Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Model predictive control for epidemic management based on neural network model

Bachelor's Thesis

Author
Tamás Epres

Advisor
Dr. Márton Vaitkus
Dr. Tamás Péni

December 6, 2024

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Epres Tamás*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, December 6, 2024

Epres Tamás
hallgató

Kivonat

A COVID-19 okozta világjárvány megmutatta, milyen nehéz olyan társadalmi intézkedéseket bevezetni, amelyek minimalizálják a gazdasági károkat, ugyanakkor megakadályozzák az egészségügyi rendszer túlterhelését. A megfelelő intézkedések megtalálása nem egyszerű feladat; a problémát felfoghatjuk szabályozástechnikai kihívásként, ahol az irányítandó szakasz a járvány lefolyását reprezentáló modell, míg a kontrol bemenetek az egyes intézkedéseknek feletehetőek meg.

Járványmodellezésre számos módszer létezik, ilyen módszer például a kompartmentális leírás, melyek során a járvány dinamikáját differenciálegyenletek írják le. Pontosabb leírást adnak a multiágens modellek, melyekben az egyéneket különálló ágensekként szimulálják. Ebben a dolgozatban a PanSim modell lett felhasználva, mint multiágens modell.

A multiágens modellek komplexitásuk és nagy dimenziószámuk miatt rendszerint nem alkalmasak arra, hogy közvetlenül a szabályozási körben alkalmazzuk őket, vagy irányítási algoritmusokat tervezzünk rájuk. Ez indokolja a SUBNET identifikációs eljárás alkalmazását, amely során neurális hálók segítségével a multiágens modell egy olyan reprezentációját állítjuk elő, amely alacsony dimenziójú, miközben dinamikai viselkedésében képes jól közelíteni az eredeti multiágens modellt.

A dolgozatban Model Predictive Control (MPC) algoritmus került implementálásra különböző járványmodellek irányításához. Először egy alacsony dimenziójú kompartmentális modellt tekintünk a járvány leírásaként. Ezt követően egy multiágens modellhez, a PanSim-hez, tervezünk kontroll input-szekvenciát úgy, hogy a SUBNET indentifikációs eljárás során kapott modell adja a tervezés alapját.

A tervezés során rámutatok azokra a nehézségekre, amelyekkel szembesülhetünk (pl. a kontroll bemenetek értékkészletének korlátozása, zaj hatása a dinamikán), és bemutatom, illetve implementálom a különböző stratégiákkal megvalósított (rolling és shrinking horizont) szabályozásokat.

Abstract

The COVID-19 pandemic has demonstrated how challenging it is to implement social measures that minimize economic damage while preventing the overburdening of the healthcare system. Finding the appropriate measures is not a straightforward task; the problem can be approached as a control engineering challenge, where the controlled process is represented by the model of the epidemic's progression, and the control inputs correspond to specific interventions.

There are numerous methods for epidemic modeling, one of which is the compartmental approach, where the dynamics of the epidemic are described using differential equations. More precise descriptions are provided by multi-agent models, in which individuals are simulated as distinct agents. In this thesis, the PanSim model has been utilized as a multi-agent model.

Due to their complexity and high dimensionality, multi-agent models are generally not suitable for direct application in control loops or for designing control algorithms. This necessitates the use of the SUBNET identification procedure, which employs neural networks to derive a low-dimensional representation of the multi-agent model that closely approximates the dynamic behavior of the original model.

In this thesis, a Model Predictive Control (MPC) algorithm was implemented for controlling various epidemic models. First, a low-dimensional compartmental model was considered to describe the epidemic. Subsequently, a control input sequence was designed for a multi-agent model, PanSim, using the model obtained through the SUBNET identification procedure as the basis for the design.

The design process highlights the challenges that may arise (e.g., the limitations on the range of control inputs and the impact of noise on the dynamics) and presents as well as implements control strategies based on different approaches, including rolling and shrinking horizon methods.

Chapter 1

Introduction

In control engineering, it is of paramount importance to have an accurate mathematical description of the system to be controlled. There are systems that exhibit linear behavior. For such linear systems, simple algorithms, such as a series compensator (PID) algorithm, can be implemented. These systems are either simple in terms of their operation, or they can be derived by linearizing a more complex system at a given operating point. However, in reality, describing most physical systems is complex, and the linearization of the system accurately describes the system only in the vicinity of a specific operating point or trajectory. For nonlinear systems the control design is challenging in general.

One of the most straightforward solutions is provided by the Model Predictive Control (MPC) algorithm, which offers an optimal control input sequence. MPC algorithms rely on mathematical optimization. During the solution of an optimization problem, a function is either minimized or maximized while satisfying given constraints. The function to be optimized is referred to as the *cost function*, and in the MPC algorithm, its value is minimized. The solution is sought under specific conditions, which are referred to as *constraints*. The variables that are unknown during the optimization process are called *decision variables* [17].

In epidemic modeling, mathematical models are developed to describe the dynamics of disease transmission within a population. The input to these models typically represents a set of intervention measures. Depending on the type of intervention implemented, the rate of disease spread within the population can be mitigated. It is characteristic that even the most basic epidemic models exhibit nonlinear behavior [14].

In the thesis, two models describing the spread of COVID-19 are presented. The first model is a compartmental model that divides the population into compartments based on the state of the disease. A characteristic feature of this model is that the transitions between compartments are formulated using ordinary differential equations. This model will be presented based on the [14] article.

The second model presented is an agent-based model, where individuals in the population are modeled as agents. These agents are simulated within a given environment and spread the disease through interactions with each other, incorporating various stochastic elements. This model is the PanSim model [15].

The goal of the control is to determine a sequence of control signals that prevents the formation of a peak in infections, thereby avoiding the overloading of the healthcare system. It is important to emphasize that this intervention should be minimal to also reduce the economic damage caused by the restrictions.

Since the dynamics of the PanSim model are complex and the dimensionality is high, the PanSim model is not suitable for directly applying the MPC algorithm design. This issue is addressed by the SUBNET identification procedure. During the SUBNET identification process, a reduced-dimensionality model is derived using neural networks, whose dynamics align with the operation of the PanSim model. The model provided by SUBNET is suitable for designing the MPC algorithm. In light of these considerations, the control of the PanSim model is implemented by applying the MPC algorithm not directly to the PanSim model, but to the model provided by SUBNET.

The structure of the thesis is as follows. The thesis commences with a concise theoretical overview of the MPC algorithm, including its mathematical formalism and the strategies employed. Subsequently, the epidemic models and their associated control designs are introduced. Initially, the compartmental model described in [14] will be presented. This is followed by a description of a multi-agent model, the PanSim model. Thereafter, a neural architecture facilitating general system identification will be introduced; this is the SUBNET. Following this, a control input sequence will be designed for the PanSim model based on the SUBNET model. Finally, the architecture of the implemented code is outlined, accompanied by a description of the utilized software and a discussion on potential directions for future development in this field.

Chapter 2

Theory of the Model Predictive Control

In this section, the primary focus is on the theoretical considerations of control and the associated mathematical formalism, rather than on the specifics of the epidemic model. A detailed exposition of the Model Predictive Control (MPC) algorithm is provided, along with a formal description of the underlying optimization problem. In addition, the applied strategies, specifically the rolling and shrinking horizon strategies, will be presented.

2.1 State-Space Representation

To provide an accurate description of the MPC algorithm, it is essential to first define the model of the real system. This model, expressed in state-space form, is presented in the following section. Consider a dynamic system whose evolution is governed by the following differential equation:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t)), \quad \mathbf{x}(0) = \mathbf{x}_0, \tag{2.1a}$$

$$\mathbf{y}_t = h(\mathbf{x}_t), \tag{2.1b}$$

where $\mathbf{x}(t) \in \mathbb{R}^n$ denotes the state vector at time t, $\mathbf{u}(t) \in \mathbb{R}^m$ represents the control input, and $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ is a continuously differentiable function describing the system dynamics. The initial state of the system is specified by $\mathbf{x}_0 \in \mathbb{R}^n$, and $h : \mathbb{R}^n \to \mathbb{R}^p$ is a function mapping the state vector to the output $\mathbf{y}_t \in \mathbb{R}^p$.

In this thesis, the MPC algorithm will be implemented for discrete-time systems. Consequently, the continuous-time dynamic model described in 2.1 must be transformed into its discrete-time representation. This transformation can be achieved, e. g. by applying, a fourth-order Runge-Kutta method [20].

Using the Runge-Kutta method, the differential equation system in 2.1 is solved with a sampling time of T_s and constant control input. As a result, the following discrete-time representation can be written:

$$\mathbf{x}_{k+1} = F(\mathbf{x}_k, \mathbf{u}_k), \quad \mathbf{x}(0) = \mathbf{x}_0, \tag{2.2a}$$

$$\mathbf{y}_k = h(\mathbf{x}_k), \tag{2.2b}$$

where $\mathbf{x}_k \in \mathbb{R}^n$ represents the state vector at discrete time step k, $\mathbf{u}_k \in \mathbb{R}^m$ denotes the control input at step $k = T_s t$, and $F : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ is the discrete-time state transition function derived through numerical integration of the continuous-time dynamics. The function $h : \mathbb{R}^n \to \mathbb{R}^p$ maps the state vector $\mathbf{x}_k$ to the output vector $\mathbf{y}_k \in \mathbb{R}^p$. The initial state of the model is $\mathbf{x}_0$.

The notation system is introduced based on the article [14]. In the expression $x[m]_k$, $\mathbf{x}$ represents a state vector, the index m refers to the m-th element of the state vector and the index k refers to the k-th time point.

2.2 The MPC algorithm

Given the representation of the dynamic model of the system, the behavior of the model is utilized to predict the system's future behavior. Based on the predicted behavior, a sequence of control actions is determined over a specified prediction horizon. In the MPC algorithm, a *cost function* is minimized based on the predicted behavior over the chosen time horizon, and the sequence of control actions obtained through this optimization represents the control input sequence [19]. A simple example is shown:

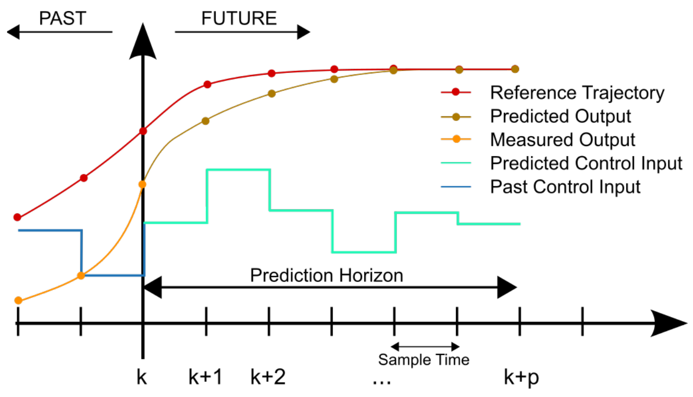

Figure 2.1: The basics of the MPC algorithm. This figure is reproduced from [19].

The MPC algorithm determines a sequence of control actions over a given horizon at an initial time point. This control sequence is applied to the real system for one time step, after which the system's response is used to re-initialize the MPC algorithm. Depending on how the MPC horizon is manipulated during the iterations, *rolling horizon* and *shrinking horizon* strategies can be distinguished.

The first strategy is the *rolling horizon* strategy. In this case, the time horizon remains fixed at N, and the time window shifts forward. Given the initial state $\mathbf{x}_0$ of the system, the optimization over the horizon N is performed starting from this initial point. The resulting control input is then applied for a duration of one time step, moving the system to the next state. At this point, the controller reinitializes the optimization with the new state over the horizon N, and the algorithm starts again [12]. An example of this strategy is shown in the following figure.

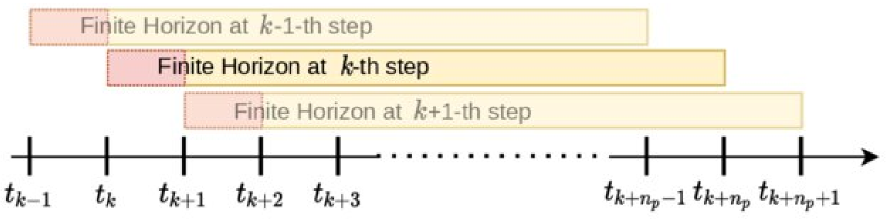
Finite Horizon at k-1-th step
Finite Horizon at k-th step
Finite Horizon at k+1-th step
t_{k-1}
t_k
t_{k+1}
t_{k+2}
t_{k+3}
t_{k+n_p-1}
t_{k+n_p}
t_{k+n_p+1}

Figure 2.2: An example of the application of the *rolling horizon* strategy. After each step, the length of the horizon over which the controller optimizes remains constant at n_p. This figure is reproduced from [9].

The second strategy is the *shrinking horizon* approach. In this scenario, the time horizon gradually shrinks as the process progresses, as the end of the time horizon is fixed at a certain point in each iteration. Initially, the time horizon is set to N, starting from the initial state $\mathbf{x}_0$. After performing the optimization from this initial point (similar to the rolling horizon strategy), the computed control input is applied for one time step, moving the system to the next state. At this point, the controller restarts the optimization using the new state of the system and a horizon that is one time step shorter than in the previous iteration [8]. This concept is illustrated in the following figure:

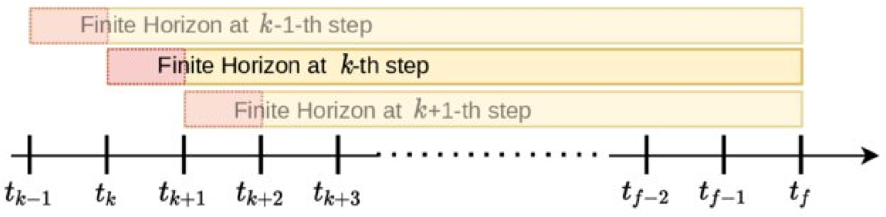
Finite Horizon at k-1-th step
Finite Horizon at k-th step
Finite Horizon at k+1-th step
t_{k-1}
t_k
t_{k+1}
t_{k+2}
t_{k+3}
t_{f-2}
t_{f-1}
t_f

Figure 2.3: Example for the *shrinking horizon* strategy. After each step, the time horizon over which the controller optimizes decreases by one. In the k-th step, the horizon length is $f - k$, while in the $(k + 1)$-th step, the horizon length becomes $f - k - 1$. This figure is reproduced from [9].

2.3 The optimization problem

As shown in section 2.2 the key concept of the MPC algorithm is the numerical optimization. In this thesis, we will use the definition of the optimization problem based on the book [17]. In this context, the optimization problem is as follows:

$$
\begin{aligned}
\text{minimize} \quad & f_0(\mathbf{x}) && \text{(2.3a)} \\
\text{subject to} \quad & f_i(\mathbf{x}) < 0, \quad i = 1, \ldots, m && \text{(2.3b)} \\
& h_i(\mathbf{x}) = 0, \quad i = 1, \ldots, p && \text{(2.3c)}
\end{aligned}
$$

Where:

- $\mathbf{x}$ is the *optimization variable* vector (each element is a optimization variable).
- $f_0(\mathbf{x})$ is the *objective function* or the *cost function.*
- $f_i(\mathbf{x}) < 0$ *inequality constraints.*
- $h_i(\mathbf{x}) = 0$ *equality constraints.*

Depending on the domains of the decision variables and the properties of the functions involved, the literature distinguishes between different types of mathematical problems (such as linear programming problems and quadratic programming problems).

If the domain of any of the decision variables is the set of integers, then we can refer to the problem as a Mixed-Integer Non-Linear Programming (MINLP) problem. Solving such a problem can be significantly more challenging than when the variables are defined over the set of real numbers. To address this, if the optimization problem associated with the MPC involves control inputs that can only take integer values, the problem can be solved over the real numbers, and then the resulting solution can be rounded to the nearest integer. This approach introduces a disturbance to the system, whose effects must be analyzed.

We shall see that, in our case, the execution time of the algorithm is not critical due to the large sampling time, which is typically one day. Therefore, unlike examples where the planning time is critical (e.g., trajectory planning for a drone), this does not pose a problem in our case.

At this stage, the relationship between the optimization problem and the MPC algorithm can be discussed. Initially, a correspondence is established between the model state vector and the vectors of the optimization variables. The controller can alter the model's states through the control inputs, implying that the controller minimizes an objective function where the optimization variables represent the model's states and the control inputs over the prediction horizon.

Typically, the function f_0 defines the *cost function* which the control is implemented, while the functions f_i impose *constraints* on the control input and the state vector, specifying the range within which the system is controlled. The functions h_i, on the other hand, describe the system dynamics, outlining the relationship between the state vector x_k and x_{k+1}, for $k = 0, \ldots, N-1$.

Let us now examine the formulation of the general equation within the context of the thesis.

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=0}^{N} u_{k+i|k}^2 + \sum_{i=0}^{N} \max\Big(H(\hat{\mathbf{x}}_{k+i|k}) - T, 0\Big)^p \cdot P && \text{(2.4a)}\\
\text{subject to} \quad & \hat{\mathbf{x}}_{k+i+1|k} = F(\hat{\mathbf{x}}_{k+i|k}, u_{k+i|k}) \quad k = 0, \ldots N-1 && \text{(2.4b)}\\
& \hat{\mathbf{x}}_0 = \mathbf{x}_0 && \text{(2.4c)}\\
& u_{k+i|k} = 0 \quad \text{for } k > N - G && \text{(2.4d)}\\
& u_{l|l} = u_{l+1|l} = \cdots = u_{l+h-1|l} \quad l = 0, h, 2h, \ldots N-h && \text{(2.4e)}\\
& U_{min} \le u_{k+i|k} \le U_{max} \quad k = 0, \ldots N && \text{(2.4f)}
\end{aligned}
$$

Where:

- $u \in U$ is the control input.
- U is the domain of the control input which is a finite set.
- $\hat{\mathbf{x}} \in \mathbb{R}$ is the predicted state vector of the system.
- $\mathbf{x}_0$ is the initial state vector.
- F is the function that represents the dynamics of the system.
- H is the function that represents the mapping from the state vector to the output.
- N is the control horizon.
- G is the time duration until 0 control input is used.
- i is the running index over the control horizon.
- k is the time variable.
- T is constant constraint value.
- h is the holding time.
- P is a penalty weight.
- p is a penalty power.

The terms in the *cost function* 2.4a are as follows. While the first term is responsible for minimizing the control input sequence, the second term implements a soft constraint. The idea is that if we impose a simple inequality constraint on the output, represented by the function H from the current state vector, it is possible for the actual system response to present a state that violates the constraint due to stochastic behavior. This could lead to an infeasible problem for the controller. To address this, we introduce an additional term in the cost function that penalizes any output that exceeds a certain threshold. The penalty term is proportional to the extent by which the output exceeds this limit, raised to a given power.

When applying *rolling horizon* strategy or *shrinking horizon* strategy, the parameters of the optimization are adjusted, while $\mathbf{x}_0$ is taken as the actual system response. Applying the *rolling horizon* strategy, the parameter N remains constant, while the value of G changes. If the *shrinking horizon* strategy is applied, G remains constant while N changes.

Chapter 3

Control design based on Compartmental model

In the following section, the compartmental epidemic model describing the progression of the COVID-19 pandemic in Hungary will be presented and analyzed. The core concept of this model is to divide the population into distinct groups, referred to as compartments, each representing a specific stage of infection. The transitions between these compartments are described by differential equations, characterizing the model as a continuous-time framework. This methodology is based on the research documented in [14], conducted by the Institute for Computer Science and Control (SZTAKI). While this model differs from the approaches presented in subsequent chapters, it provides a clear demonstration of the fundamental principles underlying epidemic dynamics. In this chapter, both the controlled plant and the model predicted by the MPC are based on the compartmental model. Assume that the complete state vector of the model is measurable.

3.1 Description of the model dynamics

The uninfected population (S) consists of individuals who have not yet contracted the infection but remain susceptible to becoming infected. The latent population (L) includes those who have been exposed to the infection but do not yet exhibit any symptoms. Subsequently, individuals transition to the pre-symptomatic infectious group (P), from which they either progress to the symptomatic infected group (I) or the asymptomatic infected group (A). Members of the asymptomatic group are likely to recover from the disease, necessitating the inclusion of the recovered group (R). Individuals exhibiting symptoms may recover and transition to the R group or require hospitalization (H). Patients in the hospital either recover or succumb to the disease (D).

The dynamics of these transitions are described by the following system of differential equations:

$$\dot{S}(t) = -\beta(1-u(t))\frac{[P(t)+I(t)+\delta A(t)]S(t)}{N} \tag{3.1a}$$

$$\dot{L}(t) = \beta(1-u(t))\frac{[P(t)+I(t)+\delta A(t)]S(t)}{N} - \alpha L(t) \tag{3.1b}$$

$$\dot{P}(t) = \alpha L(t) - pP(t) \tag{3.1c}$$

$$\dot{I}(t) = qpP(t) - \rho_1 I(t) \tag{3.1d}$$

$$\dot{A}(t) = (1-q)pP(t) - \rho_A A(t) \tag{3.1e}$$

$$\dot{H}(t) = \rho_1 \nu I(t) - hH(t) \tag{3.1f}$$

$$\dot{R}(t) = \rho_1(1-\nu)I(t) + \rho_A A(t) + (1-\mu)hH(t) \tag{3.1g}$$

$$\dot{D}(t) = \mu hH(t) \tag{3.1h}$$

Where:

- $u(t)$: The control input. The physical meaning of the control input is the magnitude of the intervention.
- β: The infection rate.
- δ: The infectiousness ratio in the $A(t)$ group.
- N: The total population size.
- α: The rate of transition from the $L(t)$ state.
- p: The rate of transition from the $P(t)$ state.
- q: The probability that an individual in the $P(t)$ state moves to the $I(t)$ state.
- ρ_1: The recovery or transition rate of infected individuals ($I(t)$).
- ρ_A: The recovery or transition rate of asymptomatic individuals ($A(t)$).
- ν: The probability that an infected individual ($I(t)$) requires hospitalization.
- h: The mortality rate of hospitalized patients ($H(t)$).

The dynamics are illustrated in the following graph:

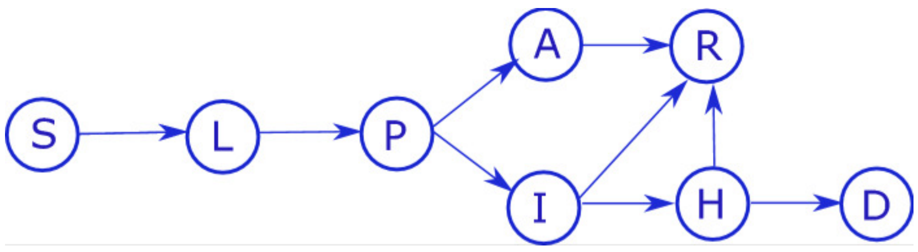

Figure 3.1: Population dynamics over the course of the epidemic. This figure is reproduced from [14].

Furthermore, the system dynamics are non-linear due to the components represented by equations (3.1a) and (3.1b). It is evident that when $u(t) = 0$, no intervention is applied, whereas $u(t) = 1$ corresponds to the implementation of the strictest intervention level. For numerical stability, the states have been normalized by dividing each compartment by the total population size.

The model exhibits a conserved property, meaning that when all equations are summed, the result is equal to one when normalized by the total population size. This implies that no new individuals are introduced into the model. Moreover, it can be demonstrated that, with normalized states in the system:

$$\lim_{t \to \infty} |S(t) + R(t) + H(t)| = 1$$

This implies that all individuals will eventually belong to either the S, R, or D groups. In this sight the S, R, D are the permanent groups, while L, P, I, A, H represent the transient groups. The model does not consider the mechanism of reinfection.

As discussed in Section 2.2, we transition to discrete time, since the MPC operates in discrete time in this thesis.

3.2 The Goal of the control

The primary objective of the control strategy is to design a control input sequence that prevents the healthcare system from reaching its capacity limit while simultaneously minimizing the magnitude of interventions. In light of the above, the goal of the control is to direct the system using a minimal control input sequence such that the number of hospitalized patients does not exceed a given threshold (20,000). During the control process, the system is controlled for 217 days, with no intervention after day 175, meaning that the control input is zero after this point.

The significance of seeking a control input whose values are uniformly zero after a certain point in time is as follows: We aim to find control input sequences that manage the epidemic in such a way that, at the end of the control period, the epidemic does not resume and no second wave emerges. In other words, without further intervention, the number of patients in hospitals should not exceed the healthcare capacity for a given period.

3.3 Simulation results

In this section, the results obtained during the control of an epidemic are presented. The MPC algorithm computes the control input sequence based on a discrete-time compartmental model. The system to be controlled is represented by a continuous-time compartmental model. Initially, the analysis is conducted without incorporating noise; subsequently, the impact of noise on the real system is introduced. Additionally, the effect of disturbances caused by rounding the control input is systematically investigated.

The compartments were initialized as follows:

$$\mathbf{x}_0 = \left[1 - \frac{10}{9{,}800{,}000}, \frac{10}{9{,}800{,}000}, 0, 0, 0, 0, 0, 0\right],$$

where the compartments in the state vector are ordered as follows: S, L, P, I, A, H, R, D.

Both the *shrinking horizon* and *rolling horizon* strategies were implemented, and their respective execution times are documented.

The 'Ipopt' solver was used to solve the optimizations [1]. However, the problem was implemented in Python using CasADi [2].

3.3.1 Parameters of the control

The initial parameters of the plant model and the optimization problem are as follows:

- In expression 2.4a, $N = 175 + 42$ days if the shrinking horizon strategy is applied, and $N = 119$ days for the rolling horizon strategy.
- In 2.4a, the capacity of the healthcare system is set to 20,000.
- In 2.4a, H is the mapping function; in this case, the number of hospitalized patients is the sixth element of the state vector.
- In 2.4a, the P penalty constant is set to 10^9.
- In 2.4a, the p penalty power constant is set to 1.
- In 2.4b, F represents the numerical solution of the dynamics obtained using the fourth-order Runge-Kutta method with a 1-day sampling time.
- In expression 2.4d, $G = 42$ days if the shrinking horizon strategy is applied, and $G = 0$ days for the rolling horizon strategy.
- In expression 2.4e, the holding time is set to $h = 7$ days.

The parameters N and G change during the iterations depending on the strategy applied.

When using the *shrinking horizon* strategy, G remains constant, while N decreases by 7 days (the holding time) with each iteration.

When using the *rolling horizon* strategy, N remains constant, but once the horizon reaches 175 days from the starting point, the parameter G increases by 7 days (the holding time) with each iteration.

The algorithm runs until the system reaches day 217, meaning that a total of 31 optimization iterations are performed for each algorithm.

The initial state $\mathbf{x}_0$ value changes each week depending on the system's response. Assuming that we can estimate the state vector, the controller is re-initialized directly based on the system's response.

For the sake of completeness, the case of control with zero input is illustrated in the following figure.

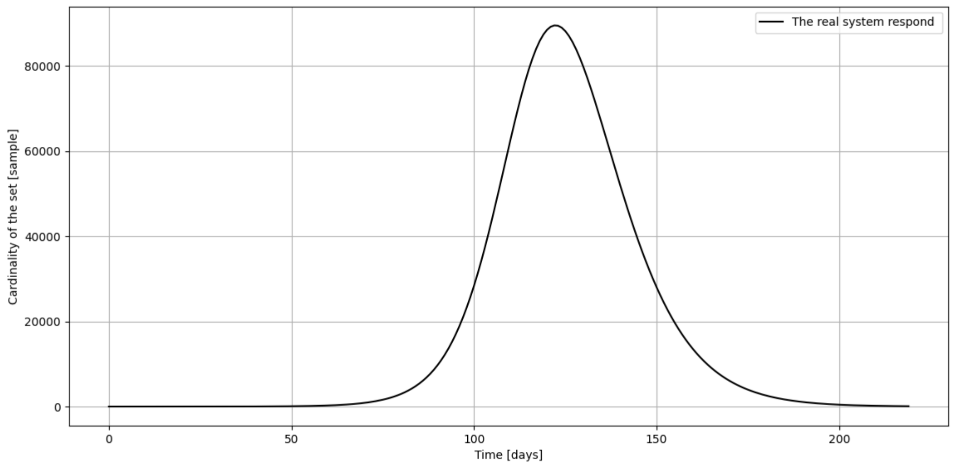

Figure 3.2: The time evolution of patient numbers in the absence of control. The black curve represents the system response.

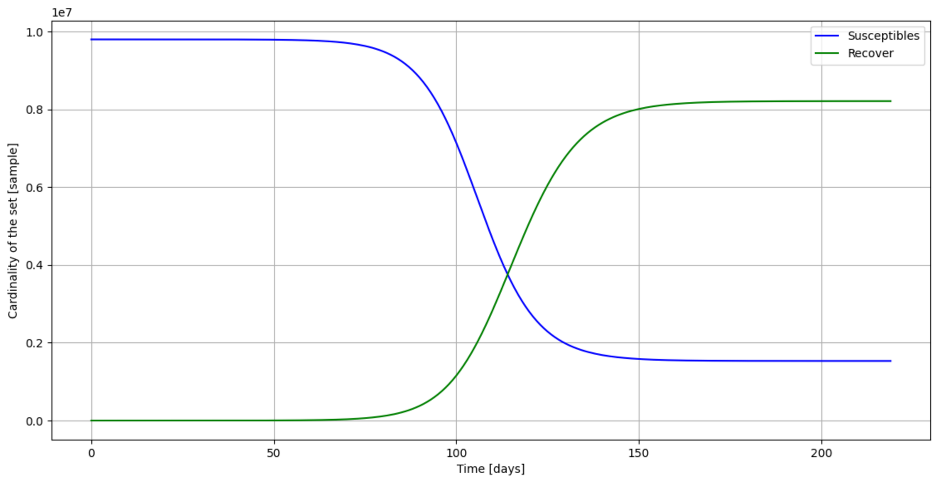

Figure 3.3: The time evolution of the population sizes in compartments S and R under zero-control conditions. The blue curve represents the susceptibles, and the green curve represents the recovereds.

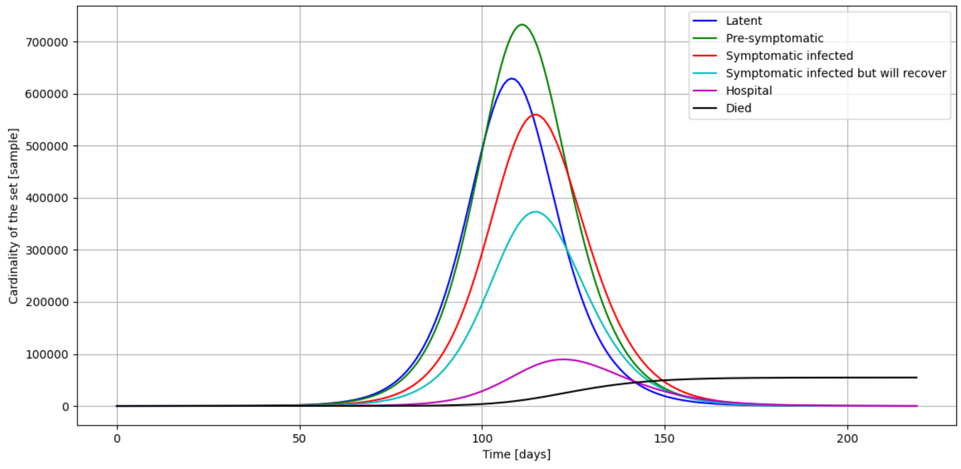

Figure 3.4: The time evolution of the population sizes in compartments L, P, S, A, H, and D under zero-control conditions. The blue curve represents L, the green curve represents P, the red curve represents S, the cyan curve represents A, the pink curve represents H (hospital), and the black curve represents D.

3.3.2 Results without noise

In the following, we apply a *shrinking horizon* strategy with a 217-day horizon. In this subsection, we analyze the system's behavior in the absence of noise, considering only the disturbances caused by the rounding of the control input. Under these conditions, the behavior prescribed by the model aligns perfectly with the actual system's behavior. The results presented below were obtained after optimization. The x-axis represents time in days, while the y-axis displays the number of individuals in each compartment. In the control plot, the y-axis corresponds to the index of the intervention.

In Scenario 1, the control input sequence is not subjected to rounding. In this case, u can take any real value between 0 and 1, and no noise is introduced to the real system dynamics.

In Scenario 2, the range of u is restricted by dividing the interval from 0 to 1 into 18 equal parts. Thus, $u \in \left\{0, \frac{1}{17}, \frac{2}{17}, \ldots, \frac{16}{17}, 1\right\}$. No noise is added to the real system's dynamics.

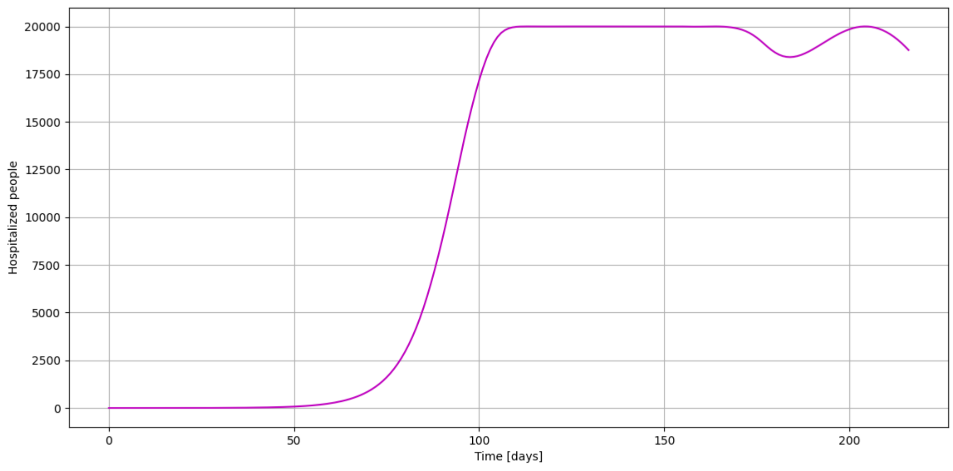

Figure 3.5: The time evolution of patient numbers in Scenario 1. The magenta curve represents the system response.

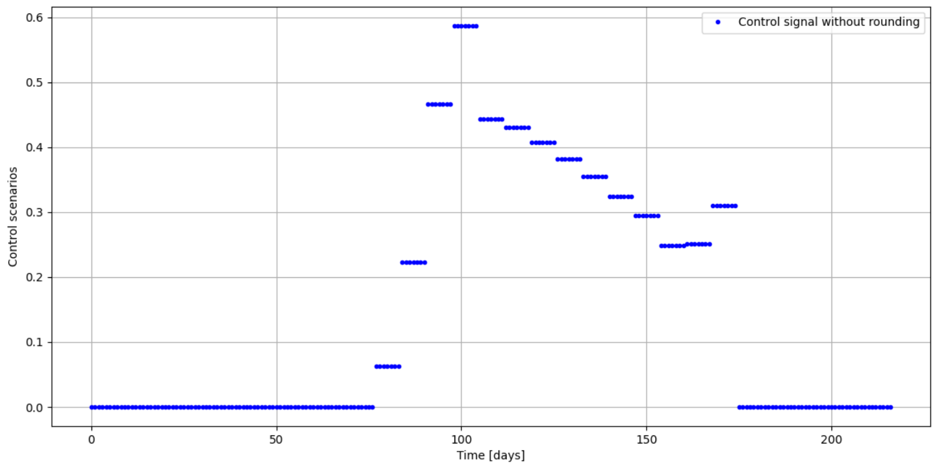

Figure 3.6: Control input applied to the system without rounding in Scenario 1.

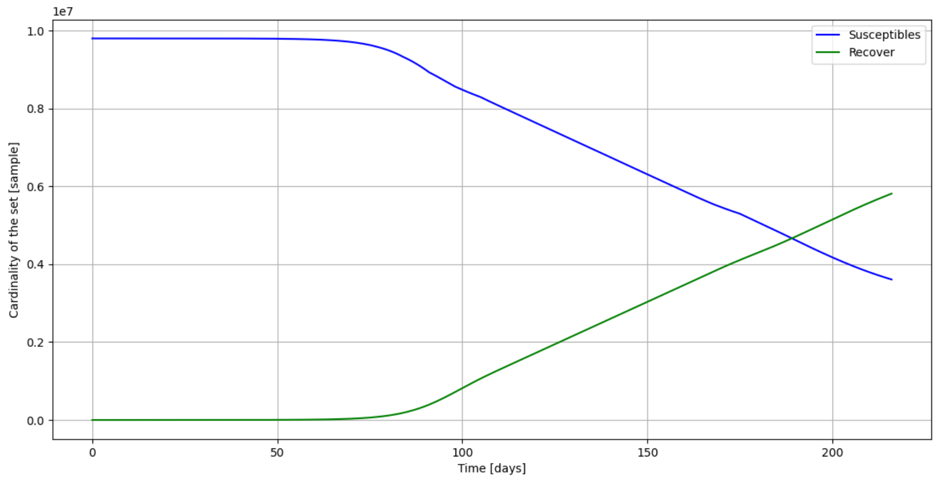

Figure 3.7: The time evolution of the S and R compartments with the control input in Scenario 1. The blue curve represents the susceptibles, and the green curve represents the recovers.

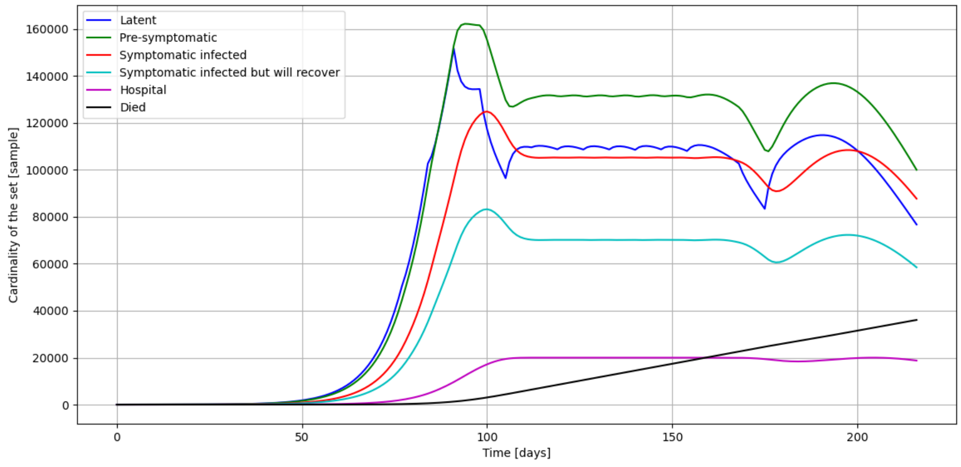

Figure 3.8: The time evolution of the L, P, S, A, H, and D compartments in Scenario 1. The blue curve represents L, the green curve represents P, the red curve represents S, the cyan curve represents A, the pink curve represents H (hospital), and the black curve represents D.

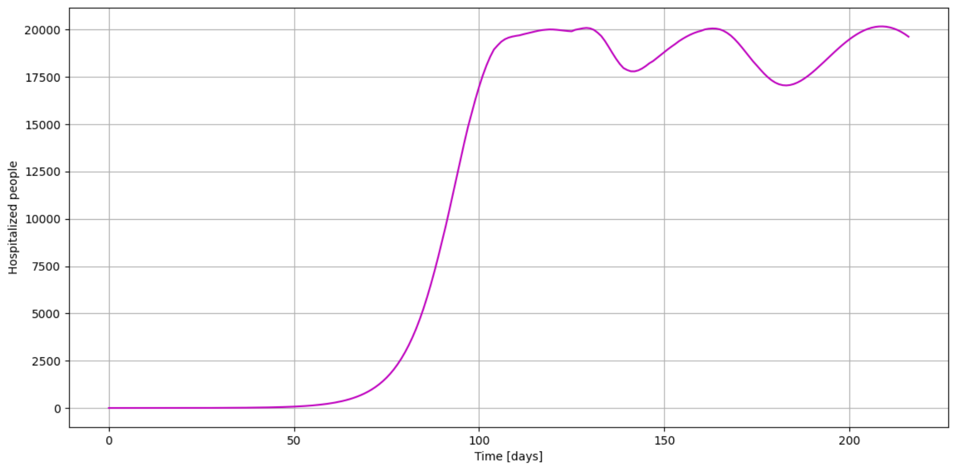

Figure 3.9: The time evolution of patient numbers in Scenario 2. The magenta curve represents the system response.

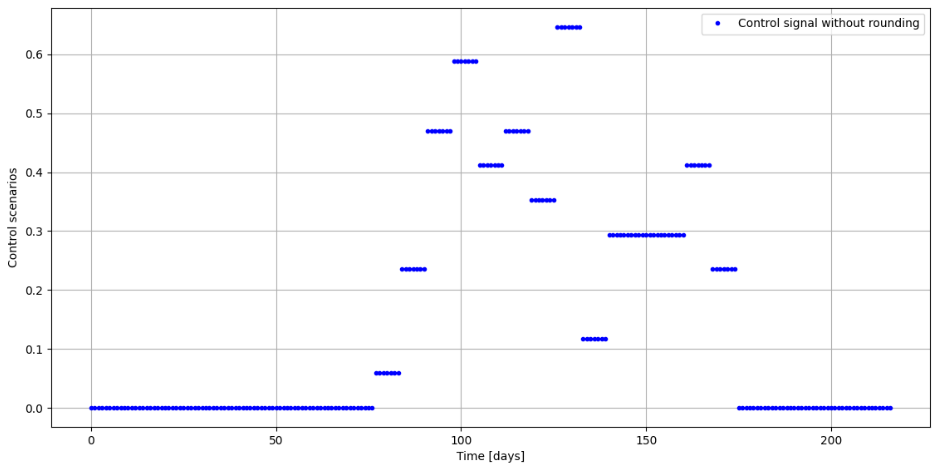

Figure 3.10: Control input applied to the system with rounding in Scenario 2.

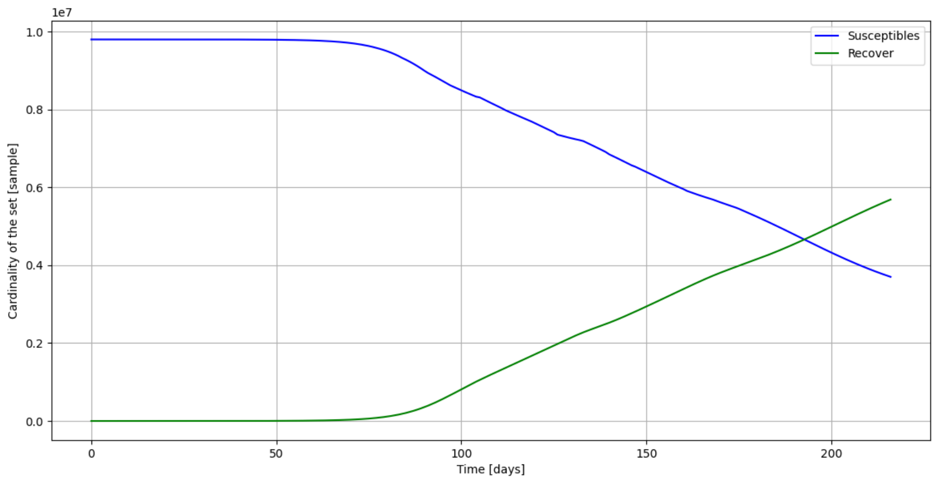

Figure 3.11: The time evolution of the S and R compartments in Scenario 2. The blue curve represents the susceptibles, and the green curve represents the recovereds.

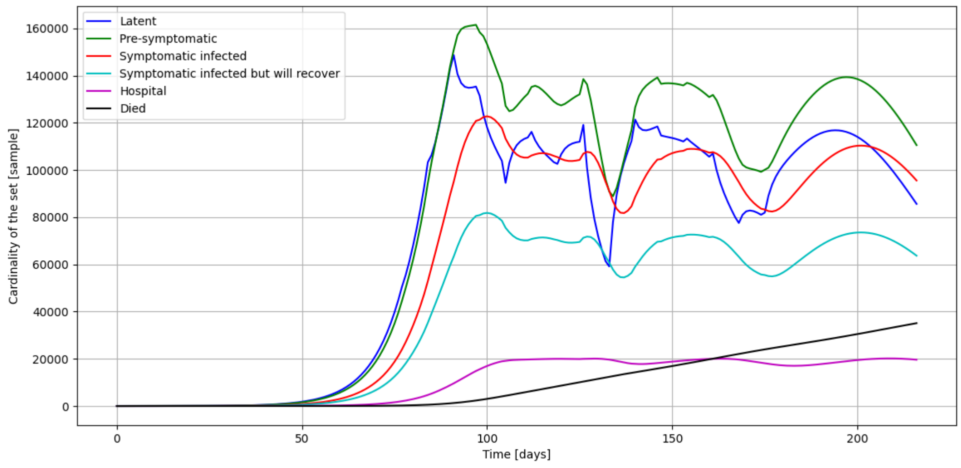

Figure 3.12: The time evolution of the L, P, S, A, H, and D compartments with the ideal control input in Scenario 2. The blue curve represents L, the green curve represents P, the red curve represents S, the cyan curve represents A, the pink curve represents H (hospital), and the black curve represents D.

The following conclusions can be drawn from the figures: Rounding the control input does not lead to significantly different results; it introduces a certain type of oscillation, but the MPC still generates a control input sequence that respects the constraints. This oscillation is a consequence of the coarse discretization. While in the non-rounded case, the average execution time of the MPC (i.e., the average time during which the solver solved each optimization problem) was 1.35 seconds, in the rounded case, it was 1.25 seconds.

3.3.3 Results with noise

In this section, we examine the effects of introducing noise to the real system. The model used in this analysis operates in discrete time as a compartmental model, while the real system is simulated using a continuous-time compartmental model. We apply white noise with a mean of 0 and a uniform deviation of $\frac{800}{9{,}800{,}000}$ to the system's state vector at each time step, at every k time point. This noise represent disturbances caused by measurement inaccuracies. Consequently, the behavior of the model differs from that of the real system. The control input was quantized such that $u \in \left\{0, \frac{1}{17}, \frac{2}{17}, \ldots, 1\right\}$.

The *shrinking horizon* strategy is first applied with a time horizon of 217 days. The average runtime was 1.49 seconds.

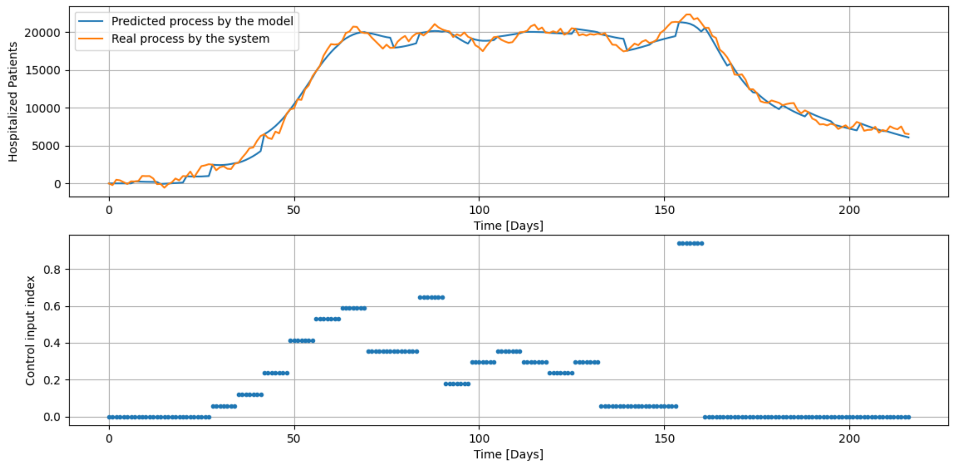

Figure 3.13: The upper plot illustrates the progression of hospitalized patient numbers, with the blue curve indicating model-based calculations and the yellow curve showing the actual patient count. The lower plot presents the control input sequence obtained using the *shrinking horizon* strategy.

Subsequently, a *rolling horizon* strategy was applied with a time horizon of 119 days. The average runtime was 1.1 seconds.

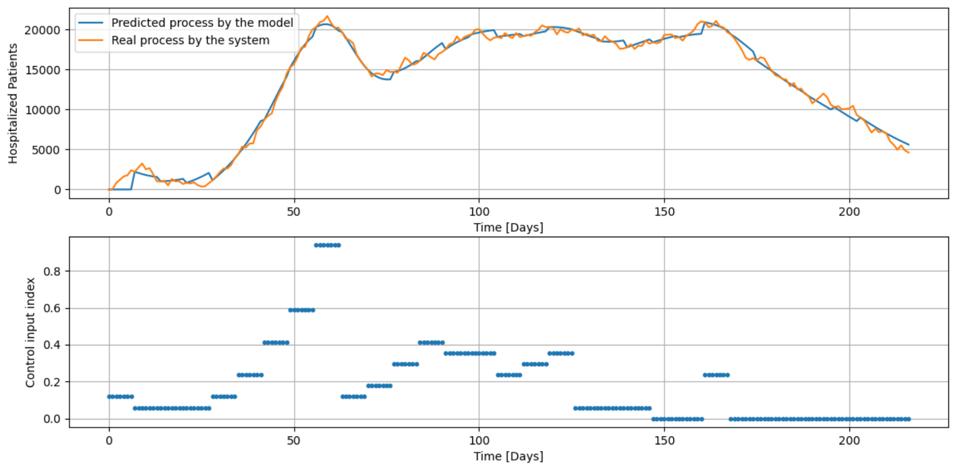

Figure 3.14: The upper plot illustrates the temporal evolution of the number of hospitalized patients, where the blue curve corresponds to the data computed by the model, and the yellow curve represents the actual patient count. The lower plot depicts the applied control input sequence, generated using the *rolling horizon* strategy.

When noise is applied to the system (see Figure 3.13 and Figure 3.14), similar control trajectories are obtained as in the noiseless case (see Figure 3.5 and Figure 3.9): the number of patients in the hospital increases for a while, then stabilizes around the hospital capacity value, and finally decreases around the control horizon. It can be stated that this type of control trajectory is suitable for our purposes. During the control process, we avoided the formation of an infection peak, and at the end of the control period, the number of patients did not start to increase.

It can also be observed that both the predicted number of patients by the model and the actual number of patients exceed the allowed number. This is due to the noise applied to the real system. Nevertheless, it can be seen that the predicted trajectory by the model does not significantly differ from the actual system's behavior.

It can also be stated that similar results were obtained during the control process for both the *shrinking* and *rolling horizon* applications.

Chapter 4

PanSim model

In this chapter, the multi-agent model is presented, whose dynamics were learned by the SUBNET neural network. The multi-agent model was developed by the National Laboratory of Health Security at the Faculty of Information Technology and Bionics, Pázmány Péter Catholic University, and is implemented in the PanSim simulator (PanSim model) [11].

The PanSim model simulates the epidemiological progression of COVID-19 in the city of Szeged with high accuracy, offering a wide range of adjustable parameters, such as population distribution and the emergence of new mutations. From this point onwards, the controlled plant will be based on the PanSim model.

As mentioned in the introduction, providing an exact model for the progression of an epidemic with high accuracy is a very challenging task. As noted in [15], the compartmental model of epidemics described by differential equations (see Chapter 3) is easy to fit to the data of a given pandemic, but it struggles with addressing the issues of population heterogeneity and spatial distribution. For instance, we could consider how the epidemic response might differ between regions.

In light of these challenges, the PanSim multi-agent based model was developed. It is a stochastic model capable of addressing the difficulties arising from the issues mentioned above, although it requires significant computational capacity.

4.1 Description of the model dynamics

The detailed operation of PanSim is comprehensively described in [15]. Based on this reference, its functionality is presented in the following sections.

PanSim implements an agent-based model in which individual agents simulate people living in a city and carrying out their daily routines. These activities may include tasks such as commuting to work or school, along with stochastic elements such as shopping or engaging in entertainment. During these activities, agents can randomly transmit infections to one another.

As mentioned, PanSim models the epidemic progression in the city of Szeged, defining 179,500 agents and 60,000 different locations. The agents' key characteristics include age, gender, medical history, as well as a set of locations they visit daily during their movement. The population is randomly initialized.

The simulation models agents as individuals residing in a virtual city, following predefined daily routines. Their movements are guided by a time schedule derived from time-use statistics, with updates occurring every 10 minutes. Based on their schedules, agents move to designated locations where interactions take place. Transmission of infection occurs at these locations, influenced by the number of infectious agents present and the nature of the environment (indoor or outdoor). At the end of each simulated day, the epidemic state evolves, and agents transition to their corresponding health states. The movement patterns of the agents are demonstrated in the accompanying video [13].

4.2 The Input and the Output of the PanSim

This section introduces the inputs and outputs of the PanSim model. As mentioned in [15], the spread of the epidemic is determined by the routines of the agents and the probability of disease transmission. The fundamental idea is to intervene in the course of the epidemic by manipulating these routines and the likelihood of disease transmission.

The PanSim model accepts a scalar input, which can range from 0 to 17. Using the notation introduced earlier, $u \in \{0, \ldots, 17\}$. Each input value corresponds to different interventions. Essentially, each input value is associated with six types of interventions, which are as follows: testing frequency, various lockdowns, measures related to curfews, school opening rules, quarantine regulations, and mask usage requirements. The higher the value of the intervention, i.e., the closer u approaches its maximum value, the stricter the regulations imposed.

The PanSim model simulates the agents' behavior on a given day as described at the beginning of the chapter, based on the intervention implemented. The sampling interval is one day, and the simulation results are recorded daily.

The PanSim model produces a vector, which encompasses various epidemic-related data. This includes the count of individuals who remain susceptible to the virus, as well as the count of individuals who have experienced reinfection. During the control process, we base our analysis on the output of PanSim, which represents the number of patients treated in hospitals.

Chapter 5

SUBNET model

In this chapter, the applied neural model is presented. The neural model was developed collaboratively by the Control Systems Group of Eindhoven University of Technology and the Systems and Control Laboratory of the HUN-REN Institute for Computer Science and Control (SZTAKI), as documented in [18]. From this point onward, the neural model will be referred to as "SUBNET," following the terminology in [18].

The PanSim model provides a detailed description of the spread of the COVID-19 pandemic. However, the simulation is computationally intensive and does not offer relationships between the states of the epidemic that can be directly used in an MPC algorithm. The PanSim model can be regarded as a black-box model: while its exact functioning is not fully known, its responses to given inputs are well-defined. In the application of MPC, a model is required that explicitly describes the relationships between the system's states. Therefore, we need a model that captures the connections between the system's states and whose dynamics match those of the PanSim model. One possible solution is provided by the SUBNET identification method.

The SUBNET is a general nonlinear identification procedure and is used for approximating nonlinear systems. An example of such nonlinear system identification is found in [18], where this architecture was used to identify a vehicle.

In the SUBNET identification procedure, three neural networks are defined: the encoder, the system-updating network, and the output network. While the encoder estimates an initial state based on the last n input-output pairs, the state-updating network estimates the next state based on the current state and the current input. The output network estimates an output based on the current state.

5.1 Description of the SUBNET

The SUBNET can operate in both continuous and discrete time. The SUBNET model consists of three distinct neural networks, whose notation will be presented based on [18].

These are the encoder ψ_θ, the state update network f_θ, and the output network h_θ. The introduction of the encoder will be presented first.

The encoder implements the neural network that estimates an initial state vector based on the system's past input-output pairs. In light of the above, the encoder ψ_θ provides an initial state $\hat{\mathbf{x}}_0 = \psi_\theta(\mathbf{U}, \mathbf{Y})$, where $\mathbf{U}$ and $\mathbf{Y}$ are defined as:

$$\mathbf{U} = \begin{bmatrix} \mathbf{u}_0 \\ \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_{n-1} \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_{n-1} \end{bmatrix}$$

The initial state vector estimator depends on the previous n input-output pairs. $\mathbf{U}$ symbolizes the past n inputs to the system, and $\mathbf{Y}$ represents the system's responses for the inputs at the given indices.

Subsequently, the state update network f_θ is used to provide the next predicted state vector based on the previous state and the previous intervention, $\hat{\mathbf{x}}_{k+1} = f_\theta(\hat{\mathbf{x}}_k, \mathbf{u}_k)$. The state update network f_θ is responsible for encoding the dynamics.

Once the estimated state vectors are obtained, these values must be mapped to the physical quantity on which we intend to control the system, i.e., the estimated state vectors must be mapped to the estimated system responses. This is realized by the h_θ network, $\hat{\mathbf{y}}_k = h_\theta(\hat{\mathbf{x}}_k)$.

It is understood that here $k \geq 0$, since the initial state $\hat{\mathbf{x}}_0$ is provided by the encoder. Since this model operates in discrete time, unlike the compartmental model, there is no need for the use of the Runge-Kutta method. The following figure shows the SUBNET architecture in discrete time.

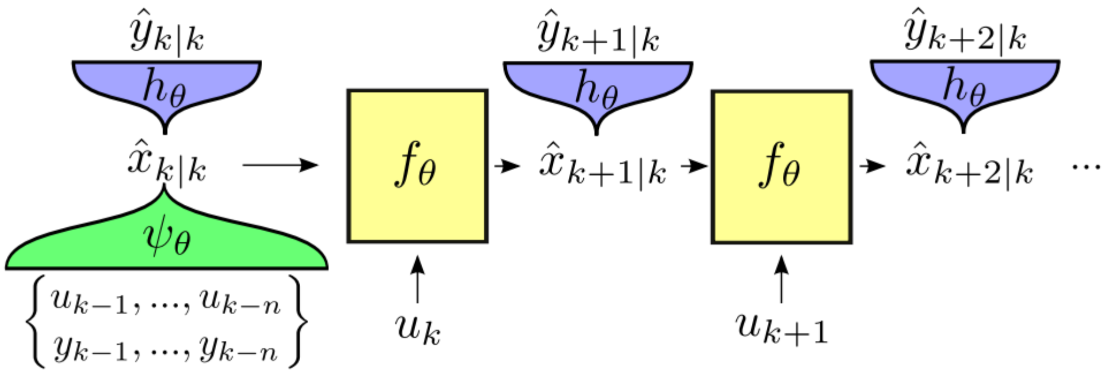

Figure 5.1: SUBNET structure in discrete time. This figure is reproduced from [18].

5.2 Identification of the PanSim by the SUBNET

Due to the complex nature and high dimensionality of PanSim's operation, it is not possible to directly design an input sequence for controlling PanSim. To address this challenge, the SUBNET identification procedure is employed. The training of the SUBNET networks to capture PanSim's dynamics is comprehensively detailed in [10].

Since PanSim expects a scalar value as input and returns a scalar value as output, the control input and the number of hospitalized people as outputs of SUBNET are also scalar, while its state vectors are column vectors with 16 elements. In the context of the existing notation, $u_k \in \mathbb{R}$ and $y_k \in \mathbb{R}$ denote the scalar input and output at discrete time step k, and $\mathbf{x}_k \in \mathbb{R}^{16}$ represents the 16-element row state vector at time step k. Additionally, $\mathbf{U} \in \mathbb{R}^{30}$ and $\mathbf{Y} \in \mathbb{R}^{30}$ represent the 30-element column vectors symbolizing the input-output pairs from the past 30 days. The functions are defined as:

$$\hat{\mathbf{x}}_{k+1} = f_\theta(\hat{\mathbf{x}}_k, u_k),$$

$$\hat{y}_k = h_\theta(\hat{\mathbf{x}}_k),$$

$$\hat{\mathbf{x}}_0 = \psi_\theta(\mathbf{U}, \mathbf{Y}),$$

where $f_\theta : \mathbb{R}^{16} \times \mathbb{R} \to \mathbb{R}^{16}$ maps the current state and input to the next state, $h_\theta : \mathbb{R}^{16} \to \mathbb{R}$ maps the state to the output, and $\psi_\theta : \mathbb{R}^{30} \times \mathbb{R}^{30} \to \mathbb{R}^{16}$ maps the input-output history to the initial state.

In the following part of the section, a comparison is made between the data provided by PanSim and the outputs estimated by SUBNET.

In these simulations, PanSim was simulated with a uniform input of 0 over 30 days. Subsequently, the encoder ψ_θ estimated an initial state $\hat{x}_0$ based on the input-output pairs of the past 30 days. Specifically, U is a 30-element column vector in which each element is equal to 0, whereas Y represents the simulation results obtained from the first 30 days by PanSim. Afterward, the SUBNET provides a state estimate for the next state based on the current state and the current input corresponding to the scenario using f_θ, while h_θ provides an estimate for the output based on the current state. Additionally, the PanSim model is updated with the control input value corresponding to the scenario.

Subsequently, two scenarios are distinguished. Scenario 1: a control input with a uniform value of 8 is applied after the encoder has estimated the initial state. Scenario 2: a control input with a uniform value of 0 is applied.

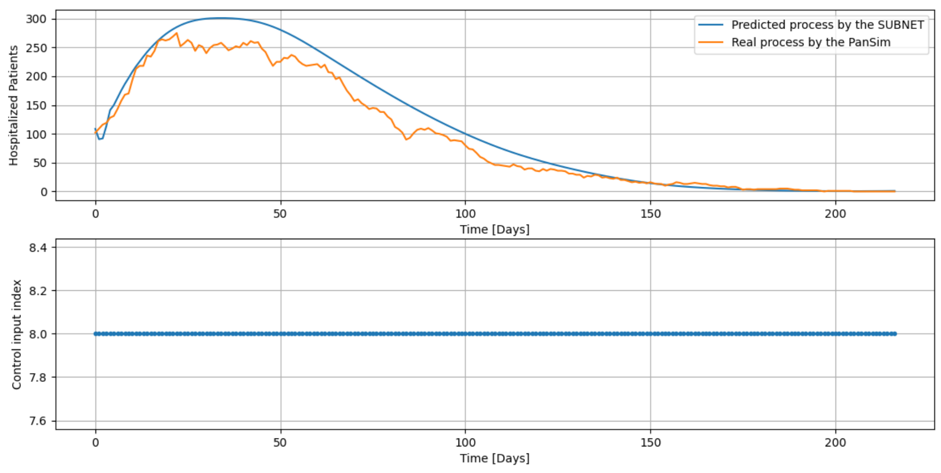

Figure 5.2: The predicted outputs of the SUBNET (blue curve) and the outputs of the PanSim model (yellow curve) with a 8 control input in Scenario 1.

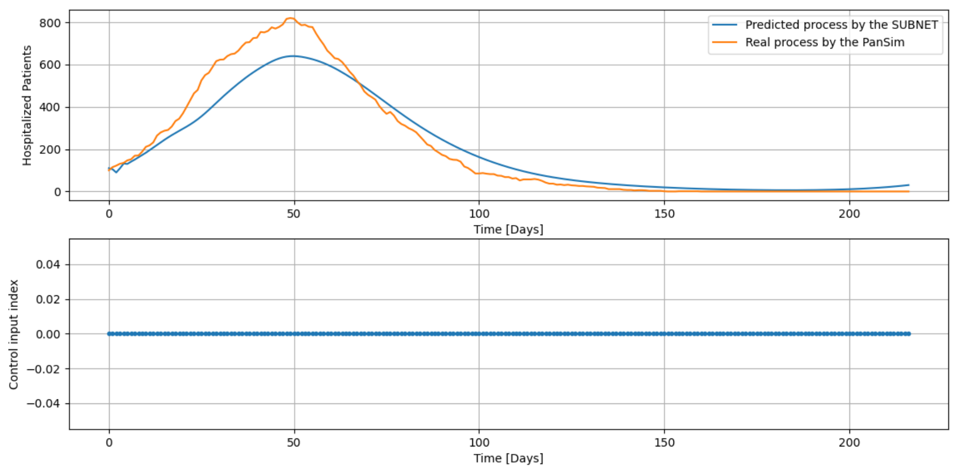

Figure 5.3: The predicted outputs of the SUBNET (blue curve) and the outputs of the PanSim model (yellow curve) with a 0 control input in Scenario 2.

Based on the Figure 5.2 and Figure 5.3, we can conclude that the SUBNET currently predicts the behavior of the PanSim model well. However, we must also note that the discrepancy between the actual number of infected individuals provided by PanSim and the estimates given by the SUBNET can vary by as much as 200 individuals (see the peak of the pandemic in Figure 5.3).

We can also conclude that the encoder correctly provided the initial state for the SUBNET model, as both the yellow and blue curves start from nearly the same position.

Chapter 6

MPC Design Based on SUBNET Model

This chapter will present the design of the MPC algorithm based on SUBNET for the PanSim model. In this case, the number of patients simulated by PanSim is considered the real system, while the SUBNET architecture is regarded as the system model. The controller then optimizes based on the stepper and mapping network of SUBNET. During the control process, a closed-loop control system is implemented with shrinking and rolling horizon strategies. Since PanSim only accepts integer intervention inputs ranging from 0 to 17, the control input results obtained during the optimization were rounded accordingly.

The optimization problems were solved using the 'Ipopt' solver [1]. The implementation of the problems was carried out in Python, employing CasADi [2].

6.1 Control Design

The initial state vector is estimated by $\hat{\mathbf{x}}_0 = \psi_\theta(\mathbf{U}, \mathbf{Y})$, where ψ_θ denotes the encoder function that estimates the initial state from the input-output pair, $\mathbf{U} \in \mathbb{R}^{30}$ is a 30-element column vector where each element is 0, and $\mathbf{Y} \in \mathbb{R}^{30}$ represents the responses obtained from PanSim.

Subsequently, the controller optimizes over a given time horizon based on the SUBNET's f_θ and h_θ networks, the initial state $\mathbf{x}_0$, and generates a control input sequence.

After the control input sequence is generated during optimization, it is applied to the PanSim model for seven days, meaning that the first 7 elements of the control input sequence are used. The $\mathbf{U}$ and $\mathbf{Y}$ vectors are always updated after applying the control input, so they always contain the most recent 30 input-output pairs.

Based on the updated $\mathbf{U}$ and $\mathbf{Y}$ vectors, the encoder predicts a new state, which initializes the controller. Depending on the strategy used, the controller optimizes again over the given horizon based on the SUBNET's f_θ and h_θ networks, and the process starts over. The control continues until the PanSim simulation reaches a specified time point.

Thus, we can conclude that a closed-loop control system is implemented, where the responses from PanSim are fed back to the controller through the encoder.

In this control setup, an example will be presented where a uniform zero control input was applied to both the SUBNET and PanSim over 217 days. The result is shown in the following figure.

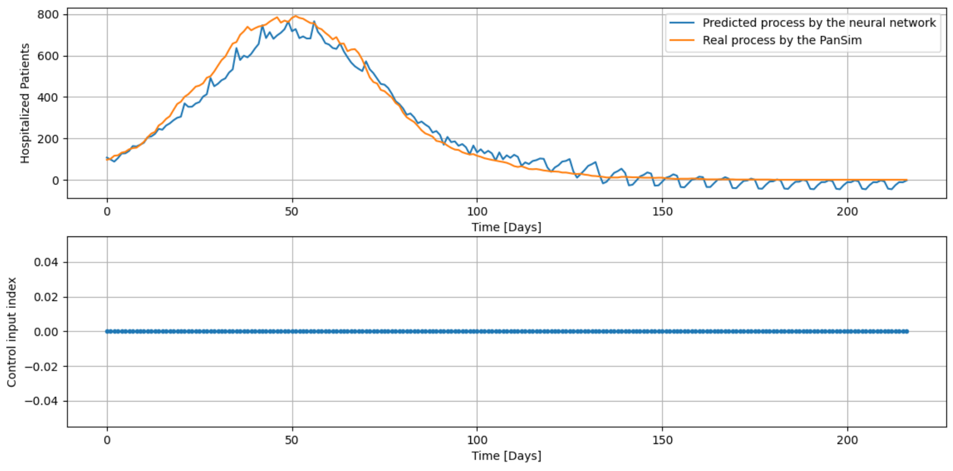

Figure 6.1: The SUBNET and the PanSim model with zero control input, in closed loop.

As shown in the Figure 6.1, the behavior of PanSim is similar to that in Figure 5.3, as it received the same control input. Additionally, it can be observed that in each iteration, the encoder provides a state that aligns the blue curve with the yellow curve, resulting in a smaller discrepancy between the output estimated by SUBNET and the actual output compared to what is shown in Figure 5.2 or Figure 5.3.

6.2 Goal of the control

The objective of the control is as follows. During the control process, we seek a control input that minimizes its value while ensuring that the number of hospitalized patients does not exceed 200, thus preventing the healthcare system from being overwhelmed. The control is applied for 217 days, with no interventions applied uniformly after day 175.

The rationale for applying a uniform zero intervention after day 175 is similar to the reasons introduced in Chapter 3. We seek control input sequences that guide the pandemic in such a way that, by the end of the control period, the pandemic does not show an increasing trend, and the number of infected individuals decreases, even without any intervention.

6.3 Parameters of the Control

The MPC algorithm solves the mathematical optimization problem presented in 2.4 during each iteration with the given parameters. The initial parameters are as follows.

- In 2.4c, the initial state vector of the system is given by the encoder network of the SUBNET, based on $\mathbf{U}$ and $\mathbf{Y}$. $\hat{\mathbf{x}}_0 = \psi_\theta(\mathbf{U}, \mathbf{Y}))$
- In 2.4a, $N = 175 + 42$ days if the shrinking horizon strategy is applied, and $N = 119$ days for the rolling horizon strategy.
- In 2.4a, the capacity of the healthcare system is set to 200.
- In 2.4a, H in this case, the mapping function of the SUBNET, h_θ.
- In 2.4a, the P penalty constant is set to 10^6.
- In 2.4a, the p penalty power constant is set to 2.
- In 2.4b, F represents the stepper function of the SUBNET, f_θ.
- In expression 2.4d, $G = 42$ days if the shrinking horizon strategy is applied, and $G = 0$ days for the rolling horizon strategy.
- In expression 2.4e, the holding time is set to $h = 7$ days.

Depending on the strategy applied, the values of N and G change during each iteration. In the case of the *shrinking horizon*, the value of G remains constant, while N decreases by 7 in each iteration. In the case of the *rolling horizon*, N remains constant, but as the initial time point reaches 175 days, G increases by 7.

The initial value $\hat{\mathbf{x}}_0$ also depends on the $\mathbf{U}$ and $\mathbf{Y}$ vectors updated in each iteration.

6.4 Simulation results

In the following, the results achieved will be presented. Firstly, the control was performed using the *shrinking horizon* strategy with a time horizon of 217 days. The average optimization time was 10.12 seconds.

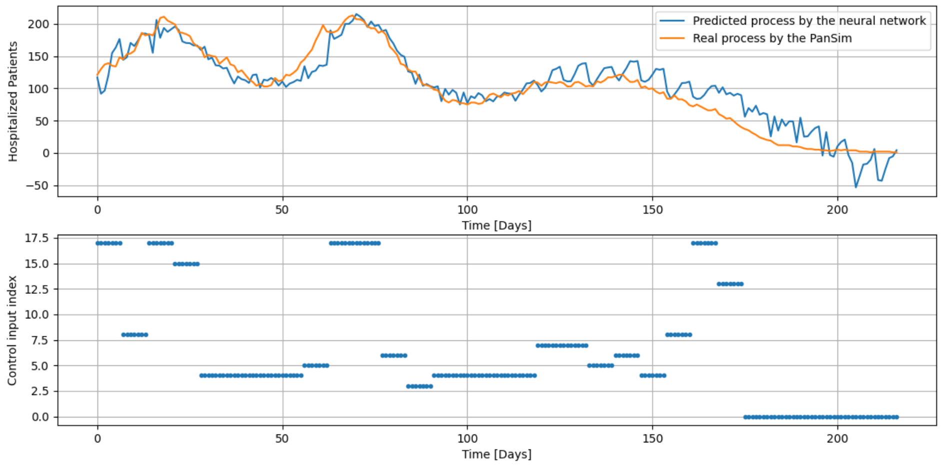

Figure 6.2: The SUBNET and the PanSim model with the *shrinking horizon* strategy in a closed-loop configuration.

Afterwards, the control was performed using the *rolling horizon* strategy with a time horizon of 119 days. The average optimization time was 12.57 seconds.

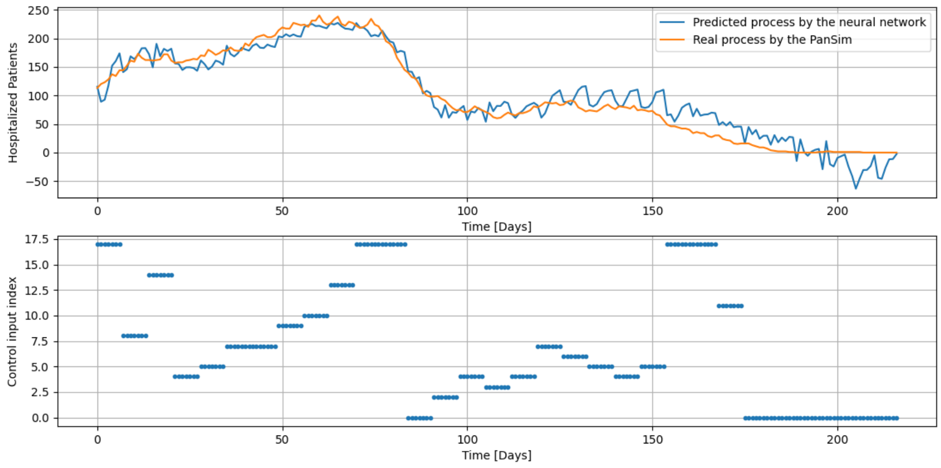

Figure 6.3: The SUBNET and the PanSim model with the *rolling horizon* strategy in a closed-loop configuration.

Based on the Figure 6.2 and Figure 6.3, we can conclude that the control was successful, as the number of hospitalized individuals does not significantly exceed the prescribed hospital capacity, and by the end of the control period, the pandemic is nearly subdued, with values approaching zero.

However, it should be noted that the tendencies predicted by the SUBNET differ significantly from those provided by PanSim. This results in the sawtooth pattern visible in Figure 6.3 around days 130-150. Here, the f_θ network of the SUBNET predicted an upward trend, while PanSim indicated a stagnating trend. After each 7-day period, the encoder updates the state for the controller based on PanSim's data, causing the output predicted by the SUBNET model to get closer to the actual output and the sawtooth pattern to emerge. A similar phenomenon can be observed in Figure 6.2 around days 130-150.

It can also be observed that, at nearly zero case numbers, the encoder network of the SUBNET estimates states that result in a negative number of cases, which is clearly unrealistic.

Chapter 7

Implementation

This section provides a detailed description of the development environments, tools, and libraries employed during the implementation process. Furthermore, it outlines the fundamental operation of the code, highlighting its core functionalities and structural design. During the implementation, it was ensured that the program is easily extendable with arbitrary model descriptions, and that they can be seamlessly integrated into the MPC algorithm with any strategy.

7.1 The Software and Hardware Tools Employed

The implemented program was written in Python (3.14.4) using Visual Studio Code, utilizing the following libraries and repositories.

The use of CasADi accelerates nonlinear optimization through algorithmic differentiation [2]. The *Matplotlib.pyplot* library was used for plotting the curves [3]. The *Numpy* library was used for defining simple expressions, such as vector representations, and processing the obtained values [4]. The *Scipy.integrate* library was responsible for solving the differential equations [6]. The *Torch* library was responsible for handling the neural networks [7]. Finally, the PanSim framework is open source on GitHub [5].

Additionally, the 'Ipopt' solver was used for optimization, which is an open-source software package designed to solve nonlinear, large-scale problems [1].

The control and optimization performed on the compartmental model were executed on a Dell laptop equipped with 8.00 GB of RAM and an Intel(R) Core(TM) i5-7300U @ 2.60GHz 2.70 GHz CPU, running Windows 11. The device does not include a dedicated graphics card.

Due to the computational demands of running PanSim simulations, access to the Titan server was granted by the Control Laboratory of the HUN-REN Institute for Computer Science and Control (SZTAKI). As a result, the simulations were executed via the PowerShell application.

The Titan server operates with 64 GB of RAM and is equipped with 8 Intel(R) Xeon(R) W-2123 CPUs running at 3.60 GHz, under a Linux operating system (Ubuntu 22.04.5).

7.2 Program structure

The foundation of the implemented code is as follows. The optimization problem is defined and managed by the `Problem` class. This class, in its constructor, accepts key parameters such as the time horizon over which the optimization problem is defined, the initial state, the holding time, and the model class describing the plant.

The `Problem` class constructs the appropriate constraints. Furthermore, the optimization problem can be solved using the specified solver. It is important to note that the problem must be initialized, meaning all decision variables must be assigned initial values.

The class also includes a time attribute, which stores the duration required to solve the problem. Until the problem is solved, this time attribute is uniformly set to zero.

A derived class of `Problem` is `Problem_With_Grace_Time`, which includes an additional parameter in its constructor: `gracetime`. This parameter specifies the duration at the end of the entire time horizon during which the intervention is set to zero. The definitions of the `Problem` class and its derived class `Problem_With_Grace_Time` are as follows:

```python
class Problem:
    def __init__(self, x0, time_horizont, holding_time, Model, margin):
        self.control_time = int(np.ceil(time_horizont / holding_time))
        self.dim = Model.dim
        self.x = cs.MX.sym('x', self.dim, time_horizont)
        self.u = cs.MX.sym('u', 1, self.control_time)
        self.y = cs.MX.sym('y', 1, time_horizont)
        self.system_step = Model.dynamic
        self.mapping = Model.map
        constraints_for_step_state = []
        constraints_for_output = []
        constraints_for_step_state.append(self.x[:, 0] - x0)
        penalty_weight = 1e6
        penalty_power = 2
        self.time = 0
        penalty = penalty_weight * cs.sumsqr(cs.fmax(self.y - hospital_capacity, 0)**penalty_power)
        self.objective = cs.sumsqr(self.u) + penalty

        for i in range(time_horizont):
            if i == time_horizont - 1:
                x_next = self.system_step(self.x[:, i].T, self.u[:, int(i / holding_time)].T)
                y_out = self.mapping(self.x[:, i].T)
                constraints_for_output.append(self.y[:, i] - y_out)
            else:
                x_next = self.system_step(self.x[:, i].T, self.u[:, int(i / holding_time)].T)
                y_out = self.mapping(self.x[:, i].T)
                constraints_for_step_state.append(self.x[:, i + 1] - x_next.T)
                constraints_for_output.append(self.y[:, i] - y_out)

        step_state = cs.vertcat(*constraints_for_step_state)
        step_output = cs.vertcat(*constraints_for_output)
        g = cs.vertcat(step_state, step_output)

        for i in range(self.control_time):
            g = cs.vertcat(g, self.u[:, i])
        for i in range(time_horizont):
            g = cs.vertcat(g, self.y[:, i])

        nlp = {'x': cs.vertcat(cs.vec(self.x), cs.vec(self.u), cs.vec(self.y)),
               'f': self.objective,
               'g': g}

        lbg = np.zeros((self.dim + 1) * time_horizont + self.control_time + time_horizont)
        ubg = np.zeros((self.dim + 1) * time_horizont + self.control_time + time_horizont)

        for i in range((self.dim + 1) * time_horizont + self.control_time + time_horizont):
            if i < (self.dim + 1) * time_horizont:
                lbg[i] = 0
                ubg[i] = 0
            else:
                if i < (self.dim + 1) * time_horizont + self.control_time:
                    lbg[i] = min_control_value
                    ubg[i] = max_control_value
                else:
                    lbg[i] = -100
                    ubg[i] = hospital_capacity + margin

        self.nlp = nlp
        self.floor_constraints = lbg
        self.ceilloing_constraints = ubg

    def get_solution(self, solver_type, x_init):
        solver = cs.nlpsol('solver', solver_type, self.nlp)
        start = time.time()
        solution = solver(lbg=self.floor_constraints, ubg=self.ceilloing_constraints, x0=x_init)
        end = time.time()
        self.time = end - start
        print(solver.stats()['return_status'])
        return solution

class Problem_With_Grace_time(Problem):
    def __init__(self, x0, time_horizont, grace_time, holding_time, model, margin):
        super().__init__(x0, time_horizont, holding_time, model, margin)

        for i in range(self.control_time):
            if i < self.control_time - np.ceil(grace_time / holding_time):
                self.floor_constraints[i + (self.dim + 1) * time_horizont] = min_control_value
                self.ceilloing_constraints[i + (self.dim + 1) * time_horizont] = max_control_value
            else:
                self.floor_constraints[i + (self.dim + 1) * time_horizont] = 0
                self.ceilloing_constraints[i + (self.dim + 1) * time_horizont] = 0

```

The CasADi library is used to define *decision variables*, which are arranged into vectors. As shown in the code snippet, *constraints* are assigned to these variables, with names representing their purpose (e.g., the *constraint* that determines the next state based on the previous state and the current control input is named *constraints_for_step_state*). In CasADi, these *constraints* must arrange the *decision variables* on one side, and an upper and lower bound must be specified. Equality *constraints* can be defined by setting both the upper and lower bounds to zero.

As shown in the code, a structure must be created that contains information related to the optimization problem. This includes the *decision variables*, the expression to be minimized, and the *constraints* applied to the *decision variables*. Subsequently, each *constraint* must have both a lower and an upper bound specified. In addition, the `Problem` class allows us to set hard *constraints* on the model output with a specified margin. This feature was not utilized during the simulations.

The `get_solution` method can be invoked on instances of the `Problem` class. By specifying the solver type as an attribute and providing the initialized *decision variables*, the optimization problem can be solved. During the solution process, the time required for solving is stored in the `time` attribute of the class.

The `Model` class, which must be provided to the `Problem` class, describes the system model. Its constructor requires the following parameters: the dimension of the model's state vector, the transition function for the state vectors, and the mapping function that relates the state vector to the system's output.

A derived class of the `Model` class is the `Plant` class, which can be interpreted as a description of a real system. In addition to the functionality of the `Model` class, it includes a `response` function. This function provides the system's output for an arbitrary-length input sequence starting from a given initial position. If required by the application, the dynamics can be perturbed with noise.

In addition, a `PanSim` class has been defined. Since the PanSim model is a 'black box' type of model, its dynamic description is unknown. The class constructor requires the PanSim simulator and the necessary encoder as inputs and updates the internal input-output attributes. Its attributes include the input-output pairs for the last 30 days, the encoder defined by `SUBNET`, and the interface class of the PanSim simulator itself. Similar to the `Plant` class, this class also has a `response` function, which provides the output corresponding to a given input sequence. Furthermore, it includes `get_initial_state` and `get_next_state` functions, which return the initial state for initialization and the next state based on the internal state of the `PanSim` class, respectively. The definitions of the `Model` class and its derived classes `Plant` and `Pansim` are as follows:

```python
class Model:
    def __init__(self,dimension, dynamic_for_one_step, output_mapping):
        self.dim = dimension
        self.dynamic = dynamic_for_one_step
        self.map = output_mapping
        if self.dynamic==system_step_neural:
            self.model_type = "neural"
            net_models = get_net_models()
            self.dynamic = partial(dynamic_for_one_step, net_models['f'])
            self.map = partial(output_mapping, net_models['h'])
            self.rounding=np.round
        if dynamic_for_one_step == runge_kutta_4_step:
            self.model_type = "compartmental"
            self.dynamic = lambda x, u: runge_kutta_4_step(dydt_casadi, x, u)
            self.rounding=rounding_for_compartmental
class Plant(Model):
    def __init__( self, dimension, dynamic_for_one_step, output_mapping):
        super().__init__( dimension, dynamic_for_one_step, output_mapping)
        if self.model_type == "compartmental":
            self.dynamic = lambda x, u: runge_kutta_4_step(dydt_numpy, x, u)
    def response(self,U,x,noise_dec):
        Y=[]
        for i in range(len(U)):
            x_next = self.dynamic( x, U[i])
            y_out= self.map(x)
            x=x_next
            if noise_dec==1 :
                if self.model_type == "compartmental":
                    noise = np.random.rand() * 2*800/real_population-800/real_population
                    x=x_next+noise
                if self.model_type == "neural":
                    noise = np.random.rand() * 0.025*2-0.025
                    x=x_next+noise
            else:

                x=x_next
            Y.append(np.squeeze(y_out))

        Y = np.array(Y)
        return [Y,x_next]

class PanSim:
    def __init__(self,simulator,encoder):
        self.simulator=simulator
        self.simulator.initSimulation(init_options)
        self.encoder=encoder
        self.Input=np.zeros(30)
        self.Output=np.zeros(30)
    def get_initial_state(self,U_init):
        results_agg = []
        inputs_agg = []
        run_options_agg = []
        for i in range (30):
            input_idx,run_options=U_init[i],input_sets[int(U_init[i])]
            results = self.simulator.runForDay(run_options)
            results_agg.append(results)
            inputs_agg.append(input_idx)
            run_options_agg.append(run_options)
        hospitalized_agg=get_results(results_agg)
        [uhist,yhist]=norm_and_unsqueeze(inputs_agg,hospitalized_agg)
        x0 = self.encoder(uhist, yhist)
        self.Input=inputs_agg
        self.Output=hospitalized_agg
        return x0.detach().numpy()
    def response(self,U,delta_time):
        results_agg = []
        inputs_agg = []
        run_options_agg = []
        for i in range (len(U)):
            input_idx,run_options=U[i],input_sets[int(U[i])]
            results = self.simulator.runForDay(run_options)
            results_agg.append(results)
            inputs_agg.append(input_idx)
            run_options_agg.append(run_options)
        hospitalized_agg=get_results(results_agg)
        self.Input = np.roll(self.Input, -delta_time)
        self.Input[-delta_time:] = inputs_agg

        self.Output = np.roll(self.Output, -delta_time)
        self.Output[-delta_time:] = hospitalized_agg
        return hospitalized_agg
    def get_next_state(self):
        [uhist,yhist]=norm_and_unsqueeze(self.Input,self.Output)
        x0 = self.encoder(uhist, yhist)
        x0=x0.detach().numpy()
        return x0
```

These classes form the foundation for the various algorithms, which have been implemented as standalone functions. The strategies are implemented under the names `rolling_MPC` and `shrinking_MPC`. These functions implement closed-loop control using the appropriate strategy. As input, they take the initial state of the system, the time duration for which the control is planned, the time duration for which a zero input is imposed at the end of the planned time horizon, as well as the attributes of the model used by the MPC and the actual system functioning as the plant. The `plant` attribute can be an instance of either the `Plant` or `Pansim` class. The implementation of `shrinking_MPC` is shown in the following code snippet:

```
def shrinking_MPC(noise_MPC,noise_plant,time_horizont,x,grace_time,model,plant,discr) :
    shrinking_time_horizont=time_horizont
    x_first=x
    time_step=holding_time
    U=np.empty((1,time_horizont))
    X_model=np.empty((model.dim,time_horizont))
    Y_model=np.empty((1,time_horizont))
    Y_real=np.empty((1,time_horizont))
    execution_times = []
    x_init=np.zeros((model.dim+1)*time_horizont+int(np.ceil((time_horizont)/holding_time)))

    for i in range(int(np.ceil(time_horizont/holding_time))):
        MyProblem=Problem_With_Grace_time(x_first.T,shrinking_time_horizont,grace_time,holding_time,model,margin)
        if noise_MPC:
            MyProblem.add_noise(shrinking_time_horizont)
        MySolution=MyProblem.get_soultion('ipopt',x_init)
        execution_times.append(MyProblem.time)
        [x_opt,u_opt,y_opt]=from_solution_to_x_u_y(MySolution,shrinking_time_horizont,model.dim)

        x_init=from_x_u_y_to_solution(x_opt,u_opt,y_opt,shrinking_time_horizont,model.dim)
        u_opt_extended=u_extended(u_opt,shrinking_time_horizont)
        if discr==1:
            u_opt_extended=model.rounding(u_opt_extended)

        U[:,i*time_step:+time_step*(i+1)]=u_opt_extended[:,:time_step]
        Y_model[:,i*time_step:time_step*(i+1)]=y_opt[:,:time_step]
        X_model[:,i*time_step:time_step*(i+1)]=x_opt[:,:time_step]
        shrinking_time_horizont=shrinking_time_horizont-time_step
        x_init=from_x_u_y_to_solution(x_opt[:,time_step:],u_opt[:,1:],y_opt[:,time_step:],shrinking_time_horizont,model
        if isinstance(plant, Plant):
            [Y,x_next]=plant.response(np.squeeze(u_opt_extended[:,:time_step]),x_first,noise_plant)

            x_first=x_next
        if isinstance(plant, PanSim):
            Y=plant.response(np.squeeze(u_opt_extended[:,:time_step]),time_step)
            x_next=plant.get_next_state()
            x_first=x_next
        Y_real[:,i*time_step:time_step*(i+1)]=Y[:time_step]


    Y_real=np.squeeze(Y_real)
    Y_model=np.squeeze(Y_model)
    U=np.squeeze(U)
    return [Y_real,Y_model,U,X_model,execution_times]
```

When implementing different strategies, it is crucial to properly initialize the optimization problem, as we are dealing with a nonlinear optimization problem. In the first iteration, all decision variables are initialized to zero uniformly. Except for the first iteration, the optimization problem is initialized with the solution obtained from the previous iteration.

The code is further supplemented with additional functions, such as those responsible for visualization and vector manipulation, which help maintain the organization and structure of the code (for example, visualization, .txt reader and writer functions).

The extensibility of the code is demonstrated by the fact that both the compartmental and neural models' optimization are carried out using the aforementioned classes. If a new model is to be applied, it is necessary to provide the description of the model's step function and mapping function.

7.3 The PanSim framework

This section will briefly present the functioning and usage of the PanSim framework. It will provide an overview of how PanSim is used and explain, from an implementation perspective, what measures correspond to the input values and what feedback PanSim returns during a simulation.

PanSim can be found on GitHub in the following repository: [5]. If Python is used as the programming language, navigate to the file `pansim_matlab_ReadMe_How_to_python.txt` in the mentioned repository and follow the instructions provided there.

Once the appropriate instructions have been executed, PanSim can be used. First, PanSim needs to be included as if it were any other module. In this thesis, it is done as follows: `import pyPanSim as sp`. After this, PanSim can be invoked.

To use PanSim, it is necessary to call the `SimulatorInterface` method and initialize the simulator. Subsequently, it is necessary to call the `initSimulation(init_options)` method to initialize the simulation. The initialization includes parameters such as the length of the intervention in days and the type of COVID-19 testing applied. A detailed description of these options can be found in the `README` file under the 'Input options' section in the repository [5].

As mentioned in Chapter 4, specific interventions correspond to each input. A detailed description of the interventions associated with each input can be found in the appendices (see A.1). This is based on the `Compute_beta.txt` file, located in the `matlab/ReadMe/` directory.

After each simulated day, PanSim returns a `result_avg` vector containing data related to the progression of the epidemic. A detailed description of this vector can be found in the `Compute_beta.txt` file located in the `matlab/ReadMe/` folder. Below, the most important return values will be highlighted.

- $NI =$ `result_avg[27]`: New cases of infection.
- $I1 =$ `result_avg[2]`: Number of presymptomatic individuals.
- $I2 =$ `result_avg[3]`: Number of asymptomatic individuals.
- $I3 =$ `result_avg[4]`: Number of infectious individuals.
- $I4 =$ `result_avg[5]`: Number of individuals with severe infections.
- $I5 =$ `result_avg[6]`: Number of hospitalized individuals.
- $I6 =$ `result_avg[7]`: Number of severely hospitalized individuals.
- $IM =$ `result_avg[37]`: Number of recovered individuals.

For control purposes, the sum of the elements $I5$ and $I6$ represents the number of hospitalized individuals by the COVID-19 virus. This sum indicates the system's response to a given control input.

Chapter 8

Conclusion and Outlook

As we have seen, it was possible to implement the control of the PanSim model using the MPC algorithm in a closed-loop system with the help of the SUBNET network, which was the main goal of this thesis.

It is also worth noting that the control of PanSim using the MPC algorithm is fundamentally a mixed integer non-linear problem. Although the problem was approached as a continuous one in this thesis, the results showed minimal deviation between the optimal input response and the response corresponding to the rounded inputs.

In future work, the implementation of so-called waiting time constraints could be considered, providing a more sophisticated solution compared to the rigid, weekly intervention-based approach. The essence of these constraints is that the intervention must be maintained for a minimum duration and can only be applied for a maximum duration. A detailed and exact description of these constraints can be found in [16].

Nevertheless, I would like to optimize the implemented code, and extend it with additional functionalities.

Acknowledgements

I would like to express my gratitude to Dr. Tamás Péni from the Control Laboratory of the HUN-REN Institute for Computer Science and Control (SZTAKI) for providing guidance during the writing of this thesis and for the opportunity to work at SZTAKI. Additionally, I would like to thank Dr. Márton Vaitkus for being my supervisor at the Budapest University of Technology and Economics (BME).

I would also like to thank Bendegúz Györök for his help in understanding and using the SUBNET and PanSim models.

Finally, I would like to express my thanks to my family for their support throughout my studies.

Bibliography

[1] *Ipopt Solver Documentation.* https://coin-or.github.io/Ipopt/.

[2] *CasADi Description.* https://web.casadi.org/docs/.

[3] *Matplotlib 3.9.2 Documentation.* https://matplotlib.org/stable/index.html.

[4] *Numpy Documentation.* https://numpy.org/doc/.

[5] *Pansim Framework.* https://github.com/khbence/pansim.

[6] *SciPy Documentation.* https://docs.scipy.org/doc/scipy/tutorial/integrate.html.

[7] *PyTorch Documentation.* https://pytorch.org/docs/stable/index.html.

[8] Mads Almassalkhi. Optimization and model-predictive control for overload mitigation in resilient power system. *ResearchGate*, pages 71–74, 2024. https://www.researchgate.net/publication/295276699_Optimization_and_Model-predictive_Control_for_Overload_Mitigation_in_Resilient_Power_Systems.

[9] Andrea Capannolo, Giovanni Zanotti, Michèle Lavagna, and Giuseppe Cataldo. Model predictive control for formation reconfiguration exploiting quasi-periodic tori in the cislunar environment. *ResearchGate*, 2022. https://www.researchgate.net/publication/362018740_Model_Predictive_Control_for_formation_reconfiguration_exploiting_quasi-periodic_tori_in_the_cislunar_environment.

[10] Bendegúz Györök and Tamás Péni. Pandemic model identification by subnet.

[11] Gergely Horváth, Gábor Szederkényi, and István Zoltán Reguly. Quantifying and comparing the impact of combinations of non-pharmaceutical interventions on the spread of covid-19. In *Mediterranean Conference on Control and Automation (MED)*, 2023.

[12] Florian Rösel Lukas Glomb, Frauke Liers. A rolling-horizon approach for multi-period optimization. *Optimization Online*, pages 4–6, 2021. https://optimization-online.org/2020/05/7809/.

[13] Movement of agents video. https://www.youtube.com/watch?v=OCfbHjLeCbY.

[14] Tamás Péni, Balázs Csutak, Gábor Szederkényi, and Gergely Röst. Nonlinear model predictive control with logic constraints for covid-19 management. *Springer*, 2020.

[15] Istvan Z. Reguly, David Csercsik, Janos Juhasz, Kalman Tornai, Zsofia Bujtar, Gergely Horvath, Bence Keomley-Horvath, Tamas Kos, Gyorgy Cserey, Kristof Ivan, Sandor Pongor, Gabor Szederkenyi, Gergely Rost, and Attila Csikasz-Nagy. Microsimulation based quantitative analysis of covid-19 management strategies. *PLOS COMPUTATIONAL BIOLOGY*, 2022.

[16] J.E. Serenoa, A. D'Jorgea, A. Ferramoscab, E.A. Hernandez-Vargasc, and A.H. Gonzáleza. Switched nmpc for epidemiological and social-economic control objectives in sir-type systems. *ScienceDirect*, 2023. https://www.sciencedirect.com/science/article/pii/S1367578823000652.

[17] Lieven Vandenberghe Stephen Boyd. *Convex Optimization*, pages 1–2. Cambridge University Press, 2004. https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf.

[18] M. Szecsi, B. Györök, A. Weinhardt-Kovács, G.I. Beintema, M. Schoukens, T. Péni, and R. Toth. Deep learning of vehicle dynamics.

[19] Wikipedia. Model predictive control. https://en.wikipedia.org/wiki/Model_predictive_control, .

[20] Wikipedia. Runge-kutta method. https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods, .

Appendix

A.1 Measures associated with each intervention input.

According to the notation system in [5], the following abbreviations and symbols will be used for interventions:

- **Testing:**
  - TPdef - 0.15% of the population is tested daily.
  - TP015 - 1.5% of the population tested daily.
  - TP035 - 3.5% of the population tested daily.
- **Closures:**
  - PLNONE - None.
  - PL0 - Mandatory closure of entertainment and hospitality businesses, and universities.
- **Curfew:**
  - CFNONE - None.
  - CF2000-0500 - Agents are not allowed to leave their homes between 8 p.m. and 5 a.m., unless they work night shifts at essential services.
- **School closure:**
  - SONONE - All children go to school.
  - SO12 - Only children below age 12 go to school.
  - SO3 - All schools and daycare facilities are closed.
- **Quarantine:**
  - QU0 - No quarantine, even for those diagnosed.
  - QU1 - Only the diagnosed individual is quarantined (this option was not available previously and is not in the table).
  - QU2 - Quarantine for those diagnosed and their household members.

  - QU3 - Similar to QU2, plus classmates if the diagnosed individual is a child, and some colleagues if the diagnosed individual is an adult worker.

- **Mask:**
  - MA1.0 - None.
  - MA0.8 - Mandatory mask-wearing at non-residential locations.

The control inputs correspond to the following sets of measures (u_i denotes $u = i$, where $i = 0, \ldots, 17$).

- u_0: [TPdef, PLNONE, CFNONE, SONONE, QU0, MA1.0]
- u_1: [TPdef, PL0, CFNONE, SONONE, QU0, MA1.0]
- u_2: [TPdef, PLNONE, CF2000-0500, SONONE, QU0, MA1.0]
- u_3: [TPdef, PLNONE, CFNONE, SO12, QU0, MA1.0]
- u_4: [TPdef, PLNONE, CFNONE, SO3, QU0, MA1.0]
- u_5: [TPdef, PLNONE, CFNONE, SONONE, QU2, MA1.0]
- u_6: [TPdef, PLNONE, CFNONE, SONONE, QU3, MA1.0]
- u_7: [TPdef, PLNONE, CFNONE, SONONE, QU0, MA0.8]
- u_8: [TP015, PLNONE, CFNONE, SONONE, QU2, MA1.0]
- u_9: [TP015, PLNONE, CFNONE, SONONE, QU3, MA1.0]
- u_{10}: [TP015, PLNONE, CFNONE, SO12, QU2, MA1.0]
- u_{11}: [TP015, PLNONE, CFNONE, SO3, QU2, MA1.0]
- u_{12}: [TP015, PLNONE, CFNONE, SO12, QU3, MA1.0]
- u_{13}: [TP015, PLNONE, CFNONE, SO3, QU3, MA1.0]
- u_{14}: [TP015, PLNONE, CFNONE, SONONE, QU2, MA0.8]
- u_{15}: [TP035, PLNONE, CFNONE, SONONE, QU3, MA0.8]
- u_{16}: [TP035, PL0, CFNONE, SO3, QU3, MA0.8]
- u_{17}: [TP035, PLNONE, CF2000-0500, SO3, QU3, MA0.8]