Department of Mechanical Engineering
Dynamics and control

Memory Saving Multi-Observer

A Comparative Analysis of Multi-Observer Architectures for Secure State Estimation

Julian Gootzen
Student number: 1676512

Supervisor:
dr. Michelle Chong

Mechanical Engineering Bachelor End Project

Eindhoven, January 24 2025

Contents

1 Problem Statement

Cyber physical systems (CPSs) are physical systems that are controlled by computer algorithms, actuator input is based on sensors that measure the state of the physical system. CPSs are often large, geographically dispersed, safety-critical systems that play a crucial role in modern infrastructure, such as power grids or water distribution systems. There are two categories in which most cyberattacks on CPSs can be classified: Denial of Service attakcs (DoS) and deception attacks [9]. A DoS attack aims to consume a significant amount of the victims resources: CPU cycles, memory bandwith or network bandwidth. The victim is then not able to use these resources for the intended purpose, for CPSs this often means that sensors do not provide any readings [21]. A deception attack aims to completely take over a sensor, the attacker injects false data into the system [18], such an attack is also known as a false data injection (FDI) attack. A system that is under an FDI attack still takes readings from the attacked sensor and makes decisions based on those readings. An FDI attack cannot only cause disruption of service, it can inject false data that aims to maximize long-term damage to a system.

These types of attacks on CPSs can have severe consequences on our (industrial) infrastructure. The Industroyer and Stuxnet attacks serve as infamous examples [14], [15]. Stuxnet first lay dormant recording normal operating conditions, later it would manipulate the sensor data and cause damage to the industrial equipment. During the periods that Stuxnet manipulated sensors, it would replay the recorded data which was sent to the verification systems and were thus unaware of any abnormal operating conditions [10]. There are a number of strategies that can be employed to prevent such an attack, the obvious one: preventing a malicious actor from gaining access to your system in the first place. Another approach, the focus of this report, is designing a system that is robust with respect to attacks.

The control systems within CPSs often need information about all state variables used to describe the system. Measuring all state variables can be expensive, impractical or even impossible [5]. An *observer* or *state estimator* can be used to create an estimate of all state variables from the subset of measured state variables. Considering the previously discussed possibility of attacks on these systems, it is strongly desirable to have the ability of providing a secure state estimate: a correct state estimate constructed from a set of noisy sensors under attack [19]. This problem is also known as *secure state estimation* (SSE).

This report will focus on the SSE method presented in [6] and [7], where an observer-based estimator is introduced. Multiple observers are created and observe the system in parallel, such an observer will be referred to as a *multi-observer* (MO) in this report. An MO can be implemented by storing each observer separately, this will be referred to as a *conventional multi-observer* (CMO). This approach has an implementation bottleneck due to the large number of observers required to provide SSE, requiring large amounts of memory. A strategy to reduce memory usage is presented in [8], where a common state is shared between all observers. An observer with this architecture will be referred to as a *state-sharing multi-observer* (SSMO). The main aim of this BEP is to provide a comparison between the two multi-observers based on memory usage.

In Chapter 2 a mass-spring-damper system will be introduced, this system will serve as a case study throughout the report. Chapter 3 discusses some important concepts regarding state estimation, such as observability and a single state estimator. These concepts underlie the MOs that are in the rest of the report. Chapter 4 is the first chapter that discusses the CMO, the state estimates are constructed and, more importantly, the selection procedure that governs selecting the final state estimate is laid out. Two variants of a CMO are discussed, both MO architectures are explained and a size estimation is presented. Chapter 5 discusses the SSMO and its architecture and compares the CMOs to the SSMO in terms of memory usage. In the final chapter, Chapter 7, the nonlinear extension to the MO will be investigated. It is not an exhaustive discussion of nonlinear MOs, but serves as a brief look at what happens when a standard nonlinearity is applied on an MO.

2 System definition

Let us define the following continuous, nonlinear, time invariant, multiple-input multiple-output system:

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t) + E\phi(x)\\ y_i(t) &= C_i x(t) + Du(t) + v_i(t) + \tau_i(t) \quad i \in \mathcal{N} = \{1, 2, \dots, N_O\}.\end{aligned} \tag{1}$$

Where $x \in \mathbb{R}^{n_x}$, $y \in \mathbb{R}^{n_y}$, $u \in \mathbb{R}^{n_u}$ and the nonlinearity $\phi(x) \in \mathbb{R}^{n_\phi}$. Each i indicates a single output, so each y_i corresponds to a $C_i \in \mathbb{R}^{1 \times n_x}$. The variables without subscript y and C, denote all y_i and C_i stacked on top of each other $y = \begin{bmatrix} y_1 & y_2 & \cdots & y_{n_y} \end{bmatrix}^T$ and $C = \begin{bmatrix} C_1^T & C_2^T & \cdots & C_{n_y}^T \end{bmatrix}^T$ Let us consider the scenario where an attacker attacks a number of outputs y_i, where the attacker then has full control over the attack signal τ_i. No assumptions are made about τ_i, it can be any unbounded signal. Let us denote the set of attacked outputs as

$$\mathcal{M} \subset \mathcal{N}. \tag{2}$$

Figure 1 shows an example where $\mathcal{N} = \{1, 2, 3, 4, 5\}$ and $\mathcal{M} = \{2, 5\}$.

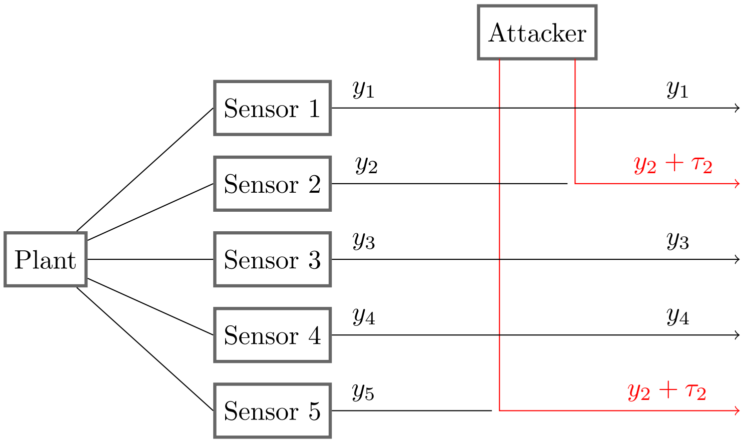


Figure 1: A system with 2 out of 5 outputs attacked

2.1 Case study: multi-mass-spring-damper system

Let us now introduce the system that will be used as a case study: a mass-spring-damper system. This system consists of b mass-spring-dampers in series. Where $x_a \in \mathbb{R}, a = 1, 2, \dots, b$ denotes the relative position with respect to the previous mass. So the absolute position of a block with respect to the fixed-world is

$$\sum_{i=1}^{a} x_i. \tag{3}$$

The other variables denote the following: $k_a \in \mathbb{R} > 0$ is the linear spring constant, $\zeta_a \in \mathbb{R} > 0$ is the nonlinear spring constant, $c_a \in \mathbb{R} \geq 0$ is the damping constant and F_a is a force acting on the mass.

We will now derive the state-space form of the system as drawn in Figure 2. We will start by deriving the equation of motion for the first $b - 1$ masses. The last mass has a different equation of motion because there is only a single other mass that enacts a force on the mass. The sum of forces on a single mass is

$$m_a \ddot{x}_a = F_a - F_s(x_a, k_a, \zeta_a) - F_d(\dot{x}_a, c_a) + F_s(x_{a+1}, k_{a+1}, \zeta_{a+1}) + F_d(\dot{x}_{a+1}, c_{a+1}). \tag{4}$$

Let us now define the spring and damping forces $F_s(x_a, k_a, \zeta_a)$ and $F_d(\dot{x}_a, c_a)$ respectively

$$F_s(x_a, k_a, \zeta_a) = k_a x_a + k_a \zeta_a^2 x_a^3, \quad F_d(\dot{x}_a, c_a) = c_a \dot{x}_a.$$

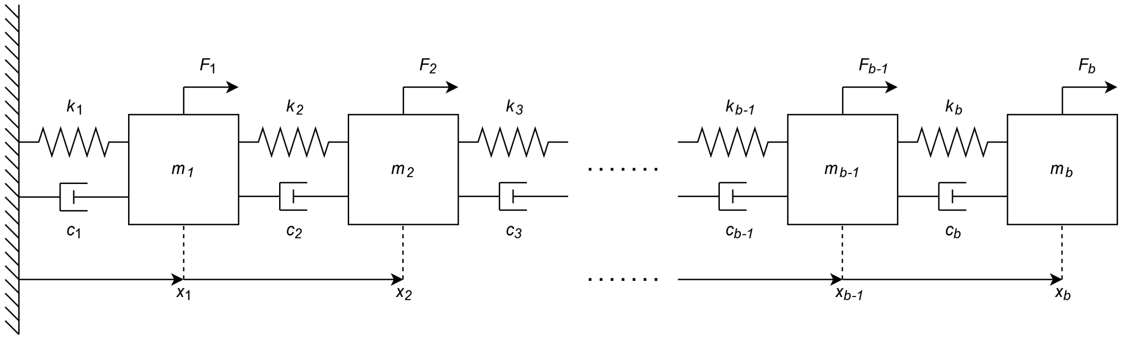


Figure 2: Multi-mass-spring-damper system for b mass-spring-dampers in series.

The formula for the nonlinear spring force is taken from the hardening spring example in [13, Section 1.2.3], the damping force is linear. The expressions for the spring force and damping force will be written as $F_s(x_a)$ and $F_d(\dot{x}_a)$ respectively, where the constants k_a, ζ_a and c_a are implicitly passed to the function. F_s will be separated into a linear and nonlinear part as

$$\begin{aligned} F_s(x_a) &= F_s^L(x_a) + F_s^{NL}(x_a) \\ F_s^L(x_a) &= k_a x_a \quad ,F_s^{NL}(x_a) = k_a \zeta_a^2 x_a^3 \end{aligned} \tag{5}$$

We will now apply Equation (4) on the first mass, using the split up F_s as in Equation (5)

$$m_1\ddot{x}_1 = u_1 - k_1 x_1 - c_1\dot{x}_1 + k_2 x_2 + c_2\dot{x}_2 - F_s^{NL}(x_1) + F_s^{NL}(x_2), \quad u_1 = F_1,$$

which can be rewritten into

$$\ddot{x}_1 = -\frac{k_1}{m_1}x_1 - \frac{c_1}{m_1}\dot{x}_1 + \frac{k_2}{m_1}x_2 + \frac{c_2}{m_1}\dot{x}_2 + \frac{u_1}{m_1} - \frac{F_s^{NL}(x_1)}{m_1} + \frac{F_s^{NL}(x_2)}{m_1}. \tag{6}$$

Doing this for the second mass gives the following result:

$$\ddot{x}_2 = -\frac{k_2}{m_2}x_2 - \frac{c_2}{m_2}\dot{x}_2 + \frac{k_3}{m_2}x_3 + \frac{c_3}{m_2}\dot{x}_3 + \frac{u_2}{m_2} - \frac{F_s^{NL}(x_2)}{m_1} + \frac{F_s^{NL}(x_3)}{m_1}.$$

which can be generalised into

$$\ddot{x}_a = -\frac{k_a}{m_a}x_a - \frac{c_a}{m_a}\dot{x}_a + \frac{k_{a+1}}{m_a}x_{a+1} + \frac{c_{a+1}}{m_a}\dot{x}_{a+1} + \frac{u_a}{m_a} - \frac{F_s^{NL}(x_a)}{m_a} + \frac{F_s^{NL}(x_{a+1})}{m_a}. \tag{7}$$

which holds for $a \in \{2, 3, \dots, b-2, b-1\}$. Now only the last mass remains

$$\ddot{x}_b = -\frac{k_b}{m_b}x_b - \frac{c_b}{m_b}\dot{x}_b + \frac{u_b}{m_b} - \frac{F_s^{NL}(x_b)}{m_b} \tag{8}$$

Let us now define the state vector x as in Equation (1) as

$$x = \begin{bmatrix} x_1 & \dot{x}_1 & x_2 & \dot{x}_2 & \cdots & x_b & \dot{x}_b \end{bmatrix}^T \tag{9}$$

Equations (6),(7) and(8) can now be used to construct the A and B matrices in equation (1)

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ -\frac{k_1}{m_1} & -\frac{c_1}{m_1} & \frac{k_2}{m_1} & \frac{c_2}{m_1} & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{k_2}{m_2} & -\frac{c_2}{m_2} & \frac{k_3}{m_2} & \frac{c_3}{m_2} & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & -\frac{k_b}{m_b} & -\frac{c_b}{m_b} \end{bmatrix}_{n_x \times n_x} \tag{10}$$

and

$$B = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ \frac{1}{m_1} & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{m_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & \frac{1}{m_b} \end{bmatrix}_{n_x \times n_u} . \tag{11}$$

Let us now derive the nonlinear contributions ϕ and E from equation (1) and from equations (6),(7),(8). First we need to define the intermediary matrix

$$H_x = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 \end{bmatrix}_{n_\phi \times n_x} , \tag{12}$$

which extracts the first n_ϕ relative positions from x performing the multiplication Hx and where n_ϕ denotes the 'size' of the nonlinearity. Performing the function ϕ on a vector implies performing F_s^{NL} on every element

$$\phi(H_x x) = \begin{bmatrix} F_s^{NL}(x_1) & F_s^{NL}(x_2) & \cdots & F_s^{NL}(x_{n_\phi}) \end{bmatrix}^T_{1 \times n_\phi}$$

and

$$E = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ -\frac{1}{m_1} & \frac{1}{m_1} & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & -\frac{1}{m_2} & \frac{1}{m_2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & -\frac{1}{m_b} \end{bmatrix}_{n_x \times n_\phi} \tag{13}$$

can be defined. We will now construct the C matrix, in order to do so let us note that only the absolute positions (3) of each mass are measured. This leads to

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 1 & 0 & \cdots & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & \cdots & 1 & 0 & 1 & 0 \end{bmatrix}_{n_y \times n_x} \tag{14}$$

where each row is a single output. In system as in Figure 2 the D matrix is equal to 0.

Example 1. *Let us define a linear system with two masses and the properties: $m_i = 1$[kg], $k_i = 15$[N/m] and $c_i = 2$[Ns/m] for $i = 1, 2$. The system matrices are as in Equations* (10),(11),(14)

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -15 & -2 & 15 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -15 & -2 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}, \quad D = 0. \tag{15}$$

3 State estimation

In this chapter the motivation for full state estimation is discussed. Secondly the concept of observability is discussed, a concept which will turn out to be very relevant to all observers that are to be used later. Starting with the single observer, that is constructed in this chapter.

3.1 Motivation for full state estimation

Let us first discuss why there is a need for state estimation, consider the continuous linear time invariant system

$$\dot{x}(t) = Ax(t) + Bu(t), \quad y(t) = Cx(t) + Du(t). \tag{16}$$

The solution to this system can be written as

$$\begin{aligned} x(t) &= e^{tA}x_0 + \int_0^t e^{(t-\tau)A}Bu(\tau)d\tau \\ y(t) &= Ce^{tA}x_0 + \int_0^t Ce^{(t-\tau)A}Bu(\tau)d\tau + Du(t). \end{aligned}$$

where $x_0 = x(0)$ [11, Equation. 6.4]. When there is no input, $u(t) = 0$ for $t \geq 0$, system (16) reduces to

$$\dot{x}(t) = Ax(t), \quad y(t) = Cx(t)$$

which has the solution

$$x(t) = e^{tA}x_0, \quad y(t) = Ce^{tA}x_0. \tag{17}$$

Let us now consider the definition of Lyapunov stability as in [11, Theorem 8.2], which states that a system is stable if and only if all the eigenvalues $\lambda_i, i = 1, 2, \ldots, n$ of A have strictly negative real parts. A system or a matrix that satisfies those requirements is called stable. The matrix exponential in equation (17) computed by writing the matrix A into its Jordan normal form $J = PAP^{-1}$, where P is a change of basis matrix. The solution can then be written as

$$x(t) = P^{-1}e^{tJ}Px_0 \tag{18}$$

[11, Section 7.3]. The matrix J takes the following form

$$J = \begin{bmatrix} J_1 & 0 & \cdots & 0 \\ 0 & J_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & J_l \end{bmatrix} \tag{19}$$

where each J_i is a Jordan block that has the form

$$J_i = \begin{bmatrix} \lambda_i & 1 & 0 & \cdots & 0 \\ 0 & \lambda_i & 1 & \cdots & 0 \\ 0 & 0 & \lambda_i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_i \end{bmatrix}, \quad i = 1, 2, \ldots, l. \tag{20}$$

The size of each Jordan block J_i is determined by the algebraic and geometric multiplicities of the eigenvalues of matrix A. The algebraic multiplicity of an eigenvalue indicates the number of times an eigenvalue appears. Or in other words, the number of duplicate eigenvalues. The geometric multiplicity indicates the number of independent eigenvector associated with a certain eigenvalue. The number of Jordan blocks for each eigenvalue is equal to the geometric multiplicity of that eigenvalue. An eigenvalue with an algebraic multiplicity of 3 and a geometric multiplicity of 2 corresponds to one 1 × 1 and one 2 × 2 Jordan block [11, Section 7.1]. Substituting equation (20) into (19) gives the full Jordan matrix. Which in turn can be substituted into equation (18). Using the Equations as in [11, Section 7.3] shows that if A is stable, $x \to 0$ as $t \to \infty$.

Example 2. *Let us take a closer look at the stability of the system presented in Example 1 and Equation (15). The eigenvalues and eigenvectors of the matrix A are as in Table 1. $\lambda_1 = \lambda_3$ and $\lambda_2 = \lambda_4$ so the algebraic multiplicity of both eigenvalues is 2. $\mathbf{v}_1 = -\mathbf{v}_3$ and $\mathbf{v}_2 = -\mathbf{v}_4$ so the geometric multiplicity of both eigenvalues is 1, since the eigenvectors are dependent on each other.*

i	1	2	3	4
λ_i	$-1+3.7417i$	$-1-3.7417i$	$-1+3.7417i$	$-1-3.7417i$
$\mathbf{v}_i$	$\begin{bmatrix}0.0645+0.2415i\\-0.9682\\0\\0\end{bmatrix}$	$\begin{bmatrix}0.0645-0.2415i\\-0.9682\\0\\0\end{bmatrix}$	$\begin{bmatrix}-0.0645-0.2415i\\0.9682\\0\\0\end{bmatrix}$	$\begin{bmatrix}-0.0645+0.2415i\\0.9682\\0\\0\end{bmatrix}$

Table 1: Eigenvalues and eigenvectors of the A matrix in (15)

This leads to the following change of basis matrix

$$P = \begin{bmatrix} 0.2679+1.0022i & 0.5000+0.0620i & 0.2679-1.0022i & 0.5000-0.0620i \\ 3.4821-2.0045i & 0.9306i & 3.4821+2.0045i & 0.9306i \\ 0 & 0.5000+0.1336i & 0 & 0.5000-0.1336i \\ 0 & 2.0045i & 0 & 2.0045i \end{bmatrix}$$

and the Jordan normal form

$$J = \begin{bmatrix} -1.0000-3.7417i & 1 & 0 & 0 \\ 0 & -1.0000-3.7417i & 0 & 0 \\ 0 & 0 & -1.0000+3.7417i & 1 \\ 0 & 0 & 0 & -1.0000+3.7417i \end{bmatrix}.$$

Let us choose the following initial conditions $x(0) = x_0 = \begin{bmatrix}0.3 & -0.1 & 0.5 & 0.2\end{bmatrix}^T$ and calculate the right-hand side of Equation (18) for a number of values of $0 \leq t \leq 5$, the results are plotted in Figure 3. The calculations and plot have been made with Matlab, the script can be found in Appendix C.

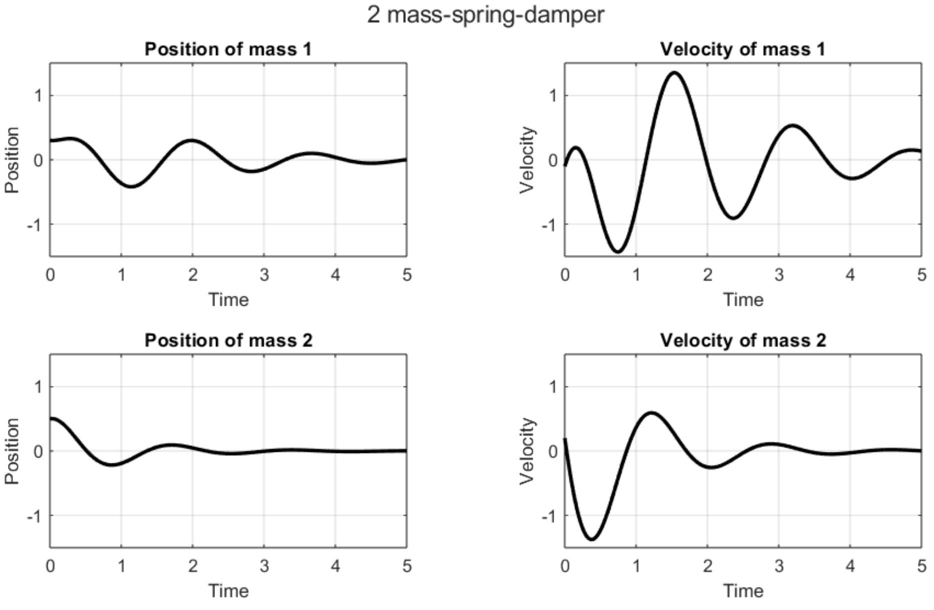


Figure 3: Time response of a double mass-spring-damper system calculated using Equation (18).

If the matrix A is not stable, but stability properties are still desired. The control law

$$u = Kx \implies x = Ax + BKx \implies \dot{x} = (A + BK)x \tag{21}$$

can be used to stabilize the system, if and only if the system (16) is stabilizable [11, Theorem 14.5]. As can be seen in Equation (21), the control law requires measuring the full state x, which can be very costly or not possible to measure [5]. Figure 4 shows a simple feedback controlled system, where it can also be seen that y feeds into the controller and can thus only stabilize the system when $y = x$. So we will now work towards creating a state estimate $\hat{x} \to x$ as $t \to \infty$.

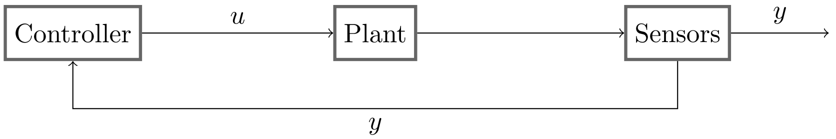


Figure 4: Feedback control system

3.2 Observability

We now aim to reconstruct the full state x from the output y and the input u. The following derivation is based on [20], consider system (16) where x is unknown and y and u are known. The derivatives of y

$$\begin{aligned}
y &= Cx + Du \\
\dot{y} &= C\dot{x} + Du = CAx + CBu + D\dot{u} \\
\ddot{y} &= CA\dot{x} + CBu + Du = CA^2x + CABu + CB\dot{u} + D\ddot{u} \\
&\vdots \\
y^{(n_x)} &= CA^{n_x}x + CA^{n_x-1}Bu + CA^{n_x-2}B\dot{u} + \cdots + CABu^{(n_x-2)} + CBu^{(n_x-1)} + Du^{(n_x)}
\end{aligned}$$

will be used to reconstruct x from the derivatives of y and u. These derivatives of y can be combined into

$$\mathbf{y} = \mathcal{O}x + \mathcal{K}\mathbf{u},$$

where

$$\mathbf{y} = \begin{bmatrix} y \\ y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n-1)} \end{bmatrix} \quad \text{and} \quad \mathbf{u} = \begin{bmatrix} u \\ u^{(1)} \\ u^{(2)} \\ \vdots \\ u^{(n-1)} \end{bmatrix},$$

are vectors containing the derivatives of the output and the input,

$$\mathcal{O} = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix} \tag{22}$$

is the *observability matrix* and

$$\mathcal{K} = \begin{bmatrix} D & 0 & 0 & \dots & 0 \\ CB & D & 0 & \dots & 0 \\ CAB & CB & D & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ CA^{n-2}B & CA^{n-3}B & CA^{n-4}B & \dots & D \end{bmatrix}.$$

Isolating x results in

$$x = \mathcal{O}^{-1}(\mathbf{y} - \mathcal{K}\mathbf{u}). \tag{23}$$

As can be seen in (23) observability matrix $\mathcal{O}$ is required to be invertible to reconstruct x from the time derivatives of y and u. If a system's A and C matrix fulfil this requirement, the system can be described as *observable*. The invertibility requirement is equivalent to the statement that the matrix $\mathcal{O}$ needs to be full rank [16, Section 2.9], this theorem that can also be found in [3, Corollary 3.8]. The implication of this is that it is only possible to reconstruct the state x if the pair (A, C) is observable. Actually reconstructing the state requires a system without noise and perfect knowledge on all system parameters, which is unrealistic. A realistic observer will be presented in Subsection 3.3.

Example 3. *Let us work out an example on the system presented in Example 1 where only the position of the first mass is measured. This leads to the following system matrices:*

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -15 & -2 & 15 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -15 & -2 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}.$$

We now compute the observability matrix

$$\mathcal{O} = \begin{bmatrix} C \\ CA \\ CA^2 \\ CA^3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -15 & -2 & 15 & 2 \\ 30 & -11 & -60 & 7 \end{bmatrix}$$

which has rank 4 *and so the pair* (A, C) *is observable.*

Example 3 considers a system with only one output, we will now investigate the case with multiple outputs. Let us expand the observability matrix (22) for a system where $n_y > 1$, a system with more than one output.

$$\mathcal{O} = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1} \end{bmatrix} = \begin{bmatrix} c_1 \\ \vdots \\ c_{n_y} \\ c_1 A \\ \vdots \\ c_{n_y} A \\ \vdots \\ c_1 A^{n-1} \\ \vdots \\ c_{n_y} A^{n-1} \end{bmatrix}_{n_x n_y \times n_x}.$$

The same rank condition still holds, the system is fully observable as long as the $\mathcal{O}$ matrix is full rank. This implies that certain combinations of sensors that do not suffice on their own can make a system fully observable [3, Section 3.4 D 2].

3.3 Single observer

Let us now reconstruct the full state of system (16) from y and u by defining the state estimator as in [11, Section 16.5], a copy of the original system

$$\dot{\hat{x}} = A\hat{x} + Bu, \quad \hat{y} = C\hat{x} + Du. \tag{24}$$

We now define the *state estimation error*

$$e = \hat{x} - x \tag{25}$$

which we differentiate and substitute equations (16) and (24) into to give

$$\dot{e} = \dot{\hat{x}} - \dot{x} = A\hat{x} + Bu - Ax - Bu = Ae.$$

We can now conclude that $e \to 0$ as $t \to \infty$ if the matrix A is a stability matrix. Let us now define a state estimator that provides an asymptotically correct state estimate even when A is not a stability matrix as in [11, Section 16.5]

$$\dot{\hat{x}} = A\hat{x} + Bu + L(\hat{y} - y), \quad \hat{y} = C\hat{x} + Du. \tag{26}$$

We now perform the same analysis on the derivative of the state estimation error

$$\dot{e} = A\hat{x} + Bu + L(C\hat{x} + Du - Cx - Du) - Ax - Bu = (A + LC)e$$

which is similar to the solution in (17). From which we can conclude that if $A + LC$ is a stability matrix $e \to 0$ as $t \to \infty$. Figure 5 shows the observer placed between the sensors and the controller

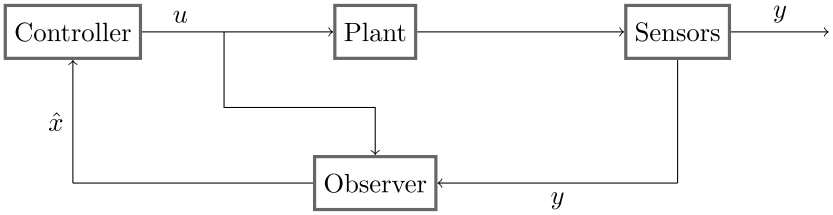


Figure 5: An observer in a control system

It is important to note that no restrictions are set on the structure of y and therefore C as long as the pair (A, C) is observable.

Let us now extend the observer (26) to also correctly estimate systems with a nonlinear contribution $\phi(H_x x)$. Consider the system

$$\dot{x} = Ax + Bu + E\phi(H_y y), \quad y = Cx + Du,$$

where $H_y \in \mathbb{R}^{n_\phi \times n_y}$ is a matrix that extracts usable data for the nonlinearity from the output, analogous to $H_x \in \mathbb{R}^{n_\phi \times n_x}$ as in Equation (12). The observer will be constructed as

$$\dot{\hat{x}} = A\hat{x} + Bu + E\phi(H_y y) + L(\hat{y} - y), \quad \hat{y} = C\hat{x} + Du. \tag{27}$$

Note that the input to the nonlinearity is y. Let us now substitute these definitions into the derivative of the error $e = \hat{x} - x$

$$\begin{aligned} \dot{e} = \dot{\hat{x}} - \dot{x} &= A\hat{x} + Bu + E\phi(H_y y) + L(\hat{y} - y) - Ax - Bu - E\phi(H_y y) \\ &= (A + LC)e + E(\phi(H_y y) - \phi(H_y y)) \\ &= (A + LC)e. \end{aligned}$$

It is now evident why $\phi(H_y y)$ is used in equation (3.3), otherwise $e \to 0$ as $t \to \infty$ cannot be guaranteed. Because $\phi(H_y y)$ does not necessarily equal $\phi(H_x x)$, $\phi(H_y y) - \phi(H_x x)$ does not necessarily equal 0. This does set certain requirements on y, variables that affect the nonlinearity need to be able to be reconstructed from y. For the multi mass-spring-damper system as in Chapter 2 this means that all positions $x_a, a = 1, 2, \dots, b$ need to be measured directly.

3.4 Eigenvalue placement

Since we have full control over L, we will now discuss methods to place eigenvalues of $A + LC$ at desired locations and what constraints are limiting our control over these eigenvalues. Let us first connect the concept of observability to eigenvalue placement.

Theorem 1 (Observer eigenvalues)**.** *If the pair (A, C)(16) is observable there exists an $L \in \mathbb{R}^{n \times m}$ that influences all eigenvalues of $A + LC$.*

Proof. This proof is based on [3, Section 4.2]. Suppose that the pair (A, C) is not fully observable and that all eigenvalues of $A+LC$ have been influenced by L, it will be shown that this leads to a contradiction. There exists a similarity transformation that separates the observable from the unobservable part: the observable decomposition [11, Section 16.1]. Specifically there exists a matrix Q that

$$\begin{aligned} Q^{-1}(A+LC)Q = Q^{-1}AQ + Q^{-1}LCQ &= \begin{bmatrix} A_o & 0 \\ A_{21} & A_u \end{bmatrix} + \begin{bmatrix} L_1 \\ L_2 \end{bmatrix} \begin{bmatrix} C_o & 0 \end{bmatrix} \\ &= \begin{bmatrix} A_o + L_1 C_o & 0 \\ A_{21} + L_2 C_o & A_u \end{bmatrix} \end{aligned}$$

where the pair (A_o, C_o) is observable. A similarity transformation leaves the eigenvalues unchanged [16, Section 5.2]. The lower triangular structure of the matrix implies that the eigenvalues of A_u remain the same, which leads to a contradiction: not all the eigenvalues of $A + LC$ have been influenced. Thus, if the pair (A, C) is observable there exists an L that can influence all eigenvalues of $A + LC$. □

An analytical method to place the eigenvalues at arbitrary locations in a MIMO system is presented in [3, Section 4.2 B], where the system is first transformed into *observer form* by a similarity transformation. After this transformation the deriving the matrix L is convenient. Another option is numerically deriving L. Matlab provides the function `place`, based on the algorithm presented in [12]. All eigenvalue placement in this report will be done using this function, the code is discussed in Chapter 6. The downside of using this numerical method is the numerical inaccuracy, numerically placed eigenvalues are placed within some tolerance. This has some implications for the models developed later in the report, especially larger systems suffer from inaccuracies.

The remaining question is what eigenvalues to select. Choosing a high gain L matrix will result in quick error suppresion and, unfortunately, high sensitivity to noise. A low gain observer will have the opposite characteristics. The L matrix can also be selected by solving an LQR problem [5].

4 Conventional multi-observer

In this section the CMO will be introduced and two implementations will be discussed, it is named *conventional* to differentiate it from the state-sharing variant discussed in Chapter 5. Let us first further elaborate on the what the CMO should achieve. Consider the case where a 'single'-observer (26) is employed to construct the state estimate of system (1), where $w = 0$ and $v = 0$ (no noise). When an attacker gains control over some of the outputs y_i and attacks it with the attack signal τ_i. Performing the same analysis on the derivative of error (25) leads to

$$\begin{aligned} \dot{e} = \dot{\hat{x}} - \dot{x} &= A\hat{x} + Bu + L(C\hat{x} - Cx - \tau) - Ax - Bu \\ &= (A + LC)e - L\tau, \end{aligned}$$

which does not guarantee $e \to 0$ as $t \to \infty$, since the attacker has full control over τ. The CMO aims to still provide a correct state estimate even when a subset $\mathcal{M} \subset \mathcal{N}$ outputs are under attack, where $2|\mathcal{M}| < |\mathcal{N}|$, with $\mathcal{M}$ as in Equation (2). The MOs discussed in this chapter will deal with linear systems, considerations regarding nonlinear MOs can be found in Chatper 7.

4.1 Constructing the state estimates

The multi-observer described in this section is based on [6, Section 3B]. This CMO is able to correctly estimate the state when up to $N_M = |\mathcal{M}|$ outputs are under attack. N_M is required to satisfy

$$2N_M < N_O, \tag{28}$$

in other words: more than half the sensors need to remain attack free for all $t \geq 0$. Consider system (1) with $N_O = n_y$ outputs and no nonlinearity (i.e. $\phi(x) = 0 \ \forall \ x \in \mathbb{R}^{n_x}$)

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y_i &= C_i x + Du + v_i + \tau_i \quad i \in \{1, 2, \dots, N_O\}. \end{aligned}$$

Let us define

$$\mathcal{J} \subset \{1, 2, \dots, N_O\}, \quad \mathcal{P} \subset \{1, 2, \dots, N_O\} \tag{29}$$

as all subsets of $\mathcal{N}$ with $J = N_O - N_M$ and $P = 1$ elements respectively. In theory P can also be taken as any value smaller than J, we restrict ourselves to systems that are fully observable trough any single output $y_i = C_i x$. The cardinalities of $\mathcal{J}$ and $\mathcal{P}$ are

$$N_J = |\mathcal{J}| = \binom{N_O}{J} = \binom{N_O}{N_O - N_M}, \quad N_P = |\mathcal{P}| = \binom{N_O}{P} = N_O. \tag{30}$$

For example, $N_O = 17$ and $N_M = 8$ ($N_O > 2N_M$) leads to $N_J = 24,310$ and $N_P = 17$. We now define the $j = 1, 2, \dots, N_J$ *J-observers* as

$$\begin{aligned} \dot{\hat{x}}_j^{\mathcal{J}} &= A\hat{x}_j^{\mathcal{J}} + Bu + L_j^{\mathcal{J}}(\hat{y}_j^{\mathcal{J}} - y_j) \\ \hat{y}_j^{\mathcal{J}} &= C_j^{\mathcal{J}} \hat{x}_j^{\mathcal{J}} + Du \end{aligned}$$

where j is a single combination out of all outputs and L_j is an output injection gain that can be chosen in such a way to place the eigenvalues at a desired location if and only if the pair (A, C_j) is observable.

Example 4. *Consider the system as in Equation* (1) *where* $N_O = 5$ *and* $N_M = 2$*. These values indicate that* $J = 5 - 2 = 3$*, which leads to the following possible J-observers*

j	$\mathcal{J}_j$	j	$\mathcal{J}_j$
1	$\{1, 2, 3\}$	6	$\{1, 4, 5\}$
2	$\{1, 2, 4\}$	7	$\{2, 3, 4\}$
3	$\{1, 2, 5\}$	8	$\{2, 3, 5\}$
4	$\{1, 3, 4\}$	9	$\{2, 4, 5\}$
5	$\{1, 3, 5\}$	10	$\{3, 4, 5\}$

Table 2: Possible values of $\mathcal{J}_j$ for an observer with $N_O = 5$ and $N_M = 2$.

Let us take $j = 4$, this would lead to the following output matrix

$$C_4^{\mathcal{J}} = \begin{bmatrix} C_1 \\ C_3 \\ C_4 \end{bmatrix}.$$

The resultant observer is as in Equation (31)*.*

The $\mathcal{J}$ in the superscript indicates that this is a variable that relates to an observer that uses J out of N_O outputs. Substituting $\hat{y}_j^{\mathcal{J}}$ and y_j in $\dot{\hat{x}}_j^{\mathcal{J}}$ leads to

$$\begin{aligned} \dot{\hat{x}}_j^{\mathcal{J}} &= Ax_j^{\mathcal{J}} + Bu + L_j^{\mathcal{J}}(C\hat{x}_j^{\mathcal{J}} + Du - C_j^{\mathcal{J}}x - Du - v_j^{\mathcal{J}} - \tau_j^{\mathcal{J}}) \\ \dot{\hat{x}}_j^{\mathcal{J}} &= (A + L_j^{\mathcal{J}}C_j^{\mathcal{J}})\hat{x}_j^{\mathcal{J}} - L_j^{J}C_j^{\mathcal{J}}x + Bu - L_j^{\mathcal{J}}(v_j^{\mathcal{J}} + \tau_j^{\mathcal{J}}). \end{aligned} \tag{31}$$

Doing the same for the the $p = 1, 2, \ldots, N_P$ *P-observers* leads to

$$\dot{\hat{x}}_p^{\mathcal{P}} = (A + L_p^{\mathcal{P}}C_p^{\mathcal{P}})\hat{x}_p^{\mathcal{P}} - L_p^{\mathcal{P}}C_p^{\mathcal{P}}x + Bu - L_p^{\mathcal{P}}(v_p^{\mathcal{P}} + \tau_p^{\mathcal{P}}). \tag{32}$$

All observers in (31) and (32) form a multi-observer. Note that for a system with N_M the largest possible number so that $N_O > 2N_M$, there is exactly one possible combination with J elements out of all N_J possible combinations that does not contain any of the N_M attacked outputs. We now need to derive a method to select this one attack free observer from all observers. That is where the P-observers become relevant.

4.2 Final estimate selection procedure

Now that all state estimates are constructed, a 'final' estimate $\hat{x}$ needs to be selected from all $\hat{x}_j^{\mathcal{J}}$ First, a comparison will be made between all J-observers and the P-observers that are *sub-observers*, implying that all outputs used to construct the state estimate $\hat{x}_p^{\mathcal{P}}$ are also used in the state estimate $\hat{x}_j^{\mathcal{J}}$. For example, when $N_O = 4$ $N_M = 1$ one of the J-observer uses $\mathcal{J}_2 = \{1, 2, 4\}$ as outputs. The P-estimates would be constructed with $\mathcal{P}_p \in \{1, 2, 4\}$. Let us define

$$\pi_j = \max_{p \subset j} |\hat{x}_j^{\mathcal{J}} - \hat{x}_p^{\mathcal{P}}| \tag{33}$$

which is the largest difference between a single J-estimate and its P-estimates, where the p-observers are all sub-observers of the j-observer. We now collect all π_j in

$$\pi_{\mathcal{J}} = \max_{\mathcal{P} \subset \mathcal{J}} |\hat{x}^{\mathcal{J}} - \hat{x}^{\mathcal{P}}|$$

where $\hat{x}^{\mathcal{J}}$ are all state estimates from (31) and $\hat{x}^{\mathcal{P}}$ are all sub-observers of each $j \in \mathcal{J}$. Let us now choose the smallest π_J

$$\sigma = \arg\min(\pi_{\mathcal{J}}), \quad \sigma \subset \{1, 2, \ldots, N_J\}.$$

We can now select the final state estimate as

$$\hat{x} = \hat{x}_\sigma^{\mathcal{J}}. \tag{34}$$

This selection procedure selects the observer which has the smallest difference between itself and its sub-observers. Figure 6 shows a schematic representation of a CMO.

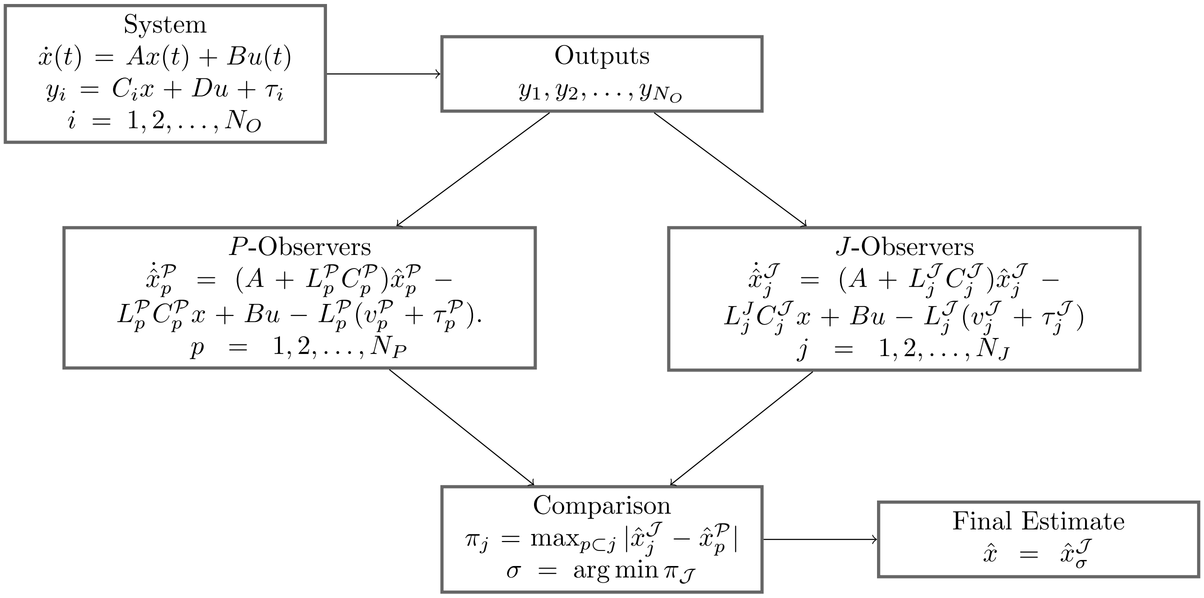

System
$\dot{x}(t) = Ax(t) + Bu(t)$
$y_i = C_i x + Du + \tau_i$
$i = 1, 2, \ldots, N_O$
Outputs
$y_1, y_2, \ldots, y_{N_O}$
P-Observers
$\dot{\hat{x}}_p^{\mathcal{P}} = (A + L_p^{\mathcal{P}} C_p^{\mathcal{P}})\hat{x}_p^{\mathcal{P}} -$
$L_p^{\mathcal{P}} C_p^{\mathcal{P}} x + Bu - L_p^{\mathcal{P}}(v_p^{\mathcal{P}} + \tau_p^{\mathcal{P}}).$
$p = 1, 2, \ldots, N_P$
J-Observers
$\dot{\hat{x}}_j^{\mathcal{J}} = (A + L_j^{\mathcal{J}} C_j^{\mathcal{J}})\hat{x}_j^{\mathcal{J}} -$
$L_j^J C_j^{\mathcal{J}} x + Bu - L_j^{\mathcal{J}}(v_j^{\mathcal{J}} + \tau_j^{\mathcal{J}})$
$j = 1, 2, \ldots, N_J$
Comparison
$\pi_j = \max_{p \subset j} |\hat{x}_j^{\mathcal{J}} - \hat{x}_p^{\mathcal{P}}|$
$\sigma = \arg\min \pi_{\mathcal{J}}$
Final Estimate
$\hat{x} = \hat{x}_\sigma^{\mathcal{J}}$

Figure 6: Conventional multi-observer diagram

4.3 Observer architecture

The state estimates constructed in 4.1 only serve as a basis of the CMO, in this subsection two different 'architectures' will be discussed. These architectures will be implemented in Matlab in the next chapter 6. The first one stores all the observers in a 2D matrix and will thus be known as the 2D-CMO. The second implementation stores the observers in a 3D matrix and will thus be known as the 3D-CMO.

4.3.1 2D conventional multi-observer

Let us write all observers (31)(32) in the following form

$$\dot{\tilde{x}}_{2D} = \tilde{A}_{2D}\tilde{x}_{2D} + F_{2D}\eta_{2D}$$

where

$$\tilde{x}_{2D} = \begin{bmatrix} \hat{x}_1^{\mathcal{J}} \\ \vdots \\ \hat{x}_{N_J}^{\mathcal{J}} \\ \hat{x}_1^{\mathcal{P}} \\ \vdots \\ \hat{x}_{N_P}^{\mathcal{P}} \end{bmatrix}, \quad \tilde{A}_{2D} = \begin{bmatrix} -L_1^{\mathcal{J}} C_1^{\mathcal{J}} & A + L_1^{\mathcal{J}} C_1^{\mathcal{J}} & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\ -L_{N_J}^{\mathcal{J}} C_{N_J}^{\mathcal{J}} & 0 & \cdots & A + L_{N_J}^{\mathcal{J}} C_{N_J}^{\mathcal{J}} & 0 & \cdots & 0 \\ -L_1^{\mathcal{P}} C_1^{\mathcal{P}} & 0 & \cdots & 0 & A + L_1^{\mathcal{P}} C_1^{\mathcal{P}} & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots \\ -L_{N_P}^{\mathcal{P}} C_{N_P}^{\mathcal{P}} & 0 & \cdots & 0 & 0 & \cdots & A + L_{N_P}^{\mathcal{P}} C_{N_P}^{\mathcal{P}} \end{bmatrix} \tag{35}$$

and

$$\eta_{2D} = \begin{bmatrix} u \\ v_1^{\mathcal{J}} + \tau_1^{\mathcal{J}} \\ \vdots \\ v_{N_J}^{\mathcal{J}} + \tau_{N_J}^{\mathcal{J}} \\ v_1^{\mathcal{P}} + \tau_1^{\mathcal{P}} \\ \vdots \\ v_{N_P}^{\mathcal{P}} + \tau_{N_P}^{\mathcal{P}} \end{bmatrix}, \quad F_{2D} = \begin{bmatrix} B & -L_1^{\mathcal{J}} & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\ B & 0 & \cdots & -L_{|\mathcal{J}|}^{\mathcal{J}} & 0 & \cdots & 0 \\ B & 0 & \cdots & 0 & -L_1^{\mathcal{P}} & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots \\ B & 0 & \cdots & 0 & 0 & \cdots & -L_{N_P}^{\mathcal{P}} \end{bmatrix} \tag{36}$$

The implementation of the $\tilde{A}_{2D}$ matrix was inspired by [11, Equation 16.10]. Where instead of calculating the error e, state estimates $\hat{x}$ are calculated. This will not provide the benefits of the *separation theorem* [11, Theorem 16.12] when feedback control is applied. It could be beneficial to change the structure of the CMO in order to take into account the separation theorem. The $\tilde{A}_{2D}$ matrix stores all J- and P-estimates: the top section stores the J-observers and on the bottom the P-observers. Multiplying this A_{2D} matrix by $\tilde{x}_{2D}$ leads to the state estimators presented in (31) and (32) without inputs, noise and attacks. The vector η_{2D} stacks the input and sum of sensor noise and attack signal on top of each other and the matrix F_{2D} stores all factors that need to be multiplied with η_{2D} to arrive at (31) and (32). We will now derive the number of elements used in the 2D-CMO, where the number of elements is the 'number of numbers' that need to be stored in all matrices need to realize the MO. For convenience we define

$$N_S = N_J + N_P,$$

the total number of observers. The sizes of F_{2D} and η_{2D} are not taken as in Equation (36), since the noise and attack signal would not be required in a real-world implementation. The matrices used in the dimenions used in Table 3 are

$$\eta_{2D} = u, \quad F_{2D} = \begin{bmatrix} B^T & B^T & \cdots & B^T \end{bmatrix}^T_{n_x N_S \times n_u}$$

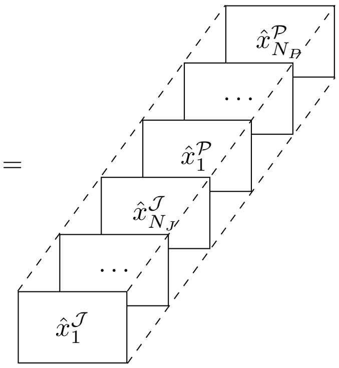
Matrix	Dimensions	Number of elements
$\tilde{x}_{2D}$	$n_x N_S \times 1$	$n_x N_S$
$\tilde{A}_{2D}$	$n_x N_S \times n_x N_S$	$n_x^2 N_S^2$
F_{2D}	$n_x N_S \times n_u$	$n_x n_u N_S$
η_{2D}	$n_u \times 1$	n_u

Table 3: 2D-CMO system matrix dimensions

4.3.2 3D conventional multi-observer

Now the second implementation of the CMO will be discussed. The goal of the 3D-CMO is to reduce the number of zeros as compared to the matrix $\tilde{A}_{2D}$ as in Equation (35). In order to do so we store all block matrices behind each other. Let us start by defining

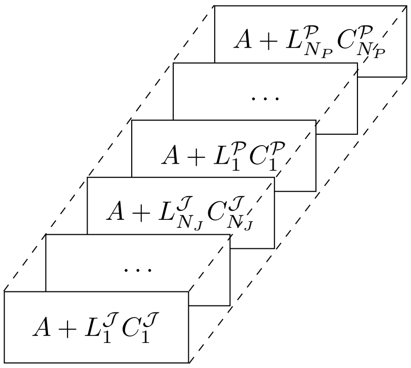
$\tilde{x}_{3D} =$



$\hat{x}^{\mathcal{P}}_{N_P}$
$\cdots$
$\hat{x}^{\mathcal{P}}_1$
$\hat{x}^{\mathcal{J}}_{N_J}$
$\cdots$
$\hat{x}^{\mathcal{J}}_1$

(37)

$\tilde{A}_{3D} =$



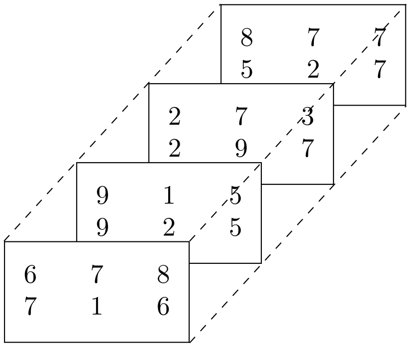
$A + L^{\mathcal{P}}_{N_P} C^{\mathcal{P}}_{N_P}$
$\cdots$
$A + L^{\mathcal{P}}_1 C^{\mathcal{P}}_1$
$A + L^{\mathcal{J}}_{N_J} C^{\mathcal{J}}_{N_J}$
$\cdots$
$A + L^{\mathcal{J}}_1 C^{\mathcal{J}}_1$

(38)

$\tilde{x}_{3D}$ is similar to $\tilde{x}_{2D}$ (35), where the state x and all its state estimates are stored below each other. Let us now define some basic notation, similar to Matlab notation, regarding three-dimensional arrays. We will define a *page* of a 3D array as a single 2D slice from that array, these slices can be along any dimension in the 3D array. Let us define $G \in \mathbb{R}^{2\times3\times4}$, slicing the third page along the third dimension will be denoted as $G(:,:,3)$ and the second page along the first dimension as $G(2,:,:)$. The slices that can be seen in equation (38) are sliced along the third dimension.

Example 5. *Let us define*

$G =$

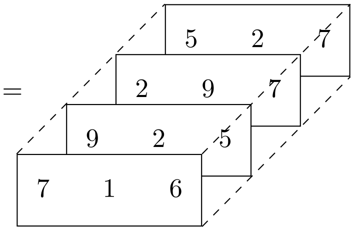


The sliced page

$$G(:,:,3) = \begin{bmatrix} 2 & 7 & 3 \\ 2 & 9 & 7 \end{bmatrix}$$

and

$G(2,:,:) =$



$\tilde{A}_{3D}$ stores all elements on the diagonal of $\tilde{A}_{2D}$ (35) on a separate page of a 3D matrix. Let us now define the operation *page multiplication*, this operation can be performed on two 3D arrays that have (at least) one equal dimension and pages sliced along this equal dimension result in two matrices that can be multiplied. In this report a page multiplication of two matrices A and B along the third dimension will be denoted as $\texttt{pm}(A, B, 3)$.

$$\texttt{pm}(A, B, 3) \implies C(:,:,\text{i}) = A(:,:,\text{i})B(:,:,\text{i}), \quad i = 1, 2, \ldots, n_3 \tag{39}$$

where

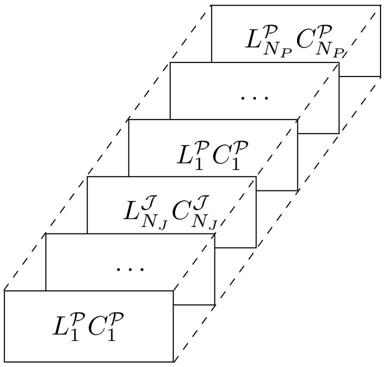
$$A \in \mathbb{R}^{n_{a_1} \times n_{a_2} \times n_3}, \quad B \in \mathbb{R}^{n_{b_1} \times n_{b_2} \times n_3}, \quad C \in \mathbb{R}^{n_{a_1} \times n_{b_2} \times n_3}, \quad n_{a_2} = n_{b_1}.$$

This operation is analogous with the Matlab function `pagemtimes` [1]. Another operation is required for the summands that are equal for each observer, Bu. The multiplication only has to be performed once and the resultant matrix M should be repeated along the third dimension n times. Let us define page multiplication as follows

$$\texttt{rep}(M,\text{n},3) \implies K(:,:,\text{i}) = M, \quad i = 1, 2, \ldots, n \tag{40}$$

This operation is analogous to the Matlab function `repmat`(M,1,1,n) [1]. Let us now define

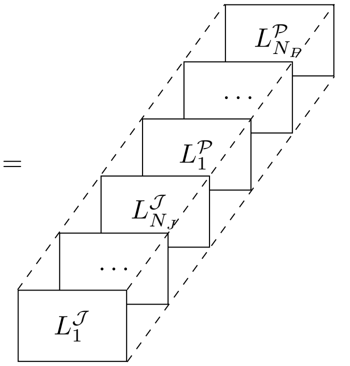
$F_{3D} =$


$L_{N_P}^{\mathcal{P}} C_{N_P}^{\mathcal{P}}$
$\dots$
$L_1^{\mathcal{P}} C_1^{\mathcal{P}}$
$L_{N_J}^{\mathcal{J}} C_{N_J}^{\mathcal{J}}$
$\dots$
$L_1^{\mathcal{P}} C_1^{\mathcal{P}}$

(41)

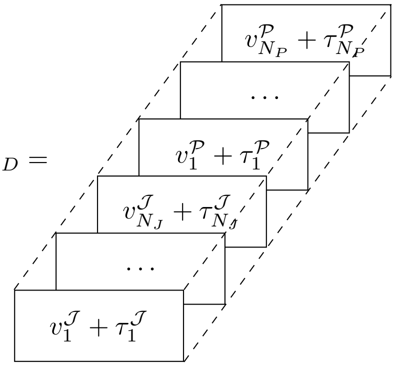
and

$L_{3D} =$



(42)

$\eta_{3D} =$


$v_{N_P}^{\mathcal{P}} + \tau_{N_P}^{\mathcal{P}}$
$\dots$
$v_1^{\mathcal{P}} + \tau_1^{\mathcal{P}}$
$v_{N_J}^{\mathcal{J}} + \tau_{N_J}^{\mathcal{J}}$
$\dots$
$v_1^{\mathcal{J}} + \tau_1^{\mathcal{J}}$

(43)

We can now construct all observers (31) and (32) with the 3D matrices (37),(38),(41),(42) and (43)

$$\dot{\tilde{x}}_{3D} = \texttt{pm}(\tilde{A}_{3D}, \tilde{x}_{3D}, 3) - \texttt{pm}(F_{3D}, x, 3) - \texttt{pm}(L_{3D}, \eta_{3D}, 3) + \texttt{rep}(Bu, N_S)$$

with page multiplication and repetition as in Equations (39) and (40) respectively. Let us now look at the sizes of all matrices that are required for state estimation. The matrices L_{3D} and η_{3D} are not taken into account, since real-world implementation.

Matrix	Dimensions	Number of elements
$\tilde{x}_{3D}$	$n_x \times 1 \times N_S$	$n_x N_S$
$\tilde{A}_{3D}$	$n_x \times n_x \times N_S$	$n_x^2 N_S$
F_{3D}	$n_x \times n_x \times N_S$	$n_x^2 N_S$

Table 4: 3D-CMO system matrix dimensions

4.4 Number of observers

From the combinatoric nature of Equation (30) it can be seen that the number of observers required to provide the secure state-estimate grows rapidly with the number of outputs N_O. Let us look more closely at N_J when N_M is the largest integer so that $N_O > 2N_M$ holds and thus Equation (30) holds.

$$N_J = \binom{N_O}{N_O - N_M} = \frac{N_O!}{(N_O - M)! N_M!} = \binom{N_O}{N_M}$$

which follows from the symmetry property of a combination [17, Section 1.1]. Substituting this into N_S results in

$$N_S = \frac{N_O!}{(N_O - N_M)!N_M!} + N_O,$$

which can be simplified into

$$N_S \approx \frac{N_O!}{(N_O - M)!N_M!}$$

for large N_O, when the combinatoric term dominates. Let us assume that $N_M = \frac{1}{2}N$ and substitute it in

$$N_S \approx \frac{N_O!}{(N_O - \frac{1}{2}N_O)!(\frac{1}{2}N_O)!} = \frac{N_O!}{\left((\frac{1}{2}N_O)!\right)^2}. \tag{44}$$

This requires taking a factorial of a non-integer, which is not defined. We therefore use *Stirling's formula* [4], which states that

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n, \tag{45}$$

where the $\sim$ implies that the ratio between the two sides of the equation approaches 1 as $N_O \to \infty$. A table comparing the values of the combination and Stirling's approximation can be found in Appendix E. Let us now substitute Equation (45) into Equation (44)

$$N_S \approx \frac{\sqrt{2\pi N_O}(\frac{N_O}{e})^{N_O}}{\left(\sqrt{\pi N_O}(\frac{1}{2}\frac{N_O}{e})^{\frac{1}{2}N_O}\right)^2} = \frac{\sqrt{2\pi N_O}}{\pi N_O}\frac{(\frac{N_O}{e})^{N_O}}{(\frac{1}{2}\frac{N_O}{e})^{N_O}} = \sqrt{\frac{2}{\pi N_O}}\frac{(\frac{N_O}{e})^{N_O}}{(\frac{1}{2})^{N_O}(\frac{N_O}{e})^{N_O}}$$

which can be further simplified into

$$N_S \approx \sqrt{\frac{2}{\pi N_O}}2^{N_O} \tag{46}$$

From Equation (46) we can conclude that the amount of observers required for secure state estimation scales with approximately $N_O^{-\frac{1}{2}}2^{N_O}$ when $N_P = 1$, which is not favourable.

5 State-sharing multi-observer

In this chapter, the state-sharing multi-observer (SSMO) will be discussed, aiming to reduce the required memory to store the CMO. The SSMO described in this chapter will provide the estimates as described in Subsection 4.1 and will employ the same selection procedure as described in Subsection 4.2.

5.1 Constructing the state-estimates

The SSMO follows the definition as in [8]. Consider the observers

$$\begin{aligned}\dot{\hat{x}}_j^{\mathcal{J}} &= \mathcal{A}_j^{\mathcal{J}}\hat{x}_j^{\mathcal{J}} - L_j^{\mathcal{J}}C_j^{\mathcal{J}}x + Bu - L_j^{\mathcal{J}}(v_j^{\mathcal{J}} + \tau_j^{\mathcal{J}}), \quad \mathcal{A}_j^{\mathcal{J}} = A + L_j^{\mathcal{J}}C_j^{\mathcal{J}} \\ \dot{\hat{x}}_p^{\mathcal{P}} &= \mathcal{A}_p^{\mathcal{P}}\hat{x}_p^{\mathcal{P}} - L_p^{\mathcal{P}}C_p^{\mathcal{P}}x + Bu - L_p^{\mathcal{P}}(v_p^{\mathcal{P}} + \tau_p^{\mathcal{P}}), \quad \mathcal{A}_p^{\mathcal{P}} = A + L_p^{\mathcal{P}}C_p^{\mathcal{P}}\end{aligned} \tag{47}$$

where $j = 1, 2, \ldots, N_J$, $p = 1, 2, \ldots, N_P$ with N_J and N_P as in Equation (30). All L must be selected in such a way that all $\mathcal{A}$ share the same characteristic polynomial

$$\det(sI - \mathcal{A}) = p(s) = s^n + q_1 s^{n-1} + \cdots + q_{n-1}s + q_n \tag{48}$$

and thus all have equal eigenvalues. Let us now define a matrix $\tilde{L}$ such that $\tilde{L}y = Ly_j$ for all J-observers or $\tilde{L}y = Ly_p$ for all P-observers. This can be achieved by padding L with zero vectors $z \in \mathbb{R}^{n_x \times 1}$ [8]. Let us now rewrite the observer (47) into the following form

$$\begin{aligned}\dot{\hat{x}}_j^{\mathcal{J}} &= \mathcal{A}_j^{\mathcal{J}}\hat{x}_j^{\mathcal{J}} + \mathcal{B}_j^{\mathcal{J}}\eta_j^{\mathcal{J}}, \quad \mathcal{B}_j^{\mathcal{J}} = \begin{bmatrix} B & -\tilde{L}_j^{\mathcal{J}}\end{bmatrix}, \quad \eta_j = \begin{bmatrix} u \\ C_j^{\mathcal{J}}x_j^{\mathcal{J}} + v_j^{\mathcal{J}} + \tau_j^{\mathcal{J}}\end{bmatrix} \\ \dot{\hat{x}}_p^{\mathcal{P}} &= \mathcal{A}_p^{\mathcal{P}}\hat{x}_p^{\mathcal{P}} + \mathcal{B}_p^{\mathcal{P}}\eta_p^{\mathcal{P}}, \quad \mathcal{B}_p^{\mathcal{P}} = \begin{bmatrix} B & -\tilde{L}_p^{\mathcal{P}}\end{bmatrix}, \quad \eta_p = \begin{bmatrix} u \\ C_p^{\mathcal{P}}x_p^{\mathcal{P}} + v_p^{\mathcal{P}} + \tau_p^{\mathcal{P}}\end{bmatrix}.\end{aligned} \tag{49}$$

Let us now derive transformation matrices $T_j^{\mathcal{J}}$ and $T_p^{\mathcal{P}}$ that transform all $\mathcal{A}_j^{\mathcal{J}}, \mathcal{B}_j^{\mathcal{J}}, \mathcal{A}_p^{\mathcal{P}}$ and $\mathcal{B}_p^{\mathcal{P}}$ as in (49) into controllable canonical form as in [11, Sec. 4.3.2]

$$\mathbf{A} = \begin{bmatrix} -q_1 I_l & -q_2 I_l & \cdots & -q_{n-1}I_l & -q_n I_l \\ I_l & 0_l & \cdots & 0_l & 0_l \\ 0_l & I_l & \cdots & 0_l & 0_l \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0_l & 0_l & \cdots & I_l & 0_l \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} I_l \\ 0_l \\ \vdots \\ 0_l \\ 0_l \end{bmatrix} \tag{50}$$

where $l = N$. The transformation matrices for each observer are,

$$\begin{aligned} T_j^{\mathcal{J}} &= R_j^{\mathcal{J}}R_q \\ T_p^{\mathcal{P}} &= R_p^{\mathcal{P}}R_q \\ R_j^{\mathcal{J}} &= \begin{bmatrix} \mathcal{B}_j^{\mathcal{J}} & \mathcal{A}_j^{\mathcal{J}}\mathcal{B}_j^{\mathcal{J}} & \left(\mathcal{A}_j^{\mathcal{J}}\right)^2\mathcal{B}_j^{\mathcal{J}} & \cdots & \left(\mathcal{A}_j^{\mathcal{J}}\right)^{n-1}\mathcal{B}_j^{\mathcal{J}}\end{bmatrix} \\ R_p^{\mathcal{P}} &= \begin{bmatrix} \mathcal{B}_p^{\mathcal{P}} & \mathcal{A}_p^{\mathcal{P}}\mathcal{B}_p^{\mathcal{P}} & \left(\mathcal{A}_p^{\mathcal{P}}\right)^2\mathcal{B}_p^{\mathcal{P}} & \cdots & \left(\mathcal{A}_p^{\mathcal{P}}\right)^{n-2}\mathcal{B}_p^{\mathcal{P}}\end{bmatrix} \\ R_q &= \begin{bmatrix} I_l & q_1 I_l & q_2 I_l & \cdots & q_{n-1}I_l \\ 0_l & I_l & q_1 I_l & \cdots & q_{n-2}I_l \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0_l & \cdots & 0_l & I_l & q_1 I_l \\ 0_l & \cdots & 0_l & 0_l & I_l \end{bmatrix}. \end{aligned} \tag{51}$$

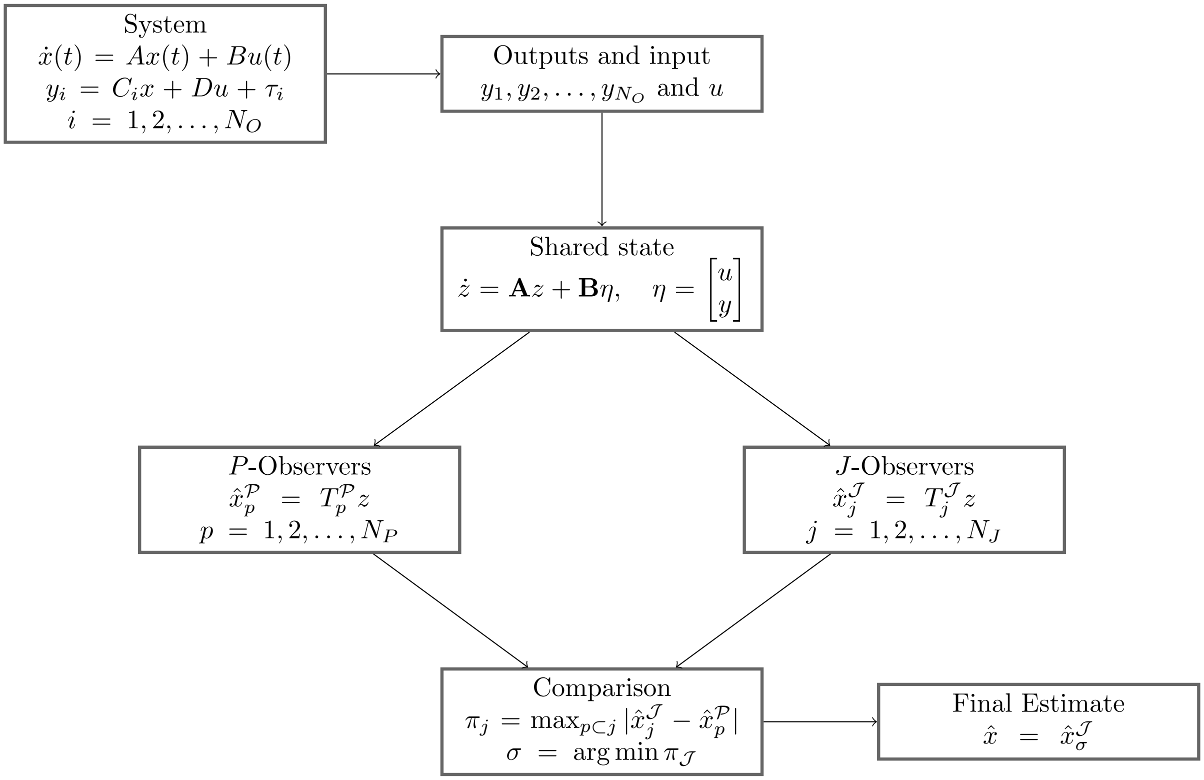
The calculation showing that this transformation holds can be found in Appendix B. It should be noted that $\mathbf{A}$ and $\mathbf{B}$ are independent of j and p. Since all observers (47) share the same characteristic polynomial (48), $\mathbf{A}$ and $\mathbf{B}$ are the same for all observers. This means that only one copy of the matrices needs to be stored and all state estimates $\hat{x}$ can be recovered using the transformation matrices $T_j^{\mathcal{J}}$ and $T_p^{\mathcal{P}}$

$$\hat{x}_j^{\mathcal{J}} = T_j^{\mathcal{J}} z, \quad \hat{x}_p^{\mathcal{P}} = T_p^{\mathcal{P}} z$$

where z is the shared state which is governed by the equation

$$\dot{z} = \mathbf{A}z + \mathbf{B}\eta, \quad \eta = \begin{bmatrix} u \\ Cx + v + \tau \end{bmatrix}. \tag{52}$$

An important requirement is that all state estimates $\hat{x}_j^{\mathcal{J}}$ and $\hat{x}_p^{\mathcal{P}}$ share the same initial condition $\hat{x}_j^{\mathcal{J}}(0) = \hat{x}_p^{\mathcal{P}}(0)$ for all $j = 1, 2, \ldots, N_J$ and $p = 1, 2, \ldots, N_P$, for simplicity it will be chosen as 0.



System
$\dot{x}(t) = Ax(t) + Bu(t)$
$y_i = C_i x + Du + \tau_i$
$i = 1, 2, \ldots, N_O$
Outputs and input
$y_1, y_2, \ldots, y_{N_O}$ and u
Shared state
$\dot{z} = \mathbf{A}z + \mathbf{B}\eta, \quad \eta = \begin{bmatrix} u \\ y \end{bmatrix}$
P-Observers
$\hat{x}_p^{\mathcal{P}} = T_p^{\mathcal{P}} z$
$p = 1, 2, \ldots, N_P$
J-Observers
$\hat{x}_j^{\mathcal{J}} = T_j^{\mathcal{J}} z$
$j = 1, 2, \ldots, N_J$
Comparison
$\pi_j = \max_{p \subset j} |\hat{x}_j^{\mathcal{J}} - \hat{x}_p^{\mathcal{P}}|$
$\sigma = \arg\min \pi_{\mathcal{J}}$
Final Estimate
$\hat{x} = \hat{x}_\sigma^{\mathcal{J}}$

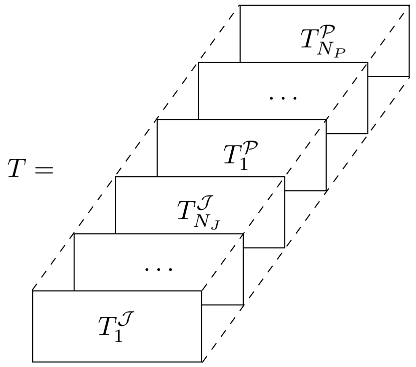
Figure 7: State sharing multi-observer diagram

5.2 Observer architecture

Let us now discuss the way an SSMO is actually implemented. The calculation of z is straightforward, Equation (52) is used. the state estimates $\hat{x}_p^{\mathcal{J}}$ and $\hat{x}_p^{\mathcal{P}}$ are calculated with the transformation matrices as follows

$$\tilde{x}_{SSMO} = \texttt{pm}(T, z, 3),$$

where



The structure of the storage matrix $\tilde{x}_{SSMO}$ is the same as $\tilde{x}_{3D}$ in Equation (38). The selection of the final estimate follows the same procedure as described in Section 4.2.

Matrix	Dimensions	Number of elements
$\tilde{x}_{SSMO}$	$n_x \times 1 \times N_S$	$n_x N_S$
$\mathbf{A}$	$n_x N_O \times n_x N_O$	$n_x^2 N_O^2$
$\mathbf{B}$	$n_x N_O \times n_x$	$n_x^2 N_O$
T	$n_x \times n_x N_O \times N_S$	$n_x^2 N_O N_S$

Table 5: SSMO system matrix dimensions

5.3 Size comparison

Let us now compare the sizes of the 2D-CMO, 3D-CMO and SSMO as presented in Tables 3, 4 and 5 respectively. First, the total number of elements for each MO is derived from the matrices that are needed to construct it.

Multi-Observer	Total number of Elements
2D-CMO	$n_x N_S + n_x^2 N_S^2 + n_x n_u N_S + n_u$
3D-CMO	$n_x N_S + 2 n_x^2 N_S$
SSMO	$n_x N_S + n_x^2 N_O^2 + n_x^2 N_O + n_x^2 N_O N_S$

Table 6: Total number of elements in each MO implementation

We now use the for N_S as in equation (46) to compare number of Gigabytes required to store the MO. This value is derived from the number of elements, each element requires 8 bytes to be stored in for example a python float or a Matlab double. The comparison is made for different combinations of the number of state variables n_x and number of total system outputs N_O. Both of these values are taken as $n \in \{1, 2, \dots, 50\}$ and the storage size for all $50^2 = 2500$ combinations are compared.

Figure 8, where the colour indicates the total size of the MO, shows the result of this comparison. The 2D-CMO performs very poorly, as to be expected due to the N_S^2 term appearing in it's A matrix (see Table 3). The 3D-CMO and the SSMO perform similarly. The size of all MOs increases much more with an increasing N_O as compared to a growing n_x, which is not strange considering N_O influences the number of combinations that need to be made from all sensors.

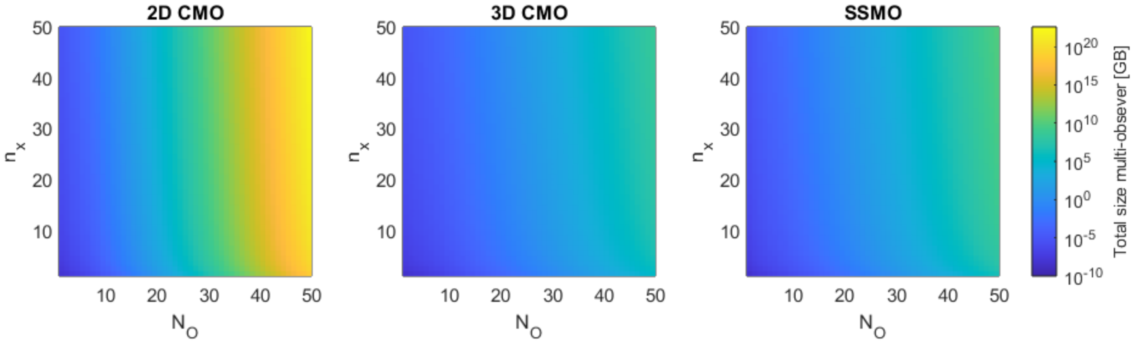


Figure 8: Storage requirements for different MOs

In Figure 9 the SSMO size is divided by the 3D CMO size, the plot shows the ratio between the two MOs. The figure clearly shows that the 3D CMO uses less memory than the SSMO under every circumstance. The difference becomes more pronounced for larger values of N_O. This can be attributed to the size of the T matrix in Equation 5.2, where its size $n_x^2 N_O N_S$ can be viewed as $n_x^3 N_S$ on the diagonal in Figure (9).

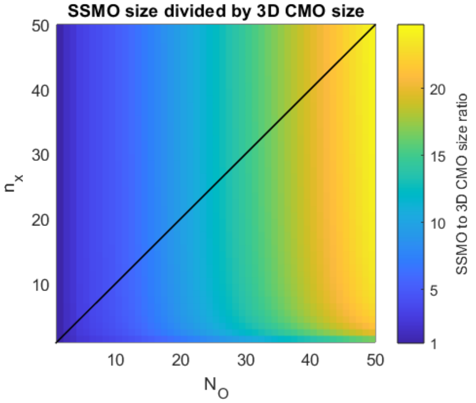


Figure 9: Ratio between the required memory for an SSMO and 3D CMO respectively

It should also be noted that none of the MOs can be implemented in scenarios with a large N_O, since the required memory is significantly too large. Even the super computer Frontier with 9.6 Petabytes (10^6 Gigabytes) of random access memory would not be able to run such an MO [2]. The calculations and plots shown in this subsection have been made with the Matlab script in Appendix D. The values in Figures 8 and 9 are all calculated with $P = 1$, different values of P could cause vastly different results if P is chosen to be large. The relative size difference between the different MOs should stay similar.

6 Matlab implementation

Both CMOs and the SSMO as defined in Subsections 4.3 and 5.2 have been implemented in Matlab. In this chapter the implementations will be discussed and the MOs will be applied on the mass-spring-damper system as described in Chapter 2. The implementation uses classes for the system, attack, shared multi-observers and the specific multi-observers. All of these will be discussed in this chapter, the code itself can be found in Appendix E.

6.1 Class explanation

The system class (`msd`) defines the system matrices for a mass-spring-damper as in Chapter 2. The class can create the state-space system consisting of the matrices in Equations (10),(11),(13) and (14). The class can create both linear and nonlinear systems and works for any number of masses in series.

The `attack` class defines an attack on the system, the outputs that are to be attacked can be specified. If no outputs are specified, a random selection is made. A list with ones indicating an attack and zeros indicating no attack is made, later on this one is substituted by the attack signal. The specific attack signal can be changed within the `attackFunction` function.

All MOs are based on a common set of J-observers and P-observers as defined in Subsection 4.1. Since all MOs observe the same system, these observers can be defined once and used by all MOs. A separate `mo` object needs to be generated for both the J and P-observers. The `mo` defines observers for each combination as defined in Equation (29), based on a given number of outputs for each observer (size of each combination). The class generates an L matrix that places the eigenvalues of $A + LC$ at specified locations. Usually, the size of C does not match the number of outputs N_O. The `mo` class automatically creates a set of outputs based on the provided C where the number of rows matches N_O.

The `cmo2d` class defines the 2D-CMO based on the two `mo` objects for the J and P-observers and the system. It constructs the system matrices as in Equations (35) and (36) and defines the 2D attack vector for use on the 2D CMO.

The `cmo3d` class defines the 3D-CMO based on the two `mo` objects. It generates the system matrices as in Equations (37),(38),(41)(42) and (43) and uses the `attack` object to create a 3D attack vector that stores the ones and zeros for each specific observer.

The `ssmo` class defines the SSMO based on the `mo` objects. It generates the shared system matrices as in Equation (50) and all transformation matrices as in Equation (51).

6.2 Attacking the system

All MOs can be simulated under the same conditions and the results can be compared. The file mainClassScript.m in Appendix E, generates all MOs and solves the ODEs as in Equations (1), (31) and (32). Figure 10 displays the solution of this ODE for the system as in Example 1. The observer quickly approaches the system state and the error decays to zero.

Let us now introduce the attack as in Figure 1, so outputs 2 and 5 are under attack. The attack signal $\tau_i = t, i = 2, 5$ is injected into both outputs (t is simply the time value). Figure 11 shows the result of such an attack, all but one of the red dashed J-estimates diverge from the actual state (the one that converges overlaps with the light blue line). The MO is able to correctly select the correct state estimate from the observer that uses the outputs $1, 3$ and 4 to construct its state estimate. The blue dashed lines show the P-estimators, each constructed with a single output. In this case three of them converge and the other two diverge.

Let us now break the condition as in Equation (28) and attack more than half of the sensors. Figure 12 shows the results, the MO gives completely unreliable estimates.

We now add some noise to the sensors, Figure 13 shows the results. The J and P-observers are not shown in the plot to better visualize the final estimate, which still tracks the state quite well. As can be seen in the top right plot showing the velocity of the first mass, the MO can choose the wrong estimate because of the noise. The selection procedure is misled by the noise, the correct J-estimate varies from its P-estimates because of the noise. It can happen that the noise causes a larger π_j as in Equation (33) than the attack signal. This results in an attacked estimator being selected as the final estimate in Equation (34).

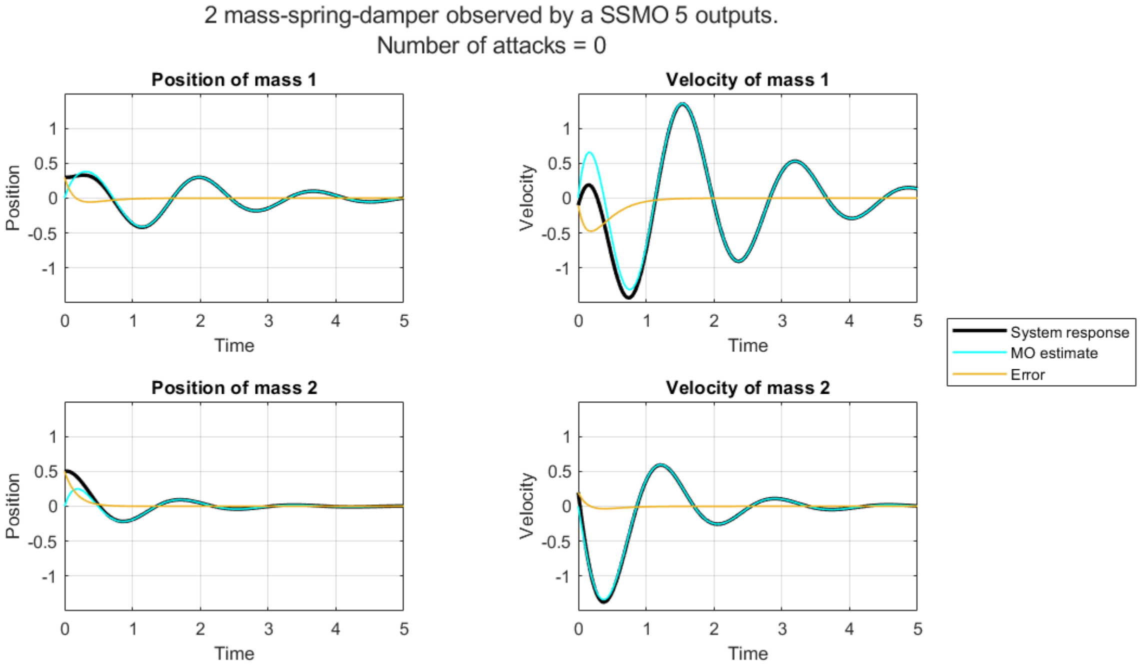


Figure 10: An unattacked, noiseless double mass-spring-damper system and an observer.

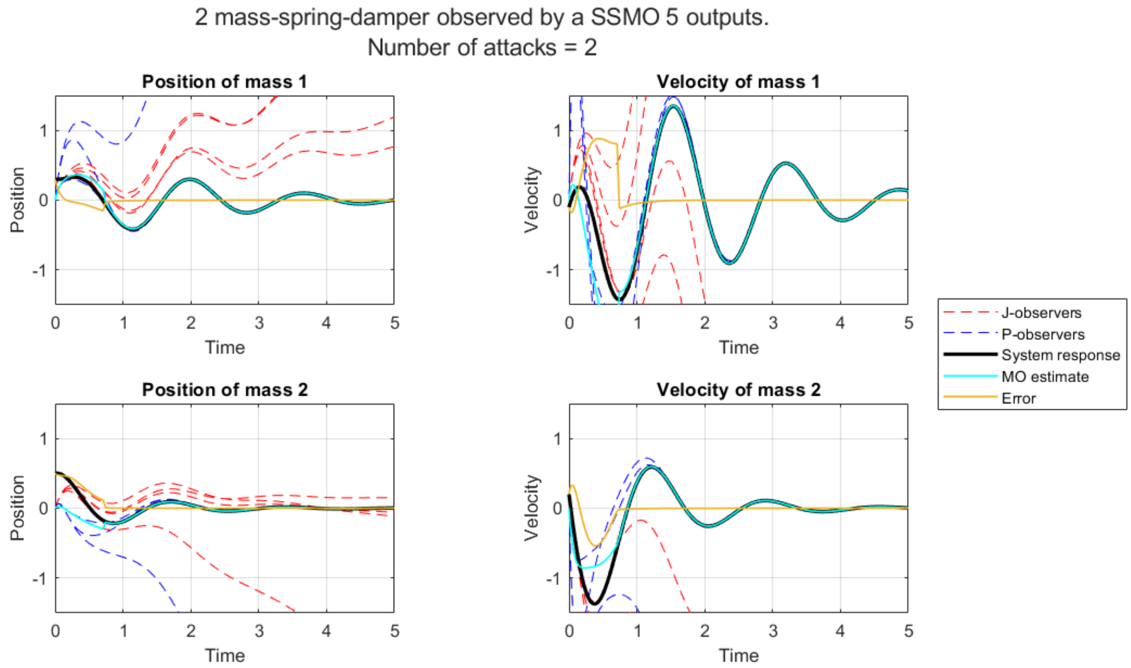


Figure 11: An attacked, noiseless, double mass-spring-damper system, observed by a SSMO. Outputs 2 and 5 are under attack with attack signal $\tau = t$.

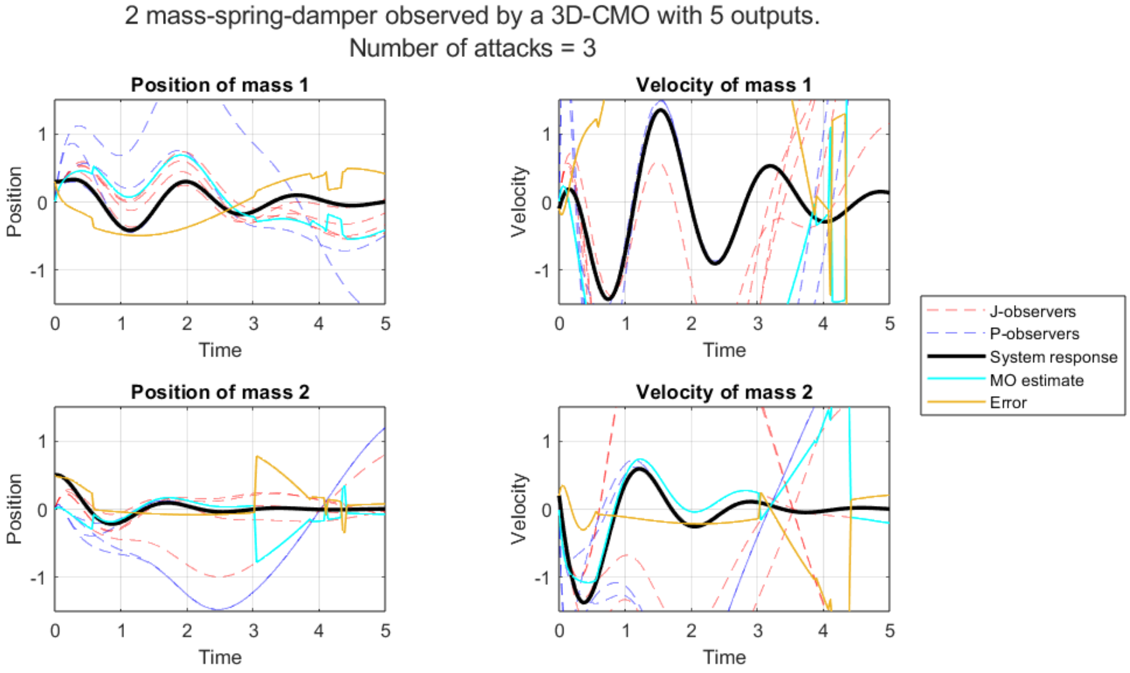


Figure 12: An attacked, noiseless, double mass-spring-damper system observed by a 3D-CMO. Outputs 1,2 and 5 are under attack with attack signal $\tau = \sin(t)$

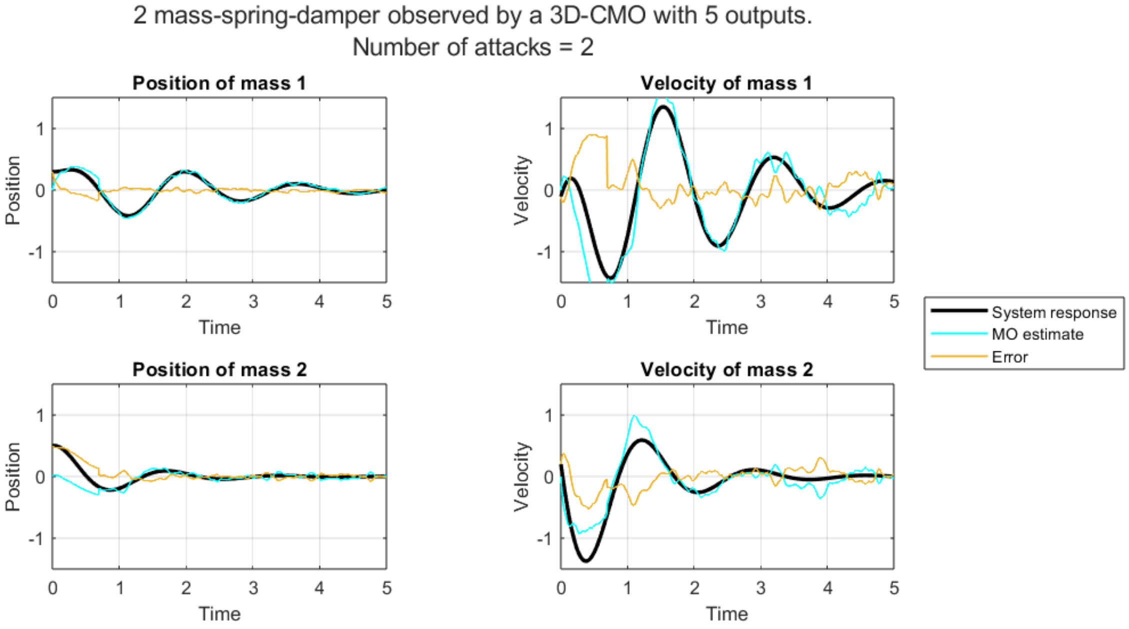


Figure 13: An attacked, noisy double mass-spring-damper system, observed by a 3D-CMO. Outputs 2 and 5 are under attack with attack signal $\tau = \sin(t)$.

7 Nonlinear multi-observers

In this chapter the issues with MOs applied on nonlinear systems will be discussed. We will employ an MO on system (1), which will be repeated here for readability

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t) + E\phi(H_x x)\\ y_i(t) &= C_i x(t) + Du(t) + v_i(t) + \tau_i(t) \quad i \in \mathcal{N} = \{1, 2, \dots, N_O\}.\end{aligned}$$

The observers will be extended with the aim of correctly observing the state of system (1) when less then half of the sensors are under attack (the same boundary condition as discussed in Chapter 4).

7.1 Extending the state-estimates

Let us extend the MO to also observe nonlinear systems as a single observer in Equation (27). First H_x as in Equation (12) has to be adapted to facilitate a nonlinearity ϕ with y as an input

$$H_y = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & \cdots & 0\\ -1 & 1 & 0 & \cdots & 0 & \cdots & 0\\ -1 & -1 & 1 & \cdots & 0 & \cdots & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots & & \vdots\\ -1 & -1 & -1 & \cdots & 1 & \cdots & 0 \end{bmatrix}_{n_\phi \times n_y} . \tag{53}$$

The multiplication $H_y y$ now gives all relative positions

$$H_y y = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & \cdots & 0\\ -1 & 1 & 0 & \cdots & 0 & \cdots & 0\\ -1 & -1 & 1 & \cdots & 0 & \cdots & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots & & \vdots\\ -1 & -1 & -1 & \cdots & 1 & \cdots & 0 \end{bmatrix} \begin{bmatrix} y_1\\ y_2\\ y_3\\ \vdots\\ y_{n_y}\end{bmatrix} = \begin{bmatrix} y_1\\ y_2 - y_1\\ y_3 - y_2 - y_3\\ \vdots\\ y_{n_\phi} - y_{n_\phi - 1} - \cdots - y_2 - y_1\end{bmatrix}.$$

Let us extend the observers as in Equation (31)

$$\begin{aligned}\dot{\hat{x}}_j^{\mathcal{J}} &= (A + L_j^{\mathcal{J}} C_j^{\mathcal{J}})\hat{x}_j^{\mathcal{J}} - L_j^{J} C_j^{\mathcal{J}} x + Bu + E\phi(H_y y) - L_j^{\mathcal{J}}(v_j^{\mathcal{J}} + \tau_j^{\mathcal{J}})\\ \dot{\hat{x}}_p^{\mathcal{P}} &= (A + L_p^{\mathcal{P}} C_p^{\mathcal{P}})\hat{x}_p^{\mathcal{P}} - L_p^{J} C_p^{\mathcal{P}} x + Bu + E\phi(H_y y) - L_p^{\mathcal{P}}(v_p^{\mathcal{P}} + \tau_p^{\mathcal{P}}),\end{aligned} \tag{54}$$

where $j = 1, 2, \dots, N_J$ and $p = 1, 2, \dots, N_P$. The resulting observer in Equation (54) is analogous to the observer in [8, Equation 5]. In this case the nonlinearity is output dependent, the full y is used as input and not the subset that corresponds to the the specific p or j. This has some implications in the SSE context, where some y_i are corrupted. If the y_i that happens to be used by the nonlinearity $\phi(H_y y)$ is attacked, all state estimates become corrupted.

Example 6. *Let us work through an example to clarify this issue, consider system* (15) *as defined in Example 1, where we change C to have $N_O = 6$ outputs*

$$C = \begin{bmatrix} 1 & 0 & 0 & 0\\ 1 & 0 & 1 & 0\\ 1 & 0 & 0 & 0\\ 1 & 0 & 1 & 0\\ 1 & 0 & 0 & 0\\ 1 & 0 & 1 & 0 \end{bmatrix}. \tag{55}$$

This means there are 3 copies of each sensor. The nonlinear spring as in Equation (5) *uses the spring constants and positions of each individual mass as input. Since we now have three copies of each position sensor we can select any combination of two sensors measuring a different position. Let us select the first two sensors, y_1 and y_2 by slicing the H_y matrix in Equation* (53)

$$H_y = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0\\ -1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

We now construct

$$\phi(H_y y) = \begin{bmatrix} F_s^{NL}(y_1) \\ F_s^{NL}(y_2 - y_1) \end{bmatrix}, \tag{56}$$

Which leads to the following E matrix

$$E = \begin{bmatrix} 0 & 0 \\ -1 & 1 \\ 0 & 0 \\ 0 & -1 \end{bmatrix}. \tag{57}$$

Figure 14 displays a scenario where $\mathcal{M} = \{3, 6\}$, so outputs 3 and 6 are under attack, with the attack signal $\tau_k = t, k \in \mathcal{M}$ The MO works as expected and provides a correct state estimate. The $\phi(H_y y)$ provides a correct value for all $t \geq 0$.

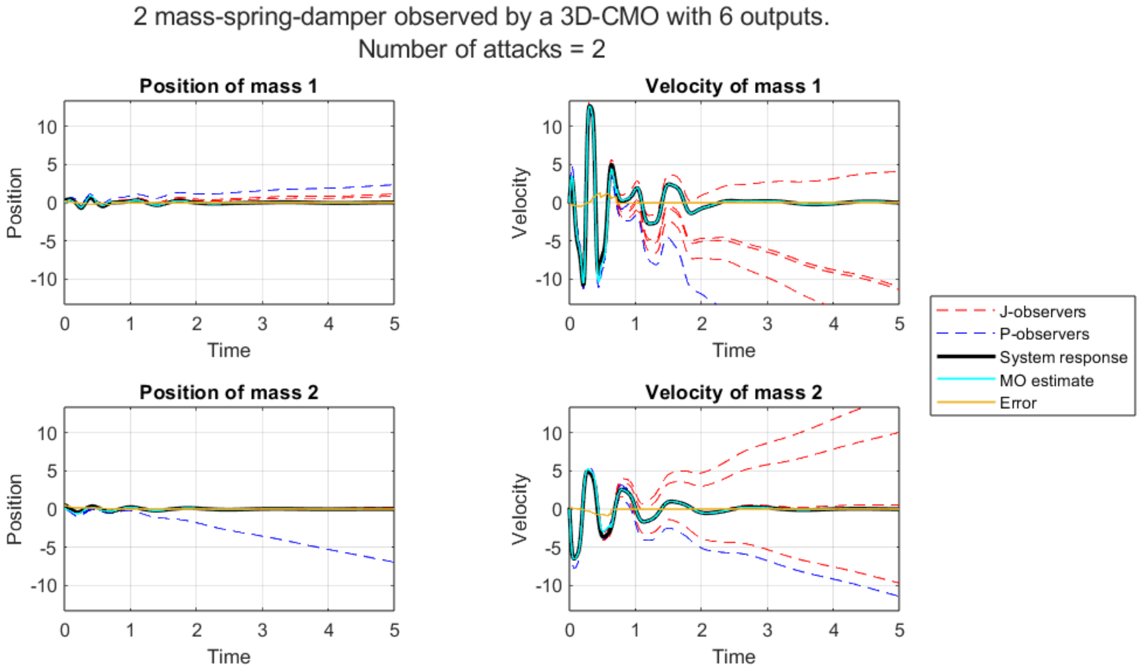


Figure 14: Nonlinear double mass-spring-damper model with outputs 3 and 6 under attack.

Figure 15 shows a scenario where $\mathcal{M} = \{1, 4\}$. Or in other words, outputs 1 and 4 are under attack. The attack signal is the same as in the previous case, $\tau_k = t, k \in \mathcal{M}$. All state estimates fail in this case, because $\phi(H_y y)$ uses an attacked output.

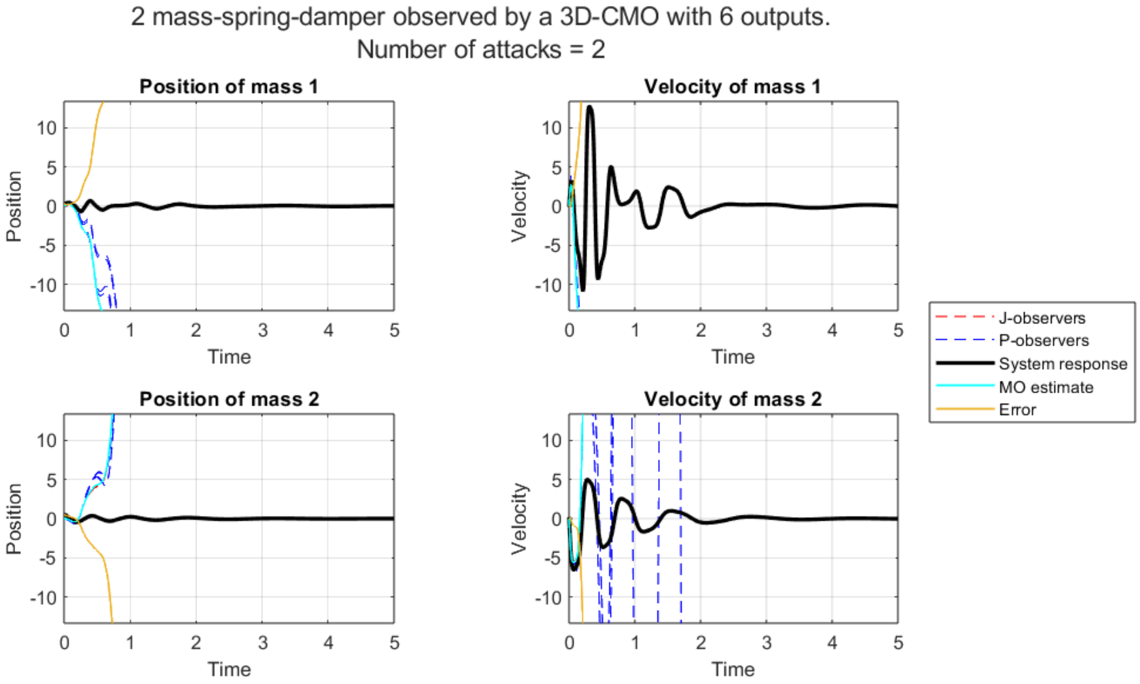


Figure 15: Nonlinear double mass-spring-damper model with outputs 1 and 4 under attack.

Both Figures 14 and 15 show the result of a 3D-CMO, the result on an SSMO is the same. Not surprising, since they realize the same set of observers.

7.2 Investigating the nonlinear issue

As discussed in Example 6 the issue arises because the observers' nonlinear extension does not employ any intelligent logic in choosing which outputs are used to calculate the nonlinear contribution. The nonlinearity $\phi(H_y y)$ should either 'know' which y_i are corrupted and avoid them, which is contradictory with the demands of SSE. Or it should be able to provide a correct state estimate even with an unknown subset of sensors under attack. Let us now discuss the implications and difficulties of this issue for both the 3D-CMO and the SSMO, we will not consider the 2D-CMO in this section.

Let us start with the SSMO, the nonlinear extension to the SSMO is presented in [8] and is as follows

$$\dot{\hat{x}}_j^{\mathcal{J}} = \mathcal{A}_j^{\mathcal{J}} \hat{x}_j^{\mathcal{J}} + \mathcal{B}_j^{\mathcal{J}} \eta_j^{\mathcal{J}}, \quad \mathcal{B}_j^{\mathcal{J}} = \begin{bmatrix} E & B & -\tilde{L}_j^{\mathcal{J}} \end{bmatrix}, \quad \eta_j = \begin{bmatrix} \phi(H_y y) \\ u \\ C_j^{\mathcal{J}} x_j^{\mathcal{J}} + v_j^{\mathcal{J}} + \tau_j^{\mathcal{J}} \end{bmatrix}$$

$$\dot{\hat{x}}_p^{\mathcal{P}} = \mathcal{A}_p^{\mathcal{P}} \hat{x}_p^{\mathcal{P}} + \mathcal{B}_p^{\mathcal{P}} \eta_p^{\mathcal{P}}, \quad \mathcal{B}_p^{\mathcal{P}} = \begin{bmatrix} E & B & -\tilde{L}_p^{\mathcal{P}} \end{bmatrix}, \quad \eta_p = \begin{bmatrix} \phi(H_y y) \\ u \\ C_p^{\mathcal{P}} x_p^{\mathcal{P}} + v_p^{\mathcal{P}} + \tau_p^{\mathcal{P}} \end{bmatrix}.$$

The derivation of the shared matrices in Equation (49) remains the same as in Appendix B. We repeat Equation (49) here for readability

$$\dot{z} = \mathbf{A} z + \mathbf{B} \eta, \quad \eta = \begin{bmatrix} \phi(H_y y) \\ u \\ y \end{bmatrix}.$$

The nonlinearity no longer appears in this structure, since it is taken into account during construction of the shared **B** matrix. The input to the nonlinear function $\phi(H_y y)$ is also 'hidden' behind the shared state.

We cannot try different combinations of sensors y_i because we only calculate a single state z, which uses a single external input η with the full output y.

We now continue with the 3D-CMO, which has more control over each individual state estimate $\hat{x}_j^{\mathcal{J}}$ and $\hat{x}_p^{\mathcal{P}}$. Since every observer is calculated separately, a different nonlinear contribution can be used for every observer.

$$\begin{aligned}\dot{\hat{x}}_j^{\mathcal{J}} &= (A + L_j^{\mathcal{J}} C_j^{\mathcal{J}})\hat{x}_j^{\mathcal{J}} - L_j^{J} C_j^{\mathcal{J}} x + Bu + E\phi(H_y^{\mathcal{J}} y_j^{\mathcal{J}}) - L_j^{\mathcal{J}}(v_j^{\mathcal{J}} + \tau_j^{\mathcal{J}}) \\ \dot{\hat{x}}_p^{\mathcal{P}} &= (A + L_p^{\mathcal{P}} C_p^{\mathcal{P}})\hat{x}_p^{\mathcal{P}} - L_p^{J} C_p^{\mathcal{P}} x + Bu + E\phi(H_y^{\mathcal{P}} y_p^{\mathcal{P}}) - L_p^{\mathcal{P}}(v_p^{\mathcal{P}} + \tau_p^{\mathcal{P}}),\end{aligned} \tag{58}$$

where

$$H_y^{\mathcal{J}} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & \cdots & 0 \\ -1 & 1 & 0 & \cdots & 0 & \cdots & 0 \\ -1 & -1 & 1 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & & \vdots \\ -1 & -1 & -1 & \cdots & 1 & \cdots & 0 \end{bmatrix}_{n_\phi \times J}, \quad H_y^{\mathcal{P}} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & \cdots & 0 \\ -1 & 1 & 0 & \cdots & 0 & \cdots & 0 \\ -1 & -1 & 1 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & & \vdots \\ -1 & -1 & -1 & \cdots & 1 & \cdots & 0 \end{bmatrix}_{n_\phi \times P}$$

The nonlinear contributions $E\phi(H_y^{\mathcal{J}} y_j^{\mathcal{J}})$ and $E\phi(H_y^{\mathcal{P}} y_p^{\mathcal{P}})$ use the subset j or p as input to the function $\phi(H_y y)$. Let us work out two examples that clarify some of the implications this extended 3D-CMO has. We start with an example that investigates a system where the nonlinearity is dependent only on one measured state variable.

Example 7. *Consider the system as in Equation* (1) *with system matrices*

$$A = \begin{bmatrix} 0 & 1 \\ -15 & -2 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}, \quad D = 0, \quad E = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

as derived from the general mass-spring-damper described in Chapter 2. The nonlinearity is

$$\begin{aligned}\phi(H_y^{\mathcal{J}} y_y^{\mathcal{J}}) &= F_s^{NL}(y_1^{\mathcal{J}}) \\ \phi(H_y^{\mathcal{P}} y_y^{\mathcal{P}}) &= F_s^{NL}(y_1^{\mathcal{P}})\end{aligned}$$

where

$$H_y^{\mathcal{J}} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}, \quad H_y^{\mathcal{P}} = 1.$$

The system has $N_O = 5$ outputs, let us attack $N_M = 2$ of those outputs. $\mathcal{M} = \{2, 4\}$, so outputs 2 and 4 are under attack. Again the attack signal $\tau_k = t, k \in \mathcal{M}$ is used. Figure 16 shows the result of such an attack, the MO successfully estimates the state and the error approaches zero.

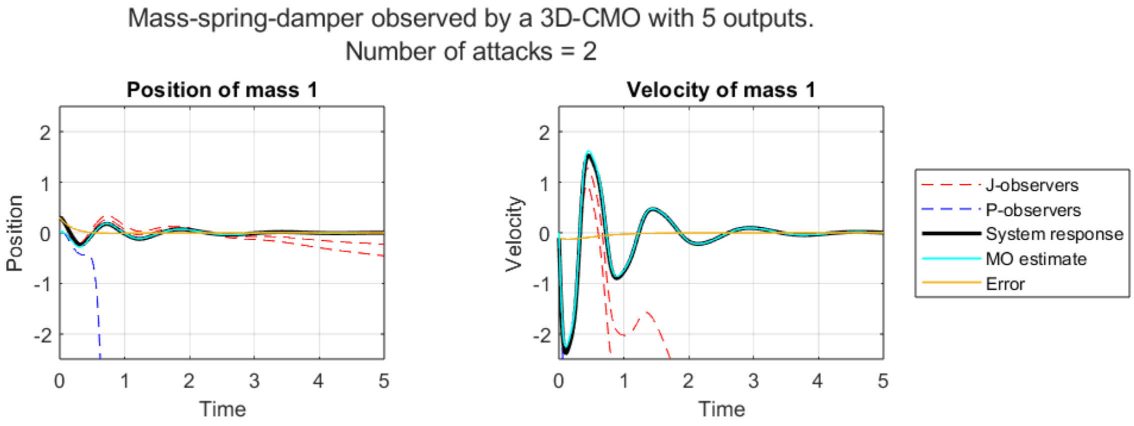


Figure 16: Single nonlinear mass-spring-damper with outputs 2 and 4 under attack.

In this scenario the MO functions because all sensors measure the same variable, the position of the first (and only) mass. So the combination $j = \{1, 3, 5\}$ provides a successful state estimate because $F_s^{NL}(y_1) = F_s^{NL}(y_3) = F_s^{NL}(y_5)$ and since the nonlinearities equal

$$\begin{aligned}
\phi\left(H_y^{\mathcal{J}}\begin{bmatrix} y_1 \\ y_3 \\ y_5 \end{bmatrix}\right) &= \phi(y_1) = F_s^{NL}(y_1) \\
\phi\left(H_y^{\mathcal{P}} y_1\right) &= \phi(y_1) = F_s^{NL}(y_1) \\
\phi\left(H_y^{\mathcal{P}} y_3\right) &= \phi(y_3) = F_s^{NL}(y_3) \\
\phi\left(H_y^{\mathcal{P}} y_5\right) &= \phi(y_5) = F_s^{NL}(y_5).
\end{aligned}$$

Let us now investigate a scenario where the nonlinearity $\phi(y)$ depends on multiple measured state variables.

Example 8. *Now, consider the same system* (15) *as in Example 1, where C is changed to have $N_O = 5$*

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

The size of the P-observers is increased to $P = 2$ and the H matrices are again sliced from (53) *as*

$$H_y^{\mathcal{J}} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{bmatrix}, \quad H_y^{\mathcal{P}} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

which leads to

$$\phi(H_y^{\mathcal{J}} y_j^{\mathcal{J}}) = \begin{bmatrix} F_s^{NL}({y_j^{\mathcal{J}}}_1) \\ F_s^{NL}({y_j^{\mathcal{J}}}_2 - {y_j^{\mathcal{J}}}_1) \end{bmatrix}, \quad y_j^{\mathcal{J}} = \begin{bmatrix} {y_j^{\mathcal{J}}}_1 \\ {y_j^{\mathcal{J}}}_2 \\ {y_j^{\mathcal{J}}}_3 \end{bmatrix}$$

and E is as in Equation (57)*. Let us now choose $N_M = 2$ in order to still satisfy the $N_O > 2N_M$ requirement. Where we specify $\mathcal{M}$ as $\{2, 4\}$, note that now all copies of the sensor measuring the position of mass 2 are attacked. Figure 17 shows that the error $\hat{x} - x$ still approaches 0 as t increases. But the estimates, especially of the velocities, do not look to be tracking the true state correctly.*

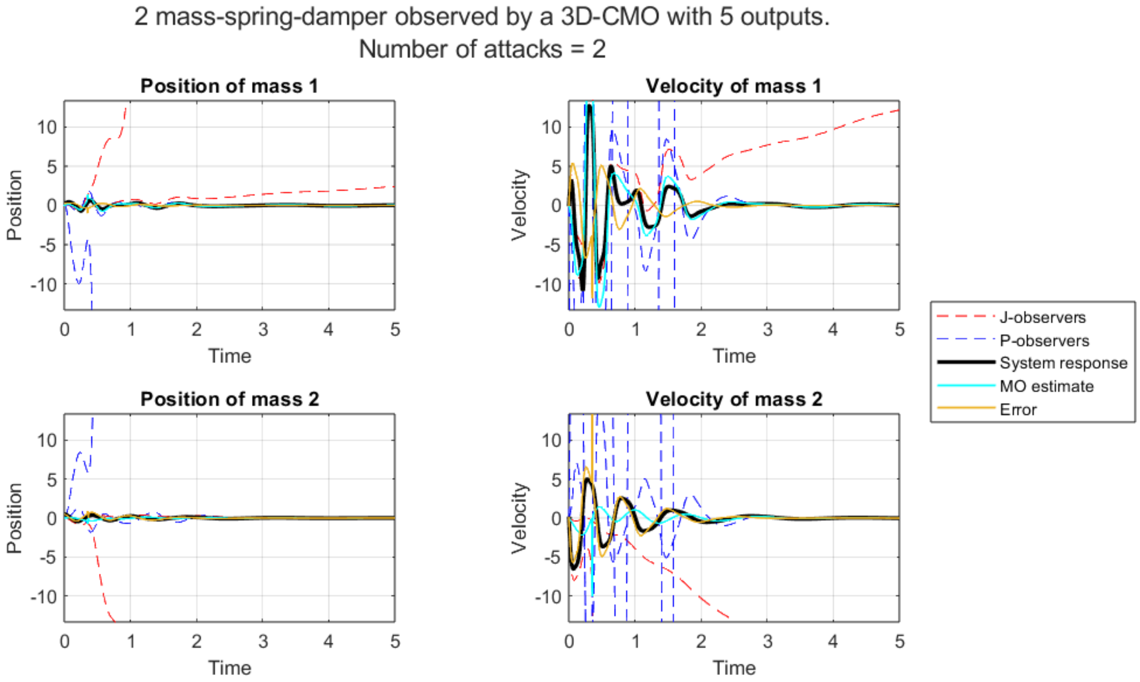


Figure 17: Double nonlinear mass-spring-damper with outputs 2 and 4 under attack.

Let us now expand the number of outputs to $N_O = 6$, and use the exact same system as in Equations (55),(57) *and* (56) *as in Example 6. We now use the observer as in Equation* (58) *and attack the system with the attack signal $\tau_k = t, k \in \mathcal{M}, \mathcal{M} = \{4, 6\}$.*

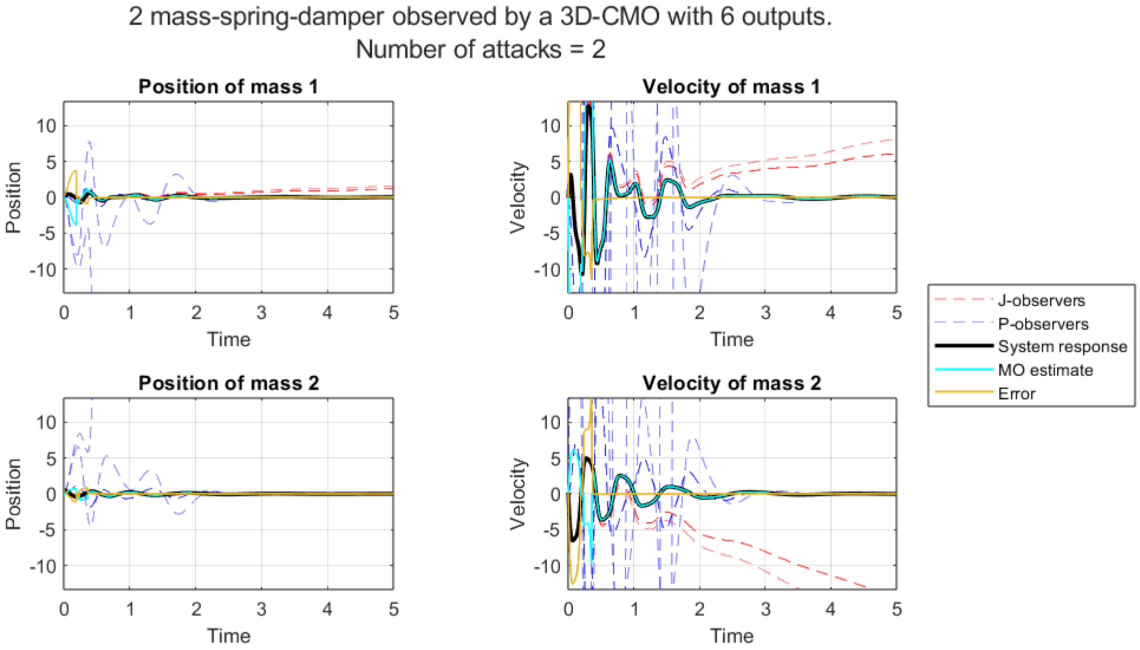


Figure 18: Double mass-spring-damper with outputs 4 and 6 under attack.

The state estimates in Figure 18 look to be tracking the true state much better. The MO now functions correctly because the state estimate using $j = \{1, 2, 3, 5\}$ has the uncorrupted outputs 1 *and* 2 *as input to $\phi(y)$ as in Equation* (56)*.*

The attacked outputs in Example 8 have, of course, been carefully chosen to provide the correct results. Had we attacked outputs 2 and 4 in the $N_O = 6$ case, no correct state estimate could have been provided. This happens because the nonlinearity $\phi(y)$ in the example takes the first two entries of $y_j^{\mathcal{J}}$ or $y_p^{\mathcal{P}}$, irrespective of what the sensor measures. So if outputs 2 and 4 are attacked, even the fully uncorrupted set of outputs $j = \{1, 3, 5, 6\}$ does not provide a correct state estimate. Because $\phi(y)$ is only considers outputs 1 and 3 in this state estimate, where output 3 is also a measurement of the position of the first mass. If the list would have been ordered as $\{1, 6, 3, 5\}$ the observer would provide a correct estimate. However, the selection procedure would still not work. During the final estimate selection procedure as described in subsection 4.2. All estimates constructed with $p \in j$ are compared against the estimate constructed with j. We now encounter the same obstacle as before: for all subsets $|p| < |j|, p \subset j$, some do not contain any sensor measuring the position of the second mass. Those estimates will suffer the same fate as the J-observers discussed before and provide a wrong state estimate. Even though its 'parent' estimate is correct.

7.3 Addressing the nonlinear issue

Let us develop a new approach in order to tackle this issue. Consider system (1) with outputs

$$y_i, i \in \{1, 2, \ldots, N_O\}.$$

We now define *aggregate sensors* that combine multiple y_i as

$$\nu_g = \begin{bmatrix} y_{g_1} \\ y_{g_2} \\ \vdots \\ y_{g_{|g|}} \end{bmatrix}, \quad g \subset \{1, 2, \ldots, N_O\} \quad \text{and} \quad 1 < n_g < N_O,$$

where $n_g = |g|$. As discovered in subsection 7.2, the order in which sensors appear should be standardized between all aggregate sensors ν_g. For simplicity, the same order as in which variables are required in the nonlinearity will be used.

Example 9. *Consider system* (15) *as in Example 1 with C as in equation* (55) *and $\phi(y)$ as in equation* (56)*. The nonlinearity expects a measurement of x_1 and a measurement of x_3, which is constructed by subtracting a measurement of absolute position of the first mass from a measurement of the absolute position of the second mass. This implies $|g| = n_\phi = 2$, where g could be $\{1, 2\}$, $\{3, 4\}$ or $\{5, 6\}$. Aggregate sensors made up of sensor groups that 'neighbour' each other in the matrix C will be called strict aggregate sensors.*

We can also be less strict on the values g can take. For example, in addition to the strict combinations g could also be $\{1, 4\}, \{1, 6\}, \{3, 2\}, \{3, 6\}, \{5, 2\}$ or $\{5, 4\}$. Such aggregate sensors will be called loose aggregate sensors.

With these aggregate sensors we now construct the state estimates as we have done before, let us start with creating the J and P-observers. We follow the same procedure as described in subsection 4.1, now we only consider the aggregate sensors ν as outputs. In Example (9) the de facto number of outputs is halved, to $N_O = 3$ outputs. For MOs employing strict aggregate sensor grouping the total number of sensors or rows of C, in this case 6, is denoted as N_S. In general

$$N_O = \frac{N_S}{n_\phi}, \quad N_O \in \mathbb{N} \implies N_S = \alpha n_\phi, \quad \alpha \in \mathbb{N}$$

so the number of sensors must equal a multiple of the number of nonlinearities. We can now construct an MO as in Equation (58), with some extra requirements on j and p. Since observers are now constructed with combinations of $\nu_i, i = 1, 2, \ldots, N_O$, we cannot simply take every possible combination of the set $\{1, 2, \ldots, N_S\}$ with sizes J and P respectively.

Let us start by discussing the number of attacks N_M, this number indicates the amount of sensors that are under attack. Those individual sensors could be in the same aggregate sensor. We will assume the worst case, all attacked sensors are in a different aggregate sensor. Let us define N_A as the number of attacked aggregate sensors. The sizes of each J and P-observer are

$$\begin{aligned} J_{agg} &= N_O - N_A, \quad P_{agg} = 1 \\ J_{ind} &= n_\phi J_{agg}, \quad P_{ind} = n_\phi P_{agg}, \end{aligned} \tag{59}$$

where the subscript *agg* denotes the number of aggregate sensors and the subscript *ind* denotes the amount of individual sensors. The condition as in Equation (28) that $N_O > 2N_M$ still holds. Equation (59) implies that more sensors are required as compared to a linear MO.

$$N_O > 2N_M \implies N_O \geq 2N_M + 1 \implies N_S \geq (2N_M + 1)n_\phi$$

which implies that the number of sensors required is n_ϕ times larger. We now construct all J-observers by creating all combinations of $\nu_i, i = 1, 2, \dots, N_O$ and all P-observers use a single ν_i, which results in the following number of observers

$$N_J = \binom{N_O}{J_{agg}}, \quad N_P = \binom{N_O}{P_{agg}}.$$

Example 10. *Consider the system as in Example 6 described by Equations* (15),(57)*. With*

$$C = \begin{bmatrix} \tilde{C}^T & \tilde{C}^T & \tilde{C}^T & \tilde{C}^T & \tilde{C}^T \end{bmatrix}^T, \quad \tilde{C} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

and

$$\phi(H^{\mathcal{J}} y_j^{\mathcal{J}}) = \begin{bmatrix} F_s^{NL}(y_j^{\mathcal{J}}{}_1) \\ F_s^{NL}(y_j^{\mathcal{J}}{}_2 - y_j^{\mathcal{J}}{}_1) \end{bmatrix} \quad \textit{and} \quad \phi(H^{\mathcal{P}} y_p^{\mathcal{P}}) = \begin{bmatrix} F_s^{NL}(y_p^{\mathcal{P}}{}_1) \\ F_s^{NL}(y_p^{\mathcal{P}}{}_2 - y_p^{\mathcal{P}}{}_2) \end{bmatrix},$$

where

$$H_y^{\mathcal{J}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad H_1^{\mathcal{P}} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}.$$

We apply strict aggregate sensor grouping on the $N_S = 10$ *sensors gives the following aggregate sensors*

$$\nu_1 = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, \quad \nu_2 = \begin{bmatrix} y_3 \\ y_4 \end{bmatrix}, \quad \nu_3 = \begin{bmatrix} y_5 \\ y_6 \end{bmatrix}, \quad \nu_4 = \begin{bmatrix} y_7 \\ y_8 \end{bmatrix}, \quad \nu_5 = \begin{bmatrix} y_9 \\ y_{10} \end{bmatrix}.$$

This system has $N_O = 5$ *and can thus handle* 2 *attacks which implies that* $N_A = 2$*. So, the* J*-observers use* $J_{agg} = 3$ *aggregate sensors. Figure 19 shows such a scenario, where the MO behaves as expected.*

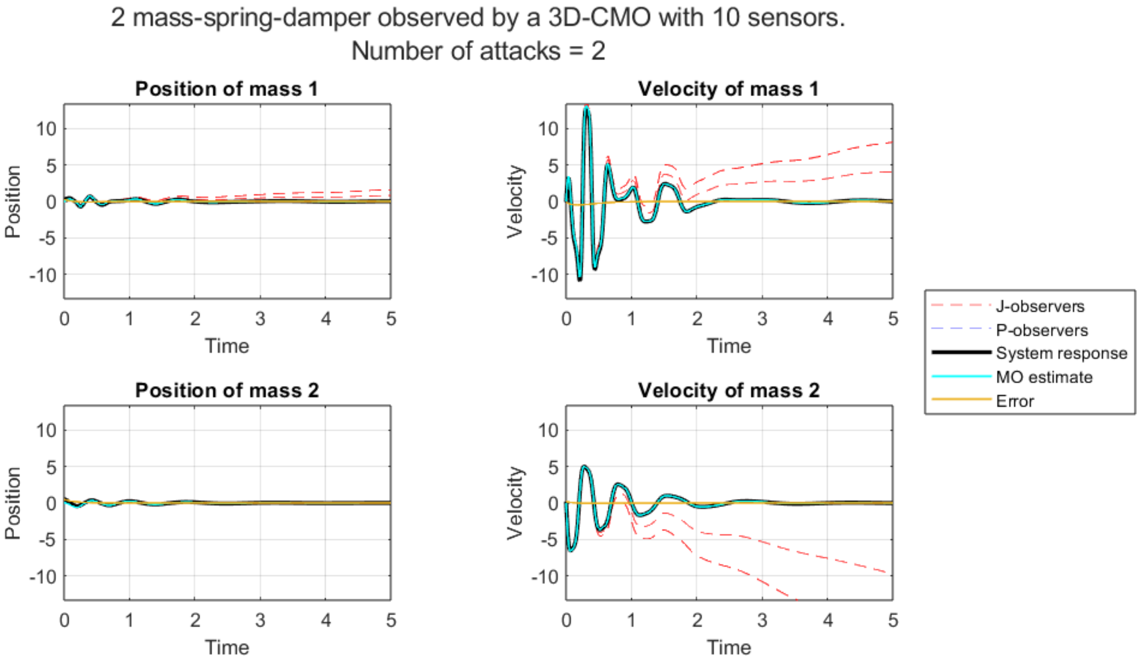


Figure 19: A double mass-spring-damper with aggregate sensors, outputs 4 and 6 are under attack

Let us define the concept of an MO being *robust to an attack*, the MO in Example 9 is robust to any attack with $N_M = 2$. It is also robust to the attack where $\mathcal{M} = \{1, 2, 3\}$, since only 2 of 5 aggregate sensors are attacked. Such an attack corrupts less than half of all aggregate sensors. The number of aggregate sensors attacked will be denoted as N_A. In general a SAMO, strict agrregate multi-observer, is robust to an attack when

$$N_O > 2N_A. \tag{60}$$

Thus N_A decides whether an MO with N_O outputs is robust to the attack. For the system in Example 9 with N_M attacks, N_A could be equal to 1 or 2. That always satisfies the condition in Equation (60). When $N_M = 3$ or $N_M = 4$, the MO is robust to some attacks. The worst case attack would be to attack as many copies of a single sensor.

Example 11. *Consider the same system as in Example 10, let us now perform the following attack:* $\mathcal{M} = \{1, 3, 5\}$ *where* $\tau_i = t, i \in \mathcal{M}$*. Figure 20 shows this scenario, the SAMO is clearly not robust with respect to this attack. Not strange considering* $N_O < 2N_A$*.*

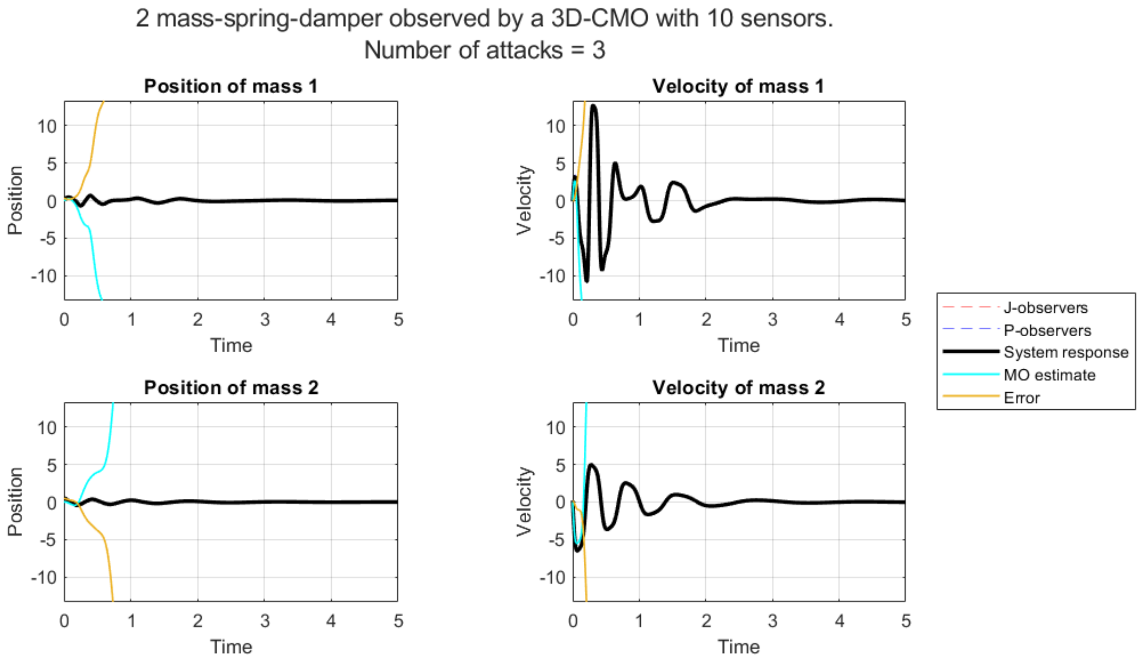


Figure 20: SAMO with $N_S = 10$ and n_ϕ, where outputs 1,3 and 5 are under attack.

So a MO is only robust to some attacks that target 3 sensors. Let us now work out the probability of the system being robust to an attack targetting 3 randomly picked sensors.

Example 12. *Consider the system as in Example 11, let us pick $N_M = 3$ random sensors out of the $N_S = 10$. We first calculate the probability of picking 3 sensors that end up being members of two distinct aggregate sensors*

$$p(N_A = 2) = \frac{5 \cdot 8}{\begin{pmatrix} 10 \\ 3 \end{pmatrix}} = \frac{40}{120} = 0.333,$$

where the numerator is the number of possible combinations with $N_A = 2$ and the denominator is the total number of combinations. Let us further explain the rationale behind the $5 \cdot 8 = 40$ combinations in the numerator. Imagine that sensors $\{1, 2\}$ are under attack, there are now 8 more sensors to choose from. So, this gives 8 combinations for a single sensor group. Since there are 5 groups in this scenario the number of combinations is 40.

Since there are only two options ($N_A = 2$ or $N_A = 3$), $p(N_A = 3) = 1 - p(N_A = 2)$. However, we will work out $p(N_A = 3)$ as well in order to get some insight into that probability

$$p(N_A = 3) = \frac{\begin{pmatrix} 5 \\ 3 \end{pmatrix} 2^3}{\begin{pmatrix} 10 \\ 3 \end{pmatrix}} = \frac{80}{120} = 0.667$$

The denominator remains the same, unsurprising since the number of combinations of 3 out of 10 does not change. The logic behind the numerator requires some more explanation. Imagine we have selected any 3 aggregate sensors, we can now choose one of two possible individual sensors from each of these aggregate sensors. Which leads to $2 \cdot 2 \cdot 2 = 8$ possible choices. Since $\binom{5}{3} = 10$ combinations of aggregate sensors can be made, there are 80 possibilities where $N_A = 3$. This means that if an attacker attacks 3 random outputs, the probability is 0.333 that the SAMO is robust to the attack.

We now expand the concept of an MO being robust to an attack, the MO in Example 12 is *fully robust to 2 attacks.* A system is fully robust to N_M attacks if there is zero probability that $2N_A \geq N_O$ for an attack targetting N_M sensors. The same MO is only *partially robust to 3 attacks.* Let us define a *robustness score* that indicates probability that an MO is robust to an attack targetting N_M randomly selected sensors

$$r_{N_M} = p(2N_A < N_O | N_M). \tag{61}$$

A robustness score of 1 for a certain N_M is equal to a system being fully robust to N_M attacks and a robustness score of 0 indicates that any attack targetting N_M sensors will be successful. Instead of calculating the probabilities analytically as in Example 12, the Matlab script in Appendix F loops over each possible attack and calculates N_A. From there the probabilities are derived by taking the fraction of of the total number of combinations.

We now aim to compare LAMOs (loose aggregate multi-observers) to SAMOs in terms of robustness scores. First, LAMO observer sizes need to be investigated, since the worst case assumption for SAMOs that $N_A = N_M$ does not apply to LAMOs. It was only possible because every individual sensor only appeared in one aggregate sensor.

Example 13. *Let us illustrate this difference using an example. Consider the same system as in Example 12, we now create loose aggregate sensors instead of strict aggregate sensors*

$$\nu_1 = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, \quad \nu_2 = \begin{bmatrix} y_1 \\ y_4 \end{bmatrix}, \quad \nu_3 = \begin{bmatrix} y_1 \\ y_6 \end{bmatrix}, \quad \nu_4 = \begin{bmatrix} y_3 \\ y_2 \end{bmatrix}, \quad \cdots \quad \nu_{24} = \begin{bmatrix} y_9 \\ y_8 \end{bmatrix}, \quad \nu_{25} = \begin{bmatrix} y_9 \\ y_{10} \end{bmatrix}.$$

The 25 aggregate sensors imply that $N_O = 25$ and thus, according to the robustness condition in Equation (60), would require at least 13 *aggregate sensors to be attacked. Every single sensor $y_i, i = 1, 2, \ldots, 10$ appears in 5 aggregate sensors $\nu_k, k = 1, 2, \ldots, 25$. This implies that the MO is not robust against an attack that corrupts 3 sensors measuring the same variable, for example, y_1, y_3 and y_5. Since such an attack corrupts* 15 *aggregate sensors.*

So the assumption $N_A = N_M$ does not hold for a LAMO, and thus the size of a LAMO should be decided in another way. In order choose a good number of aggregate sensors for each J-observer, the sizes of LAMOs should be worked out. Let us consider a system with N_S sensors and n_ϕ nonlinearities, this implies that there are N_S/n_ϕ sensors measuring each nonlinear variable. Since we use all combinations between sensors measuring a different variable, there are

$$N_O = \left(\frac{N_S}{n_\phi}\right)^{n_\phi} \tag{62}$$

aggregate sensors. Let us still consider the worst case attack, all but one sensors measuring the same variable are attacked. LAMOs would always have at least one J-observer that is constructed using the same copy of a sensor measuring the same variable. Each of these attacks takes out N_S/n_ϕ aggregate sensors and thus

$$N_A = \frac{N_S}{n_\phi} N_M. \tag{63}$$

This is a worst case estimation, an attack on sensors 1 and 2 would only attack $N_A = 9$ aggregate sensors. We now apply Equation (59) as before to find J_{agg}, the number of aggregate sensors in each J-observer. The robustness requirement in Equation (60) limits the number of attacks the system can handle. The system discussed in Example 13 can handle at most $N_A = 12$ aggregate sensors simultaneously under attack. Consider a LAMO with 2 outputs under attack, resulting in $N_A = 10$. This leads to $J_{agg} = 15$, which leads to in $\binom{25}{15} = 3,268,760$ J-observers and 25 P-observers. In general we can say that a LAMO requires

$$N_J = \binom{N_O}{J_{agg}}, \quad N_P = N_O$$

observers. Let us now compare robustness scores as in Equation (61) between SAMOs and LAMOs, Table 7 displays the robustness scores for SAMOs and Table 8 displays the robustness scores for LAMOs.

N_M		1	2	3	4	5	6	7	8
N_S	6	1	0.2	0	0	0	0	0	0
	8	1	0.1429	0	0	0	0	0	0
	10	1	1	0.3333	0.0476	0	0	0	0
	12	1	1	0.2727	0.0303	0	0	0	0
	14	1	1	1	0.4406	0.1049	0.0117	0	0
	16	1	1	1	0.3846	0.0769	0.0070	0	0
	18	1	1	1	1	0.5294	0.1674	0.0317	0.0029
	20	1	1	1	1	0.4799	0.1331	0.0217	0.0017

Table 7: Robustness scores of a SAMO for different combinations of N_S and N_M

N_M		1	2	3	4	5	6	7	8
N_S	6	1	0	0	0	0	0	0	0
	8	1	0.5741	0	0	0	0	0	0
	10	1	1	0	0	0	0	0	0
	12	1	1	0.8182	0	0	0	0	0
	14	1	1	1	0.4406	0	0	0	0
	16	1	1	1	0.9231	0	0	0	0
	18	1	1	1	1	0.5294	0	0	0
	20	1	1	1	1	0.9675	0	0	0

Table 8: Robustness scores of a LAMO for different combinations of N_S and N_M

Tables 7 and 8 clearly show that both sensor grouping methods are fully robust against the same attacks. The SAMO is partially robust against more attacks, although the LAMO does outperforms or equals the performance of the SAMO in the cases where it is partially robust.

7.4 Beyond assurances

In this subsection we will explore the behaviour of nonlinear MOs beyond the limit of $N_A > 2N_A$ as in Equation (60). Let us start with

Example 14. *Consider a LAMO as in Example 13, with* $N_S = 10$ *and* $n_\phi = 2$*. This leads to* $N_O = (10/2)^2 = 25$ *and* $N_A = (10/2) \cdot 4 = 20$ *which implies that* $J_{agg} = 25 - 20 = 5$ *as per Equation (59). The attack has* $\mathcal{M} = \{1, 3, 5, 7\}$ *where the attack signal is* $\tau_i = t, i \in \mathcal{M}$*. Figure 21 shows this scenario, where the final estimate converges towards the state. The state estimate around* $t = 0.25$ *does seem to be unstable: the selection procedure seems to easily select an attacked J-observer. After* $t = 0.5$ *the LAMO seems to stabilize and stick to a correct state estimate.*

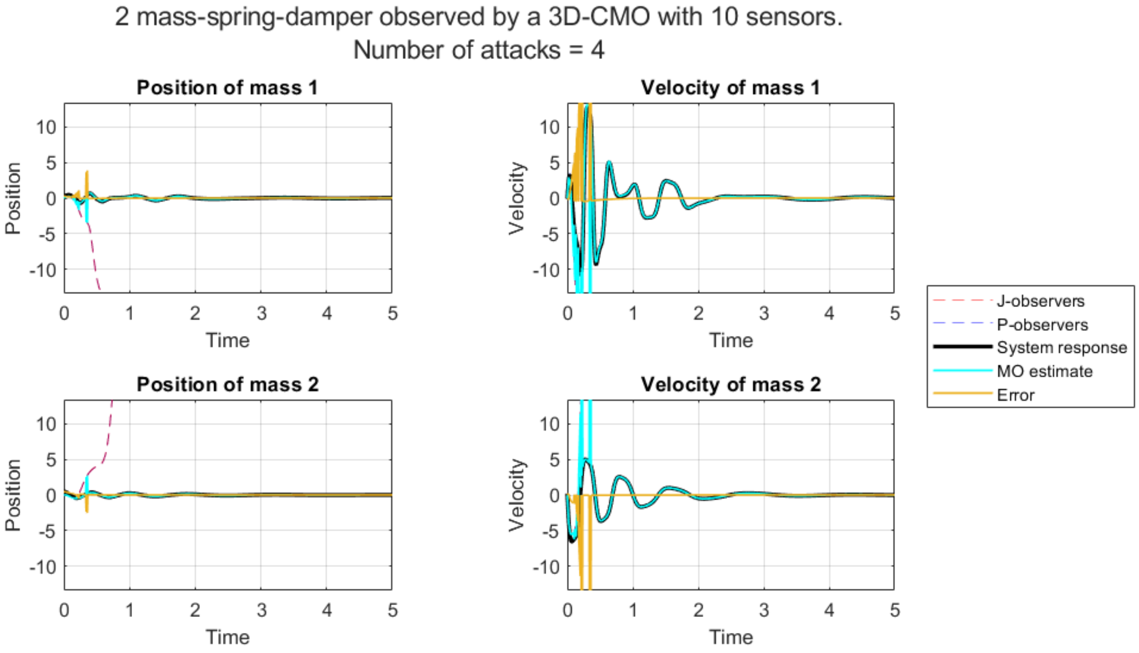


Figure 21: LAMO with $N_S = 10$ and $n_\phi = 2$, outputs 1,3,5 and 7 are under attack.

This is strange behaviour, since the condition in Equation (60) is clearly violated ($N_O = 25$ and $N_A = 20$). The selection procedure still decides on the correct state estimate after switching around rapidly at the start. This could be caused by large numerical inaccuracies in the attacked observers, since their values grow very large. Another possibility is that the loose sensor pairing allows the SAMO to function better under a large number of attacks.

8 Conclusion

This BEP has implemented three multi-observer variants and compared them based on memory usage. The 2D-CMO and the 3D-CMO store all state estimates individually, where 3D-CMO is much more memory efficient as compared to the 2D-CMO. Unlike the CMOs, the SSMO condenses all observers into a single state. In order to perform the final estimate selection procedure the original states are still required. These are recovered through transformation matrices. Although the shared state of the SSMO requires much less memory than all the individual states of the CMOs, the transformation matrices used to recover the original states still requires significant memory.

In a direct comparison between the 2D-CMO, 3D-CMO and the SSMO, the 3D-CMO uses the least memory. It should be noted that the 3D-CMOs memory requirements are still significantly too large for an implementation in a system with a large number of outputs.

The MOs extension to observe nonlinear systems remains incomplete, creating aggregate sensors seems to be a promising solution to the issues presented in this BEP. Although the SAMO seems to function well under the tested circumstances, no statements have been made about its implementation on general systems. The LAMO does not show immediate improvements over the SAMO.

Let us note several limitations of this BEP. The size comparison between multiple linear MOs does not consider systems with P-observers using a size larger than 1, selecting a different value can be required in order to properly observe a system. This could cause a different total MO size, although the size differences between the MO implementations are believed to be similar. This report has not considered MOs observing a system while feedback control is applied to the system and no effort has been made to select observer eigenvalues based on noise rejection and fast error suppression. Implementing feedback control could require changes to the structure of the CMO in order to select the desired eigenvalues. The Matlab implementations themselves have become overcomplicated in order to run the observers simultaneously.

In order to make a real-world MO implementation possible, more effort needs to be put in reducing the required memory. There is also still a need for further research into the nonlinear extension of the 3D-CMO, the feasibility of the SAMO is questionable regarding the increased size as compared to a linear MO. The unexplained behaviour of the LAMO could be key to expanding the functionality beyond the limit of the attacked aggregate sensors. Although it is also possible that it can never guarantee a secure state estimate beyond this limit.

References

[1] MATLAB ® Function Reference R2022b. Technical report, Mathworks, 9 2022.

[2] Frontier User Guide — OLCF User Documentation, 2024.

[3] Panos J Antsaklis and Anthony N Michel. *Linear Systems.* Birkhauser, Boston, 2nd edition, 2006.

[4] Richard Beals and Roderick Wong. Gamma, beta, zeta. In *Special Functions*, pages 18–56. Cambridge University Press, 6 2012.

[5] Roland Büchi. *State Space Control, LQR and Observer step by step introduction, with Matlab examples.* Books on Demand GmbH, Norderstedt, 2010.

[6] Michelle Chong, Wakaiki Masashi, and J Hespanha. Observability of linear systems under adversarial attacks. *2015 American Control Conference (ACC)*, pages 2439–2444, 2015.

[7] Michelle S. Chong, Henrik Sandberg, and Joao P. Hespanha. A secure state estimation algorithm for nonlinear systems under sensor attacks. *2020 59th IEEE Conference on Decision and Control (CDC)*, pages 5743–5748, 12 2020.

[8] Michelle S. Chong, Masashi Wakaiki, and João P. Hespanha. Memory Saving State-Sharing Multi-Observer for a Class of Multi-Observer-Based Algorithms. *IEEE Control Systems Letters*, 7:1772–1777, 2023.

[9] Derui Ding, Qing Long Han, Xiaohua Ge, and Jun Wang. Secure State Estimation and Control of Cyber-Physical Systems: A Survey. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 51(1):176–190, 1 2021.

[10] David P. Fidler. Was_Stuxnet_an_Act_of_War_Decoding_a_Cyberattack. *IEEE Security and Privacy*, 9:56–59, 8 2011.

[11] João P Hespanha. *Linear Systems Theory.* Princeton University Press, second edition edition, 2018.

[12] J. Kautsky, N.K. Nichols, and P Van Dooren. Robust pole assignment in linear state feedback. *International Journal of Control*, 41:1129–1155, 1985.

[13] K. Hassan Khalil. *Nonlinear Systems.* Prentice Hall, New Jersey, 2002.

[14] David Kushner. The Real Story of Stuxnet. *IEEE Spectrum*, 2013.

[15] André Lameiras. Industroyer: A cyber-weapon that brought down a power grid, 6 2022.

[16] David C.. Lay, Steven R.. Lay, and Judith. McDonald. *Linear algebra and its applications.* Pearson Education Limited, 2016.

[17] David R Mazur. Combinatorics: A Guided Tour. chapter 1. The Mathematical Association of America, 2010.

[18] Dimitrios Serpanos. False Data Injection Attacks on Sensor Systems. *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2022.

[19] Yasser Shoukry, Pierluigi Nuzzo, Alberto Puggelli, Alberto L Sangiovanni-Vincentelli, Sanjit A Seshia, and Paulo Tabuada. Secure State Estimation for Cyber Physical Systems under Sensor Attacks: A Satisfiability Modulo Theory Approach. *IEEE Transactions on Automatic Control*, 62:4917–4932, 2017.

[20] Stephen Boyd. Lecture 5 Observability and state estimation, 2009.

[21] Shui Yu. An Overview of DDoS Attacks. chapter 1, pages 1–14. Springer New York, 2014.

Appendices

A Stirling's approximation

n	Exact Factorial	Stirling's Approximation
1	1	0.9221
2	2	1.9190
3	6	5.8362
4	24	23.5062
5	120	118.0192
6	720	710.0782
7	5040	4.9804×10^{3}
8	40320	3.9902×10^{4}
9	362880	3.5954×10^{5}
10	3628800	3.5987×10^{6}
11	39916800	3.9616×10^{7}
12	479001600	4.7569×10^{8}
13	6.2270×10^{9}	6.1872×10^{9}
14	8.7178×10^{10}	8.6661×10^{10}
15	1.3077×10^{12}	1.3004×10^{12}
16	2.0923×10^{13}	2.0814×10^{13}
17	3.5569×10^{14}	3.5395×10^{14}
18	6.4024×10^{15}	6.3728×10^{15}
19	1.2165×10^{17}	1.2111×10^{17}
20	2.4329×10^{18}	2.4228×10^{18}

Table 9: Comparison of Exact Factorial and Stirling's Approximation

B SSMO transformation matrix

Let us now show that the transformation matrix $T = R_p R_q$,

$$R_p = \begin{bmatrix} \mathcal{B} & \mathcal{A}\mathcal{B} & \mathcal{A}^2\mathcal{B} & \cdots & \mathcal{A}^{n-1}\mathcal{B} \end{bmatrix}$$

$$R_q = \begin{bmatrix} I_l & q_1 I_l & q_2 I_l & \cdots & q_{n-1} I_l \\ 0 & I_l & q_1 I_l & \cdots & q_{n-2} I_l \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & I_l & q_1 I_l \\ 0 & \cdots & 0 & 0 & I_l \end{bmatrix}.$$

transforms the system (49) into controllable canonical form as in Equation (50), with the transformation

$$\begin{gathered} T\mathbf{A} = \mathcal{A}T \\ R_p R_q \mathbf{A} = \mathcal{A} R_p R_q. \end{gathered} \tag{64}$$

Let us start by expanding

$$\begin{aligned} R_q\mathbf{A} &= \begin{bmatrix} I_l & q_1 I_l & q_2 I_l & \cdots & q_{n-1} I_l \\ 0 & I_l & q_1 I_l & \cdots & q_{n-2} I_l \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & I_l & q_1 I_l \\ 0 & \cdots & 0 & 0 & I_l \end{bmatrix} \begin{bmatrix} -q_1 I_l & -q_2 I_l & \cdots & -q_{n-1} I_l & -q_n I_l \\ I_l & 0 & \cdots & 0 & 0 \\ 0 & I_l & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & I_l & 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ I_l & q_1 I_l & q_2 I_l & \cdots & q_{n-3} I_l & q_{n-2} I_l & 0 \\ 0 & I_l & q_1 I_l & \cdots & q_{n-4} I_l & q_{n-3} I_l & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & q_1 I_l & q_2 I_l & 0 \\ 0 & 0 & 0 & \cdots & I_l & q_1 I_l & 0 \\ 0 & 0 & 0 & \cdots & 0 & I_l & 0 \end{bmatrix}. \end{aligned}$$

We now premultiply this by R

$$\begin{aligned} R_p R_q \mathbf{A} &= \begin{bmatrix} \mathcal{B} & \mathcal{A}\mathcal{B} & \mathcal{A}^2\mathcal{B} & \cdots & \mathcal{A}^{n-1}\mathcal{B} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 0 \\ I_l & q_1 I_l & q_2 I_l & \cdots & q_{n-2} I_l & 0 \\ 0 & I_l & q_1 I_l & \cdots & q_{n-3} I_l & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ddots & q_1 I_l & 0 \\ 0 & 0 & 0 & \cdots & I_l & 0 \end{bmatrix} \\ &= \begin{bmatrix} \mathcal{A}\mathcal{B} \\ q_1\mathcal{A}\mathcal{B} + \mathcal{A}^2\mathcal{B} \\ q_2\mathcal{A}\mathcal{B} + q_1\mathcal{A}^2\mathcal{B} + \mathcal{A}^3\mathcal{B} \\ \cdots \\ q_{n-2}\mathcal{A}\mathcal{B} + q_{n-3}\mathcal{A}^2\mathcal{B} + \cdots + q_1\mathcal{A}^{n-2}\mathcal{B} + \mathcal{A}^{n-1}\mathcal{B} \\ 0 \end{bmatrix}^T \end{aligned}$$

Where by the Cayley-Hamilton theorem we can rewrite penultimate column as

$$\begin{bmatrix} \mathcal{A}\mathcal{B} \\ q_1\mathcal{A}\mathcal{B} + \mathcal{A}^2\mathcal{B} \\ q_2\mathcal{A}\mathcal{B} + q_1\mathcal{A}^2\mathcal{B} + \mathcal{A}^3\mathcal{B} \\ \vdots \\ -\mathcal{A}^n\mathcal{B} \\ 0 \end{bmatrix}^T$$

We now expand

$$\mathcal{A}R_pR_q = \begin{bmatrix}\mathcal{A}\mathcal{B} & \mathcal{A}^2\mathcal{B} & \mathcal{A}^3\mathcal{B} & \cdots & \mathcal{A}^n\mathcal{B}\end{bmatrix}\begin{bmatrix} I_l & q_1 I_l & q_2 I_l & \cdots & q_{n-1} I_l \\ 0 & I_l & q_1 I_l & \cdots & q_{n-2} I_l \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & I_l & q_1 I_l \\ 0 & \cdots & 0 & 0 & I_l \end{bmatrix}$$

$$= \begin{bmatrix} \mathcal{A}\mathcal{B} \\ q_1\mathcal{A}\mathcal{B} + \mathcal{A}^2\mathcal{B} \\ q_2\mathcal{A}\mathcal{B} + q_1\mathcal{A}^2\mathcal{B} + \mathcal{A}^3\mathcal{B} \\ \vdots \\ q_{n-1}\mathcal{A}\mathcal{B} + q_{n-2}\mathcal{A}^2\mathcal{B} + \cdots + \mathcal{A}^{n-1}\mathcal{B} \\ q_{n-1}\mathcal{A}\mathcal{B} + q_{n-2}\mathcal{A}^2\mathcal{B} + \cdots + \mathcal{A}^{n-1}\mathcal{B} + \mathcal{A}^n\mathcal{B} \end{bmatrix}^T,$$

where the bottom two rows can be simplified by using the Cayley-Hamilton theorem

$$\mathcal{A}R_pR_q = \begin{bmatrix} \mathcal{A}\mathcal{B} \\ q_1\mathcal{A}\mathcal{B} + \mathcal{A}^2\mathcal{B} \\ q_2\mathcal{A}\mathcal{B} + q_1\mathcal{A}^2\mathcal{B} + \mathcal{A}^3\mathcal{B} \\ \vdots \\ \mathcal{A}^n\mathcal{B} \\ 0 \end{bmatrix}^T,$$

which is equal to $RR_q\mathcal{A}$. We can now conclude that the matrix $T = RR_q$ satisfies (64). Now we will show the same for

$$\begin{aligned} T\mathbf{B} &= \mathcal{B} \\ R_pR_q\mathbf{B} &= \mathcal{B}. \end{aligned} \tag{65}$$

Let us expand

$$R_pR_q\mathbf{B} = \begin{bmatrix}\mathcal{B} & \mathcal{A}\mathcal{B} & \mathcal{A}^2\mathcal{B} & \cdots & \mathcal{A}^{n-1}\mathcal{B}\end{bmatrix}\begin{bmatrix} I_l & q_1 I_l & q_2 I_l & \cdots & q_{n-1} I_l \\ 0 & I_l & q_1 I_l & \cdots & q_{n-2} I_l \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & I_l & q_1 I_l \\ 0 & \cdots & 0 & 0 & I_l \end{bmatrix}\begin{bmatrix} I_l \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix}\mathcal{B} & \mathcal{A}\mathcal{B} & \mathcal{A}^2\mathcal{B} & \cdots & \mathcal{A}^{n-1}\mathcal{B}\end{bmatrix}\begin{bmatrix} I_l \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} = \mathcal{B}$$

which shows that (65) holds.

C Matlab code Chapter 3

Listing 1: jordan_form.m

```matlab
% This code simulates the response of system matrix A on tspan by
% caluclating the Jordan normal form and calculating x at each timestep in
% tspan. It then plots the result in a tiledplot.
close all; clearvars;
tspan = 0:0.01:5;
x0 = [0.3;-0.1;0.5;0.2];

A = [0 1 0 0; -15 -2 15 2; 0 0 0 1; 0 0 -15 -2];
[P,J] = jordan(A);
invP = inv(P);

x = zeros(4,size(tspan,2));
for t = 1:1:size(tspan,2)
    x(:,t) = P*expm(tspan(t)*J)*invP*x0;
end

fig = tiledlayout('flow');
sgtitle("2 mass-spring-damper")

for l = 1:1:4
    nexttile
    plot(tspan,x(l,:),LineWidth=2,Color='black')
    xlabel('Time')
    mass = floor((l+1)/2);

    if (-1)^l == -1
        % odd
        ylabel('Position')
        title(['Position of mass ' num2str(mass)])
    else
        % even
        ylabel('Velocity')
        title(['Velocity of mass ' num2str(mass)])
    end

    ylim([-1.5 1.5]);
    grid on;

end

set(gcf, 'Position', 0.4*get(0, 'Screensize'));
hold on;
```

D Matlab code Chapter 4

Listing 2: sizeComparison.m

```
clearvars;
clc;
% close all;

%% System parameter setup
m = 50;
k = 1;
st = 1;

n = 1:st:m;
N = 1:st:m;

lim = size(N,2);

size2d   = zeros(lim);
size3d   = zeros(lim);
sizessmo = zeros(lim);

%% size calculations
for i = 1:1:lim
    for j = 1:1:lim
        M = floor((N(j)-1)/2);
        J = N(j)-M;
        P = 1;

        NJ = nchoosek(N(j),J);
        NP = nchoosek(N(j),P);

        NS = NJ+NP;

        size2d(i,j) = n(i)*NS + (n(i)^2)*(NS^2) + n(i)*k*NS + k;
        size3d(i,j) = n(i)*NS + (n(i)^2)*NS + (n(i)^2)*NS;
        sizessmo(i,j) = n(i)*NS + (n(i)^2)*(N(j)^2) + (n(i)^2)*N(j) + (n(i)^2)*N(j)*NS;
    end
end

GB(:,:,1) = log10(size2d.*1e-9);
GB(:,:,2) = log10(size3d.*1e-9);
GB(:,:,3) = log10(sizessmo.*1e-9);


%% Plots
zmax = max(max(GB(:,:,1)));

tcl = tiledlayout(1,3);

for i = 1:1:3
    nexttile()
    s = pcolor(N,n,GB(:,:,i));
    s.EdgeColor = 'none';
    xlabel('N')
    ylabel('n')
    title('2D CMO')
    clim([-10 zmax])
end

cb = colorbar();
cb.Layout.Tile = 'east'; % Assign colorbar location
set(gcf, 'Position', [1 1 0.4 0.2].*get(0, 'Screensize'))

dif = (sizessmo - size3d)*1e-9;
rat = sizessmo./size3d;
```

E Matlab code Chapter 6

Firstly, the main scripts is shown. This is the script than should be executed to run the model. Secondly, all classes are shown and finally all functions are shown. The documentation that sits between the function definitions and the start of the code are made by inputting the full function into ChatGPT-4 and with the prompt: ”Can you provide documentation for this function? <code in between these brackets>” All output has been checked and corrected by the author in this report, so that the documentation accurately describes the code. All code can also be found in the github repo: https://github.com/julian1501/MultiObserver.

Listing 3: mainClassScript.m

```matlab
clearvars; close all;

% initialize bools indicating user input (dialog) and if the results should
% be plotted (plot)
dialog = true; Plot = true;
fprintf(['\n' repmat('-',1,100) '\n'])
inputs = inputDialog(dialog);

% extract input string values into number type variables
sysNum = int32(str2double(inputs{1}));
numOutputs = int32(str2double(inputs{2}));
if strcmp(inputs{3},'max')
    numAttackedOutputs = floor((numOutputs-1)/2);
else
    numAttackedOutputs = int32(str2double(inputs{3}));
end
attackedOutputs = str2num(inputs{4});
% correct number of attakced outputs
if ~isempty(attackedOutputs)
    numAttackedOutputs = size(attackedOutputs,2);
end

if strcmp(inputs{5},'max')
    numOutputsJObservers = numOutputs-numAttackedOutputs;
else
    numOutputsJObservers = int32(str2double(inputs{5}));
end
if strcmp(inputs{6},'max')
    numOutputsPObservers = numOutputs-2*numAttackedOutputs;
else
    numOutputsPObservers = int32(str2double(inputs{6}));
end
tspan = str2num(inputs{7});
x0Options = str2num(inputs{8})';
whichMO = str2num(inputs{9});
linear = int32(str2double(inputs{10}));
aggregate_grouping = inputs{11};
noiseVariance = str2double(inputs{12});


%% CALCULATIONS
% display information in the command window
fprintf('The number of senors is %3.0f: \n',numOutputs)
fprintf('The number of attacked sensors %3.0f: \n',numAttackedOutputs)

try numJObservers = nchoosek(numOutputs,numOutputsJObservers);
    catch ME
        numJObservers = numOutputs;
end
numPObservers = nchoosek(numOutputs,numOutputsPObservers);

fprintf('The size of each J observer is: %3.0f \n',numOutputsJObservers)
fprintf('The size of each P observer is: %3.0f \n',numOutputsPObservers)
fprintf('The number of J observers is: %3.0f \n',numJObservers)
fprintf('The number of P observers is: %3.0f \n',numPObservers)

% system definition
sys = msd(linear,sysNum,1,15,2.0);

% check if the system is stable
if ~isMatrixStable(sys.A)
    warning('The system is unstable')
end
if sys.D ~= 0
    error('Implementation for systems with D still needs work.')
end

% check if there is no nonlinearity in the 2D CMO
if ~linear && whichMO(1) == 1
    error('The 2D CMO does not support nonlinear systems.')
end

% setup the attack and noise
Attack = attack(numOutputs,numAttackedOutputs,attackedOutputs);
Noise = noise(numOutputs,tspan,noiseVariance);

% setup the J and P observers
Pmo = mo(sys,Attack,numOutputs,numOutputsPObservers,aggregate_grouping);
Jmo = mo(sys,Attack,numOutputs,numOutputsJObservers,aggregate_grouping);

% find which P observers are subobservers of J
sys.COutputs = Jmo.COutputs;
[numOfPsubsetsInJ,PsubsetOfJIndices] = findIndices(Jmo,Pmo);

% Setup the different type of multi-observers
CMO2D = 0; CMO3D = 0; SSMO = 0;
if whichMO(1) == 1
    CMO2D = cmo2d(sys,Jmo,Pmo);
end
if whichMO(2) == 1
    CMO3D = cmo3d(sys,Jmo,Pmo);
end
if whichMO(3) == 1
    SSMO  =  ssmo(sys,Jmo,Pmo);
end


```

```
[x0, xIds] = x0setup(x0Options,whichMO,sys,Jmo,Pmo);


% create waitbar
wb = waitbar(0,'Solver is currently at time: 0','Name','Solving the ODE');


[t,x] = ode45(@(t,x) multiObserverODE(wb,tspan(2),sys,t,x,Attack,CMO2D,CMO3D,SSMO,whichMO,Noise,Jmo,Pmo,xIds),tspan,x0);
t = t';
x = x';

close(wb)

state = x(1:sys.nx,:);
if whichMO(1) == 1
    CMO2Dest = x(xIds.xcmo2dStart:xIds.xcmo2dEnd,:);
    % create waitbar
    wb = waitbar(0,'Selection is currently at time: 0','Name','Selecting best estimates 2D-CMO');
    CMO2DbestEst = sbeCPU([state; CMO2Dest],size(t,2),PsubsetOfJIndices,numOfPsubsetsInJ,Jmo,Pmo,sys,wb);
    CMO2Derr = state - CMO2DbestEst;
    close(wb)
end

if whichMO(2) == 1
    CMO3Dest = x(xIds.xcmo3dStart:xIds.xcmo3dEnd,:);
    % create waitbar
    wb = waitbar(0,'Selection is currently at time: 0','Name','Selecting best estimates 3D-CMO');
    CMO3DbestEst = sbeCPU([state; CMO3Dest],size(t,2),PsubsetOfJIndices,numOfPsubsetsInJ,Jmo,Pmo,sys,wb);
    CMO3Derr = state - CMO3DbestEst;
    close(wb)
end

if whichMO(3) == 1
    SSMOz = x(xIds.xssmoStart:xIds.xssmoEnd,:);
    SSMOest = flatten(pagemtimes(SSMO.T,SSMOz));
    % create waitbar
    wb = waitbar(0,'Selection is currently at time: 0','Name','Selecting best estimates SSMO');
    [SSMObestEst,jBE] = sbeCPU([state; SSMOest],size(t,2),PsubsetOfJIndices,numOfPsubsetsInJ,Jmo,Pmo,sys,wb);
    SSMOerr = state - SSMObestEst;
    close(wb)
end

% calculate difference
if sum(whichMO) > 1
    if whichMO(1) + whichMO(2) == 2
        diff = sqrt((CMO2Dest - CMO3Dest).^2);
    elseif whichMO(1) + whichMO(3) == 2
```

```
        diff = sqrt((CMO2Dest - SSMOest).^2);
    elseif whichMO(2) + whichMO(3) == 2
        diff = sqrt((CMO3Dest - SSMOest).^2);
    elseif sum(whichMO) == 3
        ... % Standard deviation?
        diff = 0;
    end
    sco = max(max(diff));
    disp(sco)
    if sco < 1e-3
        disp('Tolerance within numerical tolerance.')
    else
        disp('The solutions are not similar.')
    end
end

% Only the last run plot stays open, run the code block by clicking the
% left blue bar in order to plot a specific multi observer.
%% 2D CMO plot
if Plot && whichMO(1) == 1
      MOplot(t,[state; CMO2Dest],CMO2Derr,CMO2DbestEst,sys,CMO2D,Jmo,Pmo);
end
%% 3D CMO plot
if Plot && whichMO(2) == 1
      MOplot(t,[state; CMO3Dest],CMO3Derr,CMO3DbestEst,sys,CMO3D,Jmo,Pmo);
end
%% SSMO plot
if Plot && whichMO(3) == 1
      MOplot(t,[state; SSMOest],SSMOerr,SSMObestEst,sys,SSMO,Jmo,Pmo);
end
```

Now all classes follow alphabetically

Listing 4: attack.m

```
classdef attack
% ATTACK Class
%
% The 'attack' class models an attack on a specified number of outputs.
% It allows for the initialization of the system, random selection of
% attacked outputs, and assignment of attack signals.
%
% This documentation was written by ChatGPT. Th
%
% Properties:
% ------------
% - 'numOutputs' (integer):
%   The total number of outputs in the system.
%
% - 'numAttacks' (integer):
%   The number of outputs to be attacked.
%
% - 'attackList' (array):
%   A column vector of size 'numOutputs x 1' where each element is 1 for attacked outputs
%   and 0 otherwise.
%
% Methods:
% --------
% - 'attack(numOutputs, numAttacks, attackedOutputs':
%   Constructor that initializes an 'attack' object with the given number of outputs and
%   attacks. It ensures that 'numOutputs > 2 * numAttacks' and creates an array
%   'attackList' to indicate which outputs are attacked. If attackedOutputs is
%   empty, the attacked outputs are randomly selected by selectAB. If the condition
%   is not met, an error is thrown.
%
%   Example Usage:
%       obj = attack(10, 3);
%
%   Outputs:
%       Displays the indices of the attacked outputs and initializes 'attackList'.
%
% - '[corruptA, corruptB, uncorrupt] = selectAB()':
%   Selects two subsets ('corruptA' and 'corruptB') of size 'numAttacks' each from
%   the range '1:numOutputs'. The remaining elements form the 'uncorrupt' set.
%
%   Outputs:
%       - 'corruptA': Array of indices for the first corrupted subset.
%       - 'corruptB': Array of indices for the second corrupted subset.
%       - 'uncorrupt': Array of indices for uncorrupted outputs.
%
% Internal Helper Function:
% -------------------------
% - 'selectRandomSubset(set, M)':
%   (Not defined in the provided code) Selects a random
%   subset of size 'M' from the input 'set' and returns the selected subset and the
%   remaining elements. selectRandomSubset.m
%
% Notes:
% ------
% - The 'selectAB' method is intended to be called internally within the constructor
%   to set up the initial attack configurations.
%
% Error Conditions:
% -----------------
% - If 'numOutputs' is not greater than '2 * numAttacks', an error is raised with a
%   descriptive message. This ensures that sufficient outputs remain uncorrupted.

```

```

    properties
        numOutputs
        numAttacks
        attackList
    end

    methods
        function obj = attack(numOutputs,numAttacks,attackedOutputs)
            % attack creates an array attack
            obj.numOutputs = numOutputs;
            obj.numAttacks = numAttacks;
            if ~ (numOutputs > 2* numAttacks)
               warning('The number of outputs is not larger then twice the number of attacked outputs %3.0f <= %3.0f',numOutputs,numAttacks);
            end

            if size(attackedOutputs) == [0,0]
                attackedOutputsA = selectAB(obj);
            else
                attackedOutputsA = attackedOutputs;
            end

            fprintf('The attacked outputs are: \n')
            disp(attackedOutputsA)
            % loop over attacked outputs and add attack signal
            attackList = zeros(obj.numOutputs,1);
            for i = 1:1:obj.numAttacks
                outputA = attackedOutputsA(i);
                attackList(outputA) = 1;
            end
            obj.attackList = attackList;
        end

        function [corruptA,uncorrupt] = selectAB(obj)
            % [corruptA,corruptB,uncorrupt] = selectAB(CMOdict) selects two sets
            % (A,B)
            % sized M out of the set 1:1:numOutputs.
            %
            % For example:
            %   - numOutputs = 12
            %     M = 5
            %       -> corruptA  = [1,2,3,4,5]
            %          corruptB  = [6,7,8,9,10]
            %          uncorrupt = [11,12]

            M = obj.numAttacks;
            outputSet = 1:1:obj.numOutputs;

            % Select the first M outputs to be in set A
            [corruptA, uncorrupt] = selectRandomSubset(outputSet,M);

        end
    end
end
```

Listing 5: cmo2d.m

```
classdef cmo2d
% cmo2d- creates a 2D conventional multi-observer using a system, the set
% of J-observers and the set of P-observers.
%
% Syntax:  obj = cmo2d(sys,Jmo,Pmo)
%
% Inputs:
%    sys - System model containing:
%        - 'nx': Number of states.
%        - 'A': State transition matrix.
%        - 'COutputs': Output matrix.
%        - 'E': Coupling matrix for nonlinear springs.
%        - 'NLsize': Size of the nonlinear system components.
%        - 'Linear': Logical indicating whether the system is linear.
%    Jmo - mo object for all J-observers
%        - 'sys': (same as input)
%        - 'nx': number of states
%        - 'ny': number of distinct outputs
%        - 'nu': number of inputs
%        - 'numOutputs': number of multi-observer outputs
%        - 'numObservers': number of distinct observers in this mo object
%        - 'numOutputsObservers': number of outputs per observer
%        - 'Ci': (numOutputsObserver,nx,numObservers) sized array
%           containing all Cis on each slice along the third dimension.
%        - 'CiIndices': (numObservers,numOutputsObservers) sized array
%           where each row contains the indices of the outputs used in the
%           observer.
%    Pmo - mo object for all J-observers
%
% Properties:
%    sys - see input
%    numObservers - total number of observers
%    A - shared state matrix for all observers
%    E - shared 'input' matrix for all observers
%    attack -
%    B
%    u
%    x0
%    PSubsetOfJIndices
%
% See also: cmo3d,  ssmo

    properties
        Name
        sys
        numObservers
        numOutputs
        Attack % class
        A
        F
        attack
        B
        u
        x0
        PSubsetOfJIndices
    end

    methods
        function obj = cmo2d(sys,Jmo,Pmo)
            obj.Name = "2D-CMO";
            obj.sys = sys;
            obj.numObservers = Jmo.numObservers + Pmo.numObservers;
            obj.numOutputs = Jmo.numOutputs;
            obj.Attack = Jmo.Attack;

            [ApLCJ,LCJ] = ApLCSetup(Jmo);
            [ApLCP,LCP] = ApLCSetup(Pmo);

            % A matrix subblocks
            A13 = zeros(Jmo.numObservers*sys.nx,Pmo.numObservers*sys.nx);
            A22 = A13';

            % reshape all matrices to 2d format
            ApLCJ = diag3d(ApLCJ);
            LCJ = flatten(LCJ);
            ApLCP = diag3d(ApLCP);
            LCP = flatten(LCP);

            obj.A = [-LCJ, ApLCJ,   A13;
                     -LCP,   A22, ApLCP];

            B = repmat(sys.B,Jmo.numObservers+Pmo.numObservers,1);
            LJ = diag3d(Jmo.Li);
            LP = diag3d(Pmo.Li);
            L12 = zeros(size(LJ,1),size(LP,2));
            L21 = zeros(size(LP,1),size(LJ,2));
            L = [-LJ,L12; L21,-LP];
            obj.F = [B L];

            obj.attack = cat(1,reshape(Jmo.attack3d,[],1,1),reshape(Pmo.attack3d,[],1,1));

        end

    end
end
```

Listing 6: cmo3d.m

```
classdef cmo3d
% CMO3D Class
%
```

```
% The 'cmo3d' class creates a 3D Conventional Multi-Observer (CMO) system
% by combining a system object ('sys'), a set of J-observers ('Jmo'), and a
% set of P-observers ('Pmo'). The observers are stored in 3D arrays
%
% This documentation was written by ChatGPT.
%
% Syntax:
% -------
% 'obj = cmo3d(sys, Jmo, Pmo)'
%
% Inputs:
% -------
% - 'sys' (msd object):
%   The system object containing all relevant system details.
%
% - 'Jmo' (mo object):
%   An object containing the J-observers.
%
% - 'Pmo' (mo object):
%   An object containing the P-observers.
%
% Properties:
% -----------
% - 'Name' (string):
%   A descriptive name for the system, initialized as ''3D-CMO''.
%
% - 'sys' (msd object):
%   The input system object.
%
% - 'numObservers' (integer):
%   The total number of observers, calculated as the sum of J-observers and P-observers.
%
% - 'numOutputs' (integer):
%   The total number of outputs.
%
% - 'ApLC' (3D array):
%   The 'A+LC' matrices for each observer, stacked along the third dimension.
%
% - 'LC' (3D array):
%   The 'LC' matrices for each observer, stacked along the third dimension.
%
% - 'C' (3D array):
%   The 'C' matrices for each observer, stacked along the third dimension.
%
% - 'L' (3D array):
%   The 'L' matrices for each observer, stacked along the third dimension, with appropriate padding for alignment.
%
% - 'attack' (3D array):
%   The attack list ('Attack.attackList') sliced based on the outputs for
```

```
    each observer, stacked along the third dimension.
%
% - 'B' (3D array):
%   The shared input matrix for all observers, repeated along the third dimension.
%
% - 'E' (3D array):
%   The shared nonlinear contribution matrix for all observers, repeated along the third dimension.
%
% - 'x0' (array):
%   Initial conditions for the observers (not explicitly initialized in the constructor).
%
% - 'Attack' (attack object):
%   The attack object associated with the system, inherited from 'Jmo'.
%
% Methods:
% --------
% - cmo3d(sys, Jmo, Pmo):
%   Constructor method that initializes the 'cmo3d' object, combining the properties of the 'sys', 'Jmo', and 'Pmo' inputs.
%   Key steps:
%     - Retrieves and combines the 'ApLC' and 'LC' matrices for J- and P-observers using the 'systemStarSetup3D' function.
%     - Aligns dimensions of matrices ('C', 'L', 'attack') by adding appropriate padding.
%     - Constructs shared properties ('B', 'E') by replicating the respective matrices from 'sys'.
%
% Example Usage:
% --------------
% sys = msd();                 % Create a system object
% Jmo = mo();                  % Define a set of J-observers
% Pmo = mo();                  % Define a set of P-observers
% cmo = cmo3d(sys, Jmo, Pmo); % Create a 3D-CMO object
%
% Notes:
% ------
% - This class assumes the availability of the 'systemStarSetup3D' function for processing observer matrices.
% - Padding ensures that the dimensions of J- and P-observer matrices align correctly for stacking operations.
% - The 'Attack' property directly links to the attack object of the 'Jmo'.
%
% Dependencies:
% -------------
% - systemStarSetup3D.m : External function required for processing observer matrices.
% - msd.m and mo.m: Classes representing the system and observer models, respectively.
%
% See Also:
```

```
% ---------
% - ssmo.m : Related class or function (as mentioned in the header comment).


    properties
        Name
        sys
        numObservers
        numOutputs
        ApLC
        LC
        C
        L
        attack3d
        B
        E
        x0
        Attack
        Jmo
        Pmo
    end

    methods
        function obj = cmo3d(sys,Jmo,Pmo)
            obj.sys = sys;
            obj.Name = '3D-CMO';
            obj.Attack = Jmo.Attack;
            obj.numObservers = Jmo.numObservers + Pmo.numObservers;
            obj.numOutputs = Jmo.numOutputs;
            obj.Jmo = Jmo;
            obj.Pmo = Pmo;


            [ApLCJ,LCJ] = ApLCSetup(Jmo);
            [ApLCP,LCP] = ApLCSetup(Pmo);

            % Pad the right side of LP with zeros to match the cross sectional size of
            % LJ
            padding = zeros(Jmo.numIndOutputsObservers-Pmo.numIndOutputsObservers,Jmo.nx,Pmo.numObservers);
            obj.C = cat(3,Jmo.Ci,cat(1,Pmo.Ci,padding));
            padding = zeros(Jmo.nx,Jmo.numIndOutputsObservers-Pmo.numIndOutputsObservers,Pmo.numObservers);
            PmoLPadded = cat(2,Pmo.Li,padding);
            padding = zeros(Jmo.numIndOutputsObservers-Pmo.numIndOutputsObservers,1,Pmo.numObservers);
            PAttackPadded = cat(1,Pmo.attack3d,padding);


            % Create page arrays of each ApLC
            obj.ApLC = cat(3,ApLCJ,ApLCP);
            obj.LC   = cat(3,LCJ,LCP);
            obj.L    = cat(3,Jmo.Li,PmoLPadded);
            obj.attack3d = cat(3,Jmo.attack3d,PAttackPadded);
            obj.B    = repmat(sys.B,1,1,obj.numObservers);
            obj.E    = repmat(sys.E,1,1,obj.numObservers);

        end

    end

end
```

Listing 7: ssmo.m

```
classdef ssmo
% ssmo Class
%
% The 'ssmo' class sets up a state-space multi-observer (SSMO) for a given
% system, using provided configurations of primary (Jmo) and secondary
% (Pmo) observers.
%
% Documentation written by ChatGPT.
%
% Properties:
% -----------
% - 'Name': A string representing the name of the SSMO object.
% - 'sys': The original system model for which the SSMO is constructed.
% - 'numOutputs': The number of outputs in the SSMO.
% - 'T': The transformation matrices for the observers.
% - 'A': The state-space A matrix for the overall SSMO system.
% - 'B': The state-space B matrix for the overall SSMO system.
% - 'COutputs': The output matrices for the observers.
% - 'Attack': Attack parameters from the primary observer (Jmo).
%
% Constructor ('ssmo'):
% ---------------------
% Creates an instance of the 'ssmo' class by integrating the primary (Jmo)
% and secondary (Pmo) observer models into the overall SSMO framework.
%
% Inputs:
% -------
% - 'sys': The original system model (state-space representation).
% - 'Jmo': The primary observer model (with required observer properties).
% - 'Pmo': The secondary observer model (with required observer properties).
%
% Functionality:
% --------------
% 1. Stores the input 'sys' model and observer-related configurations in
%    the SSMO object properties.
% 2. Pads the observer gain matrices ('LJ' and 'LP') to make them
%    compatible with the full output.
% 3. Computes the transformed state-space matrices 'AJp', 'BJp' for the
%    primary observer, and 'APp', 'BPp' for the secondary observer.
% 4. Derives the transformation matrices ('TJ' and 'TP') for the observers
%    using the roots of the characteristic polynomial derived from the
%    eigenvalues of 'Jmo'.
% 5. Constructs the state-space matrices ('A', 'B') for the combined SSMO
%    system.
%
% Methods:
% --------
% - 'ssmo': Constructor to initialize the 'ssmo' object with system and
%   observer models.
%
% Implementation Steps:
% ---------------------
% 1. The 'sys', 'Jmo', and 'Pmo' inputs are assigned to corresponding
%    properties of the 'ssmo' object.
% 2. Observer gain matrices ('LJ', 'LP') are padded using the 'pad3DL'
%    function to align them with the full output space.
% 3. The 'systemPSetup' function is used to derive 'AJp', 'BJp' (for Jmo)
%    and 'APp', 'BPp' (for Pmo), representing the system matrices for the
%    primary and secondary observers, respectively.
% 4. Transformation matrices are derived by solving for polynomial
%    coefficients ('q') from the eigenvalues of 'Jmo' and passing them
%    through the 'SSMOTransformationSetup' function for both 'Jmo' and
%    'Pmo'.
% 5. The final state-space matrices 'A' and 'B' for the SSMO system are
%    constructed using 'ssmoSysSetup'.
%
% Notes:
% ------
% - The 'pad3DL', 'systemPSetup', 'rootsToCoefficients',
%   'SSMOTransformationSetup', and 'ssmoSysSetup' functions must be
%   available in the same workspace or path for this class to function
%   properly.
% - The transformation matrices ('T') and system matrices ('A', 'B') are
%   stored as part of the 'ssmo' object for use in further simulations or
%   analysis.
%
% Example Usage:
% --------------
% % Define the system model
% sys = msd()
%
% % Define the primary and secondary observers
% Jmo = mo();
% Pmo = mo();
%
% % Create an instance of the SSMO class
% ssmoObj = ssmo(sys, Jmo, Pmo);
%
% % Access the state-space matrices
% A = ssmoObj.A;
% B = ssmoObj.B;
%
% See also:
% ---------
```

```matlab
% pad3DL, systemPSetup, rootsToCoefficients, SSMOTransformationSetup, ssmoSysSetup


    properties
        Name
        sys
        numOutputs
        T
        A
        B
        COutputs
        Attack
        Jmo
        Pmo
    end

    methods
        function obj = ssmo(sys,Jmo,Pmo)
            %UNTITLED Construct an instance of this class
            %   Detailed explanation goes here
            % Define Ap and Bp for each observer
            if sys.Linear == 0
                warning("The SSMO uses aggregate sensors in this git commit, this is not supposed to work. Revert to a commit before the aggregate sensors to replicate results from the report.")
            end
            obj.sys = sys;
            obj.Name = 'SSMO';
            obj.Attack = Jmo.Attack;
            obj.numOutputs = Jmo.numOutputs;
            obj.COutputs = Jmo.COutputs;
            obj.Jmo = Jmo;
            obj.Pmo = Pmo;

            % Pad the L matrices so that they are compatible with the full output
            LJpadded = pad3DL(Jmo);
            LPpadded = pad3DL(Pmo);

            % Define Ap and Bp for each observer
            [AJp,BJp] = systemPSetup(Jmo,LJpadded);
            [APp,BPp] = systemPSetup(Pmo,LPpadded);

            % Derive the transfomration matrix for each observer
            q = rootsToCoefficients(Jmo.eigenvalues);
            q = q(2:end);
            [TJ] = SSMOTransformationSetup(AJp,BJp,q,Jmo);
            [TP] = SSMOTransformationSetup(APp,BPp,q,Pmo);
            obj.T = cat(3,TJ,TP);

            % Define THE A and B matrices
            [obj.A,obj.B] = ssmoSysSetup(q,obj);


        end

    end
end
```

Listing 8: mo.m

```
classdef mo
% mo Class
%
% The 'mo' class defines a multi-observer system that observes a given
% dynamic system. It contains the configuration and initialization of
% observers and includes attack handling, observer matrices (A and L), as
% well as the setup for different system outputs.
%
% Documentation written with help of ChatGPT.
%
% Properties:
% -----------
% - 'sys': The system that the multi-observer observes. It includes system
%   matrices like 'A', 'B', and 'C'.
% - 'nx': The number of system states, which corresponds to the size of the
%   'A' matrix.
% - 'ny': The number of outputs of the system (e.g., number of sensors).
% - 'nu': The number of inputs to the system.
% - 'numOutputs': The total number of outputs for the whole system.
% - 'numObservers': The number of J-observers used in the multi-observer setup.
% - 'numOutputsObservers': The number of outputs for each J-observer.
% - 'Ci': The outputs associated with each J-observer.
% - 'CiIndices': The indices of the outputs for each J-observer.
% - 'attack3d': A 3D matrix storing attack values for each J-observer.
% - 'Ai': A cell array that stores all the A matrices for each J-observer.
% - 'Li': A cell array that stores all the L matrices for each J-observer.
% - 'eigenvalues': A vector storing system eigenvalues.
% - 'COutputs': The matrix of all outputs of the system.
% - 'Attack': The attack object, which contains information about the
%   attacks on the system.
%
% Methods:
% --------
% - 'mo(sys, Attack, numOutputs, numOutputsObserver)': Constructor to
%   initialize an instance of the 'mo' class.
%   - 'sys': The system being observed. This includes matrices such as 'A',
%     'B', 'C', etc.
%   - 'Attack': The attack object, which specifies the details of attacks.
%   - 'numOutputs': The total number of outputs for the system.
%   - 'numOutputsObserver': The number of outputs for each observer.
%
% Constructor Description:
```

```
% ------------------------
% The constructor initializes the 'mo' class object by setting various
% properties including system matrices, the number of outputs and observers,
% attack configurations, and observer matrices. It performs checks to ensure
% that the number of system outputs is greater than twice the number of
% attacked outputs and initializes the necessary matrices and configurations
% for both system and observers.
%
% Initialization Steps:
% ---------------------
% - The constructor checks the system's number of outputs and ensures the
%   number of attacked outputs is valid.
% - It computes the 'Ci' matrix (outputs associated with each observer) and
%   the 'CiIndices' (indices for each observer).
% - A 3D matrix 'attack3d' is created to store the attack values for each
%   J-observer.
% - It sets default eigenvalue options for the system and assigns them to
%   the system.
% - The constructor calls a function 'defineObservers()' to initialize the
%   observer matrices 'Ai' and 'Li'.
%
% Example:
% --------
% To create an instance of the 'mo' class, you would use:
% sys = msd();  % A predefined system object
% Attack = attack();  % A predefined attack object
% numOutputs = 6;
% numOutputsObserver = 3;
% moObj = mo(sys, Attack, numOutputs, numOutputsObserver);
% This would initialize an 'mo' object with the provided system and attack
% details.
%
% Notes:
% ------
% - The constructor automatically handles system and observer initialization.
% - The class uses predefined eigenvalue options, which can be modified in
%   the future for different systems.
% - The 'COutputs', 'Ci', and 'attack3d' matrices are computed during
%   initialization using helper functions ('CNSetup', 'CsetSetup').
%
% See also:
% ---------
% CNSetup, CsetSetup, defineObservers, msd, attack


  properties
    % System that the multi-observer observes
    sys
    % Number of system states (size of A matrix)
```

```
        nx
        % Number of different possible outputs
        ny
        % Number of inputs
        nu
        % Number of outputs of the whole system (e.g. number of sensors)
        numOutputs
        % Number of J-observers
        numObservers
        % Number of outputs of each J-observer (often
        % numOuptus-numAttackedOutputs)
        numOutputsObservers
        numIndOutputsObservers
        % Outputs of each J-observer
        Ci
        % Indices of each J-observer
        CiIndices
        % 3D matrix that stores all the attack values for J observers
        attack3d
        % All A matrices for the J observer
        Ai
        % All L matrices for the J observer
        Li
        % system eigenvalues
        eigenvalues
        % all outputs of the system
        COutputs

        Attack
    end

    methods
        function obj = mo(sys,Attack,numOutputs,numOutputsObserver,aggregate_grouping)
            %UNTITLED6 Construct an instance of this class
            %   Detailed explanation goes here
            obj.sys = sys;
            obj.nx = size(sys.A,1);
            obj.ny = size(sys.C,1);
            obj.nu = size(sys.B,2);
            obj.Attack = Attack;
            % Check whether numoutputs > 2*numAttackedOutputs
            if ~ (numOutputs > 2* Attack.numAttacks)
               warning('The number of outputs is not larger then twice the number of attacked outputs %3.0f <= %3.0f',numOutputs,Attack.numAttacks);
            end
            obj.numOutputs = numOutputs;

            obj.numOutputsObservers = numOutputsObserver;

            obj.COutputs = CNSetup(obj);
            [Ci,CiIndices,attack3D] = CsetSetup(obj.COutputs,Attack,aggregate_grouping,obj);

            obj.numObservers = size(Ci,3);
            obj.numIndOutputsObservers = size(Ci,1);

            obj.Ci = Ci;
            obj.CiIndices = CiIndices;
            obj.attack3d = attack3D;

            eigenvalueOptions = [-3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14];
            obj.eigenvalues = eigenvalueOptions(1:obj.nx);

            [Ai,Li] = defineObservers(sys.A,Ci,obj.eigenvalues,obj);

            obj.Ai = Ai;
            obj.Li = Li;

        end

    end

end
```

Listing 9: msd.m

```
classdef msd
% msd Class
%
% The 'msd' class defines a mass-spring-damper (MSD) system using state-space
% representation. It sets up a linear or nonlinear system with multiple masses
% in series and calculates the system matrices A, B, C, D, and optionally E
% for nonlinearities.
%
% Documentation written by ChatGPT.
%
% Properties:
% -----------
% - 'numMass': The number of masses in the system.
% - 'Name': The name of the system, constructed using the number of masses.
% - 'A': The state-space A matrix, describing the system's dynamics.
% - 'B': The state-space B matrix, representing the system inputs.
% - 'C': The state-space C matrix, representing the system outputs.
% - 'D': The state-space D matrix, which typically relates inputs to outputs.
% - 'E': The nonlinear E matrix (if applicable), defining the nonlinear behavior.
% - 'Linear': A boolean indicating whether the system is linear or nonlinear.
% - 'nx': The number of states in the system, corresponding to the size of the A matrix.
% - 'ny': The number of outputs in the system, corresponding to the size of the C matrix.
% - 'nu': The number of inputs to the system, corresponding to the size of the B matrix.
% - 'xsize': The size of the state vector (calculated based on the number of masses).
% - 'k': The spring constants for each mass.
% - 'a': A constant used in the nonlinear model (default value of 10).
% - 'm': The masses in the system.
% - 'NLsize': The size of the nonlinear matrix E (if applicable).
% - 'COutputs': The matrix of valid outputs for the system.
%
% Methods:
% --------
% - 'msd(linear, numMass, m, k, c)': Constructor to initialize an instance of the
%   'msd' class.
%   - 'linear': A boolean indicating whether the system is linear (true) or nonlinear (false).
%   - 'numMass': The number of masses in the system.
%   - 'm': A vector containing the masses in the system.
%   - 'k': A vector containing the spring constants for each mass.
%   - 'c': A vector containing the damping constants for each mass.
%
% Constructor Description:
% ------------------------
% The constructor initializes the mass-spring-damper system based on the
% provided parameters. It constructs the system matrices A, B, C, D, E, and P
% depending on whether the system is linear or nonlinear. If the system has
% nonlinearities, it will set up the appropriate nonlinear matrix E.
% The constructor also checks that the system is observable and validates the
% input constants. If multiple constants are provided, they are repeated for
% each mass.
%
% Initialization Steps:
% ---------------------
% - The constructor checks that only one constant for spring, mass, and damping
%   is provided, and replicates it for each mass if necessary.
% - If the system is linear, it sets up the A matrix for a linear system with
%   the appropriate entries for each mass and its interactions.
% - If the system is nonlinear, it sets up the A matrix similarly but includes
%   additional nonlinear terms and sets the nonlinear matrices E and P.
% - The system matrices B and C are set up based on the number of masses.
% - The constructor also checks that the system is observable by calling
%   the MATLAB function `isObsv()`.
%
% Example:
```

```
% --------
% linear = true;  % Set to false for a nonlinear system
% numMass = 2;
% m = [1; 1];  % Masses of each block
% k = [10; 10];  % Spring constants
% c = [0.5; 0.5];  % Damping constants
%
% sys = msd(linear, numMass, m, k, c);
% This creates a linear 2-mass mass-spring-damper system with the specified
% parameters.
%
% Notes:
% ------
% - If a nonlinear system is chosen, the matricex E is populated to
%   represent nonlinear contributions to the system.
% - The system matrices (A, B, C, D, E, and P) are computed based on the
%   configuration of the mass-spring-damper system (linear or nonlinear).
% - The constructor automatically checks for observability and throws an error
%   if the system is not observable.
% - The class is capable of handling multiple masses in series and adjusting
%   the system dynamics accordingly.
%
% See also:
% ---------
% isObsv


    properties
        % Number of masses in series
        numMass
        % Name of the system
        Name
        % State-space A matrix
        A
        % State-space B matrix
        B
        % State-space C matrix
        C
        % State-space D matrix
        D
        % Nonlinear E multiplication matrix
        E
        % Linear
        Linear
        % number of states
        nx
        % number of outputs
        ny
        % number of inputs
        nu
        % size of eventual x
        xsize

        k

        a

        m

        NLsize

        COutputs
    end

    methods
        function obj = msd(linear,numMass,m,k,c)
            %UNTITLED7 Construct an instance of this class
            %   Detailed explanation goes here
            obj.numMass = numMass;
            obj.Name = strcat(num2str(numMass), " mass-spring-damper");
            obj.Linear = linear;
            obj.k = k;
            obj.a = 10;
            obj.m = m;

            % Check if only one constant is provided and create multiples
            if size(k,1) > 1 || size(m,1) > 1 || size(c,1) > 1
                error('Provide only one constant, it will be repeated.')
            else
                m = repmat(m,numMass,1);
                k = repmat(k,numMass,1);
                c = repmat(c,numMass,1);
            end

            if numMass == 1
                if linear
                    A = [0, 1; -k(1)/m(1), -c(1)/m(1)];
                    E = [];
                elseif ~linear
                    A = [0, 1; -k(1)/m(1), -c(1)/m(1)];
                    E = [0; -1];
                end
                B = [0; 1/m(1)];
                % C should contain all rows that are valid outputs all rows should
                % individu
                C = [1 0];
                D = 0;
            else
                A = zeros(2*numMass);
                if linear
                    % Add state matrix entries for the first mass
                    A(1,2) = 1;
                    A(2,1:4) = [-(k(1)), -c(1), k(2), c(2)]./m(1);

                    % Add state matrix entries for the last mass
                    A(end-1,end) = 1;
                    A(end,end-3:end) = [0 0 -k(end) -c(end)]./m(end);

                    % Add state matrix entries for intermediate matrices
                    for i = 2:1:numMass-1
                        A(2*i-1,2*i) = 1;
                        slice = [0 0 -k(i), -c(i), k(i+1), c(i+1)]./m(i);
                        A(2*i,2*i-3:2*i+2) = slice;
                    end

                    % No non-linearities
                    E = [];



                elseif ~linear

                   % Add state matrix entries for the first mass
                    A(1,2) = 1;
                    A(2,1:4) = [-k(1), -(c(1)), k(2), c(2)]./m(1);

                    % Add state matrix entries for the last mass
                    A(end-1,end) = 1;
                    A(end,end-3:end) = [0 0 -k(end) -c(end)]./m(end);

                    % Add state matrix entries for intermediate matrices
                    for i = 2:1:numMass-1
                        A(2*i-1,2*i) = 1;
                        slice = [0 0 -k(i) -c(i) k(i+1) c(i+1)]./m(i);
                        A(2*i,2*i-3:2*i+2) = slice;
                    end

                    % non-linearities
                    E = zeros(2*numMass,numMass);
                    for i = 1:1:numMass
                        if i < numMass
                            E(2*i,i:i+1) = [-1/m(i) 1/m(i)];
                        else
                            E(2*i,i) = -1/m(i);
                        end
                    end

%                     error('Non linear model not possible for numMass: %2.0 > 1.',numMass)
                end

                % Set up B
                B = zeros(2*numMass,numMass);
                for i = 1:1:numMass
                    B(2*i,i) = 1/m(i);
                end


                % C should contain all rows that are valid outputs
                C = zeros(numMass,2*numMass);
                for i = 1:1:numMass
                    C(i,1:2*i) = [repmat([1 0],1,i)];
%                     C = [1 zeros(1, 2*numMass-1)];
                end

                D = 0;

            end

            % check for observability
            if ~isObsv(A,C)
                error('The system is not observable')
            end

            obj.A = A;
            obj.B = B;
            obj.C = C;
            obj.D = D;
            obj.E = E;
            obj.nx = size(A,1);
            obj.ny = size(C,1);
            obj.nu = size(B,2);
            obj.NLsize = size(E,2);

        end

    end
end
```

Now all functions follow alphabetically

Listing 10: ApLCSetup.m

```
function [ApLCi,LCi] = ApLCSetup(mo)
% AmLCSetup Function
%
% Constructs the 'ApLCi' and 'LCi' matrices for each observer in the
   system,
% incorporating the observer gains ('Li') and output matrices ('Ci').
%
% Documentation written by ChatGPT.
%
% Syntax:
% -------
% '[ApLCi, LCi] = AmLCSetup(mo)'
%
% Inputs:
% -------
% - 'mo': Multi-observer system object containing the following properties
   :
%   - 'nx': Number of states in the system.
%   - 'numObservers': Number of observers in the system.
%   - 'Ai': State-space 'A' matrices for each observer, organized in a 3D
```

```
array.
%   - 'Li': Observer gain matrices ('L'), organized in a 3D array.
%   - 'Ci': Output matrices ('C'), organized in a 3D array.
%
% Outputs:
% --------
% - 'ApLCi': A 3D array of modified 'A' matrices with observer gains included.
%   Each slice of 'ApLCi' represents the modified 'A' matrix for each observer.
%   Dimensions: '(nx x nx x numObservers)'.
% - 'LCi': A 3D array of matrices resulting from the multiplication of
%   observer gains ('Li') and output matrices ('Ci'). Each slice of 'LCi'
%   represents this product for each observer. Dimensions: '(nx x nx x numObservers)'.
%
% Implementation Steps:
% ---------------------
% 1. Initialize 'LCi' as a 3D array of zeros, with dimensions '(nx x nx x numObservers)'.
% 2. Initialize 'ApLCi' as a 3D array of zeros, with dimensions '(nx x nx x numObservers)'.
% 3. Loop over each observer:
%    - Compute 'LCl = Li(:,:,l) * Ci(:,:,l)' for each observer.
%    - Modify 'ApLCi(:,:,l)' by adding 'LCl' to the 'Ai(:,:,l)' matrix.
%    - Store the result of 'LCl' in 'LCi(:,:,l)'.
%
% Notes:
% ------
% - The function computes two key matrices: 'ApLCi' and 'LCi' for each observer.
% - 'ApLCi' is the modified 'A' matrix for each observer, incorporating the
%   contribution of the observer's gain and output matrices.
% - 'LCi' stores the contribution from the observer's gain and output matrices
%   alone (without the state-space dynamics from 'Ai').
%
% Matrix Structure:
% -----------------
% - 'ApLCi(:,:,l)' is the modified state-space matrix for the 'l'th observer
%   after incorporating the observer gain and output matrix.
% - 'LCi(:,:,l)' stores the product of the observer gain matrix 'Li' and the
%   output matrix 'Ci' for the 'l'th observer.
%
% See also:
% ---------
% ssmo, systemPSetup, pad3DL


    % Define empty LC
    LCi = zeros(mo.nx,mo.nx,mo.numObservers);

    % Create empty matrix to store A's on the diagonal
    ApLCi =  zeros(mo.nx,mo.nx,mo.numObservers);

    % Add LjCj from Aj and place on the diagonal
    for l = 1:1:mo.numObservers
        LCl = mo.Li(:,:,l)*mo.Ci(:,:,l);
        ApLCi(:,:,l) = mo.Ai(:,:,l) + LCl;
        LCi(:,:,l) = LCl;
    end

end
```

Listing 11: attackFunction.m

```
function attackVector = attackFunction(t,a)
% ATTACKFUNCTION
%
% The 'attackFunction' generates an attack vector based on a given time ('t') and
% an attack configuration array ('a'). The function allows different attack signals
% to be applied based on the value of elements in 'a'. When 'a' is applied
% in the 3D-CMO context the 3D attack configuration array is first
% flattened and then reshaped.
%
% This documentation was written by ChatGPT.
%
% Syntax:
% -------
% 'attackVector = attackFunction(t, a)'
%
% Inputs:
% -------
% - 't' (numeric):
%   The current time or scalar value used for generating the attack signals.
%
% - 'a' (array):
%   A matrix or array where each element represents the attack configuration for
%   the corresponding output:
%     - '1': Linear attack, returns the time 't'.
%     - '2': Sinusoidal attack, returns 'sin(t)'.
%     - Any other value: No attack, returns '0'.
%
% Outputs:
% --------
% - 'attackVector' (array):
%   An array of the same size as 'a', where each element corresponds to the attack signal
%   generated based on the configuration in 'a':
%     - If 'a(i) == 1', 'attackVector(i) = t'.
%     - If 'a(i) == 2', 'attackVector(i) = sin(t)'.
%     - Otherwise, 'attackVector(i) = 0'.
%
% Example Usage:
% --------------
% Given an attack configuration matrix 'a' and a time 't':
% t = 2;
% a = [1; 0; 2; 2; 1; 0];
% attackVector = attackFunction(t, a);
%
% The resulting 'attackVector' will be:
% attackVector =
%      [2.0000; 0; 0.9093; 0.9093; 2.0000; 0];
%
% Notes:
% ------
% - Users can customize the attack logic by modifying the 'if-elseif' block for
%   different attack functions.


    aFlat = a(:);
    attackFlat = zeros(size(aFlat));
    for i = 1:1:size(aFlat,1)
        if aFlat(i) == 1
            % Edit function below for changing the attack function
            attackFlat(i) = sin(t);
        elseif aFlat(i) == 2
            attackFlat(i) = sin(t);
        end
    end
    attackVector = reshape(attackFlat,size(a));
end
```

Listing 12: CNSetup.m

```
function COutputs = CNSetup(obj)
% CNSetup Function
%
% The 'CNSetup' function generates a matrix 'COutputs' containing the
% outputs of the Multi-Observer (MO) system. The rows of the system's 'C'
% matrix ('sys.C') are treated as possible output options and are either
% repeated or truncated to ensure that the resulting matrix has exactly
% 'numOutputs' rows.
%
% This documentation was written by ChatGPT.
%
% Syntax:
% -------
% 'COutputs = CNSetup(obj)'
%
% Inputs:
% -------
% - 'obj' (cmo3d object):
%   The 'cmo3d' object containing the system ('obj.sys') and the desired
```

```matlab
%   number of outputs ('obj.numOutputs').
%
% Outputs:
% --------
% - 'COutputs' (matrix):
%   A matrix with 'numOutputs' rows, constructed by repeating or truncating
%   the rows of 'sys.C'.
%
% Behavior:
% ---------
% The function constructs 'COutputs' based on the following rules:
% 1. If 'numOutputs' is less than or equal to the number of rows in 'sys.C',
%    only the first 'numOutputs' rows are used.
% 2. If 'numOutputs' exceeds the number of rows in 'sys.C', the rows of
%    'sys.C' are repeated until there are at least 'numOutputs' rows, and
%    any excess rows are truncated to match 'numOutputs'.
%
% Example Scenarios:
% ------------------
% - Case 1: 'sys.C' has enough rows to satisfy 'numOutputs':
%   sys.C = [1 0; 0 1];
%   obj.numOutputs = 2;
%   COutputs = CNSetup(obj);
%   % Result:
%   % COutputs =
%   %     [1 0;
%   %      0 1]
% - Case 2: 'sys.C' has fewer rows than 'numOutputs':
%   sys.C = [1 0; 0 1];
%   obj.numOutputs = 3;
%   COutputs = CNSetup(obj);
%   % Result:
%   % COutputs =
%   %     [1 0;
%   %      0 1;
%   %      1 0]
% - Case 3: 'sys.C' has more rows than 'numOutputs':
%   sys.C = [1 0; 0 1; -1 0];
%   obj.numOutputs = 2;
%   COutputs = CNSetup(obj);
%   % Result:
%   % COutputs =
%   %     [1 0;
%   %      0 1]
%
% Internal Logic:
% ---------------
% 1. Extracts 'sys.C' as the base matrix of possible output rows.
% 2. Computes the number of copies of 'sys.C' needed to meet or exceed
%    'numOutputs' rows.
% 3. Creates a repeated matrix by duplicating 'sys.C' as many times as needed.
% 4. Truncates the resulting matrix to exactly 'numOutputs' rows.
%
% Notes:
% ------
% - This function assumes that the 'sys.C' matrix is defined in the 'sys'
%   property of the 'obj' object.
% - The function dynamically adapts the output matrix to any 'numOutputs'
%   value, ensuring compatibility with varying system requirements.
%
% Dependencies:
% -------------
% - The 'obj' must be an instance of a class (e.g., 'mo') with the
%   properties 'sys.C' and 'numOutputs'.

    % Extract the number of possible ouputs
    COptions = obj.sys.C;
    numOptions = size(COptions,1);

    % If the number of actual outputs (numOuputs) is smaller then the
    % number of possible outputs: take the first numOuputs rows of Coptions.
    % If the number of actual outputs (numOutputs) is larger then the
    % number of possible outputs: duplicate Coptions until there are enough
    % it has a length longer then numOutputs and trim off the bottom rows
    % untill it matches numOutputs.

    copiesRequired = ceil(obj.numOutputs/numOptions);
    % Create empty matrix to store all duplicates of the options
    COutputOptions = repmat(COptions,copiesRequired,1);
    COutputs = COutputOptions(1:obj.numOutputs,:);

end
```

Listing 13: CsetSetup.m

```
function [Cset,CsetIndices,setAttack] = CsetSetup(CN,Attack,aggregate_grouping,obj)
% CsetSetup Function
%
% The 'CsetSetup' function creates a 3D array ('Cset') that specifies
% subsets of observer output matrices from the larger matrix 'CN' for each
% individual observer. It also generates 'CsetIndices', a matrix that
% records the indices of rows from 'CN' used in each subset, and
% 'setAttack', which maps attack signals to the selected outputs in each
% subset. The behavior varies depending on whether the system is linear or
% nonlinear.
%
% This documentation was written by ChatGPT.
%
% Syntax:
% -------
% '[Cset, CsetIndices, setAttack] = CsetSetup(CN, Attack, aggregate_grouping, obj)'
```

```
%
% Inputs:
% -------
% - 'CN' (matrix): The larger output matrix containing all possible
%   observer outputs.
% - 'Attack' (attack object): The attack object containing attack details,
%   particularly 'attackList'.
% - 'aggregate_grouping' (integer): The grouping parameter used in nonlinear systems
%   to form strict sensor groups.
% - 'obj' (mo object): The system object, including details such as
%   'sys.nx', 'sys.numMass', 'numOutputs', and 'numObservers'.
%
% Outputs:
% --------
% - 'Cset' (3D array): A 3D array where each slice ('Cset(:,:,i)') is a
%   subset of rows from 'CN' representing the outputs of an observer.
% - 'CsetIndices' (matrix): A 2D matrix where each row contains indices
%   that specify which rows of 'CN' were used to construct the subsets in
%   'Cset'.
% - 'setAttack' (3D array): A 3D array where each slice ('setAttack(:,:,i)')
%   contains the attack signals corresponding to the outputs in 'Cset(:,:,i)'.
%
% Behavior:
% ---------
% For Linear Systems ('obj.sys.Linear = 1'):
% 1. The function computes all combinations of outputs (rows) from 'CN'
%    that can form subsets of a specified size ('obj.numOutputsObservers').
% 2. 'CsetIndices' stores the row indices of 'CN' used in each subset.
% 3. 'Cset' and 'setAttack' are populated based on these combinations.
%
% For Nonlinear Systems ('obj.sys.Linear = 0'):
% 1. Sensor groups are created by aggregating related sensors (e.g., those
%    measuring the same mass position).
% 2. Larger groups are formed based on 'aggregate_grouping', ensuring all
%    outputs per observer are unique.
% 3. 'CsetIndices' is constructed to avoid duplicate rows within each subset.
% 4. Rows with duplicates are removed, and valid subsets are retained for
%    further processing.
% 5. 'Cset' and 'setAttack' are built from these valid subsets.
%
% Notes:
% ------
% - 'nchoosek' is used for linear systems to compute all combinations.
% - Nonlinear systems rely on 'aggregate_grouping' and strict sensor pairings.
% - The function assumes 'Attack.attackList' and 'CN' have compatible
%   dimensions.
%
% Dependencies:
```

```
% -------------
% - 'obj.sys.nx': The number of states in the system.
% - 'obj.sys.numMass': The number of masses in the system (for nonlinear systems).
% - 'obj.numOutputs': The total number of outputs available.
% - 'obj.numObservers': The total number of observers.
% - 'obj.numOutputsObservers': The number of outputs per observer.



    % Extract the number of states
    numStates = obj.sys.nx;

    % Define a list with all indices, so 1,2,...,N
    outputList = 1:1:obj.numOutputs;

    switch obj.sys.Linear
        case 1 % linear system
            % Select the indices of the combinations of Cj's
            CsetIndices = nchoosek(outputList,obj.numOutputsObservers);

        case 0 % nonlinear system

            % create strict pairs
            % check if N_O / nx has no remainder
            if rem(obj.numOutputs,obj.sys.nx/2) ~= 0
                error("Strict pairs are not possible numOutputs =%2.0f and " + ...
                    " nx/2 =%3.0f, which leaves a remainder of%2.0f.", ...
                    obj.numOutputs,obj.sys.nx/2,rem(obj.numOutputs,obj.sys.nx))
            end

            % create correct pairs !! very dependent on reshape behaviour
            % each row contains all sensors measuring the same mass
            % position
            sameSensors = reshape(outputList,obj.sys.numMass,[],1);
            sensorGroups = combinations(sameSensors,aggregate_grouping);
            groupSize = size(sensorGroups,2);
            numSensorGroups = size(sensorGroups,1);
            % make large groups out of the sensorGroups  obj.numOutputsObservers/numSensorCopies
            CgroupIndices = nchoosek(1:1:numSensorGroups,obj.numOutputsObservers);
            CsetSize = groupSize*size(CgroupIndices,2);
            CsetIndices = zeros(size(CgroupIndices,1),CsetSize);

            for g = 1:1:size(CgroupIndices,1)
                Cgroup = CgroupIndices(g,:);
                for i = 1:1:size(Cgroup,2)
                    CsetIndices(g,(i-1)*groupSize+1:i*groupSize) = sensorGroups(Cgroup(i),:);
                end

                % check for duplicates
                if size(unique(CsetIndices(g,:)),2) < CsetSize
                    % not unique
%                     CsetIndices(g,:) = zeros(1,CsetSize);
                end


            end

            % select all rows that have nonzero elements
            rowsToKeep = any(CsetIndices,2);
            CsetIndices = CsetIndices(rowsToKeep,:);

    end

    % Loop over the combinations and add them to the empty CJ
    obj.numObservers = size(CsetIndices,1);
    Cset = zeros(size(CsetIndices,2),numStates,obj.numObservers);
    setAttack = zeros(size(CsetIndices,2),1,obj.numObservers);
    for j = 1:1:size(CsetIndices,1)
        % In every j of CJ
        Cselection = CsetIndices(j,:);
        Cset(:,:,j) = CN(Cselection,:);
        setAttack(:,:,j) = Attack.attackList(Cselection,:);
    end

end
```

Listing 14: defineObservers.m

```
function [Ai,Li] = defineObservers(A,CJ,eigenvalues,obj)
% defineObservers Function
%
% The 'defineObservers' function configures the observer matrices 'Ai' and
% 'Li' for a set of systems defined by matrix 'A' and a set of output
% matrices 'CJ'. It ensures that the systems are observable and stable by
% placing eigenvalues for each observer's dynamics.
%
% This documentation was written by ChatGPT.
%
% Syntax:
% -------
% '[Ai, Li] = defineObservers(A, CJ, eigenvalues, obj)'
%
% Inputs:
% -------
% - 'A' (matrix): The system state matrix, shared among all observers.
% - 'CJ' (3D array): A collection of output matrices for the observers,
%   where 'CJ(:,:,i)' corresponds to the 'i'th observer.
% - 'eigenvalues' (vector): Desired eigenvalues for the closed-loop
%   dynamics of each observer.
% - 'obj' (object): Contains system properties, including:
%     - 'obj.nx': Number of states in the system.
%     - 'obj.numOutputsObservers': Number of outputs per observer.
%     - 'obj.numObservers': Total number of observers.
%
% Outputs:
% --------
% - 'Ai' (3D array): A collection of state matrices for the observers,
%   where 'Ai(:,:,i)' corresponds to the 'i'th observer.
% - 'Li' (3D array): A collection of observer gain matrices, where
%   'Li(:,:,i)' corresponds to the 'i'th observer.
%
% Behavior:
% ---------
% 1. Initializes 'Ai' and 'Li' as empty 3D arrays to hold matrices for all
%    observers.
% 2. For each observer:
%    - Checks if the pair '(A, Cj)' is observable, where 'Cj = CJ(:,:,i)'.
%    - Uses the 'place' function to calculate the observer gain matrix 'L'
%      such that the closed-loop eigenvalues of 'A + L*Cj' match the
%      specified 'eigenvalues'.
%    - Ensures that 'A + L*Cj' is stable. If not, an error is raised.
%
% Example Scenarios:
% ------------------
% Inputs:
% - 'A = [0 1; -10 -1]'
% - 'CJ(:,:,1) = [1 0; 0 1; 1 0];
%    CJ(:,:,2) = [1 0; 0 1; 0 1];
%    CJ(:,:,3) = [1 0; 1 0; 0 1];
%    CJ(:,:,4) = [0 1; 1 0; 0 1];'
% - 'eigenvalues = [-2, -3]'
% - 'obj.numObservers = 4; obj.nx = 2; obj.numOutputsObservers = 2;'
%
% Outputs:
% - 'Ai' will be a stack of the input matrix 'A' for each observer.
% - 'Li' will contain the gain matrices computed for each '(A, Cj)' pair.
%
% Example Stability Check:
% ------------------------
% For each observer, the function checks the eigenvalues of 'A + L*Cj' to
% confirm stability. If unstable, an error with diagnostic information is
% raised.
%
% Dependencies:
% -------------
% - isObsv.m: Verifies if the pair '(A, Cj)' is observable.
% - place.m: Calculates the observer gain matrix 'L' to place eigenvalues.
% - isMatrixStable.m: Confirms the stability of 'A + L*Cj'.
%
% Notes:
% ------
% - All output matrices 'Cj' must result in an observable pair '(A, Cj)'.
%   If not, the function raises an error.
% - The eigenvalues in 'eigenvalues' must be carefully chosen to ensure
%   stability of the closed-loop dynamics.
% - Stability is checked iteratively for each observer, providing detailed
```

```matlab
%   diagnostics if a problem occurs.
%
% See also:
% ---------
% isObsv, place, isMatrixStable

    Ai = zeros(size(A,1),size(A,2),obj.numObservers);
    Li = zeros(size(place(A',CJ(:,:,1)',eigenvalues),2),size(place(A',CJ(:,:,1)',eigenvalues),1),obj.numObservers);

    % Loop through all rows of CJ and place the eigenvalues at
    % 'eigenvalues'.
    for l = 1:1:obj.numObservers
        % Select the observer for which to calculate the Aj + LjCj and Bi
        Cj = CJ(:,:,l);
        if ~isObsv(A,Cj)
            disp('A =')
            disp(A);
            disp('Cj =')
            disp(Cj)
            error('A pair (A,Cj) is not observable')
        end

        Ai(:,:,l) = A;
        L = -place(A',Cj',eigenvalues)';
        Li(:,:,l) = L;
        % Stability check
        if ~isMatrixStable(A+L*Cj)
            disp("The desired eigenvalues of A+LC")
            disp(eigenvalues)
            disp('The actual eigenvalues of A+LC:')
            disp(eig(A+L*Cj))
            error('The chosen Lj does not make Aj + LjCj stable')
        end

    end

end
```

Listing 15: etaSetup.m

```matlab
function eta = etaSetup(setA,CJIndices,CPIndices,attackValue,CMOstruct)
    % eta = etaSetup(setA,CJIndices,CPIndices,attackValue,noiseFactor,noisePower,CMOdict)
    % sets up the state-space system input eta, where eta = [u;v+tau;w]. u
    % is system input, v is sensor noise, tau is the attack signal and w is
    % the disturbance.

    % Generate u
    u = zeros(CMOstruct.numOriginalInputs,1);

    % Generate N (m x tsize) sensor noise signals
    numJOuptuts = CMOstruct.numJObservers*CMOstruct.numOutputsJObservers;
    numPOutputs = CMOstruct.numPObservers*CMOstruct.numOutputsPObservers;
    v = zeros(numJOuptuts+numPOutputs,1);

    % Generate N (n x tsize) attack signals
    % Select which outputs will be attacked, form 2 subsets of numOutputs
    % sized n. the first subset will be denoted by a value of 1 and the
    % second by a value of 2.
    tau = zeros(numJOuptuts+numPOutputs,1);
    % Loop over J observers
    for j = 1:1:CMOstruct.numJObservers
        % Loop over outputs of each J observer
        row = CJIndices(j,:);
        for k = 1:1:CMOstruct.numOutputsJObservers
            outputID = row(k);
            if isMemberOf(setA,outputID)
                tau((j-1)*CMOstruct.numOutputsJObservers+k,:) =
                    attackValue;
            end

        end

    end

    for p = 1:1:CMOstruct.numPObservers
        % Loop over outputs of each P observer
        row = CPIndices(p,:);
        for k = 1:1:CMOstruct.numOutputsPObservers
            outputID = row(k);
            if isMemberOf(setA,outputID)
                tau(numJOuptuts + (p-1)*CMOstruct.numOutputsPObservers+k
                    ,:) = attackValue;
            end

        end

    end

    % Generate an (n x tsize) process noise signal
    w = zeros(CMOstruct.numOriginalStates,1);

    eta = [u;v+tau;w];
end
```

Listing 16: findIndices.m

```
function [numOfPsubsetsInJ, PsubsetOfJIndices] = findIndices(Jmo,Pmo)
% findIndices Function
%
% The 'findIndices' function determines which observers from a smaller set
% ('Pmo') are subsets of observers from a larger set ('Jmo'). The function
% returns the count of such subsets and their corresponding indices.
%
% Documentation written by ChatGPT.
%
```

```
% Syntax:
% -------
% '[numOfPsubsetsInJ, PsubsetOfJIndices] = findIndices(Jmo, Pmo, sys)'
%
% Inputs:
% -------
% - 'Jmo': Structure representing the larger observer set, containing:
%    - 'numOutputsObservers': Number of outputs per observer in 'Jmo'.
%    - 'numObservers': Number of observers in 'Jmo'.
%    - 'CiIndices': Matrix of observer indices for 'Jmo' (each row
%      corresponds to an observer's indices).
% - 'Pmo': Structure representing the smaller observer set, containing:
%    - 'numOutputsObservers': Number of outputs per observer in 'Pmo'.
%    - 'numObservers': Number of observers in 'Pmo'.
%    - 'CiIndices': Matrix of observer indices for 'Pmo' (each row
%      corresponds to an observer's indices).
% - 'sys': System object (optional; not directly used in this implementation).
%
% Outputs:
% --------
% - 'numOfPsubsetsInJ': Number of subsets in 'Pmo' that are contained within
%   each observer in 'Jmo' (scalar).
% - 'PsubsetOfJIndices': Matrix of size '(Jmo.numObservers x numOfPsubsetsInJ)',
%   where each row lists the indices of 'Pmo' observers that are subsets
%   of the corresponding 'Jmo' observer.
%
% Description:
% ------------
% For each observer in 'Jmo', the function identifies the observers in
% 'Pmo' whose indices are subsets of the 'Jmo' observer's indices.
%
% Example:
% --------
% Inputs:
% - 'Jmo.CiIndices = [1 2 3; 1 2 4; 1 3 4; 2 3 4]'
% - 'Pmo.CiIndices = [1 2; 1 3; 1 4; 2 3; 2 4; 3 4]'
% - 'Jmo.numOutputsObservers = 3', 'Pmo.numOutputsObservers = 2'
% - 'Jmo.numObservers = 4', 'Pmo.numObservers = 6'
%
% Outputs:
% - 'numOfPsubsetsInJ = 3'
% - 'PsubsetOfJIndices = [1 2 4;
%                           1 3 5;
%                           2 3 6;
%                           4 5 6]'
%
% Algorithm:
% ----------
% 1. Compute the total number of subsets ('numOfPsubsetsInJ') using the
%    'nchoosek' function.
% 2. Iterate over all observers in 'Jmo':
```

```
%    - For each observer, check if the indices of each observer in 'Pmo'
%      are a subset.
%    - If a subset is found, append its index to the current row.
% 3. Store the indices of subsets in 'PsubsetOfJIndices'.
%
% Notes:
% ------
% - The isSubsetOf.m function must be defined to check whether one set
%   of indices is a subset of another.
% - The function assumes that the observer indices in 'CiIndices' are
%   sorted and unique.
%
% Dependencies:
% -------------
% - isSubsetOf.m: Utility function to check subset relations.
%
% See also:
% ---------
% - nchoosek.m: MATLAB function for combinations.
% - isSubsetOf.m: Custom subset-checking function.

    PsubsetOfJIndices = zeros(Jmo.numObservers,1);

    for j = 1:1:Jmo.numObservers
        CjIndices = Jmo.CiIndices(j,:);
        % create new emtpy row to fill and append to the bottom of
        % PsubsetOfJIndices
        newRow = [];
        for p = 1:1:Pmo.numObservers
            CpIndices = Pmo.CiIndices(p,:);
            isPSubset = isSubsetOf(CjIndices,CpIndices);
            % If the indices of p are a subset of those of j: find the
            if isPSubset
                newRow(1,end+1) = p;
            end
        end
        % append with zeros if necessary
        if size(PsubsetOfJIndices,2) < size(newRow,2)
            PsubsetOfJIndices = [PsubsetOfJIndices zeros(size(PsubsetOfJIndices,1),size(newRow,2)-size(PsubsetOfJIndices,2))];
        elseif size(PsubsetOfJIndices,2) > size(newRow,2)
            newRow = [newRow zeros(1,size(PsubsetOfJIndices,2)-size(newRow,2))];
        end
        PsubsetOfJIndices(j,:) = newRow;
    end
    numOfPsubsetsInJ = size(PsubsetOfJIndices,2);

end
```

Listing 17: flatten.m

```
function flatM = flatten(M)
```

```
% flatten Function
%
% The 'flatten' function takes a 3D matrix and flattens it into a 2D matrix
% by stacking the slices of the 3rd dimension vertically. Each slice along
% the 3rd dimension of the input matrix becomes a consecutive block of rows
% in the output matrix.
%
% Documentation written by ChatGPT.
%
% Syntax:
% -------
% flatM = flatten(M)
%
% Inputs:
% -------
% - 'M': A 3D matrix of size '(h x w x l)' to be flattened, where:
%    - 'h': Number of rows in each slice (1st dimension).
%    - 'w': Number of columns in each slice (2nd dimension).
%    - 'l': Number of slices along the 3rd dimension.
%
% Outputs:
% --------
% - 'flatM': A 2D matrix of size '(h*l x w)', where each block of 'h' rows
%   corresponds to a slice of 'M' along the 3rd dimension.
%
% Description:
% ------------
% The function rearranges the elements of the input 3D matrix 'M' into a
% single 2D matrix 'flatM' by stacking the 'l' slices along the 3rd
% dimension of 'M' into vertical blocks. The output matrix has a number of
% rows equal to the total rows in all slices ('h*l') and the same number of
% columns as the input ('w').
%
% Example:
% --------
% Input:
% M = cat(3, [1 2; 3 4], [5 6; 7 8], [9 10; 11 12]);
% Output:
% flatM = [ 1  2;
%           3  4;
%           5  6;
%           7  8;
%           9 10;
%          11 12];
%
% Algorithm:
% ----------
% 1. Determine the number of slices ('l') in the 3rd dimension.
% 2. Compute the number of rows per slice ('h').
% 3. Initialize an empty 2D matrix 'flatM' with dimensions '(h*l x w)'.
% 4. Loop through each slice in the 3rd dimension and assign its contents
%    to the corresponding block of rows in 'flatM'.
%
% Notes:
% ------
% - The function assumes that the input matrix 'M' is non-empty and has
%   consistent dimensions.
% - This operation is useful for data reshaping or preparing 3D data for
%   algorithms that require 2D inputs.
%
% See also:
% ---------
% - reshape: MATLAB function for reshaping arrays.
% - cat.m : MATLAB function for concatenating arrays.


    l = size(M,3);
    h = size(M,1);
    flatM = zeros(size(M,1)*l,size(M,2));

    for i = 1:1:size(M,3)
        flatM(h*(i-1)+1:h*i,:) = M(:,:,i);
    end
end
```

Listing 18: generateCombination.m

```
function combinations = generateCombination(Id, n, k)
    % cgen returns the i-th combination of k numbers chosen from 1,2,...,n
    % Input:
    %   Id - the index of the combination (1-based indexing)
    %   n - total number of items
    %   k - number of items to choose
    % Output:
    %   c - the i-th combination as a row vector

    combinations = zeros(1,k); % Initialize the combination array
    remaining = Id;  % Initialize the remaining index
    init = 0;  % Initialize starting point for the combination

    for s = 1:k
        cs = init + 1;
        while remaining - nchoosek(n - cs, k - s) > 0
            remaining = remaining - nchoosek(n - cs, k - s);
            cs = cs + 1;
        end
        combinations(s) = cs; % Append the current selection
        init = cs;      % Update the starting point
    end
end
```

Listing 19: inputDialog.m

```
function inputs = inputDialog(dialog)
% inputDialog Function
%
```

```
% The 'inputDialog' function collects user inputs needed for configuring
% a multi-observer system. It displays a dialog box prompting for key
% parameters or uses predefined default inputs if the dialog is disabled.
%
% Documentation written by ChatGPT.
%
% Syntax:
% -------
% inputs = inputDialog(dialog)
%
% Inputs:
% -------
% - 'dialog': A logical value:
%     - 'true': Displays an input dialog box for the user to input values.
%     - 'false': Returns predefined default inputs without showing the dialog box.
%
% Outputs:
% --------
% - 'inputs': A cell array containing user-provided or default input values.
%
%
% Example:
% --------
% Using the dialog:
% inputs = inputDialog(true);
% Prompts the user for inputs using an input dialog box.
%
% Using default inputs:
% inputs = inputDialog(false);
% Returns:
% inputs = {'2', '5', '3', '2 5', '3', '1', '0 5', ...
%           '0.3 -0.1 0.5 0.2 -0.4 0.6 0.3 0.3 -0.7 0.4 -0.2 0.6', ...
%           '0 1 0', '1', 'loose', '0'};

    % Input dialog box
    inputPrompt = {'System selection (number indicates amount of mass-spring-dampers in series)',...
        'Number of system outputs',...
        'M, number of attacked outputs (max: largest integer M so that 2M<N holds)',...
        'Attacked outputs (empty string will attack random outputs)',...
        'Size of each J-observer (max: N-M)',...
        'Size of each P-observer (max: N-2M)',...
        'Timespan (enter tmin and tmax separtated by spaces)',...
        'x0 (enter x0 seperated by spaces)',...
        'CMO2D CMO3D SSMO',...
        'Linear (1) or Nonlinear (0) model',...
        'loose or strict aggregate sensors',...
        'Noise variance (generated with randn)'};

    % Prefilled inputs
    definputs = {'2',...
```

```
        '5',...
        '3',...
        '2 5',...
        '3',...
        '1',...
        '0 5',...
        '0.3 -0.1 0.5 0.2 -0.4 0.6 0.3 0.3 -0.7 0.4 -0.2 0.6',...
        '0 1 0',...
        '1',...
        'loose',...
        '0'};

    % Check if input is required, ohterwise use prefilled inputs
    if dialog
        inputs = inputdlg(inputPrompt,'CMO inputs',[1 40],definputs);
    else
        inputs = definputs;
    end
end
```

Listing 20: isMatrixStable.m

```
function stableBool = isMatrixStable(A)
% isMatrixStable Function
%
% The 'isMatrixStable' function checks whether the given matrix 'A' is
% stable, which means all eigenvalues of 'A' should have strictly negative
% real parts.
%
% Documentation written by ChatGPT.
%
% Syntax:
% -------
% stableBool = isMatrixStable(A)
%
% Inputs:
% -------
% - 'A': A square matrix for which stability is to be checked.
%     - 'A' should be a square matrix (i.e., number of rows equals the
%       number of columns).
%
% Outputs:
% --------
% - 'stableBool': A logical value indicating if the matrix 'A' is stable:
%     - 'true': All eigenvalues of 'A' have a strictly negative real part
%       (the matrix is stable).
%     - 'false': At least one eigenvalue of 'A' has a non-negative real
%       part (the matrix is unstable).
%
% Description:
% ------------
% This function calculates the eigenvalues of the matrix 'A' and checks whether
% all of them have strictly negative real parts. If any eigenvalue has a
```

```
    real
% part greater than or equal to 0, the matrix is considered unstable, and
    the
% function returns 'false'. Otherwise, it returns 'true', indicating that
    the
% matrix is stable.
%
% Example:
% --------
% Check matrix stability:
%
% A = [0 1; -2 -3];  % Example system matrix
% stableBool = isMatrixStable(A);
%
% If 'A' is stable, 'stableBool' will be 'true', otherwise 'false'.
%
% Notes:
% ------
% - The function assumes the input matrix 'A' is square.
%
% See also:
% ---------
% eig, real

    % Initialize stableBool as true
    stableBool = true;
    eigenvalues = eig(A);
    n = size(eigenvalues,1);
    % Loop through eigenvalues and compare each eigenvalue to 0
    for i = 1:1:n
        if eigenvalues(i) > 0
            stableBool = false;
        end
    end
end
```

Listing 21: isMemberOf.m

```
function bool = isMemberOf(superset,member)
% ISMEMBEROF Function
%
% The 'isMemberOf' function checks whether a specified 'member' is present
% in the given 'superset'.
%
% Documentation written by ChatGPT.
%
% Syntax:
% -------
% bool = isMemberOf(superset, member)
%
% Inputs:
% -------
% - 'superset': A vector containing the set of elements to check against.
%     - Can be a row or column vector.
% - 'member': The element to check if it exists in the 'superset'.
%     - A single scalar value (numeric, string, etc.).
%
% Outputs:
% --------
% - 'bool': A logical value indicating whether the 'member' is in the 'superset':
%     - 'true': if the 'member' is found in the 'superset'.
%     - 'false': if the 'member' is not found in the 'superset'.
%
% Description:
% ------------
% This function iterates through the elements of the 'superset' and compares
% each element to the 'member'. If the 'member' is found in the 'superset',
% the function returns 'true'. Otherwise, it returns 'false'. The function
% uses a simple loop to check each element of the 'superset' for equality
% with the 'member'.
%
% Example:
% --------
% Check if a number is in a set:
% superset = [1 2 3];
% member = 1;
% bool = isMemberOf(superset, member);
% bool = true
%
% Check if a number is not in a set:
% superset = [2 3 4];
% member = 1;
% bool = isMemberOf(superset, member);
% bool = false
%
% Notes:
% ------
% - This function compares the 'member' to each element in the 'superset'
%   using equality ('=='). It may not work for non-scalar or complex types.
% - It assumes 'superset' is a vector (either row or column).
%
% See also:
% ---------
% ismember

    sizeSuperset = size(superset,2);
    % Initialize bool as false
    bool = false;
    for l = 1:1:sizeSuperset
        if member == superset(l)
            bool = true;
            break
        end
    end
```

```
end
```

Listing 22: isObsv.m

```
function bool = isObsv(A,C)
% isObsv Function
%
% The 'isObsv' function determines whether the pair of matrices A and C is
% observable. It checks if the rank of the observability matrix matches the
% size of the system's state matrix.
%
% Documentation written by ChatGPT.
%
% Syntax:
% -------
% bool = isObsv(A, C)
%
% Inputs:
% -------
% - 'A': The state transition matrix of the system, which must be a square
%   matrix of size (n x n).
% - 'C': The output matrix of the system, which must have dimensions (m x n),
%   where m is the number of outputs and n is the number of states.
%
% Outputs:
% --------
% - 'bool': A logical value indicating whether the system is observable:
%     - 'true': If the pair (A, C) is observable.
%     - 'false': If the pair (A, C) is not observable.
%
% Description:
% ------------
% This function performs the Observability test (PBH test) on the system
% defined by matrices 'A' (state transition matrix) and 'C' (output matrix).
% It checks the rank of the matrix formed by combining 'C' and each
% eigenvalue of 'A'. If the rank of the matrix does not match the size of
% the system's state matrix 'A', then the system is considered unobservable.
%
% Example:
% --------
% Observable system:
% A = [0 1; 0 0];
% C = [1 0];
% bool = isObsv(A, C);
% % bool will be true (system is observable).
%
% Unobservable system:
% A = [0 1; 0 0];
% C = [0 1];
% bool = isObsv(A, C);
% % bool will be false (system is unobservable).
%
% Notes:
% ------
% - This function uses the PBH (Popov-Belevitch-Hautus) test for observability,
%   which checks if the rank of a specific matrix formed by the system matrices is full.
% - 'A' must be a square matrix, and 'C' should have the same number of columns as 'A'.
% - Eigenvalues of 'A' are used to form the matrices in the test.
%
% See also:
% ---------
% eig, rank

    n = size(A,1);
    ev = eig(A);
    I = eye(n);
    % PBH test
    bool = true;
    for i = 1:1:n
        if rank([ev(i)*I - A; C]) ~= n
            bool = false;
            break
        end
    end

end
```

Listing 23: isSubsetOf.m

```
function bool = isSubsetOf(superset,subset)
% isSubsetOf Function
%
% The 'isSubsetOf' function checks if one set (subset) is a subset of
% another set (superset). It verifies if all elements of the subset exist
% in the superset. The sets can be represented as either vertical or
% horizontal arrays.
%
% Documentation written by ChatGPT.
%
% Syntax:
% -------
% bool = isSubsetOf(superset, subset)
%
% Inputs:
% -------
% - 'superset': A 1-dimensional array (either row or column vector) that
%   represents the superset.
% - 'subset': A 1-dimensional array (either row or column vector) that
%   represents the subset.
%
% Outputs:
```

```
% --------
% - 'bool': A logical value indicating whether the subset is contained
%   within the superset:
%     - 'true': If the subset is a valid subset of the superset.
%     - 'false': If the subset is not a valid subset of the superset.
%     - An error is raised if the superset or subset are not 1-dimensional
%       or if the subset is larger than the superset.
%
% Description:
% ------------
% This function first ensures that both the superset and the subset are
% 1-dimensional arrays. It then checks if the subset is smaller than or
% equal to the superset in size. If the subset is smaller or equal, it
% proceeds to verify that every element of the subset is contained in the
% superset using the 'isMemberOf' function. If any element of the subset is
% not found in the superset, the function returns 'false'. If the subset is
% larger than the superset, an error is raised.
%
% Example:
% --------
% Valid subset:
% superset = [1 2 3 4];
% subset = [1 2];
% bool = isSubsetOf(superset, subset);
% % bool will be true (subset is a valid subset of superset).
%
% Invalid subset:
% superset = [1 2 3 4];
% subset = [1 5];
% bool = isSubsetOf(superset, subset);
% % bool will be false (subset contains an element not in superset).
%
% Error due to non-1D array:
% superset = [1 2; 3 4];  % 2D array (error case)
% subset = [1 3];
% bool = isSubsetOf(superset, subset);
% % Error: superset is not 1-dimensional.
%
% Notes:
% ------
% - The function will raise an error if either the superset or the subset
%   is not a 1-dimensional array.
% - The subset must not be larger than the superset.
% - The 'isMemberOf' function is used to check if each element of the
%   subset exists in the superset.
%
% See also:
% ---------
% isMemberOf


    % Transform superset into horizontal array.
    if size(superset,1) > 1 && size(superset,2) == 1
        superset = superset';
    elseif size(superset,1) > 1 && size(superset) > 1
        error('superset is not 1-dimensional.')
    end
    % Transform subset into horizontal array.
    if size(subset,1) > 1 && size(subset,2) == 1
        subset = subset';
    elseif size(subset,1) > 1 && size(subset) > 1
        error('subset is not 1-dimensional.')
    end

    % Check if subset is smaller or equal in size to superset
    sizeSubset = size(subset,2);
    sizeSuperset = size(superset,2);
    if sizeSubset > sizeSuperset
        error('The subset is larger then the superset.')
    end

    % Initialize bool as true
    bool = true;

    for l = 1:1:sizeSubset
        if ~isMemberOf(superset,subset(l))
            bool = false;
            break
        end
    end

end
```

Listing 24: MOplot.m

```
function MOplot(t,x,err,estimate,sys,MO,Jmo,Pmo)
% MOplot Function
%
% The 'MOplot' function generates a series of plots to visualize the results
% of a Composite Multi-Observer (CMO) system. It creates a separate plot for
% each system state, displaying the system response, J-observers' estimates,
% P-observers' estimates, the final CMO estimate, and the error for each state
% over time. The plots are arranged in a grid depending on the number of
% states in the system.
%
% Documentation written by ChatGPT.
%
% Syntax:
% -------
% MOplot(t, x, err, estimate, sys, MO, Jmo, Pmo)
%
% Inputs:
```

```
% -------
% - 't' : A vector containing time values (e.g., 0:0.01:5).
% - 'x' : A matrix containing the system states, observer estimates, and CMO
%   estimates across the time vector.
% - 'err' : A matrix containing the error for each system state over time.
% - 'estimate' : A matrix containing the CMO's final estimates for each system
%   state over time.
% - 'sys' : A structure representing the system being observed, with fields
%   such as 'nx' (number of states) and 'Name' (system name).
% - 'MO' : The multi-observer object that contains information about the system's
%   outputs and attack settings.
% - 'Jmo' : The J-observer configuration, containing the number of outputs
%   observed by each J-observer.
% - 'Pmo' : The P-observer configuration, containing the number of outputs
%   observed by each P-observer.
%
% Description:
% ------------
% The 'MOplot' function plots the system response, J-observers' estimates,
% P-observers' estimates, the final CMO estimate, and the error for each
% system state. It generates a grid of plots, where each subplot corresponds
% to a specific state of the system. The function includes various customization
% options like labels, legends, and plot titles for each state, as well as
% different line styles for each plot element (e.g., system response, estimates,
% and errors).
%
% Initialization Steps:
% ---------------------
% - The function first calculates the number of rows and columns for the plot
%   grid based on the number of system states (sys.nx).
% - It separates the system's true response, J-observers' estimates, P-observers'
%   estimates, and the CMO's estimate from the input 'x' matrix.
% - For each state, the system response, J-observer estimates, P-observer
%   estimates, CMO estimate, and error are plotted in individual subplots.
% - A legend is added for each plot to distinguish between different curves
%   (system response, J-observers, P-observers, CMO estimate, and error).
%
% Notes:
% ------
% - The function dynamically generates subplots for each system state.
% - The legend is adjusted to show the J-observers' and P-observers' estimates,
%   the system response, CMO estimate, and error for each state.
% - The number of rows and columns in the plot grid is calculated based on
%   the total number of system states (sys.nx).
% - The error and CMO estimate are only plotted if the 'estimate' matrix
%   contains multiple columns.
%
% See also:
% ---------
% mo, 3dcmo, ssmo

    fig = tiledlayout('flow');
    sgtitle({[char(sys.Name),' observed by a ' char(MO.Name) ' with ', num2str(MO.numOutputs),' outputs.'],...
        [ 'Number of attacks = ',num2str(MO.Attack.numAttacks)]});
        %,',N_J=',num2str(Jmo.numOutputsObservers),' and N_P=',num2str(Pmo.numOutputsObservers)]});

    % cmoEstimate =
    trueResponse = x(1:sys.nx,:);
    JEstimates = x(sys.nx+1:sys.nx+Jmo.numObservers*sys.nx,:);
    PEstimates = x(sys.nx+Jmo.numObservers*sys.nx+1:end,:);
    yl = 1.05 * max(max(abs(trueResponse)));

    % create tiled plot
    for l = 1:1:sys.nx
        nexttile
        leg = [];
        % select subplot to edit

        % plot all p and j estimators
        try
            for k=1:1:min(5,Jmo.numObservers)
                leg(2*k-1) = plot(t,JEstimates((k-1)*sys.nx+l,:),LineStyle="--",Color=[1 0 0 0.5]);
                hold on;
                if k < Pmo.numObservers
                    leg(2*k) = plot(t,PEstimates((k-1)*sys.nx+l,:),LineStyle="--",Color=[0 0 1 0.5]);
                    hold on;
                end

            end
        catch ME
            if strcmp(ME.identifier,"Unrecognized function or variable")
                warning("No Jmo supplied, will plot without Jmo.")
            end
        end

        % plot system response
        leg(end + 1) = plot(t,trueResponse(l,:),LineWidth=2,Color='black');
        hold on;

        % plot the cmo final estimate
        try
            if size(estimate,2) > 1
```

```
                leg(end + 1) = plot(t,estimate(l,:),LineWidth=1,Color='cyan');
                hold on;
            end
        catch ME
            if strcmp(ME.identifier,'Unrecognized function or variable')
                warning("No final esitmate is supplied, will plot without it.")
            end
        end

        % plot error
        try
            if size(estimate,2) > 1
                leg(end + 1) = plot(t,err(l,:),LineWidth=1,Color="#EDB120");
                hold on;
            end
        catch ME
            if strcmp(ME.identifier,"Unrecognized function or variable")
                warning("No error is supplied, will plot without it.")
            end
        end

        grid on;

        ylim([-yl yl])

        xlabel('Time')
        mass = floor((l+1)/2);

        if (-1)^l == -1
            % odd
            ylabel('Position')
            title(['Position of mass ' num2str(mass)])
        else
            % even
            ylabel('Velocity')
            title(['Velocity of mass ' num2str(mass)])
        end

    end

    % select the correct plots to add to legend
    if MO.Attack.numAttacks == 0 || size(leg,2) == 3
        lgd = legend([leg(end-2:end)],'System response','MO estimate','Error');
    elseif MO.Attack.numAttacks > 0
        lgd = legend([leg(1:2) leg(end-2:end)],'J-observers','P-observers','System response','MO estimate','Error');
    end

    lgd.Layout.Tile = 'east';

    set(gcf, 'Position', 0.4*get(0, 'Screensize'));
    hold on;
end
```

Listing 25: multiObserverODE

```
function dx = multiObserverODE(wb,tmax,sys,t,x,Attack,CMO2D,CMO3D,SSMO,whichMO,Noise,Jmo,Pmo,xIds)
% multiObserverODE Function
%
% The 'multiObserverODE' function computes the time derivative of the system
% state ('dx') for a dynamic system with potential attacks and multiple types
% of observers, including 2D-CMO, 3D-CMO, and SSMO. Active observers are
% specified using the 'whichMO' array.
%
% This documentation was written by ChatGPT.
%
% Syntax:
% -------
% 'dx = multiObserverODE(sys, t, x, Attack, CMO2D, CMO3D, SSMO, whichMO)'
%
% Inputs:
% -------
% - 'sys': System model structure
% - 't': Current time (scalar).
% - 'x': State vector, containing:
%     - 'xsys': System states.
%     - States for 'CMO2D', 'CMO3D', and 'SSMO' observers (if active).
% - 'Attack': Attack model structure containing
% - 'CMO2D': Configuration for the 2D observer (optional).
% - 'CMO3D': Configuration for the 3D observer, containing
% - 'SSMO': Configuration for the SSMO observer (optional).
% - 'whichMO': Boolean array specifying active observers:
%     - 'whichMO(1)': Use 'CMO2D' (1 = active, 0 = inactive).
%     - 'whichMO(2)': Use 'CMO3D' (1 = active, 0 = inactive).
%     - 'whichMO(3)': Use 'SSMO' (1 = active, 0 = inactive).
%
% Outputs:
% --------
% - 'dx': Time derivative of all states, including system and observer
%   states (vector).
%
% Observer Dynamics:
% ------------------
% 1. System State ('dxsys'):
%     - If nonlinear components are present ('NLsize > 0'), 'dxsys' includes
%       the dynamics of nonlinear springs.
%     - Otherwise, 'dxsys' depends only on the system matrix 'A'.
% 2. 2D-CMO Dynamics ('dxcmo2d'):
%     - Currently inactive (placeholder).
% 3. 3D-CMO Dynamics ('dxcmo3d'):
```

```
%    - Includes attack signals, nonlinear contributions, and coupling terms
%      defined in 'CMO3D'.
% 4. SSMO Dynamics ('dxssmo'):
%    - Includes nonlinear dynamics and attack signals, with coupling through
%      'SSMO.B'.
%
% Notes:
% ------
% - Ensure input dimensions are consistent with system and observer models.
% - Attack dynamics are applied via 'attackFunction'.
% - Nonlinear spring dynamics are applied using 'NLspring'.
%
% See also:
% -------------
% NLspring.m, attackFunction.m, cmo3d.m, ssmo.m

    % update wait bar and catch exception if the user closed it
    try
        waitbar(t/tmax,wb,sprintf('Solver is currently at time: %2.4f',t))
    catch ME
        switch ME.identifier
            case 'MATLAB:waitbar:InvalidSecondInput'
                error('User terminated the solver.')
            otherwise
                rethrow(ME)
        end

    end

    % extract Jmo and Pmo for noise
    xsys = x(1:sys.nx);
    y = sys.COutputs*xsys;
    a = attackFunction(t,Attack.attackList);
    if sys.NLsize > 0
        dxsys = sys.A*xsys + sys.E*NLspring(sys,y);
    else
        dxsys = sys.A*xsys;
    end

    % Calculations for the CMO2D
    if whichMO(1) == 1
        xcmo2d = x(xIds.xcmo2dStart:xIds.xcmo2dEnd);
        u2D = zeros(Jmo.sys.nu,1);
        Attack2d = attackFunction(t,CMO2D.attack);
        v2D = Noise.getMONoise(t,"2D",Jmo,Pmo);
        eta2D = [u2D; v2D + Attack2d];
        dxcmo2d = CMO2D.A*[xsys; xcmo2d] + CMO2D.F*eta2D;
    else
        dxcmo2d = [];
    end

    % Calculations for the CMO3D
    if whichMO(2) == 1
        xcmo3d = reshape(x(xIds.xcmo3dStart:xIds.xcmo3dEnd),sys.nx,1,CMO3D.numObservers);
        v3D = Noise.getMONoise(t,"3D",Jmo,Pmo);
        Attack3d = attackFunction(t,CMO3D.attack3d);
        dxcmo3d = pagemtimes(CMO3D.ApLC,xcmo3d) - pagemtimes(CMO3D.LC,xsys) - pagemtimes(CMO3D.L,Attack3d + v3D);

        if ~sys.Linear
            y3D = get3Dy(y,Jmo,Pmo);
            numObservers = Jmo.numObservers + Pmo.numObservers;
            phi = zeros(sys.NLsize,1,numObservers);
            for i = 1:1:numObservers
                yi = y3D(:,:,i) + Attack3d(:,:,i) + v3D(:,:,i);
                phi(:,:,i) = NLspring(sys,yi);
            end
            dxcmo3d = dxcmo3d +  pagemtimes(CMO3D.E,phi);
        end

    else
        dxcmo3d = [];
    end

    % Calculations for the SSMO
    if whichMO(3) == 1
        v = Noise.interpNoise("all",t);
        xssmo = x(xIds.xssmoStart:xIds.xssmoEnd);
        dxssmo = SSMO.A*xssmo + SSMO.B*([NLspring(sys,y+a+v) ;y+v] + [zeros(sys.NLsize,1) ;a]);
    else
        dxssmo = [];
    end

    dx = [dxsys(:); dxcmo2d(:); dxcmo3d(:); dxssmo(:)];

end
```

Listing 26: NLspring.m

```
function NLk = NLspring(sys,y)
% NLspring Function
%
% The 'NLspring' function calculates the nonlinear spring force based on a
% hardening spring model. It computes the force for each spring in the system
% based on the displacement (y) of the masses and the parameters of the system
% such as spring constant (k), mass (m), and nonlinear factor (a). The function
% is used when the system has nonlinearities (i.e., when 'NLsize > 0').
%
% Documentation written by ChatGPT.
%
```

```
% Inputs:
% -------
% - 'sys': A mass-spring-damper system object that contains the properties
%   defining the system, including spring constants (k), mass (m), nonlinear
%   size (NLsize), and the nonlinear factor (a).
% - 'y': A vector of displacements of the masses in the system.
%
% Outputs:
% --------
% - 'NLk': A vector containing the nonlinear spring forces for each mass.
%   If the system has no nonlinearities (i.e., 'sys.NLsize == 0'), the output
%   is an empty array.
%
% Function Description:
% ---------------------
% The function calculates the nonlinear spring force for each spring in the
% system based on the displacement from the previous spring and the system's
% properties. The nonlinear spring force is determined by the formula:
%
%     NLk = (k * a^2 * d^3) / m
%
% where:
% - 'k' is the spring constant,
% - 'a' is the nonlinear factor,
% - 'd' is the displacement difference between the current and previous masses,
%   and
% - 'm' is the mass.
%
% If the system does not have nonlinearities (i.e., 'sys.NLsize == 0'),
% the function returns an empty array.
%
% Initialization Steps:
% ---------------------
% - The function first checks if the system has nonlinearities by examining
%   'sys.NLsize'. If no nonlinearities are present, it returns an empty array.
% - If nonlinearities exist, it initializes a zero vector 'NLk' to store
%   the nonlinear spring forces.
% - The function iterates through each mass, calculating the displacement
%   difference ('d') between the current mass and the previous one, and
%   computes the corresponding nonlinear spring force using the specified formula.
%
% Example:
% --------
% To calculate the nonlinear spring forces for a given system:
%
% sys = msd(true, 2, [1; 1], [10; 10], [0.5; 0.5]);  % A system object
% y = [0.2; 0.3];  % Displacements of the masses
% NLk = NLspring(sys, y);
%
% This will compute the nonlinear spring forces for the 2-mass system with
% specified displacements.
%
% Notes:
% ------
% - This function applies a nonlinear hardening spring model. The spring force
%   is cubic with respect to the displacement difference between successive masses.
% - The function handles both systems with and without nonlinearities, providing
%   flexibility in the modeling of different systems.
% - If the system has no nonlinearities, it returns an empty array, indicating
%   that no nonlinear forces need to be computed.
%
% See also:
% ---------
% msd, multiObserverODE

    if sys.NLsize == 0
        NLk = [];
    else
        NLk = zeros(sys.NLsize,1);
        dPrev = 0;
        for j = 1:1:sys.NLsize
            d = y(j) - dPrev;
            dPrev = dPrev + d;
            NLk(j) = (sys.k*sys.a^2*d^3)/sys.m;
        end

    end

end
```

Listing 27: pad3DL.m

```
function Lpadded = pad3DL(mo)
% pad3DL Function
%
% The 'pad3DL' function generates a 3D matrix of padded observer matrices ('L')
% for a 3D-CMO system. Each observer's matrix is padded to match the
% total system outputs, with each resulting padded matrix placed in the third
% dimension of the output.
%
% Documentation written by ChatGPT.
%
% Inputs:
% -------
% - 'mo': An instance of the multi-observer ('mo') class. The class should
%   contain the following relevant properties:
%   - 'nx': The number of states in the system.
%   - 'numOutputs': The total number of outputs in the system.
%   - 'numObservers': The total number of observers in the multi-observer.
%   - 'Li': A 3D array of observer matrices for each observer, where the
%     third dimension corresponds to the observer index.
%   - 'CiIndices': A 2D matrix where each row contains the indices of the
%     outputs used by the corresponding observer.
%
% Outputs:
% --------
% - 'Lpadded': A 3D matrix containing the padded 'L' matrices for all observers.
%   Each 2D slice along the third dimension corresponds to the padded observer
%   matrix for one observer.
%
% Function Description:
% ---------------------
% The function iterates through the observer matrices in the multi-observer
% system and applies zero-padding to each matrix using the 'padL' function.
% The padding ensures that each observer matrix aligns with the total number
% of system outputs. The padded matrices are then stacked along the third
% dimension to form the output 'Lpadded'.
%
% Initialization Steps:
% ---------------------
% - The function initializes 'Lpadded' as a zero matrix with dimensions:
%   - Rows: Number of system states ('nx').
%   - Columns: Total number of system outputs ('numOutputs').
%   - Third dimension: Number of observers ('numObservers').
% - It iterates over each observer:
%   - Extracts the observer's matrix ('L') from 'Li'.
%   - Extracts the corresponding output indices ('cIndices') from 'CiIndices'.
%   - Calls the 'padL' function to pad the observer's matrix to the total
%     number of system outputs.
%   - Assigns the resulting padded matrix to the appropriate slice of 'Lpadded'.
%
% Notes:
% ------
% - The 'padL' function is used to handle the zero-padding for individual
%   observer matrices.
% - The 'mo' object must include the required properties ('nx', 'numOutputs',
%   'numObservers', 'Li', 'CiIndices') for the function to work correctly.
%
% See also:
% ---------
% padL, mo, cmo3d


    Lpadded = zeros(mo.nx,mo.numOutputs,mo.numObservers);

    for c = 1:1:mo.numObservers
        L = mo.Li(:,:,c);
        cIndices = mo.CiIndices(c,:);
        Lpadded(:,:,c) = padL(L,cIndices,mo.numOutputs);
    end

end
```

Listing 28: padL.m

```
function paddedL = padL(L,outputsID,numOutputs)
% padL Function
%
% The 'padL' function creates a zero-padded version of an observer matrix ('L')
% based on a set of specified output indices ('outputsID'). It ensures that
% the resulting matrix has the desired dimensions for use in the SSMO, with
% zeros in positions corresponding to outputs not included in 'outputsID'.
%
% Documentation written by ChatGPT.
%
% Inputs:
% -------
% - 'L': The observer matrix to be padded. This matrix has dimensions
%   corresponding to the number of original states and the number of outputs
%   defined by the observer.
% - 'outputsID': A vector of indices specifying which outputs are present in 'L'.
% - 'numOutputs': The total number of outputs in the system. This determines
%   the number of columns in the padded matrix.
%
% Outputs:
% --------
% - 'paddedL': The padded observer matrix. It has the same number of rows
%   as the input matrix 'L' but includes additional columns filled with zeros
%   for outputs not specified in 'outputsID'. The total number of columns
%   equals 'numOutputs'.
%
% Function Description:
% ---------------------
% The function takes the observer matrix 'L' and pads it with zeros in the
% columns corresponding to outputs not listed in 'outputsID'. It ensures that
% the resulting matrix aligns with the system's total number of outputs.
% It is used to create an L matrix that can be multiplied with the full
% output instead of its corresponing subset of the outputs and still give
% the same result.

    numOriginalStates  = size(L,1);
    numObserverOutputs = size(L,2);
    paddedL = zeros(numOriginalStates,numOutputs);

    for i = 1:1:numObserverOutputs
        colId = outputsID(i);
        paddedL(:,colId) = L(:,i);
    end

end
```

Listing 29: rootsToCoefficients.m

```
function coefficients = rootsToCoefficients(roots)
% rootsToCoefficients Function
%
% The 'rootsToCoefficients' function computes the coefficients of a polynomial
% given its roots. The resulting coefficients are returned in descending order
% of powers, starting with the highest order term.
%
% Documentation written by ChatGPT.
%
% Inputs:
% -------
% - 'roots': A vector (either row or column) of polynomial roots. The input
%   must be either a (1,n) or (n,1) matrix, where 'n' is the number of roots.
%
% Outputs:
% --------
% - 'coefficients': A row vector containing the coefficients of the polynomial
%   in descending order of powers, with the coefficient of the highest order
%   term appearing first.
%
% Function Description:
% ---------------------
% The function takes the roots of a polynomial and computes its coefficients
% by expanding the polynomial expression (x - r_1)(x - r_2)*...*(x - r_n).
% The resulting coefficients are formatted into a vector with the highest
% order term's coefficient at the start.
%
% Implementation Steps:
% ---------------------
% 1. The input 'roots' is checked to ensure it is either a row or column
%    vector. If not, an error is thrown.
% 2. If 'roots' is a column vector, it is transposed to a row vector for
%    consistency.
% 3. A symbolic polynomial (x - r_1)(x - r_2)*...*(x - r_n) is constructed
%    using the roots.
% 4. The 'coeffs' function extracts the polynomial's coefficients.
% 5. The coefficients are flipped ('fliplr') to ensure they are returned in
%    descending order of powers.
%
% Notes:
% ------
% - The function uses symbolic math ('sym') for polynomial expansion and
%   coefficient extraction.
% - Ensure that the input 'roots' is a real or complex vector; matrices or
%   multi-dimensional arrays are not supported.
% - The function ensures that the output is a numeric array by converting
%   symbolic coefficients to doubles.
%
% Error Handling:
% ---------------
% - If 'roots' is neither a row nor a column vector, the function throws an
%   error with the message:
%   'Input variable roots is not (1 x n) or (n x 1).'
%
% See also:
% ---------
% sym, expand, coeffs, fliplr


    % Format roots into (1xn) matrix
    rootsSize = size(roots);
    % Throw error if roots is not (1 x n) or (n x 1)
    if rootsSize(1) > 1 && rootsSize(2) > 1
        error('Input variable roots is not (1 x n) or (n x 1).', roots);
    % Flip roots into (1 x n) if it is (n x 1).
    elseif rootsSize(1) > 1
        roots = roots';
    end
    % Define polynomial (x - root1)...(x-rootn) and extract coefficients.
    x = sym("x");
    desiredPolynomial = expand(prod(x-roots));
    coefficients = double(coeffs(desiredPolynomial));
    % flip coefficients to arrange highest order coefficient left and
    % counting down
    coefficients = fliplr(coefficients);

end
```

Listing 30: sbeCPU.m

```
function [bestStateEstimate, jBestEstimate] = sbeCPU(x,tsteps,PsubsetOfJIndices,numOfPsubsetsInJ,Jmo,Pmo,sys,wb)
```

```
% selectBestEstimate Function
%
% The 'selectBestEstimate' function determines the best state estimate
% (xhat) from a set of subobservers P within J observers at each time step.
% The function identifies the observer index sigma(t) that provides the best
% estimate based on a predefined selection criterion, and returns the
% corresponding state estimate and observer index for all time steps.
%
% Documentation written by ChatGPT.
%
% Inputs:
% -------
% - 'x': A matrix containing the combined state trajectories for all observers.
%   The state trajectories are divided into:
%   1. J observer states
%   2. P observer states
%
% - 'tsteps': The number of time steps for which the state estimates are provided.
%
% - 'PsubsetOfJIndices': A matrix containing indices that map each J
%   observer to its corresponding P subobservers.
%
% - 'numOfPsubsetsInJ': The number of P subobservers within each
%   J observer.
%
% - 'Jmo': A structure containing information about the J observers,
%   specifically:
%   - 'Jmo.numObservers': The total number of J observers.
%
% - 'Pmo': A structure containing information about the P observers,
%   specifically:
%   - 'Pmo.numObservers': The total number of P observers.
%
% - 'sys': A structure containing system properties, including:
%   - 'sys.nx': The number of state variables for each observer.
%
% Outputs:
% --------
% - 'bestStateEstimate': A matrix of size n_x by tsteps containing the
%   best state estimate at each time step.
%
% - 'jBestEstimate': A row vector of length 'tsteps' containing the index of
%   the J observer that provided the best state estimate at each time step.
%
% Function Description:
% ---------------------
% The function follows these steps:
% 1. Extract the state trajectories of the J observers and P observers
```

```
   from
%    'x'.
% 2. For each time step t:
%    a. Reshape the states of the J and P observers into 3D matrices for
%       easier manipulation.
%    b. Loop through each J observer:
%       - Compute the maximum norm difference Pi_j between the
%         J observer's state and the states of its corresponding
%         P subobservers.
%    c. Identify the J observer with the minimum Pi_j
%       value and select its state as the best estimate for that time step
   .
% 3. Return the best state estimates and the corresponding observer
   indices.
%
% Selection Criterion:
% --------------------
% The selection is based on minimizing the observability degradation,
% as described in Chong (2015):
% 1. The best estimate is defined as:
%    x_hat(t) = x_hat{_sigma(t)}(t)
%    where \sigma(t) is the observer index minimizing Pi_j(t).
% 2. Pi_j(t) is calculated as:
%    Pi_j(t) = max(x_hat_j(t) - x_hat_p(t))
%
% Example:
% --------
% Consider a system with n_x = 2, J = 3 observers, and P = 2
% subobservers. Given:
% - 'x': A matrix of size nx times tsteps containing state estimates
%   for all observers.
% - 'PsubsetOfJIndices = [1 2; 1 3; 2 3]': Maps each J observer
%   to its corresponding P subobservers.
% - 'tsteps = 1': Single time step.
%
% The output will provide:
% - 'bestStateEstimate': The best state estimate for t = 1.
% - 'jBestEstimate': The index of the J observer that provided
%   the best estimate.
%
% Notes:
% ------
% - The function assumes that the input data is correctly formatted.
% - If multiple J observers have the same Pi_j value,
%   the first observer (lowest index) is selected.
%
% See also:
% ---------
% norm, reshape, find



    % xJ contains the states of the J observers
    xJ = x(sys.nx+1:(1+Jmo.numObservers)*sys.nx,:);
    % xP contains the states of the P observers
    xP = x((1+Jmo.numObservers)*sys.nx+1:(1+Jmo.numObservers+Pmo.numObservers)*sys.nx,:);

    % Initialize PiJ, the array that will house all Pi j (the maximum
    % difference between a J observer and all its P observers).
    PiJ = zeros(Jmo.numObservers,1);

    % create emtpy array to store best estimate and which j supplies it
    bestStateEstimate = zeros(sys.nx,tsteps);
    jBestEstimate = zeros(1,tsteps);


    for t = 1:1:tsteps
        % update wait bar and catch exception if the user closed it
        try
            waitbar(t/tsteps,wb,sprintf('Selector is currently at timestep: %d/%d',t,tsteps))
        catch ME
            switch ME.identifier
                case 'MATLAB:waitbar:InvalidSecondInput'
                    error('User terminated the solver.')
                otherwise
                    rethrow(ME)
            end

        end

        xJ3D = reshape(xJ(:,t),sys.nx,1,[]);
        xP3D = reshape(xP(:,t),sys.nx,1,[]);
        for j = 1:1:Jmo.numObservers
            xj = xJ3D(:,:,j);
            % select the row of PsubsetOfJIndices that contains the ids of
            % p that are a subset of J
            pSubsetofjIndices = PsubsetOfJIndices(j,:);
            % diflist will store the difference between solj and all its
            % subsets solp
            difflist = zeros(numOfPsubsetsInJ,1);
            % Loop over the P observers that are a subset of j
            for p = 1:1:numOfPsubsetsInJ
                % select the index of p that will be checked
                pIndex = pSubsetofjIndices(p);
                if pIndex ~= 0
                    % select the solution of p that corresponds to this index
                    xp = xP3D(:,:,pIndex);
                    % calculate and store the difference between solj and solp
                    dif = norm(xj-xp);
                    difflist(p,:) = dif;
                end

            end

            % we now select Pi j as the maximum of this list
            Pij = max(difflist);
            PiJ(j,1) = Pij;

        end

        % Select the extimate xj with j being the best smallest value
        % in PiJ
        jBestEstimateTstep = find(PiJ==min(PiJ));
        jBestEstimateTstep = jBestEstimateTstep(1);
        jBestEstimate(:,t) = jBestEstimateTstep;
        bestEstimateTstep = xJ((jBestEstimateTstep-1)*sys.nx+1:jBestEstimateTstep*sys.nx,t);
        bestStateEstimate(:,t) = bestEstimateTstep;

    end

end
```

Listing 31: selectRandomSubset.m

```
function [subset,remainder] = selectRandomSubset(set,subsetSize)
% selectRandomSubset Function
%
% The 'selectRandomSubset' function selects a random subset of specified
% size from a given set and returns both the subset and the remaining
% elements of the set.
%
% Documentation written by ChatGPT.
%
% Inputs:
% -------
% - 'set': A row vector containing the elements of the set from which a
%   subset will be selected.
% - 'subsetSize': The number of elements to include in the randomly
%   selected subset.
%
% Outputs:
% --------
% - 'subset': A row vector containing the randomly selected elements
%   (subset of the input 'set') of size 'subsetSize'.
% - 'remainder': A row vector containing the elements of the input 'set'
%   that were not selected as part of the subset.
%
% Function Description:
% ---------------------
% 1. The function determines the total number of elements in the input
%    'set' using 'size'.
% 2. It generates a random permutation of indices from the set using
%    'randperm' and selects the first 'subsetSize' indices to define
%    the subset.
% 3. The subset is extracted from the input set based on the selected
%    random indices.
% 4. The 'setdiff' function is used to compute the elements of the
%    input 'set' that were not included in the subset, forming the remainder.
%
% Example:
% --------
% Given:
% - 'set = [1, 2, 3, 4]'
% - 'subsetSize = 2'
%
% A possible output could be:
% - 'subset = [1, 3]'
% - 'remainder = [2, 4]'
%
% Notes:
% ------
% - The function assumes that the input 'set' is a row vector.
% - 'subsetSize' must be less than or equal to the total number of
%   elements in 'set'. An error will occur otherwise.
% - The random selection process ensures a unique subset for each
%   function call, provided the random number generator state changes.
%
% Error Handling:
% ---------------
% - Ensure that 'subsetSize' does not exceed the size of the input 'set'.
%   Otherwise, the function will throw an error.
%
% Implementation Steps:
% ---------------------
% 1. Determine the size of the input 'set' ('setSize').
% 2. Use 'randperm' to generate a random permutation of indices and
%    select the first 'subsetSize' indices.
% 3. Extract the elements corresponding to the selected indices
%    from 'set' to form the 'subset'.
% 4. Use 'setdiff' to determine the 'remainder'.
%
% See also:
% ---------
% randperm, setdiff

    % choose indices to take randomly
    setSize = size(set,2);
    indices = randperm(setSize,subsetSize);
    % loop through indices and add it to subset matrix
    subset = zeros(1,subsetSize);
    for index = 1:1:subsetSize
        subset(1,index) = set(1,indices(1,index));
    end
    remainder = setdiff(set,subset);
end
```

Listing 32: SSMOTransformationSetup.m

```
function T = SSMOTransformationSetup(Ap,Bp,q,mo)
% SSMOTransformationSetup Function
```

```
%
% Constructs the transformation matrices 'T' for a State-Space Multi-Observer
% (SSMO) system.
%
% Documentation written by ChatGPT.
%
% Syntax:
% -------
% 'T = SSMOTransformationSetup(Ap, Bp, q, mo)'
%
% Inputs:
% -------
% - 'Ap': A 3D array of state-space 'A' matrices, where each slice along the
%   third dimension corresponds to an observer's 'A' matrix.
% - 'Bp': A 3D array of state-space 'B' matrices, where each slice along the
%   third dimension corresponds to an observer's 'B' matrix.
% - 'q': A vector of coefficients from the characteristic polynomial (excluding
%   the leading coefficient).
% - 'mo': An object containing properties of the multi-observer system, including:
%   - 'nx': Number of states in the system.
%   - 'numOutputs': Number of system outputs.
%   - 'sys.NLsize': Number of nonlinear components (if any).
%   - 'numObservers': Total number of observers.
%
% Outputs:
% --------
% - 'T': A 3D array of transformation matrices, where each slice corresponds
%   to a specific observer. Dimensions are
%   '(nx x ((numOutputs + NLsize) * nx) x numObservers)'.
%
% Implementation Steps:
% ---------------------
% 1. Compute the controllability matrices ('Rp') for each observer using the
%    'ctrb' function, which constructs the controllability matrix from 'Ap'
%    and 'Bp' for each observer.
% 2. Generate 'RqValues', a companion-form-like matrix based on the coefficients
%    'q', where each row shifts the coefficients.
% 3. Expand 'RqValues' to accommodate all outputs and nonlinear states by
%    taking the Kronecker product ('kron') with an appropriately sized identity matrix.
% 4. Multiply the controllability matrix 'Rp' with the expanded 'Rq' matrix
%    to create the transformation matrix 'T' for each observer.
%
% Example:
```

```
% --------
% % Define Ap and Bp as 3D arrays (A and B matrices for multiple observers)
% Ap = rand(3,3,5); % Example: 5 observers, 3x3 A matrices
% Bp = rand(3,1,5); % Example: 5 observers, 3x1 B matrices
%
% % Define the polynomial coefficients and observer object
% q = [2.5; -3.0; 1.5];
% mo.nx = 3;
% mo.numOutputs = 2;
% mo.sys.NLsize = 1;
% mo.numObservers = 5;
%
% % Call SSMOTransformationSetup
% T = SSMOTransformationSetup(Ap, Bp, q, mo);
%
% Notes:
% ------
% - The 'RqValues' matrix is constructed using the companion matrix structure
%   derived from the coefficients 'q'.
% - The Kronecker product ('kron') ensures the transformation is scaled to
%   include all outputs and nonlinear states.
%
% Matrix Structure:
% -----------------
% - 'Rp' is the controllability matrix for each observer.
% - 'Rq' is derived by expanding the companion-form coefficients of the
%   characteristic polynomial.
% - Each transformation matrix 'T(:,:,i)' is the product of 'Rp(:,:,i)' and 'Rq'.
%
% See also:
% ---------
% ctrb, kron, ssmo, ssmoSysSetup


    % precompute dimensions
    a = mo.nx*(mo.numOutputs + mo.sys.NLsize);
    c = mo.nx;

    % Generate Rp matrices
    Rp = zeros(c,a,mo.numObservers);
    for i = 1:1:mo.numObservers
       Rp(:,:,i) = ctrb(Ap(:,:,i),Bp(:,:,i));
    end

    % Define correct I to use in kron product
    I = eye(mo.numOutputs + mo.sys.NLsize);
    RqValues = zeros(mo.nx);

    % Place single values of Rq correctly before expanding them with kron
    % prouct.
    for i = 1:1:mo.nx
```

```
        newRow = [zeros(1,i-1) 1 q(1:end-1)];
        RqValues(i,:) = newRow(1:mo.nx);
    end

    % multiply all individual eigenvalues by the appropriatly sized
    % identity matrix
    Rq = kron(RqValues,I);

    % Create all T matrices by multiplying Rp and Rq
    T = zeros(mo.nx,(mo.numOutputs + mo.sys.NLsize)*mo.nx,mo.numObservers);
    for i = 1:1:mo.numObservers
        T(:,:,i) = Rp(:,:,i)*Rq;
    end

end
```

Listing 33: systemPSetup.m

```
function [Ap,Bp] = systemPSetup(mo,Lpadded)
% systemPSetup Function
%
% Constructs the 'Ap' and 'Bp' matrices for a multi-observer system, where
% 'Ap' incorporates observer dynamics, and 'Bp' accounts for input and output corrections.
%
% Documentation written by ChatGPT.
%
% Syntax:
% -------
% '[Ap, Bp] = systemPSetup(mo, Lpadded)'
%
% Inputs:
% -------
% - 'mo': Multi-observer system object containing the following properties:
%   - 'nx': Number of states in the system.
%   - 'numObservers': Number of observers in the system.
%   - 'sys.E': Input-to-state matrix ('E') of the original system (optional).
%   - 'Ai': State-space 'A' matrices for each observer, organized in a 3D array.
%   - 'Li': Observer gain matrices ('L'), organized in a 3D array.
%   - 'Ci': Output matrices ('C'), organized in a 3D array.
% - 'Lpadded': A 3D array of gain matrices ('L') padded to include all output dimensions.
%
% Outputs:
% --------
% - 'Ap': A 3D array of modified state-space 'A' matrices, incorporating
%   observer dynamics. Dimensions: '(nx x nx x numObservers)'.
% - 'Bp': A 3D array of input-to-state correction matrices. If the system
%   has nonlinear dynamics ('sys.E'), 'Bp' combines the 'E' matrix with the
%   padded gain matrix. Otherwise, it consists of the negative padded gain matrix.
%
% Implementation Steps:
% ---------------------
% 1. Initialize 'Ap' as a 3D array with dimensions '(nx x nx x numObservers)'.
% 2. Define 'Bp' based on whether the system includes a non-empty input-to-state
%    matrix ('sys.E'):
%    - If 'sys.E' exists, concatenate 'E' with the negated 'Lpadded'.
%    - If 'sys.E' is empty, 'Bp' is set to '-Lpadded'.
% 3. Loop over the observers to compute each slice of 'Ap' using the formula:
%    'Ap(:,:,i) = Ai(:,:,i) + Li(:,:,i) * Ci(:,:,i)'.
%
% Matrix Structure:
% -----------------
% - 'Ap(:,:,i)' represents the modified state-space matrix for the 'i'th observer.
% - 'Bp(:,:,i)' includes contributions from the system's 'E' matrix (if available)
%   and the negated observer gains.
%
% See also:
% ---------
% ctrb, ssmo, pad3DL


    Ap = zeros(mo.nx,mo.nx,mo.numObservers);
    if size(mo.sys.E,2) > 0
        Bp = cat(2,repmat(mo.sys.E,1,1,size(Lpadded,3)), -Lpadded);
    else
        Bp = -Lpadded;
    end

    for i = 1:1:mo.numObservers
        Ap(:,:,i) = mo.Ai(:,:,i) + mo.Li(:,:,i)*mo.Ci(:,:,i);
    end

end
```

Listing 34: x0setup.m

```
function [x0, xIds] = x0setup(x0input,whichMO,sys,Jmo,Pmo)
% x0setup Function
%
% This function sets up the initial state vector 'x0' based on the input
% 'x0input'and the specified structure of the system, considering the
% number of observers and the configuration of the system (such as 2D-CMO,
% 3D-CMO, and SSMO).
%
% Syntax:
% -------
% 'x0 = x0setup(x0input, whichMO, sys, Jmo, Pmo)'
%
% Inputs:
% -------
% - 'x0input': The initial input state vector for the system, where the
%   first
%   'sys.nx' are selected. This avoids errors when the list is longer.
% - 'whichMO': A vector specifying which modes of operation are active.
%   It is a 3-element vector where:
%     - 'whichMO(1)' indicates if 2D CMO is active ('1' for active, '0'
%       for inactive).
%     - 'whichMO(2)' indicates if 3D CMO is active.
%     - 'whichMO(3)' indicates if SSMO is active.
% - 'sys': A structure representing the system, containing the number of
%   states ('nx') and the size of the nonlinearity ('NLsize').
% - 'Jmo': A structure representing the J-mode observers, including the
%   number of observers ('numObservers') and other parameters.
% - 'Pmo': A structure representing the P-mode observers, also including
%   'numObservers' and other parameters.
%
% Outputs:
% --------
% - 'x0': The initial state vector ('x0') as a column vector that includes
%   all the states for the system and the observers, based on the mode
%   configurations specified.
%
% Process:
% --------
% 1. Extract the system's state vector ('x0sys') from the first 'sys.nx'
%    elements of 'x0input'.
% 2. Compute the total number of observers as the sum of 'Jmo.numObservers'
%    and 'Pmo.numObservers'.
% 3. Set up 'x0cmo2d' if the 2D CMO mode is active (if 'whichMO(1)' equals '1').
%    - It is initialized as a zero vector of size '(numObservers * sys.nx)'.
% 4. Set up 'x0cmo3d' if the 3D CMO mode is active (if 'whichMO(2)' equals '1').
%    - It is initialized as a zero 3D array of size '(sys.nx, 1, numObservers, 1)'.
% 5. Set up 'x0ssmo' if the SSMO mode is active (if 'whichMO(3)' equals '1').
%    - It is initialized as a zero vector of size '(sys.nx * (Jmo.numOutputs + sys.NLsize))'.
% 6. Concatenate the initialized vectors and arrays ('x0sys', 'x0cmo2d', 'x0cmo3d', and 'x0ssmo')
%    to form the final initial state vector 'x0'.
%
%
% Matrix Structure:
% -----------------
% - 'x0sys': The first 'sys.nx' elements of 'x0input', representing the system's
%   state vector.
% - 'x0cmo2d': The state vector for 2D CMO observers (if active), all zeros.
% - 'x0cmo3d': The state array for 3D CMO observers (if active), all zeros.
% - 'x0ssmo': The state vector for the SSMO (if active), all zeros.
%
% See also:
% ---------
% multiObserverODE


    x0sys = x0input(1:sys.nx);
%     xIds = struct([]);
    numObservers = Jmo.numObservers + Pmo.numObservers;
    if whichMO(1) == 1
        x0cmo2d = zeros(numObservers*sys.nx,1);
        xIds.xcmo2dStart = sys.nx + 1;
        xIds.xcmo2dEnd   = sys.nx + numObservers*sys.nx;
    else
        x0cmo2d = [];
        xIds.xcmo2dEnd = sys.nx;
    end

    if whichMO(2) == 1
        x0cmo3d = zeros(sys.nx,1,numObservers,1);
        xIds.xcmo3dStart = xIds.xcmo2dEnd + 1;
        xIds.xcmo3dEnd   = xIds.xcmo2dEnd + numObservers*sys.nx;
    else
        x0cmo3d = [];
        xIds.xcmo3dEnd = xIds.xcmo2dEnd;
    end

    if whichMO(3) == 1
        x0ssmo = zeros(sys.nx*(Jmo.numOutputs + sys.NLsize),1);
        xIds.xssmoStart = xIds.xcmo3dEnd + 1;
        xIds.xssmoEnd    = xIds.xcmo3dEnd + (sys.NLsize + Jmo.numOutputs)*sys.nx;
    else
        x0ssmo = [];
        xIds.xssmoStart = xIds.xcmo3dEnd;
    end

    x0 = [x0sys(:); x0cmo2d(:); x0cmo3d(:); x0ssmo(:)];
end
```

F Matlab code chapter 7

Listing 35: probabilities.m

```
clearvars
% Define system parameters
max_N_S = 20;
N_phi = 2;
```

```
max_N_M = 10;

looseProbs =  zeros((max_N_S-6)/2,max_N_S);
strictProbs = zeros((max_N_S-6)/2,max_N_S);

for N_S = 6:2:max_N_S
    looseProbsRow = findProbabilities(N_S, N_phi, N_S, 'loose');
    % fprintf(append('N_S = ',string(N_S),', N_phi = ', string(N_phi),' ', 'loose grouping','\n'))
    % looseT = array2table(looseProbs);
    % looseT = renamevars(looseT,1:1:max_N_M,string(1:1:max_N_M));
    % looseT.Properties.RowNames = {'p(robust)','p(not robust)'};
    % disp(looseT)

    strictProbsRow = findProbabilities(N_S, N_phi, N_S, 'strict');
    % fprintf(append('N_S = ',string(N_S),', N_phi = ', string(N_phi),' ', 'strict grouping','\n'))
    % strictT = array2table(strictProbs);
    % strictT = renamevars(strictT,1:1:max_N_M,string(1:1:max_N_M));
    % strictT.Properties.RowNames = {'p(robust)','p(not robust)'};
    % disp(strictT)

    looseProbs((N_S-6)/2 + 1,1:size(looseProbsRow,2))  =  looseProbsRow(1,:);
    strictProbs((N_S-6)/2 + 1,1:size(looseProbsRow,2)) = strictProbsRow(1,:);
end

function probs = findProbabilities(N_S, N_phi, max_N_M, grouping)
    % create groups
    S = 1:1:N_S;
    same_sensors = reshape(S,N_phi,[],1);
    grouped_sensors = combinations(same_sensors,grouping);
    N_O = size(grouped_sensors,1);

    probs = zeros(2,max_N_M);
    % loop over all number of attacks until max_N_M
    for N_M = 1:1:max_N_M
        % number of possible attack combinations
        num_attack_combs = nchoosek(N_S,N_M);
        attack_combs  = nchoosek(S,N_M);

        % intialize counters
        robust = 0;
        unrobust = 0;

        % loop over all attack combinations
        for c = 1:1:num_attack_combs
            comb = attack_combs(c,:);

            % intialize counter
            N_A = 0;
            for g = 1:1:N_O
```

```
                % if any entry of the combination is a member of the sensor
                % group add one to N_A
                if any(ismember(comb,grouped_sensors(g,:)))
                    N_A = N_A + 1;
                end
            end

            % check if robustness condition holds, add one to robust if it
            % does, add one to unrobust if it doesn't
            if 2*N_A < N_O
                robust = robust + 1;
            else
                unrobust = unrobust + 1;
            end

        end

        % Calculate share of combinations that are robust and not robust
        probs(1,N_M) = robust/num_attack_combs;
        probs(2,N_M) = unrobust/num_attack_combs;
    end
end
```