

immrax: A Parallelizable and Differentiable Toolbox for Interval Analysis and Mixed Monotone Reachability in JAX ^{*}

Akash Harapanahalli^{*} Saber Jafarpour^{**} Samuel Coogan^{*}

^{*} School of Electrical and Computer Engineering, Georgia Institute of Technology, GA, USA 30318, {*aharapan, sam. coogan*}@gatech.edu

^{**} Department of Electrical, Computer, and Energy Engineering, University of Colorado, Boulder, CO, USA 80309, *saber.jafarpour@colorado.edu*.

Abstract: We present an implementation of interval analysis and mixed monotone interval reachability analysis as function transforms in Python, fully composable with the computational framework JAX. The resulting toolbox inherits several key features from JAX, including computational efficiency through Just-In-Time Compilation, GPU acceleration for quick parallelized computations, and Automatic Differentiability. We demonstrate the toolbox’s performance on several case studies, including a reachability problem on a vehicle model controlled by a neural network, and a robust closed-loop optimal control problem for a swinging pendulum.

Keywords: Interval analysis, Reachability analysis, Automatic differentiation, Parallel computation, Computational tools, Optimal control, Robust control

1. INTRODUCTION

Interval analysis is a classical field concerned with bounding the output of mappings across uncertain inputs (Jaulin et al., 2001). For dynamical systems, interval analysis provides a computationally cheap, scalable, and sound approach for studying the effects of uncertainty, through differential inequalities (Shen and Scott, 2017) and mixed monotone embeddings (Jafarpour et al., 2023). While these methods have been studied extensively in the literature, most implementations either fail to (i) utilize the key computational breakthroughs in the last decade in parallel processing on GPUs and TPUs; and/or (ii) address the challenges of the modern learning-enabled control system, such as efficient gradient computation. JAX (Bradbury et al., 2018) is an evolving numerical computation framework for Python developed by researchers at Google Deepmind. At its core, JAX is a framework for *composable function transformations*, i.e., transforms that take functions as input and return new functions with some desired property. For instance, `jit` uses XLA to transform a function into a compiled program executed on CPU/GPU/TPU, `grad/jacfdw/jacrev` return a new function evaluating the input function’s derivative using either reverse- or forward-mode autodifferentiation from Autograd (*autodiff*), and `vmap` transforms a function into a new version that parallelizes its execution over several different inputs (*vectorization*). These transformations can be composed an arbitrary number of times. In this paper, we use these features to create an efficient, differentiable framework for interval analysis and interval reachability.

Literature Review There are several existing tools for interval analysis and interval reachability analysis. To our knowledge, most of these tools do not support GPU parallelization and/or computation of gradients. CORA is a MATLAB toolbox with interval and polytopic arithmetic capabilities (Althoff, 2015), JuliaReach is a Julia toolbox supporting interval analysis and Taylor model abstractions (Bogomolov et al., 2019). In previous work, we developed an interval analysis extension for `numpy` called `npinterval` (Harapanahalli et al., 2023), and a mixed monotone interval reachability (Coogan and Arcaç, 2015) tool called `ReachMM` (Jafarpour et al., 2023). However, these lack support beyond basic CPU capabilities. For parallelization, there have been some recent developments in the reachability literature including `ReachNN*` (Fan et al., 2020) which is GPU accelerated and POLAR-Express (Wang et al., 2023) which supports CPU threading, but not GPU processing. DRIP (Everett et al., 2023) uses `jax_verify` for backward reachability of discrete-time linear systems controlled by neural networks.

`jax_verify` is a Python library that can compute the natural inclusion function from Proposition 4—however, it is restricted to the class of functions used for neural networks. The user interface is not compositional, and the usage with existing JAX transformations such as `jit` and `vmap` requires the user to manually convert the `IntervalBound` into a `JittableInputBound` at the input and output of any function to be transformed.

Contributions In this paper, we present a toolbox called `immrax`¹, introducing several new function transforma-

^{*} This work was supported in part by the Air Force Office of Scientific Research under award FA9550-23-1-0303 and the National Science Foundation under award #2219755.

¹ The most recent code for `immrax` can be found at <https://github.com/gtfactslab/immrax>, and the documentation can be found at <https://immrax.readthedocs.io>.

tions to facilitate interval analysis and mixed monotone reachability analysis. These transforms are fully composable with existing JAX transformations, allowing the toolbox to support (i) Just-In-Time (JIT) Compilation for significant improvements in runtime versus the baseline, (ii) GPU parallelizability for rapid, accurate online reachable set estimation; (iii) Automatic Differentiation for learning relationships between reachable set outputs and input parameters. In Section 2, we discuss the theory behind interval analysis and inclusion functions. In particular, in Proposition 7 Part (i), we provide a novel analytical Jacobian-based bound for functions which is vital for capturing stabilizing interactions in closed-loop system analysis. In Section 3, we discuss the theory behind using inclusion functions to build embedding systems which efficiently and scalably bound the output of a dynamical system under uncertainty. In each Section, we present their corresponding implementations in `immrax` as composable function transforms, keeping consistent with the rest of the JAX ecosystem. Finally, in Section 4, we demonstrate the usage of `immrax` on several case studies including efficient reachable set estimation of a nonlinear system controlled by a neural network using GPU parallelization for partitioning, and finding locally optimal solutions to a robust closed-loop optimal control problem on a damped inverted pendulum using Automatic Differentiation.

Notation For $\underline{x}, \bar{x} \in \mathbb{R}^n$, define the partial ordering $\underline{x} \leq \bar{x} \iff \underline{x}_i \leq \bar{x}_i$ for every $i = 1, \dots, n$. Let $[\underline{x}, \bar{x}] := \{x : \underline{x} \leq x \leq \bar{x}\}$ denote a closed and bounded interval, and let $\mathbb{I}\mathbb{R}^n$ be the set of all such intervals. The partial order \leq on \mathbb{R}^n induces the southeast order \leq_{SE} on \mathbb{R}^{2n} , where $\begin{bmatrix} \underline{x} \\ \bar{x} \end{bmatrix} \leq_{\text{SE}} \begin{bmatrix} \underline{y} \\ \bar{y} \end{bmatrix} \iff x \leq y \text{ and } \hat{y} \leq \hat{x}$. Define the upper triangle $\mathcal{T}_{\geq 0}^{2n} := \{\begin{bmatrix} \underline{x} \\ \bar{x} \end{bmatrix} \in \mathbb{R}^{2n} : \underline{x} \leq \bar{x}\}$, and note $\mathbb{I}\mathbb{R}^n \simeq \mathcal{T}_{\geq 0}^{2n}$. We denote this equivalence with $[[\begin{bmatrix} \underline{x} \\ \bar{x} \end{bmatrix}]] := [\underline{x}, \bar{x}]$. For $[\underline{a}, \bar{a}], [\underline{b}, \bar{b}] \in \mathbb{I}\mathbb{R}$ and $[\underline{A}, \bar{A}] \in \mathbb{I}\mathbb{R}^{m \times p}$, $[\underline{B}, \bar{B}] \in \mathbb{I}\mathbb{R}^{p \times n}$,

- (1) $[\underline{a}, \bar{a}] + [\underline{b}, \bar{b}] := [\underline{a} + \underline{b}, \bar{a} + \bar{b}]$ (also on $\mathbb{I}\mathbb{R}^n$ element-wise);
- (2) $[\underline{a}, \bar{a}] \cdot [\underline{b}, \bar{b}] := [\min\{\underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b}\}, \max\{\underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b}\}]$;
- (3) $([\underline{A}, \bar{A}][\underline{B}, \bar{B}])_{i,j} := \sum_{k=1}^p [\underline{A}_{i,k}, \bar{A}_{i,k}] \cdot [\underline{B}_{k,j}, \bar{B}_{k,j}]$.

For $x_1 \in \mathbb{R}^{n_1}$, $x_2 \in \mathbb{R}^{n_2}$, \dots , $x_m \in \mathbb{R}^{n_m}$, let $(x_1, x_2, \dots, x_m) \in \mathbb{R}^{n_1+n_2+\dots+n_m}$ denote their concatenation. For $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, let $df : \mathbb{R}^n \rightarrow \mathbb{R}^{m \times n}$ be its Jacobian, i.e., for $x' \in \mathbb{R}^n$, $df_{x'} = \frac{\partial f}{\partial x} \Big|_{x=x'}$.

2. INCLUSION MODULE: INCLUSION FUNCTION TRANSFORMS IN JAX

The `inclusion` module provides a streamlined interface to work with interval objects and inclusion functions.²

2.1 Inclusion Functions

Interval analysis provides a scalable, compositional approach to bound the output of a function along an interval input, and the key building block is the inclusion function.

Definition 1. (Inclusion Function). Given a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the function $F = \begin{bmatrix} \underline{F} \\ \bar{F} \end{bmatrix} : \mathcal{T}_{\geq 0}^{2n} \rightarrow \mathcal{T}_{\geq 0}^{2m}$ is an *inclusion function* for f if for every $x \in [\underline{x}, \bar{x}]$,

$$\underline{F}(\underline{x}, \bar{x}) \leq f(x) \leq \bar{F}(\underline{x}, \bar{x}).$$

² At this time, `immrax` does not bound floating point rounding errors.

An inclusion function is *monotone* if $[\underline{x}, \bar{x}] \subseteq [\underline{y}, \bar{y}]$ implies

$$\underline{F}(\underline{y}, \bar{y}) \leq \underline{F}(\underline{x}, \bar{x}) \leq f(x) \leq \bar{F}(\underline{x}, \bar{x}) \leq \bar{F}(\underline{y}, \bar{y}).$$

An inclusion function is *thin* if for every $x \in \mathbb{R}^n$,

$$\underline{F}(x, x) = f(x) = \bar{F}(x, x).$$

Remark 2. Notationally, we use the upper triangular interpretation $\mathcal{T}_{\geq 0}^{2n}$ for convenience in Section 3. Most references instead think of inclusion functions as mappings on $\mathbb{I}\mathbb{R}^n$. Given the equivalence between $\mathcal{T}_{\geq 0}^{2n}$ and $\mathbb{I}\mathbb{R}^n$, we use the notation $[F] : \mathbb{I}\mathbb{R}^n \rightarrow \mathbb{I}\mathbb{R}^m$ to denote the equivalent interval input to output mapping.

There are several methods for constructing inclusion functions. For some functions, it is possible to compute the minimal inclusion function, which returns the tightest possible output for a given interval input bound (Harapanahalli et al., 2023, Theorem 2.2).

Proposition 3. (Minimal inclusion function). Given a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the unique, monotone and thin inclusion function returning the tightest bounds the image of f on $[\underline{x}, \bar{x}]$ is $F = \begin{bmatrix} \underline{F} \\ \bar{F} \end{bmatrix}$, where for every $i \in \{1, \dots, n\}$,

$$\underline{F}_i(\underline{x}, \bar{x}) = \inf_{x \in [\underline{x}, \bar{x}]} f_i(x), \quad \bar{F}_i(\underline{x}, \bar{x}) = \sup_{x \in [\underline{x}, \bar{x}]} f_i(x),$$

Denote this as the *minimal inclusion function* of f .

Computing the minimal inclusion function is not generally viable. Instead, we provide several computationally efficient approaches to construct inclusion functions using known inclusion functions as building blocks. First, we present the natural inclusion function, which is the simplest technique (Jaulin et al., 2001), Proof in (Harapanahalli et al., 2023, Theorem 2.3).

Proposition 4. (Natural inclusion function). Given a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, such that $f = f_1 \circ f_2 \circ \dots \circ f_\ell$ is the composition of functions/operators $\{f_i\}_{i=1}^\ell$ with (monotone/thin) inclusion functions $\{F_i\}_{i=1}^\ell$, the following is a (monotone/thin) inclusion function of f

$$F(\underline{x}, \bar{x}) = (F_1 \circ F_2 \circ \dots \circ F_\ell)(\underline{x}, \bar{x}).$$

Denote this as the $\{F_i\}_{i=1}^\ell$ -*natural inclusion function* of f .

While the natural inclusion function provides a general approach, it is often overly conservative. Instead, if the function is differentiable, one can use a bound on the first order Taylor expansion of the function, which may provide better results in practice (Jaulin et al., 2001).

Proposition 5. (Jacobian-based inclusion function).

Consider a differentiable function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, with an inclusion function J for the Jacobian matrix df , i.e., $df_x \in [J(\underline{x}, \bar{x})]$ for every $x \in [\underline{x}, \bar{x}]$. Then, any center $\hat{x} \in [\underline{x}, \bar{x}]$ induces a valid inclusion function as follows

$$[F(\underline{x}, \bar{x})] = [J(\underline{x}, \bar{x})](\underline{x}, \bar{x}) - \hat{x} + f(\hat{x}).$$

Denote this as the (J, \hat{x}) -*Jacobian-based inclusion function* of f .

By bounding each component of the vector input x as separate variables, we can further reduce the overconservatism of the Jacobian-based approach. The following definition helps build the mixed Jacobian-based inclusion function.

Definition 6. (Permutation). Given a dimension n , a *permutation* σ is a bijection of $\{1, \dots, n\}$ onto itself, characterized by a tuple of n unique integers $1 \leq \sigma(i) \leq$

n . For an n -permutation $\sigma = (\sigma(1), \dots, \sigma(n))$, the j -th *subpermutation* is $\sigma_j = (\sigma(1), \dots, \sigma(j))$. Define the *replacement* $x_{\sigma_j:y} \in \mathbb{R}^n$ such that

$$(x_{\sigma_j:y})_i := \begin{cases} y_i & i \in \sigma_j \\ x_i & i \notin \sigma_j \end{cases}.$$

Proposition 7. (Mixed Jacobian-based inclusion function). Consider a differentiable function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, with an inclusion function J for the Jacobian matrix df . Given a center $\hat{x} \in [\underline{x}, \bar{x}]$ and an n -permutation σ , let $M_\sigma^{\hat{x}} : \mathcal{T}_{\geq 0}^{2n} \rightarrow \mathcal{T}_{\geq 0}^{2(m \times n)}$ be defined such that for every $i = 1, \dots, n$, the $\sigma(i)$ -th column $[M_\sigma^{\hat{x}}(\underline{x}, \bar{x})]_{\sigma(i)} := [J(\hat{x}_{\sigma_i:\underline{x}}, \hat{x}_{\sigma_i:\bar{x}})]_{\sigma(i)}$. Then, the following statements hold:

(i) For every $x \in [\underline{x}, \bar{x}]$,

$$f(x) \in [M_\sigma^{\hat{x}}(\underline{x}, \bar{x})](x - \hat{x}) + f(\hat{x});$$

(ii) The function $F : \mathcal{T}_{\geq 0}^{2n} \rightarrow \mathcal{T}_{\geq 0}^{2m}$, defined by

$$[F(\underline{x}, \bar{x})] = [M_\sigma^{\hat{x}}(\underline{x}, \bar{x})](\underline{x}, \bar{x}) - \hat{x} + f(\hat{x}),$$

is an inclusion function for f , denoted as the (J, \hat{x}, σ) -mixed Jacobian-based inclusion function of f .

2.2 `immrax.inclusion` Module Implementation

In this subsection, we discuss the implementation of the interval analysis theory from the previous section in the submodule `immrax.inclusion`. In particular, the various inclusion functions are implemented as function transforms composable with any existing JAX transformations.

Interval Class In `immrax`, intervals are implemented in the `Interval` class, which is made up of two main attributes: `lower` and `upper`, which are `jax.numpy.ndarray` objects of the same shape and dtype representing the lower and upper bound of the interval. `Interval` is registered as a Pytree node, so JAX can internally handle any `Interval` object as if it were a standard JAX type. This implementation differs from the `IntervalBound` class from `jax_verify`, which instead requires the user to manually swap between `IntervalBound` objects and `JittableInputBound` objects as needed.

`immrax` provides several helper functions with input validation to safely construct and manipulate `Interval` objects. For example, `interval` creates an `Interval` from a lower and upper bound; `icentpert` creates an `Interval` from a center x_0 and a vector ϵ from the center as $[x_0 - \epsilon, x_0 + \epsilon]$; `ut2i` converts a `jax.Array` element of the upper triangle $\mathcal{T}_{\geq 0}^{2n}$ to its representation in \mathbb{R}^n ; and `i2lu`, `i2centpert`, `i2ut` perform the inverse operations respectively.

Inclusion Functions We provide minimal inclusion functions for basic `jax.lax` primitives (and applicable class operations), such as `add (+)`, `sub (-)`, `mul (*)`, `div (/)`, `pow (**)`, `sin`, `cos`, `sqrt`, `dot_general`. A full list of supported minimal inclusion functions can be found at the `immrax` documentation, and this list will continue to grow.

The most versatile transform that `immrax` provides is `natif`, which implements the natural inclusion function from Proposition 4. Given a function f acting on usual `jax.Array` inputs, defined as a composition of primitives with defined inclusion functions, `natif(f)` creates a new

function replacing each primitive, or each functional building block, with their corresponding minimal inclusion function counterparts. Internally, `immrax` builds these inclusion functions by tracing the original f into a `ClosedJaxpr`, the JAX internal representation of the pure functional inputs, outputs, and intermediate operations. Then, the inclusion function F is built by traversing the `ClosedJaxpr`, replacing `jax.Array` inputs with `Intervals` and replacing primitives with their inclusion function.

In addition to `natif`, `immrax` provides transforms for the Jacobian-based inclusion function from Proposition 5 as `jacif` and the mixed Jacobian-based inclusion function as `mjacif`. Internally, `immrax` builds these inclusion functions by composing `natif` with `jacfwd`, which creates an inclusion function for the Jacobian matrix of f . For example, a simple implementation of `jacif` for single vector input functions of the form $f(x)$ is

```
def jacif (f) :
    df = immrax.natif(jax.jacfwd(f))
    def F (x:Interval, xc:jax.Array) -> Interval :
        return df(x) @ (x - xc) + f(xc)
    return F
```

Note that the usage of `@` here calls `Interval.__matmul__`, which is explicitly defined in the `inclusion` module as

```
Interval.__matmul__ = immrax.natif(jnp.matmul)
```

In turn, `jnp.matmul` uses the `jax.lax.dot_general` primitive, for which the minimal inclusion function is provided. In practice, the true implementations of `jacif` and `mjacif` work for functions of any number of inputs—and one can even specify multiple centers and/or multiple permutations using `kwargs`, for which the minimum and maximum are taken accordingly.

Finally, all of these function transforms were carefully written to retain the ability to be composed with existing JAX transforms, such as `jax.jit` for JIT compilation, `jax.vmap` for parallelization, and `jax.grad/jax.jacfwd` for Automatic Differentiation. The derivative of an interval or with respect to an interval is not directly a well defined object—instead, one can take the derivative of a real-valued function of an interval output, *e.g.*, `(out.upper - out.lower)`, the objective function for the pendulum in Section 4, or simply the upper/lower bound.

The following example compares the inclusion functions generated from a call of `natif`, `jacif`, and `mjacif`.

Example 8. For the function $f(x_1, x_2) = ((x_1 + x_2)^2, x_1 + x_2 + 2x_1x_2)$, we compare the natural, Jacobian-based, and mixed Jacobian-based inclusion functions on the input $[-0.1, 0.1] \times [-0.1, 0.1]$, for $\hat{x} = 0$ and $\sigma = (1, 2)$:

```
f = lambda x : jnp.array([
    (x[0] + x[1])**2, x[0] + x[1] + 2*x[1]*x[2]])
Fnat = immrax.natif(f)
Fjac = immrax.jacif(f)
Fmix = immrax.mjacif(f)
x0 = immrax.icentpert(jnp.zeros(2), 0.1)
for F in [Fnat, Fjac, Fmix] :
    F(x0) # JIT Compile
    ret, times = utils.run_times(1000, F, x0)
```

F	Output	Average Runtime
Fnat	$[0, 0.04] \times [-0.22, 0.22]$	4.778×10^{-5}
Fjac	$[-0.08, 0.08] \times [-0.24, 0.24]$	8.611×10^{-5}
Fmix	$[-0.06, 0.06] \times [-0.24, 0.24]$	6.856×10^{-5}

Table 1. Inclusion function outputs and runtimes for Example 8

While the implementation of `mjacif` might seem more complicated upon first glance, after being Just-In-Time Compiled, the performance is better than the standard `jacif`. This is because a natural inclusion function computation takes on the order of at least twice as much as a standard computation (upper and lower bound)—and while the Jacobian-based approach builds a full matrix of interval components using natural inclusion functions, the mixed Jacobian approach is able to reduce the number of interval computations by fixing some elements to the center \hat{x} , reducing to a standard singleton computation.

3. EMBEDDING MODULE: MIXED MONOTONE EMBEDDING SYSTEMS IN JAX

In this section, we consider the theory and the implementation of continuous-time embedding systems in JAX, which provides an efficient and scalable method for bounding the reachable sets of dynamical systems.

3.1 Mixed Monotone Embedding Systems

Consider the nonlinear system

$$\dot{x} = f(x, w), \quad (1)$$

where $x \in \mathbb{R}^n$ is the state of the system and $w \in \mathbb{R}^q$ is the disturbance input to the system. Assume that F is an inclusion function for f . Then, F induces the associated embedding system

$$\begin{aligned} \dot{\underline{x}}_i &= \underline{E}_i(\underline{x}, \bar{x}, \underline{w}, \bar{w}) := \underline{F}_i(\underline{x}, \bar{x}_{i:\underline{x}}, \underline{w}, \bar{w}), \\ \dot{\bar{x}}_i &= \bar{E}_i(\underline{x}, \bar{x}, \underline{w}, \bar{w}) := \bar{F}_i(\underline{x}_{i:\bar{x}}, \bar{x}, \underline{w}, \bar{w}), \end{aligned} \quad (2)$$

where $i \in \{1, \dots, n\}$ and the new state $\begin{bmatrix} \underline{x} \\ \bar{x} \end{bmatrix} \in \mathcal{T}_{\geq 0}^{2n}$ evolves on the upper triangle, $\begin{bmatrix} \underline{w} \\ \bar{w} \end{bmatrix} \in \mathcal{T}_{\geq 0}^{2q}$. In the next proposition, we use a single trajectory of the embedding system to overapproximate the true reachable set of the system (1) (Jafarpour et al., 2023, Proposition 5).

Proposition 9. (Reachability via embedding). Consider the system (1) with an inclusion function F and its induced embedding system E (2). If $t \mapsto \begin{bmatrix} \underline{x}(t) \\ \bar{x}(t) \end{bmatrix}$ denotes the trajectory of E starting from initial condition $\begin{bmatrix} \underline{x}_0 \\ \bar{x}_0 \end{bmatrix} \in \mathcal{T}_{\geq 0}^{2n}$ at t_0 with disturbance $\begin{bmatrix} \underline{w} \\ \bar{w} \end{bmatrix} \in \mathcal{T}_{\geq 0}^{2q}$, then for every $t \geq t_0$, $x(t) \in [\underline{x}(t), \bar{x}(t)]$, where $t \mapsto x(t)$ is the trajectory of (1) from initial condition $x_0 \in [\underline{x}_0, \bar{x}_0]$.

The problem of evaluating an infinite number of trajectories for the reachable set is replaced with overapproximated interval bounds using a single trajectory of the embedding system, which provides an efficient, scalable approach for online reachable set estimation.

Remark 10. If F is a monotone inclusion function, then the induced embedding system E is a monotone dynamical system (Angeli and Sontag, 2003) with respect to the southeast partial order \leq_{SE} . If F is additionally a thin

inclusion function, then the approach is equivalent to the decomposition-based approach from Coogan and Arcak (2015). Thinness ensures that the decomposition function

$$d_i(x, \hat{x}, w, \hat{w}) := \begin{cases} \underline{F}_i(x, \hat{x}_{i:x}, w, \hat{w}) & x \leq \hat{x}, w \leq \hat{w} \\ \bar{F}_i(\hat{x}_{i:x}, x, \hat{w}, w) & \hat{x} \leq x, \hat{w} \leq w \end{cases}$$

satisfies $d(x, x, w, w) = f(x, w)$.

3.2 `immrax.embedding` Module Implementation

Similar to the usage of the inclusion function transforms, `immrax` provides transforms on dynamical systems, generating dynamical embedding systems. First, one defines a `System` object, which evolves on a state $x \in \mathbb{R}^n$ and defines the vector field $f : \mathbb{R}^n \times \dots \rightarrow \mathbb{R}^n$. Here, \dots represents any number of additional inputs to the system. Given a `System` object `sys`, and an inclusion function F for the dynamics `sys.f`, the transform `ifemb(sys, F)` returns an `EmbeddingSystem` object whose dynamics are constructed using (2) on the inclusion function F . For convenience, the transforms `natemb(sys)`, `jacemb(sys)`, and `mjacemb(sys)` automatically construct the `EmbeddingSystem` induced by the natural inclusion function (`natif(sys.f)`), the Jacobian-based inclusion function (`jacif(sys.f)`), and the Mixed Jacobian-based inclusion function (`mjacif(sys.f)`).

4. APPLICATIONS

We demonstrate the usage of `immrax` through reachability on a nonlinear vehicle controlled by a neural network and robust closed-loop control synthesis on a pendulum.³

4.1 GPU Acceleration for Neural Network Feedback Loops

In this example, we reimplement the interaction-aware first-order inclusion function from our previous work `ReachMM` (Jafarpour et al., 2023) using `immrax` with `jax_verify` for neural network verification. We compare to the nonlinear bicycle model (Jafarpour et al., 2023, Section VII.A)—and the full implementation details can be found in the corresponding Jupyter notebook in the `immrax` documentation. In Table 2, we compare the runtimes (after JIT compilation) across several different numbers of initial partitions on the CPU/GPU, as well as to a similar (hybrid mode) implementation in `ReachMM` (Jafarpour et al., 2023). The compiled `immrax` implementation sees substantial improvement in the runtime on the CPU, allowing it to perform the higher fidelity Tsit5 algorithm in a similar runtime as Euler integration on `ReachMM`. Additionally, while the GPU performance on a single initial partition is worse than the CPU, as the number of initial partitions increase, the benefits of the parallelization is clear as it can compute, e.g., 625 partitions in less than 10× the runtime of a single partition on the CPU.

4.2 Automatic Differentiation for Robust Optimal Control

In this example, we use Automatic Differentiation to solve a robust optimal control problem in the embedding space of a nonlinear pendulum. In particular, we provide

³ All experiments were performed on a computer running Kubuntu 22.01, with a Ryzen 5 5600X, Nvidia RTX 3070, and 32 GB of RAM.

# Part.	ReachMM (Euler)	immrax (Euler) CPU	immrax (Euler) GPU	immrax (tsit5) CPU	immrax (tsit5) GPU
$1^4 = 1$.0476	.0112	.0178	.0649	.0983
$2^4 = 16$.690	.143	.0207	.856	.112
$3^4 = 81$	3.44	.627	.0306	3.86	.187
$4^4 = 256$	11.0	1.44	.0489	8.87	.302
$5^4 = 625$	27.1	4.60	.095	27.9	.588
$6^4 = 1296$	55.8	11.1	.198	67.1	1.13

Table 2. Summary of average runtimes (over 10 runs) in seconds for the bicycle model on different numbers of initial partitions.

and discuss the relevant `immrax` code needed to build an objective function with robust constraints, automatically create and compile functions evaluating their gradients, Jacobians, and Hessians, and finally setup an IPOPT minimization problem to find a locally optimal solution.

Consider the dynamics of a forced, damped pendulum

$$ml^2\ddot{\theta} + b\dot{\theta} + mgl \sin \theta = \tau, \quad (3)$$

with $m = 0.15\text{kg}$, $l = 0.5\text{m}$, $b = 0.1\text{N} \cdot \text{m} \cdot \text{s}$, and $g = 9.81\text{m/s}^2$. The torque $\tau := (1 + w)u$, where $u \in \mathbb{R}$ is the desired torque input and $w \in [\underline{w}, \bar{w}] := [-0.02, 0.02]$ is a bounded multiplicative disturbance on the control input. We implement this as a 2-state system with $x := (\theta, \dot{\theta})$,

$$\dot{x} = f(x, u, w) = \begin{bmatrix} \frac{(1+w)u - bx_2}{ml^2} - \frac{g}{l} \sin x_1 \\ x_2 \end{bmatrix} \quad (4)$$

This is implemented in `immrax` as a `System`, with the specified dynamics written using `jax.numpy`.

```
import jax.numpy as jnp
import immrax
class Pendulum (immrax.System) :
    def __init__(self, m=0.15, l=0.5, b=0.1) :
        self.evolution = 'continuous'
        self.xlen = 2
        self.m = m; self.l = l; self.b = b
    def f (self, t, x, u, w) :
        return jnp.array([ x[1],
            (((1 + w[0])*u[0] - self.b*x[1]) /
            (self.m * self.l**2))
            - (g/self.l)*jnp.sin(x[0]) ])
sys = Pendulum()
```

In the preceding code, we specified two key properties for the `Pendulum`: `evolution` tells `immrax` that the system is in continuous time, which is needed to build the proper continuous embedding system, and `xlen` tells `immrax` the length of the state vector.

We seek to find a finite-time closed-loop optimal control policy $\pi : [0, T] \times \mathbb{R}^n \rightarrow \mathbb{R}$ to swing up the pendulum to an *a priori* safe region at the top. We consider linear feedback control policies of the form $\pi(t, x) := K(x(t) - x_{\text{nom}}(t)) + u_{\text{ff}}(t)$, where K is a time invariant linear closed-loop stabilizing term to help counter the disturbance, $u_{\text{ff}} : [0, T] \rightarrow \mathbb{R}$ is a feedforward control policy, and $x_{\text{nom}} : [0, T] \rightarrow \mathbb{R}^n$ is the nominal trajectory of the deterministic system under the feedforward control law u_{ff} with known disturbance mapping $w_{\text{nom}} : [0, T] \rightarrow \mathbb{R}$. The closed-loop system is thus

$$\dot{x} = f(x, \pi(t, x), w) = f^\pi(t, x, w). \quad (5)$$

Consider the following function

$$\begin{aligned} [F^\pi(t, \underline{x}, \bar{x}, \underline{w}, \bar{w})] := & ([M_x] + [M_u]K)([\underline{x}, \bar{x}] - x_{\text{nom}}(t)) \\ & + [M_w](\underline{w}, \bar{w}) - w_{\text{nom}}(t) \\ & + f(x_{\text{nom}}(t), u_{\text{ff}}(t), w_{\text{nom}}(t)), \end{aligned} \quad (6)$$

with $[M_x \ M_u \ M_w] := [M_\sigma^{\xi_{\text{nom}}(t)}(\underline{x}, \bar{x}, u_{\text{ff}}(t), u_{\text{ff}}(t), \underline{w}, \bar{w})]$, where M is defined as Proposition 7 for the map $\hat{f} : \mathbb{R}^{n+p+q} \rightarrow \mathbb{R}^n$ such that $\hat{f}((x, u, w)) := f(x, u, w)$, for some $(n + p + q)$ -permutation σ , and $\xi_{\text{nom}}(t) := (x_{\text{nom}}(t), u_{\text{ff}}(t), w_{\text{nom}}(t))$. This is a valid inclusion function for the closed-loop dynamics f^π (5) (proof uses Proposition 7 Part (i)). We use the `mjacM` transform to implement the inclusion function in `immrax` as follows

```
sys_mjacM = immrax.mjacM(sys.f)
def F (t, x, w, K, nominal) :
    tc, xc, uc, wc = nominal
    iuc = immrax.interval(uc)
    iK = immrax.interval(K)
    Mt, Mx, Mu, Mw = sys_mjacM(t, x,
        iuc, w, centers=(nominal,)) [0]
    return ((Mx + Mu @ iK) @ (x - xc)
        + Mw @ (w - wc) + sys.f(tc, xc, uc, wc))
embsys = immrax.ifemb(sys, F)
```

The final step creates the embedding system `embsys`, *i.e.* E from (2) induced by the inclusion function F^π (6). Using the embedding system, we would like to solve the following robust optimal control problem,

$$\begin{aligned} \min_{u_{\text{ff}}, K} \sum_{i=1}^N |u_{\text{ff}}(t_i)|^2 + \|K\|_F^2 + \sum_{i=1}^N \|\bar{x}(t_i) - \underline{x}(t_i)\|_2^2 \\ \text{s.t. } \underline{x}_f \leq \underline{x}(t_j), \quad \bar{x}(t_j) \leq \bar{x}_f, \quad j = N_e, \dots, N, \\ \underline{x}(0) = \bar{x}(0) = (0, 0), \\ \begin{bmatrix} \underline{x}(t_{i+1}) \\ \bar{x}(t_{i+1}) \end{bmatrix} = \begin{bmatrix} \underline{x}(t_i) \\ \bar{x}(t_i) \end{bmatrix} + \Delta t E(t_i, \underline{x}(t_i), \bar{x}(t_i), \underline{w}, \bar{w}), \end{aligned} \quad (7)$$

where the embedding dynamics E are discretized using Euler integration with step size Δt . The first and second terms of the objective are typical quadratic conditioning of the decision variables. The third term is a regularization factor intended to help curb the expansion of the gap between the upper and lower bound, which empirically helps the optimization problem converge to a feasible solution. Finally, in the inequality constraints, we require that the pendulum reach a target set $[\underline{x}_f, \bar{x}_f]$ and stay within these constraints for $t \in [t_{N_e}, t_N]$.

To setup the minimization problem in IPOPT, we first implement a function called `rollout_cl_embsys`, which uses `jax.lax.scan` to perform Euler integration on the dynamics for a given control sequence $(u_{\text{ff}}(t_i))_{i=1}^N$ and gain matrix K , returning the state sequence $(x_{t_i})_{i=1}^N$.

```
def rollout_cl_embsys (u) :
    u, K = split_u(u)
    def f_euler (xt, ut) :
        xtut, xnomt = xt
        xtutp1 = xtut + dt*embsys.E(0., xtut, w, K,
            (jnp.array([0.]), xnomt,
            jnp.array([ut]), jnp.array([0.])))
        xnomtp1 = xnomt + dt*sys.f(0., xnomt,
            jnp.array([ut]), jnp.array([0.])))
        return ((xtutp1, xnomtp1), xtutp1)
    _, x = jax.lax.scan(f_euler, (x0out, x0cent), u)
    return x
```

5. CONCLUSIONS

In this paper, we presented a differentiable and parallelizable framework for interval analysis in JAX. We applied this framework to two case studies demonstrating the toolbox’s potential for efficient closed-loop reachability analysis and robust controller design. Future work will involve certified robust training of neural network controllers.

REFERENCES

- Althoff, M. (2015). An introduction to CORA 2015. In *Proc. of the 1st and 2nd Workshop on Applied Verification for Continuous and Hybrid Systems*, 120–151. EasyChair.
- Angeli, D. and Sontag, E.D. (2003). Monotone control systems. *IEEE Transactions on Automatic Control*, 48(10), 1684–1698.
- Bogomolov, S., Forets, M., Frehse, G., Potomkin, K., and Schilling, C. (2019). JuliaReach: a toolbox for set-based reachability. In *Proc. of the 22nd International Conference on Hybrid Systems: Computation and Control*, 39–44.
- Bradbury, J., Frostig, R., Hawkins, P., Johnson, M.J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs. URL <http://github.com/google/jax>.
- Coogan, S. and Arcak, M. (2015). Efficient finite abstraction of mixed monotone systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, 58–67.
- Everett, M., Bunel, R., and Omidshafiei, S. (2023). Drip: Domain refinement iteration with polytopes for backward reachability analysis of neural feedback loops. *IEEE Control Systems Letters*, 7, 1622–1627.
- Fan, J., Huang, C., Chen, X., Li, W., and Zhu, Q. (2020). ReachNN*: A tool for reachability analysis of neural-network controlled systems. In *International Symposium on Automated Technology for Verification and Analysis*, 537–542. Springer.
- Harapanahalli, A., Jafarpour, S., and Coogan, S. (2023). A toolbox for fast interval arithmetic in numpy with an application to formal verification of neural network controlled system. In *2nd ICML Workshop on Formal Verification of Machine Learning*.
- Jafarpour, S., Harapanahalli, A., and Coogan, S. (2023). Efficient interaction-aware interval analysis of neural network feedback loops. *arXiv preprint arXiv:2307.14938*.
- Jaulin, L., Kieffer, M., Didrit, O., and Walter, É. (2001). *Applied Interval Analysis*. Springer London.
- Shen, K. and Scott, J.K. (2017). Rapid and accurate reachability analysis for nonlinear dynamic systems by exploiting model redundancy. *Computers & Chemical Engineering*, 106, 596–608. ESCAPE-26.
- Wang, Y., Zhou, W., Fan, J., Wang, Z., Li, J., Chen, X., Huang, C., Li, W., and Zhu, Q. (2023). Polar-Express: Efficient and precise formal reachability analysis of neural-network controlled systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

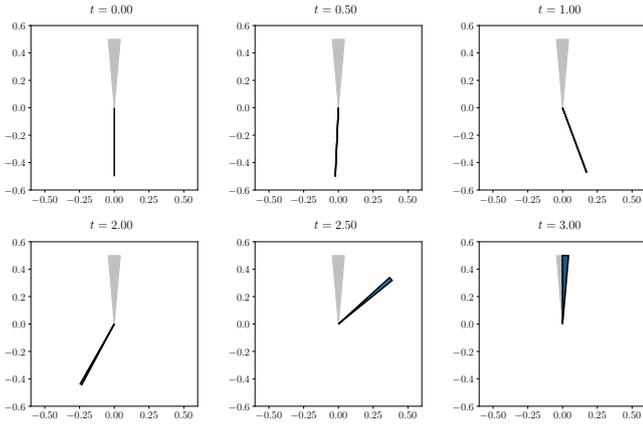


Fig. 1. The swinging trajectory of the pendulum embedding system induced by (6) controlled by the closed-loop control policy generated by the problem (7) is visualized for various time instances. The angle $[\underline{\theta}, \bar{\theta}]$ is represented as a wedge, where blue represents the interval of possible angles. The gray wedge represents the desired final state of the pendulum.

Note that the input $\mathbf{u} \in \mathbb{R}^{N+2}$ holds all decision variables, *i.e.*, $\mathbf{u} := (u_{t_1}, u_{t_2}, \dots, u_{t_N}, K_{1,1}, K_{1,2})$. To compute the nominal trajectory x_{nom} , we simulate the undisturbed system ($w_{\text{nom}} = 0$). Next, we implement the objective,

```
def obj (u) :
    x = rollout_cl_embsys(u)
    return (jnp.sum(u**2)
            + jnp.sum((x[:, :2] - x[:, :2])**2))
```

The final set is implemented as $\text{con_ineq}(\mathbf{u}) \geq 0$,

```
xf = immrax.icentpert([jnp.pi, 0.],
                    [10*(jnp.pi/360), .1])
xfl, xfu = immrax.i2lu(xf)
def con_ineq (u) :
    x = rollout_cl_embsys(u)
    return jnp.concatenate((
        (x[Ne:, :2] - xfl).reshape(-1),
        (xfu - x[Ne:, :2]).reshape(-1)))
```

Next, we use JAX’s autodiff transforms to automatically create functions to compute the objective’s gradient and Hessian, as well as the Jacobian and Hessian vector product of the constraints with respect to Lagrange multipliers.

```
obj_grad = jax.grad(obj)
obj_hess = jax.jacfwd(jacrev(obj))
con_ineq_jac = jax.jacfwd(con_ineq)
def con_ineq_hessvp (u, v) :
    def hessvp (u) :
        _, hvp = jax.vjp(con_ineq, u)
        return hvp(v)[0]
    return jax.jacrev(hessvp)(u)
```

After JIT compiling all of these functions, we use `cyipopt` with the MA57 linear solver to find a feasible solution. The setup and compilation steps took 147.91 seconds, and IPOPT was run for 100 iterations, taking 2.60 seconds and satisfying all constraints with a tolerance of -1.40×10^{-4} . The resulting trajectory is visualized in Figure 1.

Appendix A. ADDITIONAL FIGURES

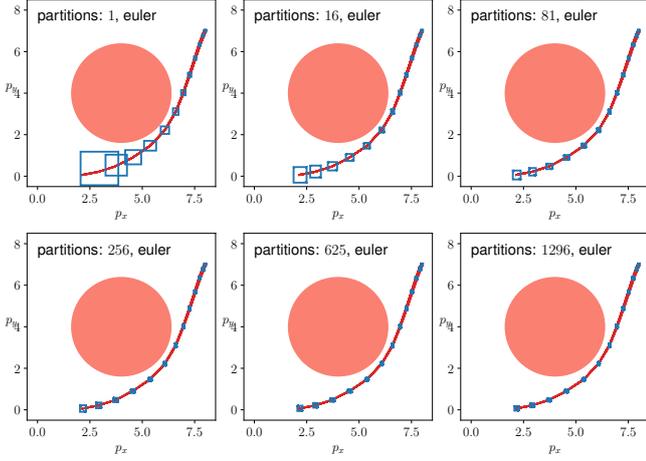


Fig. A.1. The reachable set over-approximations computed by simulating the embedding system from (Jafarpour et al., 2023) using Euler integration are visualized in light blue. The initial set $[7.95, 8.05] \times [6.95, 7.05] \times [-\frac{2\pi}{3} - 0.01, -\frac{2\pi}{3} + 0.01] \times [1.99, 2.01]$ is divided into different numbers of partitions. 100 Monte Carlo trajectories are pictured in dark red. In all cases, the vehicle is certified to avoid the obstacle pictured in light red, with varying degrees of accuracy to the true reachable set.

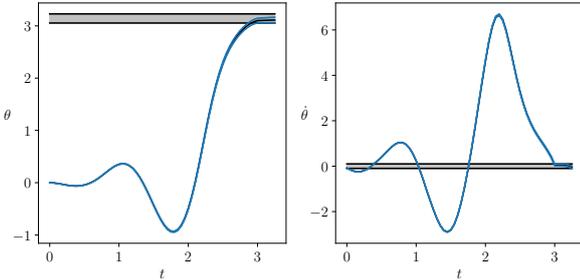


Fig. A.2. The swinging trajectory of the embedding system induced by (6) controlled by the closed-loop control policy generated by the optimization problem (7) is plotted versus time. **Left:** The angle $[\underline{\theta}, \bar{\theta}]$ vs. t in seconds (blue), with the terminal set constraint $\theta \in [\pi - \frac{10\pi}{360}, \pi + \frac{10\pi}{360}]$ (gray). **Right:** The angular velocity $[\underline{\dot{\theta}}, \bar{\dot{\theta}}]$ vs. t in seconds, with the terminal set constraint $\dot{\theta} \in [-0.1, 0.1]$ (gray).

Appendix B. PROOF OF CLOSED-LOOP PENDULUM INCLUSION FUNCTION

In this appendix, we prove that (6) is an inclusion function for the system (5). Consider the nonlinear system

$$\dot{x} = f(x, u, w), \quad (\text{B.1})$$

where $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^q \rightarrow \mathbb{R}^n$ is a parameterized vector field. Consider a piecewise continuous feedforward control curve $[0, T] \ni t \mapsto u_{\text{ff}}(t) \in \mathbb{R}^p$, a piecewise continuous disturbance trajectory $[0, T] \ni t \mapsto w_{\text{nom}}(t) \in [\underline{w}, \bar{w}]$ and some initial condition $x_0 \in \mathbb{R}^n$. Let $[0, T] \ni t \mapsto x_{\text{nom}}(t) \in \mathbb{R}^n$ denote the corresponding trajectory of (B.1)

from initial condition x_0 , under control mapping u_{ff} and disturbance mapping w_{nom} . For some gain matrix $K \in \mathbb{R}^{p \times n}$, define the feedback control policy $\pi : [0, T] \times \mathbb{R}^n \rightarrow \mathbb{R}^p$, such that $\pi(t, x) = K(x - x_{\text{nom}}(t)) + u_{\text{ff}}(t)$. Denote the closed-loop system as

$$\dot{x} = f(x, \pi(t, x), w) =: f^\pi(t, x, w). \quad (\text{B.2})$$

Fix some $t \in [0, T]$, $[\underline{x}, \bar{x}] \in \mathbb{R}^n$, and $[\underline{w}, \bar{w}] \in \mathbb{R}^q$. One can consider the vector field $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^q \rightarrow \mathbb{R}^n$ as a mapping $\hat{f} : \mathbb{R}^{n+p+q} \rightarrow \mathbb{R}^n$. Let $(x, u, w) := \xi$, $(\dot{x}, \dot{u}, \dot{w}) := \dot{\xi}$, and define $\underline{\xi} := (\underline{x}, u_{\text{ff}}(t), \underline{w})$ and $\bar{\xi} := (\bar{x}, u_{\text{ff}}(t), \bar{w})$. Then, for any center $\check{\xi} \in [\underline{\xi}, \bar{\xi}]$ and any permutation σ on $(n+p+q)$, Proposition 7(i) implies that for every $\xi \in [\underline{\xi}, \bar{\xi}]$,

$$\hat{f}(\xi) \in [M_\sigma^\xi(\underline{\xi}, \bar{\xi})](\xi - \check{\xi}) + \hat{f}(\check{\xi}). \quad (\text{B.3})$$

With the definition $[M_x^\xi \ M_u^\xi \ M_w^\xi] := [M_\sigma^\xi]$, the previous statement is equivalent to

$$\begin{aligned} f(x, u, w) \in & [M_x^\xi(\underline{x}, \bar{x}, u_{\text{ff}}(t), u_{\text{ff}}(t), \underline{w}, \bar{w})](x - \check{x}) \\ & + [M_u^\xi(\underline{x}, \bar{x}, u_{\text{ff}}(t), u_{\text{ff}}(t), \underline{w}, \bar{w})](u - u_{\text{ff}}(t)) \\ & + [M_w^\xi(\underline{x}, \bar{x}, u_{\text{ff}}(t), u_{\text{ff}}(t), \underline{w}, \bar{w})](w - \check{w}) \\ & + f(\check{x}, u_{\text{ff}}(t), \check{w}). \end{aligned} \quad (\text{B.4})$$

Let $\check{x} := x_{\text{nom}}(t)$, $\check{w} := w_{\text{nom}}(t)$, $u := \pi(t, x) = K(x - x_{\text{nom}}(t)) + u_{\text{ff}}(t)$, and $\xi_{\text{nom}}(t) := (x_{\text{nom}}(t), u_{\text{ff}}(t), w_{\text{nom}}(t))$. If $x_{\text{nom}}(t) \in [\underline{x}, \bar{x}]$ and $w_{\text{nom}}(t) \in [\underline{w}, \bar{w}]$,

$$\begin{aligned} f^\pi(t, x, w) \in & [M_x^{\xi_{\text{nom}}(t)}(\underline{x}, \bar{x}, u_{\text{ff}}(t), u_{\text{ff}}(t), \underline{w}, \bar{w})](x - x_{\text{nom}}(t)) \\ & + [M_u^{\xi_{\text{nom}}(t)}(\underline{x}, \bar{x}, u_{\text{ff}}(t), u_{\text{ff}}(t), \underline{w}, \bar{w})](K(x - x_{\text{nom}}(t))) \\ & + [M_w^{\xi_{\text{nom}}(t)}(\underline{x}, \bar{x}, u_{\text{ff}}(t), u_{\text{ff}}(t), \underline{w}, \bar{w})](w - w_{\text{nom}}(t)) \\ & + f(x_{\text{nom}}(t), u_{\text{ff}}(t), w_{\text{nom}}(t)). \end{aligned} \quad (\text{B.5})$$

Combining terms,

$$\begin{aligned} f^\pi(t, x, w) \in & ([M_x^{\xi_{\text{nom}}(t)}(\underline{x}, \bar{x}, u_{\text{ff}}(t), u_{\text{ff}}(t), \underline{w}, \bar{w})] \\ & + [M_u^{\xi_{\text{nom}}(t)}(\underline{x}, \bar{x}, u_{\text{ff}}(t), u_{\text{ff}}(t), \underline{w}, \bar{w})]K)(x - x_{\text{nom}}(t)) \\ & + [M_w^{\xi_{\text{nom}}(t)}(\underline{x}, \bar{x}, u_{\text{ff}}(t), u_{\text{ff}}(t), \underline{w}, \bar{w})](w - w_{\text{nom}}(t)) \\ & + f(x_{\text{nom}}(t), u_{\text{ff}}(t), w_{\text{nom}}(t)). \end{aligned} \quad (\text{B.6})$$

Thus, as long as $x_{\text{nom}}(t) \in [\underline{x}, \bar{x}]$ and $w_{\text{nom}}(t) \in [\underline{w}, \bar{w}]$,

$$\begin{aligned} [F^\pi(t, \underline{x}, \bar{x}, \underline{w}, \bar{w})] = & ([M_x^{\xi_{\text{nom}}(t)}(\underline{x}, \bar{x}, u_{\text{ff}}(t), u_{\text{ff}}(t), \underline{w}, \bar{w})] \\ & + [M_u^{\xi_{\text{nom}}(t)}(\underline{x}, \bar{x}, u_{\text{ff}}(t), u_{\text{ff}}(t), \underline{w}, \bar{w})]K)([\underline{x}, \bar{x}] - x_{\text{nom}}(t)) \\ & + [M_w^{\xi_{\text{nom}}(t)}(\underline{x}, \bar{x}, u_{\text{ff}}(t), u_{\text{ff}}(t), \underline{w}, \bar{w})](\underline{w}, \bar{w}) - w_{\text{nom}}(t)) \\ & + f(x_{\text{nom}}(t), u_{\text{ff}}(t), w_{\text{nom}}(t)) \end{aligned} \quad (\text{B.7})$$

is an inclusion function of the closed-loop vector field f^π .

In particular, note that the condition $x_{\text{nom}}(t) \in [\underline{x}, \bar{x}]$ is satisfied when (B.7) is used to build an embedding system, as long as the initial condition $x_0 \in [\underline{x}_0, \bar{x}_0]$ used as the initial condition of the embedding system trajectory, and for every $t \in [0, T]$, $w_{\text{nom}}(t) \in [\underline{w}, \bar{w}]$ used as the disturbance bounds in the embedding system.