



PIETOOLS 2021a: User Manual

Sachin Shivakumar, Amritam Das, Declan Jagt and Matthew Peet

June 3, 2021

Copyrights and license information

PIETOOLS is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

Notation

\mathbb{R}	Set of real numbers $(-\infty, \infty)$
$\partial_s^i \mathbf{x}$	$\frac{\partial^i \mathbf{x}}{\partial s^i}$ where s is in a compact subset of \mathbb{R}
$\dot{\mathbf{x}}$	$\frac{\partial \mathbf{x}}{\partial t}$ where t is in $[0, \infty)$
$L_2^n[a, b]$	Set of lebesgue integrable functions from $[a, b] \rightarrow \mathbb{R}^n$
$RL^{m,n}[a, b]$	$\mathbb{R}^m \times L_2^n[a, b]$
$H_k^n[a, b]$	$\{f \in L_2^n[a, b] \mid \partial_s^i f \in L_2^n[a, b] \forall i \leq k\}$
$0_{m \times n}$	Zero matrix of dimension $m \times n$
0_n	Zero matrix of dimension $n \times n$
I_n	Identity matrix of dimension $n \times n$
Δ_a	Dirac operator on $f : C \rightarrow X$, $\Delta_a(f) = f(a)$, for $a \in C$
$\mathcal{B}(X, Y)$	Space of bounded linear operators from X to Y

Contents

1	About PIETOOLS	7
I	Theory of Partial Integral Operators and Partial Integral Equations	8
2	Introduction	9
2.1	Motivation	9
2.2	A simple, yet useful application...	10
3	Partial Integral Operators	11
3.1	Definition	11
3.2	Algebra Of Partial Integral Operators	11
3.2.1	Addition Of PI Operators	12
3.2.2	Composition Of PI operators	12
3.2.3	Adjoint Of A PI Operator	13
3.3	Properties of Partial Integral Operators	14
3.3.1	Differentiation of a PI operator	14
3.3.2	Dirac operator on a PI operator	14
3.4	Matrix Parametrization of Positive Definite PI operators	15
3.5	Inverse of a 4-PI operator	16
4	Linear Partial Integral Inequalities	17
4.1	A General Class of LPI problems	17
4.2	Application of LPIs	17
5	Partial Integral Equations	19
5.1	Definition	19
5.2	Application of PIE framework	19
5.2.1	DDE to PIE	20
5.2.2	ODE-PDE to PIE	20
II	PIETOOLS: A computational toolbox	23
6	PIETOOLS 2021a	24

6.1	Getting Started	24
6.1.1	Installation	24
6.2	Before you Start	25
6.3	Demos of PDEs and Integral Operators in PIETOOLS	25
7	opvar: A MATLAB class for PI operators	26
7.1	Initialization and Assignment	27
7.2	opvar class methods	27
7.2.1	+	27
7.2.2	*	28
7.2.3	'	28
7.2.4	Concatenation	28
7.2.5	isvalid	28
7.2.6	inv_opvar	29
7.2.7	Conditional statements involving PI operators	29
7.2.8	subsref or sliced indexing	29
7.3	Additional methods for opvar objects	29
8	PIETOOLS Functions: Setting up LPIs and more	31
8.1	Optimization problem structure	31
8.2	4-PI Decision Variables	31
8.2.1	poslpivar	32
8.2.2	lpivar	32
8.2.3	getsol_lpivar	33
8.3	4-PI Constraints	33
8.3.1	lpi_eq	33
8.3.2	lpi_ineq	33
III	PIEs: Representation, Analysis, Control, and Simulation	34
9	Representation of PIEs	35
9.1	Elements of the PIE data structure	35
9.2	Initializing and modifying a PIE data structure	36
10	LPIs for Analysis of PIEs	38
11	LPIs for Optimal Estimation of PIEs	40
12	LPIs for Optimal Control of PIEs	41
IV	PDEs: Representation, Analysis, Control, and Simulation	42
13	A GUI for Defining PDEs	43
13.1	Step 1: Define States, Outputs and Inputs	44

13.2	Step 2: Select an Equation to Add a Term	46
13.3	Step 3: (Optional) Add or Remove BC	50
13.4	Step 4-5: Parse PDE Parameters and Convert Them to PIE	50
14	The PDE Input Formats	51
14.1	The Batch Input Format	51
14.1.1	Dynamics of the PDE	52
14.1.2	Dynamics of the ODE	52
14.1.3	The Output Equation	52
14.1.4	The Boundary Conditions	53
14.2	The Term-Based Input Format	54
14.2.1	The Variables and their Domain	54
14.2.2	The Size of the Problem	55
14.2.3	The ODE	56
14.2.4	The PDE	57
14.2.5	The Boundary Conditions	61
14.3	Initializing your PDE	62
15	Library of PDE Example Problems	63
16	Analysis, Control, and Estimation of PDEs	66
16.1	Defining the PDE object	66
16.2	Convert to a PIE	66
16.3	Choose Executive Mode	67
16.4	Specify Settings and SDP Solver	67
16.5	Solving the problem, interpreting the output, and implementing the controller	68
V	Delayed Systems: Representation, Analysis and Control	70
17	Constructing and Representing Systems with Delay	71
17.1	The Delay Differential Equation (DDE) Format	71
17.1.1	Initializing a DDE Data structure	73
17.2	Input of Neutral Type Systems	73
17.2.1	Initializing a NDS Data structure	73
17.3	The Differential Difference Equation (DDF) Format	73
17.3.1	Initializing a DDF Data structure	74
18	Converting between DDEs, NDSs, DDFs, and PIEs	76
18.1	DDF to PIE	76
18.1.1	Minimal DDF Realization of a DDF	76
18.1.2	Converting a DDF to a PIE	76
18.2	DDE to DDF or PIE	77
18.2.1	Minimal DDF and PIE Realizations of DDEs	77
18.2.2	DDE direct to PIE [NOT RECOMMENDED!]	77
18.3	NDS to DDF or PIE	77

18.3.1	Converting NDS to DDF	78
19	DDE, NDS, and DDF: Library of Examples	79
19.1	DDE Examples	79
19.2	NDS and DDF Examples	79
20	DDEs, NDSs, DDFs: Stability Analysis and Controller Synthesis	81
20.1	Input DDE or DDF Representation	81
20.2	Convert to a PIE	81
20.3	Choose Executive Mode	82
20.4	Specify Settings and SDP Solver	82
20.5	Solving the problem, interpreting the output, and implementing the controller	83
VI	List of Files, Functions and Scripts	85
21	PIETOOLS Scripts: Initialize Systems, Convert To PIEs And More	86
21.1	converters	86
21.1.1	initialize_PIETOOLS_DDE(DDE)	86
21.1.2	initialize_PIETOOLS_DDF(DDF)	87
21.1.3	initialize_PIETOOLS_NDS(NDS)	87
21.1.4	initialize_PIETOOLS_PDE_batch(PDE)	87
21.1.5	initialize_PIETOOLS_PDE_terms(PDE)	87
21.1.6	convert_PIETOOLS_DDE(DDE)	87
21.1.7	convert_PIETOOLS_DDF(DDF)	88
21.1.8	convert_PIETOOLS_NDS2DDF(NDS)	88
21.1.9	convert_PIETOOLS_PDE_batch(PDE)	88
21.1.10	convert_PIETOOLS_PDE_terms(PDE)	88
21.1.11	convert_PIETOOLS_PDE_batch2terms(PDE)	88
21.2	executives	88
21.2.1	executive_PIETOOLS_stability(PIE, settings)	89
21.2.2	executive_PIETOOLS_stability_dual(PIE, settings)	89
21.2.3	executive_PIETOOLS_Hinf_gain(PIE, settings)	89
21.2.4	executive_PIETOOLS_Hinf_gain_dual(PIE, settings)	89
21.2.5	executive_PIETOOLS_Hinf_estimator(PIE, settings)	89
21.2.6	executive_PIETOOLS_Hinf_control(PIE, settings)	89
21.3	settings	90
22	Troubleshooting	91
22.1	Troubleshooting: Installation	91
22.2	Troubleshooting: Solving LPIs	92
22.3	Contact Details	92

Chapter 1

About PIETOOLS

PIETOOLS is a free MATLAB toolbox for manipulating Partial Integral (PI) operators and solving Linear PI Inequalities (LPIs) which are convex optimization problems involving PI variables and PI constraints. PIETOOLS can be used to:

- define 3-PI or 4-PI operators
- declare 3-PI or 4-PI operators variables (postive semidefinite or indefinite)
- add operator inequality constraints
- solve LPI optimization problems

The interface is inspired by YALMIP and the program structure is based on that used by SOSTOOLS. By default the LPIs are solved using SeDuMi, however, the toolbox also supports use of other SDP solvers such as Mosek, sdpt3 and sdpnal.

To install and run PIETOOLS, you need:

- MATLAB version 2014a or later (we recommend MATLAB 2020a or higher. Please note some features of PIETOOLS, for example PDE input GUI, might be unavailable if an older version of MATLAB is used)
- The current version of the MATLAB Symbolic Math Toolbox (This is installed in most default versions of Matlab.)
- An SDP solver (SeDuMi is included in the installation script.)

Part I

Theory of Partial Integral Operators and Partial Integral Equations

Chapter 2

Introduction

In this chapter, we briefly talk about the need for a computational tool that handle PI operators and lightly touch upon, without going into details of solution, some popular applications.

2.1 Motivation

Semidefinite programming (SDPs) are a class of optimization problems which involve optimization of a linear objective over the cone of positive semidefinite (PSD) matrices. The development of efficient interior-point methods for semi definite programming (SDPs) problems made LMIs a powerful tool in modern control theory. As Doyle stated in [2], LMIs played a central role in postmodern control theory akin to the role played by graphical methods like Bode, Nyquist plots etc in classical control theory. However, most of the application of LMI techniques were restricted to finite dimensional systems, until sum-of-squares method came into limelight. The sum-of-squares (SOS) optimization methods found application in control theory, for example searching for Lyapunov functions or finding bounds on singular values. SOS polynomials were also used in constructing relaxations to some optimization problems with boolean decision variables. This gave rise to many toolboxes such as SOS-TOOLS [5], SOSOPT [7] etc. that can handle SOS polynomials in MATLAB. However, unlike the use of LMIs for linear ODEs, SOS methods could not be used for analysis and control of PDEs without ad-hoc interventions. For example, in order to search for a Lyapunov function that proves stability of a PDE one would usually hit a roadblock in the form of boundary conditions which are typically resolved by using integration by parts, Poincare inequality, Hölder's Inequality etc.

In an ideal world, we would prefer to define a PDE, specify the boundary conditions and let a computational tool take care of the rest. To resolve this problem, either we teach a computer to perform these "ad-hoc" interventions (Artificial Intelligence?) or come up with a method that does not require such interventions to begin with. To achieve the latter, we developed the Partial Integral Equation (PIE) representation of PDEs, which is an algebraic representation of dynamical systems using Partial Integral (PI) operators. The development of PIE representation led to a framework that can extend LMI-based methods to infinite-dimensional systems. The PIE representation encompasses a broad class

of distributed parameter systems and is algebraic - eliminating the use of boundary conditions and continuity constraints [8], [1]. The development of the toolbox, PIETOOLS, that can solve optimization problems involving these PI operators was an obvious consequence of the PIE representation.

2.2 A simple, yet useful application...

Since PIETOOLS was developed to tackle problems that earlier methods could not directly solve, let's take a brief peek into what PIETOOLS can do. Recall, in case of a linear ODE given by

$$\dot{x}(t) = Ax(t), \quad t \geq 0, x(0) = x_0$$

one can prove stability by finding a positive definite matrix P such that $A^T P + PA$ is negative semidefinite. Feasibility of such LMI constraints can easily be tested by using an SDP solver. Now consider a similar problem on infinite dimensional space. Let

$$\dot{\mathbf{x}}(t, s) = \mathcal{A}\mathbf{x}(t, s), \quad t \geq 0, s \in (a, b), \mathbf{x}(0) = \mathbf{x}_0 \in D(\mathcal{A})$$

where $\mathcal{A} = A_0 + A_1 \partial_s + A_2 \partial_s^2$ is a differential operator in spatial variable s and

$$D(\mathcal{A}) := \left\{ \mathbf{x} \in H^2[a, b] \mid B \begin{bmatrix} \mathbf{x}(a) \\ \mathbf{x}(b) \\ \mathbf{x}_s(a) \\ \mathbf{x}_s(b) \end{bmatrix} = 0 \right\}.$$

The solution space is a subspace of $H^2[a, b]$, specified by the boundary conditions given by B . To test for stability using LMIs or SOS, one would try to find a Lyapunov function

$$V(t) = \int_a^b \mathbf{x}(t, s)^T P(s) \mathbf{x}(t, s) ds$$

such that time derivative $\dot{V}(t) \leq 0$ for all time t and initial conditions \mathbf{x}_0 . However, this is not enough since the solution space is not entire $H^2[a, b]$ and some ad-hoc steps, such as integration by parts and Poincare/Wirtinger's inequality, must be taken to restrict the solution space to $D(\mathcal{A})$. With **PIETOOLS**, users can just define A_i , B and then let the solver take care of the rest. In other words, PIETOOLS can directly search for a lyapunov function $V(t) = \langle \mathbf{x}(t), \mathcal{P}\mathbf{x}(t) \rangle$ which proves the stability. We will discuss the particulars about how this is done in coming chapters. For now, just accept that we are *one-step closer to an ideal world* described in the previous section.

Chapter 3

Partial Integral Operators

Any operator that maps from a finite dimensional space to another finite dimensional space is bounded and can be written as a matrix. On infinite dimensional inner product spaces, such as Hilbert spaces, such a generalization does not exist, however, PI operators do provide a class of parameterizable bounded operators. Even though parametrization using PI operators is not be exhaustive, it is useful in solving many problems as will be seen in later chapters.

3.1 Definition

PI operator is a map from $RL^{m,n}$ to $RL^{p,q}$ and can be broadly categorized as 4-PI and 3-PI. 4-PI operators have 6 parameters whereas 3-PI have 3 parameters.

Definition 1. (4-PI:) A 4-PI operator is a bounded linear operator between that maps from $RL^{m,n}$ to $RL^{p,q}$ and is parametrized as

$$\mathcal{P} \begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix} \begin{bmatrix} x \\ \mathbf{y} \end{bmatrix} (s) = \begin{bmatrix} Px + \int_a^b Q_1(s)\mathbf{y}(s)ds \\ Q_2(s)x + \mathcal{P}_{\{R_i\}}\mathbf{y}(s) \end{bmatrix} \quad (3.1)$$

where $P \in \mathbb{R}^{p \times m}$ is a matrix, $Q_1 : [a, b] \rightarrow \mathbb{R}^{p \times n}$, $Q_2 : [a, b] \rightarrow \mathbb{R}^{q \times m}$ are bounded integrable functions and $\mathcal{P}_{\{R_i\}} : L_2^n \rightarrow L_2^q$ is a 3-PI operator of the form

$$(\mathcal{P}_{\{R_i\}}\mathbf{x})(s) := R_0(s)\mathbf{x}(s) + \int_a^s R_1(s, \theta)\mathbf{x}(\theta)d\theta + \int_s^b R_2(s, \theta)\mathbf{x}(\theta)d\theta.$$

3.2 Algebra Of Partial Integral Operators

The following subsections show that the set of PI operators is closed under addition (Lemma 2) and composition (Lemma 4). Furthermore, the adjoint of a PI operator is also a PI operator (Lemma 6). The proofs for the results are omitted as it is not the purpose of this article, however, readers can refer to [8] for proofs.

3.2.1 Addition Of PI Operators

In this subsection, we show that PI operators are closed under addition operation. Further, we show that the set of PI operators parametrized by matrix valued polynomials is also closed under addition operation.

Lemma 2. *Suppose $A, L \in \mathbb{R}^{m \times p}$ and $B_1, M_1 : [a, b] \rightarrow \mathbb{R}^{m \times q}$, $B_2, M_2 : [a, b] \rightarrow \mathbb{R}^{n \times p}$, $C_0, N_0 : [a, b] \rightarrow \mathbb{R}^{n \times q}$, $C_i, N_i : [a, b] \times [a, b] \rightarrow \mathbb{R}^{n \times q}$, for $i \in \{1, 2\}$, are bounded functions. P, Q_1, Q_2 and R_i , for $j \in \{0, 1, 2\}$ are such that*

$$P = A + L, \quad Q_i = B_i + M_i, \quad R_i = C_i + N_i,$$

if and only if

$$\mathcal{P} \begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix} = \mathcal{P} \begin{bmatrix} A, & B_1 \\ B_2, & \{C_i\} \end{bmatrix} + \mathcal{P} \begin{bmatrix} L, & M_1 \\ M_2, & \{N_i\} \end{bmatrix}.$$

Corollary 3. *Suppose $A, L \in \mathbb{R}^{m \times p}$ and $B_1, M_1 : [a, b] \rightarrow \mathbb{R}^{m \times q}$, $B_2, M_2 : [a, b] \rightarrow \mathbb{R}^{n \times p}$, $C_0, N_0 : [a, b] \rightarrow \mathbb{R}^{n \times q}$, $C_i, N_i : [a, b] \times [a, b] \rightarrow \mathbb{R}^{n \times q}$, for $i \in \{1, 2\}$, are matrix valued polynomials. If*

$$\mathcal{P} \begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix} = \mathcal{P} \begin{bmatrix} A, & B_1 \\ B_2, & \{C_i\} \end{bmatrix} + \mathcal{P} \begin{bmatrix} L, & M_1 \\ M_2, & \{N_i\} \end{bmatrix},$$

then $\mathcal{P} \begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix}$ is parametrized by matrix valued polynomials.

To see PIETOOLS implementation, check Section 7.2.1.

3.2.2 Composition Of PI operators

In this subsection, we show that 4-PI operators, with compatible dimensions, can be composed to get an equivalent PI operator. Furthermore, since PI operators are bounded and linear, the composition operation is associative and hence the set of PI operators forms an associative algebra. We also show that PI operators parametrized by matrix valued polynomials form an associative subalgebra.

Lemma 4. *For any matrices $A \in \mathbb{R}^{m \times k}$, $P \in \mathbb{R}^{k \times p}$ and integrable functions $B_1 : [a, b] \rightarrow \mathbb{R}^{m \times l}$, $Q_1 : [a, b] \rightarrow \mathbb{R}^{k \times q}$, $B_2 : [a, b] \rightarrow \mathbb{R}^{n \times k}$, $Q_2 : [a, b] \rightarrow \mathbb{R}^{l \times p}$, $C_0 : [a, b] \rightarrow \mathbb{R}^{n \times l}$, $R_0 : [a, b] \rightarrow \mathbb{R}^{l \times q}$, $C_i : [a, b]^2 \rightarrow \mathbb{R}^{n \times l}$, $R_i : [a, b]^2 \rightarrow \mathbb{R}^{l \times q}$, for $i \in \{1, 2\}$, the following identity holds.*

$$\mathcal{P} \begin{bmatrix} \hat{P}, & \hat{Q}_1 \\ \hat{Q}_2, & \{\hat{R}_i\} \end{bmatrix} = \mathcal{P} \begin{bmatrix} A, & B_1 \\ B_2, & \{C_i\} \end{bmatrix} \mathcal{P} \begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix}$$

where

$$\begin{aligned} \hat{P} &= AP + \int_a^b B_1(s)Q_2(s)ds, \quad \hat{R}_0(s) = C_0(s)R_0(s), \\ \hat{Q}_1(s) &= AQ_1(s) + B_1(s)R_0(s) + \int_s^b B_1(\eta)R_1(\eta, s)d\eta + \int_a^s B_1(\eta)R_2(\eta, s)d\eta, \end{aligned}$$

$$\begin{aligned}
\hat{Q}_2(s) &= B_2(s)P + C_0(s)Q_2(s) + \int_a^s C_1(s, \eta)Q_2(\eta)d\eta + \int_s^B C_2(s, \eta)Q_2(\eta)d\eta, \\
\hat{R}_1(s, \eta) &= B_2(s)Q_1(\eta) + C_0(s)R_1(s, \eta) + C_1(s, \eta)R_0(\eta) + \int_a^\eta C_1(s, \theta)R_2(\theta, \eta)d\theta \\
&\quad + \int_\eta^s C_1(s, \theta)R_1(\theta, \eta)d\theta + \int_s^b C_2(s, \theta)R_1(\theta, \eta)d\theta, \\
\hat{R}_2(s, \eta) &= B_2(s)Q_1(\eta) + C_0(s)R_2(s, \eta) + C_2(s, \eta)R_0(\eta) + \int_a^s C_1(s, \theta)R_2(\theta, \eta)d\theta \\
&\quad + \int_s^\eta C_2(s, \theta)R_2(\theta, \eta)d\theta + \int_\eta^b C_2(s, \theta)R_1(\theta, \eta)d\theta.
\end{aligned}$$

Corollary 5. Suppose $\mathcal{P} \begin{bmatrix} A, & B_1 \\ B_2, & \{C_i\} \end{bmatrix} : RL^{l,k} \rightarrow RL^{m,n}$ and $\mathcal{P} \begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix} : RL^{p,q} \rightarrow RL^{l,k}$ are PI operators parametrized by matrix valued polynomials. Then $\mathcal{P} \begin{bmatrix} \hat{P}, & \hat{Q}_1 \\ \hat{Q}_2, & \{\hat{R}_i\} \end{bmatrix}$ is a PI operator parametrized by matrix valued polynomials where

$$\mathcal{P} \begin{bmatrix} \hat{P}, & \hat{Q}_1 \\ \hat{Q}_2, & \{\hat{R}_i\} \end{bmatrix} = \mathcal{P} \begin{bmatrix} A, & B_1 \\ B_2, & \{C_i\} \end{bmatrix} \mathcal{P} \begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix}.$$

To see PIETOOLS implementation, check Section 7.2.2.

3.2.3 Adjoint Of A PI Operator

On the set of PI operators, we define the involution to be the adjoint operator with respect to RL -inner product. First, we provide the formulae for the adjoint of a PI operator and then show that the set of PI operators is closed under adjoint operation and hence forms a $*$ -algebra.

Lemma 6. Suppose $P \in \mathbb{R}^{m \times p}$ and $Q_1 : [a, b] \rightarrow \mathbb{R}^{m \times q}$, $Q_2 : [a, b] \rightarrow \mathbb{R}^{n \times p}$, $R_0 : [a, b] \rightarrow \mathbb{R}^{n \times q}$, $R_1, R_2 : [a, b]^2 \rightarrow \mathbb{R}^{n \times q}$ are bounded functions. Then for any $x \in RL^{m,n}$, $y \in RL^{p,q}$.

$$\left\langle \mathcal{P} \begin{bmatrix} \hat{P}, & \hat{Q}_1 \\ \hat{Q}_2, & \{\hat{R}_i\} \end{bmatrix} x, y \right\rangle_{RL} = \left\langle x, \mathcal{P} \begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix} y \right\rangle_{RL}, \quad (3.2)$$

where

$$\begin{aligned}
\hat{P} &= P^\top, & \hat{Q}_1(s) &= Q_2^\top(s), & \hat{Q}_2(s) &= Q_1^\top(s), \\
\hat{R}_0(s) &= R_0^\top(s), & \hat{R}_1(s, \eta) &= R_2^\top(\eta, s), & \hat{R}_2(s, \eta) &= R_1^\top(\eta, s).
\end{aligned} \quad (3.3)$$

To see PIETOOLS implementation, check Section 7.2.3.

For, bounded linear maps, addition and adjoint operations commute. If $\mathcal{A}, \mathcal{B} : RL^{m,n} \rightarrow RL^{p,q}$ are PI operators, then $(\mathcal{A} + \mathcal{B})^* = \mathcal{A}^* + \mathcal{B}^*$ [6]. Furthermore, for PI operators $\mathcal{A} : RL^{l,k} \rightarrow RL^{m,n}$ and $\mathcal{B} : RL^{p,q} \rightarrow RL^{l,k}$, $\mathcal{A}\mathcal{B}^* = \mathcal{B}^*\mathcal{A}^*$ [6]. We also note, without proof, that set of PI operators are closed under scalar multiplication. Hence, the set of PI operators

forms a $*$ -algebra with binary operations of addition and composition. Additionally, note that the transpose of a matrix valued polynomial is a matrix valued polynomial. Consequently, we can conclude that the adjoint of a PI operators parameterized by matrix valued functions is also a PI operator with matrix valued functions as its parameters. Thus, the subset of PI operators that are parametrized by matrix valued polynomials form a $*$ -subalgebra.

3.3 Properties of Partial Integral Operators

We list some properties in this section, such as derivative of PI operator and Dirac delta operator on PI operator, that become useful later on in Chapter 5 for reformulation of systems in PIE form.

3.3.1 Differentiation of a PI operator

For a general PI operator $\mathcal{P} : \mathbb{R}^m \times L_2^n \rightarrow \mathbb{R}^p \times L_2^q$, differentiation with respect to spatial variable s may not always be well-defined. However, in this subsection, we will show that if the domain $D(\mathcal{P}) \subseteq \mathbb{R}^m \times H_1^n$, then the differentiation is a well defined operation. Note that, the parameters of a PI operator are matrix valued polynomials and hence always differentiable. Hence differentiability of $\mathcal{P}\mathbf{x}$ depends entirely on differentiability of \mathbf{x} .

Lemma 7. Suppose $\mathcal{P} \begin{bmatrix} P, Q_1 \\ Q_2, \{R_i\} \end{bmatrix} : \mathbb{R}^m \times H_1^n \rightarrow \mathbb{R}^p \times L_2^q$, and define $\partial_s \mathcal{P} \begin{bmatrix} P, Q_1 \\ Q_2, \{R_i\} \end{bmatrix} : \mathbb{R}^m \times H_1^n \times L_2^n \rightarrow \mathbb{R}^p \times L_2^q$ as

$$\partial_s \mathcal{P} \begin{bmatrix} P, Q_1 \\ Q_2, \{R_i\} \end{bmatrix} = \mathcal{P} \begin{bmatrix} 0, 0 \\ \bar{Q}_2, \{\bar{R}_i\} \end{bmatrix} \quad (3.4)$$

where $\bar{Q}_2(s) = \partial_s Q_2(s)$, $\bar{R}_0(s) = [\partial_s R_0(s) + R_1(s, s) - R_2(s, s) \quad R_0(s)]$ and $\bar{R}_i(s, \theta) = [\partial_s R_i(s, \theta) \quad 0]$ for $i \in \{1, 2\}$. Then, for any $x \in \mathbb{R}^m$, $\mathbf{x} \in H_1^n$,

$$\partial_s \left(\mathcal{P} \begin{bmatrix} P, Q_1 \\ Q_2, \{R_i\} \end{bmatrix} \begin{bmatrix} x \\ \mathbf{x} \end{bmatrix} \right) = \left(\partial_s \mathcal{P} \begin{bmatrix} P, Q_1 \\ Q_2, \{R_i\} \end{bmatrix} \right) \begin{bmatrix} x \\ \mathbf{x} \\ \partial_s \mathbf{x} \end{bmatrix}$$

To see PIETOOLS implementation, check Section 7.3.

3.3.2 Dirac operator on a PI operator

We now consider the composition of the Dirac operator with a 4-PI operator, evaluating $\mathcal{P}\mathbf{x}$ at some point inside the interval on which the 4-PI operator acts.

Lemma 8. Suppose $\mathcal{P} \begin{bmatrix} P, Q_1 \\ Q_2, \{R_i\} \end{bmatrix} : \mathbb{R}^m \times L_2^n \rightarrow \mathbb{R}^p \times L_2^q$. Then composition of a Dirac operator

Δ_c with \mathcal{P} is a 4-PI operator, $\Delta_c \mathcal{P} \begin{bmatrix} P, Q_1 \\ Q_2, \{R_i\} \end{bmatrix} : \mathbb{R}^m \times H_1^n \times L_2^n \rightarrow \mathbb{R}^{p+q}$, given by

$$\Delta_c \mathcal{P} \begin{bmatrix} P, Q_1 \\ Q_2, \{R_i\} \end{bmatrix} \begin{bmatrix} x \\ \mathbf{x} \end{bmatrix} = \mathcal{P} \begin{bmatrix} \bar{P}, \bar{Q}_1 \\ \emptyset, \{\emptyset\} \end{bmatrix} \begin{bmatrix} x \\ \Delta_c \mathbf{x} \\ \mathbf{x} \end{bmatrix} \quad (3.5)$$

where

$$\bar{P} = \begin{bmatrix} P & 0 \\ Q_2(c) & R_0(c) \end{bmatrix}, \quad \bar{Q}_1(s) = \begin{cases} \begin{bmatrix} Q_1(s) \\ R_1(c, s) \end{bmatrix}, & \text{for } s \leq c \\ \begin{bmatrix} Q_1(s) \\ R_2(c, s) \end{bmatrix}, & \text{for } s > c \end{cases}$$

3.4 Matrix Parametrization of Positive Definite PI operators

SOS polynomials are parametrized by a PSD matrix. In a similar manner, it is possible to parameterize positive semidefinite 4-PI operators by using PSD matrices. The following theorem gives a sufficient condition for the positivity of a 4-PI operator which is the key idea behind solving optimization problems with 4-PI constraints.

Theorem 9. *For any functions $Z_1 : [a, b] \rightarrow \mathbb{R}^{d_1 \times n}$, $Z_2 : [a, b] \times [a, b] \rightarrow \mathbb{R}^{d_2 \times n}$, if $g(s) \geq 0$ for all $s \in [a, b]$ and*

$$\begin{aligned} P &= T_{11} \int_a^b g(s) ds, \\ Q(\eta) &= g(\eta) T_{12} Z_1(\eta) + \int_\eta^b g(s) T_{13} Z_2(s, \eta) ds + \int_a^\eta g(s) T_{14} Z_2(s, \eta) ds, \\ R_1(s, \eta) &= g(s) Z_1(s)^\top T_{23} Z_2(s, \eta) + g(\eta) Z_2(\eta, s)^\top T_{42} Z_1(\eta) + \int_s^b g(\theta) Z_2(\theta, s)^\top T_{33} Z_2(\theta, \eta) d\theta \\ &\quad + \int_\eta^s g(\theta) Z_2(\theta, s)^\top T_{43} Z_2(\theta, \eta) d\theta + \int_a^\eta g(\theta) Z_2(\theta, s)^\top T_{44} Z_2(\theta, \eta) d\theta, \\ R_2(s, \eta) &= g(s) Z_1(s)^\top T_{32} Z_2(s, \eta) + g(\eta) Z_2(\eta, s)^\top T_{24} Z_1(\eta) + \int_\eta^b g(\theta) Z_2(\theta, s)^\top T_{33} Z_2(\theta, \eta) d\theta \\ &\quad + \int_s^\eta g(\theta) Z_2(\theta, s)^\top T_{34} Z_2(\theta, \eta) d\theta + \int_a^s g(\theta) Z_2(\theta, s)^\top T_{44} Z_2(\theta, \eta) d\theta, \\ R_0(s) &= g(s) Z_1(s)^\top T_{22} Z_1(s). \end{aligned} \tag{3.6}$$

where

$$T = \begin{bmatrix} T_{11} & T_{12} & T_{13} & T_{14} \\ T_{21} & T_{22} & T_{23} & T_{24} \\ T_{31} & T_{32} & T_{33} & T_{34} \\ T_{41} & T_{42} & T_{43} & T_{44} \end{bmatrix} \succcurlyeq 0,$$

then the operator $\mathcal{P} \begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix}$ as defined in (3.1) is positive semidefinite, i.e. $\left\langle x, \mathcal{P} \begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix} \right\rangle \geq 0$ for all $x \in \mathbb{R}^m \times L_2^n[a, b]$.

To see PIETOOLS implementation, check Section 8.2.1.

In the next chapter, we will take a brief look at the general structure of an optimization problem involving 4-PI variables and 4-PI constraints and a couple of examples on where these generally appear.

3.5 Inverse of a 4-PI operator

In [3], an analytical expression for inverse of a 4-PI operator $\mathcal{P} \begin{bmatrix} P, & Q \\ Q^T, & \{R_i\} \end{bmatrix}$ was derived with the assumption that the kernel of the integral terms in R_i are separable and the 4-PI operator is self-adjoint, i.e. $P = P^T$, $Q_2 = Q_1^T$, $R_0 = R_0^T$ and $R_1 = R_2$. The closed form the solution is given below.

Lemma 10. *Suppose that $Q(s) = HZ(s)$ and $R_1(s, \theta) = Z(s)^T \Gamma Z(\theta)$ and $\mathcal{P} := \mathcal{P}_{\{P, Q, R_0, R_1\}}$ is a coercive and self-adjoint operator where $\mathcal{P} : X \rightarrow X$. Then we get $\mathcal{P}^{-1} := \mathcal{P}_{\{\hat{P}, \hat{Q}, \hat{R}_0, \hat{R}_1\}}$ where*

$$\begin{aligned} \hat{P} &= (I - \hat{H} K H^T) P^{-1}, & \hat{Q}(s) &= \hat{H} Z(s) R_0(s)^{-1} \\ \hat{R}_0(s) &= R_0(s)^{-1}, & \hat{R}_1(s, \theta) &= \hat{R}_0^T(s) Z(s)^T \hat{\Gamma} Z(\theta) \hat{R}_0(\theta), \end{aligned}$$

if we define

$$\begin{aligned} K &= \int_{-1}^0 Z(s) R_0(s)^{-1} Z(s)^T ds \\ \hat{H} &= P^{-1} H (K H^T P^{-1} H - I - K \Gamma)^{-1} \\ \hat{\Gamma} &= -(\hat{H}^T H + \Gamma)(I + K \Gamma)^{-1} \end{aligned}$$

and \mathcal{P}^{-1} is self-adjoint where $\mathcal{P}^{-1} : X \rightarrow X$, and $\mathcal{P}^{-1} \mathcal{P} \mathbf{x} = \mathcal{P} \mathcal{P}^{-1} \mathbf{x} = \mathbf{x}$ for any $\mathbf{x} \in Z_{m,n,K}$

There is no known closed form solution for the inverse in case of a general 4-PI operator. Moreover, even in the case of sign definite 4-PI operators, the inverse has not be found analytical when the kernel is not separable. There are alternatives involving numerical approximation of the inverse for 4-PI operators, however, more research into this topic is necessary. To see PIETOOLS implementation, check Section 7.2.6.

Chapter 4

Linear Partial Integral Inequalities

In general, a convex optimization problems have the general form

$$\begin{aligned} \min_{x \in \mathbb{R}} f(x) \\ g_i(x) &\geq 0, \quad 1 \leq i \leq n \\ h_j(x) &= 0, \quad 1 \leq j \leq m \end{aligned}$$

where $f : X \rightarrow \mathbb{R}$, $g_i : X \rightarrow \mathbb{R}$ and $h_j : X \rightarrow \mathbb{R}$ are convex functions. LPIs are no different and have the same form. However, the decision variables, x , in the case of an LPI are 4-PI operators and the inequality constraints are positive semidefinite constraints on a 4-PI operator.

4.1 A General Class of LPI problems

A convex optimization problem with PI operator decision variables and Linear PI Inequality constraints are called Linear PI Inequalities (LPIs) and take the form

$$\begin{aligned} \min_{P, Q_i, R_i} f \left(\mathcal{P} \begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix} \right) \\ \sum_{i=1}^n \mathcal{E}_{ij}^* \mathcal{P} \begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix} \mathcal{E}_{ij} + \mathcal{Q}_j \succcurlyeq 0, \quad 1 \leq j \leq m \end{aligned} \quad (4.1)$$

where $P \in \mathbb{R}^{p \times m}$, $Q_1 : \mathbb{R} \rightarrow \mathbb{R}^{p \times n}$, $Q_2 : \mathbb{R} \rightarrow \mathbb{R}^{q \times m}$, $R_0 : \mathbb{R} \rightarrow \mathbb{R}^{q \times n}$, $R_1, R_2 : \mathbb{R}^2 \rightarrow \mathbb{R}^{q \times n}$ are decision variables, $f : \mathcal{B}(RL^{m,n}, RL^{p,q}) \rightarrow \mathbb{R}$ is a convex linear functional. \mathcal{E}_{ij} , \mathcal{Q}_j are known 4-PI operators where \mathcal{Q}_j must necessarily be self-adjoint 4-PI operators. For details on setting up an LPI in PIETOOLS, see Section ??.

4.2 Application of LPIs

LPIs appear in optimization problems involving infinite dimensional vector spaces. For example, estimation of Poincare constant can be posed as an LPI optimization problem.

Similarly, operator norms for integral operators such as Volterra, Fredholm operators etc can be estimated by solving LPIs. A major application of LPIs is in the field of control theory. Analysis and control of infinite dimensional systems such as PIEs (see Chapter 5) can be done using LPIs. Any LMI used in analysis and control of linear ODEs can, in theory, be extended to an LPI for linear PIEs. For example, test for stability or classical results such as bounded-real lemma and positive-real lemma can be extended to PIEs using an LPI. See Section 12 for a list of standard LPIs that arise in analysis and control of PIEs.

Chapter 5

Partial Integral Equations

In this chapter, we focus on one of the key applications of PIETOOLS which is the analysis and control of infinite dimensional systems called Partial Integral Equations (PIEs). First, we define a general PIE system with inputs and disturbances in state-space form and the notion of solutions of a PIE system and stability of solutions of a PIE system.

5.1 Definition

A PIE is a dynamical system, whose system parameters are given by 3-PI or 4-PI operators. The general form a PIE is

$$\begin{aligned}\mathcal{T}\dot{\mathbf{v}}(t) + \mathcal{T}_w\dot{w}(t) + \mathcal{T}_u\dot{u}(t) &= \mathcal{A}\mathbf{v}(t) + \mathcal{B}_1w(t) + \mathcal{B}_2u(t), \quad \mathbf{v}(0) = \mathbf{v}_0 \in RL^{m,n} \\ z(t) &= \mathcal{C}_1\mathbf{v}(t) + \mathcal{D}_{11}w(t) + \mathcal{D}_{12}u(t), \\ y(t) &= \mathcal{C}_2\mathbf{v}(t) + \mathcal{D}_{21}w(t) + \mathcal{D}_{22}u(t),\end{aligned}\tag{5.1}$$

where the $\mathcal{T}, \mathcal{A} : RL^{m,n} \rightarrow RL^{m,n}$, $\mathcal{T}_w : \mathbb{R}^p \rightarrow RL^{m,n}$, $\mathcal{T}_u : \mathbb{R}^q \rightarrow RL^{m,n}$, $\mathcal{B}_1 : \mathbb{R}^p \rightarrow RL^{m,n}$, $\mathcal{B}_2 : \mathbb{R}^q \rightarrow RL^{m,n}$, $\mathcal{C}_1 : RL^{m,n} \rightarrow \mathbb{R}^k$, $\mathcal{C}_2 : RL^{m,n} \rightarrow \mathbb{R}^l$, $\mathcal{D}_{11} \in \mathbb{R}^{k \times p}$, $\mathcal{D}_{12} \in \mathbb{R}^{k \times q}$, $\mathcal{D}_{21} \in \mathbb{R}^{l \times p}$ and $\mathcal{D}_{22} \in \mathbb{R}^{l \times q}$ are 4-PI operators.

5.2 Application of PIE framework

As discussed in Chapter 2, PIE framework was developed for PDEs, however, is not restricted to only PDEs. Many standard systems, such as Time-delay systems, DDEs, ODE coupled with PDEs etc. can be written in PIE form when certain conditions are met. In this section, we list two classes of systems, namely DDEs and PDEs, that can be converted to PIEs. We will not go into method of conversion as it is beyond the scope of this manual, however, relevant references on the topic will be introduced in appropriate locations.

5.2.1 DDE to PIE

A general class of linear DDEs, with delay in disturbance and inputs, can be written in the form

$$\begin{aligned}
\dot{x}(t) &= A_0 x(t) + \sum_{i=1}^K A_i x(t - \tau_i) + \sum_{i=1}^K \int_{-\tau_i}^0 A_{di}(s) x(t + s) ds + B_1 w(t) + \sum_{i=1}^K B_{1i} w(t - \tau_i) \\
&\quad + \sum_{i=1}^K \int_{-\tau_i}^0 B_{1di} w(t + s) ds + B_2 u(t) + \sum_{i=1}^K B_{2i} u(t - \tau_i) + \sum_{i=1}^K \int_{-\tau_i}^0 B_{2di} u(t + s) ds \\
z(t) &= C_{10} x(t) + \sum_{i=1}^K C_{1i} x(t - \tau_i) + \sum_{i=1}^K \int_{-\tau_i}^0 C_{1di}(s) x(t + s) ds + D_{11} w(t) + \sum_{i=1}^K D_{11i} w(t - \tau_i) \\
&\quad + \sum_{i=1}^K \int_{-\tau_i}^0 D_{11di} w(t + s) ds + D_{12} u(t) + \sum_{i=1}^K D_{12i} u(t - \tau_i) + \sum_{i=1}^K \int_{-\tau_i}^0 D_{12di} u(t + s) ds \\
y(t) &= C_{20} x(t) + \sum_{i=1}^K C_{2i} x(t - \tau_i) + \sum_{i=1}^K \int_{-\tau_i}^0 C_{2di}(s) x(t + s) ds + D_{21} w(t) + \sum_{i=1}^K D_{21i} w(t - \tau_i) \\
&\quad + \sum_{i=1}^K \int_{-\tau_i}^0 D_{21di} w(t + s) ds + D_{22} u(t) + \sum_{i=1}^K D_{22i} u(t - \tau_i) + \sum_{i=1}^K \int_{-\tau_i}^0 D_{22di} u(t + s) ds.
\end{aligned} \tag{5.2}$$

Any DDE in the form Eq.(5.2) can be converted to a PIE of the form Eq. (5.1) using the conversion formulae provided in [4] (see Eq. (12)-(14)). PIETOOLS allows conversion of DDEs to PIEs via the script files (see Section 21.1.6). User can define the DDE parameters mentioned in Eq. (5.2) to obtain corresponding PI operators in Eq. (5.1).

5.2.2 ODE-PDE to PIE

We begin by defining a general representation and parameterization of ODE-PDE-PIDE interconnected systems. First, we divide the ODE-PDE-PIDE system into PDE and ODE subsystems. The ODE subsystem maps signals $(w, u, r) \mapsto (z, y, v)$ ($(w(t), u(t), r(t)) \in \mathbb{R}^{n_w+n_u+n_r}$ and $(z(t), y(t), v(t)) \in \mathbb{R}^{n_z+n_y+n_v}$) where (for given initial conditions $x(0)$) the mapping is uniquely defined by

$$\begin{bmatrix} \dot{x}(t) \\ z(t) \\ y(t) \\ v(t) \end{bmatrix} = \begin{bmatrix} A_x & B_{xw} & B_{xu} & B_{xr} \\ C_z & D_{zw} & D_{zu} & D_{zr} \\ C_y & D_{yw} & D_{yu} & D_{yr} \\ C_v & D_{vw} & D_{vu} & 0 \end{bmatrix} \begin{bmatrix} x(t) \\ w(t) \\ u(t) \\ r(t) \end{bmatrix}. \tag{5.3}$$

The partition of both the input and output signals into 3 components is included so as to specify which input channels are used for control (u), for exogenous disturbances or forcing (w) and which channels interact the PDE subsystem (r). Furthermore, the output channel is partitioned into channels for regulation (z), sensor measurements (y), and for driving of

the PDE subsystem (v). Note that the driving signal, v can affect the PDE both through the boundary conditions and the interior of the domain.

The PDE subsystem is defined by three constraints: the continuity constraints, the dynamics, and the boundary conditions. The continuity constraint is first used to specify the state dimensions and corresponding continuity so that for a given order N , we have $n \in \mathbb{N}^{N+1} = \{n_0, \dots, n_N\}$ implies $\mathbf{x}_i \in W_i^{n_i}$ with the concatenated state \mathbf{x} defined as

$$\mathbf{x}(t, s) = \begin{bmatrix} \mathbf{x}_0(t, s) \\ \vdots \\ \mathbf{x}_N(t, s) \end{bmatrix} \in W^n.$$

Furthermore, to simplify the presentation, for a given set of state dimensions $n \in \mathbb{N}^{N+1} = \{n_0, \dots, n_N\}$, we introduce the nilpotent shift operator $S^i : \mathbb{R}^{\sum_{j=0}^N n_j} \rightarrow \mathbb{R}^{\sum_{j=i}^N n_j}$ where

$$S^i = \begin{bmatrix} 0_{(\sum_{j=i}^N n_j) \times (\sum_{j=0}^{i-1} n_j)} & I_{(\sum_{j=i}^N n_j)} \end{bmatrix},$$

so that

$$(S^i \mathbf{x})(s) = \begin{bmatrix} \mathbf{x}_i(s) \\ \vdots \\ \mathbf{x}_N(s) \end{bmatrix} \in \mathbb{R}^{\sum_{j=i}^N n_j} \quad \text{and hence} \quad (\partial_s^j S^i \mathbf{x})(s) = \begin{bmatrix} \partial_s^j \mathbf{x}_i(s) \\ \vdots \\ \partial_s^j \mathbf{x}_N(s) \end{bmatrix}.$$

The PDE is parameterized using the parameter set $\{A_{0,i}, A_{1,i}, A_{2,i}, C_{r,i}, B_{xv}, B_{xb}, D_{rv}, D_{rb}, B_{x,i}, B_v, B\}$ where $A_{0,i}(s), A_{1,i}(s, \theta), A_{2,i}(s, \theta) \in \mathbb{R}^{\sum_{j=1}^N n_j}$, $D_v(s) \in \mathbb{R}^{\sum n_i \times n_v}$ and $D_b(s) \in \mathbb{R}^{\sum n_i \times 2 \sum_{i=1}^N n_i}$. In the following format, we combine three equations: the dynamics of the PDE; an output, $r(t)$, which influences the ODE state; and a set of boundary conditions. For simplicity, we do not directly include a regulated or sensed output, since such terms can be routed through $r(t)$ to the ODE which has regulated and sensed outputs. All three equations are influenced by the distributed state, \mathbf{x} , the signal from the ODE, $v(t)$, and the set of boundary values, $\mathbf{x}_b(t)$. Again, for simplicity, we do not include disturbances or control inputs in the PDE, as these can be routed from the ODE using the signal v . Finally, recall that for the spatial derivatives to exist and for the boundary values to be well-posed, we also have the implicit constraint $\mathbf{x} \in W^n$. The PDE can now be defined as

$$\begin{bmatrix} \dot{\mathbf{x}}(t, s) \\ r(t) \\ 0 \end{bmatrix} = \sum_{i=0}^N \begin{bmatrix} A_{p0,i}(s) + \int_a^s A_{p1,i}(s, \theta) + \int_s^b A_{p2,i}(s, \theta) \\ \int_a^b C_{rp,i}(s) \\ \int_a^b E_{bp,i}(s) \end{bmatrix} \partial^i S^i \mathbf{x}(t) + \begin{bmatrix} B_{pv}(s) & B_{pb}(s) \\ D_{rv} & D_{rb} \\ E_{bv} & E_{bb} \end{bmatrix} \begin{bmatrix} v(t) \\ \mathbf{x}_b(t) \end{bmatrix}, \quad (5.4)$$

where the boundary values are defined using a Dirac operator as

$$\mathbf{x}_b(t) = \begin{bmatrix} \Delta_a \\ \Delta_b \end{bmatrix} \begin{bmatrix} S\mathbf{x}(t, s) \\ \partial_s S^2 \mathbf{x}(t, s) \\ \vdots \\ \partial_s^{N-1} S^N \mathbf{x}(t, s) \end{bmatrix} \in \mathbb{R}^{2 \sum_{j=1}^N \sum_{i=j}^N n_i}, \quad \Delta_a \mathbf{x}(s) := \mathbf{x}(a), \quad (5.5)$$

Note the form of boundary conditions can be used to represent any linear boundary condition, for example, Dirichlet, Neumann, Robin, periodic, integral boundary conditions, etc. For convenience, for given $n \in \mathbb{N}^{N+1}$, we define the following state-space, X_v , where the v subscript represents the explicit dependence of this space on the time-varying value of $v(t)$.

$$X_v := \left\{ \mathbf{x} \in W^n[a, b] : E_{bb} \begin{bmatrix} \Delta_a \\ \Delta_b \end{bmatrix} \begin{bmatrix} S\mathbf{x}(s) \\ \partial_s S^2\mathbf{x}(s) \\ \vdots \\ \partial_s^{N-1} S^N\mathbf{x}(s) \end{bmatrix} + \int_a^b \sum_{i=0}^N E_{bp,i}(s) \partial_s^i S^i\mathbf{x}(s) ds + E_{bv}v = 0, \right\}. \quad (5.6)$$

Having defined X_v , we may also include initial conditions $\mathbf{x}(0) = \mathbf{x}^0 \in X_{v(0)}$.

Obviously, PIETOOLS provides functions (see Section 21.1) to perform the conversion. The reference listed above is to satiate any curiosity about things happening behind the black box. The users need not perform those steps for converting a PDE to PIE.

Part II

PIETOOLS: A computational toolbox

Chapter 6

PIETOOLS 2021a

PIETOOLS 2021a is compatible with Windows, Mac or Linux systems and has been verified to work with MATLAB version 2014 or higher, however, we suggest to use the latest version of MATLAB.

6.1 Getting Started

Before you start, **make sure** that you have

1. MATLAB with version 2014a or newer. (MATLAB 2020a or newer for GUI input)
2. MATLAB has permission to create and edit folders/files in your working directory.

6.1.1 Installation

PIETOOLS 2021a can be installed in two ways.

1. **Using install script:** The script installs the following files — tbxmanager (skipped if already installed), SeDuMi 1.3 (skipped if already installed), Modified SOSTOOLS 3.03 (always installed), PIETOOLS 2021a (always installed). Adds all the files to MATLAB path.
 - Go to the website control.asu.edu/pietools/.
 - Download the file **pietools_install.m** and run it in MATLAB.
 - **Run the script from the folder it is downloaded in to avoid path issues.**
2. **Setting up PIETOOLS 2021a manually:**
 - Download and install C/C++ compiler for the OS.
 - Install an SDP solver. SeDuMi can be obtained from [this link](#).
 - Download SeDuMi and run **install_sedumi.m** file.
 - Alternatively, install MOSEK, obtain license file and add to MATLAB path.
 - Download **PIETOOLS_2021a.zip** from [this link](#), unzip, and add to MATLAB path.

6.2 Before you Start

6.3 Demos of PDEs and Integral Operators in PIETOOLS

To make your first program in PIETOOLS_2021a, the best way is to run one of the demo files located in `GetStarted_DOCS_DEMOS` folder. Specifically, the file `Getting_Started_DEMO.m` includes the example described here. The remainder of this section will give everything essential you need to know to start your journey with PIETOOLS.

Chapter 7

opvar: A MATLAB class for PI operators

PIETOOLS introduces a MATLAB class, named `opvar`, to store all the parameters related to a 4-PI operator. Recall that a 4-PI operator has 6 parameters:

1. a matrix P
2. three matrix valued polynomials in s — Q_1 , Q_2 and R_0
3. two matrix valued polynomials in s and θ — R_1 and R_2

The `opvar` class has 8 accessible properties, including the 6 parameters mentioned above. The properties of an `opvar` object, $T: RL^{p,q}[a, b] \rightarrow RL^{m,n}[a, b]$, can be accessed using “.” as shown below:

- $T.P$: $m \times p$ matrix
- $T.Q1/T.Q2$: $m \times q/n \times p$ matrix of polynomials in pvar s
- $T.R$: A MATLAB struct
 - $R.R0$: a $n \times q$ matrix of polynomials in pvar s
 - $R.R1, R.R2$: $n \times q$ matrices of polynomials in pvars s and θ
- $T.dim$: $\begin{bmatrix} m & p \\ n & q \end{bmatrix}$
- $T.I$: $[a \ b]$

There are two additional parameters $T.var1$ and $T.var2$ that are set as pvars s and θ . Although, these can be modified, it is highly recommended not to. Note that $T.dim$ is dependent on size of the 6 parameters P , Q_i and $R.R_i$. Modifying those parameters automatically changes the value stored in `dim` property. If the dimensions of the parameters are incompatible, `dim` property stores `Nan` as its value to alert the user about the discrepancy.

7.1 Initialization and Assignment

4-PI operators can be defined in PIETOOLS using the `opvar` class command. To initialize an `opvar` object use the syntax

```
opvar T;
```

This command creates an empty `opvar` object with all dimensions 0. Consequently, the parameter `P`, `Qi`, `Ri` are initialized to 0×0 matrices. If the desired dimension of the `opvar` `T` is known, lets say $[2, 3; 3, 4]$ for example, it can be assigned by using the command

```
T.dim = [2 3; 3 4];
```

The above code, when executed, allocates the dimensions of the parameters by setting them zero matrices of appropriate dimensions. In this example, the parameter `P` is set to $0_{2 \times 3}$, `Q1` to $0_{2 \times 4}$, `Q2` to $0_{3 \times 3}$ and `Ri` to $0_{3 \times 4}$. Similarly, to assign a value to the interval of `T` (for example, lets set it to $[2, 3]$), just use the command

```
T.I = [2,3];
```

The parameters `P`, `Qi`, `R`, `Ri` can also be modified separately by using the method described above using `.` operation.

```
T.Q2 = [s^2 0 0; 3 s^3 0; 0 s+2*s^2 0];
```

Note that, assigning parameters with values that have incompatible dimensions will lead to invalid 4-PI operator whose `dim` will store `Nan`. For example, since `R`, `Ri` are set to map to $L_2^3[2, 3]$, assigning `T.Q2 = [s^2 0 0; 0 s+2*s^2 0];` will lead to a discrepancy in output dimensions since `Q2` now maps to $L_2^2[2, 3]$.

7.2 opvar class methods

In this section, we go over various methods that help in manipulating and handling of `opvar` objects in PIETOOLS.

7.2.1 +

`opvar` objects, `A` and `B`, can be added simply by using the command

```
A+B
```

For two `opvar` objects to be added, they **must** have same values stored in `dim` and `I`. Furthermore, if `A` (or `B`) is a scalar then PIETOOLS considers that as adding `A*I` (or `B*I`) where `I` is an identity matrix. Again, this operation is appropriate if and only if dimensions match. Similarly, if `A` (or `B`) is a matrix with matching dimension, it can be added to `opvar` `B` (or `A`) using the same command.

7.2.2 *

opvar objects, **A** and **B**, can be composed simply by using the command

A*B

For two opvar objects to be composed, they **must** have same values stored in **I** and the second column in **A.dim** must match first column in **B.dim**. Furthermore, if **A** (or **B**) is a scalar then PIETOOLS considers that as a scalar multiplication operation, thus multiplying all parameters of **B** (or **A**) by that value.

7.2.3 '

Adjoint of an opvar object, **A**, can be calculated by using the command

A'

7.2.4 Concatenation

opvar objects can be concatenated in any order just like matrices, provided the dimensions match. For example, opvar **A** and **B**, can be horizontally or vertically concatenated by using the command

[A B] %for horizontal

[A; B] %for vertical

Note that **A** and **B** can be horizontally concatenated if and only if they have same value in **I** and first columns of **A.dim** and **B.dim** are same. Similarly, two opvars **A** and **B** can be vertically concatenated if and only if they have same interval **I** and second columns of **A.dim** and **B.dim** are equal.

7.2.5 isvalid

This function is mostly used for internal error checking. However, the users can verify if the opvar object is initialized and assigned compatible values by using this function.

isvalid(P)

The function returns a logical value. 0 is everything is in order, 1 if the object has incompatible dimensions, 2 if property **P** is not a matrix, 3 if properties **Q1**, **Q2** or **R0** are not polynomials in s , 4 if properties **R1** or **R2** are not polynomials in s and θ . The function also returns an error message as an optional output describing the exact cause of error.

7.2.6 inv_opvar

Inverse of an opvar object T can be numerically calculated in the case when $T.R.R1=T.R.R2$. This is performed using the function

```
inv_opvar(T)
```

Note that, currently there is no known closed form solution for an inverse when T is not self-adjoint or when $R1$ and $R2$ are not equal.

7.2.7 Conditional statements involving PI operators

4-PI operators can also be used in conditions statements to test for equality or inequality. For example, if A and B are two opvar objects, user can write a conditional test to verify if they are equal using the syntax

```
A==B
```

This code returns a boolean value. Additionally, there is a function `isempty_opvar` which is essentially verifies if $A==0$ (a zero operator) or if $A==\phi$ (a null operator).

7.2.8 subsref or sliced indexing

4-PI operators can also be sliced the way matrices are sliced in matrices. The index slicing is performed in the same manner as matrices.

```
T(row_ind, col_ind)
```

Indexing 4-PI operators is slightly different due to presence of multiple components. The general idea is described here. Assume components of 4-PI operator is stacked in a big matrix as shown below.

$$B = \left[\begin{array}{c|c} P & Q1 \\ \hline Q2 & Ri \end{array} \right]$$

Then, row indices specified in `row_ind` correspond to the rows in this big matrix. Column indices, `col_ind`, are associated with the columns of this big matrix in similar manner. The retrieved slices are put in appropriate components and a 4-PI operator is returned. Note the bottom-right block of the big matrix B has 3 components in Ri . If indices in the slice correspond to rows and columns in this block, then the slice is extracted from all three components and stored in a Ri part of the new sliced PI operator.

7.3 Additional methods for opvar objects

There are some additional, quality of life, functions included in PIETOOLS that can be used in debugging or as the user sees fit. In this section, we compile the list of those functions,

without going into details or explanation. However, users can find additional information by using `help` command in MATLAB.

Function Name	Description
<code>degbalance(T)</code>	Estimates polynomial degrees needed to create a poslpi-var, \mathbf{Q} , such that $\mathbf{T}=\mathbf{Q}$ has at least one solution
<code>getdeg(T)</code>	Returns highest and lowest degree of s and θ in the components of the opvar object \mathbf{T}
<code>rand_opvar(dim, deg)</code>	Creates a random opvar object of specified dimensions dim and polynomial degrees deg
<code>show(T,opts)</code>	Alternative display format for opvar objects with optional argument to omit selected properties from display output
<code>opvar_postest(T)</code>	Numerically test for sign definiteness of \mathbf{T} . Returns -1 if negative definite, 0 if indefinite and 1 if positive definite.
<code>diff_opvar(T)</code>	Returns composition of derivative operator with opvar \mathbf{T} as described in Lem. 7

Chapter 8

PIETOOLS Functions: Setting up LPIs and more

In this chapter, we focus on the functions required to set up an LPI optimization problem. To set up and solve an optimization problem we need functions that can declare variable 4-PI operators and constraints involving 4-PI operators. Furthermore, we need a structure that can store the information related to the optimization problem. For this purpose, we utilize the SOSTOOLS [5]. Using `sosprogram` we initialize a structure that can store decision variables, objective functions and constraints related to LPI problem.

8.1 Optimization problem structure

An optimization problem is initialized using

```
prog=sosprogram([s,theta]);
```

where `s` and `theta` are declared as pvar objects. Decision variables can be added to this optimization problem by using the code

```
prog=sosdecvar(prog, gamma);
```

where `gamma` is a pvar which can then be specified as the objective function using

```
prog=sossetobj(prog, gamma);
```

Note that objective function need not be a pvar object and can instead be a linear functional on other pvar objects. For example, `sossetobj(prog, gamma1+5*gamma2)` is also valid assuming `gamma1` and `gamma2` are pvars themselves. The linear functional can also be on 4-PI variable operators.

8.2 4-PI Decision Variables

In this section, we introduce functions related to declaration of variable opvar objects and retrieval of solution once the LPI is solved.

8.2.1 poslpivar

Positive definite opvar variables can be defined by using `poslpivar` function as shown below.

```
[prog,T] = poslpivar(prog,n,I,d,opts);
```

This function takes three mandatory inputs.

- **prog**: an sosprogram that stores all decision variables and constraints
- **n**: 2×1 vector specifying the dimensions of T
- **I**: 2×1 vector specifying the interval of T
- **d** (optional): specifies the degrees of PI operator. **d** is cell structure of the form $\{a, [b0, b1, b2]\}$ where a is the degree of s in $Z_1(s)$, $b0$ is degree of s in $Z_2(s, \theta)$, $b1$ degree of θ in $Z_2(s, \theta)$ and $b2$ is degree of s and θ combined in $Z_2(s, \theta)$. (see Theorem 9)
- **opts** (optional): This is a structure with fields
 - **exclude**: 4×1 vector with 0 and 1 values. Excludes the block T_{ij} (see Theorem 9) if i -th value is 1.
 - **psatz**: Sets $g(s) = 1$ if 0 and $g(s) = (b - s)(s - a)$ if 1
 - **sep**: Constrains $R1=R2$ if set to 1

The output is an sosprogram with new decision variables and a positive definite T opvar variable. `poslpivar` functions has other experimental features involving sparsity constraints on T matrix of Theorem 9 which should be used with caution. Use `help poslpivar` command for more info. Note that positive definite opvars have only 2 dimensions since they need to be self-adjoint.

8.2.2 lpivar

Indefinite opvar variables can be defined by using `lpivar` function as shown below.

```
[prog,T] = lpivar(prog,n,I,d);
```

This function takes three mandatory inputs.

- **prog**: an sosprogram that stores all decision variables and constraints
- **n**: 2×2 vector specifying the dimensions of T
- **I**: 2×1 vector specifying the interval of T
- **d** (optional): specifies the degrees of PI operator. **d** is an array structure of the form $[b0, b1, b2]$ where $b0$ is degree of s in $Q1, Q2, R0$, $b1$ degree of θ in $R1, R2$ and $b2$ is degree of s in $R1, R2$.

The output is an sosprogram with new decision variables and an indefinite T opvar variable.

8.2.3 getsol_lpivar

Optimization problems are solved using `ssolve(prog)` command. Once the LPI is solved, the solution to any opvar decision variables can be retrieved by using the command

```
T = getsol_opvar(prog,T)
```

Note that sosprogram `prog` must be in a solved state (`ssolve` must be used) to retrieve the solution for the decision variable `T`.

8.3 4-PI Constraints

Apart from decision variables and objective functions, setting up constraints another important step in setting up any optimization problem. In LPIs, we often have equality or inequality constraints involving 4-PI objects. To set up these constraints, we introduce two functions.

8.3.1 lpi_eq

`lpi_eq` function sets up equality constraints on an opvar object. Setting up equality constraints in PIETOOLS is quite simple. If `T` needs to be constrained as $T=0$ then use the command

```
prog = lpi_eq(prog,T)
```

This returns a modified optimization structure `prog` with the constraints $T=0$ included.

8.3.2 lpi_ineq

`lpi_ineq` function sets up inequality constraints on an opvar object. Inequality constraints on opvar objects are in fact positivity constraints and hence are imposed by using the result in Theorem 9. If `T` needs to be constrained as $T \geq 0$ then use the command

```
prog = lpi_ineq(prog,T)
```

This returns a modified optimization structure `prog` with the constraints $T \geq 0$ included. Note the constraints imposed are always non-strict. For strict positivity, user should use an offset such as `lpi_ineq(prog,T-ep)` where `ep` is a small positive number.

Part III

PIEs: Representation, Analysis, Control, and Simulation

Chapter 9

Representation of PIEs

In part I, we briefly described some benefits of PIE representation over classical representations such as DDEs, PDEs, DDFs, etc. Now, we describe how PIE systems are defined in MATLAB followed by a few examples.

9.1 Elements of the PIE data structure

A PIE system, which is free from any constraint on the PIE state, is defined by 12-PI operators as shown below. To allow convenient transfer of parameters to functions that requires all the PI operators that define a PIE system we bundle all the PI operators under a single MATLAB structure with 12 fields that correspond to each of the 12 PI operators (refer to Table 9.1), respectively. Let

$$\begin{aligned}\mathcal{T}\dot{\mathbf{v}}(t) + \mathcal{T}_w\dot{w}(t) + \mathcal{T}_u\dot{u}(t) &= \mathcal{A}\mathbf{v}(t) + \mathcal{B}_1w(t) + \mathcal{B}_2u(t), \quad \mathbf{v}(0) = \mathbf{v}_0 \in \mathbb{R}^{m,n} \\ z(t) &= \mathcal{C}_1\mathbf{v}(t) + \mathcal{D}_{11}w(t) + \mathcal{D}_{12}u(t), \\ y(t) &= \mathcal{C}_2\mathbf{v}(t) + \mathcal{D}_{21}w(t) + \mathcal{D}_{22}u(t),\end{aligned}\tag{9.1}$$

be a general PIE system whose signals $\mathbf{v}(t) \in \mathbb{R}^{n_x} \times L_2^{n_p}(a, b)$, $w(t) \in \mathbb{R}^{n_w}$, $u(t) \in \mathbb{R}^{n_u}$, $z(t) \in \mathbb{R}^{n_z}$ and $y(t) \in \mathbb{R}^{n_y}$. A standard PIE structure for the above system, named `PIE` (variable name can be different from `PIE`), in MATLAB would have the fields each PI operator as described below:

1. `PIE.T`: opvar class object of dimension `[nx nx; np np]`
2. `PIE.Tw`: opvar class object of dimension `[nx nw; np 0]`
3. `PIE.Tu`: opvar class object of dimension `[nx nu; np 0]`
4. `PIE.A`: opvar class object of dimension `[nx nx; np np]`
5. `PIE.B1`: opvar class object of dimension `[nx nw; np 0]`
6. `PIE.B2`: opvar class object of dimension `[nx nu; np 0]`
7. `PIE.C1`: opvar class object of dimension `[nz nx; 0 np]`

8. PIE.C2: opvar class object of dimension [ny nx; 0 np]
9. PIE.D11: opvar class object of dimension [nz nw; 0 0]
10. PIE.D12: opvar class object of dimension [nz nu; 0 0]
11. PIE.D21: opvar class object of dimension [ny nw; 0 0]
12. PIE.D22: opvar class object of dimension [ny nu; 0 0]

PIE parameters in:					
Eqn. (9.1)	PIE.	Eqn. (9.1)	PIE.	Eqn. (9.1)	PIE.
\mathcal{T}	T	\mathcal{T}_w	Tw	\mathcal{T}_u	Tu
\mathcal{A}	A	\mathcal{B}_1	B1	\mathcal{B}_2	B2
\mathcal{C}_1	C1	\mathcal{D}_{11}	D11	\mathcal{D}_{12}	D12
\mathcal{C}_2	C2	\mathcal{D}_{21}	D21	\mathcal{D}_{22}	D22

Table 9.1: Equivalent names of Matlab elements of the PIE structure terms for terms in Eqn. (9.1). For example, to set term T to I, we use PIE.T=I.

While all the fields in a PIE structure is not necessary for every LPI test, it is good rule of thumb to initialize unnecessary PI operators by opvars of appropriate dimensions to have a consistent MATLAB structure.

CAUTION: Currently, the PIE structure is not implemented as a class in PIETOOLS — a PIE MATLAB class will be included in a future release. The user must be careful and ensure that the dimensions of the opvar objects stored in the PIE structure are consistent because there are no error checks implemented by MATLAB itself to prevent any dimension mismatch.

9.2 Initializing and modifying a PIE data structure

In MATLAB, a PIE structure can be initialized by directly assigning fields with opvar objects of appropriate dimensions. For example, to define **sysA** as a PIE structure that is compatible with PIETOOLS, we can use the following set of commands.

```
opvar T A;
T.I = [0,1]; A.I = [0,1];
T.dim = [2,2;2,2]; A.dim = [2,2;2,2];
sysA.T = T; sysA.A = A;
```

The above snippet of code initializes two opvar objects T and A whose PI parameters are all zero-valued matrices of dimension 2×2 . Then the opvars are assigned to a PIE structure **sysA**. Once initialized, parameters of PI operators can be modified directly by using `.` access. For example,

```
sysA.T.R0 = eye(2);
```

changes the R_0 component of `sysA.T` to an identity matrix. Similarly, other properties like dimensions, interval, polynomial variables can be modified using dot access supported by manipulation routines described in [7.1](#).

Note: All opvar objects in `sysA` are assumed to be defined on the same interval as `T.I`.

Chapter 10

LPIs for Analysis of PIEs

Consider the following PIEs

$$\begin{aligned}\mathcal{T}\dot{\mathbf{v}}(t) + \mathcal{T}_w\dot{w}(t) + \mathcal{T}_u\dot{u}(t) &= \mathcal{A}\mathbf{v}(t) + \mathcal{B}_1w(t) + \mathcal{B}_2u(t), \quad \mathbf{v}(0) = \mathbf{v}_0 \in RL^{m,n} \\ z(t) &= \mathcal{C}_1\mathbf{v}(t) + \mathcal{D}_{11}w(t) + \mathcal{D}_{12}u(t), \\ y(t) &= \mathcal{C}_2\mathbf{v}(t) + \mathcal{D}_{21}w(t) + \mathcal{D}_{22}u(t),\end{aligned}\tag{10.1}$$

where all the operators are 4-PI operators. The LPIs listed below are extensions of classical results used in analysis of ODEs using LMIs.

PIETOOLS, primarily has 4 functions that help in analysis of PIEs, specifically help in testing stability of PIE and finding H_∞ -norm of the PIE. These functions are:

1. **Stability:** For given 4-PI operators \mathcal{T} and \mathcal{A} , the following LPIs are needed to be solved to test stability of PIEs.

$$\begin{aligned}\mathcal{P} &\succ 0 \\ \mathcal{T}^*\mathcal{P}\mathcal{A} + \mathcal{A}^*\mathcal{P}\mathcal{T} &\preceq 0\end{aligned}$$

If \mathcal{P} is feasible, PIE is stable. The executive file that solves this problem is called `executive_PIETOOLS_stability`.

2. **Dual Stability:** For given 4-PI operators \mathcal{T} and \mathcal{A} , the following LPIs are needed to be solved to test stability of PIEs.

$$\begin{aligned}\mathcal{P} &\succ 0 \\ \mathcal{T}\mathcal{P}\mathcal{A}^* + \mathcal{A}\mathcal{P}\mathcal{T}^* &\preceq 0\end{aligned}$$

If \mathcal{P} is feasible, PIE is stable. The executive file that solves this problem is called `executive_PIETOOLS_stability_dual`.

3. **L_2 -gain from w to z** This amounts to determining the minimum value of $\gamma > 0$ such that $\|z\| \leq \gamma\|w\|$. This is achieved by solving the following LPIs

$$\min_{\gamma, \mathcal{P}} \gamma$$

$$\begin{aligned} & \mathcal{P} \succ 0 \\ & \begin{bmatrix} -\gamma I & \mathcal{D}_{11}^* & \mathcal{B}_1^* \mathcal{P} \mathcal{T} \\ (\cdot)^* & -\gamma I & \mathcal{C}_1 \\ (\cdot)^* & (\cdot)^* & \mathcal{T}^* \mathcal{P} \mathcal{A} + \mathcal{A}^* \mathcal{P} \mathcal{T} \end{bmatrix} \preccurlyeq 0 \end{aligned}$$

The executive file that solves this problem is called `executive_PIET00LS_Hinf_gain`

4. **Dual L_2 -gain from w to z** This amounts to determining the minimum value of $\gamma > 0$ such that $\|z\| \leq \gamma \|w\|$ for the dual PIEs. This is achieved by solving the following LPIs

$$\begin{aligned} & \min_{\gamma, \mathcal{P}} \gamma \\ & \mathcal{P} \succ 0 \\ & \begin{bmatrix} -\gamma I & \mathcal{D}_{11} & \mathcal{T} \mathcal{P} \mathcal{C}_1 \\ (\cdot)^* & -\gamma I & \mathcal{B}_1^* \\ (\cdot)^* & (\cdot)^* & \mathcal{T} \mathcal{P} \mathcal{A}^* + \mathcal{A} \mathcal{P} \mathcal{T}^* \end{bmatrix} \preccurlyeq 0 \end{aligned}$$

The executive file that solves this problem is called
`executive_PIET00LS_Hinf_gain_dual`

Chapter 11

LPIs for Optimal Estimation of PIEs

For the following PIE

$$\begin{aligned}\mathcal{T}\dot{\mathbf{v}}(t) + \mathcal{T}_w \dot{w}(t) &= \mathcal{A}\mathbf{v}(t) + \mathcal{B}_1 w(t), \\ z(t) &= \mathcal{C}_1 \mathbf{v}(t) + \mathcal{D}_{11} w(t), \\ y(t) &= \mathcal{C}_2 \mathbf{v}(t) + \mathcal{D}_{21} w(t),\end{aligned}\tag{11.1}$$

a state estimator has the following structure:

$$\begin{aligned}\mathcal{T}\dot{\hat{\mathbf{v}}}(t) &= \mathcal{A}\hat{\mathbf{v}}(t) + \mathcal{L}(y(t) - \hat{y}(t)), \\ \hat{z}(t) &= \mathcal{C}_1 \hat{\mathbf{v}}(t) \\ \hat{y}(t) &= \mathcal{C}_2 \hat{\mathbf{v}}(t).\end{aligned}\tag{11.2}$$

The H_∞ optimal estimation problem amounts to synthesizing \mathcal{L} such that the estimation error $z - \hat{z}$ admits $\|z - \hat{z}\| \leq \gamma \|w\|$ for a particular $\gamma > 0$.

The following LPIs yield a feasible PI operators \mathcal{P} and \mathcal{Z} for which $\mathcal{L} := \mathcal{P}^{-1}\mathcal{Z}$ results in a minimum value of $\gamma > 0$ such that $\|z - \hat{z}\| \leq \gamma \|w\|$.

$$\begin{aligned}& \min_{\gamma, \mathcal{P}} \quad \gamma \\ & \mathcal{P} \succ 0 \\ & \begin{bmatrix} \mathcal{T}_w^*(\mathcal{P}\mathcal{B}_1 + \mathcal{Z}\mathcal{D}_{21}) + (\cdot)^* & 0 & (\cdot)^* \\ 0 & 0 & 0 \\ -(\mathcal{P}\mathcal{A} + \mathcal{Z}\mathcal{C}_2)^*\mathcal{T}_w & 0 & 0 \end{bmatrix} + \begin{bmatrix} -\gamma I & -\mathcal{D}_{11}^\top & -(\mathcal{P}\mathcal{B}_1 + \mathcal{Z}\mathcal{D}_{21})^*\mathcal{T} \\ (\cdot)^* & -\gamma I & \mathcal{C}_1 \\ (\cdot)^* & (\cdot)^* & (\mathcal{P}\mathcal{A} + \mathcal{Z}\mathcal{C}_2)^*\mathcal{T} + (\cdot)^* \end{bmatrix} \preceq 0\end{aligned}$$

The executive function for solving the above problem is called

`executive_PIETOOLS_Hinf_estimator`

Chapter 12

LPIs for Optimal Control of PIEs

In this chapter, we discuss the synthesis of H_∞ optimal control of PIEs. To this end, consider the PIE that has the form

$$\begin{aligned}\mathcal{T}\dot{\mathbf{v}}(t) &= \mathcal{A}\mathbf{v}(t) + \mathcal{B}_1 w(t) + \mathcal{B}_2 u(t), \quad \mathbf{v}(0) = \mathbf{v}_0 \in RL^{m,n} \\ z(t) &= \mathcal{C}_1 \mathbf{v}(t) + \mathcal{D}_{11} w(t) + \mathcal{D}_{12} u(t),\end{aligned}$$

where all the operators are 4-PI operators.

The problem of synthesizing an H_∞ optimal controller amounts to determining a PIE operator $u(t) = \mathcal{K}\mathbf{v}(t)$ such that the regulated output z admits $\|z\| \leq \gamma\|w\|$ for a particular $\gamma > 0$.

The following LPIs yield a feasible PI operators \mathcal{P} and \mathcal{Z} for which $\mathcal{K} := \mathcal{Z}\mathcal{P}^{-1}$ results in a minimum value of $\gamma > 0$ such that $\|z\| \leq \gamma\|w\|$.

$$\begin{aligned}& \min_{\gamma, \mathcal{P}} \quad \gamma \\& \mathcal{P} \succ 0 \\& \begin{bmatrix} -\gamma I & \mathcal{D}_{11} & (\mathcal{C}_1 \mathcal{P} + \mathcal{Z} \mathcal{D}_{12} \mathcal{Z}) \mathcal{T}^* \\ \mathcal{D}_{11}^* & -\gamma I & \mathcal{B}_1^* \\ ()^* & \mathcal{B}_1 & ()^* + (\mathcal{A} \mathcal{P} + \mathcal{B}_2 \mathcal{Z}) \mathcal{T}^* \end{bmatrix} \preceq 0\end{aligned}$$

The executive function for solving the above problem is called

`executive_PIETOOLS_Hinf_controller`

Part IV

PDEs: Representation, Analysis, Control, and Simulation

Chapter 13

A GUI for Defining PDEs

In this latest edition of PIETOOLS 2021a, we have gone the extra mile to ease the process of inputting a user-defined model! As a user, there is now no need to wrap your head around the new, extensive notations and definitions of define the model you aim to work on. Utilize the new feature, PIETOOLS Graphical User Interface, that provides a simple, intuitive and interactive visual interface to directly input the model. So take a seat back, relax, and let PIETOOLS take all the burden behind the background. You can open the GUI by simply clicking on `PIETOOLS_2021a.mlapp`. You see something similar to the picture below:

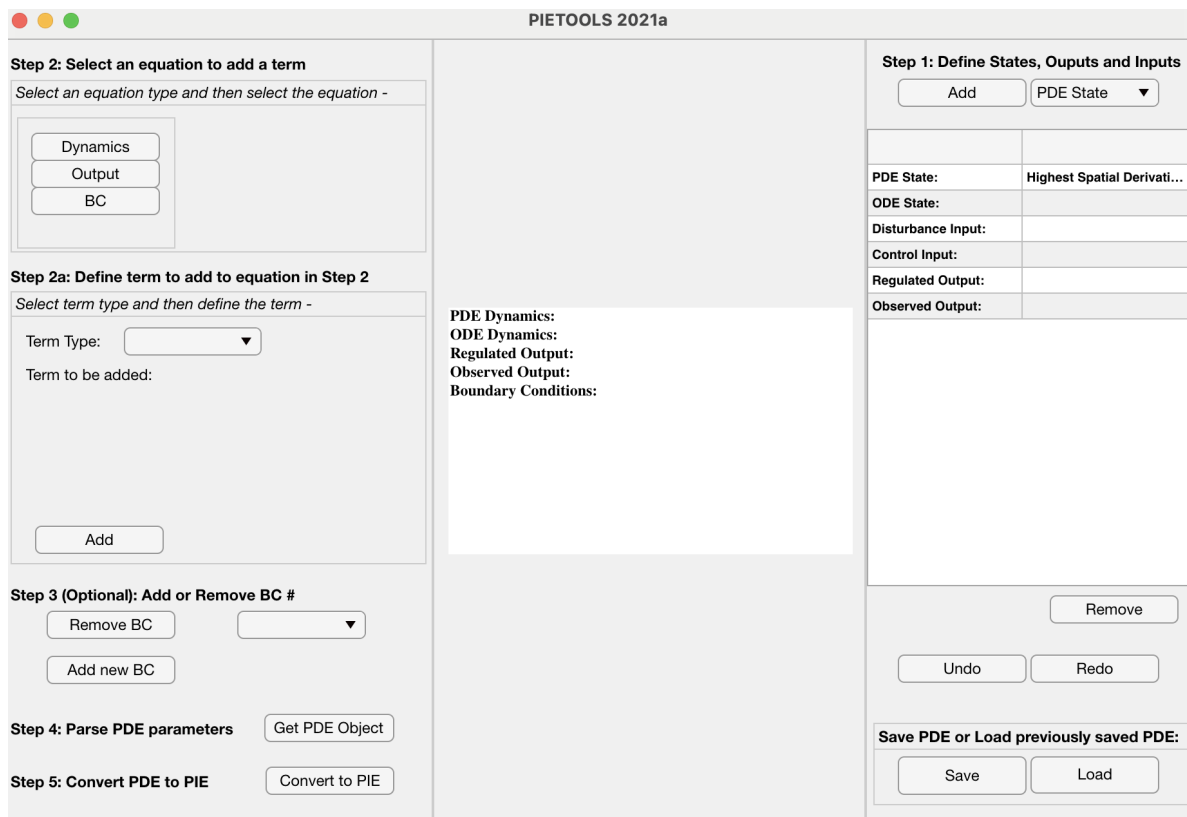


Figure 13.1: GUI overview.

Now we will go over the GUI step-by-step to demonstrate how to define your own model in PIETOOLS.

13.1 Step 1: Define States, Outputs and Inputs

First, we start with the right side of the screen as follows:

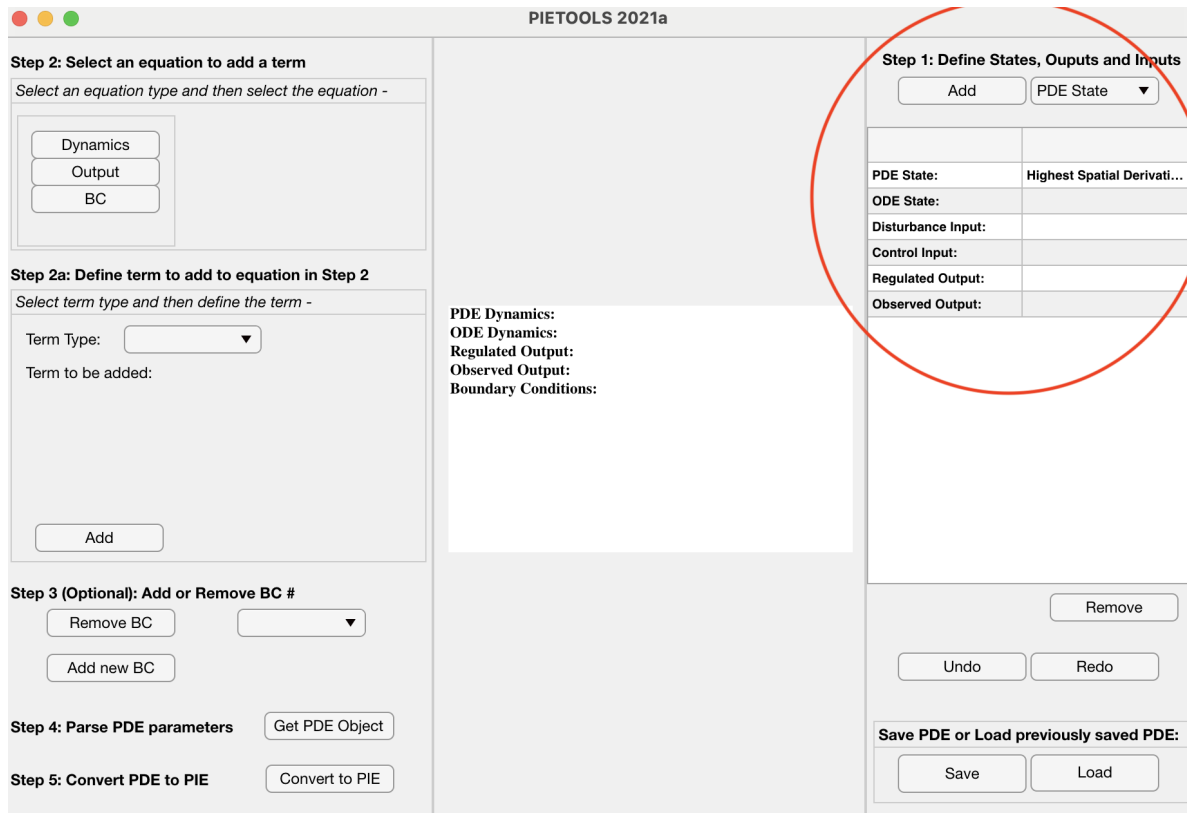


Figure 13.2: Step 1: Define States, Outputs and Inputs

1. The drop-down menu **PDE State** provides a list of all the possible variables to be defined on your model. Clicking on the **PDE State** menu reveals the list

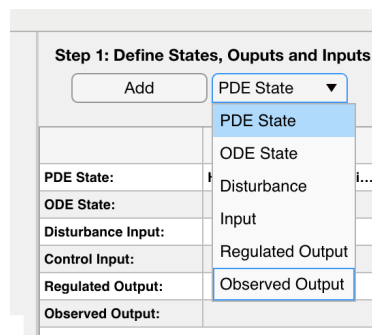
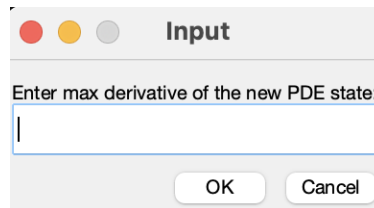


Figure 13.3: Adding variables your model

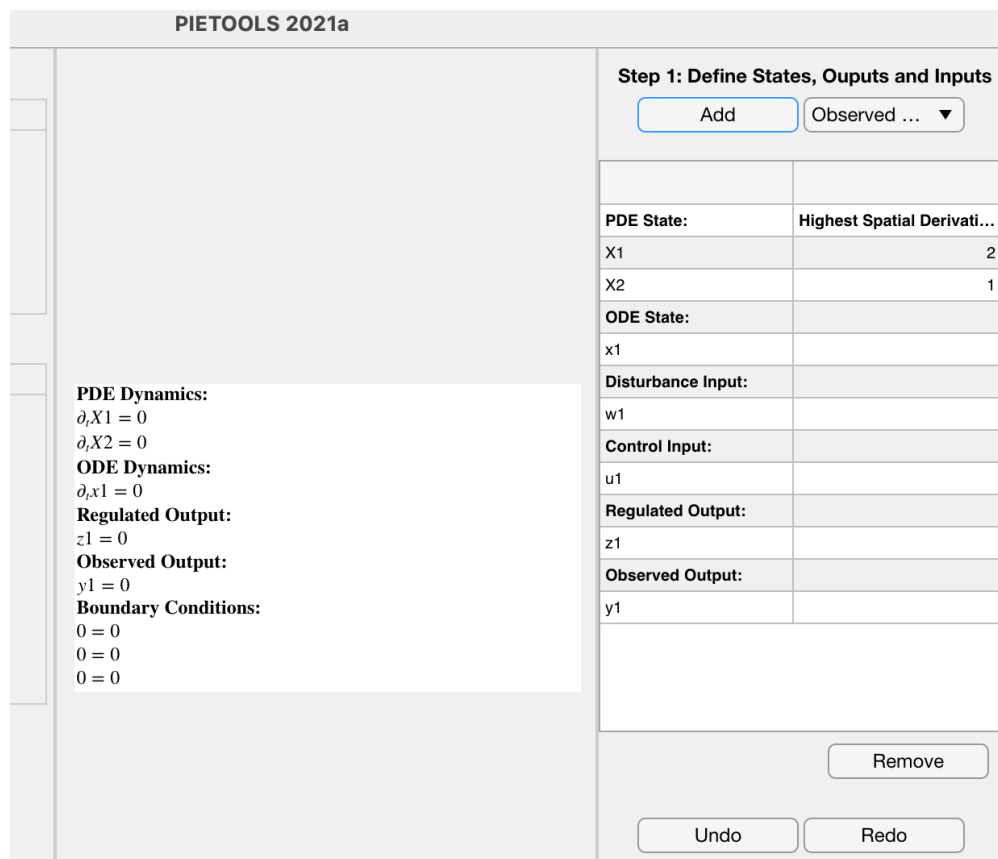
2. After selecting your intended variable, you can add them by clicking on the **Add** button.
3. Specifically, when you select **PDE State** from the drop-down menu and attempt to **PDE State**, you also have to specify what is highest order of derivative the particular state admits.



A small dialog box titled "Input" with a red, yellow, and grey window control bar. It contains a text prompt "Enter max derivative of the new PDE state:" followed by a text input field with a cursor. At the bottom are "OK" and "Cancel" buttons.

Figure 13.4: Enter the highest order of derivative the particular state admits

4. Once the variables are added, it automatically gets displayed in the display panel in the middle. As, no dynamics is specified to the model so far, all the variables are set to the default setting temporarily.



The main interface of PIETOOLS 2021a. On the left is a large display panel showing the current model configuration. On the right is a configuration panel titled "Step 1: Define States, Outputs and Inputs".

Display Panel (Left):

PDE Dynamics:
 $\partial_t X1 = 0$
 $\partial_t X2 = 0$

ODE Dynamics:
 $\partial_t x1 = 0$

Regulated Output:
 $z1 = 0$

Observed Output:
 $y1 = 0$

Boundary Conditions:
 $0 = 0$
 $0 = 0$
 $0 = 0$

Configuration Panel (Right):

Buttons: **Add**, **Observed ... ▼**

PDE State:	Highest Spatial Derivati...
X1	2
X2	1
ODE State:	
x1	
Disturbance Input:	
w1	
Control Input:	
u1	
Regulated Output:	
z1	
Observed Output:	
y1	

Buttons at the bottom: **Remove**, **Undo**, **Redo**

Figure 13.5: After adding the variables to the model

5. At the bottom there are options of **Remove**, **Undo**, **Redo** to delete or recover variables.

13.2 Step 2: Select an Equation to Add a Term

Now we specify the dynamics and the terms corresponding to each variables defined in Step 1. This is located on the left hand side of the GUI.

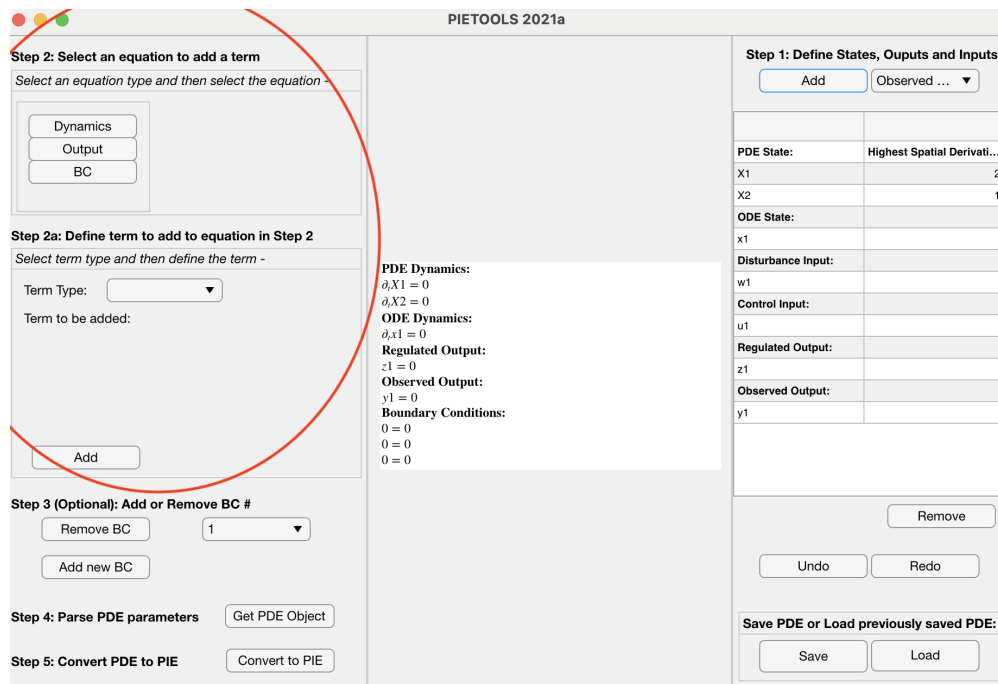
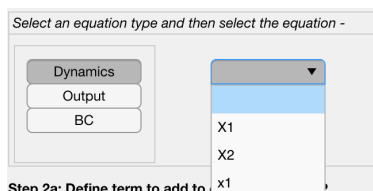


Figure 13.6: Step 2: Select an Equation to Add a Term

This has two parts. On the top, we have a panel for **Select an Equation type** and the **select the equation-** to choose which part of the model to be defined. Then, for each of the part, there an another panel below that has to be used to **Select term type** and then **define the term-**.

1. In the panel titled **Select an Equation type** and the **select the equation-**, select either **Dynamics**, **Output** or **BC** (i.e. Boundary Conditions).
2. If you select **Dynamics**, all the PDE and ODE states that you specified in Step 1 appears.



3. Once you select the desired state for which the dynamics term is needed to be added to, go down to the **Step 2.a Select term type** and then **define the term-**.

Step 2: Select an equation to add a term

Select an equation type and then select the equation -

Dynamics
Output
BC

X1 ▼

Step 2a: Define term to add to equation in Step 2

Select term type and then define the term -

Term Type: ▼
Term to be added: Standard
Integral

Add

- For individual PDE state, you may have two kinds of terms, a **Standard**, a multiplier coefficient associated with a state or a **Integral**, an integral term associated with a state.
- The **Standard** option allows to define all the multiplier terms associated with each variables. On the other hand, the **Integral** option is only available for the PDE states.

Term Type: Standard ▼
Term to be added:
Coefficient multiplying the state/input is entered in the box below
1
Add
Step 3 (Optional): Add or Remove BC
Remove BC
X1
X2
x1
w1
u1

▼

Step 2a: Define term to add to equation in Step 2

Select term type and then define the term -

Term Type: Integral ▼
Term to be added:
Coefficient multiplying the state/input is entered in the box below
s
1
0
Add
X1
X2

(θ ▼) | dθ ▼

- Now in **Standard** option, one has to select the variable and add the coefficient in the adjacent panel. Moreover, the PDE states may also contain its derivatives. If you select PDE state, you can input the order of derivative (from 0 up to the highest order derivative for that state), the independent variable with respect to which the function is defined (it is s for in-domain, 0, 1 for boundary), and the corresponding coefficient terms. Then, by clicking on the **Add** button, we can add that term to the model and it gets shown in the display panel readily.

Select term type and then define the term -

Term Type: Standard

Term to be added:

Coefficient multiplying the state/input is entered in the box below

s^2

2

∂_s

X1

(

s

)

Add

PDE Dynamics:

$\partial_t X1 = s^2 \partial_s^2 X1(s)$

$\partial_t X2 = 0$

ODE Dynamics:

$\partial_t x1 = 0$

Regulated Output:

$z1 = 0$

Observed Output:

$y1 = 0$

Boundary Conditions:

$0 = 0$

$0 = 0$

$0 = 0$

Note: Only the PDE states can be a function of 's'. For other terms option of adding 's' as an independent variable is not available. Moreover, the order of derivative can not exceed the highest order derivative for that state. If the input value exceed that, while adding it throws the following error

- One can also add an integral term by selecting **Integral**. Here, identical to the **Standard** option, we can define the order or derivative, the coefficient and the limits of integral which can be 0 or s for lower limit and s or 1 for upper limit. The functions are always with respect to θ .

Select term type and then define the term -

Term Type: Integral

Term to be added:

Coefficient multiplying the state/input is entered in the box below

s

0

\int

theta^3+3

∂_s

X2

(

θ

)

d θ

0

Add

PDE Dynamics:

$\partial_t X1 = s^2 \partial_s^2 X1(s) + \int_0^s (\theta^3 + 3) X2(\theta) d\theta$

$\partial_t X2 = 0$

ODE Dynamics:

$\partial_t x1 = 0$

Regulated Output:

$z1 = 0$

Observed Output:

$y1 = 0$

Boundary Conditions:

$0 = 0$

$0 = 0$

$0 = 0$

Note: The integral term of PDE states can only be a function of ' θ '. Moreover, the order of derivative can not exceed the highest order derivative for that state. If the input value exceed that, while adding it throws an error.

Note: Addition of these terms are made for one variable at a time. Once you select the corresponding variable and one of the options (Dynamics, Output, BC) from the top, the terms can be added from the bottom part (Step 2.a). To select an another variable for which you aim to add terms, one must repeat the above steps.

48

Step 2: Select an equation to add a term

Select an equation type and then select the equation -

Dynamics

Output

BC

X2 ▼

Step 2a: Define term to add to equation in Step 2

Select term type and then define the term -

Term Type: Standard ▼

Term to be added:

Coefficient multiplying the state/input is entered in the box below

1

5*s

∂_s

X2 ▼

(s ▼)

Add

PDE Dynamics:

1. $\partial_t X1 = s^2 \partial_s^2 X1(s) + \int_0^s (\theta^3 + 3) X1(\theta) d\theta$

2. $\partial_t X2 = 5s \partial_s X2(s)$

ODE Dynamics:

3. $\partial_t x1 = 0$

Regulated Output:

4. $z1 = 0$

Observed Output:

5. $y1 = 0$

Boundary Conditions:

6. $0 = 0$

7. $0 = 0$

8. $0 = 0$

8. To define the outputs and boundary conditions, one must follow the same steps as above.

Step 2: Select an equation to add a term

Select an equation type and then select the equation -

Dynamics

Output

BC

▼

z1

y1

Step 2: Select an equation to add a term

Select an equation type and then select the equation -

Dynamics

Output

BC

6 ▼

6

7

8

Step 2a: Define term to add to

Select term type and then define

After adding all the corresponding terms for dynamics, outputs and boundary conditions, the complete description of an example model looks something like below:

PDE Dynamics:

1. $\partial_t X1 = s^2 \partial_s^2 X1(s) + \int_0^s (\theta^3 + 3) X1(\theta) d\theta$
2. $\partial_t X2 = 5s \partial_s X2(s)$

ODE Dynamics:

3. $\partial_t x1 = 0$

Regulated Output:

4. $z1 = \int_0^1 (s^2 + s) X1(s) ds$

Observed Output:

5. $y1 = X2(0)$

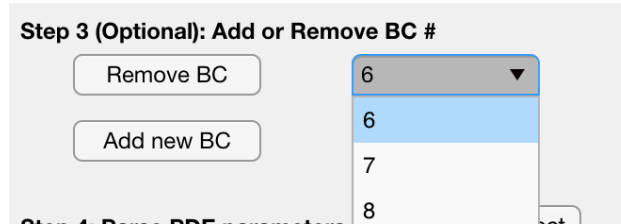
Boundary Conditions:

6. $0 = \partial_s X1(0)$
7. $0 = X1(1)$
8. $0 = X2(1)$

Figure 13.7: An example of a complete model as displayed in the GUI

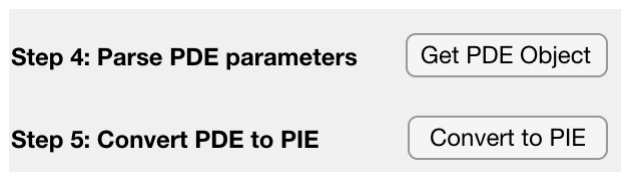
13.3 Step 3: (Optional) Add or Remove BC

As an option, the user can either add a new boundary condition or remove one. Note that the number of boundary condition has to be coherent with the number of state variables.



13.4 Step 4-5: Parse PDE Parameters and Convert Them to PIE

1. Now clicking **Get PDE Object**, you can extract all the parameters related to your model and store them in an object called **PDE_GUI** which directly gets loaded into the MATLAB workspace.
2. Now clicking **convert to PIE**, you can convert your model to PIE and store them in an object called **PIE_GUI** which directly gets loaded into the MATLAB workspace.



Chapter 14

The PDE Input Formats

14.1 The Batch Input Format

The batch input format restricts the types of terms which can be used, but is useful for defining large systems of relatively uncomplicated linear 1-D PDEs. This input format is also compatible with PIETOOLS 2020a. In this format, there are relatively few options and we can split the governing equations into three parts, which define the dynamics of the ODE, dynamics of the PDE, the output, and the boundary conditions. Any term not specified in the PDE structure is assumed to be zero by the initialization routine. In PIETOOLS 2020a, the ODE-PDE coupled models had the following form:

$$\begin{bmatrix} z(t) \\ y(t) \\ \dot{x}(t) \\ \dot{\mathbf{x}}(t) \end{bmatrix} = \begin{bmatrix} D_{11} & D_{12} & C_1 & \mathcal{C}_{1p} \\ D_{21} & D_{22} & C_2 & \mathcal{C}_{2p} \\ B_{11} & B_{12} & A & \mathcal{E}_p \\ \mathcal{B}_{21} & \mathcal{B}_{22} & \mathcal{E} & \mathcal{A}_p \end{bmatrix} \begin{bmatrix} w(t) \\ u(t) \\ x(t) \\ \mathbf{x}(t) \end{bmatrix}. \quad (14.1)$$

The model is stored in an object named `PDE`. The dimension of each variable is defined as follows:

1. `PDE.nx` number of ODE states x
2. `PDE.nw` number of disturbances w
3. `PDE.nu` number of inputs u
4. `PDE.nz` number of regulated outputs z
5. `PDE.ny` number of observed outputs y

Moreover, the PDE state \mathbf{x} is partitioned based on the order differentiation it admits.

1. `PDE.n0` number of undifferentiated PDE states \mathbf{x}_0
2. `PDE.n1` number of single differentiated PDE states \mathbf{x}_1
3. `PDE.n2` number of twice differentiated PDE states \mathbf{x}_1
4. `PDE.a` starting point of the spatial interval
5. `PDE.b` end point of the spatial interval

14.1.1 Dynamics of the PDE

PDE dynamics is defined by the operator \mathcal{A}_p which is defined as follows

$$\begin{aligned}
(\mathcal{A}_p \mathbf{x})(s) &:= A_0(s) \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} (s) + A_1(s) \partial_s \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} (s) + A_2(s) \partial_s^2 [\mathbf{x}_2] (s). \\
(\mathcal{E}x)(s) &:= E(s)x. \\
(\mathcal{B}_{21}w)(s) &:= B_{21}(s)w. \\
(\mathcal{B}_{22}u)(s) &:= B_{22}(s)u.
\end{aligned} \tag{14.2}$$

1. PDE.A0 matrix function of s of dimension n0+n1+n2 x n0+n1+n2
2. PDE.A1 matrix function of s of dimension n0+n1+n2 x n1+n2
3. PDE.A2 matrix function of s of dimension n0+n1+n2 x n2
4. PDE.E pvar matrix in variable s of dimension n0+n1+n2 x nx
5. PDE.B21 polynomial matrix in pvar s of dimension (n0+n1+n2) x nw
6. PDE.B22 polynomial matrix in pvar s of dimension (n0+n1+n2) x nu

14.1.2 Dynamics of the ODE

The ODE dynamics is given by \mathcal{E}_p according to

$$\begin{aligned}
(\mathcal{E}_p \mathbf{x})(s) &:= \int_a^b \left(E_a(s) \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} (s) + E_b(s) \partial_s [\mathbf{x}_1 \mathbf{x}_2] (s) + E_c(s) \partial_s^2 [\mathbf{x}_2] (s) \right) ds \\
&\quad + E_0 \begin{bmatrix} \mathbf{x}_1(a) \\ \mathbf{x}_1(b) \\ \mathbf{x}_2(a) \\ \mathbf{x}_2(b) \\ \partial_s \mathbf{x}_2(a) \\ \partial_s \mathbf{x}_2(b) \end{bmatrix},
\end{aligned} \tag{14.3}$$

Moreover, A, B_{11}, B_{12} are constant matrices

1. PDE.E0 matrix of dimension nx x 2*n1+4*n2
2. PDE.Ea polynomial matrix in pvar s of dimension nx x (n0+n1+n2)
3. PDE.Eb polynomial matrix in pvar s of dimension nx x n1+n2
4. PDE.Ec polynomial matrix in pvar s of dimension nx x n2
5. PDE.A matrix of dimension nx x nx
6. PDE.B11 matrix of dimension nx x nw
7. PDE.B12 matrix of dimension nx x nu

14.1.3 The Output Equation

The output operators \mathcal{C}_{1p} and \mathcal{C}_{2p} are defined as follows:

$$(\mathcal{C}_{1p} \mathbf{x})(s) := \int_a^b \left(C_{a1}(s) \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} (s) + C_{b1}(s) \partial_s \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} (s) + C_{c1}(s) \partial_s^2 [\mathbf{x}_2] (s) \right) ds$$

$$\begin{aligned}
& + C_{10} \begin{bmatrix} \mathbf{x}_1(a) \\ \mathbf{x}_1(b) \\ \mathbf{x}_2(a) \\ \mathbf{x}_2(b) \\ \partial_s \mathbf{x}_2(a) \\ \partial_s \mathbf{x}_2(b) \end{bmatrix}, \\
(\mathcal{C}_{2p} \mathbf{x})(s) &:= \int_a^b \left(C_{a2}(s) \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} (s) + C_{b2}(s) \partial_s \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} (s) + C_{c2}(s) \partial_s^2 [\mathbf{x}_2] (s) \right) ds \\
& + C_{20} \begin{bmatrix} \mathbf{x}_1(a) \\ \mathbf{x}_1(b) \\ \mathbf{x}_2(a) \\ \mathbf{x}_2(b) \\ \partial_s \mathbf{x}_2(a) \\ \partial_s \mathbf{x}_2(b) \end{bmatrix}, \tag{14.4}
\end{aligned}$$

Moreover, $C_1, C_2, D_{11}, D_{12}, D_{21}, D_{22}$ are constant matrices.

1. PDE.C10 matrix of dimension $\text{nz} \times 2^*n1+4^*n2$
2. PDE.Ca1 polynomial matrix in pvar s of dimension $\text{nz} \times (n0+n1+n2)$
3. PDE.Cb1 polynomial matrix in pvar s of dimension $\text{nz} \times n1+n2$
4. PDE.Cc1 polynomial matrix in pvar s of dimension $\text{nz} \times n2$
5. PDE.C1 matrix of dimension $\text{nz} \times \text{nx}$
6. PDE.D11 matrix of dimension $\text{nz} \times \text{nw}$
7. PDE.D12 matrix of dimension $\text{nz} \times \text{nu}$
8. PDE.C20 matrix of dimension $\text{ny} \times 2^*n1+4^*n2$
9. PDE.Ca2 polynomial matrix in pvar s of dimension $\text{ny} \times (n0+n1+n2)$
10. PDE.Cb2 polynomial matrix in pvar s of dimension $\text{ny} \times n1+n2$
11. PDE.Cc2 polynomial matrix in pvar s of dimension $\text{ny} \times n2$
12. PDE.C2 matrix of dimension $\text{ny} \times \text{nx}$
13. PDE.D21 matrix of dimension $\text{ny} \times \text{nw}$
14. PDE.D22 matrix of dimension $\text{ny} \times \text{nu}$

14.1.4 The Boundary Conditions

The boundary conditions are defined by

$$\begin{bmatrix} B_w & B_u & B_x & \mathcal{B}_c \end{bmatrix} \begin{bmatrix} w(t) \\ u(t) \\ x(t) \\ \mathbf{x}(t) \end{bmatrix} = 0, \tag{14.5}$$

where,

$$(\mathcal{B}_c \mathbf{x}) := -B \begin{bmatrix} \mathbf{x}_1(a) \\ \mathbf{x}_1(b) \\ \mathbf{x}_2(a) \\ \mathbf{x}_2(b) \\ \partial_s \mathbf{x}_2(a) \\ \partial_s \mathbf{x}_2(b) \end{bmatrix} \quad (14.6)$$

Moreover, B_w, B_u, B_x are constant matrices.

1. `PDE.B` matrix of dimension $n1+2*n2 \times n0+n1+n2$
(must have row rank $n1+2n2$ and contain no prohibited boundary conditions)
2. `PDE.Bw` matrix of dimension $n1+2*n2 \times n_w$ (must be full row rank)
3. `PDE.Bu` matrix of dimension $n1+2*n2 \times n_u$ (must be full row rank)
4. `PDE.Bx` matrix of dimension $n1+2*n2 \times n_x$

14.2 The Term-Based Input Format

To define an ODE-PDE system in PIETOOLS_2021a, we use a Matlab struct called `PDE`. This structure contains the following fields:

1. `PDE.vars` Contains information concerning the variables used by PIETOOLS.
2. `PDE.dom` Contains information concerning the domain of the PDE.
3. `PDE.n` Contains information concerning the size of our PDE.
4. `PDE.ODE` Contains information concerning the ODE dynamics.
5. `PDE.PDE` Contains information concerning the PDE dynamics.
6. `PDE.BC` Contains information concerning the boundary conditions (BCs).

We will describe how each of these fields should be specified in the following sections.

14.2.1 The Variables and their Domain

The first two fields describing our PDE are `vars` and `dom`, specifying the spatial variables used in the PDE and the domain on which they exist, respectively. Crucially, `PDE.vars` should always be set to `['s'; 'theta']`, specifying s and θ as variables, independent of what your PDE may look like on paper. This is because these variables are used by various codes in the toolbox, and changing them may lead to errors. As such, we suggest you **do NOT specify the value of `PDE.vars`, and do NOT change its value at any point**. Once you convert your PDE to a PIE, PIETOOLS will automatically add these variables to your structure `PDE`.

What you do need to specify, though, is the spatial domain $[a, b]$ on which your PDE exists. This domain can be specified assigning `PDE.dom.a` the value of your lower boundary, and `PDE.dom.b` that of your upper boundary.

14.2.2 The Size of the Problem

The third field describing your ODE-PDE concerns the size of the variables involved. This field is split into several subcomponents as:

<code>PDE.n.nx</code>	Integer specifying the number of ODE state variables.
<code>PDE.n.nw</code>	Integer specifying the number of disturbances in the ODE.
<code>PDE.n.nu</code>	Integer specifying the number of controlled inputs in the ODE.
<code>PDE.n.nz</code>	Integer specifying the number of regulated outputs of the ODE.
<code>PDE.n.ny</code>	Integer specifying the number of observed outputs of the ODE.
<code>PDE.n.nv</code>	Integer specifying the number of ODE to PDE interconnection signals.
<code>PDE.n.nr</code>	Integer specifying the number of PDE to ODE interconnection signals.
<code>PDE.n.n_pde</code>	Array specifying the number of PDE state variables as divided according to order differentiability.

For most of these elements, assigning an explicit value is not required, so long as you describe the dynamics correctly. For example, for a simple system

$$\begin{aligned}\dot{x}(t) &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

so long as you specify the matrices A , B , C and D correctly (as described in the next section), PIETOOLS will be able to derive the values of `no`, `nu` and `ny` from the size of your matrices. If you do not specify any other terms, the fields `nw`, `nz`, `nv` and `nr` will also be assigned a value zero upon converting your PDE to a PIE. However, **the field `PDE.n.n_pde` must always be specified**. In order to do this, you must first split the PDE state into $N + 1$ components as

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_N \end{bmatrix} \in \begin{bmatrix} L_2 \\ H_1 \\ \vdots \\ H_N \end{bmatrix},$$

where \mathbf{x}_n for $n \in \{0, \dots, N\}$ contains all state variables differentiable up to order n , so that N corresponds to the highest order of differentiability of any state variable. Then, the size of the PDE state can be specified by setting the n th element of `PDE.n.n_pde` equal to the size of state component x_n . For example, for a PDE

$$\begin{aligned}\dot{\omega}(t, s) &= \partial_s \phi(t, s) \\ \dot{\phi}(t, s) &= \partial_s \psi(t, s) \\ \dot{\psi}(t, s) &= -\partial_s \phi(t, s) \partial_s^3 \omega(t, s)\end{aligned}$$

we see that state variable ω must be three times differentiable, whilst state variables ϕ and ψ need only be once differentiable. Our PDE state in this case may thus be given as

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_3 \end{bmatrix} = \begin{bmatrix} \phi \\ \psi \\ \omega \end{bmatrix},$$

so we can assign a size to our state in PIETOOLS by setting:

```
PDE.PDE.n_pde = [0,2,0,1]
```

Note here that, our array is of size $N + 1 = 3 + 1 = 4$, as the highest order of differentiability of any state variable is 3. Also, we specify the lack of zeroth and second order differentiable variables by setting the corresponding elements in `PDE.PDE.n_pde` equal to zero. It is important to keep track of the order of differentiability of your state components, as this will also be used when defining the actual dynamics in section 4.

14.2.3 The ODE

The fourth field of our PDE struct, the field `PDE.ODE`, is used to describe the dynamics of the ODE component of our ODE-PDE system. This field assumes the ODE to be of the form:

$$\begin{bmatrix} \dot{x}(t) \\ z(t) \\ y(t) \\ v(t) \end{bmatrix} = \begin{bmatrix} A & B_{xw} & B_{xu} & B_{xr} \\ C_z & D_{zw} & D_{zu} & D_{zr} \\ C_y & D_{yw} & D_{yu} & D_{yr} \\ C_v & D_{vw} & D_{vu} & 0 \end{bmatrix} \begin{bmatrix} x(t) \\ w(t) \\ u(t) \\ r(t) \end{bmatrix}, \quad (14.7)$$

with x denoting our ODE state, w , u and r denoting disturbances, controlled inputs, and PDE to ODE interconnection signals, and z , y and v denoting regulated outputs, observed outputs, and ODE to PDE interconnection signals. To specify this ODE, we simply have to assign values to each of the matrices in this system:

```
PDE.ODE.A       $n_x \times n_x$  array mapping  $x$  to  $\dot{x}$ .
PDE.ODE.Bxw     $n_x \times n_w$  array mapping  $w$  to  $\dot{x}$ .
PDE.ODE.Bxu     $n_x \times n_u$  array mapping  $u$  to  $\dot{x}$ .
PDE.ODE.Bxr     $n_x \times n_r$  array mapping  $r$  to  $\dot{x}$ .
PDE.ODE.Cz      $n_z \times n_x$  array mapping  $x$  to  $z$ .
PDE.ODE.Dzw     $n_z \times n_w$  array mapping  $w$  to  $z$ .
PDE.ODE.Dzu     $n_z \times n_u$  array mapping  $u$  to  $z$ .
PDE.ODE.Dzr     $n_z \times n_r$  array mapping  $r$  to  $z$ .
PDE.ODE.Cy      $n_y \times n_x$  array mapping  $x$  to  $y$ .
PDE.ODE.Dyw     $n_y \times n_w$  array mapping  $w$  to  $y$ .
PDE.ODE.Dyu     $n_y \times n_u$  array mapping  $u$  to  $y$ .
PDE.ODE.Dyr     $n_y \times n_r$  array mapping  $r$  to  $y$ .
PDE.ODE.Cv      $n_v \times n_x$  array mapping  $x$  to  $v$ .
PDE.ODE.Dvw     $n_v \times n_w$  array mapping  $w$  to  $v$ .
PDE.ODE.Dvu     $n_v \times n_u$  array mapping  $u$  to  $v$ .
```

Note that there is no ODE map from r to v , as this relation is described by the PDE. Now, to specify the ODE, you need only assign values to the nonzero matrices in the standardized format. For example, for our simple system

$$\begin{aligned} \dot{x}(t) &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

we only need to specify the values of A , B_{xu} , C_y and D_{yu} . Then, upon converting the PDE to a PIE, PIETOOLS will assign all other fields a zero array of appropriate size. Of course, for the matrices that you do define, it is important that their sizes match up (e.g. A must have the same number of rows as B_{xu}), as otherwise PIETOOLS will set your matrix to zero. This also means that, if you do insist on specifying a size value such as `PDE.n.no` yourself, you must make sure this value matches the size of the matrices that you define, as PIETOOLS will assume the value assigned to `PDE.n.no` to be correct, and adjust the fields of `PDE.ODE` accordingly.

14.2.4 The PDE

The fifth field of the `PDE` struct specifies the actual PDE dynamics, and the interconnection with the ODE. We will describe each of these inputs in separate subsections.

The Dynamics

To describe the PDE dynamics, we once again split our PDE state \mathbf{x} into $N + 1$ components $\mathbf{x}_0, \dots, \mathbf{x}_N$ ordered according to differentiability, and consider the evolution equation for each component separately. In particular, for each component $\ell \in \{0, \dots, N\}$, we assume this evolution equation to be of the form

$$\dot{\mathbf{x}}_\ell(t, s) = \sum_{k=0}^N \sum_{d=0}^k \sum_{i=0}^2 \int_i A_{idk\ell}(s, \theta) \partial_\theta^d \mathbf{x}_k(t, \theta) d\theta + \sum_{k=1}^N \sum_{d=0}^{k-1} \sum_{j=0}^1 B_{pb,jdk\ell}(s) \Delta_j \partial_s^d \mathbf{x}_r(t, s) + B_{pv,\ell}(s) v(t) \quad (14.8)$$

where we define the integral \int_i for $i \in \{0, 1, 2\}$, and the Delta operator Δ_j for $j \in \{0, 1\}$ such that, for any function $F \in L_2[a, b]$,

$$\int_0 F(\theta) d\theta = F(s), \quad \int_1 F(\theta) d\theta = \int_a^s F(\theta) \theta, \quad \int_2 F(\theta) d\theta = \int_s^b F(\theta) \theta,$$

and

$$\Delta_0 F = F(a), \quad \Delta_1 F = F(b).$$

Then, our structure `PDE.PDE` can be used to specify the different functions A , B_{pb} and B_{pv} , being described by fields:

- `PDE.PDE.A` Cell specifying the contribution of the PDE state to the evolution equation.
- `PDE.PDE.Bpb` Cell specifying the contribution of the boundary state to the evolution equation.
- `PDE.PDE.Bpv` Polynomial object or array specifying the contribution of the ODE-PDE interconnection signal to the evolution equation.

Let us start with `PDE.PDE.A`. This field is described by the cell, in of which each element corresponds to a single term in the evolution equation of some state component. For example, consider an arbitrary element m of this cell. This element will be a struct, with the

following fields:

<code>PDE.PDE.A{m}.Lstate</code>	Integer specifying which evolution equation the term is added to.
<code>PDE.PDE.A{m}.Rstate</code>	Integer specifying which state component is considered in the term.
<code>PDE.PDE.A{m}.D</code>	Integer specifying what derivative of the state component is considered.
<code>PDE.PDE.A{m}.I</code>	Integer specifying what integral of the term is considered.
<code>PDE.PDE.A{m}.coeff</code>	Polynomial object or array specifying the pre-multiplication factor of the term.

Now, we can add a term to the evolution equation of some state component $\ell \in \{0, \dots, N\}$ as

$$\underbrace{\mathbf{x}_\ell(t, s)}_{\text{Lstate}} \stackrel{+}{=} \underbrace{\int_i}_{\text{I}} \left(\underbrace{A_{idk\ell}(s, \theta)}_{\text{coeff}} \underbrace{\partial_\theta^d}_{\text{D}} \underbrace{\mathbf{x}_k(t, \theta)}_{\text{Rstate}} \right) d\theta$$

by specifying

```

PDE.PDE.A{m}.Lstate = ell
PDE.PDE.A{m}.Rstate = k
PDE.PDE.A{m}.D = d
PDE.PDE.A{m}.I = i
PDE.PDE.A{m}.coeff = A_pol

```

where `A_pol` is some polynomial (or array, or double) specifying the function (or matrix, or scalar) $A_{idk\ell}(s, \theta)$. Note that, it does not matter what element m of the cell we are assigning these values to, and so it also doesn't matter in what order you specify your terms. Also, you need not specify combinations of `Lstate`, `Rstate`, `D` and `I` that do not appear in your PDE, as PIETOOLS will automatically set these to zero. However, any term that does appear in your system must be assigned to a separate cell element, and any cell element must correspond to a single term. For example, for the simple PDE

$$\begin{aligned}
\dot{\phi}(s, t) &= \partial_s \psi(t, s) \\
\dot{\psi}(s, t) &= -\partial_s \phi(t, s) \partial_s^3 \omega(t, s) \\
\dot{\omega}(s, t) &= \partial_s \phi(t, s)
\end{aligned}$$

with state components $\mathbf{x}_1 = \begin{bmatrix} \phi \\ \psi \end{bmatrix}$ and $\mathbf{x}_3 = \omega$, we may write it in the standardized format as

$$\begin{aligned}
\dot{\mathbf{x}}_1(t, s) &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \partial_s \mathbf{x}_1(t, s) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \partial_s^3 \mathbf{x}_3(t, s) \\
\dot{\mathbf{x}}_3(t, s) &= \begin{bmatrix} 1 & 0 \end{bmatrix} \partial_s \mathbf{x}_1(t, s)
\end{aligned}$$

and we can implement this by setting

```

PDE.PDE.A{1}.Lstate = 1      PDE.PDE.A{2}.Lstate = 1      PDE.PDE.A{3}.Lstate = 3
PDE.PDE.A{1}.Rstate = 1      PDE.PDE.A{2}.Rstate = 3      PDE.PDE.A{3}.Rstate = 1
PDE.PDE.A{1}.D = 1           PDE.PDE.A{2}.D = 3           PDE.PDE.A{3}.D = 1
PDE.PDE.A{1}.I = 0           PDE.PDE.A{2}.I = 0           PDE.PDE.A{3}.I = 0
PDE.PDE.A{1}.coeff = [0,1;-1,0] PDE.PDE.A{2}.coeff = [0;1] PDE.PDE.A{3}.coeff = [1,0]

```

Then, exchanging the order of any of these cell elements (e.g. `PDE.PDE.A{1}` and `PDE.PDE.A{2}`) will still specify the same PDE. However, if you were to divide the information of the first element `PDE.PDE.A{1}` over two elements as

```

PDE.PDE.A{1}.Lstate = 1      PDE.PDE.A{4}.Lstate = 1
PDE.PDE.A{1}.Rstate = 1      PDE.PDE.A{4}.Rstate = 1
PDE.PDE.A{1}.D = 1           PDE.PDE.A{4}.D = 1
PDE.PDE.A{1}.I = 0           PDE.PDE.A{4}.I = 0
PDE.PDE.A{1}.coeff = [0,1;0,0] PDE.PDE.A{4}.coeff = [0,0;-1,0]

```

(e.g. to separate the contributions of ϕ and ψ), PIETOOLS will get rid of one of these elements. This is because they have the same values of `Lstate`, `Rstate`, `D` and `I`, specifying a single term in the evolution equation, and must thus be assigned a single cell element.

Now, to specify contributions of the boundary state to the evolution equation, we use the cell `PDE.PDE.Bpb`, which is structured in a very similar way to `PDE.PDE.A`. That is, an arbitrary element m of this cell is a struct with fields

<code>PDE.PDE.Bpb{m}.Lstate</code>	Integer specifying which evolution equation the term is added to.
<code>PDE.PDE.Bpb{m}.Rstate</code>	Integer specifying which state component is considered in the term.
<code>PDE.PDE.Bpb{m}.D</code>	Integer specifying what derivative of the state component is considered.
<code>PDE.PDE.Bpb{m}.delta</code>	Integer specifying what at what boundary the term is evaluated.
<code>PDE.PDE.Bpb{m}.coeff</code>	Polynomial object or array specifying the pre-multiplication factor of the term.

replacing the integral field `I` with a field `delta` specifying whether the term should be evaluated at the lower boundary `PDE.dom.a` (setting `delta=0`) or the upper boundary `PDE.dom.b` (setting `delta=1`). Then, we can add a term to the evolution equation of some state component $\ell \in \{0, \dots, N\}$ as

$$\underbrace{\dot{\mathbf{x}}_\ell(t, s)}_{\text{Lstate}} \stackrel{+}{=} \underbrace{B_{pb,jdk\ell}(s)}_{\text{coeff}} \underbrace{\Delta_j}_{\text{delta}} \underbrace{\partial_s^d}_{\text{D}} \underbrace{\mathbf{x}_k(t, s)}_{\text{Rstate}}$$

by specifying

```

PDE.PDE.Bpb{m}.Lstate = ell
PDE.PDE.Bpb{m}.Rstate = k
PDE.PDE.Bpb{m}.D = d
PDE.PDE.Bpb{m}.delta = j
PDE.PDE.Bpb{m}.coeff = B_pol

```

where `B_pol` is some polynomial (or array, or double) specifying the function (or matrix, or scalar) $B_{jdk\ell}(s, \theta)$. Once again, it does not matter in what order these cell elements are specified, so long as each term is assigned a single element, and each element corresponds to

a single term.

Finally, we consider the contribution of the ODE-PDE interconnection signal to the evolution equation. For a particular state component $\ell \in 0, \dots, N$, this contribution is defined by a matrix-valued function $B_{pv,\ell}$ as

$$\dot{\mathbf{x}}_\ell(t, s) = \dots B_{pv,\ell}(s)v(t)$$

To specify this contribution in PIETOOLS, you have to concatenate the different polynomials $B_{pv,\ell}$ into a single object

$$B_{pv} = \begin{bmatrix} B_{pv,0} \\ \vdots \\ B_{pv,N} \end{bmatrix}$$

which can then be assigned to `PDE.PDE.Bpv`. Note that it is important to concatenate the different functions $B_{pv,\ell}$ in accordance with the way the different components of your state are concatenated, increasing in order of differentiability.

The PDE to ODE Output

We now consider how to specify the equation for the PDE to ODE output signal r . For this, we assume the equation may be written in the form

$$r(t) = \sum_{k=0}^N \sum_{d=0}^k \int_a^b C_{rp,dk}(s) \partial_s^d \mathbf{x}_k(t, s) ds + \sum_{k=1}^N \sum_{d=0}^{k-1} \sum_{j=0}^1 D_{rb,jdk} \Delta_j \partial_s^d \mathbf{x}_r(t, s) + D_{rv} v(t)$$

where we define the Delta operator Δ_j as before. Then, we can specify the different function C_{rp} and matrices D_{rb} and D_{rv} in PIETOOLS by assigning values to fields:

- `PDE.PDE.Crp` Cell specifying the contribution of the PDE state to the PDE to ODE signal.
- `PDE.PDE.Drb` Cell specifying the contribution of the boundary state to the PDE to ODE signal.
- `PDE.PDE.Drv` Array specifying the contribution of the ODE-PDE signal to the PDE to ODE signal.

Here, the cell `PDE.PDE.Crp` is structured very similarly to cell `PDE.PDE.A`, with each element m being specified by fields `Rstate`, `D` and `coeff`, so that a term

$$r(t) \stackrel{+}{=} \int_a^b \left(\underbrace{C_{rp,dk}(s)}_{\text{coeff}} \underbrace{\partial_s^d}_{\text{D}} \underbrace{\mathbf{x}_k(t, s)}_{\text{Rstate}} \right) ds$$

may be added to the output by specifying some cell m as

```
PDE.PDE.Crp{m}.Rstate = k
PDE.PDE.Crp{m}.D = d
PDE.PDE.Crp{m}.coeff = C_pol
```

with `C_pol` describing the desired matrix-valued polynomial $C_{rp,dk}$. Similarly, the field `PDE.PDE.Drb` is set up in the same way as `PDE.PDE.Brpb`, so that a term

$$r(t) \stackrel{+}{=} \underbrace{D_{rb,jdk}}_{\text{coeff}} \underbrace{\Delta_j}_{\text{delta}} \underbrace{\partial_s^d}_{\text{D}} \underbrace{\mathbf{x}_k(t, s)}_{\text{Rstate}}$$

can be added to the output by specifying

```
PDE.PDE.Drb{m}.Rstate = k
PDE.PDE.Drb{m}.D = d
PDE.PDE.Drb{m}.delta = j
PDE.PDE.Drb{m}.coeff = D_mat
```

where `D_mat` describes the desired matrix $D_{rb,jdk}$. Finally, to add a contribution $D_{rv}v(t)$ to the output, simply set `PDE.PDE.Drv` equal to this matrix, and you are done!

14.2.5 The Boundary Conditions

The final elements that must be specified (no PDE is complete without them) are the boundary conditions. To describe these, we assume (once again) that our PDE state has been split into different components according to differentiability, and that the boundary conditions are written as

$$0 = \sum_{k=0}^N \sum_{d=0}^k \int_a^b E_{bp,dk}(s) \partial_s^d \mathbf{x}_k(t, s) ds + \sum_{k=1}^N \sum_{d=0}^{k-1} \sum_{j=0}^1 E_{bb,jdk} \Delta_j \partial_s^d \mathbf{x}_r(t, s) + E_{bv}v(t)$$

Each of the components in this system can then be described by assigning appropriate values to:

- `PDE.BC.Ebp` Cell specifying the contribution of the PDE state to the boundary conditions.
- `PDE.BC.Ebb` Cell specifying the contribution of the boundary state to the boundary conditions.
- `PDE.BC.Ebv` Polynomial object or array specifying the contribution of the ODE to PDE signal to the boundary conditions.

Here, the fields `PDE.BC.Ebp` is structured identically to cell `PDE.PDE.Crpb`, with each element m being specified by fields `Rstate`, `D` and `coeff`, so that a term

$$0 \stackrel{\pm}{=} \int_a^b \left(\underbrace{E_{bp,dk}(s)}_{\text{coeff}} \underbrace{\partial_s^d}_{\text{D}} \underbrace{\mathbf{x}_k(t, s)}_{\text{Rstate}} \right) ds$$

may be added to the boundary conditions by specifying some cell m as

```
PDE.BC.Ebp{m}.Rstate = k
PDE.BC.Ebp{m}.D = d
PDE.BC.Ebp{m}.coeff = E_pol
```

where `E_pol` describes the desired polynomial $E_{bp,dk}$. Note that this array `E_pol` must always have a total number of rows equal to the total number of boundary conditions considered

in your system, even if the particular state component `Rstate=k` appears in only one (or a few) of these conditions. This can (of course) be achieved by setting all rows equal to zero that correspond to BCs in which this particular component does not appear. Furthermore, remember that for any state variable differentiable up to order n , you must add n boundary conditions to your system, so that the total number of BCs is always equal to `sum(PDE.n.n_pde(1:end))`.

Now, regarding the field `PDE.BC.Ebb`, this cell is structured identically to cell `PDE.PDE.Drb`, so that a term

$$0 \pm \underbrace{E_{bb,jdk}}_{\text{coeff}} \underbrace{\Delta_j}_{\text{delta}} \underbrace{\partial_s^d}_{\text{D}} \underbrace{\mathbf{x}_k(t,s)}_{\text{Rstate}}$$

may be added to the boundary conditions by specifying some cell m as

```
PDE.BC.Ebb{m}.Rstate = k
PDE.BC.Ebb{m}.D = d
PDE.BC.Ebb{m}.delta = j
PDE.BC.Ebb{m}.coeff = Ebb_mat
```

with `Ebb_mat` describing the desired matrix. Once again, the number of rows of this matrix must be equal to the total number of boundary. Finally, to add a term $E_bvv(t)$ to the boundary conditions, simply set `PDE.BC.Ebv` equal to the desired matrix.

14.3 Initializing your PDE

Using the format presented above, you can specify any 1D ODE-PDE system you desire to PIETOOLS. In doing so, note that you do not need to specify terms that do not appear in your PDE. If your system has no ODE subcomponent, you don't need to define `PDE.ODE`. If your PDE contains no terms involving the boundary values, you don't need to define the field `PDE.Bpb`. If your PDE involves only a single term, you only need to define a single element of `PDE.PDE.A`. That being said, in order to convert your PDE to a PIE, PIETOOLS will require all of these fields to be specified. For this reason, we have created the script `initialize_PIETOOLS_PDE_terms`, which will fill in all the blanks and rearrange the different cell elements to be in the correct order. This file will be automatically run when you convert your PDE to a PIE using `convert_PIETOOLS_PDE_terms`, so you do not need to worry about initializing yourself. However, initialization will produce several warnings if you did not specify all the fields, informing you of some of the changes that might have been made. Note also that, because of these changes, your PDE struct may look somewhat different after initialization (with the cell elements having been rearranged and such), so be careful with making changes after initialization.

For some examples of the PDE input format, see the `examples_PDE_library_PIETOOLS` file. Also, check out the `GetStarted_DEMO` file for an idea on how to set up a simple PDE stability test. Finally, don't forget that there is a GUI in which you can define your PDE, and which will convert it to the proper format. Have fun!

Chapter 15

Library of PDE Example Problems

To help you get started with PIETOOLS, and give you something to do when you're bored, we have implemented a variety of PDEs for you. These systems are described in the function `examples_PDE_library_PIETOOLS.m`, and include common PDEs, and examples from the literature. Each example has been specified in each of the three input formats: using the GUI, using the batch input format, and using the term-based input format. This allows you to see how the same PDE may be specified in the different formats, hopefully giving you a better understanding of how each of the formats works.

To get the PDE structure corresponding to one of the examples, the first thing you will need to do is open the actual file `examples_PDE_library_PIETOOLS.m`, and check the header. Each example has been assigned a label, and listed in a "table of contents" at the top of the file. In this table, the center column describes the PDE with boundary conditions that is considered, with the left-most column providing the label assigned to this system. The right-most column provides the default values of potential parameters occurring in the system, which we will get back to in a minute. Right of this column, just outside the table, notes and/or references concerning some of the systems are provided.

Now, say you want to load example 10 (good choice!). To do so, simply call the function with as argument your desired example:

```
>> PDE = examples_PDE_library_PIETOOLS(10);
```

Then, the output PDE will be a MATLAB structure describing the PDE example 10 from the file in the **batch** input format. If you want to get your PDE in the term-based input format, this can be achieved by specifying an optional argument '**terms**':

```
>> PDE = examples_PDE_library_PIETOOLS(10, 'terms');
```

Running this code, the output PDE struct will describe the same system, but using the term-based input format. Similarly, if you want to load the example into the GUI, you can use the optional argument '**GUI**':

```
>> examples_PDE_library_PIETOOLS(10, 'GUI');
```


Note that in this case, we get no output structure. Instead, a file describing the system is loaded into the GUI, at which point you can use the `Get_PDE_Object` button to add the corresponding PDE structure `PDE_GUI` (in term-based input format) to your workspace. If you want to compare the different input structures, you can also specify multiple optional arguments. For example, running

```
>> [PDE_batch,PDE_terms] = examples_PDE_library_PIETOOLS(10,'batch','terms');
```

the output consists of two structures, with `PDE_batch` describing the PDE in batch format, and `PDE_terms` describing the PDE in terms-based format. Here, the order in which you specify the formats will determine the order in which you receive them, so that we may achieve the same result by running:

```
>> [PDE_terms,PDE_batch] = examples_PDE_library_PIETOOLS(10,'terms','batch');
```

Also, if you prefer if you're not a fan of using characters, you can replace `'batch'` by a value 1, `'terms'` by a value 2, and `'GUI'` by a value 3. Thus, running

```
>> [PDE_terms,PDE_batch] = examples_PDE_library_PIETOOLS(10,2,1);
```

will produce the same result.

When you run one of the examples, you may note that a message such as `stability=1`, `Hinf_gain=1`, `Hinf_estimator=1` or `Hinf_control=1` appears in your command window. This is because each system corresponds to a particular problem, which is either stability analysis, H_∞ -gain computation, or constructing an H_∞ -optimal observer or controller. Each of these problems has been specified as an LPI in PIETOOLS, implemented in the “executives” folder. By specifying e.g. `stability=1`, the script `PIETOOLS_PDE` in the root directory will know that you wish to perform stability analysis, and will call the corresponding executable. Of course, if you want to solve another problem for some example PDE, for example computing the H_∞ -gain, you can specify this after running the `examples` function by setting `Hinf_gain=1`. In doing so, keep in mind that not every problem is well-defined for each example. For example, there's no sense in computing an H_∞ -gain if your system has no outputs.

Let's return now to the parameters mentioned earlier. Certain examples, such as our example 10, rely on parameters to be specified. In any such case, default parameters have been set, often corresponding to the values presented in the paper from which an example might have been extracted. However, if you want to use different parameters, it is also possible to specify these as optional arguments. For example, our example 10 relies on a parameter R to be specified, which is defaulted to a value $R = 30$. If you wish to change this value to, say, $R = 20$, you can do so by adding the argument `'R=20'` to your function call. That is, running

```
>> PDE = examples_PDE_library_PIETOOLS(10,'batch','R=20');
```

your output structure `PDE` will describe example 10 in the batch format, and with parameter value $R = 20$. If you want to specify the value of multiple parameters, you can do so by

adding multiple arguments, each corresponding to a different parameter. Note that this will not adjust the parameters used in the GUI definition of the system, as the GUI will load a predefined file.

As a warning, please realize that the way in which the function executes the parameter specifications allows you to specify all kinds of nonsense, that may not produce the desired result, or even adversely affect your program. As such, we strongly advise you to only specify values of parameters presented in the right-most column of the table of contents. Each row in this column describes the parameter names, along with their default values. If no parameters are mentioned, you shouldn't specify any values. If you do specify parameters, make sure that the name you specify does indeed match a parameter name for your considered example **exactly** (case-sensitive). Otherwise, the default value will be retained, and you may be adjusting some internal value necessary for proper execution. Remember: With great power, comes great responsibility.

Chapter 16

Analysis, Control, and Estimation of PDEs

The workflow for analysis, estimation and control of PDEs is specified in the `PIETOOLS_PDE` script file placed in the root folder. To use the script `PIETOOLS_PDE` for analysis, first, define the PDE using any of the methods discussed in the previous chapter. Analysis can be performed on PDEs either in batch format or terms format, however, the name of the variable must be `PDE` (other names cannot be used with the script).

16.1 Defining the PDE object

At this point we have discussed four different ways you can specify a PDE in `PIETOOLS`: using the GUI (highly recommended), using the batch input format (useful for larger 2nd order systems), using the term-based input format (useful for larger higher order systems), and by simply calling an example from the library (if you want to see how this works). Using examples defined in the `examples.PIETOOLS_PDE` function automatically defined the PDE object in workspace under the name `PDE`, whereas using the GUI creates the PDE object under the name `PDE_GUI`. If the PDE object is manually defined using either batch or terms format, it is recommended that the variable be named `PDE`.

16.2 Convert to a PIE

Once the PDE object has been defined, we can determine different properties of PDEs by solving an appropriate LPI. However, to use the LPIs PDE objects must be converted to PIEs objects first by using conversion functions (see [21.1](#)) or through the GUI.

For example, if the PDE object has been defined using the GUI, PIE can be obtained by using the "Convert to PIE" button which dumps the PIE object in the workspace as `PIE_GUI`. Alternatively, if the PDE is stored under a variable name `PDE_sys` in terms format, then the following command converts it to a PIE object.

```
PIE = convert_PIETOOLS_PDE_terms(PDE_sys);
```

Binary variable specification	Operation in PIETOOLS_PDE
<code>stability=1</code>	Test stability of the provided PDE
<code>stability_dual=1</code>	Test stability using dual LPI method
<code>Hinf_gain=1</code>	Compute the H_∞ gain of the specified PDE
<code>Hinf_gain_dual=1</code>	Compute the H_∞ gain using dual LPI method
<code>Hinf_estimator=1</code>	Construct an H_∞ -optimal observer for the specified PDE
<code>Hinf_controller=1</code>	Construct an H_∞ -optimal controller for the specified PDE

Table 16.1: Binary variables available to specify a PDE analysis problem

Similarly, for batch format, the following command can be used.

```
PIE = convert_PIETOOLS_PDE_batch(PDE_sys);
```

16.3 Choose Executive Mode

After obtaining the PIE representation of the PDE, we can solve LPIs to test stability, compute the H_∞ gain, and search for optimal controllers and observers for the PDE (see Chapters 4 and 10). Each of these LPIs has been implemented in a function in the “executives” folder, allowing you to test a property by calling the appropriate function. However, we recommend you use the script `PIETOOLS_PDE` (in the root directory) instead.

The executive mode, or the LPI to be tested, is specified by defining the binary variables presented in Table 16.1, either running the command in your command line, or uncommenting the corresponding command in the `PIETOOLS_PDE` script.

16.4 Specify Settings and SDP Solver

In addition to specifying what test you wish to perform, you can also specify the solver settings used when executing this test by uncommenting certain lines. In particular, recall that in solving LPIs, the PI operators are parameterized by matrices using monomials up to a certain degree. If this degree is higher, you are expanding the domain of feasible solutions in your program, increasing the accuracy but also the computational complexity. For parameters such as this degree, different options have been predefined in the `settings_PIETOOLS` scripts, which you can call by uncommenting the corresponding line in the `PIETOOLS_PDE` script. For example, uncommenting `settings_PIETOOLS_extreme` rather than the default `settings_PIETOOLS_light`, your program may be more successful, but this will come at a higher computational cost. As such, we suggest you adapt the settings according to your problem: if light settings do not yield the expected result, try heavier settings. Keep in mind that optimal controller or observer constructing may also need more effort than stability analysis. The root script `PIETOOLS_PDE` default the settings to light by running the script `settings_PIETOOLS_light`. These settings specify rather simple opvar variables in the LPI and should be sufficient for all simple PDE problems. However, if you have a more

difficult example, spatially varying coefficients, or require additional accuracy (at the cost of additional computation time), simply comment out this script and uncomment ONE of the following.

```
settings_PIETOOLS_heavy;
settings_PIETOOLS_veryheavy;
```

Alternatively, if you have a very large-scale systems with lots of states but constant coefficients, you may uncomment ONE of the following to decrease the computation time (at the cost of decreased accuracy).

```
settings_PIETOOLS_stripped;
settings_PIETOOLS_extreme;
```

In addition to these settings, you can also specify how positive your PI operators should be in each LPI. That is, PIETOOLS can only restrict PI operators to be positive (negative) semidefinite, not to be strictly positive (negative) definite. To get enforce this strict condition, rather than enforcing $\mathcal{P} \succcurlyeq 0$, we enforce $\mathcal{P} \succcurlyeq \epsilon$ for some small value $\epsilon \ll 1$, ensuring the PI operator \mathcal{P} is strictly positive. You can specify the value of epsilon for both the positivity of the Lyapunov function and the negativity of its derivative, by changing the default value in the script PIETOOLS_PDE. Finally, you may tinker with the settings by altering and running the script `settings_PIETOOLS_custom`.

There are several supported SDP solvers. By default the root script runs SeDuMi. However, if you have one of the other supported SDP solvers installed, you may specify this solver by commenting `settings.sos_opts.solver='sedumi'` and uncommenting ONE of the following.

```
settings.sos_opts.solver='mosek';
settings.sos_opts.solver='sdpnalplus';
settings.sos_opts.solver='sdpt3';
```

16.5 Solving the problem, interpreting the output, and implementing the controller

If the LPI is successfully solved, the opvar (PI operator) solving the LPI will be added to your workspace, named `P`. The solved SOS program `prog` will also be added to your workspace, along with potential additional outputs. For example, if the H_∞ gain was successfully computed, this gain will be specified by `gamma` in the workspace. If you set `Hinf_estimator=1` or `Hinf_controller=1`, the optimal observer or controller will be specified by opvar `L` or `K` respectively, provided `settings.option1.sep=1` is chosen. Whatever output you get, be

sure to check whether your problem was actually solved (for example by looking at the `prog.solinfo`, as the presented output may not always a valid solution if the optimization problem was infeasible.

- The first output you will notice when running the root script is the initialization algorithms setting all undeclared parameters to zero.
- Next, the chosen executive will set up the LPI and convert it to an LMI. This is arguably the slowest part of the code. If it is taking too long, you can reduce the complexity by using one of the settings script.
- Next, the SDP will be passed to your desired SDP solver and the solver will execute. The output format differs with solver. However, you should take note of whether the solver is able to find a feasible solution. For sedumi, this means a fearatio of around +1.
- After the SDP solver has completed, the executive will summarize the result. For stability analysis, this is a binary answer. For H_∞ gain analysis, optimal estimation, and optimal control, the summary will state the bound on the closed or open loop H_∞ -gain as appropriate.
- Finally, for the optimal control and estimation problems, the executive will construct the feedback gains (\mathcal{K} or \mathcal{L}) in the form of a 4PI-operator. For an optimal controller, this means the control input should be

$$u(t) = \mathcal{K} \begin{bmatrix} x(t) \\ \mathbf{x}_0(t) \\ \vdots \\ \partial_s^N \mathbf{x}_N(t) \end{bmatrix}$$

where recall $\mathbf{x}_i(t, s)$ is the i -times differentiable PDE state and x is the ODE state from the PDE representation. The construction of observer gains is similar. The feedback gains are output from the executive function and placed in the workspace only if settings are chosen such that `settings.option1.sep=1`. Of course, the simplest way to implement the controller is to simply construct the closed loop PIE using, e.g. $\mathcal{A}_{cl} = \mathcal{A} + \mathcal{B}_2 \mathcal{K}$ which can, alternatively obtained by using the function `closedLoopPIE`.

Part V

Delayed Systems: Representation, Analysis and Control

Chapter 17

Constructing and Representing Systems with Delay

In contrast to PDE systems, systems with delay have a widely recognized input format. For this reason, the input format for systems with delay is only available in batch mode. However, there are three distinct types of delay system which can be included in PIETOOLS. We address each of these separately. In each case, the problem data should be included in a DDE, NDS, or DDF structure and terms which are not included are assumed to be zero.

17.1 The Delay Differential Equation (DDE) Format

The DDE data structure allows the user to declare any of the matrices in the following general form of Delay-Differential equation.

$$\begin{aligned} \begin{bmatrix} \dot{x}(t) \\ z(t) \\ y(t) \end{bmatrix} &= \begin{bmatrix} A_0 & B_1 & B_2 \\ C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{bmatrix} \begin{bmatrix} x(t) \\ w(t) \\ u(t) \end{bmatrix} + \sum_{i=1}^K \begin{bmatrix} A_i & B_{1i} & B_{2i} \\ C_{1i} & D_{11i} & D_{12i} \\ C_{2i} & D_{21i} & D_{22i} \end{bmatrix} \begin{bmatrix} x(t - \tau_i) \\ w(t - \tau_i) \\ u(t - \tau_i) \end{bmatrix} \\ &+ \sum_{i=1}^K \int_{-\tau_i}^0 \begin{bmatrix} A_{di}(s) & B_{1di}(s) & B_{2di}(s) \\ C_{1di}(s) & D_{11di}(s) & D_{12di}(s) \\ C_{2di}(s) & D_{21di}(s) & D_{22di}(s) \end{bmatrix} \begin{bmatrix} x(t+s) \\ w(t+s) \\ u(t+s) \end{bmatrix} ds \end{aligned} \quad (17.1)$$

In this representation, it is understood that

- The present state is $x(t)$.
- The disturbance or exogenous input is $w(t)$. These signals are not typically known or alterable. They can account for things like unmodelled dynamics, changes in reference, forcing functions, noise, or perturbations.
- The controlled input is $u(t)$. This is typically the signal which is influenced by an actuator and hence can be accessed for feedback control.
- The regulated output is $z(t)$. This signal typically includes the parts of the system to be minimized, including actuator effort and states. These signals need not be measured using sensors.

- The observed or sensed output is $y(t)$. These are the signals which can be measured using sensors and fed back to an estimator or controller.

To add any term to the DDE structure, simply declare its value. For example, to represent

$$\dot{x}(t) = -x(t-1), \quad z(t) = x(t-2)$$

we use

```
DDE.tau = [1 2];
```

```
DDE.A{1} = -1;
```

```
DDE.C1{2} = 1;
```

All terms not declared are assumed to be zero. The exception is that we require the user to specify the values of the delay in `DDE.tau`. When you are done adding terms to the DDE structure, use the function `DDE=PIETOOLS_initialize_DDE(DDE)`, which will check for undeclared terms and set them all to zero. It also checks to make sure there are no incompatible dimensions in the matrices you declared and will return a warning if it detects such malfeasance. The complete list of terms and DDE structural elements is listed in Table 17.1.

ODE Terms:					
Eqn. (17.1)	DDE.	Eqn. (17.1)	DDE.	Eqn. (17.1)	DDE.
A_0	A0	B_1	B1	B_2	B2
C_1	C1	D_{11}	D11	D_{12}	D12
C_2	C2	D_{21}	D21	D_{22}	D22
Discrete Delay Terms:					
Eqn. (17.1)	DDE.	Eqn. (17.1)	DDE.	Eqn. (17.1)	DDE.
A_i	Ai{i}	B_{1i}	B1i{i}	B_{2i}	B2i{i}
C_{1i}	C1i{i}	D_{11i}	D11i{i}	D_{12i}	D12i{i}
C_{2i}	C2i{i}	D_{21i}	D21i{i}	D_{22i}	D22i{i}
Distributed Delay Terms: May be functions of pvar s					
Eqn. (17.1)	DDE.	Eqn. (17.1)	DDE.	Eqn. (17.1)	DDE.
A_{di}	Adi{i}	B_{1di}	B1di{i}	B_{2di}	B2di{i}
C_{1di}	C1di{i}	D_{11di}	D11di{i}	D_{12di}	D12di{i}
C_{2di}	C2di{i}	D_{21di}	D21di{i}	D_{22di}	D22di{i}

Table 17.1: Equivalent names of Matlab elements of the DDE structure terms for terms in Eqn. (17.1). For example, to set term `XX` to `YY`, we use `DDE.XX=YY`. In addition, the delay τ_i is specified using the vector element `DDE.tau(i)` so that if $\tau_1 = 1, \tau_2 = 2, \tau_3 = 3$, then `DDE.tau=[1 2 3]`.

17.1.1 Initializing a DDE Data structure

The user need only add non-zero terms to the DDE structure. All terms which are not added to the data structure are assumed to be zero. Before conversion to another representation or data structure, the data structure will be initialized using the command

```
DDE = initialize_PIETOOLS_DDE(DDE)
```

This will check for dimension errors in the formulation and set all non-zero parts of the DDE data structure to zero. Not that, to make the code robust, all PIETOOLS conversion utilities perform this step internally.

17.2 Input of Neutral Type Systems

The input format for a Neutral Type System (NDS) is identical to that of a DDE except for 6 additional terms:

$$\begin{aligned} \begin{bmatrix} \dot{x}(t) \\ z(t) \\ y(t) \end{bmatrix} &= \begin{bmatrix} A_0 & B_1 & B_2 \\ C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{bmatrix} \begin{bmatrix} x(t) \\ w(t) \\ u(t) \end{bmatrix} + \sum_{i=1}^K \begin{bmatrix} A_i & B_{1i} & B_{2i} & E_i \\ C_{1i} & D_{11i} & D_{12i} & E_{1i} \\ C_{2i} & D_{21i} & D_{22i} & E_{2i} \end{bmatrix} \begin{bmatrix} x(t - \tau_i) \\ w(t - \tau_i) \\ u(t - \tau_i) \\ \dot{x}(t - \tau_i) \end{bmatrix} \\ &+ \sum_{i=1}^K \int_{-\tau_i}^0 \begin{bmatrix} A_{di}(s) & B_{1di}(s) & B_{2di}(s) & E_{di}(s) \\ C_{1di}(s) & D_{11di}(s) & D_{12di}(s) & E_{1di}(s) \\ C_{2di}(s) & D_{21di}(s) & D_{22di}(s) & E_{2di}(s) \end{bmatrix} \begin{bmatrix} x(t+s) \\ w(t+s) \\ u(t+s) \\ \dot{x}(t+s) \end{bmatrix} ds \end{aligned} \quad (17.2)$$

These new terms are parameterized by E_i , E_{1i} , and E_{2i} for the discrete delays and by E_{di} , E_{1di} , and E_{2di} for the distributed delays. As for the DDE case, these terms should be included in a NDS object as, e.g. `NDS.E{1}=1`.

17.2.1 Initializing a NDS Data structure

The user need only add non-zero terms to the NDS structure. All terms which are not added to the data structure are assumed to be zero. Before conversion to another representation or data structure, the data structure will be initialized using the command

```
NDS = initialize_PIETOOLS_NDS(NDS)
```

This will check for dimension errors in the formulation and set all non-zero parts of the NDS data structure to zero. Not that, to make the code robust, all PIETOOLS conversion utilities perform this step internally.

17.3 The Differential Difference Equation (DDF) Format

A Differential Difference Equation (DDF) is a more general representation than either the DDE or NDS. Most importantly, unlike the DDE or NDS, it allows one to represent the

structure of the delayed channels. As such, it is the only representation for which the minimal realization features of PIETOOLS are defined. Nevertheless, the general form of DDF is more compact than that of the DDE or NDS. The distinguishing feature of the DDF is decomposition of the output signal from the ODE part into sub-components, r_i , each of which is delayed by amount τ_i . Identification of these r_i is often challenging and hence most users will input the system as an ODE or NDS and then convert to a minimal DDF representation. The form of a DDF is given as follows.

$$\begin{bmatrix} \dot{x}(t) \\ z(t) \\ y(t) \\ r_i(t) \end{bmatrix} = \begin{bmatrix} A_0 & B_1 & B_2 \\ C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \\ C_{ri} & B_{r1i} & B_{r2i} \end{bmatrix} \begin{bmatrix} x(t) \\ w(t) \\ u(t) \end{bmatrix} + \begin{bmatrix} B_v \\ D_{1v} \\ D_{2v} \\ D_{rvi} \end{bmatrix} v(t)$$

$$v(t) = \sum_{i=1}^K C_{vi} r_i(t - \tau_i) + \sum_{i=1}^K \int_{-\tau_i}^0 C_{vdi}(s) r_i(t + s) ds. \quad (17.3)$$

As for a DDE or NDS, each of the non-zero parameters in Eqn. (17.3) should be added to the DDF structure, along with the vector of values of the delays `DDF.tau`. The elements of the DDF structure which can be defined by the user are included in Table 17.3.

17.3.1 Initializing a DDF Data structure

The user need only add non-zero terms to the DDF structure. All terms which are not added to the data structure are assumed to be zero. Before conversion to another representation or data structure, the data structure will be initialized using the command

```
DDF = initialize_PIETOOLS_DDF(DDF)
```

This will check for dimension errors in the formulation and set all non-zero parts of the DDF data structure to zero. Not that, to make the code robust, all PIETOOLS conversion utilities perform this step internally.

ODE Terms:							
Eqn. (17.2)	NDS.	Eqn. (17.2)	NDS.	Eqn. (17.2)	NDS.		
A_0	A0	B_1	B1	B_2	B2		
C_1	C1	D_{11}	D11	D_{12}	D12		
C_2	C2	D_{21}	D21	D_{22}	D22		

Discrete Delay Terms:							
Eqn. (17.2)	NDS.	Eqn. (17.2)	NDS.	Eqn. (17.2)	NDS.	Eqn. (17.2)	NDS.
A_i	Ai{i}	B_{1i}	B1i{i}	B_{2i}	B2i{i}	E_i	Ei{i}
C_{1i}	C1i{i}	D_{11i}	D11i{i}	D_{12i}	D12i{i}	E_{1i}	E1i{i}
C_{2i}	C2i{i}	D_{21i}	D21i{i}	D_{22i}	D22i{i}	E_{2i}	E2i{i}

Distributed Delay Terms: May be functions of pvar s							
Eqn. (17.2)	NDS.	Eqn. (17.2)	NDS.	Eqn. (17.2)	NDS.	Eqn. (17.2)	NDS.
A_{di}	Adi{i}	B_{1di}	B1di{i}	B_{2di}	B2di{i}	E_{di}	E di{i}
C_{1di}	C1di{i}	D_{11di}	D11di{i}	D_{12di}	D12di{i}	E_{1di}	E1di{i}
C_{2di}	C2di{i}	D_{21di}	D21di{i}	D_{22di}	D22di{i}	E_{2di}	E2di{i}

Table 17.2: Equivalent names of Matlab elements of the NDS structure terms for terms in Eqn. (17.2). For example, to set term XX to YY , we use $NDS.XX=YY$. In addition, the delay τ_i is specified using the vector element $NDS.tau(i)$ so that if $\tau_1 = 1, \tau_2 = 2, \tau_3 = 3$, then $NDS.tau=[1 \ 2 \ 3]$.

ODE Terms:							
Eqn. (17.3)	DDF.	Eqn. (17.3)	DDF.	Eqn. (17.3)	DDF.	Eqn. (17.3)	DDF.
A_0	A0	B_1	B1	B_2	B2	B_v	Bv
C_1	C1	D_{11}	D11	D_{12}	D12	D_{1v}	D1v
C_2	C2	D_{21}	D21	D_{22}	D22	D_{2v}	D2v
C_{ri}	Cri{i}	B_{r1i}	Br1i{i}	B_{r2i}	Br2i{i}	D_{rvi}	Drvi{i}

Discrete Delay Terms:							
Eqn. (17.3)	DDF.						
C_{vi}	Cvi{i}						

Distributed Delay Terms: May be functions of pvar s							
Eqn. (17.3)	DDF.						
$C_{vdi}(s)$	Cvdi{i}						

Table 17.3: Equivalent names of Matlab elements of the DDF structure terms for terms in Eqn. (17.3). For example, to set term XX to YY , we use $DDF.XX=YY$. In addition, the delay τ_i is specified using the vector element $DDF.tau(i)$ so that if $\tau_1 = 1, \tau_2 = 2, \tau_3 = 3$, then $DDF.tau=[1 \ 2 \ 3]$.

Chapter 18

Converting between DDEs, NDSs, DDFs, and PIEs

For a given delay system, there are several alternative representations of that system. For example, a DDE can be represented in the DDE, DDF, or PIE format. However, only the DDF and PIE formats allow one to specify structure in the delayed channels, which are infinite-dimensional. For that reason, it is almost always preferable to efficiently convert the DDE or NDS to either a DDF or PIE - as this will dramatically reduce computational complexity of the analysis, control, and simulation problems (assuming you have tools for analysis, control and simulation of DDFs and PIEs - which we do!). However, identifying an efficient DDF or PIE representation of a given DDF/NDS is laborious for large systems and requires detailed understanding of the DDF format. For this reason, we introduce a set of functions for automating this conversion process.

18.1 DDF to PIE

To convert a DDF data structure to an equivalent PIE representation, we have two utilities which are typically called sequentially. The first uses the SVD to identify and eliminate unused delay channels. The second naively converts a DDF to an equivalent PIE.

18.1.1 Minimal DDF Realization of a DDF

The typical first step in analysis, simulation and control of a DDF is elimination of unused delay channels. This is accomplished using the SVD to identify such channels in a DDF structure and output a smaller, equivalent DDF structure. To use this utility, simply declare your DDE and enter the command

```
DDF = minimize_PIETOOLS_DDF(DDF)
```

18.1.2 Converting a DDF to a PIE

Having constructed a minimal (or not) DDF representation of a DDE, NDS or DDF, the next step is conversion to an equivalent PIE. To use this utility, simply declare your DDF

structure and enter the command

```
PIE = convert_PIETOOLS_DDF2PIE(DDF)
```

18.2 DDE to DDF or PIE

We next address the problem of converting a DDE data structure to a DDF or PIE data structure. Because the DDE representation does not allow one to represent structure, this conversion is almost always performed using the `minimize_PIETOOLS_DDE2DDF` function followed possibly by `convert_PIETOOLS_DDF2PIE`. However, we also include a `convert_PIETOOLS_DDE2PIE` function, although we do not recommend using this feature.

18.2.1 Minimal DDF and PIE Realizations of DDEs

The utility `minimal_PIETOOLS_DDE2DDF` function takes a DDE data structure and constructs an equivalent DDF representation with associated structure. This utility uses the SVD to eliminate unused delay channels in the DDF - resulting in a much more compact representation of the same system. To use this utility, simply declare your DDE and enter the command

```
DDF = minimize_PIETOOLS_DDE2DDF(DDE)
```

After constructing the minimal DDF realization of the DDE, the user may then convert to a PIE using

```
PIE = convert_PIETOOLS_DDF2PIE(DDF)
```

18.2.2 DDE direct to PIE [NOT RECOMMENDED!]

Although it should never be used in practice, we also include a utility to construct the equivalent naïve PIE representation of a DDE. This is occasionally useful for purposes of comparison. To use this utility, simply declare your DDE and enter the command

```
DDF = convert_PIETOOLS_DDE2PIE(DDE)
```

Because of the limited utility of the unstructured representation, we have not included a naïve DDE to DDF utility.

18.3 NDS to DDF or PIE

Finally, we next address the problem of converting a NDS data structure to a DDF or PIE data structure. Like the DDE, the NDS representation does not allow one to represent structure and so the typical process involves 3 steps: direct conversion of the NDS to a DDF using `convert_PIETOOLS_NDS2DDF`; constructing a minimal representation of the resulting DDF using `minimize_PIETOOLS_DDF`; and conversion of the reduced DDF to a PIE (if desired) using `convert_PIETOOLS_DDF2PIE`. Since the latter two functions are already described in Section 21.1.7, we only describe the first utility.

18.3.1 Converting NDS to DDF

Given an NDS data structure, the user may construct an equivalent DDF data structure and associated representation using the utility

```
DDF = convert_PIETOOLS_NDS2DDF(NDS)
```

The typical user should follow this up with the utility

```
DDF = minimize_PIETOOLS_DDE2DDF(DDE)
```

After constructing the minimal DDF realization of the NDS, the user may then convert to a PIE using

```
PIE = convert_PIETOOLS_DDF2PIE(DDF)
```

Chapter 19

DDE, NDS, and DDF: Library of Examples

In this chapter, we give a brief explanation of the example problem libraries for DDEs, NDSs, and DDFs. At present, these example files are implemented as scripts (in contrast to the PDE example library). This means the user must edit the file to uncomment the desired example prior to calling the script. We expect to update this file in a future release to match the format used for the PDE example library.

19.1 DDE Examples

We have compiled a list of 23 DDE numerical examples, grouped into: stability analysis problems; input-output systems; Estimator design problems; and feedback control problems. These examples are drawn from the literature and citations are used to indicate the source of each example. For each group, the relevant flags have been included to indicate which executive mode should be called after the example has been loaded.

19.2 NDS and DDF Examples

There are relatively few DDFs which do not arise from a DDE or NDS. Hence, we have combined the DDF and NDS example libraries into the script `examples_NDSDDF_library_PIETOOLS`. The Neutral Type systems are listed first, and currently consist only of stability analysis problems - of which we include 13. As for the DDE case, the library is a script, so the user must uncomment the desired example and call the script from the root file or command window. For the NDS problems, after calling the example library, in order to convert the NDS to a DDF or PIE, the user can use the following commands:

```
NDS=initialize_PIETOOLS_NDS(NDS)
DDF=convert_PIETOOLS_NDS2DDF(NDS)
DDF=minimize_PIETOOLS_DDF(DDF)
PIE=convert_PIETOOLS_DDF2PIE(DDF)
```


In contrast to the NDS case, we only include 3 DDF examples. The first two are difference equations which cannot be represented in either the NDS or DDE format. The third is a network control problem, which is also included in the DDE library in DDE format.

Chapter 20

DDEs, NDSs, DDFs: Stability Analysis and Controller Synthesis

The workflow for analysis, estimation and control of DDEs, NDSs, or DDFs is included in the PIETOOLS_DDE and PIETOOLS_DDF root scripts.

20.1 Input DDE or DDF Representation

A description of the DDE and DDF input formats are included in the header of the root scripts PIETOOLS_DDE and PIETOOLS_DDF. A description of the NDS format is included at the end of the PIETOOLS_DDF root script. These formats are also described in Chapter 17. The user can input a desired DDE, NDS, or DDF by hand or may uncomment one of the examples in the examples script examples_DDE.PIETOOLS and run the script from the command line or root script.

20.2 Convert to a PIE

The Stability, Analysis, Estimation, and Control functions of PIETOOLS require the DDE/NDS/DDF to be first converted to PIE format. This is done in the root script, defaulting to a minimal realization of the DDE or NDS format. To override this option (Not recommended) and use the original DDE/NDS/DDF representation, use the command

```
DDE_minimal_rep=0
```

Otherwise, the root script PIETOOLS_DDE converts a DDE to PIE using

```
DDE=initialize_PIETOOLS_DDE(DDE);  
DDF=minimize_PIETOOLS_DDE(DDE)  
PIE=convert_PIETOOLS_DDF2PIE(DDF)
```

For a DDF, the script `PIETOOLS_DDF` converts a DDF to PIE using

```
DDF=initialize_PIETOOLS_DDF(DDF);  
DDF=minimize_PIETOOLS_DDF(DDF)  
PIE=convert_PIETOOLS_DDF2PIE(DDF)
```

For an NDS, the script `PIETOOLS_DDF` determines whether a data structure called `NDS` exists and if so, it converts it to a DDF using

```
DDF=convert_PIETOOLS_NDS2DDF(NDS);
```

The script then proceeds as for a DDF.

20.3 Choose Executive Mode

If the executive mode has not already been defined in the `examples_DDE_PIETOOLS` script, then the user should uncomment one of the following lines in the `PIETOOLS_DDE` root script.

```
stability=1;  
stability_dual=1;  
Hinf_gain=1;  
Hinf_gain_dual=1;  
Hinf_control=1;  
Hinf_estimator=1;
```

These options call the corresponding executive as described in Chapters [10](#), [11](#), and [12](#).

20.4 Specify Settings and SDP Solver

The root scripts `PIETOOLS_DDE` and `PIETOOLS_DDF` default the settings to light by running the script `settings_PIETOOLS_light`. These settings specify rather simple opvar variables in the LPI and should be sufficient for all but the most complex DDE, NDS, or DDF problems. However, if you have a more difficult example or require additional accuracy (at the cost of additional computation time), simply comment out this script and uncomment ONE of the following.

```
settings_PIETOOLS_heavy;  
settings_PIETOOLS_veryheavy;
```

Alternatively, if you have a very large-scale systems with lots of delayed channels, you may uncomment ONE of the following to decrease the computation time (at the cost of decreased accuracy).

```
settings_PIETOOLS_stripped;  
settings_PIETOOLS_extreme;
```

Finally, you may tinker with the settings by altering and running the script `settings_PIETOOLS_custom`.

There are several supported SDP solvers. By default the root script runs SeDuMi. However, if you have one of the other supported SDP solvers installed, you may specify this solver by commenting `settings.sos_opts.solver='sedumi'` and uncommenting ONE of the following.

```
settings.sos_opts.solver='mosek';  
settings.sos_opts.solver='sdpnalplus';  
settings.sos_opts.solver='sdpt3';
```

20.5 Solving the problem, interpreting the output, and implementing the controller

After all the proceeding steps have been completed, the root script calls the chosen executives as specified above. These executives perform the desired task and output the Lyapunov variables and controller/estimator gains as relevant. The following outputs are produced in the Matlab command window:

- The first output you will notice when running the root script is the initialization algorithms setting all undeclared parameters to zero.
- Next, the chosen executive will set up the LPI and convert it to an LMI. This may take some time, so please be patient. If it is taking too long, you can reduce the complexity by using one of the settings script.
- Next, the SDP will be passed to your desired SDP solver and the solver will execute. The output format differs with solver. However, you should take note of whether the solver is able to find a feasible solution. For sedumi, this means a feastratio of around +1.
- After the SDP solver has completed, the executive will summarize the result. For stability analysis, this is a binary answer. For H_∞ gain analysis, optimal estimation, and optimal control, the summary will state the bound on the closed or open loop H_∞ -gain as appropriate.

- Finally, for the optimal control and estimation problems, the executive will construct the feedback gains (\mathcal{K} or \mathcal{L}) in the form of a 4PI-operator. For an optimal controller, this means the control input should be

$$u(t) = \mathcal{K} \begin{bmatrix} x(t) \\ \partial_s \begin{bmatrix} r_1(t + \tau_1 s) \\ \vdots \\ r_K(t + \tau_K s) \end{bmatrix} \end{bmatrix} = \mathcal{K} \begin{bmatrix} x(t) \\ \partial_s \begin{bmatrix} \frac{1}{\tau_1} r_1(t + s) \\ \vdots \\ \frac{1}{\tau_K} r_K(t + s) \end{bmatrix} \end{bmatrix} = \mathcal{K} \begin{bmatrix} x(t) \\ \begin{bmatrix} \frac{1}{\tau_1} \dot{r}_1(t + s) \\ \vdots \\ \frac{1}{\tau_K} \dot{r}_K(t + s) \end{bmatrix} \end{bmatrix}.$$

where recall $r_i(t, s)$ is the delayed signal from the DDF representation. If you don't recall how r_i is defined, refer to Chapter 17 where we let

$$\begin{aligned} [r_i(t)] &= [C_{ri} \quad B_{r1i} \quad B_{r2i}] \begin{bmatrix} x(t) \\ w(t) \\ u(t) \end{bmatrix} + [D_{rvi}] v(t) \\ v(t) &= \sum_{i=1}^K C_{vi} r_i(t - \tau_i) + \sum_{i=1}^K \int_{-\tau_i}^0 C_{vdi}(s) r_i(t + s) ds. \end{aligned}$$

These parameters can be found in the DDF data structure created using `minimize_PIETOOLS_DDE` or `minimize_PIETOOLS_DDF`. The construction of observer gains is similar. The feedback gains are output from the executive function and placed in the workspace. Of course, the simplest way to implement the controller is to simply construct the closed loop PIE using, e.g. $\mathcal{A}_{cl} = \mathcal{A} + \mathcal{B}_2 \mathcal{K}$. This can be simulated using one the PIETOOLS PIE Simulators which will be released in the next version of PIETOOLS.

Part VI

List of Files, Functions and Scripts

Chapter 21

PIETOOLS Scripts: Initialize Systems, Convert To PIEs And More

A large number of applications involving 4-PI operators and LPIs revolve around analysis and control of infinite dimensional systems — more specifically, analysis and control of PIEs. In Chapter 5, we described to class of systems which can be converted to PIEs. Traditionally, these systems are not written in PIE form and reformulating them as PIEs usually requires many, tedious, steps. To alleviate the burden of conversion from the users, PIETOOLS performs all the heavy-lifting by providing ”helper” functions, that usually start as `convert_...`. These functions can convert systems from a standard form to PIE form provided all the parameters are appropriately defined. Additionally, there are `initialize_...` files which help in verifying if the parameters are appropriately defined and populate MATLAB workspace with variables that maybe necessary but missing in the definition.

21.1 converters

Since the standard form of DDEs/DDFs and PDEs shown earlier in Chapter 5 is a large class, they involve many parameters which can be overwhelming to a new user. Furthermore, in most real-life systems, most of the parameters may in fact be empty or just zeroes. To avoid defining unnecessary parameters, `initialize_...` script files are used to get the minimal required information about the DDE/DDF/PDE from the user and then converted to PIE using `convert_...` script files.

21.1.1 `initialize_PIETOOLS_DDE(DDE)`

This function file takes input from the user, which describe the DDE system in the standard form Eq. (17.1) whose structure is as described in Section 17.1 and verifies if the parameters defined are consistent in dimension. The script also automatically defines any missing parameters. The inputs to this script file is the DDE object. The function returns another object which too is in the standard format mentioned in Section 17.1, however, the latter object is verified to have correct dimensions and parameters.

21.1.2 `initialize_PIETOOLS_DDF(DDF)`

This function file takes input from the user, which describe the DDF system in the standard form (see (17.3)) and verifies if the parameters defined are consistent in dimension. The script also automatically defines any missing parameters. The inputs to this script file is the DDF object. The function returns another object which too is in the standard format mentioned in Section 17.3, however, the latter object is verified to have correct dimensions and parameters.

21.1.3 `initialize_PIETOOLS_NDS(NDS)`

This function file takes input from the user, which describe the NDS system in the standard form (see (17.2)) and verifies if the parameters defined are consistent in dimension. The script also automatically defines any missing parameters. The inputs to this script file is the NDS object. The function returns another object which too is in the standard format mentioned in Section 17.2, however, the latter object is verified to have correct dimensions and parameters.

21.1.4 `initialize_PIETOOLS_PDE_batch(PDE)`

This function file takes input from the user, which describe the PDE system in the standard form Eq. (14.7)-(14.8) and verifies if the parameters defined are consistent in dimension. The script also automatically defines any missing parameters. The inputs to this script file is the PDE object. The function returns another object which too is in the standard format mentioned in Section 14.1, however, the latter object is verified to have correct dimensions and parameters.

21.1.5 `initialize_PIETOOLS_PDE_terms(PDE)`

This function file takes input from the user, which describe the PDE system in the standard form Eq. (14.7)-(14.8) and verifies if the parameters defined are consistent in dimension. The script also automatically defines any missing parameters. The inputs to this script file is the PDE object. The function returns another object which too is in the standard format mentioned in Section 14.2, however, the latter object is verified to have correct dimensions and parameters.

21.1.6 `convert_PIETOOLS_DDE(DDE)`

This function takes a DDE object in the format specified in Section 17.1 is converted to PIE structure with all the PI operators in 9.1 which is then sent as an output. Additionally, this script file calls `initialize_PIETOOLS_DDE` function in the beginning to verify the input format of DDE.

21.1.7 `convert_PIETOOLS_DDF(DDF)`

This function takes a DDF object in the format specified in Section 17.3 is converted to PIE structure with all the PI operators in 9.1 which is then sent as an output. Additionally, this script file calls `initialize_PIETOOLS_DDF` function in the beginning to verify the input format of DDF.

21.1.8 `convert_PIETOOLS_NDS2DDF(NDS)`

This function takes a NDS object in the format specified in Section 17.2 is converted to PIE structure with all the PI operators in 17.3 which is then sent as an output. Additionally, this script file calls `initialize_PIETOOLS_NDS` function in the beginning to verify the input format of NDS.

21.1.9 `convert_PIETOOLS_PDE_batch(PDE)`

This function takes a PDE object in the format specified in Section 14.1 is converted to PIE structure with all the PI operators in 9.1 which is then sent as an output. Additionally, this script file calls `initialize_PIETOOLS_PDE_batch` function in the beginning to verify the input format of PDE.

21.1.10 `convert_PIETOOLS_PDE_terms(PDE)`

This function takes a PDE object in the format specified in Section 14.2 is converted to PIE structure with all the PI operators in 9.1 which is then sent as an output. Additionally, this script file calls `initialize_PIETOOLS_PDE_terms` function in the beginning to verify the input format of PDE.

21.1.11 `convert_PIETOOLS_PDE_batch2terms(PDE)`

This function takes a PDE object in the format specified in Section 14.1 is converted to PDE structure in terms format 14.2 which is then sent as an output. Additionally, this script file calls `initialize_PIETOOLS_PDE_batch` function in the beginning to verify the input format of PDE.

21.2 executives

Executive files are preset function files that have standard LPIs hardwired in them. These files can be used to solve standard LPIs for any given PIE, provided the PIE system is passed as an argument along with settings related to the LPI optimization problem. For example, after defining a PIE structure `PIE` (refer 9.1), stability LPI can be solved by just calling `executive_PIETOOLS_stability(PIE, settings)` file. However, note that a executive files are deeply integrated with settings files. Currently, at least one `settings_...` file must be called before calling an executive.

21.2.1 executive_PIETOOLS_stability(PIE, settings)

This function file solves the stability LPI for a PIE. 4-PI operators \mathcal{T} and \mathcal{A} are mandatory inputs and must be defined before calling this script. Displays whether the LPI has a feasible solution or not and returns the Lyapunov operator P which proves stability. If a feasible solution exists, then the PIE is stable.

21.2.2 executive_PIETOOLS_stability_dual(PIE, settings)

This function file, same as the previous one, solves the dual LPI for stability of a PIE. 4-PI operators \mathcal{T} and \mathcal{A} are mandatory inputs and must be defined before calling this script. Displays whether the LPI has a feasible solution or not and returns the Lyapunov operator P which proves stability. If a feasible solution exists, then the PIE is stable.

21.2.3 executive_PIETOOLS_Hinf_gain(PIE, settings)

This function file finds an upper bound on the L_2 gain from disturbances w to outputs z in a PIE Eq. (5.1). 4-PI operators \mathcal{T} , \mathcal{A} , \mathcal{B}_1 , \mathcal{C}_1 and \mathcal{D}_{11} are mandatory inputs and must be defined before calling this script. Displays the smallest γ such that $\frac{\|z\|_{L_2}}{\|w\|_{L_2}} \leq \gamma$. The function also outputs **gamma** and returns the Lyapunov operator P which proves BIBO-stability.

21.2.4 executive_PIETOOLS_Hinf_gain_dual(PIE, settings)

This function file also finds an upper bound on the L_2 gain from disturbances w to outputs z in a PIE Eq. (5.1), however, it uses dual LPI instead of the primal. 4-PI operators \mathcal{T} , \mathcal{A} , \mathcal{B}_1 , \mathcal{C}_1 and \mathcal{D}_{11} are mandatory inputs and must be defined before calling this script. Displays the smallest γ such that $\frac{\|z\|_{L_2}}{\|w\|_{L_2}} \leq \gamma$. The function also outputs **gamma** and returns the Lyapunov operator P which proves BIBO-stability.

21.2.5 executive_PIETOOLS_Hinf_estimator(PIE, settings)

This function file finds an H_∞ -optimal estimator for the PIE Eq. (5.1) if one exists and returns the gains of the \mathcal{P} and \mathcal{Z} which can then be used to find the observer gains using the relation $\mathcal{L} = \mathcal{P}^{-1}\mathcal{Z}$. 4-PI operators \mathcal{T} , \mathcal{A} , \mathcal{B}_1 , \mathcal{C}_1 , \mathcal{C}_2 , \mathcal{D}_{11} and \mathcal{D}_{21} are mandatory inputs and must be defined before calling this script. The function also outputs observer gains **Lop**, H_∞ norm **gamma** and the Lyapunov operator P which proves observer stability.

21.2.6 executive_PIETOOLS_Hinf_control(PIE, settings)

This function file finds an H_∞ -optimal controller for the PIE Eq. (5.1) if one exists and returns the opvar objects P and Z which can then be used to find the controller gains using the relation $\mathcal{K} = \mathcal{Z}P^{-1}$. The function also outputs observer gains **Lop** and H_∞ norm **gamma** of the closed loop PIE.

21.3 settings

Settings files are used to set up parameters related to LPI optimization problems such as `d` and `opts` in `poslpivar` and `lpivar` functions. These settings also declare strictness of sign-definite constraints on `opvar` objects. Although, these are not universal for all examples of PIEs, they seem to work well in most cases. Feel free to use different settings from the one given in this scripts to tailor the code for specific examples. Current list of settings files and a brief overview of what they do is given below. Additionally, we provide a `settings_PIETOOLS_custom` file to allow users to freely modify and explore different settings without having out to worry about breaking the default values.

Settings	Description
<code>settings_PIETOOLS_extreme</code>	low degree polynomials in \mathcal{P} and \mathcal{Z} , highly sparse PI variables
<code>settings_PIETOOLS_stripped</code>	low degree polynomials in \mathcal{P} and \mathcal{Z} , moderately sparse PI variables
<code>settings_PIETOOLS_light</code>	low degree polynomials in \mathcal{P} and \mathcal{Z} , non sparse
<code>settings_PIETOOLS_heavy</code>	high degree polynomials in \mathcal{P} and \mathcal{Z} , non sparse
<code>settings_PIETOOLS_veryheavy</code>	very high degree polynomials in \mathcal{P} and \mathcal{Z} , non sparse

Chapter 22

Troubleshooting

This chapter is dedicated to tackling issues related to installation and setting up of PIETOOLS 2021a. Additionally, we also discuss some common issues users may run into while solving LPIs.

22.1 Troubleshooting: Installation

When installing the PIETOOLS toolbox using the install script, in rare circumstances, the user may run into one of following errors. In case of an error, a message is displayed explaining the issue, which can be one of the following.

1. *An error appeared when trying to create the folder ...*
Check if the folder already has a folder named **PIETOOLS_2021a**. If that does not resolve the issue try running MATLAB from an administrator account which has the authority to create or modify folders. As a last resort, try installing in a different folder. If that does not fix this, contact us.
2. *The installation directory “ ” already exists...*
Obvious error. However, if there is no folder with the name PIETOOLS_2021a, check for a hidden folder.
3. *‘tbxmanager’ or ‘SeDuMi’ were not downloaded or installed...*
Check your internet connection. Verify that MATLAB is allowed to download files using the internet connection. Check if the websites for tbxmanager and SeDuMi are operational.
4. *‘PIETOOLS’ was not downloaded or installed...*
Check the suggestions for the previous error. If that does not fix the issue, contact us.
5. *Could not modify the initialization file “startup.m”...*
Try running MATLAB from an administrator account which has the authority to create or modify folders. If that does not fix the issue, manually add SeDuMi or the relevant SDP solver (like MOSEK, sdpt3, sdpanl) to your MATLAB path, and extract the files in **PIETOOLS_2021a.zip**. Also add PIETOOLS to your MATLAB path.

6. *Could not save the path to a default location...*

Try running MATLAB from an administrator account which has the authority to create or modify folders. If that does not fix it, just add PIETOOLS and SeDuMi to your MATLAB path and skip this step.

22.2 Troubleshooting: Solving LPIs

PIETOOLS can be used for solving LPI optimization problems and users may run into errors while setting up and solving them. For any errors in setting up an LPI, refer to the function and script headers to ensure that input-output formats are correct. Ensure that PI objects, defined in the LPI problem, are well-defined and valid PI operators. You can use the `isvalid` function to check if a PI operator is well defined. If that does not fix the problem, feel free to contact us.

PIETOOLS 2021a relies on (a modified version of) SOSTOOLS 3.03 with solver SeDuMi to solve optimization problems. If you are unfamiliar with SOSTOOLS, and are unsure how to interpret the results of a solved optimization problem, please check the SOSTOOLS manual available at <http://www.cds.caltech.edu/sostools/>, or the SeDuMi manual available at https://sedumi.ie.lehigh.edu/?page_id=58 for more information. A brief overview of how to interpret results, and what to do in case of error, is provided below:

1. **How do I interpret the results of a solved optimization problem?**

A general rule of thumb is to look at: `pinf`, `dinf`, `feasratio` and `Residual norm`. `pinf` and `dinf` should be 0, while `feasratio` is in between -1 and 1 (preferably closer to 1). The lower the residual norm the better. Refer to SeDuMi manual to interpret other output parameters and more details.

2. **What if `pinf` is 1?**

Verify if the LPI constraints are in fact feasible. Verify if the sign-definiteness of the PI operator is on a compact interval (use `psatz` term if local sign-definite is needed) or the entire real line. Use `'getdeg'` function to check if your LPI constraint has high degree polynomials. If yes, make sure that all `opvar` variables used in `lpi_eq` function have high enough degrees to match it. If this does not resolve the issue contact us and attach the files that you are trying to run along with a snapshot of the error/output.

3. **What if `dinf` is 1 and `feasratio` is -1?**

This issue typically occurs when the objective function is unbounded from below and becomes $-\infty$. Check if the objective function is bounded below. If this does not resolve the issue contact us by email and attach the files that you are trying to run along with a snapshot of the error/output.

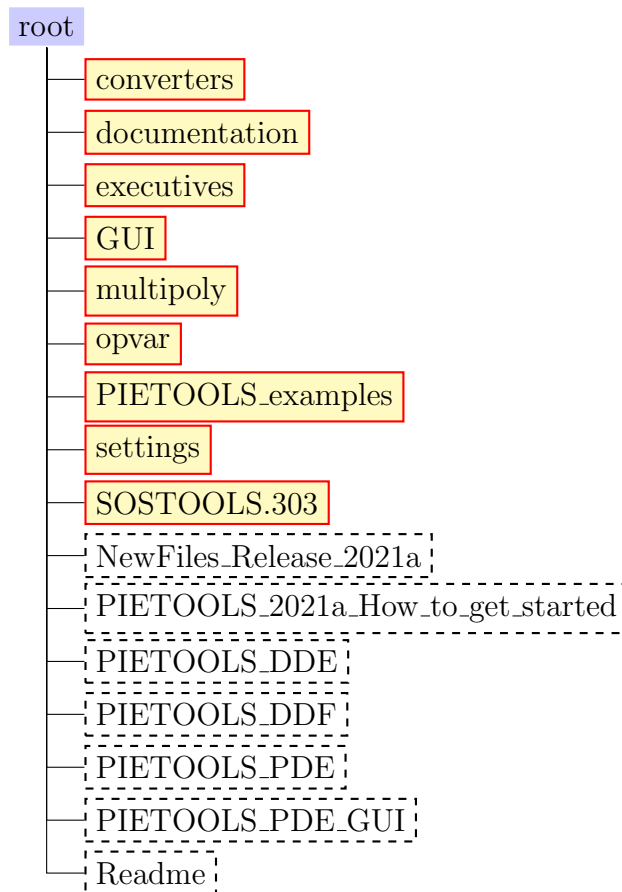
22.3 Contact Details

To resolve issues, report bugs or to collaborate on any development work regarding PIETOOLS, please contact us through email and we will get back to you as quickly as possible. In case of issues with installation, solving problems or bugs identified, please include the script file

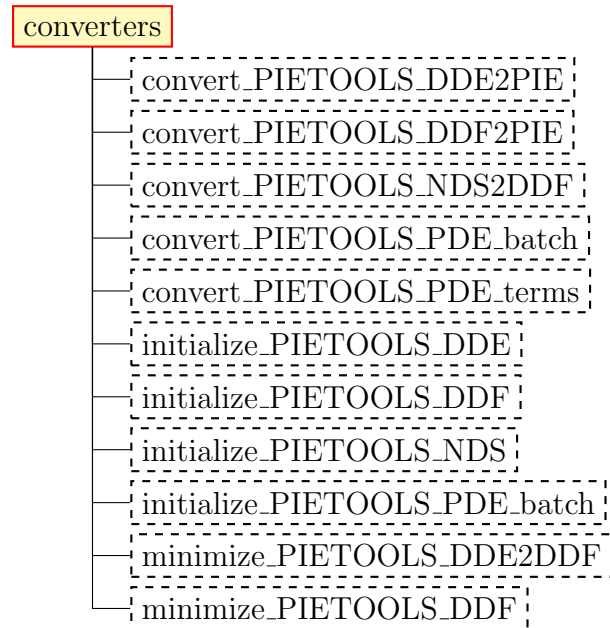
that generates the error along with images of the error generated in MATLAB. You can reach us through email at: sshivak8@asu.edu, ad2079@cam.ac.uk, djagt@asu.edu and mpeet@asu.edu

List Of Folders and Files

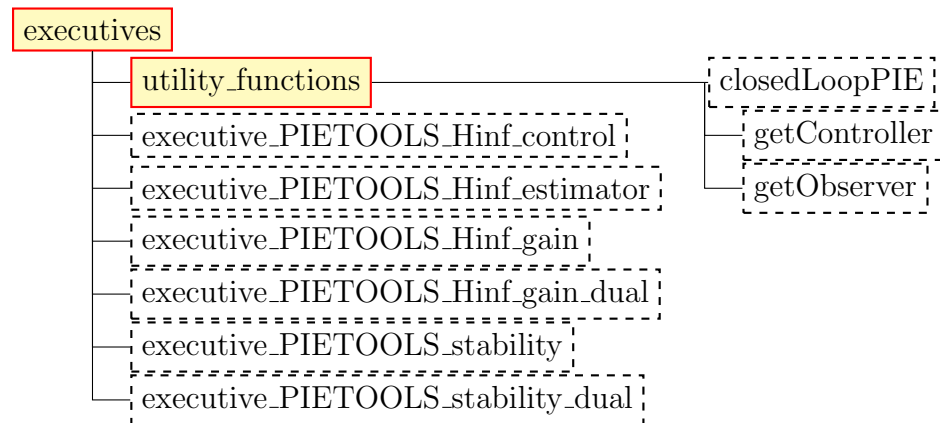
Directory Structure of PIETOOLS



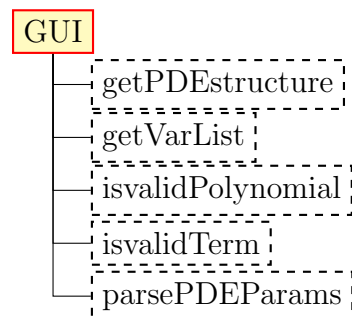
converters



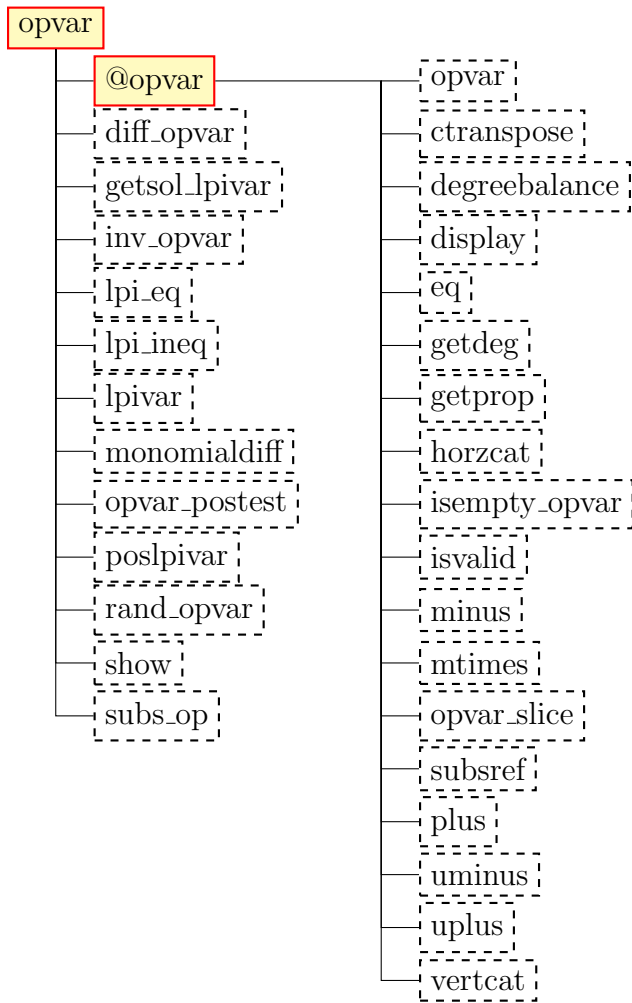
executives



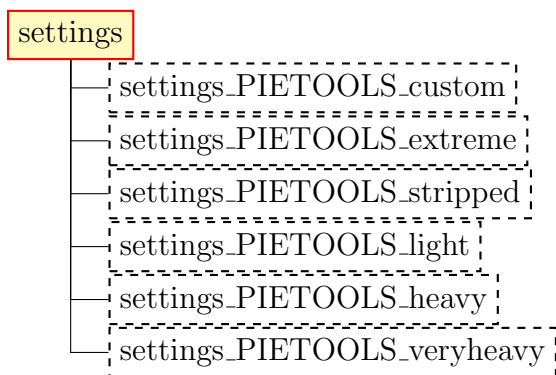
GUI



opvar



settings



Bibliography

- [1] A. Das, S. Shivakumar, S. Weiland, and M. Peet. H_∞ optimal estimation for linear coupled PDE systems. In *Proceedings of the IEEE Conference on Decision and Control*, 2019.
- [2] John Doyle, Andy Packard, and Kemin Zhou. Review of LFTs, LMIs, and μ . 1991.
- [3] M. Peet. A convex solution of the H_∞ -optimal controller synthesis problem for multi-delay systems. *SIAM Journal on Control and Optimization*, 2019. Submitted.
- [4] Matthew M Peet. Representation of networks and systems with delay: Ddes, ddfs, ode-pdes and pies. *Automatica*, 127:109508, 2021.
- [5] Stephen Prajna, Antonis Papachristodoulou, and Pablo A Parrilo. Introducing SOS-TOOLS: A general purpose sum of squares programming solver. In *Proceedings of the 41st IEEE Conference on Decision and Control, 2002.*, volume 1, pages 741–746. IEEE, 2002.
- [6] Walter Rudin. Functional analysis, 1973.
- [7] Peter Seiler. SOSOPT: A toolbox for polynomial optimization. *arXiv preprint arXiv:1308.1889*, 2013.
- [8] S. Shivakumar, A. Das, S. Weiland, and M. Peet. A generalized lmi formulation for input-output analysis of linear systems of odes coupled with pdes. In *Proceedings of the IEEE Conference on Decision and Control*, 2019.