



PIETOOLS 2022: User Manual

S. Shivakumar

A. Das

D. Braghini

D. Jagt

Y. Peet

M. Peet

December 9, 2022

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
1.1	PIETOOLS 2022 Release Notes	7
1.2	Installing PIETOOLS	8
1.2.1	Installation	8
2	Scope of PIETOOLS	10
2.1	Motivation	10
2.2	PIETOOLS for Analysis and Control of ODE-PDE Systems	11
2.2.1	Defining Models in PIETOOLS	11
2.2.2	Setting the control signal	12
2.2.3	Declaring Boundary Conditions	12
2.2.4	Simulating ODE-PDE Model	12
2.2.5	Analysis and Control of the ODE-PDE Model Using PIEs	13
2.3	Summary	16
3	PI Operators in PIETOOLS	18
3.1	Declaring PI Operators in 1D	18
3.1.1	Declaring 3-PI Operators	18
3.1.2	Declaring 4-PI Operators	20
3.2	Declaring PI Operators in 2D	21
3.2.1	Declaring 9-PI Operators	22
3.2.2	Declaring General 2D PI Operators	24
3.3	Overview of <code>opvar</code> and <code>opvar2d</code> Structure	27
3.3.1	<code>opvar</code> class	27
3.3.2	<code>opvar2d</code> class	27
I	PIETOOLS Workflow for ODE-PDE and DDE Models	29
4	Setup and Representation of PDEs and DDEs	30
4.1	Command Line Parser for 1D ODE-PDEs	30
4.1.1	Defining a coupled ODE-PDE system	31
4.1.2	More examples of command line parser format	34
4.2	Alternative Input Formats for PDEs	38
4.2.1	A GUI for Declaring PDEs	38
4.2.2	An Input Format for 2D PDEs	40

4.3	Batch Input Format for DDEs	41
4.3.1	Initializing a DDE Data structure	42
4.4	Alternative Input Formats for TDSs	42
5	Converting PDEs and DDEs to PIEs	45
5.1	What is a PIE?	45
5.2	Converting a PDE to a PIE	46
5.3	Converting a DDE to a PIE	47
5.4	Converting a System with Inputs and Outputs to a PIE	50
6	Simulating PDE, DDE and PIE Solutions with PIESIM	54
6.1	Simulating Solutions in the PIE Representation	54
6.2	PIE Simulation Using PIETOOLS	55
6.2.1	Following the solver_PIESIM Template Script	55
6.2.2	Using the PIESIM Function	56
6.3	Plotting the solution	57
6.3.1	PIESIM Demonstration A: PDE example	57
6.3.2	PIESIM Demonstration B: DDE example	59
6.3.3	PIESIM Demonstration C: PIE example	60
7	Declaring and Solving Convex Optimization Programs on PI Operators	65
7.1	Initializing an Optimization Problem Structure	66
7.2	Declaring Decision Variables	68
7.2.1	sosdecvar	68
7.2.2	poslpivar	68
7.2.3	lpivar	71
7.3	Imposing Constraints	73
7.3.1	lpi_ineq	73
7.3.2	lpi_eq	74
7.4	Defining an Objective Functions	75
7.5	Solving the Optimization Problem	76
7.6	Extracting the Solution	77
7.6.1	sosgetol	77
7.6.2	getsol_lpivar	78
7.7	Running Pre-Defined LPis: Executives and Settings	80
7.7.1	Settings in PIETOOLS Executives	81
7.7.2	Executive Functions Available in PIETOOLS	83
II	Additional PIETOOLS Functionality	85
8	Alternative Input Formats for ODE-PDE Systems	86
8.1	A GUI for Defining PDEs	86
8.1.1	Step 1: Define States, Outputs and Inputs	87
8.1.2	Step 2: Select an Equation to Add a Term	89
8.1.3	Step 3: (Optional) Add or Remove BC	93

8.1.4	Step 4-5: Parse PDE Parameters and Convert Them to PIE	94
8.2	The Terms-Based Input Format	94
8.2.1	Declaring a System of Coupled 2D PDEs	96
8.2.2	Declaring an ODE-PDE System	99
8.2.3	Declaring a System with (Delayed) Inputs and Outputs	102
8.2.4	Additional Options	106
8.3	The Command Line Parser format Format	110
8.3.1	<code>state</code> class objects	110
8.3.2	<code>sys</code> class objects	113
9	Batch Input Formats for Time-Delay Systems	116
9.1	Representing Systems with Delay	116
9.1.1	Input of Delay Differential Equations	116
9.1.2	Input of Neutral Type Systems	117
9.1.3	The Differential Difference Equation (DDF) Format	118
9.2	Converting between DDEs, NDSs, DDFs, and PIEs	119
9.2.1	DDF to PIE	120
9.2.2	DDE to DDF or PIE	120
9.2.3	NDS to DDF or PIE	121
10	Operations on PI Operators in PIETOOLS: <code>opvar</code> and <code>dopvar</code>	123
10.1	Declaring <code>opvar</code> and <code>dopvar</code> Objects	123
10.1.1	The <code>opvar</code> Class	123
10.1.2	<code>dopvar</code> objects and the <code>dopvar</code> class	126
10.2	Algebraic Operations on <code>opvar</code> Objects	127
10.2.1	Addition ($A+B$)	127
10.2.2	Multiplication ($A*B$)	128
10.2.3	Adjoint (A')	129
10.2.4	Inverse (<code>inv_opvar(A)</code>)	129
10.3	Matrix Operations on <code>opvar</code> Objects	130
10.3.1	Concatenation ($[A,B]$)	131
10.3.2	Subs-indexing ($A(i,j)$)	132
10.4	Additional Methods for <code>opvar</code> Objects	132
III	Examples and Applications	134
11	PIETOOLS Demonstrations	135
11.1	DEMO 1: Simple Stability, Simulation and Control Problem	135
11.2	DEMO 2: Estimating the Volterra Operator Norm	135
11.3	DEMO 3: Solving the Poincaré Inequality	136
11.4	DEMO 4: Finding an Optimal Stability Parameter	138
11.5	DEMO 5: Constructing and Simulating an Optimal Estimator	140
11.6	DEMO 6: H_∞ -optimal Controller synthesis for PDEs	145
11.7	DEMO 7: Observer-based Controller design and simulation for PDEs	149

12 Libraries of PDE and TDS Examples in PIETOOLS	155
12.1 A Library of PDE Example Problems	155
12.2 Libraries of DDE, NDS, and DDF Examples	157
12.2.1 DDE Examples	157
12.2.2 NDS and DDF Examples	157
13 Standard Applications of LPI Programming	159
13.1 LPIs for Analysis of PIEs	159
13.1.1 Operator Norm	159
13.1.2 Stability	160
13.1.3 Dual Stability	160
13.1.4 Input-Output Gain	161
13.1.5 Dual Input-Output Gain	161
13.1.6 Positive Real Lemma	161
13.2 LPIs for Optimal Estimation of PIEs	162
13.3 LPIs for Optimal Control of PIEs	163
IV Appendices	164
A PI Operators and their Properties	165
A.1 PI Operators on Different Function Spaces	165
A.2 Addition of PI Operators	168
A.3 Composition of PI Operators	168
A.4 Adjoint of PI Operators	170
A.5 Inversion of PI operators	171
A.5.1 Inversion of 3-PI operators	171
A.5.2 Inversion of 4-PI operators	173
A.6 Composition of Differential and PI operator	174
A.7 Matrix Parametrization of Positive Definite PI Operators	174
B Troubleshooting	176
B.1 Troubleshooting: Installation	176
B.2 Troubleshooting: Solving LPIs	177
B.3 Contact Details	177

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 PI operators in 1D and 2D
- declare PI operator decision variables (positive 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 [10], 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.)

1.1 PIETOOLS 2022 Release Notes

PIETOOLS 2022 introduces several improvements to PIETOOLS, including a new interface for declaring 1D PDEs, and updating many functions to allow for representation and analysis of 2D PDEs. We list the main updates below:

1. **Introduction of the command line user interface:** In PIETOOLS 2022, 1D linear PDEs can be constructed on-the-fly using simple command line instructions. In particular, PIETOOLS 2022 adds the following functionality:

- A new function `state` that allows state variables, inputs and outputs to be easily declared.
- Built-in routines for performing addition, multiplication, differentiation, substitution, and integration of declared state variables and inputs, allowing for straightforward symbolic declaration of differential equations and boundary conditions.
- A new overarching structure `sys` for representing linear 1D ODE-PDE systems, to which declared equations can be easily added using `addequation`.
- A built-in routines `convert` to directly convert declared ODE-PDE systems to equivalent PIEs.

Using the new command line parser, a wide variety of linear 1D ODE-PDE systems, including systems with delay, can be easily declared from the Command Window, and converted to equivalent PIEs for further analysis. See Chapter 4 for more information.

2. **Introduction of 2D PDE and PIE functionality:** Expanding upon the functionality for 1D ODE-PDEs, PIETOOLS 2022 now also allows coupled systems of linear ODEs, 1D PDEs and 2D PDEs to be declared and analysed. In particular, PIETOOLS 2022 adds the following functionality:

- A new terms-format that allows for representation of coupled linear ODE - 1D PDE - 2D PDE systems. See Section 8.2.
- `opvar2d` and `dopvar2d` functions for representing PI operators and PI operator decision variables in 2D. See Section 3.2
- LPI programming functions for setting up and solving LPI programs involving 2D PI operators. See Chapter 7.

1.2 Installing PIETOOLS

PIETOOLS 2022 is compatible with Windows, Mac or Linux systems and has been verified to work with MATLAB version 2020a or higher, however, we suggest to use the latest version of MATLAB.

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.

1.2.1 Installation

PIETOOLS 2022 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), `SOSTOOLS 4.00` (always installed), `PIETOOLS 2022` (always installed). Adds all the files to MATLAB path.

- Go to <https://github.com/CyberneticSCL/PIETOOLS> or 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 2022 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_2022.zip** from [this link](#), unzip, and add to MATLAB path.

Chapter 2

Scope of PIETOOLS

In this chapter, we briefly talk about the need for a new computational tool for the analysis and control of ODE-PDE systems as well as DDEs. We lightly touch upon, without going into details, the class of problems PIETOOLS can solve.

2.1 Motivation

Semidefinite programming (SDP) is a class of optimization problems that involve the optimization of a linear objective over the cone of positive semidefinite (PSD) matrices. The development of efficient interior-point methods for semidefinite 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 applications of LMI techniques were restricted to finite dimensional systems, until the sum-of-squares method came into the lime-light. 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 SOSTOOLS [6], 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 the analysis and control of PDEs without ad-hoc interventions. For example, to search for a Lyapunov function that proves the 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 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 PIETOOLS for Analysis and Control of ODE-PDE Systems

Using PIETOOLS 2022 for controlling ODE-PDE models has been made intuitive, easy, and with very few mathematical details about the PIE operators and the maths behind them. Let's take a closer look at how this works.

2.2.1 Defining Models in PIETOOLS

Any control problem necessarily starts with declaring the model, and PIETOOLS makes it extremely simple to do so. Suppose that we are interested in modeling a coupled ODE-PDE system, such as a system with ODE dynamics given by

$$\dot{x}(t) = -x(t) + u(t), \quad (2.1)$$

with controlled input u , and PDE dynamics given by

$$\ddot{\mathbf{x}}(t, s) = c^2 \partial_s^2 \mathbf{x}(t, s) - b \partial_s \mathbf{x}(t, s) + s w(t), s \in (0, 1), t \geq 0, \quad (2.2)$$

i.e, the one-dimensional wave equation with velocity c , added viscous damping coefficient b and external disturbance w . Since the PDE has a second-order derivative in time, we should make a change of variables to appropriately define a state space. On this example, we do so by calling $\phi = (\partial_s \mathbf{x}, \dot{\mathbf{x}})$. Thus the dynamic equation becomes

$$\dot{\phi}(t, s) = \begin{bmatrix} 0 & 1 \\ c & 0 \end{bmatrix} \partial_s \phi(t, s) + \begin{bmatrix} 0 & 0 \\ 0 & -b \end{bmatrix} \phi(t, s) + \begin{bmatrix} 0 \\ s \end{bmatrix} w(t), s \in (0, 1), t \geq 0. \quad (2.3)$$

We can also add to our system a regulated output equation, for instance

$$z(t) = \begin{bmatrix} r(t) \\ u(t) \end{bmatrix}, \quad (2.4)$$

where $r(t) = \int_0^1 [1 \ 0] \phi(t, s) ds$ gives $\mathbf{x}(t, s = 1) - \mathbf{x}(t, s = 0)$. Adding u to the regulated output is a way of obtaining this signal from the simulations.

To define these models, we first create the following variables in MATLAB using the `state()` class:

```
>> x = state('ode');    phi = state('pde',2);
>> w = state('in');    u = state('in');
>> z = state('out',2)
```

Then, we can define/add equations to this `sys()` object using standard operators, such as '+', '-', '*', 'diff', 'subs', 'int', etc., as shown below.

```
>> odepde = sys();
>> eq_dyn = [diff(x,t,1) == -x+u diff(phi,t,1)==[0 1; c 0]*diff(phi,s,1)+[0;s]*w+[0
0;0 -b]*phi];
>>eq_out= z ==[int([1 0]*phi,s,[0,1]) u];
>>odepde = addequation(odepde,[eq_dyn;eq_out]);
```

```

    Initialized sys() object of type "pde"

    5 equations were added to sys() object

```

Whenever, equations are successfully added to the `sys()` object, a text message confirming the same is displayed in the command output window. To verify if PIETOOLS got the right equations, the user just needs to type the system variable ("odepde" in this example) on the command window for PIETOOLS to display the added equations. We encourage the user to always check the equations before proceeding.

2.2.2 Setting the control signal

Since our system has a controlled input, which could have any name, we must pass this information to PIETOOLS. This is done by the following command:

```

>> odepde= setControl(odepde,[u]);

    1 inputs were designated as controlled inputs

```

Similarly, observed outputs also need to be specified. For details, see chapter 4.

2.2.3 Declaring Boundary Conditions

A general PDE model is incomplete without boundary conditions, but in PIETOOLS, boundary conditions can be declared in much the same way as the system dynamics. For example, to declare the following Dirichlet,

$$\dot{\mathbf{x}}(t, s = 0) = 0,$$

and Neuman,

$$\partial_s \mathbf{x}(t, s = 1) = x(t),$$

boundary conditions, we can simply call

```

>> bc1 = [0 1]*subs(phi,s,0) == 0;
>> bc2 = [1 0]*subs(phi,s,1) == x;
>> odepde = addequation(odepde,[bc1;bc2]);

    2 equations were added to sys() object

```

2.2.4 Simulating ODE-PDE Model

One of the first things a practitioner might do is to simulate the system. If you look at traditional PDE literature, one of the biggest challenges is simulation. Every different kind of PDE requires different techniques to discretize. More importantly, there is no guarantee that the simulation result will stay bounded. In PIETOOLS, there is only one command to simulate any linear ODE-PDE coupled system of your choice:

```

>> solution = PIESIM(odepde, opts, uinput, ndiff);

```

For instance, suppose, we want to simulate the ODE-PDE model corresponding to (2.1) and (2.3) with constant velocity $c = 1$, damping coefficient $b = 0.1$, under the previously defined

boundary conditions, and a specific choice of disturbance $w(t)$. In particular, we aim to investigate how $w(t)$ affects $\dot{\mathbf{x}}$ and the value $\mathbf{x}(t, s = 1) = \mathbf{x}(t, s = 0)$ given by the previously defined output.

To run this simulation in PIESIM, we use the following opts to the function, which commands PIESIM to don't automatically plot the solution, to use 8 Chebyshev polynomials, to simulate the solutions up to $t = 1s$ with a time-step of $10^{-2}s$, and to use Backward Differentiation Formula (BDF):

```
>>opts.plot = 'no';  
>>opts.N = 8;  
>>opts.tf = 10;  
>>opts.dt = 1e-2;  
>>opts.intScheme=1;
```

Another important piece of information to PIESIM is regarding the initial conditions and external perturbations for the simulation. This is done as follows for the zero-state response of the system perturbed by an exponentially decaying sinusoidal signal:

```
>>uinput.ic.PDE = [0,0];  
>>uinput.ic.ODE = 0;  
>>uinput.u=0;  
>>uinput.w = sin(5*st)*exp(-st);
```

Note that the control input must also be zero since we want to simulate an open-loop response. The last argument is regarding the space differentiability of the states. In this example, the PDE state involves 2 first order differentiable state variables and this is passed through the input `ndiff` as:

```
>>ndiff = [0,2,0];
```

For details on PIESIM arguments, we refer the reader to chapter 6. PIESIM gives us discretized time-dependent arrays corresponding to the time vector used in the simulations and the resulting state variables and output. The result is depicted on Figures 2.2 and 2.1.

2.2.5 Analysis and Control of the ODE-PDE Model Using PIEs

Apart from simulation, you may be interested in knowing whether the model is internally stable or not. Moreover, what would be a good control input such that the effect of external disturbances for a specific choice of output can be suppressed? In PIETOOLS, such an analysis and synthesis are typically performed by first converting the ODE-PDE model to a new representation called Partial Integral Equations (PIEs), which is parametrized by a special class of operators, and then solving convex optimization problems (see chapter. 7) for more details.

Thus, PIEs are an equivalent representation of ODE-PDE models which provide a convenient and efficient way to analyze ODE-PDE models by numerically treatable methods. The conversion from the original system to the PIE representation is simply done using the following command, resulting in the next output on the command window (considering that every previous step was correctly done):

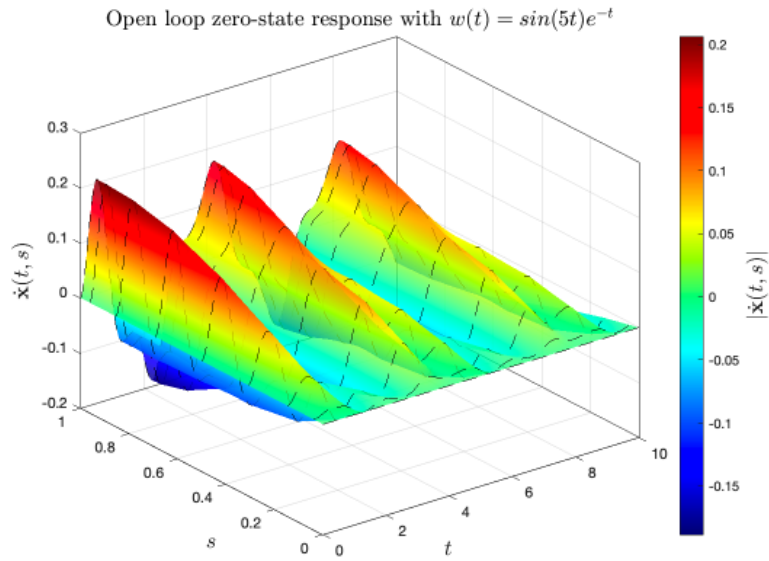


Figure 2.1: Transient response of the state variable $\dot{\mathbf{x}}(t, s)$ by simulating the ODE-PDE model (2.1) and (2.3) with $u(t) = 0$ for external disturbance $w(t) = \sin(5t)e^{-t}$.

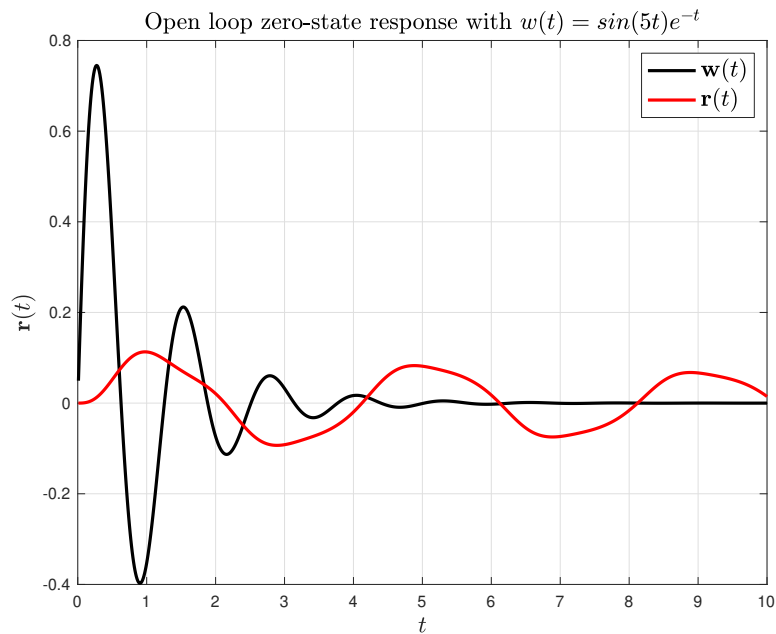


Figure 2.2: Transient response $r(t)$ of the ODE-PDE model (2.1) and (2.3) with $u(t) = 0$ for external disturbance $w(t) = \sin(5t)e^{-t}$.

```

>>PIE = convert(odepde,'pie');

    --- Reordering the state components to allow for representation as PIE ---

The order of the state components x has not changed.

    --- Converting ODE-PDE to PIE ---
Initialized sys() object of type 'pde'
Conversion to pie was successful

```

Once the model is converted to a PIE, analysis, and control can be performed by calling one of the executive functions. There are plenty of executive functions available, starting from stability, computing H_∞ gain, H_∞ optimal state estimator as well as state feedback controllers.

For instance, the example of this chapter is asymptotically stable only when $b > 0$. This can be shown by calling the executive after one of the following predefined settings had been chosen: `extreme`, `stripped`, `light`, `heavy`, `veryheavy`, or `custom`. For details on the optimization settings, the reader is referred to section. 7.7.

```

>>settings = lpiettings('heavy');
>>[prog, P] = PIETOOLS_stability(PIE,settings);

```

If the resultant optimization problem can be solved with these settings, the following message will be displayed after the optimization outputs:

```

    The System of equations was successfully solved.

```

, which provides an exponential stability certificate for the system.

Now, if we want to improve the system's rejection of disturbances, the optimal solution is to design a state-feedback controller that provides a control input $u(t)$ to be applied in (2.1), which minimizes the H_∞ norm of the closed-loop system, provided that such a controller exists. To compute this performance measurement on the open-loop, we just need to call the executive:

```

>> [prog, P, gamma] = PIETOOLS_Hinf_gain(PIE,settings);

```

Provided, again, that the optimization problem can be solved, the command window output will display the H_∞ norm. In this example, we have:

```

    The H-infty norm of the given system is upper bounded by:
    5.1631

```

For PIETOOLS to synthesize a state feedback controller which minimizes this metric, we need to call a third executive:

```

>> [prog, Kval, gam_val] = PIETOOLS_Hinf_control(PIE, settings);

```

, which will make PIETOOLS search for the operator \mathcal{K} stored in variable `Kval` corresponding to the controller, and display the closed loop H_∞ norm if succeeded. For this example, the result is a great increase in performance, in terms of this metric:

The closed-loop H_∞ norm of the given system is upper bounded by:
0.9779

The controller is generally a 4-PI linear operator, as detailed described in Chapter. 3, which has an image parameterized by matrix-valued polynomials. The resultant controller can be displayed by entering its variable name on the command window. Keep in mind that PIETOOLS disregard the monomials with coefficients lower than an accuracy defaulted to 10^{-4} .

We can again use PIESIM to simulate the response of the resultant closed-loop system, as depicted in Figures 2.3 and 2.2.5. Moreover, we can compare the closed-loop with the open-loop performances, as shown by Figure 2.2.5. We encourage the user to look at the file DEMO1_Simple_Stability_Simulation_and_Control.m, included in the PIETOOLS_demos folder of PIETOOLS, which provides a complete guide to reproduce the results described and depicted on this section.

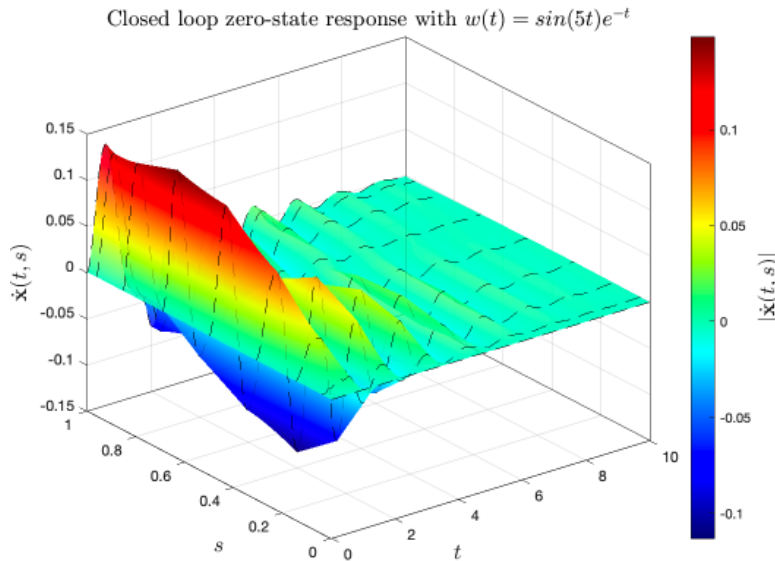


Figure 2.3: Transient response of the state variable $\dot{\mathbf{x}}(t, s)$ on the closed-loop system for external disturbance $w(t) = \sin(5t)e^{-t}$.

2.3 Summary

In this chapter, we gave an introduction to how PIETOOLS can be used to solve various control-relevant problems involving linear ODE-PDE models. The example depicted here was highly sensitive to disturbances infinite-dimensional system. Figures. 2.1 and. 2.2 show that, even after the applied disturbance has ceased, the output signal $r(t)$ remains affected, taking more time than the final time of the presented simulation to reject the disturbance. This behavior is measured by the computed H_∞ norm of the open-loop system.

On the other hand, with the synthesized feedback controller given by PIETOOLS, the closed-loop system quickly rejects the disturbance, as is clear from Figures. 2.3 and. 2.2.5. The increase in performance can be certified by the considerable reduction in the value of the

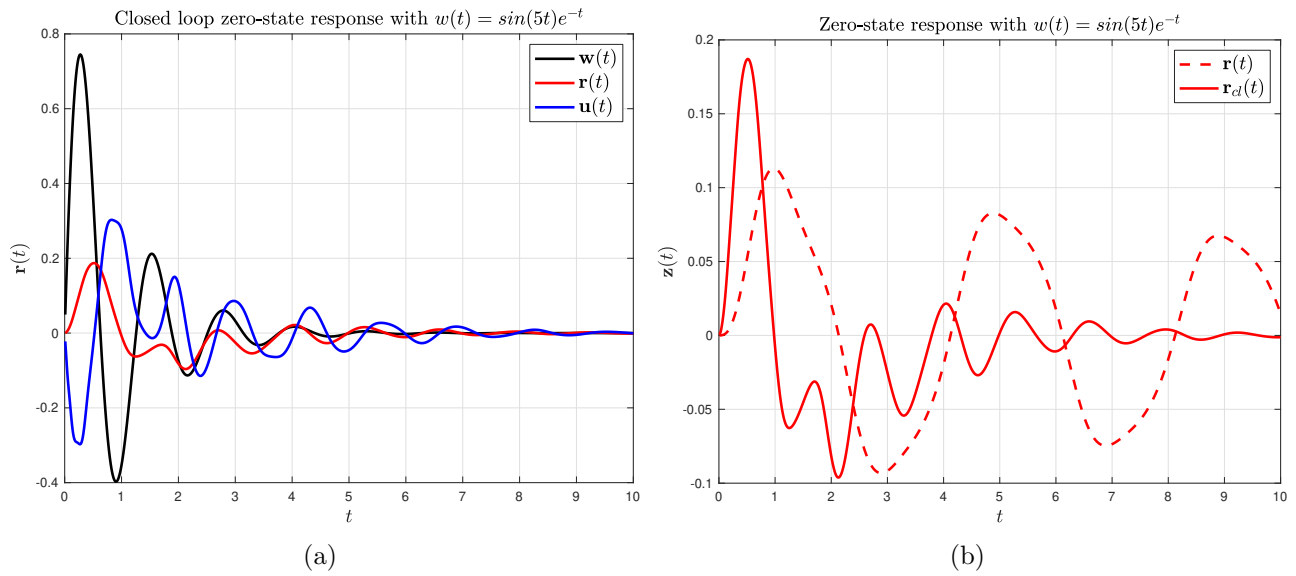


Figure 2.4: (a) Transient response of the output $r(t)$ and controlled input $u(t)$ of the closed-loop system for external disturbance $w(t) = \sin(5t)e^{-t}$. (b) Comparison of the transient responses of the open-loop system- $r(t)$ - with the closed-loop system- $r_{cl}(t)$.

H_∞ norm and by comparing the behavior of the outputs without and with the controller, in Figure. 2.2.5.

Chapter 3

PI Operators in PIETOOLS

PIETOOLS primarily functions by manipulation of Partial Integral (PI) operators which is made simple by introduction of MATLAB classes that represent PI operators. In PIETOOLS 2022, there are two types of PI operators: PI operators with known parameters, `opvar/opvar2d` class objects, and PI operators with unknown parameters, `dopvar/dopvar2d` class objects. In this Chapter, we outline the classes used to represent PI operators with known parameters. The information in this chapter is divided as follows: Section 3.1 and Section 3.2 provide brief mathematical background, and corresponding MATLAB implementation, about PI operators in 1D and 2D, respectively. Section 3.3 provides an overview of the structure of `opvar/opvar2d` classes in PIETOOLS. For more theoretical background on PI operators, we refer to Appendix A. For more information on operations that can be performed on `opvar/opvar2d` class objects, we refer to Chapter 10.

3.1 Declaring PI Operators in 1D

In this Section, we illustrate how 1D PI operators can be represented in PIETOOLS using `opvar` class objects. Here, we say that an operator \mathcal{P} is a 1D PI operator if it acts on functions $\mathbf{v}(s)$ depending on just one spatial variable s , and the operation it performs can be described using partial integrals. We further distinguish 3-PI operators, acting on functions $\mathbf{v} \in L_2^n[a, b]$, and 4-PI operators, acting on functions $\begin{bmatrix} v_0 \\ \mathbf{v}_1 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a, b] \end{bmatrix}$. Both types of operators can be represented using `opvar` class objects, as we show in the remainder of this section.

3.1.1 Declaring 3-PI Operators

We first consider declaring a 3-PI operator in PIETOOLS. Here, for given parameters $R = \{R_0, R_1, R_2\}$, the associated 3-PI operator $\mathcal{P}[R] : L_2^n[a, b] \rightarrow L_2^n[a, b]$ is given by

$$(\mathcal{P}[R]\mathbf{v})(s) = R_0(s)\mathbf{v}(s) + \int_a^s R_1(s, \theta)\mathbf{v}(\theta)d\theta + \int_s^b R_2(s, \theta)\mathbf{v}(\theta)d\theta, \quad s \in [a, b], \quad (3.1)$$

for any $\mathbf{v} \in L_2^n[a, b]$. In PIETOOLS, we represent such 3-PI operators using `opvar` class objects. For example, suppose we wish to declare a very simply PI operator $\mathcal{A} : L_2^2[-1, 1] \rightarrow L_2^2[-1, 1]$,

defined by

$$(\mathcal{A}\mathbf{v})(s) = \int_{-1}^s \underbrace{\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}}_{R_1} \mathbf{v}(\theta) d\theta, \quad s \in [-1, 1]. \quad (3.2)$$

To declare this operator, we first initialize an empty `opvar` object `A`, by simply calling `opvar` as:

```
>> opvar A
A =
  [] | []
-----
  [] | []
A.R =
  [] | [] | []
>> A.I = [-1,1];
```

Here, the first line initialize a 0×0 `opvar` object with all empty parameters `[]`. The second line, `A.I = [-1, 1]`, then sets the spatial interval associated to the operator equal to $[-1, 1]$, indicating that it maps the function space $L_2[-1, 1]$.

Next, we set the parameters of the operator. For a 3-PI operator such as \mathcal{A} , only the parameters in the field `A.R` will be nonzero, where `A.R` itself has fields `R0`, `R1` and `R2`. For our simple operator, only the parameter R_1 in the 3-PI Expression (3.1) is nonzero, so we only have to assign a value to the field `R1`:

```
>> A.R.R1 = [1,2; 3,4];
A =
  [] | []
-----
  [] | []
A.R =
  [0,0] | [1,2] | [0,0]
  [0,0] | [3,4] | [0,0]
```

where the fields `A.R.R0` and `A.R.R2` automatically default to zero-arrays of the appropriate dimensions. With that, the `opvar` object `A` represents the PI operator \mathcal{A} as defined in (3.2).

Next, suppose we wish to implement a slightly more complicated operator $\mathcal{B} : L_2[0, 1] \rightarrow L_2^2[0, 1]$, defined as

$$(\mathcal{B}\mathbf{x})(s) = \underbrace{\begin{bmatrix} 1 \\ s^2 \end{bmatrix}}_{R_0} \mathbf{v}(s) + \int_0^s \underbrace{\begin{bmatrix} 2s \\ s(s-\theta) \end{bmatrix}}_{R_1} \mathbf{x}(\theta) d\theta + \int_s^1 \underbrace{\begin{bmatrix} 3\theta \\ \frac{3}{4}(s^2-s) \end{bmatrix}}_{R_2} \mathbf{x}(\theta) d\theta, \quad s \in [0, 1].$$

For this operator, the parameters $R_i(s, \theta)$ are all polynomial functions. Such polynomial functions can be represented in PIETOOLS using the `polynomial` class (from the ‘multipoly’ toolbox), for which operations such as addition, multiplication and concatenation have already been implemented. This means that polynomials such as the functions R_i can be implemented by simply initializing polynomial variables s and θ , and then using these variables to define the desired functions:

```

>> pvar s theta
>> R0 = [1; s^2]
R0 =
 [ 1]
 [ s^2]

>> R1 = [2*s; s*(s-theta)]
R1 =
 [ 2*s]
 [ s^2 - s*theta]

>> R2 = [3*theta; (3/4)*(s^2-s)]
R2 =
 [ 3*theta]
 [ 0.75*s^2 - 0.75*s]

```

Here, the first line calls the function `pvar` to initialize the two polynomial variables `s` and `theta`, which we use to represent the spatial variable s and dummy variable θ respectively. Then, we can add and multiply these variables to represent any desired polynomial in (s, θ) , allowing us to implement the parameters $R_0(s)$, $R_1(s, \theta)$ and $R_2(s, \theta)$. Having defined these parameters, we can then represent the operator \mathcal{B} as an `opvar` object `b` as before:

```

>> opvar B;
>> B.I = [0,1];
>> B.var1 = s;      B.var2 = theta;
>> B.R.R0 = R0;    B.R.R1 = R1;    B.R.R2 = R2
B =
  [] | []
  -----
  [] | B.R

B.R =
  [1] | [2*s] | [3*theta]
 [s^2] | [s^2-s*theta] | [0.75*s^2-0.75*s]

```

Note here that, in addition to specifying the spatial domain $[0, 1]$ of the variables using the field `B.I`, we also have to specify the actual variables s and θ that appear in the parameters, using the fields `B.var1` and `B.var2`. Here `var1` should correspond to the primary spatial variable, i.e. the variable s on which the function $\mathbf{u}(s) := (\mathcal{B}\mathbf{v})(s)$ will actually depend, and `var2` should correspond to the dummy variable, i.e. the variable θ which is used solely for integration.

3.1.2 Declaring 4-PI Operators

In addition to 3-PI operators, 4-PI operators can also be represented using the `opvar` structure. Here, for a given matrix P , given functions Q_1, Q_2 , and 3-PI parameters $R = \{R_0, R_1, R_2\}$, we define the associated 4-PI operator $\mathcal{P} \begin{bmatrix} P & Q_1 \\ Q_2 & R \end{bmatrix} : \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a,b] \end{bmatrix}$

$$\left(\mathcal{P} \begin{bmatrix} P & Q_1 \\ Q_2 & R \end{bmatrix} \mathbf{v} \right)(s) = \begin{bmatrix} P v_0 + \int_a^b Q_1(s) \mathbf{v}_1(s) ds \\ Q_2(s) v_0 + (\mathcal{P}[R] \mathbf{v}_1)(s) \end{bmatrix}, \quad s \in [a, b],$$

for $\mathbf{v} = \begin{bmatrix} v_0 \\ \mathbf{v}_1 \end{bmatrix} \in \left[L_2^{\mathbb{R}^{n_0}} \right]_{L_2^1[a,b]}$. To represent operators of this form, we use the same `opvar` structure as before, only now also specifying values of the fields `P`, `Q1` and `Q2`. For example, suppose we wish to declare a 4-PI operator $\mathcal{C} : \left[L_2^{\mathbb{R}^2} \right]_{L_2[0,3]} \rightarrow \left[L_2^{\mathbb{R}} \right]_{L_2^2[0,3]}$ defined as

$$(\mathcal{C}\mathbf{x})(s) = \left[\underbrace{\begin{bmatrix} -1 & 2 \\ 0 & -s \\ s & 0 \end{bmatrix}}_{Q_2} \begin{matrix} x_0 \\ v_0 \end{matrix} + \underbrace{\int_0^3 (3-s^2)}_{Q_1} \mathbf{x}_1(s) ds + \underbrace{\int_0^s \begin{bmatrix} 1 \\ -s^3 \end{bmatrix}}_{R_0} \mathbf{v}_1(s) + \underbrace{\int_0^s \begin{bmatrix} s-\theta \\ \theta \end{bmatrix}}_{R_1} \mathbf{v}_1(\theta) d\theta + \int_s^3 \underbrace{\begin{bmatrix} \theta-s \\ \theta-s \end{bmatrix}}_{R_2} \mathbf{v}_1(\theta) d\theta \right], \quad s \in [0, 3].$$

for $\mathbf{v} = \begin{bmatrix} v_0 \\ \mathbf{v}_1 \end{bmatrix} \in \left[L_2^{\mathbb{R}^2} \right]_{L_2^1[0,3]}$. To declare this operator, we first construct the polynomial functions defining the parameters `P` through `R2`, using `pvar` objects `s` and `tt` to represent `s` and `θ`:

```
>> pvar s tt
>> P = [-1,2];
>> Q1 = (3-s^2);
>> Q2 = [0,-s; s,0];
>> R0 = [1; s^3];          R1 = [s-tt; tt];          R2 = [s; tt-s];
```

Having defined the desired parameters, we can then define the operator `C` as

```
>> opvar C;
>> C.I = [0,3];
>> C.var1 = s;          C.var2 = tt;
>> C.P = P;
>> C.Q1 = Q1;
>> C.Q2 = Q2;
>> C.R.R0 = R0;        C.R.R1 = R1;          C.R.R2 = R2
C =
  [-1,2] | [-s^2+3]
-----
  [0,-s] | C.R
  [s,0]  |

C.R =
  [1] | [s-tt] | [s]
[s^3] | [tt]  | [-s+tt]
```

using the field `R` to specify the 3-PI sub-component, and using the fields `P`, `Q1` and `Q2` to set the remaining parameters.

3.2 Declaring PI Operators in 2D

In addition to PI operators in 1D, PI operators in 2D can also be represented in `PIETOOLS`, using the `opvar2d` data structure. Here, similarly to how we distinguish 3-PI operators and 4-PI operators for 1D function spaces, we will distinguish 2 classes of 2D operators. In particular, we distinguish the standard 9-PI operators, which act on just functions $\mathbf{v} \in L_2[[a, b] \times [c, d]]$,

and the more general 2D PI operator, acting on coupled functions $\begin{bmatrix} v_0 \\ \mathbf{v}_x \\ \mathbf{v}_y \\ \mathbf{v}_2 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_x}[a,b] \\ L_2^{n_y}[c,d] \\ L_2^{n_2}[[a,b] \times [c,d]] \end{bmatrix}$.

3.2.1 Declaring 9-PI Operators

For given parameters $N = \begin{bmatrix} N_{00} & N_{01} & N_{02} \\ N_{10} & N_{11} & N_{12} \\ N_{20} & N_{21} & N_{22} \end{bmatrix}$, the associated 9-PI operator $\mathcal{P}[N] : L_2^n [[a, b] \times [c, d]] \rightarrow L_2^m [[a, b] \times [c, d]]$ is given by

$$\begin{aligned} (\mathcal{P}[N]\mathbf{v})(x, y) &= N_{00}(x, y)\mathbf{v}(x, y) + \int_c^y N_{01}(x, y, \nu)\mathbf{v}(x, \nu)d\nu + \int_y^d N_{02}(x, y, \nu)\mathbf{v}(x, \nu)d\nu \\ &+ \int_a^x N_{20}(x, y, \theta)\mathbf{v}(\theta, y)d\theta + \int_a^x \int_c^y N_{11}(x, y, \theta, \nu)\mathbf{v}(\theta, \nu)d\nu d\theta + \int_a^x \int_y^d N_{12}(x, y, \theta, \nu)\mathbf{v}(\theta, \nu)d\nu d\theta \\ &+ \int_x^b N_{20}(x, y, \theta)\mathbf{v}(\theta, y)d\theta + \int_x^b \int_c^y N_{21}(x, y, \theta, \nu)\mathbf{v}(\theta, \nu)d\nu d\theta + \int_x^b \int_y^d N_{22}(x, y, \theta, \nu)\mathbf{v}(\theta, \nu)d\nu d\theta, \end{aligned}$$

for any $\mathbf{v} \in L_2 [[a, b] \times [c, d]]$. In PIETOOLS 2022, we represent such operators using `opvar2d` class objects, which are declared in a similar manner to `opvar` objects. For example, to declare a simple operator $\mathcal{D} : L_2^2 [[0, 1] \times [1, 2]] \rightarrow L_2^2 [[0, 1] \times [1, 2]]$ defined as

$$[\mathcal{D}\mathbf{v}](s_1, s_2) = \int_0^{s_1} \int_{s_2}^2 \underbrace{\begin{bmatrix} s_1^2 & s_1 s_2 \\ s_1 s_2 & s_2^2 \end{bmatrix}}_{N_{12}} \mathbf{v}(\theta_1, \theta_2) d\theta_2 d\theta_1, \quad (s_1, s_2) \in [0, 1] \times [1, 2],$$

we first declare the parameter N_{12} defining this operator by representing s_1 and s_2 by `pvar` objects `s1` and `s2`

```
>> pvar s1 s2
>> N12 = [s1^2, s1*s2; s1*s2, s2^2];
```

Then, we initialize an empty `opvar2d` object `D` to represent \mathcal{D} , and assign the variables (s_1, s_2) and their domain $[0, 1] \times [1, 2]$ to this operator as

```
>> opvar2d D;
>> D.var1 = [s1; s2];
>> D.I = [0, 1; 1, 2];
```

Note here that, in `opvar2d` objects, `var1` is a column vector listing each of the spatial variables (s_1, s_2) on which the result $\mathbf{u}(s_1, s_2) = (\mathcal{D}\mathbf{v})(s_1, s_2)$ depends. Accordingly, the field `I` in an `opvar2d` object also has two rows, with each row specifying the interval on which the variable in the associated row of `var1` exists. Having initialized the operator, we then assign the parameter `N12` to the appropriate field. Here, the parameters defining a 9-PI operator are stored in the 3×3 cell array `D.R22`, with `R22` referring to the fact that these parameters map 2D functions to 2D functions. Within this array, element $\{i, j\}$ for $i, j \in \{1, 2, 3\}$ corresponds to parameter $N_{i-1, j-1}$ in the operator, and so we can specify parameter N_{12} using element $\{2, 3\}$:

```
>> D.R22{2,3} = N12
D =
[] | [] | [] | []
-----
[] | D.Rxx | [] | D.Rx2
-----
[] | [] | D.Ryy | D.Ry2
-----
[] | D.R2x | D.R2y | D.R22
```

```

D.Rxx =
  [] | [] | []

D.Rx2 =
  [] | [] | []

D.Ryy =
  [] | [] | []

D.Ry2 =
  [] | [] | []

D.R2x =
  [] | [] | []

D.R2y =
  [] | [] | []

D.R22 =
  [0,0] | [0,0] | [0,0]
  [0,0] | [0,0] | [0,0]
  -----
  [0,0] | [0,0] | [s1^2,s1*s2]
  [0,0] | [0,0] | [s1*s2,s2^2]
  -----
  [0,0] | [0,0] | [0,0]
  [0,0] | [0,0] | [0,0]

```

We note that, in the resulting structure, there are a lot of empty parameters, such as `D.Rxx`. As we will discuss in the next subsection, these parameters correspond to maps to and from other functions spaces, just like the parameters `P` and `Qi` in the `opvar` structure. Since the operator \mathcal{D} maps only functions $L_2^2[[0, 1] \times [1, 2]] \rightarrow L_2^2[[0, 1] \times [1, 2]]$, all parameters mapping different function spaces are empty for the object `D`.

Suppose now we want to declare a 9-PI operator $\mathcal{E} : L_2[[0, 1] \times [-1, 1]] \rightarrow L_2[[0, 1] \times [-1, 1]]$ defined by

$$\begin{aligned}
(\mathcal{E}\mathbf{v})(s_1, s_2) &= \overbrace{x^2y^2}^{N_{00}} \mathbf{v}(s_1, s_2) + \int_{-1}^{s_2} \overbrace{s_1(s_2 - \theta_2)}^{N_{01}} \mathbf{v}(s_1, \theta_2) d\theta_2 \\
&\quad + \int_{s_1}^1 \underbrace{(s_1 - \theta_1)s_2}_{N_{20}} \mathbf{v}(\theta_1, s_2) d\theta_1 + \int_{s_1}^1 \int_{-1}^{s_2} \underbrace{(s_1 - \theta_1)(s_2 - \theta_2)}_{N_{21}} \mathbf{v}(\theta_1, \theta_2) d\theta_2 d\theta_1
\end{aligned}$$

As before, we first set the values of the parameters N_{ij} , using `s1`, `s2`, `th1` and `th2` to represent s_1 , s_2 , θ_1 and θ_2 respectively:

```

>> pvar s1 s2 th1 th2
>> N00 = s1^2 * s2^2;
>> N01 = s1*(s2-th2);

```



```
>> N20 = (s1-th1)*s2;
>> N21 = (s1-th1)*(s2-th2);
```

Next, we initialize an `opvar2d` object `E` with the appropriate variables and domain as

```
>> opvar2d E;
>> E.var1 = [s1;s2];      E.var2 = [th1; th2];
>> E.I = [0,1; -1,1];
```

where in this case we set both the primary variables, using `var1`, and the dummy variables, using `var2`. Note here that the domains of the first and second dummy variables are the same as those of the first and second primary variables, and are defined in the first and second row of `I` respectively. Finally, we assign the parameters N_{ij} to the appropriate elements of `R22`

```
>> E.R22{1,1} = N00;      E.R22{1,2} = N01;
>> E.R22{3,1} = N20;      E.R22{3,2} = N21
E =
  [] |      [] |      [] |      []
-----
  [] | E.Rxx |      [] | E.Rx2
-----
  [] |      [] | E.Ryy | E.Ry2
-----
  [] | E.R2x | E.R2y | E.R22

E.R22 =

      [s1^2*s2^2] |      [s1*s2-s1*th2] | [0]
-----
              [0] |      [0] | [0]
-----
[s1*s2-s2*th1] | [s1*s2-s1*th2-s2*th1+th1*th2] | [0]
```

so that `E` represents the desired operator.

3.2.2 Declaring General 2D PI Operators

The most general PI operators that can be represented in PIETOOLS 2022 are those mapping

$$\begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_x}[a,b] \\ L_2^{n_y}[c,d] \\ L_2^{n_2}[a,b] \times [c,d] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_x}[a,b] \\ L_2^{m_y}[c,d] \\ L_2^{m_2}[a,b] \times [c,d] \end{bmatrix}, \text{ defined by parameters } R = \begin{bmatrix} R_{00} & R_{0x} & R_{0y} & R_{02} \\ R_{x0} & R_{xx} & R_{xy} & R_{x2} \\ R_{y0} & R_{yx} & R_{yy} & R_{y2} \\ R_{20} & R_{2x} & R_{2y} & R_{22} \end{bmatrix} \text{ as}$$

$$(\mathcal{P}[R]\mathbf{x})(s) = \begin{bmatrix} R_{00}v_0 & + \int_a^b R_{0x}(x)\mathbf{v}_x(x)dx & + \int_c^d R_{0y}(y)\mathbf{v}_y(y)dy & + \int_a^b \int_c^d R_{02}(x,y)\mathbf{v}_2(x,y)dydx \\ R_{x0}(x)v_0 & + (\mathcal{P}[R_{xx}]\mathbf{v}_x)(x) & + \int_c^d R_{xy}(x,y)\mathbf{v}_y(y)dy & + \int_c^d (\mathcal{P}[R_{x2}]\mathbf{v}_2)(x,y)dy \\ R_{y0}(y)v_0 & + \int_a^b R_{yx}(x,y)\mathbf{v}_x(x)dx & + (\mathcal{P}[R_{yy}]\mathbf{v}_y)(y) & + \int_a^b (\mathcal{P}[R_{y2}]\mathbf{v}_2)(x,y)dx \\ R_{20}(x,y)v_0 & + (\mathcal{P}[R_{2x}]\mathbf{v}_x)(x,y) & + (\mathcal{P}[R_{2y}]\mathbf{v}_y)(x,y) & + (\mathcal{P}[R_{22}]\mathbf{v}_2)(x,y) \end{bmatrix}$$

for $\mathbf{v} = \begin{bmatrix} v_0 \\ \mathbf{v}_x \\ \mathbf{v}_y \\ \mathbf{v}_2 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_x}[a,b] \\ L_2^{n_y}[c,d] \\ L_2^{n_2}[a,b] \times [c,d] \end{bmatrix}$, where $\mathcal{P}[R_{xx}]$, $\mathcal{P}[R_{yy}]$, $\mathcal{P}[R_{x2}]$, $\mathcal{P}[R_{y2}]$, $\mathcal{P}[R_{2x}]$ and $\mathcal{P}[R_{2y}]$ are 3-PI operators, and where $\mathcal{P}[R_{22}]$ is a 9-PI operator. These types of PI operators are also

represented using the `opvar2d` class, specifying each of the parameters R_{ij} using the associated fields `Rij`. For example, suppose we want to implement a PI operator $\mathcal{F} : \begin{bmatrix} L_2^{\mathbb{R}}[0,2] \\ L_2[0,2] \times [2,3] \end{bmatrix} \rightarrow$

$\begin{bmatrix} L_2^2[0,2] \\ L_2[0,2] \times [2,3] \end{bmatrix}$, defined as

$$(\mathcal{F}\mathbf{v})(x, y) = \left[\begin{array}{c} \overbrace{\begin{bmatrix} 1 \\ x \end{bmatrix}}^{R_{x0}} v_0 + \overbrace{\begin{bmatrix} x \\ x^2 \end{bmatrix}}^{R_{xx}^0} \mathbf{v}_1(x) + \int_x^2 \overbrace{\begin{bmatrix} 1 \\ (\theta-x) \end{bmatrix}}^{R_{xx}^2} \mathbf{v}_1(\theta) d\theta + \int_2^3 \int_0^x \overbrace{\begin{bmatrix} y \\ y^2(x-\theta) \end{bmatrix}}^{R_{x2}^1} \mathbf{v}_2(\theta, y) dy \\ \underbrace{y^2}_{R_{2x}^0} \mathbf{v}_1(x) + \underbrace{\int_0^x y}_{R_{2x}^1} \mathbf{v}_1(\theta) d\theta + \underbrace{\int_0^x \int_2^y \theta \nu}_{R_{22}^{11}} \mathbf{v}_2(\theta, \nu) d\nu d\theta \end{array} \right],$$

for $\mathbf{v} = \begin{bmatrix} v_0 \\ \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix} \in \begin{bmatrix} L_2^{\mathbb{R}}[0,2] \\ L_2[0,2] \times [2,3] \end{bmatrix}$. To declare this operator, we define the parameters as before as

```
>> pvar x y theta nu
>> Rx0 = [1; x];
>> Rxx_0 = [x; x^2];      Rxx_2 = [1; theta-x];
>> Rx2_1 = [y; y^2 * (x-theta)];
>> R2x_0 = y^2;          R2x_1 = y;
>> R22_11 = theta*nu;
```

and then declare the `opvar2d` object as

```
>> opvar2d F;
>> F.var1 = [x; y];      F.var2 = [theta; nu];
>> F.I = [0,2; 2,3];
>> F.Rx0 = Rx0;
>> F.Rxx{1} = Rxx_0;     F.Rxx{3} = Rxx_2;
>> F.Rx2{2} = Rx2_1;
>> F.R2x{1} = R2x_0;     F.R2x{2} = R2x_1;
>> F.R22{2,2} = R22_11;
```

yielding a structure

```
>> F
F =

    [] |    [] |    [] |    []
-----
 [1] | F.Rxx |    [] | F.Rx2
 [x] |      |      |
-----
    [] |    [] | F.Ryy | F.Ry2
-----
 [0] | B.R2x | F.R2y | F.R22

F.Rxx =

    [x] | [0] |    [1]
 [x^2] | [0] | [theta-x]
```

```

F.Rx2 =
    [0] | [y] | [0]
    [0] | [-theta*y^2+x*y^2] | [0]

F.Ryy =
    [] | [] | []

F.Ry2 =
    [] | [] | []

F.R2x =
    [y^2] | [y] | [0]

F.R2y =
    [] | [] | []

F.R22 =
    [0] | [0] | [0]
    -----
    [0] | [nu*theta] | [0]
    -----
    [0] | [0] | [0]

```

Representing the operator \mathcal{F} .

In the following subsection, we provide an overview of how the `opvar` and `opvar2d` data structures are defined.

3.3 Overview of opvar and opvar2d Structure

3.3.1 opvar class

Let $\mathcal{B} : \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a, b] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a, b] \end{bmatrix}$ be a 4-PI operator of the form

$$(\mathcal{B}\mathbf{x})(s) = \begin{bmatrix} Px_0 + \int_a^b Q_1(s)\mathbf{x}_1(s)ds \\ Q_2(s)x_0 + R_0(s)\mathbf{x}_1(s) + \int_a^s R_1(s, \theta)\mathbf{x}_1(\theta)d\theta + \int_s^b R_2(s, \theta)\mathbf{x}_1(\theta)d\theta \end{bmatrix} \quad (3.3)$$

for $\mathbf{x} = \begin{bmatrix} x_0 \\ \mathbf{x}_1 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a, b] \end{bmatrix}$. Then, we can represent this operator as an opvar object B with fields as defined in Table 3.1.

B.dim	= [m0,n0; m1,n1]	2×2 array of type <code>double</code> specifying the dimensions of the function spaces $\begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a, b] \end{bmatrix}$ and $\begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a, b] \end{bmatrix}$ the operator maps to and from;
B.var1	= <code>s</code>	1×1 <code>pvar</code> (<code>polynomial class</code>) object specifying the spatial variable s ;
B.var2	= <code>theta</code>	1×1 <code>pvar</code> (<code>polynomial class</code>) object specifying the dummy variable θ ;
B.I	= [a,b]	1×2 array of type <code>double</code> , specifying the interval $[a, b]$ on which the spatial variables s and θ exist;
B.P	= <code>P</code>	$m_0 \times n_0$ array of type <code>double</code> or <code>polynomial</code> defining the matrix P ;
B.Q1	= <code>Q1</code>	$m_0 \times n_1$ array of type <code>double</code> or <code>polynomial</code> defining the function $Q_1(s)$;
B.Q2	= <code>Q2</code>	$m_1 \times n_0$ array of type <code>double</code> or <code>polynomial</code> defining the function $Q_2(s)$;
B.R.R0	= <code>R0</code>	$m_1 \times n_1$ array of type <code>double</code> or <code>polynomial</code> defining the function $R_0(s)$;
B.R.R1	= <code>R1</code>	$m_1 \times n_1$ array of type <code>double</code> or <code>polynomial</code> defining the function $R_1(s, \theta)$;
B.R.R2	= <code>R2</code>	$m_1 \times n_1$ array of type <code>double</code> or <code>polynomial</code> defining the function $R_2(s, \theta)$;

Table 3.1: Fields in an opvar object B, defining a general 4-PI operator as in Equation (3.3)

3.3.2 opvar2d class

Let $\mathcal{D} : \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_x}[a, b] \\ L_2^{n_y}[c, d] \\ L_2^{n_2}[[a, b] \times [c, d]] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_x}[a, b] \\ L_2^{m_y}[c, d] \\ L_2^{m_2}[[a, b] \times [c, d]] \end{bmatrix}$ be a PI operator of the form

$$(\mathcal{D}\mathbf{x})(s) = \begin{bmatrix} R_{00}v_0 + \int_a^b R_{0x}(x)\mathbf{v}_x(x)dx + \int_c^d R_{0y}(y)\mathbf{v}_y(y)dy + \int_a^b \int_c^d R_{02}(x, y)\mathbf{v}_2(x, y)dydx \\ R_{x0}(x)v_0 + (\mathcal{P}[R_{xx}]\mathbf{v}_x)(x) + \int_c^d R_{xy}(x, y)\mathbf{v}_y(y)dy + \int_c^d (\mathcal{P}[R_{x2}]\mathbf{v}_2)(x, y)dy \\ R_{y0}(y)v_0 + \int_a^b R_{yx}(x, y)\mathbf{v}_x(x)dx + (\mathcal{P}[R_{yy}]\mathbf{v}_y)(y) + \int_a^b (\mathcal{P}[R_{y2}]\mathbf{v}_2)(x, y)dx \\ R_{20}(x, y)v_0 + (\mathcal{P}[R_{2x}]\mathbf{v}_x)(x, y) + (\mathcal{P}[R_{2y}]\mathbf{v}_y)(x, y) + (\mathcal{P}[R_{22}]\mathbf{v}_2)(x, y) \end{bmatrix} \quad (3.4)$$

for $\mathbf{v} = \begin{bmatrix} v_0 \\ \mathbf{v}_x \\ \mathbf{v}_y \\ \mathbf{v}_2 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_x}[a, b] \\ L_2^{n_y}[c, d] \\ L_2^{n_2}[[a, b] \times [c, d]] \end{bmatrix}$, where $\mathcal{P}[R_{xx}]$, $\mathcal{P}[R_{yy}]$, $\mathcal{P}[R_{x2}]$, $\mathcal{P}[R_{y2}]$, $\mathcal{P}[R_{2x}]$ and $\mathcal{P}[R_{2y}]$

are 3-PI operators, and where $\mathcal{P}[R_{22}]$ is a 9-PI operator. We can represent the operator \mathcal{D} as an opvar2d object D with fields as defined in Table 3.2.

D.dim	= [m0,n0; mx,nx; my,ny; m2,n2;]	4 × 2 array of type double specifying the dimensions of the function spaces $\begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_x}[a,b] \\ L_2^{m_y}[c,d] \\ L_2^{m_2}[[a,b] \times [c,d]] \end{bmatrix}$ and $\begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_x}[a,b] \\ L_2^{n_y}[c,d] \\ L_2^{n_2}[[a,b] \times [c,d]] \end{bmatrix}$ the operator maps to and from;
D.var1	= [x; y]	2 × 1 pvar (polynomial class) object specifying the spatial variables (x, y);
D.var2	= [theta; nu]	2 × 1 pvar (polynomial class) object specifying the dummy variables (θ, ν);
D.I	= [a,b; c,d]	2 × 2 array of type double, specifying the domain [a, b] × [c, d] on which the spatial variables (x, θ) and (y, ν) exist;
D.R00	= R00	$m_0 \times n_0$ array of type double or polynomial defining the matrix R_{00} ;
D.R0x	= R0x	$m_0 \times n_x$ array of type double or polynomial defining the function $R_{0x}(x)$;
D.R0y	= R0y	$m_0 \times n_y$ array of type double or polynomial defining the function $R_{0y}(y)$;
D.R02	= R02	$m_0 \times n_2$ array of type double or polynomial defining the function $R_{02}(x, y)$;
D.Rx0	= Rx0	$m_x \times n_0$ array of type double or polynomial defining the function $R_{x0}(x)$;
D.Rxx	= Rxx	3 × 1 cell array specifying the 3-PI parameters R_{xx} ;
D.Rxy	= Rxy	$m_x \times n_y$ array of type double or polynomial defining the function $R_{xy}(x, y)$;
D.Rx2	= Rx2	3 × 1 cell array specifying the 3-PI parameters R_{x2} ;
D.Ry0	= Ry0	$m_y \times n_0$ array of type double or polynomial defining the function $R_{y0}(y)$;
D.Ryx	= Ryx	$m_y \times n_x$ array of type double or polynomial defining the function $R_{yx}(x, y)$;
D.Ryy	= Ryy	1 × 3 cell array specifying the 3-PI parameters R_{yy} ;
D.Ry2	= Ry2	1 × 3 cell array specifying the 3-PI parameters R_{y2} ;
D.R20	= R20	$m_2 \times n_0$ array of type double or polynomial defining the function $R_{20}(x, y)$;
D.R2x	= R2x	3 × 1 cell array specifying the 3-PI parameters R_{2x} ;
D.R2y	= R2y	1 × 3 cell array specifying the 3-PI parameters R_{2y} ;
D.R22	= R22	3 × 3 cell array specifying the 9-PI parameters R_{22} ;

Table 3.2: Fields in an opvar2d object D, defining a general PI operator in 2D as in Equation (3.4)

Part I

PIETOOLS Workflow for ODE-PDE and DDE Models

Chapter 4

Setup and Representation of PDEs and DDEs

Using PIETOOLS, a wide variety of linear differential equations and time-delay systems can be simulated and analysed by representing them as partial integral equations (PIEs). To facilitate this, PIETOOLS includes several input format to declare partial differential equations (PDEs) and delay-differential equations (DDEs), which can then be easily converted to equivalent PIEs using the PIETOOLS function `convert`, as we show in Chapter 5. In this chapter, we present two of these input formats, discussing in detail how linear 1D PDE and DDE systems can be easily implemented using the Command Line Parser for PDEs and Batch-Based input format for DDEs. We refer to Chapter 8 for information on two alternative input formats for PDEs, including a format to parse 2D PDEs, and we refer to Chapter 9 for two alternative input formats for time-delay systems, namely the Neutral Delay System (NDS) and Delay Differential Equation (DDF) formats.

4.1 Command Line Parser for 1D ODE-PDEs

In PIETOOLS 2022, by far the simplest and most intuitive of these input formats is the Command Line Parser format. Command Line Parser format utilizes MATLAB variables of class `state` to define symbols that can be freely manipulated to express PDEs as MATLAB expressions that are stored in the `sys` class object. `sys` class objects are used to store parameters of a coupled ODE-PDE and to perform analysis, control, and simulation of the stored system. Refer to 8.3.2 for more details.

Requirements

- Command Line Parser format can be used to defined ODE-PDEs with ONLY one spatial dimension.
- Command Line Parser format supports ODEs/PDEs with time-delayed terms.
- Command Line Parser format requires MATLAB 2021a or higher.
- Variables `s`, `t`, and `theta` are protected and cannot be used for other purposes.

4.1.1 Defining a coupled ODE-PDE system

For the purpose of demonstration, consider the following coupled ODE-PDE system in control theory framework

$$\begin{aligned}\dot{x}(t) &= -5x(t) + \int_0^1 \partial_s \mathbf{x}(t, s) ds + u(t) \\ \dot{\mathbf{x}}(t, s) &= 9\mathbf{x}(t, s) + \partial_s^2 \mathbf{x}(t, s) + sw(t) \\ \mathbf{x}(t, 0) &= 0, \quad \partial_s \mathbf{x}(t, 1) + x(t) = 2w(t) \\ z(t) &= \begin{bmatrix} \int_0^1 \mathbf{x}(t, s) ds \\ u(t) \end{bmatrix} \\ y(t) &= \mathbf{x}(t, 0).\end{aligned}$$

Code Block 1

The above set of equations can be defined and converted to a PIE using the code shown below.

```
>> pvar t s theta;
>> x = state('ode'); X = state('pde');
>> w = state('in'); z = state('out',2);
>> u = state('in'); y = state('out');
>> odepde= sys();
>> odepde = addequation(odepde, z==[int(X,s,[0,1]); u]);
>> eqns = [diff(x,t)==-5*x+int(diff(X,s,1),s,[0,1])+u;
diff(X,t)==9*X+diff(X,s,2)+s*w;
subs(X,s,0)==0;
subs(diff(X,s),s,1)==-x+2*w;
y==subs(X,s,0)];
>> odepde = addequation(odepde,eqns);
>> odepde = setControl(odepde,[u]);
>> odepde = setObserve(odepde,[y]);
>> sys_pie = convert(odepde,'pie');
```

Next we will breakdown each step used in the code above and explain the action performed by each line of the above code block. Specifying any PDE system using the 'Command Line Parser' format follows the steps listed below:

1. Define independent variables (s, t)
2. Define dependent variables (\mathbf{x}, x, z, y, w and u)
3. Define a `sys()` object to store the equations
4. Add equations
5. Specify control inputs and observed outputs

4.1.1a Define independent variables

To define equations symbolically, first, the independent variables (spatial variable and time variable) and dependent variables (states, inputs, and outputs) have to be declared. For example, if the PDE is defined on space s and time t , we would start by defining these variables as `polynomial` objects as shown below.

```
| >> pvar t s theta; % independent variables are polynomial objects
```

Note that we have defined additional variable `theta` which will be used as a dummy spatial variable if needed (for example, in operations involving integration).

4.1.1b Define dependent variables

After defining independent variables, we need to define dependent variables such as ODE/PDE states, inputs, and outputs (See 2 for details). Dependent variables are defined as `state` class objects. For example:

```
| >> x = state('ode'); X = state('pde');  
| >> w = state('in'); z = state('out',2);  
| >> u = state('in'); y = state('out');
```

The above code, when executed in MATLAB, creates four symbolic variables, namely `x`, `X`, `w`, `u`, `z`, `y`, and assigns them the type ODE, PDE, input (`w`, `u`), and output (`z`, `y`) respectively. For more details on the usage of `state` class see 8.3.1. Now that the dependent and independent variables have been established, we define the ODE-PDE equations using these symbolic variables.

In this example, the output `z` must be of length 2 because

$$z(t) = \begin{bmatrix} \int_0^1 \mathbf{x}(t, s) ds \\ u(t) \end{bmatrix}.$$

The length is specified using the second argument as shown in the code

```
| >> z = state('out',2);
```

where the second argument to `state()` function always specifies length of the vector.

4.1.1c Define a PDE object, `sys()`

Now that all the required symbols are defined, we can initialize a `sys` class object and add equations to the object.

```
| >> odepde= sys();  
|  
| Initialized sys() object of type ‘‘pde’’
```

By default, `sys` objects are initialized as type ‘pde’ on the domain $[0, 1]$. The domain can be modified, if needed, by using the following command.

```
| >> odepde.dom = [0,3];
```

The above command changes the domain of all relevant PDE states to $[0, 3]$. For the current example, we assume the domain is $[0, 1]$ and proceed.

Warning

The domain parameter MUST be set before adding the equations to the system.

4.1.1d Define equations and add to PDE object

We can individually add equations one at a time using `addequation()` method or group the equations into a column vector and add all the equations together. For example, we can add the following equation to the `sys()`

$$z(t) = \begin{bmatrix} \int_0^1 \mathbf{x}(t, s) ds \\ u(t) \end{bmatrix}$$

using the command

```
>> odepde = addequation(odepde, z==[int(X,s,[0,1]); u]); % one equation at a time
      2 equations were added to sys() object
```

Alternatively, we can define all the equations together in a column vector and add them to the system in one command. We can add the remaining 5 equations and boundary conditions listed below

$$\begin{aligned} \dot{x}(t) &= -5x(t) + \int_0^1 \partial_s \mathbf{x}(t, s) ds + u(t) \\ \dot{\mathbf{x}}(t, s) &= 9\mathbf{x}(t, s) + \partial_s^2 \mathbf{x}(t, s) + s w(t), \quad \mathbf{x}(t, 0) = 0, \quad \partial_s \mathbf{x}(t, 1) + x(t) = 2w(t) \\ y(t) &= \mathbf{x}(t, 0). \end{aligned}$$

using the code

```
>> eqns = [diff(x,t)==-5*x+int(X,s,[0,1])+u; diff(X,t)==9*X+diff(X,s,2)+s*w;
subs(X,s,0)==0; subs(diff(X,s),s,1)==-x+2*w; y== subs(x,s,0)];
>> odepde = addequation(odepde,eqns);
      5 equations were added to sys() object
```

Any expression passed to `addequation` function in the form `addequation(odepde,expr)` results in addition of the equation '`expr=0`' to the `odepde` object.

Note

The `=` symbol is not used while defining equations. Instead `==` is used because MATLAB uses `=` as a protected symbol for assignment operation. Thus, any symbolic expression that needs to be added takes the form `expr==0` or `exprA==exprB`.

4.1.1e Specify control inputs and observed outputs

By default, all inputs are defined as disturbance inputs and all outputs are defined as regulated outputs. To explicitly assign, certain inputs as control inputs, one must use `setControl()` function. In the above example, we can specify `u` to be control input by using the following command.

```
odepde = setControl(odepde,[u]); % designate control inputs
    1 inputs were designated as controlled inputs
```

Likewise, observed outputs can be added to the system as shown below.

```
odepde = setObserve(odepde,[y]);
    1 outputs were designated as observed outputs
```

Functions such as `removeControl` and `removeObserve` are included to reset a particular input as disturbance (output as regulated output). While this is the last step in defining a system using the command line parser format, we can directly convert the `sys` objects to PIE objects using a function as shown the next subsection.

4.1.1f Getting PIE from `sys` class objects

Once the ODE-PDE class object has been defined, we can obtain the PIE class object, namely `pie_struct`, using the `convert` function.

```
sys_pie = convert(odepde,'pie');
Conversion to 'pie' was successful
```

After executing the code in the above ‘Code block’, the PIE system parameters are stored under `params` property which can be accessed to obtain the following output

```
>> sys_pie.params
ans =
pie_struct with properties:

    dim: 1;
   vars: [1×2 polynomial];
    dom: [1×2 double];

    T: [2×2 opvar];    Tw: [2×1 opvar];    Tu: [2×1 opvar];
    A: [2×2 opvar];    B1: [2×1 opvar];    B2: [2×1 opvar];
   C1: [2×2 opvar];    D11: [2×1 opvar];    D12: [2×1 opvar];
   C2: [1×2 opvar];    D21: [1×1 opvar];    D22: [1×1 opvar];
```

Once the PIE structure is obtained, we can proceed to perform analysis, control, and simulation, as discussed in detail in Chapters 6 and 7.

4.1.2 More examples of command line parser format

More details on the implementation of Command Line Parser format can be found in Section 8.3. In this subsection, we provide a few more examples to demonstrate the typical use of the command line parser. More specifically, we focus on examples involving inputs, outputs, delays, vector-valued PDEs, etc., to demonstrate the capabilities of command line parser.

4.1.2a Example: Transport equation

Consider the Transport equation which is modeled as a PDE with 1st derivatives in time and space given by

$$\begin{aligned}\dot{\mathbf{x}}(t, s) &= 5\partial_s \mathbf{x}(t, s) + u(t), \quad s \in [0, 2] \\ y(t) &= x(t, 2), \\ \mathbf{x}(t, 0) &= 0.\end{aligned}$$

Here, we use a control input in the domain and an observer at the right boundary with an intention to design an observer based controller. This system can be defined using the command line parser format as shown below.

Code Block 2

The above set of equations can be defined and converted to a PIE using the code shown below.

```
>> pvar t s theta;
>> X = state('pde');
>> u = state('in'); y = state('out');
>> odepde = sys();
>> odepde.dom = [0,2];
>> eqns = [diff(X,t)==5*diff(X,s)+u;
subs(X,s,0)==0; y==subs(x,s,2)];
>> odepde = addequation(odepde,eqns);
>> odepde = setControl(odepde,[u]);
>> odepde = setObserve(odepde,[y]);
>> sys_pie = convert(odepde,'pie');
```

The steps are identical to the one specified in the previous section. The only exception is the change of domain to [0,2]. First, we define all the independent variables as pvar objects and dependent variables as state objects. Then we define the equations and add the equations to a system storage object. Finally, we specify which symbols are control inputs and observed outputs and convert the system to a PIE system. By accessing `params` property, we obtain the following output

```
>> sys_pie.params

ans =
pie_struct with properties:

    dim: 1;
   vars: [1x2 polynomial];
    dom: [1x2 double];

    T: [1x1 opvar];    Tw: [1x0 opvar];    Tu: [1x1 opvar];
    A: [1x1 opvar];    B1: [1x0 opvar];    B2: [1x1 opvar];
   C1: [0x1 opvar];    D11: [0x0 opvar];    D12: [0x1 opvar];
   C2: [1x1 opvar];    D21: [0x0 opvar];    D22: [0x1 opvar];
```

Once the PIE system is obtained, we can use the LPIs in Chapter 7 to design an observer and controller for this PDE.

4.1.2b Example: PDE with delay terms

Consider a reaction-diffusion equation which are modeled as a PDE with 2^{nd} -order derivatives in both space and first order derivative in time. Let there be an dynamic ODE system coupled with the PDE through a channel that is delayed by an amount $\tau = 2$. The coupled ODE-PDE model for this system is given by the equations

$$\begin{aligned}\dot{x}(t) &= -5x(t), \\ \dot{\mathbf{x}}(t, s) &= 10\mathbf{x}(t, s) + \partial_s^2 \mathbf{x}(t, s) + x(t - 2), \\ \mathbf{x}(t, 0) = 0 &= \mathbf{x}(t, 1).\end{aligned}$$

Code Block 3

The above set of equations can be defined and converted to a PIE using the code shown below.

```
>> x = state('ode');
>> X = state('pde');
>> pvar s t theta;
>> ss = sys();
>> eqns = [diff(x,t)==-5*x; diff(X,t)==diff(X,s,2)+subs(x,t,t-2);
subs(X,s,0)==0;subs(X,s,1)==0;];
>> ss = addequation(ss,eqns);
>> ss = convert(ss,'pie');
```

In the above system, the delay term is converted to a new state, \mathbf{v} , that is governed by a transport equation. The new set of equations is then given by

$$\begin{aligned}\dot{x}(t) &= -5x(t), \\ \dot{\mathbf{x}}(t, s) &= 10\mathbf{x}(t, s) + \partial_s^2 \mathbf{x}(t, s) + \mathbf{v}(t, -2), \\ \dot{\mathbf{v}}(t, \theta) &= \partial_\theta \mathbf{v}(t, \theta), \\ \mathbf{x}(t, 0) = 0 &= \mathbf{x}(t, 1) \quad \mathbf{v}(t, 0) = x(t).\end{aligned}$$

Note: The above conversion is performed internally, and the users only need to use the input format shown in the code block above.

Thus, the resulting PIE will have two distributed states and one finite dimensional state which can be verified by looking at the dimensions of the parameters stored in `ss.params` property:

```
>> ss.params

ans =
pie_struct with properties:

    dim: 2;
   vars: [2x2 polynomial];
    dom: [2x2 double];

    T: [3x3 opvar2d];    Tw: [3x0 opvar2d];    Tu: [3x0 opvar2d];
```

A: [3×3 opvar2d];	B1: [3×0 opvar2d];	B2: [3×0 opvar2d];
C1: [0×3 opvar2d];	D11: [0×0 opvar2d];	D12: [0×0 opvar2d];
C2: [0×3 opvar2d];	D21: [0×0 opvar2d];	D22: [0×0 opvar2d];

4.1.2c Example: Beam equation

Here we consider the Timoshenko Beam equations which are modeled as a PDE with 2^{nd} -order derivatives in both space and time. While this system cannot be directly input using the command line parser format, we can redefine the state variables to convert it to a PDE with first order temporal derivative as shown below.

$$\begin{aligned}\ddot{w} &= \partial_s(w_s - \phi), & \ddot{\phi} &= \phi_{ss} + (w_s - \phi) \\ \phi(0) = w(0) &= 0, & \phi_s(1) &= 0, & w_s(1) - \phi(1) &= 0.\end{aligned}$$

By choosing $\mathbf{x} = [\dot{w}, w_s - \phi, \dot{\phi}, \phi_s]$, we get

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

which is a vector-valued transport equation with a reaction term. Next, we define this system using the command line parser as shown below.

Code Block 4

The above set of equations can be defined and converted to a PIE using the code shown below.

```
>> x = state('pde',4);
>> pvar s t theta;
>> ss = sys();
>> A0 = [0,0,0,0;0,0,-1,0;0,1,0,0;0,0,0,0];
>> A1 = [0,1,0,0;1,0,0,0;0,0,0,1;0,0,1,0];
>> B = [1,0,0,0,0,0,0,0;0,0,1,0,0,0,0,0;0,0,0,0,0,0,0,1;0,0,0,0,0,1,0,0];
>> eqns = [diff(x,t)==A0*x+A1*diff(x,s); B*[subs(x,s,0); subs(x,s,1)]==0];
>> ss = addequation(ss,eqns);
>> ss = convert(ss,'pie');
```

As seen above, the presence of vector-valued states does not change the typical workflow to define the PDE. As long as the dimensions of the parameters and vectors used in the equations match, the process and steps remain the same.

The above code should generate a PIE system with `params` as shown below:

```

>> ss.params

ans =
  pie_struct with properties:

    dim: 1;
   vars: [1×2 polynomial];
    dom: [1×2 double];

    T: [4×4 opvar];    Tw: [4×0 opvar];    Tu: [4×0 opvar];
    A: [4×4 opvar];    B1: [4×0 opvar];    B2: [4×0 opvar];
   C1: [0×4 opvar];    D11: [0×0 opvar];    D12: [0×0 opvar];
   C2: [0×4 opvar];    D21: [0×0 opvar];    D22: [0×0 opvar];

```

4.2 Alternative Input Formats for PDEs

In addition to the command line parser input format, PIETOOLS 2022 offers two additional methods for declaring PDEs. In particular, PIETOOLS comes with a graphical user interface (GUI) that allows users to simultaneously visualize the PDE that they are specifying, as well as “terms-based” input format, which is the only input format to declare PDEs in multiple spatial variables in PIETOOLS 2022. We briefly introduce both of these input formats here, focusing on the GUI in Subsection 4.2.1, and the terms-based input format in Subsection 4.2.2. Note that we provide only a brief introduction of each format here, referring to Chapter 8 for more details.

4.2.1 A GUI for Declaring PDEs

Aside from the command line parser, the GUI is the easiest way to declare linear 1D ODE-PDE systems in PIETOOLS, providing a simple, intuitive and interactive visual interface to directly input the model. The GUI can be opened by running `PIETOOLS.PIETOOLS_GUI` from the command line, opening a window like the one displayed in the picture below:

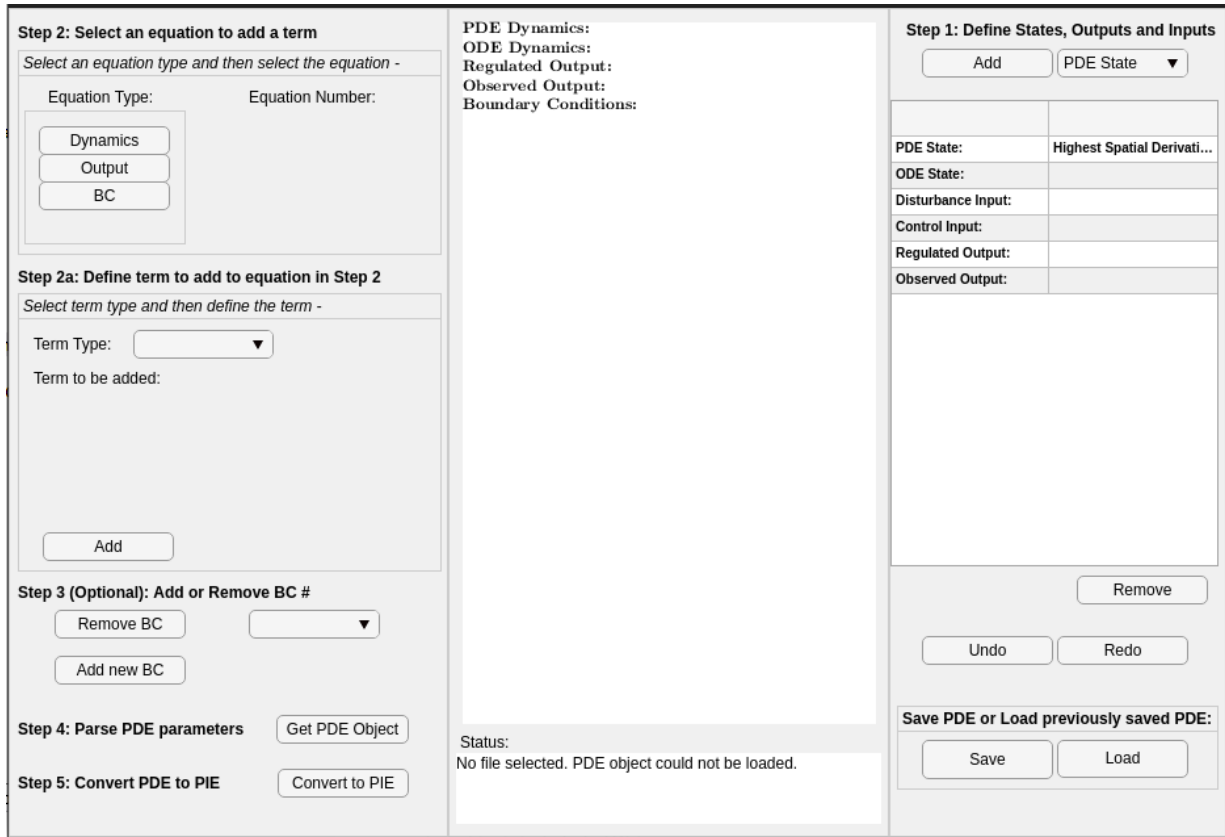


Figure 4.1: Example of empty GUI window.

Then, the desired PDE can be declared following steps 1 through 4, first specifying the state variables, inputs and outputs, then declaring the different equations term by term, and finally adding any boundary conditions. It also allows PDE models to be saved and loaded, so that e.g. the system

$$\begin{aligned} \dot{\mathbf{x}}(t, s) &= \partial_s^2 \mathbf{x}(t, s) + sw(t), & s \in [0, 1] \\ z(t) &= \int_0^1 \mathbf{x}(t, s) ds, \\ \mathbf{x}(t, 0) &= 0, \quad \mathbf{x}(t, 1) = 0, \end{aligned}$$

can be retrieved by simply loading the file
 PIETOOLS_PDE.Ex.Heat.Eq.with.Distributed.Disturbance.GUI
 from the library of PDE examples, returning a window that looks like

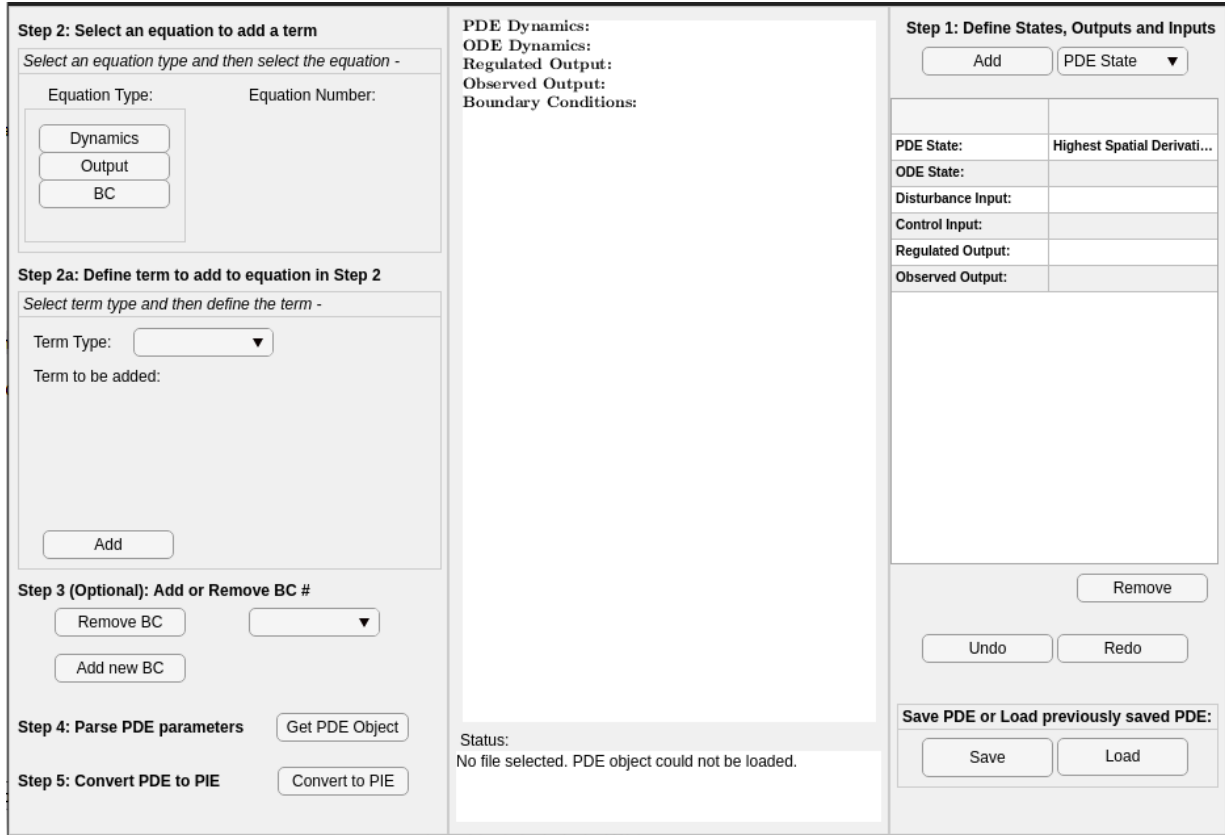


Figure 4.2: GUI window after loading the file `PIETOOLS_PDE_Ex_Heat_Eq_with_Distributed_Disturbance_GUI` from the library of PDE examples.

Then, the system can be parsed by clicking `Get PDE Objects`, returning a structure `PDE_GUI` in the MATLAB workspace that can be used for further analysis.

For more details on how to use the GUI, we refer to Section 8.1.

4.2.2 An Input Format for 2D PDEs

In PIETOOLS 2022, the terms-based input format is the only way to declare 2D PDEs. In this format, a PDE is represented as a `pde_struct` object. Each term in each equation in the PDE is then implemented separately.

For example, to declare a 2D heat equation,

$$\begin{aligned} \dot{\mathbf{x}}(t, s_1, s_2) &= \partial_{s_1}^2 \mathbf{x}(t, s_1, s_2) + \partial_{s_2}^2 \mathbf{x}(t, s_1, s_2) + w(t), & (s_1, s_2) &\in [-1, 1] \times [0, 1], \\ z(t) &= \int_{-1}^1 \int_0^1 \mathbf{x}(t, s_1, s_2) ds_2 ds_1, \\ \mathbf{x}(t, -1, s_2) &= 0, & \mathbf{x}(t, 1, s_2) &= 0, \\ \mathbf{x}(t, s_1, 0) &= 0, & \mathbf{x}(t, s_1, 1) &= 0, \end{aligned}$$

we first declare spatial variables (s_1, s_2) as `s1` and `s2`, and initialize an empty `pde_struct` object `PDE` as

```

>> pvar s1 s2
>> PDE = pde_struct();

```

Then, we declare the state $x(t, s_1, s_2)$, input $w(t)$ and output $z(t)$ as

```

>> PDE.x{1}.vars = [s1;s2];
>> PDE.x{1}.dom = [-1,1; 0,1];
>> PDE.w{1}.vars = [];
>> PDE.z{1}.vars = [];

```

using the field `vars` to specify the spatial variables on which each component depends, and the field `dom` to specify the spatial domain on which these variables exist. Then, the PDE can be implemented one term at a time as

```

>> PDE.x{1}.term{1}.x = 1;      PDE.x{1}.term{2}.x = 1;      PDE.x{1}.term{3}.w = 1;
>> PDE.x{1}.term{1}.D = [2,0];  PDE.x{1}.term{2}.D = [0,2];

```

using the fields `x` and `w` to indicate whether each term involves a state component or input, and the field `D` to specify the order of derivative of the state component in the term. The output equation can be similarly specified as

```

>> PDE.z{1}.term{1}.x = 1;
>> PDE.z{1}.term{1}.I{1} = [-1,1];    PDE.z{1}.term{1}.I{2} = [0,1];

```

using the field `I` to specify the desired domain of integration of the state component along each spatial direction. Finally, the boundary conditions can be declared as

```

>> PDE.BC{1}.term{1}.x = 1;      PDE.BC{2}.term{1}.x = 1;
>> PDE.BC{1}.term{1}.loc = [-1,s2];  PDE.BC{2}.term{1}.loc = [1,s2];
>> PDE.BC{3}.term{1}.x = 1;      PDE.BC{4}.term{1}.x = 1;
>> PDE.BC{3}.term{1}.loc = [s1,0];  PDE.BC{4}.term{1}.loc = [s1,1];

```

Then, the PDE can be initialized by calling `PDE = initialize(PDE)`, returning a structure that can be used for analysis and simulation.

For more details on how to use the terms-based input format to declare (2D) PDEs, we refer to Section 8.2.

4.3 Batch Input Format for DDEs

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

$$\begin{aligned}
\begin{bmatrix} \dot{x}(t) \\ z(t) \\ y(t) \end{bmatrix} &= \begin{bmatrix} A_0 & B_1 & B_2 \\ C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{bmatrix} \begin{bmatrix} x(t) \\ w(t) \\ u(t) \end{bmatrix} + \sum_{i=1}^K \begin{bmatrix} A_i & B_{1i} & B_{2i} \\ C_{1i} & D_{11i} & D_{12i} \\ C_{2i} & D_{21i} & D_{22i} \end{bmatrix} \begin{bmatrix} x(t - \tau_i) \\ w(t - \tau_i) \\ u(t - \tau_i) \end{bmatrix} \\
&+ \sum_{i=1}^K \int_{-\tau_i}^0 \begin{bmatrix} A_{di}(s) & B_{1di}(s) & B_{2di}(s) \\ C_{1di}(s) & D_{11di}(s) & D_{12di}(s) \\ C_{2di}(s) & D_{21di}(s) & D_{22di}(s) \end{bmatrix} \begin{bmatrix} x(t+s) \\ w(t+s) \\ u(t+s) \end{bmatrix} ds
\end{aligned} \tag{4.1}$$

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.Ai{1} = -1;
>> DDE.C1i{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 4.1.

4.3.1 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.

4.4 Alternative Input Formats for TDSs

Although the delay differential equation (DDE) format is perhaps the most intuitive format for representing time-delay systems (TDS), it is not the only representation of TDS systems, and not every TDS can be represented in this format. For this reason, PIETOOLS includes two additional input format for TDSs, namely the Neutral Type System (NDS) representation,

ODE Terms:					
Eqn. (4.1)	DDE.	Eqn. (4.1)	DDE.	Eqn. (4.1)	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. (4.1)	DDE.	Eqn. (4.1)	DDE.	Eqn. (4.1)	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. (4.1)	DDE.	Eqn. (4.1)	DDE.	Eqn. (4.1)	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 4.1: Equivalent names of Matlab elements of the DDE structure terms for terms in Eqn. (4.1). 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].

and Differential-Difference Equation (DDF) representation. Here, the structure of a NDS is identical to that of a DDE except for 6 additional terms:

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

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, and should be included in a NDS object as, e.g. NDS.E{1}=1. On the other hand, the DDF representation is more compact but less transparent than the DDE and NDS representation, taking the form

$$\begin{aligned}
 \begin{bmatrix} \dot{x}(t) \\ z(t) \\ y(t) \\ r_i(t) \end{bmatrix} &= \begin{bmatrix} A_0 & B_1 & B_2 \\ C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \\ C_{ri} & B_{r1i} & B_{r2i} \end{bmatrix} \begin{bmatrix} x(t) \\ w(t) \\ u(t) \end{bmatrix} + \begin{bmatrix} B_v \\ D_{1v} \\ D_{2v} \\ D_{rvi} \end{bmatrix} v(t) \\
 v(t) &= \sum_{i=1}^K C_{vi} r_i(t - \tau_i) + \sum_{i=1}^K \int_{-\tau_i}^0 C_{vdi}(s) r_i(t+s) ds.
 \end{aligned}$$

In this representation, the output signal from the ODE part is decomposed into sub-components r_i , each of which is delayed by amount τ_i . Identifying these sub-components is often challenging, so in most cases it will be preferable to use the NDS or DDE representation instead. However, the DDF representation is more general than either the DDE or NDS representation, so PIETOOLS also includes an input format for declaring DDF systems. For more information on how to declare systems in the DDF or NDS representation, and how to convert between different representations, we refer to Chapter 9.

Chapter 5

Converting PDEs and DDEs to PIEs

In the previous chapter, we showed how general linear 1D ODE-PDE and DDE systems can be declared in PIETOOLS. In order to analyze such systems, PIETOOLS represents each of them in a standardized format, as a partial integral equation (PIE). This format is parameterized by partial integral, or PI operators, rather than by differential operators, allowing PIEs to be analysed by solving optimization problems on these PI operators (see Chapter 7).

In this chapter, we show how an equivalent PIE representation of PDE and DDE systems can be computed in PIETOOLS. In particular, in Section 5.1, we first provide a simple illustration of what a PIE is. In Sections 5.2 and 5.3, we then show how a PDE and a DDE can be converted to a PIE, and in Section 5.4, we show how a PDE with inputs and outputs can be converted to a PIE. To reduce notation, we demonstrate the PDE conversion only for 1D systems, though we note that the same steps also work for 2D PDEs.

5.1 What is a PIE?

To illustrate the concept of partial integral equations, suppose we have a simple 1D PDE

$$\begin{aligned}\dot{\mathbf{x}}(t, s) &= 2\partial_s \mathbf{x}(t, s) + 10\mathbf{x}(t, s), & s \in [0, 1], \\ \mathbf{x}(t, 0) &= 0.\end{aligned}\tag{5.1}$$

In this system, the PDE state $\mathbf{x}(t)$ at any time $t \geq 0$ is a function of s , that has to satisfy the boundary condition (BC) $\mathbf{x}(t, 0) = 0$. Moreover, the state must be at least first order differentiable with respect to s , for us to be able to evaluate the derivative $\partial_s \mathbf{x}(t, s)$. As such, a more fundamental state would actually be this first order derivative $\partial_s \mathbf{x}(t)$ of the state, which does not need to be differentiable, nor does it need to satisfy any boundary conditions. We therefore define $\mathbf{x}_f(t, s) := \partial_s \mathbf{x}(t, s)$ as the *fundamental state* associated to this PDE. Using the fundamental theorem of calculus, we can then express the PDE state in terms of the fundamental state as

$$\mathbf{x}(t, s) = \mathbf{x}(t, 0) + \int_0^s \partial_s \mathbf{x}(t, \theta) d\theta = \mathbf{x}(t, 0) + \int_0^s \mathbf{x}_f(t, \theta) d\theta = \int_0^s \mathbf{x}_f(t, \theta) d\theta,$$

where we invoke the boundary condition $\mathbf{x}(t, 0) = 0$. Substituting this result into the PDE, we arrive at an equivalent representation of the system as

$$\int_0^s \dot{\mathbf{x}}_f(t, \theta) d\theta = 2\mathbf{x}_f(t, s) + \int_0^s 10\mathbf{x}_f(t, \theta) d\theta, \quad s \in [0, 1],\tag{5.2}$$

in which the fundamental state \mathbf{x}_f does not need to satisfy any boundary conditions, nor does it need to be differentiable with respect to s . We refer to this representation as the Partial Integral Equation, or PIE representation of the system, involving only partial integrals, rather than partial derivatives with respect to s . It can be shown that for any well-posed linear PDE – meaning that the solution to the PDE is uniquely defined by the dynamics and the BCs – there exists an equivalent PIE representation. In PIETOOLS, this equivalent representation can be obtained by simply calling `convert` for the desired PDE structure `PDE`, returning a structure `PIE` that corresponds to the equivalent PIE representation.

5.2 Converting a PDE to a PIE

Suppose that we have a PDE structure `PDE`, defining a 1D heat equation with integral boundary conditions:

$$\begin{aligned} \dot{\mathbf{x}}(t, s) &= \partial_s^2 \mathbf{x}(t, s), & s \in [0, 1] \\ \text{with BCs } \mathbf{x}(t, 0) + \int_0^1 \mathbf{x}(t, s) ds &= 0, & \mathbf{x}(t, 1) + \int_0^1 \mathbf{x}(t, s) ds = 0 \end{aligned} \quad (5.3)$$

In this system, the state $\mathbf{x}(t, s)$ at each time $t \geq 0$ must be at least second order differentiable with respect to s , so we define the associated fundamental state as $\mathbf{x}_f(t, s) = \partial_s^2 \mathbf{x}(t, s)$. We implement this system in PIETOOLS using the command line parser as follows:

```
>> pvar s t
>> x = state('pde');
>> PDE_dyn = diff(x,t) == diff(x,s,2);
>> PDE_BC1 = [subs(x,s,0) + int(x,s,[0,1]) == 0;
              subs(x,s,1) + int(x,s,[0,1]) == 0];
>> PDE = sys();
>> PDE = addequation(PDE, [PDE_dyn; PDE_BC1]);
```

Then, we can derive the associated PIE representation by simply calling

```
>> PIE = convert(PDE, 'pie')
PIE =
  pie_struct with properties:

    dim: 1;
   vars: [1x2 polynomial];
    dom: [0 1];

    T: [1x1 opvar];    Tw: [1x0 opvar];    Tu: [1x0 opvar];
    A: [1x1 opvar];    B1: [1x0 opvar];    B2: [1x0 opvar];
   C1: [0x1 opvar];    D11: [0x0 opvar];    D12: [0x0 opvar];
   C2: [0x1 opvar];    D21: [0x0 opvar];    D22: [0x0 opvar];
```

In this structure, the field `dim` corresponds to the spatial dimensionality of the system, with `dim=1` indicating that this is a 1D PIE. The fields `vars` and `dom` define the spatial variables in the PIE and their domain, with

```

>> PIE.vars
ans =
    [ s, theta]

>> PIE.dom
ans =
     0     1

```

indicating that \mathbf{s} is the primary variable, \mathbf{theta} the dummy variable, and both exist on the domain $s, \theta \in [0, 1]$. We note that the remaining fields in the PIE structure are all `opvar` objects, representing PI operators in 1D. Moreover, most of these operators are empty, being of dimension 1×0 , 0×1 or 0×0 . This is because the PDE (5.3) does not involve any inputs or outputs, and therefore its associated PIE has the simple structure

$$(\mathcal{T}\dot{\mathbf{x}}_f)(t, s) = (\mathcal{A}\mathbf{x}_f)(t, s),$$

where the operator \mathcal{T} maps the fundamental state \mathbf{x}_f back to the PDE state \mathbf{x} as

$$(\mathcal{T}\mathbf{x}_f)(t, s) = \mathbf{x}(t, s).$$

For the PDE (5.3), we know that $\mathbf{x}_f(t, s) = \partial_s \mathbf{x}(t, s)$. The associated operators \mathcal{T} and \mathcal{A} are represented by the `opvar` objects `T` and `A` in the PIE structure, for which we find that

```

>> T = PIE.T
T =
    [] | []
    -----
    [] | T.R

T.R =
    [0] | [s^2 - 0.25*s - theta] | [0.75*(s^2 - s)]
>> A = PIE.A
A =
    [] | []
    -----
    [] | A.R

A.R =
    [1] | [10] | [0]

```

We conclude that the PDE (5.3) is equivalently represented by the PIE

$$\int_0^s \underbrace{\left(s^2 - \frac{s}{4} - \theta\right)}_{\text{PIE.T.R.R1}} \dot{\mathbf{x}}_f(t, \theta) d\theta + \int_s^1 \underbrace{\frac{3}{4}(s^2 - s)}_{\text{PIE.T.R.R2}} \dot{\mathbf{x}}_f(t, \theta) d\theta = \underbrace{1}_{\text{PIE.A.R.R0}} \mathbf{x}_f(t, s).$$

5.3 Converting a DDE to a PIE

Just like PDEs, DDEs (and other delay-differential equations) can also be equivalently represented as PIEs. For example, consider the following DDE

$$\dot{x}(t) = \begin{bmatrix} -1.5 & 0 \\ 0.5 & -1 \end{bmatrix} x(t) + \int_{-1}^0 \begin{bmatrix} 3 & 2.25 \\ 0 & 0.5 \end{bmatrix} x(t+s) ds + \int_{-2}^0 \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} x(t+s) ds,$$

where $x(t) \in \mathbb{R}^2$ for $t \geq 0$. We declare this system as a structure `DDE` in `PIETOOLS` as


```

>> DDE.A0 = [-1.5, 0; 0.5, -1];
>> DDE.Adi{1} = [3, 2.25; 0, 0.5];      DDE.tau(1) = 1;
>> DDE.Adi{2} = [-1, 0; 0, -1];      DDE.tau(2) = 2;

```

We can then convert the DDE to a PIE by calling

```

>> PIE = convert_PIETOOLS_DDE(DDE, 'pie')
PIE =
  pie_struct with properties:

    dim: 1;
  vars: [1x2 polynomial];
  dom: [1x2 double];
    T: [6x6 opvar];      Tw: [6x0 opvar];      Tu: [6x0 opvar];
    A: [6x6 opvar];      B1: [6x0 opvar];      B2: [6x0 opvar];
    C1: [0x6 opvar];     D11: [0x0 opvar];     D12: [0x0 opvar];
    C2: [0x6 opvar];     D21: [0x0 opvar];     D22: [0x0 opvar];

```

In this structure, we note that $\text{dim}=1$, indicating that the PIE is 1D, even though the state $x(t) \in \mathbb{R}^2$ in the DDE is finite-dimensional. This is because, in order to incorporate the delayed signals, the state is augmented to $\mathbf{x}(t) = \begin{bmatrix} x(t) \\ \mathbf{x}_1(t) \\ \mathbf{x}_2(t) \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^2 \\ L_2^2[-1,0] \\ L_2^2[-1,0] \end{bmatrix}$, where

$$\mathbf{x}_1(t, s) = x(t + \tau_1 s) = x(t + s), \quad \text{and}, \quad \mathbf{x}_2(t, s) = x(t + \tau_2 s) = x(t - 2s)$$

for $s \in [-1, 0]$. Here, the artificial states $\mathbf{x}_1(t)$ and $\mathbf{x}_2(t)$ will have to satisfy

$$\begin{aligned} \dot{\mathbf{x}}_1(t, s) &= \dot{x}(t + s) = \partial_r x(r) = \partial_s x(t + s) = \partial_s \mathbf{x}_1(t, s), \\ \dot{\mathbf{x}}_2(t, s) &= \dot{x}(t + 2s) = \partial_r x(r) = \frac{1}{2} \partial_s x(t + 2s) = \frac{1}{2} \partial_s \mathbf{x}_2(t, s) \quad s \in [-1, 0] \end{aligned}$$

and we can equivalently represent the DDE as a PDE

$$\begin{aligned} \dot{x}(t) &= \begin{bmatrix} -1.5 & 0 \\ 0.5 & -1 \end{bmatrix} x(t) + \int_{-1}^0 \begin{bmatrix} 3 & 2.25 \\ 0 & 0.5 \end{bmatrix} \mathbf{x}_1(t, s) ds + \int_{-2}^0 \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \mathbf{x}_2(t, s) dt, \\ \dot{\mathbf{x}}_1(t, s) &= \partial_s \mathbf{x}_1(t, s) \\ \dot{\mathbf{x}}_2(t, s) &= \frac{1}{2} \partial_s \mathbf{x}_2(t, s) \end{aligned}$$

$$\text{with BCs} \quad \mathbf{x}_1(t, -1) = x(t), \quad \mathbf{x}_1(t, -2) = x(t).$$

In this system, \mathbf{x}_1 and \mathbf{x}_2 must be first-order differentiable with respect to s , suggesting that the fundamental state associated to this PDE is given by $\mathbf{x}_f(t) = \begin{bmatrix} x_{f,0}(t) \\ \mathbf{x}_{f,1}(t) \\ \mathbf{x}_{f,2}(t) \end{bmatrix} = \begin{bmatrix} x(t) \\ \partial_s \mathbf{x}_1(t) \\ \partial_s \mathbf{x}_2(t) \end{bmatrix}$ for $t \geq 0$.

The PIE structure derived from the DDE will describe the dynamics in terms of this fundamental state \mathbf{x}_f , where we note that, indeed, the objects **T** and **A** are of dimension 6×6 . In particular, we find that

```

>> T = PIE.T
T =
    [1,0] | [0,0,0,0]
    [0,1] | [0,0,0,0]
-----
    [1,0] | T.R

```

```

      [0,1] |
      [1,0] |
      [0,1] |

T.R =
      [0,0,0,0] | [0,0,0,0] | [-1,0,0,0]
      [0,0,0,0] | [0,0,0,0] | [0,-1,0,0]
      [0,0,0,0] | [0,0,0,0] | [0,0,-1,0]
      [0,0,0,0] | [0,0,0,0] | [0,0,0,-1]

```

where T.P is simply a 2×2 identity operator, as the first two state variables of the augmented state \mathbf{x} and the fundamental state \mathbf{x}_f are both identical, and equal to the finite-dimensional state $x(t)$. More generally, we find that the augmented state can be retrieved from the associated fundamental state as

$$\mathbf{x}(t, s) = (\mathcal{T}\mathbf{x}_f)(t, s) = \begin{bmatrix} I_{2 \times 2} & \int_{-1}^0 d\theta \begin{bmatrix} 0_{2 \times 2} & 0_{2 \times 2} \end{bmatrix} \\ \begin{bmatrix} I_{2 \times 2} \\ I_{2 \times 2} \end{bmatrix} & \int_s^0 d\theta \begin{bmatrix} -I_{2 \times 2} & 0_{2 \times 2} \\ 0_{2 \times 2} & -I_{2 \times 2} \end{bmatrix} \end{bmatrix} \begin{bmatrix} x_{f,0}(t) \\ \mathbf{x}_{f,1}(t, \theta) \\ \mathbf{x}_{f,2}(t, \theta) \end{bmatrix} = \begin{bmatrix} x_{f,0}(t) \\ x_{f,0}(t) - \int_s^0 \mathbf{x}_{f,1}(t, \theta) d\theta \\ x_{f,0}(t) - \int_s^0 \mathbf{x}_{f,2}(t, \theta) d\theta \end{bmatrix}$$

Then, studying the value of the object A

```

>> A = PIE.A
A =
      [-0.5,2.25] | [-3*s-3,-2.25*s-2.25,2*s+2,0]
      [0.5,-2.5] | [0,-0.5*s-0.5,0,2*s+2]
      -----
      [0,0] | A.R
      [0,0] |
      [0,0] |
      [0,0] |

A.R =
      [1,0,0,0] | [0,0,0,0] | [0,0,0,0]
      [0,1,0,0] | [0,0,0,0] | [0,0,0,0]
      [0,0,0.5000,0] | [0,0,0,0] | [0,0,0,0]
      [0,0,0,0.5000] | [0,0,0,0] | [0,0,0,0]

```

we find that the DDE can be equivalently represented by the PIE

$$\begin{aligned} (\mathcal{T}\dot{\mathbf{x}}_f)(t, s) &= \begin{bmatrix} \dot{x}_{f,0}(t) \\ \dot{x}_{f,0}(t) - \int_s^0 \dot{\mathbf{x}}_{f,1}(t, \theta) d\theta \\ \dot{x}_{f,0}(t) - \int_s^0 \dot{\mathbf{x}}_{f,2}(t, \theta) d\theta \end{bmatrix} \\ &= \begin{bmatrix} \begin{bmatrix} -0.5 & 2.25 \\ 0.5 & -2.5 \end{bmatrix} x_{f,0}(t) + \int_{-1}^0 (s+1) \left(\begin{bmatrix} -3 & -2.25 \\ 0 & -0.5 \end{bmatrix} \mathbf{x}_{f,1}(t, s) + 2\mathbf{x}_{f,2}(t, s) \right) ds \\ \mathbf{x}_{f,1}(t, s) \\ \frac{1}{2}\mathbf{x}_{f,2}(t, s) \end{bmatrix} = (\mathcal{A}\mathbf{x}_f)(t, s) \end{aligned}$$

5.4 Converting a System with Inputs and Outputs to a PIE

In addition to autonomous differential systems, systems with inputs and outputs can also be represented as PIEs. In this case, the PIE takes a more general form

$$\begin{aligned}\mathcal{T}_u\dot{u}(t) + \mathcal{T}_w\dot{w} + \mathcal{T}\dot{\mathbf{x}}_f(t) &= \mathbf{A}\mathbf{x}_f(t) + \mathbf{B}_1w(t) + \mathbf{B}_2u(t), \\ z(t) &= \mathcal{C}_1\mathbf{x}_f(t) + \mathcal{D}_{11}w(t) + \mathcal{D}_{12}u(t), \\ y(t) &= \mathcal{C}_2\mathbf{x}_f(t) + \mathcal{D}_{21}w(t) + \mathcal{D}_{22}u(t),\end{aligned}\tag{5.4}$$

where w denotes the exogenous inputs, u the actuator inputs, z the regulated outputs, and y the observed outputs. Here, the operator \mathcal{T}_u , \mathcal{T}_w and \mathcal{T} define the map from the fundamental state \mathbf{x}_f back to the PDE state as

$$\mathbf{x}(t) = \mathcal{T}_u u(t) + \mathcal{T}_w w(t) + \mathcal{T} \mathbf{x}_f(t),$$

where the operators \mathcal{T}_u and \mathcal{T}_w will be nonzero only if the inputs u and w contribute to the boundary conditions enforced upon the PDE state \mathbf{x} . As such, the temporal derivatives \dot{u} and \dot{w} will also contribute to the PIE only if these inputs appear in the boundary conditions, which may be the case when performing e.g. boundary or delayed control.

In PIETOOLS, systems with inputs and outputs can be converted to PIEs in the same manner as autonomous systems. For example, consider a 1D heat equation with distributed disturbance w , and boundary control u , where we can observe the state at the upper boundary, and we wish to regulate the integral of the state over the entire domain:

$$\begin{aligned}\dot{\mathbf{x}}(t, s) &= \frac{1}{2}\partial_s^2\mathbf{x}(t, s) + s(2-s)w(t), & s \in [0, 1] \\ z(t) &= \int_0^1 \mathbf{x}(t, s)ds, \\ y(t) &= \mathbf{x}(t, 1), \\ \text{with BCs } \mathbf{x}(t, 0) &= u(t), & \partial_s\mathbf{x}(t, 1) = 0,\end{aligned}\tag{5.5}$$

This system too can be represented as a partial integral equation, describing the dynamics of the fundamental state $\mathbf{x}_f = \partial_s^2\mathbf{x}$. To arrive at this PIE representation, we once more implement the PDE using the command line parser as

```
>> pvar s t
>> x = state('pde');
>> w = state('in');    u = state('in');
>> z = state('out');    y = state('out');
>> PDE = sys();
>> PDE_dyn = diff(x,t) == 0.5*diff(x,s,2) + s*(2-s)*w;
>> PDE_z    = z == int(x,s,[0,1]);
>> PDE_y    = y == subs(x,s,1);
>> PDE_BCs = [subs(x,s,0) == u; subs(diff(x,s),s,1) == 0];
>> PDE = addequation(PDE,[PDE_dyn; PDE_z; PDE_y; PDE_BCs]);
>> PDE = setControl(PDE,u);          PDE = setObserve(PDE,y);
```

where we use the commands `setControl` and `setObserve` to indicate that u and y are a controlled input and observed output respectively. Then, we can convert this system to an equivalent PIE as before, finding a structure

```
>> PIE = convert(PDE, 'pie')
PIE =
  pie_struct with properties:

    dim: 1;
    vars: [1x2 polynomial];
    dom: [0 1];

    T: [1x1 opvar];    Tw: [1x1 opvar];    Tu: [1x1 opvar];
    A: [1x1 opvar];    B1: [1x1 opvar];    B2: [1x1 opvar];
    C1: [1x1 opvar];    D11: [1x1 opvar];    D12: [1x1 opvar];
    C2: [1x1 opvar];    D21: [1x1 opvar];    D22: [1x1 opvar];
```

In this structure, the fields `T` through `D22` describe the PI operators \mathcal{T} through \mathcal{D}_{22} in the PIE (5.4). Here, since the exogenous input w does not contribute to the boundary conditions, it also will not contribute to the map $\mathbf{x} = \mathcal{T}_u u + \mathcal{T}_w w + \mathcal{T} \mathbf{x}_f$ from the fundamental state \mathbf{x}_f to the PDE state \mathbf{x} . As such, we also find that the associated `opvar` object `Tw` has all parameters equal to zero, whereas `Tu` and `T` are distinctly nonzero

```
>> Tw = PIE.Tw
Tw =
  [] | []
  -----
  [0] | Tw.R

>> Tu = PIE.Tu
Tu =
  [] | []
  -----
  [1] | Tu.R

>> T = PIE.T
T =
  [] | []
  -----
  [] | T.R

T.R =
  [0] | [-theta] | [-s]
```

Note here that only the parameter `Tu.Q2` is non-empty for `Tu`, and only `T.R` is nonempty for `T`, as \mathcal{T}_u maps a finite-dimensional state $u \in \mathbb{R}$ to an infinite dimensional state $\mathbf{x} \in L_2[0, 1]$, whilst \mathcal{T} maps an infinite-dimensional state $\mathbf{x}_f \in L_2[0, 1]$ to an infinite-dimensional state $\mathbf{x} \in L_2[0, 1]$. Studying the values of `Tu` and `T`, we find that we can retrieve the PDE state as

$$\mathbf{x} = \mathcal{T}_u u + \mathcal{T} \mathbf{x}_f = u - \int_0^s \theta \mathbf{x}_f(\theta) d\theta - \int_s^1 s \mathbf{x}_f(\theta) d\theta = u - \int_0^s \theta \partial_\theta^2 \mathbf{x}(\theta) d\theta - \int_s^1 s \partial_\theta^2 \mathbf{x}(\theta) d\theta$$

Next, we look at the operators \mathcal{A} , \mathcal{B}_1 and \mathcal{B}_2 . Here, \mathcal{B}_2 will be zero, as the input u does not appear in the equation for $\dot{\mathbf{x}}$, nor does the value of $\mathbf{x}_f = \partial_s^2 \mathbf{x}$ depend on u . For the remaining operators, we find that they are equal to

```
>> A = PIE.A
A =
      [] | []
      -----
      [] | T.R

A.R =
      [0.5] | [0] | [0]

>> B1 = PIE.B1
B1 =
                [] | []
      -----
      [-s^2+2*s] | B1.R
```

suggesting that the fundamental state \mathbf{x}_f must satisfy

$$\dot{u}(t) - \int_0^s \theta \dot{\mathbf{x}}_f(t, \theta) d\theta - \int_s^1 s \dot{\mathbf{x}}_f(t, \theta) d\theta = \frac{1}{2} \mathbf{x}_f(t) + [-s^2 + 2s]w(t), \quad s \in [0, 1]$$

This leaves only the output equations. Here, since there is no feed-through from w into z or y , the operators \mathcal{D}_{11} and \mathcal{D}_{21} will both be zero. However, despite the actuator input u not appearing in the PDE equations for z and y , the contribution of u to the BCs means that the value of the PDE state $\mathbf{x} = \mathcal{T} \mathbf{x}_f + \mathcal{T}_u u$ also depends on the value of u , and therefore \mathcal{D}_{12} and \mathcal{D}_{22} are nonzero. In particular, we find that

```
>> C1 = PIE.C1
C1 =
      [] | [0.5*s^2-s]
      -----
      [] | C1.R

>> D12 = PIE.D12
D12 =
      [1] | []
      -----
      [] | D12.R

>> D22 = PIE.D22
D22 =
      [1] | []
      -----
      [] | D22.R

>> C2 = PIE.C2
C2 =
      [] | [-s]
      -----
```

Here, only the parameters Q1 of C1 and C2 are non-empty, as the operators \mathcal{C}_1 and \mathcal{C}_2 map infinite-dimensional states $\mathbf{x}_f \in L_2[0, 1]$ to finite-dimensional outputs $z, y \in \mathbb{R}$. Similarly, only the parameters P of D12 and D22 are non-empty, as \mathcal{D}_{12} and \mathcal{D}_{22} map the finite-dimensional input $u \in \mathbb{R}$ to finite-dimensional outputs $z, y \in \mathbb{R}$. Combining with the earlier results, we find that the PDE (5.5) may be equivalently represented by the PIE

$$\begin{aligned} \dot{u}(t) - \int_0^s \theta \dot{\mathbf{x}}_f(t, \theta) d\theta - \int_s^1 s \dot{\mathbf{x}}_f(t, \theta) d\theta &= \frac{1}{2} \mathbf{x}_f(t, s) + s(2-s)w(t), & s \in [0, 1] \\ z(t) &= \int_0^1 \left(\frac{1}{2} s^2 - s \right) \mathbf{x}_f(t, s) ds + u(t), \\ y(t) &= - \int_0^1 s \mathbf{x}_f(t, s) ds + u(t), & \text{where } \mathbf{x}_f(t, s) = \partial_s^2 \mathbf{x}(t, s). \end{aligned}$$

Chapter 6

Simulating PDE, DDE and PIE Solutions with PIESIM

In the previous chapter, we showed how an equivalent PIE representation of any well-posed, linear PDE or DDE can be computed in PIETOOLS. This partial integral representation offers several benefits to the partial differential and delay-differential formats, including analysis of e.g. stability properties using convex optimization as we show in Chapter 7. In this chapter, we demonstrate another benefit of the PIE representation, namely the relative ease of numerical simulation of solutions in this format. In particular, in Section 6.1, we provide some theoretical background on how solutions to PIEs can be numerically simulated. In Section 6.2, we then show how the PIETOOLS function PIESIM can be used to simulate solutions to PDE, DDE and PIE systems. Finally, in Section 6.3, we show how solutions computed with PIESIM can be plotted, and several examples of how PIESIM can be used.

6.1 Simulating Solutions in the PIE Representation

Consider a PIE system of the form

$$\begin{aligned} \mathcal{T}\dot{\mathbf{v}}(t) + \mathcal{T}_w\dot{w}(t) + \mathcal{T}_u\dot{u}(t) &= \mathcal{A}\mathbf{v}(t) + \mathcal{B}_1w(t) + \mathcal{B}_2u(t), & \mathbf{v}(0, s) &= \mathbf{v}_0(s), & s \in [-1, 1], & t \geq 0, \\ z(t) &= \mathcal{C}\mathbf{v}(t) + \mathcal{D}_{11}w(t) + \mathcal{D}_{12}u(t), & & & & (6.1) \end{aligned}$$

where $\mathbf{v}(t) \in L_2^n[-1, 1]$ for $t \geq 0$. To simulate solutions to this system, the state $\mathbf{v}(t)$ at each time $t \geq 0$ is projected on to a finite dimensional vector space spanned by Chebyshev polynomials up to order N . The resulting projected solution, $\mathbf{v}_N \approx \mathbf{v}$, is of the form

$$\mathbf{v}_N(t, s) = \sum_{i=0}^N \alpha_i(t) P_i(s), \quad s \in [-1, 1], \quad t \geq 0, \quad (6.2)$$

where P_i is a Chebyshev polynomial of degree i .

By substituting \mathbf{v}_N in the form of (6.2) into the PIE and taking an inner product with each basis Chebyshev polynomial function, we obtain an ODE approximation of the PIE as

$$T\dot{\alpha}(t) + T_w\dot{w}(t) + T_u\dot{u}(t) = A\alpha(t) + B_1w(t) + B_2u(t), \quad \alpha = [\alpha_0 \quad \alpha_1 \quad \cdots \quad \alpha_N]^T, \quad (6.3)$$

where T, T_w, T_u, A and B_i are matrices obtained by spectral discretization of the PI operators \mathcal{T} through \mathcal{B}_i . Given an initial condition $\alpha(0) = \alpha_I \in \mathbb{R}^{N+1}$, the resulting ODE (6.3) can then be solved analytically if T is invertible, or numerically using any time-stepping scheme. Here, an initial value for α can be obtained from the initial conditions for \mathbf{v} in the PIE (6.1) using the identity

$$\alpha_k(0) = \frac{2}{N+1} \sum_{i=0}^{N+1} \mathbf{v}(0, s_i) P_K(s_i), \quad s_i = \cos \frac{i\pi}{N}, \quad k = 0, 1, \dots, N.$$

Given the ODE solution $\alpha(t)$ to 6.3, an approximate solution \mathbf{v}_N to the PIE can be reconstructed using the equation (6.2). If the PIE corresponds to a PDE or DDE system, the PIE solution \mathbf{v}_N can then be used to find an approximate solution \mathbf{x}_N of the original PDE or DDE using the identity

$$\mathbf{x}(t, s) = \mathcal{T}\mathbf{v}(t, s) + \mathcal{T}_w w(t) + \mathcal{T}_u u(t) \approx T\mathbf{v}_N(t, s) + T_w w(t) + T_u u(t). \quad (6.4)$$

6.2 PIE Simulation Using PIETOOLS

PIETOOLS 2022 supports simulation of PIEs obtained by transforming a DDE/DDF or a PDE with spatial derivatives up to order 2. Users can run PIE simulations either by using the script `solver_PIESIM.m` located in the `PIESIM` folder or by directly calling the function `PIESIM()`. A guide to use both the methods will be presented in the following subsections.

6.2.1 Following the `solver_PIESIM` Template Script

The script file is organized as a template for the user to demonstrate a typical workflow involved in simulation of the PIEs. The simulation procedure can be broken down into the following steps.

1. Rescale the spatial dimension in case of PDEs (the delay interval in case of DDEs) to the interval $[-1, 1]$, or alternatively use `rescalePIE()` function after conversion
2. (Mandatory) Define a PDE/DDE model
 - Alternatively, the above step can be skipped if the PIE is already known, however, that feature is restricted to executive function file
3. Convert PDE/DDE to a PIE (Use converter functions)
4. (Optional) Define all the simulation settings listed below under `opts` structure
 - Order of approximation `N`
 - Time of simulation `tf`
 - Time integration scheme `intScheme`
 - Order of time integration scheme `Norder` (only if `intScheme=1`)
 - System type `type`

- Time step `dt`
 - Flag to turn plotting on or off `plot`
5. (Optional) Define all the system inputs listed below under `uinput` structure
 - Initial condition `ic.ODE` and `ic.ODE` (or `ic.DDE` for DDE model, `ic.PDE` for PIE model) as MATLAB symbolic expression in `sx` (space) and `st`
 - Disturbance (MATLAB symbolic expression in time `st`) `w`
 - Control input (MATLAB symbolic expression in time `st`) `u`
 - Flag for comparing with exact solution `ifexact`
 - Exact solution as a MATLAB symbolic expression in time `st` and space `sx` under `exact`
 6. Call `PIESIM(model,opts,uinput)` with the above inputs
 7. Reconstruct PDE/DDE solution (performed inside `PIESIM()`)

Warning

`uinput.ic.PDE` is used for both PDE and PIE inputs to `PIESIM()` function, however, when a **PDE** is passed `uinput.ic.PDE` stores initial conditions for the PDE, whereas when a **PIE** is passed `uinput.ic.PDE` stores initial conditions for the PIE!

6.2.2 Using the PIESIM Function

By using the function, the user can directly employ simulation results in a other scripts. While the systems definitions for PDE/DDE/PIE and `uinput`, and `opts` have the same format. This function can be called using the syntax

```
| >> solution = PIESIM(system, opts, uinput, n_pde);
```

where `system` is a PDE, DDE, or PIE structure whereas `opts` is the simulation options structure and `uinput` is a structure as described in the previous subsection both of which are optional for PDE/DDE systems. Notice, the additional input `n_pde` that is required only if the `system` defined is of the type 'PIE'. In case a PIE is directly passed, the user has to provide order of differentiability of the original PDE/DDE states that generated the PIE form as the fourth argument. Furthermore, the information in the `uinput`, such as IC and input, must now correspond to the PIE (and NOT the original system). The syntax to simulate a PIE directly using executive function is given by

```
| >> solution = PIESIM(PIE, opts, uinput, n_pde);
```

where `n_pde` is a vector describing the number of continuous, differentiable and twice differentiable states in the original PDE/DDE, in the same order. For example, `n_pde = [n0,n1,n2]` implies the original PDE/DDE has `n0` continuous states, `n1` differentiable states, and `n2` twice differentiable states.

This function returns an output structure `solution` with the fields

- `tf` - scalar - actual final time of the solution
- `final.pde` - array of size $(N + 1) \times n_s$, $n_s = n_0 + n_1 + n_2$ - PDE (distributed state) solution at a final time
- `final.ode` - array of size n_x - ODE solution at a final time
- `final.observed` - array of size n_y - final value of observed outputs
- `final.regulated` - array of size n_z - final value of regulated outputs
- `timedep.dtime` - array of size $1 \times N_{steps}$ - array of temporal stamps (discrete time values) of the time-dependent solution (only if `intScheme=1`)
- `timedep.pde` - array of size $(N + 1) \times n_s \times N_{steps}$ - time-dependent solution of n_s PDE (only if `intScheme=1`) (distributed) states of the primary PDE system
- `timedep.ode` - array of size $n_x \times N_{steps}$ - time-dependent solution of n_x ODE states, where $N_{steps} = tf/dt$ (only if `intScheme=1`)
- `timedep.observed` - array of size $n_y \times N_{steps}$ - time-dependent value of observed outputs (only if `intScheme=1`)
- `timedep.regulated` - array of size $n_z \times N_{steps}$ - time-dependent value of regulated outputs (only if `intScheme=1`)

Warning

In contrast to `uinput.ic.pde`, irrespective of the input, the `solution.timedep.pde` always stores $\mathcal{T}\mathbf{v} + \mathcal{T}_w w + \mathcal{T}_u u$ as the value as described in Equation 6.4.

6.3 Plotting the solution

Simulation of PIEs, either by using solver file or directly using the executive file, generates figures that plot time-varying ODE states (from 0 to final simulation time) for each ODE state. Further, a plot showing the spatial distribution (only at final simulation time) is generated for all distributed states in the PIE. Note, that plots correspond to the solution of the **original PDE/DDE** and not the PIE solution. In general, given a PIE of the form (6.1), the solution which is plotted is (6.4). However, the value of time-varying distributed state at each simulation time step is stored under the solution output which is given by the executive file. The user can use this output to generate further plots to calculate outputs z as defined in (6.1).

6.3.1 PIESIM Demonstration A: PDE example

In this section, we will demonstrate the standard process involved in simulation of PIEs using an example from the `examples_pde_library_PIESIM` file.

Code Block 5

First, we load an example from the library file and then edit simulation parameters directly by accessing the corresponding properties. The full code for the simulation demonstrated in this section is given below.

```
>> syms sx st;
>> init_option=1;
>> [PDE,uinput]=examples_pde_library_PIESIM(4);
>> uinput.exact(1) = -2*sx*st-sx^2;
>> uinput.ifexact=true;
>> uinput.w(1) = -4*st-4;
>> uinput.ic.PDE=-sx^2;
>> opts.plot='yes';
>> opts.N=8;
>> opts.tf=0.1;
>> opts.intScheme=1;
>> opts.Norder = 2;
>> opts.dt=1e-3;
>> solution = PIESIM(PDE,opts,uinput);
```

Next, we will explain each line used in the above code.

First, to choose an example from the examples library, we set the library flag to one. Then, an example can be selected by specifying the example number (between 1 and 34) to load the example.

```
>> init_option=1;
>> [PDE,uinput]=examples_pde_library_PIESIM(4);
```

In this demonstration, we choose the example

$$\begin{aligned}\dot{\mathbf{x}}(t, s) &= s\partial_s^2\mathbf{x}(t, s), & s \in [0, 2], t \geq 0 \\ \mathbf{x}(t, 0) &= 0, & \mathbf{x}(t, 2) = w_1(t), & \mathbf{x}(0, s) = -s^2.\end{aligned}$$

where $w_1(t) = -4t - 4$. For this PDE, the exact solution is known and is given by the expression $\mathbf{x}(t, s) = -2st - s^2$ which can be specified under `uinput` structure for verification as shown below.

```
>> uinput.exact(1) = -2*sx*st-sx^2;
>> uinput.ifexact=true;
```

Likewise, other input parameters such as, initial conditions and inputs at the boundary are specified as

```
>> uinput.w(1) = -4*st-4;
>> uinput.ic.PDE=-sx^2;
```

where `sx`, `st` are MATLAB symbolic objects. However, the example automatically defines the `uinput` structure and the above expressions are provided for demonstration only and not necessary when using a PDE from example library. Once the PDE and system inputs are defined, we have to specify simulation parameters under `opts` structure. First, we turn on the plotting by specifying the plotting flag as show below.

```

>> opts.plot='yes';
>> opts.N=8;
>> opts.tf=0.1;
>> opts.intScheme=1;
>> opts.Norder = 2;
>> opts.dt=1e-3;

```

We specify the order of discretization (order of chebyshev polynomials to be used in approximation of PDE solution N) and time of simulation. Then, we select a time-integration scheme (backward difference scheme is used in this demonstration, however, one can chose symbolic integration by setting `intScheme=2`). In solver file, time step and order of truncation are automatically chosen for backward difference scheme as shown below, however, the user can modify these parameters as needed.

Now that we have defined all necessary parameters we can run the simulation using the command for the PDE example

```
| solution = PIESIM(PDE,opts,uinput);
```

which produces the plot Fig. 6.1, where we see that simulation result in dots whereas the analytical solution is plotted using the solid line. If the analytical solution is not passed, then only the dots are plotted.

6.3.2 PIESIM Demonstration B: DDE example

Simulation of DDEs can be performed using the same steps as the simulation of PDEs, however, there is a main difference which is the first argument to `PIESIM()` function a DDE model. Consider a DDE system,

$$\begin{aligned} \dot{x}(t) &= \begin{bmatrix} -1 & 2 \\ 0 & 1 \end{bmatrix} x(t) + \begin{bmatrix} 0.6 & -0.4 \\ 0 & 0 \end{bmatrix} x(t - \tau_a) + \begin{bmatrix} 0 & 0 \\ 0 & -0.5 \end{bmatrix} x(t - \tau_b) + \begin{bmatrix} 1 \\ 1 \end{bmatrix} w(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t) \\ y(t) &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 0 \\ 0.1 \end{bmatrix} u(t) \end{aligned}$$

where $\tau_a = 1$ and $\tau_b = 2$. This system can be stored in a DDE structure as shown below.

```

>> DDE.A0=[-1 2;0 1]; DDE.Ai1=[.6 -.4; 0 0];
>> DDE.Ai2=[0 0; 0 -.5]; DDE.B1=[1;1];
>> DDE.B2=[0;1]; DDE.C1=[1 0;0 1;0 0];
>> DDE.D12=[0;0;.1]; DDE.tau=[1,2];
>> solution = PIESIM(DDE,opts,uinput);

```

We can use the same `opts` and `uinput` from previous section (except initial conditions, which we will leave undefined and default to zero). Note `uinput.u` must be set to zero since we are simulating without a controller.

```

>> uinput.exact(1) = -2*sx*st-sx^2;
>> uinput.ifexact=true;
>> uinput.w(1) = -4*st-4;
>> uinput.u(1) =0;
>> opts.plot='yes';

```

```

>> opts.N=8;
>> opts.tf=0.1;
>> opts.intScheme=1;
>> opts.Norder = 2;
>> opts.dt=1e-3;

```

Then, use the command to simulate the system which gives us a default plot. However, using the code below, we can change the plot features by accessing the data in `solution` structure.

```

>> plot(solution.timedep.dtime,solution.timedep.ode,'--o',...
'MarkerIndices',1:50:length(solution.timedep.dtime));
>> ax = gca;
>> set(ax,'XTick',solution.timedep.dtime(1:150:end));
>> lgd1 = legend('x1','x2','Interpreter','latex'); lgd1.FontSize = 10.5;
>> lgd1.Location = 'northeast';
>> title('Time evolution of the Delay system states, x, ...
without state feedback control');
>> ylabel('x1(t), x2(t)','Interpreter','latex','FontSize',15);
>> xlabel('t','FontSize',15,'Interpreter','latex');

```

All the code presented here can be found in the demo file packaged with PIETOOLS named “PIE_simulation_DEMO”.

6.3.3 PIESIM Demonstration C: PIE example

In the above DDE example, the solution is clearly unstable. For this system, we can design a stabilizing controller for the PIE form and then simulate the solution again to see the behaviour of the control system under the action of the controller. However, conversion of a PIE back to DDE/PDE format is often tricky. Fortunately, a PIE need not be converted back to DDE or PDE format to simulate its solutions, because `PIESIM()` can be used to simulate a system in PIE form directly. For example, for the given DDE,

$$\dot{x}(t) = \begin{bmatrix} -1 & 2; 0 & 1 \end{bmatrix} x(t) + \begin{bmatrix} 0.6 & -0.4 \\ 0 & 0 \end{bmatrix} x(t-1) + \begin{bmatrix} 0 & 0 \\ 0 & -0.5 \end{bmatrix} x(t-2) + \begin{bmatrix} 1 \\ 1 \end{bmatrix} w(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 0 \\ 0.1 \end{bmatrix} u(t)$$

we can find the controller by using the following code.

```

>> DDE.A0=[-1 2;0 1]; DDE.Ai1=[.6 -.4; 0 0];
>> DDE.Ai2=[0 0; 0 -.5]; DDE.B1=[1;1];
>> DDE.B2=[0;1]; DDE.C1=[1 0;0 1;0 0];
>> DDE.D12=[0;0;.1]; DDE.tau=[1,2];
>> DDE=initialize_PIETOOLS_DDE(DDE);
>> DDF=minimize_PIETOOLS_DDE2DDF(DDE);
>> PIE=convert_PIETOOLS_DDF(DDF);
>> [prog, K, gamma, P, Z] = PIETOOLS_Hinf_control(PIE);
>> PIE = closedLoopPIE(PIE,K);
>> ndiff = [0, PIE.T.dim(2,1)];

```

Then, we can define the `uinput` and `opt` to define simulation parameters for the PIE SIM. Using the same inputs and options as before, we have

```
>> uinput.exact(1) = -2*sx*st-sx^2;
>> uinput.ifexact=true;
>> uinput.w(1) = -4*st-4;
>> opts.plot='yes';
>> opts.N=8;
>> opts.tf=0.1;
>> opts.intScheme=1;
>> opts.Norder = 2;
>> opts.dt=1e-3; >> solution=PIESIM(PIE,opts,uinput,ndiff);
```

See the file ‘DDE_simulation_demo.m’ in the ‘GetStarted_DOCS_DEMOS’. Given the (optimal) controller gains K , we construct a closed loop PIE system using `closedLoopPIE(PIE,K)`. This closed system can then be simulated using the `PIESIM()` function. Finally, we plot the system behavior under the state feedback designed for the PIE using the following code.

```
>> plot(solution.timedep.dtime,solution.timedep.ode,'--o',...
'MarkerIndices',1:50:length(solution.timedep.dtime));
>> ax = gca;
>> set(ax,'XTick',solution.timedep.dtime(1:150:end));
>> lgd1 = legend('x1','x2','Interpreter','latex'); lgd1.FontSize = 10.5;
>> lgd1.Location = 'northeast';
>> title('Time evolution of the Delay system states, x,...
with state feedback control');
>> ylabel('x1(t), x2(t)','Interpreter','latex','FontSize',15);
>> xlabel('t','FontSize',15,'Interpreter','latex');
```

The main difference from PDE/DDE simulations to PIE simulations is the new input argument `ndiff` which describes how many states are differentiable. In case of DDEs, all states with delays are at least once differentiable along the delay variable and hence are all placed under `n1`. The other arguments such as `opts` and `uinput` are same as the inputs described in earlier sections. Behaviour of the DDE solution (for same initial conditions) simulated using a PIE is shown in Figure 6.3.

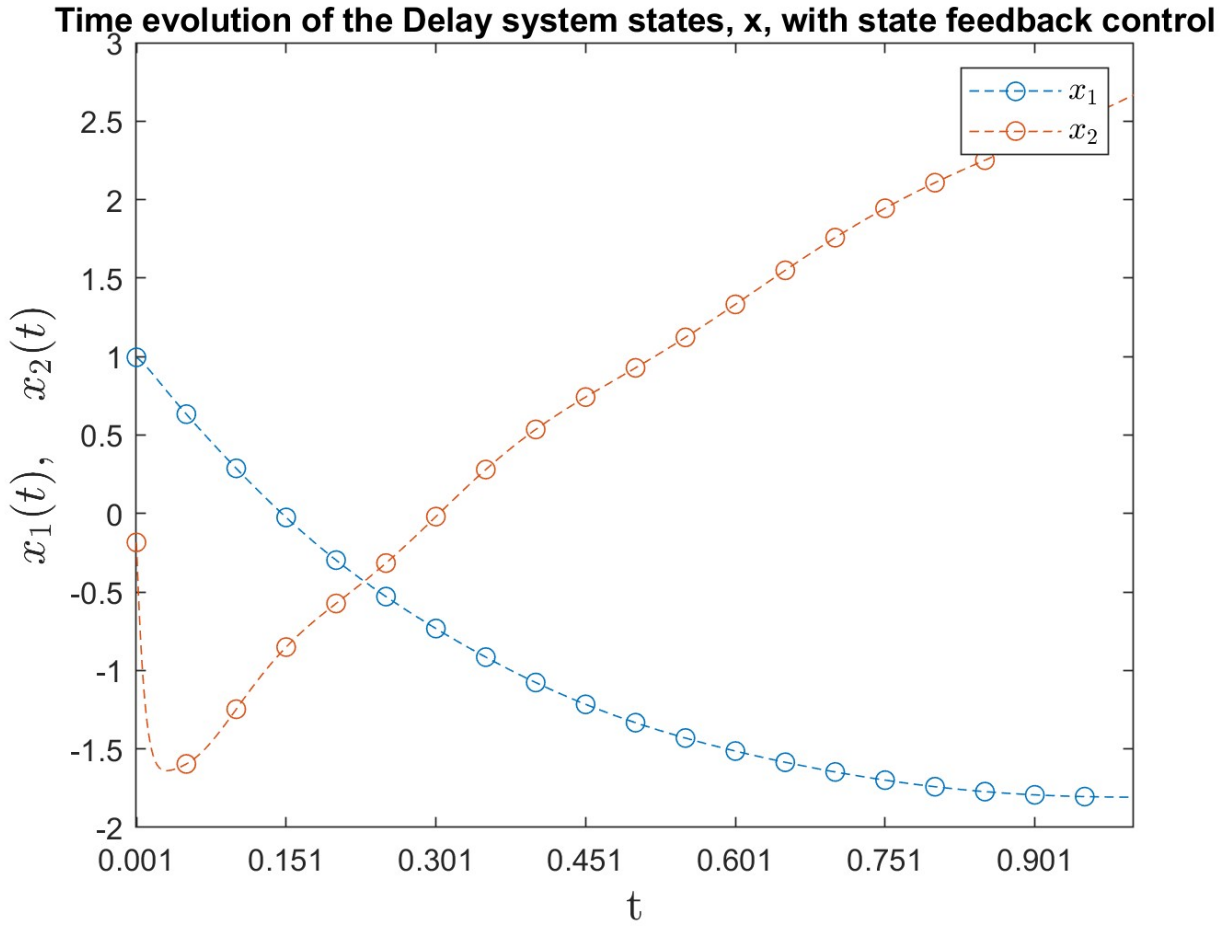


Figure 6.3: DDE solution $x_1(t)$ and $x_2(t)$ obtained by PIE simulation for the closed loop PIE of the previous DDE example is plotted against time t

All the code presented here can be found in the demo file packaged with PIETOOLS named “PIE_simulation_DEMO”.

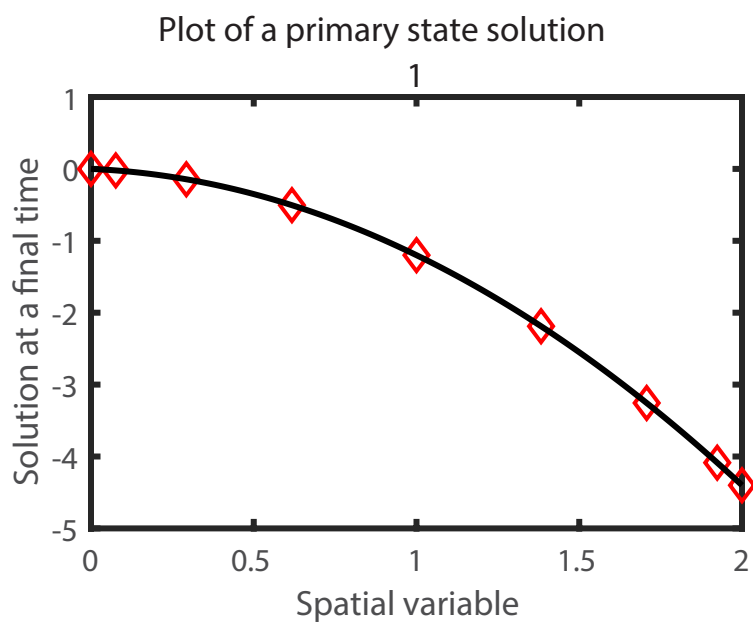


Figure 6.1: Default PIESIM plot: Final solution $\mathbf{x}(t, \cdot)$ at $t = 0.1s$ obtained by analytical expression (solid line) and by PIE simulation (dots) are plotted against space $[0, 2]$

Time evolution of the Delay system states, x , without state feedback control

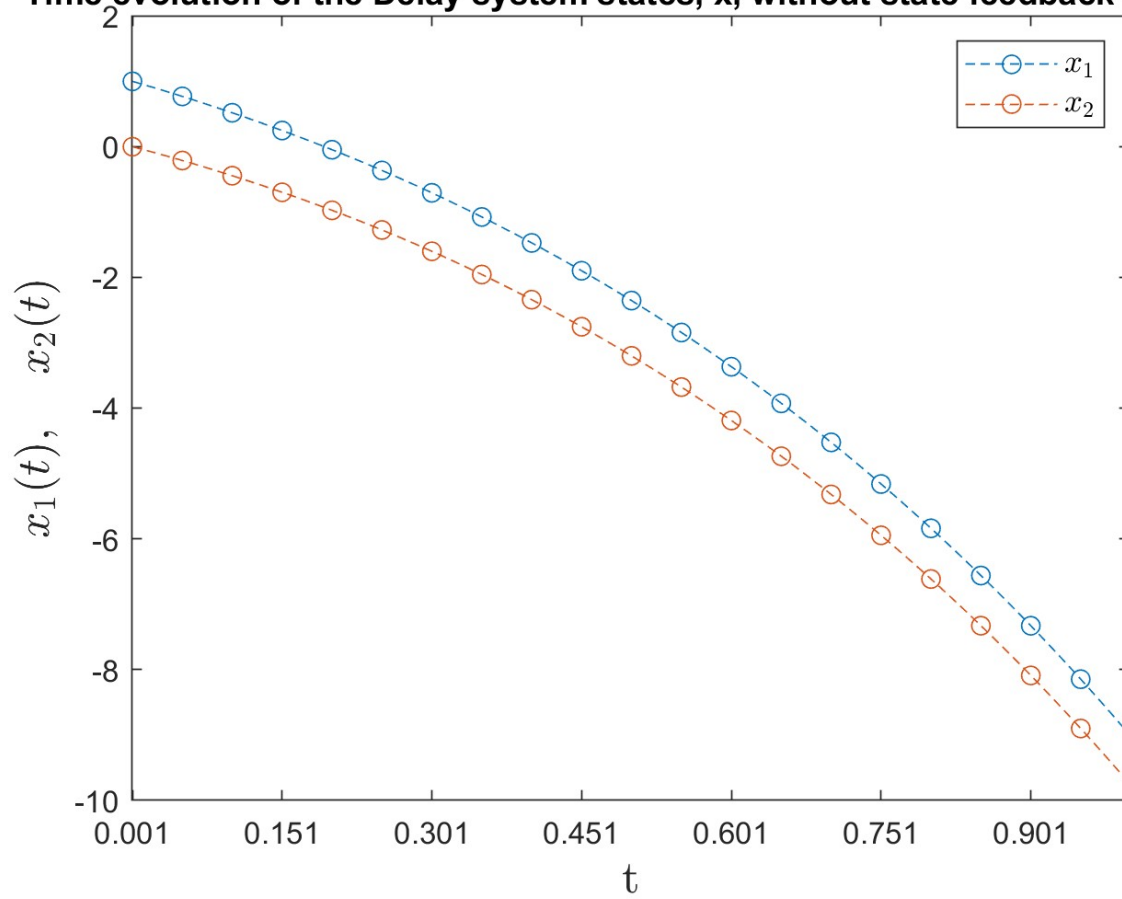


Figure 6.2: DDE solutions $x_1(t)$ and $x_2(t)$ obtained by by PIE simulation are plotted against time t

Chapter 7

Declaring and Solving Convex Optimization Programs on PI Operators

In Chapter 5 we showed that, using PIETOOLS, we can derive an equivalent PIE representation of any well-posed system of linear partial differential and delay-differential equations. This PIE representation is completely free of boundary conditions and continuity constraints that appear in any well-posed PDE, allowing analysis of PIEs to be performed without having to explicitly account for such additional constraints. In addition, PIEs are parameterized by PI operators, which can be added and multiplied, and for which concepts of e.g. positivity are well-defined. This allows us to impose positivity and negativity constraints on PI operators, referred to as linear PI inequalities (LPIs), to define convex optimization programs for testing properties (such as stability) of PIEs.

In this chapter, we show how these convex optimization problems can be implemented in PIETOOLS. In particular, in Sections 7.1 and 7.2, we show how an LPI optimization program can be initialized, and how (PI operator) decision variables can be added to this program structure. Next, in Section 7.3 we show how PI operator equality and inequality constraints can be specified, followed by how an objective function can be set for the program in Section 7.4. In Sections 7.5 and 7.6, we then show how the optimization program can be solved, and how the obtained solution can be extracted respectively. Finally, in Section 7.7, we show how pre-defined executive files can be used to solve standard LPI optimization programs for PIEs, and how properties in these optimization programs can be modified using the `settings` files.

We illustrate the use of each of the functions in this chapter with the following example. A demo file “`Hinf_optimal_estimator_DEMO`” declaring and solving the LPI in this example has also been included in PIETOOLS; see Section 11.5

Example

Consider the problem of designing an H_∞ -optimal estimator of the form

$$\begin{aligned} \mathcal{T}\dot{\hat{\mathbf{v}}}(t) &= \mathcal{A}\hat{\mathbf{v}}(t) + \mathcal{L}(y(t) - \hat{y}(t)), & \mathcal{T}\dot{\mathbf{v}}(t) &= \mathcal{A}\mathbf{v}(t) + \mathcal{B}_1w(t), \\ \hat{z}(t) &= \mathcal{C}_1\hat{\mathbf{v}}(t), & z(t) &= \mathcal{C}_1\mathbf{v}(t) + \mathcal{D}_{11}w(t), \\ \hat{y}(t) &= \mathcal{C}_2\hat{\mathbf{v}}(t), & y(t) &= \mathcal{C}_2\mathbf{v}(t) + \mathcal{D}_{21}w(t), \end{aligned} \quad \text{for a PIE,} \quad (7.1)$$

aiming to find an operator \mathcal{L} that minimize the gain $\frac{\|\hat{z}-z\|_{L_2}}{\|w\|_{L_2}}$. To construct such an operator, we solve the LPI,

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

$$\mathcal{P} \succ 0, \quad Q := \begin{bmatrix} -\gamma I & -\mathcal{D}_{11}^\top & -(\mathcal{P}\mathcal{B}_1 + \mathcal{Z}\mathcal{D}_{21})^* \mathcal{T} \\ (\cdot)^* & -\gamma I & \mathcal{C}_1 \\ (\cdot)^* & (\cdot)^* & (\mathcal{P}\mathcal{A} + \mathcal{Z}\mathcal{C}_2)^* \mathcal{T} + (\cdot)^* \end{bmatrix} \preceq 0 \quad (7.2)$$

so that, for any solution $(\gamma, \mathcal{P}, \mathcal{Z})$ to this problem, letting $\mathcal{L} := \mathcal{P}^{-1}\mathcal{Z}$, the estimation error will satisfy $\|z - \hat{z}\| \leq \gamma\|w\|$. For more details on this LPI, and additional examples of LPIs and their applications, we refer to Chapter 13.

We will solve the LPI (7.2) for the PIE associated to the PDE

$$\begin{aligned} \dot{\mathbf{x}}(t, s) &= \partial_s^2 \mathbf{x}(t, s) + 4\mathbf{x}(t, s) + w(t), & s \in [0, 1] \\ \text{with BCs} & \quad 0 = \mathbf{x}(t, 0) = \partial_s \mathbf{x}(t, 1), \\ \text{and outputs} & \quad z(t) = \int_0^1 \mathbf{x}(t, s) ds + w(t), \\ & \quad y(t) = \mathbf{x}(t, 1). \end{aligned} \quad (7.3)$$

We declare this PDE using the command line parser as

```
>> pvar s t
>> PDE = sys();
>> x = state('pde');    w = state('in');
>> y = state('out');    z = state('out');
>> eqs = [diff(x,t) == diff(x,s,2) + 4*x + w;
          z == int(x,s,[0,1]) + w;
          y == subs(x,s,1);
          subs(x,s,0) == 0;
          subs(diff(x,s),s,1) == 0];
>> PDE = addequation(PDE,eqs);
>> PDE = setObserve(PDE,y);
```

We convert the PDE to an equivalent PIE using `convert`, and extract the defining PI operators so that these can be used to declare the LPI (7.2)

```
>> PIE = convert(PDE,'pie');    PIE = PIE.params;
>> T = PIE.T;
>> A = PIE.A;    C1 = PIE.C1;    C2 = PIE.C2;
>> B1 = PIE.B1;    D11 = PIE.D11;    D21 = PIE.D21;
```

7.1 Initializing an Optimization Problem Structure

In PIETOOLS, optimization programs are stored as program structures `prog`. These structures keep track of the free variables in the optimization program, the decision variables in the optimization program, the constraints imposed upon these decision variables, and the objective function in terms of these decision variables. To initialize an optimization program structure,

call the function `sosprogram`, passing as primary argument a vector of free variables that appear in the problem, and possibly passing decision variables to appear in the program as second argument:

```
>> pvar s1 s2; % Declare free (polynomial) variables
>> dpvar d1 d2; % Initialize decision variables
>> prog = sosprogram([s1;s2], [d1;d2]); % Initialize the program structure
```

When initializing a program structure for an LPI, it is crucial that the free variables `poly_vars` include the spatial variables and dummy variables that appear in the relevant PI operators.

Example

For the LPI (7.2), the free variables that appear in the optimization program will be s and θ , the spatial variables that appear in the PI operators. We initialize the optimization program structure as

```
>> vars = PIE.vars(:)
vars =
    [ s]
    [ theta]

>> prog = sosprogram(vars)
prog =

    struct with fields:

        var: [1×1 struct]
        expr: [1×1 struct]
        extravar: [1×1 struct]
        objective: []
        solinfo: [1×1 struct]
        variable: {2×1 cell}
        varmat: [1×1 struct]
        decvariable: {}
```

This initialize an empty optimization program in the free variables `vars=[s;theta]`. The field `variable` will then contain the names of the free variables that appear in the optimization program,

```
>> prog.variable
ans =
    2×1 cell array

    {'s' }
    {'theta'}
```

matching the variables that appear in the PIE structure.

Note:

To represent LPI optimization programs, PIETOOLS utilizes the `sosprogram` optimization

program structure from SOSTOOLS 4.00 [6]. For additional options allowed by SOSTOOLS not mentioned here, we refer to the SOSTOOLS 4.00 manual.

7.2 Declaring Decision Variables

Having discussed how to initialize an optimization program structure `prog`, in this section, we show how decision variables can be added to the optimization program structure. For the purposes of implementing LPIs, we distinguish three types of decision variables: standard scalar decision variables (Subsection 7.2.1), positive semidefinite PI operator decision variables (Subsection 7.2.2), and indefinite PI operator decision variables (Subsection 7.2.3).

7.2.1 `sosdecvar`

The simplest decision variables in LPI programs are represented by scalar `dpvar` objects, and can be declared using `dpvar`. The function `sosdecvar` must then be used to add the decision variable to the optimization program structure:

```
>> dpvar d1; % Initialize a new decision variable
>> prog = sosdecvar(prog,d1); % Add the decision variable to the program
```

Example

In the LPI (7.2), γ is a scalar decision variables, that appears both in the constraints and the objective function. To declare this variable, we simply call

```
>> dpvar gam;
>> prog = sosdecvar(prog, gam)
prog =

  struct with fields:

      var: [1×1 struct]
      expr: [1×1 struct]
      extravar: [1×1 struct]
      objective: 0
      solinfo: [1×1 struct]
      vartable: {2×1 cell}
      varmat: [1×1 struct]
      decvartable: {'gam'}
```

first initializing the variable `gam` as a `dpvar` object representing γ , and subsequently declaring it as decision variables in the optimization program, adding it to the field `decvartable`.

7.2.2 `poslpivar`

In PIETOOLS, positive semidefinite PI operator decision variables $\mathcal{P} \succcurlyeq 0$ are parameterized by positive matrices $P \succcurlyeq 0$ as $\mathcal{P} = \mathcal{Z}_d^* T \mathcal{Z}_d \succeq 0$, where \mathcal{Z}_d is a PI operator parameterized by monomials of degree of at most d (see Theorem 11 in Appendix A). Such PI operators can be declared using the function `poslpivar`:

```
| >> [prog,P] = poslpivar(prog,n,I,d,opts);
```

This function takes three mandatory inputs.

- **prog**: An sosprogram structure to which to add the PI operator decision variable.
- **n**: A 2×1 vector $[\mathbf{n0};\mathbf{n1}]$ specifying the dimensions of $\mathcal{P} : \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \end{bmatrix}$ for a 1D operator, or a 4×1 vector $[\mathbf{n0};\mathbf{n1};\mathbf{n2};\mathbf{n3}]$ specifying the dimensions for a 2D operator $\mathcal{P} : \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \\ L_2^{n_2}[c,d] \\ L_2^{n_3}[[a,b] \times [c,d]] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \\ L_2^{n_2}[c,d] \\ L_2^{n_3}[[a,b] \times [c,d]] \end{bmatrix}$.
- **I**: a 1×2 array $[\mathbf{a},\mathbf{b}]$ specifying the interval $[a,b]$ of the variables in \mathbf{P} in 1D, or a 2×2 array $[\mathbf{a},\mathbf{b};\mathbf{c},\mathbf{d}]$ specifying the spatial domain $[a,b] \times [c,d]$ of the variables in \mathbf{P} in 2D.
- **d** (optional):
 - 1D: A cell structure of the form $\{\mathbf{a}, [\mathbf{b0},\mathbf{b1},\mathbf{b2}]\}$, specifying the degree \mathbf{a} of s in $Z_1(s)$, the degree $\mathbf{b0}$ of s in $Z_2(s,\theta)$, the degree $\mathbf{b1}$ of θ in $Z_2(s,\theta)$, and the degree $\mathbf{b2}$ of s and θ combined in $Z_2(s,\theta)$ (see Thm. 11).
 - 2D: A structure with fields $\mathbf{d}.\mathbf{dx},\mathbf{d}.\mathbf{dy},\mathbf{d}.\mathbf{d2}$, specifying degrees for operators along $x \in [a,b]$, along $y \in [c,d]$, and along both $(x,y) \in [a,b] \times [c,d]$; call `help poslpivar_2d` for more information.
- **opts** (optional): This is a structure with fields
 - **exclude**: 4×1 vector with 0 and 1 values. Excludes the block T_{ij} (see Thm. 11) if i -th value is 1. Binary 16×1 array in 2D; call `help poslpivar_2d` for more information.
 - **psatz**: Sets $g(s) = 1$ if set to 0, and $g(s) = (b-s)(s-a)$ if set to 1 in 1D. In 2D, sets $g(x,y) = (b-x)(x-a)(d-y)(y-c)$ if **psatz**=1.
 - **sep**: Binary scalar value, constrains $\mathbf{P}.\mathbf{R}.\mathbf{R1}=\mathbf{P}.\mathbf{R}.\mathbf{R2}$ if set to 1. In 2D, this field is a 6×1 array; call `help poslpivar_2d` for more information.

The output is a program structure **prog** with new decision variables and a **dopvar** class object **P** representing a positive semidefinite PI operator decision variable. The `poslpivar` function has other experimental features to impose sparsity constraints on the T matrix of Theorem 11, which should be used with caution. Use `help poslpivar` for more information.

Note that, to enforce $\mathcal{P} \succeq 0$ only for $s \in [a,b]$ (or $(x,y) \in [a,b] \times [c,d]$), the option **psatz** can be used as

```
| >> [prog,P] = poslpivar(prog,n,I,d);
| >> opts.psatz = 1;
| >> [prog,P2] = poslpivar(prog,n,I,d,opts);
| >> P = P+P2;
```

This will allow P to be negative definite outside of the specified domain **I**, allowing for more freedom in the optimization problem. However, since it involves declaring a second PI operator decision variable **P2**, it may also substantially increase the computational complexity associated with setting up and solving the optimization problem.

Note also that the output of operator \mathcal{P} of `poslpivar` is only positive semidefinite, i.e. $\mathcal{P} \succeq 0$. To ensure strict positive definiteness, a (small) positive constant ϵ can be added to the operator as e.g.

```
| >> ep = 1e-5;
| >> P = P + ep;
```

This amounts to adding a matrix ϵI to the parameters \mathcal{P} and $\mathcal{P} \cdot \mathbf{R} \cdot \mathbf{R}^0$ in 1D, or $\mathcal{P} \cdot \mathbf{R}^0$, $\mathcal{P} \cdot \mathbf{R}_{xx}\{1\}$, $\mathcal{P} \cdot \mathbf{R}_{yy}\{1\}$ and $\mathcal{P} \cdot \mathbf{R}_{22}\{1,1\}$ in 2D, ensuring that $\mathcal{P} \succ 0$.

Example

In the LPI (7.2), we have one positive definite decision variable $\mathcal{P} \succeq 0$. This operator \mathcal{P} should have the same dimensions $\begin{bmatrix} n_0 \\ n_1 \end{bmatrix}$ as the operator $\mathcal{T} : \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[0,1] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[0,1] \end{bmatrix}$, which in our case are $n_0 = 0$ and $n_1 = 1$:

```
| >> Pdim = T.dim(:,1)
| Pdim =
|
|      0
|      1
```

Similarly, we require the spatial domain of the operator \mathcal{P} to match the spatial domain of the PIE as

```
| >> Pdom = PIE.dom
| Pdom =
|
|      0      1
```

We then specify the degrees of the monomials in the operator \mathcal{Z}_d defining $\mathcal{P} = \mathcal{Z}_d^* \mathcal{P} \mathcal{Z}_d$ as

```
| >> Pdeg = {6, [2,3,5], [2,3,5]}
| Pdeg =
|
| 1×3 cell array
|
|      {[6]}      {[2 3 5]}      {[2 3 5]}
```

These degrees were chosen based on some trial and error, finding that they produce an accurate solution, whilst still being relatively small.

To reduce the size of the optimization program, we also require $\mathcal{P} \cdot \mathbf{R} \cdot \mathbf{R}^1 = \mathcal{P} \cdot \mathbf{R} \cdot \mathbf{R}^2$ using the option `sep`,

```
| >> opts.sep = 1;
```

decreasing the freedom in our choice of \mathcal{P} , but also decreasing computational complexity. Then, we can declare a positive PI operator \mathcal{P} of the proposed specifications as

```

>> [prog,P] = poslpivar(prog,Pdim,Pdom,Pdeg,opts)
prog =

    struct with fields:

        var: [1×1 struct]
        expr: [1×1 struct]
        extravar: [1×1 struct]
        objective: [362×1 double]
        solinfo: [1×1 struct]
        vartable: {2×1 cell}
        varmat: [1×1 struct]
        decvartable: {362×1 cell}

P =

    [] | []
    -----
    [] | P.R

P.R =

    Too big to display. Use ans.R.R0 | Too big to display. Use ans.R.R1 |

```

The output object P is a `dopvar` objects, representing a PI operator decision variable rather than a fixed PI operator. As such, the parameters P , $Q1$, $Q2$ and R defining this PI operator are `dpvar` class objects, parameterized by decision variables `coeff_i`. These decision variables `coeff_i` are collected in the field `decvartable` of the program structure (along with the previously declared decision variable `gam`), and represent the matrix T in the expansion $\mathcal{P} = \mathcal{Z}_d^* T \mathcal{Z}_d$, constrained to satisfy $T \succcurlyeq 0$.

In the LPI (7.2), the operator \mathcal{P} is required to be strictly positive definite, satisfying $\mathcal{P} \succeq \epsilon I$ for some $\epsilon > 0$. To enforce this, we let $\epsilon = 10^{-6}$, and ensure strict positivity of \mathcal{P} by calling

```

>> eppos = 1e-6;
>> P.R.R0 = P.R.R0 + eppos*eye(size(P));

```

7.2.3 lpivar

A general (indefinite) PI operator decision variable \mathcal{Z} can be declared in PIETOOLS using the `lpivar` function as shown below.

```

| >> [prog,Z] = lpivar(prog,n,I,d);

```

This function takes three mandatory inputs.

- `prog`: An sosprogram structure to which to add the PI operator decision variable.
- `n`: a 2×2 array `[m0,n0;m1,n1]` specifying the dimensions of $\mathcal{Z} : \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a,b] \end{bmatrix}$ for a 1D operator, or 4×2 array `[m0,n0;m1,n1;m2,n2;m3,n3]` specifying the dimensions for a 2D operator $\mathcal{Z} : \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \\ L_2^{n_2}[c,d] \\ L_2^{n_3}[[a,b] \times [c,d]] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a,b] \\ L_2^{m_2}[c,d] \\ L_2^{m_3}[[a,b] \times [c,d]] \end{bmatrix}$.

- **I**: a 1×2 array $[a, b]$ specifying the interval $[a, b]$ of the variables in Z in 1D, or a 2×2 array $[a, b; c, d]$ specifying the spatial domain $[a, b] \times [c, d]$ of the variables in Z in 2D.
- **d** (optional):
 - 1D: An array structure of the form $[b0, b1, b2]$, specifying the degree **b0** of s in Q1, Q2, R0, the degree **b1** of θ in Z.R.R1, Z.R.R2, and the degree **b2** of s in Z.R.R1, Z.R.R2.
 - 2D: A structure with fields **d.dx**, **d.dy**, **d.d2**, specifying degrees for operators along $x \in [a, b]$, along $y \in [c, d]$, and along both $(x, y) \in [a, b] \times [c, d]$; call `help poslpivar_2d` for more information.

The output is a `dopvar` object Z representing an indefinite PI decision variable, and an updated program structure to which the decision variable has been added.

Note:

Indefinite PI operator decision variables \mathcal{Z} need not be symmetric. As such, the second argument **n** to the function `lpivar` must specify both the output (row) dimensions of the operator \mathcal{Z} ($n(:, 1)$), and the input (column) dimensions of the operator \mathcal{Z} ($n(:, 2)$).

Example

For the LPI (7.2), we need to declare a PI operator decision variable \mathcal{Z} which need not be positive or negative definite. Here, if $\mathcal{C}_2 : \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a, b] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a, b] \end{bmatrix}$, then $\mathcal{Z} : \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a, b] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a, b] \end{bmatrix}$, so we specify the dimensions and domain of \mathcal{Z} as

```
>> Zdim = C2.dim(:, [2, 1])
Zdim =
     0     1
     1     0
>> Zdom = PIE.dom;
```

so that in our case $\mathcal{Z} : \mathbb{R} \rightarrow L_2[0, 1]$. As such, only the parameter $Z.Q2$ will be non-empty, and we will allow this parameter to be a quartic function of s , specifying degrees

```
>> Zdeg = [4, 0, 0];
```

Using the function `lpivar`, we then declare an indefinite PI operator decision variable as

```

>> [prog,Z] = lpivar(prog,Zdim,Zdom,Zdeg)
prog =

  struct with fields:

      var: [1×1 struct]
      expr: [1×1 struct]
      extravar: [1×1 struct]
      objective: [367×1 double]
      solinfo: [1×1 struct]
      vartable: {2×1 cell}
      varmat: [1×1 struct]
      decvaritable: {367×1 cell}

Z =

      [ ] | [ ]
-----
[coeff_362+coeff_363*s+coeff_364*s^2+coeff_365*s^3+coeff_366*s^4] | Z.R

Z.R =

[ ] | [ ] | [ ]

```

where we note that $Z.Q2$ is indeed a quartic function of s . The resulting `dopvar` object Z only involves five decision variables `coeff`, increasing the total number of decision variables in `prog.decvaritable` to 367.

7.3 Imposing Constraints

Constraints form a crucial aspect of most optimization problems. In LPIs, constraints often appear as inequality or equality conditions on 4-PI objects (e.g. $Q \leq 0$ or $Q = 0$). These constraints can be set up using the functions `lpi_ineq` and `lpi_eq` respectively, as we show in the next subsections.

7.3.1 `lpi_ineq`

Given a program structure `prog` and `dopvar` object Q (representing a PI operator variable Q), an inequality constraint $Q \succeq 0$ can be added to the program by calling

```
| >> prog = lpi_ineq(prog,Q,opts);
```

This function takes three input arguments

- `prog`: An sosprogram structure to which to add the constraint $Q \succeq 0$.
- `Q`: A `dopvar` or `dopvar2d` object representing the PI operator Q .
- `opts` (optional): This is a structure with fields
 - `psatz`: Binary scalar indicating whether to enforce the constraint only locally. If `psatz=0` (default), a constraint $Q = \mathcal{R}_1$ will be enforced, where $\mathcal{R} \succeq 0$ will be declared as a `dopvar` or `dopvar2d` object

```
| >> R1=poslpivar(prog,n,I,d,opts1)
```

with `opts1.psatz=0`. If `psatz=1`, a constraint $Q = R_1 + R_2$ will be enforced, with R_1 as before, and $R_2 \succeq 0$ declared as a `dopvar` or `dopvar2d` object

```
| >> R2=poslpivar(prog,n,I,d,opts2)
```

with `opts2.psatz=0`. Using `psatz=1` allows the constraint $Q \succeq 0$ to be violated outside of the spatial domain I , but may also substantially increase the computational effort in solving the problem.

Calling `lpi_ineq`, a modified optimization structure `prog` is returned with the constraints $Q \succeq 0$ included. Note that the constraint imposed by `lpi_ineq` is always **non-strict**. For strict positivity, an offset $\epsilon > 0$ may be introduced, enforcing $Q - \epsilon \succeq 0$ to ensure $Q \succeq \epsilon I \succ 0$. This may be implemented as `lpi_ineq(prog,Q-ep)` where `ep` is a small positive number.

Example

For the LPI (7.2), we impose the constraint $Q \preceq 0$ by calling

```
>> nw = size(B1,2);      nz = size(C1,1);
>> Q = [-gam*eye(nw),    -D11',      -(P*B1+Z*D21)']*T;
        -D11,            -gam*eye(nz),  C1;
        -T'*(P*B1+Z*D21), C1',        (P*A+Z*C2)']*T+T'*(P*A+Z*C2)];
>> prog = lpi_ineq(prog,-Q);
prog =

  struct with fields:

      var: [1×1 struct]
     expr: [1×1 struct]
  extravar: [1×1 struct]
 objective: [20248×1 double]
   solinfo: [1×1 struct]
  vartable: {2×1 cell}
   varmat: [1×1 struct]
 decvartable: {20248×1 cell}
```

We note that `lpi_ineq` enforces the constraint $-Q \succeq 0$ by introducing a new PI operator decision variable $R \succeq 0$, and imposing the equality constraint $-Q = R$. In doing so, `lpi_ineq` tries to ensure that the degrees of the polynomial parameters defining R match those of the parameters defining $-Q$, in this case parameterizing R by $20248 - 367 = 19881$ decision variables (check the size of `decvartable`). An operator R involving more or fewer decision variables can also be declared manually, at which point the constraint $-Q = R$ can be enforced using `lpi_eq`, as we show next.

7.3.2 lpi_eq

Given a program structure `prog` and `dopvar` object `Q` (representing a PI operator decision variable Q), an equality constraint $Q = 0$ can be added to the program by calling

```
| >> prog = lpi_eq(prog,Q);
```

This returns a modified optimization structure `prog` with the constraints $Q=0$ included. Note that equality constraints such as $Q_1 = Q_2$ may be equivalently written as e.g. $Q_1 - Q_2 = 0$, which may be implemented as `prog = lpi_eq(prog,Q-R)`.

Example

For the LPI (7.2), we can enforce the constraint $Q \preceq 0$ by declaring a new positive semidefinite PI operator decision variable $R \succeq 0$, and enforcing $Q = -R \preceq 0$ as

```
>> nw = size(B1,2);      nz = size(C1,1);
>> Q = [-gam*eye(nw),   -D11',      -(P*B1+Z*D21)']*T;
        -D11,          -gam*eye(nz),  C1;
        -T'*(P*B1+Z*D21), C1',      (P*A+Z*C2)']*T+T'*(P*A+Z*C2)];
>> [prog_alt,R] = poslpivar(prog,Q.dim(:,1),Q.I);
>> prog_alt = lpi_eq(prog_alt,Q+R)
prog_alt =

    struct with fields:

        var: [1×1 struct]
        expr: [1×1 struct]
        extravar: [1×1 struct]
        objective: [467×1 double]
        solinfo: [1×1 struct]
        vartable: {2×1 cell}
        varmat: [1×1 struct]
        decvar: {467×1 cell}
```

The new program structure will include the constraints $R \succeq 0$ and $Q + R = 0$. The resulting number of decision variables (467) is substantially smaller than in the program obtained using `lpi_ineq`, as the default degrees used by `poslpivar` are relatively small. Specifying higher degrees d in declaring $R = \text{poslpivar}(\text{prog},Q.\text{dim}(:,1),Q.I,d)$, we can increase the freedom in the optimization problem, potentially increasing accuracy but also increasing computational complexity.

7.4 Defining an Objective Functions

Aside from constraints, many optimization problems also involve an objective function, aiming to minimize or maximize some function of the decision variables. To **minimize** the value of a **linear** objective function $f(d_1, \dots, d_n)$, where d_1, \dots, d_n are decision variables, call `sossetobj` with as first argument the program structure, and as second argument the function $f(d)$:

```
| >> prog = sossetobj(prog, f);
```

where `f` must be a `dpvar` object representing the objective function. For example, a function $f(\gamma_1, \gamma_2) = \gamma_1 + 5\gamma_2$ could be specified as objective function using

```
>> dpvar gam1 gam2;
>> f = gam1 + 5*gam2;
>> prog = sosdecvar(prog, [gam1;gam2]);
>> prog = sossetobj(prog, f);
```

Note:

- The objective function must always be linear in the decision variables.
- Only one (scalar) objective function can be specified.
- In solving the optimization program, the value of the objective function will always be minimized. Thus, to maximize the value of the objective function $f(d)$, specify $-f(d)$ as objective function.

Example

In the LPI (7.2), the value of the decision variable γ is minimized. As such, the objective function in this problem is just $f(\gamma) = \gamma$, which we declare as

```
>> prog = sossetobj(prog, gam);
prog =

    struct with fields:

        var: [1×1 struct]
        expr: [1×1 struct]
        extravar: [1×1 struct]
        objective: [23776×1 double]
        solinfo: [1×1 struct]
        vartable: {2×1 cell}
        varmat: [1×1 struct]
        decvar: {23776×1 cell}
```

The field `objective` in the resulting structure `prog` will be a vector with all elements equal to zero, except the first element equal to 1, specifying that the objective function is equal to 1 times the first decision variable in `decvar`, which is γ .

7.5 Solving the Optimization Problem

Once an optimization program has been specified as a program structure `prog`, it can be solved by calling `solSolve`

```
>> prog_sol = solSolve(prog,opts);
```

Here `opts` is an optional argument to specify settings in solving the optimization program, with fields

- `opts.solve`: char type object specifying which semidefinite programming (SDP) solver to use. Options include ‘`sedumi`’ (default), ‘`mosek`’, ‘`sdpt3`’, and ‘`sdpnalplus`’. Note that these solvers must be separately installed in order to use them.

- `opts.simplify`: Binary value indicating whether the solver should attempt to simplify the SDP before solving. The simplification process may take additional time, but may reduce the time of actually solving the SDP.

After calling `soossolve`, it is important to check whether the problem was actually solved. Using the solver SeDuMi, this can be established looking at e.g. the value of `feasratio`, which will be close to +1 if the problem was successfully solved, and the values of `pinf` and `dinf`, which should both be zero if the problem is primal and dual solvable. If `soossolve` is unsuccessful in solving the problem, the problem may be infeasible, or different settings must be used in declaring the decision variables and constraints (e.g. include higher degree monomials in the positive operators).

Example

Having declared the full LPI (7.2), we can finally solve the problem as

```
>> opts.solver = 'sedumi';
>> opts.simplify = true;
>> prog_sol = soossolve(prog,opts);

Residual norm: 1.1879e-05

      iter: 19
feasratio: 0.8308
      pinf: 0
      dinf: 0
      numerr: 1
      timing: [0.1475 5.4485 0.0113]
      wallsec: 5.6073
      cpusec: 6.3700
```

We note that the problem was not found to be either primal or dual infeasible, and the value of `feasratio` is fairly close to 1. However, the solver did run into numerical errors, as signified by `numerr: 1`. Nevertheless, we will refrain from performing an additional test (e.g. rerunning with greater degrees for the monomials defining \mathcal{P}) for the purposes of this demonstration.

7.6 Extracting the Solution

Calling `prog_sol=soossolve(prog)`, a program structure `prog_sol` is returned that is very similar to the input structure `prog`, defining the solved optimization problem. From this solved structure, solved values of the decision variables can be extracted using `soosgetsol` and `getsol_lpivar`, as we show next.

7.6.1 soosgetol

To obtain the (optimal) value of a decision variable after an LPI optimization program has been solved, the function `soosgetsol` can be used, passing the solved optimization program structure as first argument, and the considered decision variable as second argument:

```
| >> f_val = sosgetsol(prog_sol,f);
```

Here, `f` must be a `dpvar` object, representing any polynomial function that is affine in the decision variables that appear in the optimization program.

To extract the optimal value of γ found when solving the LPI (7.2), we call

```
| >> gam_val = sosgetsol(prog_sol,gam)
| gam_val =
| 1.0028
```

We find that, through proper choice of the operator \mathcal{L} , the estimator can achieve an H_∞ -norm $\frac{\|\tilde{z}\|_{L_2}}{\|w\|_{L_2}} \leq 1.0028$.

7.6.2 getsol_lpivar

To retrieve the values of `dopvar` decision variables after an optimization program has been solved, the function `getsol_lpivar` can be used, passing the solved optimization program structure `prog_sol` as first argument, and the considered `dopvar` decision variable as second argument:

```
| >> Pop_val = sosgetsol(prog_sol,Pop);
```

Note that `sosprogram prog_sol` must be in a solved state (`sossolve` must have been called) to retrieve the solution for the input `dopvar` or `dopvar2d` object `Pop`. The output `Pop_val` will then be `opvar` or `opvar2d` class object, representing a fixed PI operator, with the solved (optimal) values of the decision variables substituted into the associated parameters.

Example

To extract the values of the operators \mathcal{P} and \mathcal{Z} for which the LPI (7.2) has been found feasible, we call

```
| >> Pval = getsol_lpivar(prog_sol,P);
| >> Zval = getsol_lpivar(prog_sol,Z);
```

The resulting object `Pval` and `Zval` are `opvar` objects, representing the values of the operator \mathcal{P} and \mathcal{Z} for which the LPI (7.2) holds. Using these values, we can compute an operator \mathcal{L} such that the Estimator (7.1) satisfies $\frac{\|\tilde{z}\|_{L_2}}{\|w\|_{L_2}} \leq \gamma = 1.0028$, as we show next.

When performing estimator or controller synthesis (see also Chapter 13), the optimal estimator or controller associated to a solved problem has to be constructed from the solved PI operator decision variables. For example, for the estimator in (7.1), the value of \mathcal{L} is determined by the values of \mathcal{P} and \mathcal{Z} in the LPI (7.2) as $\mathcal{L} = \mathcal{P}^{-1}\mathcal{Z}$. To facilitate this post-processing of the solution, PIETOOLS includes several utility functions. We note that these functions have not been included for 2D operators in PIETOOLS 2022.

7.6.2a getObserver

For a solution $(\mathcal{P}, \mathcal{Z})$ to the optimal estimator LPI (7.2), the operator \mathcal{L} in the Estimator (7.1) can be computed as

```
| >> Lval = getObserver(Pval,Zval);
```

where Pval and Zval are opvar objects representing the (optimal) values of \mathcal{P} and \mathcal{Z} in the LPI, and Lval is an opvar object representing the associated (optimal) value of \mathcal{L} in the estimator.

Example

Given the opvar objects Pval and Zval, we can finally construct an optimal observer operator \mathcal{L} for the System (7.1) by calling

```
| >> Lval = getObserver(Pval,Zval)
| Lval =
|
|          [] | []
| -----
| Too big to display. Use ans.Q2 | Lval.R
|
| Lval.R =
|   [] | [] | []
```

where the expression for Lval.Q2 is rather complicated. Nevertheless, using this value for the operator \mathcal{L} , an H_∞ norm $\frac{\|z\|_{L_2}}{\|w\|_{L_2}} \leq \gamma = 1.0028$ can be achieved. Increasing the freedom in our optimization problem, e.g. by increasing the degrees of the monomials defining \mathcal{P} and \mathcal{Z} , we may be able to achieve a tighter bound on the H_∞ norm for the obtained operator \mathcal{L} , or find another operator \mathcal{L} achieving a smaller value of the norm.

7.6.2b getController

For a solution $(\mathcal{P}, \mathcal{Z})$ to the optimal control LPI (13.13) (see Section 13.3, the operator $\mathcal{K} = \mathcal{Z}\mathcal{P}^{-1}$ defining the feedback law $u = \mathcal{K}\mathbf{v}$ for optimal control of the PIE (13.12) can be computed as

```
| >> Kval = getController(Pval,Zval);
```

where Pval and Zval are opvar objects representing the (optimal) values of \mathcal{P} and \mathcal{Z} in the LPI, and Kval is an opvar object representing the associated (optimal) value of \mathcal{K} in the feedback law $u = \mathcal{K}\mathbf{v}$. Note that this feedback law is described in terms of the PIE state \mathbf{v} , not the state of the associated PDE or TDS. Deriving an optimal controller for the associated PDE or TDS system will require careful consideration of how the PIE state relates to the PDE or TDS state.

7.6.2c closedLoopPIE

For a PIE (13.12) and an operator \mathcal{K} defining a feedback law $u = \mathcal{K}\mathbf{v}$, a PIE corresponding to the closed-loop system for the given feedback law can be computed as

```
| >> PIE_CL = getController(PIE_OL,Kval);
```


where `Kval` is an `opvar` object representing the (optimal) value of \mathcal{K} in the feedback law $u = \mathcal{K}\mathbf{v}$, `PIE_OL` is a `pie_struct` object representing the PIE system without feedback, and `PIE_CL` is a `pie_struct` object representing the closed-loop PIE system with the feedback law $u = \mathcal{K}\mathbf{v}$ enforced. Note that the resulting system takes no more actuator inputs u , so that operators `PIE_CL.Tu`, `PIE_CL.B2`, `PIE_CL.D12`, and `PIE_CL.D22` are all empty.

Example

A full code declaring and solving the optimal estimator LPI (7.2) for the PDE (7.3) has been included as a demo file “Hinf_optimal_observer_DEMO” in PIETOOLS. See Section 11.5 for more information. Section 11.5 also shows how the estimated PDE state associated to the obtained operator \mathcal{L} can be simulated with PIESIM.

7.7 Running Pre-Defined LPis: Executives and Settings

Combining the steps from the previous sections, we find that the H_∞ -optimal estimator LPI (7.2) can be declared and solved for any given PIE structure `PIE` using roughly the same code. Therefore, to facilitate solving the H_∞ -optimal estimator problem, the code has been implemented in an executive file `PIETOOLS_Hinf_estimator`, that is structured roughly as

```
>> % Extract the operators and initialize the LPI program
>> T = PIE.T;      A = PIE.A;      B1 = PIE.B1;
>> C1 = PIE.C1;   D11 = PIE.D11;  C2 = PIE.C2;   D12 = PIE.D12;
>> prog = sosprogram([PIE.vars(:,1);PIE.vars(:,2)]);

>> % Declare the objective function min{gamma}
>> dpvar gam;
>> prog = sosdecvar(prog, gam);
>> prog = sossetobj(prog, gam);

>> % Declare the positive operator P>=0
>> [prog,P] = poslpivar(prog,T.dim(:,1),PIE.dom,dd1,options1);
>> if override1==0
>>     % Allow P<=0 outside domain PIE.dom
>>     [prog,P2] = poslpivar(prog,T.dim(:,1),PIE.dom,dd12,options12);
>>     P = P + P2;
>> end
>> % Enforce strict positivity P>0
>> P.P = P.P + eppos*eye(size(P.P));
>> P.R.R0 = eppos2*eye(size(P.R.R0));

>> % Declare the indefinite operator Z
>> [prog,Z] = lpivar(prog,C2.dim(:,[2,1]),PIE.dom,ddZ);

>> % Enforce the negativity constraint Q<=0
>> Q = [-gam*eye(nw),      -D11',      -(P*B1+Z*D21)']*T;
>>      -D11,              -gam*eye(nz),   C1;
>>      -T'*(P*B1+Z*D21), C1',          (P*A+Z*C2)']*T+T'*(P*A+Z*C2) + epneg*T'*T];
```

```

>> if sosineq
>>     % Enforce using lpi_ineq
>>     prog = lpi_ineq(prog,-Q,opts);
>> else
>>     % Enforce using lpi_eq
>>     [prog,R] = poslpivar(prog,Q.dim(:,1),Q.I,dd2,options2);
>>     if override2==0
>>         % Allow R<=0 outside of domain Q.I
>>         [prog,R2] = poslpivar(prog,Q.dim(:,1),Q.I,dd3,options3);
>>         R = R+R2;
>>     end
>>     % Enforce Q=-R<=0
>>     prog = lpi_eq(prog,Q+R);
>> end

>> % Solve the optimization program and extract the solution
>> prog_sol = sossolve(prog, sos_opts);
>> gam_val = sosgetsol(prog_sol, gam);
>> Pval = getsol_lpivar(prog_sol, P);
>> Zval = getsol_lpivar(prog_sol, Z);
>> Lval = getObserver(Pval, Zval);

```

We note that, in this code, that there are several parameters that can be set, including what degrees to use for the PI operator decision variables (`dd1`, `ddZ`, `dd2`, etc.), whether or not to enforce positivity/negativity strictly and/or locally (`eppos`, `epneg`, `override1`, etc.), and what options to use in calling each of the different functions (`options1`, `opts`, `sos_opts`, etc.). To specify each of the options, the executive files such as `PIETOOLS_Hinf_estimator` can be called with an optional argument `settings`, as we describe in the following subsection.

7.7.1 Settings in PIETOOLS Executives

When calling an executive function such as `PIETOOLS_Hinf_estimator` in `PIETOOLS`, a second (optional) argument can be used to specify settings to use in declaring the LPI program. This argument should be a MATLAB structure with fields as defined in Table 7.1, specifying a value for each of the different options to be used in declaring the LPI program.

To help in declaring settings for the executive files, `PIETOOLS` includes several pre-defined `settings` structures, allowing LPI programs of varying complexity to be constructed. In particular, we distinguish `extreme`, `stripped`, `light`, `heavy` and `veryheavy` settings, corresponding to LPI programs of increasing complexity. A `settings` structure associated to each can be extracted by calling the function `lpisettings`, using

```
| >> settings = lpisettings(complexity, epneg, simplify, solver);
```

This function takes the following arguments:

- `complexity`: A `char` object specifying the complexity for the settings. Can be one of `'extreme'`, `'stripped'`, `'light'`, `'heavy'`, `'veryheavy'` or `'custom'`.
- `epneg`: (optional) Positive scalar ϵ indicating how strict the negativity condition $Q \preceq \epsilon \|\mathcal{T}\|^2$ would need to be in e.g. the LPI for stability. Defaults to 0, enforcing $Q \preceq 0$.

settings Field	Application
eppos	Nonnegative (small) scalar to enforce strict positivity of Pop.P
eppos2	Nonnegative (small) scalar to enforce strict positivity of Pop.R0
epneg	Nonnegative (small) scalar to enforce strict negativity of Dop
sosineq	Binary value, set 1 to use <code>sosineq</code>
override1	Binary value, set 1 to let <code>P2op = 0</code>
override2	Binary value, set 1 to let <code>De2op = 0</code>
dd1	1x3 cell structure defining monomial degrees for Pop
dd12	1x3 cell structure defining monomial degrees for P2op
dd2	1x3 cell structure defining monomial degrees for Deop
dd3	1x3 cell structure defining monomial degrees for De2op
ddZ	1x3 array defining monomial degrees for Zop
options1	Structure of <code>poslpivar</code> options for Pop
options12	Structure of <code>poslpivar</code> options for P2op
options2	Structure of <code>poslpivar</code> options for Deop
options3	Structure of <code>poslpivar</code> options for De2op
opts	Structure of <code>lpi_ineq</code> options for enforcing $Dop \leq 0$
sos_opts	Structure of <code>sosolve</code> options for solving the LPI

Table 7.1: Fields of settings structure passed on to PIETOOLS executive files

- `simplify`: (optional) A `char` object set to `'psimplify'` if the user wishes to simplify the SDP produced in the executive before solving it, or set to `''` if not. Defaults to `''`.
- `solver`: (optional) A `char` object specifying which solver to use to solve the SDP in the executive. Options include `'sedumi'` (default), `'mosek'`, `'sdpt3'`, and `'sdpnalplus'`. Note that these solvers must be separately installed in order to use them.

Note that, using higher-complexity settings, the number of decision variables in the optimization problem will be greater. This offers more freedom in solving the optimization program, thereby allowing for (but not guaranteeing) more accurate results, but also (substantially) increasing the computational effort. We therefore recommend initially trying to solve with e.g. `stripped` or `light` settings, and only using heavier settings if the executive fails to solve the problem. Note also that PIETOOLS includes a `custom` settings file, which can be used to declare custom settings for the executives.

Once settings have been specified, the desired LPI can be declared and solved for a PIE represented by a structure `PIE` by simply calling the corresponding executive file, solving e.g. the H_∞ -optimal estimator LPI (7.2) by calling

```
>> settings = lpisettings('light');
>> [prog, Lop, gam, P, Z] = PIETOOLS_Hinf_estimator(PIE, settings);
```

If successful, this returns the program structure `prog` associated to the solved problem, as well as a `opvar` object `Lop` and scalar `gam` respectively corresponding to the operator \mathcal{L} in the estimator (7.1) and associated estimation error gain γ . The function also returns `dopvar` objects `P` and `Z`, corresponding to the unsolved PI operator decision variables in the LPI.

7.7.2 Executive Functions Available in PIETOOLS

In addition to the H_∞ -optimal estimation LPI, PIETOOLS includes several other executive files to run standard LPI programming tests for a provided PIE. For example, stability of the PIE defined by PIE when $w = 0$ can be tested by calling

```
>> settings = lpsettings('light');
>> [prog] = PIETOOLS_stability(PIE, settings);
```

returning the optimization program structure `prog` associated to the solved program, and displaying a message of whether the system was found to be stable or not in the command window.

Table 7.2 lists the different executive functions that have already been implemented in PIETOOLS. For each executive, a brief description of its purpose is provided, along with a mathematical description of the LPI that is solved. Each executive can be called for a PIE structure with fields

T:	[nx × nx opvar];	Tw:	[nx × nw opvar];	Tu:	[nx × nu opvar];
A:	[nx × nx opvar];	B1:	[nx × nw opvar];	B2:	[nx × nu opvar];
C1:	[nz × nx opvar];	D11:	[nz × nw opvar];	D12:	[nz × nu opvar];
C2:	[ny × nx opvar];	D21:	[ny × nw opvar];	D22:	[ny × nu opvar];

representing a PIE of the form

$$\begin{aligned} \mathcal{T}_u \dot{u}(t) + \mathcal{T}_w \dot{w} + \mathcal{T} \dot{\mathbf{x}}_f(t) &= \mathcal{A} \mathbf{x}_f(t) + \mathcal{B}_1 w(t) + \mathcal{B}_2 u(t), \\ z(t) &= \mathcal{C}_1 \mathbf{x}_f(t) + \mathcal{D}_{11} w(t) + \mathcal{D}_{12} u(t), \\ y(t) &= \mathcal{C}_2 \mathbf{x}_f(t) + \mathcal{D}_{21} w(t) + \mathcal{D}_{22} u(t). \end{aligned}$$

For more information on the origin and application of each LPI, see the references provided in the table, as well as Chapter 13.

Problem	LPI
[prog, P] = PIETOOLS_stability(PIE, settings)	
Test stability of the PIE for $w = 0$ and $u = 0$, by verifying feasibility of the primal LPI [8].	$\mathcal{P} \succ 0$ $\mathcal{T}^* \mathcal{P} \mathcal{A} + \mathcal{A}^* \mathcal{P} \mathcal{T} \preceq 0$
[prog, P] = PIETOOLS_stability_dual(PIE, settings)	
Test stability of the PIE for $w = 0$ and $u = 0$ by verifying feasibility of the dual LPI [9].	$\mathcal{P} \succ 0$ $\mathcal{T} \mathcal{P} \mathcal{A}^* + \mathcal{A} \mathcal{P} \mathcal{T}^* \preceq 0$
[prog, P, gam] = PIETOOLS_Hinf_gain(PIE, settings)	
Determine an upper bound γ on the \mathcal{H}_∞ -norm $\sup_{w,z \in L_2} \frac{\ z\ _{L_2}}{\ w\ _{L_2}}$ of the PIE for $u = 0$, by solving the primal LPI [8].	$\min_{\gamma, \mathcal{P}} \gamma$ $\mathcal{P} \succ 0$ $\begin{bmatrix} -\gamma I & \mathcal{D}_{11}^* & \mathcal{B}_1^* \mathcal{P} \mathcal{T} \\ (\cdot)^* & -\gamma I & \mathcal{C}_1 \\ (\cdot)^* & (\cdot)^* & \mathcal{T}^* \mathcal{P} \mathcal{A} + \mathcal{A}^* \mathcal{P} \mathcal{T} \end{bmatrix} \preceq 0$
[prog, P, gam] = PIETOOLS_Hinf_gain_dual(PIE, settings)	
Determine an upper bound γ on the \mathcal{H}_∞ -norm $\sup_{w,z \in L_2} \frac{\ z\ _{L_2}}{\ w\ _{L_2}}$ of the PIE for $u = 0$, by solving the dual LPI [9].	$\min_{\gamma, \mathcal{P}} \gamma$ $\mathcal{P} \succ 0$ $\begin{bmatrix} -\gamma I & \mathcal{D}_{11} & \mathcal{T} \mathcal{P} \mathcal{C}_1 \\ (\cdot)^* & -\gamma I & \mathcal{B}_1^* \\ (\cdot)^* & (\cdot)^* & \mathcal{T} \mathcal{P} \mathcal{A}^* + \mathcal{A} \mathcal{P} \mathcal{T}^* \end{bmatrix} \preceq 0$
[prog, L, gam, P, Z] = PIETOOLS_Hinf_estimator(PIE, settings)	
Establish an \mathcal{H}_∞ -optimal observer $\mathcal{T} \dot{\hat{\mathbf{x}}}_f = \mathbf{A} \hat{\mathbf{x}}_f + \mathcal{L}(\mathcal{C}_1 \hat{\mathbf{x}}_f - y)$ for the PIE with $u = 0$ by solving the LPI and returning $\mathcal{L} = \mathcal{P}^{-1} \mathcal{Z}$ [1].	$\min_{\gamma, \mathcal{P}, \mathcal{Z}} \gamma$ $\mathcal{P} \succ 0$ $\begin{bmatrix} -\gamma I & \mathcal{D}_{11}^* & (\mathcal{P} \mathcal{B}_1 + \mathcal{Z} \mathcal{D}_{21})^* \mathcal{T} \\ (\cdot)^* & -\gamma I & \mathcal{C}_1 \\ (\cdot)^* & (\cdot)^* & \mathcal{T}^* (\mathcal{P} \mathcal{A} + \mathcal{Z} \mathcal{C}_2) + (\mathcal{P} \mathcal{A} + \mathcal{Z} \mathcal{C}_2)^* \mathcal{P} \mathcal{T} \end{bmatrix} \preceq 0$
[prog, K, gam, P, Z] = PIETOOLS_Hinf_control(PIE, settings)	
Establish an \mathcal{H}_∞ -optimal controller $u = \mathcal{K} \mathbf{x}_f$ for the PIE by solving the LPI and returning $\mathcal{K} = \mathcal{Z} \mathcal{P}^{-1}$ [9].	$\min_{\gamma, \mathcal{P}, \mathcal{Z}} \gamma$ $\mathcal{P} \succ 0$ $\begin{bmatrix} -\gamma I & \mathcal{D}_{11} & \mathcal{T} (\mathcal{P} \mathcal{C}_1 + \mathcal{Z} \mathcal{D}_{12}) \\ (\cdot)^* & -\gamma I & \mathcal{B}_1^* \\ (\cdot)^* & (\cdot)^* & \mathcal{T} (\mathcal{A} \mathcal{P} + \mathcal{B}_2 \mathcal{Z})^* + (\mathcal{A} \mathcal{P} + \mathcal{B}_2 \mathcal{Z}) \mathcal{T}^* \end{bmatrix} \preceq 0$

Table 7.2: List of pre-defined executives for analysis and control of PIEs. See also Chapter 13.

Part II

Additional PIETOOLS Functionality

Chapter 8

Alternative Input Formats for ODE-PDE Systems

In PIETOOLS, coupled ODE-PDE systems can be defined using three different approaches: Command Line Parser (via command line or MATLAB scripts), graphical user interface (a MATLAB app), and ‘terms input’ format structure. The former two input methods are useful when the system is defined by using entire equations, whereas the last method (terms input format method) requires the user to identify the parameters from each term of set of equations and define them in a PIETOOLS compatible class called `pde_struct`.

As such, Command Line Parser and GUI input methods are the simplest and most intuitive methods to define a coupled ODE-PDE. However, in this chapter we describe only the GUI and terms-based input methods, referring to Chapter 4 for information on the Command Line Parser input format. In particular, in Section 8.1, we show how any well-posed, linear, coupled 1D ODE-PDE system can be declared in PIETOOLS using the GUI. In Section 8.2, we then show how any well-posed, linear, coupled ODE - 1D PDE - 2D PDE system can be declared in PIETOOLS using the terms-based input format. We note that, although most ODE-PDE coupled systems can be defined by using any of the three input methods, certain type of PDEs (for example, PDEs with 2 spatial dimensions and PDEs with second order time differential terms) can only be defined using the terms input format in PIETOOLS 2022.

8.1 A GUI for Defining PDEs

In addition to the Command Line Parser, PIETOOLS 2022 also allows PDEs to be declared using a graphical user interface (GUI), that provides a simple, intuitive and interactive visual interface to directly input the model. It also allows declared PDE models to be saved and loaded, so that the same system can be used in different sessions without having to declare the model from scratch each time.

To open the GUI, simply call `PIETOOLS_PIETOOLS_GUI` from the command line. You will see something similar to the picture below:

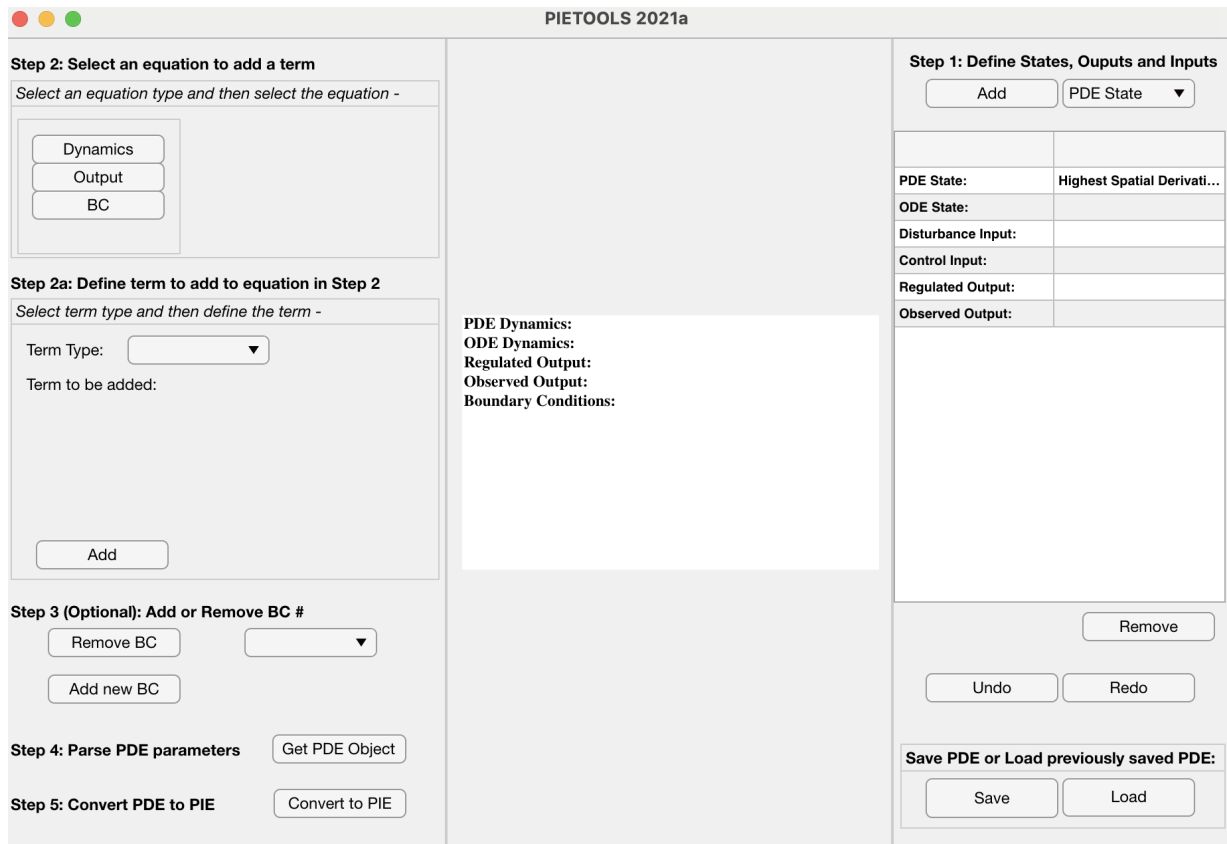


Figure 8.1: GUI overview.

Now we will go over the GUI step-by-step to demonstrate how to define your own linear, 1D ODE-PDE model in PIETOOLS.

8.1.1 Step 1: Define States, Outputs and Inputs

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

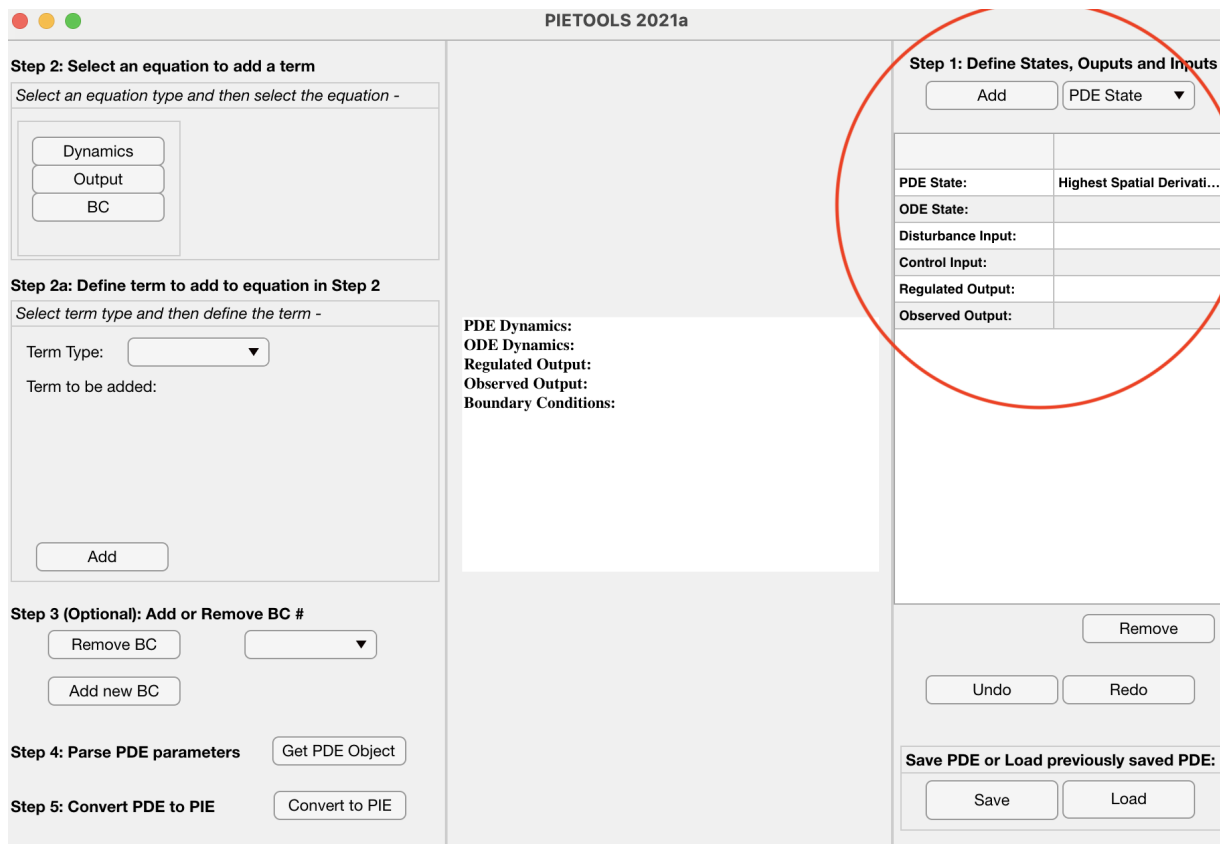


Figure 8.2: Step 1: Define States, Outputs and Inputs

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

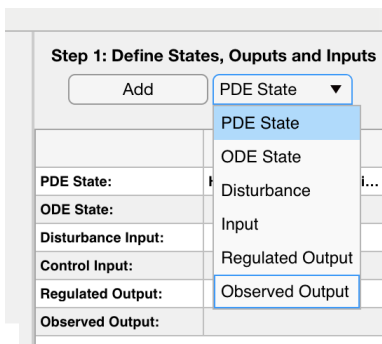


Figure 8.3: Adding variables your model

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.

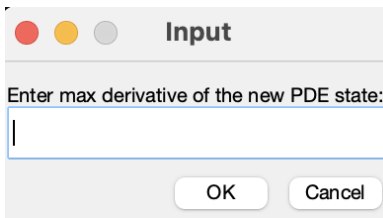


Figure 8.4: Enter the highest order of derivative the particular state admits

- 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.

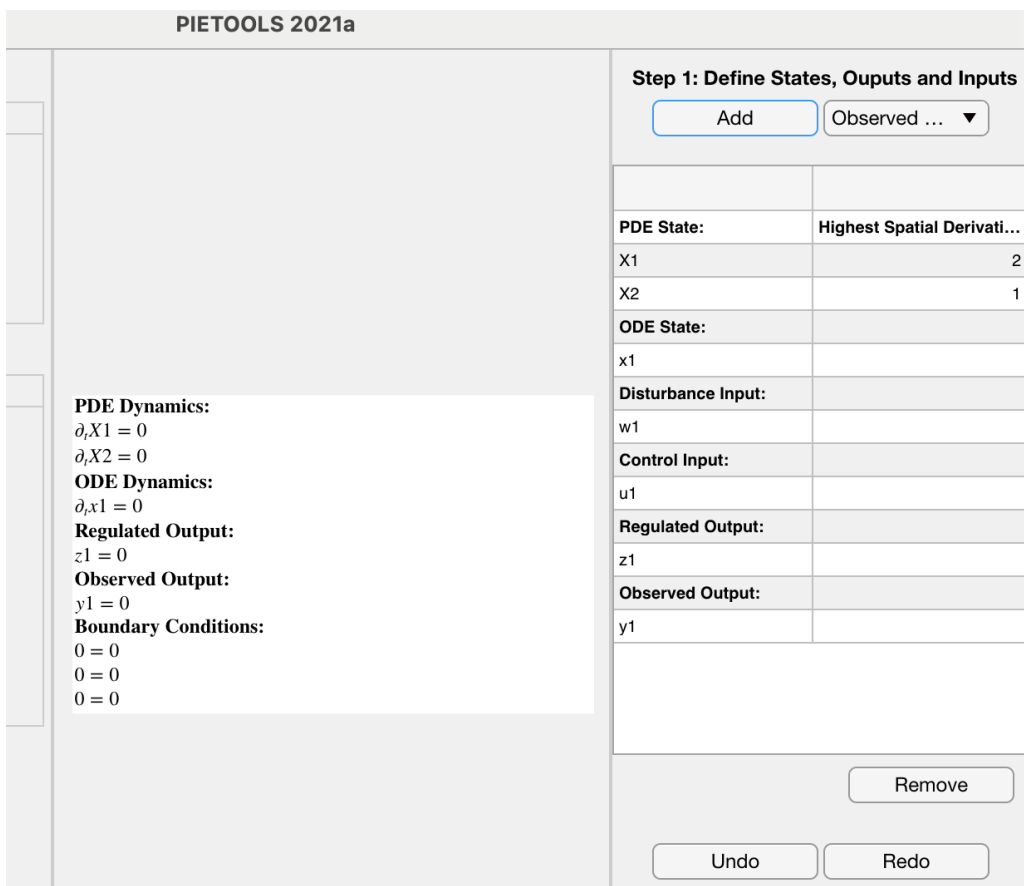


Figure 8.5: After adding the variables to the model

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

8.1.2 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.

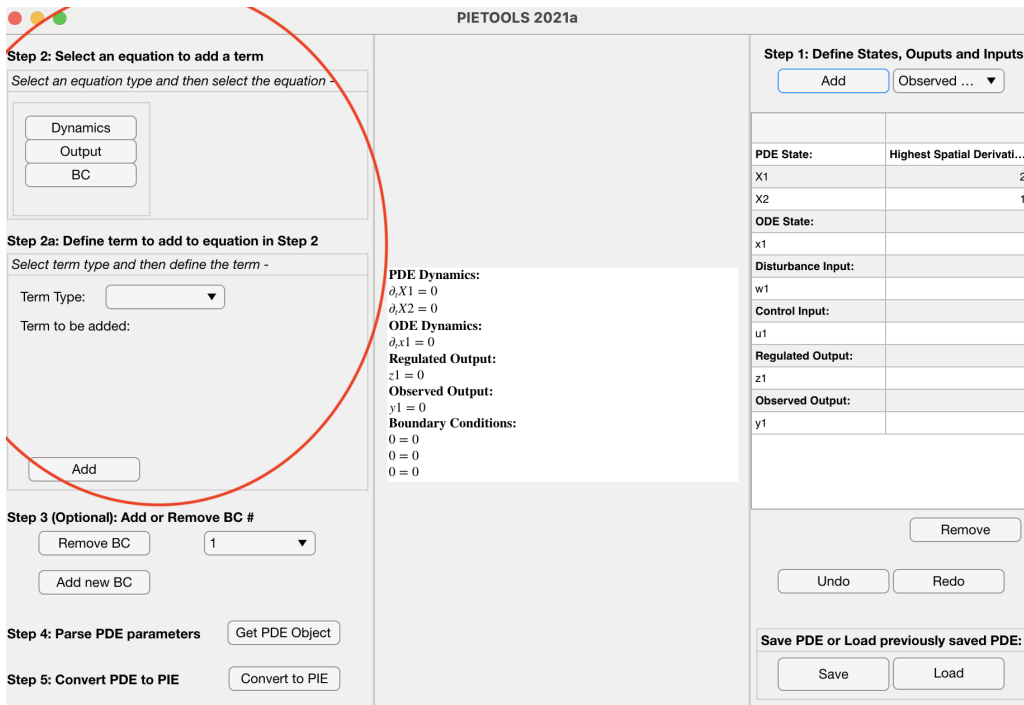
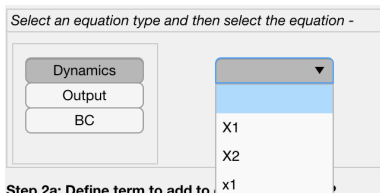


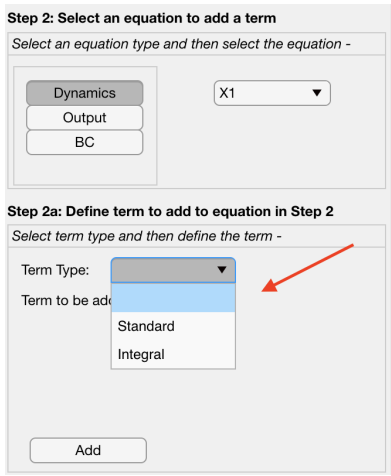
Figure 8.6: Step 2: Select an Equation to Add a Term

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-**.

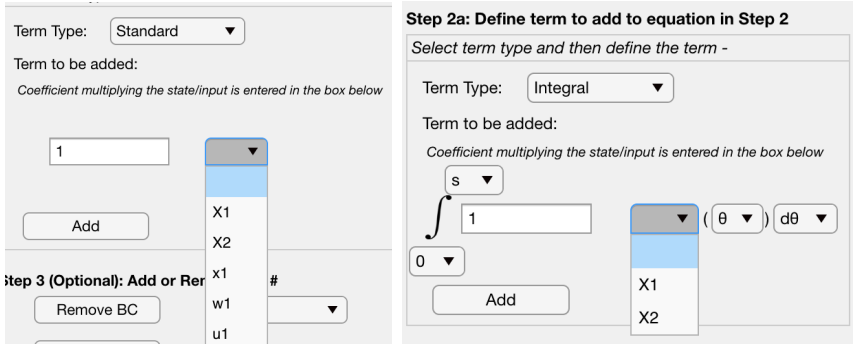
1. In the panel titled **Select an Equation type and the select the equation-**, select either **Dynamics**, **Output** or **BC** (i.e. **Boundary Conditions**).
2. If you select **Dynamics**, all the **PDE** and **ODE** states that you specified in **Step 1** appears.



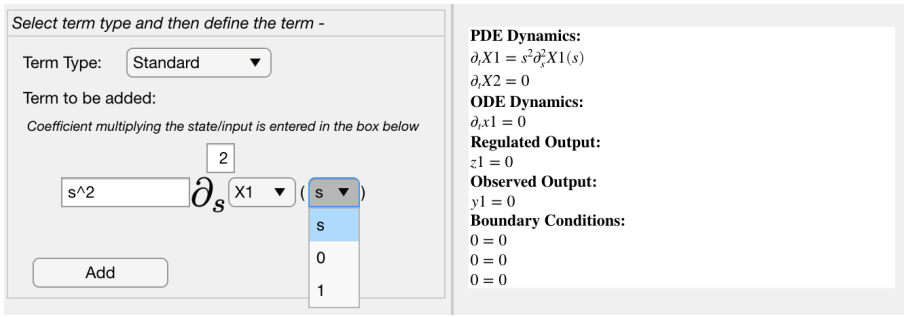
3. Once you select the desired state for which the dynamics term is needed to be added to, go down to the **Step 2.a Select term type and then define the term-**.



4. 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.
5. 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.



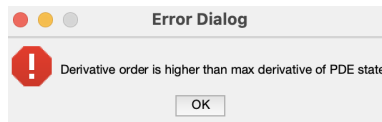
6. Now in **Standard** option, one has to select the variable and add the coefficient in the adjacent panel. Moreover, the PDE states may also contain its derivatives. If you select PDE state, you can input the order of derivative (from 0 up to the highest order derivative for that state), the independent 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.



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.

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



- 7. 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

$\int_0^s \text{theta}^3+3 \partial_s X2 (\theta) d\theta$

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:
 $0 = 0$
 $0 = 0$
 $0 = 0$

Step 2: Select an equation to add a term

Select an equation type and then select the equation -

Dynamics **X2**

Output

BC

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

$5*s \partial_s X2 (s)$

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

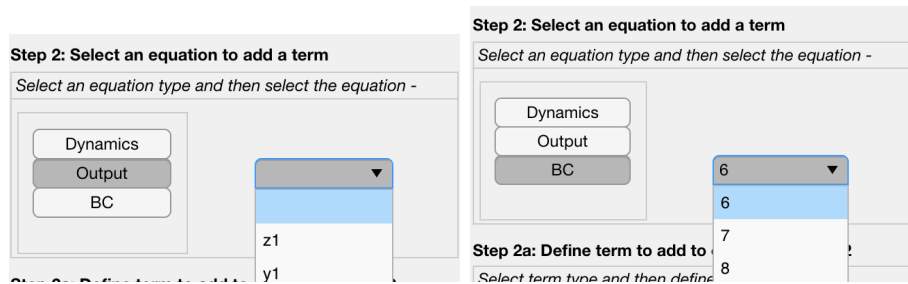
ODE Dynamics:
3. $\partial_t x1 = 0$

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.



Additional Remarks:

A) The integral term of PDE states can only be a function of ' θ '. Moreover, the order of derivative can not exceed the specified order of differentiability of that state. If the input value exceeds this order of differentiability, and error is thrown when trying to add the term.

B) Terms can be specified and added only for one variable at a time. Once a desired variable and one of the options (Dynamics, Output, BC) has been selected at the top, the term can be added following the instructions at the bottom (Step 2a). In order to select another variable for which to add a term, the above steps must be repeated.

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. $y1 = X2(0)$

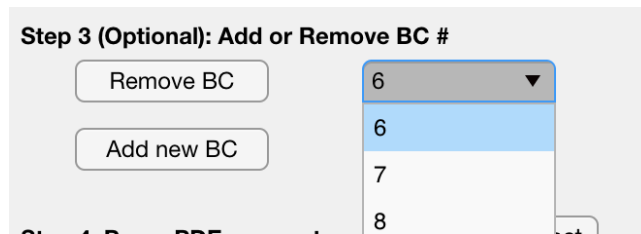
Boundary Conditions:

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

Figure 8.7: An example of a complete model as displayed in the GUI

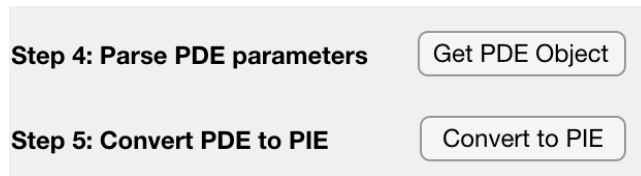
8.1.3 Step 3: (Optional) Add or Remove BC

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.



8.1.4 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.



8.2 The Terms-Based Input Format

In addition to the Command Line Parser and GUI input methods, PIETOOLS also allows ODE-PDE systems to be specified in a terms-based format. Although this format is not quite as intuitive as the alternative input methods, it is the most general input format, used internally by PIETOOLS for e.g. the PDE to PIE conversion. In PIETOOLS 2022, it is also the only format that allows for the implementation of PDEs involving 2 spatial variables.

In the terms-based input format, each term in each equation defining the system is specified as a separate cell element in `pde_struct` class object. These `pde_struct` objects collect all information on the state variables, inputs and outputs, as well as the equations defining their dynamics, through the following fields:

<code>pde_struct</code>	
<code>PDE.x</code>	a cell with each element i specifying a state component \mathbf{x}_i in the system;
<code>PDE.w</code>	a cell with each element i specifying an exogenous input \mathbf{w}_i ;
<code>PDE.u</code>	a cell with each element i specifying an actuator input \mathbf{u}_i ;
<code>PDE.z</code>	a cell with each element i specifying a regulated output \mathbf{z}_i ;
<code>PDE.y</code>	a cell with each element i specifying an observed output \mathbf{y}_i ;
<code>PDE.BC</code>	a cell with each element i specifying a boundary condition for the PDE.

Each element of each of the cells in the above list is yet another struct, with fields

size a scalar integer specifying the size of the state component, input or output;
vars a $p \times 2$ pvar (polynomial) array (for $p \leq 2$), specifying the spatial variables of the state component, input, or output;
dom a $p \times 2$ array specifying the interval on which each spatial variable exists;
term a cell defining the (differential) equation associated to the state component, output, or boundary condition,

specifying e.g. a state component $\mathbf{x}_i(t) \in L_2^n[[a, b] \times [c, d]]$ on variables $(s_1, s_2) \in [a, b] \times [c, d]$ and an exogenous input $w_k(t) \in \mathbb{R}^m$ as

PDE.x{i}.size = n;	PDE.w{k}.size = m;
PDE.x{i}.vars = [s1; s2];	PDE.w{k}.vars = [];
PDE.x{i}.dom = [a,b; c,d];	

The remaining field **term** can then be used to specify the differential equation for the state component \mathbf{x}_i , output \mathbf{y}_i or \mathbf{z}_i , or the i th boundary condition $0 = \mathbf{F}_i$. Here, the j th term in each of these equations is specified through the fields

term{j}.x; integer specifying which state component,
or term{j}.w; **or** exogenous input,
or term{j}.u; **or** actuator input appears in the term;
term{j}.D $1 \times p$ integer array specifying the order of the derivative of the **state component term{j}.x** in each variable;
term{j}.loc $1 \times p$ polynomial or “double” array specifying the spatial position at which to evaluate the **state component term{j}.x**;
term{j}.I $p \times 1$ cell array specifying the lower and upper limits of the integral to take of the state or input;
term{j}.C polynomial or constant factor with which to multiply the state or input,

where p denotes the number of spatial variables on which the component x or input w or u depends. For example, a term involving a state component $\mathbf{x}_k(s_1, s_2)$ depending on two spatial variables of the form

$$\underbrace{\int_{L_1}^{U_1} \int_{L_2}^{U_2}}_{\mathbf{I}} \left(\underbrace{C(s_1, s_2, \theta_1, \theta_2)}_{\mathbf{C}} \overbrace{\partial_{\theta_1}^{d_1} \partial_{\theta_2}^{d_2}}^{\mathbf{D}} \underbrace{\mathbf{x}_k(t - \tau, \theta_1, \theta_2)}_{\mathbf{x}} \right) d\theta_2 d\theta_1. \quad (8.1)$$

can be added to the equation for $\dot{\mathbf{x}}_i$ as

PDE.x{i}.term{j}.x = k;
PDE.x{i}.term{j}.D = [d1,d2];
PDE.x{i}.term{j}.loc = [theta1, theta2];
PDE.x{i}.term{j}.I = {[L1,U1]; [L2,U2]};
PDE.x{i}.term{j}.C = C;

| PDE.x{i}.term{j}.delay = tau;

Similarly, a term allowing a term involving an input as

$$\underbrace{\int_{L_1}^{U_1} \int_{L_2}^{U_2}}_{\mathbf{I}} \left(\underbrace{C(s_1, s_2, \theta_1, \theta_2)}_{\mathbf{C}} \underbrace{\mathbf{w}_k(t - \tau, \theta_1, \theta_2)}_{\mathbf{w}} \right) d\theta_2 d\theta_1, \quad \text{or}$$

$$\underbrace{\int_{L_1}^{U_1} \int_{L_2}^{U_2}}_{\mathbf{I}} \left(\underbrace{C(s_1, s_2, \theta_1, \theta_2)}_{\mathbf{C}} \underbrace{\mathbf{u}_k(t - \tau, \theta_1, \theta_2)}_{\mathbf{u}} \right) d\theta_2 d\theta_1, \quad (8.2)$$

can be added to the PDE of the i th state component as

$$\left| \begin{array}{l} \text{PDE.x}\{i\}.\text{term}\{j\}.\mathbf{w} = \mathbf{k}; \quad \text{or} \quad \text{PDE.x}\{i\}.\text{term}\{j\}.\mathbf{u} = \mathbf{k}; \\ \text{PDE.x}\{i\}.\text{term}\{j\}.\mathbf{I} = \{[L1,U1]; [L2,U2]\}; \\ \text{PDE.x}\{i\}.\text{term}\{j\}.\mathbf{C} = \mathbf{C}; \\ \text{PDE.x}\{i\}.\text{term}\{j\}.\text{delay} = \text{tau}; \end{array} \right.$$

Here, in any term, at most one of the fields \mathbf{x} , \mathbf{w} , and \mathbf{u} can be specified, indicating whether the term involves a state component or input, and specifying which state component or input this would be. If the term involves an input, no field \mathbf{D} or loc can be specified, as PIETOOLS cannot take a derivative of an input, nor evaluate it at any particular position. Using a similar structure, terms in the equations for the outputs and the boundary conditions can also be specified.

In the remainder of this section, we will illustrate through several examples how this terms-based input format can be used to declare general linear ODE-PDE systems in PIETOOLS. In particular:

1. In Section 8.2.1, we show how a system of coupled 2D PDEs can be implemented.
2. In Section 8.2.2, we show how a coupled ODE - 1D PDE - 2D PDE system can be implemented.
3. In Section 8.2.3, we show how systems with inputs and outputs can be implemented.
4. Finally, in Section 8.2.4, we outline some additional options to implement more complicated systems, involving partial integrals, kernel functions, and stricter continuity constraints.

8.2.1 Declaring a System of Coupled 2D PDEs

Suppose we want to implement a PDE given by

$$\begin{aligned} \dot{\mathbf{x}}_1(t, s_1, s_2) &= \begin{bmatrix} 1 \\ 2 \end{bmatrix} \mathbf{x}_2(t, s_1, s_2) + \begin{bmatrix} 10 & 0 \\ 0 & -1 \end{bmatrix} \partial_{s_1} \partial_{s_2} \mathbf{x}_1(t, s_1, s_2), & (s_1, s_2) \in [0, 3] \times [-1, 1], \\ \dot{\mathbf{x}}_2(t, s_1, s_2) &= [1 \quad 1] \partial_{s_1}^2 \mathbf{x}_1(t, s_1, s_2) + [2 \quad -1] \partial_{s_2}^2 \mathbf{x}_1(t, s_1, s_2), & t \geq 0, \end{aligned}$$

$$\begin{aligned}
0 &= \mathbf{x}_1(t, 0, s_2) & 0 &= \partial_{s_1} \mathbf{x}_1(t, 0, s_2), \\
0 &= \mathbf{x}_1(t, s_1, -1), & 0 &= \partial_{s_2} \mathbf{x}_1(t, s_1, -1),
\end{aligned} \tag{8.3}$$

where $\mathbf{x}_1 \in L_2^2[[0, 3] \times [-1, 1]]$ and $\mathbf{x}_2 \in L_2[[0, 3] \times [-1, 1]]$. To implement this PDE, we first initialize an empty `pde_struct` object as

```
| >> PDE = pde_struct();
```

Next, we declare the state components that appear in the system. In (8.3), we have two PDEs, involving two state components $\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}$. Each component and associated PDE is implemented using a separate field `PDE.x{i}`, for which we specify the spatial variables and their domain as

```
| >> pvar s1 s2
| >> PDE.x{1}.vars = [s1;s2];      PDE.x{2}.vars = [s1;s2];
| >> PDE.x{1}.dom = [0,3;-1,1];   PDE.x{2}.dom = [0,3;-1,1];
```

Here, we use the first line to initialize polynomial variables `s1` and `s2`. With the second line, we then indicate that both \mathbf{x}_1 and \mathbf{x}_2 depend on these spatial variables, and we set the domain of these variables using the third line. Note that the variables should be specified as a column vector, so that each row in `dom` defines the domain of the variable(s) in the associated row of `vars`.

Having declared the state components, we now define the PDEs associated to each component. Here, for $i \in \{1, 2\}$, the PDE for state component \mathbf{x}_i is defined one term at a time using the cell structure `PDE.x{i}.term`. For example, to specify the first term in the first PDE,

$$\underbrace{\begin{bmatrix} 1 \\ 2 \end{bmatrix}}_C \underbrace{\mathbf{x}_2}_x(t, s_1, s_2),$$

we specify the following elements

```
| >> PDE.x{1}.term{1}.x = 2;
| >> PDE.x{1}.term{1}.C = [1;2];
```

Here, we set `x{1}.term{1}.x=2` to indicate that the first term in the PDE of the first state component \mathbf{x}_1 is expressed in terms of the second state component \mathbf{x}_2 . Setting `C=[1;2]`, we pre-multiply this term with the matrix $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ before adding it to the equation.

In a similar manner, we declare the second term in the first PDE,

$$\underbrace{\begin{bmatrix} 10 & 0 \\ 0 & -1 \end{bmatrix}}_C \overbrace{\partial_{s_1}^1 \partial_{s_2}^1}^D \underbrace{\mathbf{x}_1}_x(t, s_1, s_2),$$

by calling

```
| >> PDE.x{1}.term{2}.x = 1;
| >> PDE.x{1}.term{2}.D = [1,1];
| >> PDE.x{1}.term{2}.C = [10,0; 0,-1];
```

Here, in addition to specifying the considered state component and coefficient matrix using `term{2}.x` and `term{2}.C`, we also specify a derivative to be taken of the state using the field `term{2}.D`. This field `term{j}.D` specifies the order of the derivative that should be taken of the state component `term{j}.x` with respect to each of the spatial variables on which this state component depends. Setting `D=[1,1]`, a first order derivative will be taken with respect to each of the two variables (s_1, s_2) on which state component \mathbf{x}_1 depends. Note that if no field `D` is specified for a term, it will default to 0, assuming that no differentiation is desired for this term.

Now, to declare the terms in the PDE for the second state component \mathbf{x}_2 ,

$$\underbrace{\begin{bmatrix} 1 & 1 \end{bmatrix}}_C \overbrace{\partial_{s_1}^2 \partial_{s_2}^0}^D \underbrace{\mathbf{x}_1}_{\mathbf{x}}(t, s_1, s_2) + \underbrace{\begin{bmatrix} 2 & -1 \end{bmatrix}}_C \overbrace{\partial_{s_1}^0 \partial_{s_2}^2}^D \underbrace{\mathbf{x}_1}_{\mathbf{x}}(t, s_1, s_2)$$

we can specify each term separately as we did the first equation, by calling

```
>> PDE.x{2}.term{1}.x = 1;          PDE.x{2}.term{2}.x = 1;
>> PDE.x{2}.term{1}.D = [2,0];      PDE.x{2}.term{2}.D = [0,2];
>> PDE.x{2}.term{1}.C = [1,1];      PDE.x{2}.term{2}.C = [2,-1];
```

Here, since both terms depend on the same state component \mathbf{x}_1 , we set the field `x` in both terms equal to 1. However, in such a situation, we can also combine the terms to represent them as

$$\begin{bmatrix} 1 & 1 & 2 & -1 \end{bmatrix} \overbrace{\begin{bmatrix} \partial_{s_1}^2 \partial_{s_2}^0 \\ \partial_{s_1}^0 \partial_{s_2}^2 \end{bmatrix}}^D \underbrace{\begin{bmatrix} \mathbf{x}_1(t, s_1, s_2) \\ \mathbf{x}_1(t, s_1, s_2) \end{bmatrix}}_{\mathbf{x}}$$

which we can implement using a single term

```
>> PDE.x{2}.term{1}.x = 1;
>> PDE.x{2}.term{1}.D = [2,0; 0,2];
>> PDE.x{2}.term{1}.C = [1,1, 2,-1];
```

Having declared the PDE, it remains only to specify the boundary conditions,

$$\begin{aligned} 0 &= \underbrace{\mathbf{x}_1}_{\mathbf{x}}(t, \underbrace{0, s_2}_{\text{loc}}) & 0 &= \underbrace{\mathbf{x}_1}_{\mathbf{x}}(t, \underbrace{s_1, -1}_{\text{loc}}), \\ 0 &= \overbrace{\partial_{s_1}^1 \partial_{s_2}^0}^D \underbrace{\mathbf{x}_1}_{\mathbf{x}}(t, \underbrace{0, s_2}_{\text{loc}}), & 0 &= \overbrace{\partial_{s_1}^0 \partial_{s_2}^1}^D \underbrace{\mathbf{x}_1}_{\mathbf{x}}(t, \underbrace{s_1, -1}_{\text{loc}}) \end{aligned}$$

For these, in addition to specifying e.g. the desired state component and order of the derivative, we also have to specify at which boundary to evaluate the state component. For this, we use the field `loc`, specifying the first (and only) term in each of the four boundary conditions as

```
>> PDE.BC{1}.term{1}.x = 1;          PDE.BC{2}.term{1}.x = 1;
>> PDE.BC{1}.term{1}.loc = [0,s2];  PDE.BC{2}.term{1}.loc = [s1,-1];

>> PDE.BC{3}.term{1}.x = 1;          PDE.BC{4}.term{1}.x = 1;
```

```

| >> PDE.BC{3}.term{1}.D = [1,0];      PDE.BC{4}.term{1}.D = [0,1];
| >> PDE.BC{3}.term{1}.loc = [0,s2];   PDE.BC{4}.term{1}.loc = [s1,-1];

```

Note here that, by not specifying a field C , PIETOOLS assumes the term is not pre-multiplied by any matrix, and the value of C defaults to an identity matrix of the appropriate size.

Having declared the full PDE, we use the function `initialize` to check for errors and fill in any gaps, which produces the output

```

| >> PDE = initialize(PDE);
|
| Encountered 2 state components:
|   x1(t,s1,s2), of size 2, differentiable up to order (2,2) in variables (s1,s2);
|   x2(t,s1,s2), of size 1, differentiable up to order (0,0) in variables (s1,s2);
|
| Encountered 4 boundary conditions:
|   F1(t,s2) = 0, of size 2;
|   F2(t,s1) = 0, of size 2;
|   F3(t,s2) = 0, of size 2;
|   F4(t,s1) = 0, of size 2;

```

matching the desired specifications of our state components and boundary conditions.

8.2.2 Declaring an ODE-PDE System

Having illustrated how to implement general linear 2D PDEs, we now illustrate how systems of coupled ODEs and PDEs can be implemented in PIETOOLS. For example, suppose we have a system of the form

$$\begin{aligned} \dot{x}_1(t) &= -x_1(t) + \int_0^3 2\mathbf{x}_2(t, s_1)ds_1 - \int_{-1}^1 3\mathbf{x}_3(t, 3, s_2)ds_2, & (s_1, s_2) \in [0, 3] \times [-1, 1] \\ \dot{\mathbf{x}}_2(t, s_1) &= s_1x_1(t) + \partial_{s_1}\mathbf{x}_2(t, s_1) - \int_{-1}^1 \mathbf{x}_3(t, s_1, s_2)ds_2, & t \geq 0 \\ \dot{\mathbf{x}}_3(t, s_1, s_2) &= s_1s_2x_1(t) + \mathbf{x}_2(t, s_1) + \partial_{s_2}\mathbf{x}_3(t, s_1, s_2), \\ 0 &= \mathbf{x}_2(t, 3) - x_1(t) \\ 0 &= \mathbf{x}_3(t, 3, -1) - x_1(t - 3) \\ 0 &= \partial_{s_1}\mathbf{x}_3(t, s_1, -1) - \mathbf{x}_2(t, s_1) \end{aligned}$$

where $x_1 \in \mathbb{R}$ is a finite-dimensional state, $\mathbf{x}_2 \in L_2[0, 3]$ varies only along the first spatial dimension, and $\mathbf{x}_3 \in L_2[[0, 3] \times [-1, 1]]$ varies along both spatial directions. As before, we first initialize an empty PDE structure as

```

| >> PDE = pde_struct;

```

Next, we implement each state component and their associated differential equation using a separate field `PDE.x{i}`. In this case, different states vary in (different numbers of) variables, which we indicate through the fields `vars` as

```

| >> pvar s1 s2
| >> PDE.x{1}.vars = [];      PDE.x{2}.vars = s1;      PDE.x{3}.vars = [s1; s2];
| >> PDE.x{1}.dom = [];      PDE.x{2}.dom = [0,3];   PDE.x{3}.dom = [0,3; -1,1];

```

Note here that, for the finite-dimensional state component x_1 , we can indicate that this state does not vary in space by specifying an empty set of variables `vars` and/or specifying an empty spatial domain `dom`.

Having initialized the different state components, we now specify the PDE for each. Here, the first term in the first PDE,

$$\underbrace{-}_{\mathbf{c}} \underbrace{x_1}_{\mathbf{x}}(t),$$

can be easily implemented as

```
>> PDE.x{1}.term{1}.x = 1;
>> PDE.x{1}.term{1}.C = -1;
```

However, for the second term, we have to perform integration, to remove the dependence of the state $\mathbf{x}_2(s_1)$ on the spatial variable s_1 . To declare this, we use the field `I`, which is a cell array of which the number of elements is equal to the number of variables that the state component depends on. Each element of this cell array should specify the spatial domain over which to integrate the state component along the associated spatial direction. Since $\mathbf{x}_2(s_1)$ depends only on s_1 , the field `I` will have just one element, which we set equal to the domain of integration `[0,3]`. That is, we implement the second term in the first PDE,

$$\underbrace{2}_{\mathbf{c}} \underbrace{\int_0^3}_{\mathbf{I}\{1\}} \underbrace{\mathbf{x}_2}_{\mathbf{x}}(t, s_1) ds_1,$$

as

```
>> PDE.x{1}.term{2}.x = 2;
>> PDE.x{1}.term{2}.C = 2;
>> PDE.x{1}.term{2}.I{1} = [0,3];
```

Finally, for the third term, we also integrate the state, but this time along the second spatial variable s_2 on which the state component $\mathbf{x}_3(s_1, s_2)$ depends – meaning we need to specify this integral using the second element `I{2}` of the field `I`. To also remove the dependence of $\mathbf{x}_3(s_1, s_2)$ on s_1 , the state is evaluated at the boundary $s_1 = 3$, which we can specify using the field `loc`. Doing so, we implement the third term in the first PDE,

$$\underbrace{-3}_{\mathbf{c}} \underbrace{\int_{-1}^1}_{\mathbf{I}\{2\}} \underbrace{\mathbf{x}_3}_{\mathbf{x}}(t, \underbrace{3}_{\mathbf{loc}}, s_2) ds_2$$

as

```
>> PDE.x{1}.term{3}.x = 3;           PDE.x{1}.term{3}.C = -3;
>> PDE.x{1}.term{3}.I{2} = [-1,1]; PDE.x{1}.term{3}.loc = [3,s2];
```

Note here that we do not specify a value for `I{1}`. Leaving this field empty, we indicate that no integral should be taken along the direction of the first spatial variable s_1 on which the state component depends.

Now, to specify the second PDE,

$$\dot{\mathbf{x}}_2(t, s_1) = \underbrace{s_1}_C \underbrace{x_1(t)}_x + \overbrace{\partial_{s_1}^1}^D \underbrace{\mathbf{x}_2(t, s_1)}_x - \underbrace{\int_{-1}^1}_{I\{2\}} \underbrace{\mathbf{x}_3(t, s_1, s_2)}_x ds_2$$

we set

```
>> PDE.x{2}.term{1}.x = 1;          PDE.x{2}.term{1}.C = s1;
>> PDE.x{2}.term{2}.x = 2;          PDE.x{2}.term{2}.D = 1
>> PDE.x{2}.term{3}.x = 3;          PDE.x{2}.term{3}.C = -1;
>> PDE.x{2}.term{3}.I{2} = [-1,1];
```

where we note that, since the state component $\mathbf{x}_2(s_1)$ depends only on a single spatial variable s_1 , we need to specify only one order of the derivative $D=1$. Similarly, no order of a derivative should be specified for state component x_1 under any circumstance, as this state does not vary in space at all. Finally, to implement the third PDE,

$$\dot{\mathbf{x}}_3(t, s_1, s_2) = \underbrace{s_1 s_2}_C \underbrace{x_1(t)}_x + \underbrace{\mathbf{x}_2(t, s_1)}_x + \overbrace{\partial_{s_1}^0 \partial_{s_2}^1}^D \underbrace{\mathbf{x}_3(t, s_1, s_2)}_x,$$

we set

```
>> PDE.x{3}.term{1}.x = 1;          PDE.x{3}.term{1}.C = s1*s2;
>> PDE.x{3}.term{2}.x = 2;
>> PDE.x{3}.term{3}.x = 3;          PDE.x{3}.term{3}.D = [0,1];
```

With that, we have specified the PDE, and it remains only to set the boundary conditions. We implement the first condition

$$0 = \underbrace{\mathbf{x}_2(t, \overbrace{3}^{\text{loc}})}_x - \underbrace{x_1(t)}_x,$$

as

```
>> PDE.BC{1}.term{1}.x = 2;          PDE.BC{1}.term{2}.x = 1;
>> PDE.BC{1}.term{1}.loc = 3;        PDE.BC{1}.term{2}.C = -1;
```

where we note that only a single position needs to be indicated in `loc` for state component $\mathbf{x}_2(s_1)$, as this component depends on only one spatial variable. On the other hand, for the second boundary condition,

$$0 = \underbrace{\mathbf{x}_3(t, \overbrace{3, -1}^{\text{loc}})}_x - \underbrace{x_1(t - \underbrace{3}_{\text{delay}})}_x,$$

we do need to specify a spatial position for both s_1 and s_2 in state component \mathbf{x}_3 ,

```

>> PDE.BC{2}.term{1}.x = 3;           PDE.BC{2}.term{2}.x = 1;
>> PDE.BC{2}.term{1}.loc = [3,-1];    PDE.BC{2}.term{2}.C = -1;
>>                                     PDE.BC{2}.term{2}.delay = 3;

```

where we use the field `delay` to indicate that we wish to evaluate x_1 at time $t - 1$. Finally, for the third boundary condition

$$0 = \overbrace{\partial_{s_1}^1 \partial_{s_2}^0}^D \underbrace{\mathbf{x}_3}_{\mathbf{x}}(t, \overbrace{s_1, -1}^{\text{loc}}) - \underbrace{\mathbf{x}_2}_{\mathbf{x}}(t, \overbrace{s_1}^{\text{loc}}),$$

we specify a location at which to evaluate both $\mathbf{x}_1(s_1)$ and $\mathbf{x}_2(s_1, s_2)$,

```

>> PDE.BC{3}.term{1}.x = 3;           PDE.BC{3}.term{2}.x = 2;
>> PDE.BC{3}.term{1}.loc = [s1,-1];  PDE.BC{3}.term{2}.loc = s1;
>> PDE.BC{3}.term{1}.D = [1,0];      PDE.BC{3}.term{2}.C = -1;

```

Note that, although we are not evaluating $\mathbf{x}_2(t, s_1)$ here at any boundary, we do specify its position as `loc=s1`, as it is good practice to always specify the position at which to evaluate (non-ODE) state components when declaring the boundary conditions.

Finally, having declared the full PDE, we run `initialize`,

```

>> PDE = initialize(PDE);

Encountered 3 state components:
x1(t),      of size 1, finite-dimensional;
x2(t,s1),   of size 1, differentiable up to order (1) in variables (s1);
x3(t,s1,s2), of size 1, differentiable up to order (1,1) in variables (s1,s2);

Encountered 3 boundary conditions:
F1(t) = 0,   of size 1;
F2(t) = 0,   of size 1;
F3(t,s1) = 0, of size 1;

```

indicating that the state components and boundary conditions have been properly implemented.

8.2.3 Declaring a System with (Delayed) Inputs and Outputs

Suppose now we want to implement the following system

$$\begin{aligned} \dot{\mathbf{x}}(t, s_1, s_2) &= \partial_{s_1} \partial_{s_2} \mathbf{x}(t, s_1, s_2) + (3 - s_1)(1 - s_2)(s_2 + 1)w(t) & (s_1, s_2) \in [0, 3] \times [-1, 1] \\ z(t) &= 10 \int_0^3 \int_{-1}^1 \mathbf{x}(t, s_1, s_2) ds_2 ds_1 \\ \mathbf{y}_1(t, s_1) &= \int_{-1}^1 \mathbf{x}(t, s_1, s_2) ds_2 + s_1 w(t) \\ \mathbf{y}_2(t, s_2) &= \mathbf{x}(t, 0, s_2) \\ 0 &= \mathbf{x}(t, 0, -1) \\ 0 &= \partial_{s_1} \mathbf{x}(t, s_1, -1) - u_1(t - 0.5) \\ 0 &= \partial_{s_2} \mathbf{x}(t, 0, s_2) - \mathbf{u}_2(t, s_2) \end{aligned}$$

In this system, we have a finite-dimensional disturbance w , finite-dimensional regulated output z , infinite dimensional actuator inputs $\mathbf{u}(t, s_1, s_2) = \begin{bmatrix} u_1(t) \\ \mathbf{u}_2(t, s_2) \end{bmatrix}$, and infinite-dimensional observed outputs $\mathbf{y}(t, s_1, s_2) = \begin{bmatrix} \mathbf{y}_1(t, s_1) \\ \mathbf{y}_2(t, s_2) \end{bmatrix}$, in addition to our infinite dimensional state $\mathbf{x}(t, s_1, s_2)$. Here, we initialize the PDE structure as before as

```
| >> PDE = pde_struct();
```

and then implement the state as

```
| >> pvar s1 s2
| >> PDE.x{1}.vars = [s1,s2];
| >> PDE.x{1}.dom = [0,3; -1,1];
```

Similarly, we indicate the spatial variation of the different inputs and outputs as

```
| >> PDE.z{1}.vars = [];      PDE.w{1}.vars = [];
| >> PDE.y{1}.vars = s1;      PDE.u{1}.vars = [];
| >> PDE.y{1}.dom = [0,3];
| >> PDE.y{2}.vars = s2;      PDE.u{2}.vars = s2;
| >> PDE.y{2}.dom = [-1,1];  PDE.u{2}.dom = [-1,1];
```

where we set $\text{PDE.z}\{1\}.\text{vars} = []$; and $\text{PDE.w}\{1\}.\text{vars} = []$; to indicate that the regulated output and exogenous input are finite-dimensional.

Next, we implement the PDE. The first term here,

$$\overbrace{\partial_{s_1}^1 \partial_{s_2}^1}^{\text{D}} \underbrace{\mathbf{x}}_{\text{x}}(t, s_1, s_2)$$

can be implemented simply as

```
| >> PDE.x{1}.term{1}.x = 1;
| >> PDE.x{1}.term{1}.D = [1,1];
```

where we use the field \mathbf{x} to indicate that the term involves the first (and only) state component. Similarly, to add the term

$$\underbrace{(3 - s_1)(1 - s_2)(s_2 + 1)}_{\text{C}} \underbrace{w}_{\text{w}}(t),$$

we indicate that we are adding an input by using the field \mathbf{w} as

```
| >> PDE.x{1}.term{2}.w = 1;
| >> PDE.x{1}.term{2}.C = (3-s1)*(1-s2)*(s2+1);
```

Note that, in any term, we can only specify one of the fields \mathbf{x} , \mathbf{w} and \mathbf{u} , as any one term cannot involve both a state and an input, or both an exogenous and an actuator input. PIETOOLS will check whether a field \mathbf{x} , \mathbf{w} , or \mathbf{u} is specified, and act accordingly. Note also that, since the inputs may describe arbitrary external forcings, PIETOOLS cannot take the derivative of an input or evaluate it at any particular position. As such, in any term involving an input \mathbf{w} or \mathbf{u} , we cannot specify an order of a derivative (D) or spatial position (loc).

Now, to specify the regulated output

$$z = \underbrace{10}_C \int_{\underbrace{0}_{I\{1\}}}^{\underbrace{3}_{I\{1\}}} \int_{\underbrace{-1}_{I\{2\}}}^{\underbrace{1}_{I\{2\}}} \underbrace{\mathbf{x}}_x(t, s_1, s_2) ds_2 ds_1$$

we simply add one term to `PDE.z{1}` as

```
| >> PDE.z{1}.term{1}.x = 1;           PDE.z{1}.term{1}.C = 10;
| >> PDE.z{1}.term{1}.I{1} = [0,3];    PDE.z{1}.term{1}.I{2} = [-1,1];
```

where we use `I{1}` and `I{2}` to define the domain of integration for the first and second variable on which the state component depends.

Similarly, to specify the equation for y_1 ,

$$y_1(s_1) = \int_{\underbrace{-1}_{I\{2\}}}^{\underbrace{1}_{I\{2\}}} \underbrace{\mathbf{x}}_x(t, s_1, s_2) ds_2 + \underbrace{s_1}_C \underbrace{w}_w(t),$$

we set

```
| >> PDE.y{1}.term{1}.x = 1;           PDE.y{1}.term{2}.w = 1;
| >> PDE.y{1}.term{1}.I{2} = [-1,1];   PDE.y{1}.term{2}.C = s1;
```

this time specifying only an integral along the second spatial direction through `I{2}`.

Finally, we implement the second regulated output equation,

$$y_2(s_2) = \underbrace{\mathbf{x}}_x(t, \underbrace{0}_{loc}, s_2),$$

as

```
| >> PDE.y{2}.term{1}.x = 1;
| >> PDE.y{2}.term{1}.loc = [0,s2];
```

This leaves only the boundary conditions, the first of which

$$0 = \underbrace{\mathbf{x}}_x(t, \underbrace{0}_{loc}, -1),$$

can be easily implemented as

```
| >> PDE.BC{1}.term{1}.x = 1;
| >> PDE.BC{1}.term{1}.loc = [0,-1];
```

Next, for the second boundary condition

$$0 = \overbrace{\partial_{s_1}^1 \partial_{s_2}^0}^D \underbrace{\mathbf{x}}_x(t, s_1, \underbrace{-1}_{loc}) \underbrace{-}_{C} \underbrace{u_1}_u(t - \underbrace{0.5}_{delay})$$

we specify the two terms as

```

>> PDE.BC{2}.term{1}.x = 1;          PDE.BC{2}.term{2}.u = 1;
>> PDE.BC{2}.term{1}.D = [1,0];     PDE.BC{2}.term{2}.C = -1;
>> PDE.BC{2}.term{1}.loc = [s1,-1]; PDE.BC{2}.term{2}.delay = 0.5;

```

Here, although we have to specify the spatial position at which to evaluate the state \mathbf{x} , we do not specify the spatial position of the input \mathbf{u}_1 , as we cannot evaluate an input at any specific position. Similarly, the third boundary condition

$$0 = \overbrace{\partial_{s_1}^0 \partial_{s_2}^1}^D \underbrace{\mathbf{x}}_x(t, \overbrace{0, s_2}^{\text{loc}}) \underbrace{-}_{C} \underbrace{\mathbf{u}_2}_u(t, s_2),$$

can be implemented as

```

>> PDE.BC{3}.term{1}.x = 1;          PDE.BC{3}.term{2}.u = 2;
>> PDE.BC{3}.term{1}.D = [0,1];     PDE.BC{3}.term{2}.C = -1;
>> PDE.BC{3}.term{1}.loc = [0,s2];

```

Having declared the full system, we can then call `initialize`,

```

>> PDE = initialize(PDE);

Encountered 1 state component:
  x(t,s1,s2),    of size 1, differentiable up to order (1,1) in variables (s1,s2);

Encountered 2 actuator inputs:
  u1(t),         of size 1;
  u2(t,s2),      of size 1;

Encountered 1 exogenous input:
  w(t),          of size 1;

Encountered 2 observed outputs:
  y1(t,s1),      of size 1;
  y2(t,s2),      of size 1;

Encountered 1 regulated output:
  z(t),          of size 1;

Encountered 3 boundary conditions:
  F1(t) = 0,     of size 1;
  F2(t,s1) = 0,  of size 1;
  F3(t,s2) = 0,  of size 1;

```

indicating that the state, inputs, and outputs have been correctly declared.

Note:

Although the terms-based input format allows infinite-dimensional inputs and outputs to be specified in the PDE structure (as we did here for \mathbf{u} and \mathbf{y}), PIETOOLS 2022 may not always be able to convert such PDEs to PIEs. This functionality will be included in a later release.

8.2.4 Additional Options

Using the methods from the previous sections, a wide variety of ODE-PDE input-output systems can be implemented in PIETOOLS. However, for those users interested in implementing even more complicated (linear) systems, there are a few additional options for PDEs that PIETOOLS allows. In particular, in defining linear systems, PIETOOLS also allows partial integrals of state variables and inputs to be added to the dynamics, using the field `I`. In addition, PDEs with higher-order temporal derivatives can be declared using the field `tdiff`. Finally, the inherent continuity constraints imposed for PDEs in PIETOOLS can also be adjusted, using the field `diff`.

8.2.4a Partial Integrals and Kernel Functions

In the previous subsections, we showed that an integral $\int_{a_k}^{b_k} \mathbf{x}(s_1, \dots, s_n) ds_k$ of a state \mathbf{x} over $s_k \in [a_k, b_k]$ could be specified in the PDE structure by setting `I{k}=[ak,bk]`. Integrating over the entire domain of the variable $s_k \in [a_k, b_k]$, this integral will remove the dependence of the state \mathbf{x} on the variable s_k . However, rather than performing a full integral, we can also specify a partial integral, integrating over either $[a_k, s_k]$ or $[s_k, b_k]$. For example, to add a term

$$\underbrace{\int_{s_1}^3}_{I\{1\}} \underbrace{\int_{-1}^{s_2}}_{I\{2\}} \underbrace{\mathbf{x}_j}_{\mathbf{x}}(t, \theta_1, \theta_2) d\theta_2 d\theta_1$$

to the PDE of the i th state variable, we can set

```
>> PDE.x{i}.term{1}.x = j;
>> PDE.x{i}.term{1}.I{1} = [s1,3];
>> PDE.x{i}.term{1}.I{2} = [-1,s2];
```

Similarly, a term

$$\underbrace{10s_2}_C \underbrace{\int_{s_2}^1}_{I\{2\}} \underbrace{\partial_{s_1}^1 \partial_{s_2}^0}_{\mathbf{D}} \underbrace{\mathbf{x}_j}_{\mathbf{x}}(t, \underbrace{3}_{\text{loc}}, \theta_2) d\theta_2$$

can be added to the i th boundary condition as

```
>> PDE.BC{i}.term{1}.x = j;
>> PDE.BC{i}.term{1}.D = [1,0];
>> PDE.BC{i}.term{1}.loc = [3,s2];
>> PDE.BC{i}.term{1}.I{2} = [s2,1];
>> PDE.BC{i}.term{1}.C = 10*s2;
```

Note here that, although a dummy variable θ_2 is introduced in the mathematical representation of the integral $\int_{s_2}^1 \mathbf{x}(t, 3, \theta_2) d\theta$, the location `loc=[3,s2]` at which we evaluate the state can still be specified using the primary variable s_2 . However, for the factor $C(s_2) = 10s_2$ with which to multiply the state, the use of the primary variable s_2 ensures that this factor will not be integrated itself. Suppose now that we wish for C to act as a kernel instead, adding e.g. a term

$$\int_{s_2}^1 10\theta_2 \partial_{s_1}^1 \mathbf{x}_j(t, 3, \theta_2) d\theta_2$$

To implement this term, we first have to specify dummy variables for the state component. This can be done using the second column of the field $\mathbf{x}\{j\}.\mathbf{vars}$, specifying dummy variables (θ_1, θ_2) for the j th state component as

```
>> pvar s1 s2 theta1 theta2
>> PDE.x{j}.vars = [s1,theta1; s2,theta2];
>> PDE.x{j}.dom = [0,3; -1,1]
```

Here, each dummy variable θ_i is linked to a particular spatial variable s_i by specifying the two variables in the same **row** of the field **vars**. The domain of the dummy variable will then be the same as that of its associated primary variable, as specified in the corresponding row of **dom**.

Having specified the dummy variables, we can now add the term

$$\underbrace{\int_{s_2}^1}_{I\{2\}} \underbrace{10\theta_2}_{C} \overbrace{\partial_{s_1}^1 \partial_{s_2}^0}^D \underbrace{\mathbf{x}_j}_{x}(t, \overbrace{3, \theta_2}^{\text{loc}}) d\theta_2$$

to the i th boundary condition by specifying

```
>> PDE.BC{i}.term{1}.x = j;
>> PDE.BC{i}.term{1}.D = [1,0];
>> PDE.BC{i}.term{1}.loc = [3,theta2];    or    PDE.BC{i}.term{1}.loc = [3,s2];
>> PDE.BC{i}.term{1}.I{2} = [s2,1];
>> PDE.BC{i}.term{1}.C = 10*theta2;
```

Here, we can choose to set either $\text{loc}=[3, \text{theta2}]$ or $\text{loc}=[3, s2]$, as the field $I\{2\}=[s2, 1]$ ensures that both options will be recognized as integrating the state over the domain $[s_2, 1]$. However, dummy variables can only be used (in the fields **loc** and **C**) if integration is performed along the associated spatial direction. For example, a term

$$\underbrace{\int_0^{s_1}}_{I\{1\}} \underbrace{s_2(s_1 - \theta_1)}_C \overbrace{\partial_{s_1}^1 \partial_{s_2}^2}^D \underbrace{\mathbf{x}_j}_{x}(t, \overbrace{\theta_1, -1}^{\text{loc}}) d\theta_1$$

could also be added to e.g. the PDE of the i th state component as

```
>> PDE.BC{i}.term{1}.x = j;
>> PDE.BC{i}.term{1}.D = [1,2];
>> PDE.BC{i}.term{1}.loc = [theta1,-1];    or    PDE.BC{i}.term{1}.loc = [s1,-1];
>> PDE.BC{i}.term{1}.I{1} = [0,s1];
>> PDE.BC{i}.term{1}.C = s2*(s1-theta1);
```

in which case θ_2 cannot be used, since no integration is performed along the second spatial direction. However, dummy variables can also be used to implement kernels for full integrals, adding e.g. a term

$$\underbrace{\int_0^{s_1}}_{I\{1\}} \underbrace{\int_{-1}^1}_{I\{2\}} \underbrace{\theta_2(s_1 - \theta_1)}_C \overbrace{\partial_{s_1}^1 \partial_{s_2}^2}^D \underbrace{\mathbf{x}_j}_{x}(t, \theta_1, \theta_2) d\theta_2 d\theta_1$$

as

```

>> PDE.BC{i}.term{1}.x = j;
>> PDE.BC{i}.term{1}.D = [1,2];
>> PDE.BC{i}.term{1}.I{1} = [0,s1];
>> PDE.BC{i}.term{1}.I{2} = [-1,1];
>> PDE.BC{i}.term{1}.C = theta2*(s1-theta1);

```

where in this case, no location needs to be specified, as the state is not evaluated at any boundary.

Note:

In specifying the domain of integration for a variable $s_k \in [a_k, b_k]$, only the full domain $I\{k\}=[ak, bk]$ or the partial domains $I\{k\}=[ak, sk]$ or $I\{k\}=[sk, bk]$ are supported. Specifying any other domain for the integral will produce an error.

8.2.4b Higher Order Temporal Derivatives

In the previous sections, only PDEs involving first-order temporal derivatives have been implemented, using the element of $\mathbf{x}\{i\}.\text{term}$ to declare the terms in the equation for $\dot{\mathbf{x}}_i$. However, PDEs involving higher order temporal derivatives can also be specified, by adjusting the value of $\mathbf{x}\{i\}.\text{tdiff}$. For example, suppose we want to declare the wave equation

$$\begin{aligned} \ddot{\mathbf{x}}(t, s) &= \partial_s^2 \mathbf{x}(t, s), & s \in [0, 1], \quad t \geq 0, \\ \mathbf{x}(t, 0) &= 0, \quad \mathbf{x}(t, 1) &= 0, \end{aligned}$$

where $\mathbf{x} \in L_2[0, 1]$. We initialize the PDE structure and the state \mathbf{x} as before,

```

>> PDE = pde_struct();
>> pvar s
>> PDE.x{1}.vars = s;
>> PDE.x{1}.dom = [0,1];

```

Then, before specifying the terms in the PDE for this state component $\mathbf{x}_1 = \mathbf{x}$, we indicate that this PDE concerns a second order temporal derivative of \mathbf{x} , using the field `tdiff`:

```

>> PDE.x{1}.tdiff = 2;

```

Note that if no field `tdiff` is specified (as in the previous sections), its value will default to 1. Having specified the desired order of the temporal derivative, we can declare the PDE and BCs as

```

>> PDE.x{1}.term{1}.D = 2;
>> PDE.BC{1}.term{1}.loc = 0;
>> PDE.BC{2}.term{1}.loc = 1;

```

and we can initialize

```

>> PDE = initialize(PDE);

```

```

Encountered 1 state component:

```

```

  x(t,s),    of size 1, differentiable up to order (2) in variables (s);

```

```

Encountered 2 boundary conditions:

```

```

  F1(t) = 0, of size 1;

```

```
| F2(t) = 0, of size 1;
```

We note that, although higher order temporal derivatives can be specified in the PDE format, the PIE representation cannot represent such higher order temporal derivatives. Therefore, when calling `convert(PDE, 'pie')`, the higher order temporal derivatives will first be expanded, expressing the PDE as a system involving only first order temporal derivatives. This is done using the function `expand_tderivatives`, which can also be called to manually expand the delays

```
| >> PDE = expand_tderivatives(PDE)
|
| Added 1 state component:
|     x2(t,s) := (d/dt) x1(t,s)
```

Introducing the new state component $x_2(t, s) = \dot{x}_1(t, s)$, PIETOOLS then expresses the system as a PDE involving only first-order temporal derivatives as

$$\begin{aligned} \dot{\mathbf{x}}_1(t, s) &= \mathbf{x}_2(t, s), & s \in [0, 1], \quad t \geq 0, \\ \dot{\mathbf{x}}_2(t, s) &= \partial_s^2 \mathbf{x}_1(t, s), \\ \mathbf{x}_1(t, 0) &= 0, \quad \mathbf{x}_1(t, 1) = 0. \end{aligned}$$

Note:

Expanding higher order temporal derivatives using `expand_tderivatives`, no boundary conditions will be imposed upon the newly introduced state components (such as $x_2(t, s) = \dot{x}_1(t, s)$). Boundary conditions can be imposed upon the new states by manually declaring them in terms-based format. Results from e.g. stability tests may vary depending on whether or not boundary conditions are imposed upon the newly introduced state components.

8.2.4c Continuity Constraints

In any PDE, there are inherent continuity constraints imposed upon the PDE state, requiring the state to be differentiable in each spatial variable up to some order. For example, for a PDE

$$\dot{\mathbf{x}}(t, s_1, s_2) = \partial_{s_1}^2 \mathbf{x}(t, s_1, s_2) + \partial_{s_2} \mathbf{x}(t, s_1, s_2)$$

the state \mathbf{x} must be at least second order differentiable with respect to s_1 , and first order differentiable with respect to s_2 . Specifying this system in PIETOOLS, a continuity constraint will automatically be imposed upon the PDE state as

$$\partial_{s_1}^2 \partial_{s_2} \mathbf{x}(t, s_1, s_2) \in L_2.$$

Accordingly, a well-posed solution to the PDE will require (at least) two boundary conditions to be specified along boundaries in the first spatial dimension (e.g. $s_1 = 0$ and/or $s_1 = 3$), and one boundary condition to be specified along boundaries in the second spatial dimension (e.g. $s_2 = -1$ or $s_2 = 1$). Moreover, at the boundaries along the first spatial direction, we can only evaluate first order derivatives of the state with respect to s_1 , and at the boundaries along the

second spatial direction, then we cannot differentiate the state with respect to s_2 at all. That is, where the boundary conditions

$$0 = \partial_{s_1} \mathbf{x}(t, 0, s_2) \qquad 0 = \mathbf{x}(t, s_1, -1)$$

are permitted, the boundary conditions

$$0 = \partial_{s_1}^2 \mathbf{x}(t, 0, s_2) \qquad 0 = \partial_{s_2} \mathbf{x}(t, s_1, -1)$$

are prohibited.

In some cases, however, it may be desirable for a state component to be differentiable up to a greater order than the maximal order of the derivative appearing in the PDE. In PIETOOLS, this can be indicated in the PDE structure through the optional field `PDE.x{i}.diff`, allowing the order of spatial differentiability of the i th PDE state component to be explicitly specified as

```
| >> PDE.x{i}.diff = [dmax1, dmax2];
```

where `dmax1` denotes the maximal order of differentiability in s_1 , and `dmax2` the maximal order of differentiability in s_2 . Note that this order of differentiability cannot be smaller than the maximal order of the spatial derivatives of the i th PDE state in any term of the system. That is, specifying a term such as

```
| >> PDE.x{k}.term{j}.x = i; PDE.x{k}.term{j}.D = [dmax1+1, dmax2];
```

PIETOOLS will automatically update the order of differentiability of the i th state component to `PDE.x{i}.diff=[dmax1+1, dmax2]`. Furthermore, the number of columns of the field `diff` should always match the number of rows of the field `vars`, specifying an order of differentiability for each spatial variable on which the state component depends. For example, if a state component $\mathbf{x}_j(s_2)$ varies only in a single variable s_2 , only a single order of differentiability can be specified as

```
| >> PDE.x{j}.vars = s2; PDE.x{j}.diff = dmax2;
```

8.3 The Command Line Parser format Format

In this subsection, we explain how defining and manipulating ODE-PDE systems revolves around two main classes: `state` and `sys`. Furthermore, we will also specify valid modes of manipulating these objects in MATLAB and potential caveats while using these objects.

8.3.1 state class objects

All symbols used to define a systems are either `polynomial` type (part of SOSTOOLS) or `state` type (part of PIETOOLS). Here, we will focus on `state` class objects and methods defined for such objects. First, any `state` class object has the following properties that can be freely accessed (but **should not** be modified directly).

`state`

This class has the following properties:

1. **type**: Type of variable; It is a cell array of strings that can take values in `{'ode', 'pde'}`

```
'out' , 'in'}
```

2. `veclength`: Positive integer
3. `var`: Cell array of polynomial row vectors (Multipoly class object)
4. `diff_order`: Cell array of non-negative integers (same size as `var`)

The first independent variable stored in each row of the `state.var` cell structure is always the time variable `t`. Spatial variables are stored in location 2 and on-wards. For example,

```
>> X = state('pde'); x = state('ode');
>> X.var

ans =
     [ t, s]

>> x.var

ans =
     [ t ]
```

Differentiation information is stored as a cell array where the cell structure has the same size as `state.var` with non-negative integers specifying order of differentiation w.r.t. the independent variable based on the location. For the above example, we have

```
>> X.diff_order

ans =
     [ 0, 0]

>> y = diff(X,s,2);
>> y.diff_order

ans =
     [ 0, 2]
```

Note, user can indeed edit these properties directly by assignment. For example, the code

```
>> x = state('pde');
>> x.diff_order = [0,2];
```

defines the symbol `x` as a function $x(t,s)$, and converts it to the second derivative $\partial_s^2 x(t,s)$. This is same as the code

```
>> x = state('pde');
>> x = diff(x,s,2);
```

Since, this permanently changes `x` to its second spatial derivative in the workspace, such direct manipulation of the properties should be avoided at all costs.

Declaring/initializing state variables The initialization function `state()` takes two input arguments (both are optional):

- `type`: The argument is reserved to specification of the type of the state object (defaults to 'ode', if not specified)

- `veclength`: The size of the vector-valued state (defaults to one, if not specified)

```
| d = state('pde',3);
```

Alternatively, multiple states can be defined collectively using the command shown below, however, all such states will default to the type 'ode' and length 1.

```
| state a b c;
```

Operations on state class objects All of the following operations should give us a **terms** (an internal class that cannot be accessed or modified by users) class object which is defined by some PI operator times a vector of states. Operators/functions that are used to manipulate **state** objects are:

1. addition: `x+y` or `x-y`
2. multiplication: `K*x`
3. vertical concatenation: `[x;y]`
4. differentiation: `diff(x,s,3)`
5. integration: `int(x,s,[0,s])`
6. substitution: `subs(x,s,0)`

Caveats in operations on state class objects While manipulation of state class objects, the users must adhere the following rules stated in the table 8.1. All the operations listed in the table are invalid.

Addition of time derivatives is not allowed, since that usually leads to a descriptor dynamical PDE system which is not supported by PIETOOLS. For example, consider the following PDE.

$$\begin{aligned}\dot{\mathbf{x}}(t) + \dot{\mathbf{y}}(t) &= \partial_s^2 \mathbf{x}(t, s) \\ 2\dot{\mathbf{y}}(t) &= 5\partial_s^2 \mathbf{y}(t, s).\end{aligned}$$

This PDE cannot be implemented directly using the command line parser. Since, the left hand side of the equation has a coefficient different from identity, the user needs to first separate it as

$$\begin{aligned}\dot{\mathbf{x}}(t) &= \partial_s^2 \mathbf{x}(t, s) - 2.5\partial_s^2 \mathbf{y}(t, s) \\ \dot{\mathbf{y}}(t) &= 2.5\partial_s^2 \mathbf{y}(t, s).\end{aligned}$$

Now, we can define this PDE using the following code:

```
\begin{matlab}
>> pvar ~t ~s ~theta; \\
>> x = state('pde'); ~y = state('pde'); \\
>> odepde= sys(); \\
>> odepde = addequation(odepde, diff(x,t)-diff(x,s,2)-2.5*diff(y,s,2)); \\
>> odepde = addequation(odepde, diff(y,t)-2.5*diff(y,s,2));
\end{matlab}
```

Likewise, we do not permit adding outputs with outputs, outputs with time derivatives, or right multiplication which also lead to descriptor type systems. Coupling on left hand side of these equations must be manually resolved before defining the PDE in PIETOOLS.

Other limitations to note are, PIETOOLS does not support temporal-spatial mixed derivatives, integration in time, and evaluation of functions at specific time or inside the spatial domain. For example, for a state $x(t, s)$ with $s \in [0, 1]$ we cannot find $x(t = 2, s)$ or $x(t, s = 0.5)$. x can only be evaluated at the boundary $s = 0$ or $s = 1$.

Note:

While PIETOOLS terms format (introduced in Section 8.2) supports higher order temporal derivatives, the command line parser does not support it currently.

8.3.2 sys class objects

`sys`

This class has following accessible properties:

- **equation:** stores all the equations added to the system object in a column vector where every row is an equation with zero on the right hand side (i.e., `row(i)=0` for every `i`)
- **type:** type of the system (currently supports 'pde' and 'pie')

Operation type	Incorrect or ‘not-permitted’ operations
Addition	<ul style="list-style-type: none"> ✘ Adding two time derivatives: <code>diff(x,t)+diff(x,t)</code> ✘ Adding two outputs: <code>z1+z2</code> ✘ Adding time derivative and output: <code>diff(x,t)+z</code>
Multiplication	<ul style="list-style-type: none"> ✘ Multiplying two states: <code>x*x</code> ✘ Multiplying non-identity with time derivative/output: <code>2*diff(x,t)</code> or <code>-1*z</code> ✘ Right multiplication: <code>x*3</code>
Differentiation	<ul style="list-style-type: none"> ✘ Higher order time derivatives: <code>diff(x,t,2)</code> ✘ Mixed derivatives of space and time: <code>diff(diff(x,t),s,2)</code>
Substitution	<ul style="list-style-type: none"> ✘ Substituting a double for time variable: <code>subs(x,t,2)</code> ✘ Substituting positive time delay: <code>subs(x,t,t+5)</code> ✘ Substitution values other than <code>pvar</code> variable or boundary values
Integration	<ul style="list-style-type: none"> ✘ Integration of time variable: <code>int(x,t,[0,5])</code> ✘ both limits being non-numeric: <code>int(x,s,[theta,eta])</code> ✘ limit same as variable of integration: <code>int(x,s,[s,1])</code>
Concatenation	<ul style="list-style-type: none"> ✘ Horizontal concatenation: <code>[x,x]</code> ✘ Blank spaces in vertical concatenation: <code>[x + y; z]</code>

Table 8.1: This table lists all the invalid forms of operations on `state` class objects. The left column specifies the type of operation whereas the right column lists the operations that are **INVALID** for that ‘type’ of operation.

- `params`: either a `pde_struct` or `pie_struct` object
- `dom`: a 1×2 vector double (value of first element must be strictly smaller than that of second element)
- Other hidden properties:
 1. `states`: a vector of all states, inputs, outputs appearing in the equation property
 2. `ControlledInputs`: A vector with length same as the states property with 0 or 1 value. This vector specifies whether a state is a controlled input or not.
 3. `ObservedOutputs`: A vector with length same as the states property with 0 or 1 value. This specifies whether a state is an observed output or not.

sys class methods Methods used to modify a `sys()` object are listed below.

- `addequation`: adds an equation to the `obj.equation` property; syntax `addequation(obj, eqn)`
- `removeequation`: removes equation in row `i` from the `obj.equation` property; syntax `removeequation(obj,i)`
- `setControl`: sets a chosen state `x` as a control input; syntax `setControl(obj,x)`
- `setObserve`: sets a chosen state `x` as an observed output; syntax `setObserve(obj,x)`
- `removeControl`: removes a chosen state `x` from the set of control inputs; syntax `removeControl(obj,x)`
- `removeObserve`: removes a chosen state `x` from the set of observed outputs; syntax `removeObserve(obj,x)`
- `getParams`: parses symbolic equations from `obj.equation` property to get `pde_struct` object which is stored in `obj.params`; syntax `getParams(obj)`
- `convert`: converts `obj.params` from `pde_struct` to `pie_struct` object; syntax `convert(obj,'pie')`

WARNING:

`sys` class object properties should not be modified directly (unless you know what you are doing); Use the methods provided above.

Chapter 9

Batch Input Formats for Time-Delay Systems

In Chapter 4, we showed how time-delay systems (TDSs) can be implemented as delay differential equations (DDEs) in PIETOOLS. In that chapter, we further hinted at the fact that PIETOOLS also allows TDSs to be declared in two alternative representations: as neutral type systems (NDSs) and as differential difference equations (DDFs). In this chapter, we will provide more details on how to work with such NDS and DDF systems in PIETOOLS. In particular, in Section 9.1, we recall the DDE representation, and show what NDS and DDF systems look like, and how systems of each type can be declared in PIETOOLS. In Section 9.2, we then show how NDS and DDE systems can be converted to the DDF representation in PIETOOLS, and how each type of TDS can be converted to a PIE.

9.1 Representing Systems with Delay

In this section, we show how time-delay systems in DDE, NDS and DDF representation can be declared in PIETOOLS, focusing on DDE systems in Subsection 9.1.1, NDS systems in Subsection 9.1.2, and DDF systems in Subsection 9.1.3. For more information on how to declare systems in DDE representation in PIETOOLS, we refer to Section 4.3

9.1.1 Input of Delay Differential Equations

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

$$\begin{aligned} \begin{bmatrix} \dot{x}(t) \\ z(t) \\ y(t) \end{bmatrix} &= \begin{bmatrix} A_0 & B_1 & B_2 \\ C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{bmatrix} \begin{bmatrix} x(t) \\ w(t) \\ u(t) \end{bmatrix} + \sum_{i=1}^K \begin{bmatrix} A_i & B_{1i} & B_{2i} \\ C_{1i} & D_{11i} & D_{12i} \\ C_{2i} & D_{21i} & D_{22i} \end{bmatrix} \begin{bmatrix} x(t - \tau_i) \\ w(t - \tau_i) \\ u(t - \tau_i) \end{bmatrix} \\ &+ \sum_{i=1}^K \int_{-\tau_i}^0 \begin{bmatrix} A_{di}(s) & B_{1di}(s) & B_{2di}(s) \\ C_{1di}(s) & D_{11di}(s) & D_{12di}(s) \\ C_{2di}(s) & D_{21di}(s) & D_{22di}(s) \end{bmatrix} \begin{bmatrix} x(t+s) \\ w(t+s) \\ u(t+s) \end{bmatrix} ds \end{aligned} \quad (9.1)$$

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.Ai{1} = -1;
>> DDE.C1i{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 9.1.

9.1.1a 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.

9.1.2 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:

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

ODE Terms:					
Eqn. (9.1)	DDE.	Eqn. (9.1)	DDE.	Eqn. (9.1)	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. (9.1)	DDE.	Eqn. (9.1)	DDE.	Eqn. (9.1)	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. (9.1)	DDE.	Eqn. (9.1)	DDE.	Eqn. (9.1)	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 9.1: Equivalent names of Matlab elements of the DDE structure terms for terms in Eqn. (9.1). 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].

$$+ \sum_{i=1}^K \int_{-\tau_i}^0 \begin{bmatrix} A_{di}(s) & B_{1di}(s) & B_{2di}(s) & E_{di}(s) \\ C_{1di}(s) & D_{11di}(s) & D_{12di}(s) & E_{1di}(s) \\ C_{2di}(s) & D_{21di}(s) & D_{22di}(s) & E_{2di}(s) \end{bmatrix} \begin{bmatrix} x(t+s) \\ w(t+s) \\ u(t+s) \\ \dot{x}(t+s) \end{bmatrix} ds \quad (9.2)$$

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.

9.1.2a 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.

9.1.3 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.

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

$$v(t) = \sum_{i=1}^K C_{vi} r_i(t - \tau_i) + \sum_{i=1}^K \int_{-\tau_i}^0 C_{vdi}(s) r_i(t + s) ds. \quad (9.3)$$

As for a DDE or NDS, each of the non-zero parameters in Eqn. (9.3) 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 9.3.

9.1.3a 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. Note that, to make the code robust, all PIETOOLS conversion utilities perform this step internally.

9.2 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.

9.2.1 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 naïvely converts a DDF to an equivalent PIE.

9.2.1a 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 DDF and enter the command

```
| >> DDF = minimize_PIETOOLS_DDF(DDF);
```

9.2.1b 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_DDF(DDF,'pie');
```

9.2.2 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 by first computing a minimized DDF representation using `minimize_PIETOOLS_DDE2DDF`, followed possibly by converting this DDF representation to a PIE. Both steps are included in the function `convert_PIETOOLS_DDE`, allowing the minimal DDF representation and PIE representation of a DDE structure DDE to be computed as

```
| >> [DDF, PIE] = convert_PIETOOLS_DDE(DDE);
```

Here, the function `convert_PIETOOLS_DDE` computes the minimal DDF representation by calling `minimal_PIETOOLS_DDE2DDF`, which uses the SVD to eliminate unused delay channels in the DDF - resulting in a much more compact representation of the same system. As such, the minimal DDF representation can be computed by calling `minimal_PIETOOLS_DDE2DDF` directly as

```
| >> DDF = minimal_PIETOOLS_DDE2DDF(DDF);
```

or by calling `convert_PIETOOLS_DDE` with a second argument `'ddf'`

```
| >> DDF = convert_PIETOOLS_DDE(DDE,'ddf');
```

Similarly, if only the PIE representation is desired, the user can also call

```
| >> PIE = convert_PIETOOLS_DDE(DDE,'pie');
```

though the procedure for computing the PIE will still involve computing the DDF representation first.

9.2.2a 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_legacy(DDE);
```

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

9.2.3 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 is involves 3 steps: direct conversion of the NDS to a DDF; constructing a minimal representation of the resulting DDF using `minimize_PIETOOLS_DDF`; and conversion of the reduced DDF to a PIE. These three steps have been combined into a single function `convert_PIETOOLS_NDS`, computing the DDF representation, minimizing this representation, and converting this representation to a PIE. All three resulting structures can be returned by calling

```
| >> [DDF_max, DDF, PIE] = convert_PIETOOLS_NDS(NDS);
```

where now `DDF_max` corresponds to the non-minimized DDF representation of NDS, and `DDF` corresponds to the minimized representation. If the user only want to compute this DDF representation, it is computationally cheaper to call the function with only two outputs,

```
| >> [DDF_max, DDF] = convert_PIETOOLS_NDS(NDS);
```

or to call the function with a second argument `'ddf'`,

```
| >> [DDF] = convert_PIETOOLS_NDS(NDS, 'ddf');
```

Similarly, if only the non-minimized DDF representation is desired, the function should called with a single output,

```
| >> DDF_max = convert_PIETOOLS_NDS(NDS, 'ddf_max');
```

or with a second argument `'ddf_max'`,

```
| >> DDF_max = convert_PIETOOLS_NDS(NDS, 'ddf_max');
```

It is also possible to pass an argument `'pie'`, calling

```
| >> PIE = convert_PIETOOLS_NDS(NDS, 'pie');
```

returning only the PIE representation, even though the DDF representations will also be computed.

ODE Terms:							
Eqn. (9.2)	NDS.	Eqn. (9.2)	NDS.	Eqn. (9.2)	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. (9.2)	NDS.	Eqn. (9.2)	NDS.	Eqn. (9.2)	NDS.	Eqn. (9.2)	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. (9.2)	NDS.	Eqn. (9.2)	NDS.	Eqn. (9.2)	NDS.	Eqn. (9.2)	NDS.
A_{di}	Adi{i}	B_{1di}	B1di{i}	B_{2di}	B2di{i}	E_{di}	Edi{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 9.2: Equivalent names of Matlab elements of the NDS structure terms for terms in Eqn. (9.2). 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. (9.3)	DDF.	Eqn. (9.3)	DDF.	Eqn. (9.3)	DDF.	Eqn. (9.3)	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. (9.3)	DDF.						
C_{vi}	Cvi{i}						

Distributed Delay Terms: May be functions of pvar s							
Eqn. (9.3)	DDF.						
$C_{vdi}(s)$	Cvdi{i}						

Table 9.3: Equivalent names of Matlab elements of the DDF structure terms for terms in Eqn. (9.3). 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 10

Operations on PI Operators in PIETOOLS: `opvar` and `dopvar`

In Chapter 5, we showed how PI operators could be declared as `opvar` and `opvar2d` objects in PIETOOLS. In Chapter 7, we showed how the similar class of `dopvar` (and `dopvar2d`) objects can be used to represent PI operator decision variables in convex optimization programs. In this Chapter, we detail some features of these `opvar` and `dopvar` classes, showing how standard operations on PI operators can be easily performed using the `opvar` classes in PIETOOLS. In particular, we first recall the structure of `opvar` and `opvar2d` objects in Section 10.1, also showing how such objects can be declared in PIETOOLS. In Section 10.2, we then show algebraic operations such as addition of `opvar` objects can be performed in PIETOOLS, after which we show how matrix operations such as concatenation can be performed on `opvar` objects in Section 10.3. Finally, in Section 10.4, we outline a few additional operations for `opvar` objects. For more information on the theory behind these operations, we refer to Appendix A, as well as papers such as [].

Note

Unless stated otherwise, the operations on `opvar` objects presented in the following sections can also be performed on `opvar2d`, `dopvar` and `dopvar2d` class objects. To reduce notation, these operations will be illustrated only for `opvar` class objects.

10.1 Declaring `opvar` and `dopvar` Objects

In this section, we briefly recall how `opvar` objects are structured, and how they represent PI operators in 1D. We also briefly introduce the `dopvar` class, showing how such objects can be declared in a similar manner to `opvar` objects. For more information on the `opvar2d` structure for PI operators in 2D, we refer to Chapter 3.

10.1.1 The `opvar` Class

In PIETOOLS, 4-PI operators are represented by `opvar` objects, which are structures with 8 accessible fields. In particular, letting $\mathcal{T} : \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a, b] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a, b] \end{bmatrix}$ be a 4-PI operator of the

form

$$(\mathcal{T}\mathbf{x})(s) = \begin{bmatrix} Px_0 + \int_a^b Q_1(s)\mathbf{x}_1(s)ds \\ Q_2(s)x_0 + R_0(s)\mathbf{x}_1(s) + \int_a^s R_1(s,\theta)\mathbf{x}_1(\theta)d\theta + \int_s^b R_2(s,\theta)\mathbf{x}_1(\theta)d\theta \end{bmatrix} \quad (10.1)$$

for $\mathbf{x} = \begin{bmatrix} x_0 \\ \mathbf{x}_1 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \end{bmatrix}$, we can declare \mathcal{T} as an `opvar` object `T` with the following fields

opvar fields

<code>dim</code>	<code>[m0,n0; m1,n1]</code>	2×2 array of type <code>double</code> specifying the dimensions of the function spaces $\begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a,b] \end{bmatrix}$ and $\begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \end{bmatrix}$ the operator maps to and from;
<code>var1</code>	<code>s</code>	1×1 <code>pvar</code> (<code>polynomial class</code>) object specifying the spatial variable s ;
<code>var2</code>	<code>theta</code>	1×1 <code>pvar</code> (<code>polynomial class</code>) object specifying the dummy variable θ ;
<code>I</code>	<code>[a,b]</code>	1×2 array of type <code>double</code> , specifying the interval $[a, b]$ on which the spatial variables s and θ exist;
<code>P</code>	<code>P</code>	$m_0 \times n_0$ array of type <code>double</code> or <code>polynomial</code> defining the matrix P ;
<code>Q1</code>	<code>Q1</code>	$m_0 \times n_1$ array of type <code>double</code> or <code>polynomial</code> defining the function $Q_1(s)$;
<code>Q2</code>	<code>Q2</code>	$m_1 \times n_0$ array of type <code>double</code> or <code>polynomial</code> defining the function $Q_2(s)$;
<code>R.R0</code>	<code>R0</code>	$m_1 \times n_1$ array of type <code>double</code> or <code>polynomial</code> defining the function $R_0(s)$;
<code>R.R1</code>	<code>R1</code>	$m_1 \times n_1$ array of type <code>double</code> or <code>polynomial</code> defining the function $R_1(s, \theta)$;
<code>R.R2</code>	<code>R2</code>	$m_1 \times n_1$ array of type <code>double</code> or <code>polynomial</code> defining the function $R_2(s, \theta)$;

As an example, suppose we want to declare the 4-PI operator $\mathcal{T} : \begin{bmatrix} \mathbb{R}^2 \\ L_2^3[2,3] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^3 \\ L_2[2,3] \end{bmatrix}$ defined as

$$(\mathcal{T}\mathbf{x})(r) = \begin{bmatrix} \overbrace{\begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 3 & 4 \end{bmatrix} x_0}^P + \int_2^3 \overbrace{\begin{bmatrix} r^2 & 0 & 0 \\ 3 & r^3 & 0 \\ 0 & r+2*r^2 & 0 \end{bmatrix} \mathbf{x}_1(r) dr}^{Q_1(r)} \\ \underbrace{\begin{bmatrix} -5r & 6 \end{bmatrix} x_0}_{Q_2(s)} + \int_2^r \underbrace{\begin{bmatrix} r & 2\nu & 3(r-\nu) \end{bmatrix} \mathbf{x}_1(\nu) d\nu}_{R_1(r,\nu)} \end{bmatrix}, \quad r \in [2, 3],$$

for any $\mathbf{x} = \begin{bmatrix} x_0 \\ \mathbf{x}_1 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^2 \\ L_2^3[2,3] \end{bmatrix}$. To declare this operator, we first initialize an `opvar` object `T`, using the syntax

```
>> opvar T
ans =
     [] | []
-----
     [] | ans.R

ans.R =
     [] | [] | []
```

This command creates an empty `opvar` object `T` with all dimensions 0. Consequently, the parameters `P`, `Qi`, `Ri` are initialized to 0×0 matrices. Since we know the operator \mathcal{T} to map $\begin{bmatrix} \mathbb{R}^2 \\ L_2^3[2,3] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^3 \\ L_2[2,3] \end{bmatrix}$, we can specify the desired dimension of the `opvar` object `T` using the command

```

>> T.dim = [3 2; 1 3]
T =
      [0,0] | [0,0,0]
      [0,0] | [0,0,0]
      [0,0] | [0,0,0]
      -----
      [0,0] | T.R

T.R =
      [0,0,0] | [0,0,0] | [0,0,0]

```

Here, by assigning a value to `dim`, the parameters are adjusted to zero matrices of appropriate dimensions. We note that, this command is not strictly necessary, as the dimensions of `T` will also be automatically adjusted once we specify the values of the parameters.

Next, we assign the interval $[2, 3]$ on which the functions $\mathbf{x}_1 \in L_2^3[0, 3]$ are defined, by setting the field `I` as

```

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

```

Since the parameters defining \mathcal{T} also depend on $r, \nu \in [2, 3]$, we have to assign these variables as well. For this, we represent them by `pvar` objects `r` and `nu`, and set the values of `T.var1` and `T.var2` as

```

>> pvar r nu
>> T.var1 = r;      T.var2 = nu;

```

Note that, if the parameters `Qi` and `R.Ri` are constant, there is no need to declare the variables `var1` or `var2`, in which case these fields will default to `var1=s` and `var2=theta`. Having declared the variables, we finally set the values of the parameters, by assigning them to the appropriate fields of `T`

```

>> T.P = [1,0; 0,2; 3,4];
>> T.Q1 = [r^2, 0, 0; 3, r^3, 0; 0, r+2*r^2, 0];
>> T.Q2 = [-5*r, 6];
>> T.R.R1 = [r, 2*nu, 3*r-nu]
T =
      [1,0] | [r^2,0,0]
      [0,2] | [3,r^3,0]
      [3,4] | [0,2*r^2+r,0]
      -----
      [-5*r,6] | T.R

T.R =
      [0,0,0] | [r,2*nu,-nu+3*r] | [0,0,0]

```

Note

`dim` is dependent on size of the 6 parameters `P`, `Qi` and `R.Ri`. Modifying those parameters automatically changes the value stored in `dim` property. If the dimensions of the parameters are incompatible, `dim` will store `Nan` as its value to alert the user about the discrepancy.

10.1.2 dpvar objects and the dopvar class

In addition to polynomial functions, polynomial decision variables can also be declared in PIETOOLS. Such decision variables are used in polynomial optimization programs, optimizing over the values of the coefficients defining these polynomials. To declare such polynomial decision variables, we use the `dpvar` class, extending the `polynomial` class to represent polynomials with decision variables. For example, to represent a variable quadratic polynomial

$$p(c_0, c_1, c_2; x) = c_0 + c_1x + c_2x^2,$$

with unknown coefficient $\{c_0, c_1, c_2\}$, we use

```
>> pvar x
>> dpvar c0 c1 c2
>> p = c0 + c1*x + c2*x^2
p =
c0 + c1*x + c2*x^2
```

Here, the second line decision variables c_0 , c_1 and c_2 , and the third line uses these decision variables to declare the decision variable polynomial $p(c_0, c_1, c_2; x)$ as a `dpvar` object `p`. Crucially, this variable is affine in the coefficients c_0 , c_1 and c_2 , as `dpvar` objects can only be used to represent decision variable polynomials that are affine with respect to their coefficients. This is because PIETOOLS cannot tackle polynomial optimization programs that are nonlinear in the decision variables, and accordingly, the `dpvar` structure has been built to exploit the linearity of the decision variables to minimize computational effort.

Using polynomial decision variables, we can also define PI operator decision variables, defining the parameters Q_1 through R_2 by polynomial decision variables rather than polynomials. For example, suppose we have an operator $\mathcal{D} : L_2^2[0, 1] \rightarrow L_2^2[0, 1]$ defined as

$$(\mathcal{D}\mathbf{x})(s) = \underbrace{\begin{bmatrix} c_1 & c_2s \\ c_2s & c_3s^2 \end{bmatrix}}_{R_0(s)} \mathbf{x}(s) + \int_s^1 \underbrace{\begin{bmatrix} c_4s^2 & c_5s\theta \\ -c_6s\theta & c_7\theta^2 \end{bmatrix}}_{R_2(s,\theta)} \mathbf{x}(\theta)d\theta, \quad s \in [0, 1]$$

for $\mathbf{x} \in L_2^2[0, 1]$, where c_1 through c_7 are unknown coefficients. Then, we can declare the parameters R_1 and R_2 as `dpvar` class objects by calling

```
>> pvar s th
>> dpvar c1 c2 c3 c4 c5 c6 c7
>> R0 = [c1, c2*s; c2*s, c3*s^2];
>> R2 = [c4, c5*s*th; c6*s*th, c7*th^2];
```

Then, we can declare \mathcal{D} as a `dopvar` object `D`. Such `dopvar` objects have a structure identical to that of `opvar` objects, with the only difference being that the parameters `P` through `R.R2` can be declared as `dpvar` objects. Accordingly, we declare the `dopvar` object `D` in a similar manner as we would and `opvar` objects:

```
>> dopvar D;
>> D.I = [0,1];
>> D.var1 = s;      D.var2 = th;
>> D.R.R0 = R0;    D.R.R1 = R2
D =
[] | []
```

```

-----
[] | D.R
D.R =
      [c1,c2*s] | [0,0] |      [c4,c5*s*th]
      [c2*s,c3*s^2] | [0,0] | [c6*s*th,c7*th^2]

```

10.2 Algebraic Operations on opvar Objects

In this section, we go over various methods that help in manipulating and handling of `opvar` objects in PIETOOLS. In particular, in Subsection 10.2.1, we show how the sum of two PI operators can be computed in PIETOOLS, followed by the composition of PI operators in Subsection 10.2.2. In Subsection 10.2.3, we then show how to take the adjoint of a PI operator, and finally, in Subsection 10.2.4, we show how a numerical inverse of a PI operator can be computed. To illustrate each of these operations, we use the PI operators $\mathcal{A}, \mathcal{B} : \left[\begin{smallmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{smallmatrix} \right] \rightarrow \left[\begin{smallmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{smallmatrix} \right]$ defined as

$$\begin{aligned}
 (\mathcal{A}\mathbf{x})(s) &= \left[\begin{array}{l} \begin{bmatrix} 1 & 0 \\ 2 & -1 \end{bmatrix} x_0 + \int_{-1}^1 \begin{bmatrix} 1-s \\ s+1 \end{bmatrix} \mathbf{x}_1(s) ds \\ \begin{bmatrix} 10s & -1 \end{bmatrix} x_0 + 2\mathbf{x}_1(s) + \int_{-1}^s (s-\theta)\mathbf{x}_1(\theta)d\theta + \int_s^1 (s-\theta)\mathbf{x}_1(\theta)d\theta \end{array} \right], \\
 (\mathcal{B}\mathbf{x})(s) &= \left[\begin{array}{l} \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix} x_0 \\ \begin{bmatrix} 5s & -s \end{bmatrix} x_0 + s^2\mathbf{x}_1(s) + \int_s^1 \theta\mathbf{x}_1(\theta)d\theta \end{array} \right], \quad s \in [-1, 1],
 \end{aligned} \tag{10.2}$$

for $\mathbf{x} = \begin{bmatrix} x_0 \\ \mathbf{x}_1 \end{bmatrix} \in \left[\begin{smallmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{smallmatrix} \right]$. We declare these operators as

```

>> pvar s th
>> opvar A B;
>> A.I = [-1,1];      B.I = [-1,1];
>> A.var1 = s;        B.var1 = s;
>> A.var2 = th;       B.var2 = th;
>> A.P = [1,0;2,-1];  B.P = [1,0;0,3];
>> A.Q1 = [1-s;s+1];  B.Q1 = [1-s;s+1];
>> A.Q2 = [10*s,-1];  B.Q2 = [5*s,-s];
>> A.R.R0 = 2;        B.R.R0 = s^2;
>> A.R.R1 = (s-th);   B.R.R1 = (s-th);
>> A.R.R2 = (s-th);   B.R.R2 = th;

```

10.2.1 Addition (A+B)

`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 dimensions (`A.dim=B.dim`), domains (`A.I=B.I`), and variables (`A.var1=B.var1`). 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.

Example Adding the opvar objects A and B corresponding to operators \mathcal{A}, \mathcal{B} defined as in Equation (10.2), we find

```

>> C = A+B
C =
          [2,0] | [-s+1]
          [2,2] | [s+1]
-----
[15*s,-s-1] | C.R

C.R =
[s^2+2] | [s-th] | [s]

```

suggesting that, for $\mathbf{x} = \begin{bmatrix} x_0 \\ \mathbf{x}_1 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{bmatrix}$,

$$(\mathcal{A}\mathbf{x})(s) + (\mathcal{B}\mathbf{x})(s) = \begin{bmatrix} \begin{bmatrix} 2 & 0 \\ 2 & 2 \end{bmatrix} x_0 + \int_{-1}^1 \begin{bmatrix} 1-s \\ s+1 \end{bmatrix} \mathbf{x}_1(s) ds \\ [15s \ -s-1] x_0 + (s^2 + 2)\mathbf{x}_1(s) + \int_{-1}^s (s - \theta)\mathbf{x}_1(\theta) d\theta + \int_s^1 s\mathbf{x}_1(\theta) d\theta \end{bmatrix}.$$

10.2.2 Multiplication (A*B)

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 the same domains ($A.I=A.B$), the same variables ($A.var1=B.var1$ and $A.var2=B.var2$), and the output dimension of B must match the input dimension of A ($A.dim(:,2)=B.dim(:,1)$). 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.

Example Composing the opvar objects A and B corresponding to operators \mathcal{A}, \mathcal{B} defined as in Equation (10.2), we find

```

>> C = A*B
C =
          [-2.3333,0.66667] | [-1.5*s^3+2*s^2+1.5*s]
          [5.3333,-3.6667] | [1.5*s^3+2*s^2+0.5*s]
-----
[20*s-3.3333,-2*s-2.3333] | C.R

C.R =
[2*s^2] | [2*s*th^2-1.5*th^3+s*th+0.5*th] | [2*s*th^2-1.5*th^3+s*th+2.5*th]

```

suggesting that, for $\mathbf{x} = \begin{bmatrix} x_0 \\ \mathbf{x}_1 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{bmatrix}$,

$$\left(\mathcal{A}(\mathcal{B}\mathbf{x}) \right) (s) = \begin{bmatrix} \begin{bmatrix} -2\frac{1}{3} & \frac{2}{3} \\ 5\frac{1}{3} & -3\frac{2}{3} \end{bmatrix} x_0 + \int_{-1}^1 \begin{bmatrix} -\frac{1}{2}s^3+2s^2+\frac{1}{2}s \\ \frac{1}{2}s^3+2s^2+\frac{1}{2}s \end{bmatrix} \mathbf{x}_1(s) ds \\ \begin{bmatrix} 20s-3\frac{1}{3} & -2s-2\frac{1}{3} \end{bmatrix} x_0 + 2s^2\mathbf{x}_1(s) + \int_{-1}^s (2s\theta^2 - 1\frac{1}{2}\theta^3 + s\theta + \frac{1}{2}\theta)\mathbf{x}_1(\theta) d\theta \\ + \int_s^1 (2s\theta^2 - 1\frac{1}{2}\theta^3 + s\theta + 2\frac{1}{2}\theta)\mathbf{x}_1(\theta) d\theta \end{bmatrix}.$$

Note

Although `dopvar` objects can be multiplied with `opvar` objects and vice versa, producing a `dopvar` object in both cases, it is not possible to compute the composition of two `dopvar` objects. This is because `dopvar` objects depend linearly (affinely) on the decision variables, and the composition of two `dopvar` objects would require taking the product of decision variables. Similarly for `dopvar2d` objects.

10.2.3 Adjoint (A')

The adjoint of an `opvar` object `A` can be calculated using the command

```
| >> A'
```

For an operator $\mathcal{A} : RL^{n_0, n_1}[a, b] \rightarrow RL^{m_0, m_1}[a, b]$, the adjoint $\mathcal{A}^* : RL^{m_0, m_1}[a, b] \rightarrow RL^{n_0, n_1}[a, b]$ will be such that, for any $\mathbf{x} \in RL^{n_0, n_1}[a, b]$ and $\mathbf{y} \in RL^{m_0, m_1}[a, b]$,

$$\langle \mathcal{A}\mathbf{x}, \mathbf{y} \rangle_{RL} = \langle \mathbf{x}, \mathcal{A}^*\mathbf{y} \rangle_{RL},$$

where for $\mathbf{x} = \begin{bmatrix} x_0 \\ \mathbf{x}_1 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a, b] \end{bmatrix} = RL^{n_0, n_1}[a, b]$ and $\mathbf{y} = \begin{bmatrix} y_0 \\ \mathbf{y}_1 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a, b] \end{bmatrix} = RL^{m_0, m_1}[a, b]$,

$$\langle \mathbf{x}, \mathbf{y} \rangle_{RL} := \langle x_0, y_0 \rangle + \langle \mathbf{x}_1, \mathbf{y}_1 \rangle_{L_2} = x_0^T y_0 + \int_a^b [\mathbf{x}_1(s)]^T \mathbf{y}_1(s) ds$$

Example Computing the adjoint of the `opvar` object `A` corresponding to operator \mathcal{A} defined as in Equation (10.2), we find

```
| >> AT = A'
| AT =
|           [1,2] | [10*s]
|           [0,-1] | [-1]
|           -----
|           [-s+1,s+1] | AT.R
|
| AT.R =
|           [2] | [-s+th] | [-s+th]
```

suggesting that, for $\mathbf{x} = \begin{bmatrix} x_0 \\ \mathbf{x}_1 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{bmatrix}$,

$$(\mathcal{A}^*\mathbf{x})(s) = \begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & -1 \end{bmatrix} x_0 + \int_{-1}^1 \begin{bmatrix} 10s \\ -1 \end{bmatrix} \mathbf{x}_1(s) ds \\ \begin{bmatrix} 1-s & s+1 \end{bmatrix} x_0 + 2\mathbf{x}_1(s) + \int_{-1}^1 (\theta - s)\mathbf{x}_1(\theta) d\theta + \int_s^1 (\theta - s)\mathbf{x}_1(\theta) d\theta \end{bmatrix}.$$

10.2.4 Inverse (inv_opvar(A))

The inverse of an `opvar` object `A` can be numerically calculated, using the function

```
| >> inv_opvar(A)
```

See Lemma 9 for details on Inversion formulae.

Example Computing the inverse of the opvar object A corresponding to operator \mathcal{A} defined as in Equation (10.2), we find

```
>> Ainv = inv_opvar(A)
Ainv =
          [-0.2,-0.4] | [ 0.3*s + 0.2]
          [1.2,0.4] | [-0.3*s - 0.7]
-----
[ 0.3*s+0.7,1.35*s+0.65] | AT.R

Ainv.R =
[0.5] | [-1.2*s*th-0.675*s-0.3*th-0.575] | [-1.2*s*th-0.675*s-0.3*th-0.575]
```

suggesting that, for $\mathbf{x} = \begin{bmatrix} x_0 \\ \mathbf{x}_1 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{bmatrix}$

$$(\mathcal{A}^{-1}\mathbf{x})(s) = \begin{bmatrix} \begin{bmatrix} -0.2 & -0.4 \\ 1.2 & 0.4 \end{bmatrix} x_0 + \int_{-1}^1 \begin{bmatrix} 0.3s+0.2 \\ -0.3s-0.7 \end{bmatrix} \mathbf{x}_1(s) ds \\ \begin{bmatrix} 0.3s+0.7 & 1.35s+0.65 \end{bmatrix} x_0 + 0.5\mathbf{x}_1(s) + \int_{-1}^1 (-1.2s\theta - 0.675s - 0.3\theta - 0.575)\mathbf{x}_1(\theta) d\theta \\ + \int_s^1 (-1.2s\theta - 0.675s - 0.3\theta - 0.575)\mathbf{x}_1(\theta) d\theta \end{bmatrix}.$$

Note

An inverse function has not been defined for opvar2d, dopvar, or dopvar2d objects.

10.3 Matrix Operations on opvar Objects

In this section, we show how matrix operations on opvar objects can be performed. In particular, in Subsection 10.3.1 we show how opvar] objects can be concatenated, and in Subsection 10.3.2, we show how desired rows and columns of opvar objects can be extracted. Although we explain these operations only for opvar objects, they can also be applied to dopvar, opvar2d, and dopvar2d objects. For the purpose of illustration, we once more use the 4-PI operators $\mathcal{A}, \mathcal{B} : \begin{bmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{bmatrix} \rightarrow \begin{bmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{bmatrix}$ defined in Equation (10.2), represented in PIETOOLS by the opvar objects A and B,

```
>> A
A =
          [1,0] | [-s+1]
          [2,-1] | [s+1]
-----
[10*s,-1] | A.R

A.R =
[2] | [s-th] | [s-th]

>> B
B =
          [1,0] | [0]
          [0,3] | [0]
-----
[5*s,-s] | B.R
```

```
B.R =
      [s^2] | [0] | [th]
```

10.3.1 Concatenation ([A,B])

Just like matrices, multiple `opvar` objects can be concatenated, provided the dimensions match. In particular, two `opvar` objects `A` and `B` can be horizontally or vertically concatenated by respectively using the command

```
>> [A B] % for horizontal concatenation
>> [A; B] % for vertical concatenation
```

Note that concatenation of `opvar` objects is allowed only if their spatial domain is the same (`A.I=B.I`), and the variables involved in each are identical (`A.var1=B.var1` and `A.var2=B.var2`). Moreover, `A` and `B` can be horizontally concatenated only if they have the same row dimensions (`A.dim(:,1)=B.dim(:,2)`), and they can be concatenated vertically only if they have the same column dimensions (`A.dim(:,2)=B.dim(:,2)`).

Example Horizontally concatenating the `opvar` objects `A` and `B` corresponding to operators \mathcal{A}, \mathcal{B} defined as in Equation (10.2), we find

```
>> C = [A, B]
C =
           [1,0,1,0] | [-s+1,0]
           [2,-1,0,3] | [s+1,0]
-----
      [10*s,-1,5*s,-s] | C.R
C.R =
      [2,s^2] | [s-th,0] | [s-th,th]
```

Note that, since $\mathcal{A}, \mathcal{B} : \left[L_2^{\mathbb{R}^2}[-1,1] \right] \rightarrow \left[L_2^{\mathbb{R}^2}[-1,1] \right]$, we have $\mathcal{C} : \left[L_2^{\mathbb{R}^4}[-1,1] \right] \rightarrow \left[L_2^{\mathbb{R}^2}[-1,1] \right]$, **not** $\mathcal{C} :$

$\left[\begin{array}{c} \mathbb{R} \\ L_2[-1,1] \\ \mathbb{R} \\ L_2[-1,1] \end{array} \right] \rightarrow \left[L_2^{\mathbb{R}^2}[-1,1] \right]$. Similarly, taking the vertical concatenation,

```
>> C = [A; B]
C =
           [1,0] | [-s+1]
           [2,-1] | [s+1]
           [1,0] | [0]
           [0,3] | [0]
-----
      [10*s,-1] | C.R
      [5*s,-s] |
C.R =
           [2] | [s-th] | [s-th]
      [s^2] |      [0] |      [th]
```

the resulting operator \mathcal{C} will map $\left[L_2^{\mathbb{R}^2}[-1,1] \right] \rightarrow \left[L_2^{\mathbb{R}^4}[-1,1] \right]$, as PIETOOLS cannot represent operators mapping e.g. $\left[L_2^{\mathbb{R}^2}[-1,1] \right] \rightarrow \begin{bmatrix} \mathbb{R} \\ L_2^{\mathbb{R}^2}[-1,1] \\ L_2^{\mathbb{R}^2}[-1,1] \end{bmatrix}$.

10.3.2 Subs-indexing ($A(i, j)$)

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 from matrix indexing due to presence of multiple components. These components can be visualized as being stacked as in a matrix:

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

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.

Example Extracting row 3 and columns 1, and 3 of the `opvar` object A corresponding to the operator \mathcal{A}, \mathcal{B} defined as in Equation (10.2), we find

```
| >> C = A(3, [1,3])
| C =
|          [] | []
|          -----
|          [10*s] | C.R
| C.R =
|          [2] | [s-th] | [s-th]
```

10.4 Additional Methods for `opvar` Objects

There are some additional 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>A==B</code>	The function checks whether the variables, domain, dimensions and parameters of the operators <code>A</code> and <code>B</code> are equal, and returns a binary value 1 if this is the case, or 0 if it is not. The function can also be used to check if an operator <code>A</code> has all parameters equal to zero operator by calling <code>A==0</code> .
<code>isvalid(P)</code>	The function returns a logical value. 0 if everything is in order, 1 if the object has incompatible dimensions, 2 if property <code>P</code> is not a matrix, 3 if properties <code>Q1</code> , <code>Q2</code> or <code>R0</code> are not polynomials in s , 4 if properties <code>R1</code> or <code>R2</code> are not polynomials in s and θ . For <code>opvar2d</code> objects, returns boolean value <code>true</code> or <code>false</code> to indicate if <code>P</code> is appropriate or not.
<code>degbalance(T)</code>	Estimates polynomial degrees needed to create an <code>opvar</code> object <code>Q</code> associated to a positive PI operator $Q \succeq 0$ in <code>poslpivar</code> , such that <code>T=Q</code> has at least one solution.
<code>getdeg(T)</code>	Returns highest and lowest degree of s and θ in the components of the <code>opvar</code> object <code>T</code> .
<code>rand_opvar(dim, deg)</code>	Creates a random <code>opvar</code> object of specified dimensions <code>dim</code> and polynomial degrees <code>deg</code> .
<code>show(T,opts)</code>	Alternative display format for <code>opvar</code> objects with optional argument to omit selected properties from display output. Not defined for <code>opvar2d</code> objects.
<code>opvar_postest(T)</code>	Numerically test for sign definiteness of <code>T</code> . Returns -1 if negative definite, 0 if indefinite and 1 if positive definite. Use <code>opvar_postest_2d</code> for <code>opvar2d</code> objects.
<code>diff_opvar(T)</code>	Returns composition of derivative operator with <code>opvar</code> <code>T</code> as described in Lem. 10. Use <code>diff(T)</code> for <code>opvar2d</code> objects.

Part III

Examples and Applications

Chapter 11

PIETOOLS Demonstrations

In this Chapter, we illustrate several applications of PIE simulation and LPI programming, and how each of these problems can be implemented in PIETOOLS. Each of these problems has also been implemented as a DEMO file in PIETOOLS, which can be found in the PIETOOLS_demos directory.

11.1 DEMO 1: Simple Stability, Simulation and Control Problem

See Chapter 2 for a description.

11.2 DEMO 2: Estimating the Volterra Operator Norm

The Volterra integral operator $\mathcal{T} : L_2[0, 1] \rightarrow L_2[0, 1]$ is perhaps the simplest example of a 3-PI operator, defined as

$$(\mathcal{T}\mathbf{x})(s) = \int_0^s \mathbf{x}(\theta)d\theta, \quad s \in [0, 1],$$

for any $\mathbf{x} \in L_2[0, 1]$. In PIETOOLS, this operator can be easily declared as

```
| a=0;    b=1;  
| opvar Top;  
| Top.R.R1 = 1;    Top.I = [a,b];
```

Then, an upper bound on the norm of this operator can be computed by solving the LPI

$$\begin{aligned} & \min_{\gamma \geq 0} \gamma, \\ \text{s.t.} \quad & \mathcal{T}^*\mathcal{T} \leq \gamma, \end{aligned}$$

so that case $C = \sqrt{\gamma}$ for any feasible value γ is an upper bound on the norm of \mathcal{T} . This optimization problem is an LPI, that can be declared and solved in PIETOOLS as


```

% First, define dvar gam and set up an optimization problem
vars = [Top.var1;Top.var2];      % Free vars in optimization problem
dvar gam;                        % Decision var in optimization problem
prob = sosprogram(vars,gam);

% Next, set gam as objective function min{gam}
prob = sossetobj(prob, gam);

% Then, enforce the constraint Top'*Top-gam<=0
opts.psatz = 1;                  % Allow Top'*Top-gam>0 outside of [a,b]
prob = lpi_ineq(prob,-(Top'*Top-gam),opts); % lpi_ineq(prob,Q) enforces Q>=0

% Finally, solve and retrieve the solution
prob = sossolve(prob);
operator_norm = sqrt(double(sosgetsol(prob,gam)));

```

This code can also be run by calling “volterra_operator_norm_DEMO”. We obtain an upper bound $C = 0.68698$ on the induced norm $\|\mathcal{T}\|$ of the Volterra operator. The exact value of the induced norm of this operator is known to be equal to $\|\mathcal{T}\| = \frac{2}{\pi} = 0.6366\dots$

11.3 DEMO 3: Solving the Poincaré Inequality

In a one dimensional domain $\Omega = [a, b]$, the Poincaré inequality imposes a bound on the norm of a function $x(s)$ in terms of the spatial derivative $\partial_s x(s)$ of this function,

$$\|x\|_{L_2} \leq C \|\partial_s x\|_{L_2}, \quad \forall x \in W_1[a, b],$$

where

$$W_1[a, b] := \{x \in L_2[a, b] \mid \partial_s x \in L_2[a, b], x(a) = x(b) = 0\}.$$

In PIETOOLS, we can find a value C that satisfies this inequality, by solving the optimization problem

$$\begin{aligned} & \min_{\gamma \geq 0} \gamma, \\ \text{s.t.} \quad & \langle x, x \rangle_{L_2} - \gamma \langle \partial_s x, \partial_s x \rangle_{L_2} \leq 0, \quad \forall x \in W_1[a, b] \end{aligned}$$

in which case $C = \sqrt{\gamma}$ satisfies the Poincaré inequality. To declare this problem, we first note that we can represent x and $\partial_s x$ in terms of a fundamental state $\partial_s^2 x \in L_2[a, b]$, which is free of the boundary conditions and continuity constraints imposed upon x and $\partial_s x$. In particular, we first declare a PDE

$$\begin{aligned} \dot{x}(t, s) &= \partial_s x(t, s), & s &\in [a, b], \\ x(t, a) &= x(t, b) = 0, \end{aligned}$$

as a `pde_struct` object by calling

```

%% Initialize the PDE structure and spatial variable s in [a,b]
pvar s theta;
pde_struct PDE;
a = 0;      b = 1;

%% Declare the state variables x(t,s)
PDE.x{1}.vars = s;
PDE.x{1}.dom = [a,b];
PDE.x{1}.diff = 2;      % Let x be second order differentiable wrt s.

%% Declare the PDE \dot{x}(t,s) = \partial_s x(t,s)
PDE.x{1}.term{1}.D = 1; % Order of the derivative wrt s

%% Declare the boundary conditions x(t,a) = x(t,b) = 0
PDE.BC{1}.term{1}.loc = a; % Evaluate x at s=a
PDE.BC{2}.term{1}.loc = b; % Evaluate x at s=b

%% Initialize the system
PDE = initialize(PDE);

```

Here, we explicitly indicate that the function x must be second order differentiable, as we wish to derive a PIE representation in terms of the fundamental state $x_f := \partial_s^2 x$. To obtain this PIE representation, we call `convert`,

```

PIE = convert(PDE,'pie');
H2 = PIE.T; % H2 x_{ss} = x
H1 = PIE.A; % H1 x_{ss} = x_{s}

```

arriving at an equivalent representation of the PDE as

$$\mathcal{H}_2 \dot{x}_f(t, s) = \mathcal{H}_1 x_f(t, s), \quad s \in [a, b].$$

In this representation, the fundamental state $x_f := \partial_s^2 x$ is free of any boundary conditions and continuity constraints. Moreover, we note that

$$\mathcal{H}_2 x_f(t, s) = x(t, s), \quad \text{and,} \quad \mathcal{H}_1 x_f(t, s) = \partial_s x(t, s).$$

As such, the Poincaré inequality optimization problem can be equivalently represented as

$$\min_{\gamma \geq 0} \gamma, \quad \text{s.t.} \quad \langle \mathcal{H}_2 x_f, \mathcal{H}_2 x_f \rangle - \gamma \langle \mathcal{H}_1 x_f, \mathcal{H}_1 x_f \rangle \leq 0, \quad \forall x_f \in L_2[a, b]$$

giving rise to an LPI

$$\min_{\gamma \geq 0} \gamma, \quad \text{s.t.} \quad \mathcal{H}_2^* \mathcal{H}_2 - \gamma \mathcal{H}_1^* \mathcal{H}_1 \leq 0.$$

We declare and solve this LPI in PIETOOLS as

```

%% First, define dpvar gam and set up an optimization problem
dpvar gam;
vars = [H2.var1; H2.var2];
prob = sosprogram(vars,gam);

%% Set gam as objective function to minimize
prob = sossetobj(prob, gam);

```

```

%% Set up the constraint H2'*H2-gam H1'*H1<=0
opts.psatz = 1;      % allow H2'*H2 > gam H1'*H1 outside of [a,b]
prob = lpi_ineq(prob,-(H2'*H2-gam*H1'*H1),opts);

% Solve and retrieve the solution
prob = sossolve(prob);
poincare_constant = sqrt(double(sosgetsol(prob,gam)));

```

This code can also be run calling the function “poincare_inequality_DEMO”, arriving at a constant $C = 0.42664$ that satisfies the Poincaré inequality on the domain $[a, b] = [0, 1]$. On this domain, a minimal value of C is known to be $C = \frac{1}{\pi} = 0.31831\dots$

11.4 DEMO 4: Finding an Optimal Stability Parameter

We consider a reaction-diffusion equation on an interval $[a, b]$,

$$\dot{\mathbf{x}}(t, s) = \lambda \mathbf{x}(t, s) + \partial_s^2 \mathbf{x}(t, s), \quad s \in [a, b], \mathbf{x}(t, a) = \mathbf{x}(t, b) = 0,$$

for some $\lambda \in \mathbb{R}$. On the interval $[a, b] = [0, 1]$, this system is known to be stable whenever $\lambda \leq \pi^2 = 9.8696\dots$. In PIETOOLS, we can numerically estimate this limit using the function `stability_PIETOOLS`. In particular, we may equivalently represent the PDE as a PIE of the form

$$(\mathcal{T}\dot{\mathbf{x}}_f)(t, s) = (\mathcal{A}(\lambda)\mathbf{x}_f)(t, s), \quad s \in [a, b],$$

where $\mathbf{x}_f(t, s) := \partial_s^2 \mathbf{x}(t, s)$. Then, a maximal value of λ for which the PIE is stable can be estimated by solving the optimization problem

$$\begin{aligned} & \max_{\lambda \in \mathbb{R}, \mathcal{P} \in \Pi} \lambda, \\ \text{s.t.} \quad & \mathcal{P} \succ 0, \\ & \mathcal{T}^* \mathcal{P} \mathcal{A}(\lambda) + \mathcal{A}^*(\lambda) \mathcal{P} \mathcal{T} \preceq 0. \end{aligned}$$

Unfortunately, both λ and \mathcal{P} are decision variables in this optimization program, and so the product $\mathcal{P}\mathcal{A}(\lambda)$ is not linear in the decision variables. As such, this problem cannot be directly implemented as a convex optimization program. However, for any fixed value of λ , stability of the PIE can be verified by testing feasibility of the LPI

$$\mathcal{P} \succ 0, \quad \mathcal{T}^* \mathcal{P} \mathcal{A}(\lambda) + \mathcal{A}^*(\lambda) \mathcal{P} \mathcal{T} \preceq 0,$$

an optimization program that has already been implemented in `stability_PIETOOLS`. Therefore, to estimate an upper bound on the value of λ for which the PDE is stable, we can test stability for given values of λ , and perform bisection over some domain $\lambda \in [\lambda_{\min}, \lambda_{\max}]$ to find an optimal value. For our demonstration, we use $\lambda_{\min} = 0$ and $\lambda_{\max} = 20$, testing stability for 8 values of λ between these upper and lower bounds:

```

%% Set bisection limits for lam.
lam_min = 0;      lam_max = 20;
lam = 0.5*(lam_min + lam_max);
n_iters = 8;

```

We initialize a PDE structure in Terms-format, so that we may easily update the value of λ :

```

%%% Initialize a PDE structure.
a = 0;  b = 1;
pvar s
pde_struct PDE;
PDE.x{1}.vars = s;
PDE.x{1}.dom = [a,b];

% Set the PDE  $\dot{x}(t,s) = \text{lam} * x(t,s) + x_{\text{ss}}(t,s)$ ;
PDE.x{1}.term{1}.C = lam;
PDE.x{1}.term{2}.D = 2;

% Set the BCs  $x(t,a) = x(t,b) = 0$ ;
PDE.BC{1}.term{1}.loc = a;
PDE.BC{2}.term{1}.loc = b;

```

Finally, we fix settings for the LPI program, using `veryheavy` settings to achieve relatively high accuracy

```

%%% Initialize settings for solving the LPI
settings = lpisettings('veryheavy');
if strcmp(settings.sos_opts.solver,'sedumi')
    settings.sos_opts.params.fid = 0;  % Suppress output in command window
end

```

Having initialized everything, we perform bisection on the value of λ . In particular, for a given value of $\lambda \in [\lambda_{\min}, \lambda_{\max}]$, we update the PDE structure `PDE`, compute the associated PIE structure `PIE`, and test stability of this PIE using `stability_PIETOOLS`. Then, we check the value of `feasratio` in the output optimization program structure, which should be close to 1 if the LPI was successfully solved, and thus the system was found to be stable.

- If the system is stable, then it is stable for any value of λ smaller than the value `lam` used in the test. As such, we update the value of $\lambda_{\min} \leftarrow \lambda$, and repeat the test with a greater value $\lambda = \frac{1}{2}(\lambda_{\min} + \lambda_{\max})$.
- If stability could not be verified, then stability can also not be verified for any value of λ greater than the value `lam` used in the test. As such, we update the value of $\lambda_{\max} \leftarrow \lambda$, and repeat the test with a greater value $\lambda = \frac{1}{2}(\lambda_{\min} + \lambda_{\max})$.

This algorithm can be implemented as

```

%%% Perform bisection on the value of lam
for iter = 1:n_iters
    % Update the value of lam in the PDE.
    PDE.x{1}.term{1}.C = lam;

    % Update the PIE.
    PIE = convert(PDE,'pie');

    % Re-run the stability test.
    prog = PIETOOLS_stability(PIE,settings);

    % Check if the system is stable

```

```

if prog.solinfo.info.dinf || prog.solinfo.info.pinf ...
    || abs(prog.solinfo.info.feasratio - 1)>0.3
    % Stability cannot be verified, decreasing the value of lam...
    lam_max = lam;
    lam = 0.5*(lam_min + lam_max);
else
    % The system is stable, trying a larger value of lam...
    lam_success = lam;
    lam_min = lam;
    lam = 0.5*(lam_min + lam_max);
end
end
end

```

This code can also be called using “stability_parameter_bisection_DEMO”. Running this demo, we find that stability can be verified whenever $\lambda \leq 9.8438$.

11.5 DEMO 5: Constructing and Simulating an Optimal Estimator

We consider a reaction-diffusion PDE, with an observed output y ,

$$\begin{aligned}
 & \dot{\mathbf{x}}(t, s) = \partial_s^2 \mathbf{x}(t, s) + 4\mathbf{x}(t, s) + w(t), & s \in [0, 1], \\
 \text{with BCs} & \quad 0 = \mathbf{x}(t, 0) = \partial_s \mathbf{x}(t, 1), \\
 \text{and outputs} & \quad z(t) = \int_0^1 \mathbf{x}(t, s) ds + w(t), \\
 & \quad y(t) = \mathbf{x}(t, 1).
 \end{aligned} \tag{11.1}$$

This PDE can be easily declared in PIETOOLS as

```

pvar s t
PDE = sys();
x = state('pde');    w = state('in');
y = state('out');    z = state('out');
eqs = [diff(x,t) == diff(x,s,2) + 4*x + w;
       z == int(x,s,[0,1]) + w;
       y == subs(x,s,1);
       subs(x,s,0) == 0;
       subs(diff(x,s),s,1) == 0];
PDE = addequation(PDE,eqs);
PDE = setObserve(PDE,y);

```

at which point an equivalent PIE representation can be derived by calling `convert`:

```

PIE = convert(PDE,'pie');    PIE = PIE.params;
T = PIE.T;
A = PIE.A;    C1 = PIE.C1;    C2 = PIE.C2;
B1 = PIE.B1;    D11 = PIE.D11;    D21 = PIE.D21;

```

Then, the PDE (11.1) can be equivalently represented by a PIE

$$(\mathcal{T}\dot{\mathbf{v}})(t, s) = (\mathcal{A}\mathbf{v})(t, s) + (\mathcal{B}_1 w)(t, s), \quad s \in [0, 1]$$

$$\begin{aligned}
z(t) &= (\mathcal{C}_1 \mathbf{v})(t) + (\mathcal{D}_{11} w)(t), \\
y(t) &= (\mathcal{C}_2 \mathbf{v})(t) + (\mathcal{D}_{12} w)(t),
\end{aligned} \tag{11.2}$$

where we define $\mathbf{v} := \partial_s^2 \mathbf{x}$, and where $\mathbf{x} = \mathcal{T} \mathbf{v}$.

We consider the problem of designing an optimal estimator for the PIE (11.2). In particular, we construct an estimator of the form

$$\begin{aligned}
\mathcal{T} \hat{\mathbf{v}}(t) &= \mathcal{A} \hat{\mathbf{v}}(t) + \mathcal{L}(y(t) - \hat{y}(t)), \\
\hat{z}(t) &= \mathcal{C}_1 \hat{\mathbf{v}}(t), \\
\hat{y}(t) &= \mathcal{C}_2 \hat{\mathbf{v}}(t),
\end{aligned} \tag{11.3}$$

so that the error $\mathbf{e}(t, s) := \hat{\mathbf{v}}(t, s) - \mathbf{v}(t, s)$ in the state and $\tilde{z}(t) := \hat{z}(t) - z(t)$ in the output satisfy

$$\begin{aligned}
\mathcal{T} \dot{\mathbf{e}}(t) &= (\mathcal{A} - \mathcal{L} \mathcal{C}_2) \mathbf{e}(t) - (\mathcal{B}_1 + \mathcal{L} \mathcal{D}_{21}) w(t) \\
\tilde{z}(t) &= \mathcal{C}_1 \mathbf{e}(t) - \mathcal{D}_{11} w(t).
\end{aligned} \tag{11.4}$$

The goal of H_∞ -optimal estimation, then, is to determine a value for the observer operator \mathcal{L} that minimizes the gain $\gamma := \frac{\|\tilde{z}\|_{L_2}}{\|w\|_{L_2}}$ from disturbances w to errors \tilde{z} in the output. See also Section 13.2. To construct such an operator, we solve the LPI,

$$\begin{aligned}
&\min_{\gamma, \mathcal{P}, \mathcal{Z}} \gamma \\
&\mathcal{P} \succ 0, \quad Q := \begin{bmatrix} -\gamma I & -\mathcal{D}_{11}^\top & -(\mathcal{P} \mathcal{B}_1 + \mathcal{Z} \mathcal{D}_{21})^* \mathcal{T} \\ (\cdot)^* & -\gamma I & \mathcal{C}_1 \\ (\cdot)^* & (\cdot)^* & (\mathcal{P} \mathcal{A} + \mathcal{Z} \mathcal{C}_2)^* \mathcal{T} + (\cdot)^* \end{bmatrix} \preceq 0
\end{aligned} \tag{11.5}$$

so that, for any solution $(\gamma, \mathcal{P}, \mathcal{Z})$ to this problem, letting $\mathcal{L} := \mathcal{P}^{-1} \mathcal{Z}$, the estimation error will satisfy $\|\hat{z} - z\| \leq \gamma \|w\|$.

In Chapter 7, we showed in detail how the LPI (11.5) can be declared for a PIE of the form (11.2). The code declaring and solving this LPI can also be found in the demo file “Hinf_Optimal_Estimator_DEMO”. We will therefore not discuss this implementation here. Instead, we compute an optimal value of the observer operator using one of the pre-defined executive files, as

```

| settings = lpisettings('heavy');
| [prog, Lval, gam_val] = PIETOOLS_Hinf_estimator(PIE, settings);

```

returning a value `Lval` of the operator \mathcal{L} such that the L_2 -gain $\frac{\|\tilde{z}\|_{L_2}}{\|w\|_{L_2}}$ is bounded from above by `gam_val`.

Given the observer operator \mathcal{L} , we can construct the Estimator (11.3), obtaining a PIE

$$\begin{aligned}
\left(\begin{bmatrix} \mathcal{T} & 0 \\ 0 & \mathcal{T} \end{bmatrix} \begin{bmatrix} \dot{\mathbf{v}} \\ \dot{\hat{\mathbf{v}}} \end{bmatrix} \right) (t, s) &= \left(\begin{bmatrix} \mathcal{A} & 0 \\ -\mathcal{L} \mathcal{C}_2 & \mathcal{A} + \mathcal{L} \mathcal{C}_2 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \hat{\mathbf{v}} \end{bmatrix} \right) (t, s) + \left(\begin{bmatrix} \mathcal{B}_1 \\ \mathcal{L} \mathcal{D}_{21} \end{bmatrix} w \right) (t) \\
\begin{bmatrix} z \\ \hat{z} \end{bmatrix} (t) &= \left(\begin{bmatrix} \mathcal{C}_1 & 0 \\ 0 & \mathcal{C}_1 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \hat{\mathbf{v}} \end{bmatrix} \right) (t) + \left(\begin{bmatrix} \mathcal{D}_{11} \\ 0 \end{bmatrix} w \right) (t)
\end{aligned}$$

We can declare this PIE in PIETOOLS as

```

% Construct the operators defining the PIE.
T_CL = [T, 0*T; 0*T, T];
A_CL = [A, 0*A; -Lval*C2, A+Lval*C2];    B_CL = [B1; Lval*D21];
C_CL = [C1, 0*C1; 0*C1, C1];            D_CL = [D11; 0*D11];

% Declare the PIE.
PIE_CL = pie_struct();
PIE_CL.vars = PIE.vars;
PIE_CL.dom = PIE.dom;
PIE_CL.T = T_CL;
PIE_CL.A = A_CL;                PIE_CL.B1 = B_CL;
PIE_CL.C1 = C_CL;                PIE_CL.D11 = D_CL;
PIE_CL = initialize(PIE_CL);

```

Then, we can use PIESIM to simulate the evolution of the PDE state $\mathbf{x}(t) = (\mathcal{T}\mathbf{v})(t)$ and its estimate $\hat{\mathbf{x}}(t) = (\mathcal{T}\hat{\mathbf{v}})(t)$ associated to the system defined by PIE_CL. For this, we first declare initial conditions $\begin{bmatrix} \mathbf{v}^{(0,s)} \\ \hat{\mathbf{v}}^{(0,s)} \end{bmatrix}$ and values of the disturbance $w(t)$ as

```

% Declare initial conditions for the state components of the PIE
syms st sx real
uinput.ic.PDE = [-10*sx;      % Actual initial PIE state value
                 0];        % Estimated initial PIE state value

% Declare the value of the disturbance w(t)
uinput.w = 2*sin(pi*st);

```

Here we use symbolic objects `st` and `sx` to represent a temporal variable t and spatial variable s respectively. Note that, since we will be simulating the PIE directly, the initial conditions `uinput.ic.PDE` will also correspond to the initial values of the PIE state $\begin{bmatrix} \mathbf{v}^{(0,s)} \\ \hat{\mathbf{v}}^{(0,s)} \end{bmatrix}$, not to those of the PDE state. Here, we let the initial PIE state $\mathbf{v}(0)$ be parabolic, and start with an estimate of this state that is just $\hat{\mathbf{v}}(0) = 0$. Given these initial conditions, we then simulate the evolution of the PDE state $\begin{bmatrix} \mathbf{x} \\ \hat{\mathbf{x}} \end{bmatrix} = \begin{bmatrix} \mathcal{T} & 0 \\ 0 & \mathcal{T} \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \hat{\mathbf{v}} \end{bmatrix}$ using PIESIM as

```

% Set options for the discretization and simulation:
opts.plot = 'no';    % Do not plot the final solution
opts.N = 8;         % Expand using 8 Chebyshev polynomials
opts.tf = 1;        % Simulate up to t = 1;
opts.dt = 1e-3;     % Use time step of 10^-3
opts.intScheme = 1; % Time-step using Backward Differentiation Formula (BDF)
ndiff = [0,0,2];    % The PDE state involves 2 twice differentiable states

% Simulate the solution to the PIE with estimator.
[solution,grid] = PIESIM(PIE_CL,opts,uinput,ndiff);

% Extract actual and estimated solution at each time step.
x_act = reshape(solution.timedep.pde(:,1,:),opts.N+1,[]);
x_est = reshape(solution.timedep.pde(:,2,:),opts.N+1,[]);
tval = solution.timedep.dtime;

```

Here, we use `opts.N=8` Chebyshev polynomials in the expansion of the solution, and simulate up to `opts.tf=1`, taking time steps of `opts.dt=1e-3`. Having obtained the solution, we then

collect the actual values $\mathbf{x}(t, s)$ of the PDE state at each time step and each grid point in `x_act`, and the estimated values $\hat{\mathbf{x}}(t, s)$ of the PDE state at each time step and each grid point in `x_est`. Plotting the obtained values at several grid points, as well as the error in estimated state, we obtain a graph as in Figure [11.1](#).

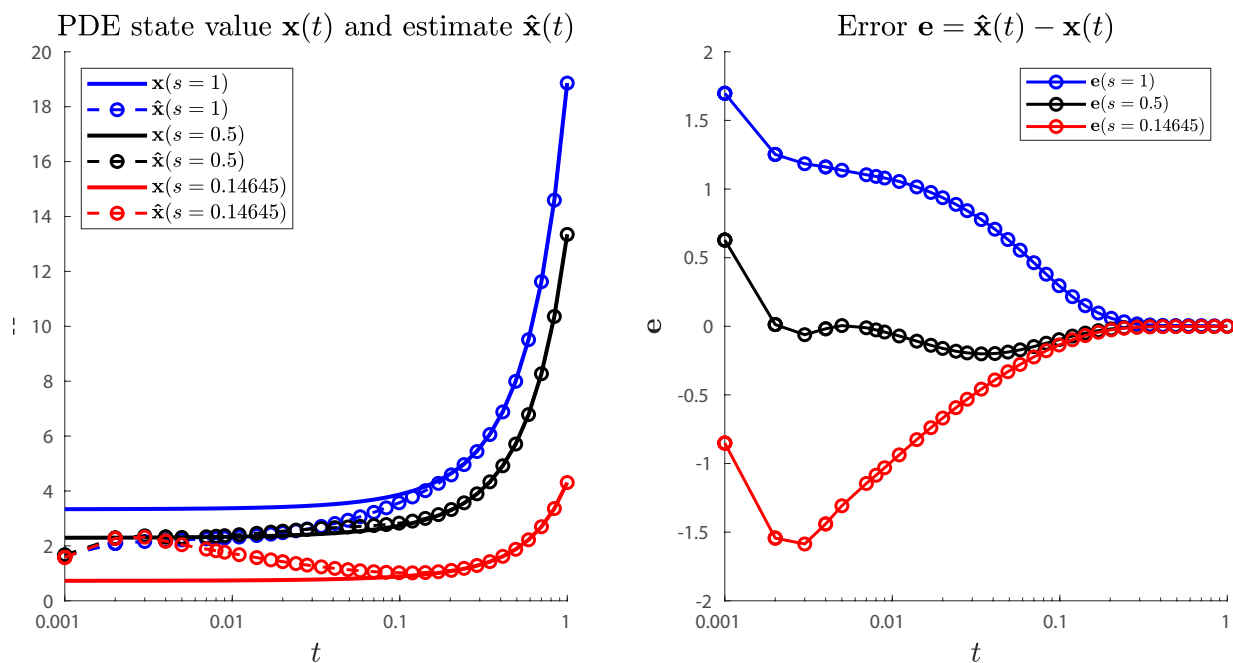


Figure 11.1: Simulated value of PDE state $\mathbf{x}(t, s)$ and estimated state $\hat{\mathbf{x}}(t, s)$, along with the error $\mathbf{e} = \hat{\mathbf{x}}(t, s) - \mathbf{x}(t, s)$ associated to the PDE (11.1) at several grid points $s \in [0, 1]$, using the Estimator (11.3) with operator \mathcal{L} computed by solving the H_∞ -optimal estimator LPI. See the demo file “Hinf_Optimal_Estimator_DEMO”.

As Figure 11.1 shows, the value of the estimate state $\hat{\mathbf{x}}$ at each of the grid points quickly converge to those of the actual state \mathbf{x} , despite starting off with a rather poor estimate of $\hat{\mathbf{x}} = 0$. Within a time of 1, the values of the actual and estimated state become indistinguishable (at the displayed scale), with the error converging to zero. This is despite the fact that the value of the state itself continues to increase, as the PDE (11.1) is unstable.

The full code constructing the optimal estimator, simulating the PDE state and its estimate, and plotting the results has been included in PIETOOLS as the demo file “Hinf_Optimal_Estimator_DEMO”.

11.6 DEMO 6: H_∞ -optimal Controller synthesis for PDEs

We consider an unstable reaction-diffusion PDE, with output z , disturbance w and control input u

$$\begin{aligned}
 & \dot{\mathbf{x}}(t, s) = \partial_s^2 \mathbf{x}(t, s) + 4\mathbf{x}(t, s) + sw(t) + su(t), & s \in [0, 1], \\
 \text{with BCs} & \quad 0 = \mathbf{x}(t, 0) = \partial_s \mathbf{x}(t, 1), \\
 \text{and outputs} & \quad z(t) = \begin{bmatrix} \int_0^1 \mathbf{x}(t, s) ds + w(t) \\ u(t) \end{bmatrix}.
 \end{aligned} \tag{11.6}$$

This PDE can be easily declared in PIETOOLS as

```

pvar s t
lam = 4;
PDE = sys();
x = state('pde'); w = state('in');
z = state('out', 2); u = state('in');
eqs = [diff(x,t) == diff(x,s,2) + lam*x + s*w + s*u;
z == [int(x,s,[0,1]) + w; u];
subs(x,s,0)==0;
subs(diff(x,s),s,1)==0];
PDE = addequation(PDE,eqs);
PDE = setControl(PDE,u);
display_PDE(PDE);

```

Next, we convert the PDE system to a PIE by using the following code and extract relevant PI operators for easier access:

```

PIE = convert(PDE,'pie'); PIE = PIE.params;
T = PIE.T;
A = PIE.A; C1 = PIE.C1; B2 = PIE.B2;
B1 = PIE.B1; D11 = PIE.D11; D12 = PIE.D12;

```

Then, the PDE (11.6) can be equivalently represented by a PIE

$$\begin{aligned}
 (\mathcal{T}\dot{\mathbf{v}})(t, s) &= (\mathcal{A}\mathbf{v})(t, s) + (\mathcal{B}_1 w)(t, s) + (\mathcal{B}_2 u)(t, s), & s \in [0, 1] \\
 z(t) &= (\mathcal{C}_1 \mathbf{v})(t) + (\mathcal{D}_{11} w)(t) + (\mathcal{D}_{12} u)(t),
 \end{aligned} \tag{11.7}$$

where we define $\mathbf{v} := \partial_s^2 \mathbf{x}$ and $\mathbf{x} = \mathcal{T}\mathbf{v}$.

We now attempt to find a state-feedback H_∞ -optimal controller for the PIE (11.7). In particular, we use $u(t) = \mathcal{K}\mathbf{v}(t)$, where $\mathcal{K} : L_2 \rightarrow \mathbb{R}$ is a PI operator of the form

$$\mathcal{K}\mathbf{v} = \int_a^b K(s)\mathbf{v}(s)ds.$$

Then we get the closed loop system

$$\begin{aligned} \mathcal{T}\dot{\mathbf{v}}(t) &= (\mathcal{A} + \mathcal{B}_2\mathcal{K})\mathbf{v}(t) + \mathcal{B}_1w(t), \\ z(t) &= (\mathcal{C}_1 + \mathcal{D}_{12}\mathcal{K})\mathbf{z}(t) + \mathcal{D}_{11}w(t). \end{aligned} \quad (11.8)$$

Next, we solve the LPI for H_∞ -optimal control to find a \mathcal{K} that minimizes the gain $\gamma := \frac{\|z\|_{L_2}}{\|w\|_{L_2}}$ from disturbances w to errors \tilde{z} in the output. See also Section 13.3. To find such an operator, we solve the LPI,

$$\min_{\gamma, \mathcal{P}, \mathcal{Z}} \gamma \text{ s.t. } \mathcal{P} \succ 0$$

$$\begin{bmatrix} -\gamma I & \mathcal{D}_{11} & \mathcal{T}(\mathcal{P}\mathcal{C}_1 + \mathcal{Z}\mathcal{D}_{12}) \\ (\cdot)^* & -\gamma I & \mathcal{B}_1^* \\ (\cdot)^* & (\cdot)^* & \mathcal{T}(\mathcal{A}\mathcal{P} + \mathcal{B}_2\mathcal{Z})^* + (\mathcal{A}\mathcal{P} + \mathcal{B}_2\mathcal{Z})\mathcal{T}^* \end{bmatrix} \preceq 0 \quad (11.9)$$

so that, for any solution $(\gamma, \mathcal{P}, \mathcal{Z})$ to this problem, letting $\mathcal{K} := \mathcal{Z}\mathcal{P}^{-1}$, we have $\|z\| \leq \gamma\|w\|$.

In Chapter 7, we showed in detail how the LPI (11.9) can be declared for a PIE of the form (11.7). The code declaring and solving this LPI can also be found in the demo file “Hinf_optimal_control_DEMO”. Here we use pre-defined `executive` files to solve the above LPI, which is a simple function call as shown below

```
| settings = lpsettings('heavy');
| [prog, kval, gam_val] = PIETOOLS_Hinf_control(PIE, settings);
```

The executive, in this particular case, returns a value `kval` of the operator \mathcal{K} such that the L_2 -gain $\frac{\|z\|_{L_2}}{\|w\|_{L_2}}$ is bounded from above by `gam_val`. Next, we construct the closed loop PIE using the function `closedLoop_PIE`,

```
| PIE_CL = closedLoopPIE(PIE, kval);
```

Note, closed loop system can be constructed manually, similar to the method used in 11.5.

Then, we can use `PIESIM` to simulate the evolution of the PDE state $\mathbf{x}(t) = (\mathcal{T}\mathbf{v})(t)$ associated to the system defined by `PIE_CL`. As described before we can set initial condition and disturbance using the commands below

```
| % Declare initial conditions for the state components of the PIE
| syms st sx real
| uinput.ic.PDE = -10*sx; % IC PIE
| uinput.w = exp(-st); % disturbance
```

Note that, since we will be simulating the PIE directly, the initial conditions `uinput.ic.PDE` will also correspond to the initial values of the PIE state $\mathbf{v}(0, s)$, not to those of the PDE state. In this example, we have used multiple initial conditions and disturbance inputs. For brevity, the code corresponding to all the different initial conditions and disturbances is not presented.

Next, we set the parameters related to the numerical scheme and simulation as shown below

```

% Set options for the discretization and simulation:
opts.plot = 'no'; % Do not plot the final solution
opts.N = 8; % Expand using 8 Chebyshev polynomials
opts.tf = 1; % Simulate up to t = 1;
opts.dt = 1e-3; % Use time step of 10^-3
opts.intScheme = 1; % Time-step using Backward Differentiation Formula (BDF)
ndiff = [0,0,1]; % The PDE state involves 2 twice differentiable states

```

These parameters are same as the ones described in 11.5, and hence the description is omitted here.

We run multiple simulations for both open (without controller) and closed loop PIEs (with controller) the results of which are plotted in Figure 11.2 and Figure 11.3, respectively.

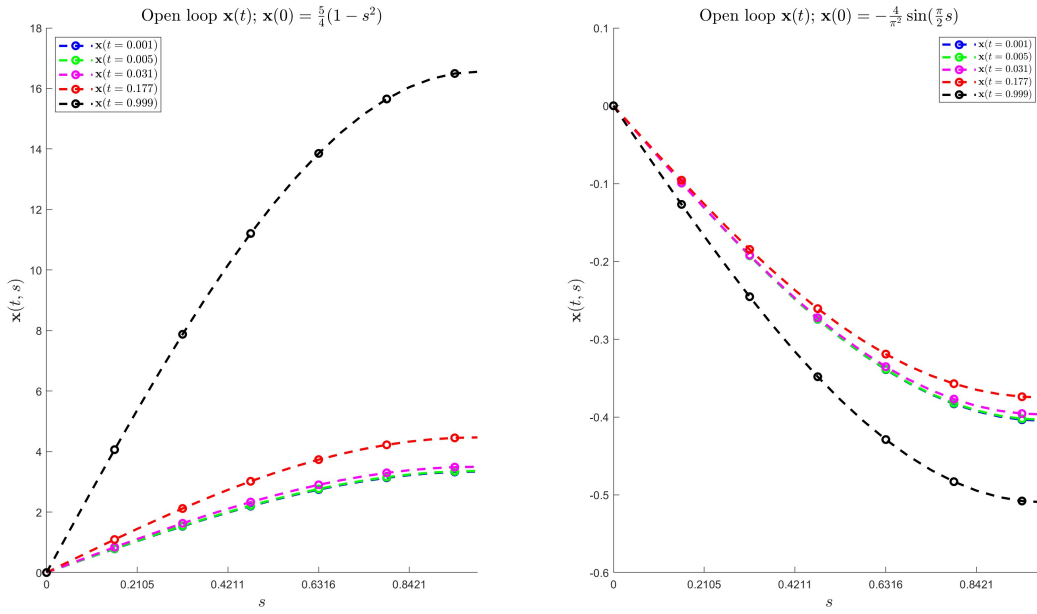


Figure 11.2: Simulated value of PDE state $\mathbf{x}(t, s)$ at several grid points $s \in [0, 1]$, without controller input, for different initial conditions and a bounded L_2 disturbance w .

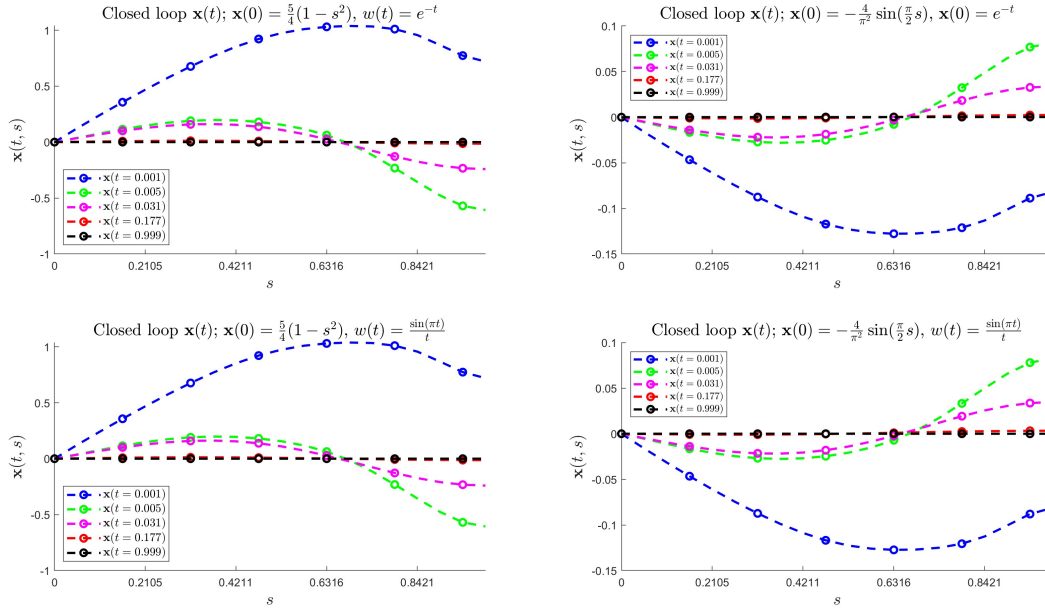


Figure 11.3: Simulated value of PDE state $\mathbf{x}(t, s)$ at several grid points $s \in [0, 1]$, with H_∞ -optimal state feedback controller, for different initial conditions and a bounded L_2 disturbance w . L_2 -gain = 1.6205.

Lastly, for the initial condition $\mathbf{v}(0, s) = -10s$, we plot the regulated output $z(t)$ when subjected to two different disturbance inputs, namely, $w(t) = \exp(-t)$ and $w(t) = \frac{\sin(\pi t)}{t}$. This can be obtained by numerical integration of the solution as shown in the code below, which can they be plotted to obtain Figure 11.4.

```

tval = linspace(0,1,2000); % grid points in time
XX = linspace(0,1,20); % grid points in space
w1_tval = subs(sin(pi*st)./(st+eps),tval);
w2_tval = subs(exp(-st),tval);
z_quadrature = double(subs(C1.Q1(1,1),s,XX));
k_quadrature = double(subs(Kval.Q1,s,XX));
ZZ1 = trapz(z_quadrature,x_CL_a)+double(w1_tval); %z1 for w1 disturbance
ZZ2 = trapz(z_quadrature,x_CL_aa)+double(w2_tval);%z1 for w2 disturbance
ZZ3 = trapz(k_quadrature,x_CL_a); %u(t) for w1 disturbance
ZZ4 = trapz(k_quadrature,x_CL_aa); %u(t) for w1 disturbance

```

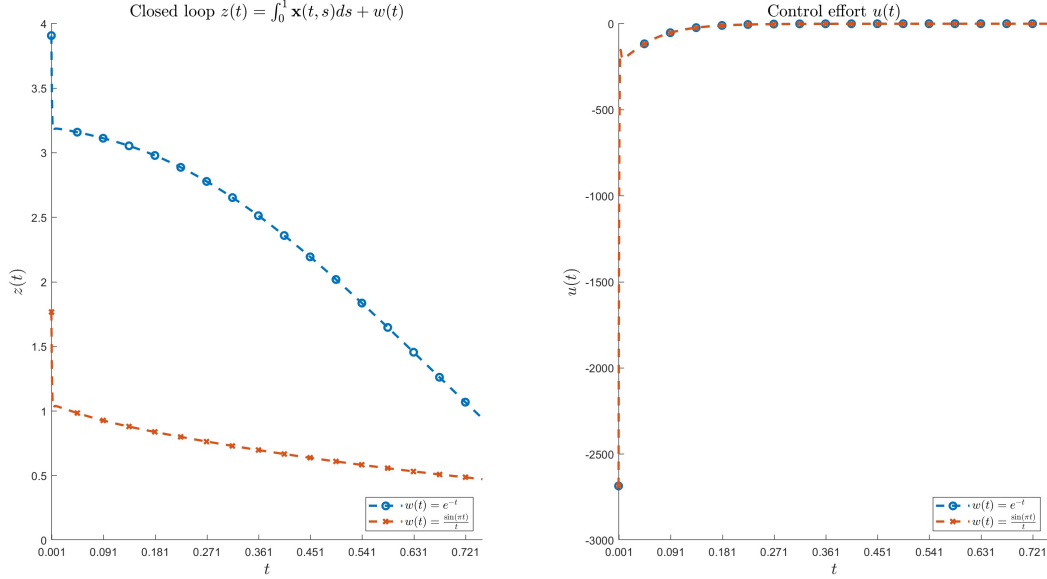


Figure 11.4: Simulated value of $Z(t)$, with H_∞ -optimal state feedback controller, for different bounded L_2 disturbances, w . L_2 -gain = 1.6205.

The full code for designing the optimal controller, simulating the PDE state, and plotting the results has been included in PIETOOLS as the demo file “Hinf_optimal_control_DEMO”

11.7 DEMO 7: Observer-based Controller design and simulation for PDEs

In Sections 11.5 and 11.6, we showed the procedure to find optimal estimator and controller for a PDE. The controller was designed based on the information of the complete PDE state, which, in practice, is unrealistic. So, in this section, we use the observer and controller obtained in the previous sections and couple them by changing the control law to be dependent on the estimated state instead of PDE state, i.e., we set $u = \mathcal{K}\hat{\mathbf{v}}$, where \mathcal{K} is the control gains obtained in 11.6 and $\hat{\mathbf{v}}$ is the state estimate from the observer found in 11.5.

For demonstration, we reuse the unstable reaction-diffusion PDE, with output z , disturbance w and control input u

$$\begin{aligned}
 & \dot{\mathbf{x}}(t, s) = \partial_s^2 \mathbf{x}(t, s) + 4\mathbf{x}(t, s) + sw(t) + su(t), & s \in [0, 1], \\
 \text{with BCs} & \quad 0 = \mathbf{x}(t, 0) = \partial_s \mathbf{x}(t, 1), \\
 \text{and outputs} & \quad z(t) = \begin{bmatrix} \int_0^1 \mathbf{x}(t, s) ds + w(t) \\ u(t) \end{bmatrix} \\
 & \quad y(t) = \mathbf{x}(t, 1).
 \end{aligned} \tag{11.10}$$

Recall, this PDE can be defined using the code

```

pvar s t;
lam = 4;
PDE = sys();
x = state('pde'); w = state('in');
z = state('out', 2); u = state('in'); y = state('out');
eqs = [diff(x,t) == diff(x,s,2) + lam*x + s*w + s*u;
       z == [int(x,s,[0,1]) + w; u];
       y == subs(x,s,1);
       subs(x,s,0)==0;
       subs(diff(x,s),s,1)==0];
PDE = addequation(PDE,eqs);
PDE = setControl(PDE,u);
PDE = setObserve(PDE,u);
display_PDE(PDE);

```

Next, we convert the PDE system to a PIE by using the following code and extract relevant PI operators for easier access:

```

PIE = convert(PDE,'pie');      PIE = PIE.params;
T = PIE.T;
A = PIE.A;      C1 = PIE.C1;   B2 = PIE.B2;
B1 = PIE.B1;    D11 = PIE.D11; D12 = PIE.D12;
C2 = PIE.C2;    D21 = PIE.D21; D22 = PIE.D22;

```

Details on observer design and controller design can be found in Sections 11.5 and 11.6 respectively, and therefore, will not be repeated here. For now, we proceed assuming the observer gains \mathcal{L} and the controller gains \mathcal{K} are known.

Recall, from Equations (11.3) and (11.7), the combined system is given by

$$\begin{aligned}
(\mathcal{T}\dot{\mathbf{v}})(t) &= (\mathcal{A}\mathbf{v})(t) + (\mathcal{B}_1 w)(t) + (\mathcal{B}_2 u)(t), \\
z(t) &= (\mathcal{C}_1 \mathbf{v})(t) + (\mathcal{D}_{11} w)(t) + (\mathcal{D}_{12} u)(t), \\
\mathcal{T}\dot{\hat{\mathbf{v}}}(t) &= \mathcal{A}\hat{\mathbf{v}}(t) + \mathcal{L}(\mathcal{C}_2 \mathbf{v}(t) - \mathcal{C}_2 \hat{\mathbf{v}}(t)), \\
\hat{z}(t) &= \mathcal{C}_1 \hat{\mathbf{v}}(t).
\end{aligned} \tag{11.11}$$

Using the observer-controller coupling, $u = \mathcal{K}\hat{\mathbf{v}}$, we get the closed loop PIE

$$\begin{aligned}
\left(\begin{bmatrix} \mathcal{T} & 0 \\ 0 & \mathcal{T} \end{bmatrix} \begin{bmatrix} \dot{\mathbf{v}} \\ \dot{\hat{\mathbf{v}}} \end{bmatrix} \right) (t, s) &= \left(\begin{bmatrix} \mathcal{A} & \mathcal{B}_2 \mathcal{K} \\ -\mathcal{L}\mathcal{C}_2 & \mathcal{A} + \mathcal{L}\mathcal{C}_2 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \hat{\mathbf{v}} \end{bmatrix} \right) (t, s) + \left(\begin{bmatrix} \mathcal{B}_1 \\ \mathcal{L}\mathcal{D}_{21} \end{bmatrix} w \right) (t) \\
\begin{bmatrix} z \\ \hat{z} \end{bmatrix} (t) &= \left(\begin{bmatrix} \mathcal{C}_1 & \mathcal{D}_{12} \mathcal{K} \\ 0 & \mathcal{C}_1 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \hat{\mathbf{v}} \end{bmatrix} \right) (t) + \left(\begin{bmatrix} \mathcal{D}_{11} \\ 0 \end{bmatrix} w \right) (t).
\end{aligned} \tag{11.12}$$

We can construct the above closed loop system using the code

```

T_CL = [T, 0*T; 0*T, T];
A_CL = [A, B2*Kval; -Lval*C2, A+Lval*C2];   B_CL = [B1; Lval*D21];
C_CL = [C1, D12*Kval; 0*C1, C1];           D_CL = [D11; 0*D11];
PIE_CL = pie_struct();
PIE_CL.vars = PIE.vars;
PIE_CL.dom = PIE.dom;
PIE_CL.T = T_CL;
PIE_CL.A = A_CL;      PIE_CL.B1 = B_CL;
PIE_CL.C1 = C_CL;    PIE_CL.D11 = D_CL;

```

```
| PIE_CL = initialize(PIE_CL);
```

Once, we have the closed loop PIE object, we can use PIESIM to simulate the system with different initial conditions and disturbances. Following the process presented in Section 11.6 and 11.5, we use the initial condition $\mathbf{v}(0, s) = -10s$ and $\hat{\mathbf{v}}(0, s) = 0$. Using the same options and disturbance as before, we simulate the open loop system (without control or observer) using the code

```
| syms st sx real
| opts.plot = 'no'; % Do not plot the final solution
| opts.N = 8; % Expand using 8 Chebyshev polynomials
| opts.tf = 1; % Simulate up to t = 1;
| opts.dt = 1e-3; % Use time step of 10^-3
| opts.intScheme=1; % Time-step using Backward Differentiation Formula (BDF)
| ndiff = [0,0,1]; % The PDE state involves 1 second order differentiable state variables
|
| % Simulate the solution to the PIE without controller for different IC.
| uinput.ic.PDE = [-10*sx]; % IC PIE
| uinput.w = exp(-st); % disturbance
| [solution_OL,grid] = PIESIM(PIE,opts,uinput,ndiff);
```

Then, we can access the time-varying PDE solution from `solution_OL` for post-processing. In Figure 11.5 we post the solution at different time values to show that the open loop system is unstable.

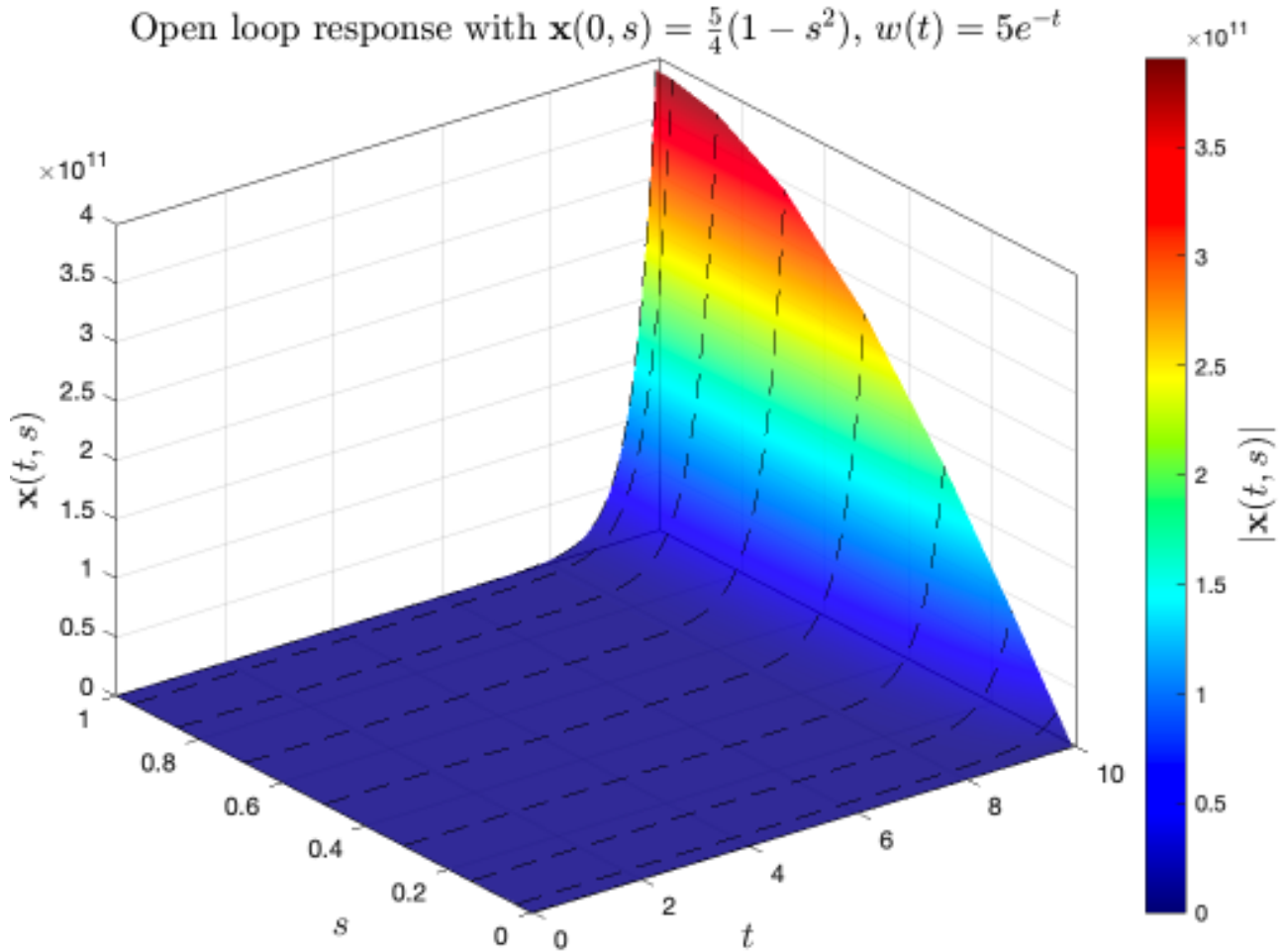


Figure 11.5: Simulated value of PDE state $\mathbf{x}(t, s)$ at several grid points $s \in [0, 1]$, without controller input, with some initial condition and a bounded L_2 disturbance w .

Likewise, we can simulate and extract the closed loop system solution using the code

```
% Simulate the solution to the PIE with controller for different IC and disturbance.
ndiff = [0,0,2];
uinput.ic.PDE = [-10*sx; 0]; % IC PIE and observed state
uinput.w = exp(-st); % disturbance
[solution_CL_a,grid] = PIESIM(PIE_CL,opts,uinput,ndiff);
uinput.ic.PDE = [sin(sx*pi/2); 0]; % IC PIE
uinput.w = 5*sin(pi*st)./(st+eps); % disturbance
[solution_CL_b,grid] = PIESIM(PIE_CL,opts,uinput,ndiff);
% Extract actual solution at each time step.
tval = solution_CL_a.timedep.dtime;
x_CL_a = reshape(solution_CL_a.timedep.pde(:,1,:),opts.N+1,[]);
hatx_CL_a = reshape(solution_CL_a.timedep.pde(:,2,:),opts.N+1,[]);
x_CL_b = reshape(solution_CL_b.timedep.pde(:,1,:),opts.N+1,[]);
hatx_CL_b = reshape(solution_CL_b.timedep.pde(:,2,:),opts.N+1,[]);
```

where, **note** that `ndiff` is changed to `[0,0,2]` because the closed loop PIE system has two

distributed states corresponding to two PDE states (PDE state and observer state) that are twice differentiable in space. Using the solutions stored in $\mathbf{x}_{CL_a(b)}$, we can plot the response of the closed loop system for different initial conditions and disturbances as shown below in Figure 11.6.

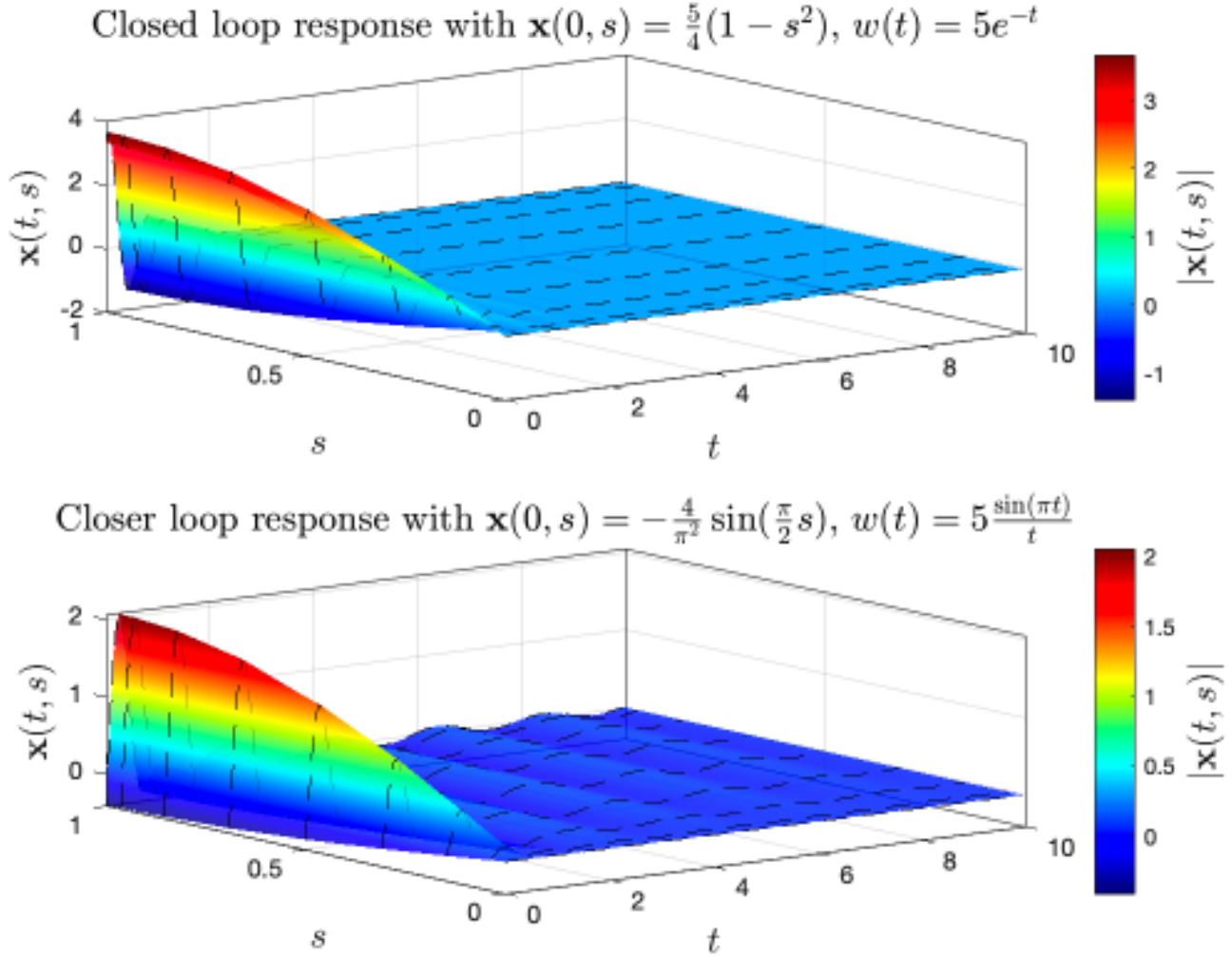


Figure 11.6: Simulated value of PDE state $\mathbf{x}(t, s)$ at several grid points $s \in [0, 1]$, with H_∞ -optimal state feedback controller, for different initial conditions and bounded L_2 disturbances w . L_2 -gain = 1.6205 for controller, L_2 -gain = 1.005 for observer.

Lastly, for the initial condition $\mathbf{v}(0, s) = -10s$ and $\hat{\mathbf{v}}(0) = 0$, we plot (see Figure 11.7 the regulated output $z(t)$ when subjected to two different disturbance inputs, namely, $w(t) = \exp(-t)$ and $w(t) = 5 \frac{\sin(\pi t)}{t}$, where z is obtained by numerical integration of the solution as described in Section 11.6.

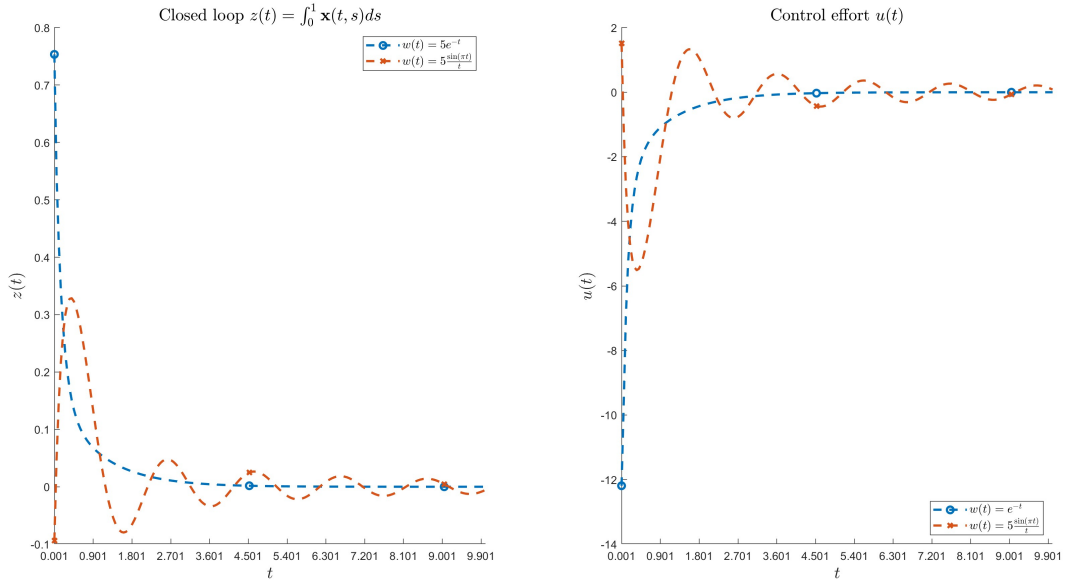


Figure 11.7: Simulated value of $Z(t)$, with H_∞ -optimal state feedback controller, for different bounded L_2 disturbances, w .

Chapter 12

Libraries of PDE and TDS Examples in PIETOOLS

In Chapters 4, 8 and 9, we have shown how partial differential equations and time-delay systems can be declared in PIETOOLS through different input formats. To help get started with each of these input formats, PIETOOLS includes a variety of example pre-defined PDE and TDS systems, including common examples and particular models from the literature. These examples are collected in the `PIETOOLS_examples` folder, and can be accessed calling the function `examples_PDE_library_PIETOOLS` and the scripts `examples_DDE_library_PIETOOLS` and `examples_NDSDDF_library_PIETOOLS`. In this Chapter, we illustrate how this works, focusing on PDE examples in Section 12.1, and TDS examples in Section 12.2.

12.1 A Library of PDE Example Problems

To help get started with analysing and simulating PDEs in PIETOOLS, a variety of PDE models have been included in the directory `PIETOOLS_examples/Examples_Library`. These systems include common PDE models, as well as examples from the literature, and are defined through separate MATLAB functions. Each of these functions takes two arguments

1. `GUI`: A binary index (0 or 1) indicating whether the example should be opened in the graphical user interface;
2. `params`: A string for specifying allowed parameters in the system.

For example, calling `help PIETOOLS_PDE_Ex_Reaction_Diffusion_Eq`, PIETOOLS indicates that this function declares a reaction-diffusion PDE

$$\dot{\mathbf{x}}(t, s) = \lambda \mathbf{x}(t, s) + \partial_s^2 \mathbf{x}(t, s), \quad s \in [0, 1] \quad \mathbf{x}(t, 0) = \mathbf{x}(t, 1) = 0,$$

where the value of the parameter λ can be set. Then, calling

```
| >> PDE = PIETOOLS_PDE_Ex_Reaction_Diffusion_Eq(0,{'lam=10;'})
```

we obtain a `pde_struct` object `PDE` representing the reaction-diffusion equation with $\lambda = 10$. Calling

```
| >> PDE = PIETOOLS_PDE_Ex_Reaction_Diffusion_Eq(1,{'lam=10;'})
```

The PDE will also be loaded in the GUI, though a default value of $\lambda = 9.86$ will be used, as the GUI will always load a pre-defined file, which cannot be adjusted from the command line.

To simplify the process of extracting PDE examples, PIETOOLS includes a function `examples_PDE_library_PIETOOLS()`. In this function, each of the pre-defined PDEs is assigned an index, allowing desired PDEs to be extracted by calling `examples_PDE_library_PIETOOLS` with the associated index. For example, scrolling through this function we find that the reaction-diffusion equation is the fifth system in the list, and therefore, we can obtain a `pde_struct` object defining this system by calling the library function with argument “5”, returning

```
| >> PDE = examples_PDE_library_PIETOOLS(5);
| --- Extracting ODE-PDE example 5 ---
|
| (d/dt) x(t,s) = 9.86 * x(t,s) + (d^2/ds^2) x(t,s);
| 0 = x(t,0);
| 0 = x(t,1);
|
| Would you like to run the executive associated to this problem? (y/n)
| -->
```

We note that the function asks whether an executive should be run for the considered PDE. This is because, for each of the PDE examples, an associated LPI problem has also been declared, matching one of the `executive` files (see also Chapter 13). For the reaction-diffusion equation, the proposed executive is the `PIETOOLS_stability` function, testing stability of the PDE. Entering `yes` in the command line window, this executive will be automatically run, whilst entering `no` will stop the function, and just return the `pde_struct` object `PDE`.

Using the `examples_PDE_library_PIETOOLS` function, parameters in the PDE can also be adjusted, calling e.g.

```
| >> PDE = examples_PDE_library_PIETOOLS(5,'lam=10;');
| --- Extracting ODE-PDE example 5 ---
|
| (d/dt) x(t,s) = 10 * x(t,s) + (d^2/ds^2) x(t,s);
| 0 = x(t,0);
| 0 = x(t,1);
```

Similarly, if multiple parameters can be specified, we specify each of these parameters separately. For example, we note that example 7 corresponds to a PDE

$$\dot{\mathbf{x}}(t,s) = c(s)\mathbf{x}(t,s) + b(s)\partial_s\mathbf{x}(t,s) + a(s)\partial_s^2\mathbf{x}(t,s), \quad s \in [0,1], \mathbf{x}(t,0) = \partial_s\mathbf{x}(t,1) = 0,$$

where the values of the functions $a(s)$, $b(s)$ and $c(s)$ can be specified. As such, we can declare this PDE for $a = 1$, $b = 2$ and $c = 3$ by calling

```
| >> PDE = examples_PDE_library_PIETOOLS(7,'a=1;', 'b=2;', 'c=3;');
| --- Extracting ODE-PDE example 7 ---
|
| (d/dt) x(t,s) = 3 * x(t,s) + 2 * (d/ds) x(t,s) + (d^2/ds^2) x(t,s);
| 0 = x(t,0);
| 0 = (d/ds) x(t,1);
```

Finally, we can also open the PDE in the GUI by calling

```
| >> examples_PDE_library_PIETOOLS(7,'GUI');
| --- Extracting ODE-PDE example 7 ---
```

or extract the PDE as a `pde_struct` (terms-format) and open it in the GUI by calling

```
| >> PDE = examples_PDE_library_PIETOOLS(7,'TERM','GUI');
| --- Extracting ODE-PDE example 7 ---
```

In each case, the function will still ask whether the executive associated with the PDE should be run as well. Of course, you can also convert the PDE to a PIE yourself using `convert`, and then run any desired executive manually, assuming this executive makes sense (e.g. there's no sense in computing an H_∞ -gain if your system has no outputs).

12.2 Libraries of DDE, NDS, and DDF Examples

Aside from the PDE examples, a list of TDS examples is also included in PIETOOLS, in DDE, NDS, and DDF format. Unlike the PDE problems, however, these TDS examples are all not separated into distinct functions, but are divided over two scripts: `examples_DDE_library_PIETOOLS` and `examples_NDSDDF_library_PIETOOLS`. In each of these scripts, most examples are commented, and only one example should be uncommented at any time. This example can then be extracted by calling the script, adding a structure DDE, NDS or DDF to the MATLAB workspace. To extract a different example, the desired example must be uncommented, and all other examples must be commented, at which point the script can be called again to obtain a structure representing the desired system.

We expect to update the DDE and NDS/DDF example files in a future release to match the format used for the PDE example library.

12.2.1 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.

12.2.2 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\_NDS(NDS,'ddf');
| >> DDF = minimize\_PIETOOLS\_DDF(DDF);
```

```
| >> PIE = convert\_PIETOOLS\_DDF(DDF,'pie');
```

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 13

Standard Applications of LPI Programming

In Chapter 7, we showed how general LPI optimization programs can be declared and solved in PIETOOLS. In this chapter, we provide several applications of LPI programming for analysis, estimation, and control of PIEs. Recall that such PIEs take the form

$$\begin{aligned}\mathcal{T}\dot{\mathbf{v}}(t) + \mathcal{T}_w\dot{w}(t) + \mathcal{T}_u\dot{u}(t) &= \mathcal{A}\mathbf{v}(t) + \mathcal{B}_1w(t) + \mathcal{B}_2u(t), \quad \mathbf{v}(0) = \mathbf{v}_I \\ z(t) &= \mathcal{C}_1\mathbf{v}(t) + \mathcal{D}_{11}w(t) + \mathcal{D}_{12}u(t), \\ y(t) &= \mathcal{C}_2\mathbf{v}(t) + \mathcal{D}_{21}w(t) + \mathcal{D}_{22}u(t),\end{aligned}\tag{13.1}$$

where $\mathbf{v} = \begin{bmatrix} v_0 \\ \mathbf{v}_1 \\ \mathbf{v}_2 \\ \mathbf{v}_3 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \\ L_2^{n_2}[c,d] \\ L_2^{n_3}[[a,b] \times [c,d]] \end{bmatrix}$, and where \mathcal{T} through \mathcal{D}_{22} are all PI operators. In Section 13.1, we provide several LPIs for stability analysis and H_∞ -gain estimation of such PIEs, setting $u = 0$. Then, in Section 13.2, we present an LPI for H_∞ -optimal estimation of PIEs of the form (13.1), followed by an LPI for H_∞ -optimal full-state feedback control in Section 13.3. We note that, almost all of these LPIs have already been implemented as `executive` functions in PIETOOLS, and we will refer to these executives when applicable.

13.1 LPIs for Analysis of PIEs

Using LPIs, several properties of a PIE as in Equation (13.1) may be tested, as listed in this section. In particular, the LPIs listed below are extensions of classical results used in analysis of ODEs using LMIs. For most of these LPIs, PIETOOLS includes an executive function that may be run to solve it for a given PIE.

13.1.1 Operator Norm

For a PI operator \mathcal{T} , an upper bound $\sqrt{\gamma}$ on the operator norm $\|\mathcal{T}\|$ can be computed by solving the LPI

$$\begin{aligned}\min_{\gamma, \mathcal{P}} \quad & \gamma \\ \mathcal{T}^*\mathcal{T} - \gamma \preceq & 0\end{aligned}\tag{13.2}$$

This LPI has not been implemented as an executive in PIETOOLS, but has been implemented in the demo file `volterra_operator_norm_DEMO` (see also Section 11.2).

13.1.2 Stability

For given PI operators \mathcal{T} and \mathcal{A} , stability of the PIE

$$\mathcal{T}\dot{\mathbf{v}}(t) = \mathcal{A}\mathbf{v}(t)$$

can be tested by solving the LPI

$$\begin{aligned} \mathcal{P} &\succ 0 \\ \mathcal{T}^*\mathcal{P}\mathcal{A} + \mathcal{A}^*\mathcal{P}\mathcal{T} &\preceq 0 \end{aligned} \quad . \quad (13.3)$$

If there exists a PI operator \mathcal{P} such that this LPI is feasible, then the PIE is stable. Given a structure PIE, this LPI may be solved for the associated PIE by calling

```
| >> [prog, Pop] = PIETOOLS_stability(PIE, settings);
```

Here `prog` will be an SOS program structure describing the solved problem and `Pop` will be a `dopvar` object describing the (unsolved) decision operator \mathcal{P} from which the solved operator can be derived using

```
| >> Pop = getsol_lpivar(prog,Pop);
```

See Chapter 7 for more information on the operation of this function and the `settings` input.

13.1.3 Dual Stability

For given PI operators \mathcal{T} and \mathcal{A} , stability of the PIE

$$\mathcal{T}\dot{\mathbf{v}}(t) = \mathcal{A}\mathbf{v}(t)$$

can also be tested by solving the LPI

$$\begin{aligned} \mathcal{P} &\succ 0 \\ \mathcal{T}\mathcal{P}\mathcal{A}^* + \mathcal{A}\mathcal{P}\mathcal{T}^* &\preceq 0 \end{aligned} \quad (13.4)$$

If there exists a PI operator \mathcal{P} such that this LPI is feasible, then the PIE is stable. Given a structure PIE, this LPI may be solved for the associated PIE by calling

```
| >> [prog, Pop] = PIETOOLS_stability_dual(PIE, settings);
```

Here `prog` will be an SOS program structure describing the solved problem and `Pop` will be a `dopvar` object describing the (unsolved) decision operator \mathcal{P} .

13.1.4 Input-Output Gain

Consider a system of the form

$$\begin{aligned}\mathcal{T}\dot{\mathbf{v}}(t) &= \mathcal{A}\mathbf{v}(t) + \mathcal{B}_1 w(t), \quad \mathbf{v}(0) = \mathbf{0} \\ z(t) &= \mathcal{C}_1 \mathbf{v}(t) + \mathcal{D}_{11} w(t),\end{aligned}\tag{13.5}$$

where $w \in L_2^{n_w}[0, \infty)$. Then, $z \in L_2^{n_z}[0, \infty)$, and $\|z\|_{L_2} \leq \gamma \|w\|_{L_2}$, if the following LPI is feasible

$$\begin{aligned}\min_{\gamma, \mathcal{P}} \quad & \gamma \\ \mathcal{P} \succ & 0 \\ \begin{bmatrix} -\gamma I & \mathcal{D}_{11}^* & \mathcal{B}_1^* \mathcal{P} \mathcal{T} \\ (\cdot)^* & -\gamma I & \mathcal{C}_1 \\ (\cdot)^* & (\cdot)^* & \mathcal{T}^* \mathcal{P} \mathcal{A} + \mathcal{A}^* \mathcal{P} \mathcal{T} \end{bmatrix} \preceq & 0\end{aligned}\tag{13.6}$$

Given a structure PIE, this LPI may be solved for the associated PIE by calling

```
| >> [prog, Pop, gam] = PIETOOLS_Hinf_gain(PIE, settings);
```

Here `prog` will be an SOS program structure describing the solved problem, and `gam` will be the smallest value of γ for which the LPI was found to be feasible, offering a bound on the L_2 -gain from w to z of the system. The output `Pop` will be a `dopvar` object describing the (unsolved) decision operator \mathcal{P} .

13.1.5 Dual Input-Output Gain

For a System (13.5) with $w \in L_2^{n_w}[0, \infty)$, an upper bound γ on the L_2 -gain from w to z can also be obtained by solving the LPI

$$\begin{aligned}\min_{\gamma, \mathcal{P}} \quad & \gamma \\ \mathcal{P} \succ & 0 \\ \begin{bmatrix} -\gamma I & \mathcal{D}_{11} & \mathcal{T} \mathcal{P} \mathcal{C}_1 \\ (\cdot)^* & -\gamma I & \mathcal{B}_1^* \\ (\cdot)^* & (\cdot)^* & \mathcal{T} \mathcal{P} \mathcal{A}^* + \mathcal{A} \mathcal{P} \mathcal{T}^* \end{bmatrix} \preceq & 0\end{aligned}\tag{13.7}$$

Given a structure PIE, this LPI may be solved for the associated PIE by calling

```
| >> [prog, Pop, gam] = PIETOOLS_Hinf_gain_dual(PIE, settings);
```

Here `prog` will be an SOS program structure describing the solved problem, and `gam` will be the found optimal value for γ . The output `Pop` will be a `dopvar` object describing the (unsolved) decision operator \mathcal{P} .

13.1.6 Positive Real Lemma

For a PIE of the form

$$\mathcal{T}\dot{\mathbf{v}}(t) = \mathcal{A}\mathbf{v}(t) + \mathcal{B}_1 w(t), \quad \mathbf{v}(0) = \mathbf{0}$$

$$z(t) = \mathcal{C}_1 \mathbf{v}(t) + \mathcal{D}_{11} w(t),$$

we can test whether the system is passive by solving the LPI

$$\begin{aligned} \mathcal{P} \succ 0 \\ \begin{bmatrix} -\mathcal{D}_{11}^* - \mathcal{D}_{11} & \mathcal{B}_1^* \mathcal{P} \mathcal{T} - \mathcal{C}_1 \\ (\cdot)^* & \mathcal{T}^* \mathcal{P} \mathcal{A} + \mathcal{A}^* \mathcal{P} \mathcal{T} \end{bmatrix} \preceq 0 \end{aligned} \quad (13.8)$$

If there exists a PI operator \mathcal{P} such that this LPI is feasible, then the system is passive. Note that this LPI has not been implemented as an executive in PIETOOLS.

13.2 LPIs for Optimal Estimation of PIEs

For the following PIE

$$\begin{aligned} \mathcal{T} \dot{\mathbf{v}}(t) + \mathcal{T}_w \dot{w}(t) &= \mathcal{A} \mathbf{v}(t) + \mathcal{B}_1 w(t), \\ z(t) &= \mathcal{C}_1 \mathbf{v}(t) + \mathcal{D}_{11} w(t), \\ y(t) &= \mathcal{C}_2 \mathbf{v}(t) + \mathcal{D}_{21} w(t), \end{aligned} \quad (13.9)$$

a state estimator has the following structure:

$$\begin{aligned} \mathcal{T} \dot{\hat{\mathbf{v}}}(t) &= \mathcal{A} \hat{\mathbf{v}}(t) + \mathcal{L}(\hat{y}(t) - y(t)), \\ \hat{z}(t) &= \mathcal{C}_1 \hat{\mathbf{v}}(t) \\ \hat{y}(t) &= \mathcal{C}_2 \hat{\mathbf{v}}(t). \end{aligned} \quad (13.10)$$

so that the errors $\mathbf{e} := \hat{\mathbf{v}} - \mathbf{v}$ and $\tilde{z} := \hat{z} - z$ in respectively the state and regulated output estimates satisfy

$$\begin{aligned} \mathcal{T} \dot{\mathbf{e}} - \mathcal{T}_w \dot{w}(t) &= (\mathcal{A} + \mathcal{L} \mathcal{C}_2) \mathbf{e}(t) - (\mathcal{B}_1 + \mathcal{L} \mathcal{D}_{21}) w(t), \\ \tilde{z}(t) &= \mathcal{C}_1 \mathbf{e}(t) - \mathcal{D}_{11} w(t) \end{aligned}$$

The H_∞ -optimal estimation problem amounts to synthesizing \mathcal{L} such that the estimation error $\tilde{z} := \hat{z} - z$ admits $\|\tilde{z}\| \leq \gamma \|w\|$ for a particular $\gamma > 0$. To establish such an estimator, we can solve the LPI

$$\begin{aligned} \min_{\gamma, \mathcal{P}, \mathcal{Z}} \quad & \gamma \\ \mathcal{P} \succ 0 \\ \begin{bmatrix} \mathcal{T}_w^* (\mathcal{P} \mathcal{B}_1 + \mathcal{Z} \mathcal{D}_{21}) + (\cdot)^* & 0 & (\cdot)^* \\ 0 & 0 & 0 \\ -(\mathcal{P} \mathcal{A} + \mathcal{Z} \mathcal{C}_2)^* \mathcal{T}_w & 0 & 0 \end{bmatrix} + \begin{bmatrix} -\gamma I & -\mathcal{D}_{11}^\top & -(\mathcal{P} \mathcal{B}_1 + \mathcal{Z} \mathcal{D}_{21})^* \mathcal{T} \\ (\cdot)^* & -\gamma I & \mathcal{C}_1 \\ (\cdot)^* & (\cdot)^* & (\mathcal{P} \mathcal{A} + \mathcal{Z} \mathcal{C}_2)^* \mathcal{T} + (\cdot)^* \end{bmatrix} \preceq 0 \end{aligned} \quad (13.11)$$

Then, if this LPI is feasible for some $\gamma > 0$ and PI operators \mathcal{P} and \mathcal{Z} , then, letting $\mathcal{L} := \mathcal{P}^{-1} \mathcal{Z}$, the estimation error will satisfy $\|z - \hat{z}\| \leq \gamma \|w\|$. Given a structure PIE, this LPI may be solved for the associated PIE by calling

```
| >> [prog, Lop, gam, Pop, Zop] = PIETOOLS_Hinf_estimator(PIE, settings);
```

Here `prog` will be an SOS program structure describing the solved problem, `gam` will be the found optimal value for γ , and `Lop` will be an `opvar` object describing the optimal estimator \mathcal{L} . Outputs `Pop` and `Zop` will be `dopvar` objects describing the (unsolved) decision operators \mathcal{P} and \mathcal{Z} , from which the solved operators can be derived using

```
| >> Pop = getsol_lpivar(prog,Pop);          Zop = getsol_lpivar(prog,Zop);
```

See Chapter 7 for more information on the operation of this function and the `settings` input.

13.3 LPIs for Optimal Control of PIEs

In this section, we discuss the synthesis of H_∞ optimal control of a PIE of the form

$$\begin{aligned} \mathcal{T}\dot{\mathbf{v}}(t) &= \mathcal{A}\mathbf{v}(t) + \mathcal{B}_1w(t) + \mathcal{B}_2u(t), & \mathbf{v}(0) &= \mathbf{v}_0 \\ z(t) &= \mathcal{C}_1\mathbf{v}(t) + \mathcal{D}_{11}w(t) + \mathcal{D}_{12}u(t), \end{aligned} \quad (13.12)$$

where $w \in L_2[0, \infty)$. The problem of synthesizing an H_∞ -optimal controller amounts to determining a PIE operator \mathcal{K} such that, using the full-state feedback law $u(t) = \mathcal{K}\mathbf{v}(t)$, the regulated output z admits $\|z\|_{L_2} \leq \gamma\|w\|_{L_2}$ for a particular $\gamma > 0$. To establish such a controller, we can solve the LPI

$$\begin{aligned} & \min_{\gamma, \mathcal{P}, \mathcal{Z}} \gamma \\ & \mathcal{P} \succ 0 \\ dj & \begin{bmatrix} -\gamma I & \mathcal{D}_{11} & (\mathcal{C}_1\mathcal{P} + \mathcal{D}_{12}\mathcal{Z})\mathcal{T}^* \\ \mathcal{D}_{11}^* & -\gamma I & \mathcal{B}_1^* \\ (*)^* & \mathcal{B}_1 & (*)^* + (\mathcal{A}\mathcal{P} + \mathcal{B}_2\mathcal{Z})\mathcal{T}^* \end{bmatrix} \preceq 0 \end{aligned} \quad (13.13)$$

If this LPI is feasible for some $\gamma > 0$ and PI operators \mathcal{P} and \mathcal{Z} , then, letting $\mathcal{K} := \mathcal{Z}\mathcal{P}^{-1}$, the L_2 -gain for the controlled system with $u = \mathcal{K}\mathbf{v}$ will be such that $\|z\| \leq \gamma\|w\|$. Given a structure PIE, this LPI may be solved for the associated PIE by calling

```
| >> [prog, Kop, gam, Pop, Zop] = PIETOOLS_Hinf_control(PIE, settings);
```

Here `prog` will be an SOS program structure describing the solved problem, `gam` will be the found optimal value for γ , and `Kop` will be an `opvar` object describing the optimal feedback \mathcal{K} . Outputs `Pop` and `Zop` will be `dopvar` objects describing the (unsolved) decision operators \mathcal{P} and \mathcal{Z} , from which the solved operators can be derived using

```
| >> Pop = getsol_lpivar(prog,Pop);          Zop = getsol_lpivar(prog,Zop);
```

See Chapter 7 for more information on the operation of this function and the `settings` input.

Part IV
Appendices

Appendix A

PI Operators and their Properties

In this appendix, we discuss in a bit more detail the crucial properties of PI operators that PIETOOLS relies on for implementation and analysis of PIEs. For this, in Section A.1, we first recap the definitions of PI operators as presented in Chapter 5, also introducing some notation that we will continue to use throughout the appendix. In Section A.2, A.3, and A.4, we show that respectively the sum, composition, and adjoint of PI operators can be expressed as PI operators. In Section A.5, and A.6, we then show how the respectively the inverse of a PI operator, and the composition of a PI operator with a differential operator can be computed. Finally, in Section A.7, we show how a cone of positive PI operators can be parameterized by positive matrices, allowing an LPI constraint $\mathcal{P} \succcurlyeq 0$ to be posed as an LMI $P \succcurlyeq 0$.

For more information on PI operators, and full proofs of each of the results, we refer to e.g. [5] [8] (1D) and [4] (2D).

A.1 PI Operators on Different Function Spaces

Recall that we denote the space of square integrable functions on a domain Ω as $L_2[\Omega]$, with inner product

$$\langle \mathbf{x}, \mathbf{y} \rangle_{L_2} = \int_{\Omega} [\mathbf{x}(s)]^T \mathbf{y}(s) ds.$$

In defining the different PI operators, we will restrict ourselves to domains of 1D or 2D hypercubes. In 1D, such a hypercube is simply an interval $[a, b]$, for which we define the following PI operator:

Definition 1 (3-PI Operator). *For given parameters*

$$R := \{R_0, R_1, R_2\} \in \{L_2^{m \times n}[a, b], L_2^{m \times n}[[a, b]^2], L_2^{m \times n}[[a, b]^2]\} =: \mathcal{N}_{1D}^{m \times n}[a, b],$$

we define the associated 3-PI operator $\mathcal{P}[R] := \mathcal{P}_{\{R_0, R_1, R_2\}} : L_2^n[a, b] \rightarrow L_2^m[a, b]$ as

$$(\mathcal{P}[R]\mathbf{x})(s) := R_0(s)\mathbf{x}(s) + \int_a^s R_1(s, \theta)\mathbf{x}(\theta)d\theta + \int_s^b R_2(s, \theta)\mathbf{x}(\theta)d\theta, \quad (\text{A.1})$$

for any $\mathbf{x} \in L_2^n[a, b]$.

We note that 3-PI operators can be seen as (one possible) generalization of matrices to infinite dimensional vector spaces. In particular, suppose we have a matrix $P \in \mathbb{R}^{m \times n}$, which we decompose as $P = D + L + U$, where D is diagonal, L is strictly lower triangular, and U is strictly upper-triangular. Then, for any $x \in \mathbb{R}^n$, the i th element of the product Px is given by:

$$(Px)_i = D_{ii}x_i + \sum_{j=1}^{i-1} L_{ij}x_j + \sum_{j=i+1}^n U_{ij}x_j.$$

Compare this to the value of $\mathcal{P}[R]\mathbf{x}$ at a position $s \in [a, b]$ for some $\mathbf{x} \in L_2[a, b]$ and 3-PI parameters R :

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

Replacing row and column indices (i, j) by primary and dummy variables (s, θ) , and performing integration instead of summation, 3-PI operators have a structure very similar to that of matrices, wherein we can recognize a diagonal, lower-triangular, and upper-triangular part. Accordingly, we will occasionally refer to a PI operator of the form $\mathcal{P}_{\{R_0, 0, 0\}}$ as a diagonal 3-PI operator, and to PI operators of the forms $\mathcal{P}_{\{0, R_1, 0\}}$ and $\mathcal{P}_{\{0, 0, R_2\}}$ as lower- and upper-triangular PI operators respectively. The similar structure between matrices and PI operator also ensures that matrix operations such as addition and multiplication are valid for PI operators as well, as we will discuss in more detail in the next sections.

To map functions on a domain $[a, b] \times [c, d] \subset \mathbb{R}^2$, we also define the 9-PI operator:

Definition 2 (9-PI Operator). *For given parameters*

$$R := \begin{bmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{bmatrix} \in \begin{bmatrix} L_2^{m \times n}[[a, b] \times [c, d]] & L_2^{m \times n}[[a, b] \times [c, d]^2] & L_2^{m \times n}[[a, b] \times [c, d]^2] \\ L_2^{m \times n}[[a, b]^2 \times [c, d]] & L_2^{m \times n}[[a, b]^2 \times [c, d]^2] & L_2^{m \times n}[[a, b]^2 \times [c, d]^2] \\ L_2^{m \times n}[[a, b]^2 \times [c, d]] & L_2^{m \times n}[[a, b]^2 \times [c, d]^2] & L_2^{m \times n}[[a, b]^2 \times [c, d]^2] \end{bmatrix} =: \mathcal{N}_{2D}^{m \times n}[[a, b] \times [c, d]]$$

we define the associated 9-PI operator $\mathcal{P}[R] := \mathcal{P} \begin{bmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{bmatrix} : L_2^n[[a_1, b_1] \times [a_2, b_2]] \rightarrow L_2^m[[a_1, b_1] \times [a_2, b_2]]$ as

$$\begin{aligned} (\mathcal{P}[R]\mathbf{x})(s, r) &= R_{00}(s, r)\mathbf{x}(s, r) + \int_c^r R_{01}(s, r, \nu)\mathbf{x}(s, \nu)d\nu + \int_r^d R_{02}(s, r, \nu)\mathbf{x}(s, \nu)d\nu \\ &+ \int_a^s R_{10}(s, r, \theta)\mathbf{x}(\theta, r)d\theta + \int_a^s \int_c^r R_{11}(s, r, \theta, \nu)\mathbf{x}(\theta, \nu)d\nu d\theta + \int_a^s \int_r^d R_{12}(s, r, \theta, \nu)\mathbf{x}(\theta, \nu)d\nu d\theta \\ &+ \int_s^b R_{20}(s, r, \theta)\mathbf{x}(\theta, r)d\theta + \int_s^b \int_c^r R_{21}(s, r, \theta, \nu)\mathbf{x}(\theta, \nu)d\nu d\theta + \int_s^b \int_r^d R_{22}(s, r, \theta, \nu)\mathbf{x}(\theta, \nu)d\nu d\theta \end{aligned} \quad (\text{A.2})$$

for any $\mathbf{x} \in L_2^n[[a_1, b_2] \times [a_2, b_2]]$.

Note that, similar to how 3-PI operators can be seen as a generalization of matrices, operating on infinite-dimensional states $\mathbf{x}(s)$ instead of a finite-dimensional vectors x_i , 9-PI operators are a generalization of (a particular class of) tensors, operating on infinite-dimensional states $\mathbf{x}(s, r)$ instead of matrix-valued states x_{ij} . However, this comparison is not quite as easy to visualize as that between 3-PI operators and matrices, so we will mostly use 3-PI operators to illustrate the different properties of PI operators in the remaining sections.

Finally we define a general class of PI operators, encapsulating 3-PI operators and 9-PI operators, as well as matrices and ‘‘cross-operators’’. In particular, we consider operators defined on the set $Z^n[[a, b], [c, d]] := \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_s}[a, b] \\ L_2^{n_r}[c, d] \\ L_2^{n_2}[[a, b] \times [c, d]] \end{bmatrix}$, where $n := \{n_0, n_s, n_r, n_2\}$, with each element being a coupled state of finite-dimensional variables $x_0 \in \mathbb{R}^{n_0}$, 1D functions $\mathbf{x}_s \in L_2^{n_s}[a, b]$ and $\mathbf{x}_r \in L_2^{n_r}[a, b]$, and 2D functions $\mathbf{x}_2 \in L_2^{n_2}[[a, b] \times [c, d]]$.

Definition 3 (PI Operator). *For any operator $\mathcal{R} : Z^n[[a, b], [c, d]] \rightarrow Z^m[[a, b], [c, d]]$ with $m := \{m_0, m_s, m_r, m_2\}$ and $n := \{n_0, n_s, n_r, n_2\}$, we say that \mathcal{R} is a PI operator, denoted by $\mathcal{R} \in \Pi^{m \times n}$ if there exist parameters*

$$\mathcal{R} := \begin{bmatrix} R_{00} & R_{0s} & R_{0r} & R_{02} \\ R_{s0} & R_{ss} & R_{sr} & R_{s2} \\ R_{r0} & R_{rs} & R_{rr} & R_{r2} \\ R_{20} & R_{2s} & R_{2r} & R_{22} \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{m_0 \times n_0} & L_2^{m_s \times n_s}[a, b] & L_2^{m_r \times n_r}[c, d] & L_2^{m_2 \times n_2}[[a, b] \times [c, d]] \\ L_2^{m_s \times n_0}[a, b] & \mathcal{N}_{1D}^{m_s \times n_s}[a, b] & L_2^{m_s \times n_r}[[a, b] \times [c, d]] & \mathcal{N}_{1D \leftarrow 2D}^{m_s \times n_2}[[a, b], [c, d]] \\ L_2^{m_r \times n_0}[c, d] & L_2^{m_r \times n_s}[[a, b] \times [c, d]] & \mathcal{N}_{1D}^{m_r \times n_r}[c, d] & \mathcal{N}_{1D \leftarrow 2D}^{m_r \times n_2}[[c, d], [a, b]] \\ L_2^{m_2 \times n_0}[[a, b] \times [c, d]] & \mathcal{N}_{2D \leftarrow 1D}^{m_2 \times n_s}[[a, b], [c, d]] & \mathcal{N}_{2D \leftarrow 1D}^{m_2 \times n_r}[[c, d], [a, b]] & \mathcal{N}_{2D}^{m_2 \times n_2}[[a, b] \times [c, d]] \end{bmatrix} =: \mathcal{N}^{m \times n}[[a, b] \times [c, d]]$$

such that

$$\begin{aligned} \mathcal{R} &= (\mathcal{P}[R]\mathbf{x})(s, r) \\ &= \mathcal{P} \begin{bmatrix} R_{00} & R_{0s} & R_{0r} & R_{02} \\ R_{s0} & R_{ss} & R_{sr} & R_{s2} \\ R_{r0} & R_{rs} & R_{rr} & R_{r2} \\ R_{20} & R_{2s} & R_{2r} & R_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ \mathbf{x}_s \\ \mathbf{x}_r \\ \mathbf{x}_2 \end{bmatrix} \\ &:= \begin{bmatrix} R_{00}x_0 & + \int_a^b R_{0s}(s)\mathbf{x}_s(s)ds & + \int_c^d R_{0r}(r)\mathbf{x}_r(r)dr & + \int_a^b \int_c^d R_{02}\mathbf{x}_2(s, r)drds \\ R_{s0}(s)x_0 & + (\mathcal{P}[R_{ss}]\mathbf{x}_s)(s) & + \int_c^d R_{sr}(s, r)\mathbf{x}_r(r)dr & + (\mathcal{P}[R_{s2}]\mathbf{x}_2)(s) \\ R_{r0}(r)x_0 & + \int_a^b R_{rs}(s, r)\mathbf{x}_s(s)ds & + (\mathcal{P}[R_{rr}]\mathbf{x}_r)(r) & + (\mathcal{P}[R_{r2}]\mathbf{x}_2)(r) \\ R_{20}(s, r)x_0 & + (\mathcal{P}[R_{2s}]\mathbf{x}_s)(s, r) & + (\mathcal{P}[R_{2r}]\mathbf{x}_r)(s, r) & + (\mathcal{P}[R_{22}]\mathbf{x}_2)(s, r) \end{bmatrix}, \quad (\text{A.3}) \end{aligned}$$

for any $\mathbf{x} = \begin{bmatrix} x_0 \\ \mathbf{x}_s \\ \mathbf{x}_r \\ \mathbf{x}_2 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_s}[a, b] \\ L_2^{n_r}[c, d] \\ L_2^{n_2}[[a, b] \times [c, d]] \end{bmatrix} =: Z^n$, where for given parameters

$$P := \{P_0, P_1, P_2\} \in \{L_2^{m_s \times n_2}[[a, b] \times [c, d]], L_2^{m_s \times n_2}[[a, b]^2 \times [c, d]], L_2^{m_s \times n_2}[[a, b]^2 \times [c, d]]\} =: \mathcal{N}_{1D \leftarrow 2D}^{m_s \times n_2}[[a, b], [c, d]],$$

$$Q := \{Q_0, Q_1, Q_2\} \in \{L_2^{m_s \times n_2}[[a, b] \times [c, d]], L_2^{m_s \times n_2}[[a, b]^2 \times [c, d]], L_2^{m_s \times n_2}[[a, b]^2 \times [c, d]]\} =: \mathcal{N}_{2D \leftarrow 1D}^{m_s \times n_2}[[a, b], [c, d]],$$

we define

$$(\mathcal{P}[P]\mathbf{x}_2)(s) := \int_c^d \left[P_0(s, r)\mathbf{x}_2(s, r) + \int_a^s P_1(s, r, \theta)\mathbf{x}_2(\theta, r)d\theta + \int_s^b P_2(s, r, \theta)\mathbf{x}_2(\theta, r)d\theta \right] dr,$$

$$(\mathcal{P}[Q]\mathbf{x}_s)(s, r) := Q_0(s, r)\mathbf{x}_s(s) + \int_a^s Q_1(s, r, \theta)\mathbf{x}_s(\theta)d\theta + \int_s^b Q_2(s, r, \theta)\mathbf{x}_s(\theta)d\theta,$$

for any $\mathbf{x}_2 \in L_2^{n_2}[[a, b] \times [c, d]]$ and $\mathbf{x}_s \in L_2^{n_s}[a, b]$.

A.2 Addition of PI Operators

An obvious but crucial property of PI operators is that the sum of two PI operators (of appropriate dimensions) is again a PI operator.

Lemma 4. *For any PI parameters $Q, R \in \mathcal{N}^{m \times n}[[a, b], [c, d]]$, there exist unique parameters $P \in \mathcal{N}^{m \times n}[[a, b], [c, d]]$ such that*

$$\mathcal{P}[R] + \mathcal{P}[Q] = \mathcal{P}[P].$$

That is, for any $\mathbf{x} \in Z^n[[a, b], [c, d]]$,

$$((\mathcal{P}[Q] + \mathcal{P}[R])\mathbf{x})(s) = (\mathcal{P}[P]\mathbf{x})(s).$$

Proof. We outline the proof for 3-PI operators, for which it is easy to see that, by linearity of the integral,

$$\begin{aligned} & (\mathcal{P}_{\{R_0, R_1, R_2\}}\mathbf{x})(s) + (\mathcal{P}_{\{Q_0, Q_1, Q_2\}}\mathbf{x})(s) \\ &= [R_0(s) + Q_0(s)]\mathbf{x}(s) + \int_a^s [R_1(s, \theta) + Q_1(s, \theta)]\mathbf{x}(\theta)d\theta + \int_s^b [R_2(s, \theta) + Q_2(s, \theta)]\mathbf{x}(\theta)d\theta \\ &= (\mathcal{P}_{\{R_0+Q_0, R_1+Q_1, R_2+Q_2\}}\mathbf{x})(s). \end{aligned}$$

Similar results can be easily derived for more general PI operators. For a full proof, we refer to [5]. \square

Comparing the addition operation for 3-PI operators to that for matrices $A, B \in \mathbb{R}^{m \times n}$, we can draw direct parallels. In particular, where the sum $C = A + B$ of two matrices is simply computed by adding the elements $[C]_{ij} = [A]_{ij} + [B]_{ij}$ for each row i and column j , the sum of two 3-PI operators is computed by simply adding the values of the parameters $P(s, \theta) = Q(s, \theta) + R(s, \theta)$ at each position s and θ within the domain.

A.3 Composition of PI Operators

In addition to the sum of two PI operators being a PI operator, the composition of two PI operators can also be shown to be a PI operator, as stated in the following lemma:

Lemma 5. *For any PI parameters $R_1 \in \mathcal{N}^{m \times p}[[a, b], [c, d]]$ and $R_2 \in \mathcal{N}^{p \times n}[[a, b], [c, d]]$, there exist unique parameters $R_3 \in \mathcal{N}^{m \times n}[[a, b], [c, d]]$ such that*

$$\mathcal{P}[R_1] \circ \mathcal{P}[R_2] = \mathcal{P}[R_3].$$

That is, for any $\mathbf{x} \in Z^n[[a, b], [c, d]]$,

$$\left(\mathcal{P}[R_1](\mathcal{P}[R_2]\mathbf{x}) \right)(s) = (\mathcal{P}[R_3]\mathbf{x})(s).$$

Proof. We once again outline the proof only for 3-PI operators. For this, we define the indicator function

$$\mathbf{I}(s - \theta) = \begin{cases} 1, & \text{if } s \geq \theta \\ 0. & \text{else} \end{cases}$$

allowing us to write, e.g.

$$(\mathcal{P}_{\{R_0, R_1, R_2\}} \mathbf{x})(s) = R_0(s) \mathbf{x} + \int_a^b \left[\mathbf{I}(s - \theta) R_1(s, \theta) + \mathbf{I}(\theta - s) R_2(s, \theta) \right] \mathbf{x}(\theta) d\theta.$$

Furthermore, we have the following relations for the indicator function

$$\begin{aligned} \mathbf{I}(s - \eta) \mathbf{I}(\eta - \theta) &= \begin{cases} \mathbf{I}(s - \theta), & \text{if } \eta \in [\theta, s], \\ 0, & \text{else,} \end{cases} \\ \mathbf{I}(s - \eta) \mathbf{I}(\theta - \eta) &= \mathbf{I}(s - \theta) \mathbf{I}(\theta - \eta) + \mathbf{I}(\theta - s) \mathbf{I}(s - \eta) \end{aligned}$$

Using the first relation, it follows that for any $R_1, Q_1 \in L_2^{m_s \times n_s} [[a, b]^2]$,

$$\begin{aligned} \left(\mathcal{P}_{\{0, R_1, 0\}} (\mathcal{P}_{\{0, Q_1, 0\}} \mathbf{x}) \right) (s) &= \int_a^s R_1(s, \eta) \int_a^\eta Q_1(\eta, \theta) \mathbf{x}(\theta) d\theta d\eta \\ &= \int_a^b \int_a^b \mathbf{I}(s - \eta) \mathbf{I}(\eta - \theta) R_1(s, \eta) Q_1(\eta, \theta) \mathbf{x}(\theta) d\theta d\eta \\ &= \int_a^s \left[\int_\theta^s R_1(s, \eta) Q_1(\eta, \theta) d\eta \right] \mathbf{x}(\theta) d\theta = (\mathcal{P}_{\{0, P_{11}, 0\}} \mathbf{x})(s), \end{aligned}$$

where $P_{11}(s, \theta) := \int_\theta^s R_1(s, \eta) Q_1(\eta, \theta) d\eta$. Similarly, we can show that

$$\begin{aligned} \left(\mathcal{P}_{\{0, R_1, 0\}} (\mathcal{P}_{\{0, 0, Q_2\}} \mathbf{x}) \right) (s) &= \int_a^s \left[\int_a^\theta R_1(s, \eta) Q_2(\eta, \theta) d\eta \right] \mathbf{x}(\theta) d\theta + \int_s^b \left[\int_a^s R_1(s, \eta) Q_2(\eta, \theta) d\eta \right] \mathbf{x}(\theta) d\theta \\ \left(\mathcal{P}_{\{0, 0, R_2\}} (\mathcal{P}_{\{0, Q_1, 0\}} \mathbf{x}) \right) (s) &= \int_a^s \left[\int_s^b R_2(s, \eta) Q_1(\eta, \theta) d\eta \right] \mathbf{x}(\theta) d\theta + \int_s^b \left[\int_\theta^b R_2(s, \eta) Q_1(\eta, \theta) d\eta \right] \mathbf{x}(\theta) d\theta \\ \left(\mathcal{P}_{\{0, 0, R_2\}} (\mathcal{P}_{\{0, 0, Q_2\}} \mathbf{x}) \right) (s) &= \int_s^b \left[\int_s^\theta R_2(s, \eta) Q_2(\theta, \eta) d\eta \right] \mathbf{x}(\theta) d\theta = (\mathcal{P}_{\{0, 0, P_{22}\}} \mathbf{x})(s), \end{aligned}$$

proving that the composition of lower-triangular and upper-triangular partial integrals can always be expressed as partial integrals as well. It is also easy to see that

$$\begin{aligned} \left(\mathcal{P}_{\{R_0, 0, 0\}} (\mathcal{P}_{\{0, Q_1, Q_2\}} \mathbf{x}) \right) (s) &= \int_a^s R_0(s) Q_1(s, \theta) \mathbf{x}(\theta) d\theta + \int_s^b R_0(s) Q_2(s, \theta) \mathbf{x}(\theta) d\theta = (\mathcal{P}_{\{0, P_{01}, P_{02}\}} \mathbf{x})(s), \\ \left(\mathcal{P}_{\{0, R_1, R_2\}} (\mathcal{P}_{\{Q_0, 0, 0\}} \mathbf{x}) \right) (s) &= \int_a^s R_1(s, \theta) Q_0(\theta) \mathbf{x}(\theta) d\theta + \int_s^b R_2(s, \theta) Q_2(\theta) \mathbf{x}(\theta) d\theta = (\mathcal{P}_{\{0, P_{10}, P_{20}\}} \mathbf{x})(s), \end{aligned}$$

from which it follows that the composition of any 3-PI operators can be expressed as a 3-PI operator as well. Moreover, since we can repeat these steps along any spatial directions, this result extends to more general (2D) PI operators as well. For a full proof, we refer to [].

□

We note again the similarity to matrices: just like the product of two lower triangular matrices L_1, L_2 is a lower triangular matrix L_3 , the composition of two lower-triangular 3-PI operators $\mathcal{P}_{\{0, R_1, 0\}}, \mathcal{P}_{\{0, Q_1, 0\}}$ is also a lower-triangular 3-PI operator $\mathcal{P}_{\{0, P_{11}, 0\}}$. Similarly, the product of two upper-triangular 3-PI operators $\mathcal{P}_{\{0, 0, R_2\}}, \mathcal{P}_{\{0, 0, Q_2\}}$ is also an upper-triangular 3-PI operator $\mathcal{P}_{\{0, 0, P_{22}\}}$, but the composition of lower- and upper-triangular PI operators need not be lower- or upper-triangular – just as with matrices. Finally, the composition of a diagonal operator $\mathcal{P}_{\{R_0, 0, 0\}}$ with a lower- or upper-diagonal PI operator is also respectively lower- or upper-diagonal.

A.4 Adjoint of PI Operators

To define the adjoint of a PI operator $\mathcal{R} \in \Pi^{\text{m} \times \text{normaln}}$, we first recall the definition of the function space that these operators map: $Z^n[[a, b], [c, d]] := \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_s}[a, b] \\ L_2^{n_r}[c, d] \\ L_2^{n_2}[[a, b] \times [c, d]] \end{bmatrix}$, where $\mathbf{n} := \{n_0, n_s, n_r, n_2\}$, with each element being a coupled state of finite-dimensional variables $x_0 \in \mathbb{R}^{n_0}$, 1D functions $\mathbf{x}_s \in L_2^{n_s}[a, b]$ and $\mathbf{x}_r \in L_2^{n_r}[c, d]$, and 2D functions $\mathbf{x}_2 \in L_2^{n_2}[[a, b] \times [c, d]]$. We endow this space with the inner product

$$\begin{aligned} \langle \mathbf{x}, \mathbf{y} \rangle_Z &= \langle x_0, y_0 \rangle + \langle \mathbf{x}_s, \mathbf{y}_s \rangle_{L_2} + \langle \mathbf{x}_r, \mathbf{y}_r \rangle_{L_2} + \langle \mathbf{x}_2, \mathbf{y}_2 \rangle_{L_2} \\ &= x_0^T y_0 + \int_a^b [\mathbf{x}_s(s)]^T \mathbf{y}(s) ds + \int_c^d [\mathbf{x}_r(r)]^T \mathbf{y}(r) dr + \int_a^b \int_c^d [\mathbf{x}_2(s, r)]^T \mathbf{y}(s, r) dr ds \end{aligned}$$

Defining this inner product, we can also define the adjoint of PI operators.

Lemma 6. *For any PI parameters $R \in \mathcal{N}^{\text{m} \times \text{n}}[[a, b], [c, d]]$, there exist unique parameters $Q \in \mathcal{N}^{\text{n} \times \text{m}}[[a, b], [c, d]]$ such that*

$$(\mathcal{P}[R])^* = \mathcal{P}[Q],$$

where \mathcal{P}^* denotes the adjoint of a PI operator \mathcal{P} . That is, for any $\mathbf{x} \in Z^n[[a, b], [c, d]]$ and $\mathbf{y} \in Z^m[[a, b], [c, d]]$,

$$\langle \mathcal{P}[R]\mathbf{x}, \mathbf{y} \rangle_Z = \langle \mathbf{x}, \mathcal{P}[Q]\mathbf{y} \rangle_Z.$$

Proof. We outline the proof only for 4-PI operators. In particular, let $\mathbf{n} = \{n_0, n_1, 0, 0\}$ and $\mathbf{m} = \{m_0, m_1, 0, 0\}$, and let $B = \begin{bmatrix} P & Q_1 \\ Q_2 & \{R_0, R_1, R_2\} \end{bmatrix}$ define a 4-PI operator $\mathcal{P}[B] \in \Pi^{\text{n} \times \text{m}}$. Define $\hat{B} = \begin{bmatrix} \hat{P} & \hat{Q}_1 \\ \hat{Q}_2 & \{\hat{R}_0, \hat{R}_1, \hat{R}_2\} \end{bmatrix}$, where

$$\begin{aligned} \hat{P} &= P^T, & \hat{Q}_1(s) &= Q_2^T(s), & \hat{R}_1(s, \theta) &= R_2^T(\theta, s), \\ \hat{Q}_2(s) &= Q_1^T(s), & \hat{R}_0(s) &= R_0^T(s), & \hat{R}_2(s, \theta) &= R_1^T(\theta, s), \end{aligned}$$

Then, for arbitrary $\mathbf{x} = \begin{bmatrix} x_0 \\ \mathbf{x}_1 \end{bmatrix} \in Z^n$ and $\mathbf{y} = \begin{bmatrix} y_0 \\ \mathbf{y}_1 \end{bmatrix} \in Z^m$, we note that

$$\langle \mathcal{P}[B]\mathbf{x}, \mathbf{y} \rangle_Z = [P x_0]^T y_0 + \left[\int_a^b Q_1(s) \mathbf{x}_1(s) ds \right]^T y_0 + \int_a^s [Q_2(s) x_0]^T \mathbf{y}_1(s) ds + \int_a^b [R_0(s) \mathbf{x}_1(s)]^T \mathbf{y}_1(s) ds$$

$$\begin{aligned}
& + \int_a^b \left[\left(\int_a^b \mathbf{I}(s-\theta)R_1(s,\theta) + \mathbf{I}(\theta-s)R_2(s,\theta) \right) \mathbf{x}_1(\theta) d\theta \right]^T \mathbf{y}_1(s) ds \\
& = x_0 [P^T y_0] + x_0^T \left[\int_a^b Q_2^T(s) \mathbf{y}_1(s) ds \right] + \int_a^b \mathbf{x}_1^T(s) [Q_1^T(s) y_0] ds + \int_a^b \mathbf{x}_1^T(s) [R_0^T(s) \mathbf{y}_1(s)] ds \\
& \quad + \int_a^b \mathbf{x}_1^T(s) \left[\left(\int_a^b \mathbf{I}(-\theta-s)R_1(\theta,s) + \mathbf{I}(s-\theta)R_2(\theta,s) \right) \mathbf{y}_1(\theta) d\theta \right] ds \\
& = \left\langle \mathbf{x}, \mathcal{P}[\hat{B}]\mathbf{y} \right\rangle_Z.
\end{aligned}$$

□

We note again the similarities with matrices: Just like the adjoint of a matrix can be determined by switching the rows and columns, the adjoint of a 3-PI operator is determined by switching the primary and dual variables (s, θ) , as well as switching the lower- and upper-triangular parts.

A.5 Inversion of PI operators

In this section, we focus on constructing the controller from the feasible solutions \mathcal{P} and \mathcal{Z} to the LPIs described in previous sections. The controller gains \mathcal{K} is given by the relation $\mathcal{K} = \mathcal{Z}\mathcal{P}^{-1}$. However, \mathcal{P}^{-1} , although is a 4-PI operator, may not have polynomial parameters. Hence the inverse is calculated numerically. First, we propose a numerical method to find the inverse of \mathcal{P} . Then, we use the numerical inverse to construct the controller gains. To find the inverse of a 4-PI operator, $\mathcal{P} \left[\begin{smallmatrix} P, & Q \\ Q^T, & \{R_i\} \end{smallmatrix} \right]$, we first find the inverse of 3-PI operators of the form $\mathcal{P}_{\{I, H_1, R_2\}}$ and then express the inverse of $\mathcal{P} \left[\begin{smallmatrix} P, & Q \\ Q^T, & \{R_i\} \end{smallmatrix} \right]$ in terms of $\mathcal{P}_{\{I, H_1, H_2\}}^{-1}$ where H_i are dependent on the parameters $\{P, Q, R_i\}$.

A.5.1 Inversion of 3-PI operators

First, we note that any matrix-valued polynomial $H(s, \theta)$ can be factored as $F(s)G(\theta)$. Then, for any given 3-PI operator $\mathcal{P}_{\{I, H_1, H_2\}}$ with matrix-valued polynomial parameters H_1 and H_2 , we have

$$\begin{aligned}
\mathcal{P}_{\{I, H_1, H_2\}} & = \mathcal{P}_{\{I, -F_1 G_1, -F_2 G_2\}}, \text{ where} \\
H_i(s, \theta) & = -F_i(s)G_i(\theta),
\end{aligned}$$

for some matrix-valued polynomials F_i and G_i . We can now find an inverse for $\mathcal{P}_{\{I, H_1, H_2\}}$ using the following result.

Lemma 7. *Suppose $F_1 : [a, b] \rightarrow \mathbb{R}^{n \times p}$, $G_1 : [a, b] \rightarrow \mathbb{R}^{p \times n}$, $F_2 : [a, b] \rightarrow \mathbb{R}^{n \times q}$, $G_2 : [a, b] \rightarrow \mathbb{R}^{q \times n}$ and U is the unique function that satisfies the equation*

$$U(s) = I_{(p+q)} + \int_a^s U(t) \begin{bmatrix} G_1(t)F_1(t) & G_1(t)F_2(t) \\ -G_2(t)F_1(t) & -G_2(t)F_2(t) \end{bmatrix} dt,$$

where U is partitioned as

$$U = \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{bmatrix}, \quad U_{11}(s) \in \mathbb{R}^{p \times p}, U_{22}(s) \in \mathbb{R}^{q \times q}.$$

Then, the 3-PI operator $\mathcal{P}_{\{I, -F_1 G_1, -F_2 G_2\}}$ is invertible if and only if $U_{22}(b)$ is invertible and

$$(\mathcal{P}_{\{I, -F_1 G_1, -F_2 G_2\}})^{-1} = \mathcal{P}_{\{I, L_1, L_2\}},$$

where

$$L_1(s, t) = \begin{bmatrix} F_1(s) & F_2(s) \end{bmatrix} U(s) V(t) \begin{bmatrix} G_1(t) \\ -G_2(t) \end{bmatrix} - L_2(s, t), \quad (\text{A.4a})$$

$$L_2(s, t) = - \begin{bmatrix} F_1(s) & F_2(s) \end{bmatrix} U(s) P V(t) \begin{bmatrix} G_1(t) \\ -G_2(t) \end{bmatrix}, \quad (\text{A.4b})$$

$$P = \begin{bmatrix} 0_{p \times p} & 0_{p \times q} \\ U_{22}(b)^{-1} U_{21}(b) & I_q \end{bmatrix},$$

and V is the unique function satisfying the equation

$$V(t) = I_{(p+q)} - \int_a^t \begin{bmatrix} G_1(s) F_1(s) & G_1(s) F_2(s) \\ -G_2(s) F_1(s) & -G_2(s) F_2(s) \end{bmatrix} V(s) ds.$$

Proof. Proof can be found in [3, Chapter IX.2]. \square

Using Lemma 2.2. of [3, Chapter IX.2], we can use an iterative process and numerical integration to approximate U and V functions in the above result to find an inverse for the 3-PI operators of the form $\mathcal{P}_{\{I, R_1, R_2\}}$ where R_i are matrix-valued polynomials. By extension, given an R_0 invertible, we can obtain the inverse of a general 3-PI operator as shown below.

Corollary 8. *Suppose $R_0 : [a, b] \rightarrow \mathbb{R}^{n \times n}$, $R_1, R_2 : [a, b]^2 \rightarrow \mathbb{R}^{n \times n}$, with R_0 invertible on $[a, b]$. Then, the inverse of the 3-PI operator, $\mathcal{P}_{\{R_i\}}$, is given by $\mathcal{P}_{\{\hat{R}_0, \hat{R}_1, \hat{R}_2\}}$ where*

$$\hat{R}_0(s) = R_0(s)^{-1}, \quad \hat{R}_i(s, \theta) = L_i(s, \theta) \hat{R}_0(\theta), \quad i \in \{1, 2\},$$

where L_1 and L_2 are as defined in (A.4) for functions F_i and G_i such that $F_i(s) G_i(\theta) = R_0(s)^{-1} R_i(s, \theta)$.

Proof. Let R_i be as stated above.

$$\begin{aligned} & (\mathcal{P}_{\{R_0(s), R_1(s, \theta), R_2(s, \theta)\}})^{-1} \\ &= (\mathcal{P}_{\{R_0(s), 0, 0\}} \mathcal{P}_{\{I, R_0(s)^{-1} R_1(s, \theta), R_0(s)^{-1} R_2(s, \theta)\}})^{-1} \\ &= (\mathcal{P}_{\{I, R_0(s)^{-1} R_1(s, \theta), R_0(s)^{-1} R_2(s, \theta)\}})^{-1} (\mathcal{P}_{\{R_0(s), 0, 0\}})^{-1} \\ &= \mathcal{P}_{\{I, L_1(s, \theta), L_2(s, \theta)\}} \mathcal{P}_{\{R_0(s)^{-1}, 0, 0\}} \\ &= \mathcal{P}_{\{R_0(s)^{-1}, L_1(s, \theta) R_0(\theta)^{-1}, L_2(s, \theta) R_0(\theta)^{-1}\}}. \end{aligned}$$

where L_i are obtained from the Lemma 7 and the composition of PI operators is performed using the formulae in [8]. \square

Note the above expressions for the inverse are exact, however, in practice, R_0^{-1} may not have an analytical expression (or very hard to determine). Thus, finding F_i and G_i such that $F_i(s)G_i(\theta) = R_0(s)^{-1}R_i(s, \theta)$ may not be possible. To overcome this problem, we approximate R_0^{-1} by a polynomial which guarantees that $R_0^{-1}R_i$ are polynomials and can be factorized into F_i and G_i . Using this approach, we can find an approximate inverse for $\mathcal{P}_{\{R_i\}}$ using Lemma 7.

A.5.2 Inversion of 4-PI operators

Given, R_0, R_1, R_2 with R_0 invertible, we proposed a way to find the inverse of the operator $\mathcal{P}_{\{R_i\}}$. Now, we use this method to find the inverse of a 4-PI operator $\mathcal{P} \left[\begin{matrix} P, & Q \\ Q^T, & \{R_i\} \end{matrix} \right]$. Given P , Q and R_i with invertible P and R_0 , define the 3-PI operator $\mathcal{P}_{\{H_i\}}$ with parameters H_i

$$H_0(s) = R_0(s), \quad H_i(s, \theta) = R_i(s, \theta) - Q(s)^T P^{-1} Q(\theta), \quad i \in \{1, 2\}.$$

Next we suppose that $\mathcal{P}_{\{H_i\}}$ is invertible. We will use the inverse of this 3-PI operator to find the inverse of the 4-PI operator as follows.

Corollary 9. *Suppose $P \in \mathbb{R}^{m \times m}$, $Q : [a, b] \rightarrow \mathbb{R}^{m \times n}$, $R_0 : [a, b] \rightarrow \mathbb{R}^{n \times n}$ and $R_1, R_2 : [a, b]^2 \rightarrow \mathbb{R}^{n \times n}$ are matrices and matrix-valued polynomials with P and $\mathcal{P}_{\{H_i\}}$ invertible such that $(\mathcal{P}_{\{H_i\}})^{-1} = \mathcal{P}_{\{L_i\}}$, where*

$$H_0(s) = R_0(s), \quad H_i(s, \theta) = R_i(s, \theta) - Q(s)^T P^{-1} Q(\theta),$$

for $i \in \{1, 2\}$. Then, the inverse of $\mathcal{P} \left[\begin{matrix} P, & Q \\ Q^T, & \{R_i\} \end{matrix} \right]$ is given by

$$\left(\mathcal{P} \left[\begin{matrix} P, & Q \\ Q^T, & \{R_i\} \end{matrix} \right] \right)^{-1} = \mathcal{P} \left[\begin{matrix} \hat{P}, & \hat{Q} \\ \hat{Q}^T, & \{\hat{R}_i\} \end{matrix} \right],$$

where $\hat{P}, \hat{Q}, \hat{R}_i$ are defined as

$$\begin{aligned} \hat{R}_0(s) &= R_0(s)^{-1}, \quad \hat{R}_i(s, \theta) = L_i(s, \theta), \quad \text{for } i \in \{1, 2\}, \\ \hat{P} &= P^{-1} + P^{-1} \left(\int_a^b (Q(s)\hat{R}_0(s)Q(s)^T \right. \\ &\quad \left. + \int_s^b Q(\theta)L_1(\theta, s)Q(\theta)^T d\theta \right. \\ &\quad \left. + \int_a^s Q(\theta)L_2(\theta, s)Q(\theta)^T d\theta \right) ds P^{-1} \\ \hat{Q}(s) &= -P^{-1} \left(Q(s)\hat{R}_0(s) + \int_s^b Q(\theta)L_1(\theta, s)d\theta \right. \\ &\quad \left. + \int_a^s Q(\theta)L_2(\theta, s)d\theta \right). \end{aligned} \tag{A.5}$$

Proof. This result can be proved by performing the composition of \mathcal{P} and its inverse \mathcal{P}^{-1} where the inverse is provided by the formulae stated above. Using composition formulae for 4-PI operators (See [8]), we show that the resulting operator is an identity. \square

A.6 Composition of Differential and PI operator

Given the well-known relationship between integrals and derivatives (think e.g. the fundamental theorem of calculus), it is natural to assume that the composition of a differential operator and a PI operator may be expressed as a PI operator as well. Unfortunately, this is not true in general, as e.g. the operator \mathcal{P} defined as $(\mathcal{P}\mathbf{v})(s) = P(s)\mathbf{v}(s)$ is a PI operator, but there clearly does not exist a PI operator \mathcal{Q} such that $\partial_s(\mathcal{P}\mathbf{v})(s) = (\mathcal{Q}\mathbf{v})(s)$. Nevertheless, if the function \mathbf{v} is differentiable, i.e. $\mathbf{v} \in H_1$, then we can always express the derivative $(\mathcal{P}\mathbf{v})(s)$ for a PI operator \mathcal{P} in terms of $\mathbf{v}(s)$ and $\partial_s\mathbf{v}(s)$ as $\partial_s(\mathcal{P}\mathbf{v})(s) = (\mathcal{Q}[\partial_s\mathbf{v}^v])(s)$, as we show in the next lemma.

Lemma 10. 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

$$\partial_s\mathcal{P} \begin{bmatrix} P, Q_1 \\ Q_2, \{R_i\} \end{bmatrix} = \mathcal{P} \begin{bmatrix} 0, 0 \\ \bar{Q}_2, \{\bar{R}_i\} \end{bmatrix} \quad (\text{A.6})$$

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 $x \in \mathbb{R}^m$, $\mathbf{x} \in H_1^n$,

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

Proof. The result can be easily derived using the Leibniz integral rule, stating that for any $F \in H_1[a, b]^2$,

$$\frac{d}{ds} \left(\int_{L(s)}^{U(s)} F(s, \theta) d\theta \right) = F(s, U(s)) \frac{d}{ds} U(s) - F(s, L(s)) \frac{d}{ds} L(s) + \int_{L(s)}^{U(s)} \frac{d}{ds} F(s, \theta) d\theta.$$

For a similar result for PI operators in 2D, we refer to [4]. □

A.7 Matrix Parametrization of Positive Definite PI Operators

In order to be able to solve optimization programs involving PI operator \mathcal{P} , we need to be able to enforce positivity constraints $\mathcal{P} \succcurlyeq 0$. For this, we parameterize PI operators by positive matrices, expanding them as $\mathcal{P} = \mathcal{Z}^* P \mathcal{Z}$ for a fixed operator \mathcal{Z} , and positive semidefinite matrix $P \succcurlyeq 0$. The following theorem provides a sufficient condition for positivity of a 4-PI operator defined as

$$(\mathcal{P} \begin{bmatrix} P, Q_1 \\ Q_2, \{R_i\} \end{bmatrix} \mathbf{x})(s) = \begin{bmatrix} Px_0 + \int_a^b Q_1(s) \mathbf{x}_1(s) ds \\ Q_2(s)x_0 + R_0(s)\mathbf{x}_1(s) + \int_a^s R_1(s, \theta)\mathbf{x}_1(\theta)d\theta + \int_s^b R_2(s, \theta)\mathbf{x}_1(\theta)d\theta \end{bmatrix} \quad (\text{A.7})$$

for $\mathbf{x} = \begin{bmatrix} x_0 \\ \mathbf{x}_1 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a, b] \end{bmatrix}$. This result allows us to parameterize a cone of positive PI operators as positive matrices, implement LPI constraints as LMI constraints, allowing us to solve optimization problems with PI operators using semi-definite programming solvers.

Theorem 11. *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*

$$\begin{aligned}
P &= T_{11} \int_a^b g(s) ds, \\
Q(\eta) &= g(\eta) T_{12} Z_1(\eta) + \int_\eta^b g(s) T_{13} Z_2(s, \eta) ds + \int_a^\eta g(s) T_{14} Z_2(s, \eta) ds, \\
R_1(s, \eta) &= g(s) Z_1(s)^\top T_{23} Z_2(s, \eta) + g(\eta) Z_2(\eta, s)^\top T_{42} Z_1(\eta) + \int_s^b g(\theta) Z_2(\theta, s)^\top T_{33} Z_2(\theta, \eta) d\theta \\
&\quad + \int_\eta^s g(\theta) Z_2(\theta, s)^\top T_{43} Z_2(\theta, \eta) d\theta + \int_a^\eta g(\theta) Z_2(\theta, s)^\top T_{44} Z_2(\theta, \eta) d\theta, \\
R_2(s, \eta) &= g(s) Z_1(s)^\top T_{32} Z_2(s, \eta) + g(\eta) Z_2(\eta, s)^\top T_{24} Z_1(\eta) + \int_\eta^b g(\theta) Z_2(\theta, s)^\top T_{33} Z_2(\theta, \eta) d\theta \\
&\quad + \int_s^\eta g(\theta) Z_2(\theta, s)^\top T_{34} Z_2(\theta, \eta) d\theta + \int_a^s g(\theta) Z_2(\theta, s)^\top T_{44} Z_2(\theta, \eta) d\theta, \\
R_0(s) &= g(s) Z_1(s)^\top T_{22} Z_1(s). \tag{A.8}
\end{aligned}$$

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 (A.7) is positive semidefinite, i.e. $\left\langle \mathbf{x}, \mathcal{P} \begin{bmatrix} P, Q_1 \\ Q_2, \{R_i\} \end{bmatrix} \mathbf{x} \right\rangle \geq 0$ for all $\mathbf{x} \in \mathbb{R}^m \times L_2^n[a, b]$.

To see the PIETOOLS implementation, check Section 7.2.2. For an extension of this result to 2D PI operators, we refer to [4].

Appendix B

Troubleshooting

This appendix is dedicated to tackling issues related to installation and setting up of PIETOOLS 2022. Additionally, we discuss some common issues users may run into while solving LPIs.

B.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_2022**. 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_2022, 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, sdpnal) to your MATLAB path, and extract the files in **PIETOOLS_2022.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.

B.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 2022 relies on recently released SOSTOOLS 4.00 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 [10].

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.

B.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

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 . In *[1991] Proceedings of the 30th IEEE Conference on Decision and Control*, pages 1227–1232 vol.2. IEEE, 1991.
- [3] Israel Gohberg, Seymour Goldberg, and Marius A Kaashoek. *Classes of linear operators*, volume 63. Birkhäuser, 2013.
- [4] Declan S. Jagt and Matthew M. Peet. A PIE representation of coupled 2D PDEs and stability analysis using LPIs. In *Proceedings of the American Control Conference*, 2021.
- [5] M. Peet. A partial integral equation representation of coupled linear PDEs and scalable stability analysis using LMIs. *Automatica*, 2020. Submitted.
- [6] 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.
- [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.
- [9] Sachin Shivakumar, Amritam Das, Siep Weiland, and Matthew M Peet. Duality and H_∞ -optimal control of coupled ODE-PDE systems. *arXiv preprint arXiv:2004.03638*, 2020.
- [10] Jos F Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. In *Optimization methods and software*, volume 11, pages 625–653. Taylor & Francis, 1999.