PIETOOLS 2024: User Manual

S. Shivakumar A. Das D. Braghini D. Jagt Y. Peet
M. Peet

January 20, 2025

Copyrights and license information

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 $[a, b] \to \mathbb{R}^n$
$RL^{m,n}[a, b]$	$\mathbb{R}^m \times L_2^n[a, b]$
$H_k^n[a, b]$	$\{f \in L_2^n[a, b] \mid \partial_s^i f \in L_2^n[a, b] \forall i \leq k\}$
$0_{m \times n}$	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 $f : C \to X$, $\Delta_a(f) = f(a)$, for $a \in C$
$\mathcal{B}(X, Y)$	Space of bounded linear operators from X to Y

Contents

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 [11], 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 2024 Release Notes

PIETOOLS 2024 introduces several functional and quality of life improvements to PIETOOLS, including a unified and improved command-line interface for declaring both 1D and 2D PDEs, support for H_2 norm and improved demos.

Major Updates:

1. PIETOOLS and PIESIM now include native support for 2D PDEs defined on a hyper-rectangle.
   - 2D PDEs can be declared using the command line interface
   - 2D PDEs can be simulated in PIESIM (See Chap 6)
   - 2D opvars and decision variables can be created using `lpivar`, `poslpivar`, etc
   - function-valued inputs and boundary conditions can be declared (See Chap 4)
   - the gain analysis and estimator design scripts now support 2D PDEs and PIEs
   - Demo 9 now illustrates L_2-gain analysis for a 2D PDE
2. Support for H_2 analysis, estimation, and control are now supported (See Chap 13).
   - New scripts include '`lpiscript(pie,h2norm)`' and '`lpiscript(pie,h2-observer)`'
   - Demo 8 now illustrates H_2 norm analysis
3. Scripts now allow for non-coercive Lyapunov function candidates, improving the accuracy of gain analysis and performance of optimal controller/observer synthesis.
4. All Demos have been streamlined and shortened to better illustrate functionality and workflow.
   - Two new demos have been added illustrating analysis of 2D PDEs and H2 norm bounding

Minor Updates

1. Improved operator inversion routines have been incorporated in getobserver/getcontroller to allow for more numerically reliable controller/observer gains.
2. Several routines in declaring and solving LPIs have been updated and renamed.
   - `sosprogram` is now `lpiprogram` and updated to require that the spatial domain must be specified at instantiation
   - `sosdecvar` is now `lpidecvar` and `sossetobj` is now `lpisetobj`
   - Routines `lpi_eq` and `lpi_ineq` have been updated to allow for scalar equality and inequality constraints to be declared as well
   - `sossolve` is now `lpisolve`
   - `sosgetsol` is now `lpigetsol` and allows one to access operator-valued variables directly
3. The function `piess` now allows one to construct a PIE object directly, using similar state-space syntax to the Matlab command '`ss`' – e.g. `piess(T,A,B,C,D)` where `T,A,B,C,D` are opvar objects.

4. The function `pielft` allows for the closed-loop interconnection (LFT) of two PIE objects. e.g. `pielft(pie1,pie2)`.

5. The directory `lpi_programming` has been added. Some users may need to update their Matlab path definitions accordingly, simply run the script `pietools_path_update`.

1.2 Installing PIETOOLS

PIETOOLS 2024 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 2024 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 2024 (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 2024 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_2024.zip** from this link, unzip, and add to MATLAB path.

Chapter 2

Scope of PIETOOLS

In this chapter, we briefly motivate 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 matrices. The development of efficient interior-point methods for SDP 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 plots, 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 limelight. The sum-of-squares (SOS) optimization methods found application in control theory, for example searching for Lyapunov functions or finding bounds on singular values. This gave rise to many toolboxes such as SOSTOOLS [6], SOSOPT [7], etc., that can handle SOS polynomials in MATLAB. However, unlike the LMI methods for linear ODEs, SOS methods for analysis and control of PDEs still required ad-hoc interventions. For example, searching for a Lyapunov function that certifies stability of a PDE, one usually hits a roadblock in the form of boundary conditions, requiring the use of e.g. integration by parts, Poincaré inequality, or Hölder's Inequality to resolve.

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 alternative representation of dynamical systems, parameterized by Partial Integral (PI) operators. The PIE representation can be used to represent a broad class of distributed parameter systems and is algebraic – eliminating the use of boundary conditions and continuity constraints [8], [1]. Consequently, many LMI-based methods for analysis of finite-dimensional systems can be extended to infinite-dimensional ones using PIEs. The PIETOOLS software offers the tools to do exactly that – making e.g. stability analysis, controller synthesis, and simulation of linear infinite-dimensional systems as intuitive as it is for finite-dimensional ones.

2.2 PIETOOLS for Analysis, Control, and Simulation of ODE-PDE Systems

Using PIETOOLS 2024 for controlling and simulating ODE-PDE models has been made intuitive, requiring little knowledge of the mathematical details on PIE operators. To illustrate, let us see how we can simulate a simple ODE-PDE model, and synthesize a controller.

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

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

with controlled input u, and PDE dynamics given by a one-dimensional wave equation

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

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. For this example, we do so by introducing $\phi = (\partial_s\mathbf{x}, \dot{\mathbf{x}})$, so that the dynamics of ϕ are governed by

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

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

$$z(t) = \begin{bmatrix} r(t) \\ u(t) \end{bmatrix} = \begin{bmatrix} \mathbf{x}(t,1) - \mathbf{x}(t,0) \\ u(t) \end{bmatrix} = \begin{bmatrix} \int_0^1 \phi_1(t,s)ds \\ u(t) \end{bmatrix}. \tag{2.4}$$

To define the presented model in MATLAB, we first declare the spatial and temporal variable s and t, and create the state, input, and output variables, using the `pvar` and `pde_var` functions:

```
>> pvar s t
>> x = pde_var();       phi = pde_var(2,s,[0,1]);
>> w = pde_var('in');   u = pde_var('control');
>> z = pde_var('out',2);
```

Then, we can define an equation in terms of these variables as a `pde_struct` object, using standard operators such as `'+','-','*','diff','subs','int'`, etc., declaring e.g. our wave equation for $c = 1$ and $b = 0.01$ as:

```
>> b = 0.01;    c = 1;
>> 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 = [eq_dyn;eq_out];
```

2.2.2 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 and Neumann boundary conditions,

$$\dot{\mathbf{x}}(t, s = 0) = 0, \qquad \partial_s \mathbf{x}(t, s = 1) = x(t),$$

we can simply call

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

Once the full system is declared, we clean up the structure and fill in any gaps by calling the function `initialize` as

```
» odepde = initialize(odepde);
```

Whenever equations are successfully initialized, a summary of the encountered states, inputs, and outputs is displayed in the Command Window. To verify if PIETOOLS got the right equations, the user just needs to type the name of the system variable ("`odepde`" in this example) in the command window without a semicolon for PIETOOLS to display the added equations. We encourage the user to always check the equations before proceeding.

2.2.3 Simulating ODE-PDE Models

Having declared an ODE-PDE system, one of the first things a practitioner may do is to simulate the system. If you look at traditional PDE literature, a big challenge in simulation of PDEs is that every different kind of PDE requires different techniques to discretize. In PIETOOLS, however, there is only one command to simulate any linear ODE-PDE coupled system of your choice:

```
» solution = PIESIM(odepde, opts, uinput);
```

Here, aside from the desired system to simulate (our `odepde`), we have to declare two additional arguments. The first are the options for simulation, determining e.g. the number of spatial and temporal points in discretization of the solution. For our illustration, we pass the following options to the function, informing PIESIM not to automatically plot the solution, to use 8 Chebyshev polynomials in expanding the solution in space, and to simulate the solutions up to $t = 12$ seconds with a time-step of $3 \cdot 10^{-2}$ seconds:

```
» opts.plot = 'no';
» opts.N = 8;
» opts.tf = 12;
» opts.dt = 3*1e-2;
```

Next, in order to simulate solutions, we must of course also pass values for all of the inputs, as well as initial values of the states, which is all done with the input `uinput`. In our case, we wish to simulate the zero-state response of the system, perturbed by an exponentially decaying sinusoidal signal, which we specify as

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

Note that we set the control input to zero to simulate an open-loop response.

For more details on the PIESIM input arguments, we refer the reader to Chapter 6. With the output structure `solution`, PIESIM returns discretized time-dependent arrays corresponding to the time vector used in the simulations and the resulting state variables and output. The simulated evolution of the second PDE state variable $\phi_2(t) = \dot{\mathbf{x}}(t)$ and regulated output $r(t)$ for our example is depicted in Figure 2.1.

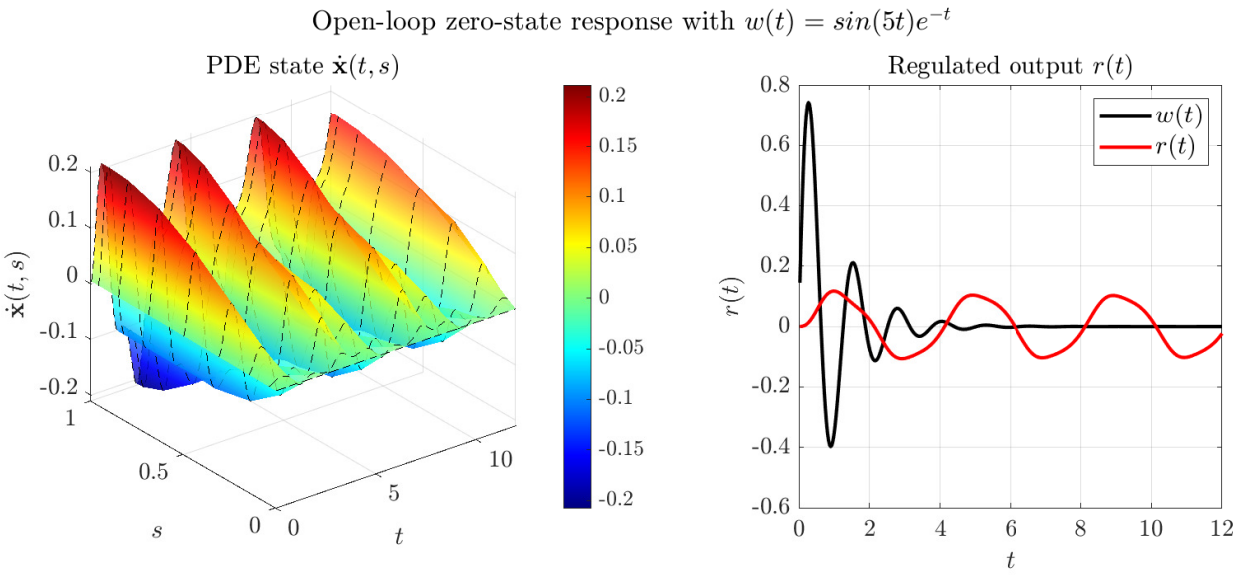


Figure 2.1: Transient response of the state variable $\dot{\mathbf{x}}(t,s)$ and regulated output $r(t)$ by simulating the ODE-PDE model (2.1) and (2.3) with $u(t) = 0$ for external disturbance $w(t) = sin(5t)e^{-t}$.

2.2.4 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 analysis and synthesis tasks are typically performed by first converting the ODE-PDE model to a new representation called a Partial Integral Equation (PIE), which is parametrized by a special class of operators, and then solving convex optimization problems (see Chapter. 7 for more details).

To illustrate, for our example (as for any model declared in PIETOOLS), conversion to a PIE is simply done by calling the function `convert` as

```
» 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 the PDE to an equivalent PIE ---

    --- Conversion to PIE was successful ---
```

Note here that, although conversion to a PIE is done automatically, the user should be aware that the order of e.g. state variables may be changed in the process. PIETOOLS will always display a message informing the user of such changes, and in this case, we find that no reordering is performed.

Once the model is converted to a PIE, analysis, and control can be performed by calling one of the executive functions. There are numerous executive functions available, including for stability analysis, for computing norms of the system, and for performing optimal estimator and controller synthesis – see Chapter 13.

For our ODE-PDE model defined by (2.1) and (2.3), it can be proven that the system is asymptotically stable only when $b > 0$. We can verify stability for the value $b = 0.01$ by calling the stability executive for the PIE representation of our ODE-PDE model as follows

```
» settings = lpisettings('heavy');
» lpiscript(PIE,'stability',settings);
```

Here, we must also declare settings used for running the stability test, which must be specified as one of the following: `extreme`, `stripped`, `light`, `heavy`, `veryheavy`, or `custom`. For details on the optimization settings, the reader is referred to section. 7.7.

For our system, PIETOOLS is able to successfully solve the stability program, and it will inform the user of this fact by displaying the following output:

```
        The System of equations was successfully solved.
```

Although our system is indeed found to be stable, our simulation results (Fig. 2.1) show that solutions do not converge to zero very quickly, continuing to oscillate long after the disturbance has already vanished. One way to quantify stability of the system is with the H_∞ norm or L_2-gain of the system, measuring the worst-case amplification of the "energy" of the regulated output over that of the disturbance. To compute an upper bound γ on the value of this H_∞ norm, we can run the corresponding executive for the PIE representation of our system as

```
» [~,~, gam] = lpiscript(PIE,'l2gain',settings);
```

If successful, PIETOOLS will display the obtained bound on the H_∞ norm as follows

```
    The H-infty norm of the given system is upper bounded by:
    51.5744
```

The obtained bound suggest that the H_∞ norm of the system is quite large. To improve the system's rejection of disturbances, we therefore design a state-feedback controller that provides a control input $u(t)$ which minimizes the H_∞ norm of the closed-loop system, provided that such a controller exists. To synthesize this state feedback controller we can call yet another executive:

```
» [~, Kval, gam_val] = lpiscript(PIE,'hinf-controller',settings);
```

which will make PIETOOLS search for an operator $\mathcal{K}$ (stored in variable `Kval`) corresponding to the optimal controller gain, and display the closed-loop H_∞ norm if successful. For this example, the resulting controller substantially improves performance, achieving an H_∞ norm of the closed-loop system of just 0.8183

```
The closed-loop H-infty norm of the given system is upper bounded by:
0.8183
```

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

Using PIESIM, we can also simulate the response of the resulting closed-loop system. The simulated evolution of the PDE state $\dot{\mathbf{x}}(t)$ and regulated output $r(t)$ are plotted in Figure 2.2, along with the feedback control effort $u(t)$ used to achieve this response. The regulated output response of the open- and closed-loop system are displayed together in Figure 2.3, showing that the imposed feedback indeed makes the system "more stable", driving the output to 0 substantially faster.

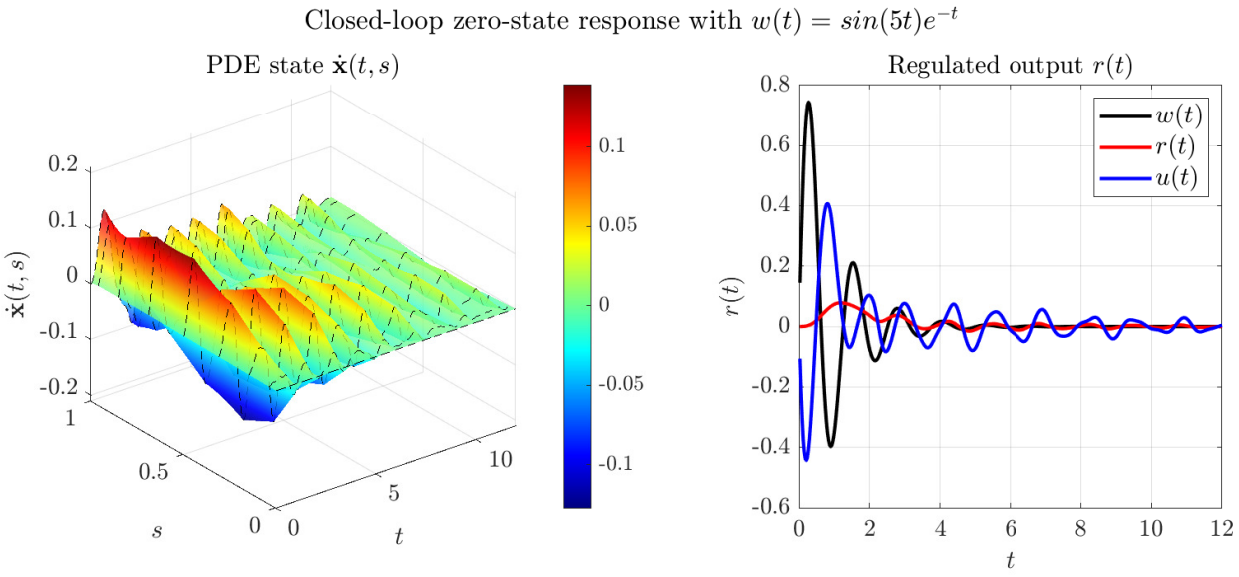


Figure 2.2: Transient response of the state variable $\dot{\mathbf{x}}(t,s)$ and regulated output $r(t)$ on the closed-loop system for external disturbance $w(t) = sin(5t)e^{-t}$.

The full code used to produce the presented plots is provided in the file "PIETOOLS_Code_Illustrations_Ch2_Introduction.m". We also encourage the user to look at the various PIETOOLS demo files for an overview on how to perform controller synthesis of ODE-PDE synthesis and simulate the resulting closed-loop system.

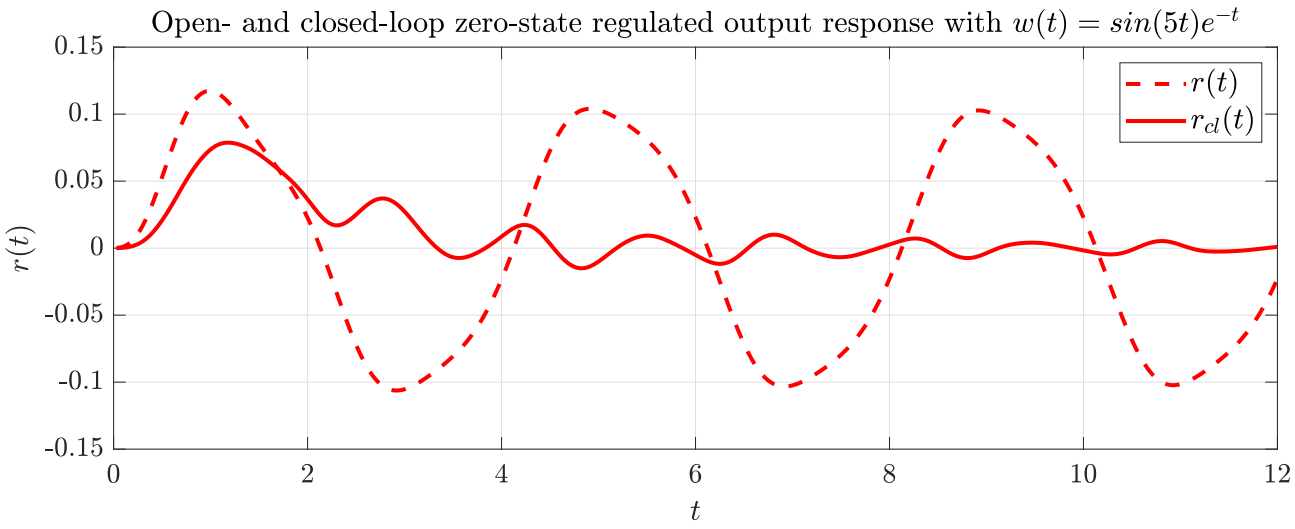


Figure 2.3: Open- ($r(t)$) and closed-loop ($r_{cl}(t)$) transient response of the regulated output $r(t)$ for external disturbance $w(t) = sin(5t)e^{-t}$.

2.3 Summary

In this chapter, we gave an introduction on 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. Figure. 2.1 shows that, even after the applied disturbance has ceased, the output signal $r(t)$ remains affected, failing to converge to zero within the simulation time. This behaviour 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. 2.2. The increase in performance can be certified by the considerable reduction in the value of the H_∞ norm and by comparing the behaviour of the outputs without and with the controller in Figure. 2.3.

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 2024, 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 $\left[\begin{smallmatrix} v_0 \\ \mathbf{v}_1 \end{smallmatrix}\right] \in \left[\begin{smallmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \end{smallmatrix}\right]$. 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 $\mathcal{P}[R] : L_2^n[a,b] \to L_2^m[a,b]$ is given by

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

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 simple PI operator $\mathcal{A} : L_2^2[-1,1] \to L_2^2[-1,1]$,

defined by

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

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 paramaters 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 $\mathcal{B} : L_2[0,1] \to L_2^2[0,1]$, defined as

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

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 s_dum
>> R0 = [1; s^2]
R0 =
  [   1]
  [ s^2]

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

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

Here, the first line calls the function `pvar` to initialize the two polynomial variables `s` and `s_dum`, 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 = s_dum;
>> B.R.R0 = R0;     B.R.R1 = R1;     B.R.R2 = R2
B =
       [] | []
       ---------
       [] | B.R

B.R =
       [1] |          [2*s] |          [3*s_dum]
     [s^2] | [s^2-s*s_dum] | [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 (`s`) and θ (`s_dum`) 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 $\mathbf{u}(s) := \big(\mathcal{B}\mathbf{v}\big)(s)$ will actually depend, and `B.var2` should correspond to the dummy variable, i.e. the variable θ which is used solely for integration.

Warning

By default, dummy variables in PIETOOLS are always assigned the same name as the primary variable, but with `_dum` added (e.g. `s` and `s_dum`). If users declare their own PI operators, they are highly recommended to use the same convention when setting their primary spatial variables and dummy variables, to avoid unintended errors when performing e.g. analysis and simulation.

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 $\mathcal{P}\left[\begin{smallmatrix} P & Q_1 \\ Q_2 & R \end{smallmatrix}\right] : \left[\begin{smallmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \end{smallmatrix}\right] \to \left[\begin{smallmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a,b] \end{smallmatrix}\right]$

$$\left(\mathcal{P}\left[\begin{smallmatrix} P & Q_1 \\ Q_2 & R \end{smallmatrix}\right]\mathbf{v}\right)(s) = \left[\begin{array}{ll} Pv_0 & + \int_a^b Q_1(s)\mathbf{v}_1(s)ds \\ Q_2(s)v_0 & + \left(\mathcal{P}[R]\mathbf{v}_1\right)(s) \end{array}\right], \qquad s \in [a,b],$$

for $\mathbf{v} = \left[\begin{smallmatrix} v_0 \\ \mathbf{v}_1 \end{smallmatrix}\right] \in \left[\begin{smallmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \end{smallmatrix}\right]$. 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 $\mathcal{C} : \left[\begin{smallmatrix} \mathbb{R}^2 \\ L_2[0,3] \end{smallmatrix}\right] \to \left[\begin{smallmatrix} \mathbb{R} \\ L_2^2[0,3] \end{smallmatrix}\right]$ defined as

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

for $\mathbf{v} = \left[\begin{smallmatrix} v_0 \\ \mathbf{v}_1 \end{smallmatrix}\right] \in \left[\begin{smallmatrix} \mathbb{R}^2 \\ L_2^1[0,3] \end{smallmatrix}\right]$. To declare this operator, we first construct the polynomial functions defining the parameters P through R_2, 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 $\mathcal{C}$ 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 $\mathbf{v} \in L_2\Big[[a,b]\times[c,d]\Big]$, 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 $\mathcal{P}[N] : L_2^n\Big[[a,b]\times[c,d]\Big] \to L_2^m\Big[[a,b]\times[c,d]\Big]$ 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 $\mathbf{v} \in L_2\Big[[a,b]\times[c,d]\Big]$. In PIETOOLS 2024, we represent such operators using `opvar2d` class objects, which are declared in a similar manner to `opvar` objects. For example, to delcare a simple operator $\mathcal{D} : L_2^2\Big[[0,1]\times[1,2]\Big] \to L_2^2\Big[[0,1]\times[1,2]\Big]$ defined as

$$\Big[\mathcal{D}\mathbf{v}\Big](s_1,s_2) = \int_0^{s_1}\int_{s_2}^2 \underbrace{\begin{bmatrix} s_1^2 & s_1s_2 \\ s_1s_2 & s_2^2 \end{bmatrix}}_{N_{12}} \mathbf{v}(\theta_1,\theta_2)d\theta_2 d\theta_1, \qquad (s_1,s_2)\in[0,1]\times[1,2],$$

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 $\mathbf{u}(s_1,s_2) = \Big(\mathcal{D}\mathbf{v}\Big)(s_1,s_2)$ 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 $L_2^2\Big[[0,1]\times[1,2]\Big] \to L_2^2\Big[[0,1]\times[1,2]\Big]$, all parameters mapping different function spaces are empty for the object `D`.

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

defined by

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

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 2024 are those mapping $\begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_x}[a,b] \\ L_2^{n_y}[c,d] \\ L_2^{n_2}\left[[a,b]\times[c,d]\right] \end{bmatrix} \to \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_x}[a,b] \\ L_2^{m_y}[c,d] \\ L_2^{m_2}\left[[a,b]\times[c,d]\right] \end{bmatrix}$, defined by parameters $R = \begin{bmatrix} R_{00} & R_{0x} & R_{0y} & R_{02} \\ R_{x0} & R_{xx} & R_{xy} & R_{x2} \\ R_{y0} & R_{yx} & R_{yy} & R_{y2} \\ R_{20} & R_{2x} & R_{2y} & R_{22} \end{bmatrix}$ 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 $\mathbf{v} = \begin{bmatrix} v_0 \\ \mathbf{v}_x \\ \mathbf{v}_y \\ \mathbf{v}_2 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_x}[a,b] \\ L_2^{n_y}[c,d] \\ L_2^{n_2}\left[[a,b]\times[c,d]\right] \end{bmatrix}$, 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 $\mathcal{F} : \begin{bmatrix} \mathbb{R} \\ L_2[0,2] \\ L_2\left[[0,2]\times[2,3]\right] \end{bmatrix} \to \begin{bmatrix} L_2^2[0,2] \\ L_2\left[[0,2]\times[2,3]\right] \end{bmatrix}$, defined as

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

for $\mathbf{v} = \begin{bmatrix} v_0 \\ \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix} \in \begin{bmatrix} \mathbb{R} \\ L_2[0,2] \\ L_2\left[[0,2]\times[2,3]\right] \end{bmatrix}$. 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 $\mathcal{B}: \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \end{bmatrix} \to \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a,b] \end{bmatrix}$ be a 4-PI operator of the form

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

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 `double` 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 = s1`	1×1 `pvar` (`polynomial` class) object specifying the spatial variable s;
`B.var2 = s1_dum`	1×1 `pvar` (`polynomial` class) object specifying the dummy variable θ;
`B.I = [a,b]`	1×2 array of type `double`, specifying the interval $[a,b]$ on which the spatial variables s and θ exist;
`B.P = P`	$m_0 \times n_0$ array of type `double` or `polynomial` defining the matrix P;
`B.Q1 = Q1`	$m_0 \times n_1$ array of type `double` or `polynomial` defining the function $Q_1(s)$;
`B.Q2 = Q2`	$m_1 \times n_0$ array of type `double` or `polynomial` defining the function $Q_2(s)$;
`B.R.R0 = R0`	$m_1 \times n_1$ array of type `double` or `polynomial` defining the function $R_0(s)$;
`B.R.R1 = R1`	$m_1 \times n_1$ array of type `double` or `polynomial` defining the function $R_1(s,\theta)$;
`B.R.R2 = R2`	$m_1 \times n_1$ array of type `double` or `polynomial` 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 $\mathcal{D}: \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_x}[a,b] \\ L_2^{n_y}[c,d] \\ L_2^{n_2}\left[[a,b]\times[c,d]\right] \end{bmatrix} \to \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_x}[a,b] \\ L_2^{m_y}[c,d] \\ L_2^{m_2}\left[[a,b]\times[c,d]\right] \end{bmatrix}$ be a PI operator of the form

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

for $\mathbf{v} = \begin{bmatrix} v_0 \\ \mathbf{v}_x \\ \mathbf{v}_y \\ \mathbf{v}_2 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_x}[a,b] \\ L_2^{n_y}[c,d] \\ L_2^{n_2}\left[[a,b]\times[c,d]\right] \end{bmatrix}$, 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 $\begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_x}[a,b] \\ L_2^{m_y}[c,d] \\ L_2^{m_2}\left[[a,b]\times[c,d]\right] \end{bmatrix}$ and $\begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_x}[a,b] \\ L_2^{n_y}[c,d] \\ L_2^{n_2}\left[[a,b]\times[c,d]\right] \end{bmatrix}$ the operator maps to and from;
`D.var1`	`= [s1; s2]`	2×1 `pvar` (`polynomial` class) object specifying the spatial variables (x, y);
`D.var2`	`= [s1_dum; s2_dum]`	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 $[a, b] \times [c, d]$ 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 formats 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 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, and we refer to Chapter 9 for two alternative input formats for time-delay systems, namely the Neutral Delay System (NDS) and Delay Difference Equation (DDF) formats.

4.1 Command Line Parser for Coupled ODE-PDEs

In PIETOOLS 2024, the simplest and most intuitive format for declaring coupled ODE-PDE systems is the Command Line Parser format. The Command Line Parser format represents ODE-PDE systems in MATLAB as `pde_struct` objects, for which a variety of operations (addition, multiplication, substitution) have been defined to allow for easy declaration of a broad class of systems. In this section, we provide an overview on how to declare such systems as `pde_struct` objects, referring to Section 8.2 for more background.

Note

In PIETOOLS 2022, a Command Line Parser format for declaring 1D ODE-PDE systems was introduced, generating dependent variables using the `state` function and representing the system as a `sys` class object. These functions are still available in PIETOOLS 2024, but do not currently support declaration of 2D PDE systems, and are therefore not discussed here. See Section 8.3 instead.

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

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

The following code shows how this system can be declared using the Command Line Parser format, and subsequently converted to a PIE.

Code Block 1

```
>> pvar t s;
>> x = pde_var();          X = pde_var(s,[0,1]);
>> w = pde_var('in');      z = pde_var('out',2);
>> u = pde_var('control');
>> y = pde_var('sense');
>> out_eq = 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 = [eqns; out_eq];
>> odepde = initialize(odepde)
>> PIE = convert(odepde,'pie');
```

We will break down each step used in the code above and explain the action performed by each line of the code. Specifying any PDE system using the 'Command Line Parser' format follows the same three simple steps listed below:

1. Define independent variables (s, t)
2. Define dependent variables ($\mathbf{x}$, x, z, y, w and u)
3. Define the equations

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 in terms of spatial variable s and temporal variable t, we would start by defining these variables as `polynomial` objects, using the function `pvar` as shown below:

```
» pvar  t  s;  % independent variables are polynomial objects
```

Note that `t` **will always be interpreted as the temporal variable** in PIETOOLS. Although we highly recommend always using `s` or `s1` as spatial variable (as this is the default used by PIETOOLS), the spatial variable can feasibly be given any name, so long as it is properly assigned to e.g. a PDE state as we show next.

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 aslo Chapter 2). Dependent variables are defined as `pde_struct` objects, and can be declared using the function `pde_var`. For example:

```
>> x = pde_var();          X = pde_var(s,[0,1]);
>> w = pde_var('in');      z = pde_var('out',2);
>> u = pde_var('control');
>> y = pde_var('sense');
```

The above code, when executed in MATLAB, creates six symbolic variables, namely `x, X, w, u, z, y`. Here, the variables `x` and `X` are not explicitly assigned a particular type, and will therefore default to be interpreted as state variables. Passing the polynomial variable `s` as well as the interval `[0,1]` in declaring `X`, the variable is interpreted to be a PDE state variable $\mathbf{x}(t,s)$ with spatial domain $s \in [0,1]$. For the remaining variables, a type is explicitly specified, declaring `w` to be an exogenous input, `z` to be a regulated output, `u` to be a controlled input, and `y` to be a sensed output. In addition, the output `z` is declared to be vector-valued, with length 2, which will be crucial when declaring the equation

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

4.1.1c Define the equations

Having declared the dependent variables that appear in the PDE, we can now use standard algebraic operations such as addition (`+`) and multiplication (`*`), as well as operations such as integration (`int`), differentiation (`diff`), and substitution (`subs`) to declare our system. For example, to declare the equation for $z(t)$, we call

```
>> out_eq = z==[int(X,s,[0,1]); u];
```

We can also declare multiple equations together in a column vector, e.g. specifying the remaining 5 equations and boundary conditions listed below

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

by calling

```
>> 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)];
```

To combine these equations into a single structure, we simply concatenate, and initialize, as

```
>> odepde = [eqns; out_eq];
>> odepde = initialize(odepde);
```

Here, the `initialize` function cleans up the PDE structure and checks for errors in the declaration, providing an overview of the variables it encounters as

```
Encountered 2 state components:
 x1(t),      of size 1, finite-dimensional;
 x2(t,s),    of size 1, differentiable up to order (2) in variables (s);

Encountered 1 actuator input:
 u(t),    of size 1;

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

Encountered 1 observed output:
 y(t),    of size 1;

Encountered 1 regulated output:
 z(t),    of size 2;

Encountered 2 boundary conditions:
 F1(t) = 0, of size 1;
 F2(t) = 0, of size 1;
```

After initialization, the system can be converted to an equivalent PIE as

```
>> PIE = convert(odepde)
PIE =
  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];
```

The output is a `pie_struct` object, storing `opvar` objects representing the PI operators defining the PIE representation of the input system – see Chapter 5 for more details. Once the PIE structure is obtained, we can proceed to perform analysis, control, and simulation, as discussed in detail in Chapters 6 and 7.

Note on declaring equations

The = symbol is not used while defining equations. Instead == is used, since 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`.

Note on PDE display

When displaying PDE variables and equations in the MATLAB Command Window, one must keep the following in mind:

- PDE variables are always represented by a letter `x` for states, `y` for observed outputs, `z` for regulated outputs, `u` for controlled inputs, and `w` for exogenous inputs;
- Each variable is displayed with an integer subscript corresponding to the unique ID assigned to this variable. This ID is crucial for PIETOOLS to distinguish different PDE variables, but may become cumbersomely large when declaring multiple systems. To avoid this issue, the ID counter can be reset by calling `clear stateNameGenerator`;
- When converting to a PIE, equations are always re-ordered to start with the ODE states, followed by the PDE (PIE) states, observed outputs, regulated outputs, and finally the boundary conditions. As such, the order of the different variables and equations after initialization or conversion may not be the same as initially declared.

4.1.2 Declaring 2D PDEs

The Command Line Input format can also be used to declare PDE systems involving multiple spatial variables. To illustrate, consider the following system of a coupled ODE, 1D PDE, and 2D PDE, with a distributed disturbance $\mathbf{w}$ and output $\mathbf{y}$:

$$
\begin{aligned}
\frac{d}{dt}x_1(t) &= -x_1(t) + \mathbf{x}_4(t,b,d) + u_1(t), && t \geq 0,\\
\partial_t \mathbf{x}_2(t,s_1) &= \partial_{s_1}^2 \mathbf{x}_2(t,s_1) + \mathbf{w}(t,s_1), && s_1 \in [a,b],\\
\partial_t \mathbf{x}_3(t,s_2) &= \partial_{s_2}^2 \mathbf{x}_3(t,s_2) + s_2 u2(t), && s_2 \in [c,d],\\
\partial_t \mathbf{x}_4(t,s_1,s_2) &= \partial_{s_1}^2 \mathbf{x}_4(t,s_1,s_2) + \partial_{s_2}^2 \mathbf{x}_4(t,s_1,s_2) + 4\mathbf{x}_4(t,s_1,s_2),\\
\mathbf{y}(t,s_2) &= \begin{bmatrix} \mathbf{x}_3(t,s_2) \\ \mathbf{x}_4(t,b,s_2) \end{bmatrix},\\
z(t) &= \int_a^b \int_c^d \mathbf{x}_4(t,s_1,s_2) ds_2 ds_1,\\
\mathbf{x}_2(t,a) &= x_1(t), \qquad \partial_{s_1}\mathbf{x}_2(t,b) = 0,\\
\mathbf{x}_3(t,c) &= x_1(t), \qquad \mathbf{x}_3(t,d) = 0,\\
\mathbf{x}_4(t,s_1,c) &= \mathbf{x}_2(t,s_1), \qquad \mathbf{x}_4(t,s_1,d) = 0,\\
\mathbf{x}_4(t,a,s_2) &= \mathbf{x}_3(t,s_2), \qquad \partial_{s_1}\mathbf{x}_4(t,b,s_2) = 0.
\end{aligned}
$$

In this case, we have an ODE state $x_1(t) \in \mathbb{R}$, two 1D PDE states $\mathbf{x}_2(t) \in L_2[a,b]$ and $\mathbf{x}_3(t) \in L_2[c,d]$, and a 2D PDE state $\mathbf{x}_4(t) \in L_2[[a,b]\times[c,d]]$. In addition, we have two

controlled inputs $u_1(t), u_2(t) \in \mathbb{R}$ and a regulated output $z(t) \in \mathbb{R}$, as well as a distributed disturbance $\mathbf{w}(t) \in L_2[a,b]$, and vector-valued sensed output $\mathbf{y}(t) \in L_2^2[c,d]$ at all times $t \geq 0$. We declare this system for $[a,b] = [0,1]$ and $[c,d] = [-1,1]$ as follows.

Code Block 2

```
>> clear stateNameGenerator
>> pvar t s1 s2
>> a = 0;   b = 1;   c = -1;   d = 1;
>> x1 = pde_var();
>> x2 = pde_var(s1,[a,b]);
>> x3 = pde_var(s2,[c,d]);
>> x4 = pde_var([s1;s2],[a,b;c,d]);
>> w = pde_var('in',s1,[a,b]);
>> z = pde_var('out');
>> u1 = pde_var('control');     u2 = pde_var('control');
>> y = pde_var('sense',2,s2,[c,d]);

>> odepde = [diff(x1,t)==-x1+subs(x4,[s1;s2],[b;d])+u1;
             diff(x2,t)==diff(x2,s1,2)+w;
             diff(x3,t)==diff(x3,s2,2)+s2*u2;
             diff(x4,t)==diff(x4,s1,2)+diff(x4,s2,2)+4*x4;
             y==[x3;subs(x4,s1,b)];
             z==int(x3,[s1;s2],[a,b;c,d]);
             subs(x2,s1,a)==x1; subs(diff(x2,s1),s1,b)==0;
             subs(x3,s2,c)==x1; subs(x3,s2,d)==0;
             subs(x4,s2,c)==x2; subs(x4,s2,d)==0;
             subs(x4,s1,a)==x3; subs(diff(x4,s1),s1,b)==0];
>> odepde = initialize(odepde);
>> PIE = convert(odepde);
```

Note here that the two spatial variables on which $\mathbf{x}_4$ depends are declared as a column array `[s1;s2]`, with the corresponding interval on which each variable exists being specified in the respective rows of the second argument `[a,b;c,d]`. Furthermore, since the disturbance $\mathbf{w}$ and output $\mathbf{y}$ are distributed as well, the variables on which they depend (as well as the domain of those variables) must be passed in the call to `pde_var` when declaring these objects. Since, in addition, the output $\mathbf{y}$ is vector-valued, the size `2` of the output must be specified as well, preceding the declaration of the spatial variable `s2`.

Note on declaring ND PDEs

Although `pde_struct` objects can be used to represents PDEs in arbitrary numbers of spatial variables, PIETOOLS does not currently offer tools for analysis or simulation of PDEs in three or more variables. Such features will be added in a later release.

4.1.3 More examples of command line parser format

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.

4.1.3a Example: Transport equation

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

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

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 Input format as shown below.

Code Block 3

```
>> clear stateNameGenerator
>> pvar t s;
>> pde_var state X control u sense y;
>> X.vars = s;    X.dom = [0,2];
>> odepde = [diff(X,t)==5*diff(X,s)+u;
             subs(X,s,0)==0;
             y==subs(X,s,2)];
>> odepde = initialize(odepde);
>> PIE = convert(odepde,'pie');
```

In this case, we declare the PDE variables in a manner similar to how we declare the independent variables, using the arguments `state`, `control`, and `sense` to declare the subsequent variables to be state, controlled inputs, and sensed output variables. In doing so, the variable X will be initially interpreted as an ODE state, which we resolve by manually setting the variables `X.vars` and domain `X.dom`.

4.1.3b Example: PDE with delay terms

The Command Line Input format can also be used to declare systems with temporal delay. To illustrate, consider a reaction-diffusion equation coupled to an ODE through a channel that is delayed by an amount $\tau = 2$. Specifically, we consider the following equations

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

This system can be declared in PIETOOLS and converted to a PIE using the following code.

Code Block 4

```
>> clear stateNameGenerator
>> pvar s t
>> x = pde_var();
>> X = pde_var(s,[0,1]);
>> odepde = [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];
>> odepde = initialize(odepde);
>> PIE = convert(odepde,'pie');
```

Running this code, and in particular the last line `PIE=convert(odepde)`, PIETOOLS will display several warnings as

```
Added 1 state components:
   x3(t,ntau2)  := x1(t-ntau2);

Variable s has been merged with variable ntau_2.
All spatial variables have been rescaled to exist on the interval [-1,1].

The state components have been re-indexed as:
   x1(t)       -->   x1(t)
   x3(t,s1)    -->   x2(t,s1)
   x2(t,s1)    -->   x3(t,s1)
```

This is because the PIE representation does not support temporal delays in any of the variables. Instead, an additional state variable $\mathbf{v}(t,r) = x_1(t-r)$ is introduced to represent the delayed state. This state variable will be governed by a 1D transport equation, satisfying

$$\partial_t \mathbf{v}(t,r) = -\partial_r \mathbf{v}(t,r), \qquad r \in [0,2], \qquad \mathbf{v}(t,2) = x(t). \tag{4.1}$$

However, simply adding this equation to our reaction-diffusion PDE would yield a 2D system: existing on $(s,r) \in [0,1] \times [0,2]$. To reduce complexity, therefore, PIETOOLS automatically rescales the variables s and r to both exist on the spatial domain $[-1,1]$, rescaling the PDE variables and equations accordingly to represent the system as a 1D ODE-PDE system

$$\begin{aligned}
\dot{x}_1(t) &= -5x_1(t), \\
\partial_t \mathbf{x}_2(t,s) &= \partial_s \mathbf{x}_2(t,s), && s \in [-1,1], \\
\partial_t \mathbf{x}_3(t,s) &= 10\mathbf{x}_3(t,s) + 4\partial_s^2 \mathbf{x}_3(t,s) + \mathbf{x}_2(t,-1), \\
\mathbf{x}_3(t,-1) = 0 = \mathbf{x}_3(t,1) \quad & \mathbf{x}_2(t,1) = x_1(t),
\end{aligned}$$

where now $x_1(t) = x(t)$, $\mathbf{x}_2(t,s) = \mathbf{v}(t,1-s) = x(t+s-1)$ and $\mathbf{x}_3(t,s) = \mathbf{x}(t,0.5(1+s))$. Consequently, the PIE representation will also involve three state variables, $(x_1(t), \partial_s \mathbf{x}_2(t), \partial_s^2 \mathbf{x}_3(t,s))$, as indicated by the dimensions of e.g. the operator `PIE.T`

```
>> PIE
PIE =
  pie_struct with properties:
     dim: 1;
    vars: [1×2 polynomial];
     dom: [1×2 double];

       T: [3×3 opvar];     Tw: [3×0 opvar];     Tu: [3×0 opvar];
       A: [3×3 opvar];     B1: [3×0 opvar];     B2: [3×0 opvar];
      C1: [0×3 opvar];    D11: [0×0 opvar];    D12: [0×0 opvar];
      C2: [0×3 opvar];    D21: [0×0 opvar];    D22: [0×0 opvar];
```

4.1.3c 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.

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

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

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

which is a vector-valued transport equation with a reaction term. We define this system using the Command Line Input format as shown below.

Code Block 5

```
>> clear stateNameGenerator
>> pvar s t
>> x = pde_var(4,s,[0,1]);
>> 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];
>> PDE = initialize(eqns);
>> PIE = convert(PDE,'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.

4.2 Alternative Input Formats for PDEs

In addition to the command line parser input format, PIETOOLS 2024 offers a graphical user interface (GUI) for declaring 1D PDEs, that allows users to simultaneously visualize the PDE that they are specifying. We briefly introduce this input format in Subsection 4.2.1, referring to Chapter 8 for more details.

4.2.1 A GUI for Declaring PDEs

Aside from the Command Line Input format, 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:

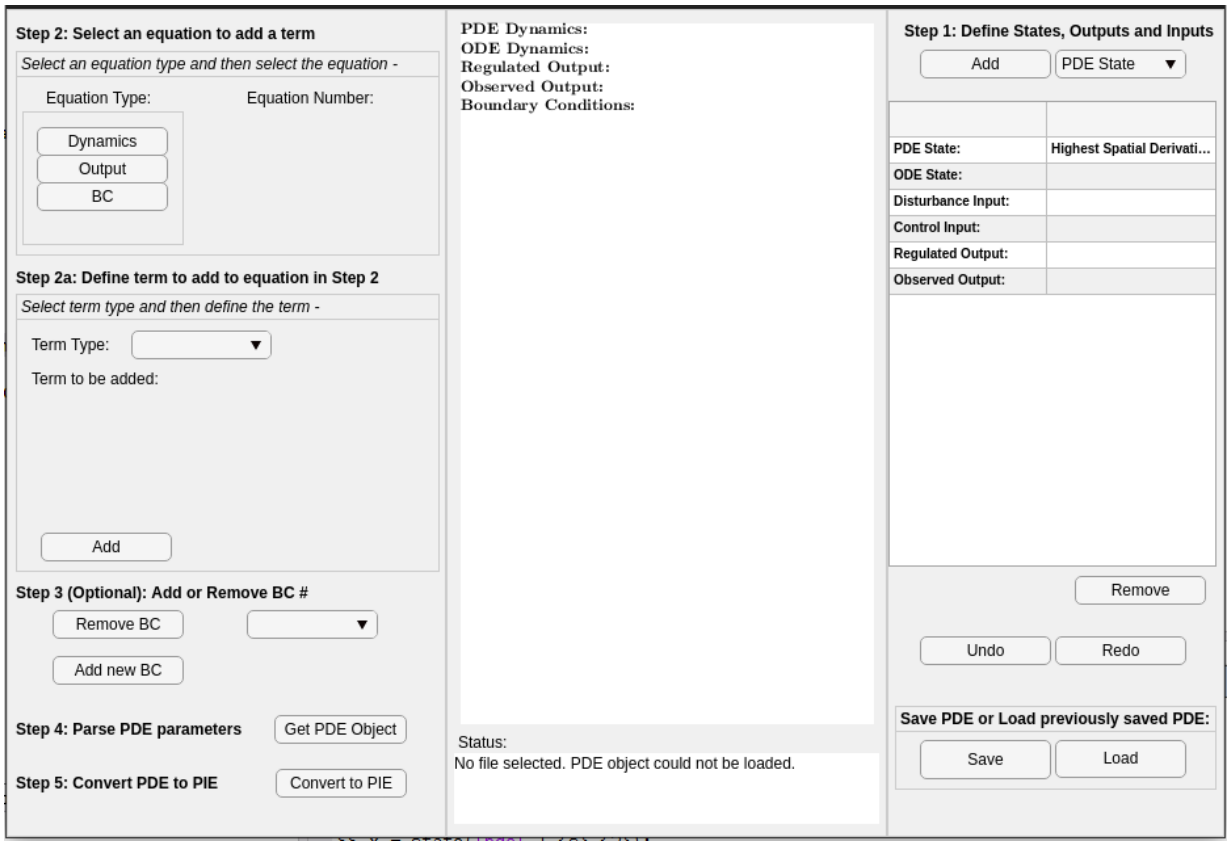


Figure 4.1: Example of empty GUI window.

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

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

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

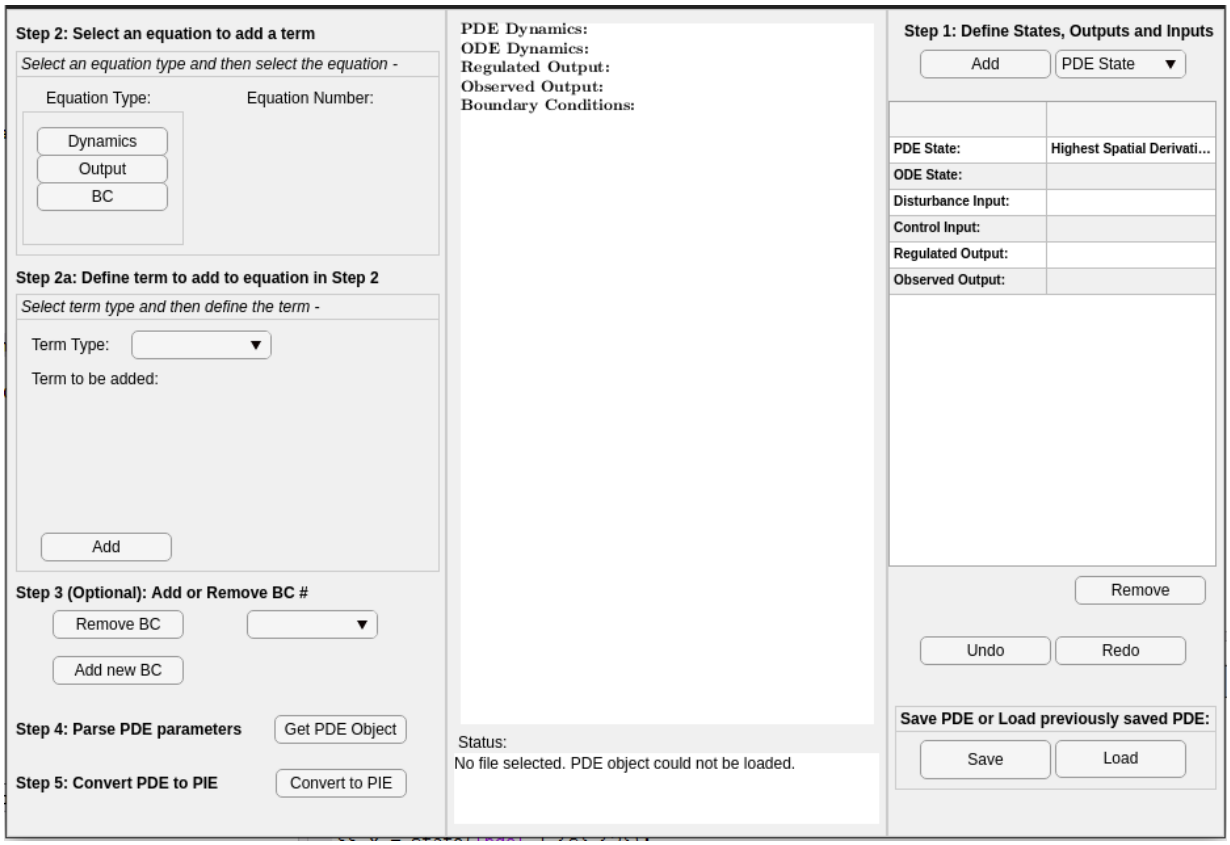


Figure 4.2: GUI window after loading the file `PIETOOLS_PDE_Ex_Heat_Eq_with_Distributed_Disturbance_GUI` from the library of PDE examples.

The declared 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.

4.3 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.

$$\begin{bmatrix}\dot{x}(t)\\ z(t)\\ y(t)\end{bmatrix} = \begin{bmatrix}A_0 & B_1 & B_2\\ C_1 & D_{11} & D_{12}\\ C_2 & D_{21} & D_{22}\end{bmatrix}\begin{bmatrix}x(t)\\ w(t)\\ u(t)\end{bmatrix} + \sum_{i=1}^{K}\begin{bmatrix}A_i & B_{1i} & B_{2i}\\ C_{1i} & D_{11i} & D_{12i}\\ C_{2i} & D_{21i} & D_{22i}\end{bmatrix}\begin{bmatrix}x(t-\tau_i)\\ w(t-\tau_i)\\ u(t-\tau_i)\end{bmatrix} + \sum_{i=1}^{K}\int_{-\tau_i}^{0}\begin{bmatrix}A_{di}(s) & B_{1di}(s) & B_{2di}(s)\\ C_{1di}(s) & D_{11di}(s) & D_{12di}(s)\\ C_{2di}(s) & D_{21di}(s) & D_{22di}(s)\end{bmatrix}\begin{bmatrix}x(t+s)\\ w(t+s)\\ u(t+s)\end{bmatrix} ds \tag{4.2}$$

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 senors.
- 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.

Note that this input format extends the possibilities of the command-line parser, which does offer support for the user to add the terms with distributed delays $A_{di}(s)$,$B_{1di}(s)$, $B_{2di}(s)$, $C_{1di}(s)$, $D_{11di}(s)$, $D_{12di}(s)$, $C_{2di}(s)$, $D_{21di}(s)$, and $D_{22di}(s)$. To add any term to the DDE structure in this Batch input format, simply declare its value. For example, to represent

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

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.

ODE Terms:

Eqn. (4.2)	DDE.	Eqn. (4.2)	DDE.	Eqn. (4.2)	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. (4.2)	DDE.	Eqn. (4.2)	DDE.	Eqn. (4.2)	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. (4.2)	DDE.	Eqn. (4.2)	DDE.	Eqn. (4.2)	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. (4.2). 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]`.

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

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

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

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

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

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

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

Conversion of PDEs and DDEs to PIEs and Closing the Loop

In the previous chapter, we showed how general linear 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 $\mathbf{x}_{\mathrm{f}}(t,s) := \partial_s \mathbf{x}(t,s)$ as the *fundamental state* associated with this PDE. Using the fundamental theorem of calculus, we can then express the PDE state in terms of the fundamental state as

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

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

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

in which the fundamental state $\mathbf{x}_{\mathrm{f}}(t)$ 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:

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

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 $\mathbf{x}_{\mathrm{f}}(t,s)=\partial_s^2\mathbf{x}(t,s)$. We implement this system in PIETOOLS using the Command Line Input format as follows:

```
>> pvar s t
>> x = pde_var(s,[0,1]);
>> PDE_dyn = diff(x,t) == diff(x,s,2);
>> PDE_BCs = [subs(x,s,0) + int(x,s,[0,1]) == 0;
              subs(x,s,1) + int(x,s,[0,1]) == 0];
>> PDE = [PDE_dyn; PDE_BCs];
```

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

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

    dim: 1;
   vars: [1×2 polynomial];
    dom: [0 1];

      T: [1×1 opvar];     Tw: [1×0 opvar];     Tu: [1×0 opvar];
      A: [1×1 opvar];     B1: [1×0 opvar];     B2: [1×0 opvar];
     C1: [0×1 opvar];    D11: [0×0 opvar];    D12: [0×0 opvar];
     C2: [0×1 opvar];    D21: [0×0 opvar];    D22: [0×0 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, s_dum]

>> PIE.dom
ans =
     0     1
```

indicating that `s` is the primary variable, `s_dum` the dummy variable (used for integration), and both exist on the domain $s, s_{\text{dum}} \in [0,1]$. 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

$$\big(\mathcal{T}\dot{\mathbf{x}}_{\text{f}}\big)(t,s) = \big(\mathcal{A}\mathbf{x}_{\text{f}}\big)(t,s),$$

where the operator $\mathcal{T}$ maps the fundamental state $\mathbf{x}_{\text{f}}$ back to the PDE state $\mathbf{x}$ as

$$\big(\mathcal{T}\mathbf{x}_{\text{f}}\big)(t,s) = \mathbf{x}(t,s).$$

For the PDE (5.3), we know that $\mathbf{x}_{\text{f}}(t,s) = \partial_s^2\mathbf{x}(t,s)$. 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*s_dum-0.25*s_dum^2-0.75*s_dum] | [s*s_dum-0.25*s_dum^2-s+0.25*s_dum]
>> A = PIE.A
A =
    [] | []
    ---------
    [] | A.R

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

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

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

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: [1×2 polynomial];
    dom: [1x2 double];
      T: [6×6 opvar];     Tw: [6×0 opvar];     Tu: [6×0 opvar];
      A: [6×6 opvar];     B1: [6×0 opvar];     B2: [6×0 opvar];
     C1: [0×6 opvar];    D11: [0×0 opvar];    D12: [0×0 opvar];
     C2: [0×6 opvar];    D21: [0×0 opvar];    D22: [0×0 opvar];
```

In this structure, we note that `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

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

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

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

and we can equivalently represent the DDE as a PDE

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

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 $\mathbf{x}_{\mathrm{f}}(t) = \begin{bmatrix} x_{\mathrm{f},0}(t) \\ \mathbf{x}_{\mathrm{f},1}(t) \\ \mathbf{x}_{\mathrm{f},2}(t) \end{bmatrix} = \begin{bmatrix} x(t) \\ \partial_s \mathbf{x}_1(t) \\ \partial_s \mathbf{x}_2(t) \end{bmatrix}$ for $t \geq 0$. The `PIE` structure derived from the `DDE` will describe the dynamics in terms of this fundamental state $\mathbf{x}_{\mathrm{f}}$, 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 $\mathbf{x}_{\mathrm{f}}$ 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

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

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

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

5.4 Converting an Input-Output System to a PIE

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

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

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 $\mathbf{x}_{\rm f}(t)$ back to the PDE state as

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

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:

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

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 Input format as

```
>> pvar s t
>> x = pde_var(s,[0,1]);
>> w = pde_var('in');    u = pde_var('control');
>> z = pde_var('out');   y = pde_var('observe');
>> 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 = [PDE_dyn; PDE_z; PDE_y; PDE_BCs];
```

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: [1×2 polynomial];
    dom: [0 1];

      T: [1×1 opvar];     Tw: [1×1 opvar];     Tu: [1×1 opvar];
      A: [1×1 opvar];     B1: [1×1 opvar];     B2: [1×1 opvar];
     C1: [1×1 opvar];    D11: [1×1 opvar];    D12: [1×1 opvar];
     C2: [1×1 opvar];    D21: [1×1 opvar];    D22: [1×1 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 $\mathbf{x} = \mathcal{T}_u u + \mathcal{T}_w w + \mathcal{T}\mathbf{x}_\mathrm{f}$ from the fundamental state $\mathbf{x}_\mathrm{f}$ 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] | [-s_dum] | [-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 $\mathbf{x}_\mathrm{f} \in L_2[0,1]$ 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

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

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 $\mathbf{x}_\mathrm{f} = \partial_s^2\mathbf{x}$ 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 $\mathbf{x}_\mathrm{f}$ must satisfy

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

This leaves only the output equations. Here, since there is no feedthrough 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 $\mathbf{x} = \mathcal{T}\mathbf{x}_\mathrm{f} + \mathcal{T}_u u$ 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]
     ----------
     [] | C2.R
```

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 $\mathbf{x}_\mathrm{f} \in L_2[0,1]$ 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

$$\dot{u}(t) - \int_0^s \theta \dot{\mathbf{x}}_\mathrm{f}(t,\theta)d\theta - \int_s^1 s\dot{\mathbf{x}}_\mathrm{f}(t,\theta)d\theta = \frac{1}{2}\mathbf{x}_\mathrm{f}(t,s) + s(2-s)w(t), \qquad s \in [0,1],\ t \geq 0, \tag{5.6}$$

$$z(t) = \int_0^1 \left(\frac{1}{2}s^2 - s\right)\mathbf{x}_\mathrm{f}(t,s)ds + u(t),$$

$$y(t) = -\int_0^1 s\mathbf{x}_\mathrm{f}(t,s)ds + u(t), \qquad \text{where } \mathbf{x}_\mathrm{f}(t,s) = \partial_s^2 \mathbf{x}(t,s).$$

5.5 Declaring and Manipulating PIEs

In the previous sections, we showed how equivalent PIE representations of PDE and DDE systems can be easily computed by simply calling the command `convert`. However, it is of course also possible to construct PIE systems directly, which can be convenient when e.g. building the closed-loop representation after performing PIE estimator or controller synthesis. In this section, we show how such PIEs can be generated for given PI operators using the function `piess`, focusing on systems without inputs and outputs in Subsection 5.5.1, systems with disturbances and regulated outputs in Subsection 5.5.2, and adding controlled inputs and observed outputs in Subsection 5.5.3. We will also show how feedback interconnections of PIE systems can be performed to construct closed-loop PIE representations in Subsection 5.5.4.

5.5.1 Declaring a simple PIE

In its simplest form, a PIE governing the dynamics of a fundamental state $\mathbf{x}_\mathrm{f}(t) \in L_2^n[a,b]$ is defined by only two PI operators: $\mathcal{T}$ and $\mathcal{A}$. For example, the PIE representation of the transport equation in (5.1) at the start of this section is given by

$$\partial_t(\mathcal{T}\mathbf{x}_\mathrm{f})(t,s) = \overbrace{\int_0^s \partial_t \mathbf{x}_\mathrm{f}(t,\theta)d\theta}^{(\mathcal{T}\mathbf{x}_\mathrm{f})(t,s)} = \overbrace{2\mathbf{x}_\mathrm{f}(t,s) + \int_0^s 10\mathbf{x}_\mathrm{f}(t,\theta)d\theta}^{(\mathcal{A}\mathbf{x}_\mathrm{f})(t,s)} = (\mathcal{A}\mathbf{x}_\mathrm{f})(t,s), \qquad s \in [0,1],\ t \geq 0,$$

with fundamental state $\mathbf{x}_\mathrm{f}(t) \in L_2[0,1]$. Here, the operators $\mathcal{T}$ and $\mathcal{A}$ defining this representation can be declared as `opvar` objects as

```
>> opvar T A;
>> T.I = [0,1];     A.I = [0,1];
>> T.R.R1 = 1;
>> A.R.R0 = 2;      A.R.R1 = 10;
```

See Chapter 3 for more details. Now, to construct the PIE representation defined by these operators, we call the function `piess` as

```
>> PIE1 = piess(T,A)
PIE1 =
  pie_struct with properties:

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

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

This returns a `pie_struct` object `PIE` with the fields `PIE.T` and `PIE.A` set to the input operators `T` and `A`, respectively. Note that the domain and variables defining the PIE are automatically set equal to those of the operators `T` and `A`, and PIETOOLS will throw an error if the variables or domains of these operators don't match.

5.5.2 Declaring a PIE with outputs and disturbances

Suppose now that we have a more elaborate PIE of the form,

$$\begin{aligned} \partial_t(\mathcal{T}\mathbf{x}_{\mathrm{f}})(t) &= \mathcal{A}\mathbf{x}_{\mathrm{f}}(t) + \mathcal{B}w(t), \\ z(t) &= \mathcal{C}\mathbf{x}_{\mathrm{f}}(t) + \mathcal{D}w(t), \end{aligned} \tag{5.7}$$

involving a disturbance $w(t)$ and regulated output $z(t)$, both finite or infinite-dimensional, at each time $t \geq 0$. Now, the dynamics are represented by five PI operators, so declaring this system with `piess` also requiring passing all five operators as e.g.

```
» PIE_zw = piess(T,A,B,C,D);
```

To illustrate, suppose we have the same system as in the previous subsection, but now with a finite-dimensional disturbance and regulated output as

$$\begin{aligned} \partial_t \int_0^s \mathbf{x}_{\mathrm{f}}(t,\theta)d\theta &= 2\mathbf{x}_{\mathrm{f}}(t,s) + \int_0^s 10\mathbf{x}_{\mathrm{f}}(t,\theta)d\theta + 5sw(t), \\ z(t) &= \int_0^1 (1-s)\mathbf{x}_{\mathrm{f}}(t,s)ds. \end{aligned}$$

In this example, the PI operators $\mathcal{A}$ and $\mathcal{T}$ are the same as in the previous subsection, and the multiplier operator $\mathcal{B} : \mathbb{R} \to L_2[0,1]$ and integral operator $\mathcal{C} : L_2[0,1] \to \mathbb{R}$ are given by

$$(\mathcal{B}w)(s) := 5sw, \quad \forall w \in \mathbb{R}, \qquad (\mathcal{C}\mathbf{v}) := \int_0^1 (1-s)\mathbf{v}(s)ds, \quad \forall \mathbf{v} \in L_2[0,1]. \tag{5.8}$$

We can declare the integral operator as an `opvar` object `C` using the field `C.Q2` as

```
>> opvar C
>> C.I = [0,1];    s = C.var1;
>> C.Q1 = 1-s;
```

Here, we set `s=C.var1` to use the default spatial variable that PIETOOLS uses for `opvar` objects, which is always a safe choice. Formally, we should then also specify the multiplier operator $\mathcal{B} : \mathbb{R} \to L_2[0,1]$ as an `opvar` object `B` with `B.Q1=5*s`. However, passing polynomial functions to `piess`, these functions are automatically interpreted as corresponding multiplier operators, so that we can simply declare our input-output PIE as

```
>> PIE2 = piess(T,A,5*s,C,0)
PIE2 =
  pie_struct with properties:

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

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

passing `0` as fifth argument to declare a zero feedthrough $\mathcal{D} = 0$. Note that the resulting operator `PIE2.B1` indeed represents the desired multiplier operator,

```
>> PIE2.B1
ans =
         [] | []
     ---------------
     [5*s1] | ans.R

ans.R =
   [] | [] | []
```

Note also that the field `PIE2.Tw`, representing the term $\mathcal{T}_w \dot{w}(t)$ in the left-hand side of the PIE dynamics is automatically populated with a zero operator of appropriate dimensions,

```
>> PIE2.Tw
ans =
      [] | []
     -----------
     [0] | ans.R

ans.R =
   [] | [] | []
```

5.5.3 Declaring a PIE with sensing and control

Finally, let us consider a PIE in its most general form as in (5.4), involving not only disturbance and regulated outputs, but also controlled inputs and observed outputs. This representation is parameterized by 12 PI operators: three operators $\{\mathcal{T}, \mathcal{T}_w, \mathcal{T}_u\}$ defining the left-hand side

of the PIE dynamics, a state operator $\mathcal{A}$, two input operators $\{\mathcal{B}_1, \mathcal{B}_2\}$, two output operators $\{\mathcal{C}_1, \mathcal{C}_2\}$, and four feedthrough operators $\{\mathcal{D}_{11}, \mathcal{D}_{12}, \mathcal{D}_{21}, \mathcal{D}_{22}\}$. To declare such a PIE for given operators, we pass the values of the different operator to `piess` using cell structures as follows:

```
>> PIE = piess({T,Tw,Tu}, A, {B1,B2}, {C1;C2}, {D11,D12;D21,D22});
```

Note here that the operators $\mathcal{B}_1$ and $\mathcal{B}_2$ must be declared using a 1×2 cell array, whereas the operators $\mathcal{C}_1$ and $\mathcal{C}_2$ must be declared using a 2×1 cell array. The operators $\mathcal{D}_{ij}$ should be declared accordingly as a 2×2 cell array. For example, consider the PIE representation of the PDE in (5.5), given in (5.6), defined by the operators

$$
\begin{aligned}
&(\mathcal{T}\mathbf{v})(s) := -\int_0^s \theta \mathbf{v}(\theta)d\theta - \int_s^1 s\mathbf{v}(\theta)d\theta, && (\mathcal{T}_u u)(s) := u, && \forall s \in [0,1].\\
&(\mathcal{A}\mathbf{v})(s) := \frac{1}{2}\mathbf{v}(s), && (\mathcal{B}_1 w)(s) = s(2-s), &&\\
&\mathcal{C}_1\mathbf{v} := \int_0^1 \left(\frac{1}{2}s^2 - s\right)\mathbf{v}(s)ds, && \mathcal{D}_{12}u := u, &&\\
&\mathcal{C}_2\mathbf{v} := -\int_0^1 s\mathbf{v}(s)ds, && \mathcal{D}_{22}u := u. &&
\end{aligned}
$$

for $\mathbf{v} \in L_2[0,1]$ and $u, w \in \mathbb{R}$. We can declare these operators as `opvar` objects as

```
>> opvar T C1 C2;
>> T.I = [0,1];    C1.I = [0,1];    C2.I = [0,1];
>> s = T.var1;         theta = T.var2;
>> T.R.R1 = -s;        T.R.R2 = -theta;
>> C1.Q1 = 0.5*s^2-s;  C2.Q1 = -s;
```

where again, we use `s=T.var1` and `theta=T.var2` to ensure that the primary spatial variable and dummy variable match the default variables used by PIETOOLS. Of course, we can also declare the multiplier operators $\mathcal{A} : L_2[0,1] \to L_2[0,1]$, $\mathcal{B}_1, \mathcal{T}_u : \mathbb{R} \to L_2[0,1]$, and $\mathcal{D}_{12}, \mathcal{D}_{22} : \mathbb{R} \to \mathbb{R}$ as `opvar` operators, but for the purpose of passing them to `piess` it suffices to declare them as just

```
>> A = 0.5;    Tu = 1;    B1 = s*(2-s);
>> D12 = 1;    D22 = 1;
```

Then, calling

```
>> PIE3 = piess({T,0,Tu},A,{B1,0},{C1;C2},{0,D12;0,D22})
PIE3 =
  pie_struct with properties:

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

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

we obtain a `pie_struct` object representing our desired PIE, where we note that e.g. the input parameter `A` is automatically augmented to a suitable `opvar` object as

```
>> PIE3.A
ans =
      [] | []
      -----------
      [] | ans.R

ans.R =
    [0.5000] | [0] | [0]
```

representing a multiplier operator $\mathcal{A} : L_2[0,1] \to L_2[0,1]$. Note also that in the call to `piess`, we can always use `0` or `[]` for an argument to indicate that the corresponding operator is a zero operator, and the function will automatically generate an associated `opvar` object will all zero parameters. Of course, users should be careful that this only works if `piess` is able to deduce the row and column dimensions of the operator from the remaining input arguments.

5.5.4 Taking interconnections of PIEs

A crucial property of the PIE representation of a system is that, due to the lack of boundary conditions and the algebraic nature of PI operators, we can easily take (feedback) interconnections of PIEs. To facilitate the computation of such interconnections of `pie_struct` objects, PIETOOLS currently offers two functions: `closedLoopPIE`, for imposing a simple feedback law $u = \mathcal{K}\mathbf{v}$ in a PIE representation, and `pielft`, for taking the linear fractional transformation of two PIE systems. To illustrate, consider again the PIE in (5.6), declared as a `pie_struct` object in the previous subsection. This PIE involves a finite-dimensional scalar input $u(t) \in \mathbb{R}$, and finite-dimensional scalar output $y(t)$. Suppose now that, e.g. by using the controller synthesis LPI (see Chapter 13.3, we find that the control law

$$u(t) = \mathcal{K}\mathbf{x}_{\mathrm{f}}(t) := \int_0^1 \mathbf{x}_{\mathrm{f}}(t,s)ds$$

for PIE state $\mathbf{x}_{\mathrm{f}}(t) \in L_2[0,1]$ is a good control law to e.g. stabilize the system. We can declare the operator $\mathcal{K} : L_2[0,1] \to \mathbb{R}$ representing this feedback law as

```
>> opvar K;
>> K.I = [0,1];    K.Q1 = 1;
```

Then, to impose this feedback law in our PIE, we can call `closedLoopPIE` as

```
>> PIE3_CL1 = closedLoopPIE(PIE3,K)
PIE3_CL1 =
  pie_struct with properties:

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

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

returning a `pie_struct` object representing the closed-loop PIE

$$\int_0^s [1-\theta]\dot{\mathbf{x}}_{\mathrm{f}}(t,\theta)d\theta + \int_s^1 [1-s]\dot{\mathbf{x}}_{\mathrm{f}}(t,\theta)d\theta = \frac{1}{2}\mathbf{x}_{\mathrm{f}}(t,s) + s(2-s)w(t), \qquad s\in[0,1],\ t\geq 0,$$
$$z(t) = \int_0^1 \left(1+\frac{1}{2}s^2 - s\right)\mathbf{x}_{\mathrm{f}}(t,s)ds,$$
$$y(t) = \int_0^1 (1-s)\mathbf{x}_{\mathrm{f}}(t,s)ds.$$

Note that this PIE representation no longer involves any controlled input $u(t)$, as we have imposed the feedback law $u(t) = \int_0^1 \mathbf{x}_{\mathrm{f}}(t,s)ds$.

Now, suppose that we instead want to synthesize a Luenberger-type estimator for our PIE, simulating an estimate $\hat{\mathbf{x}}_{\mathrm{f}}(t)$ of the PIE state and $\hat{z}(t)$ of the output state as

$$\dot{u}(t) - \int_0^s \theta\dot{\hat{\mathbf{x}}}_{\mathrm{f}}(t,\theta)d\theta - \int_s^1 s\dot{\hat{\mathbf{x}}}_{\mathrm{f}}(t,\theta)d\theta = \frac{1}{2}\hat{\mathbf{x}}_{\mathrm{f}}(t,s) + \mathcal{L}(\hat{y}(t)-y(t)), \qquad s\in[0,1],\ t\geq 0,$$
$$\hat{z}(t) = \int_0^1 \left(\frac{1}{2}s^2 - s\right)\hat{\mathbf{x}}_{\mathrm{f}}(t,s)ds + u(t),$$
$$\hat{y}(t) = -\int_0^1 s\hat{\mathbf{x}}_{\mathrm{f}}(t,s)ds + u(t),$$

for some gain $\mathcal{L} : R \to L_2[0,1]$. Let a suitable gain $\mathcal{L}$ (computed using e.g. the LPI from Section 13.2) be given by $\mathcal{L}y = s(1-s)y$ for $y\in\mathbb{R}$. Then, we can construct the "closed-loop" system for this value of the gain as

```
>> opvar L;          s = L.var1;
>> L.I = [0,1];     L.Q2 = s*(1-s);
>> PIE3_CL2 = closedLoopPIE(PIE3,L,'observer')
PIE3_CL2 =
  pie_struct with properties:

      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: [0×2 opvar];    D21: [0×1 opvar];    D22: [0×1 opvar];
```

representing the augmented system of our original PIE with the resulting estimator as

$$\dot{u}(t) - \int_0^s \theta\dot{\mathbf{x}}_{\mathrm{f}}(t,\theta)d\theta - \int_s^1 s\dot{\mathbf{x}}_{\mathrm{f}}(t,\theta)d\theta = \frac{1}{2}\mathbf{x}_{\mathrm{f}}(t,s) + s(2-s)w(t), \qquad s\in[0,1],\ t\geq 0,$$
$$\dot{u}(t) - \int_0^s \theta\dot{\hat{\mathbf{x}}}_{\mathrm{f}}(t,\theta)d\theta - \int_s^1 s\dot{\hat{\mathbf{x}}}_{\mathrm{f}}(t,\theta)d\theta = \frac{1}{2}\hat{\mathbf{x}}_{\mathrm{f}}(t,s) - s(1-s)\int_0^1 \theta\hat{\mathbf{x}}_{\mathrm{f}}(t,s)d\theta + s(1-s)\int_0^1 \theta\mathbf{x}_{\mathrm{f}}(t,s)d\theta,$$
$$z(t) = \int_0^1 \left(\frac{1}{2}s^2 - s\right)\mathbf{x}_{\mathrm{f}}(t,s)ds + u(t),$$
$$\hat{z}(t) = \int_0^1 \left(\frac{1}{2}s^2 - s\right)\hat{\mathbf{x}}_{\mathrm{f}}(t,s)ds + u(t).$$

Note that this system involves no more observed output y, instead implicitly feeding this output back into the estimator dynamics. Further note that the PIE now involves two state variables, $(\mathbf{x}_{\mathrm{f}}(t), \hat{\mathbf{x}}_{\mathrm{f}})$, and two regulated outputs, $(z(t), \hat{z}(t))$, corresponding to the true values and their estimates.

Finally, aside from imposing simple feedback laws, we can also take a linear fractional transformation of two complete PIE systems using `pielft`. To illustrate, suppose that we combine our earlier estimator and controller, to generate a control effort $u(t)$ for our PIE using the estimate $\hat{\mathbf{x}}_{\mathrm{f}}(t)$ of the state as

$$-\int_0^s \theta\dot{\hat{\mathbf{x}}}_{\mathrm{f}}(t,\theta)d\theta - \int_s^1 s\dot{\hat{\mathbf{x}}}_{\mathrm{f}}(t,\theta)d\theta = \frac{1}{2}\hat{\mathbf{x}}_{\mathrm{f}}(t,s) - s(1-s)\int_0^1 \theta\hat{\mathbf{x}}_{\mathrm{f}}(t,s)d\theta - s(1-s)\hat{y}(t),$$
$$\hat{z}(t) = \int_0^1 \Big(\frac{1}{2}s^2 - s\Big)\hat{\mathbf{x}}_{\mathrm{f}}(t,s)ds,$$
$$\hat{u}(t) = \int_0^1 \hat{\mathbf{x}}_{\mathrm{f}}(t,s)ds.$$

Note now that the output $y(t)$ of our original PIE is used as input $\hat{y}(t)$ to this estimator PIE, and we will use the output $\hat{u}(t)$ to this estimator PIE as input $u(t)$ to our original system. To declare this estimator system, we again use `piess` as

```
>> opvar T A C1;
>> T.I = [0,1];    A.I = [0,1];    C1.I = [0,1];
>> s = T.var1;          theta = T.var2;
>> T.R.R1 = -s;         T.R.R2 = -theta;
>> A.R.R0 = 0.5;        A.R.R1 = -s*(1-s)*theta;    A.R.R2 = A.R.R1;
>> C1.Q1 = 0.5*s^2-s;
>> PIE3_est = piess(T,A,{[],-L},{C1;K});
```

where we use the same operators `K` and `L` as before, passing `[],-L` as argument to declare a controlled input $\mathcal{L}\hat{u}(t)$, and passing `C1;K` as argument to declare an observed output $\hat{y}(t) = \mathcal{K}\hat{\mathbf{x}}_{\mathrm{f}}(t)$, in addition to the regulated output $\hat{z}(t) = \mathcal{C}_1\hat{\mathbf{x}}(t)$. Then, to take the linear fractional transformation of our PIE with this estimator system, using the interconnection signals $\hat{u}(t) = y(t)$ and $u(t) = \hat{y}(t)$, we simply call `pielft` as

```
>> PIE3_CL3 = pielft(PIE3,PIE3_est)
PIE3_CL3 =
  pie_struct with properties:

      T: [2×2 opvar];     Tw: [2×1 opvar];     Tu: [2×0 opvar];
      A: [2×2 opvar];     B1: [2×1 opvar];     B2: [2×0 opvar];
     C1: [2×2 opvar];    D11: [2×1 opvar];    D12: [2×0 opvar];
     C2: [0×2 opvar];    D21: [0×1 opvar];    D22: [0×0 opvar];
```

yielding a `pie_struct` object representing the system

$$\int_0^1 \dot{\hat{\mathbf{x}}}_{\mathrm{f}}(t) - \int_0^s \theta\dot{\mathbf{x}}_{\mathrm{f}}(t,\theta)d\theta - \int_s^1 s\dot{\mathbf{x}}_{\mathrm{f}}(t,\theta)d\theta = \frac{1}{2}\mathbf{x}_{\mathrm{f}}(t,s) + s(2-s)w(t), \qquad s\in[0,1],\ t\geq 0,$$
$$-\int_0^s \theta\dot{\hat{\mathbf{x}}}_{\mathrm{f}}(t,\theta)d\theta - \int_s^1 s\dot{\hat{\mathbf{x}}}_{\mathrm{f}}(t,\theta)d\theta = \frac{1}{2}\hat{\mathbf{x}}_{\mathrm{f}}(t,s) - s(1-s)\int_0^1 \theta\hat{\mathbf{x}}_{\mathrm{f}}(t,s)d\theta + s(1-s)\int_0^1 \theta\mathbf{x}_{\mathrm{f}}(t,s)d\theta,$$
$$z(t) = \int_0^1 \Big(\frac{1}{2}s^2 - s\Big)\mathbf{x}_{\mathrm{f}}(t,s)ds + \int_0^1 \hat{\mathbf{x}}_{\mathrm{f}}(t,s)ds,$$
$$\hat{z}(t) = \int_0^1 \Big(\frac{1}{2}s^2 - s\Big)\hat{\mathbf{x}}_{\mathrm{f}}(t,s)ds.$$

This PIE has no more controlled inputs or observed outputs, as the controlled inputs of the first PIE have been set equal to the observed outputs of the second PIE, and vice versa. However,

the closed-loop PIE system is expressed in terms of the states, disturbances, and regulated outputs of both PIE systems.

For additional examples on using `piess` and `pielft` to construct closed-loop PIE representations after controller and observer synthesis, see also demos 5 through 7 in Chapter 11.

Note

When imposing feedback laws using `closedLoopPIE`, **all** controlled inputs are assumed to be governed by the feedback. Thus, the output dimensions of the specified gain must match the dimensions of the controlled input in the PIE. Similarly, when constructing an estimator using `closedLoopPIE`, the input dimensions of the Luenberger gain must match the dimensions of the observed output. Finally, taking the linear fractional transformation of two PIEs, the dimensions of the controlled input of the first PIE must match those of the observed output of the second PIE, and vice versa.

Warning

The structure of `opvar` objects restricts them to map only the function space $\mathbb{R}^m \times L_2^n[a,b]$, not e.g. $L_2[a,b] \times \mathbb{R}^m$. As such, the state, input, and output variables in the PIE representation will always be reordered to place the finite-dimensional variables (e.g ODE states) ahead of the infinite-dimensional variables (e.g. PDE states). Similarly, state, input, and output variables on lower-dimensional domains (e.g. 1D PDE states) will always be placed ahead of variables on higher-dimensional domains (e.g. 2D PDE states). Consequently, both when converting ODE-PDE or DDE systems to PIEs and when taking e.g. linear fractional transformations of PIE systems, the order of the states, inputs, and outputs may be changed.

Chapter 6

PIESIM: A General-Purpose Simulation Tool based on PIETOOLS

PIESIM is a general-purpose, versatile, high-fidelity simulation tool, distributed with the PIETOOLS package. It numerically solves PDEs, as well as coupled PDE/ODEs, and DDE systems, in one and two dimensions. PIESIM utilizes high-order methods based on Chebyshev polynomial approximation for spatial discretization and offers temporal discretization schemes of the order one through four based on implicit backward-difference formulas (BDF). PIESIM is based on the PDE-to-PIE transformation framework native to PIETOOLS and seamlessly applies high-order methods to PDE problems while exactly satisfying any arbitrary set of admissible boundary conditions.

6.1 Organization of PIESIM

PIESIM is based on a set of MATLAB functions that perform the following tasks:

1. Discretization of the PDE/DDE/PIE solution domain and the PI operators.
2. Temporal integration using the BDF scheme.
3. Transformation of solution from the PIE (fundamental) state back to the PDE (primary) state and plotting the solution.

6.2 `PIESIM.m` - the Main Routine of PIESIM

The main routine of the PIESIM solver is the function `PIESIM.m` located in the folder `PIESIM/PIESIM_routines`. This function can be called using the syntax

```
» solution = PIESIM(system,opts,uinput,ndiff);
```

where the inputs are characterized by:

1. `system` - a PDE, DDE, or a PIE structure (supported in 1D only) - **required**;
2. `opts` - simulation options structure - **optional**;

3. `uinput` - structure that describes initial conditions and other options - **optional**;

4. `ndiff` - **required only** if the `system` defined is of the type 'PIE',

and the output:

1. `solution` - returns the variables associated with the solution field.

Detailed description of the inputs (and their default values if not specified by user) is provided below.

1. `system` is the *required* input that declares the structure of the problem to be numerically solved. It can take a value of 'PDE', 'DDE', or a 'PIE' (in 1D only). The user is responsible for declaring this variable using available options in PIETOOLS prior to calling PIESIM. The `system` variable does not have a default value. The simulation will not run and an error will be issued if the variable `system` is not declared.

2. `opts` is the *optional* structure with the fields:

   - `N` (integer $\geq 2+d$, where d is the order of the highest spatial derivative in the problem) - polynomial order used in a spatial discretization of a **primary** state solution - default `N`=8
   - `tf` (real > 0) - final time of simulation - default `tf`=1
   - `Norder` (integer $\{1, 2, 3, 4\}$) - order of time integration scheme - default `Norder`=2
   - `dt` (real > 0) - time step value used in numerical time integration - default `dt`=0.01
   - `plot` ('yes' or 'no') - option for plotting solution: returns solution plots if 'yes', no plots if 'no' - default `plot`='no'
   - `ploteig` ('yes' or 'no') - option for plotting discrete eigenvalues of a temporal propagator: returns eigenvalue plot if 'yes', no plot if 'no' - default `ploteig`='no'

3. `uinput` is the *optional* structure with the fields:

   - `ic` - initial conditions defined as a MATLAB symbolic object. Initial conditions should be specified for the **primary** states if the system type is 'PDE' or 'DDE', and for the **fundamental** states if the system type is 'PIE'. `ic` consists of the following sub-fields:
     - `ic.pde` - initial conditions for the PDE states defined as symbolic objects in 'sx' (1D), or 'sx', 'sy' (2D)
     - `ic.ode` - initial conditions for the ODE states defined as scalars

     The order of entries in the `ic.pde`, `ic.ode` arrays should correspond to the order of states in the originally declared problem (of the type 'PDE', 'DDE' or 'PIE').
   - `w` - external disturbances defined as a MATLAB symbolic object in 'st', 'sx' (1D), or 'st', 'sx', 'sy' (2D)

- `ifexact` (true or false, false default) - indicates whether exact solution will be used for comparison with numerical solution (if true) or not (if false). `uinput.ifexact=true` should only be used if exact solution is known. In this case, exact solution needs to be specified in the `uinput.ifexact` field - it is provided automatically if PIESIM built-in examples are used. If `uinput.ifexact=true` and `opts.plot='yes'`, the exact solution will be plotted together with the numerical solution.
- `exact` - used only if `uinput.ifexact=true` and contains exact solution to the declared problem, specified as a symbolic object in 'sx' (1D), or 'sx', 'sy' (2D)

4. `ndiff` is a *required* structure if the `system` type is 'PIE'. `ndiff` is a vector of integers specifying the number of differentiable states based on the index location, such as `ndiff`(i) is the number of states that are $(i-1)$ times differentiable in space. For example, [0, 2, 1] stands for (0) continuous, (2) continuously differentiable, and (1) twice continuously differentiable states.

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!

`PIESIM.m` function returns an output structure `solution` with the fields:

- `tf` - scalar - actual final time of the solution
- `final.pde` - PDE solution at the final time
  - In 1D problems - array of size `N+1` $\times$ `ns`, `ns` - total number of PDE states
  - In 2D problems - cell array `final.pde`{1, 2}
    `final.pde`{1} - array containing the solution for states that are only the functions of one variable - array of size `(N+1)` $\times$ (`nx`+`ny`), `nx` - number of states depending only on s_1, `ny` - number of states depending only on s_2
    `final.pde`{2} - array containing the solution for states that are the functions of two variables - array of size (`N`+1) $\times$ (`N`+1) $\times$ `ns`, `ns` - number of states depending on both s_1 and s_2
- `final.ode` - array of size `no` (number of ODE states) - ODE solution at the final time
- `final.observed` - array of size `noo` (number of observed outputs) - final value of observed outputs
- `final.regulated` - array of size `nro` (number of regulated outputs) - final value of regulated outputs
- `timedep.dtime` - array of size 1$\times$ `Nsteps` - array of discrete time values at which the time-dependent solution is computed. `Nsteps` - number of time steps - is calculated as `Nsteps= floor(tf/dt)`

- `timedep.pde` - time-dependent PDE solution
  - In 1D problems - array of size `N+1` $\times$ `ns` $\times$ `Nsteps`, `ns` - total number of PDE states
  - In 2D problems - cell array `timedep.pde`{1, 2}
    `timedep.pde`{1} - array containing the solution for states that are only the functions of one variable - array of size `(N+1)` $\times$ (`nx`+`ny`) $\times$ `Nsteps`, `nx` - number of states depending only on s_1, `ny` - number of states depending only on s_2
    `timedep.pde`{2} - array containing the solution for states that are the functions of two variables - array of size (`N`+1) $\times$ (`N`+1) $\times$ `ns` $\times$ `Nsteps`, `ns` - number of states depending on both s_1 and s_2
- `timedep.ode` - array of size `no` $\times$ `Nsteps` - time-dependent solution of ODE states
- `timedep.observed` - array of size `noo` $\times$ `Nsteps` - time-dependent value of observed outputs
- `timedep.regulated` - array of size `nro` $\times$ `Nsteps` - time-dependent value of regulated outputs

Note on final time

If (tf/dt) is not an integer value, i.e $Nsteps = floor(tf/dt) \neq (tf/dt)$, the final time tf is adjusted as $tf = Nsteps \times dt$. The time step value dt is always used as specified by the user and is never changed intrinsically.

Note on solution output

In contrast to `uinput.ic.pde`, irrespective of the input, both `final.pde` and `solution.timedep.pde` fields always store the value of the **primary state** (reconstructed) solution to the original PDE/DDE problem, i.e. $\mathcal{T}\mathbf{v} + \mathcal{T}_w w + \mathcal{T}_u u$ as the value.

6.3 Running PIESIM

PIESIM can be run from any location within PIETOOLS, as long as the desired 'PDE', 'DDE' or 'PIE' structure has already been defined, by calling the `PIESIM.m` function as

```
» solution = PIESIM(system);
```

when the default options for `opts` and `uinput` will be used, or via

```
» solution = PIESIM(system,opts,uinput);
```

when `opts` and `uinput` will be declared by the user prior to calling the `PIESIM.m` function.

Note that the user may choose to declare either `opts` or `uinput` independently while leaving the other input argument as default, with calling

```
» solution = PIESIM(system,opts);
```

or

```
» solution = PIESIM(system,uinput);
```

respectively.

If 'PIE' argument is passed into PIESIM as the system type, the additional input `ndiff` specifying the order of differentiability of the original PDE/DDE states is required as described in Section 6.2, i.e.

```
» solution = PIESIM(PIE,ndiff);
```

with the default values of `opts` and `uinput` or

```
» solution = PIESIM(PIE,opts,uinput,ndiff);
```

with the user-specified values of `opts` and `uinput`.

PIETOOLS distribution has built-in examples and demonstrations of how to run PIESIM. These examples can be found in:

1. PIESIM demonstrations accessible from "PIETOOLS_Code_Illustrations_Ch6_PIESIM" located in the folder "PIETOOLS_demos/snippets_from_manual". These are described in details in Chapter 6.4 of this manual.

2. PIETOOLS demonstrations located in the folder "PIETOOLS_demos" of PIETOOLS (see Chapter 11 of this manual). Some of the demonstrations (specifically, demonstrations 1, 5, 6, 7 and 9) feature PIESIM.

3. PIETOOLS examles in the file `PIETOOLS_PDE.m`. The script `PIETOOLS_PDE.m` allows user to select predefined examples (1 through 40) from the PIETOOLS example library (located in the folder "PIETOOLS_examples"), featuring both 1D and 2D problems. It then offers the user an option to run stability or a controller synthesis script (based on the example) by selecting the 'y' option. Regardless of whether 'y' or 'n' is selected, PIESIM is executed afterwards.

4. PIESIM examples located in the files `examples_pde_library_PIESIM_1D` and `examples_pde_library_PIESIM_2D` within the "PIESIM" folder, for 1D and 2D problems respectively. To run examples from the "PIESIM" folder, a MATLAB executable `solver_PIESIM.m` is provided in the same folder. To execute the examples using `solver_PIESIM.m`, the user should first adjust the desired problem dimension by setting `dim=1` for 1D problems or `dim=2` for 2D problems within `solver_PIESIM.m` file. The user then must choose the example number by setting the variable `example` to the corresponding example number which can range between 1 and 38 for 1D problems and between 1 and 19 for 2D problems. All examples in the "PIESIM" folder come with the provided analytical solution that will automatically be plotted alongside the numerical solution if `uinput.ifexact=true` and `opts.plot='yes'` are selected. This feature provides a good option for benchmarking PIESIM numerical solutions and performing convergence checks, if needed.

Note on Numerical Stability

The backward-difference formula is an implicit scheme whose region of stability lies outside of a small circle in the right half of the complex plane. This means that numerical simulations may be unstable if a discrete system has eigenvalues with small positive real parts. These eigenvalues can arise if the underlying physical problem is unstable or if the discretization errors move otherwise stable eigenvalues into the right half-plane. This situation may occur especially if the actual system has eigenvalues close to purely imaginary, as in systems with very little dissipation. PIESIM has a built-in check of numerical stability and issues a warning if the scheme is unstable. It also provides a recommendation on the time step size when the scheme may stabilize, if the recommended value is less than one. Increasing time step to a value higher than one is not recommended, since it will lead to large numerical errors. If this situation occurs, it is likely that the underlying physical problem is unstable, and the corresponding warning is issued.

6.4 PIESIM Demonstrations

This section presents several illustrative examples on how PIESIM can be simulated systems in PDE, DDE and PIE representation. Note that, although the figures displayed in this section have been modified slightly from the default figures returned by PIESIM, the script "PIETOOLS_Code_Illustrations_Ch6_PIESIM" located in the folder "PIETOOLS_demos/snippets_from_manual" contains full codes to reproduce each of these plots.

6.4.1 PIESIM Demonstration A: 1D PDE example

In this section, we will demonstrate the standard process involved in simulation of 1D PDEs, simulating Example 4 from `examples_pde_library_PIESIM_1D` using the following code.

Code Block 6

```
» syms sx st;
» [PDE,uinput]=examples_pde_library_PIESIM_1D(4);
» uinput.exact(1) = -2*sx*st-sx^2;
» uinput.ifexact=true;
» uinput.w(1) = 0;
» uinput.w(2) = -4*st-4;
» uinput.ic.PDE = -sx^2;
» opts.plot = 'yes';
» opts.N = 8;
» opts.tf = 1;
» opts.Norder = 2;
» opts.dt=1e-3;
» solution = PIESIM(PDE,opts,uinput);
» tval = solution.timedep.dtime;
» xval = reshape(solution.timedep.pde(:,1,:),opts.N+1,[]);
```

We will explain each line used in the above code. First, we load an example from the PIESIM examples library. An example can be selected by specifying the example number (between 1 and 38 for 1D problems) to load the example.

```
[PDE,uinput]=examples_pde_library_PIESIM_1D(4);
```

In this demonstration, we choose the example

$$\begin{aligned} &\dot{\mathbf{x}}(t,s) = s\partial_s^2\mathbf{x}(t,s), \qquad s\in[0,2], t\geq 0 \\ &\mathbf{x}(t,0) = w_1(t), \qquad \mathbf{x}(t,2) = w_2(t), \qquad \mathbf{x}(0,s) = -s^2. \end{aligned} \tag{6.1}$$

where $w_1(t) = 0$ and $w_2(t) = -4t-4$. 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) = 0;
» uinput.w(2) = -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 may choose to specify simulation parameters under `opts` structure.

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

First, we turn on the plotting by setting the plotting option to 'yes'. Next, we specify the order of spatial discretization `N` (order of Chebyshev polynomials to be used in the approximation of the PDE solution) and the time of simulation, `tf`. Then, we select the order of the temporal integration scheme (backward-difference scheme) as 2 and the time step as 1e-3. If these parameters are not specified, they will be set at their default values as listed in Section 6.2. However, the user can modify these parameters as needed. Now that we have defined all necessary and optional parameters, we can run the simulation using the command for the PDE example:

```
» solution = PIESIM(PDE,opts,uinput);
» tval = solution.timedep.dtime;
» xval = reshape(solution.timedep.pde(:,1,:),opts.N+1,[]);
```

We obtain the time steps at which the solution is computed from `soution.timedep.dtime`, and the value of the PDE state $x(t,s)$ at these times from `solution.timedep.pde`. Note that the number of columns of this latter field matches the number of state variables in our system,

so we extract the first (and in this case only) column of this array. Element `xval(i,j)` will then specify the value of our PDE state at grid point `i` and time step `j`. An isosurface plot of these values is shown in Figure 6.1, along with the simulated and exact value of the PDE state at the final time $t = 1$. These plots are produced by PIESIM if `opts.plot='yes'` is chosen.

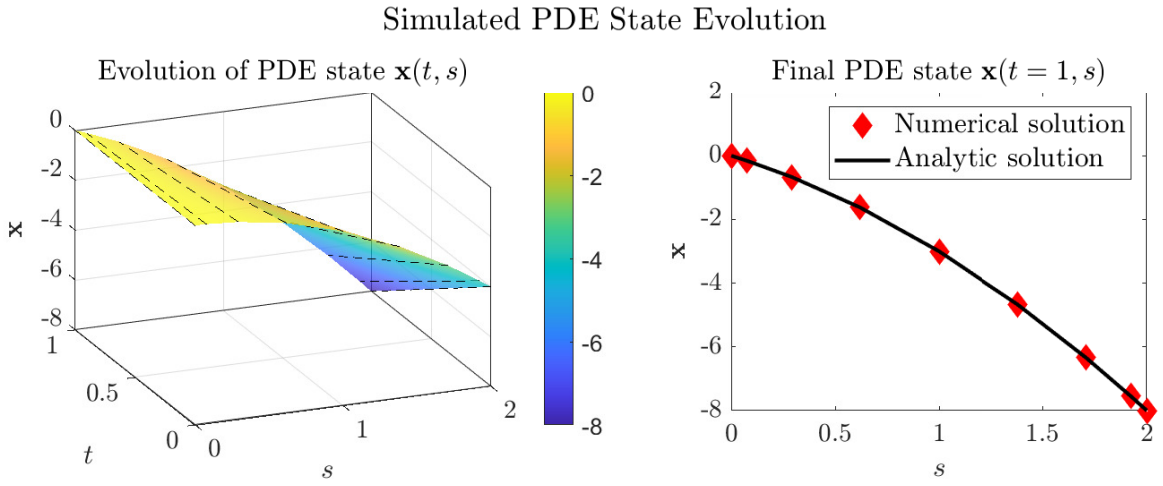


Figure 6.1: Simulated evolution of the PDE state $\mathbf{x}(t,s)$ (left) and final value $\mathbf{x}(t=1,s)$ (right) for the PDE (6.1), along with the analytic solution at the final time.

6.4.2 PIESIM Demonstration B: 2D PDE example

Simulation of 2D ODE-PDE systems can be done with PIESIM in much the same way to that of 1D systems. To illustrate, consider the following system, consisting of a 2D PDE coupled to 1D PDEs and an ODE at the boundaries,

$$
\begin{aligned}
\dot{x}_1(t) &= -x_1(t), && t \geq 0, \\
\partial_t \mathbf{x}_1(t,s_1) &= \frac{1}{\pi^2}\partial_{s_1}^2 \mathbf{x}_2(t,s_1), && s_1 \in [0,1], \\
\partial_t \mathbf{x}_3(t,s_2) &= \frac{1}{\pi^2}\partial_{s_2}^2 \mathbf{x}_3(t,s_2), && s_2 \in [0,5], \\
\partial_t \mathbf{x}_4(t,s_1,s_2) &= \frac{1}{2\pi^2}\Big(\partial_{s_1}^2 \mathbf{x}_4(t,s_1) + \partial_{s_2}^2 \mathbf{x}_4(t,s_1)\Big), \\
\mathbf{x}_2(t,0) &= x(t), && \partial_{s_1}\mathbf{x}_2(t,1) = 0, \\
\mathbf{x}_3(t,0) &= x(t), && \partial_{s_2}\mathbf{x}_3(t,5) = 0, \\
\mathbf{x}_4(t,0,s_2) &= \mathbf{x}_3(t,s_2) && \partial_{s_1}\mathbf{x}_4(t,1,s_2) = 0, \\
\mathbf{x}_4(t,s_1,0) &= \mathbf{x}_2(t,s_1) && \partial_{s_2}\mathbf{x}_4(t,s_1,5) = 0,
\end{aligned}
\tag{6.2}
$$

This system can be readily declared using the Command Line Input format as shown in Chapter 4, but is also included in the PIESIM 2D example library as example number 9. Starting with initial conditions

$$
x_1(0) = 10, \qquad \mathbf{x}_2(0,s_1) = 10\cos(\pi s_1), \quad \mathbf{x}_3(0,s_2) = 10\cos(\pi s_2),
$$

$$\mathbf{x}_4(0, s_1, s_2) = 10\cos(\pi s_1)\cos(\pi s_2), \tag{6.3}$$

the exact solution to the PDE, $\mathbf{x}(t) = (x_1(t), \mathbf{x}_2(t), \mathbf{x}_3(t), \mathbf{x}_4(t)$, is given by $\mathbf{x}(t) = e^{-t}\mathbf{x}(0)$. We can numerically approximate this solution through simulation with PIESIM by running the following code.

Code Block 7

First, extract our desired example PDE:

```
>> [PDE,~] = examples_pde_library_PIESIM_2D(9);
```

Next, set the initial conditions and exact solution:

```
>> syms sx sy st real
>> u_ex = 10*cos(sym(pi)*sx)*cos(sym(pi)*sy)*exp(-st);
>> uinput.exact(1) = subs(subs(u_ex,sy,0),sx,0);
>> uinput.exact(2) = subs(u_ex,sy,0);
>> uinput.exact(3) = subs(u_ex,sx,0);
>> uinput.exact(4) = u_ex;
>> uinput.ic.x = subs(uinput.exact,st,0);
>> uinput.ifexact = true;
```

Also, set the options for simulation

```
>> opts.plot = 'yes';
>> opts.N = 16;
>> opts.tf = 1;
>> opts.dt = 1e-3;
```

Finally, simulate and extract solution

```
>> [solution,grids] = PIESIM(PDE,opts,uinput);
>> x1val = solution.timedep.ode;          % x1(t)
>> x2fin = solution.final.pde{1}(:,1);    % x2(t=tf,s1);
>> x3fin = solution.final.pde{1}(:,2);    % x3(t=tf,s2);
>> x4fin = solution.final.pde{2};         % x4(t=tf,s1,s2);
>> s1_grid = grids.phys(:,1);
>> s2_grid = grids.phys(:,2);
```

Running this code, the solution to the PDE (6.2) is simulated up to time $t = 1$, using a time step of $\Delta t = 10^{-3}$. Spatial discretization is performed using 16×16 Chebyshev polynomials in the spatial variables s_1 and s_2.

Note that, since we are simulating a 2D PDE, the output field `solution.final.pde` (as well as `solution.timedep.pde`) will be a cell, with the first element containing the solution of the 1D PDE states at the final time and each grid point, and the second element containing the solution of the 2D PDE state at the final time and all grid points. In this case, the simulated value of $\mathbf{x}_2(1, s_1)$ at all grid points in $s_1 \in [0, 1]$ is stored in the first column `solution.final.pde1(:,1)`, and the simulated value of $\mathbf{x}_3(1, s_2)$ is stored in the second column `solution.final.pde1(:,2)`. The output `x4fin=solution.final.pde2` will be an $(16+1)\times(16+1)$ array, with each element `x4fin(i,j)` specifying the value of $\mathbf{x}_4(t, s_1, s_2)$ at the `i`th grid point along $s_2 \in [0, 5]$, and the `j`th gridpoint along $s_1 \in [0, 1]$. The values of these grid points are stored in the output field

`grid.phys`, with the first column specifying grid points along the first spatial direction s_1, and the second column specifying grid points along the second spatial direction s_2.

The simulated value of the 1D and 2D PDE states at the final time $t = 1$ are displayed in Fig. 6.2 and Fig. 6.3, respectively, along with the value of the exact solution at that time.

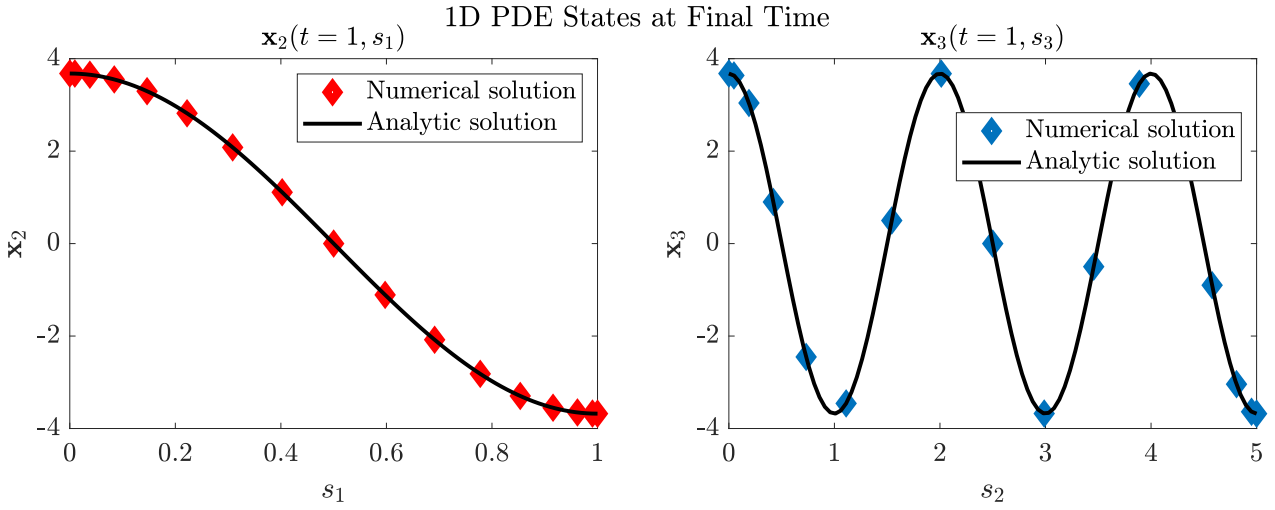


Figure 6.2: Numerical and true values of the 1D PDE states $\mathbf{x}_2(t)$ and $\mathbf{x}_3(t)$ from the ODE-PDE (6.2) at $t = 1$, simulated with initial conditions as in (6.3).

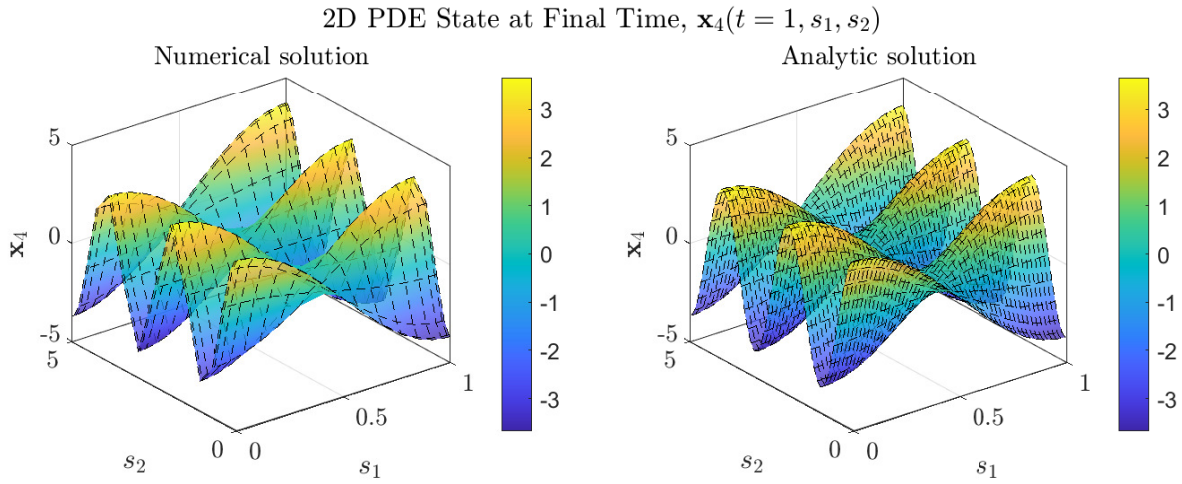


Figure 6.3: Numerical and true values of the 2D PDE state $\mathbf{x}_4(t)$ from the ODE-PDE (6.2) at $t = 1$, simulated with initial conditions as in (6.3).

6.4.3 PIESIM Demonstration C: DDE example

Simulation of DDEs can be performed using the same steps as the simulation of PDEs, only now passing a DDE structure rather than PDE structure as first argument to the `PIESIM()` function. To illustrate, consider a DDE system,

$$\dot{x}(t) = \begin{bmatrix} -1 & 2 \\ 0 & 1 \end{bmatrix} x(t) + \begin{bmatrix} 0.6 & -0.4 \\ 0 & 0 \end{bmatrix} x(t-\tau_a) + \begin{bmatrix} 0 & 0 \\ 0 & -0.5 \end{bmatrix} x(t-\tau_b) + \begin{bmatrix} 1 \\ 1 \end{bmatrix} w(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t) \quad (6.4)$$

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

where $\tau_a = 1$ and $\tau_b = 2$. As shown in Chapter 4, this system can be represented in PIETOOLS as a structure `DDE` as follows:

```
>> DDE.A0=[-1 2;0 1];          DDE.Ai{1}=[.6 -.4; 0 0];
>> DDE.Ai{2}=[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];
```

We simulate the system with an initial state $x_1(-s) = x_2(-2s) = 1$ for all $s \in [0, 1]$, and with a disturbance $w(t) = -4t - 4$, which we declare as simply

```
>> uinput.w(1) = -4*st-4;
>> uinput.u(1) = 0;
```

Here, by not specifying any initial conditions, the initial values will default to 1. Now, for the remaining simulation options, we use the same settings as in Subsection 6.4.1:

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

Then, we can simulate the system and extract the solution as simply

```
>> solution = PIESIM(DDE,opts,uinput);
>> tval = solution.timedep.dtime;
>> xval = solution.timedep.ode;
>> zval = solution.timedep.regulated;
```

Note here that the simulated value of the DDE state is stored in `solution.timedep.ode` – there is no separate field for DDE solutions. Element `xval(i,j)` will then give the simulated value of the `i`th state at time `tval(j)`. The simulated evolution of the DDE states is displayed in Figure 6.4

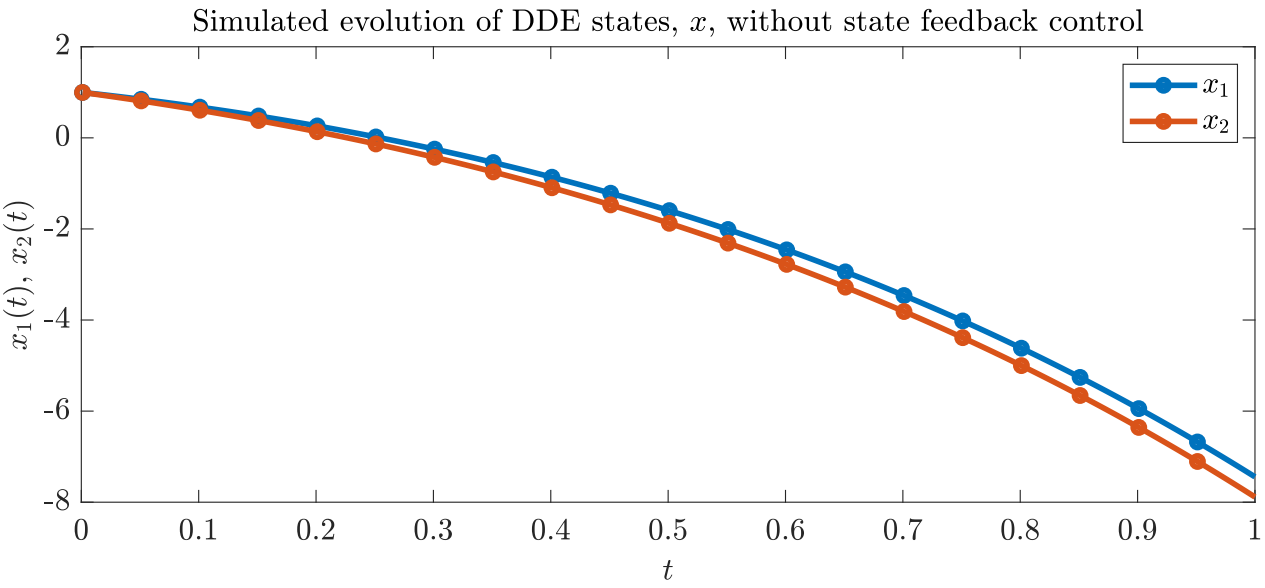


Figure 6.4: State solutions $x_1(t)$ and $x_2(t)$ to the DDE (6.4) simulated up to $t = 1$, with unit initial conditions and disturbance $w(t) = -4t - 4$.

6.4.4 PIESIM Demonstration D: PIE example

Simulating the DDE from Section 6.4.3, it appears that this system is unstable. To resolve this, we can use PIETOOLS to design a stabilizing controller for the PIE, and then visualize the behaviour of the system under the action of the controller by simulating the closed-loop system. However, controller synthesis in PIETOOLS is performed in terms of the PIE representation, and conversion of a PIE back to DDE/PDE format is often tricky. Fortunately, `PIESIM()` also supports simulation of systems in the PIE representation, by passing a `pie_struct` object as first argument to the function. To illustrate, consider again the DDE (6.4), which we declared as a `DDE` structure in the previous subsection. We can synthesize a controller for this system by first converting it to a PIE, and then calling the controller synthesis LPI (see also Sec. 13.3) as follows

```
>> PIE = convert_PIETOOLS_DDE(DDE,'pie');
>> [~, K, gam] = lpiscript(PIE,'hinf-controller','light');
>> PIE_CL = closedLoopPIE(PIE,K);
```

Here, `PIE` will be a `pie_struct` objecting representing the PIE representation of our DDE, expressed in terms of a fundamental state $\mathbf{x}_\mathrm{f}(t)$. The output `K` will then be an `opvar` object representing the optimal controller gain $\mathcal{K}$ defining the control law $u(t) = \mathcal{K}\mathbf{x}_\mathrm{f}(t)$. Finally, `PIE_CL` will be another `pie_struct` object, representing the PIE dynamics of the system with the feedback law $u(t) = \mathcal{K}\mathbf{x}_\mathrm{f}(t)$ imposed. Now, to simulate the behaviour of this system, we can simply call PIESIM for the object `PIE_CL`. Here, we will use the same options for numerical integration and discretization as in the previous subsection, setting

```
>> opts.plot = 'yes';
>> opts.N = 8;
>> opts.tf = 1;
>> opts.Norder = 2;
```

```
>> opts.dt = 1e-3;
```

However, for the field `uinput`, we note that in this case we have no exact solution, nor any controlled input (as `PIE_CL` represents the closed-loop system), so we declare only the disturbance $w(t) = -4t - 4$

```
>> clear uinput;    syms st
>> uinput.w(1) = -4*st-4;
```

Note that, since we are now simulating a PIE, initial conditions would have to be declared for the PIE state – not for the DDE or PDE state. However, since we want to model the response to a zero initial state, we declare no initial conditions in this case, so that the initial state will default to zero. Finally, in order to simulate a PIE system, we must also specify how many state variables there are differentiable up to each order. In our case, each of our two DDE state $x_1(t), x_2(t)$ will introduce a corresponding fundamental state component $\mathbf{x}_1(t,s), \mathbf{x}_2(t,s)$ that is first-order differentiable with respect to s (see also Sec. 4.3. Thus, we indicate that our system involves 2 first-order differentiable distributed state as

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

Then, we can finally simulate the closed-loop response of the system using PIESIM as

```
>> solution = PIESIM(PIE,opts,uinput,ndiff);
>> tval = solution.timedep.dtime;
>> xval = solution.timedep.ode;
>> zval = solution.timedep.regulated;
```

Here, even though we pass a PIE system to the PIESIM, the values `solution.timedep.ode` and `solution.timede.pde` will still correspond to simulated values of the primary state, not the fundamental state. The simulated evolution of the DDE state variables is displayed in Figure 6.5.

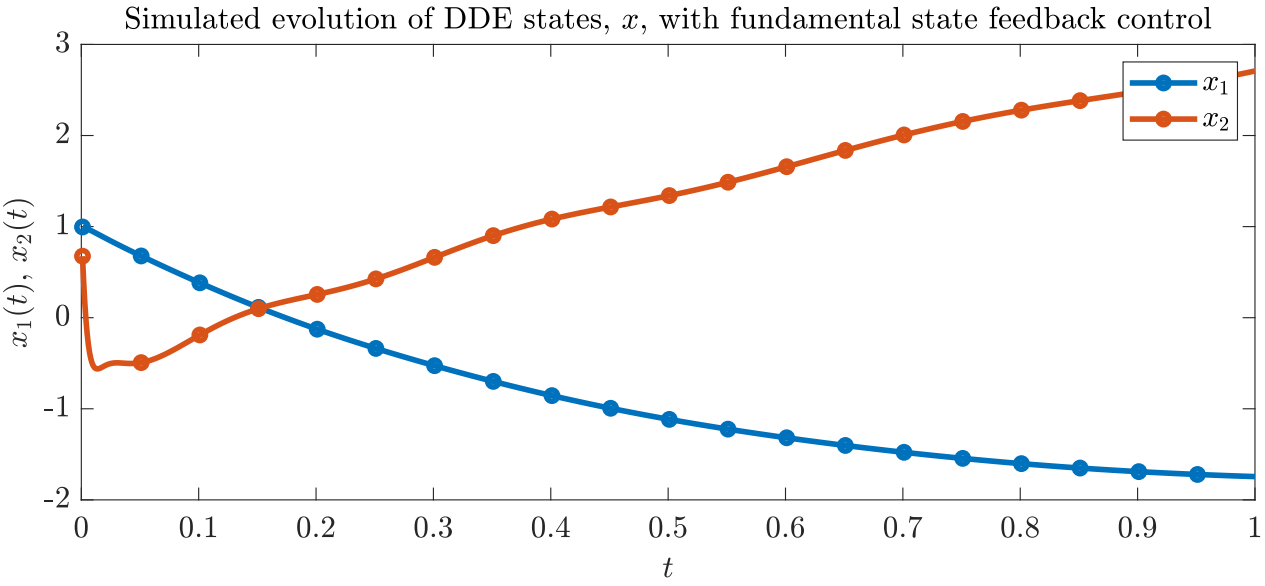


Figure 6.5: State solutions $x_1(t)$ and $x_2(t)$ to the DDE (6.4) with optimal feedback synthesized with PIETOOLS, simulated up to $t = 1$, with unit initial conditions and disturbance $w(t) = -4t - 4$.

Note

PIESIM does not currently support simulation of 2D PIEs directly. This feature will be added in a later release.

Chapter 7

Declaring and Solving Convex Optimization Programs on PI Operators

In Chapter 5 we showed how PIETOOLS can be used to derive an equivalent PIE representation of any well-posed system of linear partial differential and delay-differential equations. This PIE representation is free of the boundary conditions and continuity constraints that appear in the PDE representation, 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.

To illustrate each of these functions, we consider the problem of constructing an H_∞-optimal estimator for a simple diffusive 2D PDE. This can be done in the PIE representation by solving an LPI (see Sec. 13.2), and we will illustrate how this LPI can be declared and solved in PIETOOLS in the following sections. The full codes presented throughout this section are also included in the script "PIETOOLS_Code_Illustrations_Ch7_LPI_Programming".

Example

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

$$\begin{aligned}
\partial_t(\mathcal{T}\hat{\mathbf{x}}_{\mathrm{f}})(t) &= \mathcal{A}\hat{\mathbf{x}}_{\mathrm{f}}(t) + \mathcal{L}\big(y(t)-\hat{y}(t)\big), & \partial_t(\mathcal{T}\mathbf{x}_{\mathrm{f}}(t)) &= \mathcal{A}\mathbf{x}_{\mathrm{f}}(t) + \mathcal{B}_1 w(t),\\
\hat{z}(t) &= \mathcal{C}_1\hat{\mathbf{x}}_{\mathrm{f}}(t), & \text{for a PIE,} \qquad z(t) &= \mathcal{C}_1\mathbf{x}_{\mathrm{f}}(t) + \mathcal{D}_{11}w(t),\\
\hat{y}(t) &= \mathcal{C}_2\hat{\mathbf{x}}_{\mathrm{f}}(t), & y(t) &= \mathcal{C}_2\mathbf{x}_{\mathrm{f}}(t) + \mathcal{D}_{21}w(t), \qquad (7.1)
\end{aligned}$$

aiming to find an operator $\mathcal{L}$ that minimize the gain $\sup_{w\in L_2[0,\infty),\ w\neq 0} \frac{\|\hat{z}-z\|_{L_2}}{\|w\|_{L_2}}$. To construct such an operator, we solve the LPI,

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

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 $\|z-\hat{z}\|_{L_2}\leq\gamma\|w\|_{L_2}$. 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 corresponding to the PDE

$$\begin{aligned}
\partial_t\mathbf{x}(t,s_1,s_2) &= \partial_{s_1}^2\mathbf{x}(t,s_1,s_2) + \partial_{s_1}^2\mathbf{x}(t,s_1,s_2) + 4\mathbf{x}(t,s_1,s_2) && s_1\in[0,1],\\
&\quad + s_1(1-s_1)(s_2+1)(3-s_2)w(t), && s_2\in[-1,1],\\
\text{with BCs} \qquad 0 &= \mathbf{x}(t,0,s_2) = \mathbf{x}(t,1,s_2),\\
0 &= \mathbf{x}(t,s_1,-1) = \partial_{s_2}\mathbf{x}(t,s_1,1),\\
\text{and outputs} \qquad z(t) &= \int_0^1\int_{-1}^1 \mathbf{x}(t,s_1,s_2)ds_2ds_1 + w(t),\\
y(t,s_1) &= \mathbf{x}(t,s_1,1). \qquad (7.3)
\end{aligned}$$

We declare this PDE using the Command Line Input format as

```
>> pvar s1 s2 t
>> pde_var state x input w output z sense y;
>> x.vars = [s1;s2];   x.dom = [0,1;-1,1];
>> y.vars = s1;        y.dom = [0,1];
>> PDE = [diff(x,t)==diff(x,s1,2)+diff(x,s2,2)+4*x +s1*(1-s1)*(s2-1)*(3-s2)*w;
                  z==int(x,[s1;s2],[0,1;-1,1]) +w;
                  y==subs(x,s2,1);
               subs(x,s1,0)==0;    subs(x,s1,1)==0;
               subs(x,s2,-1)==0;   subs(diff(x,s2),s2,1)==0];
```

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');
>> 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. Such an optimization program structure must always be initialized with the function `lpiprogram` as

```
>> prog = lpiprogram(vars, dum_vars, dom, dvars, free_vars);
```

This function takes 5 inputs, only two of which are mandatory:

- `vars`: A $n \times 1$ array specifying the n spatial variables that appear in the optimization program. These variables will be independent variables in the optimization program.
- `dum_vars`: (optional) A $n \times 1$ array specifying for each of the spatial variables an associated dummy variable used for integration in the PI operators. In most cases, this argument need not be declared, in which case the function automatically generates a dummy variable for each of the specified spatial variables by adding `_dum` to the variable name (e.g. `s` yields `s_dum`). This matches the default dummy variables used throughout PIETOOLS. However, if you explicitly used different dummy variable names to define the PI operators in your optimization program (e.g. if `T.var1=s` but `T.var2=theta`), it is crucial that you specify these dummy variables when initializing the optimization program as well.
- `dom`: A $n \times 2$ array specifying for each of the n variables the lower boundary (first column) and upper boundary (second column) of the interval on which this variable exists.
- `dvars`: (optional) A $q \times 1$ array of decision variables that appear in the optimization program. Decision variables can also be added to the optimization program structure later using the functions described in Section 7.2.
- `freevars`: (optional) A $m \times 1$ array of additional independent variables in the optimization program, that are not necessary spatial variables, and therefore are not restricted to a particular domain. This field is usually left empty.

The output is a structure representing an optimization program, to which (additional) decision variables and constraints can be added as shown in the following sections.

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.

Example

For the LPI (7.2), the relevant spatial variables are $(s_1, s_2) \in [0,1] \times [-1,1]$. We initialize the optimization program structure as

```
>> prog = lpiprogram([s1;s2], [0,1;-1,1])
prog =

  struct with fields:

           var: [1×1 struct]
          expr: [1×1 struct]
      extravar: [1×1 struct]
     objective: []
       solinfo: [1×1 struct]
      vartable: [4×1 polynomial]
        varmat: [1×1 struct]
   decvartable: {}
           dom: [2x2 double]
```

This initializes an empty optimization program in the spatial variables `s1` and `s2`, existing on the intervals `[0,1]` and `[-1,1]`, respectively, and stored in the field `vartable`

```
>> prog.vartable
ans =
  [     s1]
  [     s2]
  [ s1_dum]
  [ s2_dum]
```

Note that `lpiprogram` automatically adds dummy variables `s1_dum` and `s2_dum` to the program, which match the dummy variable defining the PI operators in our PIE:

```
>> vars = PIE.vars
vars =
  [ s1, s1_dum]
  [ s2, s2_dum]
```

If, for whatever reason, you explicitly declared different dummy variables to define your PI operators (e.g. `pvar theta nu`), it is vital that you pass these dummy variables to `lpiprogram` instead (e.g. using `lpiprogram([s1;s2],[theta;nu],[0,1;-1,1]]`).

7.2 Declaring Decision Variables

Having shown 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 lpidecvar

The simplest decision variables in LPI programs are represented by scalar dpvar objects, and can be declared using lpidecvar, as e.g.

```
>> [prog,d1] = lpidecvar(prog,'d1');    % Generate decision variable with name d1
```

This function takes an optimization program structure prog, and returns the same structure but with the decision variable d1 added to it, where the output d1 is a dpvar object which can then be manipulated using standard operations such as addition and multiplication. Note that the second argument 'd1' is an optional input that merely specifies the desired name of the output decision variable, so that d1.dvarname='d1'. If the name of the decision variable is not of importance, an $m \times n$ array of decision variables can also be declared as

```
>> [prog,d_arr] = lpidecvar(prog,[m,n]);  % Generate mxn decision variable array
```

where now d_arr is an $m \times n$ dpvar object, with each element d_arr(i,j) being a decision variable named coeff_k for some k.

Example

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

```
>> [prog,gam] = lpidecvar(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: [4×1 polynomial]
        varmat: [1×1 struct]
   decvartable: {'gam'}
           dom: [2x2 double]
```

returning an optimization program structure with the variable 'gam' added to the field decvartable. The output gam is a dpvar object representing this variable, which we will use to declare constraints in Section 7.3.

7.2.2 poslpivar

In PIETOOLS, positive semidefinite PI operator decision variables $\mathcal{P} \succcurlyeq 0$ are parameterized by positive matrices $P \succcurlyeq 0$ and positive scalar-valued functions $g(s) \geq 0$ (for s in the domain) as $\mathcal{P} = \mathcal{Z}_d^*(gP)\mathcal{Z}_d \succeq 0$, 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,d,opts);
```

This function takes four inputs, two of which are mandatory:

- `prog`: An LPI program structure to which to add the PI operator decision variable.
- `n`: A 2×1 vector `[n0;n1]` specifying the dimensions of $\mathcal{P}:\left[\begin{smallmatrix}\mathbb{R}^{n0}\\ L_2^{n1}[a,b]\end{smallmatrix}\right]\to\left[\begin{smallmatrix}\mathbb{R}^{n0}\\ L_2^{n1}[a,b]\end{smallmatrix}\right]$ for a 1D operator, or a 4×1 vector `[n0;n1;n2;n3]` specifying the dimensions for a 2D operator $\mathcal{P}:\left[\begin{smallmatrix}\mathbb{R}^{n0}\\ L_2^{n1}[a,b]\\ L_2^{n2}[c,d]\\ L_2^{n3}[[a,b]\times[c,d]]\end{smallmatrix}\right]\to\left[\begin{smallmatrix}\mathbb{R}^{n0}\\ L_2^{n1}[a,b]\\ L_2^{n2}[c,d]\\ L_2^{n3}[[a,b]\times[c,d]]\end{smallmatrix}\right]$.
- `d` (optional):
  - 1D: A cell structure of the form `{a,[b0,b1,b2]}`, specifying the degree `a` of s in $Z_1(s)$, the degree `b0` of s in $Z_2(s,\theta)$, the degree `b1` of θ in $Z_2(s,\theta)$, and the degree `b2` of s and θ combined in $Z_2(s,\theta)$ (see Thm. 11).
  - 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.
- `opts` (optional): 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`, or $g(x,y)=((x-\frac{b+a}{2})-\frac{b-a}{2})^2((y-\frac{d+c}{2})-\frac{d-c}{2})^2$.
  - `sep`: Binary scalar value, constrains `P.R.R1=P.R.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 `dopvar` class object `P` representing a positive semidefinite PI operator decision variable, and an updated program structure `prog` including this decision variable. The function `poslpivar` has other experimental features to impose sparsity constraints on the T matrix of Thm. 11, which should be used with caution. Call `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,d);
>> opts.psatz = 1;
>> [prog,P2] = poslpivar(prog,n,d,opts);
>> P = P+P2;
```

This will allow P to be nonpositive 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 operator `P` of `poslpivar` is only positive semidefinite, i.e. $\mathcal{P}\succeq 0$. To ensure positive definiteness, we can add a strictly positive operator ϵI for (small) $\epsilon>0$ to $\mathcal{P}$ as e.g.

```
>> P = P + 1e-5*eye(size(P));
```

Example

In the LPI (7.2), we have one positive definite PI operator decision variable $\mathcal{P} \succ 0$. This operator $\mathcal{P}$ should have the same dimensions $\begin{bmatrix} n_0 \\ n_1 \\ n_2 \\ n_3 \end{bmatrix}$ as the operator $\mathcal{T}$, which we declare as follows:

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

With this, we indicate that the operator $\mathcal{P}$ should map $L_2[[0,1] \times [-1,1]] \to L_2[[0,1] \times [-1,1]]$. In its most general form, such an operator may be defined using multiplier and integral operators, but to minimize computational cost, we will simply choose $\mathcal{P}$ here to be a (scalar-valued) matrix, excluding any of the integral terms by using the settings

```
>> P_opts.exclude = [1;
                     1;1;1;
                     1;1;1;
                     0;1;1;1;1;1;1;1;1;1];
```

Here, the first three lines are just to exclude all operator components mapping $\mathbb{R} \to \mathbb{R}$, $L_2[0,1] \to L_2[0,1]$, and $L_2[-1,1] \to L_2[-1,1]$, respectively – which will also be excluded automatically by the specified dimensions of the operator. However, with the last line, we indicate that we also wish to exclude all integral-type operators mapping $L_2[[0,1] \times [-1,1]] \to L_2[[0,1] \times [-1,1]]$, only including a multiplier-type operator. Although this multiplier could still be defined by a polynomial, we will set the degree of this polynomial to just 0 as

```
>> Pdeg = 0;
```

We declare a positive semidefinite PI operator $\mathcal{P}$ with these specifications as

```
>> [prog,P] = poslpivar(prog,Pdim,Pdeg,P_opts)
prog =

  struct with fields:

           var: [1×1 struct]
          expr: [1×1 struct]
      extravar: [1×1 struct]
     objective: [2×1 double]
       solinfo: [1×1 struct]
      vartable: [4×1 polynomial]
        varmat: [1×1 struct]
   decvartable: {2×1 cell}
           dom: [2x2 double]
```

The output `P` here is a `dopvar2d` object, representing a PI operator decision variable rather than a fixed PI operator. As such, the parameters (e.g `P.R00`, `P.R0x`, `P.R0y`, etc.) defining this PI operator are `dpvar` class objects, parameterized by decision variables `coeff_i`. These decision variables are collected in the field `decvartable` of the program structure, and represent the matrix T in the expansion $\mathcal{P} = \mathcal{Z}_d^* T \mathcal{Z}_d$, constrained to satisfy $T \succcurlyeq 0$. With our settings, the variable $\mathcal{P}$ will just be a multiplier operator mapping $L_2[[0,1]\times[-1,1]] \to L_2[[0,1]\times[-1,1]]$, with the only non-zero parameter being stored in `P.R22{1,1}`:

```
>> P.R22{1,1}
ans =
  coeff_1
```

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 $\epsilon = 10^{-4}$, and ensure strict positivity of $\mathcal{P}$ by calling

```
>> eppos = 1e-4;
>> P = P + 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,d);
```

This function takes three inputs, two of which are mandatory:

- `prog`: An LPI program structure to which to add the PI operator decision variable.
- `n`: a 2×2 array `[m0,n0;m1,n1]` specifying the dimensions of $\mathcal{Z} : \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \end{bmatrix} \to \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a,b] \end{bmatrix}$ for a 1D operator, or 4×2 array `[m0,n0;m1,n1;m2,n2;m3,n3]` specifying the dimensions for a 2D operator $\mathcal{Z} : \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \\ L_2^{n_2}[c,d] \\ L_2^{n_3}[[a,b]\times[c,d]] \end{bmatrix} \to \begin{bmatrix} \mathbb{R}^{m_0} \\ L_2^{m_1}[a,b] \\ L_2^{m_2}[c,d] \\ L_2^{m_3}[[a,b]\times[c,d]] \end{bmatrix}$.
- `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 lpivar_2d` for more information.

The output is a `dopvar` (or `dopvar2d`) object `Z` representing an indefinite PI decision variable, and an updated program structure to which the decision variable has been added. Note that, since PI operator decision variables $\mathcal{Z}$ need not be symmetric, 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, given that $\mathcal{C}_2 : L_2[[0,1] \times [-1,1]] \to L_2[0,1]$, the operator $\mathcal{Z}$ should map $L_2[0,1] \to L_2[[0,1] \times [-1,1]]$, so we specify the dimensions of $\mathcal{Z}$ as

```
>> Zdim = C2.dim(:,[2,1])
Zdim =

     0     0
     0     1
     0     0
     1     0
```

so that in our case $\mathcal{Z} : L_2[0,1] \to L_2[[0,1] \times [-1,1]]$. As such, only the parameter `Z.R2x` will be non-empty, but this parameter may still involve both a multiplier term (`Z.R2x{1}`) and partial integral terms (`Z.R2x{2}` and `Z.R2x{3}`). We will allow each of these terms to be defined by polynomials of degree at most 2 in each of the variables, specifying degrees

```
>> Zdeg = 2;
```

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

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

  struct with fields:

           var: [1×1 struct]
          expr: [1×1 struct]
      extravar: [1×1 struct]
     objective: [33×1 double]
       solinfo: [1×1 struct]
      vartable: [4×1 polynomial]
        varmat: [1×1 struct]
  decvartable: {33×1 cell}
           dom: [2x2 double]
```

returning another `dopvar2d` object `Z`. We can verify that only the field `Z.R2x` is defined by non-zero parameters, with e.g.

```
>> Z.R2x{1}
ans =
  coeff_02 + coeff_04*s2 + coeff_07*s2^2 + coeff_03*s1 + coeff_06*s1*s2
  + coeff_09*s1*s2^2 + coeff_05*s1^2 + coeff_08*s1^2*s2 + coeff_10*s1^2
  *s2^2
```

indicating that the multiplier term is defined by a polynomial of degree 2, as desired. In total, the object `Z` is defined by 32 decision variables `coeff`, increasing the total number of decision variables in `prog.decvartable` to 33.

7.3 Imposing Constraints

Constraints form a crucial aspect of almost every optimization problem. In LPIs, constraints often appear as inequality or equality conditions on PI operators (e.g. $\mathcal{Q} \preceq 0$ or $\mathcal{Q} = 0$). 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` (or `dopvar2d`) object `Q` (representing a PI operator variable $\mathcal{Q}$), an inequality constraint $\mathcal{Q} \succeq 0$ can be added to the program by calling

```
» prog = lpi_ineq(prog,Q,opts);
```

This function takes three input arguments

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

    ```
    » R1=poslpivar(prog,n,d,opts1)
    ```

    with `opts1.psatz=0`. If `psatz=1` (or `psatz=2`), a constraint $\mathcal{Q} = \mathcal{R}_1 + \mathcal{R}_2$ will be enforced, with $\mathcal{R}_1$ as before, and $\mathcal{R}_2 \succeq 0$ declared as a `dopvar` or `dopvar2d` object

    ```
    » R2=poslpivar(prog,n,d,opts2)
    ```

    with `opts2.psatz=psatz`. Using `psatz=1` (or alternatively `psatz=2` in 2D) allows the constraint $\mathcal{Q} \succeq 0$ 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>=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 $\mathcal{Q} - \epsilon I \succeq 0$ to ensure $\mathcal{Q} \succeq \epsilon I \succ 0$. This may be implemented as e.g.

```
>> prog = lpi_ineq(prog,Q-ep*eye(size(Q)));
```

where `ep` is a small positive number.

Example

For the LPI (7.2), we impose the constraint $\mathcal{Q} \preccurlyeq 0$ by calling

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

  struct with fields:

           var: [1×1 struct]
          expr: [1×1 struct]
      extravar: [1×1 struct]
     objective: [102185×1 double]
       solinfo: [1×1 struct]
      vartable: [4×1 polynomial]
        varmat: [1×1 struct]
   decvartable: {102185×1 cell}
           dom: [2x2 double]
```

We note that `lpi_ineq` enforces the constraint $-\mathcal{Q} \succcurlyeq 0$ by introducing a new PI operator decision variable $\mathcal{R} \succcurlyeq 0$, and imposing the equality constraint $-\mathcal{Q} = \mathcal{R}$. In doing so, `lpi_ineq` tries to ensure that the degrees of the polynomial parameters defining $\mathcal{R}$ match those of the parameters defining $-\mathcal{Q}$, in this case parameterizing $\mathcal{R}$ by $102185-33 = 102152$ decision variables (check the size of `decvartable`). An operator $\mathcal{R}$ involving more or fewer decision variables can also be declared manually, at which point the constraint $-\mathcal{Q} = \mathcal{R}$ can be enforced using `lpi_eq`, as we show next.

7.3.2 `lpi_eq`

Given a program structure `prog` and `dopvar` (or `dopvar2d`) object `Q` (representing a PI operator decision variable $\mathcal{Q}$), an equality constraint $\mathcal{Q} = 0$ 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. Since `opvar` objects are defined by parameters (`Q.P`, `Q.Q1`, `Q.Q2`, etc.), each of these parameters will be set to zero in the optimization program. Here, if `Q` is symmetric, we note that e.g. `Q.Q1=Q.Q2`, and therefore only `Q.Q1=0` needs to be enforced. To exploit this fact, an argument `'symmetric'` can be passed to `lpi_eq` as

```
» prog = lpi_eq(prog,Q,'symmetric');
```

This argument is not mandatory, but can reduce the number of equality constraints in the optimization program, and therefore the computational cost of solving it. Of course, this argument should only be passed if `Q` is indeed symmetric!

Example

For the LPI (7.2), we can enforce the constraint $\mathcal{Q} \preccurlyeq 0$ by declaring a new positive semidefinite PI operator decision variable $\mathcal{R} \succeq 0$, and enforcing $\mathcal{Q} = -\mathcal{R} \preccurlyeq 0$. In particular, as an alternative to using `lpi_ineq` as in the previous subsection, we can call

```
>> Rdeg = 3;       R_opts.psatz = 1;
>> [prog_alt,R1] = poslpivar(prog,Q.dim(:,1),Rdeg);
>> [prog_alt,R2] = poslpivar(prog_alt,Q.dim(:,1),Rdeg-1,R_opts);
>> prog_alt = lpi_eq(prog_alt,Q+R1+R2,'symmetric');
prog_alt =

  struct with fields:

           var: [1×1 struct]
          expr: [1×1 struct]
      extravar: [1×1 struct]
     objective: [105670×1 double]
       solinfo: [1×1 struct]
      vartable: [4×1 polynomial]
        varmat: [1×1 struct]
   decvartable: {105670×1 cell}
           dom: [2x2 double]
```

Here, we chose the degrees of `R` to be slightly larger than those we used to define `P`, hoping to make sure that all monomials appearing in `Q` can then be matched by those in `R`. The new program structure includes the constraints $\mathcal{R} \succcurlyeq 0$ and $\mathcal{Q} + \mathcal{R} = 0$. The resulting total number of decision variables (105670) is somewhat larger than in the program obtained using `lpi_ineq` in the previous subsection, as the specified degrees `Rdeg` used to define `R` are quite large. Naturally, we can reduce the number of decision variables and thus the computational cost of solving by specifying smaller degrees `Rdeg`, but this may come at the cost of greater conservatism.

7.4 Defining an Objective Function

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 $f(d_1, \ldots, d_1)$, where $d_1, \ldots, d_n$ are decision variables, call `lpisetobj` with as first argument the program structure, and as second argument the function $f(d)$:

```
>> prog = lpisetobj(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

```
>> [prog,gam] = lpidecvar(prog,[1,2]);
>> f = gam(1) + 5*gam(2);
>> prog = lpisetobj(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 $f(d)$, 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 = lpisetobj(prog, gam);
prog =

  struct with fields:

           var: [1×1 struct]
          expr: [1×1 struct]
      extravar: [1×1 struct]
     objective: [102185×1 double]
       solinfo: [1×1 struct]
      vartable: [4×1 polynomial]
        varmat: [1×1 struct]
   decvartable: {102185×1 cell}
           dom: [2x2 double]
```

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, indicating that the objective function is equal to 1 times the first decision variable in `decvartable`, 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 `lpisolve`

```
>> prog_sol = lpisolve(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 `lpisolve`, 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 `lpisolve` 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

```
>> solve_opts.solver = 'sedumi';
>> solve_opts.simplify = true;
>> prog_sol = lpisolve(prog,solve_opts);

Residual norm: 0.00031666

         iter: 18
    feasratio: -0.0725
         pinf: 0
         dinf: 0
       numerr: 2
       timing: [0.9840 223.2360 0.0320]
      wallsec: 224.2520
       cpusec: 114.1406
```

The returned value of `feasratio` is close to zero, and `numerr` is 2, indicating that SeDuMi has run into serious numerical problems. This is not uncommon when solving LPI optimization programs where we are simultaneously testing feasibility of some (large) LPI constraints and minimizing an objective. In such cases, it is often a good idea to run the optimization program for a fixed a value of the objective function, only testing feasibility of the LPI, and manually performing bisection on the value of the objective function coefficients if necessary. For example, rerunning this same test but now fixing a value `gam==2` a priori, `lpisolve` returns an output structure

```
         iter: 15
    feasratio: 0.8673
         pinf: 0
         dinf: 0
       numerr: 1
       timing: [0.3380 2.7187e+02 0.0850]
      wallsec: 2.7227e+02
       cpusec: 133
```

Here, we have `pinf=0` and `dinf=0`, indicating that the proposed problem was not found to be primal or dual infeasible, and the feasratio is quite close to 1. Thus, it appears that the optimization program was successfully solved, and the LPI (7.2) is feasible for $\gamma = 2$.

7.6 Extracting the Solution

Calling `prog_sol=lpisolve(prog)`, a program structure `prog_sol` is returned that is very similar to the input structure `prog`, defining the solved optimization problem. Given this solved LPI program structure, we can retrieve the (optimal) value of any decision variable appearing in the LPI – be it a function specified as an object of type `dpvar`, or an operator specified as an object of type `dopvar` – using the function `lpigetsol`, passing the solved optimization program structure `prog_sol` as first argument, and the considered `dpvar` or `dopvar` (`dopvar2d`) decision variable as second argument:

```
>> f_val = lpigetsol(prog_sol,f);
>> Pop_val = lpigetsol(prog_sol,Pop);
```

Note that the lpiprogram `prog_sol` must be in a solved state (`lpisolve` must have been called) to retrieve the solution for the input `dpvar`, `dopvar`, or `dopvar2d` object. The output `f_val` will then by a `polynomial` class object, and `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 optimal value of γ found when solving the LPI (7.2), we call

```
>> gam_val = lpigetsol(prog_sol,gam)
gam_val =
  1.1344
```

We find that, through proper choice of the operator $\mathcal{L}$, the estimator can achieve an H_∞-norm $\frac{\|\tilde{z}\|_{L_2}}{\|w\|_{L_2}} \leq 1.1344$. Of course, since the optimization program was not found to be feasible, this bound may not actually be accurate, and we should instead continue with the solved program structure obtained when fixing $\gamma = 2$. To extract the values of the operators $\mathcal{P}$ and $\mathcal{Z}$ achieving this gain, we call

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

The resulting object `Pval` and `Zval` are `opvar2d` 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 $\frac{\|\tilde{z}\|_{L_2}}{\|w\|_{L_2}} \leq \gamma = 2$, as we show next.

When performing estimator or controller synthesis (see also Chapter 13), the optimal estimator or controller associated with 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.

7.6.1 `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` (`opvar2d`) objects representing the (optimal) values of $\mathcal{P}$ and $\mathcal{Z}$ in the LPI, and `Lval` is an `opvar` (`opvar2d`) object representing the associated (optimal) value of $\mathcal{L}$ in the estimator.

Example

Given the `opvar2d` 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 =

    [] |       [] |       [] |       []
    ----------------------------------
    [] | Lval.Rxx |       [] | Lval.Rx2
    ----------------------------------
    [] |       [] | Lval.Ryy | Lval.Ry2
    ----------------------------------
    [] | Lval.R2x | Lval.R2y | Lval.R22

Lval.R2x =

    Too big to display | Too big to display | Too big to display
```

where the expression for `Lval.Rx2` is rather complicated. Nevertheless, using this value for the operator $\mathcal{L}$, an H_∞ norm $\frac{\|\tilde{z}\|_{L_2}}{\|w\|_{L_2}} \leq \gamma = 2$ can be achieved. Performing bisection on the value of γ, and perhaps 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.

7.6.2 `getController`

For a solution $(\mathcal{P}, \mathcal{Z})$ to the optimal control LPI (13.21) (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 (13.20) can be computed as

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

where `Pval` and `Zval` are `opvar` (`opvar2d`) objects representing the (optimal) values of $\mathcal{P}$ and $\mathcal{Z}$ in the LPI, and `Kval` is an `opvar` (`opvar2d`) 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.

7.6.3 `closedLoopPIE`

For a PIE (13.20) 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 = closedLoopPIE(PIE,Kval);
```

where `Kval` is an `opvar` (`opvar2d`) object representing the value of $\mathcal{K}$ in the feedback law $u = \mathcal{K}\mathbf{v}$, `PIE` 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.

In addition, for a PIE and an operator $\mathcal{L}$ defining a Luenberger estimator gain, a PIE for the closed-loop observed system can be computed as

```
>> PIE_CL = closedLoopPIE(PIE,Lval,'observer');
```

where `Lval` is an `opvar` (`opvar2d`) object representing the value of $\mathcal{L}$ in the estimator in (7.1), `PIE` is a `pie_struct` object representing the PIE system in (7.1), and `PIE_CL` is a `pie_struct` object representing the closed-loop PIE system of the form

$$\partial_t \left(\begin{bmatrix} \mathcal{T} & 0 \\ 0 & \mathcal{T} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{\mathrm{f}} \\ \hat{\mathbf{x}}_{\mathrm{f}} \end{bmatrix} \right)(t,s) = \left(\begin{bmatrix} \mathcal{A} & 0 \\ -\mathcal{L}\mathcal{C}_2 & \mathcal{A}+\mathcal{L}\mathcal{C}_2 \end{bmatrix} \begin{bmatrix} \mathbf{x}_{\mathrm{f}} \\ \hat{\mathbf{x}}_{\mathrm{f}} \end{bmatrix} \right)(t,s) + \left(\begin{bmatrix} \mathcal{B}_1 \\ \mathcal{L}\mathcal{D}_{21} \end{bmatrix} w \right)(t)$$
$$\begin{bmatrix} z \\ \hat{z} \end{bmatrix}(t) = \left(\begin{bmatrix} \mathcal{C}_1 & 0 \\ 0 & \mathcal{C}_1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_{\mathrm{f}} \\ \hat{\mathbf{x}}_{\mathrm{f}} \end{bmatrix} \right)(t) + \left(\begin{bmatrix} \mathcal{D}_{11} \\ 0 \end{bmatrix} w \right)(t)$$

so that e.g. `PIE_CL.A` represents the operator $\left[\begin{smallmatrix} \mathcal{A} & 0 \\ -\mathcal{L}\mathcal{C}_2 & \mathcal{A}+\mathcal{L}\mathcal{C}_2 \end{smallmatrix}\right]$. Note that, for a PIE with state $\mathbf{x}_{\mathrm{f}}(t) \in L_2^n$ and output $z(t) \in \mathbb{R}$, the state and output of the closed-loop observed system are respectively given by $\left[\begin{smallmatrix} \mathbf{x}_{\mathrm{f}(t)} \\ \hat{\mathbf{x}}_{\mathrm{f}}(t) \end{smallmatrix}\right] \in L_2^{2n}$ and $\left[\begin{smallmatrix} z(t) \\ \hat{z}(t) \end{smallmatrix}\right]$, where $\hat{\mathbf{x}}_{\mathrm{f}}(t)$ and $\hat{z}(t)$ are estimated values of the PIE state and regulated output, respectively. However, for PIEs with coupled finite- and infinite-dimensional states $\left[\begin{smallmatrix} x(t) \\ \mathbf{x}_{\mathrm{f}}(t) \end{smallmatrix}\right] \in \left[\begin{smallmatrix} \mathbb{R}^m \\ L_2^n \end{smallmatrix}\right]$, the state of the closed-loop system will be of the form $\left[\begin{smallmatrix} x(t) \\ \hat{x}(t) \\ \mathbf{x}_{\mathrm{f}}(t) \\ \hat{\mathbf{x}}_{\mathrm{f}}(t) \end{smallmatrix}\right] \in \left[\begin{smallmatrix} \mathbb{R}^{2m} \\ L_2^{2n} \end{smallmatrix}\right]$. This is because `opvar` objects can only represent maps of states in $\mathbb{R} \times L_2$, not e.g. states in $L_2 \times \mathbb{R}$, and so the state components and their estimates will be organized accordingly.

Example

A full code declaring and solving the optimal estimator LPI (7.2) for the PDE (7.3) has been included as a script "PIETOOLS_Code_Illustrations_Ch7_LPI_Programming" in PIETOOLS. An alternative demonstration for how an optimal estimator can be constructed and simulated is also given in Section 11.5.

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 = lpiprogram(PIE.vars,PIE.dom);

>> % Declare the objective function min{gamma}
>> [prog,gam] = lpidecvar(prog, 'gam');
>> prog = lpisetobj(prog, gam);

>> % Declare the positive operator P>=0
>> [prog,P] = poslpivar(prog,T.dim,dd1,options1);
>> if override1==0
>>     % Allow P<=0 outside domain PIE.dom
>>     [prog,P2] = poslpivar(prog,T.dim,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
>> Iw = eye(size(B1,2));    Iz = eye(size(C1,1));
>> Q = [-gam*Iw,           -D11',    -(P*B1+Z*D21)'*T;
        -D11,              -gam*Iz,  C1;
        -T'*(P*B1+Z*D21),  C1',      (P*A+Z*C2)'*T+T'*(P*A+Z*C2)+epneg*T'*T];
>> if use_ineq
>>     % 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,'symmetric');
>> end

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

We note that, in this code, 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.

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 sosineq
override1	Binary value, set 1 to let P2op = 0
override2	Binary value, set 1 to let De2op = 0
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 poslpivar options for Pop
options12	Structure of poslpivar options for P2op
options2	Structure of poslpivar options for Deop
options3	Structure of poslpivar options for De2op
opts	Structure of lpi_ineq options for enforcing Dop ≤ 0
sos_opts	Structure of sossolve options for solving the LPI

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

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 $Q \preccurlyeq \epsilon\|\mathcal{T}\|^2$ would need to be in e.g. the LPI for stability. Defaults to 0, enforcing $Q \preccurlyeq 0$.

- 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);
```

Alternatively, this executive can also be called using the function `lpiscript` as

```
>> [prog, Lop, gam, P, Z] = lpiscript(PIE,'hinf-observer','light');
```

If successful, this returns the program structure `prog` associated to the solved problem, as well as an `opvar` object `Lop` and scalar `gam` xcorresponding to the operator $\mathcal{L}$ in the estimator (7.1) and associated estimation error gain γ, respectively. 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

```
>> [prog] = lpiscript(PIE,'stability','light');
```

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

```
      dim: N;
     vars: [N×2 polynomial];
      dom: [N×2 double];

        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

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

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] = lpiscript(PIE,'stability',settings)`	
Test stability of the PIE for $w = 0$ and $u = 0$, by verifying feasibility of the primal LPI [8].	$\mathcal{P} \succ 0$ $\mathcal{T}^*\mathcal{P}\mathcal{A} + \mathcal{A}^*\mathcal{P}\mathcal{T} \preccurlyeq 0$
`[prog, P] = lpiscript(PIE,'stability-dual',settings)`	
Test stability of the PIE for $w = 0$ and $u = 0$ by verifying feasibility of the dual LPI [9].	$\mathcal{P} \succ 0$ $\mathcal{T}\mathcal{P}\mathcal{A}^* + \mathcal{A}\mathcal{P}\mathcal{T}^* \preccurlyeq 0$
`[prog, P, gam] = lpiscript(PIE,'l2gain',settings)`	
Determine an upper bound γ on the $\mathcal{H}_\infty$-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].	$\min\limits_{\gamma,\mathcal{P}} \quad \gamma$ $\mathcal{P} \succ 0$ $\begin{bmatrix} -\gamma I & \mathcal{D}_{11}^* & \mathcal{B}_1^*\mathcal{P}\mathcal{T} \\ (\cdot)^* & -\gamma I & \mathcal{C}_1 \\ (\cdot)^* & (\cdot)^* & \mathcal{T}^*\mathcal{P}\mathcal{A} + \mathcal{A}^*\mathcal{P}\mathcal{T} \end{bmatrix} \preccurlyeq 0$
`[prog, P, gam] = lpiscript(PIE,'l2gain-dual',settings)`	
Determine an upper bound γ on the $\mathcal{H}_\infty$-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].	$\min\limits_{\gamma,\mathcal{P}} \quad \gamma$ $\mathcal{P} \succ 0$ $\begin{bmatrix} -\gamma I & \mathcal{D}_{11} & \mathcal{T}\mathcal{P}\mathcal{C}_1 \\ (\cdot)^* & -\gamma I & \mathcal{B}_1^* \\ (\cdot)^* & (\cdot)^* & \mathcal{T}\mathcal{P}\mathcal{A}^* + \mathcal{A}\mathcal{P}\mathcal{T}^* \end{bmatrix} \preccurlyeq 0$
`[prog, L, gam, P, Z] = lpiscript(PIE,'hinf-observer',settings)`	
Establish an $\mathcal{H}_\infty$-optimal observer $\mathcal{T}\dot{\hat{\mathbf{x}}}_\mathrm{f} = A\hat{\mathbf{x}}_\mathrm{f} + \mathcal{L}(\mathcal{C}_1\hat{\mathbf{x}}_\mathrm{f} - y)$ for the PIE with $u = 0$ by solving the LPI and returning $\mathcal{L} = \mathcal{P}^{-1}\mathcal{Z}$ [1].	$\min\limits_{\gamma,\mathcal{P},\mathcal{Z}} \quad \gamma$ $\mathcal{P} \succ 0$ $\begin{bmatrix} -\gamma I & \mathcal{D}_{11}^* & (\mathcal{P}\mathcal{B}_1 + \mathcal{Z}\mathcal{D}_{21})^*\mathcal{T} \\ (\cdot)^* & -\gamma I & \mathcal{C}_1 \\ (\cdot)^* & (\cdot)^* & (\cdot)^* + (\mathcal{P}\mathcal{A} + \mathcal{Z}\mathcal{C}_2)^*\mathcal{P}\mathcal{T} \end{bmatrix} \preccurlyeq 0$
`[prog, K, gam, P, Z] = lpiscript(PIE,'hinf-controller',settings)`	
Establish an $\mathcal{H}_\infty$-optimal controller $u = \mathcal{K}\mathbf{x}_\mathrm{f}$ for the PIE by solving the LPI and returning $\mathcal{K} = \mathcal{Z}\mathcal{P}^{-1}$ [9].	$\min\limits_{\gamma,\mathcal{P},\mathcal{Z}} \quad \gamma$ $\mathcal{P} \succ 0$ $\begin{bmatrix} -\gamma I & \mathcal{D}_{11} & \mathcal{T}(\mathcal{P}\mathcal{C}_1 + \mathcal{Z}\mathcal{D}_{12}) \\ (\cdot)^* & -\gamma I & \mathcal{B}_1^* \\ (\cdot)^* & (\cdot)^* & (\cdot)^* + (\mathcal{A}\mathcal{P} + \mathcal{B}_2\mathcal{Z})\mathcal{T}^* \end{bmatrix} \preccurlyeq 0$

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

Input Formats for ODE-PDE Systems

In PIETOOLS, there are three main methods for declaring coupled ODE-PDE systems: The Command Line Input format (via command line or MATLAB scripts), the graphical user interface (a MATLAB app), and using the `sys` structure. The first two of these methods have been presented in Chapter 4, and are the recommended input formats. The last of these methods, using the `sys` structure, was introduced in PIETOOLS 2022 as a Command Line Parser format, and functions similarly to the current Command Line Input format. Although the `sys` structure functionality is still available in PIETOOLS 2024, it supports at most 1D ODE-PDE systems, and it is therefore recommended to use `pde_var` instead.

In this chapter, we provide a bit more background on each of the three input formats for ODE-PDE systems. 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 shift focus to the Command Line Input format, providing a detailed description of how general ODE-PDE systems can be declared using this format, and how this format actually works "behind the scenes". Finally, in Section 8.3, we show how the `sys` structure can be used to declare 1D ODE-PDE models as well.

8.1 A GUI for Defining PDEs

In addition to the Command Line Input format, PIETOOLS 2024 also allows PDEs to be delcared 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_PDE_GUI` from the command line. You will see something similar to the picture below:

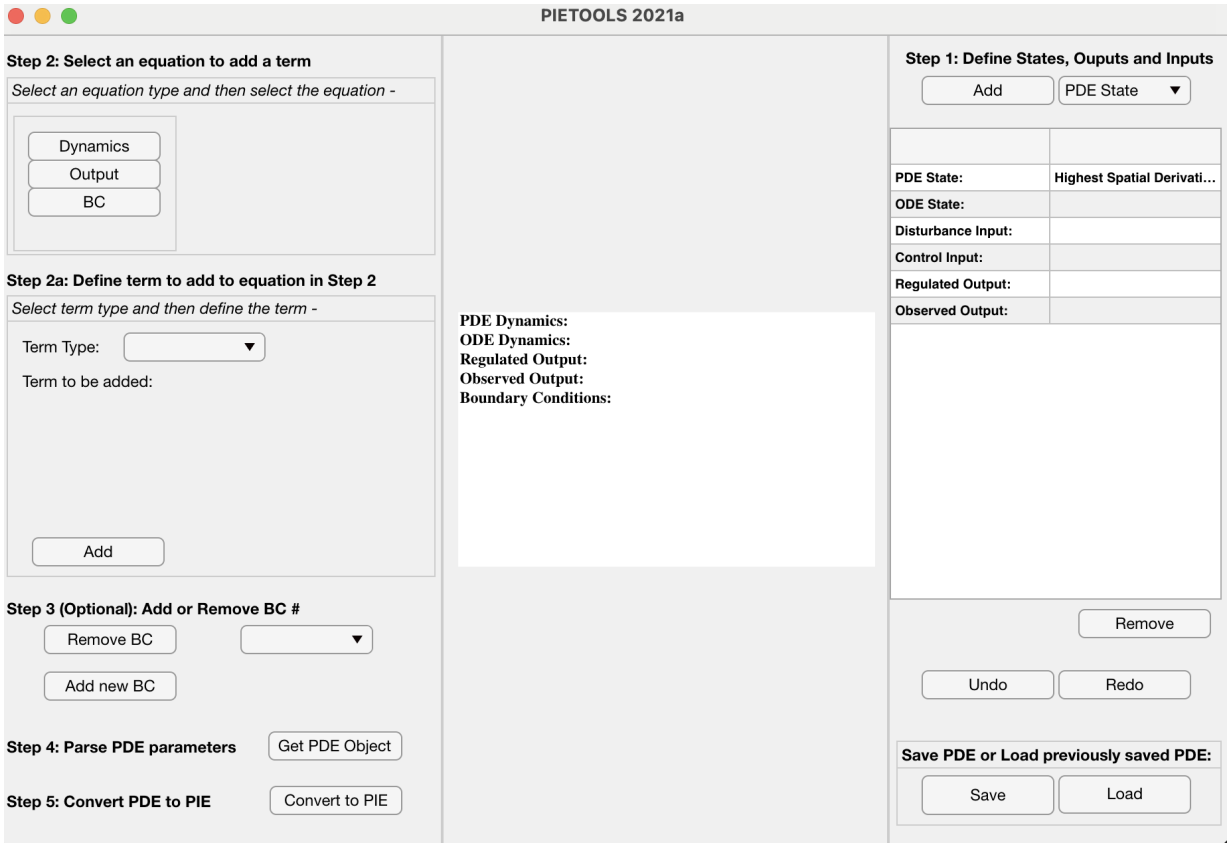


Figure 8.1: GUI overview.

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:

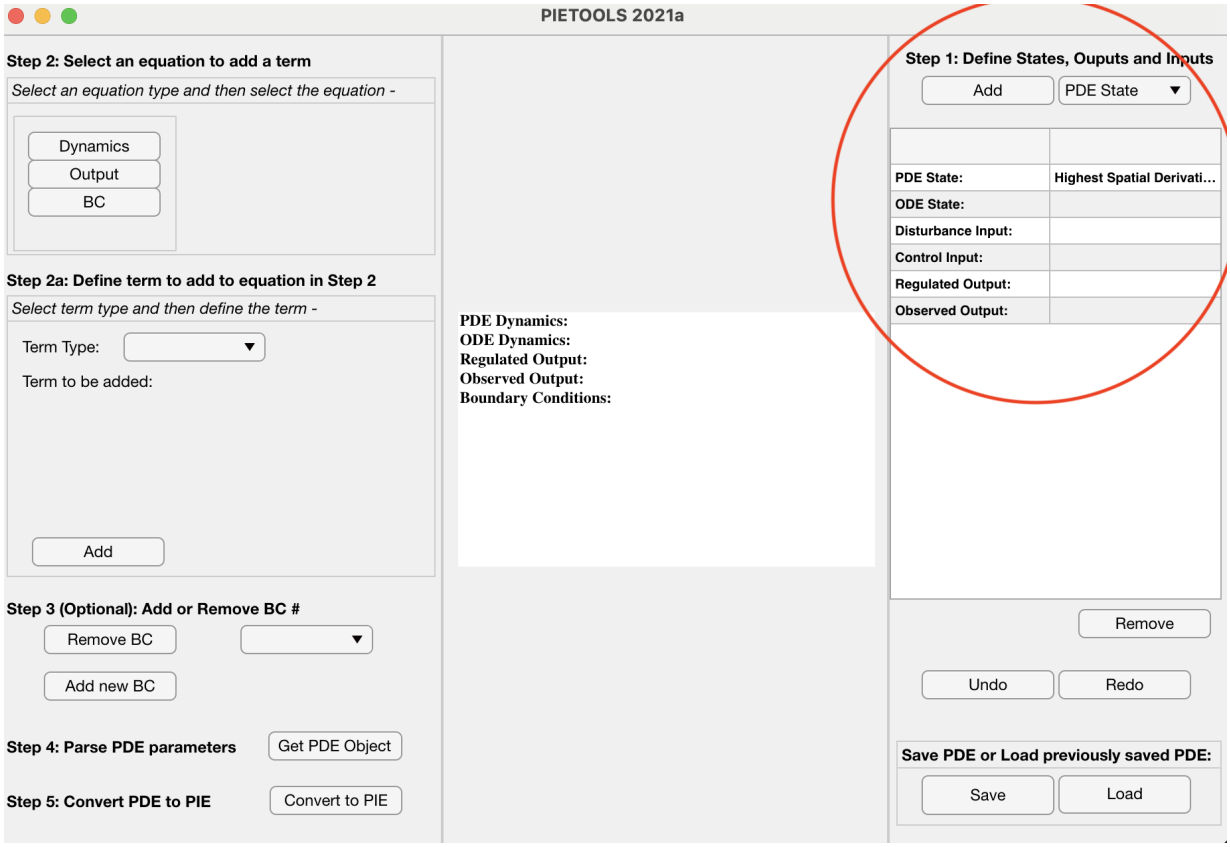


Figure 8.2: Step 1: Define States, Outputs and Inputs

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

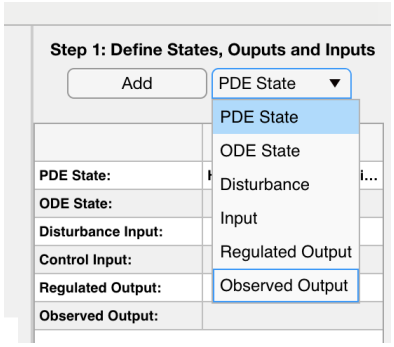


Figure 8.3: Adding variables your model

2. After selecting your intended variable, you can add it by clicking on the `Add` button. Note that ehen you select `PDE State` from the drop-down menu and attempt to a PDE state, you also have to specify the highest order of spatial derivative that the particular state admits.

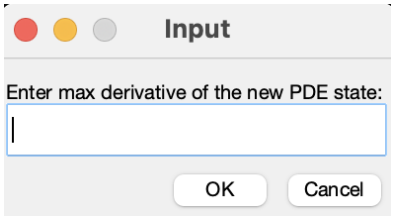


Figure 8.4: Enter the highest order of derivative the particular state admits

3. Once the variables are added, they automatically get displayed in the display panel in the middle. Since no dynamics have been specified for the model so far, all the variables are set to the default setting temporarily.

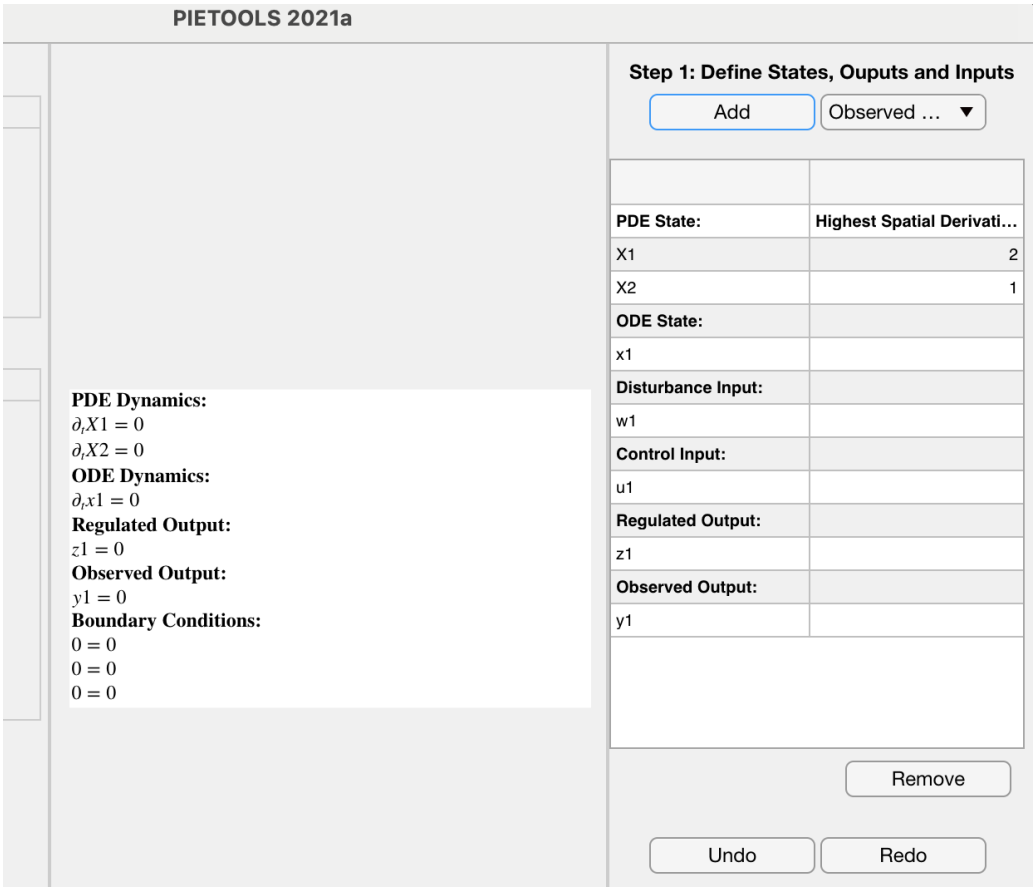


Figure 8.5: After adding the variables to the model

4. 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 of the variables defined in Step 1. This is located on the left hand side of the GUI.

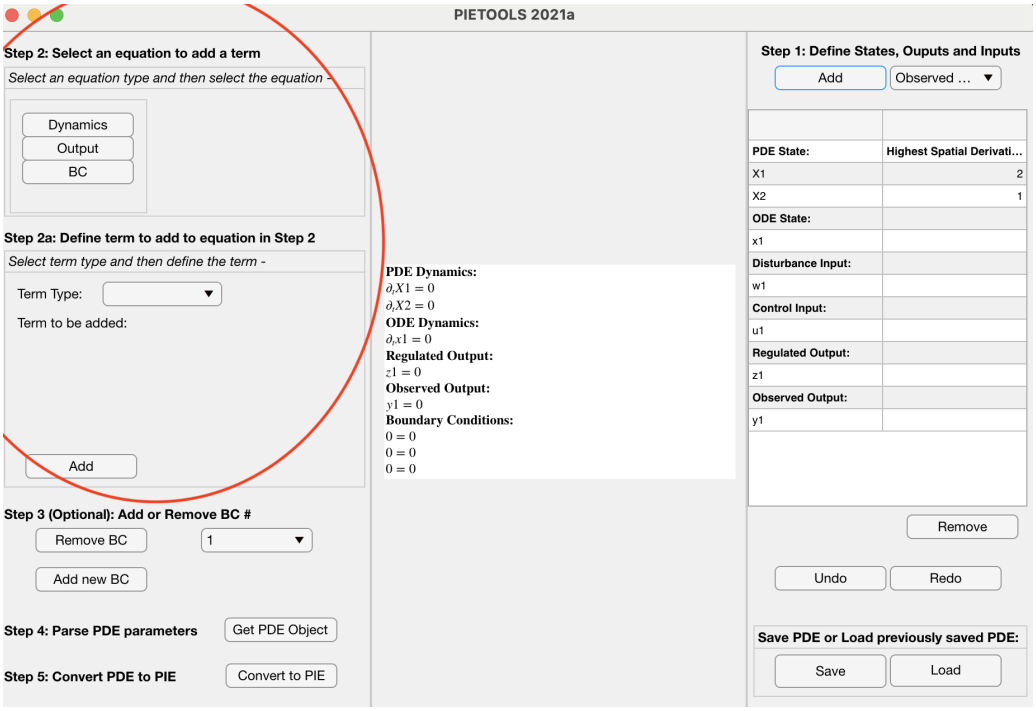
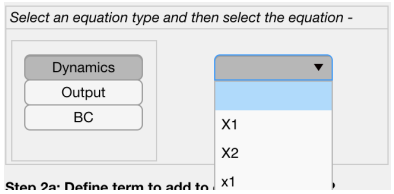


Figure 8.6: Step 2: Select an Equation to Add a Term

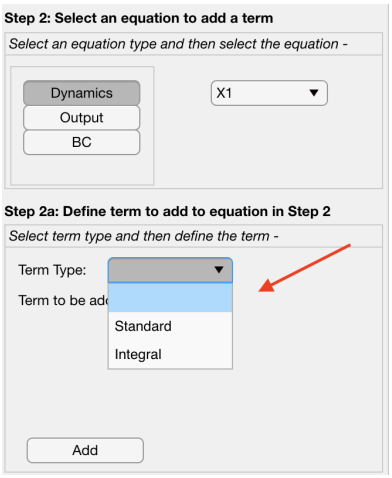
This has two parts. On the top, we have a panel for `Select an Equation type and then select the equation-` to choose which part of the model to be defined. Below that panel, there is another panel that has to be used to `Select term type and then define the term-`.

1. In the panel titled `Select an Equation type and then select the equation-`, select either `Dynamics`, `Output` or `BC` (i.e. Boundary Conditions).

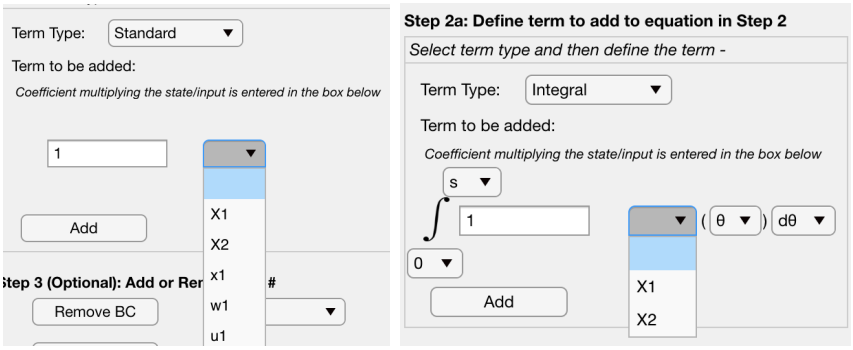
2. If you select `Dynamics`, all the PDE and ODE states that you specified in Step 1 appear.



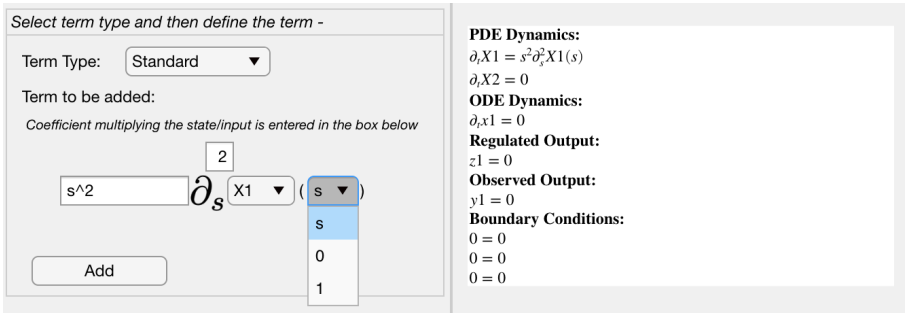
3. Once you select the desired state for which to add terms to the dynamics, go down to Step 2.a `Select term type and then define the term-`.



4. For a term involving an individual PDE state, you may have two kinds of terms, a `Standard` term, pre-multiplying the state with some coefficients, or a `Integral` term, taking some (partial) integral of the state. The `Standard` option can be used to define both terms involving states and terms involving inputs. On the other hand, the `Integral` option is only available for the PDE states.



- 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 a 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.

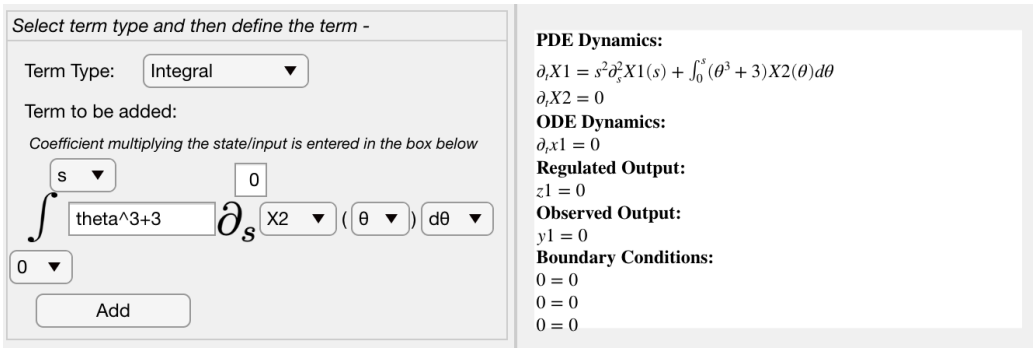


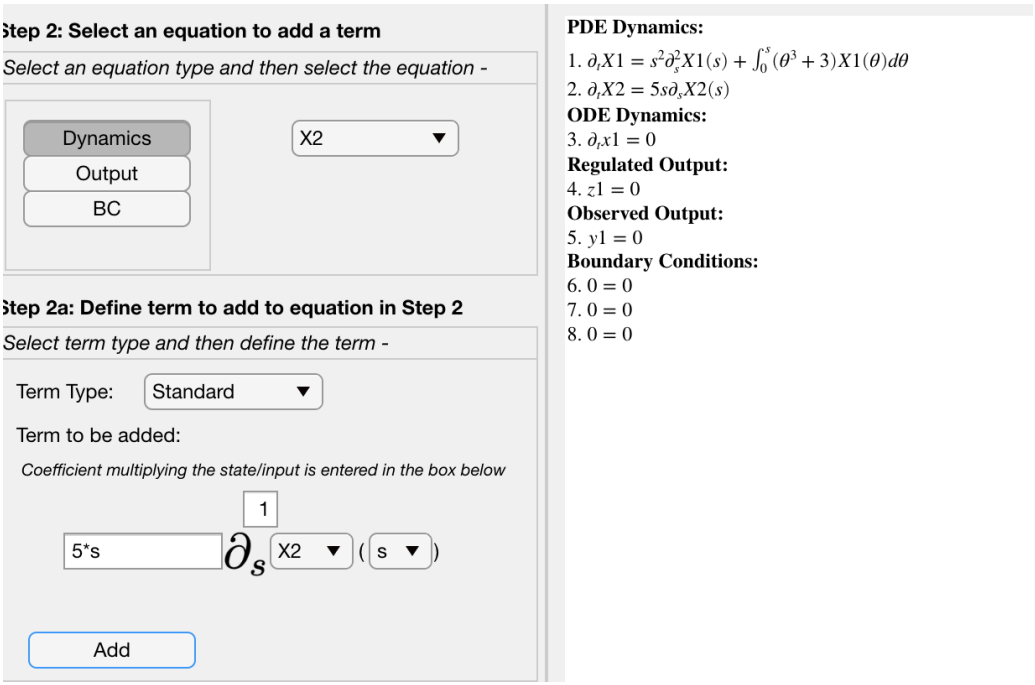
Note:

Only the PDE states can be a function of 's'. For other terms, the option of adding 's' as an independent variable is not available.

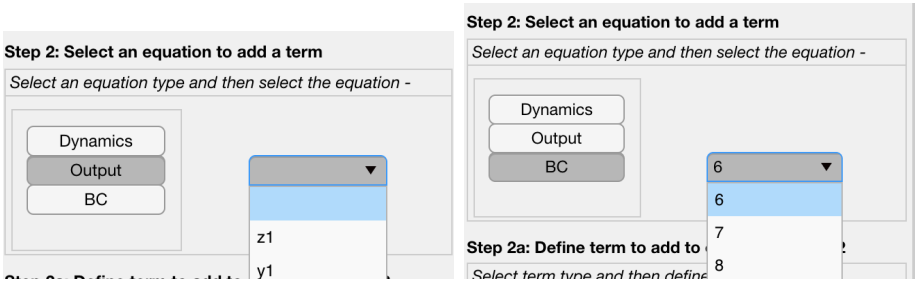
The order of derivative can not exceed the highest order derivative for that state. If the input value exceeds that, the following error will be displayed

- One can also add an integral term by selecting `Integral`. Here, identical to the `Standard` option, we can define the order or derivative, the coefficients 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.





5. 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 $'\theta'$.
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 desired 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. $y1 = X2(0)$

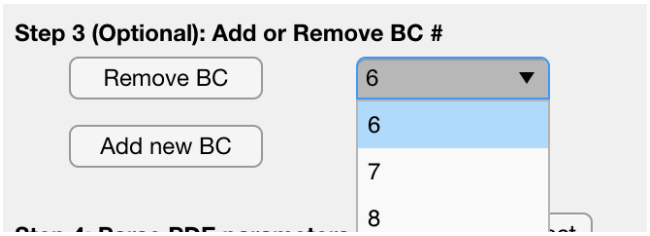
Boundary Conditions:

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

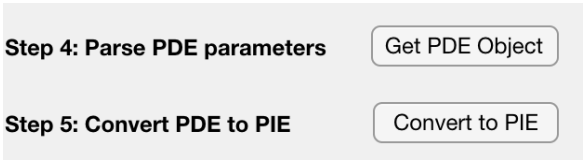
If desired, the user can also add a new boundary condition or remove one. Note that for a PDE state variable differentiable up to order N, a well-posed PDE must impose exactly N boundary conditions on this state variable.



8.1.4 Step 4-5: Parse PDE Parmeters and Convert Them to PIE

1. Once the desired model has been declared, you can extract all the parameters defining this model by pressing `Get PDE Object`, storing these parameters in an object called `PDE_GUI` which directly gets loaded into the MATLAB workspace.

2. In addition, pressing `convert to PIE`, you can convert your model to a PIE and store it in an object called `PIE_GUI` which directly gets loaded into the MATLAB workspace.



8.2 The Command Line Input Format

In PIETOOLS 2024, the easiest format for declaring ODE-PDE systems is using the Command Line Input format presented in Chapter 4. Using this format, `pde_struct` class objects can be declared using the `pde_var` function, and subsequently manipulated using e.g. algebraic operations to declare a broad class of linear ODE-PDEs. In this section, we show how the `pde_struct` object actually stores the information necessary to represent such systems, and how we can use this structure to define a broad class of linear systems. To illustrate, we will use the following example of a wave equation throughout this section.

Example

$$
\begin{aligned}
\partial_t \mathbf{x}_1(t,s_1,s_2) &= \mathbf{x}_2(t,s_1,s_2), && t\geq 0,\\
\partial_t \mathbf{x}_2(t,s_1,s_2) &= 5(\partial_{s_1}^2\mathbf{x}_1(t,s_1,s_2)+\partial_{s_2}^2\mathbf{x}_1(t,s_1,s_2))+(3-s_1)s_2u_1(t), && s_1\in[0,3],\\
\mathbf{y}_1(t,s_1) &= \mathbf{x}_1(t,s_1,1)+s_1w_1(t), && s_2\in[-1,1],\\
\mathbf{y}_2(t,s_2) &= \mathbf{x}_2(t,3,s_2)+w_2(t),\\
z(t) &= \begin{bmatrix} 10\int_0^3\int_{-1}^1 \mathbf{x}_1(t,s_1,s_2)ds_2ds_1 \\ \partial_{s_1}\partial_{s_2}\mathbf{x}_2(t,3,1)\end{bmatrix},\\
\mathbf{x}_1(t,s_1,-1) &= \partial_{s_2}\mathbf{x}_1(t,s_1,1)=0,\\
\mathbf{x}_1(t,0,s_2) &= \mathbf{u}_2(t-0.5,s_2), \qquad \partial_{s_1}\mathbf{x}_1(t,3,s_2)=0,\\
\mathbf{x}_2(t,s_1,-1) &= \partial_{s_2}\mathbf{x}_2(t,s_1,1)=0,\\
\mathbf{x}_2(t,0,s_2) &= \partial_{s_1}\mathbf{x}_1(t,3,s_2)=0.
\end{aligned}
\tag{8.1}
$$

8.2.1 Representing State, Input, and Output Variables

In Chapter 4, we showed that a state, input, or output variable can be declared in PIETOOLS 2024 using the function `pde_var`. A general call to this function takes the form

```
>> obj = pde_var(type,size,vars,dom,diff);
```

and may involve up to five inputs:

- `type`: A character array specifying the desired type of variable. Must be set to `'state'` to declare an ODE or PDE state variable, $x(t)$, `'input'` (or `'in'`) to declare an exogenous input variable, $w(t)$, `'control'` to declare an actuator input variable, $u(t)$, `'output'` (or `'out'`) to declare a regulated output variable, $z(t)$, and `'sense'` the declared a sensed output variable, $y(t)$. Defaults to `'state'`;
- `size`: An integer specifying the size of the object in case it is vector-valued. Defaults to `1`;
- `vars`: A $p \times 1$ array of type `'polynomial'`, specifying the spatial variables on which the object depends. Defaults to an empty array `[]`, indicating that the object does not vary in space;
- `dom`: A $p \times 2$ numeric array specifying for each of the spatial variables the interval on which it is defined, with the first column specifying the lower limit of the interval, and the second column the upper limit. Defaults to an array with the same number of rows as `vars`, with the first column all zeros, and the second column all ones, indicating that all spatial variables exist on $[0, 1]$;
- `diff`: A $p \times 1$ numeric array, specifying the order of differentiability of the object with respect to each of the spatial variables on which it depends. This can only be declared for PDE variables, i.e. `'state'` type objects, and is optional.

Note that, in general, the fifth input can be omitted, as PIETOOLS can usually infer the order of differentiability of PDE states from the declared PDE dynamics. In addition, most of the remaining inputs admit a default value that is used when no value is specified, **so long as the arguments are passed in the correct order**. For example, a scalar-valued PDE state $\mathbf{x}(t, s_1, s_2)$ on $s_1, s_2 \in [0, 1]$ can be declared as

```
>> x = pde_var('state',1,[s1;s2],[0,1;0,1]);
```

but can also be declared as e.g.

```
>> x = pde_var(1,[s1;s2]);
```

or even just

```
>> x = pde_var([s1;s2]);
```

However, we can simplify this call any further, as the spatial variables on which a PDE state depends **must always be specified**.

Calling `obj=pde_var(...)`, the output object `obj` is a `pde_struct` object representing the desired state, input, or output variable. To represent these variables, a `pde_struct` object `obj` has the following fields:

pde_struct

`obj.x`	a cell with each element i specifying a state component $\mathbf{x}_i$ in the system;
`obj.w`	a cell with each element i specifying an exogenous input $\mathbf{w}_i$;
`obj.u`	a cell with each element i specifying an actuator input $\mathbf{u}_i$;
`obj.z`	a cell with each element i specifying a regulated output $\mathbf{z}_i$;
`obj.y`	a cell with each element i specifying an observed output $\mathbf{y}_i$;
`obj.BC`	a cell with each element i specifying a boundary condition for the PDE.
`obj.free`	a cell with each element i specifying a free PDE variable or set of terms.

Depending on the specified `type` in the call to `pde_var`, one of these fields will be populated with a single element representing the desired state, input, or output variable. To this end, each of the cell elements of the fields will again be a structure, with the following fields:

`size`	an integer specifying the size of the state component, input or output;
`vars`	a $p \times 1$ 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 with the state component, output, or boundary condition;
`ID`	a unique integer value distinguishing the state, input, or output component from all others;
`diff`	a $1 \times p$ array specifying the order of differentiability of the state variable with respect to each spatial variable;

Naturally, the values of these fields will be populated with the inputs passed to `pde_var` (where the order of differentiability will be converted from column to row array). For example, our PDE state $\mathbf{x}(t, s_1, s_2)$ will be represented as an object `x` with all fields empty, except the field `x.x`, which will be a 1×1 cell structure with elements

```
>> x.x{1}
ans =

  struct with fields:

    size: 1
    vars: [2×1 polynomial]
     dom: [2×2 double]
      ID: 1
```

where `x.x{1}.vars=[s1;s2]`, and `x.x{1}.dom=[0,1;0,1]`. In addition, a single free term representing the state variable will be stored in the field `x.free{1}.term{1}`. We will show how exactly such terms are represented in the structure in the next subsection.

Example

Consider the wave equation example from (8.1). The system is represented as a 2D PDE, in spatial variables $(s_1, s_2) \in [0,3] \times [-1,1]$. We initialize these variables as

```
>> pvar s1 s2
```

Now, the system is defined in terms of two PDE state variables, $\mathbf{x}_1(t), \mathbf{x}_2(t) \in L_2[[0,3] \times [-1,1]$, two actuator inputs, $u_1(t) \in \mathbb{R}$ and $\mathbf{u}_2(t) \in L_2[-1,1]$, two exogenous inputs, $w_1(t), w_2(t) \in \mathbb{R}$, two sensed outputs, $\mathbf{y}_1(t) \in L_2[0,3]$ and $\mathbf{y}_2(t) \in L_2[-1,1]$, and a vector-valued regulated output $z(t) \in \mathbb{R}^2$. We can declare these variables as

```
>> x1 = pde_var('state',1,[s1;s2],[0,3;-1,1],[2;2]);
>> x2 = pde_var('state',1,[s1;s2],[0,3;-1,1],[2;2]);
>> u1 = pde_var('control',1,[],[]);
>> u2 = pde_var('control',1,s2,[-1,1]);
>> w1 = pde_var('input',1,[],[]);
>> w2 = pde_var('input',1,[],[]);
>> y1 = pde_var('sense',1,s1,[0,3]);
>> y2 = pde_var('sense',1,s2,[-1,1]);
>> z = pde_var('output',2,[],[]);
```

or, equivalently, as

```
>> x1 = pde_var([s1;s2],[0,3;-1,1]);
>> x2 = pde_var([s1;s2],[0,3;-1,1],[2;2]);
>> u1 = pde_var('control');
>> u2 = pde_var('control',s2,[-1,1]);
>> w1 = pde_var('in');
>> w2 = pde_var('in');
>> y1 = pde_var('sense',s1,[0,3]);
>> y2 = pde_var('sense',s2,[-1,1]);
>> z = pde_var('out',2);
```

Here, we do not specify the order of differentiability of state variable $\mathbf{x}_1(t)$ with respect to the spatial variables. This is because, in the PDE (8.1), a second-order derivative of $\mathbf{x}_1$ is taken with respect to both variables in the dynamics for $\mathbf{x}_2(t)$, from which PIETOOLS will be able to infer the order of differentiability automatically. However, no second-order spatial derivative is taken of the state variable $\mathbf{x}_2(t)$, so we manually specify that it is second-order differentiable with respect to both spatial variables in the call to `pde_var`. Failing to specify this may result in issues when converting the system to a PIE.

Displaying e.g. the output variable `y2`, we get something like

```
>> y2
      y8(t,s2);
```

Here, the index 8 corresponds to the ID assigned to the output: `y2.y{1}.ID=8`. This ID is uniquely generated by the function `stateNameGenerator`, and is vital for PIETOOLS to distinguish between the different PDE objects. However, since the value of the IDs tends to increase quite quickly, the user may consider calling `clear stateNameGenerator` whenever declaring a new PDE, to reset the counter for the IDs.

8.2.2 Declaring Terms

After declaring a PDE variable using `pde_var`, we can perform a variety of operations on this variable, including multiplying it with desired coefficients, performing differentiation or integration, and evaluating it at a particular position. The resulting term involving the PDE variable is stored in the field `free`, which is a cell structure with each element `free{i}` specifying a separate string of free terms to be used to declare equations. In particular, each element `free{i}` has a field `term`, which is again a cell structure with each element `free{i}.term{j}` representing a single term 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,
or `term{j}.y;`	**or** sensed output,
or `term{j}.z;`	**or** regulated output 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) factor with which to multiply the state or input;
`term{j}.delay`	scalar integer specifying the temporal delay in the state or input;

where p denotes the number of spatial variables on which the component depends. Combined, these fields can represent a general term involving e.g. a state component $\mathbf{x}_k(t,s)$ or input $\mathbf{u}_k(t,s)$ of the form

$$\underbrace{\int_{L_1}^{U_1}}_{\texttt{I}} \left(\underbrace{C(s,\theta)}_{\texttt{C}} \overbrace{\partial_\theta^d}^{\texttt{D}} \underbrace{\mathbf{x}_k}_{\texttt{x}}(t - \overbrace{\tau}^{\texttt{delay}}, \overbrace{\theta}^{\texttt{loc}}) \right) d\theta \qquad \text{or} \qquad \underbrace{\int_{L}^{U}}_{\texttt{I}} \left(\underbrace{C(s,\theta)}_{\texttt{C}} \underbrace{\mathbf{u}_k}_{\texttt{u}}(t - \overbrace{\tau}^{\texttt{delay}}, \overbrace{\theta}^{\texttt{loc}}) \right) d\theta.$$

Declaring e.g. a PDE state `x2=pde_var([s1;s2],[0,3;-1,1],[2;2]` as in the previous subsection, this state will be immediately assigned a single term in the field `x2.free{1}.term{1}`, representing just the variable itself as

```
>> x2.free{1}.term{1}
ans =

  struct with fields:

      x: 1
      C: 1
    loc: [1×2 polynomial]
      D: [0 0]
      I: {2×1 cell}
```

Here, `x2.free{1}.term{1}.loc=[s1,s2]` and `x2.free1.term1.I={[];[]}`, so that the state variable $\mathbf{x}_2$ is multiplied with 1, evaluated at $(s_1, s_2) = (s_1, s_2)$, and not differentiated or integrated. Note that the index `x2.free{1}.term{1}.x` is `1`, which is not the same as the ID `x2.x{1}.ID=2`, but rather matches the element i of the field `x2.x{i}` in which the information on the state variable is stored – no need to worry about this though. Although all the fields in the structure could be adjusted manually to specify a desired term involving $\mathbf{x}_2$, the `pde_struct` class comes with a variety of overloaded functions for declaring these values much more easily.

8.2.2a Multiplication

Given a `pde_struct` object representing some term, we can pre-multiply it with a desired factor using the standard multiplication operation `*`. The result is the same term, but now with coefficients `term{j}.C` set to the specified factor. For example, to represent $5\mathbf{x}_1(t, s_1, s_2)$, we simply call

```
>> trm1 = 5*x1

 5 * x1(t,s1,s2);
```

The result will again be a single term, stored in the field `trm1.free{1}.term{1}`, where now the coefficients are set to 5:

```
>> trm1.free{1}.term{1}.C
ans =
      5
```

Similarly, we can also pre-multiply input variables with desired coefficients, and these coefficients can also be polynomial. For example, to represent the term $(3 - s_1)s_2u_1(t)$, we call

```
>> trm2 = (3-s1)*s2*u1

       C11(s1,s2) * u3(t);
```

Again, the result is only a single term, in this case represented as

```
>> trm2.free{1}.term{1}
ans =

  struct with fields:

    u: 1
    C: [1×1 polynomial]
    I: {0×1 cell}
```

where now `trm2.free{1}.term{1}=-s1*s2+3*s2`. We can also check the value of the coefficients defining this term by calling the field `C` directly, with row and column index as displayed,

```
>> trm2.C{1,1}
ans =
   -s1*s2 + 3*s2
```

Note that `pde_struct` objects can only represent linear systems, and thus multiplication of state, input, or output variables with one another is not supported. Also, declaring output equations wherein the output is multiplied with a factor is not supported.

8.2.2b Differentiation

Naturally, any PDE will involve partial derivatives of a state variable with respect to spatial variables. This operation can be performed in the Command Line Input format using the function `diff` as

```
>> trm_new = diff(trm_old,vars,order);
```

This function takes three arguments:

- `trm_old`: A `pde_struct` object representing some term of which to take a derivative;
- `vars`: A $p \times 1$ array of type `'polynomial'`, specifying the spatial variables with respect to which to take the derivative;
- `order`: A $p \times 1$ array of integers specifying for each of the variables the desired order of the derivative of the term with respect to that variable. Defaults to 1;

The output `trm_new` is then another `pde_struct` object, representing the derivative of the term defined by `trm_old` with respect to each of the variables in `vars`, up to the order specified in `order`. Specifically, the value of the field `D` in the term will be set to the specified order. For example, to represent the term $5\partial_{s_1}^2 \mathbf{x}_1(t, s_1, s_2)$, we can take the derivative of $5\mathbf{x}_1(t, s_1, s_2)$ as

```
>> trm3 = diff(trm1,s1,2)

    5 * (d/ds1)^2 x1(t,s1,s2);
```

The result again represents a single term, stored as

```
>> trm3.free{1}.term{1}
ans =

  struct with fields:

      x: 1
      C: [1×1 polynomial]
    loc: [1×2 polynomial]
      D: [2 0]
      I: {2×1 cell}
```

Here, the field `D` is set to `[2 0]` to indicate that a second-order derivative is taken of the state with respect to its first variable. Similarly, we can take the derivative $\partial_{s_1}\partial_{s_2}\mathbf{x}_2(t, s_1, s_2)$ as

```
>> trm4 = diff(x2,[s1;s2])

  (d/ds1)(d/ds2) x2(t,s1,s2);
```

with the derivative now stored as `trm4.free{1}.term{1}.D=[1,1]`.

Note, however, that it is not possible to take derivatives of input or output signals. In addition, keep in mind that the order of multiplication and differentiation matters, so that e.g. `s1*diff(x2,s1)` yields a different result than `diff(s1*x2,s1)`.

8.2.2c Substitution

Aside from taking derivatives of PDE states, we can also evaluate these PDE states at the boundary of the domain, using the function `subs` as

```
>> trm_new = subs(trm_old,vars,loc);
```

This function takes three arguments:

- `trm_old`: A `pde_struct` object representing some term which to evaluate at some boundary position;
- `vars`: A $p \times 1$ array of type `'polynomial'`, specifying the spatial variables to be substituted for a position;
- `loc`: A $p \times 1$ numeric array specifying the value for which each of the variables are to be substituted. Each value must correspond to either the lower or upper limit of the interval on which the corresponding variable is defined;

The function returns a `pde_struct` object `trm_new`, representing the same term as `trm_old` but now evaluated with respect to each of the variables in `vars` at the position specified in `loc`. In particular, the value of the field `term{j}.loc` will be set to the specified value. For example, to evaluate our derivative $\partial_{s_1}\partial_{s_2}\mathbf{x}_2(t, s_1, s_2)$ (represented by `trm4`) at $(s_1, s_2) = (3, 1)$, we call

```
>> trm5 = subs(trm4,[s1;s2],[3;1])

    (d/ds1)(d/ds2) x2(t,3,1);
```

where now

```
>> trm5.free{1}.term{1}.loc
ans =

 [ 3, 1]
```

Similarly, we can declare the term $\mathbf{x}_2(t, s_1, -1)$ as

```
>> trm6 = subs(x2,s2,-1)

   x2(t,s1,-1);
```

where now

```
>> trm6.free{1}.term{1}.loc
ans =

  [ s1, -1]
```

indicating that the term is evaluated as $(s_1, s_2) = (s_1, -1)$.

Note that only state variables can be evaluated at a boundary – substitution of input or output variables is not supported. In addition, note that the order of substitution and multiplication is important, so that e.g. `s1*subs(x2,s1,0)` yields a different result than `subs(s1*x2,s1,0)`.

8.2.2d Integration

Regulated outputs for PDE systems frequently involve an integral of the PDE state. Such an integral for `pde_struct` objects can be declared with the function `int` as

```
>> trm_new = int(trm_old,vars,dom);
```

This function too takes three arguments:

- `trm_old`: A `pde_struct` object representing some term which to integrate;
- `vars`: A $p \times 1$ array of type `'polynomial'`, specifying the spatial variables with respect to which to integrate;
- `loc`: A $p \times 2$ numeric or `'polynomial'` array specifying the domain over which integration is to be performed for each variable, with the first column specifying the lower limit and the second column the upper limit of the integral. Lower and upper limits must either correspond to the boundaries of the domain on which the variable is defined, or, for the purpose of declaring a partial integral, the spatial variable itself;

The returned `trm_new` is again a `pde_struct` object representing the same term as `trm_old`, but now integrated over the desired domain with respect to the desired variable. Specifically, the limits of the integral with respect to each variable will be stored in the field `term{j}.I`. For example, to declare the integral $\int_0^3 \int_{-1}^1 \mathbf{x}_1(t,s_1,s_2)ds_2ds_1$, we call

```
>> trm7 = 10*int(x1,[s2;s1],[-1,1;0,3])

   int_0^3 int_-1^1 [10 * x1(t,s1,s2)] ds2 ds1;
```

where now

```
>> trm7.free{1}.term{1}.I{1}
ans =
  [ 0, 3]

>> trm7.free{1}.term{1}.I{2}
ans =
  [ -1, 1]
```

If desired, it is also possible to declare partial integrals, for which substitution must be performed as well as integration. For example, to declare an integral $\int_0^{s_1}(s_1-\theta)\mathbf{x}_2(t,\theta,s_2)d\theta$, we can call

```
>> pvar s1_dum
>> trm_alt = int((s1-s1_dum)*subs(x2,s1,s1_dum),s1_dum,[0,s1])

    int_0^s1 [C11(s1,s1_dum) * x2(t,s1_dum,s2)] ds1_dum;
```

Here, we first need to introduce a dummy variable for integration, and substitute s_1 for this dummy variable, before taking the integral. Although not strictly necessary, we highly recommend to always give the dummy variable the same name as its associated primary spatial variable, but with `_dum` added, as this is the default used by PIETOOLS.

Note that, unlike differentiation and substitution, integration **is supported** for input variables, **but not** for output variables.

8.2.2e Delay

Although we recommend using the DDE or DDF format for declaring systems with delay, `pde_struct` objects do also allow signals with temporal delay to be declared. This can be done using the `subs` function, by substituting the temporal variable t for $t-\tau$, for some desired value τ. For example, to declare the term $\mathbf{u}(t-0.5, s_2)$, we call

```
>> pvar t
>> trm8 = subs(u2,t,t-0.5)

    u4(t-0.5,s2);
```

Here, we must first declare the temporal variable `t`, and then we can substitute `t` for `t-0.5`. Note that the variable `t` **will always be interpreted as temporal variable** in PIETOOLS, and therefore should not be used for any other purpose. The value of the delay will be stored in the aptly named field `delay`, so that e.g.

```
>> trm8.free{1}.term{1}
ans =

  struct with fields:

       u: 1
       C: 1
       I: {[]}
     loc: [1×1 polynomial]
   delay: 0.5000
```

Delay can be added to state variables or exogenous inputs in a similar manner, but is not supported for output signals.

8.2.2f Addition

Having seen how we can perform a variety of operations on PDE variables to declare a single term, naturally, we will want to add these terms to create an equation. This can be readily done with the overloaded function for addition, `+`. In particular, given two `pde_struct` objects, `trm_1` and `trm_2`, each specifying their own terms, we can compute the sum of these terms as `trm_3=trm_1+trm_2`, where now `trm_3.free{i}.term` includes all elements from `trm_1.free{i}.term` as well as `trm_2.free{i}.term`. For example, having declared the term $(3-s_1)s_2u_1(t)$ as `trm2`, and $5\partial_{s_1}^2\mathbf{x}_1(t,s_1,s_2)$ as `trm3`, we can take their sum as

```
>> trms9 = trm2+trm3

    C11(s1,s2) * u3(t) + 5 * (d/ds1)^2 x1(t,s1,s2);
```

where now `trm9.free{1}.term` is a 1×2 cell array, with each element representing a separate term, combining the terms from `trm2` and `trm3`. Note that we can determine the coefficients appearing in this sum of terms by calling field `C{1,j}`, where `j` is the desired term number, so that e.g.

```
>> trms9.C{1,1}
ans =
  -s1*s2 + 3*s2
```

and

```
>> trms9.C{1,2}
ans =
  5
```

Naturally, we can also add more terms to the structure, adding e.g. $5\partial_{s_2}^2\mathbf{x}_1(t,s_1,s_2)$ as

```
>> trms10 = 5*diff(x1,s2,2) + trms9

    5 * (d/ds2)^2 x1(t,s1,s2) + C12(s1,s2) * u3(t) + 5 * (d/ds1)^2 x1(t,s1,s2);
```

Note that the coefficients in `trms10` have been shifted compared to `trms9`, so that now e.g. `trms10.C{1,2}=trms9.C{1,1}=-s1*s2+3*s2`.

8.2.2g Concatenation

Finally, `pde_struct` objects can also be concatenated vertically, using the standard MATLAB function `[ ; ]`, to combine separate sums of terms into a single structure. For example, we can concatenate the variable $\mathbf{x}_2(t,s_1,s_2)$, declared as `x2`, with the terms $5\partial_{s_1}^2\mathbf{x}_2(t,s_1,s_2) + 5\partial_{s_2}^2\mathbf{x}_2(t,s_1,s_2) = (3-s_1)s_2u_1(t)$, declared as `trms10` as

```
>> RHS_x = [x2;trms10]

   x2(t,s1,s2);
   5 * (d/ds2)^2 x1(t,s1,s2) + C12(s1,s2) * u3(t) + 5 * (d/ds1)^2 x1(t,s1,s2);
```

Here, each row of terms in `RHS1` is stored in a separate element of the cell `RHS1.free`, so that `RHS1.free{1}.term` only has a single element representing the variable $\mathbf{x}_2(t,s_1,s_2)$, and `RHS1.free{2}.term` has three elements, representing the three terms $5\partial_{s_1}^2\mathbf{x}_2(t,s_1,s_2)$, $(3-s_1)s_2u_1(t)$, and $5\partial_{s_2}^2\mathbf{x}_2(t,s_1,s_2)$. Similarly, we can concatenate the corner value $\partial_{s_1}\partial_{s_2}\mathbf{x}_2(t,3,1)$, represented by `trm5`, and the integral $10\int_0^3\int_{-1}^1\mathbf{x}_1(t,s_1,s_2)ds_2ds_1$, represented by `trm7`, as

```
>> RHS_z = [trm7; tmr5]

    int_0^3 int_-1^1 [10 * x1(t,s1,s2)] ds2 ds1;
    (d/ds1)(d/ds2) x2(t,3,-1);
```

8.2.3 Declaring Equations

Having seen how terms for PDEs can be declared and stored using `pde_struct` objects, all that remains is to use these terms to declare actual equations. Here, we distinguish three types of equations, namely state equations (ODEs and PDEs), output equations, and boundary conditions, all of which can be declared using the function `==`.

8.2.3a State equations

In PIETOOLS, all ordinary and partial differential equations are assumed to involve some temporal variable, distinct from all other (spatial) variables in the dynamics. That is, any PDE is expected to model the evolution of some state $\mathbf{x}(t) \in L_2[\Omega]$ for $t \geq 0$, governed by an equation

$$\partial_t \mathbf{x}(t,s) = f(\mathbf{x}(t,s), \mathbf{u}(t,s), \mathbf{w}(t,s), s), \qquad s \in \Omega$$

where f is some (polynomial) function of $\mathbf{x}(t,s)$ and its partial derivatives with respect to s, as well as any input signals $\mathbf{u}(t,s)$ and $\mathbf{w}(t,s)$. In the previous subsection, we have seen how we can declare such a function f in the `pde_struct` format using multiplication, addition, differentiation, etc.. Now, to declare the left-hand side of such an equation, $\partial_t \mathbf{x}(t,s)$, we can use the function `diff` just as we did to declare spatial derivatives. For example, to declare the derivative $\partial_t \mathbf{x}_1(t,s_1,s_2)$, we call

```
>> pvar t
>> LHS1 = diff(x1,t);

        d_t x1(t,s1,s2);
```

Here, we first need to declare the temporal variable as `pvar t`, noting that PIETOOLS will always interpret this object as temporal variable. Alternatively, we can also use `'t'` to represent the temporal variable in this case, declaring e.g. $\partial_t \mathbf{x}_2(t,s_1,s_2)$ as

```
>> LHS2 = diff(x2,'t');

        d_t x2(t,s1,s2);
```

The result is again a single term, stored in `LHS2.free{1}.term{1}`, where now the order of the temporal derivative is represented through the field `tdiff`:

```
>> LHS2.free{1}.term{1}.tdiff
ans =

     1
```

indicating that a first-order temporal derivative is taken of the state variable. Although it is also possible to declare higher-order temporal derivatives – calling e.g. `diff(x2,'t',2)` to declare a second-order temporal derivative of $\mathbf{x}_2$ – keep in mind that PIETOOLS will always convert the system to a format involving only first-order temporal derivatives when constructing the PIE representation. In doing so, PIETOOLS will add state components without adding boundary conditions, so that the resulting representation may not be entirely equivalent, and results of e.g. stability analysis may be conservative – see also Subsection 8.2.4f.

Given a temporal derivative of a state variable, an equation defining the dynamics of this state can be easily declared using the function `==`, equating this derivative to some desired set of terms. For example, to declare the equation $\partial_t \mathbf{x}_1(t,s_1,s_2) = \mathbf{x}_2(t,s_1,s_2)$, we simply call

```
>> x_eq1 = diff(x1,t)==x2

 d_t x1(t,s1,s2) = x2(t,s1,s2);
```

Similarly, having already declared the terms $5\partial_{s_1}^2 \mathbf{x}_2(t,s_1,s_2) + (3-s_1)s_2 u_1(t) + 5\partial_{s_2}^2 \mathbf{x}_2(t,s_1,s_2)$ through `trms10`, we can declare the dynamics for $\mathbf{x}_2$ as

```
>> x_eq2 = diff(x2,'t')==trms10

d_t x2(t,s1,s2) = 5*(d/ds2)^2x1(t,s1,s2)+C12(s1,s2)*u3(t)+5*(d/ds1)^2x1(t,s1,s2);
```

In doing so, the result `x_eq2` will store all terms from `trms10.free{1}.term` in a corresponding element of `x_eq2.x`. In particular, since `x_eq2` defines only a single state equation, the terms will be stored in `x_eq2.x{1}`, so that `x_eq2.x{1}.term=trms10.free{1}.term`. However, we can also concatenate the declared equations as

```
>> x_eqs = [x_eq1; x_eq2]

 d_t x1(t,s1,s2) = x2(t,s1,s2);
 d_t x2(t,s1,s2) = 5*(d/ds2)^2x1(t,s1,s2)+C22(s1,s2)*u3(t)+5*(d/ds1)^2x1(t,s1,s2);
```

In this output, the two state equations are stored in separate elements of `x_eqs.x`, so that in particular `x_eqs.x{2}.term=x_eq2.x{1}.term=trms10.free{1}.term`.

Note that, when concatenating equations, the order of the equations may not reflect the order in which they are specified. For the purposes of e.g. simulation and analysis, the order of the state components will be determined by the order in which their governing equations appear, so it is important to always check the final order of the state (as well as input and output) variables once the full system of equations has been declared.

8.2.3b Output equations

Aside from differential equations, we can also declare output equations, using the same function `==`. For example, to declare the observed output equations $\mathbf{y}_1(t, s_1) = \mathbf{x}_1(t, s_1) + s_1 w_1(t)$ and $\mathbf{y}_2(t, s_2) = \mathbf{x}_2(t, s_2) + w_2(t)$, we can call

```
>> y_eqs = [y1==subs(x1,s2,1)+s1*w1;
            y2==subs(x2,s1,3)+w2]

 y7(t,s1) = x1(t,s1,1) + C32(s1) * w5(t);
 y8(t,s2) = x2(t,3,s2) + w6(t);
```

In doing so, the output object `y_eqs` will have the two declared equations stored in `y_eqs.y{1}` and `y_eqs.y{2}`, so that e.g. `y_eqs.y{1}.term{2}` represents the term $s_1 w_1$ as

```
>> y_eqs.y{1}.term{2}
ans =

  struct with fields:

    w: 1
    C: [1×1 polynomial]
    I: {0×1 cell}
```

where `y_eqs.y{1}.term{2}.C=s1`. Similarly, having declared $\begin{bmatrix} 10 \int_0^3 \int_{-1}^1 \mathbf{x}_1(t, s_1, s_2) ds_2 ds_1 \\ \partial_{s_1} \partial_{s_2} \mathbf{x}_2(t, 3, 1) \end{bmatrix}$ as `RHS_z`, we can set the regulated output equation as

```
>> z_eqs = z==RHS_z

z9(t) = int_0^3 int_-1^1[C31*x1(t,s1,s2)]ds2 ds1 + C32*(d/ds1)(d/ds2) x2(t,3,1);
```

Note here that, although the output $z(t)$ is vector-valued, we declared it as only a single variable `z`. As such, it will also be assigned only one equation, where now the coefficients `C31` and `C32` are arrays mapping the scalar-valued terms to the vector-valued outputs. In particular,

```
>> z_eqs.C{3,1}
ans =
  [ 10]
  [  0]
```

and

```
>> z_eqs.C{3,2}
ans =
     0
     1
```

More generally, the full equation for z will be stored in the field `z_eqs.z{1}`. Thus, although concatenation of vector-valued objects is supported, keep in mind that only one equation will be assigned for each declared variable. Note also that, naturally, equations can only be set if the number of elements on the left-hand side and right-hand side match.

8.2.3c Boundary conditions

Any well-posed PDE also involves a number of boundary conditions on the state. These too can be declared as `pde_struct` objects using the function `==`, by setting different terms equal to one another, or setting a term equal to zero. For example, to declare the boundary condition $\mathbf{x}_2(t, s_1, -1) = 0$, we can call

```
>> BC1 = subs(x2,s2,-1)==0

 0 = x2(t,s1,-1);
```

Here, the zero will always be displayed on the left-hand side of the equation. The actual term $\mathbf{x}_2(t, s_1, -1)$ that is set equal to zero will be stored in the field `BC1.BC{1}.term`. Similarly, we can declare the condition $\mathbf{x}_1(t, 0, s_2) = \mathbf{u}_2(t - 0.5, s_2)$ as

```
>> BC2 = subs(x1,s1,0)==subs(u2,t,t-0.5)

 0 = x1(t,0,s2) - u2(t-0.5,s2);
```

Again, the equation is represented in the form $0 = F(\mathbf{x}, \mathbf{u}, \mathbf{w}, s)$, where the function F is stored in the field `BC`. In this case, the field `BC2.BC{1}.term` will have two elements, representing the terms $\mathbf{x}_1(t, 0, s_2)$ and $-\mathbf{u}_2(t - 0.5, s_2)$.

Note that any equation that does not involve a temporal derivative or an output will be interpreted as a boundary condition, and added to the field `BC`.

Example

Consider the wave equation example from (8.1). Having declared the different state, input, and output variables, the full PDE system can be declared as

```
>> pvar t
>> PDE = [diff(x1,t)==x2;
          diff(x2,t)==5*(diff(x1,s1,2)+diff(x1,s2,2))+(3-s1)*s2*u1;
          y1==subs(x1,s2,1)+s1*w1;
          y2==subs(x2,s1,3)+w2;
          z ==[10*int(x1,[s1;s2],[0,3;-1,1]);
               subs(diff(x2,[s1;s2]),[s1;s2],[3;1])];
          subs(x1,s1,0)==subs(u2,t,t-0.5);
          subs(x2,s1,0)==0;
          subs([x1;x2],s2,-1)==0;
          subs(diff([x1;x2],s1),s1,3)==0;
          subs(diff([x1;x2],s2),s2,1)==0]

 dt x1(t,s1,s2) = x2(t,s1,s2);
 dt x2(t,s1,s2) = 5 * (d/ds1)^2 x1(t,s1,s2) + 5 * (d/ds2)^2 x1(t,s1,s2)
                            + C23(s1,s2) * u3(t);
 y7(t,s1) = x1(t,s1,1) + C32(s1) * w5(t);
 y8(t,s2) = x2(t,3,s2) + w6(t);
 z9(t) = int_0^3 int_-1^1 [C51 * x1(t,s1,s2)]ds2 ds1
                            + C52 * (d/ds1)(d/ds2) x2(t,3,1);
 0 = x1(t,0,s2) - u4(t-0.5,s2);
 0 = x2(t,0,s2);
 0 = x1(t,s1,-1);
 0 = x2(t,s1,-1);
 0 = (d/ds1) x1(t,3,s2);
 0 = (d/ds1) x2(t,3,s2);
 0 = (d/ds2) x1(t,s1,1);
 0 = (d/ds2) x2(t,s1,1);
```

8.2.4 Post-Processing of PDE Structures

After you have declared a full system of equations and boundary conditions as a `pde_struct` object, you are generally ready to convert the system to a PIE with the function `convert`, for the purpose of e.g. stability analysis or estimator or controller synthesis. However, before converting the system to a PIE, PIETOOLS will first express the system in a particular manner, e.g. re-ordering the state, input and output variables, accounting for any temporal delays, and expanding any higher-order temporal derivatives in a manner that involves only first-order derivatives. Although all of this is done automatically when calling `convert`, and the user will generally be informed of these changes in the Command Window, it is important that the user be aware of what exactly is happening. Moreover, in some cases, it may be useful for the user to perform these operations themselves. In this subsection, therefore, we list several functions for post-processing of completed PDE structures that PIETOOLS generally runs when converting the system to a PIE, and that the user may also benefit from themselves.

8.2.4a Declaring controlled inputs and observed outputs

Although we highly recommend distinguishing between actuator inputs and disturbances when first declaring the PDE variables, it is possible to convert disturbances to controlled inputs after the PDE has been declared as well, using the function `setControl` as

```
>> PDE_new = setControl(PDE_old,w);
```

This function takes as input a PDE system `PDE` declared as `pde_struct`, and a desired exogenous input variable `w`, and returns a structure `PDE_new` representing the same as the input, but now with the variable `w` converted to an actuator input `u`. For example to convert the disturbance w_1 in our output equations defined by `y_eqs` to a controlled input, we call

```
>> y_eqs_u = setControl(y_eqs,w1)
1 inputs were designated as controlled inputs


 y7(t,s1) = x1(t,s1,1) + C32(s1) * u5(t);
 y8(t,s2) = x2(t,3,s2) + w6(t);
```

In the resulting system, the disturbance w_1 has been converted to a controlled input $u_3(t)$, and the equation $y_1(t,s_1) = \mathbf{x}_1(t,s_1,1) + s_1 w_1(t)$ has been updated accordingly to $y_1(t,s_1) = \mathbf{x}_1(t,s_1,1) + s_1 u_3(t)$ (although different subscripts are used in the display). Similarly, we can also convert regulated outputs to observed outputs using the function `setObserve` as

```
>> PDE_new = setObserve(PDE_old,z);
```

Rather than a disturbance, this function takes a regulated output variable as second argument, and converts this output to an observed output in the specified structure. For example, we can convert the regulated output $z(t)$, represented by `z_eqs`, to an observed output by calling

```
>> z_eqs_y = setObserve(z_eqs,z)
1 outputs were designated as observed outputs


 y9(t) = int_0^3 int_-1^1[C31*x1(t,s1,s2)]ds2 ds1 + C32*(d/ds1)(d/ds2) x2(t,3,1);
```

Keep in mind that the functions `setControl` and `setObserve` should only be called once the full PDE system has been declared, to avoid e.g. an input signal appearing as both an exogenous input and controlled input in the system.

8.2.4b Initializing a PDE structure

After declaring any PDE structure `PDE`, the user is highly recommended to initialize the structure by calling `PDE=initialize(PDE)`. This function checks that the PDE has been properly specified, throwing an error if any terms have not been properly declared, and warning the user of e.g. missing equations or insufficient boundary conditions. As such, it is important that the user **initializes the system only once all equations have been declared**. The function then displays a summary of the encountered state variables, inputs, outputs, and boundary conditions, and returns a cleaner display of the system, so that the user can check whether PIETOOLS has properly interpreted the system. In this summary and the display, the different variables are also assigned new indices separate from their IDs, instead matching the order in which PIETOOLS will consider them. However, these variables may be reordered again when converting to a PIE, using the function `reorder_comps` presented in the next subsection.

Example

Consider again the wave equation example from (8.1), declared as a `pde_struct` object `PDE` in the previous subsection. We initialize the function as follows:

```
>> PDE = initialize(PDE);

Encountered 2 state components:
 x1(t,s1,s2), of size 1, differentiable up to order (2,2) in variables (s1,s2);
 x2(t,s1,s2), of size 1, differentiable up to order (2,2) in variables (s1,s2);

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

Encountered 2 exogenous inputs:
 w1(t),    of size 1;
 w2(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 2;

Encountered 8 boundary conditions:
 F1(t,s2) = 0,   of size 1;
 F2(t,s2) = 0,   of size 1;
 F3(t,s1) = 0,   of size 1;
 F4(t,s1) = 0,   of size 1;
 F5(t,s2) = 0,   of size 1;
 F6(t,s2) = 0,   of size 1;
 F7(t,s1) = 0,   of size 1;
 F8(t,s1) = 0,   of size 1;
```

Note that the inputs and outputs have been re-indexed as `u1` and `u2`, `w1` and `w2`, and `y1` and `y2`.

8.2.4c Reordering components

In order to convert any PDE to a PIE, the state variables, inputs, and outputs have to be reordered to support representation in terms PI operators (or more accurately, in terms the `opvar` and `opvar2d` representing these operators numerically). This ordering is done primarily based on the dimension of the spatial domain on which the state, input, or output variable is defined, starting with finite-dimensional (ODE states) variables, followed by variables on a 1D domain (1D PDE states), followed by variables on a 2D domain (2D PDE states), etc. This reordering is done using the function `reorder_comps`, taking a PDE structure and returning an equivalent representation wherein the different variables have been re-indexed to match the order necessary for representation as a PIE.

Example

Consider again the wave equation example from (8.1), declared as an initialized `pde_struct` object `PDE_i`. Calling `reorder_comps`, we get the following message

```
>> PDE_r = reorder_comps(PDE_i)

The order of the state components x has not changed.
The order of the exogenous inputs w has not changed.
The order of the actuator inputs u has not changed.
The order of the regulated outputs z has not changed.
The order of the observed outputs y has not changed.
The boundary conditions have been re-indexed as:
   BC3(t,s1)   -->   BC1(t,s1)
   BC4(t,s1)   -->   BC2(t,s1)
   BC7(t,s1)   -->   BC3(t,s1)
   BC8(t,s1)   -->   BC4(t,s1)
   BC1(t,s2)   -->   BC5(t,s2)
   BC2(t,s2)   -->   BC6(t,s2)
   BC5(t,s2)   -->   BC7(t,s2)
   BC6(t,s2)   -->   BC8(t,s2)
```

Since the state components, inputs, and outputs have already been ordered correctly when declaring the system, they are not reordered here. However, the boundary conditions have been reordered, to first give the boundary conditions depending on the first spatial variable, `s1`, follows by the boundary conditions defined in terms of the second spatial variable, `s2`. Now, suppose we introduce an ODE state component $x_3(t)$, with dynamics $\dot{x}_3(t) = -x_3(t)$, added to the PDE structure as

```
>> x3 = pde_var();
PDE_2 = initialize([PDE;diff(x3,'t')==-x3]);

Encountered 3 state components:
x1(t,s1,s2), of size 1, differentiable up to order (2,2) in variables (s1,s2);
x2(t,s1,s2), of size 1, differentiable up to order (2,2) in variables (s1,s2);
x3(t),       of size 1, finite-dimensional;
```

In this system, the finite-dimensional state variable appears last, whereas it must appear first in the PIE representation. As such, calling now the function `reorder_comps`, we get a message

```
>> PDE_r2 = reorder_comps(PDE_2);

The state components have been re-indexed as:
   x3(t)          -->   x1(t)
   x1(t,s1,s2)    -->   x2(t,s1,s2)
   x2(t,s1,s2)    -->   x3(t,s1,s2)
```

indicating that in the updated system structure, the state variables have been reordered to place the finite-dimensional state first.

8.2.4d Combining spatial variables

Although `pde_struct` can be used to declare systems involving an arbitrary number of spatial variables, PIETOOLS 2024 does not support conversion of systems involving more than 2 spatial variables to PIEs. However, in some cases, it is possible to express a higher-dimensional PDE using fewer spatial variables, by rescaling the domain on which each variable is defined. For `pde_struct` objects, this can be achieved using the function `combine_vars` as

```
>> PDE_new = combine_vars(PDE_old,[a,b]);
```

taking a PDE structure representing some desired equation, and returning a structure defining an equivalent representation of the system, but now with all variables rescaled to exist on the interval `[a,b]`, and merged where possible. To illustrate, consider the following simple system of three coupled transport equations

$$\begin{aligned}
\partial_t \phi_1(t,s_1) &= \partial_{s_1}\phi_1(t,s_1), & s_1 &\in [0,1],\ t \geq 0\\
\partial_t \phi_2(t,s_2) &= \partial_{s_2}\phi_2(t,s_1), & s_2 &\in [0,2],\\
\partial_t \phi_3(t,s_3) &= \partial_{s_3}\phi_3(t,s_1), & s_3 &\in [0,3],\\
\mathbf{x}_1(t,0) &= \mathbf{x}_2(t,0) = \mathbf{x}_3(t,0) = 0.
\end{aligned}$$

We can declare this system using three different spatial variables as

```
>> pvar s1 s2 s3
>> phi1 = pde_var(s1,[0,1]); phi2 = pde_var(s2,[0,2]); phi3 = pde_var(s3,[0,3]);
>> PDE_alt = [diff(phi1,'t')==diff(phi1,s1);
              diff(phi2,'t')==diff(phi2,s2);
              diff(phi3,'t')==diff(phi3,s3);
              subs(phi1,s1,0)==0; subs(phi2,s2,0)==0; subs(phi3,s3,0)==0];
>> PDE_alt = initialize(PDE_alt)
Warning: Currently, PIETOOLS supports only problems with at most two distinct
spatial variables. Analysis of the returned PDE structure will not be possible.
Try running "combine_vars" to reduce the dimensionality of your problem.

Encountered 3 state components:
 x1(t,s1), of size 1, differentiable up to order (1) in variables (s1);
 x2(t,s2), of size 1, differentiable up to order (1) in variables (s2);
 x3(t,s3), of size 1, differentiable up to order (1) in variables (s3);

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

  d_t x1(t,s1) = d_s1 x1(t,s1);
  d_t x2(t,s2) = d_s2 x2(t,s2);
  d_t x3(t,s3) = d_s3 x3(t,s3);

  0 = x1(t,0);
  0 = x2(t,0);
  0 = x3(t,0);
```

Upon initializing this system, PIETOOLS already throws a warning that the system cannot be analyzed with PIETOOLS in this form, and recommends running `combine_vars` to reduce the dimensionality of the problem. Obeying our computer overlords, we run this function to combine the variables to exist on the domain $[0,1]$ as

```
>> PDE_alt = combine_vars(PDE_alt,[0,1])

Variables (s1,s2) have been merged with variables (s3,s3) respectively.

All spatial variables have been rescaled to exist on the interval [0,1].

  d_t x1(t,s) = d_s x1(t,s);
  d_t x2(t,s) = 0.5 * d_s x2(t,s);
  d_t x3(t,s) = 0.33333 * d_s x3(t,s);

  0 = x1(t,0);
  0 = x2(t,0);
  0 = x3(t,0);
```

In doing so, the spatial variables s_1 and s_2 are both merged with the spatial variable s_3, all now converted to a single variable s on the domain $[0,1]$. Accordingly, the differential equations are now also expressed in terms of this single spatial variable, introducing e.g. $\mathbf{x}_2(t,s) = \phi_2(t,2s)$ for $s \in [0,1]$, so that $\partial_{s_2}\phi_2(t,s_2)$ becomes $\frac{1}{2}\partial_s \mathbf{x}_2(t,s)$. In this manner, the returned system offers an equivalent representation of the original PDE, through suitable state transformation. Of course, if the system involves distributed inputs or outputs, those will be transformed accordingly as well.

Note that, even for systems that do not allow for spatial variables to be merged, the function `combine_vars` can still be used to rescale all variables to exist on the same domain, which may reduce numerical issues in e.g. stability analysis later on.

8.2.4e Expanding delays

Unlike our PDE representation, the PIE representation does not allow for temporal delay to occur in the state variables or inputs. To resolve this, any delay in the PDE is instead modeled using a transport equation. Specifically, for any variable $\mathbf{v}(t-\tau,s)$, be it a state or input variable, we can introduce a state variable $\mathbf{x}(t,r,s) = \mathbf{v}(t-r\tau,s)$ for $r \in [0,1]$, so that we can express $\mathbf{v}(t-\tau,s) = \mathbf{x}(t,1,s)$ wherever it appears in the PDE dynamics. Here, the state variable $\mathbf{x}(t)$ is governed by a transport equation

$$\partial_t \mathbf{x}(t,r,s) = -\frac{1}{\tau}\partial_r \mathbf{v}(t,r,s), \qquad \mathbf{x}(t,0,s) = \mathbf{v}(t,s).$$

This equation is just a standard PDE with a standard boundary conditions, that can be readily added to the PDE structure and converted to a PIE. In this manner, any temporal delay in a linear PDE can be equivalently represented by adding a suitable transport equation to the system, a process which can be performed for a `pde_struct` object `PDE` by simply calling `PDE=expand_delays(PDE)`. This function returns a new structure that offers an equivalent representation of the input system, but now with an added state component modeled by a transport equation for each delay present in the system.

Note that, using the function `expand_delay` to expand a PDE state $\mathbf{x}(t-\tau, s)$ with delay, the boundary conditions on the state $\mathbf{x}(t)$ do not get automatically imposed on the newly introduces state $\hat{\mathbf{x}}(t, r, s) = \mathbf{x}(t - r\tau, s)$ modeling the delay. This may introduce conservatism when e.g. analyzing stability of the system.

Example

Consider again the wave equation example from (8.1), declared as an initialized `pde_struct` object `PDE_i`. As declared, the system involves a delayed input $\mathbf{u}_2(t-0.5, s_2)$ in the boundary conditions, which we can expand to get

```
>> PDE_d = expand_delays(PDE_i)

Added 1 state components:
   x3(t,s2,ntau_3)  := u2(t-ntau_3,s2);

Variable s1 has been merged with variable ntau_3.

All spatial variables have been rescaled to exist on the interval [-1,1].

 d_t x1(t,s1,s2) = x2(t,s1,s2);
 d_t x2(t,s1,s2) = 2.2222 * d_s1^2 x1(t,s1,s2) + 5 * d_s2^2 x1(t,s1,s2)
                       + C23(s1,s2) * u1(t);
 d_t x3(t,s1,s2) = 4 * d_s2 x3(t,s1,s2);

 y1(t,s1) = x1(t,s1,1) + C42(s1) * w1(t);
 y2(t,s2) = x2(t,1,s2) + w2(t);

 z(t) = int_-1^1 int_-1^1[C61 * x1(t,s1,s2)]ds2 ds1
                       + C62 * d_s1 d_s2 x2(t,1,1);

 0 = x1(t,-1,s2) - x3(t,-1,s2);
 0 = x2(t,-1,s2);
 0 = x1(t,s1,-1);
 0 = x2(t,s1,-1);
 0 = 0.66667 * d_s1 x1(t,1,s2);
 0 = 0.66667 * d_s1 x2(t,1,s2);
 0 = d_s2 x1(t,s1,1);
 0 = d_s2 x2(t,s1,1);
 0 = u2(t,s2) - x3(t,1,s2);
```

In the returned system, a new variable $\mathbf{x}_3(t, s_2, r)$ (where r is `ntau_3`) is introduced to model the delayed input $\mathbf{u}(t-0.5, s_2)$, adding a transport equation to model $\mathbf{x}_3$, as well as a boundary condition $\mathbf{u}_2(t, s_2) = \mathbf{x}_3(t, 1, s_2)$. However, since this also increases the number of spatial variables in the system, the function `combine_vars` (presented in the previous subsection) is called automatically to rescale all variables to exist on the domain $[-1, 1]$, and merge the new variable r with the variable s_1 (since it cannot be merged with s_2). Accordingly, all the dynamics have been rescaled as well, so that e.g. the term $5\partial_{s_1}^2 \mathbf{x}(t, s_1)$ becomes $5(\frac{2}{3})^2 = \partial_{s_1}^2 \mathbf{x}_1(t, s_1)$ to account for the fact that s_1 has been rescaled from the domain $[0, 3]$ to the domain $[-1, 1]$.

8.2.4f Expanding higher-order temporal derivatives

As briefly illustrated earlier, the Command Line Input format can also be used to declare systems with higher-order temporal derivatives. However, the PIE representation does not actually support such higher-order temporal derivatives. Nevertheless, we can easily get around this issue by introducing additional state variables. For example, consider the following wave equation

$$\begin{aligned}
&\partial_t^2 \mathbf{x}_1(t,s_1,s_2) = 5\Big(\partial_{s_1}^2 \mathbf{x}_1(t,s_1,s_2) + \partial_{s_2}^2 \mathbf{x}_1(t,s_1,s_2)\Big), \qquad (s_1,s_2) \in [0,3]\times[-1,1],\ t \geq 0,\\
&\mathbf{x}_1(t,0,s_2) = \partial_{s_1}\mathbf{x}_1(t,3,s_2) = 0,\\
&\mathbf{x}_1(t,s_1,-1) = \partial_{s_2}\mathbf{x}_1(t,s_1,1) = 0,
\end{aligned}$$

which we can declare as

```
>> PDE_w = [diff(x1,'t',2)==5*(diff(x1,s1,2)+diff(x1,s2,2));
            subs(x1,s1,0)==0;  subs(diff(x1,s1),s1,3)==0;
            subs(x1,s2,-1)==0; subs(diff(x1,s2),s2,1)==0];
>> PDE_w = initialize(PDE_w)

Encountered 1 state component:
 x(t,s1,s2),    of size 1, differentiable up to order (2,2) in variables (s1,s2);

Encountered 4 boundary conditions:
 F1(t,s2) = 0,   of size 1;
 F2(t,s2) = 0,   of size 1;
 F3(t,s1) = 0,   of size 1;
 F4(t,s1) = 0,   of size 1;

  d_t^2 x(t,s1,s2) = 5 * d_s1^2 x(t,s1,s2) + 5 * d_s2^2 x(t,s1,s2);

  0 = x(t,0,s2);
  0 = d_s1 x(t,3,s2);
  0 = x(t,s1,-1);
  0 = d_s2 x(t,s1,1);
```

Now, to get rid of the second-order temporal derivative in this system, we can introduce a new state variable $\mathbf{x}_2(t) = \partial_t \mathbf{x}_1(t)$. Then, the wave equation can be equivalently expressed in a format involving only first-order temporal derivatives as

$$\begin{aligned}
&\partial_t \mathbf{x}_1(t,s_1,s_2) = \mathbf{x}_2(t,s_1,s_2), \qquad (s_1,s_2) \in [0,3]\times[-1,1],\ t \geq 0,\\
&\partial_t^2 \mathbf{x}_1(t,s_1,s_2) = 5\Big(\partial_{s_1}^2 \mathbf{x}_1(t,s_1,s_2) + \partial_{s_2}^2 \mathbf{x}_1(t,s_1,s_2)\Big)\\
&\mathbf{x}_1(t,0,s_2) = \partial_{s_1}\mathbf{x}_1(t,3,s_2) = \mathbf{x}_2(t,0,s_2) = \partial_{s_1}\mathbf{x}_2(t,3,s_2) = 0,\\
&\mathbf{x}_1(t,s_1,-1) = \partial_{s_2}\mathbf{x}_1(t,s_1,1) = \mathbf{x}_2(t,s_1,-1) = \partial_{s_2}\mathbf{x}_2(t,s_1,1) = 0.
\end{aligned}$$

Given the `pde_struct` object `PDE_w` representing our wave equation, we can similarly expand this system in a format involving only first-order temporal derivatives, using the function `expand_delays` as

```
>> PDE_w2 = expand_tderivatives(PDE_w)

Added 1 state component:
   x2(t,s1,s2)   := d_t x1(t,s1,s2)

Warning: No BCs have been imposed on the newly added state components
representing the higher order temporal derivatives.
         Results of e.g. stability tests may be very conservative.

 d_t x1(t,s1,s2) = x2(t,s1,s2);
 d_t x2(t,s1,s2) = 5 * d_s1^2 x1(t,s1,s2) + 5 * d_s2^2 x1(t,s1,s2);

 0 = x1(t,0,s2);
 0 = d_s1 x1(t,3,s2);
 0 = x1(t,s1,-1);
 0 = d_s2 x1(t,s1,1);
```

The resulting structure represents the expected (expanded) PDE dynamics in terms of $\mathbf{x}_1(t)$ and $\mathbf{x}_2(t) = \partial_t \mathbf{x}_1(t)$. However, as the function also warns, no boundary conditions are imposed upon this new state variable $\mathbf{x}_2(t)$. Analyzing the stability or norm of this system, the lack of boundary conditions on the newly introduced state may result in more conservative results.

8.3 The `sys` Format for 1D ODE-PDEs

In PIETOOLS 2022, a Command Line Parser was introduced for declaration of 1D ODE-PDE systems, using the `sys` and `state` structures. Although the `pde_var` function has replaced this `sys`-based input format in PIETOOLS 2024, 1D ODE-PDEs can still be declared using this older Command Line Parser format as well. In this section, we give an overview of how the `state` and `sys` classes can be used for defining and manipulating 1D ODE-PDE systems. 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:

```
>> pvar t s;
>> 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));
```

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 $s=1$.

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')
- `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)`

Operation type	Incorrect or 'not-permitted' operations
Addition	✖ Adding two time derivatives: `diff(x,t)+diff(x,t)` ✖ Adding two outputs: `z1+z2` ✖ Adding time derivative and outptu: `diff(x,t)+z`
Multiplication	✖ Multiplying two states: `x*x` ✖ Multiplying non-identity with time derivative/output: `2*diff(x,t)` or `-1*z` ✖ Right multiplication: `x*3`
Differentiation	✖ Higher order time derivatives: `diff(x,t,2)` ✖ Mixed derivatives of space and time: `diff(diff(x,t),s,2)`
Substitution	✖ Substituting a double for time variable: `subs(x,t,2)` ✖ Substituting positive time delay: `subs(x,t,t+5)` ✖ Substitution values other than `pvar` variable or boundary values
Integration	✖ Integration of time variable: `int(x,t,[0,5])` ✖ both limits being non-numeric: `int(x,s,[theta,eta])` ✖ limit same as variable of integration: `int(x,s,[s,1])`
Concatenation	✖ Horizontal concatenation: `[x,x]` ✖ Blank spaces in vertical concatentation: `[x + y; z]`

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.

- 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.

$$\begin{bmatrix}\dot{x}(t)\\ z(t)\\ y(t)\end{bmatrix} = \begin{bmatrix}A_0 & B_1 & B_2\\ C_1 & D_{11} & D_{12}\\ C_2 & D_{21} & D_{22}\end{bmatrix}\begin{bmatrix}x(t)\\ w(t)\\ u(t)\end{bmatrix} + \sum_{i=1}^{K}\begin{bmatrix}A_i & B_{1i} & B_{2i}\\ C_{1i} & D_{11i} & D_{12i}\\ C_{2i} & D_{21i} & D_{22i}\end{bmatrix}\begin{bmatrix}x(t-\tau_i)\\ w(t-\tau_i)\\ u(t-\tau_i)\end{bmatrix} + \sum_{i=1}^{K}\int_{-\tau_i}^{0}\begin{bmatrix}A_{di}(s) & B_{1di}(s) & B_{2di}(s)\\ C_{1di}(s) & D_{11di}(s) & D_{12di}(s)\\ C_{2di}(s) & D_{21di}(s) & D_{22di}(s)\end{bmatrix}\begin{bmatrix}x(t+s)\\ w(t+s)\\ u(t+s)\end{bmatrix} ds \tag{9.1}$$

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 senors.
- 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 is value. For example, to represent

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

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].

$$+\sum_{i=1}^{K}\int_{-\tau_i}^{0}\begin{bmatrix} A_{di}(s) & B_{1di}(s) & B_{2di}(s) & E_{di}(s) \\ C_{1di}(s) & D_{11di}(s) & D_{12di}(s) & E_{1di}(s) \\ C_{2di}(s) & D_{21di}(s) & D_{22di}(s) & E_{2di}(s) \end{bmatrix}\begin{bmatrix} x(t+s) \\ w(t+s) \\ u(t+s) \\ \dot{x}(t+s) \end{bmatrix} ds \tag{9.2}$$

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 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.

$$\begin{bmatrix} \dot{x}(t) \\ z(t) \\ y(t) \\ r_i(t) \end{bmatrix} = \begin{bmatrix} A_0 & B_1 & B_2 \\ C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \\ C_{ri} & B_{r1i} & B_{r2i} \end{bmatrix} \begin{bmatrix} x(t) \\ w(t) \\ u(t) \end{bmatrix} + \begin{bmatrix} B_v \\ D_{1v} \\ D_{2v} \\ D_{rvi} \end{bmatrix} v(t)$$
$$v(t) = \sum_{i=1}^{K} C_{vi} r_i(t-\tau_i) + \sum_{i=1}^{K} \int_{-\tau_i}^{0} C_{vdi}(s) r_i(t+s) ds. \tag{9.3}$$

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. Not 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,'dde');
```

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 represenation, 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 Setion 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 [10].

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

form

$$\big(\mathcal{T}\mathbf{x}\big)(s) = \begin{bmatrix} Px_0 & + \int_a^b Q_1(s)\mathbf{x}_1(s)ds \\ Q_2(s)x_0 & + R_0(s)\mathbf{x}_1(s) + \int_a^s R_1(s,\theta)\mathbf{x}_1(\theta)d\theta + \int_s^b R_2(s,\theta)\mathbf{x}_1(\theta)d\theta \end{bmatrix} \tag{10.1}$$

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

`dim`	`= [m0,n0; m1,n1]`	2×2 array of type `double` 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;
`var1`	`= s`	1×1 `pvar` (`polynomial` class) object specifying the spatial variable s;
`var2`	`= theta`	1×1 `pvar` (`polynomial` class) object specifying the dummy variable θ;
`I`	`= [a,b]`	1×2 array of type `double`, specifying the interval $[a,b]$ on which the spatial variables s and θ exist;
`P`	`= P`	$m_0 \times n_0$ array of type `double` or `polynomial` defining the matrix P;
`Q1`	`= Q1`	$m_0 \times n_1$ array of type `double` or `polynomial` defining the function $Q_1(s)$;
`Q2`	`= Q2`	$m_1 \times n_0$ array of type `double` or `polynomial` defining the function $Q_2(s)$;
`R.R0`	`= R0`	$m_1 \times n_1$ array of type `double` or `polynomial` defining the function $R_0(s)$;
`R.R1`	`= R1`	$m_1 \times n_1$ array of type `double` or `polynomial` defining the function $R_1(s,\theta)$;
`R.R2`	`= R2`	$m_1 \times n_1$ array of type `double` or `polynomial` defining the function $R_2(s,\theta)$;

As an example, suppose we want to declare the 4-PI operator $\mathcal{T} : \begin{bmatrix} \mathbb{R}^2 \\ L_2^3[2,3] \end{bmatrix} \to \begin{bmatrix} \mathbb{R}^3 \\ L_2[2,3] \end{bmatrix}$ defined as

$$\big(\mathcal{T}\mathbf{x}\big)(r) = \begin{bmatrix} \overbrace{\begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 3 & 4 \end{bmatrix}}^{P} x_0 & + \int_2^3 \overbrace{\begin{bmatrix} r^2 & 0 & 0 \\ 3 & r^3 & 0 \\ 0 & r+2*r^2 & 0 \end{bmatrix}}^{Q_1(r)} \mathbf{x}_1(r)dr \\ \underbrace{\begin{bmatrix} -5r & 6 \end{bmatrix}}_{Q_2(s)} x_0 & + \int_2^r \underbrace{\begin{bmatrix} r & 2\nu & 3(r-\nu) \end{bmatrix}}_{R_1(r,\nu)} \mathbf{x}_1(\nu)d\nu \end{bmatrix}, \qquad r \in [2,3],$$

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 $\begin{bmatrix} \mathbb{R}^2 \\ L_2^3[2,3] \end{bmatrix} \to \begin{bmatrix} \mathbb{R}^3 \\ L_2[2,3] \end{bmatrix}$, 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

$$p(c_0, c_1, c_2; x) = c_0 + c_1 x + c_2 x^2,$$

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 $\mathcal{D} : L_2^2[0,1] \to L_2^2[0,1]$ defined as

$$\big(\mathcal{D}\mathbf{x}\big)(s) = \underbrace{\begin{bmatrix} c_1 & c_2 s \\ c_2 s & c_3 s^2 \end{bmatrix}}_{R_0(s)} \mathbf{x}(s) + \int_s^1 \underbrace{\begin{bmatrix} c_4 s^2 & c_5 s\theta \\ -c_6 s\theta & c_7 \theta^2 \end{bmatrix}}_{R_2(s,\theta)} \mathbf{x}(\theta) d\theta, \qquad s \in [0,1]$$

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 $\mathcal{A}, \mathcal{B} : \left[\begin{smallmatrix}\mathbb{R}^2\\ L_2[-1,1]\end{smallmatrix}\right] \to \left[\begin{smallmatrix}\mathbb{R}^2\\ L_2[-1,1]\end{smallmatrix}\right]$ defined as

$$\big(\mathcal{A}\mathbf{x}\big)(s) = \begin{bmatrix} \left[\begin{smallmatrix}1 & 0\\ 2 & -1\end{smallmatrix}\right] x_0 \quad + \int_{-1}^{1} \left[\begin{smallmatrix}1-s\\ s+1\end{smallmatrix}\right] \mathbf{x}_1(s)ds \\ \left[\begin{smallmatrix}10s & -1\end{smallmatrix}\right] x_0 \quad + 2\mathbf{x}_1(s) + \int_{-1}^{s}(s-\theta)\mathbf{x}_1(\theta)d\theta + \int_{s}^{1}(s-\theta)\mathbf{x}_1(\theta)d\theta \end{bmatrix}, \tag{10.2}$$

$$\big(\mathcal{B}\mathbf{x}\big)(s) = \begin{bmatrix} \left[\begin{smallmatrix}1 & 0\\ 0 & 3\end{smallmatrix}\right] x_0 \\ \left[\begin{smallmatrix}5s & -s\end{smallmatrix}\right] x_0 \quad + s^2\mathbf{x}_1(s) + \int_{s}^{1}\theta\mathbf{x}_1(\theta)d\theta \end{bmatrix}, \qquad s \in [-1,1],$$

for $\mathbf{x} = \left[\begin{smallmatrix}x_0\\ \mathbf{x}_1\end{smallmatrix}\right] \in \left[\begin{smallmatrix}\mathbb{R}^2\\ L_2[-1,1]\end{smallmatrix}\right]$. 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];
>> A.Q2 = [10*s,-1];      B.Q2 = [5*s,-s];
>> A.R.R0 = 2;            B.R.R0 = s^2;
>> A.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 $\mathbf{x} = \left[\begin{smallmatrix} x_0 \\ \mathbf{x}_1 \end{smallmatrix}\right] \in \left[\begin{smallmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{smallmatrix}\right]$,

$$\Big(\mathcal{A}\mathbf{x}\Big)(s) + \Big(\mathcal{B}\mathbf{x}\Big)(s) = \begin{bmatrix} \left[\begin{smallmatrix} 2 & 0 \\ 2 & 2 \end{smallmatrix}\right] x_0 & + \int_{-1}^{1} \left[\begin{smallmatrix} 1-s \\ s+1 \end{smallmatrix}\right] \mathbf{x}_1(s) ds \\ \left[\begin{smallmatrix} 15s & -s-1 \end{smallmatrix}\right] x_0 & + (s^2+2)\mathbf{x}_1(s) + \int_{-1}^{s} (s-\theta)\mathbf{x}_1(\theta) d\theta + \int_{s}^{1} s\mathbf{x}_1(\theta) d\theta \end{bmatrix}.$$

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 $\mathbf{x} = \left[\begin{smallmatrix} x_0 \\ \mathbf{x}_1 \end{smallmatrix}\right] \in \left[\begin{smallmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{smallmatrix}\right]$,

$$\Big(\mathcal{A}\Big(\mathcal{B}\mathbf{x}\Big)\Big)(s) = \begin{bmatrix} \left[\begin{smallmatrix} -2\frac{1}{3} & \frac{2}{3} \\ 5\frac{1}{3} & -3\frac{2}{3} \end{smallmatrix}\right] x_0 & + \int_{-1}^{1} \left[\begin{smallmatrix} -1\frac{1}{2}s^3+2s^2+1\frac{1}{2}s \\ 1\frac{1}{2}s^3+2s^2+\frac{1}{2}s \end{smallmatrix}\right] \mathbf{x}_1(s) ds \\ \left[\begin{smallmatrix} 20s-3\frac{1}{3} & -2s-2\frac{1}{3} \end{smallmatrix}\right] x_0 & + 2s^2\mathbf{x}_1(s) + \int_{-1}^{s} (2s\theta^2 - 1\tfrac{1}{2}\theta^3 + s\theta + \tfrac{1}{2}\theta)\mathbf{x}_1(\theta) d\theta \\ & \quad + \int_{s}^{1} (2s\theta^2 - 1\tfrac{1}{2}\theta^3 + s\theta + 2\tfrac{1}{2}\theta)\mathbf{x}_1(\theta) d\theta \end{bmatrix}.$$

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 $\mathcal{A} : RL^{n_0,n_1}[a,b] \to RL^{m_0,m_1}[a,b]$, the adjoint $\mathcal{A}^* : RL^{m_0,m_1}[a,b] \to RL^{n_0,n_1}[a,b]$ 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 $\mathbf{x} = \left[\begin{smallmatrix} x_0 \\ \mathbf{x}_1 \end{smallmatrix}\right] \in \left[\begin{smallmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \end{smallmatrix}\right] = RL^{n_0,n_1}[a,b]$ and $\mathbf{y} = \left[\begin{smallmatrix} y_0 \\ \mathbf{y}_1 \end{smallmatrix}\right] \in \left[\begin{smallmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \end{smallmatrix}\right] = RL^{n_0,n_1}[a,b]$,

$$\langle \mathbf{x}, \mathbf{y} \rangle_{RL} := \langle x_0, y_0 \rangle + \langle \mathbf{x}_1, \mathbf{y}_1 \rangle_{L_2} = x_0^T y_0 + \int_a^b [\mathbf{x}_1(s)]^T \mathbf{y}_1(s) ds$$

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 $\mathbf{x} = \left[\begin{smallmatrix} x_0 \\ \mathbf{x}_1 \end{smallmatrix}\right] \in \left[\begin{smallmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{smallmatrix}\right]$,

$$\left(\mathcal{A}^*\mathbf{x}\right)(s) = \begin{bmatrix} \left[\begin{smallmatrix} 1 & 2 \\ 0 & -1 \end{smallmatrix}\right] x_0 & + \int_{-1}^{1} \left[\begin{smallmatrix} 10s \\ -1 \end{smallmatrix}\right] \mathbf{x}_1(s) ds \\ \left[\begin{smallmatrix} 1-s & s+1 \end{smallmatrix}\right] x_0 & + 2\mathbf{x}_1(s) + \int_{-1}^{1} (\theta - s)\mathbf{x}_1(\theta) d\theta + \int_{s}^{1} (\theta - s)\mathbf{x}_1(\theta) d\theta \end{bmatrix}.$$

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 $\mathbf{x} = \left[\begin{smallmatrix} x_0 \\ \mathbf{x}_1 \end{smallmatrix}\right] \in \left[\begin{smallmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{smallmatrix}\right]$

$$\left(\mathcal{A}^{-1}\mathbf{x}\right)(s) = \begin{bmatrix} \left[\begin{smallmatrix} -0.2 & -0.4 \\ 1.2 & 0.4 \end{smallmatrix}\right] x_0 & + \int_{-1}^{1} \left[\begin{smallmatrix} 0.3s+0.2 \\ -0.3s-0.7 \end{smallmatrix}\right] \mathbf{x}_1(s)ds \\ \left[\begin{smallmatrix} 0.3s+0.7 & 1.35s+0.65 \end{smallmatrix}\right] x_0 & + 0.5\mathbf{x}_1(s) + \int_{-1}^{1}(-1.2s\theta - 0.675s - 0.3\theta - 0.575)\mathbf{x}_1(\theta)d\theta \\ & \quad + \int_{s}^{1}(-1.2s\theta - 0.675s - 0.3\theta - 0.575)\mathbf{x}_1(\theta)d\theta \end{bmatrix}.$$

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 desired rows and columns of `opvar` objects can be extracted, and in Subsection 10.3.2 we show how `opvar` objects can be concatenated. 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 operator $\mathcal{A} : \left[\begin{smallmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{smallmatrix}\right] \to \left[\begin{smallmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{smallmatrix}\right]$ defined in Equation (10.2), represented in PIETOOLS by the `opvar` object `A`,

```
>> A
A =
          [1,0] | [-s+1]
         [2,-1] | [s+1]
       ----------------
       [10*s,-1] | A.R

A.R =
    [2] | [s-th] | [s-th]
```

10.3.1 Subs-indexing (`A(i,j)`)

Just like matrices, 4-PI operators can be sliced to extract desired rows or columns, returning new 4-PI operators in lower dimensions. This index slicing is performed in the same manner as for matrices, extracting rows `row_ind` and columns `col_ind` of an `opvar` object `T` as

```
» T(row_ind, col_ind)
```

However, 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:

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

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 that 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 Consider the `opvar` object `A` corresponding to the operator $\mathcal{A} : \left[\begin{smallmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{smallmatrix}\right] \to \left[\begin{smallmatrix} \mathbb{R}^2 \\ L_2[-1,1] \end{smallmatrix}\right]$ defined as in Equation (10.2). We can decompose this operator into subcomponents $\mathcal{A} = \left[\begin{smallmatrix} \mathcal{A}_{11} & \mathcal{A}_{12} \\ \mathcal{A}_{21} & \mathcal{A}_{22} \end{smallmatrix}\right]$, where $\mathcal{A}_{11} : \mathbb{R} \to \mathbb{R}^2$, $\mathcal{A}_{12} : \left[\begin{smallmatrix} \mathbb{R} \\ L_2[-1,1] \end{smallmatrix}\right] \to \mathbb{R}^2$, $\mathcal{A}_{21} : \mathbb{R} \to L_2[-1,1]$ and $\mathcal{A}_{22} : L_2[-1,1] \to L_2[-1,1]$, by taking

```
>> A11 = A([1,2],1)
A11 =
      [1] | []
      [2] |
      -----------
       [] | A11.R
A11.R =
    [] | [] | []

>> A12 = A([1,2],2:3)
A12 =
       [0] | [-s+1]
      [-1] | [s+1]
      -------------
        [] | A12.R
A12.R =
    [] | [] | []

>> A21 = A(3,1)
A21 =
          [] | []
      ---------------
      [10*s] | A21.R
A21.R =
    [] | [] | []

>> A22 = A(3,2:3)
A22 =
        [] | []
      -------------
```

```
        [-1] | A12.R
A22.R =
    [2] | [s-th] | [s-th]
```

10.3.2 Concatenation ([A,B], [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 commands

```
» [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(:,1), and they can be concatenated vertically only if they have the same column dimensions (A.dim(:,2)=B.dim(:,2)). Finally, note that opvar objects can only represent maps $\mathbb{R}^{n_0} \times L_2^{n_1} \to \mathbb{R}^{m_0} \times L_2^{m_1}$. Therefore, if e.g. A: $L_2 \to \mathbb{R}$ and B: $L_2 \to L_2$, then the concatenation [B;A]: $L_2 \to \mathbb{R} \times L_2$ can be represented as an opvar object, but the concatenation [B;A]: $L_2 \to L_2 \times \mathbb{R}$ cannot. Therefore, this latter concatenation is currently prohibited. Similarly, if A: $\mathbb{R} \to L_2$ and B: $L_2 \to L_2$, we can take the concatenation [A,B]: $\mathbb{R} \times L_2 \to L_2$, but we cannot concatenate [B,A]: $L_2 \times R \to L_2$.

Example Consider the opvar objects A11, A12, A21 and A22, corresponding to the operator $\mathcal{A} = \left[\begin{smallmatrix} \mathcal{A}_{11} & \mathcal{A}_{12} \\ \mathcal{A}_{21} & \mathcal{A}_{22} \end{smallmatrix}\right]$ defined in (10.2), decomposed as in the previous subsection. Then, we can reconstruct an opvar object A representing the full operator $\mathcal{A}$ as

```
>> A = [A11, A12; A21, A22]
A =
          [1,0] | [-s+1]
         [2,-1] | [s+1]
      ----------------
      [10*s,-1] | A.R
A.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
`A==B`	The function checks whether the variables, domain, dimensions and parameters of the operators `A` and `B` 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 `A` has all parameters equal to zero operator by calling `A==0`.
`isvalid(P)`	The function returns a logical value. 0 is everything is in order, 1 if the object has incompatible dimensions, 2 if property `P` is not a matrix, 3 if properties `Q1, Q2` or `R0` are not polynomials in s, 4 if properties `R1` or `R2` are not polynomials in s and θ. For `opvar2d` objects, returns boolean value `true` or `false` to indicate if `P` is appropriate or not.
`degbalance(T)`	Estimates polynomial degrees needed to create an `opvar` object `Q` associated to a positive PI operator $\mathcal{Q} \succeq 0$ in `poslpivar`, such that `T=Q` has at least one solution.
`getdeg(T)`	Returns highest and lowest degree of s and θ in the components of the opvar object `T`.
`rand_opvar(dim, deg)`	Creates a random opvar object of specified dimensions `dim` and polynomial degrees `deg`.
`show(T,opts)`	Alternative display format for opvar objects with optional argument to omit selected properties from display output. **Not defined for `opvar2d` objects.**
`opvar_postest(T)`	Numerically test for sign definiteness of `T`. Returns -1 if negative definite, 0 if indefinite and 1 if positive definite. Use `opvar_postest_2d` for `opvar2d` objects.
`diff_opvar(T)`	Returns composition of derivative operator with opvar `T` as described in Lem. 10. Use `diff(T)` for `opvar2d` 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. Note that, although running these demos will not produce the plots presented throughout this chapter, the code to reproduce each of these figures has been added in the script "PIETOOLS_Code_Illustrations_Ch11_Demos".

11.1 DEMO 1: Stability Analysis and Simulation

With this demo, we illustrate how an ODE-PDE model can be simulated with PIESIM, and how stability of the system can be tested using LPI programming. For this purpose, we consider the same damped 1D wave equation coupled to a stable ODE as in Chapter 2, given by

$$\begin{aligned}\dot{x}(t) &= -x(t),\\ \ddot{\mathbf{x}}(t,s) &= c^2\partial_s^2\mathbf{x}(t,s) - b\partial_s\mathbf{x}(t,s) + sw(t), s\in(0,1), t\geq 0,\\ r(t) &= \mathbf{x}(t,1) - \mathbf{x}(t,0).\end{aligned}$$

for some given velocity c and viscous damping coefficient b, and with external disturbance $w(t)\in\mathbb{R}$ and regulated output $r(t)\in\mathbb{R}$. To implement this system in PIETOOLS, we introduce the PDE state $\phi = (\partial_s\mathbf{x}, \dot{\mathbf{x}})$. Then, the system can be equivalently expressed as

$$\begin{aligned}\dot{x}(t) &= -x(t)\\ \dot{\phi}(t,s) &= \begin{bmatrix}0 & 1\\ c & 0\end{bmatrix}\partial_s\phi(t,s) + \begin{bmatrix}0 & 0\\ 0 & -b\end{bmatrix}\phi(t,s) + \begin{bmatrix}0\\ s\end{bmatrix}w(t), s\in(0,1), t\geq 0,\\ r(t) &= \int_0^1 \phi_1(t,s)ds.\end{aligned} \tag{11.1}$$

For e.g. $c=1$ and $b=0.01$, this system can be declared in PIETOOLS as

```
% Declare independent variables
pvar t s;
% Declare state, input, and output variables
phi = pde_var(2,s,[0,1]);    x = pde_var();
w = pde_var('in');           r = pde_var('out');
```

```
% Declare system equations
c=1;    b=.01;
eq_dyn = [diff(x,t,1)==-x
          diff(phi,t,1)==[0 1; c 0]*diff(phi,s,1)+[0;s]*w+[0 0;0 -b]*phi];
eq_out= r==int([1 0]*phi,s,[0,1]);
bc1 = [0 1]*subs(phi,s,0)==0;   % add the boundary conditions
bc2 = [1 0]*subs(phi,s,1)==x;
odepde = [eq_dyn;eq_out;bc1;bc2];
```

To see whether this system is stable, we first simulate it using PIESIM. For this, we consider an initial value of the ODE state $x(0) = \frac{1}{2}$, and initial PDE state values $\phi_1(0, s) = 5\sin(2\pi s)$ and $\phi_2(0, s) = 0$. We further assume the disturbance to be given by a decaying but oscillating function $w(t) = e^{-t}\sin(5t)$. We declare these values as

```
% % Declare initial values and disturbance
syms st sx;
uinput.ic.ODE = 0.5;
uinput.ic.PDE =  [0.5-sx,sin(pi*sx)];
uinput.w = sin(5*st)*exp(-st);
```

Now, to perform simulation with these values, we will use PIESIM, expanding the PDE state $\phi(t, s)$ using 16 Chebyshev in s polynomial, and simulating up to time $t = 9$, using a time step of 0.03.

```
opts.plot = 'yes';  % plot the solution
opts.N = 16;        % expand using 16 Chebyshev polynomials
opts.tf = 9;        % simulate up to t = 9;
opts.dt = 0.03;     % use time step of 3*10^-2
```

Having declared these settings, we call PIESIM to simulate as

```
[solution,grids] = PIESIM(odepde, opts, uinput);
tval = solution.timedep.dtime;
phi1 = reshape(solution.timedep.pde(:,1,:),opts.N+1,[]);
phi2 = reshape(solution.timedep.pde(:,2,:),opts.N+1,[]);
zval = solution.timedep.regulated;
wval = subs(uinput.w,st,tval);
```

extracting the values of the ODE and PDE state at each time step, as well as those of the disturbance and regulated output. Note that the PDE state solutions are provided only at the $N + 1 = 17$ grid points at each time step. Fig. 11.1 and Fig. 11.2 show the simulated evolution of the different ODE and PDE state variables, as well as the regulated output.

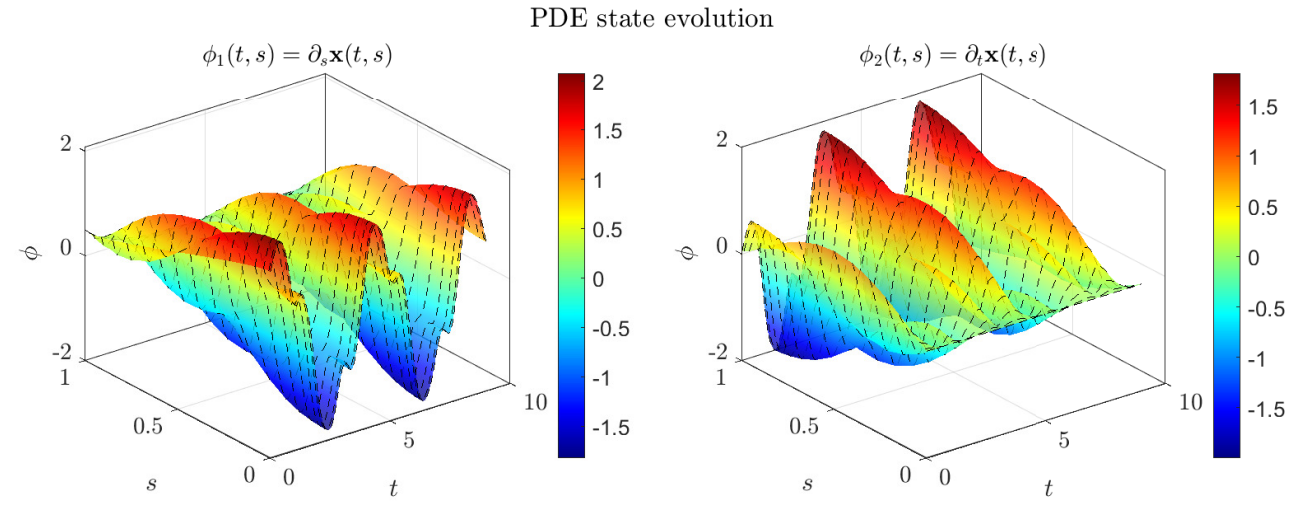


Figure 11.1: Simulated value of PDE state variables $\phi_1(t,s) = \partial_s \mathbf{x}(t,s)$ and $\phi_2(t,s) = \partial_t \mathbf{x}(t,s)$ associated to the ODE-PDE (11.1).

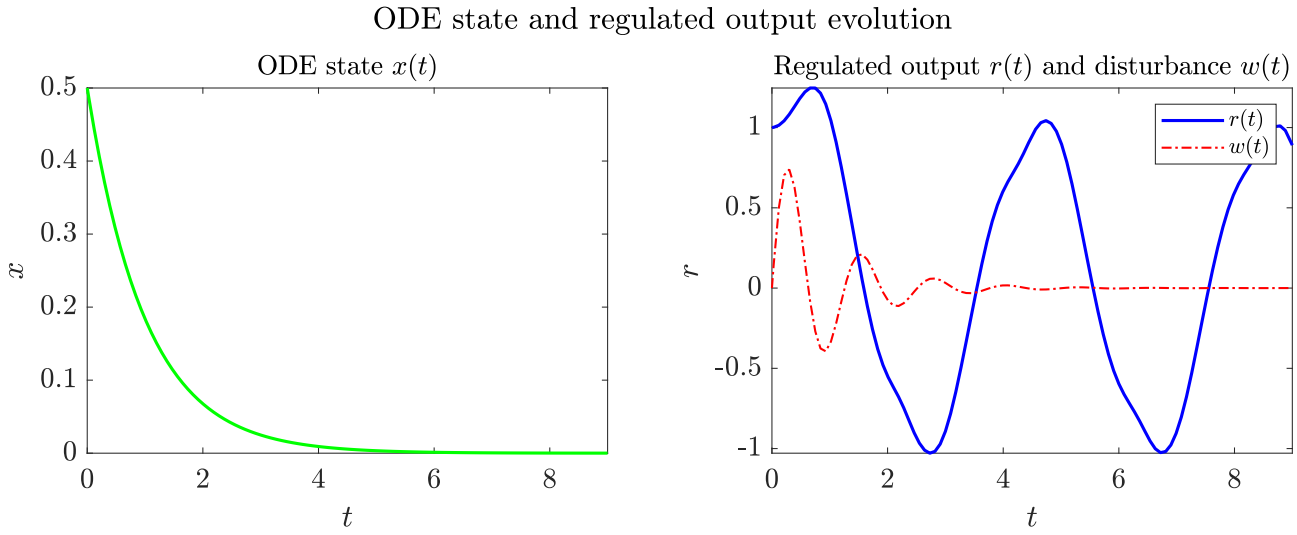


Figure 11.2: Simulated value of ODE state variable $x(t)$ and regulated output $r(t) = \int_0^1 \phi_1(t,s)ds$ associated to the ODE-PDE (11.1).

From the figures it appears that, although oscillatory, solutions to the system are stable, not growing with time (despite the presence of disturbances). To verify that the system is indeed stable, we can test stability in the PIE representation. To this end, we first convert the system to a PIE as

```
pie = convert(odepde);
T = pie.T;
A = pie.A;     B = pie.B1;
C = pie.C1;    D = pie.D11
```

For our ODE-PDE system (11.1), the associated PIE representation takes the form

$$\begin{aligned}\partial_t(\mathcal{T}\mathbf{x}_{\rm f})(t) &= \mathcal{A}\mathbf{x}_{\rm f}(t) + \mathcal{B}w(t),\\ z(t) &= \mathcal{C}\mathbf{x}_{\rm f}(t) + \mathcal{D}w(t),\end{aligned}$$

where now $\mathbf{x}_{\rm f}(t,s) = (x(t), \partial_s\phi_1(t,s), \partial_s\phi_2(t,s)) \in \mathbb{R}\times L_2^2[0,1]$. In the absence of disturbances, $w(t)=0$, stability of the autonomous system can be tested by solving the LPI

$$\mathcal{P}\succ 0, \qquad \mathcal{T}^*\mathcal{P}\mathcal{A} + \mathcal{A}\mathcal{P}\mathcal{T} \preceq 0. \tag{11.2}$$

This LPI can be declared and solved as

```
% % Initialize LPI program
prog = lpiprogram(s,[0,1]);

% % Declare decision variables:
% %   P: R x L2^2 --> R x L2^2,    P>0
[prog,P] = poslpivar(prog,[1;2]);
P = P + 1e-4;                    % enforce P>=1e-4

% % Set inequality constraints:
% %   A'*P*T + T'*P*A <= 0
Q = A'*P*T + T'*P*A;
opts.psatz = 1;                  % allow Q>=0 outside domain
prog = lpi_ineq(prog,-Q,opts);

% % Solve and retrieve the solution
solve_opts.solver = 'sedumi';    % use SeDuMi to solve
solve_opts.simplify = true;      % simplify SDP before solving
prog = lpisolve(prog,solve_opts);
```

Running this code, we find that the optimization program defined by `prog` is feasible, indicating that the operators defined by `A` and `T` indeed represent a stable system. Thus, the ODE-PDE system (11.1) is stable.

11.2 DEMO 2: Estimating the Volterra Operator Norm

The Volterra integral operator $\mathcal{T}: L_2[0,1]\to L_2[0,1]$ is perhaps the simplest example of a 3-PI operator, defined as

$$\Big(\mathcal{T}\mathbf{x}\Big)(s) = \int_0^s \mathbf{x}(\theta)d\theta, \qquad\qquad s\in[0,1],$$

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

$$\min_{\gamma\geq 0}\ \gamma,$$

$$\text{s.t.} \qquad \mathcal{T}^*\mathcal{T} \leq \gamma,$$

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

```
% % Initialize LPI program
prob = lpiprogram(Top.vars,Top.I);

% % Declare decision variables:
% %   gam \in \R
[prob,gam] = lpidecvar(prob,'gam');

% % Set inequality constraints
% %   Top'*Top-gam <= 0
opts.psatz = 1;                              % Allow gam-Top'*Top<0 outside of [a,b]
prob = lpi_ineq(prob,gam-Top'*Top,opts); % lpi_ineq(prob,Q) enforces Q>=0

% % Set objective function:
% %   min gam
prob = lpisetobj(prob, gam);

% % Solve LPI and retrieve solution
prob = lpisolve(prob);
operator_norm = sqrt(double(lpigetsol(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 $\|\mathcal{T}\| = \frac{2}{\pi} = 0.6366....$

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,

$$\|x\|_{L_2} \leq C\|\partial_s x\|_{L_2}, \qquad \forall x \in W_1[a, b],$$

where

$$W_1[a, b] := \{x \in L_2[a, b] \mid \partial_s x \in L_2[a, b], x(a) = x(b) = 0\}.$$

The challenge of finding a smallest value of C for which this inequality holds can be posed as an optimization problem

$$\begin{aligned} &\min_{\gamma \geq 0} \ \gamma, \\ \text{s.t.} \quad &\langle x, x\rangle_{L_2} - \gamma \langle \partial_s x, \partial_s x\rangle_{L_2} \leq 0, \qquad \forall x \in W_1[a, b] \end{aligned}$$

setting $C = \sqrt{\gamma}$. To declare this problem, we first note that (assuming $\partial_s^2 x$ exists) 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 $\partial_s x$. To represent this in PIETOOLS, we introduce an artificial PDE

$$\begin{aligned} \dot{x}(t,s) &= \partial_s^2 x(t,s), && s \in [a,b], \\ z(t,s) &= \partial_s x(t,s), \\ x(t,a) &= x(t,b) = 0, \end{aligned}$$

which we declare as

```
a = 0;  b = 1;
pvar t s
x = pde_var(1,s,[a,b]);
z = pde_var('output',1,s,[a,b]);
PDE = [diff(x,t)==diff(x,s,2);
       z==diff(x,s,1);
       subs(x,s,a)==0;
       subs(x,s,b)==0];
```

Here, we take a second-order derivative of x in the PDE to explicitly indicate that x must be second order differentiable, as we wish to derive a PIE representation in terms of the fundamental state $x_{\mathrm{f}} := \partial_s^2 x$. To obtain this PIE representation, we call `convert`,

```
PIE = convert(PDE);
H2op = PIE.T;       % (H2op*x_{ss}) = x;
H1op = PIE.C1;      % (H1op*x_{ss}) = x_{s}
```

arriving at an equivalent PIE representation of the system as

$$\begin{aligned} \partial_t(\mathcal{H}_2 x_{\mathrm{f}})(t,s) &= x_{\mathrm{f}}(t,s), && s \in [a,b], \\ z(t,s) &= (\mathcal{H}_1 x_{\mathrm{f}})(t,s). \end{aligned}$$

In this representation, the fundamental state $x_{\mathrm{f}} := \partial_s^2 x$ is free of any boundary conditions and continuity constraints. Moreover, we note that

$$\mathcal{H}_2 x_{\mathrm{f}}(t,s) = x(t,s), \qquad \text{and,} \qquad \mathcal{H}_1 x_{\mathrm{f}}(t,s) = \partial_s x(t,s).$$

As such, the Poincaré inequality optimization problem can be equivalently represented as

$$\min_{\gamma \geq 0} \ \gamma, \qquad \text{s.t.} \quad \langle \mathcal{H}_2 v, \mathcal{H}_2 v \rangle - \gamma \langle \mathcal{H}_1 v, \mathcal{H}_1 v \rangle \leq 0, \qquad \forall v \in L_2[a,b]$$

giving rise to an LPI

$$\min_{\gamma \geq 0} \ \gamma, \qquad \text{s.t.} \quad \gamma \mathcal{H}_1^* \mathcal{H}_1 - \mathcal{H}_2^* \mathcal{H}_2 \succeq 0.$$

We declare and solve this LPI in PIETOOLS as

```
% % Initialize LPI program
prob = lpiprogram(s,[a,b]);

% % Declare decision variables:
% %   gam \in \R
[prob,gam] = lpidecvar(prob,'gam');     % scalar decision variable

% % Set inequality constraints:
% %   gam*H1op'*H1op' - H2op'*H2op >= 0
```

```
opts.psatz = 1;    % allow gam H1op'*H1op < H2op'*H2op outside of [a,b]
prob = lpi_ineq(prob,gam*(H1op'*H1op)-H2op'*H2op,opts);

% % Set objective function:
% %   min gam
prob = lpisetobj(prob, gam);

% % Solve LPI and retrieve solution
prob = lpisolve(prob);
poincare_constant = sqrt(double(lpigetsol(prob,gam)));
```

This code can also be run calling the function "DEMO3_poincare_inequality", arriving at a constant $C = 0.4271$ that satisfies the Poincaré inequality on the domain $[a, b] = [0, 1]$. On this domain, a minimal value of C is known to be $C = \frac{1}{\pi} = 0.3183....$

11.4 DEMO 4: Finding an Optimal Stability Parameter

We consider a reaction-diffusion equation on an interval $[a, b]$,

$$\dot{\mathbf{x}}(t, s) = \lambda \mathbf{x}(t, s) + \partial_s^2 \mathbf{x}(t, s), \qquad s \in [a, b], \qquad \mathbf{x}(t, a) = \mathbf{x}(t, b) = 0,$$

for some $\lambda \in \mathbb{R}$. On the interval $[a, b] = [0, 1]$, this system is know to be stable whenever $\lambda \leq \pi^2 = 9.8696....$ In PIETOOLS, we can numerically estimate this limit using the function `stability_PIETOOLS`. In particular, we may equivalent represent the PDE as a PIE of the form

$$\partial_t \big(\mathcal{T}\mathbf{x}_\mathrm{f}\big)(t, s = \big(\mathcal{A}(\lambda)\mathbf{x}_\mathrm{f}\big)(t, s), \qquad s \in [a, b],$$

where $\mathbf{x}_\mathrm{f}(t, s) := \partial_s^2 \mathbf{x}(t, s)$. Then, a maximal value of λ for which the PIE is stable can be estimated by solving the optimization problem

$$\begin{aligned} &\max_{\lambda \in \mathbb{R}, \mathcal{P} \in \Pi} \ \lambda, \\ \text{s.t.} \quad &\mathcal{P} \succ 0, \\ &\mathcal{T}^* \mathcal{P} \mathcal{A}(\lambda) + \mathcal{A}^*(\lambda) \mathcal{P} \mathcal{T} \preccurlyeq 0. \end{aligned}$$

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

$$\mathcal{P} \succ 0, \qquad \mathcal{T}^* \mathcal{P} \mathcal{A}(\lambda) + \mathcal{A}^*(\lambda) \mathcal{P} \mathcal{T} \preccurlyeq 0,$$

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;
```

Using these settings, we perform bisection to find a largest value of λ for which stability of the system can be verified. In particular, for a given value of $\lambda \in [\lambda_{\min}, \lambda_{\max}]$, we build a PDE structure `PDE` representing the system, compute the associated PIE structure `PIE`, and test stability of this PIE by solving the LPI. 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
    % ================================================
    % === Declare the operators of interest

    % Declare system as PDE.
    x = pde_var('state',1,s,[0,1]);
    PDE = [diff(x,t)==diff(x,s,2)+lam*x;
                         subs(x,s,0)==0;
                         subs(x,s,1)==0];

    % Convert to PIE.
    PIE = convert(PDE,'pie');
    T = PIE.T;      A = PIE.A;

    % ================================================
    % === Declare the LPI

    % % Initialize LPI program
    prog = lpiprogram(s,[0,1]);

    % % Declare decision variables:
    % %   P:L2-->L2,    P>0
    [prog,P] = poslpivar(prog,T.dim);
    P = P + 1e-4;                   % enforce P>=1e-4

    % % Set inequality constraints:
    % %   A'*P*T + T'*P*A <= 0
    Q = A'*P*T + T'*P*A;
```

```
    opts.psatz = 1;                 % allow Q>=0 outside domain
    prog = lpi_ineq(prog,-Q,opts);

    % % Solve and retrieve the solution
    solve_opts.solver = 'sedumi';   % use SeDuMi to solve
    solve_opts.params.fid = 0;      % suppress output in command window
    prog = lpisolve(prog,solve_opts);

    % % Alternatively, uncomment to run pre-defined stability executive
    % prog = lpiscript(PIE,'stability',settings);

    % Check if the system is stable
    is_pinf = prog.solinfo.info.pinf;       % is primal feasible?
    is_dinf = prog.solinfo.info.dinf;       % is dual feasible?
    feasrat = prog.solinfo.info.feasratio;  % ratio should be close to 1
    if is_dinf || is_pinf || abs(feasrat-1)>0.1
        % Stability cannot be verified --> decrease value of lam...
        lam_max = lam;
        lam = 0.5*(lam_min + lam_max);
    else
        % The system is stable --> try larger value of lam...
        lam_min = lam;
        lam = 0.5*(lam_min + lam_max);
    end
end
```

This code can also be called using "DEMO4_stability_parameter_bisection". Running this demo, we find that stability can be verified whenever $\lambda \leq 9.8438$, approaching the analytic bound of $\lambda = \pi^2 \approx 9.8696$

11.5 DEMO 5: Constructing and Simulating an Optimal Estimator

We consider a reaction-diffusion PDE, with an observed output y,

$$\begin{aligned} \dot{\mathbf{x}}(t,s) &= \partial_s^2 \mathbf{x}(t,s) + 4\mathbf{x}(t,s) + w(t), && s \in [0,1], \\ \text{with BCs} \qquad 0 &= \mathbf{x}(t,0) = \partial_s \mathbf{x}(t,1), \\ \text{and outputs} \qquad z(t) &= \int_0^1 \mathbf{x}(t,s)ds + w(t), \\ y(t) &= \mathbf{x}(t,1). \end{aligned} \tag{11.3}$$

This PDE can be easily declared in PIETOOLS as

```
% Declare independent variables (time and space)
pvar t s
% Declare state, input, and output variables
x = pde_var(1,s,[0,1]);   w = pde_var('in');
z = pde_var('out');       y = pde_var('sense');
% Declare the sytem equations
```

```
lam = 4;
PDE = [diff(x,t) == diff(x,s,2) + lam*x + w;  % PDE
       z == int(x,s,[0,1]) + w;               % regulated output
       y == subs(x,s,1);                      % observed output
       subs(x,s,0) == 0;                      % first boundary condition
       subs(diff(x,s),s,1) == 0];             % second boundary condition
display_PDE(PDE);
```

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 (11.3) can be equivalently represented by a PIE

$$\begin{aligned}\partial_t\Big(\mathcal{T}\mathbf{x}_{\mathrm{f}}\Big)(t,s) &= \Big(\mathcal{A}\mathbf{x}_{\mathrm{f}}\Big)(t,s) + \Big(\mathcal{B}_1 w\Big)(t,s), \qquad s\in[0,1]\\ z(t) &= \Big(\mathcal{C}_1\mathbf{x}_{\mathrm{f}}\Big)(t) + \Big(\mathcal{D}_{11}w\Big)(t),\\ y(t) &= \Big(\mathcal{C}_2\mathbf{x}_{\mathrm{f}}\Big)(t) + \Big(\mathcal{D}_{12}w\Big)(t),\end{aligned} \tag{11.4}$$

where we define $\mathbf{x}_{\mathrm{f}} := \partial_s^2\mathbf{x}$, and where $\mathbf{x} = \mathcal{T}\mathbf{x}_{\mathrm{f}}$.

We consider the problem of designing an optimal estimator for the PIE (11.4). In particular, we construct an estimator of the form

$$\begin{aligned}\partial_t(\mathcal{T}\hat{\mathbf{x}}_{\mathrm{f}})(t) &= \mathcal{A}\hat{\mathbf{x}}_{\mathrm{f}}(t) + \mathcal{L}\Big(y(t)-\hat{y}(t)\Big),\\ \hat{z}(t) &= \mathcal{C}_1\hat{\mathbf{x}}_{\mathrm{f}}(t),\\ \hat{y}(t) &= \mathcal{C}_2\hat{\mathbf{x}}_{\mathrm{f}}(t),\end{aligned} \tag{11.5}$$

so that the error $\mathbf{e}(t,s) := \hat{\mathbf{x}}_{\mathrm{f}}(t,s) - \mathbf{x}_{\mathrm{f}}(t,s)$ in the state and $\tilde{z}(t) := \hat{z}(t) - z(t)$ in the output satisfy

$$\begin{aligned}\partial_t(\mathcal{T}\mathbf{e})(t) &= (\mathcal{A}-\mathcal{L}\mathcal{C}_2)\mathbf{e}(t) - (\mathcal{B}_1+\mathcal{L}\mathcal{D}_{21})w(t)\\ \tilde{z}(t) &= \mathcal{C}_1\mathbf{e}(t) - \mathcal{D}_{11}w(t).\end{aligned} \tag{11.6}$$

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. To construct such an operator, we solve the LPI,

$$\begin{aligned}&\min_{\gamma,\mathcal{P},\mathcal{Z}} \quad \gamma\\ &\mathcal{P}\succ 0, \qquad Q := \begin{bmatrix}-\gamma I & -\mathcal{D}_{11}^\top & -(\mathcal{P}\mathcal{B}_1+\mathcal{Z}\mathcal{D}_{21})^*\mathcal{T}\\ (\cdot)^* & -\gamma I & \mathcal{C}_1\\ (\cdot)^* & (\cdot)^* & (\mathcal{P}\mathcal{A}+\mathcal{Z}\mathcal{C}_2)^*\mathcal{T}+(\cdot)^*\end{bmatrix}\preccurlyeq 0\end{aligned} \tag{11.7}$$

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 $\|\hat{z}-z\|\le\gamma\|w\|$. For more details, see Section (13.2). This LPI can be implemented in PIETOOLS as

```
% % Initialize LPI program
prog = lpiprogram(s,[0,1]);

% % Declare decision variables:
% %   gam \in \R,     P:L2-->L2,    Z:\R-->L2
% Scalar decision variable
[prog,gam] = lpidecvar(prog,'gam');
% Positive operator variable P>=0
opts.sep = 1;                    % set P.R.R1=P.R.R2
[prog,P] = poslpivar(prog,T.dim,4,opts);
% Indefinite operator variable Z: \R-->L2
[prog,Z] = lpivar(prog,[0,1;1,0],4);

% % Set inequality constraints:
% %   Q <= 0
Q = [-gam,              -D11',        -(P*B1+Z*D21)'*T;
     -D11,              -gam,         C1;
     -T'*(P*B1+Z*D21),  C1',          (P*A+Z*C2)'*T+T'*(P*A+Z*C2)];
prog = lpi_ineq(prog,-Q);

% % Set objective function:
% %   min gam
prog = lpisetobj(prog, gam);

% % Solve and retrieve the solution
prog_sol = lpisolve(prog);
% Extract solved value of decision variables
gam_val = lpigetsol(prog_sol,gam);
Pval = lpigetsol(prog_sol,P);
Zval = lpigetsol(prog_sol,Z);
% Build optimal observer operator L
Lval = getObserver(Pval,Zval);
```

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 (11.5), obtaining a PIE

$$\partial_t \left(\begin{bmatrix} \mathcal{T} & 0 \\ 0 & \mathcal{T} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{\mathrm{f}} \\ \hat{\mathbf{x}}_{\mathrm{f}} \end{bmatrix} \right)(t,s) = \left(\begin{bmatrix} \mathcal{A} & 0 \\ -\mathcal{L}\mathcal{C}_2 & \mathcal{A}+\mathcal{L}\mathcal{C}_2 \end{bmatrix} \begin{bmatrix} \mathbf{x}_{\mathrm{f}} \\ \hat{\mathbf{x}}_{\mathrm{f}} \end{bmatrix} \right)(t,s) + \left(\begin{bmatrix} \mathcal{B}_1 \\ \mathcal{L}\mathcal{D}_{21} \end{bmatrix} w \right)(t)$$

$$\begin{bmatrix} z \\ \hat{z} \end{bmatrix}(t) = \left(\begin{bmatrix} \mathcal{C}_1 & 0 \\ 0 & \mathcal{C}_1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_{\mathrm{f}} \\ \hat{\mathbf{x}}_{\mathrm{f}} \end{bmatrix} \right)(t) + \left(\begin{bmatrix} \mathcal{D}_{11} \\ 0 \end{bmatrix} w \right)(t)$$

We can construct this PIE in PIETOOLS by first declaring the estimator dynamics with input signal y using `piess`, and then taking the linear fractional transformation of the original PIE with this estimator PIE, as

```
PIE_est = piess(T,A+Lval*C2,-Lval,C1(1,:));
PIE_CL = pielft(PIE,PIE_est);
```

Alternatively, this system could also be constructed using the function `closedLoopPIE` as

```
PIE_CL = closedLoopPIE(PIE,Lval,'observer');
```

returning the closed-loop system associated to the original PIE as in (11.4) defined by `PIE`, and the observer as in (11.5) with operator $\mathcal{L}$ defined by `Lval`. Then, we can use PIESIM to simulate the evolution of the PDE state $\mathbf{x}(t) = \big(\mathcal{T}\mathbf{x}_{\mathrm{f}}\big)(t)$ and its estimate $\hat{\mathbf{x}}(t) = \big(\mathcal{T}\hat{\mathbf{x}}_{\mathrm{f}}\big)(t)$ associated to the system defined by `PIE_CL`. For this, we first declare initial conditions $\left[\begin{smallmatrix}\mathbf{x}_{\mathrm{f}}(0,s)\\ \hat{\mathbf{x}}_{\mathrm{f}}(0,s)\end{smallmatrix}\right]$ and values of the disturbance $w(t)$ as

```
% % Declare initial values and disturbance
syms st sx real
uinput.ic.PDE = [-10*sx;    % actual initial PIE state value
                  0];       % estimated initial PIE state value
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 $\left[\begin{smallmatrix}\mathbf{x}_{\mathrm{f}}(0,s)\\ \hat{\mathbf{x}}_{\mathrm{f}}(0,s)\end{smallmatrix}\right]$, not to those of the PDE state. Here, we let the initial PIE state $\mathbf{x}_{\mathrm{f}}(0)$ be parabolic, and start with an estimate of this state that is just $\hat{\mathbf{x}}_{\mathrm{f}}(0) = 0$. Given these initial conditions, we then simulate the evolution of the PDE state $\left[\begin{smallmatrix}\mathbf{x}\\ \hat{\mathbf{x}}\end{smallmatrix}\right] = \left[\begin{smallmatrix}\mathcal{T} & 0\\ 0 & \mathcal{T}\end{smallmatrix}\right]\left[\begin{smallmatrix}\mathbf{x}_{\mathrm{f}}\\ \hat{\mathbf{x}}_{\mathrm{f}}\end{smallmatrix}\right]$ using PIESIM as

```
% % Set options for discretization and simulation
opts.plot = 'yes';  % plot the solution
opts.N = 8;         % expand using 8 Chebyshev polynomials
opts.tf = 2;        % simulate up to t = 2
opts.dt = 1e-3;     % use time step of 10^-3
ndiff = [0,0,2];    % PDE state involves 2 second order differentiable state variables

% % Simulate solution to the PIE with estimator.
[solution,grid] = PIESIM(PIE_CL,opts,uinput,ndiff);
% % Extract actual and estimated state and output at each time step.
tval = solution.timedep.dtime;
x_act = reshape(solution.timedep.pde(:,1,:),opts.N+1,[]);
x_est = reshape(solution.timedep.pde(:,2,:),opts.N+1,[]);
z_act = solution.timedep.regulated(1,:);
z_est = solution.timedep.regulated(2,:);
```

Here, we expand the solution using 8 Chebyshev polynomials, and we simulate up to `opts.tf=2`, 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 `x_eta-x_act` in estimated state, we obtain a graph as in Figure 11.3.

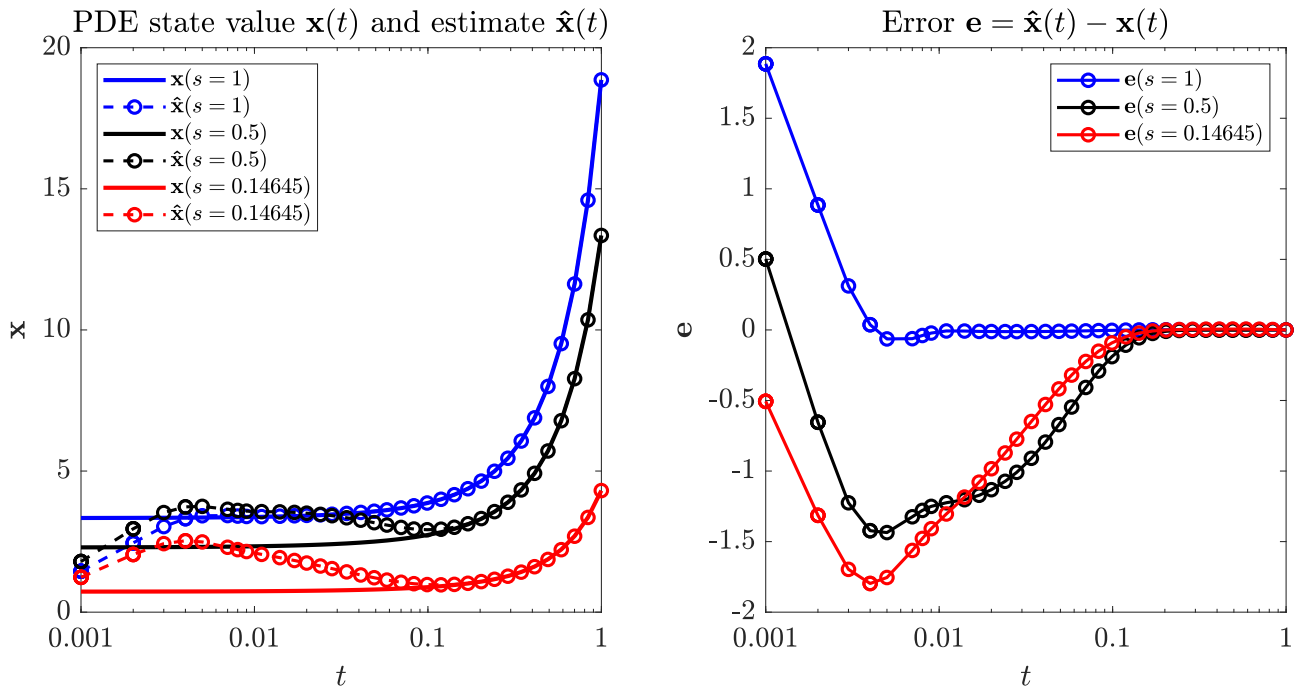


Figure 11.3: Simulated value of PDE state $\mathbf{x}(t,s)$ and estimated state $\hat{\mathbf{x}}(t,s)$, along with the error $\mathbf{e}=\hat{\mathbf{x}}(t,s)-\mathbf{x}(t,s)$ associated to the PDE (11.3) at several grid points $s\in[0,1]$, using the Estimator (11.5) with operator $\mathcal{L}$ computed by solving the H_∞-optimal estimator LPI. See the file "PIETOOLS_Code_Illustrations_Ch11_Demos" for the code to achieve these plots.

As Figure 11.3 shows, the value of the estimate state $\hat{\mathbf{x}}$ at each of the grid points quickly converges to that 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 (11.3) 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 "DEMO5_Hinf_Optimal_Estimator".

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

$$\begin{aligned}
&\dot{\mathbf{x}}(t,s)=\partial_s^2\mathbf{x}(t,s)+4\mathbf{x}(t,s)+sw(t)+su(t), && s\in[0,1],\\
\text{with BCs}\quad &0=\mathbf{x}(t,0)=\partial_s\mathbf{x}(t,1),\\
\text{and outputs}\quad &z(t)=\begin{bmatrix}\int_0^1\mathbf{x}(t,s)ds+w(t)\\ u(t)\end{bmatrix}.
\end{aligned}\tag{11.8}$$

This PDE can be easily declared in PIETOOLS as

```
% Declare independent variables (time and space)
pvar t s
% Declare state, input, and output variables
x = pde_var('state',1,s,[0,1]);
z = pde_var('output',2);
w = pde_var('input',1);          u = pde_var('control',1);
% Declare the sytem equations
lam = 5;
PDE = [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];
```

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');
T = PIE.T;
A = PIE.A;      C1 = PIE.C1;    B2 = PIE.B2;
B1 = PIE.B1;    D11 = PIE.D11;  D12 = PIE.D12;
```

Then, the PDE (11.8) can be equivalently represented by a PIE

$$\begin{aligned}\partial_t\big(\mathcal{T}\dot{\mathbf{x}}_{\mathrm{f}}\big)(t,s) &= \big(\mathcal{A}\mathbf{x}_{\mathrm{f}}\big)(t,s) + \big(\mathcal{B}_1 w\big)(t,s) + \big(\mathcal{B}_2 u\big)(t,s), \qquad s\in[0,1]\\ z(t) &= \big(\mathcal{C}_1\mathbf{x}_{\mathrm{f}}\big)(t) + \big(\mathcal{D}_{11}w\big)(t) + \big(\mathcal{D}_{12}u\big)(t),\end{aligned} \tag{11.9}$$

where we define $\mathbf{x}_{\mathrm{f}} := \partial_s^2\mathbf{x}$ and $\mathbf{x} = \mathcal{T}\mathbf{x}_{\mathrm{f}}$.

We now attempt to find a state-feedback H_∞-optimal controller for the PIE (11.9). In particular, we use $u(t) = \mathcal{K}\mathbf{x}_{\mathrm{f}}(t)$, where $\mathcal{K} : L_2 \to \mathbb{R}$ is a PI operator of the form

$$\mathcal{K}\mathbf{x}_{\mathrm{f}} = \int_a^b K(s)\mathbf{x}_{\mathrm{f}}(s)ds.$$

Then we get the closed-loop system

$$\begin{aligned}\partial_t(\mathcal{T}\mathbf{x}_{\mathrm{f}})(t) &= (\mathcal{A}+\mathcal{B}_2\mathcal{K})\mathbf{x}_{\mathrm{f}}(t) + \mathcal{B}_1 w(t),\\ z(t) &= (\mathcal{C}_1+\mathcal{D}_{12}\mathcal{K})\mathbf{x}_{\mathrm{f}}(t) + \mathcal{D}_{11}w(t).\end{aligned} \tag{11.10}$$

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. For more details, see Section (13.3). To find such an operator, we solve the LPI,

$$\begin{aligned}\min_{\gamma,\mathcal{P},\mathcal{Z}} \quad & \gamma \\ s.t. \quad & \mathcal{P}\succ 0, \quad \begin{bmatrix} -\gamma I & \mathcal{D}_{11} & \mathcal{T}(\mathcal{P}\mathcal{C}_1+\mathcal{Z}\mathcal{D}_{12}) \\ (\cdot)^* & -\gamma I & \mathcal{B}_1^* \\ (\cdot)^* & (\cdot)^* & \mathcal{T}(\mathcal{A}\mathcal{P}+\mathcal{B}_2\mathcal{Z})^* + (\mathcal{A}\mathcal{P}+\mathcal{B}_2\mathcal{Z})\mathcal{T}^* \end{bmatrix} \preccurlyeq 0,\end{aligned} \tag{11.11}$$

so that, for any solution $(\gamma,\mathcal{P},\mathcal{Z})$ to this problem, letting $\mathcal{K} := \mathcal{Z}\mathcal{P}^{-1}$, we have $\|z\| \le \gamma\|w\|$. We can declare and solve this LPI in PIETOOLS as follows:

```
% % Initialize LPI program
prog = lpiprogram(s,[0,1]);

% % Declare decision variables:
% %   gam \in \R,     P:L2-->L2,    Z:\R-->L2
% Scalar decision variable
[prog,gam] = lpidecvar(prog,'gam');
% Positive operator variable P>=0
[prog,P] = poslpivar(prog,[0,0;1,1],4);
% Enforce strict positivity P >= 1e-3
P = P + 1e-3;
% Indefinite operator variable Z
[prog,Z] = lpivar(prog,[1,0;0,1],2);

% % Set inequality constraints:
% %   Q <= 0
Q = [-gam*eye(2),       D11,    (C1*P+D12*Z)*(T');
     D11',              -gam,   B1';
     T*(C1*P+D12*Z)',   B1,     (A*P+B2*Z)*(T')+T*(A*P+B2*Z)'];
prog = lpi_ineq(prog,-Q);

% % Set objective function:
% %   min gam
prog = lpisetobj(prog, gam);

% % Solve and retrieve the solution
prog_sol = lpisolve(prog);
% Extract solved value of decision variables
gam_val = lpigetsol(prog_sol,gam);
Pval = lpigetsol(prog_sol,P);
Zval = lpigetsol(prog_sol,Z);
% Build the optimal control operator K.
Kval = getController(Pval,Zval,1e-3);
```

Alternatively, this LPI could also be solved using the executive `PIETOOLS_Hinf_control`, as e.g.

```
settings = lpisettings('heavy');
[prog, Kval, gam_val] = PIETOOLS_Hinf_control(PIE, settings);
```

Using either option, we obtain an `opvar` class object `Kval` representing a value 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 `piess` as

```
PIE_CL = piess(T,A+B2*Kval,B1,C1+D12*Kval,D11);
```

or, equivalently, using the function `closedLoop_PIE` as

```
    PIE_CL = closedLoopPIE(PIE,Kval);
```

Having constructed the closed-loop PIE representation, we can use PIESIM to simulate the evolution of the PDE state $\mathbf{x}(t) = \big(\mathcal{T}\mathbf{x}_{\mathrm{f}}\big)(t)$ associated to the system defined by `PIE_CL`. In this

case, we simulate with an initial condition $\mathbf{x}(0,s) = \frac{4}{\pi^2}\sin(\frac{\pi}{2}s)$, and disturbance $w(t) = \frac{\sin(\pi t)}{t+\epsilon}$, where we introduce $\epsilon = 2.2204 \cdot 10^{-16}$ to avoid a singularity at $t = 0$. Note that, since we are simulating a PIE, we must also specify the initial value of the PIE state $\mathbf{x}_\mathrm{f}(t) = \partial_s^2\mathbf{x}(t)$, yielding $\mathbf{x}_\mathrm{f}(0,s) = \sin(\frac{\pi}{2}s)$. We declare these values as shown below

```
% % Declare initial values and disturbance
syms st sx real
uinput.ic.PDE = sin(sx*pi/2);
uinput.w = sin(pi*st)./(st+eps);
```

Next, we set the parameters related to the numerical scheme and simulation as shown below

```
% % Set options for discretization and simulation
opts.plot = 'yes';  % plot the solution
opts.N = 16;        % expand using 16 Chebyshev polynomials
opts.tf = 2;        % simulate up to t = 2
opts.dt = 1e-2;     % use time step of 10^-2
ndiff = [0,0,1];    % PDE state involves 1 2nd-order differentiable state
```

In this case, the state involves only a single 2nd-order differentiable variable $\mathbf{x}(t,s)$, and we expand this state using 16 Chebyshev polynomials in s. We use these settings to simulate the open- (without controller) and closed-loop (with controller) PIEs as

```
% Simulate uncontrolled PIE and extract solution
[solution_OL,grid] = PIESIM(PIE,opts,uinput,ndiff);
tval = solution_OL.timedep.dtime;
x_OL = reshape(solution_OL.timedep.pde(:,1,:),opts.N+1,[]);
z_OL = solution_OL.timedep.regulated(1,:);
% Simulate controlled PIE and extract solution
[solution_CL,~] = PIESIM(PIE_CL,opts,uinput,ndiff);
x_CL = reshape(solution_CL.timedep.pde(:,1,:),opts.N+1,[]);
z_CL = solution_CL.timedep.regulated(1,:);
u_CL = solution_CL.timedep.regulated(2,:);
w = double(subs(uinput.w,st,tval));
```

The resulting values of the PDE state in the open- and closed-loop setting are plotted in Fig. 11.4, at several time instances. The corresponding evolution of the regulated output is displayed in Fig. 11.5. The figures show that, as expected, the state and regulated output blow up if no control is exerted, as the PDE (11.8) is unstable. However, using the control $u(t) = \mathcal{K}\mathbf{x}_\mathrm{f}(t)$ with gain $\mathcal{K}$ computed by solving the LPI (11.11), the PDE state converges to zero within 1 second – despite the presence of a disturbance $w(t)$ – and the regulated output $z(t) = \int_0^1 \mathbf{x}(t,s)ds + w(t)$ becomes driven entirely by the disturbance $w(t)$.

The full code for designing the optimal controller and simulating the PDE state has been included in PIETOOLS as the demo file "DEMO6_Hinf_optimal_control".

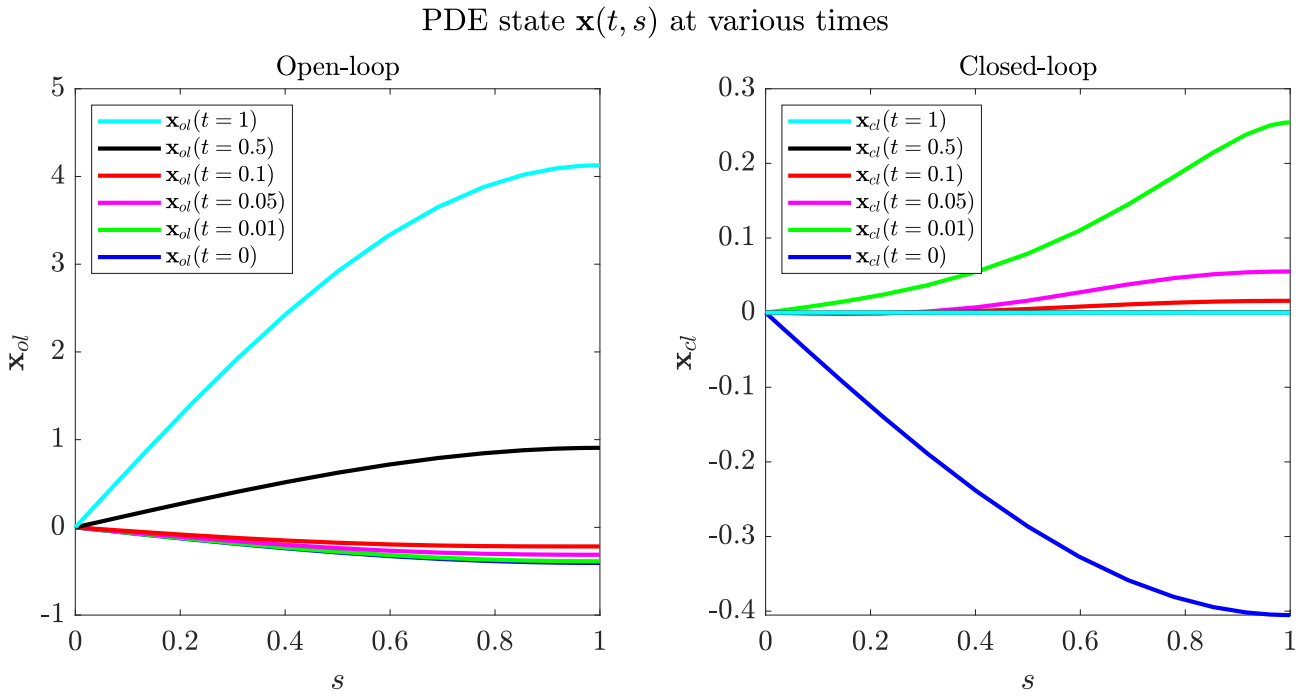


Figure 11.4: Simulated value of PDE state $\mathbf{x}(t,s)$ solution to (11.8) at several times $t \in [0,1]$, both without (left) and with (right) feedback $u(t) = (\mathcal{K}\mathbf{x}_{\mathrm{f}})(t)$, starting with $\mathbf{x}(0,s) = -\frac{4}{\pi^2}\sin(\frac{\pi}{2}s)$ and with disturbance $w(t) = \frac{\sin(\pi t)}{t+\epsilon}$.

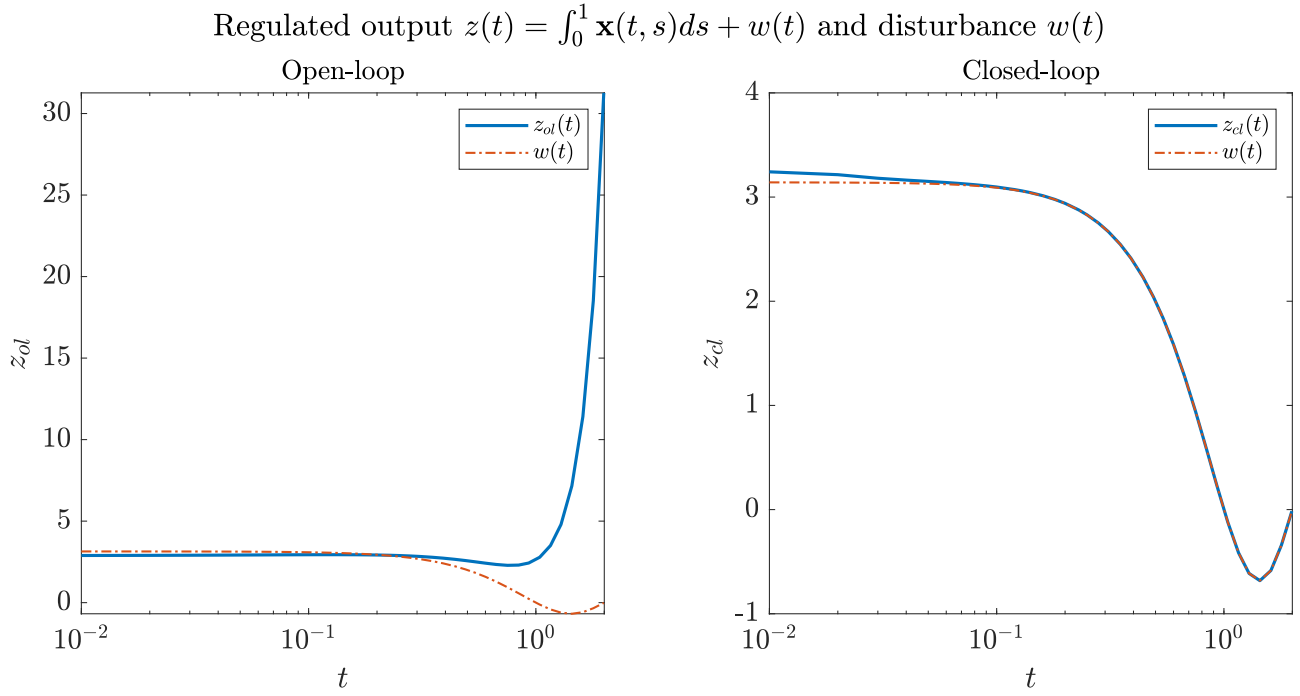


Figure 11.5: Simulated evolution of regulated output $z(t)$ to (11.8), both without (left) and with (right) feedback $u(t) = (\mathcal{K}\mathbf{x}_{\mathrm{f}})(t)$, starting with $\mathbf{x}(0,s) = -\frac{4}{\pi^2}\sin(\frac{\pi}{2}s)$ and with disturbance $w(t) = \frac{\sin(\pi t)}{t+\epsilon}$.

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 $u = \mathcal{K}\hat{\mathbf{x}}_{\mathrm{f}}$, where $\mathcal{K}$ is the control gains obtained in 11.6 and $\hat{\mathbf{x}}_{\mathrm{f}}$ is the PIE 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

$$\begin{aligned} \dot{\mathbf{x}}(t,s) &= \partial_s^2 \mathbf{x}(t,s) + 4\mathbf{x}(t,s) + sw(t) + su(t), && s \in [0,1], \\ \text{with BCs} \qquad 0 &= \mathbf{x}(t,0) = \partial_s \mathbf{x}(t,1), \\ \text{and outputs} \qquad z(t) &= \begin{bmatrix} \int_0^1 \mathbf{x}(t,s)ds \\ u(t) \end{bmatrix} \\ y(t) &= \mathbf{x}(t,1). \end{aligned} \tag{11.12}$$

Recall that this PDE can be defined using the code

```
% Declare independent variables (time and space)
pvar t s
% Declare state, input, and output variables
x = pde_var('state',1,s,[0,1]);
w = pde_var('input',1);      u = pde_var('control',1);
z = pde_var('output',2);     y = pde_var('sense',1);
% Declare the sytem equations
lam = 5;
PDE = [diff(x,t) == diff(x,s,2) + lam*x + s*w + s*u;
       z == [int(x,s,[0,1]); u];
       y == subs(x,s,1);
       subs(x,s,0)==0;
       subs(diff(x,s),s,1)==0];
```

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');
T = PIE.T;
A = PIE.A;      B1 = PIE.B1;    B2 = PIE.B2;
C1 = PIE.C1;    D11 = PIE.D11;  D12 = PIE.D12;
C2 = PIE.C2;    D21 = PIE.D21;  D22 = PIE.D22;
```

A detailed presentation on how to manually perform optimal observer and controller synthesis for PIE systems is already given in Section 11.5 and Section 11.6 respectively, and therefore, will not be repeated here. Instead, we will simply compute the optimal observer gain $\mathcal{L}$ and controller gain $\mathcal{K}$ using the respective executive functions, as

```
settings = lpisettings('heavy');
[prog_k, Kval, gam_co_val] = PIETOOLS_Hinf_control(PIE, settings);
[prog_l, Lval, gam_ob_val] = PIETOOLS_Hinf_estimator(PIE, settings);
```

Now, recall from Equations (11.5) and (11.9) that the combined system is given by

$$\begin{aligned}
\partial_t\Big(\mathcal{T}\mathbf{x}_{\mathrm{f}}\Big)(t) &= \Big(\mathcal{A}\mathbf{x}_{\mathrm{f}}\Big)(t) + \Big(\mathcal{B}_1 w\Big)(t) + \Big(\mathcal{B}_2 u\Big)(t),\\
z(t) &= \Big(\mathcal{C}_1\mathbf{x}_{\mathrm{f}}\Big)(t) + \Big(\mathcal{D}_{11}w\Big)(t) + \Big(\mathcal{D}_{12}u\Big)(t),\\
\partial_t(\mathcal{T}\hat{\mathbf{x}}_{\mathrm{f}})(t) &= \mathcal{A}\hat{\mathbf{x}}_{\mathrm{f}}(t) + \mathcal{L}\Big(\mathcal{C}_2\mathbf{x}_{\mathrm{f}}(t) - \mathcal{C}_2\hat{\mathbf{x}}_{\mathrm{f}}(t)\Big),\\
\hat{z}(t) &= \mathcal{C}_1\hat{\mathbf{x}}_{\mathrm{f}}(t).
\end{aligned} \tag{11.13}$$

Using the observer-controller coupling, $u = \mathcal{K}\hat{\mathbf{x}}_{\mathrm{f}}$, we get the closed-loop PIE

$$\begin{aligned}
\partial_t\left(\begin{bmatrix}\mathcal{T} & 0\\ 0 & \mathcal{T}\end{bmatrix}\begin{bmatrix}\mathbf{x}_{\mathrm{f}}\\ \mathbf{x}_{\mathrm{f}}\end{bmatrix}\right)(t,s) &= \left(\begin{bmatrix}\mathcal{A} & \mathcal{B}_2\mathcal{K}\\ -\mathcal{L}\mathcal{C}_2 & \mathcal{A}+\mathcal{L}\mathcal{C}_2\end{bmatrix}\begin{bmatrix}\mathbf{x}_{\mathrm{f}}\\ \mathbf{x}_{\mathrm{f}}\end{bmatrix}\right)(t,s) + \left(\begin{bmatrix}\mathcal{B}_1\\ \mathcal{L}\mathcal{D}_{21}\end{bmatrix}w\right)(t)\\
\begin{bmatrix}z\\ \hat{z}\end{bmatrix}(t) &= \left(\begin{bmatrix}\mathcal{C}_1 & \mathcal{D}_{12}\mathcal{K}\\ 0 & \mathcal{C}_1\end{bmatrix}\begin{bmatrix}\mathbf{x}_{\mathrm{f}}\\ \hat{\mathbf{x}}_{\mathrm{f}}\end{bmatrix}\right)(t) + \left(\begin{bmatrix}\mathcal{D}_{11}\\ 0\end{bmatrix}w\right)(t).
\end{aligned} \tag{11.14}$$

We can construct the above closed-loop system using the code

```
PIE_est = piess(T,A+Lval*C2,-Lval,{C1(1,:);Kval});
PIE_CL = pielft(PIE,PIE_est);
```

first constructing a PIE for the estimated state $\hat{\mathbf{x}}_{\mathrm{f}}(t,s)$, with input $y(t)$ and output $u(t) = (\mathcal{K}\hat{\mathbf{x}}_{\mathrm{f}})(t)$, and then taking the linear fractional transformation with the original PIE to obtain the closed-loop system.

Once, we have the closed-loop PIE object, we can use `PIESIM` to simulate the system for some initial conditions and disturbances. In this case, we consider the disturbance $w(t) = 10e^{-t}$ and initial state $\mathbf{x}(0,s) = -\frac{5}{3}s^3 + 5s$, so that the initial fundamental state is $\mathbf{x}_{\mathrm{f}}(0,s) = -10s$. For the state estimate $\hat{\mathbf{x}}_{\mathrm{f}}(t,s)$, we assume to start with no information, setting $\hat{\mathbf{x}}_{\mathrm{f}}(0,s) = 0$. We simulate the open-loop (without control or observer) and closed-loop systems for these values of the initial state and disturbance using the code

```
% % Declare initial values and disturbance
syms st sx real
uinput.ic.PDE = [-10*sx; 0];
uinput.w = 10*exp(-st);

% % Set options for discretization and simulation
opts.plot = 'yes';  % plot the solution
opts.N = 8;         % Expand using 8 Chebyshev polynomials
opts.tf = 2;        % Simulate up to t = 2
opts.dt = 1e-2;     % Use time step of 10^-2
ndiff = [0,0,1];    % The state involves 1 state variable
ndiff_CL = [0,0,2]; % The closed-loop system involves 2 state variables

% % Perform the actual simulation
% Simulate uncontrolled PIE and extract solution
[solution_OL,grid] = PIESIM(PIE,opts,uinput,ndiff);
tval = solution_OL.timedep.dtime;
```

```
x_OL = reshape(solution_OL.timedep.pde(:,1,:),opts.N+1,[]);
z_OL = solution_OL.timedep.regulated(1,:);
% Simulate controlled PIE and extract solution
[solution_CL,~] = PIESIM(PIE_CL,opts,uinput,ndiff_CL);
x_CL = reshape(solution_CL.timedep.pde(:,1,:),opts.N+1,[]);
xhat_CL = reshape(solution_CL.timedep.pde(:,2,:),opts.N+1,[]);
z_CL = solution_CL.timedep.regulated(1,:);
zhat_CL = solution_CL.timedep.regulated(3,:);
u_CL = solution_CL.timedep.regulated(2,:);
w = double(subs(uinput.w,st,tval));
```

Note here that the field `ndiff` for the closed-loop system is set to `[0,0,2]` (rather than `[0,0,1]`), as the closed-loop PIE system has two 2nd-order differentiable PDE states (PDE state and observer state).

The simulated evolutions of the PDE state and regulated output are plotted in Fig. 11.6 and Fig. 11.7, respectively, both for the open- and closed-loop systems. The figures show that the observer-based controller indeed stabilizes the system, with the PDE state and regulated output of the closed-loop system both converging to zero. The control effort $u(t) = \mathcal{K}\hat{\mathbf{x}}_{\mathrm{f}}(t)$ used to achieve this stabilization is displayed in Fig. 11.8. Although this control effort is initially quite large, the effort quickly decays as the PDE state converges to the equilibrium.

The full code for designing the optimal controller and simulating the PDE state has been included in PIETOOLS as the demo file "DEMO7_observer_based_control".

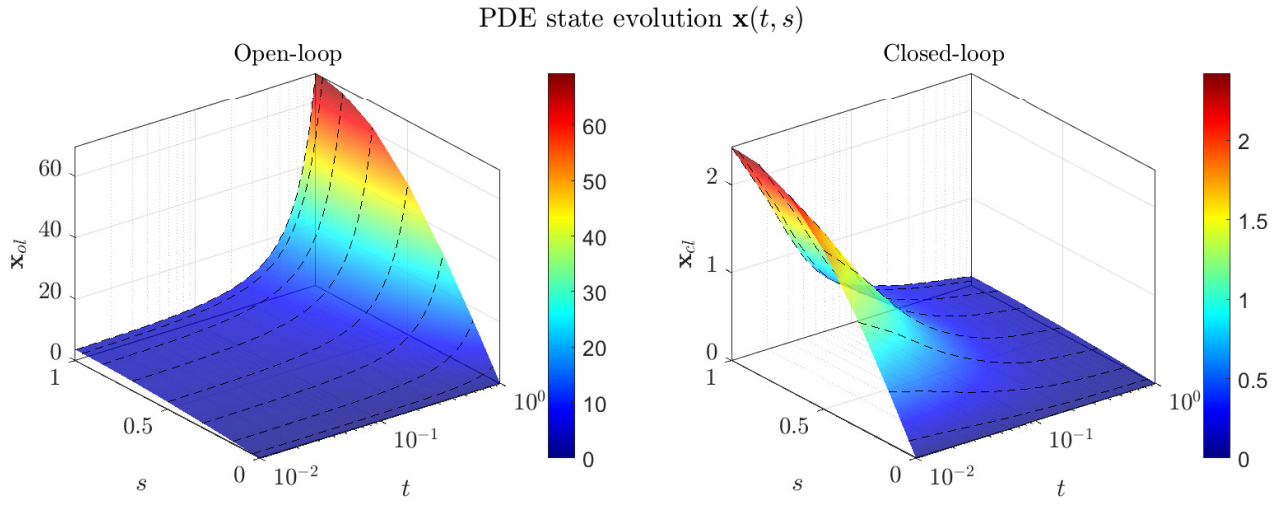


Figure 11.6: Simulated value of PDE state $\mathbf{x}(t,s)$ solution to (11.8) at several times $t \in [0,1]$, without (left) and with (right) the observer-based feedback $u(t) = (\mathcal{K}\hat{\mathbf{x}}_{\mathrm{f}})(t)$, starting with state $\mathbf{x}(0,s) = -\frac{5}{3}s^3 + 5s$ and estimate $\hat{\mathbf{x}}(0,s) = 0$, and with disturbance $w(t) = 10e^{-t}$.

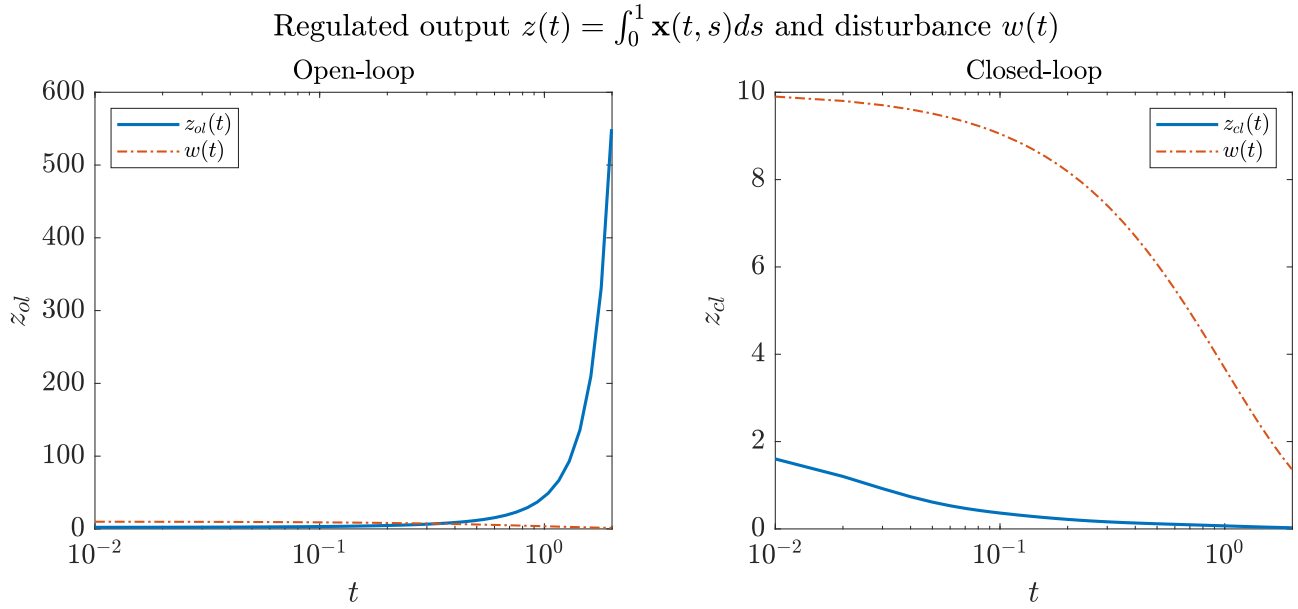


Figure 11.7: Simulated evolution of regulated output $z(t)$ to (11.8), both without (left) and with (right) the observer-based feedback $u(t) = (\mathcal{K}\hat{\mathbf{x}}_{\mathrm{f}})(t)$, starting with state $\mathbf{x}(0,s) = -\frac{5}{3}s^3 + 5s$ and estimate $\hat{\mathbf{x}}(0,s) = 0$, and with disturbance $w(t) = 10e^{-t}$.

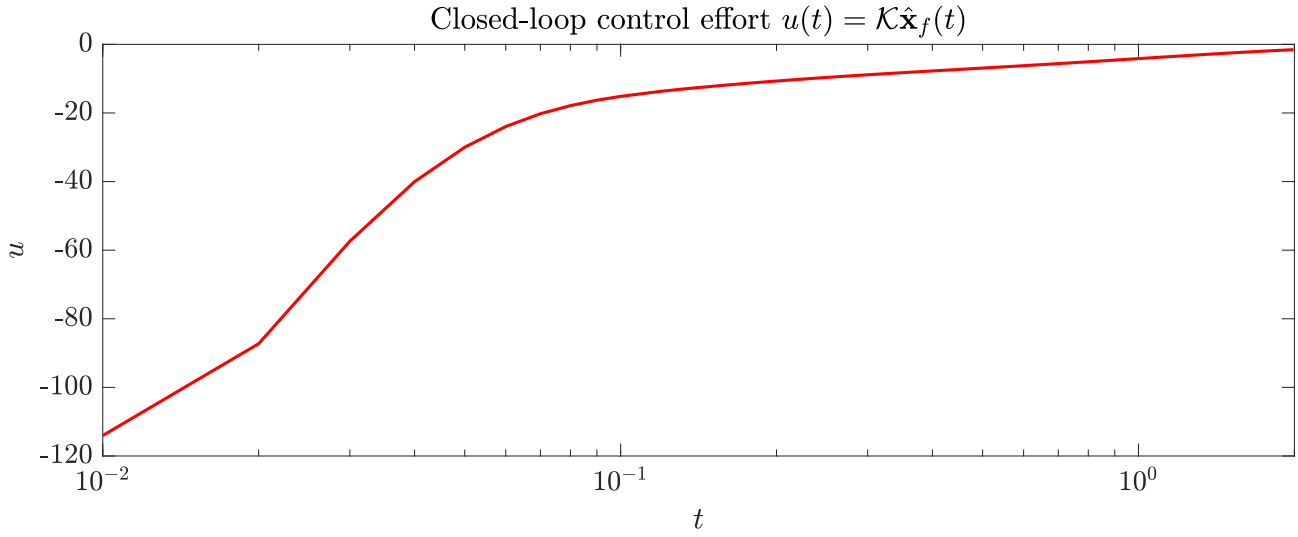


Figure 11.8: Control effort $u(t) = (\mathcal{K}\hat{\mathbf{x}}_{\mathrm{f}})(t)$ for (11.8), computed by solving the LPIE (11.11), and simulated with initial state $\mathbf{x}(0,s) = -\frac{5}{3}s^3+5s$, initial estimate $\hat{\mathbf{x}}(0,s) = 0$, and disturbance $w(t) = 10e^{-t}$.

11.8 DEMO 8: H_2-Norm Analysis of PDEs

Consider the following 1D pure-convection equation with homogeneous Dirichlet boundary condition:

$$\partial_t \mathbf{x}(t,s) = \partial_s \mathbf{x}(t,s) + (s - s^2)w(t) \quad s \in [0,1], t \in \mathbb{R}_+ \tag{11.15}$$

$$z(t) = \int_0^1 \mathbf{x}(t,s)ds,$$
$$\mathbf{x}(t,1) = 0. \tag{11.16}$$

We want to compute the H_2-norm of this system, as defined in Sec. 13.1.7. To compute the H_2 norm, the following LPI may be used. First, we declare the PDE in PIETOOLS as

```
% % Declare system as PDE
% Declare independent variables (time and space)
pvar s t
% Declare state, input, and output variables
x = pde_var('state',1,s,[0,1]);
w = pde_var('in',1);
z = pde_var('out',1);
% Declare the sytem equations
pde = [diff(x,t,1)==diff(x,s,1)+(s-s^2)*w;    % dynamics
                z==int(x,s,[0,1]);                    % output equation
                subs(x,s,1)==0];;                          % boundary condition
pde=initialize(pde);
```

Then, we convert the system to the equivalent PIE representation as

```
PIE = convert(pde,'pie');
T = PIE.T;      A = PIE.A;
B1 = PIE.B1;    C1 = PIE.C1;
```

$$\begin{aligned} &\min_{\mu,\mathcal{P}} \quad \gamma \\ &\mathcal{P} \succ 0 \\ &trace(\mathcal{C}_1\mathcal{P}\mathcal{C}_1^*) \leq \mu \\ &\mathcal{A}\mathcal{P}\mathcal{T}^* + \mathcal{T}\mathcal{P}\mathcal{A}^* + \mathcal{B}_1\mathcal{B}_1^* \preccurlyeq 0 \end{aligned} \tag{11.17}$$

If feasible for some $\mu \geq 0$ and PI operator $\mathcal{P}$, then $\|\Sigma\|_{H_2} \leq \gamma = \sqrt{\mu}$. Note that this is reduced version of (13.13) in the particular case where $\mathcal{Q} = \mathcal{P}\mathcal{T}^*$, with a coercive operator $\mathcal{P} \succ 0$.

To declare and solve the LPI, the following command-lines are used.

```
% % Initialize LPI program
prog = lpiprogram(s,[0,1]);

% % Declare decision variables:
% %   gam \in \R,     W:L2-->L2,    Z:\R-->L2
% Scalar decision variable
[prog,gam] = lpidecvar(prog,'gam');
% Positive operator variable W>=0 with default polynomial degrees up to 3.
[prog,W] = poslpivar(prog,[0;1]);

% % Set inequality constraints:
% %   A W T* + T W A* + B1 B1* <= 0
% %   gam >= trace(C1 W C1*)
% Operator inequality Dop<=0
```

```
Dop =  A*W*T'+T*W*A'+B1*B1';
prog = lpi_ineq(prog,-Dop);
% Scalar inequality gam >= trace(C1 W C1*)
Aux = C1*W*C1';
traceVal = trace(Aux.P);
prog = lpi_ineq(prog, gam-traceVal);

% % Set objective function:
% %   min gam
prog = lpisetobj(prog, gam);

% % Solve and retrieve the solution
opts.solver = 'sedumi';          % Use SeDuMi to solve the SDP
prog_sol = lpisolve(prog,opts);
% Extract solved value of decision variables
gamd = sqrt(double(lpigetsol(prog_sol,gam)));
Wc = lpigetsol(prog_sol,W);
```

PIETOOLS gives $\gamma = 0.1016$ as output with the default settings used, an upper-bound on the H_2 norm of the system. The same value of 0.1016 can also be obtained by numerical integration of the output $z(t)$ to the non-zero initial condition as defined in Eq. (13.11).

11.9 DEMO 9: L_2-Gain Analysis of (2D) PDEs

Consider the following 2D reaction-diffusion equation with Dirichlet-Neumann boundary conditions:

$$\begin{aligned} \partial_t \mathbf{x}(t,s_1,s_2) &= \partial_{s_1}^2 \mathbf{x}(t,s_1,s_2) + \partial_{s_2}^2 \mathbf{x}(t,s_1,s_2) + \lambda \mathbf{x}(t,s_1,s_2) + \mathbf{w}(t), \quad && s_1,s_2 \in [a,b]\times[c,d], \\ z(t) &= \int_a^b \int_c^d \mathbf{x}(t,s_1,s_2)\, ds_2 ds_1, && t \in \mathbb{R}_+, \\ \mathbf{x}(t,a,s_2) = \mathbf{x}(t,b,s_2) &= 0, \qquad \mathbf{x}(t,s_1,c) = \partial_{s_2}\mathbf{x}(t,s_2,d) = 0. \end{aligned} \tag{11.18}$$

We simulate the output response of the system to a bounded disturbance, for parameter values $a = c = 0$, $b = d = 1$, and $\lambda = 5$. To this end, we first declare the PDE in PIETOOLS as

```
% Declare independent variables (time and space)
pvar s1 s2 t
% Declare state, input, and output variables
a = 0;      b = 1;
c = 0;      d = 1;
x = pde_var('state',1,[s1;s2],[a,b;c,d]);
w = pde_var('in',1);
z = pde_var('out',1);
% Declare the sytem equations
lam = 5;
PDE = [diff(x,t) == diff(x,s1,2) +diff(x,s2,2) + lam*x + w;
       z == int(x,[s1;s2],[a,b;c,d]);
       subs(x,s1,a)==0;
       subs(x,s1,b)==0;
       subs(x,s2,c)==0;
```

```
        subs(diff(x,s2),s2,d)==0];
PDE = initialize(PDE);
```

Next, we declare the disturbance and initial values for simulation. For our purposes, we consider a zero initial state $\mathbf{x}(0, s_1, s_2) = 0$, and a decaying but oscillating disturbance $w(t) = 20\sin(\pi t)e^{-t/2}$, which we declare using symbolic variables as

```
% % Declare initial values and disturbance
syms st sx sy real
uinput.ic.PDE = 0;
uinput.w = 20*sin(pi*st)*exp(-st/2);
```

We simulate the solution up to a time $T = 10$, performing temporal integration using a time step of $\Delta t = 10^{-2}$, and expanding in space using 8×8 Chebyshev polynomials. To perform this simulation, we call PIESIM as

```
opts.plot = 'yes';  % don't plot the final solution
opts.N = 8;         % Expand using 8x8 Chebyshev polynomials
opts.tf = 10;       % Simulate up to t = 10
opts.dt = 1e-2;     % Use time step of 10^-2
[solution,grid] = PIESIM(PDE,opts,uinput,ndiff);
```

A plot of the simulated evolution of the regulated state is shown in Fig. 11.9.

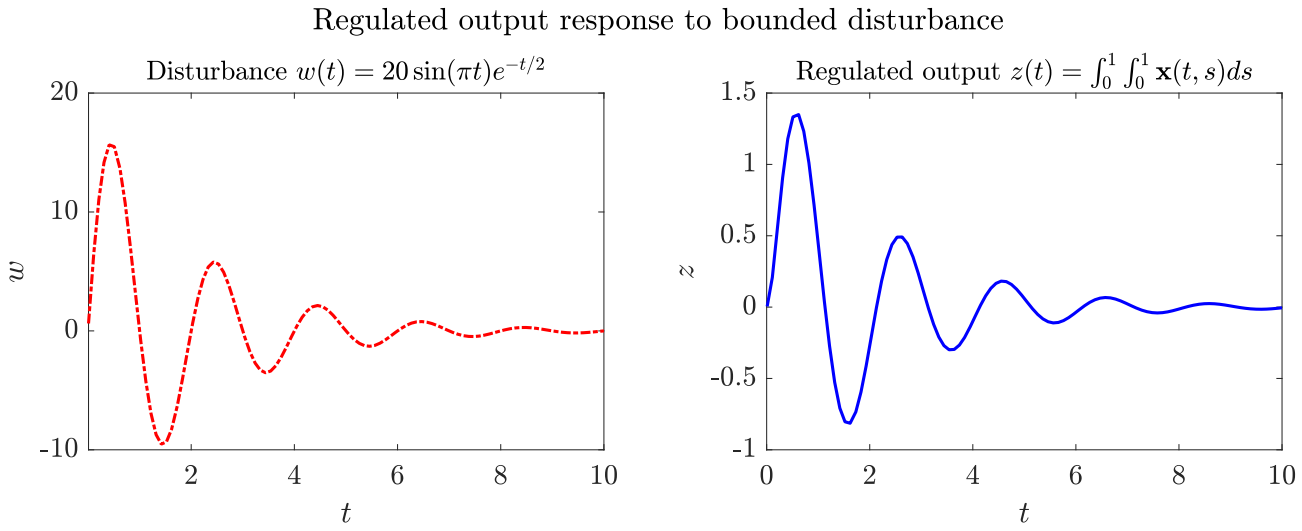


Figure 11.9: Simulated evolution of regulated output $z(t)$to (11.18) in response to a disturbance $w(t) = 20\sin(\pi t)e^{-t/2}$, starting with a zero initial state.

From the figure, it appears that the system is stable, with the bounded (decaying) disturbance $w \in L_2[0, \infty)$ also resulting in a bounded (decaying) output $z \in L_2[0, \infty)$. In fact, the value of the regulated output appears to be roughly one tenth that of the disturbance at any time $t \geq 0$, suggesting that the L_2-gain $\sup_{w,z \in L_2[0,\infty), w \neq 0} \frac{\|z\|_{L_2}}{\|w\|_{L_2}}$ of the system may also be bounded by 0.1. To compute a more accurate upper bound γ on the value of this gain, we solve the L_2-gain LPI presented in Sec. 13.1.4. To this end we first, compute the equivalent PIE representation of the PDE as

```
PIE = convert(PDE);
T = PIE.T;
A = PIE.A;      C = PIE.C1;
B = PIE.B1;     D = PIE.D11;
```

Given these operators, the PIE representation of the PDE (11.18) takes the form

$$\begin{aligned}\partial_t(\mathcal{T}\mathbf{x}_{\mathrm{f}})(t) &= \mathcal{A}\mathbf{x}_{\mathrm{f}}(t) + \mathcal{B}w(t),\\ z(t) &= \mathcal{C}\mathbf{x}_{\mathrm{f}}(t) + \mathcal{D}w(t),\end{aligned}$$

where the fundamental state $\mathbf{x}_{\mathrm{f}}(t) \in L_2[[a,b]\times[c,d]]$ corresponds to the highest-order mixed derivative of the PDE state, $\mathbf{x}_{\mathrm{f}}(t) = \partial_{s_1}^2\partial_{s_2}^2\mathbf{x}(t)$. Given this PIE representation, a smallest upper bound on the L_2-gain of the PDE can be computed by solving the LPI:

$$\begin{array}{lll}\min\limits_{\gamma\in\mathbb{R},\ \mathcal{P}} & \gamma, & \\ \text{s.t.} & \mathcal{P}\succ 0, & \begin{bmatrix} -\gamma & \mathcal{D} & \mathcal{B}^*\mathcal{P}\mathcal{T} \\ \mathcal{D} & -\gamma & \mathcal{C} \\ \mathcal{T}^*\mathcal{P}\mathcal{B} & \mathcal{C}^* & \mathcal{A}^*\mathcal{P}\mathcal{T}+\mathcal{T}^*\mathcal{P}\mathcal{A}\end{bmatrix}\end{array}$$

Note that this is reduced version of (13.7) in the particular case where $\mathcal{Q} = \mathcal{P}\mathcal{T}^*$, with a coercive operator $\mathcal{P}\succ 0$. We declare and solve this LPI as

```
% % Initialize LPI program
prog = lpiprogram([s1;s2],[a,b;c,d]);

% % Declare decision variables:
% %   gam \in \R,     P:L2-->L2,    Z:\R-->L2
% Scalar decision variable
[prog,gam] = lpidecvar(prog,'gam');
% Positive operator variable P>=0
[prog,P] = poslpivar(prog,T.dim);

% % Set inequality constraints:
% %   Q <= 0
Q = [-gam,        D',      (P*B)'*T;
     D,           -gam,    C;
     T'*(P*B),    C',     (P*A)'*T+T'*(P*A)];
opts_Q.psatz = 2;
prog = lpi_ineq(prog,-Q,opts_Q);

% % Set objective function:
% %   min gam
prog = lpisetobj(prog, gam);

% % Solve and retrieve the solution
prog_sol = lpisolve(prog,opts);
% Extract solved value of decision variable
gam_val = double(lpigetsol(prog_sol,gam));
```

Solving this LPI, we find an upper bound on the L_2-gain of the system as $\gamma = 0.09431$, approaching the true value of the L_2-gain of approximately 0.09397..

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

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

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-diffusiong 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

$$\dot{\mathbf{x}}(t,s) = c(s)\mathbf{x}(t,s) + b(s)\partial_s\mathbf{x}(t,s) + a(s)\partial_s^2\mathbf{x}(t,s), \qquad \mathbf{x}(t,0) = \partial_s\mathbf{x}(t,1) = 0,$$

where the values of the functions $a(s)$, $b(s)$ and $c(s)$ for $s \in [0,1]$ 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, the TDS examples are not declared in 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

$$\begin{aligned} \partial_t(\mathcal{T}\mathbf{x}_{\mathrm{f}})(t) + \mathcal{T}_w\dot{w}(t) + \mathcal{T}_u\dot{u}(t) &= \mathcal{A}\mathbf{x}_{\mathrm{f}}(t) + \mathcal{B}_1 w(t) + \mathcal{B}_2 u(t), \quad \mathbf{x}_{\mathrm{f}}(0) = x_{\mathrm{I}} \\ z(t) &= \mathcal{C}_1\mathbf{x}_{\mathrm{f}}(t) + \mathcal{D}_{11}w(t) + \mathcal{D}_{12}u(t), \\ y(t) &= \mathcal{C}_2\mathbf{x}_{\mathrm{f}}(t) + \mathcal{D}_{21}w(t) + \mathcal{D}_{22}u(t), \end{aligned} \tag{13.1}$$

where $\mathbf{x}_{\mathrm{f}} = \begin{bmatrix} x_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_1}[a,b] \\ L_2^{n_2}[c,d] \\ L_2^{n_3}[[a,b]\times[c,d]] \end{bmatrix}$, 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 following LPI.

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

$$\mathcal{T}^*\mathcal{T} - \gamma \preccurlyeq 0 \tag{13.2}$$

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

$$\partial_t(\mathcal{T}\mathbf{x}_{\mathrm{f}})(t) = \mathcal{A}\mathbf{x}_{\mathrm{f}}(t) \tag{13.3}$$

can be tested by solving the following LPI.

$$\begin{aligned} &\mathcal{P} \succ 0 \\ &\mathcal{T}^*\mathcal{P}\mathcal{A} + \mathcal{A}^*\mathcal{P}\mathcal{T} \preccurlyeq 0 \end{aligned} \quad . \tag{13.4}$$

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(PIE, settings);
```

or

```
>> [prog, Pop] = lpiscript(PIE, 'stability', settings);
```

Here `prog` will be an LPI 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 (13.3) can also be tested by solving the following LPI.

$$\begin{aligned} &\mathcal{P} \succ 0 \\ &\mathcal{T}\mathcal{P}\mathcal{A}^* + \mathcal{A}\mathcal{P}\mathcal{T}^* \preccurlyeq 0 \end{aligned} \tag{13.5}$$

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);
```

or

```
>> [prog, Pop] = lpiscript(PIE, 'stability-dual', settings);
```

Here `prog` will be an LPI 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

$$\begin{aligned} \partial_t(\mathcal{T}\mathbf{x}_{\mathrm{f}})(t) &= \mathcal{A}\mathbf{x}_{\mathrm{f}}(t) + \mathcal{B}_1 w(t), \quad x_{\mathrm{f}}(0) = \mathbf{0} \\ z(t) &= \mathcal{C}_1\mathbf{x}_{\mathrm{f}}(t) + \mathcal{D}_{11}w(t), \end{aligned} \tag{13.6}$$

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.

$$\begin{aligned} &\min_{\gamma,\mathcal{Q},\mathcal{R}} \quad \gamma \\ &\mathcal{T}^*\mathcal{Q} = \mathcal{Q}^*\mathcal{T} = \mathcal{R} \succeq 0 \\ &\begin{bmatrix} -\gamma I & \mathcal{D}_{11}^* & \mathcal{B}_1^*\mathcal{Q} \\ (\cdot)^* & -\gamma I & \mathcal{C}_1 \\ (\cdot)^* & (\cdot)^* & \mathcal{Q}^*\mathcal{A} + \mathcal{A}^*\mathcal{Q} \end{bmatrix} \preccurlyeq 0 \end{aligned} \tag{13.7}$$

Given a structure `PIE`, this LPI may be solved for the associated PIE by calling

```
>> [prog, Qop, gam] = PIETOOLS_Hinf_gain(PIE, settings);
```

or

```
>> [prog, Qop, gam] = lpiscript(PIE, 'l2gain', settings);
```

Here `prog` will be an LPI 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 (13.6) 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

$$\begin{aligned} &\min_{\gamma,\mathcal{Q},\mathcal{R}} \quad \gamma \\ &\mathcal{T}\mathcal{Q} = \mathcal{Q}^*\mathcal{T}^* = \mathcal{R} \succeq 0 \\ &\begin{bmatrix} -\gamma I & \mathcal{D}_{11} & \mathcal{C}_1\mathcal{Q} \\ (\cdot)^* & -\gamma I & \mathcal{B}_1^* \\ (\cdot)^* & (\cdot)^* & \mathcal{Q}^*\mathcal{A}^* + \mathcal{A}\mathcal{Q} \end{bmatrix} \preccurlyeq 0 \end{aligned} \tag{13.8}$$

Given a structure `PIE`, this LPI may be solved for the associated PIE by calling

```
>> [prog, Qop, gam] = PIETOOLS_Hinf_gain_dual(PIE, settings);
```

or

```
>> [prog, Qop, gam] = lpiscript(PIE, 'l2gain-dual', settings);
```

Here `prog` will be an LPI 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 of Eq. (13.6), we can test whether the system is passive by solving the LPI

$$\begin{aligned}&\mathcal{P}\succ 0\\&\begin{bmatrix}-\mathcal{D}_{11}^*-\mathcal{D}_{11} & \mathcal{B}_1^*\mathcal{P}\mathcal{T}-\mathcal{C}_1\\ (\cdot)^* & \mathcal{T}^*\mathcal{P}\mathcal{A}+\mathcal{A}^*\mathcal{P}\mathcal{T}\end{bmatrix}\preccurlyeq 0\end{aligned}\tag{13.9}$$

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.

13.1.7 H_2-Norm

For a PIE of the form

$$\begin{aligned}\partial_t(\mathcal{T}\mathbf{x}_{\mathrm{f}})(t)&=\mathcal{A}\mathbf{x}_{\mathrm{f}}(t)+\mathcal{B}_1 w(t),\quad x_{\mathrm{f}}(0)=\mathbf{0}\\ z(t)&=\mathcal{C}_1\mathbf{x}_{\mathrm{f}}(t),\end{aligned}\tag{13.10}$$

we can compute the H_2-norm by extending its usual definition to PIEs as follows: consider solutions of the auxiliary PIE

$$\begin{aligned}\partial_t(\mathcal{T}\mathbf{x}_{\mathrm{f}})(t)&=\mathcal{A}\mathbf{x}_{\mathrm{f}}(t),\\ z(t)&=\mathcal{C}_1\mathbf{x}_{\mathrm{f}}(t),\qquad \mathcal{T}\mathbf{x}_{\mathrm{f}}(0)=\mathcal{B}_1 x_0.\end{aligned}\tag{13.11}$$

We define the H_2 norm of System (13.10), denoted Σ, as

$$\|\Sigma\|_{H_2}:=\sup_{\substack{z,\mathbf{x}\text{ satisfy (13.11)}\\ \|x_0\|=1}}\|z\|_{L_2}.$$

Then, we can compute an optimal upper-bound on this H_2-norm by solving the following LPI:

$$\begin{aligned}&\min_{\gamma,\mathcal{R},\mathcal{Q},\mathcal{W}}\ \gamma\\&\mathcal{R}\succcurlyeq 0,\gamma>0\\&\mathcal{Q}^*\mathcal{T}=\mathcal{T}^*\mathcal{Q}=\mathcal{R}\\&\begin{bmatrix}-\gamma I & \mathcal{C}_1\\ \mathcal{C}_1^* & \mathcal{A}^*\mathcal{Q}+\mathcal{Q}^*\mathcal{A}\end{bmatrix}\preccurlyeq 0\end{aligned}$$

$$\begin{bmatrix} \mathcal{W} & \mathcal{B}_1^*\mathcal{Q} \\ \mathcal{Q}^*\mathcal{B}_1 & \mathcal{R} \end{bmatrix} \succcurlyeq 0$$
$$\text{trace}(\mathcal{W}) \leq \gamma \tag{13.12}$$

If (13.12) is feasible for some $\gamma > 0$, PI operator $\mathcal{R}, \mathcal{W} \succcurlyeq 0$, and $\mathcal{Q}$, then $\|\Sigma\|_{H_2} \leq \gamma$.
Given a structure `PIE`, this LPI may be solved for the associated PIE by calling

```
>> [prog, Wm, gam, Rop, Qop] = PIETOOLS_H2_norm_o(PIE, settings);
```

or

```
>> [prog, Wm, gam, Rop, Qop] = lpiscript(PIE, 'h2norm', settings);
```

Here `prog` will be an LPI 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 H_2-norm of the system. The outputs `Rop, Qop, Wm` will be `dopvar` objects describing the (unsolved) decision operators. Note that, in the most common case when the input $w(t)$ is finite-dimensional, the operator $\mathcal{W}$ is a matrix.

13.1.8 Dual H_2-Norm

For a system (13.10), an optimal upper bound γ on the H_2 norm can also be found by solving the LPIs

$$\begin{aligned} &\min_{\gamma,\mathcal{R},\mathcal{Q},W} \quad \gamma \\ &\mathcal{R} \succcurlyeq 0, \gamma > 0 \\ &\mathcal{Q}^*\mathcal{T}^* = \mathcal{T}\mathcal{Q} = \mathcal{R} \\ &\begin{bmatrix} -\gamma I & \mathcal{B}_1^* \\ \mathcal{B}_1 & \mathcal{A}\mathcal{Q} + \mathcal{Q}^*\mathcal{A}^* \end{bmatrix} \preccurlyeq 0 \\ &\begin{bmatrix} \mathcal{W} & \mathcal{C}_1\mathcal{Q} \\ \mathcal{Q}^*\mathcal{C}_1^* & \mathcal{R} \end{bmatrix} \succcurlyeq 0 \\ &\text{trace}(\mathcal{W}) \leq \gamma \end{aligned} \tag{13.13}$$

Given a structure `PIE`, this LPI may be solved for the associated PIE by calling

```
>> [prog, Wm, gam, Rop, Qop] = PIETOOLS_H2_norm_c(PIE, settings);
```

or

```
>> [prog, Wm, gam, Rop, Qop] = lpiscript(PIE, 'h2norm-dual', settings);
```

Here `prog` will be an LPI 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 H_2-norm of the system. The outputs `Rop, Qop, Wm` will be `dopvar` objects describing the (unsolved) decision operators. Note that, in the most common case when the output $z(t)$ is finite-dimensional, the operator $\mathcal{W}$ is a matrix.

13.2 LPIs for Optimal Estimation of PIEs

13.2.1 H_∞ Estimator

For the following PIE

$$\begin{aligned}\partial_t(\mathcal{T}\mathbf{x}_{\rm f})(t)+\mathcal{T}_w\dot{w}(t)+\mathcal{T}_u\dot{u}(t)&=\mathcal{A}\mathbf{x}_{\rm f}(t)+\mathcal{B}_1w(t)+\mathcal{B}_2u(t),\\ z(t)&=\mathcal{C}_1\mathbf{x}_{\rm f}(t)+\mathcal{D}_{11}w(t)+\mathcal{D}_{12}u(t),\\ y(t)&=\mathcal{C}_2\mathbf{x}_{\rm f}(t)+\mathcal{D}_{21}w(t)+\mathcal{D}_{22}u(t),\end{aligned}\tag{13.14}$$

a state estimator has the following structure:

$$\begin{aligned}\partial_t(\mathcal{T}\hat{\mathbf{x}}_{\rm f})(t)+\mathcal{T}_u\dot{u}(t)&=\mathcal{A}\hat{\mathbf{x}}_{\rm f}(t)+\mathcal{L}(\hat{y}(t)-y(t))+\mathcal{B}_2u(t),\\ \hat{z}(t)&=\mathcal{C}_1\hat{\mathbf{x}}_{\rm f}(t)+\mathcal{D}_{12}u(t)\\ \hat{y}(t)&=\mathcal{C}_2\hat{\mathbf{x}}_{\rm f}(t)+\mathcal{D}_{22}u(t),\end{aligned}\tag{13.15}$$

so that the errors $\mathbf{e}:=\hat{\mathbf{x}}_{\rm f}-\mathbf{x}_{\rm f}$ and $\tilde{z}:=\hat{z}-z$ in respectively the state and regulated output estimates satisfy

$$\begin{aligned}\partial_t(\mathcal{T}\mathbf{e})(t)-\mathcal{T}_w\dot{w}(t)&=(\mathcal{A}+\mathcal{L}\mathcal{C}_2)\mathbf{e}(t)-(\mathcal{B}_1+\mathcal{L}\mathcal{D}_{21})w(t),\\ \tilde{z}(t)&=\mathcal{C}_1\mathbf{e}(t)-\mathcal{D}_{11}w(t)\end{aligned}$$

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 following LPI.

$$\begin{aligned}&\min_{\gamma,\mathcal{P},\mathcal{Z}}\quad\gamma\\ &\mathcal{P}\succ 0\\ &\begin{bmatrix}\mathcal{T}_w^*(\mathcal{P}\mathcal{B}_1+\mathcal{Z}\mathcal{D}_{21})+(\cdot)^* & 0 & (\cdot)^*\\ 0 & 0 & 0\\ -(\mathcal{P}\mathcal{A}+\mathcal{Z}\mathcal{C}_2)^*\mathcal{T}_w & 0 & 0\end{bmatrix}+\begin{bmatrix}-\gamma I & -\mathcal{D}_{11}^\top & -(\mathcal{P}\mathcal{B}_1+\mathcal{Z}\mathcal{D}_{21})^*\mathcal{T}\\ (\cdot)^* & -\gamma I & \mathcal{C}_1\\ (\cdot)^* & (\cdot)^* & (\mathcal{P}\mathcal{A}+\mathcal{Z}\mathcal{C}_2)^*\mathcal{T}+(\cdot)^*\end{bmatrix}\preccurlyeq 0\end{aligned}\tag{13.16}$$

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 $\|\tilde{z}\|\leq\gamma\|w\|$. 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);
```

or

```
>> [prog, Lop, gam, Pop, Zop] = lpiscript(PIE, 'hinf-observer', settings);
```

Here `prog` will be an LPI 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 `opvar` objects describing the solved operators $\mathcal{P}$ and $\mathcal{Z}$. See Chapter 7 for more information on how LPIs are solved and on the `settings` input.

13.2.2 H_2 Estimator

For the following PIE

$$\begin{aligned}\partial_t(\mathcal{T}\mathbf{x}_{\mathrm{f}})(t)+\mathcal{T}_u\dot{u}(t)&=\mathcal{A}\mathbf{x}_{\mathrm{f}}(t)+\mathcal{B}_1w(t)+\mathcal{B}_2u(t),\\ z(t)&=\mathcal{C}_1\mathbf{x}_{\mathrm{f}}(t)+\mathcal{D}_{12}u(t),\\ y(t)&=\mathcal{C}_2\mathbf{x}_{\mathrm{f}}(t)+\mathcal{D}_{21}w(t)+\mathcal{D}_{22}u(t),\end{aligned}\tag{13.17}$$

a state estimator with the following structure:

$$\begin{aligned}\partial_t(\mathcal{T}\hat{\mathbf{x}}_{\mathrm{f}})(t)+\mathcal{T}_u\dot{u}(t)&=\mathcal{A}\hat{\mathbf{x}}_{\mathrm{f}}(t)+\mathcal{L}(\hat{y}(t)-y(t))+\mathcal{B}_2u(t),\\ \hat{z}(t)&=\mathcal{C}_1\hat{\mathbf{x}}_{\mathrm{f}}(t)+\mathcal{D}_{12}u(t)\\ \hat{y}(t)&=\mathcal{C}_2\hat{\mathbf{x}}_{\mathrm{f}}(t)+\mathcal{D}_{22}u(t).\end{aligned}\tag{13.18}$$

so that the errors $\mathbf{e}:=\hat{\mathbf{x}}_{\mathrm{f}}-\mathbf{x}_{\mathrm{f}}$ and $\tilde{z}:=\hat{z}-z$ in respectively the state and regulated output estimates satisfy

$$\begin{aligned}\partial_t(\mathcal{T}\mathbf{e})(t)&=(\mathcal{A}+\mathcal{L}\mathcal{C}_2)\mathbf{e}(t)-(\mathcal{B}_1+\mathcal{L}\mathcal{D}_{21})w(t),\\ \tilde{z}(t)&=\mathcal{C}_1\mathbf{e}(t)\end{aligned}$$

The estimation problem is to find $\mathcal{L}$ such that, for some $\gamma>0$ the above system, called Σ_e, has H_2-norm $\|\Sigma_e\|_{H_2}\leq\gamma$. The optimal estimator can be found by solving the following LPI.

$$\begin{aligned}&\min_{\gamma,\mathcal{Z},\mathcal{P},\mathcal{W}}\quad\gamma\\ &\gamma>0\\ &\mathrm{trace}(\mathcal{W})\leq\gamma\\ &\mathcal{P}\succ 0\\ &\begin{bmatrix}-\gamma I & \mathcal{C}_1\\ \mathcal{C}_1^* & \mathcal{A}^*\mathcal{P}\mathcal{T}+\mathcal{T}^*\mathcal{P}\mathcal{A}+\mathcal{T}^*\mathcal{Z}\mathcal{C}_2+\mathcal{C}_2^*\mathcal{Z}^*T\end{bmatrix}\preccurlyeq 0\\ &\begin{bmatrix}\mathcal{W} & -(\mathcal{B}_1^*\mathcal{P}+\mathcal{D}_{21}^*\mathcal{Z}^*)\\ -(\mathcal{P}\mathcal{B}_1+\mathcal{Z}\mathcal{D}_{21}) & \mathcal{P}\end{bmatrix}\succcurlyeq 0\end{aligned}\tag{13.19}$$

If this LPI is feasible for some $\gamma>0$, PI operators $\mathcal{P}$, $\mathcal{Z}$, and $\mathcal{W}$, then, letting $\mathcal{L}:=\mathcal{P}^{-1}\mathcal{Z}$, the estimation error will satisfy $\|\tilde{z}\|\leq\gamma\|w\|$. Given a structure `PIE`, this LPI may be solved for the associated PIE by calling

```
>> [prog, Lop, gam, Pop, Zop, Wop] = PIETOOLS_H2_estimator(PIE,settings);
```

or

```
>> [prog, Lop, gam, Pop, Zop, Wop] = lpiscript(PIE, 'h2-observer', settings);
```

Here `prog` will be an LPI 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`, `Zop`, and `Wop` will be `opvar` objects describing the solved operators $\mathcal{P}$, $\mathcal{Z}$ and $\mathcal{W}$. See Chapter 7 for more information on how LPIs are solved and on the `settings` input.

13.3 LPIs for Optimal Control of PIEs

13.3.1 H_∞ Control

In this section, we discuss the synthesis of H_∞ optimal control of a PIE of the form

$$\begin{aligned}\partial_t(\mathcal{T}\mathbf{x}_\mathrm{f})(t) &= \mathcal{A}\mathbf{x}_\mathrm{f}(t) + \mathcal{B}_1 w(t) + \mathcal{B}_2 u(t), \qquad \mathbf{x}_\mathrm{f}(0) = \mathbf{0}\\ z(t) &= \mathcal{C}_1\mathbf{x}_\mathrm{f}(t) + \mathcal{D}_{11}w(t) + \mathcal{D}_{12}u(t),\end{aligned} \tag{13.20}$$

where $w, z \in L_2[0,\infty)$. 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

$$\begin{aligned}&\min_{\gamma,\mathcal{P},\mathcal{Z}} \quad \gamma\\ &\mathcal{P} \succ 0\\ &\begin{bmatrix} -\gamma I & \mathcal{D}_{11} & (\mathcal{C}_1\mathcal{P} + \mathcal{D}_{12}\mathcal{Z})\mathcal{T}^* \\ \mathcal{D}_{11}^* & -\gamma I & \mathcal{B}_1^* \\ ()^* & \mathcal{B}_1 & ()^* + (\mathcal{A}\mathcal{P} + \mathcal{B}_2\mathcal{Z})\,\mathcal{T}^* \end{bmatrix} \preccurlyeq 0\end{aligned} \tag{13.21}$$

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 $u = \mathcal{K}\mathbf{x}_\mathrm{f}$ will be such that $\|z\| \leq \gamma\|w\|$. Given a structure `PIE`, this LPI may be solved for the associated PIE by calling

```
>> [prog_sol, Kop, gamma, Pop, Zop] = PIETOOLS_Hinf_control(PIE,settings);
```

or

```
>> [prog_sol, Kop, gamma, Pop, Zop] = lpiscript(PIE, 'hinf-controller', settings);
```

Here `prog` will be an LPI 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 `opvar` objects describing the solved operators $\mathcal{P}$ and $\mathcal{Z}$. See Chapter 7 for more information on how LPIs are solved and on the `settings` input.

13.3.2 H_2 Controller

For a PIE of the form

$$\begin{aligned}\partial_t(\mathcal{T}\mathbf{x}_\mathrm{f})(t) &= \mathcal{A}\mathbf{x}_\mathrm{f}(t) + \mathcal{B}_1 w(t) + \mathcal{B}_2 u(t), \quad \mathbf{x}_\mathrm{f}(0) = \mathbf{0}\\ z(t) &= \mathcal{C}_1\mathbf{x}_\mathrm{f}(t) + \mathcal{D}_{12}u(t),\end{aligned} \tag{13.22}$$

with $w, z \in L_2[0,\infty)$. The problem is to determine $\mathcal{K}$ such that, using the full-state feedback law $u(t) = \mathcal{K}\mathbf{x}_\mathrm{f}(t)$, the regulated output z of the closed loop system, denoted Σ_c has H_2 norm $\|\Sigma_c\|_{H_2} \leq \gamma$ for a particular $\gamma > 0$.

The controller can be found by solving the following LPI.

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

$$
\begin{aligned}
&\mathcal{P} \succ 0, \gamma > 0 \\
&\text{trace}(\mathcal{W}) \leq \gamma \\
&\begin{bmatrix} -\gamma I & \mathcal{B}_1^* \\ \mathcal{B}_1 & \mathcal{A}\mathcal{P}\mathcal{T}^* + \mathcal{T}\mathcal{P}\mathcal{A}^* + \mathcal{B}_2\mathcal{Z} + \mathcal{Z}^*\mathcal{B}_2^* \end{bmatrix} \preccurlyeq 0 \\
&\begin{bmatrix} \mathcal{W} & \mathcal{C}_1\mathcal{P} + \mathcal{D}_{12}\mathcal{Z} \\ \mathcal{P}^*\mathcal{C}_1^* + \mathcal{Z}^*\mathcal{D}_{12}^* & \mathcal{P} \end{bmatrix} \succcurlyeq 0
\end{aligned} \tag{13.23}
$$

If the LPI is feasible, let $\mathcal{K} = \mathcal{Z}\mathcal{P}^{-1}$. Given a structure `PIE`, this LPI may be solved for the associated PIE by calling

```
>>[prog, Kop, gam, Pop, Zop, Wop] = PIETOOLS_H2_control(PIE, settings);
```

or

```
>> [prog, Kop, gam, Pop, Zop, Wop] = lpiscript(PIE, 'h2control', settings);
```

Here `prog` will be an LPI 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 H_2-norm of Σ_c. The outputs `Pop, Zop`, and `Wop` will be `opvar` objects describing the solved decision operators $\mathcal{P}, \mathcal{Z}$, and $\mathcal{W}$. See Chapter 7 for more information on how LPIs are solved and on 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

$$\langle \mathbf{x}, \mathbf{y} \rangle_{L_2} = \int_\Omega \Big[\mathbf{x}(s)\Big]^T \mathbf{y}(s) ds.$$

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*

$$R := \{R_0, R_1, R_2\} \in \Big\{L_2^{m\times n}[a,b], L_2^{m\times n}\Big[[a,b]^2\Big], L_2^{m\times n}\Big[[a,b]^2\Big]\Big\} =: \mathcal{N}_{1D}^{m\times n}[a,b],$$

we define the associated 3-PI operator $\mathcal{P}[R] := \mathcal{P}_{\{R_0,R_1,R_2\}} : L_2^n[a,b] \to L_2^m[a,b]$ *as*

$$\Big(\mathcal{P}[R]\mathbf{x}\Big)(s) := R_0(s)\mathbf{x}(s) + \int_a^s R_1(s,\theta)\mathbf{x}(\theta)d\theta + \int_s^b R_2(s,\theta)\mathbf{x}(\theta)d\theta, \tag{A.1}$$

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 ith element of the product Px is given by:

$$\Big(Px\Big)_i = D_{ii}x_i + \sum_{j=1}^{i-1} L_{ij}x_j + \sum_{j=i+1}^{n} U_{ij}x_j.$$

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:

$$\Big(\mathcal{P}[R]\mathbf{x}\Big)(s) = R_0(s)\mathbf{x}(s) + \int_a^s R_1(s,\theta)\mathbf{x}(\theta)d\theta + \int_s^b R_2(s,\theta)\mathbf{x}(\theta)d\theta.$$

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*

$$\begin{aligned} R &:= \begin{bmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{bmatrix} \\ &\in \begin{bmatrix} L_2^{m\times n}[[a,b]\times[c,d]] & L_2^{m\times n}[[a,b]\times[c,d]^2] & L_2^{m\times n}[[a,b]\times[c,d]^2] \\ L_2^{m\times n}[[a,b]^2\times[c,d]] & L_2^{m\times n}[[a,b]^2\times[c,d]^2] & L_2^{m\times n}[[a,b]^2\times[c,d]^2] \\ L_2^{m\times n}[[a,b]^2\times[c,d]] & L_2^{m\times n}[[a,b]^2\times[c,d]^2] & L_2^{m\times n}[[a,b]^2\times[c,d]^2] \end{bmatrix} =: \mathcal{N}_{2D}^{m\times n}[[a,b]\times[c,d]] \end{aligned}$$

we define the associated 9-PI operator $\mathcal{P}[R] := \mathcal{P}\left[\begin{smallmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{smallmatrix}\right] : L_2^n\Big[[a_1,b_1]\times[a_2,b_2]\Big] \to L_2^m\Big[[a_1,b_1]\times[a_2,b_2]\Big]$ *as*

$$\begin{aligned} \left(\mathcal{P}[R]\mathbf{x}\right)(s,r) &= R_{00}(s,r)\mathbf{x}(s,r) + \int_c^r R_{01}(s,r,\nu)\mathbf{x}(s,\nu)d\nu + \int_r^d R_{02}(s,r,\nu)\mathbf{x}(s,\nu)d\nu \\ &+ \int_a^s R_{10}(s,r,\theta)\mathbf{x}(\theta,r)d\theta + \int_a^s\int_c^r R_{11}(s,r,\theta,\nu)\mathbf{x}(\theta,\nu)d\nu d\theta + \int_a^s\int_r^d R_{12}(s,r,\theta,\nu)\mathbf{x}(\theta,\nu)d\nu d\theta \\ &+ \int_s^b R_{20}(s,r,\theta)\mathbf{x}(\theta,r)d\theta + \int_s^b\int_c^r R_{21}(s,r,\theta,\nu)\mathbf{x}(\theta,\nu)d\nu d\theta + \int_s^b\int_r^d R_{22}(s,r,\theta,\nu)\mathbf{x}(\theta,\nu)d\nu d\theta \end{aligned} \tag{A.2}$$

for any $\mathbf{x} \in L_2^n\Big[[a_1,b_2]\times[a_2,b_2]\Big]$.

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 $Z^{\mathrm{n}}\Big[[a,b],[c,d]\Big] := \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_s}[a,b] \\ L_2^{n_r}[c,d] \\ L_2^{n_2}\Big[[a,b]\times[c,d]\Big] \end{bmatrix}$, where $\mathrm{n} := \{n_0, n_s, n_r, n_2\}$, 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 $\mathbf{x}_2 \in L_2^{n_2}\Big[[a,b]\times[c,d]\Big]$.

Definition 3 (PI Operator)**.** *For any operator* $\mathcal{R} : Z^{\mathrm{n}}\Big[[a,b],[c,d]\Big] \to Z^{\mathrm{m}}\Big[[a,b],[c,d]\Big]$ *with* $\mathrm{m} := \{m_0, m_s, m_r, m_2\}$ *and* $\mathrm{n} := \{n_0, n_s, n_r, n_2\}$*, we say that* $\mathcal{R}$ *is a PI operator, denoted by* $\mathcal{R} \in \Pi^{\mathrm{m}\times\mathrm{m}}$ *if there exist parameters*

$$R := \begin{bmatrix} R_{00} & R_{0s} & R_{0r} & R_{02} \\ R_{s0} & R_{ss} & R_{sr} & R_{s2} \\ R_{r0} & R_{rs} & R_{rr} & R_{s2} \\ R_{20} & R_{2s} & R_{2r} & R_{22} \end{bmatrix}$$

$$\in \begin{bmatrix} \mathbb{R}^{m_0\times n_0} & L_2^{m_0\times n_s}[a,b] & L_2^{m_0\times n_r}[c,d] & L_2^{m_0\times n_2}\Big[[a,b]\times[c,d]\Big] \\ L_2^{m_s\times n_0}[a,b] & \mathcal{N}_{1D}^{m_s\times n_s}[a,b] & L_2^{m_s\times n_r}\Big[[a,b]\times[c,d]\Big] & \mathcal{N}_{1D\leftarrow 2D}^{m_s\times n_2}\Big[[a,b],[c,d]\Big] \\ L_2^{m_r\times n_0}[c,d] & L_2^{m_r\times n_s}\Big[[a,b]\times[c,d]\Big] & \mathcal{N}_{1D}^{m_r\times n_r}[c,d] & \mathcal{N}_{1D\leftarrow 2D}^{m_r\times n_2}\Big[[c,d],[a,b]\Big] \\ L_2^{m_2\times n_0}\Big[[a,b]\times[c,d]\Big] & \mathcal{N}_{2D\leftarrow 1D}^{m_2\times n_s}\Big[[a,b],[c,d]\Big] & \mathcal{N}_{2D\leftarrow 1D}^{m_2\times n_r}\Big[[c,d],[a,b]\Big] & \mathcal{N}_{2D}^{m_2\times n_r}\Big[[a,b]\times[c,d]\Big] \end{bmatrix} =: \mathcal{N}^{\mathrm{m}\times\mathrm{n}}\Big[[a,b]\times[c,d]\Big]$$

such that

$$\begin{aligned} \mathcal{R} &= (\mathcal{P}[R]\mathbf{x})(s,r) \\ &= \mathcal{P}\begin{bmatrix} R_{00} & R_{0s} & R_{0r} & R_{02} \\ R_{s0} & R_{ss} & R_{sr} & R_{s2} \\ R_{r0} & R_{rs} & R_{rr} & R_{s2} \\ R_{20} & R_{2s} & R_{2r} & R_{22} \end{bmatrix}\begin{bmatrix} x_0 \\ \mathbf{x}_s \\ \mathbf{x}_r \\ \mathbf{x}_2 \end{bmatrix} \\ &:= \begin{bmatrix} R_{00}x_0 & + \int_a^b R_{0s}(s)\mathbf{x}_s(s)ds & + \int_c^d R_{0r}(r)\mathbf{x}_r(r)dr & + \int_a^b\int_c^d R_{02}\mathbf{x}_2(s,r)drds \\ R_{s0}(s)x_0 & + (\mathcal{P}[R_{ss}]\mathbf{x}_s)(s) & + \int_c^d R_{sr}(s,r)\mathbf{x}_r(r)dr & + (\mathcal{P}[R_{s2}]\mathbf{x}_2)(s) \\ R_{r0}(r)x_0 & + \int_a^b R_{rs}(s,r)\mathbf{x}_s(s)ds & + (\mathcal{P}[R_{rr}]\mathbf{x}_r)(r) & + (\mathcal{P}[R_{r2}]\mathbf{x}_2)(r) \\ R_{20}(s,r)x_0 & + (\mathcal{P}[R_{2s}]\mathbf{x}_s)(s,r) & + (\mathcal{P}[R_{2r}]\mathbf{x}_r)(s,r) & + (\mathcal{P}[R_{22}]\mathbf{x}_2)(s,r) \end{bmatrix}, \end{aligned} \tag{A.3}$$

for any $\mathbf{x} = \begin{bmatrix} x_0 \\ \mathbf{x}_s \\ \mathbf{x}_r \\ \mathbf{x}_2 \end{bmatrix} \in \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_s}[a,b] \\ L_2^{n_r}[c,d] \\ L_2^{n_2}\Big[[a,b]\times[c,d]\Big] \end{bmatrix} =: Z^{\mathrm{n}}$*, where for given parameters*

$$P := \{P_0, P_1, P_2\}$$
$$\in \Big\{L_2^{m_s\times n_2}\big[[a,b]\times[c,d]\big], L_2^{m_s\times n_2}\big[[a,b]^2\times[c,d]\big], L_2^{m_s\times n_2}\big[[a,b]^2\times[c,d]\big]\Big\} =: \mathcal{N}_{1D\leftarrow 2D}^{m_s\times n_2}\big[[a,b],[c,d]\big],$$
$$Q := \{Q_0, Q_1, Q_2\}$$
$$\in \Big\{L_2^{m_s\times n_2}\big[[a,b]\times[c,d]\big], L_2^{m_s\times n_2}\big[[a,b]^2\times[c,d]\big], L_2^{m_s\times n_2}\big[[a,b]^2\times[c,d]\big]\Big\} =: \mathcal{N}_{2D\leftarrow 1D}^{m_s\times n_2}\big[[a,b],[c,d]\big],$$

we define

$$(\mathcal{P}[P]\mathbf{x}_2)(s) := \int_c^d \Big[P_0(s,r)\mathbf{x}_2(s,r) + \int_a^s P_1(s,r,\theta)\mathbf{x}_2(\theta,r)d\theta + \int_s^b P_2(s,r,\theta)\mathbf{x}(\theta,r)d\theta\Big]dr,$$

$$(\mathcal{P}[Q]\mathbf{x}_s)(s,r) := Q_0(s,r)\mathbf{x}_s(s) + \int_a^s Q_1(s,r,\theta)\mathbf{x}_s(\theta)d\theta + \int_s^b Q_2(s,r,\theta)\mathbf{x}_s(\theta)d\theta,$$

for any $\mathbf{x}_2 \in L_2^{n_2}\Big[[a,b]\times[c,d]\Big]$ *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* $Q, R \in \mathcal{N}^{m\times n}\Big[[a,b],[c,d]\Big]$, *there exist unique parameters* $P \in \mathcal{N}^{m\times n}\Big[[a,b],[c,d]\Big]$ *such that*

$$\mathcal{P}[R] + \mathcal{P}[Q] = \mathcal{P}[P].$$

That is, for any $\mathbf{x} \in Z^{\mathrm{n}}\Big[[a,b],[c,d]\Big]$,

$$\Big((\mathcal{P}[Q] + \mathcal{P}[R])\mathbf{x}\Big)(s) = \Big(\mathcal{P}[P]\mathbf{x}\Big)(s).$$

Proof. We outline the proof for 3-PI operators, for which it is easy to see that, by linearity of the integral,

$$\begin{aligned}
&\Big(\mathcal{P}_{\{R_0,R_1,R_2\}}\mathbf{x}\Big)(s) + \Big(\mathcal{P}_{\{Q_0,Q_1,Q_2\}}\mathbf{x}\Big)(s)\\
&\quad = [R_0(s)+Q_0(s)]\mathbf{x}(s) + \int_a^s [R_1(s,\theta)+Q_1(s,\theta)]\mathbf{x}(\theta)d\theta + \int_s^b [R_2(s,\theta)+Q_2(s,\theta)]\mathbf{x}(\theta)d\theta\\
&\quad = \Big(\mathcal{P}_{\{R_0+Q_0,R_1+Q_1,R_2+Q_2\}}\mathbf{x}\Big)(s).
\end{aligned}$$

Similar results can be easily derived for more general PI operators. For a full proof, we refer to [5]. □

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* $R_1 \in \mathcal{N}^{m\times p}\Big[[a,b],[c,d]\Big]$ *and* $R_2 \in \mathcal{N}^{p\times n}\Big[[a,b],[c,d]\Big]$, *there exist unique parameters* $R_3 \in \mathcal{N}^{m\times n}\Big[[a,b],[c,d]\Big]$ *such that*

$$\mathcal{P}[R_1] \circ \mathcal{P}[R_2] = \mathcal{P}[R_3].$$

That is, for any $\mathbf{x} \in Z^{\mathrm{n}}\Big[[a,b],[c,d]\Big]$,

$$\Big(\mathcal{P}[R_1]\Big(\mathcal{P}[R_2]\mathbf{x}\Big)\Big)(s) = \Big(\mathcal{P}[R_3]\mathbf{x}\Big)(s).$$

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.

$$\Big(\mathcal{P}_{\{R_0,R_1,R_2\}}\mathbf{x}\Big)(s) = R_0(s)\mathbf{x} + \int_a^b \Big[\mathbf{I}(s-\theta)R_1(s,\theta) + \mathbf{I}(\theta-s)R_2(s,\theta)\Big]\mathbf{x}(\theta)d\theta.$$

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 $R_1, Q_1 \in L_2^{m_s\times n_s}\Big[[a,b]^2\Big]$,

$$\begin{aligned}
\Big(\mathcal{P}_{\{0,R_1,0\}}\Big(\mathcal{P}_{\{0,Q_1,0\}}\mathbf{x}\Big)\Big)(s) &= \int_a^s R_1(s,\eta)\int_a^\eta Q_1(\eta,\theta)\mathbf{x}(\theta)d\theta d\eta \\
&= \int_a^b\int_a^b \mathbf{I}(s-\eta)\mathbf{I}(\eta-\theta)R_1(s,\eta)Q_1(\eta,\theta)\mathbf{x}(\theta)d\theta d\eta \\
&= \int_a^s \left[\int_\theta^s R_1(s,\eta)Q_1(\eta,\theta)d\eta\right]\mathbf{x}(\theta)d\theta = \Big(\mathcal{P}_{\{0,P_{11},0\}}\mathbf{x}\Big)(s),
\end{aligned}$$

where $P_{11}(s,\theta) := \int_\theta^s R_1(s,\eta)Q_1(\theta,\eta)d\eta$. Similarly, we can show that

$$\begin{aligned}
\Big(\mathcal{P}_{\{0,R_1,0\}}\Big(\mathcal{P}_{\{0,0,Q_2\}}\mathbf{x}\Big)\Big)(s) &= \int_a^s \left[\int_a^\theta R_1(s,\eta)Q_2(\eta,\theta)d\eta\right]\mathbf{x}(\theta)d\theta + \int_s^b \left[\int_a^s R_1(s,\eta)Q_2(\eta,\theta)d\eta\right]\mathbf{x}(\theta)d\theta \\
\Big(\mathcal{P}_{\{0,0,R_2\}}\Big(\mathcal{P}_{\{0,Q_1,0\}}\mathbf{x}\Big)\Big)(s) &= \int_a^s \left[\int_s^b R_2(s,\eta)Q_1(\eta,\theta)d\eta\right]\mathbf{x}(\theta)d\theta + \int_s^b \left[\int_\theta^b R_2(s,\eta)Q_1(\eta,\theta)d\eta\right]\mathbf{x}(\theta)d\theta \\
\Big(\mathcal{P}_{\{0,0,R_2\}}\Big(\mathcal{P}_{\{0,0,Q_2\}}\mathbf{x}\Big)\Big)(s) &= \int_s^b \left[\int_s^\theta R_2(s,\eta)Q_2(\theta,\eta)d\eta\right]\mathbf{x}(\theta)d\theta = \Big(\mathcal{P}_{\{0,0,P_{22}\}}\mathbf{x}\Big)(s),
\end{aligned}$$

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 easy to see that

$$\begin{aligned}
\Big(\mathcal{P}_{\{R_0,0,0\}}\Big(\mathcal{P}_{\{0,Q_1,Q_2\}}\mathbf{x}\Big)\Big)(s) &= \int_a^s R_0(s)Q_1(s,\theta)\mathbf{x}(\theta)d\theta + \int_s^b R_0(s)Q_2(s,\theta)\mathbf{x}(\theta)d\theta = \Big(\mathcal{P}_{\{0,P_{01},P_{02}\}}\mathbf{x}\Big)(s), \\
\Big(\mathcal{P}_{\{0,R_1,R_2\}}\Big(\mathcal{P}_{\{Q_0,0,0\}}\mathbf{x}\Big)\Big)(s) &= \int_a^s R_1(s,\theta)Q_0(\theta)\mathbf{x}(\theta)d\theta + \int_s^b R_2(s,\theta)Q_2(\theta)\mathbf{x}(\theta)d\theta = \Big(\mathcal{P}_{\{0,P_{10},P_{20}\}}\mathbf{x}\Big)(s),
\end{aligned}$$

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 [10].

□

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 $\mathcal{R} \in \Pi^{\mathrm{m}\times\mathrm{normaln}}$, we first recall the definition of the function space that these operators map: $Z^{\mathrm{n}}\Big[[a,b],[c,d]\Big] := \begin{bmatrix} \mathbb{R}^{n_0} \\ L_2^{n_s}[a,b] \\ L_2^{n_r}[c,d] \\ L_2^{n_2}\big[[a,b]\times[c,d]\big] \end{bmatrix}$, where $\mathrm{n} := \{n_0, n_s, n_r, n_2\}$, 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 $\mathbf{x}_2 \in L_2^{n_2}\Big[[a,b]\times[c,d]\Big]$. 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 $R \in \mathcal{N}^{m\times n}\Big[[a,b],[c,d]\Big]$, there exist unique parameters $Q \in \mathcal{N}^{n\times m}\Big[[a,b],[c,d]\Big]$ such that*

$$\Big(\mathcal{P}[R]\Big)^* = \mathcal{P}[Q],$$

where $\mathcal{P}^*$ denotes the adjoint of a PI operator $\mathcal{P}$. That is, for any $\mathbf{x} \in Z^{\mathrm{n}}\Big[[a,b],[c,d]\Big]$ and $\mathbf{y} \in Z^{\mathrm{m}}\Big[[a,b],[c,d]\Big]$,

$$\langle \mathcal{P}[R]\mathbf{x}, \mathbf{y}\rangle_Z = \langle \mathbf{x}, \mathcal{P}[Q]\mathbf{y}\rangle_Z .$$

Proof. We outline the proof only for 4-PI operators. In particular, let $\mathrm{n} = \{n_0, n_1, 0, 0\}$ and $\mathrm{m} = \{m_0, m_1, 0, 0\}$, and let $B = \left[\begin{smallmatrix} P & Q_1 \\ Q_2 & \{R_0,R_1,R_2\} \end{smallmatrix}\right]$ define a 4-PI operator $\mathcal{P}[B] \in \Pi^{\mathrm{n}\times\mathrm{m}}$. Define $\hat{B} = \left[\begin{smallmatrix} \hat{P} & \hat{Q}_1 \\ \hat{Q}_2 & \{\hat{R}_0,\hat{R}_1,\hat{R}_2\} \end{smallmatrix}\right]$, 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 $\mathbf{x} = \left[\begin{smallmatrix} x_0 \\ \mathbf{x}_1 \end{smallmatrix}\right] \in Z^{\mathrm{n}}$ and $\mathbf{y} = \left[\begin{smallmatrix} y_0 \\ \mathbf{y}_1 \end{smallmatrix}\right] \in Z^{\mathrm{m}}$, we note that

$$\langle \mathcal{P}[B]\mathbf{x}, \mathbf{y}\rangle_Z = [Px_0]^T y_0 + \left[\int_a^b Q_1(s)\mathbf{x}_1(s) ds\right]^T y_0 + \int_a^s [Q_2(s)x_0]^T \mathbf{y}_1(s) ds + \int_a^b [R_0(s)\mathbf{x}_1(s)]^T \mathbf{y}_1(s) ds$$

$$
\begin{aligned}
&+ \int_a^b \left[\left(\int_a^b \mathbf{I}(s-\theta) R_1(s,\theta) + \mathbf{I}(\theta - s) R_2(s,\theta) \right) \mathbf{x}_1(\theta) d\theta \right]^T \mathbf{y}_1(s) ds \\
&= x_0 [P^T y_0] + x_0^T \left[\int_a^b Q_2^T(s) \mathbf{y}_1(s) ds \right] + \int_a^b \mathbf{x}_1^T(s) \left[Q_1^T(s) y_0 \right] ds + \int_a^b \mathbf{x}_1^T(s) \left[R_0^T(s) \mathbf{y}_1(s) \right] ds \\
&\quad + \int_a^b \mathbf{x}_1^T(s) \left[\left(\int_a^b \mathbf{I}(-\theta - s) R_1(\theta, s) + \mathbf{I}(s-\theta) R_2(\theta, s) \right) \mathbf{y}_1(\theta) d\theta \right] ds \\
&= \left\langle \mathbf{x}, \mathcal{P}[\hat{B}] \mathbf{y} \right\rangle_Z .
\end{aligned}
$$

□

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 address the problem of invertibility, required to constructing the controllers and estimators from the feasible solutions to the LPIs described in the main text. For example, the controller gain $\mathcal{K}$ is given by the relation $\mathcal{K} = \mathcal{Z}\mathcal{P}^{-1}$. $\mathcal{P}^{-1}$, although is a 4-PI operator, may not have polynomial parameters. Hence the inverse is approximated numerically. To find the inverse of a 4-PI operator, $\mathcal{P}\begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix}$, we first find the inverse of 3-PI operators of the form $\mathcal{P}_{\{R_i\}}$.

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 $F_1 : [a, b] \to \mathbb{R}^{n \times p}$, $G_1 : [a, b] \to \mathbb{R}^{p \times n}$, $F_2 : [a, b] \to \mathbb{R}^{n \times q}$, $G_2 : [a, b] \to \mathbb{R}^{q \times n}$ and U is the unique function that satisfies the equation*

$$
U(s) = I_{(p+q)} + \int_a^s \begin{bmatrix} G_1(t) F_1(t) & G_1(t) F_2(t) \\ -G_2(t) F_1(t) & -G_2(t) F_2(t) \end{bmatrix} U(t) dt,
$$

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 $\mathcal{P}_{\{I,-F_1G_1,-F_2G_2\}}$ is invertible if and only if $U_{22}(b)$ is invertible and

$$(\mathcal{P}_{\{I,-F_1G_1,-F_2G_2\}})^{-1} = \mathcal{P}_{\{I,L_1,L_2\}},$$

where

$$L_1(s,t) = \begin{bmatrix} F_1(s) & F_2(s) \end{bmatrix} U(s)V(t) \begin{bmatrix} G_1(t) \\ -G_2(t) \end{bmatrix} - L_2(s,t), \tag{A.4a}$$

$$L_2(s,t) = -\begin{bmatrix} F_1(s) & F_2(s) \end{bmatrix} U(s)PV(t) \begin{bmatrix} G_1(t) \\ -G_2(t) \end{bmatrix}, \tag{A.4b}$$

$$P = \begin{bmatrix} 0_{p\times p} & 0_{p\times q} \\ U_{22}(b)^{-1}U_{21}(b) & I_q \end{bmatrix},$$

and V is the unique function satisfying the equation

$$V(t) = I_{(p+q)} - \int_a^t V(s) \begin{bmatrix} G_1(s)F_1(s) & G_1(s)F_2(s) \\ -G_2(s)F_1(s) & -G_2(s)F_2(s) \end{bmatrix} ds.$$

Proof. Proof can be found in [3, Chapter IX.2]. □

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 at discrete spatial points. Them, a polynomial that best fits the data, in a least-squares sense, can be used as an approximation. Thus, we find an approximated inverse for 3-PI operators of the form $\mathcal{P}_{\{I,R_1,R_2\}}$ where R_i are matrix-valued polynomials. By extension, given an invertible R_0, we can obtain the inverse of a general 3-PI operator as shown below.

Corollary 8. *Suppose $R_0 : [a,b] \to \mathbb{R}^{n\times n}$, $R_1, R_2 : [a,b]^2 \to \mathbb{R}^{n\times n}$, 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].

□

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 $\mathcal{P}\begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix}$. Given P, Q_1, Q_2 and R_i with invertible P and R_0. For this aim, we assume invertibility of the 3-PI operator $\mathcal{P}_{\{R_i\}}$ in the following result.

Corollary 9. *Suppose $P \in \mathbb{R}^{m\times m}$, $Q_1 : [a,b] \to \mathbb{R}^{m\times n}$, $Q_2 : [a,b] \to \mathbb{R}^{n\times m}$ $R_0 : [a,b] \to \mathbb{R}^{n\times n}$ and $R_1, R_2 : [a,b]^2 \to \mathbb{R}^{n\times n}$ are matrices and matrix-valued polynomials such that $\mathcal{P}^{-1}_{\{R_i\}}$ is invertible according to Cor. 8 and call $\mathcal{P}_{\{\hat{R}_i\}} := \mathcal{P}^{-1}_{\{R_i\}}$. Then, $\mathcal{P} := \mathcal{P}\begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix} \in \Pi_4$ is invertible if and only if the matrix*

$$T = P - \mathcal{P}\begin{bmatrix} \emptyset, & Q_1 \\ \emptyset, & \{\emptyset\} \end{bmatrix} \mathcal{P}_{\{\hat{R}_i\}} \mathcal{P}\begin{bmatrix} \emptyset, & \emptyset \\ Q_2, & \{\emptyset\} \end{bmatrix}$$

is invertible. Furthermore

$$\mathcal{P}^{-1} = \mathcal{U}\mathcal{P}\begin{bmatrix} T^{-1}, & 0 \\ 0, & \{\hat{R}_i\} \end{bmatrix} \mathcal{V},$$

where

$$\mathcal{U} = \mathcal{P}\begin{bmatrix} I, & 0 \\ 0, & \{\hat{R}_i\} \end{bmatrix} \mathcal{P}\begin{bmatrix} I, & 0 \\ -Q_2, & \{R_i\} \end{bmatrix},$$
$$\mathcal{V} = = \mathcal{P}\begin{bmatrix} I, & -Q_1 \\ 0, & \{\hat{R}_i\} \end{bmatrix} \mathcal{P}\begin{bmatrix} I, & 0 \\ 0, & \{\hat{R}_i\} \end{bmatrix}.$$

Proof. A proof may be found in Sec. VII of [9]. □

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 $\big(\mathcal{P}\mathbf{v}\big)(s) = P(s)\mathbf{v}(s)$ is a PI operator, but there clearly does not exist a PI operator $\mathcal{Q}$ such that $\partial_s\big(\mathcal{P}\mathbf{v}\big)(s) = \big(\mathcal{Q}\mathbf{v}\big)(s)$. Nevertheless, if the function $\mathbf{v}$ is differentiable, i.e. $\mathbf{v} \in H_1$, then we can always express the derivative of $\big(\mathcal{P}\mathbf{v}\big)(s)$ for a PI operator $\mathcal{P}$ in terms of $\mathbf{v}(s)$ and $\partial_s\mathbf{v}(s)$ as $\partial_s\big(\mathcal{P}\mathbf{v}\big)(s) = \Big(\mathcal{Q}\big[\begin{smallmatrix}\mathbf{v}\\ \partial_s\mathbf{v}\end{smallmatrix}\big]\Big)(s)$, as we show in the next lemma.

Lemma 10. *Suppose $\mathcal{P}\begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix} : \mathbb{R}^m \times H_1^n \to \mathbb{R}^p \times L_2^q$, and define $\partial_s\mathcal{P}\begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix} : \mathbb{R}^m \times H_1^n \times L_2^n \to \mathbb{R}^p \times L_2^q$ as*

$$\partial_s\mathcal{P}\begin{bmatrix} P, & Q_1 \\ Q_2, & \{R_i\} \end{bmatrix} = \mathcal{P}\begin{bmatrix} 0, & 0 \\ \bar{Q}_2, & \{\bar{R}_i\} \end{bmatrix} \tag{A.5}$$

where $\bar{Q}_2(s) = \partial_s Q_2(s)$, $\bar{R}_0(s) = \begin{bmatrix}\partial_s R_0(s) + R_1(s,s) - R_2(s,s) & R_0(s)\end{bmatrix}$ *and* $\bar{R}_i(s,\theta) = \begin{bmatrix}\partial_s R_i(s,\theta) & 0\end{bmatrix}$ *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

$$\left(\mathcal{P}\left[\begin{smallmatrix} P, & Q_1 \\ Q_2, & \{R_i\}\end{smallmatrix}\right]\mathbf{x}\right)(s) = \begin{bmatrix} Px_0 & + \int_a^b Q_1(s)\mathbf{x}_1(s)ds \\ Q_2(s)x_0 & + R_0(s)\mathbf{x}_1(s) + \int_a^s R_1(s,\theta)\mathbf{x}_1(\theta)d\theta + \int_s^b R_2(s,\theta)\mathbf{x}_1(\theta)d\theta \end{bmatrix} \tag{A.6}$$

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* $Z_1 : [a,b] \to \mathbb{R}^{d_1 \times n}$, $Z_2 : [a,b]\times[a,b] \to \mathbb{R}^{d_2 \times n}$, *if* $g(s) \geq 0$ *for all* $s \in [a,b]$ *and*

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

$$+\int_s^\eta g(\theta)Z_2(\theta,s)^\top T_{34}Z_2(\theta,\eta)\,d\theta+\int_a^s g(\theta)Z_2(\theta,s)^\top T_{44}Z_2(\theta,\eta)\,d\theta,$$
$$R_0(s)=g(s)Z_1(s)^\top T_{22}Z_1(s). \tag{A.7}$$

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.6) *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].

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] J. Doyle, A. Packard, and K. Zhou. Review of LFTs, LMIs, and μ. In *Proceedings of the 30th IEEE Conference on Decision and Control*, pages 1227–1232 vol.2. IEEE, 1991.

[3] I. Gohberg, S. Goldberg, and M. A. Kaashoek. *Classes of linear operators*, volume 63. Birkhäuser, 2013.

[4] Declan S. J. and 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*, 125:109473, 2021.

[6] S. Prajna, A. Papachristodoulou, and P. A Parrilo. Introducing SOSTOOLS: 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] P. 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] S. Shivakumar, A. Das, S. Weiland, and M. Peet. Dual representations and H_∞-optimal control of partial differential equations. *arXiv preprint arXiv:2208.13104*, 2022.

[10] S. Shivakumar, A. Das, S. Weiland, and M. Peet. Extension of the Partial Integral Equation representation to GPDE input-output systems. *IEEE Transactions on Automatic Control*, 2024.

[11] J. 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.