SLSpy: Python-Based System-Level Controller Synthesis Framework

Shih-Hao Tseng and Jing Shuang (Lisa) Li

Abstract—Synthesizing controllers for large, complex, and distributed systems is a challenging task. Numerous proposed methods exist in the literature, but it is difficult for practitioners to apply them—most proposed synthesis methods lack ready-to-use software implementations, and existing proprietary components are too rigid to extend to general systems. To address this gap, we develop SLSpy, a framework for controller synthesis, comparison, and testing.

SLSpy implements a highly extensible software framework which provides two essential workflows: synthesis and simulation. The workflows are built from five conceptual components that can be customized to implement a wide variety of synthesis algorithms and disturbance tests. SLSpy comes pre-equipped with a workflow for System Level Synthesis (SLS), which enables users to easily and freely specify desired design objectives and constraints. We demonstrate the effectiveness of SLSpy using two examples that have been described in the literature but do not have ready-to-use implementations. We open-source SLSpy to facilitate future controller synthesis research and practical usage.

Notation: Let \( \mathcal{R}\mathcal{H}_\infty \) denote the set of stable rational proper transfer matrices, and \( z^{-1}\mathcal{R}\mathcal{H}_\infty \subset \mathcal{R}\mathcal{H}_\infty \) be the subset of strictly proper stable transfer matrices. Lower- and upper-case letters (such as \( x \) and \( A \)) denote vectors and matrices respectively, while bold lower- and upper-case characters and symbols (such as \( u \) and \( \Phi_u \)) are reserved for signals and transfer matrices. We use \( \Phi_u[\tau] \) to denote the \( \tau^{th} \) spectral element of a transfer function \( \Phi_u \), i.e.,

\[
\Phi_u = \sum_{\tau=0}^{\infty} z^{-\tau} \Phi_u[\tau].
\]

For simplicity, we write \((x * y)_{ub}[t]\) as a shorthand notation for the discrete-time finite convolution \( \sum_{\tau=ib}^{uk} x[\tau] y[t-\tau] \).

I. INTRODUCTION

Many of the systems we seek to control are large, complex, distributed, and multi-agent. Controlling such systems is a nontrivial task, and there is a rich body of work surrounding this topic, with numerous proposed controller synthesis methods [1]–[6]. However, these synthesis methods are often inaccessible to engineers and control theorists who want to test them out, customize them, or compare them with other methods. Most methods described in the literature do not come with a ready-to-use software implementation. For those that do, the software may be proprietary and expensive and/or extend poorly to general systems.

State-of-the-art control software often comes in the form of toolboxes. Toolboxes offer off-the-shelf solutions but are not readily extensible to novel methods and customizations. We instead seek a framework for controller synthesis, similar to those available in the robotics domain [7]–[9]. In this domain, frameworks allow for comparison and generalization across a vast diversity of controller algorithms, middleware modules, and hardware components. Frameworks are also found in the networking community, where ns-3 [10] is the unified platform for testing communication protocols.

Inspired by these frameworks, we implement SLSpy, an open-source Python-based framework that serves as a platform for comparison of different discrete-time controller synthesis methods. The framework provides two key workflows: synthesis and simulation. The workflows are built from conceptual components that can be customized to implement a specific synthesis method, and allow for easy extension and modification of existing synthesis methods. SLSpy can accommodate any synthesis algorithm that follows the workflow described in [11].

SLSpy comes pre-equipped with implementations of System Level Synthesis (SLS) and Input-Output Parametrization (IOP) [12]–[14]. This is the first open-source implementation of IOP, and is the first Python-based implementation of SLS; we previously developed a MATLAB toolbox for SLS, SLS-MATLAB [15]. SLSpy includes additional functionality for SLS not found in SLS-MATLAB, such as output feedback and LQG objectives.

The SLS and IOP implementations in SL Spy serve two purposes: they are examples of how the framework’s conceptual components can be easily customized to implement specific synthesis methods, and are also useful algorithms in their own right. The modularized implementations, facilitated by the modularity of the underlying objective and constraints, also make it easy to extend, modify, and apply SLS and IOP.

The main contributions of SL Spy are to introduce a framework for general discrete-time controller synthesis, and to create the first Python implementations of SLS and IOP with modularized objective and constraints. We define and further motivate the necessity of a framework and describe the benefits of modularized objective and constraint design in Section II. The architectures of the framework and the modules are described in Section III. We then demonstrate the usefulness of SL Spy in Section IV through two examples: controller synthesis via IOP and output-feedback LQG. Finally, we summarize and list open questions and extensions for future work in Section V.

II. MOTIVATION AND BACKGROUND

We begin with the preliminaries of frameworks and System Level Synthesis (SLS). We describe the inversion of control concept, which differentiates a framework from a toolbox/library, and explain why a framework could be more
TABLE I  
STATE-OF-THE-ART CONTROL & SIMULATION SOFTWARE

| Software        | Properties                          |
|-----------------|-------------------------------------|
| Simulink [16]   | general                             |
| PSpice [17]     | circuits                            |
| SOSTOOLS [18]   | optimization √                       |
| CVX [19]        | optimization √                       |
| ACADO [20]      | optimization √                       |
| pyRobots [7]    | robotics √                           |
| SMACH [21]      | robotics √                           |
| V-REP [8]       | robotics √                           |
| TuLiP [22], [23]| temporal logic planning √           |
| SLS-MATLAB [15]| controller synthesis √              |
| SLSpy           | controller synthesis √              |

A. Towards Framework: Inversion of Control

Software is a crucial tool in the synthesis and design of controllers for large-scale systems. It allows synthesis algorithms to be distributed and used without the overhead of learning them in detail and implementing them. We summarize some state-of-the-art control and simulation software in Table I. Overall, most of the available tools for controller synthesis are proprietary, which are hard to extend and usually require license (and cost) to use. There are multiple open-source alternatives, but they do not directly serve the purpose of controller synthesis and are often domain-specific. We aim to develop an open-source general-purpose software for controller synthesis.

There are several ways to pack algorithms/methods into software. Two major options are frameworks and libraries/toolboxes. A framework defines some workflows that aggregate various abstract components; the users then customize the workflows of interest by instantiating components with desired behavior. Conversely, a library/toolbox consists of several functions/tools that perform specific actions. The users plan their workflows and choose the functions that fit in their schemes.

A key property that distinguishes frameworks from libraries/toolboxes is the inversion of control [24], [25] (also dubbed the Hollywood Principle – “Don’t call us, we’ll call you”). We illustrate this in the controller synthesis example shown in Fig. 1.

To synthesize a controller, a possible workflow defined by a framework consists of three phases: establishing a system model, specifying the synthesis algorithm, and calculating the controller model. In this framework, the users create their

1 Although SLS-MATLAB is open-source, it requires MATLAB, which is proprietary.
algorithm) to synthesize a new controller (theirController). On the contrary, to extend a toolbox, we would have to add a new function or revise an existing function; this is nontrivial and requires changing the source code of the original toolbox.

B. Modularized System Level Synthesis

Synthesizing an optimal controller for a networked cyber-physical system is challenging. The recently proposed System Level Synthesis (SLS) [12], [13] method provides a solution for the following system:

\[ x[t + 1] = Ax[t] + B_1 w[t] + B_2 u[t], \]
\[ \tilde{z}[t] = C_1 x[t] + D_{11} w[t] + D_{12} u[t], \]
\[ y[t] = C_2 x[t] + D_{21} w[t] + D_{22} u[t], \]

where \( x[t] \) is the state, \( w[t] \) the noise, \( u[t] \) the control, \( \tilde{z}[t] \) the regulated output, and \( y[t] \) the measurement at time \( t \). SLS aims to synthesize a controller, the transfer function \( K \) that maps the state \( x \) or the output \( y \) to the control \( u \), subject to some system-level objective \( g \) and constraint \( S \). To do so, SLS introduces a new parametrization such that by solving

\[
\begin{align*}
\min \ g(\Phi_x, \Phi_u) \\
\text{s.t.} \quad \begin{bmatrix} zI - A & -B_2 \end{bmatrix} \begin{bmatrix} \Phi_x \\ \Phi_u \end{bmatrix} = I,
\end{align*}
\]

for a state-feedback system and

\[
\begin{align*}
\min \ g(\Phi_{xx}, \Phi_{ux}, \Phi_{xy}, \Phi_{uy}) \\
\text{s.t.} \quad \begin{bmatrix} zI - A \\ \Phi_{ux} \end{bmatrix} \begin{bmatrix} \Phi_{xx} & \Phi_{xy} \\ \Phi_{ux} & \Phi_{uy} \end{bmatrix} = \begin{bmatrix} I \\ 0 \end{bmatrix},
\end{align*}
\]

for an output-feedback system, we can derive the controllers of the corresponding systems by

state-feedback: \( u = (\Phi_u - \Phi_{ux} \Phi_{xx}^{-1} \Phi_{xy}) x \),

output-feedback: \( u = (\Phi_{uy} - \Phi_{ux} \Phi_{xx}^{-1} \Phi_{xy}) y \).

For simplicity, we denote by \( \Phi \) the set of SLS parameters, i.e., \( \{ \Phi_x, \Phi_u \} \) or \( \{ \Phi_{xx}, \Phi_{ux}, \Phi_{xy}, \Phi_{uy} \} \).

A key feature of SLS is that it enforces the constraint \( S \) explicitly through the optimization. As such, it decouples the solving procedure from the structure of constraints. This entanglement greatly confined the capability of legacy methods, e.g., [26]–[28], to approach only certain constraints and systems. With SLS, we can now specify the constraints freely and let the corresponding convex program determine the feasibility. To facilitate the usage of SLS, we have developed and released the SLS-MATLAB toolbox [15], [29].

From a practical perspective, users of SLS care more about obtaining a controller that meets their specifications than about the details of the underlying optimization. User-specified requirements on the controller correspond to objectives and constraints in the SLS problem. Motivated by this, we propose to automate the synthesis process and modularize the objectives and constraints as shown in Fig. 3. This allows the users to specify the synthesis type (state-feedback or output-feedback) and select, customize, or even combine their desired objective and constraint modules. The framework then carries out the synthesis and generates the controller model for the users. Through modularization, we aim to make SLS more accessible to not only researchers but also control practitioners.

III. ARCHITECTURE

We design SLSpy, a software framework for system-level controller synthesis. Our framework addresses the controller synthesis problem at the system level; component-wise details are omitted and the system is described by a map between its sensors and actuators.

The extensibility of a framework relies on its ability to instantiate conceptual components in pre-defined workflows. We can realize instantiation through inheritance in software; for this reason, we implement our framework in Python, an object-oriented language with good support for inheritance. An additional benefit of Python is that it is open-source and commonly used, which makes our framework more accessible. We remark that the concepts in this paper are not Python-specific; our framework can be implemented in any programming language that supports inheritance or some equivalent instantiation process.

Below, we illustrate the details of our framework and its SLS modules.

A. Framework Overview

To design a framework for system-level controller synthesis, we focus on two essential workflows: synthesis and simulation, as shown in Fig. 4. We further partition the workflows into five core conceptual components: SystemModel, SynthesisAlgorithm, ControllerModel, NoiseModel, and Simulator. The synthesis workflow takes a SystemModel and synthesizes a desired ControllerModel. The simulation workflow allows users to verify the behavior of the resulting ControllerModel fed back to the SystemModel, and examine the impact of external disturbances from the NoiseModel.
We design the simulation workflow to lie in the time domain. As a result, all conceptual components should handle and produce time domain signals with the exception of SynthesisAlgorithm. This design decision allows the components to collaborate with real cyber-physical systems. For example, with appropriate hardware-software interfaces, a ControllerModel can generate control signals to control a real system; a physical controller can be tested with different SystemModels; a NoiseModel can serve as a noise generator for robustness tests of real systems. For flexibility of the synthesis workflow, we allow SynthesisAlgorithm to deal with the frequency domain. Overall, our framework maintains flexibility to accommodate as many future synthesis algorithms as possible.

We explain the functions of each component below:

**SystemModel**, interfered by noise \( w \), takes control input \( u \) to generate state \( x \), measurement \( y \), and regulated output \( z \). A SystemModel could have internal states, which allows it to model a wide range of systems, including general linear or nonlinear, time-invariant or time-varying ones.

**ControllerModel** receives the measurement \( y \) (which equals to \( x \) under state-feedback schemes) to produce control input \( u \). ControllerModel is flexible to accommodate a wide range of parametrizations of the represented controller. For example, we can parametrize the class of linear time-invariant controllers in ControllerModel by a direct map \( K \) from \( y \) to \( u \), the Youla parameter \( Q \) [26], or the SLS parametrization [12], [13], which uses closed-loop maps from state disturbances and measurement error to the state and input (i.e., \( \Phi \)). ControllerModel contains procedures that turn measurement \( y \) into control \( u \) in time domain according to the parameters.

**SynthesisAlgorithm** takes a SystemModel and synthesizes a ControllerModel according to its design parameters and constraints. In conventional toolboxes, such as TuLiP [22], [23] and SLS-MATLAB [15], the synthesis algorithm and controller model are often coupled. However, for the framework, we separate the two for better extensibility and reduced code duplication. For example, there are many ways to design a controller \( K \), including LQR and pole-placement methods. These are two separate synthesis algorithms corresponding to the same controller model; the code for the controller model would be duplicated if we combined the synthesis algorithm and controller model.

**NoiseModel** models some disturbance or noise processes. A key design decision we made is to exclude NoiseModel from the synthesis workflow, and hence from the SystemModel and SynthesisAlgorithm. Indeed, some synthesis algorithms may assume and target specific classes of noise (e.g. Gaussian noise), but we argue that the assumptions should be part of their synthesis parameters. We instead include NoiseModel in the simulation workflow, so that we can examine the system performance under different external disturbances. Of course, users are free to choose a NoiseModel that agrees with their assumptions.

**Simulator** simulates time-domain system behavior for a specific system (SystemModel) and controller (ControllerModel) in the presence of noise (NoiseModel), and outputs the resulting history of state \( x \), measurement \( y \), regulated output \( z \), control \( u \), and noise \( w \). Users can then analyze the history, visualize it using our pre-written visualization tools, and compare simulations from different controllers.

We include the Simulator as a separate entity from the SystemModel for extensibility; when the system is known, the coupling between SystemModel and Simulator is apparent. However, for applications with plant uncertainty or related to system identification, the SystemModel used in design is not necessarily the same as the true system, which the Simulator uses. Separating the SystemModel and Simulator also allows us to test a single controller on a variety of systems.

### B. System Level Synthesis Modules

As illustrated in Section II-B, SLS provides a new parametrization for both the formulation of the synthesis problem and the corresponding controller models. Fig. 5 shows how SLS is implemented within the SLSpy framework via inheritance; below, we describe the details of the implementation.

**Constraints and Objectives** Given an LTI System, the SLS algorithm formulates an optimization problem with some specified objective \( g \) and constraint set \( \mathcal{S} \). As stated in Section II-B, we want to allow the user to specify arbitrary combinations of objectives and constraints. To this end, we
include the SLS_Objective and SLS_Constraint base classes. We then maintain two lists, as shown in Fig. 6, to keep track of the user-selected modules, which are derived from SLS_Objective and SLS_Constraint. Below we explain how to combine those modules to form the corresponding SLS optimization problem.

A naive assumption for combining objectives is that the overall objective \( g(\Phi) \) is the sum of objective modules \( g_i \), i.e., \( g(\Phi) = \sum_i g_i(\Phi) \). However, this is not the most general expression, and may lead to issues with more complex objectives. We instead make the more general assumption that the objective modules can modify the cumulative objectives from previous modules. Specifically,

\[
g(\Phi) = \ldots g_3(\Phi, g_2(\Phi, g_1(\Phi, 0)))
\]

To demonstrate the flexibility of this structure, we consider the following objective as an example. Consider

\[
g(\Phi) = \alpha \left\| \begin{bmatrix} C_1 & D_{12} \\ \Phi_x & \Phi_u \end{bmatrix} \right\|_{\mathcal{H}_2}^2 + \left\| \Phi_x \right\|_{\mathcal{L}_2}^2,
\]

which can be decomposed as

\[
g(\Phi) = g_3(\Phi, g_2(\Phi, g_1(\Phi, 0)))
\]

where

\[
g_1(\Phi, h) = \left\| \begin{bmatrix} C_1 & D_{12} \\ \Phi_x & \Phi_u \end{bmatrix} \right\|_{\mathcal{H}_2}^2 + h,
\]

\[
g_2(\Phi, h) = \alpha h,
\]

\[
g_3(\Phi, h) = \left\| \Phi_x \right\|_{\mathcal{L}_2}^2 + h.
\]

Besides \( \Phi \) and \( h \), each objective module can also take its own parameters to cover a larger class of objectives, e.g.,

\[
g_{\mathcal{H}_2}(\Phi, C_1, D_{12}, h) = g_1(\Phi, h), \quad g_{mul}(\Phi, \alpha, h) = g_2(\Phi, h).
\]

We obtain \( g \) by iterating through the SLS_Objective list and performing function compositions. Correspondingly, SLS_Objective must include a function for function composition.

Combining arbitrary constraints is trivial; we maintain a list of constraints and allow constraint modules to add to the list. Correspondingly, SLS_Constraint should include a function that adds its constraint to the list. The core SLS feasibility constraints \((1)\) for state feedback and \((2a), (2b)\) for output feedback) can be included as modules; they are generally applicable except in the case of robust SLS [30]. Some SLS problems (e.g. robust SLS) are defined using a combination of new variables defined via equality constraints, and regularization terms on these new variables in the objective. In these cases, the constraint must be defined before the objective; for this reason, SLS_Constraint inherits from SLS_Objective.

Controllers SLS proposes controller realization in block diagrams, which are in the frequency domain. The block-diagram realization of the SLS output feedback controller is shown in Fig. 7. However, as described in Section III-A, the ControllerModel requires functionality in the time domain. This necessitates the translation of Fig. 7 into time-domain equations for implementation.

Fig. 7 corresponds to the time-domain equations

\[
u[t] = (I + \Phi_{uy}[0]D_{22})^{-1} \left( u'[t] + \Phi_{uy}[0]y[t] \right)
\]

where the internal states are

\[
u'[t] = (\Phi_{ux} \ast \beta)_1[t - 1] + (\Phi_{uy} \ast \bar{y})_1^T[t - 2],
\]

\[
\beta[t + 1] = - (\Phi_{xx} \ast \beta)_2[t - 2] - (\Phi_{xy} \ast \bar{y})_1^T[t - 1],
\]

\[
\bar{y}[t] = y[t] - D_{22}u[t].
\]

SLSpy implements the output-feedback SLS controller in time domain as defined in (3), as well as the state-feedback standard SLS controller in [11].

IV. EXAMPLES

Through the following examples, we demonstrate how SLSpy can help the user perform and study controller synthesis with ease. All codes used for the examples are available online at [31].

A. Setup

For all examples, we use a 10-node fully-actuated chain-like system, as shown in Fig. 8, with the following tridiagonal

![Fig. 8](image-url)
A matrix:

$$A = \begin{bmatrix}
0.4 & 0.1 & 0 & \cdots & 0 \\
0.1 & 0.3 & \ddots & \vdots & \vdots \\
0 & \ddots & \ddots & \vdots & \vdots \\
\vdots & \ddots & \ddots & 0.3 & 0.1 \\
0 & \cdots & \ddots & 0.1 & 0.4 \\
\end{bmatrix} \tag{4}$$

The system is stable, with a spectral radius of 0.5. We zero-initialize the system and disturb it at time 0 with an impulse disturbance $w[0] = 10$. Under different controller models, we record the quantities of interest and plot their time series in log scale.

**B. Input-Output Parametrization**

The design of SLSpy framework allows the user to implement novel synthesis algorithms with ease. For example, a new parametrization – Input-Output Parametrization (IOP) – is proposed in [14] for the following system:

$$y = Gu + P_{yw}w, \quad z = P_{zu}u + P_{zw}w$$

where $y$, $u$, and $w$ are the measurement (system output), control, and noise, respectively. Given a transfer function $G$, IOP obtains the controller $K = YX^{-1}$ for $u = Ky$ by solving

$$\min \| [P_{zw} + P_{zu}YP_{yw}] \|
\text{ s.t. } [I - G] \begin{bmatrix} X & W \\ Y & Z \end{bmatrix} = [I 0]
\begin{bmatrix} X & W \\ Y & Z \end{bmatrix} [I - G] = [0 0]
X, W, Y, Z \in \mathcal{RH}_\infty.$$

We implement IOP in SLSpy with only 282 lines of code, and demonstrate the effectiveness of the IOP controller in Fig. 8. Fig. 9 shows a simple example where the disturbance hits the center of a chain-like system of 10 nodes. While the disturbance spreads, the IOP controller reacts and stabilizes the system.

**C. Output Feedback LQG**

We implement an LQG controller with nonzero expected measurement noise. The SLS formulation of LQG can be found in [32]. Since the framework decouples the expected noise (which is a parameter for controller synthesis) and the actual noise (which is used in the simulator), we can simulate the response of the controller to noises it was not designed for. We include a simulation of the LQG controller in a system with no measurement noise in Fig. 10 and a simulation of the same controller in a system with measurement noise in Fig. 11.

Compared to the IOP controller, the LQG controller allows the disturbance to spread more in both time and space. Since the LQG controller expects measurement noise, it does not act as aggressively on sensor information as the IOP controller, which expects no measurement noise. A fairer comparison would be comparing the IOP controller with the LQR controller, and in that case, we find that the two controllers are identical, which matches the discussion in [33].

**V. Conclusion**

We propose a software framework for controller synthesis and implement it as SLSpy in Python. Our framework serves as a platform for comparison of different discrete-time controller synthesis methods. We describe the architecture of the framework and its supported workflows, and use it to deploy...
modularized implementations of two synthesis methods that previously had no open-source implementations.

A direction for future work is exploring how additional optimization solvers and techniques can be incorporated into the framework. Currently, all objectives and constraints are directly specified in CVX syntax. One possible solution is the inclusion of a translator component between objectives/constraints and the solver.

References

[1] A. Nayar, A. Mahajan, and D. Teneketzis, “Decentralized stochastic control with partial history sharing: A common information approach,” *IEEE Trans. Autom. Control*, vol. 58, no. 7, pp. 1644–1658, 2013.

[2] B. Bamieh and P. G. Voulgaris, “A convex characterization of distributed control problems in spatially invariant systems with communication constraints,” *Systems and Control Letters*, vol. 54, no. 6, pp. 575–583, 2005.

[3] M. Rotkowitz and S. Lall, “A characterization of convex problems in decentralized control,” *IEEE Trans. Autom. Control*, vol. 50, no. 12, pp. 1984–1996, 2005.

[4] Y. C. Ho and K. C. Chu, “Team Decision Theory and Information Structures in Optimal Control Problems-Part I,” *IEEE Trans. Autom. Control*, vol. 17, no. 1, pp. 15–22, 1971.

[5] B. Bamieh, F. Paganini, and M. A. Dahleh, “Distributed control of spatially invariant systems,” *IEEE Trans. Autom. Control*, vol. 47, no. 7, pp. 1091–1107, 2002.

[6] A. Mahajan et al., “Information structures in optimal decentralized control,” in *Proc. IEEE CDC*, 2012, pp. 1291–1306.

[7] S. Lemaignan, A. Hosseini, and P. Dillenbourg, “PYROBOTS, a toolset for robot executive control,” in *Proc. IEEE IROS*, vol. 2015-Decem, 2015, pp. 2848–2853.

[8] E. Rohmer, S. P. Singh, and M. Freese, “V-REP: A versatile and scalable robot simulation framework,” in *Proc. IEEE VR*, 2013, pp. 1321–1326.

[9] H. Bruyninckx et al., “The BRICS component model: A model-based development paradigm for complex robotics software systems,” in *Proc. ACM SAC*, 2013, pp. 1758–1764.

[10] ns-3. [Online]. Available: https://www.nsnam.org.

[11] S.-H. Tseng and J. Anderson, “Deployment architectures for cyber-physical control systems,” in *Proc. IEEE ACC*, jul 2020.

[12] Y.-S. Wang, N. Matni, and J. C. Doyle, “A system level approach to controller synthesis,” *IEEE Trans. Autom. Control*, vol. 34, no. 8, pp. 982–987, 2019.

[13] J. Anderson et al., “System level synthesis,” *Annual Reviews in Control*, vol. 59, no. 12, pp. 3238–3251, 2019.

[14] L. Furieri et al., “An input-output parametrization of stabilizing controllers: Amidst Youla and system level synthesis,” *IEEE Control Systems Letters*, vol. 3, no. 4, pp. 1014–1019, 2019.

[15] J. S. L. Li and S.-H. Tseng, “SLS-MATLAB toolbox: Do-it-yourself system level synthesis [poster],” in *Proc. IEEE ACC*, jul 2020.

[16] Simulink. [Online]. Available: https://www.mathworks.com/products/simulink.html.

[17] Pspice. [Online]. Available: https://www.orcad.com/products/orcad-pspice-designer/overview.

[18] S. Prajna, A. Papachristodoulou, and P. A. Parrilo, “Introducing SOSTOOLS: A general purpose sum of squares programming solver,” in *Proc. IEEE CDC*, IEEE, 2002, pp. 741–746.

[19] M. Grant and S. Boyd, “CVX: Matlab software for disciplined convex programming,” http://cvxr.com/cvx Mar. 2014.

[20] B. Houska, H. J. Ferreau, and M. Diehl, “ACADO toolkit - An open-source framework for automatic control and dynamic optimization,” *Optimal Control Applications and Methods*, vol. 32, no. 3, pp. 298–312, 2011.

[21] J. Bohren and S. Cousins, “The SMACH high-level executive,” *IEEE Robotics and Automation Magazine*, vol. 17, no. 4, pp. 18–20, 2010.

[22] T. Wongpiromsarn et al., “TuLiP: A software toolbox for receding horizon temporal logic planning,” in *Proc. ACM HSCC*, 2011, pp. 313–314.

[23] I. Filippidis et al., “Control design for hybrid systems with TuLiP: The temporal logic planning toolbox,” in *Proc. IEEE CCA*. IEEE, 2016, pp. 1030–1041.

[24] R. E. Johnson and B. Foote, “Designing reusable classes,” *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22–35, 1988.

[25] M. Fowler. Inversion of control. [Online]. Available: https://martinfowler.com/bliki/InversionOfControl.html.

[26] D. C. Youla, H. A. Jabr, and J. J. Bongiorno Jr., “Modern Wiener-Hopf design of optimal controllers – part II: The multivariable case,” in *Proc. IEEE ACC*, vol. 21, no. 3, 2016, pp. 319–338.

[27] A. Lamperski and J. C. Doyle, “Dynamic programming solutions for decentralized state-feedback LQG problems with communication delays,” in *Proc. IEEE ACC*, 2012.

[28] Ş. Sabău and N. C. Martins, “Youla-like parametrizations subject to Q1 subspace constraints,” *IEEE Trans. Autom. Control*, vol. 59, no. 6, pp. 1411–1422, 2014.

[29] Compilation of System Level Synthesis Codes. [Online]. Available: https://github.com/sls-caltech/ls-code.

[30] N. Matni, Y. S. Wang, and J. Anderson, “Scalable system level synthesis for virtually localizable systems,” in *Proc. IEEE CDC*, 2018, pp. 3473–3480.

[31] SLSpy. [Online]. Available: https://github.com/shih-hao-tseng/SLSpy.

[32] Y. S. Wang and N. Matni, “Localized LQG optimal control for large-scale systems,” in *Proc. IEEE ACC*, vol. 2016-July, 2016, pp. 1954–1961.

[33] Y. Zheng et al., “On the equivalence of Youla, system-level and input-output parameterizations,” *arXiv preprint arXiv:1907.06256*, 2019.