© 2045 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Lodestar: An Integrated Embedded Real-Time Control Engine*

Hamza El-Kebir[1], Joseph Bentsman[2], Melkior Ornik[3]

***Abstract*— In this work we present *Lodestar*, an integrated engine for rapid real-time control system development. Using a functional block diagram paradigm, Lodestar allows for complex multi-disciplinary control software design, while automatically resolving execution order, circular data-dependencies, and networking. In particular, Lodestar presents a unified set of control, signal processing, and computer vision routines to users, which may be interfaced with external hardware and software packages using interoperable user-defined wrappers. Lodestar allows for user-defined block diagrams to be directly executed, or for them to be translated to overhead-free source code for integration in other programs. We demonstrate how our framework departs from approaches used in state-of-the-art simulation frameworks to enable real-time performance, and compare its capabilities to existing solutions in the realm of control software, emphasizing the convenience of using Lodestar in low-level control system design and implementation. To demonstrate the utility of Lodestar in real-time control systems design, we have applied Lodestar to implement a real-time torque-based controller for a robotic arm. To compare the algorithm design approach in Lodestar to a classical ground-up approach, we have developed a novel autofocus algorithm for use in thermography-based localization and parameter estimation in electrosurgery and other areas of robot-assisted surgery. We use this example to illustrate that Lodestar considerably eases the design process. We also show how Lodestar can seamlessly interface with existing simulation and networking framework in a number of simulation examples.**

I. Introduction

Cyber-physical systems often exhibit complex dynamic behaviors that necessitate the adoption of advanced control laws and their safe and efficient implementation. Many of these systems lie at the intersection of a variety of disciplines, with their control systems often requiring custom software and hardware solutions tailored to the specific application. As an example, one may consider autonomous vehicles, which require an understanding of the drivetrain systems, vision systems and computer vision algorithms for parsing the surroundings, and control and decision algorithms to guide the vehicle [1]. In such an application alone, one finds a need to incorporate low-level control algorithms for the drivetrain, as well as complex computer vision algorithms for image segmentation, vehicle classification and localization, and a high-level decision algorithm that ultimately guides the vehicle [2]. Another example, which is pertinent in the latter part of this work, lies in robot-assisted surgery, specifically in relation to *electrosurgery*, a widely used surgical technique. Here, one aims to provide real-time diagnostics to surgeons, a challenge that lies at the intersection of computer vision, signal processing, and control [3], while also requiring the capability of passing sensitive information in a secure manner between networked systems to safeguard the patient's medical information.

Both in industry and academia, problems such as those described above are almost exclusively solved using custom-built solutions, with little software standardization. While solutions such as Simulink present themselves as *de facto* industry standards [4], proprietary software interfaces and a lack of portability often make it hard to obtain immediately executable code from these frameworks. Moreover, during prototyping, it is possible that some of the hardware or software used in a system does not have pre-existing interfaces with the control design software of choice, necessitating the need for developing custom interfaces and wrappers. In practice, one would therefore either need to call on a team of engineers proficient in each of the subfields and their associated software stacks, or become proficient in all of the required software packages themself [5]. Recognizing this major hurdle in rapid control software prototyping and deployment, the objective of this work is to introduce a new open-source control engine, called *Lodestar*[1], that aims to address issues in hardware and software abstractions while still providing real-time performance levels. Lodestar addresses two key challenges: (i) enabling user-extensible abstraction of domain-specific hardware and software functionalities as part of a functional block diagram description of a system, and (ii) producing directly executable code for rapid prototyping, including networking capabilities. Lodestar is implemented in ISO C++11 to maximize platform support, and is licensed under the permissive BSD 3-clause license[2]. Fig. 1 shows an overview of the Lodestar engine.

This paper is structured as follows. Related work is presented in Sec. II. In Sec. III, we discuss Lodestar's core architecture, as well as the design philosophy that underpins the engine. Then, in Sec. IV, two applications are given to illustrate the ease of use of Lodestar, as well as to compare its use to conventional ground-up prototyping approaches: (i) an overview of real-time performance of Lodestar as applied to

*Research reported in this publication was supported by the National Institute of Biomedical Imaging and Bioengineering of the National Institutes of Health under award number R01EB029766, as well as the National Aeronautics and Space Administration under award number 80NSSC21K1030.

[1]H. El-Kebir (corresponding author) is with the Dept. of Aerospace Engr. at the University of Illinois Urbana-Champaign, Urbana, IL 61801, USA elkebir2@illinois.edu

[2]J. Bentsman is with the Dept. of Mech. Sci. and Engr. at the the University of Illinois Urbana-Champaign, Urbana, IL 61801, USA jbentsma@illinois.edu

[3]M. Ornik is with the Dept. of Aerospace Engr. and the Coordinated Sci. Lab. at the the University of Illinois Urbana-Champaign, Urbana, IL 61801, USA mornik@illinois.edu

[1]https://ldstr.dev

[2]https://github.com/helkebir/Lodestar/blob/master/LICENSE

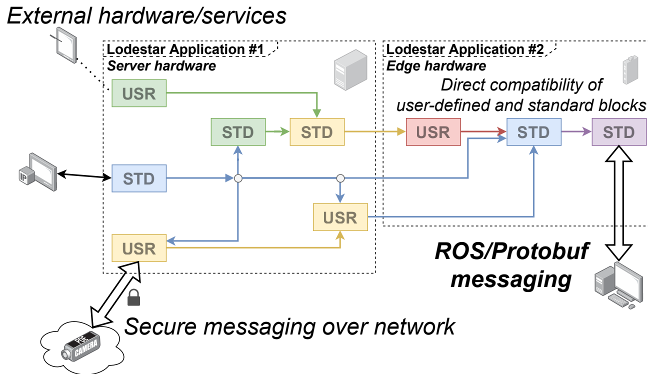

Fig. 1: Overview of the Lodestar engine. A description of the complete system is provided by the user (`USR`), with internal execution order and data dependencies being resolved automatically through the standard package (`STD`). External hardware and software services can easily be interfaced with, using existing protocols or a custom encrypted communication protocol. Lodestar directly provides a baseline implementation that is lightweight and efficient enough to execute directly on target hardware, without modifications or significant overhead.

joint tracking on a robotic arm, (ii) an application of Lodestar to real-time autofocus of an infrared thermographer.

II. Related Work

Over the past decade, the introduction of the Robot Operating System (ROS, [6]) has made a significant impact on the way experimental robotics is practiced, providing a unified ecosystem for message passing between software and hardware services. This has afforded users and vendors alike to abstract away hardware specifics, instead providing "services" that operate on their inputs, and produce a desired effect as a result. Users and other services may then subscribe to specific *topics* (channels) in which these results are broadcast, and may broadcast their own messages on topics of their choosing. While this publisher/subscriber, or *pub/sub*, framework has been regarded as a gold standard within the field of experimental robotics, it leaves the entirety of the control system synthesis to the end-user. In practice, this has compelled users to develop one-off solutions to their control problems, leaving a single unified solution to be desired, except in the most common of cases, such as PID-based joint control using `ros_control` [7].

When considering robotics simulation frameworks, tools such as Drake [8], MoveIt [9], Gazebo [10], and CARLA [11], have shown to present a great number of features for testing cyber-physical control systems. Since these tools are all simulation-focused, they often apply nondeterministic algorithms such as implicit solvers, which are not feasible in real life unless custom schemes are adopted [12]. As an example, Drake provides implicit Euler, Bogacki–Shampine, and Runge–Kutta schemes[3], all of which require access to the underlying model state at future times [12]; Lodestar chooses to adopt explicit integration schemes, including noise robust integrators. Moreover, a high-fidelity system model often precludes direct migration from a simulation environment to a real-life application, especially when this model is actively used within the controller architecture [12]. Most importantly, these tools do not aim to provide users with a comprehensive modeling toolset to design control systems, but rather supply a simulation framework that accepts the final control signals. As an illustration of this fact, while Drake does provide routines for linear quadratic regulator synthesis and PID controllers [8], it lacks a framework for modular controller design; the same observation can be made for CARLA [11]. This latter point underlines the need of a directly deployable, open-source control engine, in which users can synthesize controllers, test them in simulation, and directly apply them in real-life.

Turning our attention to controller synthesis frameworks, we find a number of specialized toolchains do indeed exist. Recently, NASA has introduced the F Prime framework, which aids in designing system specifications (templates) for flight software stacks [13]. F Prime generates boilerplate code, and leaves the actual logic specification to the user. Such an approach is patently different from Lodestar; we explicitly aim for a design framework that produces directly executable code. Previously, MATLAB's Simulink[4] and Stateflow[5] modeling frameworks were used as a basis for direct code translation [14]; this functionality is now directly provided by their publisher in the form of C/C++ code generation [4]. While Simulink does allow for user-authored custom blocks to be introduced, their input–output structure is limited to a single matrix-type input and output. More importantly, the input and output must not have a *direct algebraic dependency*, a limitation that is not present in Lodestar as is discussed later. Finally, a Simulink model may not be executed on hardware directly without writing custom interfaces that get compiled to the proprietary MEX executable format or using slower MATLAB-based interfaces. In the context of rapid prototyping, any Simulink models would run in a Java virtual machine (JVM) instance, which may incur appreciable time delays when using relatively few MEX functions[6]. Most importantly, direct prototyping in Simulink requires the platform to have a working installation of MATLAB and Simulink, which may quickly become prohibitive on (embedded) edge devices. Our proposed solution only requires an ISO C++11 compliant compiler.

In prior work on executable code generation in the context of control system design, most approaches do not allow for circular data-dependencies, or algebraic loops [15]. In Simulink, algebraic loops may appear in simulation and are promptly solved, but when using code generation capabilities, these loops may not be present in the final system [16]. Algebraic loops often arise in constrained control algorithms, such as hard-constrained model predictive control [17], which are essential in practical applications. While a common workaround to algebraic loops is to introduce a small time delay in one of the components to "break the loop," such an approach removes any analytical robustness

[3]https://drake.mit.edu/doxygen_cxx/group__integrators.html

[4]https://www.mathworks.com/products/simulink.html

[5]https://www.mathworks.com/products/stateflow.html

[6]https://www.mathworks.com/help/coder/ug/best-practices-for-using-mex-functions-to-accelerate-matlab-algorithms.html

guarantees, and may not be feasible under strict execution time constraints to start with [18]. Another simulation framework, CyPhySim [19], allows for algebraic loop solving, but these capabilities are confined to simulating models. In this work we show how Lodestar automatically detects algebraic loops, and produces efficient application-specific solvers that may be directly deployed on embedded hardware. Lodestar is based on a block-diagram-like structure, allowing for rapid prototyping of performant code in an abstracted manner.

III. Concepts & Architecture

In this section, we present several notions that are central in the design of the Lodestar control engine. We also briefly discuss how these concepts have been implemented in Lodestar, as well as how users are able to extend the capabilities of Lodestar.

A. Functional Blocks

Functional blocks, or *operators*, have been extensively recognized as a natural way of modeling data-driven systems [20]. In Lodestar, we permit functional blocks that consist of zero or more inputs, outputs, and parameters (see Fig. 2). In this context, a single input, output, or parameter may refer to a multi-dimensional object such as a matrix, or a more complex structure. Associated with each functional block is a *pure function*, i.e., a function modifying the block's internal state based on the inputs and parameters, exposing the results of this modification in its outputs.

In Lodestar, block inputs are only intended to be altered manually by a user prior to being engaged in an interconnection. Given that Lodestar is implemented in C++, compile-time checks are implemented to ensure that interconnections have the same types (i.e., connected input–output pairs must have the same type), among many other consistency checks (such as dimensions, multiplicability, etc.).

Pure functional blocks can also be viewed as atomic units of a control flow; each functional block provides well-defined behavior which is naturally implementation-agnostic. This allows for direct code generation after the execution order is determined and circular dependencies are resolved. The resolution of execution order and circular dependencies is discussed in a later section.

We distinguish between two main classes of blocks: those that exhibit *direct feedthrough*, and those that do not. The instantaneous output of a direct feedthrough block directly depends on the input (for example, a sum or product), while the instantaneous output of a block that does not have direct feedthrough may only depend on the inputs observed during the last cycle and before (e.g., an integrator or time delay block). Systems that exhibit direct feedthrough are numerous, including networked systems and systems with slow and fast dynamics that are modeled using differential-algebraic equation, such as chemical reactors and aircraft [21].

In Lodestar, a simple system of the form of Fig. 2[7] can be generated using the C++ code shown in Lst. 1. In this

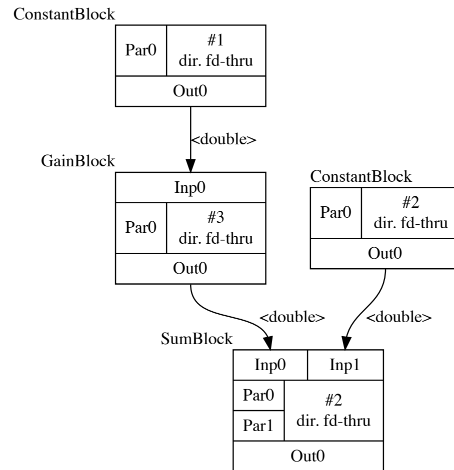

Fig. 2: Illustrative diagram of a system with complete feedthrough.

example, we multiply the first constant by a gain, and sum it with the second constant. As can be seen from the use of templated classes in Lst. 1[8], extensive use of template metaprogramming is made throughout Lodestar, providing specialized function implementations at zero overhead in many cases. Lst. 1 shows how the blocks (`ConstantBlock`, `SumBlock`, `GainBlock`) each allow for their internal type to be defined, as well as the number of inputs in case of the `SumBlock`. A key advantage of Lodestar's design is the fact that many consistency checks may be performed at compile-time. Hidden in the implementation of the `GainBlock` is a static assertion that verifies whether the input is multiplicable with the gain, and the result of that multiplication can be cast to the expected output of the `GainBlock`. Such functionality is key when considering matrix-valued inputs.

```
1  ConstantBlock<double> c{5}, c2{2};
2  SumBlock<double, 2> s;
3  GainBlock<double> g{0.5};
4
5  connect(c.o<0>(), g.i<0>());
6  connect(g.o<0>(), s.i<0>());
7  connect(c2.o<0>(), s.i<1>());
8
9  BlockPack bp{c, c2, s, g};
10 aux::Executor ex{bp};
11 ex.resolveExecutionOrder();
12 ex.trigger();
```

Lst. 1: A sample block diagram in Lodestar.

B. Interconnections and Execution Order

Lst. 1 illustrates the method in which blocks are declared (lines 1–3). Each block inherits from a `BlockProto` object, which provides a virtual `trigger` function, as well as a unique block identifier. The `trigger` function is the only level of run-time polymorphism that Lodestar requires, making it very memory efficient [22].

In Lst. 1, it can be seen how outputs and inputs are interconnected using the `connect` function (lines 5–7).

[7]This figure was auto-generated using Lodestar's internal visualization tools.

[8]A graphical user interface (GUI) companion application similar to Simulink is under development to improve ease of use.

Accessing inputs, outputs, and parameters is much the same as in Modelica [23] or the Functional Mock-up Interface (FMI) standard; the syntax is blk.h<s>(), where h is i, o, or p, and s is the *slot index*. Given the templated implementation of Lodestar, code does not compile if invalid indices are accessed, preventing out-of-range access at runtime.

Before computing the overall execution order, blocks are bundled in a BlockPack object, where each block is reduced to the information provided by the corresponding BlockProto object, after extracting block-specific information (e.g., whether a block has direct feedthrough). With this BlockPack object, we can then initialize an Executor object, which allows Lodestar to determine the execution order. After ordering, the Executor object then provides its own trigger function, which performs one *cycle* of the program in the correct order.

C. Algebraic Loops and Circular Data Dependencies

Unlike most data-driven designs, which only consider systems where the block interconnections form a directed acyclic graph (DAG) [15], [20], functional block models in Lodestar may contain cycles, or loops. Algebraic loops arise from cycles that consist exclusively of blocks with direct feedthrough [17]. In many previous control engines, these algebraic loops are 'removed' by introducing indirect feedthrough blocks, such as time delays[9] [18]. As mentioned in Sec. II, such interventions may significantly alter the validity of the model, and may even cause systems to become unstable [18]. Lodestar departs from such stop-gap solutions, instead providing for the first time a mechanism that produces application-specific nonlinear algebraic equation solvers that can run in real-time, i.e., no general purpose nonlinear algebraic equation solvers are used.

D. Control Routines

Lodestar comes bundled with an extensively tested and documented standard library of blocks, currently counting over twenty blocks dealing with linear control systems, time delays, 2D convolution for images, PID controllers, etc. In addition to existing blocks, custom behavior can quickly be tested using the special FunctionBlock, in which users can easily author their own behavior. Lodestar also provides a number of commonly encountered control routines as standalone functions in C++, where these were, to the best of our knowledge, previously only available in proprietary software, interpreted languages such as Python, and Fortran (e.g., [24]). Currently, synthesis routines (e.g., discrete-time linear quadratic regulators), discretization algorithms (e.g., zero-order hold, bilinear transformations), and conversions between control systems (state-space to transfer functions and vice versa) are provided as part of Lodestar. In addition to these capabilities, Lodestar also allows for nonlinear state space systems to be defined symbolically, automatically generating C++ code that linearizes these systems. Combining these capabilities allows for expedient control system design and implementation, as demonstrated in Sec. IV.

[9] https://www.mathworks.com/help/simulink/ug/remove-algebraic-loops.html

E. Networking & Message Passing

Message passing presents a crucial challenge in control system implementation for distributed systems; this issue is exactly where ROS has found its calling [6]. However, ROS imposes stringent constraints on the project structure and the supported platforms, as well as the way in which messages are defined. This has led to the decision that Lodestar be independent of the ROS ecosystem as a baseline, but still allows for direct integration with ROS applications. A major goal for the networking aspect of Lodestar was that it should support as many platforms as possible, while remaining lightweight and flexible to mesh well with a functional block structure. To this end, Lodestar has adopted Google's Protocol Buffer[10] framework[11]. In particular, messages sent in Lodestar are always preceded by a *herald* message that declares the origin/destination of the message, as well as information about the type of message. This allows Lodestar to discard messages that are wrongly addressed or malformed (e.g., a wrongly sized matrix). Therefore, the required buffer size is known at compile time, since the expected message types are determined by the blocks present in the system.

F. User-defined Extensions

We proceed to briefly demonstrate how users can easily extend Lodestar by authoring custom blocks. All blocks in Lodestar inherit their core functionality from a Block<TInputs..., TOutputs..., TParameters...> class. To add a new block, the user needs to take two steps: (1) inherit from the Block<> class with the desired parameters and define the new block's behavior, and (2) declare a specialization of the BlockTraits class. BlockTraits objects are used to determine whether a block has direct feedthrough, as well as information about the inputs, outputs, and parameters of the block. If these two conditions are met, a user-defined block is treated as a first-class object in the Lodestar engine, allowing users to easily interface with custom software and hardware services.

IV. Applications

In this section, we discuss two applications of Lodestar: torque-control of robotic arm joints and observation control for single-view thermography through focus adjustments.

A. Robotic Arm Joint Torque Control

We first consider the standard problem of joint configuration control using joint torques [25]. Our goal is to show that one can easily construct a proportional–integral–derivative (PID) controller framework with anti-windup and saturation constraints in Lodestar. The classical and projection-based PID controllers are both applied in practice on a

[10] https://developers.google.com/protocol-buffers

[11] ROS messages can be issued and subscribed to using the same Protocol Buffer framework and a lightweight ROS wrapper.

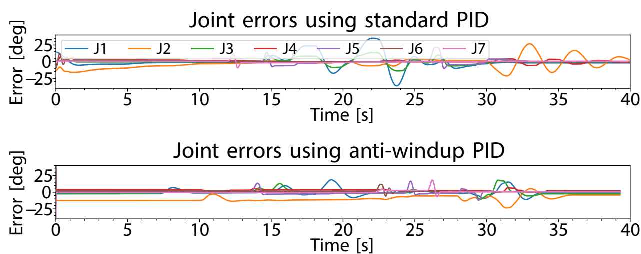

Fig. 3: Joint responses of classical and projection-based anti-windup PID, when subjected to external disturbances, with control algorithms implemented fully in Lodestar.

Franka Emika Panda 7-joint robotic arm, with real-time performance being compared. We will also demonstrate that more advanced controllers can easily be prototyped using Lodestar, e.g., the fuzzy PID controller architecture of [26]. We compare the performance of this latter controller in Lodestar to that of C++ code generated by Simulink Coder, with common use cases for either approach being discussed.

We have implemented all algorithms on a workstation running Ubuntu 20.04 with a fully preemptible real-time kernel, equipped with 128 GB of RAM and a 24-core AMD Ryzen Threadripper 3960X. All code was compiled using GCC 11.2 using C++14 with `O3` optimization; the code is available on GitHub[12]. A custom `PandaArmBlock` class was written by the authors to interface with the `libfranka` C++ drivers. The results for the classical and projection-based anti-windup PID controllers are shown in Fig. 3.

Using the Franka Control Interface (FCI) for the Panda robotic arm, a callback function may not take more than 1 millisecond for more than 5% of all controller calls, imposing strict real-time constraints on the controller implementation. If this constraint is violated, the robotic arm locks its joints and seizing operation out of safety considerations. Using Lodestar, a single evaluation cycle for the classical PID controller takes an average of 12 microseconds with a standard deviation of 1 microsecond. This particular implementation uses the `SimplePIDBlock` class provided as part of Lodestar. To demonstrate the performance of custom controllers composed of multiple blocks, we have implemented the anti-windup PID controller using time delays, sums, gains, and saturation blocks. Despite the addition of separate blocks, we obtain an average cycle time of 32 microseconds with a standard deviation of 11 microseconds. In other words, there is over 930 microseconds of additional free processing time per torque callback in 99.7% of the cases. Additionally, changes to the controller structure and gains can easily be made, allowing for rapid hardware-in-the-loop prototyping without relying on additional hardware or software, other than the compiler toolchain.

To compare performance with generated code, such as provided by the Simulink Coder software, we have considered a single-input-single-output version of the fuzzy PID controller from [26]. The generated code runs at 50 nanoseconds per cycle, while the equivalent code in Lodestar takes 390 nanoseconds on average. However, any code generation

[12]https://github.com/helkebir/Lodestar-Examples

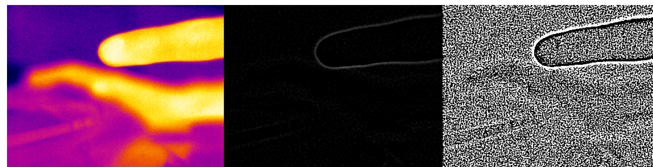

Fig. 4: Comparison of Laplacian filters when applied to thermographer imagery. On the left, the original thermal image is shown. The middle image shows the Laplacian obtained using a noise-robust filter; only the sharpest edges are clearly visible. A Sobel filter was used in the right image; high-frequency noise is strongly amplified.

solution in Simulink Coder needs to be accompanied with additional code for hardware interfacing, as well as custom hooks for interfacing with other software components; neither of these actions are needed when Lodestar is used. Such versatility renders Lodestar more convenient to use compared to existing solutions, with rapid prototyping of different controller architectures becoming a possibility even on modest edge hardware. In addition, code generation takes at least two separate steps: (i) the model has to be *transpiled* to source code, (ii) the generated source code has to be compiled and linked to auxiliary libraries. The former step already requires custom software that can often not run on modest hardware, and the latter step introduces additional vendor-specific code that is often undocumented. In Lodestar, all of the standard blocks are fully transparent and clearly defined in their respective header files, unlike packages that only provide binaries with no access to the original source files to identify potential compatibility issues.

B. Autofocus for Single-view Infrared Thermography

In energy-based surgery methods, thermal damage can often go unnoticed even by skilled physicians, oftentimes resulting in lasting damage that can only be detected days after a surgical procedure [27]. For this reason, our previous work proposes the use of infrared thermography to incorporate temperature information into sensing and control systems for robot-assisted surgery [3], [28]. For the purposes of system identification and control, careful calibration of the focus of a thermographer is essential to ensure that no artificial diffusion is introduced. Unlike in visible light imagery, image sharpness is hard to quantify in thermography [29]. In this work, we use a novel algorithm based on noise-robust filtering to allow for focusing even in the presence of small temperature differences, as seen in electrosurgery [30]. Without presenting the algorithmic details, we refer to the results of our trials are shown in Fig. 5 on two sets of optics. In terms of performance, this particular Lodestar application runs at around 26.92 Hz (limited by the thermographer refresh rate of 27 Hz). If we assume that the thermographer reports its results instantaneously at 27 Hz, we find an average processing time of 100 microseconds, with all processing times lying under 6 milliseconds in 99.7% of the cases, leaving at least 31 milliseconds for subsequent processing tasks.

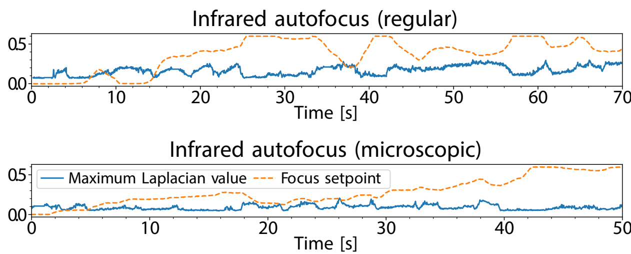

Fig. 5: Thermographic autofocus time series for two different sets of optics: mid-range optics (top), and microscopic optics (bottom). The solid blue line shows the maximum absolute Laplacian value, which is indicative of image sharpness. The orange dashed line is the focus motor set point, which saturates at 0.6. In response to changes in sharpness, our algorithm quickly refocuses in the correct direction. All algorithms are implemented fully in Lodestar.

REFERENCES

[1] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An open approach to autonomous vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, Nov. 2015.

[2] A. M. Madni, M. W. Sievers, J. Humann, E. Ordoukhanian, J. D'Ambrosio, and P. Sundaram, "Model-based approach for engineering resilient system-of-systems: Application to autonomous vehicle networks," in *Disciplinary Convergence in Systems Engineering Research*. Cham, Switzerland: Springer International Publishing, 2018, pp. 365–380.

[3] H. El-Kebir, Y. Lee, R. Berlin, E. Benedetti, P. C. Giulianotti, L. P. Chamorro, and J. Bentsman, "Online hypermodel-based path planning for feedback control of tissue denaturation in electrosurgical cutting," in *11th IFAC Symposium on Biological and Medical Systems*. Ghent, Belgium: IFAC, 2021, pp. 448–453.

[4] A. Rajhans, S. Avadhanula, A. Chutinan, P. J. Mosterman, and F. Zhang, "Graphical modeling of hybrid dynamics with Simulink and Stateflow," in *21st International Conference on Hybrid Systems: Computation and Control*, Porto, Portugal, 2018, pp. 247–252.

[5] J. El-khoury, O. Redell, and M. Torngren, "A tool integration platform for multi-disciplinary development," in *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, Porto, Portugal, 2005, pp. 442–450.

[6] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: An open-source robot operating system," in *2009 IEEE International Conference on Robotics and Automation*, Kobe, Japan, 2009, pp. 5–10.

[7] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodríguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke, and E. Fernandez Perdomo, "{{ros_control}}: A generic and simple control framework for ROS," *The Journal of Open Source Software*, vol. 2, no. 20, p. 456, 2017.

[8] R. Tedrake and the Drake Development Team, "Drake: Model-based design and verification for robotics," https://drake.mit.edu, 2019.

[9] S. Chitta, I. Sucan, and S. Cousins, "Moveit!" *IEEE Robotics & Automation Magazine*, vol. 19, no. 1, pp. 18–19, 2012.

[10] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sendai, Japan: IEEE, 2004, pp. 2149–2154.

[11] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *1st Annual Conference on Robot Learning*, vol. 78, Mountain View, CA, USA, 2017, pp. 1–16.

[12] M. Arnold, B. Burgermeister, and A. Eichberger, "Linearly implicit time integration methods in real-time applications: DAEs and stiff ODEs," *Multibody System Dynamics*, vol. 17, no. 2-3, pp. 99–117, 2007.

[13] R. Bocchino, T. Canham, G. Watney, L. Reder, and J. Levison, "F Prime: An open-source framework for small-scale flight software systems," in *32nd Annual AIAA/USU Conference on Small Satellites*, Logan, Utah, USA, 2018.

[14] H. Hanselmann, U. Kiffmeier, L. Koster, M. Meyer, and A. Rukgauer, "Production quality code generation from Simulink block diagrams," in *1999 IEEE International Symposium on Computer Aided Control System Design*. Kohala Coast, Hawaii, USA: IEEE, 1999, pp. 213–218.

[15] J. Kim and I. Lee, "Modular code generation from hybrid automata based on data dependency," in *9th IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, 2003, pp. 160–168.

[16] A. Benveniste, T. Bourke, B. Caillaud, B. Pagano, and M. Pouzet, "A type-based analysis of causality loops in hybrid systems modelers," *Nonlinear Analysis: Hybrid Systems*, vol. 26, pp. 168–189, 2017.

[17] A. Syaichu-Rohman, R. H. Middleton, and M. M. Seron, "A multivariable nonlinear algebraic loop as a QP with applications to MPC," in *2003 European Control Conference*, Cambridge, United Kingdom, 2003, pp. 1–6.

[18] A. Syaichu-Rohman and R. Middleton, "On the robustness of multivariable algebraic loops with sector nonlinearities," in *41st IEEE Conference on Decision and Control*, Las Vegas, NV, USA, 2002, pp. 1054–1059.

[19] C. Brooks, E. A. Lee, D. Lorenzetti, T. S. Nouidui, and M. Wetter, "CyPhySim: A cyber-physical systems simulator," in *18th International Conference on Hybrid Systems: Computation and Control*. Seattle, Washington, USA: ACM, 2015, pp. 301–302.

[20] M. Wcislik, K. Suchenia, and M. Łaskawski, "Programming of sequential control systems using functional block diagram language," in *13th IFAC and IEEE Conference on Programmable Devices and Embedded Systems*. Cracow, Poland: IFAC, 2015, pp. 330–335.

[21] A. Ordys and A. Pike, "State space generalized predictive control incorporating direct through terms," in *37th IEEE Conference on Decision and Control*, Tampa, FL, USA, 1998, pp. 4740–4741.

[22] S. P. Nath, S. Vikram, and K. Anand, "Better alternative run-time polymorphism in C++: A practical approach in OO design for machine intelligence," in *2012 International Conference on Computer Communication and Informatics*, Coimbatore, India, 2012, pp. 1–5.

[23] S. E. Mattsson, H. Elmqvist, and M. Otter, "Physical system modeling with Modelica," *Control Engineering Practice*, vol. 6, no. 4, pp. 501–510, 1998.

[24] P. Benner, V. Mehrmann, V. Sima, S. Van Huffel, and A. Varga, "SLICOT—A subroutine library in systems and control theory," in *Applied and Computational Control, Signals, and Circuits*. Boston, Massachusetts, USA: Birkhäuser Boston, 1999, pp. 499–539.

[25] I. Cervantes and J. Alvarez-Ramirez, "On the PID tracking control of robot manipulators," *Systems & Control Letters*, vol. 42, no. 1, pp. 37–46, 2001.

[26] H. Malki, D. Misir, D. Feigenspan, and Guanrong Chen, "Fuzzy PID control of a flexible-joint robot arm with uncertainties from time-varying loads," *IEEE Transactions on Control Systems Technology*, vol. 5, no. 3, pp. 371–378, 1997.

[27] D. Palanker, A. Vankov, and P. Jayaraman, "On mechanisms of interaction in electrosurgery," *New Journal of Physics*, vol. 10, no. 12, p. 123022, Dec. 2008.

[28] H. El-Kebir and J. Bentsman, "PDE-based modeling and non-collocated feedback control of electrosurgical-probe/tissue interaction," in *2021 American Control Conference*, New Orleans, LA, USA, 2021, pp. 4045–4050.

[29] K. Dziarski, A. Hulewicz, and G. Dombek, "Lack of thermogram sharpness as component of thermographic temperature measurement uncertainty budget," *Sensors*, vol. 21, no. 12, p. 4013, 2021.

[30] H. El-Kebir, J. Ran, Y. Lee, L. P. Chamorro, M. Ostoja-Starzewski, R. Berlin, and J. Bentsman, "Minimally invasive live tissue high-fidelity thermophysical modeling using real-time thermography," *IEEE Transactions on Biomedical Engineering*, vol. 70, no. 6, pp. 1849–1857, 2023.