ControlSystems.jl: A Control Toolbox in Julia

Fredrik Bagge Carlson, Mattias Fält, Albin Heimerson, Olof Troeng

***Abstract*— ControlSystems.jl enables the powerful features of the Julia language to be leveraged for control design and analysis. The toolbox provides types for state-space, transfer-function, and time-delay models, together with algorithms for design and analysis. Julia's mathematically-oriented syntax is convenient for implementing control algorithms, and its just-in-time compilation gives performance on par with C. The multiple-dispatch paradigm makes it easy to combine the algorithms with powerful tools from Julia's ecosystem, such as automatic differentiation, arbitrary-precision arithmetic, GPU arrays, and probability distributions. We demonstrate how these features allow complex problems to be solved with little effort.**

I. INTRODUCTION

The Julia programming language [1] has, over the last couple of years, revolutionized technical computing. It is a high-level language with mathematically-oriented syntax and semantics, but still achieves execution speeds comparable to C by relying on just-in-time compilation. Julia is free, open source and due to its suitability for technical computing, already has a rich ecosystem with high-quality packages for applied mathematics. Julia's use of *multiple-dispatch* and *duck typing* simplifies code reuse and composability across packages, making it possible to achieve complex functionality with little effort. The power of the Julia language has been demonstrated in numerous applications [2].

`ControlSystems.jl` [3] provides a large set of algorithms for control design and analysis, enabling the control community to leverage the power of the Julia language and its ecosystem. For example, `ControlSystems.jl` makes it easy to tune controller parameters using automatic differentiation and optimize performance subject to uncertainty by propagating probability distributions.

The `ControlSystems.jl` toolbox supports common functionality such as creating linear time-invariant (LTI) systems using either a state-space representation or as transfer functions with either polynomials or zero–pole–gain representations. Systems with delays are supported, with the time-response capabilities provided by the `DifferentialEquations.jl` ecosystem. The toolbox further contains tools for system synthesis using pole placement, lead-lag design, LQR/LQG and $\mathcal{H}_\infty$ synthesis. The toolbox provides frequency-domain visualizations such as Bode, Nyquist, gang-of-four and Nichols plots as well as simulation and plotting methods in the time domain for impulse- and step-responses. Standard analysis tools to compute controllability, observability and Gramian matrices, determine gain and phase margins, calculate $\mathcal{H}_2$ and $\mathcal{H}_\infty$ norms, as well as Riccati- and Lyapunov-equation solvers are also available. Furthermore, functions that facilitate the design and analysis of PID controllers [4], as well as examples that create interactive GUIs to further enhance these tools, are provided. More detailed information can be found at `ControlSystems.jl` [3] and the linked documentation.

Outline: In the next section, we elaborate on why we believe that Julia is a good language for control design. Sec. III provides a terse introduction to types and dispatch in Julia and then the implementation of the toolbox is summarized in Sec. IV. In Sec. V we provide several examples that demonstrate some of the capabilities of the toolbox when used together with the Julia ecosystem. We conclude with an outlook and conclusions.

II. WHY JULIA FOR CONTROL?

In this section, we motivate why we believe that Julia is a suitable language for control design and analysis.

Julia and `ControlSystems.jl` are *free and open-source*, with permissive licenses. This avoids the costs and hassles of licenses, which is a great advantage in both industrial and educational contexts.

Julia is fast. Julia is a compiled language that relies on LLVM to generate machine code. This means that Julia programs can match the computational speed of programs written in C or Fortran, while maintaining a relatively high-level coding style.

Julia, unlike `scipy` or `Matlab`, does not need a low-level language to implement computationally intensive algorithms, hence there is no *two-language barrier*. Since Julia is also open source, this provides unique opportunities to inspect and debug complex algorithms.

The Julia *ecosystem* provides numerous state-of-the-art packages for technical computing, despite Julia being a young language. One reason for this could be that many experts in technical computing were early to realize the potential of Julia, and have made significant contributions within their areas of expertise. Furthermore, there has always been a single, built-in package manager that makes

All authors are or have been with the Department of Automatic Control at Lund University, Sweden. A. Heimerson is still there, while F. Bagge Carlson is with Julia Computing, M. Fält is with North Link AB, Helsingborg, Sweden, and O. Troeng is with Ericsson, Lund, Sweden. All authors are or have been members of the ELLIIT Strategic Research Area at Lund University. E-mail: `baggepinnen@gmail.com`, `mattias@mfalt.com`, `albin.heimerson@control.lth.se`, `olof.troeng@gmail.com`

Code for the examples that are presented in this paper is found at `https://github.com/JuliaControl/CDC2021`

it easy to install packages, manage dependencies, and create new packages equipped with unit-test infrastructure.

Julia has, similarly to Matlab, a *syntax designed for technical computing*, and in particular linear algebra. This makes control-algorithm implementations read similar to their pseudo-code counterparts in the literature. Furthermore, the multiple-dispatch semantics of Julia, described in the next section, lends itself exceptionally well to implementing mathematical algorithms since this is naturally how mathematics is expressed. Below, we give some examples of syntax features that we find helpful for analyzing control system.

- Use of unicode symbols, e.g, as variable names.
  Example: `θ = LinRange(0, 2π, 100)`
  Example: `(1 + √5)/2 ≈ MathConstants.φ # is true`
- List comprehensions and generators.
  Example: `prod((p - z)/(conj(p) + z) for p in poles(L) if real(p) > 0)`
- Ability to index into return values (unlike in Matlab).
  Example: `xfinal = lsim(P, u, t)[3][end]`
- Convenient matrix operations (unlike in Python).
  Example: `[B A*B A^2*B]`
- Proper support for keyword arguments and default argument values.

Interoperability between Julia and other programming languages is facilitated through an easy-to-use C-call interface, and dynamic bidirectional interoperability with automatic type conversion exists between Julia and both Python and R, and to some extent Matlab.

III. TYPES AND DISPATCH IN JULIA

The semantics of Julia was designed to allow for a dynamic, high-level language, while at the same time letting the compiler reason about the code and compile efficient machine code. A key to achieve this is the type system. While the user is not required to reason about types, any invocation of a function leads to compilation of a custom method of that function specialized for the types of *all* the input arguments. On a low level, this makes it easy for the user to write generic code that works for multiple input types, and on a high level, it allows for packages across the ecosystem to work together without explicit co-design.

A. Functions and Methods

To make the concept of specialization more concrete, consider the following trivial Julia function

```
trig_identity(x) = sin(x)^2 + cos(x)^2
```

When called with a numeric argument, it works as expected

```
trig_identity(0.5) # 1.0
```

but it can also be called with symbolic variables

```
using Symbolics
@variables x
trig_identity(x) # cos(x)^2 + sin(x)^2
simplify(trig_identity(x)) # 1
```

That `trig_identity` can be called with an argument of any type for which `sin` and `cos` are defined is an example of *duck typing*, i.e., "if it quacks like a duck, it's a duck". Duck typing helps to improve code reuse, by avoiding unnecessary type restrictions on the arguments.

Duck typing is more widely applicable in Julia than in, e.g., Python, where this function would likely be defined using `numpy.sin` and consequently fail for symbolic arguments since `numpy.sin` is not the same function as `sympy.sin`. In Julia, there is only one function `Base.sin`, but this function has multiple *methods*. Namespacing is not required since a package implementing a new numeric type may also implement a new method for `Base.sin` that works on the new type, and any algorithm that uses `Base.sin / Base.cos` will work for the new type:

```
X = randn(2,2)   # Random 2×2 matrix
trig_identity(X) # = 2×2 identity matrix
```

While the example above was trivial, the same mechanism allows ODE solvers and control-analysis algorithms, that are implemented in terms of array operations, to work with standard, dense arrays of double-precision floats, but also GPU arrays, arrays of dual numbers (for forward-mode automatic differentiation), and block-banded matrices. These algorithms will also work on block-banded matrices of dual numbers that compute on the GPU, because all packages and types in Julia compose automatically.

Because of this composability, a large number of custom types have been developed by the Julia community, some of the most relevant are:

- Extended- and arbitrary-precision numbers.
- Dual numbers for automatic differentiation.
- Intervals for interval arithmetic.
- Numbers representing distributions of uncertain values.
- GPU-arrays for GPU computations.
- Matrices and arrays with structure that can be exploited for performance and memory usage, e.g., diagonal matrices, banded matrices, Toeplitz/Hankel matrices, constant arrays, and sparse arrays.

B. Multiple Dispatch

Many programming languages have basic capabilities for method dispatch on the type of a *single* function argument. However, as discussed in [1], this is often insufficient for maximizing composability and code reuse. For this reason, Julia relies on the paradigm of *multiple dispatch*, where method selection is based on the types of *all* function arguments. Multiple dispatch is fundamental to the success of Julia's well-developed ecosystem.

`ControlSystems.jl` leverages these capabilities by providing generic control algorithms, and allowing arbitrary matrix and `Number` types in system representations. In Sec. V, we show how multiple dispatch, together with traditional duck typing, then enables non-trivial control problems to be solved with little effort, e.g., symbolic control design and control design for uncertain systems.

We claimed in the previous section that multiple-dispatch is the semantics of mathematics. To lend credit to this statement, consider the following three operations

$$\alpha\beta, \quad \alpha v, \quad DE$$

where α, β are scalars, v is a vector and D, E are matrices. The mathematical *operation* here is that of multiplication, but the implementations of the different multiplications are vastly different, one dispatches to scalar multiplication, one to scalar times vector and one dispatches to a matrix-matrix multiplication, the notation, however, is the same. Multiple-dispatch at its core implies that the multiplication *operator*,[1] `*`, is implemented using any number of different *methods*, and which method is called is determined based on *all* arguments, in these cases based on both the left and right operands.

IV. DETAILS ON IMPLEMENTATION

We now present a brief introduction to some of the types and methods that are implemented in the `ControlSystems.jl` toolbox. We also provide some details on the implementation of delay systems and time-domain simulations.

A. Types

The toolbox provides two types for representing rational LTI systems, `TransferFunction` and `StateSpace`. They both have a type parameter `TimeEvolution` that is either `Continuous` or `Discrete`. This allows for efficient dispatch, while requiring only a single method when continuous-time and discrete-time systems behave the same, as is the case with, e.g., addition.

A `TransferFunction` is represented with a matrix of SISO functions, corresponding to each of the input-output channels. Each SISO function can either be a fraction of two `Polynomial`(s) or a zero-pole-gain description. The short-hand constructors `tf` and `zpk` provide a convenient way of constructing these transfer functions.

The `StateSpace` type is represented by the standard A, B, C, D matrices and can be created using the short-hand `ss`. Both of these system types, as well as the `TimeEvolution` field, are parameterized by their numerical type, which leads to great flexibility in terms of numerical accuracy and functionality as we will show in the following sections.

The type `HeteroStateSpace`, which is intended for advanced usage, allows the fields A, B, C, D to be of different array types. This type is useful for, e.g., statically-sized arrays, GPU arrays or arrays with named entries.

The type `DelayLtiSystem` represents (continuous-time) delay systems, it is further described in Sec. IV-C.

B. Algorithms

`ControlSystems.jl` provides a range of standard functions for analyzing, simulating, and converting between different types of LTI systems, as well as functions for control design. Documentation on these functions and their implementations can be found online [3].

`ControlSystems.jl` rely on functionality provided by the Julia ecosystem in several cases.

For plotting, the package `Plots.jl` [5] is used, and the toolbox provides plotting *recipes* such as `bodeplot`, `nyquistplot`, `lsimplot` and `stepplot`. These recipes allow the user to easily visualize system characteristics, while the flexibility of `Plots.jl` in terms of plotting backends and plot customization is retained.

Similarly, for simulation of continuous-time systems the ODE solvers from `DifferentialEquations.jl` [6] are used. Hundreds of integration methods are provided by `DifferentialEquations.jl`, which makes it possible for the user to select a solver based on properties such as stiffness, smoothness, number of states, and desired accuracy. This also allows the user to supply a generic Julia function as input signal and have it treated as a continuous-time signal.

C. Delay Systems

1) Representation: `ControlSystems.jl` relies on the linear fractional transformation (LFT)-based representation described in [7] for modeling delay systems with internal delays. A (rational) linear time-invariant system with r delays $\{\tau_1, \ldots, \tau_r\}$ can be represented by the equations

$$\frac{d}{dt}x = Ax(t) + B_1u(t) + B_2d_\tau(t) \tag{1a}$$
$$y(t) = C_1x(t) + D_{11}u(t) + D_{12}d_\tau(t) \tag{1b}$$
$$d(t) = C_2x(t) + D_{21}u(t) + D_{22}d_\tau(t), \tag{1c}$$

where d is the vector-valued signal entering the time delays and d_τ is the vector-valued signal leaving the time delays, its components are given by $d_{\tau,i}(t) = d_i(t - \tau_i)$. The equations (1) can be represented as a linear fractional transformation as shown in Fig. 1.

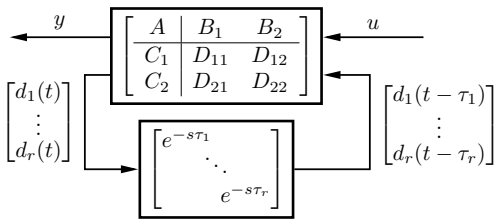

Fig. 1. Representation of an LTI system with delays as a linear fractional transformation.

[1] An operator is equivalent to a function in Julia, i.e., `A*B` is the same as `*(A,B)`. All functions follow the multiple-dispatch semantics.

The type `DelayLtiSystem` contains the partitioned state-space model in Fig. 1 together with the vector of time delays $[\tau_1, \ldots, \tau_r]$.

2) Interconnection of Delay Systems: To compute feedback interconnections of linear fractional transformations we have implemented a very general `feedback` function that efficiently computes the system resulting from connecting arbitrary inputs and outputs. The formulae are very similar to the Redheffer star-product [8], but we have opted for a more symmetric version where the first output of the first system is connected to the first input of the second system, etc. For performance reasons, `ControlSystems.jl` has specialized implementations for series and parallel connections of `DelaySystem`(s).

V. EXAMPLES

A. A First Example

To illustrate basic usage and the delay-system functionality of `ControlSystems.jl`, we reproduce some of the plots from the Smith predictor example in *Advanced PID Control* [4, Ch. 8]. For details, see [4].

The delay-free system $P_0(s) = 1/(s+1)$ can be defined with

```
P₀ = tf(1.0, [1, 1])
```

A PI-controller $C_0(s)$ for $P_0(s)$ that gives closed-loop poles with speed ω_0 and relative damping ζ is obtained with

```
ω₀ = 2; ζ = 0.7
C₀ = placePI(P₀, ω₀, ζ)
```

Given a delay τ, the delayed plant $P(s) = e^{-s\tau}P_0(s)$ and the corresponding Smith predictor $C_{\text{sp}}(s)$ are formed with

```
τ   = 8.0
P   = delay(τ) * P₀
Csp = feedback(C₀, (1 - delay(τ))*P₀)
```

The closed-loop response to step disturbances in the reference signal and at the plant input (Fig. 2) is obtained with

```
G = [feedback(P*C_sp, 1) feedback(P, C_sp)]
lsimplot(G, t -> [1; t >= 15], 0:0.1:40)
```

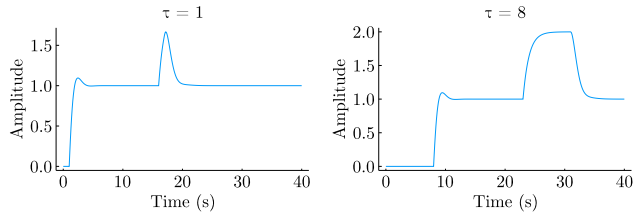

Fig. 2. Closed-loop response of $P(s)$ controlled with $C_{\text{sp}}(s)$ for step disturbances in the reference signal (at $t = 0$) and at the plant input (at $t = 15$), for delays $\tau = 1$ and $\tau = 8$. The plot was generated with the command `lsimplot`.

The Nyquist plot of the open-loop system (Fig. 3) is obtained with

```
nyquistplot(Csp*P, exp10.(-1:1e-4:2))
```

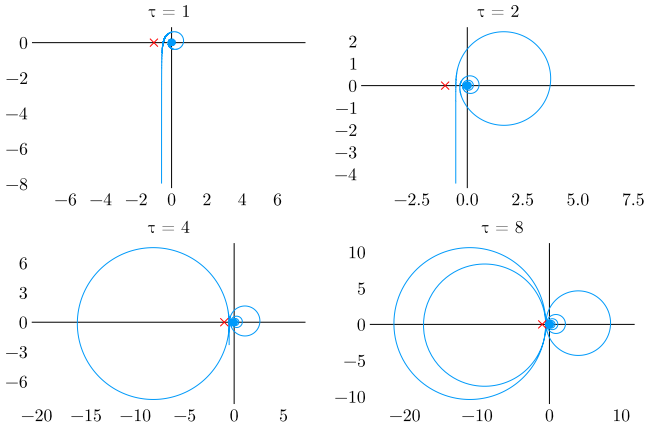

Fig. 3. Nyquist plots for $L(s) = P(s)C_{\text{sp}}(s)$ with $\tau \in \{1, 2, 4, 8\}$. The plots were generated with the command `nyquistplot`.

As discussed in [4], the "predictor part" of $C_{\text{sp}}(s)$,

$$C_{\text{pred}}(s) = \frac{1}{1 + C_0(s)(1 - e^{-s\tau})P_0(s)}, \qquad (2)$$

can be thought of as an approximation to the ideal, non-causal, predictor $e^{\tau s}$. This is illustrated in Fig. 4 where the frequency responses of $C_{\text{pred}}(s)$ and $e^{\tau s}$ are compared with

```
C_pred = feedback(1, C₀*(ss(1.0) - delay(τ))*P₀)
bodeplot([C_pred,delay(-τ)],exp10.(-1:0.002:0.4))
```

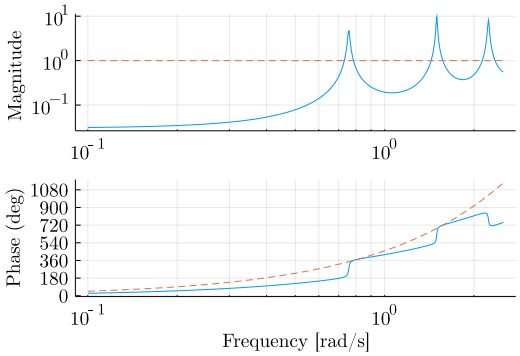

Fig. 4. Illustration of that the phase response of the predictor $C_{\text{pred}}(s)$ in (2) (solid line) is similar to that of the ideal predictor $e^{s\tau}$ (dashed line). The plot was generated with the command `bodeplot`.

B. High-Accuracy Simulation of Delay Systems

We will consider the impulse response of a feedback interconnection between an integrator and a delay. For this system, an extreme-accuracy solution `ytrue` can be computed using an analytical method of steps and 256-bit `BigFloat` numbers. This solution is shown in Fig. 5.

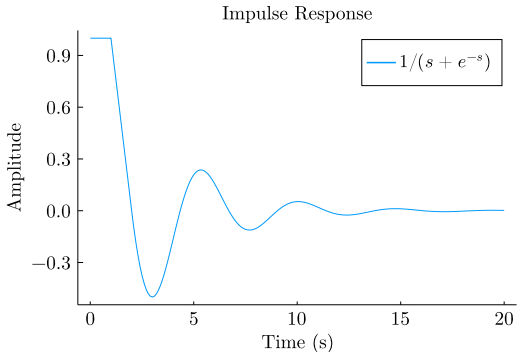

Fig. 5. The impulse response studied in Sec. V-B

The standard approach to simulate this system is

```
s   = tf("s")
sys = feedback(1/s, delay(1))
y,  = impulse(sys, t)
maximum(abs,ytrue-y) # ≈ 8.9e-8
```

Under the hood, the system is simulated using an integrator from the `DelayDiffEq.jl` package. The default settings give a solution `y` with a maximum error from `ytrue` of about 10^{-7}. This can be improved by specifying a lower error tolerance.

```
abstol = reltol = 1e-12
y,     = impulse(sys, t; abstol, reltol)
maximum(abs, ytrue-y) # ≈ 4.5e-13
```

Thanks to the flexible type system in Julia, we can achieve even better accuracy by creating a system with 256-bit `BigFloat` coefficients (the default is 64 bits). All of the following calculations are then made with this precision. By also choosing an integration method suited for high accuracy, e.g., Verner's 9/8 Runge-Kutta method, we obtain a very accurate solution.

```
using OrdinaryDiffEq, DelayDiffEq
sys    = feedback(BigFloat(1)/s,delay(1))
abstol = reltol = 1e-30
alg    = MethodOfSteps(Vern9())
y,     = impulse(sys, big.(t); alg, abstol, reltol)
maximum(abs, ytrue-y) # ≈ 2.4e-32
```

C. Symbolic Computations

As alluded to above, multiple dispatch and duck-typing enable basic computations with LTI systems with symbolic parameters. Symbolic types for different use cases are provided by, e.g., `Symbolics.jl` and `SymPy.jl`.

Example (Kailath [1], Exercise 3.1-1): Find k, δ, γ so that the feedback interconnection of

$$H(s) = \frac{s-0.5}{s(s+1)}$$

and

$$G_c(s) = -k\frac{s+\delta}{s+\gamma}$$

has all its poles in -1.

```
using ControlSystems, SymPy, Polynomials
s = tf("s")
@syms δ k γ

H      = (s - 1//2) / s / (s - 1)
Gc     = -k * (s + δ) / (s + γ)
sys_cl = feedback(H, Gc)
# Get the denominator polynomial
Q = denpoly(sys_cl)[]
# Scale so that leading coeff. equals one
Q_scaled  = Q / Q[end]
Q_desired = Polynomial([1,1])^3
SymPy.solve(coeffs(Q_scaled - Q_desired),
    (δ, k, γ))
```

This returns the correct values $(-1/9, -18, -14)$.

D. Controller Optimization for Uncertain Systems

Model-based control design must be robust to parameter uncertainties due to, e.g., wear and manufacturing tolerances. Existing software tools for control design typically have limited or no support for explicit modeling of parameter uncertainty, as this requires complicated, special-purpose code. `ControlSystems.jl` provides support for uncertainty modeling through composability of its system types and `Number` types representing uncertainty that are provided by other packages.

This section will demonstrate a simple optimization-based design of a PID controller for the system

$$P(s) = \frac{p\omega}{s^2 + 2\zeta\omega s + \omega^2} \tag{3}$$

where the parameters are uncertain with normal distributions $(\mu \pm \sigma)$

$$p = 1 \pm 0.1, \quad \zeta = 0.3 \pm 0.05, \quad \omega = 1 \pm 0.05,$$

The performance criterion is taken to be the mean absolute error in a unit step response, and robustness is ensured by a constraint on the maximum magnitude, M_S, of the sensitivity function S. To avoid excessive control signal activity due to noise, we also penalize the transfer function $G_{un} = CS$. The code below specifies the uncertain parameters using the package `MonteCarloMeasurements.jl` [9], which represents probability distributions as collections of samples, and places them in a standard transfer-function object from `ControlSystems.jl`.

```
using MonteCarloMeasurements,Optim,ControlSystems
±(m,s) = m + s*Particles(200)
unsafe_comparisons(true)
p = 1   ± 0.1
ζ = 0.3 ± 0.05
```

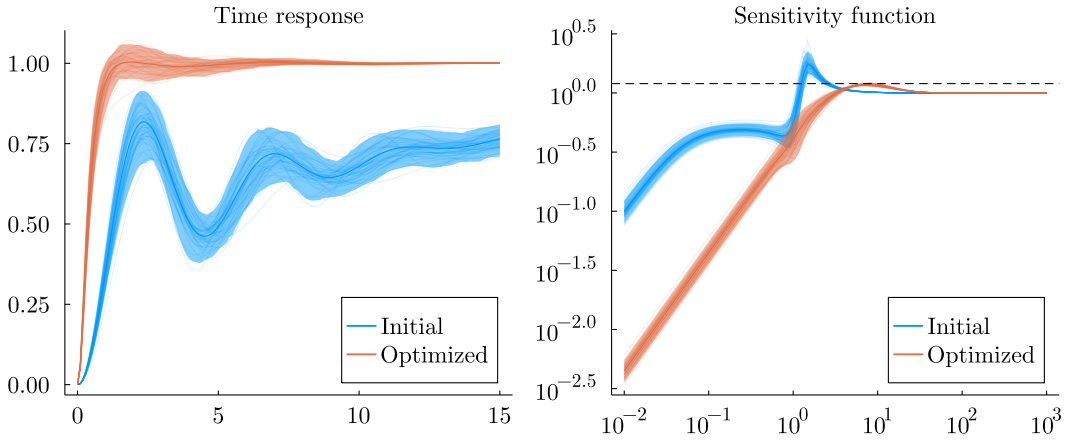

Fig. 6. Controller optimization for uncertain system. Shaded bands include 95% of realizations.

```
ω = 1   ± 0.05
P = ss(tf([p*ω], [1, 2ζ*ω, ω^2]))
Ω = exp10.(-2:0.04:3)
params = log.([1,0.1,0.1]) # Initial guess
Msc    = 1.2 # Constraint on Ms
```

We now define the cost function, which includes time-domain simulation, the frequency-domain constraint on the maximum sensitivity function as well as the noise amplification penalty. Note how we constrain the 90th percentile of the sensitivity function since we are dealing with an uncertain system.

```
function systems(params)
    kp,ki,kd = exp.(params)
    C  = ss(pid(; kp, ki, kd)*tf(1, [0.05, 1])^2)
    G  = feedback(P*C) # Closed-loop system
    S  = 1/(1 + P*C)   # Sensitivity function
    CS = C*S           # Noise amplification
    Gd = c2d(G, 0.1)   # Discretize the system
    y,t = vec.(step(Gd,15)) # Step-response
    C, G, S, CS, y, t
end

function cost(params)
    C, G, S, CS, y, t = systems(params)
    Ms = maximum(bode(S, Ω)[1]) # max sensitivity
    q  = pquantile(Ms, 0.9)
    perf = mean(abs, 1 .- y)
    robust = (q > Msc ? 10000(q-Msc) : 0.0)
    noise = pmean(sum(bode(CS, Ω[end-30:end])[1]))
    100pmean(perf) + robust + 0.002noise
end
```

We are now ready optimize the cost function:

```
res = optimize(cost, params)
```

The time responses and sensitivity functions of the closed-loop system with the controller provided by the initial guess as well as the optimized parameters are shown in Fig. 6. The figure was created by simply calling the function `plot` from `Plots.jl` on the arrays returned from `step` and `bode`. Thanks to the recipe system of `Plots.jl`, uncertain quantities are automatically plotted with confidence intervals and sample trajectories. A large number of types in the Julia ecosystem implements this recipe system and are thus automatically plottable.

In this example, the function `cost` was explicitly written for uncertain systems, using `pquantile` and `pmean` for operations on the distributions of the individual uncertain values. But none of the functions from `ControlSystems.jl`, (`step, c2d, bode` and `tf`) have been explicitly written with uncertain numbers in mind.

E. GPU Computations

Julia's high-level syntax and flexible compiler enable convenient use of hardware accelerators like graphics processing units (GPUs) [10]. This means that Julia code, through multiple dispatch, can often be compiled to optimized GPU kernels by simply changing the array types to some GPU array type, e.g., `CuArray` from the package `CUDA.jl`.

For `ControlSystems.jl`, this means that if the user defines a system using `CuArrays`, calculations with the system will automatically be performed on the GPU. The following example demonstrates this, where the function `cu` is used for converting standard arrays to `CuArrays`.

```
using ControlSystems, CUDA
A   = cu([-1.2 0; 0.1 -0.4]);  B = cu([1.2; 0])
C   = cu([1 0]);               D = cu([0])
sys = HeteroStateSpace(A, B, C, D)
x0  = cu(zeros(2, 1))
u(x,t)  = cu(ones(1, 1))
y, t, x = lsim(sys, u, 0:0.01:5; x0)
```

Not all operations are fast (or even available) on the GPU, which means that GPU usage is only beneficial in certain situations. One situation where GPU acceleration may give significant speedups is for time-domain simulations, as shown by the benchmark in Fig. 7.

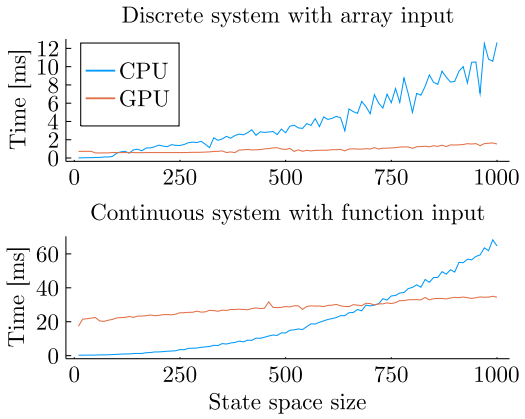

Fig. 7. Comparison of CPU and GPU speed when running `lsim` on continuous-time and discrete-time systems of increasing sizes.

VI. OUTLOOK AND CONCLUSION

Looking forward, we hope to see further use of the toolbox by the control community for solving both theoretical and real-world control problems. We are also welcoming contributions and collaboration for improved functionality.

As the surrounding ecosystem grows and matures, `ControlSystems.jl` may offload some of the heavy numerical lifting to other, special-purpose Julia packages, while focusing on control-related algorithms. There has also been promising work on bringing the advanced modeling and simulation capabilities of acausal modeling tools into Julia [11], [12], [13].

For Julia in general we expect to see a decrease in compilation times, a common annoyance of new adopters of the language. We are also expecting better utilities for static compilation of Julia programs, which will enable convenient deployment to embedded devices.

To conclude, we have introduced the Julia control toolbox `ControlSystems.jl` and demonstrated how its interoperability with the Julia ecosystem enables challenging control problems to be solved with little effort. We hope that you, the reader, will find `ControlSystems.jl` helpful for tackling the control problems that stand in your way.

ACKNOWLEDGMENT

`ControlSystems.jl` was created and originally developed by Jim Crist-Harif in early 2015, before it was adopted by Mattias and Fredrik in 2016. The online community has also provided many valuable contributions in the form of bug reports and code improvements.

REFERENCES

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, 2017.

[2] Julia Computing, "Case Studies — Julia Computing," https://juliacomputing.com/case-studies/, accessed: 2021-05-13.

[3] F. Bagge Carlson, M. Fält, A. Heimerson, and O. Troeng, "ControlSystems.jl: A Control Systems Toolbox in Julia," https://github.com/JuliaControl/ControlSystems.jl.

[4] K. J. Åström and T. Hägglund, *Advanced PID Control*. Research Triangle Park, NC: The Instrumentation, Systems, and Automation Society, 2006.

[5] T. Breloff *et al.*, "Plots.jl: Powerful convenience for Julia visualizations and data analysis," https://github.com/JuliaPlots/Plots.jl.

[6] C. Rackauckas and Q. Nie, "Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia," *Journal of Open Research Software*, vol. 5, no. 1, 2017.

[7] P. Gahinet and L. F. Shampine, "Software for modeling and analysis of linear systems with delays," in *Proceedings of the 2004 American Control Conference*, vol. 6. IEEE, 2004.

[8] K. Zhou, J. C. Doyle, K. Glover, *et al.*, *Robust and Optimal Control*. Englewood Cliffs, NJ: Prentice Hall, 1996.

[9] F. Bagge Carlson, "MonteCarloMeasurements.jl: Nonlinear Propagation of Arbitrary Multivariate Distributions by means of Method Overloading," 2020.

[10] T. Besard, V. Churavy, A. Edelman, and B. D. Sutter, "Rapid software prototyping for heterogeneous and distributed platforms," *Advances in Engineering Software*, 2019.

[11] B. Lie, A. Palanisamy, A. Mengist, L. Buffoni, M. Sjölund, A. Asghar, A. Pop, and P. Fritzson, "OMJulia: An OpenModelica API for Julia-Modelica interaction," 2019.

[12] H. Elmqvist, T. Henningsson, and M. Otter, "Systems modeling and programming in a unified environment based on Julia," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2016, pp. 198–217.

[13] Y. Ma, S. Gowda, R. Anantharaman, C. Laughman, V. Shah, and C. Rackauckas, "Modelingtoolkit: A composable graph transformation system for equation-based modeling," 2021.