Optimization of Controller Parameters in Julia using ControlSystems.jl and Automatic Differentiation

Bagge Carlson, Fredrik

2019

Document Version:
Publisher's PDF, also known as Version of record

Optimization of Controller Parameters in Julia using ControlSystems.jl and Automatic Differentiation

Fredrik Bagge Carlson
fredrikb@control.lth.se

LUND
UNIVERSITY

Department of Automatic Control

Technical Report TFRT-7656
ISSN 0280–5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Abstract

We describe how to utilize the possibility of differentiating through arbitrary Julia code to perform tasks such as controller optimization. The user specifies a cost function, for example, the integrated squared error between output and reference, and constraints, such as a maximum acceptable value of the sensitivity function. Julia performs the integration and calculates the sensitivities of the cost and constraint functions with respect to controller parameters automatically, using automatic differentiation. We conclude with a full example including gradient-based optimization of the cost function. All code required is open-source under permissive licenses.

1. Introduction

Julia [Bezanson et al., 2017] is a modern programming language designed with high-performance numerical computing in mind. As such, it has stellar support for automatic differentiation (AD) [Revels et al., 2016; Innes et al., 2018; Revels et al., 2018]. AD in Julia allows the programmer to write an arbitrary Julia program and have the gradient, Jacobians, Hessian etc. of the output with respect to the input automatically calculated. AD is the main technique for calculating the gradient of the loss function with respect to the parameters in deep learning. However, many deep-learning frameworks require you to manually construct a computation graph using a domain specific language, making the usage of their AD extremely limited. A major benefit of AD compared to finite-difference (FD) approximation is in numerical accuracy. While FD have to make a trade-off between the error arising from too large ϵ in $(f(x+\epsilon)-f(x))/\epsilon$ and numerical errors arising from too small ϵ, AD calculates derivatives to machine precision. Reverse-mode AD further enjoys a theoretical efficiency advantage compared to both forward-mode AD and FD when the differentiate function have high input arity and low output arity, such as a scalar cost function of many parameters.

Julia also offers a vast library of high-performance tools for solving differential equations, written in Julia [Rackauckas and Nie, 2017]. AD is therefore available for differentiation through the integrators [Rackauckas et al., 2018]. In this document, we will outline how the AD functionality available in Julia, together with the ControlSystems.jl [Bagge Carlson and Fält, 2016] and DifferentialEquations.jl [Rackauckas and Nie, 2017] packages, allow for convenient optimization of continuous-time PID controllers, with robustness constraints.

2. ControlSystems.jl

ControlSystems.jl is a Julia toolbox with functionality and syntax similar to the corresponding toolbox in Matlab. Of key difference is the ability of the Julia ControlSystems types, such as `StateSpace` and `TransferFunction`, to hold arrays and numbers of arbitrary types. To illustrate this, consider first the following transfer-function definition

```
julia> using ControlSystems
julia> tf(1, [1., 1., 1.])
TransferFunction{ControlSystems.SisoRational{Float64}}
      1.0
--------------------
1.0*s^2 + 1.0*s + 1.0
Continuous-time transfer function model
```

Julia indicates that the resulting object is a `TransferFunction` with coefficients of type `Float64`. Should we desire, we may instead create a transfer function with uncertain coefficients, and have all calculations done using these coefficients report uncertainties using linear uncertainty propagation (note the uncertainty in the pole locations)

```
julia> using Measurements, GenericLinearAlgebra
julia> ζ = 0.7 ± 0.2             # An uncertain value
julia> G = tf(1, [1., 2ζ, 1])
TransferFunction{ControlSystems.SisoRational{Measurement{Float64}}}
            1.0 ± 0.0
-------------------------------------
1.0 ± 0.0*s^2 + 1.4 ± 0.4*s + 1.0 ± 0.0
julia> pole(G)
2-element Array{Complex{Measurement{Float64}},1}:
 (-0.7 ± 0.2) + (0.71414 ± 0.19604)im
 (-0.7 ± 0.2) - (0.71414 ± 0.19604)im
```

It is now indicated that the type of the coefficients is `Measurement{Float64}`. This type is provided by the package Measurements.jl [Giordano, 2016] and is created using the ± operator (`\pm <TAB>`). Fundamentally, this is what allows us to calculate derivatives of arbitrary functions involving the ControlSystems types. Our exposition will make use of the package ForwardDiff.jl [Revels et al., 2016]. ForwardDiff performs AD using *dual numbers*, an AD technique suitable for functions of few parameters, such as a loss function of a few controller parameters. ForwardDiff was shown by Rackauckas et al. (2018) to be highly efficient at calculating sensitivities through DE solvers when the number of parameters was small (<100).

3. Differentiating through the integrator

In order to allow AD through the integrator, care must be taken to tell the integrator what types to initialize its caches to. In the code below, we store the type of the input in the variable `Tp`. When called with an array `p` of `Float64`, we will have `Tp == Float64`. However, when the function `simulate` is called by the AD package, `p` will be of type `ForwardDiff.Dual`. We also convert the initial guess `x0` and the time span of the integration to this type. In the code, we further construct a function `K` that creates a PID-controller from the three parameters `kp,ki,kd`. The code assumes that the process P is defined somewhere outside `simulate`. In the examples provided below, the process

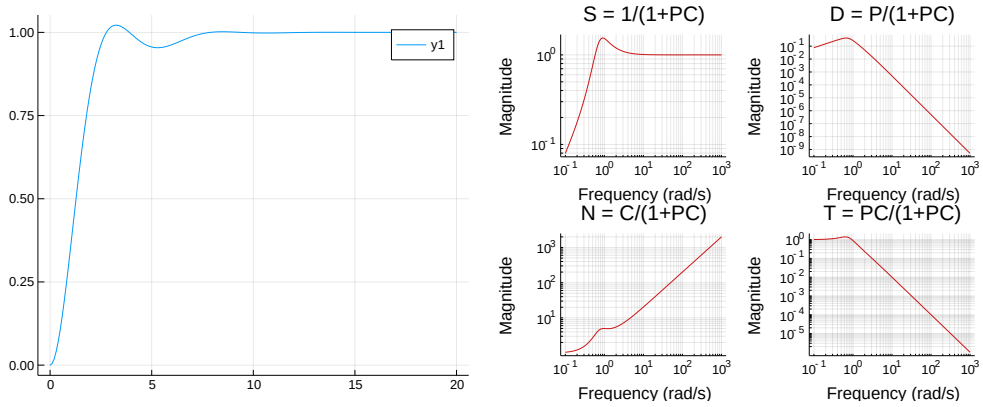
$$P(s) = \frac{1}{(2s+1)^2(0.5s+1)}$$

was used, together with the cost function

$$J(k_p, k_i, k_d) \quad = \quad \frac{1}{T_f}\int_0^{T_f} |r(t) - y(t)|dt \quad = \quad \frac{1}{T_f}\int_0^{T_f} |e(t)|dt$$

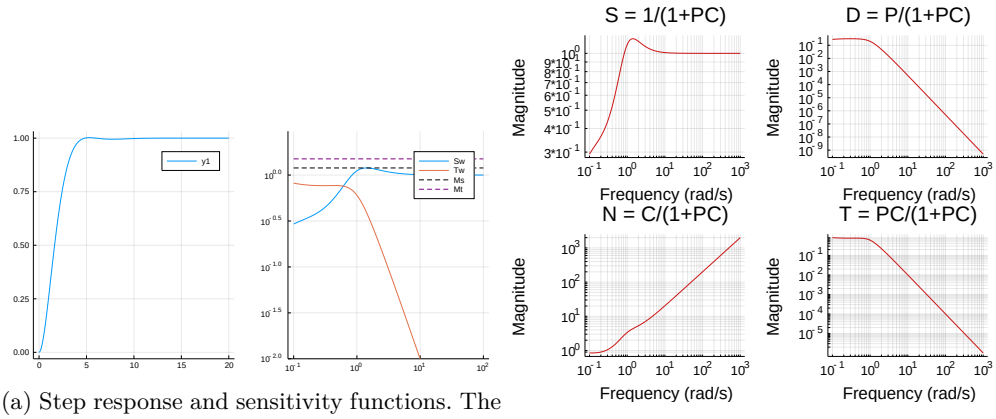
The `Simulator` type is defined in ControlSystems.jl and provides a simple interface between ControlSystems types and the solvers from DifferentialEquations.jl. Its second argument is a function that takes the state and current time and returns the system input.

```julia
using ControlSystems
K(kp,ki,kd) = pid(kp=kp, ki=ki, kd=kd)
K(p)        = K(p...)

function simulate(p)
    C     = K(p[1], p[2], p[3]) # Construct PID controller from parameters
    L     = feedback(P*C) |> ss # Form closed-loop system
    s     = Simulator(L, (x,t) -> [1]) # Sim. unit step load disturbance
    Tp    = eltype(p)           # Store type of input
    x0    = Tp.(zeros(L.nx))    # Initial state of same type as input
    tspan = (Tp(0.),Tp(Tf))     # Sim. time span of same type as input
    sol   = solve(s, x0, tspan) # Simulate the closed-loop system
    y     = L.C*sol(t)          # Output y = C*x
end

function costfun(p)
    y = simulate(p)
    mean(abs, 1 .- y) # ~ Integrated absolute error IAE
end
```



(a) Step response. (b) Gang of four.

Figure 1: Optimization of the cost function without constraints.



(a) Step response and sensitivity functions. The constraints are shown as dashed lines.

(b) Gang of four

Figure 2: Optimization of the cost function with constraints on maximum sensitivity and complimentary sensitivity functions, S and T.

4. Optimizing the cost function

In order to optimize the cost function `costfun` with respect to the parameters `p`, we use the library NLopt [Johnson, 2007]. NLopt wants us to supply a function that takes a vector of parameters and an array used to store the gradient. In Algorithm 1, we define this function, called `f`. The gradient is calculated using ForwardDiff.jl. The function `runopt` calls the optimizer from NLopt using some reasonable settings.

The results of the optimization are show in Figs. 1a and 1b. As we can see, the step-response is nice and fast, but if we inspect the gang of four in Fig. 1b, we see that the sensitivity function S has a rather high peak.

In order to obtain a more robust design, we may add constraints on the maximum value of the sensitivity function, M_S, and the same for the complimentary sensitivity function, M_T. Thankfully, we can differentiate through the Bode-diagram calculations, making the addition of these constraints straightforward. In Algorithm 2, we define the function `freqdomain` that calculates the Bode diagram of S and T, and the function `constraintfun` that specifies that those Bode-diagrams should be below the corresponding threshold for all frequencies.

The optimizer chosen for this task must support nonlinear inequality constraints, the NLopt-optimizers that support this are `:LD_MMA` and `:LD_SLSQP`.[1] NLopt further requires a function `c` that calculates the Jacobian of the constraints.

The result of the constrained optimization is shown in Figs. 2a and 2b. The step response is now slightly smoother and the peak in S is lower. Success!

[1] `https://nlopt.readthedocs.io/en/latest/NLopt_Algorithms/`

In order to keep the example nice and small, we did not include any filters on the measurements. Adding this filter is straight forward. In Sec. 5 we will describe some tips and tricks for adding functionality such as constraints in both time and frequency domains.

Algorithm 1 Optimizing the cost function using NLopt.

```
using ControlSystems, NLopt, ForwardDiff
p           = [0.1, 0.1, 0.1] # Initial guess [kp, ki, kd]
K(kp,ki,kd) = pid(kp=kp, ki=ki, kd=kd)
K(p)        = K(p...)

function simulate(p)
    C     = K(p[1], p[2], p[3]) # Construct PID controller from parameters
    L     = feedback(P*C) |> ss # Form closed-loop system
    s     = Simulator(L, (x,t) -> [1]) # Sim. unit step load disturbance
    Tp    = eltype(p)           # Store type of input
    x0    = Tp.(zeros(L.nx))    # Initial state of same type as input
    tspan = (Tp(0.),Tp(Tf))     # Sim. time span of same type as input
    sol   = solve(s, x0, tspan) # Simulate the closed-loop system
    y     = L.C*sol(t)          # Output y = C*x
end

function costfun(p)
    y = simulate(p)
    mean(abs, 1 .- y) # ~ Integrated absolute error IAE
end

f_cfg = ForwardDiff.GradientConfig(costfun, p)
function f(p::Vector, grad::Vector)
    if length(grad) > 0
        grad .= ForwardDiff.gradient(costfun,p,f_cfg)
    end
    costfun(p)
end

function runopt(p; f_tol = 1e-5, x_tol = 1e-3)
    opt = Opt(:LD_AUGLAG, 3)
    lower_bounds!(opt, 1e-6ones(3))
    xtol_rel!(opt, x_tol)
    ftol_rel!(opt, f_tol)

    min_objective!(opt, f)
    NLopt.optimize(opt, p)[2]
end

p = runopt(p, x_tol=1e-6)
y = simulate(p)
plot(t,y', show=false)
```

4.1 Initial guess

The problem of optimizing the parameters of a PID controller like above is nonconvex, and there is no guarantee that a good solution will be found. As a consequence of this, a good initial guess is required. A global optimization method can be used to provide an initial guess

Algorithm 2 Optimizing the cost function with constraints. The function `c` calculates the constraints and their Jacobian using ForwardDiff.jl.

```
Ω  = exp10.(LinRange(-1,2,150))
p0 = [0.1,0.1,0.1]
function freqdomain(p)
    C  = K(p[1], p[2], p[3])
    S  = 1/(1+P*C)               # Sensitivity fun
    T  = tf(1.) - S        # Comp. Sensitivity fun
    Sw = vec(bode(S,Ω)[1]) # Freq. domain constraints
    Tw = vec(bode(T,Ω)[1]) # Freq. domain constraints
    Sw,Tw
end
Ms = 1.2 # Maximum sensitivity
Mt = 1.5 # Maximum comp. sensitivity

function constraintfun(p)
    Sw,Tw = freqdomain(p)
    [maximum(Sw)-Ms; maximum(Tw)-Mt]
end

g_cfg = ForwardDiff.JacobianConfig(constraintfun, p)

function c(result, p::Vector, jac)
    if length(jac) > 0
        jac .= ForwardDiff.jacobian(constraintfun,p,g_cfg)'
    end
    result .= constraintfun(p)
end

function runopt(p; f_tol = 1e-5, x_tol = 1e-3, c_tol = 1e-8)
    opt = Opt(:LD_SLSQP, 3)
    lower_bounds!(opt, 1e-6ones(3))
    xtol_rel!(opt, x_tol)
    ftol_rel!(opt, f_tol)

    min_objective!(opt, f)
    inequality_constraint!(opt, c, c_tol*ones(2))
    NLopt.optimize(opt, p)[2]
end

p     = runopt(p, x_tol=1e-6)
y     = simulate(p)
Sw,Tw = freqdomain(p)
plot(t,y', layout=2, show=false)
plot!(Ω, [Sw Tw] , lab=["Sw" "Tw"], subplot=2, xscale=:log10, yscale=:log10)
plot!([Ω[1],Ω[end]], [Ms,Ms], c=:black, l=:dash, subplot=2, lab="Ms")
plot!([Ω[1],Ω[end]], [Mt,Mt], c=:purple, l=:dash, subplot=2, lab="Mt", ylims=(0.01,3))
```

if no candidate is previously available. The optimization method SAMIN from Optim.jl[2] is a suitable choice for this. SAMIN does not handle arbitrary nonlinear constraints and such constraints must thus be encoded in the cost-function. This encoding can easily be done using indicator functions, since SAMIN does not require the cost function to be smooth.

5. You may want to..

Most Julia code can be differentiated using AD. Some exceptions include when Julia call out to BLAS or LAPACK, and no generic methods written in Julia are defined. Examples of this are FFT and matrix factorizations. The solution to this problem is to either manually define the Jacobian of the function, or to define a generic fallback method written in Julia.

5.1 Differentiate through FFT

FFT can easily be differentiated by implementing the FFT in Julia. A Julia implementation exists in FastTransforms.jl[3] and it can be used for AD by lifting the type restrictions in the method definitions (by removing `T<:BigFloat`).

5.2 Differentiate through Eigenvalue factorization

Eigen calculations are used in many places in ControlSystems.jl, examples include calculation of poles, frequency-response calculations for state-space systems etc. Julia calls LAPACK to perform these calculations and it is thus not possible to differentiate through these by default. A native Julia implementations of `eigvals` exists in GenericLinearAlgebra.jl[4] (this package was used above to allow the function `pole` to calculate the poles of the system when the transfer function contained coefficients of the special type `Measurement`).

5.3 Add time-domain constraints

This is straightforward, just make `constraintfun` depend on the solution.

5.4 Add filters

A filter is easily added anywhere. As an example, this adds a filter with adjustable bandwidth on the output

```
C  = K(p[1], p[2], p[3])
ω  = p[4]
Cf = C*tf(1, [1, 2ζω, ω^2])
```

6. Notes

The code published in this report is released under the MIT license[5] and was written for Julia v1.0 and ControlSystems.jl v0.5. A notebook with the full code is available at `https://github.com/JuliaControl/ControlExamples.jl/blob/master/autotuning.ipynb`

[2] `http://julianlsolvers.github.io/Optim.jl/stable/#algo/samin/`
[3] `https://github.com/MikaelSlevinsky/FastTransforms.jl/blob/master/src/fftBigFloat.jl`
[4] `https://github.com/JuliaLinearAlgebra/GenericLinearAlgebra.jl`
[5] It means you may use it however you want without restrictions.

References

Bagge Carlson, F. and M. Fält (2016). *ControlSystems.jl : a control systems toolbox for julia*. eng. URL: https://github.com/JuliaControl/ControlSystems.jl.

Bezanson, J., A. Edelman, S. Karpinski, and V. B. Shah (2017). "Julia: a fresh approach to numerical computing". *SIAM Review* **59**:1, pp. 65–98.

Giordano, M. (2016). *Uncertainty propagation with functionally correlated quantities*. eprint: arXiv:1610.08716. URL: https://github.com/JuliaPhysics/Measurements.jl.

Innes, M., E. Saba, K. Fischer, D. Gandhi, M. C. Rudilosso, N. M. Joy, T. Karmali, A. Pal, and V. Shah (2018). *Fashionable modelling with flux*. eprint: arXiv:1811.01457.

Johnson, S. (2007). *The NLopt nonlinear-optimization package*. URL: https://github.com/JuliaOpt/NLopt.jl.

Rackauckas, C. and Q. Nie (2017). "Differentialequations.jl - a performant and feature-rich ecosystem for solving differential equations in julia". *SIAM Review* **5**:1, p. 15. DOI: http://doi.org/10.5334/jors.151. URL: https://github.com/JuliaDiffEq/DifferentialEquations.jl.

Rackauckas, C., Y. Ma, V. Dixit, X. Guo, M. Innes, J. Revels, J. Nyberg, and V. Ivaturi (2018). *A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions*. eprint: arXiv:1812.01892.

Revels, J., T. Besard, V. Churavy, B. D. Sutter, and J. P. Vielma (2018). *Dynamic automatic differentiation of GPU broadcast kernels*. eprint: arXiv:1810.08297.

Revels, J., M. Lubin, and T. Papamarkou (2016). *Forward-mode automatic differentiation in julia*. eprint: arXiv:1607.07892. URL: https://github.com/JuliaDiff/ForwardDiff.jl/.