Control Challenges: Solutions

Iacopo Moles

Table of contents

# 1. Introduction

## 1.1 Intro

This is a collection of write ups on how to solve the various problems presented by [Github user](https://janismac.github.io/ControlChallenges/) “Janismac”.

## 1.2 What do I need?

### 1.2.1 Software

* A real OS like Linux or Windows. [[1]](#footnote-22)
* The [Julia Programming Language](https://julialang.org/install/)
  + Clone the [repo](https://github.com/icpmoles/controlchallengessolutions)
  + Activate the package by running in your terminal:
  + julia --project -e 'using Pkg; Pkg.instantiate()'
* (Nice to have) [OpenModelica Editor](https://openmodelica.org/)

### 1.2.2 Theory

* Basic Julia knowledge
* Basic JS knowledge
* Control Theory knowledge
  + Frequency Based Control
  + State Space Based control
* Misc knowledge:
  + Linear Algebra
  + Differential Equations

# 2. The Damped Mass problem

## 2.1 Modeling

To better understand the problem let’s take a [peek](https://github.com/janismac/ControlChallenges/blob/gh-pages/js/models/BlockOnSlope.js) at how the simulated model works.

Models.BlockOnSlope.prototype.vars =   
{  
 g: 9.81,  
 x: 0, // distance from objective s  
 dx: 0, // velocity v  
 slope: 1, // slope coefficient alpha = dy/dx in the cartesian plane  
 F: 0, // Requested u  
 F\_cmd: 0, // Saturated u  
 friction: 0, // Coulomb friction coefficient mu  
 T: 0, // Simulation Time  
};  
  
Models.BlockOnSlope.prototype.simulate = function (dt, controlFunc)  
{  
 this.F\_cmd = controlFunc({x:this.x,dx:this.dx,T:this.T});  
 if(typeof this.F\_cmd != 'number' || isNaN(this.F\_cmd)) throw "Error: The controlFunction must return a number.";  
 this.F\_cmd = Math.max(-20,Math.min(20,this.F\_cmd));  
 integrationStep(this, ['x', 'dx', 'F'], dt);  
}  
  
Models.BlockOnSlope.prototype.ode = function (x)  
{  
 return [  
 x[1],  
 (x[2]) - (Math.sin(this.slope) \* this.g) - (this.friction \* x[1]),  
 20.0 \* (this.F\_cmd - x[2])  
 ];  
}

Line 2

The model has obviously some default values for the parameters that can be modified for the different scenarios.

Line 17

The control command is generated by the controlFunction(block) function provided by us. There are some checks to see if it’s a number. If it’s acceptable then it passes through a saturation between .

Lines 24-26

The model is a simple ODE with equations:

Converting it in state-space representation:

Obviously the gravitational term acts as a disturbance.

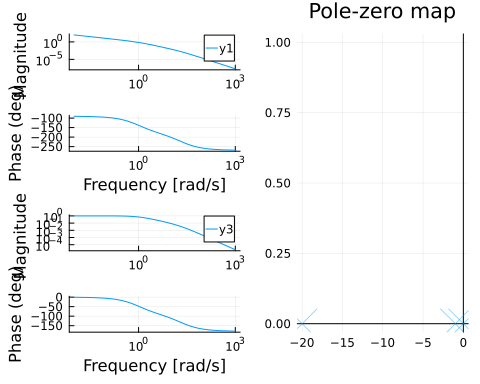
# 3. Block without Friction

Position Control with friction. Using Pole Placement + PD.

## 3.1 State Space representation

We can convert the set of ODE into a state space representation. The final bode plot of the block position is:

using DiscretePIDs, ControlSystems, Plots, LinearAlgebra  
  
# System parameters  
Ts = 0.02 # sampling time  
Tf = 2.5; #final simulation time  
g = 9.81 #gravity  
α = 0.0 # slope  
μ = 1.0 # friction coefficient  
x\_0 = -2.0 # starting position  
dx\_0 = 0.0 # starting velocity  
τ = 20.0 # torque constant   
  
# State Space Matrix  
A = [0 1 0  
 0 -μ 1  
 0 0 -τ];  
B = [0  
 0  
 τ];  
C = [1 0 0  
 0 1 0];  
  
sys = ss(A, B, C, 0.0) # Continuous  
  
plot!(bodeplot(tf(sys)),pzmap(tf(sys)))



Starting Bode Plot and PZ Map

It has the shape we expect from a motor + friction. Slow pole for the mass + friction and a faster pole for the current & inductance.

Numerically they are:

display(eigvals(A))

3-element Vector{Float64}:  
 -20.0  
 -1.0  
 0.0

We see that we start with all the pole in the left-half plane, which is good.

## 3.2 Pole Placement

We can design a controller with pole placement.

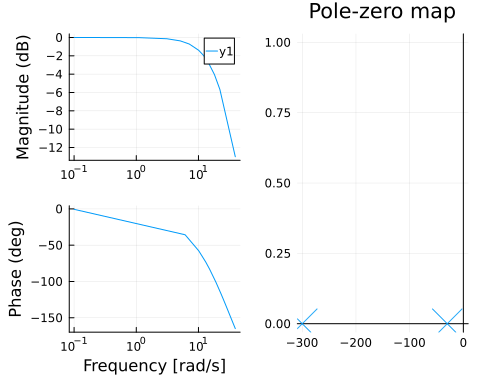
For some reason pole placement doesn’t work for the observer, I use a Kalman Filter with random fast values.

observability(A, C).isobservable &   
controllability(A, B).iscontrollable; #OK  
  
ε = 0.01;  
pp = 15.0;  
poles\_cont = -2.0 \* [pp + ε, pp - ε, pp];  
L = real(place(sys, poles\_cont, :c));  
  
poles\_obs = poles\_cont \* 10.0;  
K = place(1.0 \* A', 1.0 \* C', poles\_obs)'  
cont = observer\_controller(sys, L, K; direct=false);

We can check the effect of the new controller on the loop

closedLoop = feedback(sys \* cont)  
print(poles(closedLoop));  
setPlotScale("dB")  
plot!(bodeplot(closedLoop[1, 1], 0.1:40), pzmap(closedLoop))

ComplexF64[-29.979998924597755 + 0.0im, -30.000002152199972 + 0.0im, -30.019998923202337 + 0.0im, -300.0000000000004 + 0.0im, -300.1999999999752 + 0.0im, -299.80000000004117 + 0.0im]



We can compare this to the open-loop response in @start-bode. We can see that we achieve unitary gain throughout the whole low-frequency range.

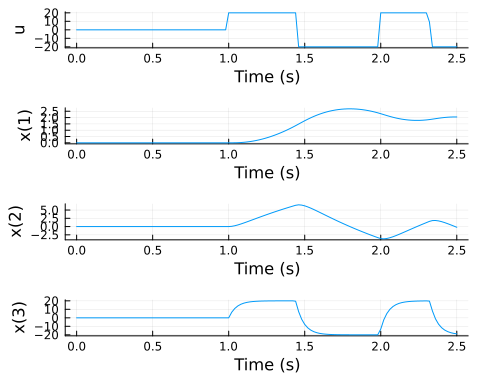
We can convert the pole placement controller into the standard PD gain form.

K = L[1];  
Ti = 0;  
Td = L[2] / L[1];  
pid = DiscretePID(; K, Ts, Ti, Td);

## 3.3 Simulation

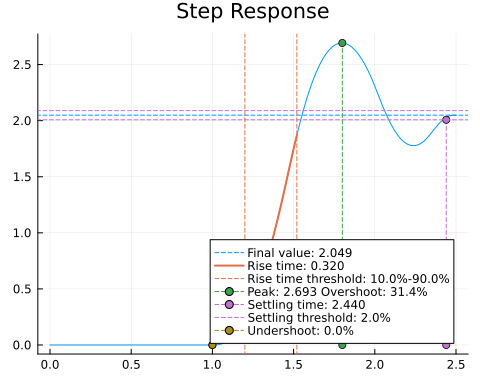
We can simulate this with a motor that only outputs the position:

sysreal = ss(A, B, [1 0 0], 0.0)  
ctrl = function (x, t)  
 y = (sysreal.C\*x)[] # measurement  
 d = 0 \* [1.0] # disturbance  
 r = 2.0 \* (t >= 1) # reference  
 # u = pid(r, y) # control signal  
 # u + d # Plant input is control signal + disturbance  
 # u =1  
 e = x - [r; 0.0; 0.0]  
 e[3] = 0.0 # torque not observable, just ignore it in the final feedback  
 u = -L \* e + d  
 u = [maximum([-20.0 minimum([20.0 u])])]  
end  
t = 0:Ts:Tf  
  
res = lsim(sysreal, ctrl, t)  
  
display(plot(res,   
 plotu=true,   
 plotx=true,   
 ploty=false  
 ))  
ylabel!("u", sp=1);  
ylabel!("x", sp=2);  
ylabel!("v", sp=3);  
ylabel!("T", sp=4);



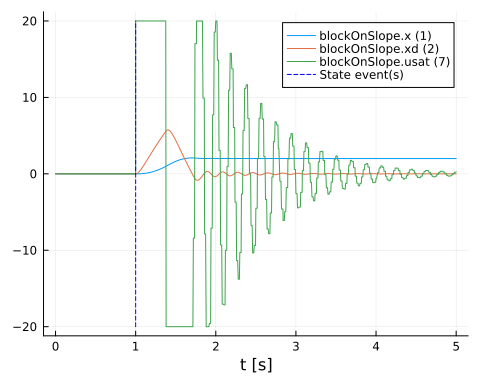
For more stats:

si = stepinfo(res);  
plot(si);title!("Step Response")



We can also simulate it in a SIMULINK-like environment:

using FMI, DifferentialEquations  
fmuPath = abspath(joinpath(@\_\_DIR\_\_,  
 "..","..",  
 "modelica",  
 "ControlChallenges",  
 "ControlChallenges.BlockOnSlope\_Challenges.Examples.WithFriction.fmu"))  
fmu = loadFMU(fmuPath);  
simData = simulateME(  
 fmu,  
 (0.0, 5.0);  
 recordValues=["blockOnSlope.x",   
 "blockOnSlope.xd",   
 "blockOnSlope.usat"],  
 showProgress=false);  
unloadFMU(fmu);  
plot(simData, states=false, timeEvents=false)



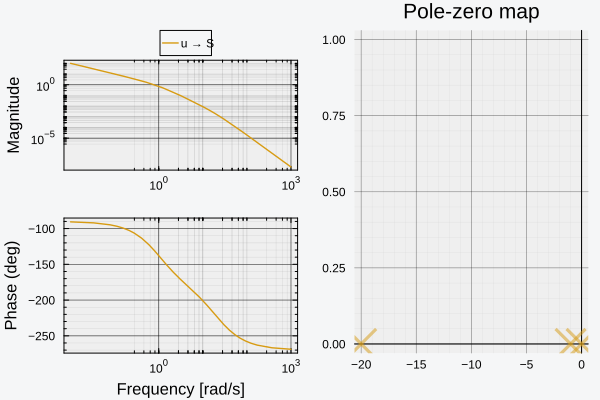
There is a slight difference between the lsim simulation and the FMU simulation. I need to recheck some stuff.

# 4. Block With Friction

Position Control with friction. Using Pole Placement + PD.

## 4.1 Response Analysis

using CCS  
using ControlSystems, Plots, LinearAlgebra, RobustAndOptimalControl  
CCS.setupEnv()  
  
contSys = CCS.blockModel.csys(;g = 0, α = 0 , μ = 1, τ =20)  
plot!(bodeplot(contSys[1,1]),pzmap(contSys))



Starting Bode Plot and PZ Map

It has the shape we expect from a motor + friction. Slow pole for the mass + friction and a faster pole for the current & inductance.

Numerically they are:

display(eigvals(contSys.A))

3-element Vector{Float64}:  
 -20.0  
 -1.0  
 0.0

We see that we start with all the poles in the left-half plane, which is good.

## 4.2 Pole Placement

We can design a controller with pole placement.

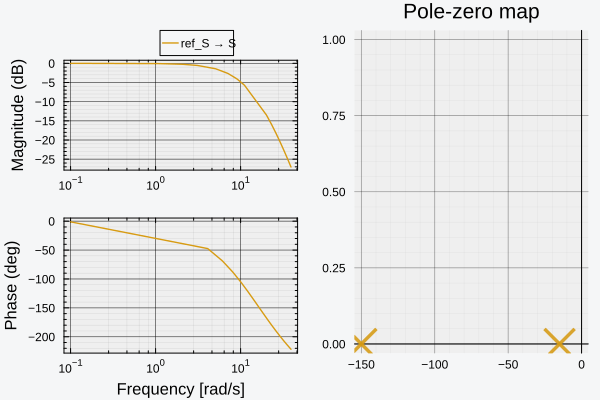
For some reason pole placement doesn’t work for the observer, I use a Kalman Filter with random fast values.

observability(contSys.A,contSys.C).isobservable || error("System is not observable")  
controllability(contSys.A,contSys.B).iscontrollable || error("System is not controllable")  
  
ε = 0.01;  
pp = 15.0;  
poles\_cont = - [pp + ε, pp - ε, pp];  
L = real(place(contSys, poles\_cont, :c));  
  
  
poles\_obs = poles\_cont \* 10.0;  
K = place(contSys, poles\_obs, :o)  
obs\_controller = observer\_controller(contSys, L, K; direct=false);  
fsf\_controller = named\_ss(obs\_controller, u = [:ref\_S, :ref\_V], y = [:u]);

We can check the effect of the new controller on the loop

closedLoop = feedback( contSys \* fsf\_controller);  
print(poles(closedLoop));  
setPlotScale("dB")  
plot!(bodeplot(closedLoop[1,1], 0.1:40), pzmap(closedLoop))

ComplexF64[-14.990000366343788 + 0.0im, -14.999999266673722 + 0.0im, -15.010000366982666 + 0.0im, -149.99999999999986 + 0.0im, -150.10000000002432 + 0.0im, -149.8999999999607 + 0.0im]



We can compare this to the open-loop response in @start-bode. We can see that we achieve unitary gain throughout the whole low-frequency range.

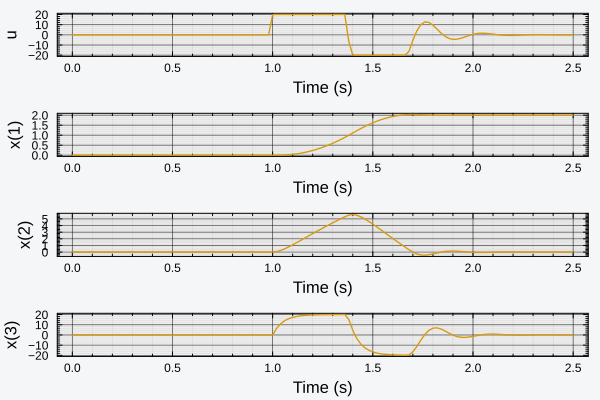
We can convert the pole placement controller into the standard PD gain form.

using DiscretePIDs  
Ts = 0.02 # sampling time  
Tf = 2.5; #final simulation time  
  
K = L[1];  
Ti = 0;  
Td = L[2] / L[1];  
  
pid = DiscretePID(; K, Ts, Ti, Td);

## 4.3 Simulation

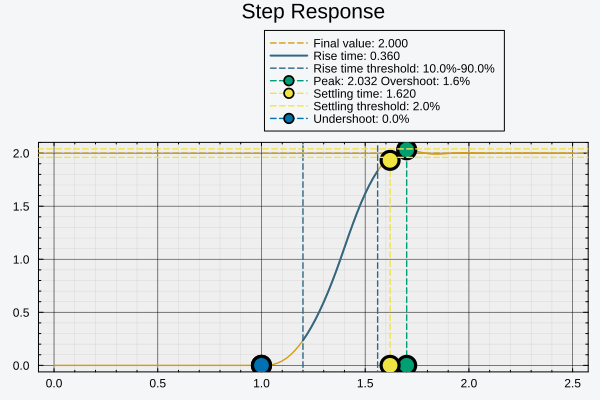
We can simulate this with a motor that only outputs the position:

sysreal = ss(contSys.A, contSys.B, [1 0 0], 0.0)  
ctrl = function (x, t)  
 y = (sysreal.C\*x)[] # measurement  
 d = 0 \* [1.0] # disturbance  
 r = 2.0 \* (t >= 1) # reference  
 # u = pid(r, y) # control signal  
 # u + d # Plant input is control signal + disturbance  
 # u =1  
 e = x - [r; 0.0; 0.0]  
 e[3] = 0.0 # torque not observable, just ignore it in the final feedback  
 u = -L \* e + d  
 u = [maximum([-20.0 minimum([20.0 u])])]  
end  
t = 0:Ts:Tf  
  
res = lsim(sysreal, ctrl, t)  
  
display(plot(res,   
 plotu=true,   
 plotx=true,   
 ploty=false  
 ))  
ylabel!("u", sp=1);  
ylabel!("x", sp=2);  
ylabel!("v", sp=3);  
ylabel!("T", sp=4);



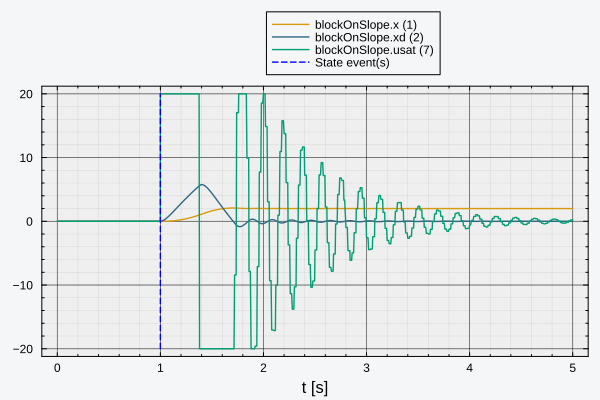
For more stats:

si = stepinfo(res);  
plot(si);title!("Step Response")



We can also simulate it in a SIMULINK-like environment:

using FMI, DifferentialEquations  
fmuPath = abspath(joinpath(@\_\_DIR\_\_,  
 "..","..",  
 "modelica",  
 "ControlChallenges",  
 "ControlChallenges.BlockOnSlope\_Challenges.Examples.WithFriction.fmu"))  
fmu = loadFMU(fmuPath);  
simData = simulateME(  
 fmu,  
 (0.0, 5.0);  
 recordValues=["blockOnSlope.x",   
 "blockOnSlope.xd",   
 "blockOnSlope.usat"],  
 showProgress=false);  
unloadFMU(fmu);  
plot(simData, states=false, timeEvents=false)



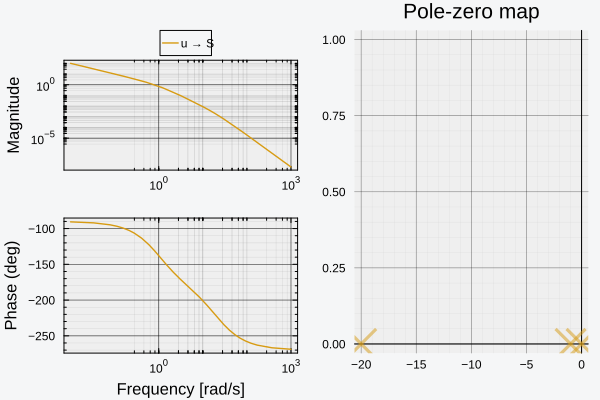
There is a slight difference between the lsim simulation and the FMU simulation. I need to recheck some stuff.

# 5. Block on a slope

Position Control with friction. Using Pole Placement + PD.

## 5.1 Response Analysis

using CCS  
using ControlSystems, Plots, LinearAlgebra, RobustAndOptimalControl  
CCS.setupEnv()  
  
contSys = CCS.blockModel.csys(;g = 0, α = 0 , μ = 1, τ =20)  
plot!(bodeplot(contSys[1,1]),pzmap(contSys))



Starting Bode Plot and PZ Map

It has the shape we expect from a motor + friction. Slow pole for the mass + friction and a faster pole for the current & inductance.

Numerically they are:

display(eigvals(contSys.A))

3-element Vector{Float64}:  
 -20.0  
 -1.0  
 0.0

We see that we start with all the poles in the left-half plane, which is good.

## 5.2 Pole Placement

We can design a controller with pole placement.

For some reason pole placement doesn’t work for the observer, I use a Kalman Filter with random fast values.

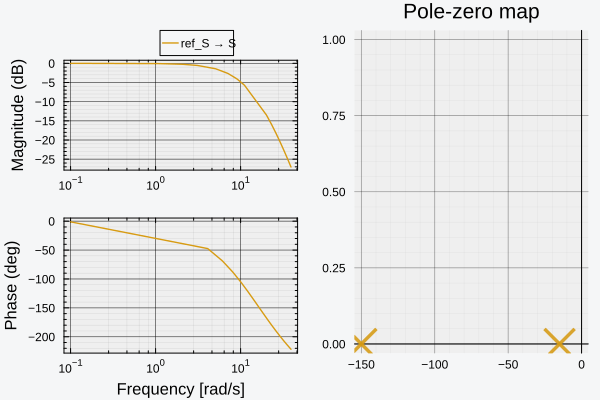
observability(contSys.A,contSys.C).isobservable || error("System is not observable")  
controllability(contSys.A,contSys.B).iscontrollable || error("System is not controllable")  
  
ε = 0.01;  
pp = 15.0;  
poles\_cont = - [pp + ε, pp - ε, pp];  
L = real(place(contSys, poles\_cont, :c));  
  
  
poles\_obs = poles\_cont \* 10.0;  
K = place(contSys, poles\_obs, :o)  
obs\_controller = observer\_controller(contSys, L, K; direct=false);  
fsf\_controller = named\_ss(obs\_controller, u = [:ref\_S, :ref\_V], y = [:u])

NamedStateSpace{Continuous, Float64}  
A =   
 -150.0 8.033684828490095e-12 0.0  
 1.0915557686859107e-5 -279.9999999999851 1.0  
 -3374.9970813429577 -17530.98989999806 -44.0  
B =   
 150.0 0.9999999999919663  
 -1.0915557686859107e-5 278.9999999999851  
 -0.0014186570429435138 16899.989999998063  
C =   
 168.74992500000002 31.549995000000003 1.2000000000000002  
D =   
 0.0 0.0  
  
Continuous-time state-space model  
With state names: x1 x2 x3  
 input names: ref\_S ref\_V  
 output names: u

We can check the effect of the new controller on the loop

closedLoop = feedback( contSys \* fsf\_controller);  
print(poles(closedLoop));  
setPlotScale("dB")  
plot!(bodeplot(closedLoop[1,1], 0.1:40), pzmap(closedLoop))

ComplexF64[-14.990000366343788 + 0.0im, -14.999999266673722 + 0.0im, -15.010000366982666 + 0.0im, -149.99999999999986 + 0.0im, -150.10000000002432 + 0.0im, -149.8999999999607 + 0.0im]



We can compare this to the open-loop response in @start-bode. We can see that we achieve unitary gain throughout the whole low-frequency range.

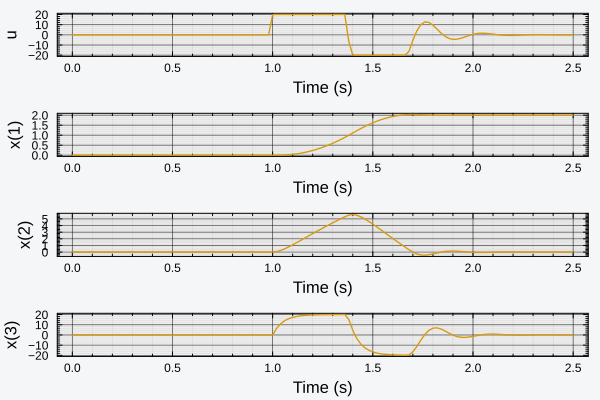
We can convert the pole placement controller into the standard PD gain form.

using DiscretePIDs  
Ts = 0.02 # sampling time  
Tf = 2.5; #final simulation time  
  
K = L[1];  
Ti = 0;  
Td = L[2] / L[1];  
  
pid = DiscretePID(; K, Ts, Ti, Td);

## 5.3 Simulation

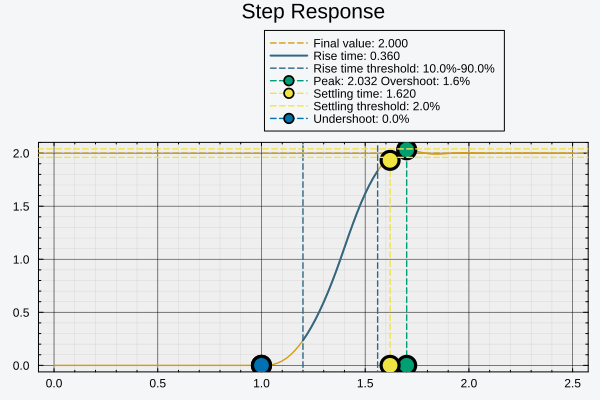
We can simulate this with a motor that only outputs the position:

sysreal = ss(contSys.A, contSys.B, [1 0 0], 0.0)  
ctrl = function (x, t)  
 y = (sysreal.C\*x)[] # measurement  
 d = 0 \* [1.0] # disturbance  
 r = 2.0 \* (t >= 1) # reference  
 # u = pid(r, y) # control signal  
 # u + d # Plant input is control signal + disturbance  
 # u =1  
 e = x - [r; 0.0; 0.0]  
 e[3] = 0.0 # torque not observable, just ignore it in the final feedback  
 u = -L \* e + d  
 u = [maximum([-20.0 minimum([20.0 u])])]  
end  
t = 0:Ts:Tf  
  
res = lsim(sysreal, ctrl, t)  
  
display(plot(res,   
 plotu=true,   
 plotx=true,   
 ploty=false  
 ))  
ylabel!("u", sp=1);  
ylabel!("x", sp=2);  
ylabel!("v", sp=3);  
ylabel!("T", sp=4);



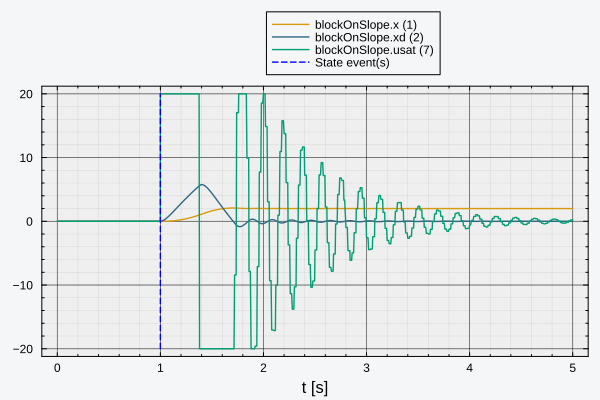
For more stats:

si = stepinfo(res);  
plot(si);title!("Step Response")



We can also simulate it in a SIMULINK-like environment:

using FMI, DifferentialEquations  
fmuPath = abspath(joinpath(@\_\_DIR\_\_,  
 "..","..",  
 "modelica",  
 "ControlChallenges",  
 "ControlChallenges.BlockOnSlope\_Challenges.Examples.WithFriction.fmu"))  
fmu = loadFMU(fmuPath);  
simData = simulateME(  
 fmu,  
 (0.0, 5.0);  
 recordValues=["blockOnSlope.x",   
 "blockOnSlope.xd",   
 "blockOnSlope.usat"],  
 showProgress=false);  
unloadFMU(fmu);  
plot(simData, states=false, timeEvents=false)



There is a slight difference between the lsim simulation and the FMU simulation. I need to recheck some stuff.

# Appendix A — Performance tricks

## A.1 Hurwitz Check

Create our nice model. Assume to have run the poles function and that you have a vector of eigenvalues. For simplicity I will create an arbitrary vector with the first 100 values in and the last 100 values as random around .

using BenchmarkTools  
  
vbig = [zeros(ComplexF64,100).-1 ; rand(ComplexF64,100).-0.5];  
  
vbig[[1,end]]

2-element Vector{ComplexF64}:  
 -1.0 + 0.0im  
 -0.030437584033531584 + 0.17283801604223903im

### A.1.1 for loop

With a naive approach we check if all the elements are in the LHP: make a function that iterates and returns false if it hits a pole with positive real part.

function isHurwitz(v)  
 for i in eachindex(v)  
 if real(v[i])>0.0  
 return false  
 end  
 end  
 return true  
end  
  
@benchmark isHurwitz($(Ref(vbig))[])

BenchmarkTools.Trial: 10000 samples with 987 evaluations per sample.  
 Range (min … max): 49.949 ns … 969.301 ns ┊ GC (min … max): 0.00% … 0.00%  
 Time (median): 54.306 ns ┊ GC (median): 0.00%  
 Time (mean ± σ): 57.041 ns ± 22.237 ns ┊ GC (mean ± σ): 0.00% ± 0.00%  
  
 ▆▇▆█▇▅▄▃▂▂▁▁▁▁ ▂  
 ██████████████▇█▇▇▆▆▆▆▇▆▆▅▅▅▆▅▆▄▄▅▄▅▅▅▅▅▅▄▄▅▄▁▅▃▃▄▄▆▇▁▁▅▆▇▆▆ █  
 49.9 ns Histogram: log(frequency) by time 126 ns <  
  
 Memory estimate: 0 bytes, allocs estimate: 0.

We have our baseline. We can probably squeeze out some more performance but I’m still a Julia noob.

### A.1.2 all()

Let’s try using some of the built-in declarative functions:

@benchmark all(real($vbig).<=0.0)

BenchmarkTools.Trial: 10000 samples with 746 evaluations per sample.  
 Range (min … max): 183.378 ns … 22.068 μs ┊ GC (min … max): 0.00% … 97.78%  
 Time (median): 261.662 ns ┊ GC (median): 0.00%  
 Time (mean ± σ): 445.850 ns ± 864.436 ns ┊ GC (mean ± σ): 32.85% ± 16.78%  
  
 █▇▅▃▁▁▁▁ ▂  
 ██████████▇▆▅▅▅▄▄▅▃▃▁▁▁▁▁▃▅▆▅▅▃▃▃▄▃▃▄▄▃▄▅▆▇▇▇▇▇▆▅▆▅▆▅▅▅▅▅▅▅▅▆ █  
 183 ns Histogram: log(frequency) by time 5.1 μs <  
  
 Memory estimate: 1.75 KiB, allocs estimate: 5.

Simple all(), when given a tuple it checks if all the values are True, otherwise it stops when it encounters the first False.

We can see that it’s a tad slower. This is because it’s creating a new vector with just the real parts, then it’s creating a new vector with only the boolean results and then it’s checking if there are any False results.

This results in a lot of allocations and wasted resources.

@benchmark all(<=(0.0),real($vbig))

BenchmarkTools.Trial: 10000 samples with 759 evaluations per sample.  
 Range (min … max): 144.664 ns … 31.398 μs ┊ GC (min … max): 0.00% … 97.98%  
 Time (median): 226.746 ns ┊ GC (median): 0.00%  
 Time (mean ± σ): 344.335 ns ± 699.060 ns ┊ GC (mean ± σ): 22.41% ± 15.26%  
  
 ▂██▆▃▂▁ ▁▂▁▁▁ ▂  
 ████████▇▇▆▅▇███████▇▇▇▆▅▅▃▄▄▄▁▄▃▃▃▁▁▄▃▃▃▁▁▁▁▁▁▁▁▃▄▆▆▇███▇▆▇▆ █  
 145 ns Histogram: log(frequency) by time 2.44 μs <  
  
 Memory estimate: 1.62 KiB, allocs estimate: 2.

A smarter way is to skip on of the allocations by creating the vector of real parts and then checking row by row if the non-positivity check fails.

@benchmark all(i -> real(i)<=0.0,$vbig)

BenchmarkTools.Trial: 10000 samples with 987 evaluations per sample.  
 Range (min … max): 49.139 ns … 2.022 μs ┊ GC (min … max): 0.00% … 0.00%  
 Time (median): 51.773 ns ┊ GC (median): 0.00%  
 Time (mean ± σ): 53.295 ns ± 22.061 ns ┊ GC (mean ± σ): 0.00% ± 0.00%  
  
 ▅▆█▃▂▂ ▁  
 ███████▇▅▆▆▅▄▅▃▄▄▁▅▅▅▄▃▄▅▄▄▄▁▃▃▃▅▁▄▃▄▃▃▃▄▁▄▃▃▃▃▃▁▃▃▃▃▃▄▄▃▁█ █  
 49.1 ns Histogram: log(frequency) by time 116 ns <  
  
 Memory estimate: 0 bytes, allocs estimate: 0.

We can do better: Instead of converting into real the full vector it checks element by element if it’s in the LHP. It returns false at the first failure. We finally have a comparable result to the benchmark function but in a more compact way.

Is it cleaner? That’s subjective.

### A.1.3 mapreduce()

Finally we try the MapReduce approach. This allows a better utilization of your processor without the necessity of learning parallel programming.

@benchmark mapreduce(i->real(i)<=0.0, &, $vbig)

BenchmarkTools.Trial: 10000 samples with 993 evaluations per sample.  
 Range (min … max): 42.095 ns … 692.346 ns ┊ GC (min … max): 0.00% … 0.00%  
 Time (median): 43.807 ns ┊ GC (median): 0.00%  
 Time (mean ± σ): 46.318 ns ± 15.245 ns ┊ GC (mean ± σ): 0.00% ± 0.00%  
  
 ▇█▆▄▅▄▂▁ ▁ ▁ ▂  
 █████████████▆▆▆▆▅▇▆█▆▆▅▄▅▄▄▃▆▆▃▁▄▅▅▁▃▄▃▁▄▄▄▃▄▄▅▆▃▄▃▄▄▅▇▆▇▆▆ █  
 42.1 ns Histogram: log(frequency) by time 102 ns <  
  
 Memory estimate: 0 bytes, allocs estimate: 0.

We squeeze the last bit of performance and beat the initial benchmark, not by much but still appreciable.

1. MacOs should be supported in theory but it’s not tested. [↑](#footnote-ref-22)