

Control Theory: Tutorial with Julia

Iacopo Moles

Table of contents

1	Introduction	3
1.1	Intro	3
1.2	What do I need?	3
1.2.1	Software	3
1.2.2	Theory	3
I	Block on a Slope	4
2	The Damped Mass problem	5
2.1	Modeling	5
3	Block without Friction	6
3.1	State Space representation	6
3.2	Pole Placement	7
3.3	Simulation	8
4	Block With Friction	12
4.1	Response Analysis	12
4.2	Pole Placement	13
4.3	Simulation	14
5	Block on a slope	18
5.1	Response Analysis	18
5.2	Pole Placement	19
5.3	Simulation	20
	Appendices	24
A	Performance tricks	24
A.1	Hurwitz Check	24
A.1.1	for loop	24
A.1.2	all()	24
A.1.3	mapreduce()	25

1 Introduction

1.1 Intro

This is a collection of write ups on how to solve the various problems presented by Github user¹ "Janismac".

1.2 What do I need?

1.2.1 Software

- A real OS like Linux or Windows. ²
- The Julia Programming Language³
 - Clone the repo⁴
 - Activate the package by running in your terminal:

```
julia --project -e 'using Pkg; Pkg.instantiate()'
```

- (Nice to have) OpenModelica Editor⁵

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

¹<https://janismac.github.io/ControlChallenges/>

²MacOs should be supported in theory but it's not tested.

³<https://julialang.org/install/>

⁴<https://github.com/icpmoles/controlchallengessolutions>

⁵<https://openmodelica.org/>

Part I

Block on a Slope

2 The Damped Mass problem

2.1 Modeling

To better understand the problem let's take a peek¹ at how the simulated model works.

Listing 2.1 BlockOnSlope.js

```
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])
  ];
}
```

- ① The model has obviously some default values for the parameters that can be modified for the different scenarios.
- ② The control command u 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 ± 20 .
- ③ The model is a simple ODE with equations:

$$\begin{cases} \dot{s} = v \\ \dot{v} = F - \sin(\alpha) \cdot g - \mu \cdot v \\ \dot{F} = -20 \cdot F + 20 \cdot u_{sat} \end{cases}$$

Converting it in state-space representation:

$$\begin{bmatrix} \dot{s} \\ \dot{v} \\ \dot{F} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & -\mu & 1 \\ 0 & 0 & -20 \end{bmatrix} \begin{bmatrix} s \\ v \\ F \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ u_{sat} \end{bmatrix} + \begin{bmatrix} 0 \\ -\sin(\alpha) \cdot g \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} s \\ v \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} s \\ v \\ F \end{bmatrix}$$

Obviously the gravitational term acts as a disturbance.

¹<https://github.com/janismic/ControlChallenges/blob/gh-pages/js/models/BlockOnSlope.js>

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
 $\alpha$  = 0.0 # slope
 $\mu$  = 1.0 # friction coefficient
x_0 = -2.0 # starting position
dx_0 = 0.0 # starting velocity
 $\tau$  = 20.0 # torque constant
```

```
# State Space Matrix
```

```
A = [ $\begin{bmatrix} 0 & 1 & 0 \\ 0 & -\mu & 1 \\ 0 & 0 & -\tau \end{bmatrix}$ ];
```

```
B = [ $\begin{bmatrix} 0 \\ 0 \\ \tau \end{bmatrix}$ ];
```

```
C = [ $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ ];
```

```
sys = ss(A, B, C, 0.0) # Continuous
```

```
plot!(bodeplot(tf(sys)),pzmap(tf(sys)))
```

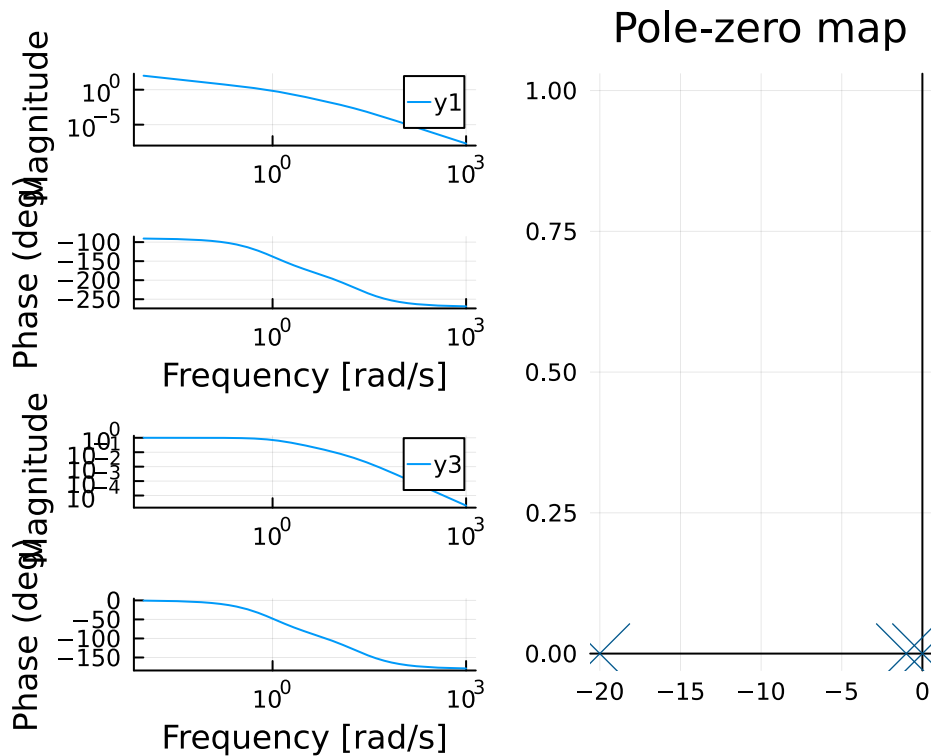


Figure 3.1: Starting Bode Plot and PZ Map

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));

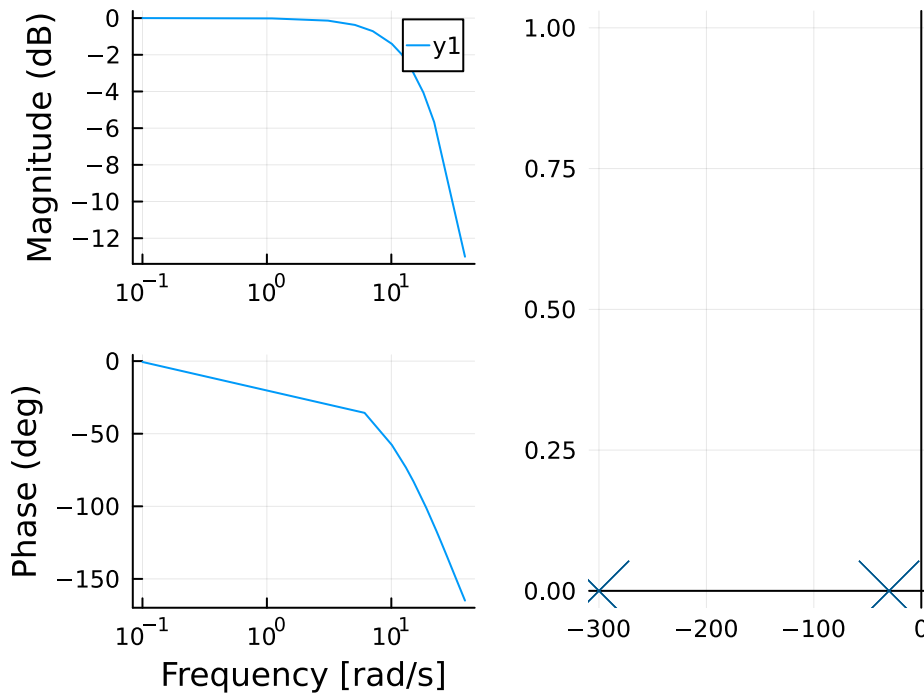
poles_obs = poles_cont * 10.0;
K = place(1.0 * A', 1.0 * C', poles_obs)';
cont = observer_controller(sys, L, K; direct=false);
```

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, -300.00000000000004 + 0.0im, -300.1999999999752 + 0.0im, -299.800000000004117 + 0.0im]
```

Pole-zero map



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);
```

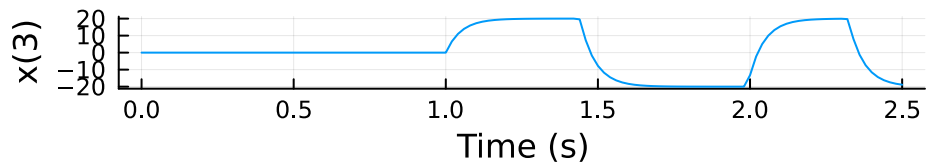
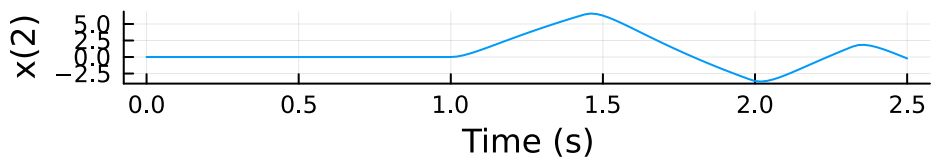
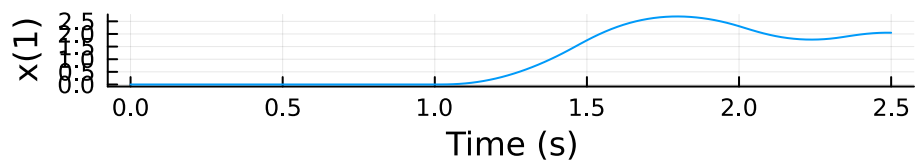
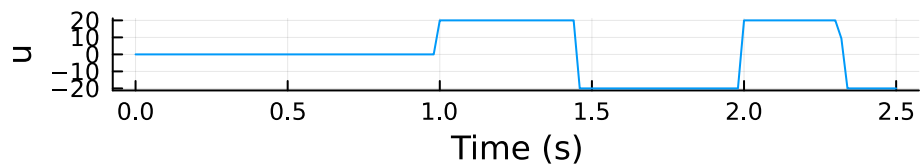
3.3 Simulation

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
```


3 Block without Friction

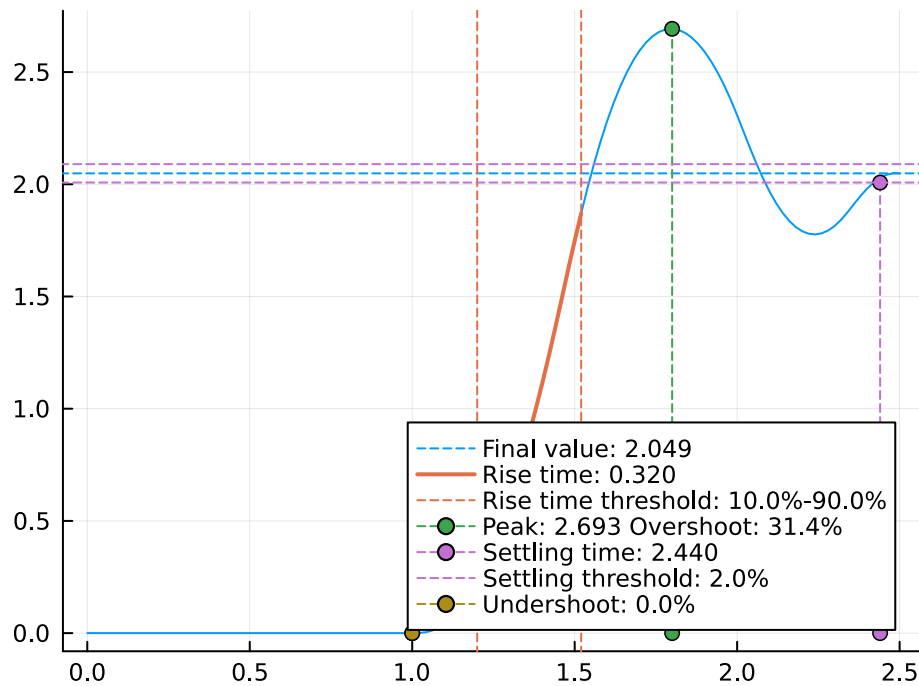
```
t = 0:Ts:Tf  
res = lsim(sysreal, ctrl, t)  
display(plot(res,  
    plotu=true,  
    plotx=true,  
    ploty=false  
))  
ylabel!("u", sp=1);  
ylabel!("x", sp=2);  
ylabel!("v", sp=3);  
ylabel!("T", sp=4);
```



For more stats:

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

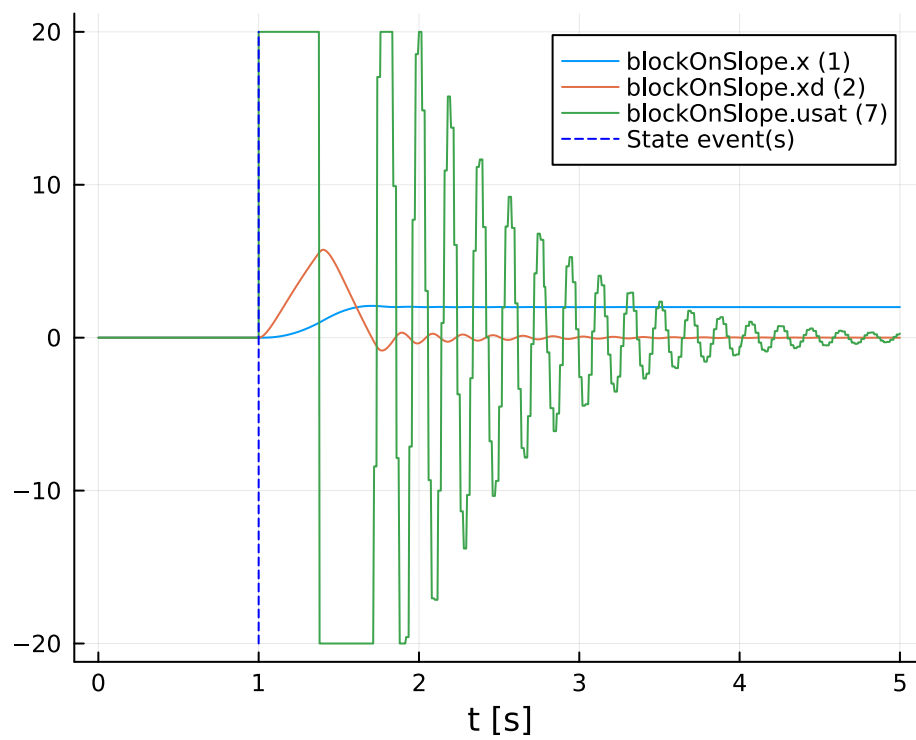
Step Response



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)
```

3 Block without Friction



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,  $\alpha$  = 0,  $\mu$  = 1,  $\tau$  = 20)
plot!(bodeplot(contSys[1,1]), pzmap(contSys))
```

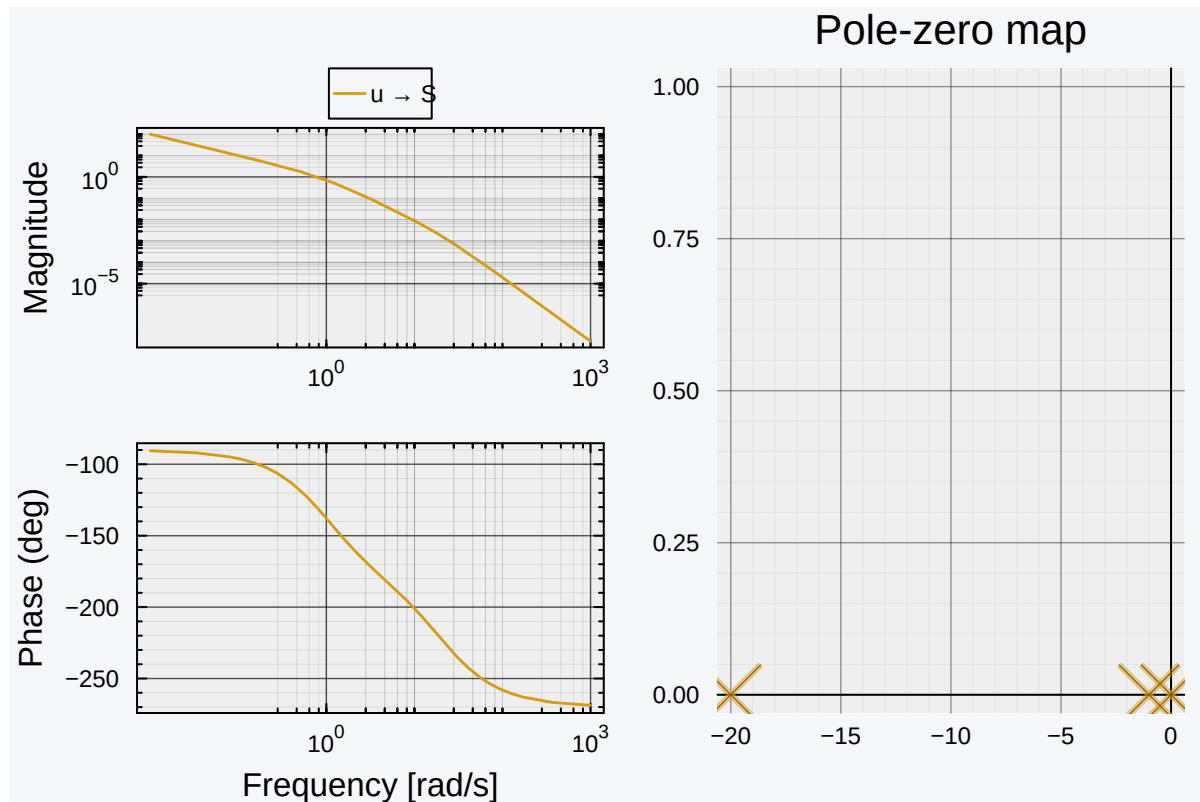


Figure 4.1: Starting Bode Plot and PZ Map

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")

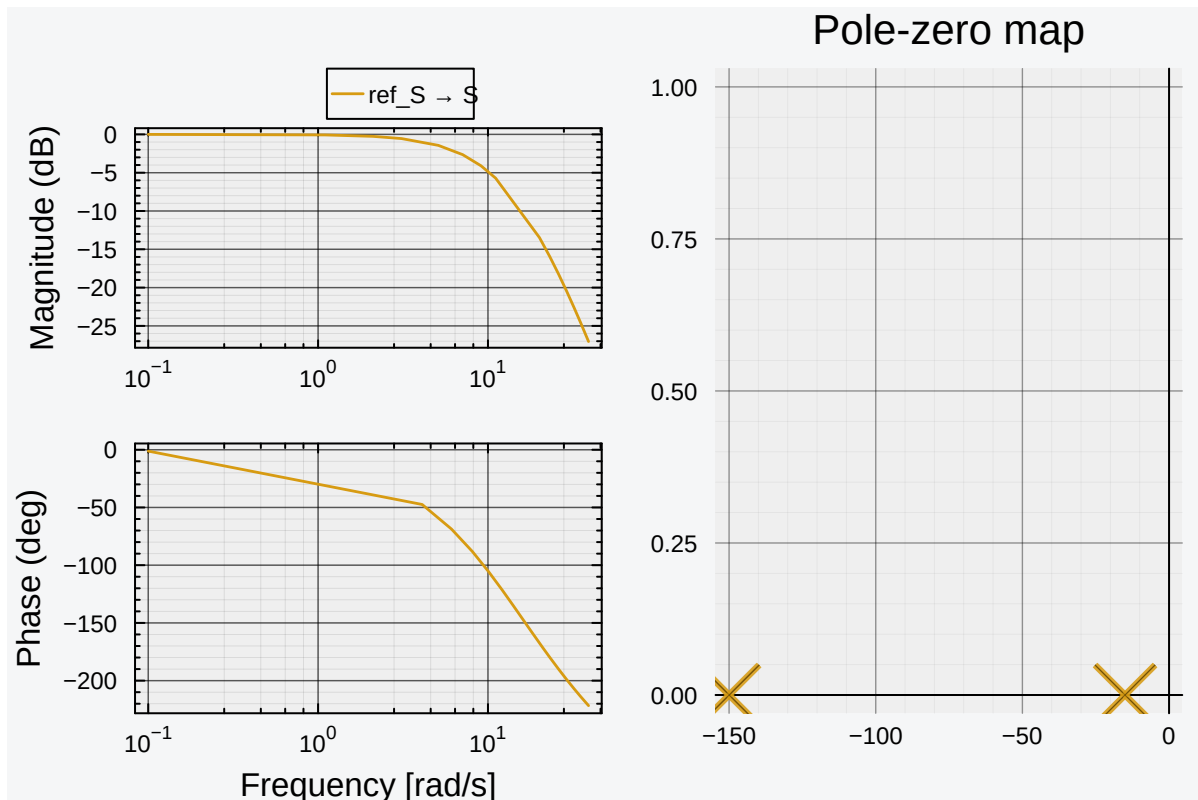
ε = 0.01;
pp = 15.0;
poles_cont = - [pp + ε, pp - ε, pp];
L = real(place(contSys, poles_cont, :c));

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

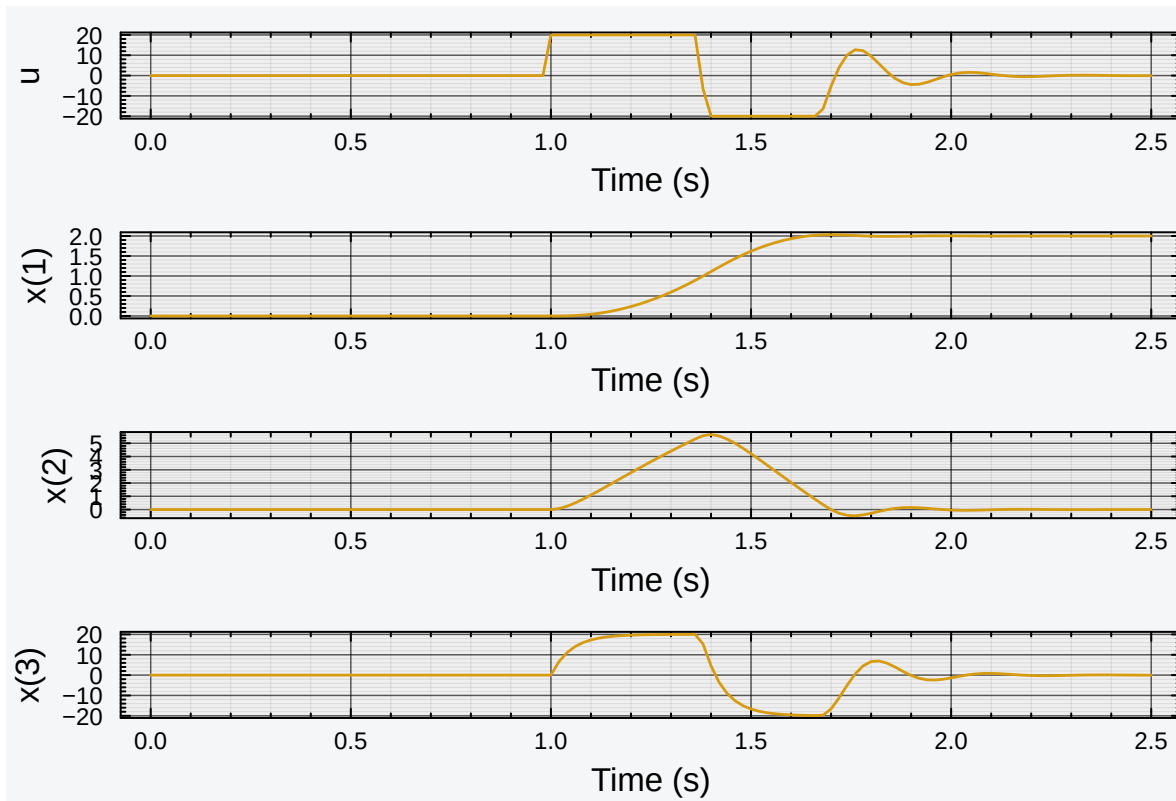
We can simulate this with a motor that only outputs the position:

```
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)
```

4 Block With Friction

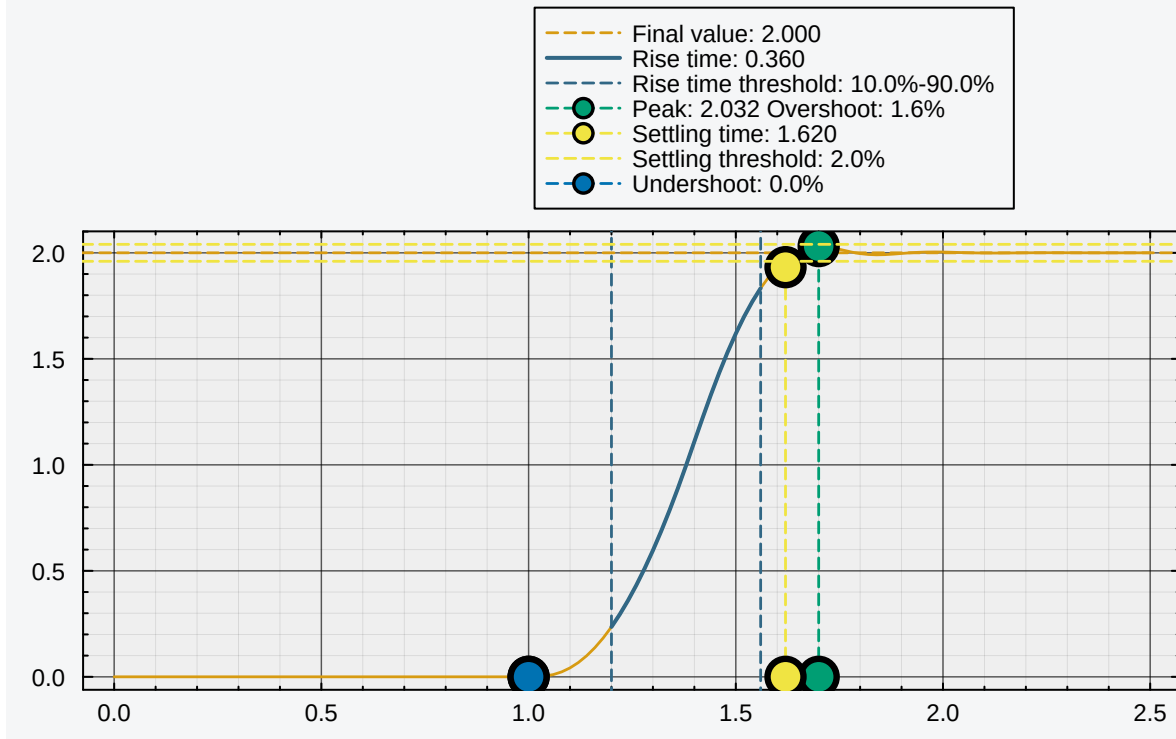
```
display(plot(res,  
    plotu=true,  
    plotx=true,  
    ploty=false  
))  
ylabel!("u", sp=1);  
ylabel!("x", sp=2);  
ylabel!("v", sp=3);  
ylabel!("T", sp=4);
```



For more stats:

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

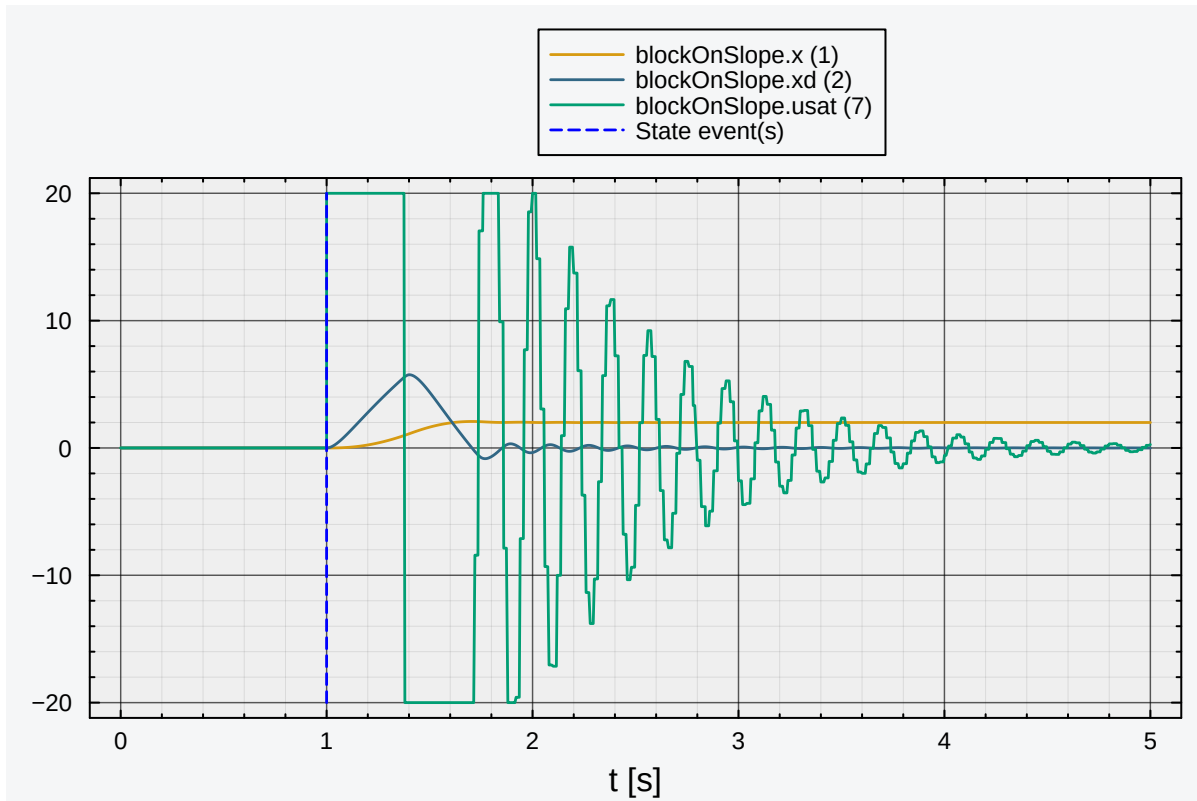
Step Response



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)
```


4 Block With Friction



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,  $\alpha$  = 0,  $\mu$  = 1,  $\tau$  = 20)
plot!(bodeplot(contSys[1,1]), pzmap(contSys))
```

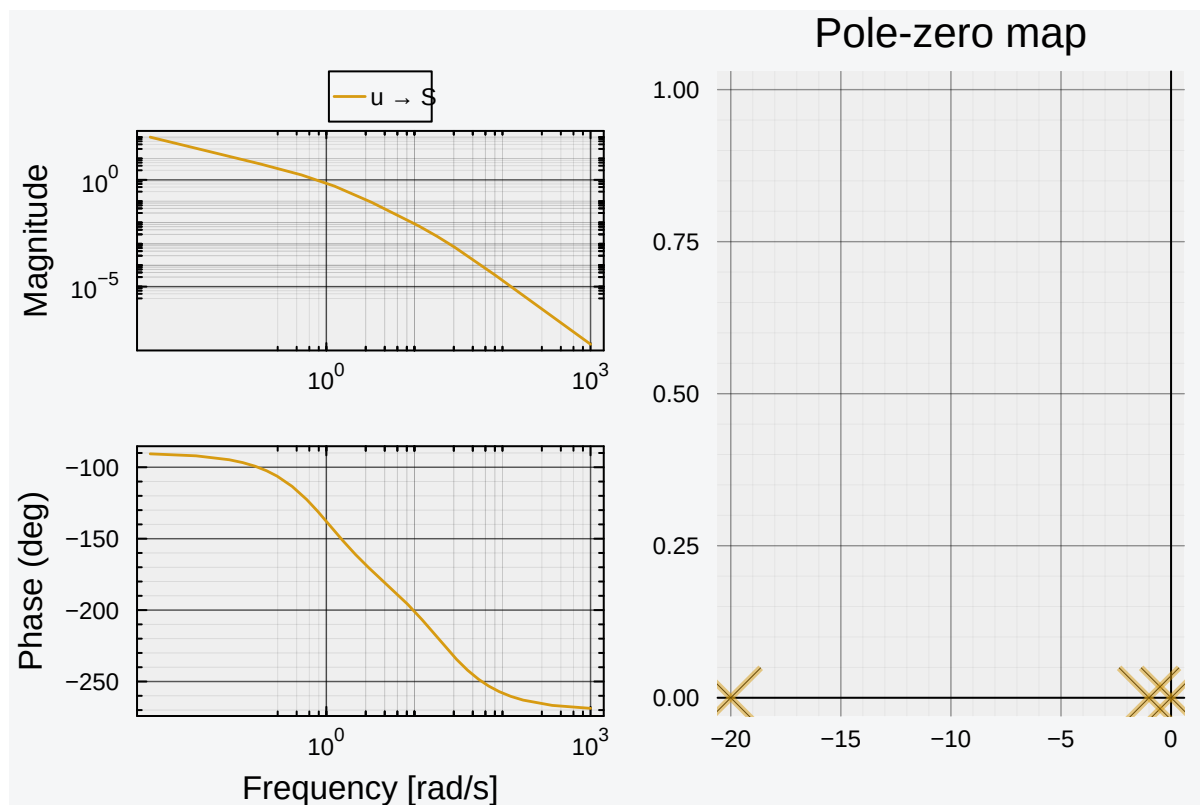


Figure 5.1: Starting Bode Plot and PZ Map

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])
```

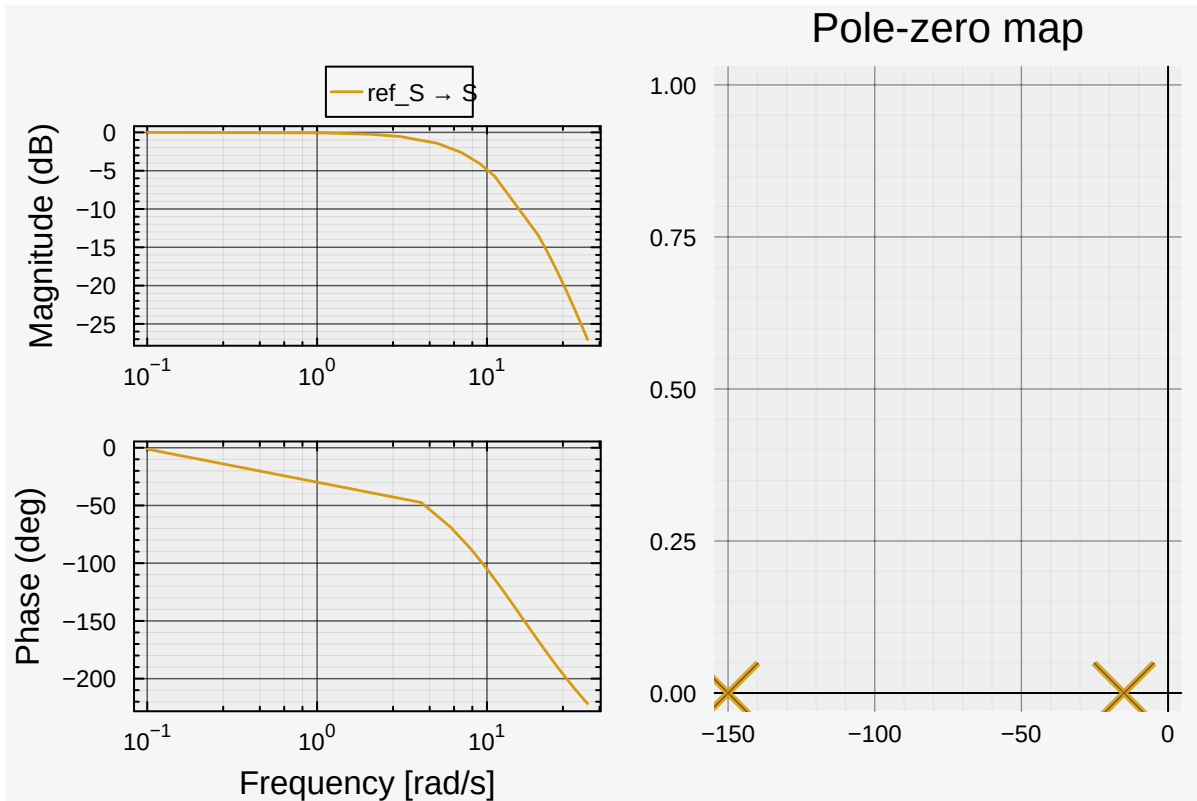
```
NamedStateSpace{Continuous, Float64}
A =
  -150.0      8.033684828490095e-12      0.0
    1.0915557686859107e-5    -279.9999999999851      1.0
  -3374.9970813429577    -17530.98989999806    -44.0
B =
  150.0      0.9999999999919663
  -1.0915557686859107e-5    278.9999999999851
  -0.0014186570429435138  16899.989999998063
C =
  168.749925000000002  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, -149.9999999999986 + 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

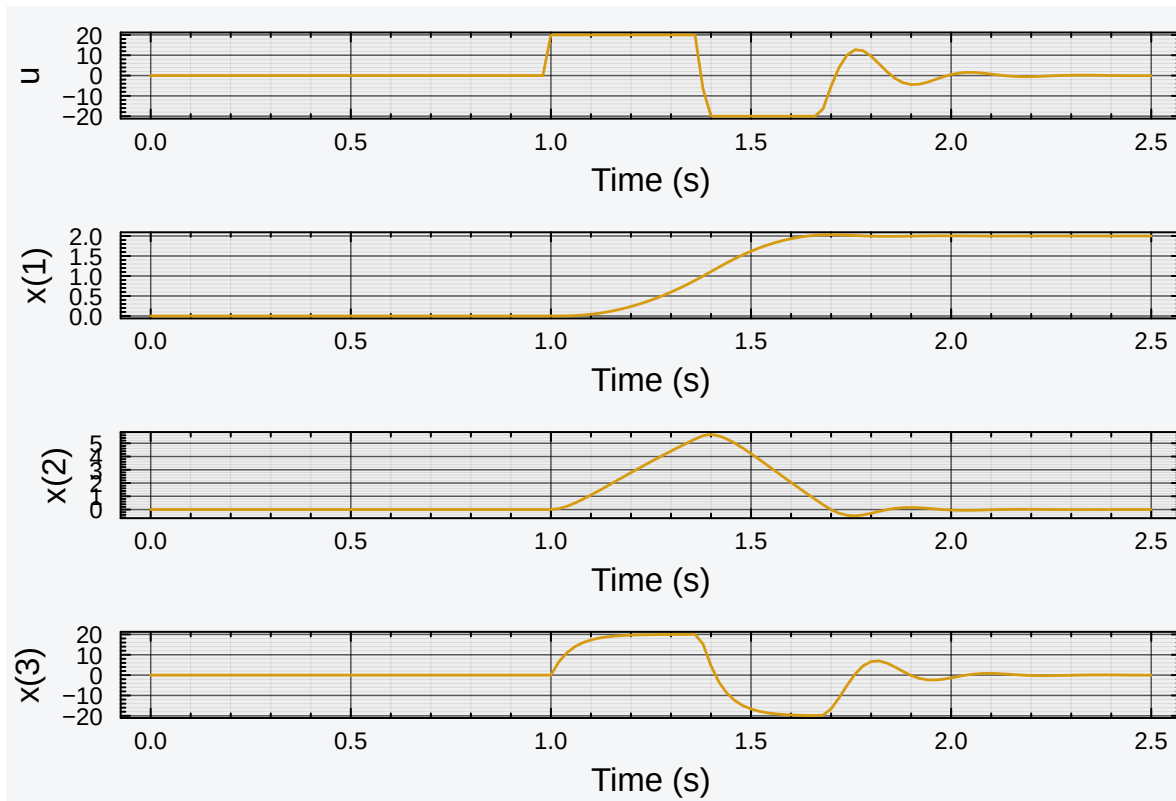
We can simulate this with a motor that only outputs the position:

```
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)
```

5 Block on a slope

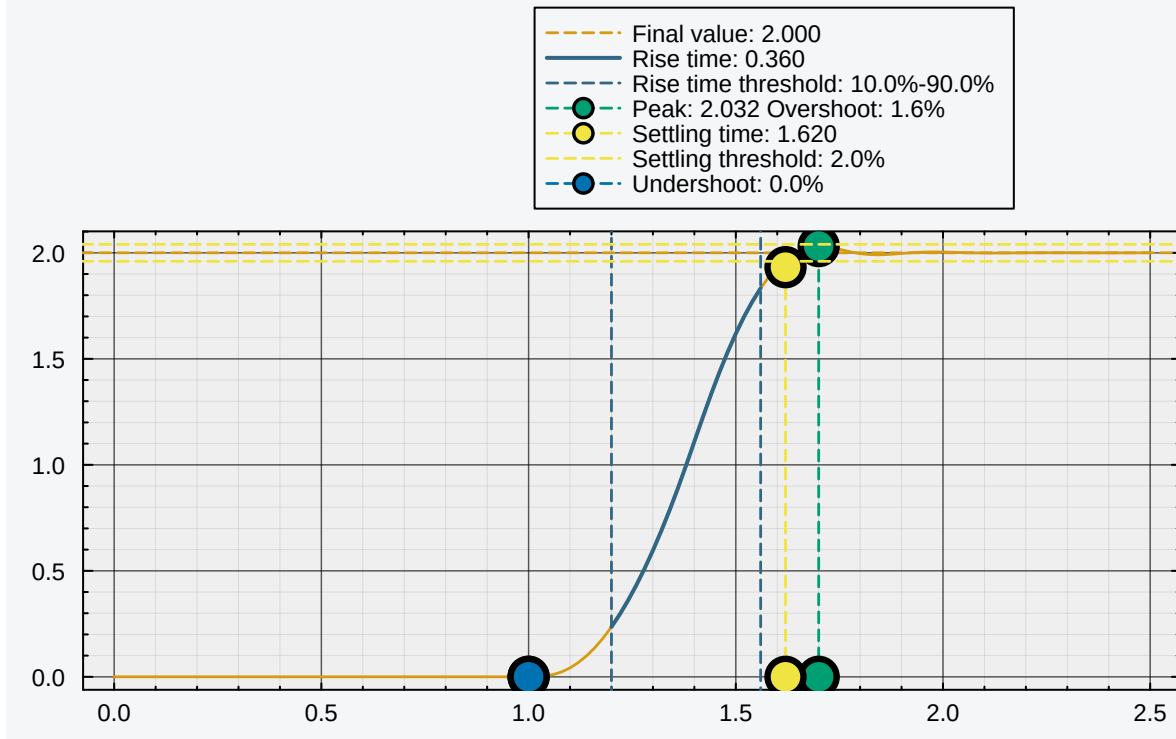
```
display(plot(res,  
    plotu=true,  
    plotx=true,  
    ploty=false  
))  
ylabel!("u", sp=1);  
ylabel!("x", sp=2);  
ylabel!("v", sp=3);  
ylabel!("T", sp=4);
```



For more stats:

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

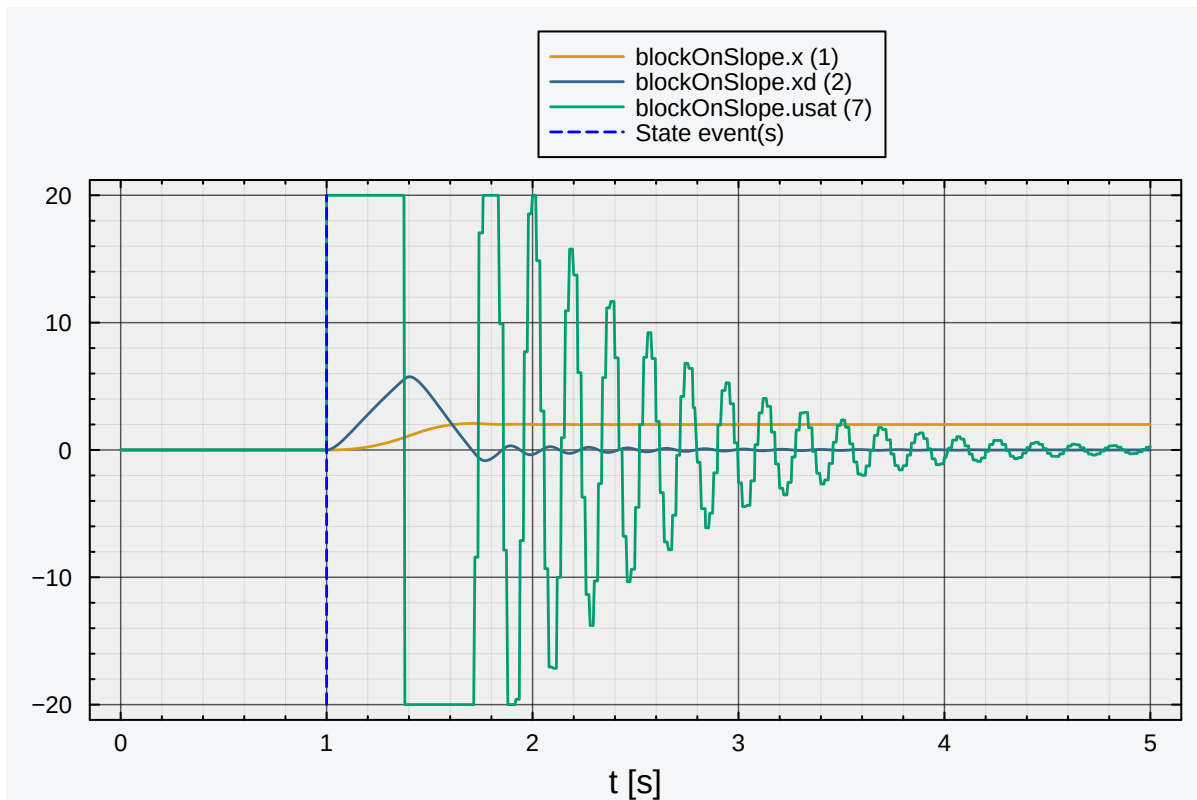
Step Response



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)
```

5 Block on a slope



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

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 -1 and the last 100 values as random around 0.

```
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
```

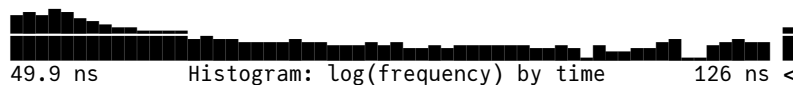
A.1.1 for loop

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%
```



Memory estimate: 0 bytes, allocs estimate: 0.

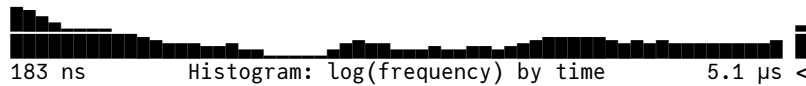
We have our baseline. We can probably squeeze out some more performance but I'm still a Julia noob.

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%
```

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.

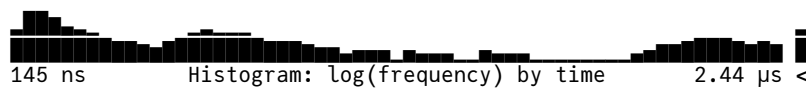
We can see that it's a tad slower. This is because it's creating a new vector with just the real parts, then it's creating a new vector with only the boolean results and then it's checking if there are any False results.

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%



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%



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%



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.