

Control Challenges: Solutions

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
1	The Damped Mass problem	5
	Modeling	6
2	Block without Friction	8
2.1	State Space representation	8
2.2	Pole Placement	10
2.3	Simulation	11
3	Block With Friction	16
3.1	Response Analysis	16
3.2	Pole Placement	17
3.3	Simulation	19
4	Block on a slope	24
4.1	Response Analysis	24
4.2	Pole Placement	25
4.3	Simulation	28
5	Performance tricks	32
5.1	Hurwitz Check	32

1 Introduction

1.1 Intro

This is a collection of write ups on how to solve the various problems presented by [Github user "Janismac"](#).

1.2 What do I need?

1.2.1 Software

- A real OS like Linux or Windows. ¹
- The [Julia Programming Language](#)
 - Clone the [repo](#)
 - Activate the package by running in your terminal:

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

- (Nice to have) [OpenModelica Editor](#)

1.2.2 Theory

- Basic Julia knowledge
- Basic JS knowledge
- Control Theory knowledge
 - Frequency Based Control
 - State Space Based control
- Misc knowledge:
 - Linear Algebra

¹MacOs should be supported in theory but it's not tested.

– Differential Equations

Part I

The Damped Mass problem

Modeling

To better understand the problem let's take a [peek](#) at how the simulated model works.

Listing 1.1 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.

2 Block without Friction

Position Control with friction. Using Pole Placement + PD.

2.1 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 = [0 1 0
     0 - $\mu$  1
     0 0 - $\tau$ ];
B = [0
     0
      $\tau$ ];
C = [1 0 0
     0 1 0];

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

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

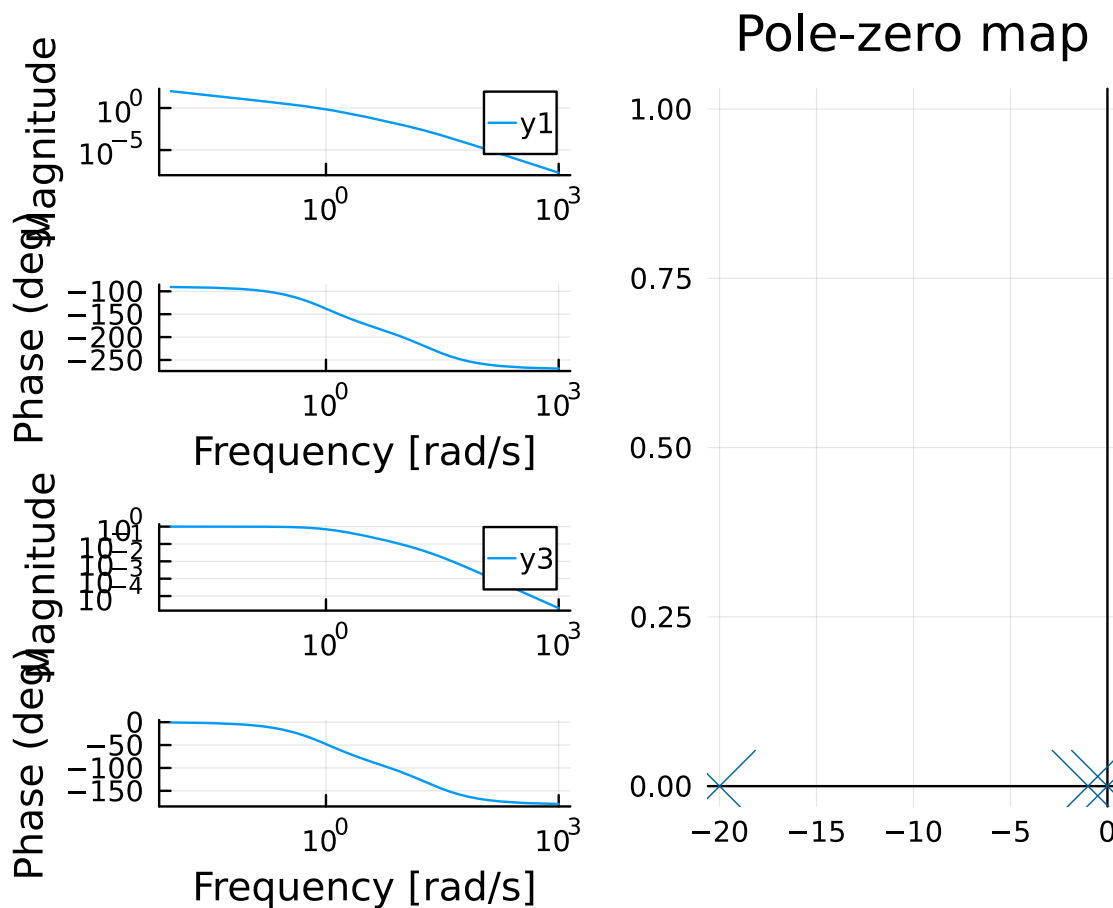



Figure 2.1: Starting Bode Plot and PZ Map

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.

2.2 Pole Placement

We can design a controller with pole placement.

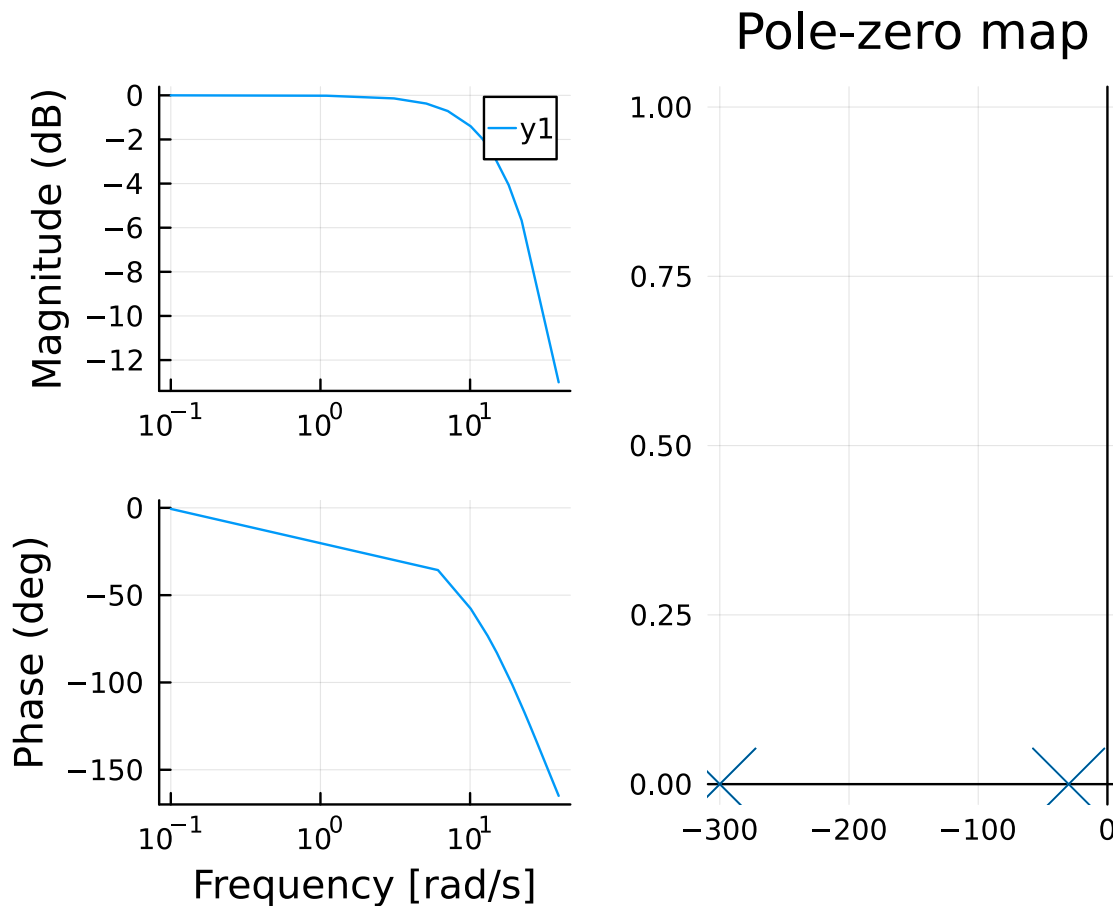
For some reason pole placement doesn't work for the observer, I use a Kalman Filter with random fast values.

```
observability(A, C).isobservable &  
controllability(A, B).iscontrollable; #OK  
  
 $\varepsilon$  = 0.01;  
pp = 15.0;  
poles_cont = -2.0 * [pp +  $\varepsilon$ , pp -  $\varepsilon$ , 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,  
-300.19999999999752 + 0.0im, -299.800000000004117 + 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);
```

2.3 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
```

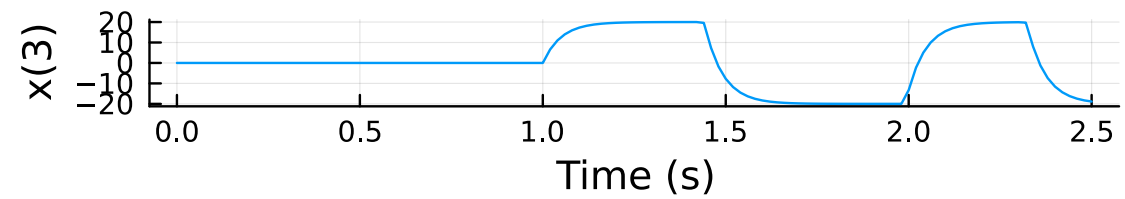
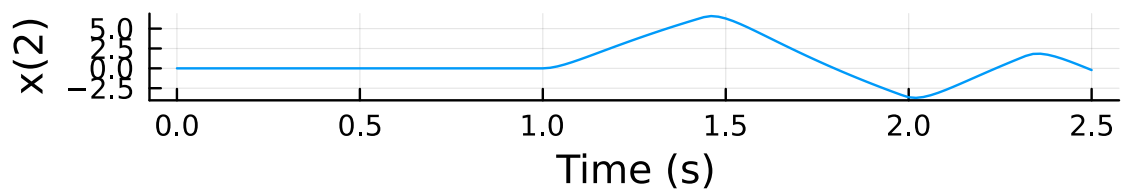
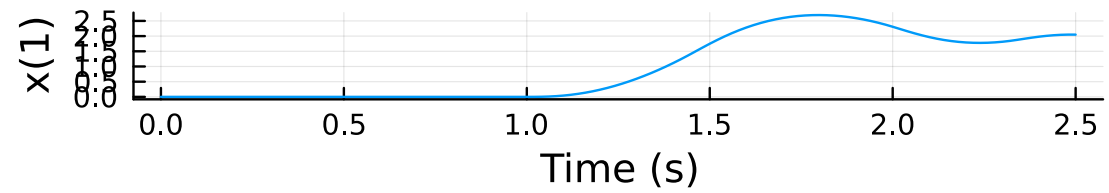
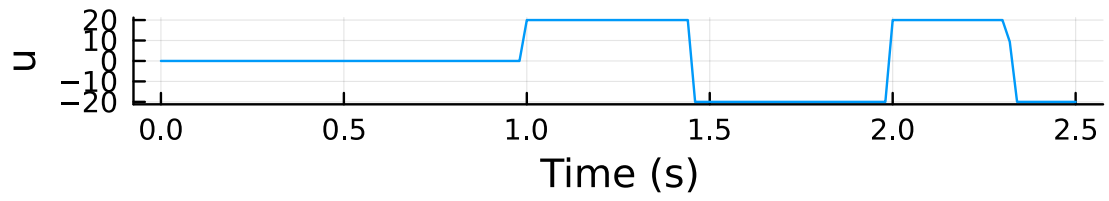
```

    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);

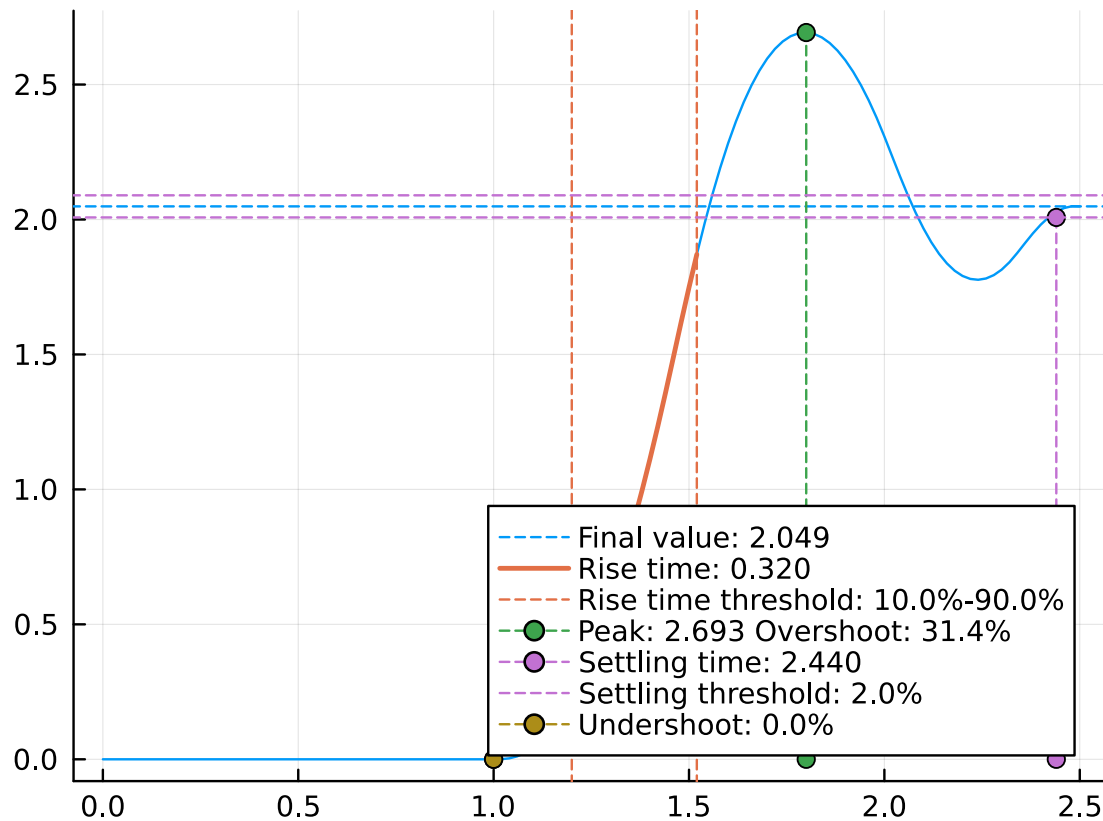
```



For more stats:

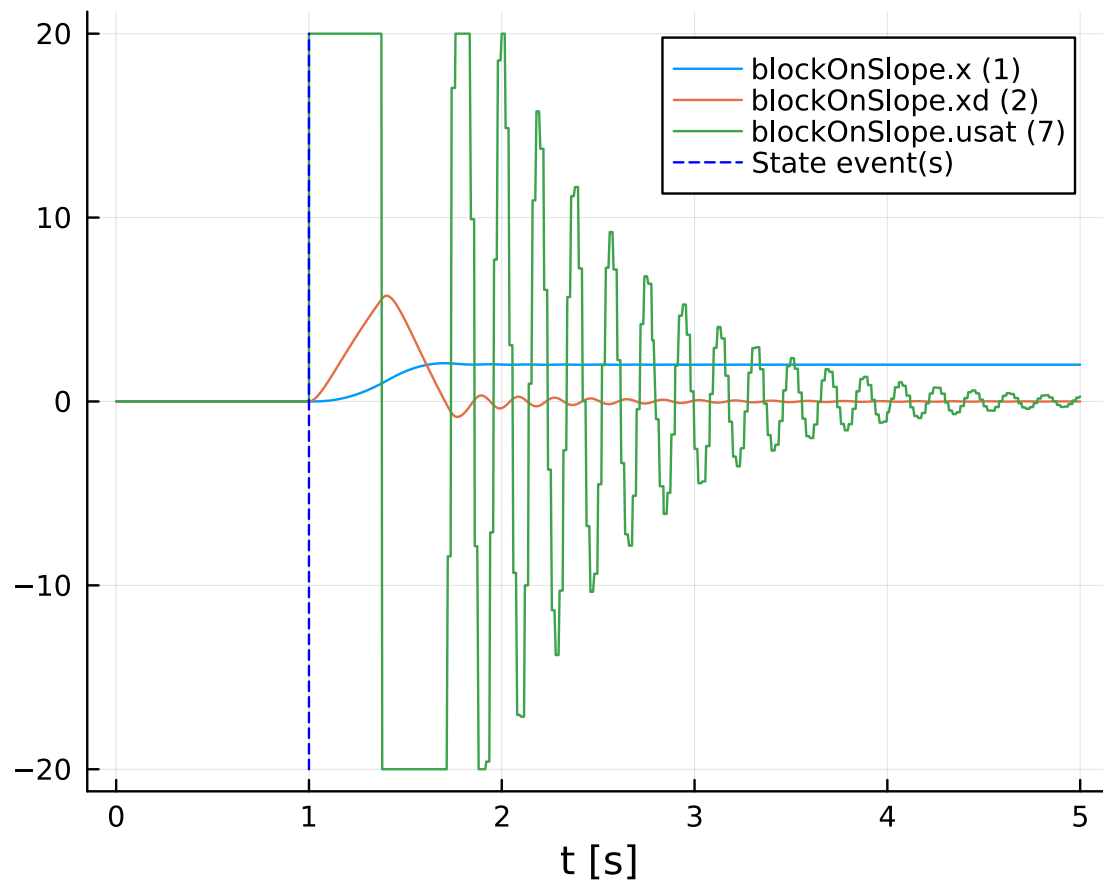
```
si = stepinfo(res);
plot(si);title!("Step Response")
```

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",
    "n.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.

3 Block With Friction

Position Control with friction. Using Pole Placement + PD.

3.1 Response Analysis

```
using CCS: blockModel
using ControlSystems, Plots, LinearAlgebra,
    ↳ RobustAndOptimalControl, PlotThemes
theme(:wong)
contSys = blockModel.csys(;g = 0,  $\alpha$  = 0,  $\mu$  = 1,  $\tau$  = 20)

plot!(bodeplot(contSys[1,1]),pzmap(contSys))
```

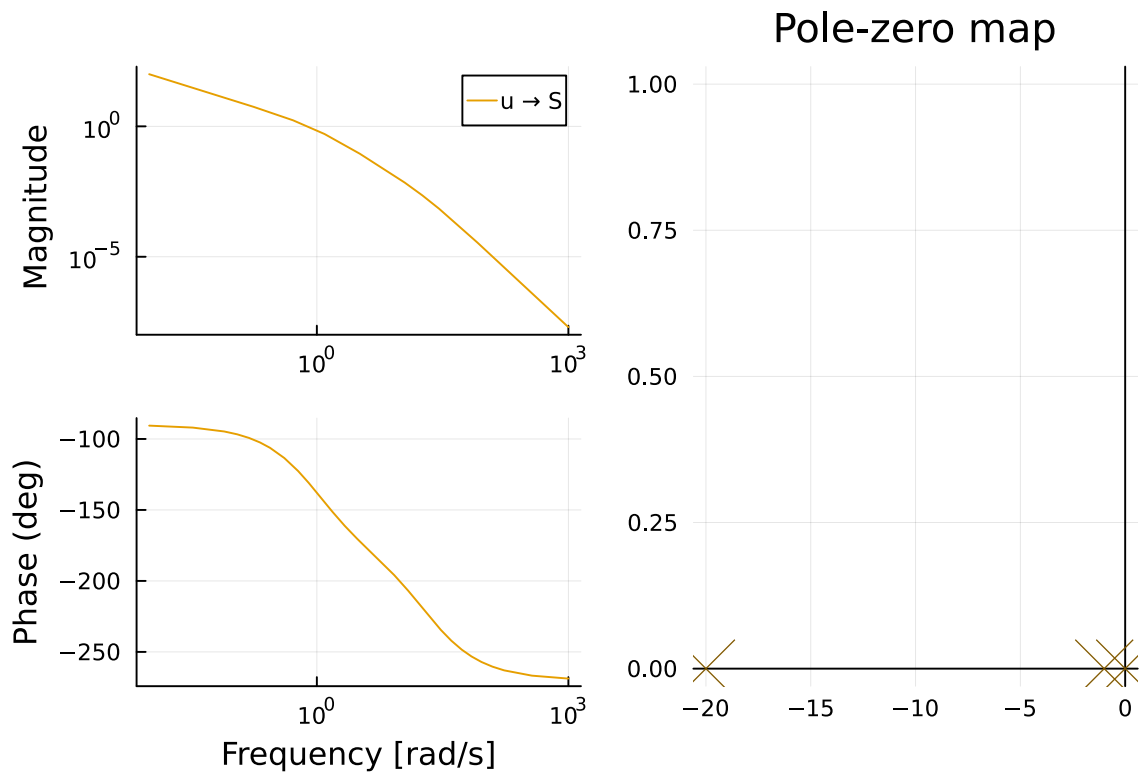



Figure 3.1: Starting Bode Plot and PZ Map

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(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.

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(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

```

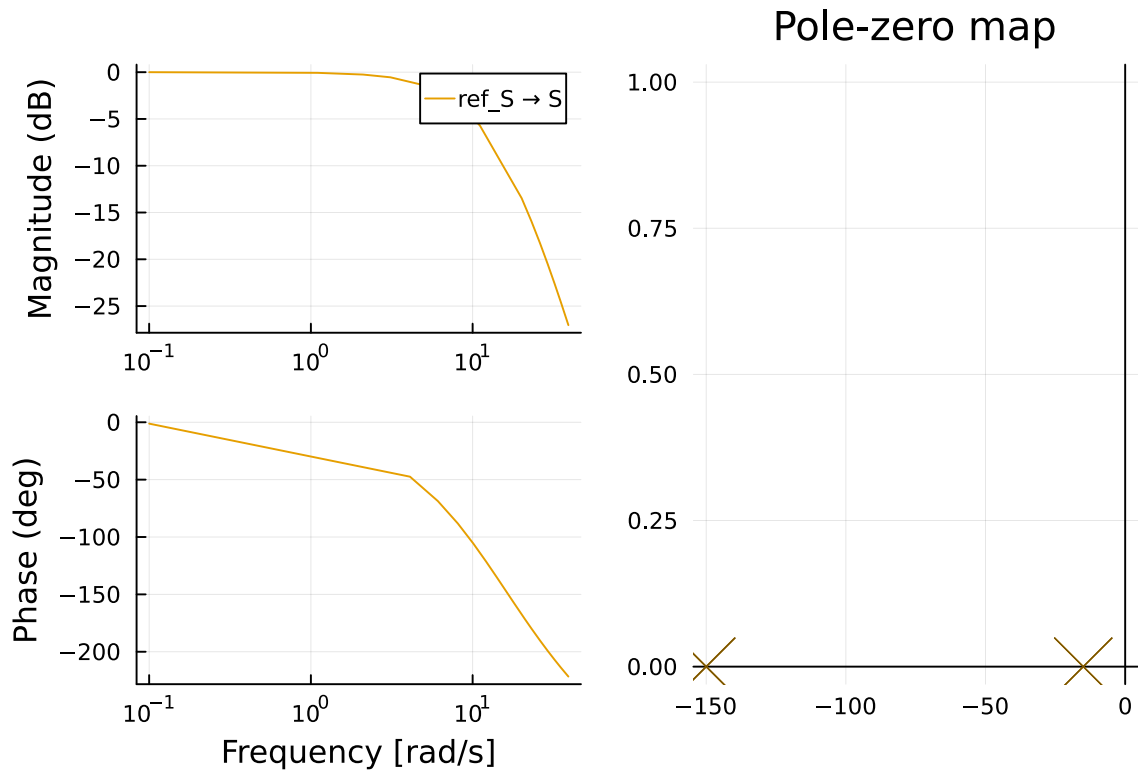
closedLoop = feedback( contSys * fsf_controller);
print(poles(closedLoop));
setPlotScale("dB")
plot!(bodeplot(closedLoop[1,1], 0.1:40), pzmap(closedLoop))

```

```

ComplexF64[-14.990000366343788 + 0.0im, -14.999999266673722 +
0.0im, -15.010000366982666 + 0.0im, -149.9999999999986 + 0.0im,
-150.100000000002432 + 0.0im, -149.89999999999607 + 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);
```

3.3 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)
```

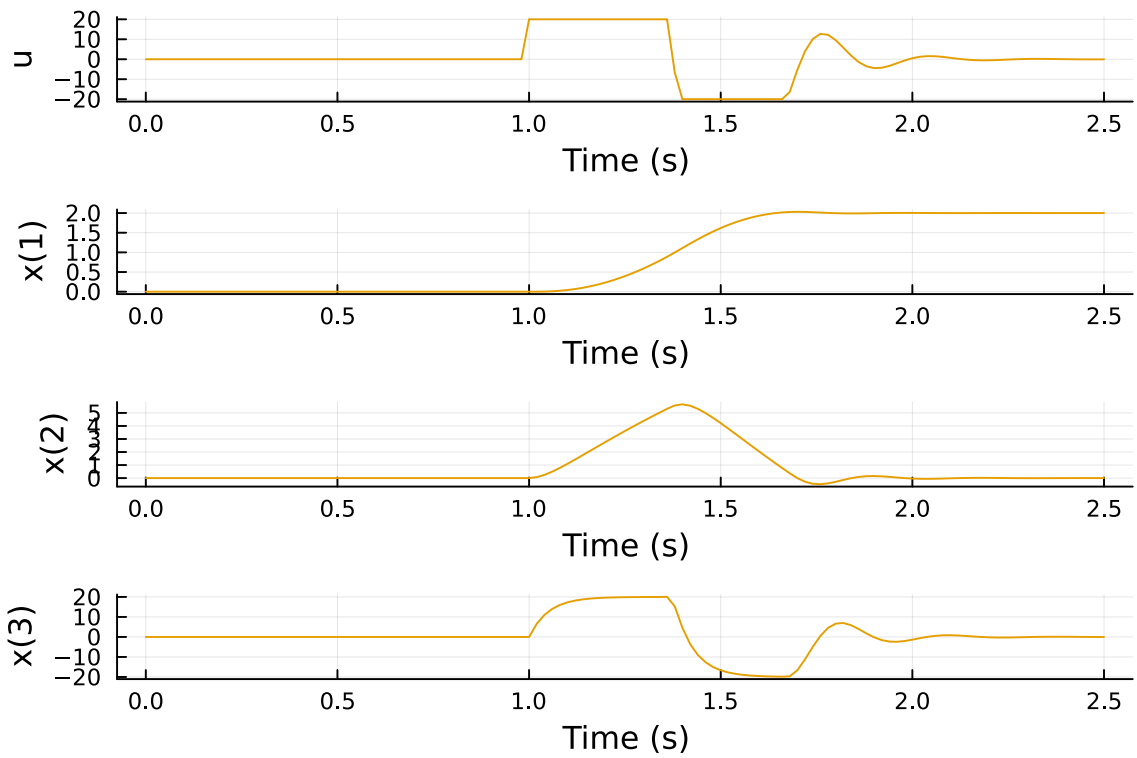
```

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)

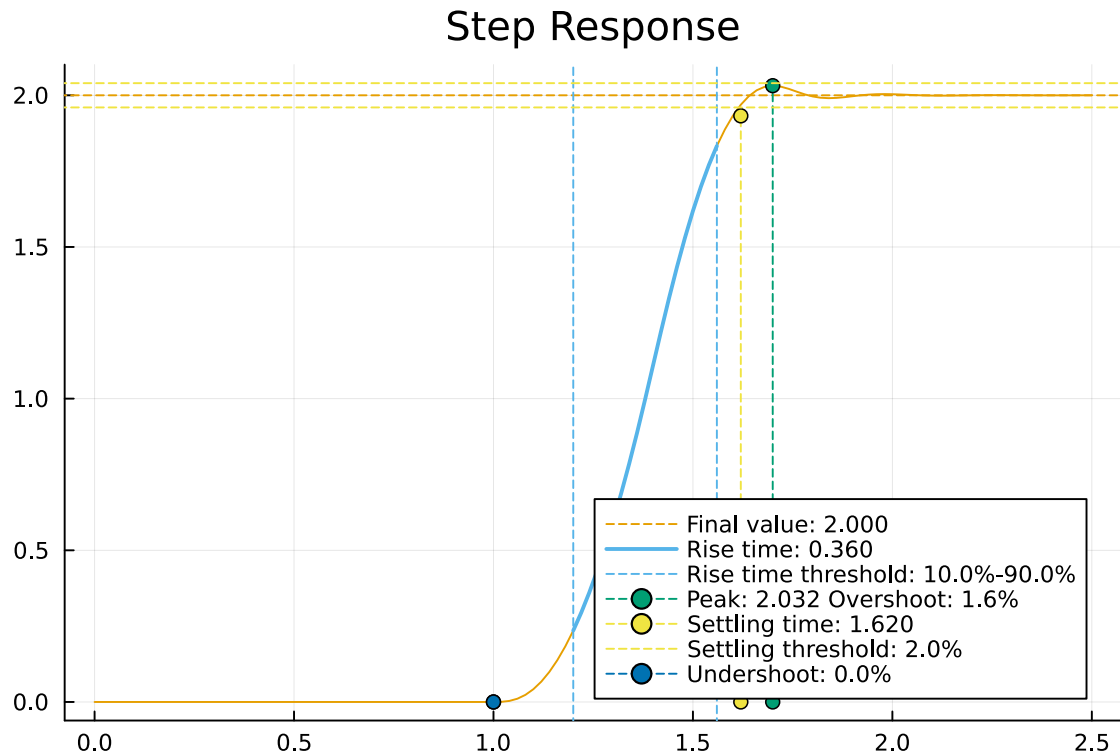
display(plot(res,
    plotu=true,
    plotx=true,
    ploty=false
))
ylabel!("u", sp=1);
ylabel!("x", sp=2);
ylabel!("v", sp=3);
ylabel!("T", sp=4);

```



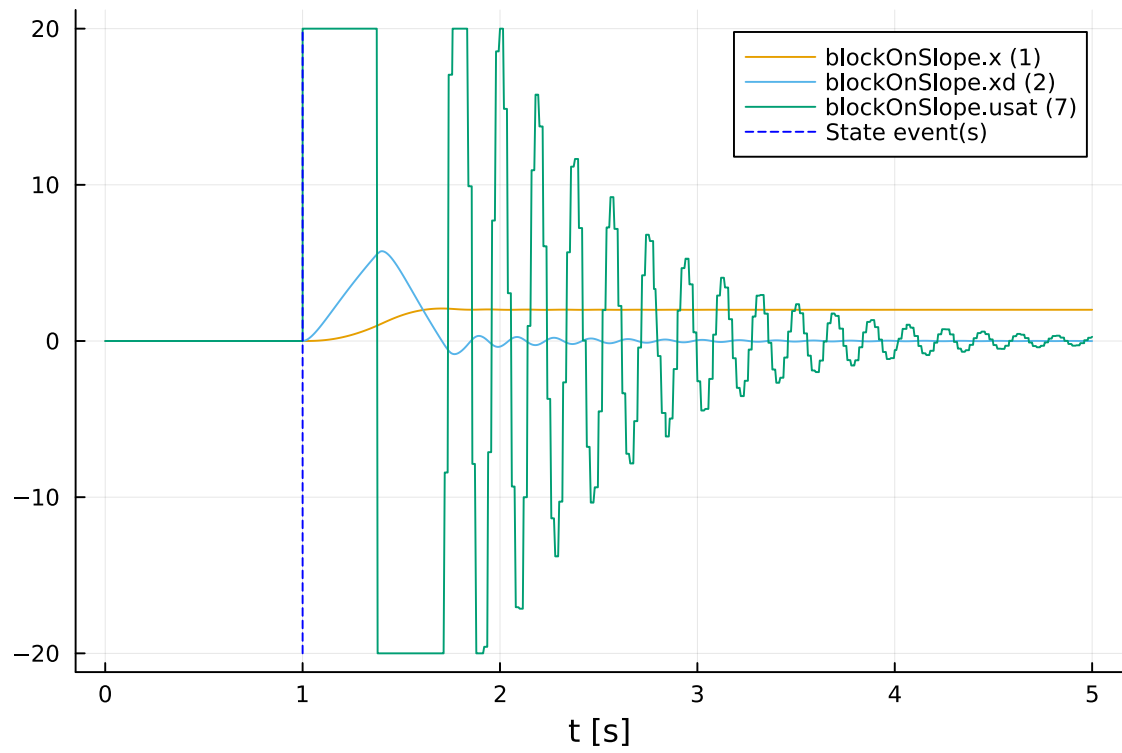
For more stats:

```
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",
  "n.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 on a slope

Position Control with friction. Using Pole Placement + PD.

4.1 Response Analysis

```
using CCS: blockModel
using ControlSystems, Plots, LinearAlgebra,
    ↳ RobustAndOptimalControl

contSys = blockModel.csys(;g = 0,  $\alpha$  = 0,  $\mu$  = 1,  $\tau$  = 20)

plot!(bodeplot(contSys[1,1]),pzmap(contSys))
```

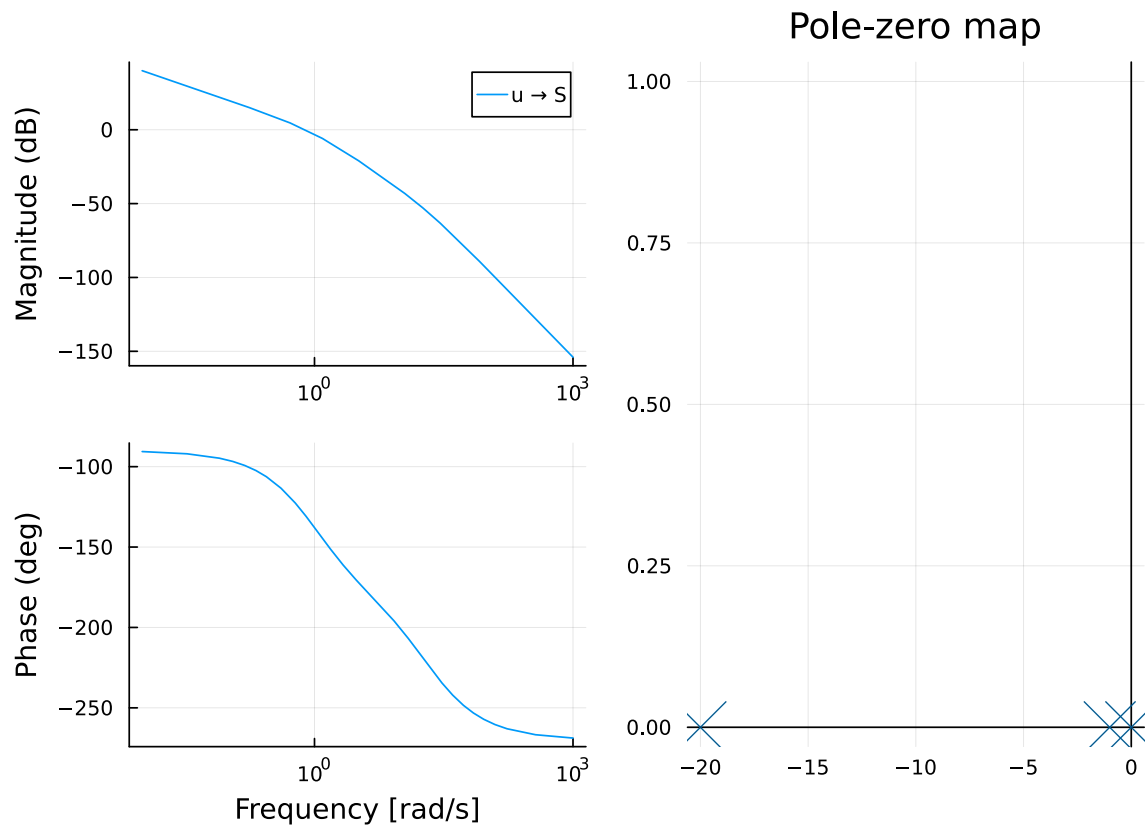



Figure 4.1: Starting Bode Plot and PZ Map

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])
```

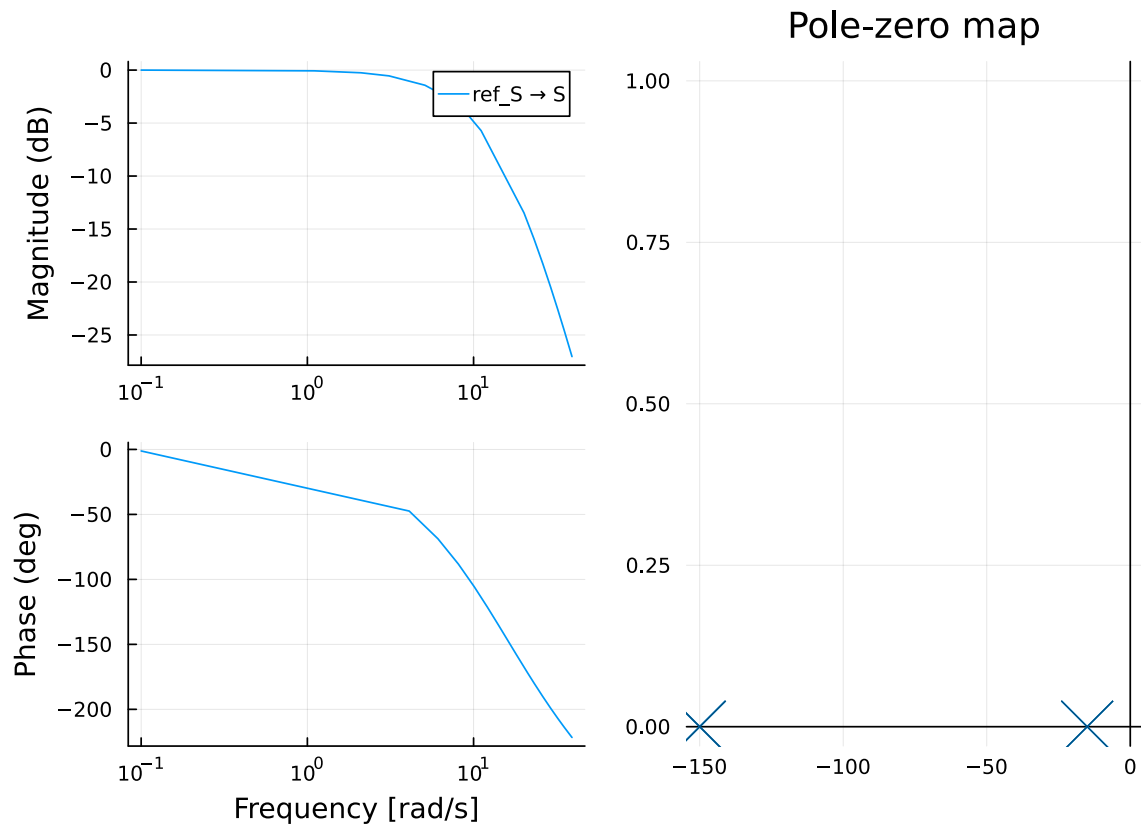
```
NamedStateSpace{Continuous, Float64}
A =
  -150.0          8.033684828490095e-12    0.0
    1.0915557686859107e-5  -279.999999999851    1.0
  -3374.9970813429577    -17530.98989999806   -44.0
B =
  150.0          0.999999999919663
  -1.0915557686859107e-5    278.999999999851
  -0.0014186570429435138  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,  
-150.100000000002432 + 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);
```

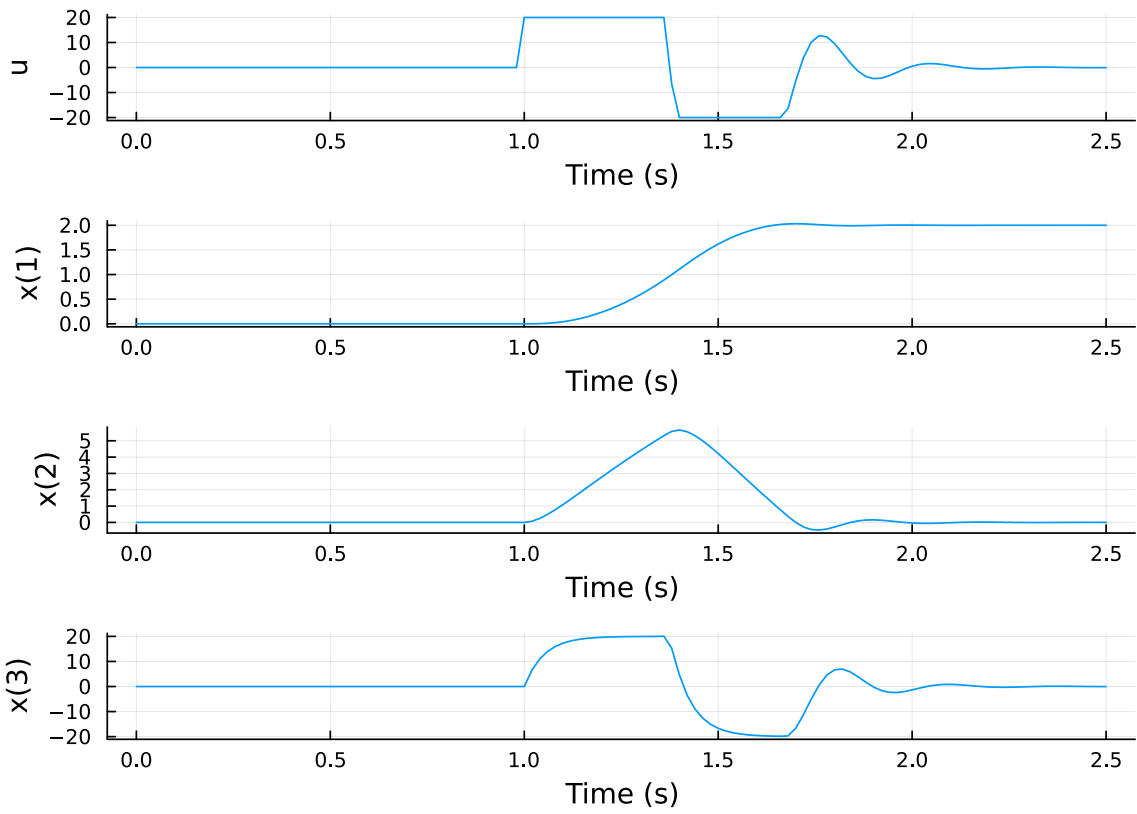
4.3 Simulation

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

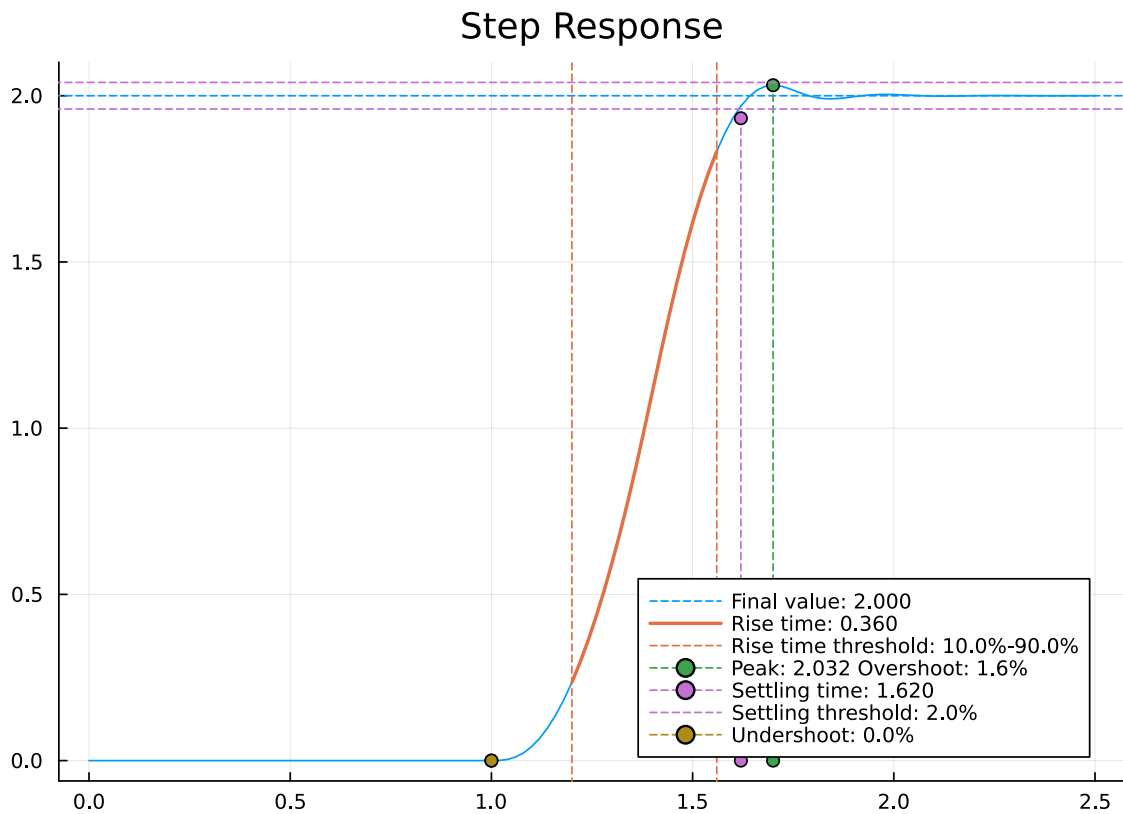
res = lsim(sysreal, ctrl, t)

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);
```



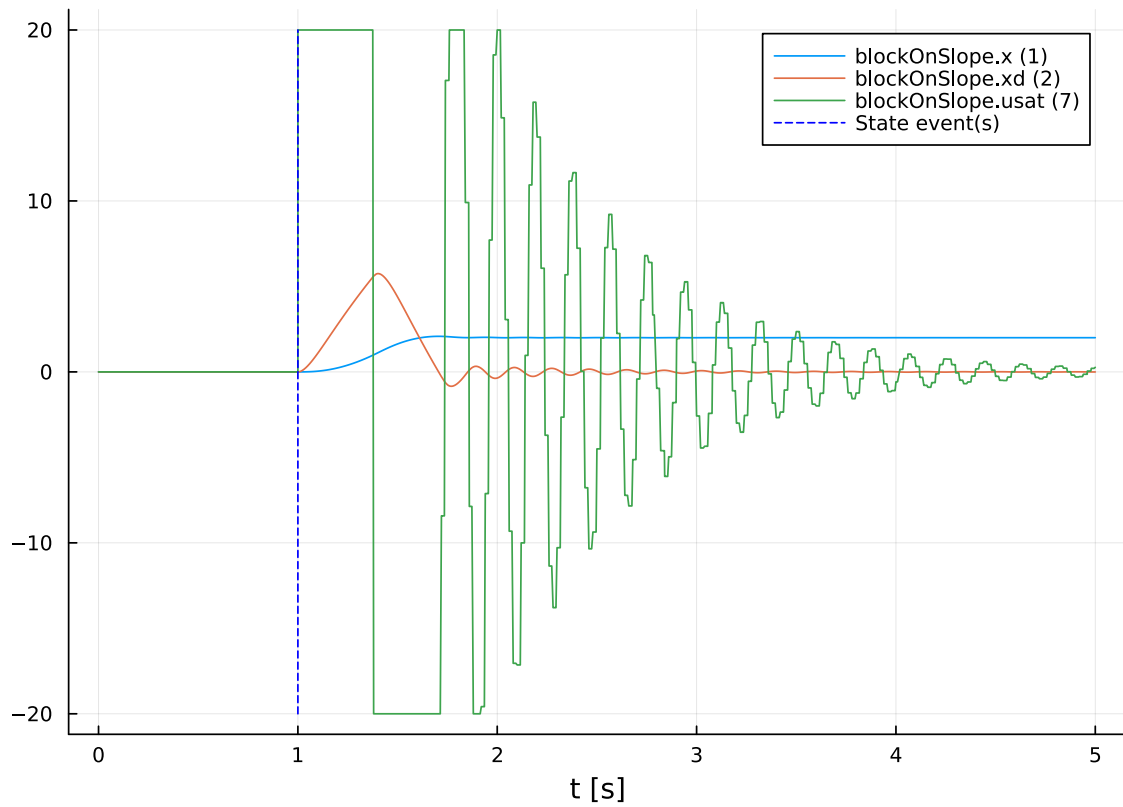
For more stats:

```
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",
    "n.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 Performance tricks

5.1 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];

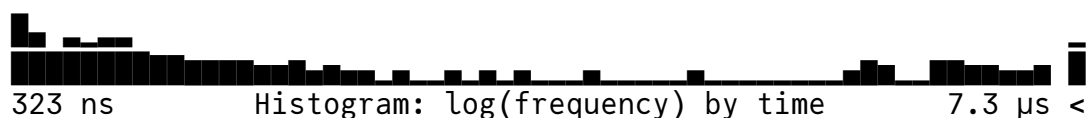
vbig[[1,end]]
```

```
2-element Vector{ComplexF64}:
  -1.0 + 0.0im
 -0.4331044312441319 + 0.2962239553125152im
```

Then with a naive approach we check if all the elements are in the LHP.

```
@benchmark all(real(vbig).<=0)
```

```
BenchmarkTools.Trial: 10000 samples with 226 evaluations per
sample.
Range (min ... max):  323.451 ns ... 71.935 μs  | GC (min ... max):
0.00% ... 98.29%
Time (median):       361.947 ns                  | GC (median):
0.00%
Time (mean ± σ):     582.819 ns ± 1.713 μs      | GC (mean ± σ):
27.61% ± 10.74%
```



Memory estimate: 1.78 KiB, allocs estimate: 6.

The *whole* vector of complex numbers gets converted to real and then we check row by row if it's non-positive. The whole check one by one results in a vector with booleans that gets checked one by one if it contains false values.


```
@benchmark all(<=(0),real(vbig))
```

BenchmarkTools.Trial: 10000 samples with 200 evaluations per sample.

```
Range (min ... max): 395.000 ns ... 102.246 µs | GC (min ... max):  
0.00% ... 99.04%  
Time (median): 457.500 ns | GC (median):  
0.00%  
Time (mean ± σ): 667.069 ns ± 1.728 µs | GC (mean ± σ):  
15.54% ± 9.07%
```



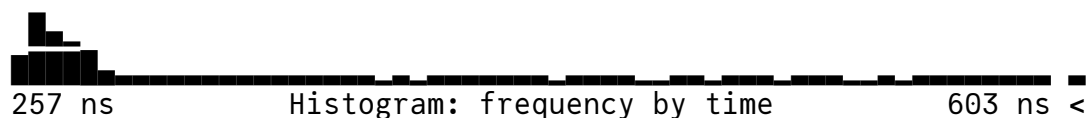
Memory estimate: 1.62 KiB, allocs estimate: 2.

For now we can skip the full evaluation of non-positivity: the first time it encounters a positive numbers it returns false. This improves the performance a little bit.

```
@benchmark all(i -> real(i)<=0,vbig)
```

BenchmarkTools.Trial: 10000 samples with 328 evaluations per sample.

```
Range (min ... max): 257.012 ns ... 1.215 µs | GC (min ... max):  
0.00% ... 0.00%  
Time (median): 270.122 ns | GC (median):  
0.00%  
Time (mean ± σ): 279.606 ns ± 55.243 ns | GC (mean ± σ):  
0.00% ± 0.00%
```



Memory estimate: 0 bytes, allocs estimate: 0.

This is the final form. 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.