

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
I	The Damped Mass problem	4
	Modeling	5
2	Block With Friction	6
2.1	State Space representation	6
2.2	Pole Placement	7
2.3	Simulation	9

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.
- 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
 - Differential Equations

Part I

The Damped Mass problem

Modeling

$$\begin{cases} \dot{x} = \mathbf{A}x + \mathbf{B}y \\ y = \mathbf{C}x + \mathbf{D}y \end{cases}$$

2 Block With 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
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)))
```

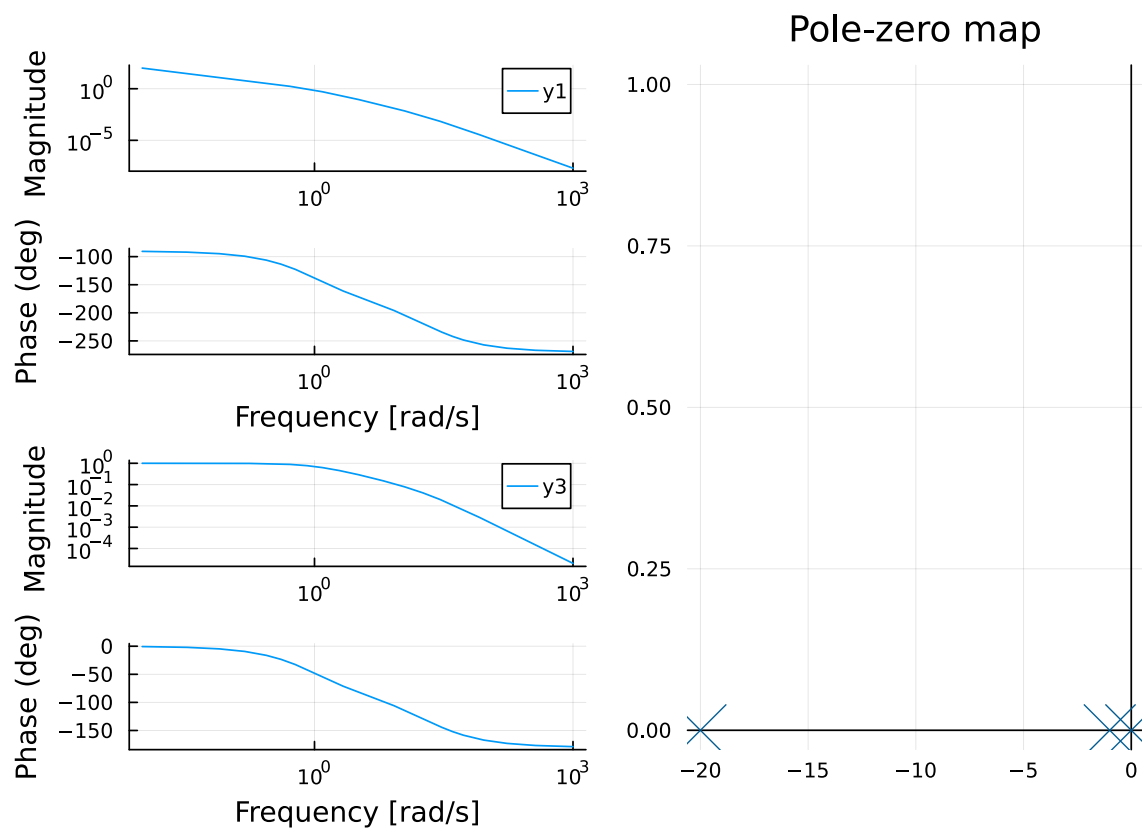


Figure 2.1: Starting Bode Plot and PZ Map

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.

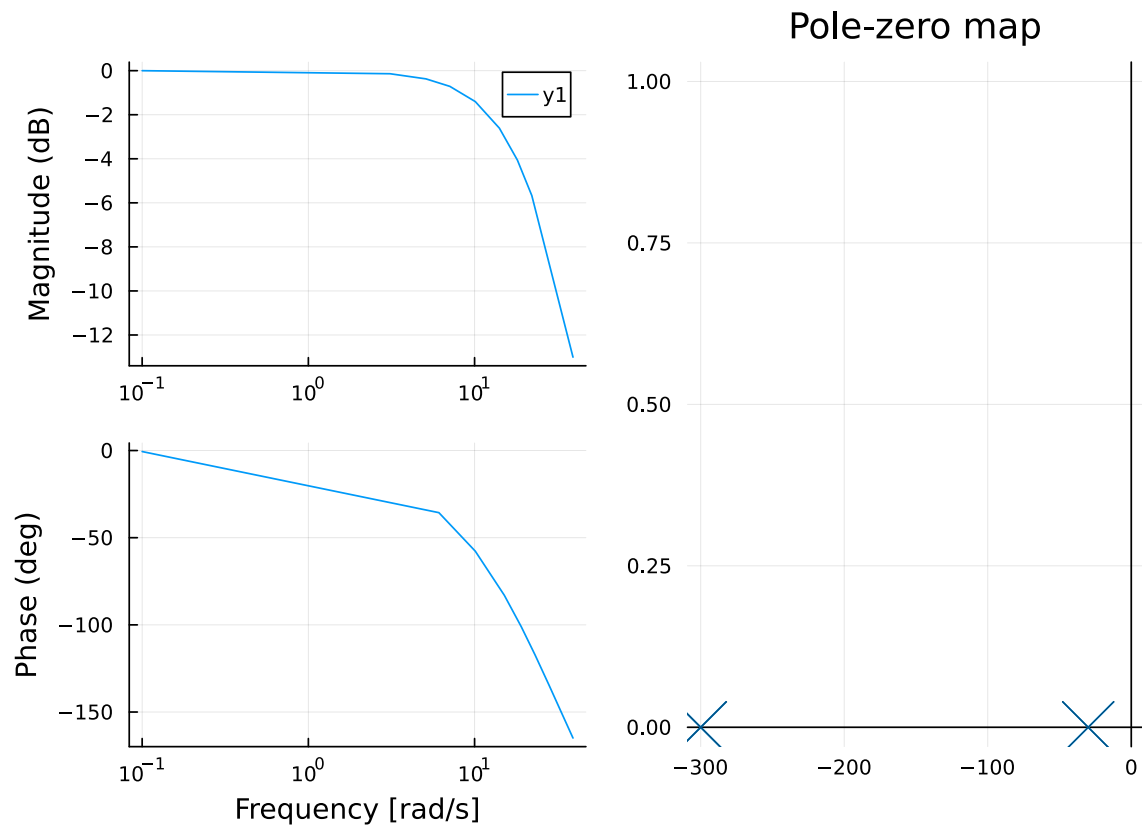
For some reason pole placement doesn't work for the observer, I use a Kalman Filter with random fast values.

```
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
```

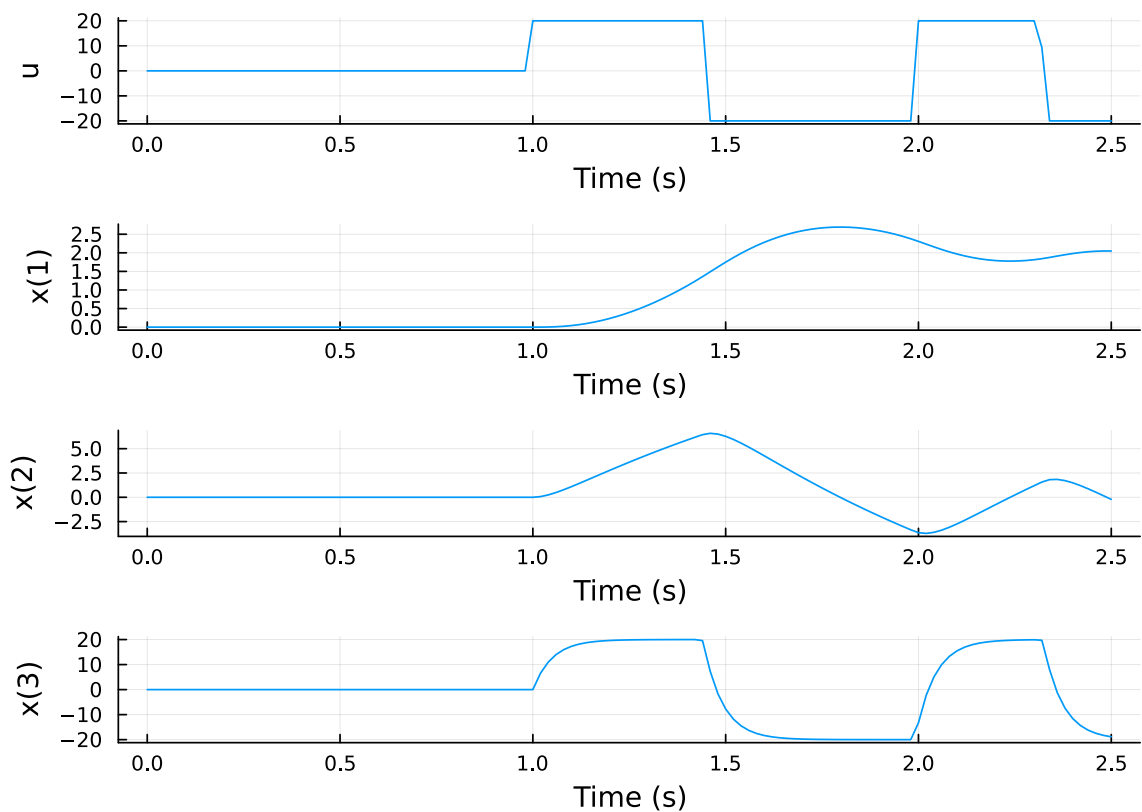
```

# 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)

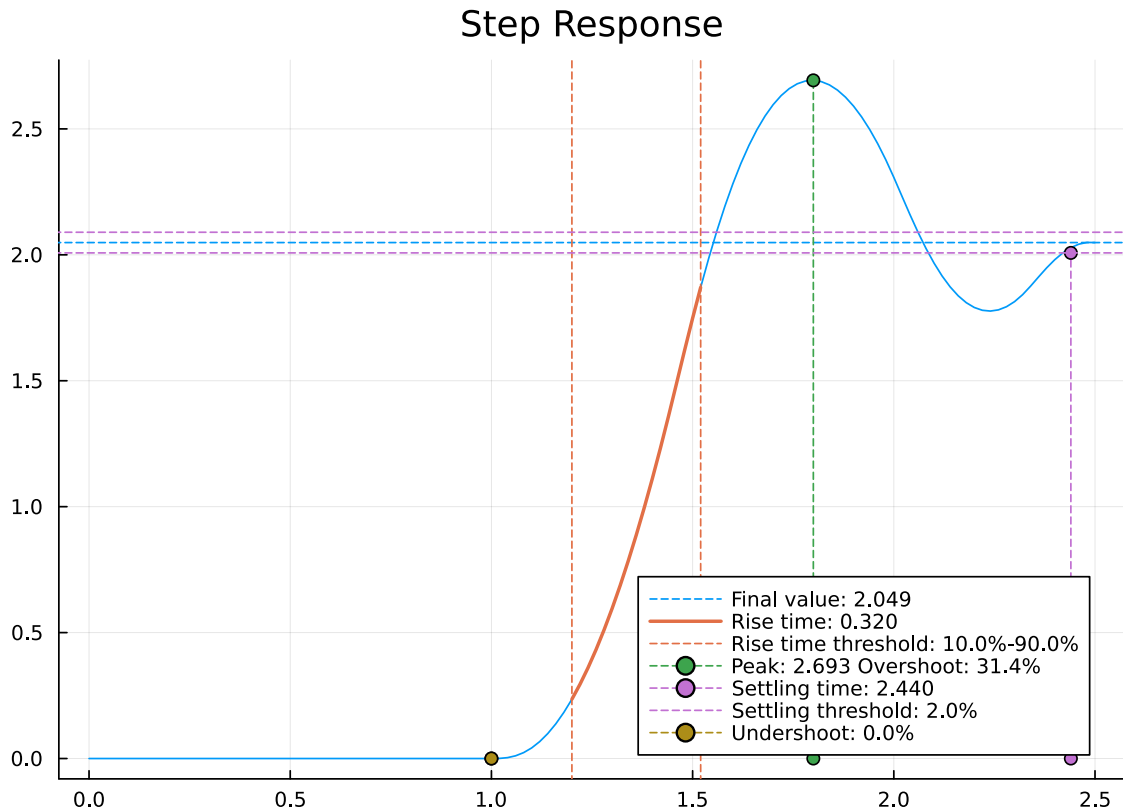
display(plot(res,
    plotu=true,
    plotx=true,
    ploty=false
))
ylabel!("u", sp=1);
ylabel!("x", sp=2);
ylabel!("v", sp=3);
ylabel!("T", sp=4);

```



For more stats:

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



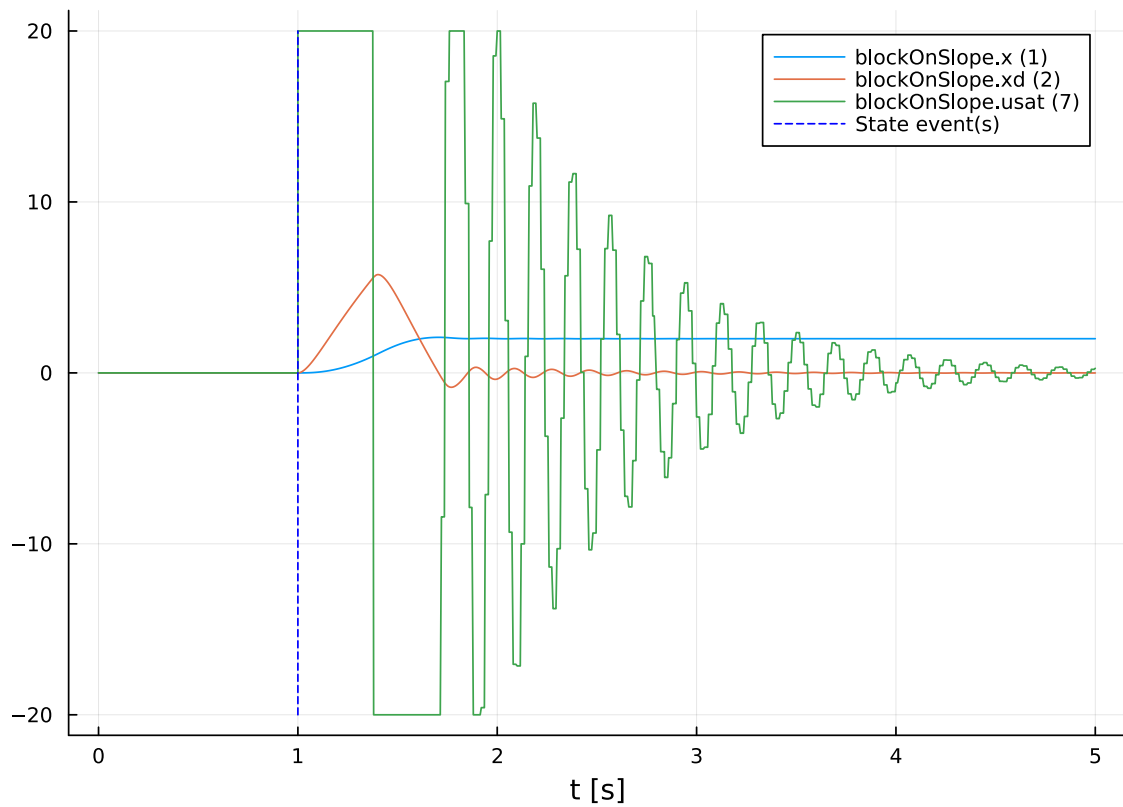
We can also simulate it in a SIMULINK-like environment:

```
using FMI, DifferentialEquations  
fmuPath = abspath(joinpath(@__DIR__,  
    "..", "..",  
    "modelica",  
    "ControlChallenges",  
    "ControlChallenges.BlockOnSlope_Challenges.Examples.WithFrict",  
    "ion.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.