Computational Dynamics HW 4

Jackson Wills

28th February 2020

1 Building equations of motion

1.1 Determine kinetic energy, potential energy, and dissipation function of the system.

```
using SymPy
using Plots
@vars t f m1 m2 k1 k2 c1 c2 z1 z2 z3 z4 u
x1 = SymFunction("x1")(t)
x2 = SymFunction("x2")(t)
xdot1 = SymFunction("xdot1")(t)
xddot1 = SymFunction("xddot1")(t)
xdot2 = SymFunction("xdot2")(t)
xddot2 = SymFunction("xddot2")(t)
diff(x1,t)
diff(x1,t,t)
diff(diff(x1,t),t);
```

defining system

```
q = [x1;x2]
```

$$\begin{bmatrix} \mathrm{x}_1(t) \\ \mathrm{x}_2(t) \end{bmatrix}$$

```
Q = [0;f]
```

$$\begin{bmatrix} 0 \\ f \end{bmatrix}$$

1.1.1 Kinetic Energy

```
T = 1//2*m1*(diff(x1,t))^2 + 1//2*m2*(diff(x2,t))^2 |> subs(diff(x1,t),xdot1) |>
subs(diff(x2,t),xdot2)
```

$$\frac{m_1 \dot{x}_1^2(t)}{2} + \frac{m_2 \dot{x}_2^2(t)}{2}$$

1.1.2 Potential Engergy

```
V = 1//2*k1*x1^2 + 1//2*k2*(x2-x1)^2
```

$$\frac{k_1\,\mathrm{x_1}^2\,(t)}{2}+\frac{k_2\,(-\,\mathrm{x_1}\,(t)+\mathrm{x_2}\,(t))^2}{2}$$

1.1.3 Dissipation Functon

```
D = 1//2*c1*(diff(x1,t))^2 + 1//2*c2*(diff(x2,t)-diff(x1,t))^2 |> subs(diff(x1,t),xdot1)
|> subs(diff(x2,t),xdot2)
```

$$\frac{c_1\dot{x}_1^2(t)}{2}+\frac{c_2\,(-\dot{x}_1(t)+\dot{x}_2(t))^2}{2}$$

1.2 Lagrange's Equation to construct the equations of motion.

```
L = T-V
```

```
Q1 = diff(diff(L,xdot1),t) - diff(L,x1) + diff(D,xdot1)|> subs(diff(xdot1,t),xddot1) |>
subs(diff(xdot2,t),xddot2)
Q2 = diff(diff(L,xdot2),t) - diff(L,x2) + diff(D,xdot2) |> subs(diff(xdot1,t),xddot1)|>
subs(diff(xdot2,t),xddot2)
eqn1 = Q[1] - Q1 #equal to zero
```

$$-c_1\dot{x}_1(t)-\frac{c_2\,(2\dot{x}_1(t)-2\dot{x}_2(t))}{2}-k_1\,\mathrm{x_1}\,(t)-\frac{k_2\,(2\,\mathrm{x_1}\,(t)-2\,\mathrm{x_2}\,(t))}{2}-m_1\ddot{x}_1(t)$$

```
eqn2 = Q[2] - Q2 #equal to zero
```

$$-\frac{c_2\,(-2\dot{x}_1(t)+2\dot{x}_2(t))}{2}+f-\frac{k_2\,(-2\,\mathrm{x_1}\,(t)+2\,\mathrm{x_2}\,(t))}{2}-m_2\ddot{x}_2(t)$$

1.3 Hamilton′s equation to construct the equations of motion

```
P1 = diff(L,xdot1)
```

$$m_1\dot{x}_1(t)$$

```
P2 = diff(L,xdot2)
```

$$m_2\dot{x}_2(t)$$

```
@vars p1 p2
zero1 = P1 - p1
zero2 = P2 - p2
sol = solve( [zero1,zero2] , [xdot1,xdot2])
```

```
Dict{Any,Any} with 2 entries:
  xdot1(t) => p1/m1
  xdot2(t) => p2/m2
```

```
Ham = T + V |> subs(sol)
```

$$\frac{k_1 \mathrm{x}_1{}^2(t)}{2} + \frac{k_2 \left(-\mathrm{x}_1(t) + \mathrm{x}_2(t)\right)^2}{2} + \frac{p_2^2}{2m_2} + \frac{p_1^2}{2m_1}$$

```
Qdamping1 = -diff(D,xdot1) |> subs(sol)
Qdamping2 = -diff(D,xdot2) |> subs(sol)


reversesol = solve( [zero1,zero2] , [p1,p2])

pdot1 = -diff(Ham,x1)+Qdamping1 + Q[1] |> subs(reversesol)
```

$$-c_1 \dot{x}_1(t) - \frac{c_2 \left(2\dot{x}_1(t) - 2\dot{x}_2(t)\right)}{2} - k_1 \mathrm{x}_1(t) - \frac{k_2 \left(2\,\mathrm{x}_1(t) - 2\,\mathrm{x}_2(t)\right)}{2}$$

```
pdot2 = -diff(Ham,x2)+Qdamping2 + Q[2] |> subs(reversesol)
```

$$-\frac{c_2 \left(-2\dot{x}_1(t) + 2\dot{x}_2(t)\right)}{2} + f - \frac{k_2 \left(-2\,\mathrm{x}_1(t) + 2\,\mathrm{x}_2(t)\right)}{2}$$

1.4 pdot1 = m1 * xddot1 and pdot2 = m2 * xddot2 so the expressions for pdot1 and pdot2 are the equations of motion

2 Control

2.1 State Space Representation of System.

z = x - x0

u = f - f0

z = [x1; x2; xdot1; xdot2]

u = [0;f]

```
rule = Dict(x1=>z1,x2=>z2,xdot1=>z3,xdot2=>z4,f=>u)
zdot = [xdot1;xdot2;pdot1/m1;pdot2/m2]
zdot = zdot.subs(reversesol)
zdot = zdot.subs(rule)
```

$$\begin{bmatrix} z_3 \\ z_4 \\ \frac{-c_1 z_3 - \frac{c_2(2z_3 - 2z_4)}{2} - k_1 z_1 - \frac{k_2(2z_1 - 2z_2)}{2}}{m_1} \\ \frac{-\frac{c_2(-2z_3 + 2z_4)}{2} - \frac{k_2(-2z_1 + 2z_2)}{2} + u}{m_2} \end{bmatrix}$$

---

now I'll plug in values and get the A Matrix

```
values = Dict(m1=>1,m2=>1,k1=>20,k2=>10,c1=>.4,c2=>.2)
A = fill(exp(xddot2^xddot1),(4,4))
let A = A
    for i = 1:4
        A[i,1] = diff(zdot[i],z1)
```

```
        A[i,2] = diff(zdot[i],z2)
        A[i,3] = diff(zdot[i],z3)
        A[i,4] = diff(zdot[i],z4)
    end
    return A
end
A = A.subs(values)
afloat = fill(NaN,(size(A,1),size(A,2)))
A = oftype(afloat,A)
```

```
4×4 Array{Float64,2}:
   0.0    0.0   1.0   0.0
   0.0    0.0   0.0   1.0
 -30.0   10.0  -0.6   0.2
  10.0  -10.0   0.2  -0.2
```

Now for the B Matrix

```
B = fill(exp(xddot2^xddot1),(4,1))
let B = B
    for i = 1:4
        B[i,1] = diff(zdot[i],u)
    end
    return B
end
B = B.subs(values)
bfloat = fill(NaN,(size(B,1),size(B,2)))
B = oftype(bfloat,B)
```

```
4×1 Array{Float64,2}:
 0.0
 0.0
 0.0
 1.0
```

```
C = [0 1 0 0.]
D = [0.];
```

2.2 Determine if the system is controllable and observable

2.2.1 Controlability

```
Cm = hcat(B,A*B,A^2*B,A^3*B)
Om = vcat(C,C*A,C*A^2,C*A^3)
```

```
import LinearAlgebra
check1 = LinearAlgebra.det(Cm) #not zero so full rank
```

```
-100.00000000000003
```

```
(blah1,check2,blah2) = LinearAlgebra.svd(Cm)
    check2 # 4 non zero singular values, so rank=4
```

```
4-element Array{Float64,1}:
 25.172073368211034
  8.582299701082098
  0.8275406052494161
  0.5593556407784597
```

```
check3 = LinearAlgebra.eigvals(Cm) # no zero eigenvalues, so full rank
```

```
4-element Array{Complex{Float64},1}:
 -4.901911175483425 + 0.0im
 0.9612402445965968 - 0.11416855069044601im
 0.9612402445965968 + 0.11416855069044601im
 21.771430686290227 + 0.0im
```

```
check4 = LinearAlgebra.rank(Cm) # rank = 4, so full rank
```

```
4
```

2.2.2 Observability

```
check5 = LinearAlgebra.det(Om) # not zero so full rank
```

```
-100.00000000000001
```

```
(blah3,check6,blah4) = LinearAlgebra.svd(Om)
    check2 # 4 non zero singular values, so rank=4
```

```
4-element Array{Float64,1}:
 25.172073368211034
  8.582299701082098
  0.8275406052494161
  0.5593556407784597
```

```
check7 = LinearAlgebra.eigvals(Om) # no zero eigenvalues, so full rank
```

```
4-element Array{Complex{Float64},1}:
 -11.03804378694272 + 0.0im
 0.2128580846285345 - 3.179219742966267im
 0.2128580846285345 + 3.179219742966267im
 0.8923276176856609 + 0.0im
```

```
check8 = LinearAlgebra.rank(Om) # rank = 4, so full rank
```

```
4
```

3 Make A Controller

3.1 This time using Ackerman's Formula

```
import Polynomials
OS = 5/100
Ts = .25
zeta = sqrt(log(OS)^2/(pi^2+log(OS)^2))
omega = 4/Ts/zeta
Ts2 = 5*Ts
omega2 = 4/Ts2/zeta
# first 2 roots
s1 = -zeta*omega+1im*omega*sqrt(1 - zeta^2)
```

```
-16.0 + 16.77903025619982im
```

```
#second 2 roots
s2 = -zeta*omega2+1im*omega2*sqrt(1 - zeta^2)
```

```
-3.2 + 3.3558060512399646im
```

```
p1 = Polynomials.poly([s1, conj(s1), s2, conj(s2)  ])
Lambda = zeros(4,4)
for i = 0:4
        global Lambda += p1.a[i+1]*A^i
end
Lambda = real(Lambda)

e_c = hcat(B, A*B, A^2*B, A^3*B)
K = (e_c\Lambda)[end,:]
```

```
4-element Array{Float64,1}:
 -801.9055120259093
  645.8948998080115
  276.51195391998175
   37.6
```

check that the placement of the poles is where I wanted them

```
LinearAlgebra.eigvals(A - B*K')
```

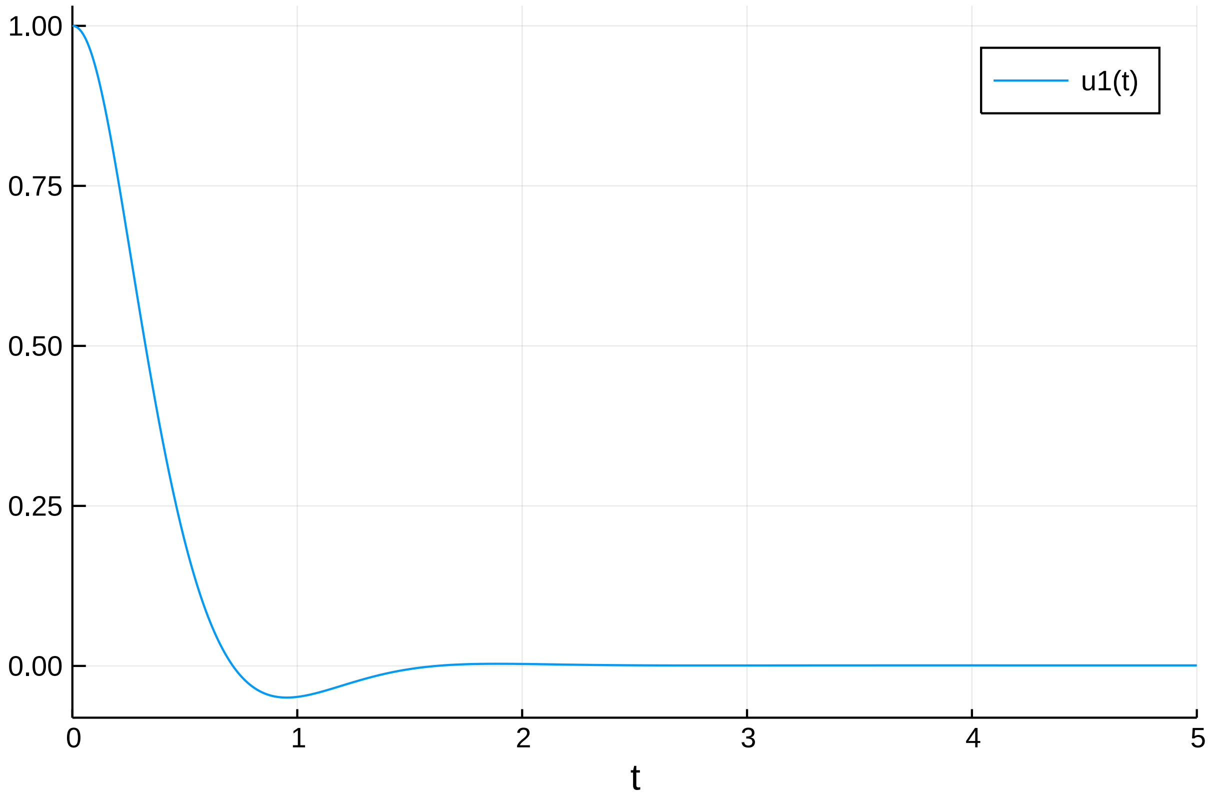
```
4-element Array{Complex{Float64},1}:
 -15.999999999999998 - 16.779030256199825im
 -15.999999999999998 + 16.779030256199825im
  -3.199999999999999 - 3.355806051239971im
  -3.199999999999999 + 3.355806051239971im
```

That is what I wanted! Now I'll test the controller

```
function ode2MassSprings(dz,z,p,t)
    # p = [M1,M2,K1,K2,C1,C2]

    x0 = [0 0 0 0.]
    r(t) = 1
    x = z - x0
    U = r(t) - LinearAlgebra.dot(K,x)

    dz[1] = z[3]
    dz[2] = z[4]
    dz[3] = (-p[5]*z[3] - p[6]*(2*z[3] - 2*z[4])/2 -p[3]*z[1] - p[4]*(2*z[1] -
2*z[2])/2)/p[1]
    dz[4] = (-p[6]*(-2*z[3] + 2*z[4])/2 - p[4]*(-2*z[1] + 2*z[2])/2 + U)/p[2]
end

import DifferentialEquations
tspan = (0.0,5)
z0 = [1 2 0 0.]
p = [1,1,20,10,.4,.2]
prob = DifferentialEquations.ODEProblem(ode2MassSprings,z0,tspan,p)
sol = DifferentialEquations.solve(prob);
```

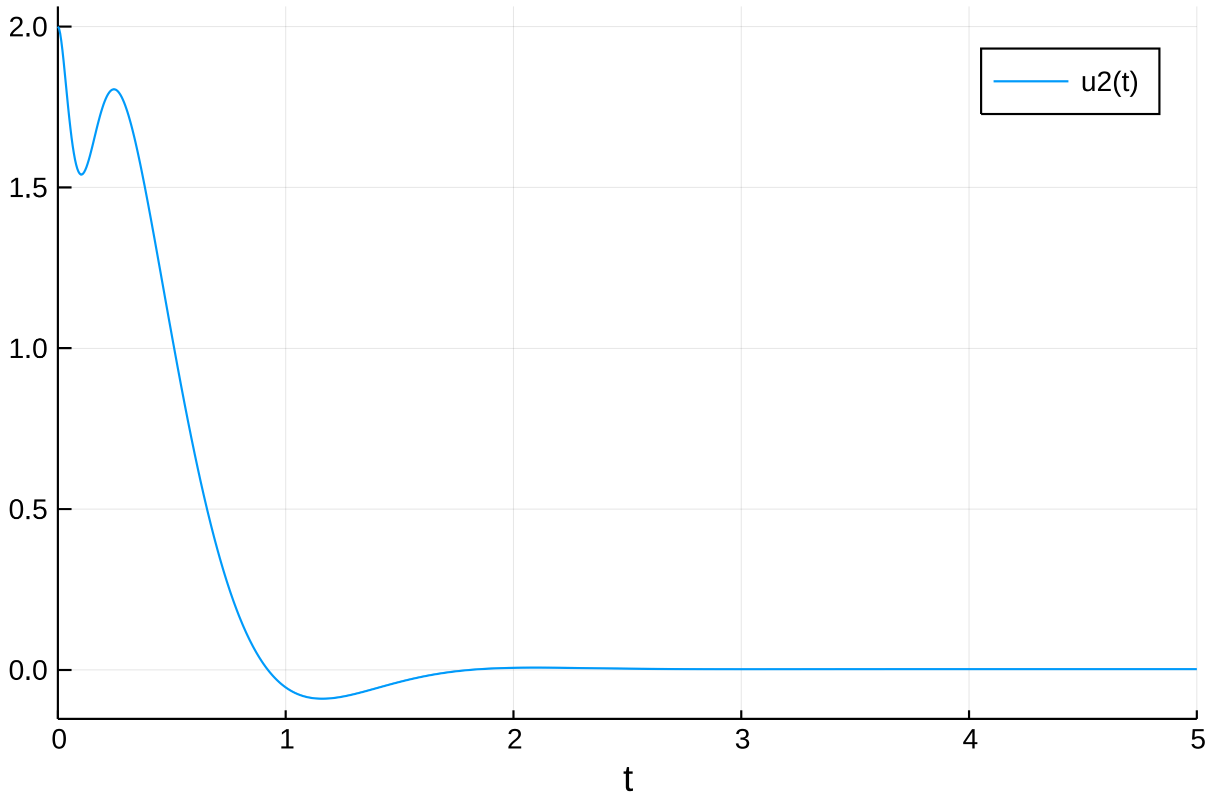
The time response of x1 in the controlled system

```
plot(sol, vars = (0,1))
```



The time response of x1 in the controlled system

```
plot(sol, vars = (0,2))
```

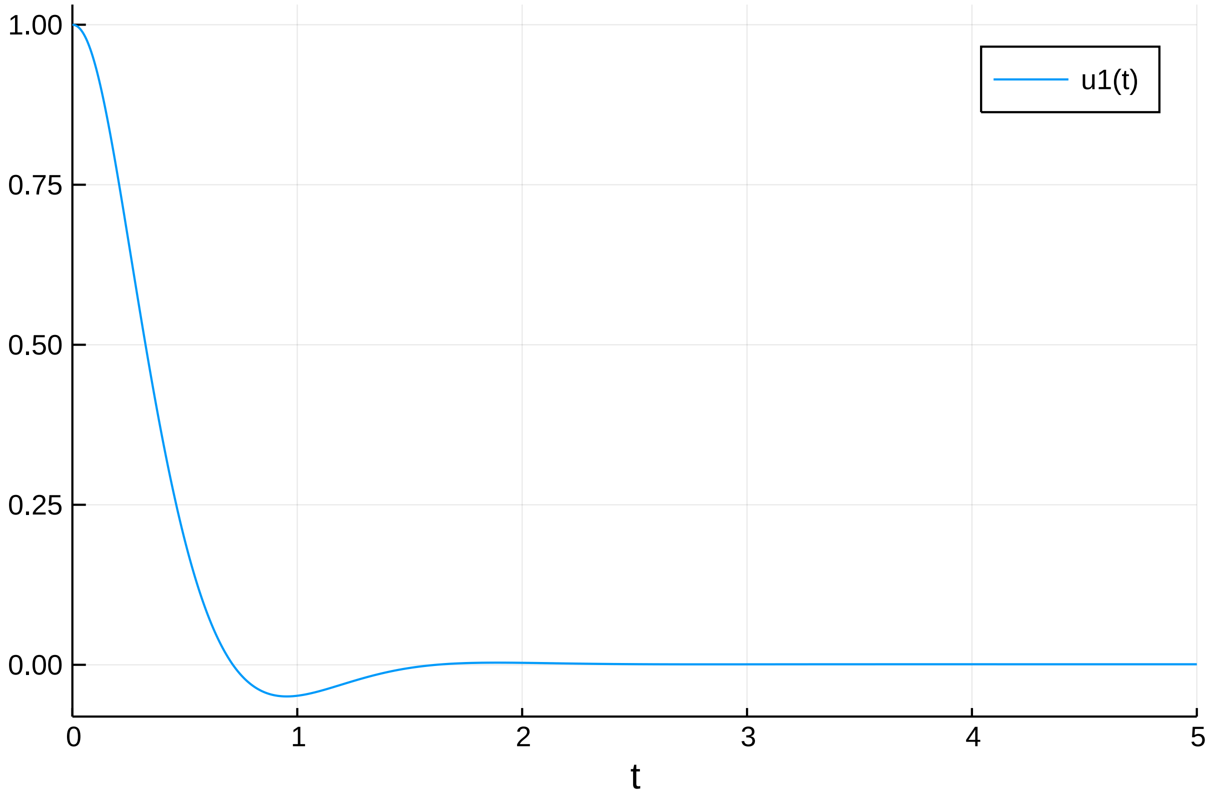


3.1.1 NOW im going to do the problem again in controller cononical form to see if I get the same K values

```
eigsA = LinearAlgebra.eigvals(A)
OLCharEqn = Polynomials.poly(eigsA)
OLCharEqnCoeffs = Polynomials.coeffs(OLCharEqn)

Abar = [0 1 0 0;0 0 1 0;0 0 0 1;-OLCharEqnCoeffs[1:end-1]'] |> real
Bbar = [0;0;0;1]

e_cz = hcat(Bbar, Abar*Bbar, Abar^2*Bbar, Abar^3*Bbar)
P = e_c/e_cz

@vars K1 K2 K3 K4
Kchecksymbols = [K1 K2 K3 K4]
Matrix = Abar-Bbar*Kchecksymbols

# p1 is the previously defined desired characteristic equation

DesiredCoeffs = Polynomials.coeffs(p1) |> real
zero3 = Matrix[4] + DesiredCoeffs[1]
zero4 = Matrix[8] + DesiredCoeffs[2]
zero5 = Matrix[12] + DesiredCoeffs[3]
zero6 = Matrix[16] + DesiredCoeffs[4]
Kvals = solve([zero3,zero4,zero5,zero6],[K1,K2,K3,K4])
Kcheckz = oftype(zeros(1,4),Kchecksymbols.subs(Kvals))
Kcheckx = Kcheckz/P

1×4 Array{Float64,2}:
 -801.906  645.895  276.512  37.6

LinearAlgebra.eigvals(Abar-Bbar*Kcheckx)
A_z = Abar-Bbar*Kcheckz
A_x = A-B*Kcheckx;

LinearAlgebra.eigvals(A - B*Kcheckx)

function odeCheck(dz,z,p,t)
        #p = [M1,M2,K1,K2,C1,C2]

    x0 = [0 0 0 0.]
    r(t) = 1
    x = z - x0
    U = r(t) - LinearAlgebra.dot(Kcheckx,x)

    dz[1] = z[3]
    dz[2] = z[4]
    dz[3] = (-p[5]*z[3] - p[6]*(2*z[3] - 2*z[4])/2 -p[3]*z[1] - p[4]*(2*z[1] -
2*z[2])/2)/p[1]
    dz[4] = (-p[6]*(-2*z[3] + 2*z[4])/2 - p[4]*(-2*z[1] + 2*z[2])/2 + U)/p[2]

end

tspan = (0.0,5)
z0 = [1 2 0 0.]
p = [1,1,20,10,.4,.2]
probCheck = DifferentialEquations.ODEProblem(odeCheck,z0,tspan,p)
solCheck = DifferentialEquations.solve(probCheck);
```

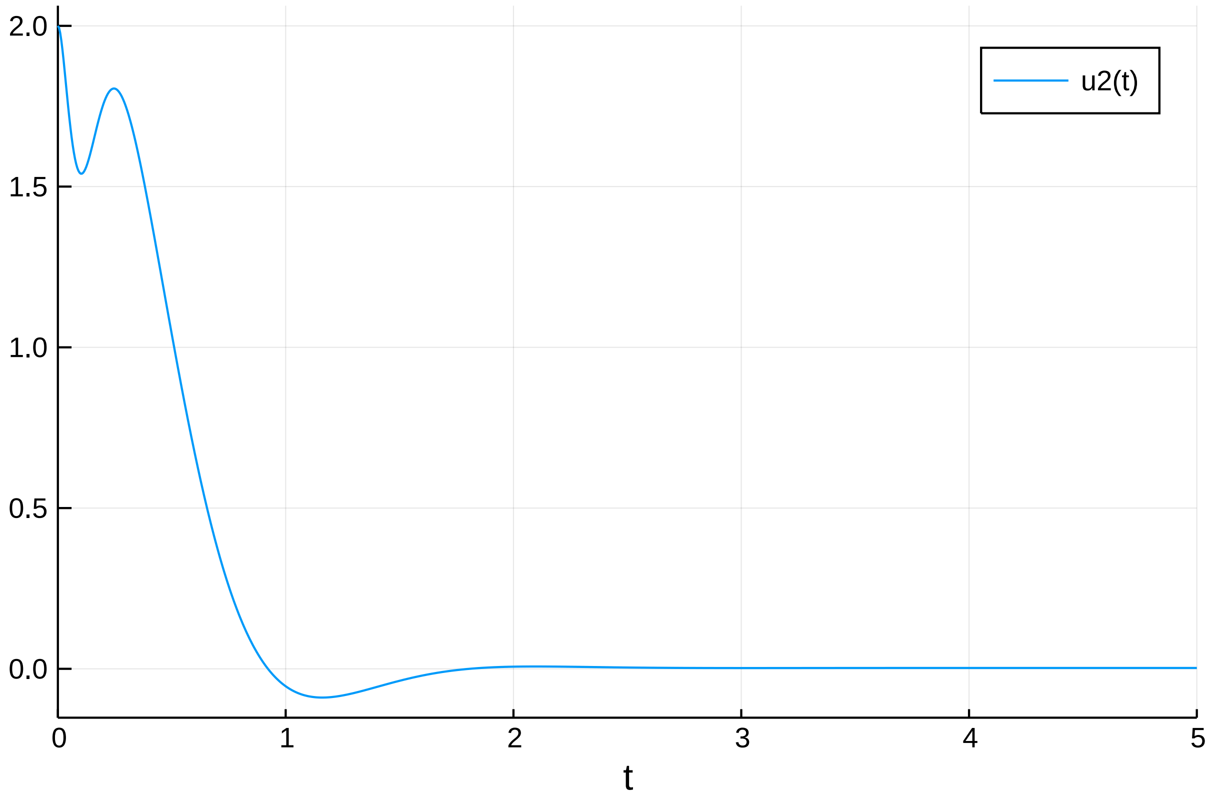
A check on the time response of x1 in the controlled system

```
plot(solCheck, vars = (0,1))
```



A check on the time response of x1 in the controlled system

```
plot(solCheck, vars = (0,2))
```



OKAY thats good! The plots look the same as when I did this with Ackermans!!!

```
s01 = 10*real(s1)+imag(s1)*1im
s02 = 10*real(s2)+imag(s2)*1im
p01 = Polynomials.poly([s01,conj(s01),s02,conj(s02)])

LambdaObs = zeros(4,4)
for i = 0:4
        global LambdaObs += p01.a[i+1]*A'^i
end
LambdaObs = real(LambdaObs)

e_cObs = hcat(C', A'*C', A'^2*C', A'^3*C')
L = (e_cObs\LambdaObs)[end,:]
```

```
4-element Array{Float64,1}:
 144031.88351404472
    383.19999999999993
      2.538091841259959e6
  47050.157290592004
```

check that the poles went where I wanted them to

```
LinearAlgebra.eigvals(A' - C'*L')
```

```
4-element Array{Complex{Float64},1}:
  -159.9999999999999 - 16.77903025620029im
  -159.9999999999999 + 16.77903025620029im
 -31.999999999999982 - 3.3558060512401im
 -31.999999999999982 + 3.3558060512401im
```

They did!

3.1.2 NOW to make an Observer in Controller cononical form

First, I'll make the desired polynomial

```
s01 = 10*real(s1)+imag(s1)*1im
s02 = 10*real(s2)+imag(s2)*1im
p01 = Polynomials.poly([s01, conj(s01), s02, conj(s02)  ])

ObsA = LinearAlgebra.transpose(A)
ObsB =  LinearAlgebra.transpose(C)
e_cxObs =  hcat(ObsB, ObsA*ObsB, ObsA^2*ObsB, ObsA^3*ObsB)
eigsObsA = LinearAlgebra.eigvals(ObsA)
OLCharEqnObs = Polynomials.poly(eigsObsA)
OLCharEqnCoeffsObs = Polynomials.coeffs(OLCharEqnObs)

AbarObs = [0 1 0 0;0 0 1 0;0 0 0 1;-OLCharEqnCoeffsObs[1:end-1]'] |> real

BbarObs = [0;0;0;1]

e_czObs = hcat(BbarObs, AbarObs*BbarObs, AbarObs^2*BbarObs, AbarObs^3*BbarObs)
PObs = e_cxObs/e_czObs

@vars L1 L2 L3 L4
L = [L1 L2 L3 L4]
MatrixObs = AbarObs-BbarObs*L

# p1 is the previously defined desired characteristic equation

DesiredCoeffsObs = Polynomials.coeffs(p01) |> real
zero7 = MatrixObs[4] + DesiredCoeffsObs[1]
zero8 = MatrixObs[8] + DesiredCoeffsObs[2]
zero9 = MatrixObs[12] + DesiredCoeffsObs[3]
zero10 = MatrixObs[16] + DesiredCoeffsObs[4]
Lvals = solve([zero7,zero8,zero9,zero10],[L1,L2,L3,L4])
Lcheckz = oftype(zeros(1,4),L.subs(Lvals))
Lcheckx = Lcheckz*inv(PObs)
```

```
1×4 Array{Float64,2}:
 1.44032e5  383.2  2.53809e6  47050.2
```

These are the same values as the first L.

But we'll check them anyway

```
AbarObs - BbarObs*Lcheckz
LinearAlgebra.eigvals(A'-C'*Lcheckx)
```

```
4-element Array{Complex{Float64},1}:
 -160.00000000000023 - 16.77903025620038im
 -160.00000000000023 + 16.77903025620038im
 -31.999999999999766 - 3.355806051241438im
 -31.999999999999766 + 3.355806051241438im
```

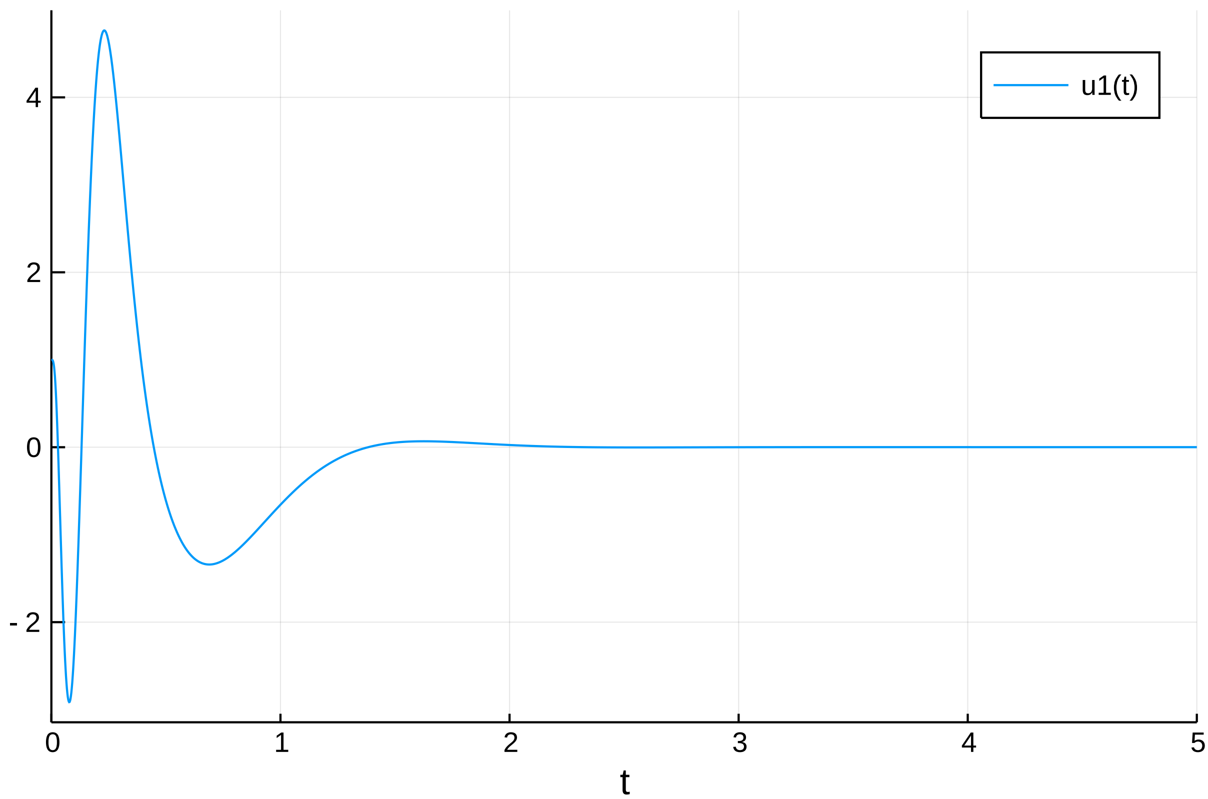
They went where I wanted them to go

Now, I'll test the response of the contolled and observed system:

```
function ode2MassSpringsObserver(dz,z,p,t)
	#p = [M1,M2,K1,K2,C1,C2]

	r(t) = 1
	x = [z[1];z[2];z[3];z[4]]
	xhat = [z[5];z[6];z[7];z[8]]
	y = C*x
	yhat = C*xhat
	U = r(t) - LinearAlgebra.dot(K,xhat)

	dz[1] = z[3]
	dz[2] = z[4]
	dz[3] = (-p[5]*z[3] - p[6]*(2*z[3] - 2*z[4])/2 -p[3]*z[1] - p[4]*(2*z[1] -
2*z[2])/2)/p[1]
	dz[4] = (-p[6]*(-2*z[3] + 2*z[4])/2 - p[4]*(-2*z[1] + 2*z[2])/2 + U)/p[2]
	dxhat = A*xhat + B*U + Lcheckx'*(y-yhat)

	dz[5:8] = dxhat

end


tspan = (0.0,5)
z0 = [1 2 0 0 0 0 0 0.]
p = [1,1,20,10,.4,.2]
prob = DifferentialEquations.ODEProblem(ode2MassSpringsObserver,z0,tspan,p)
sol = DifferentialEquations.solve(prob);
```

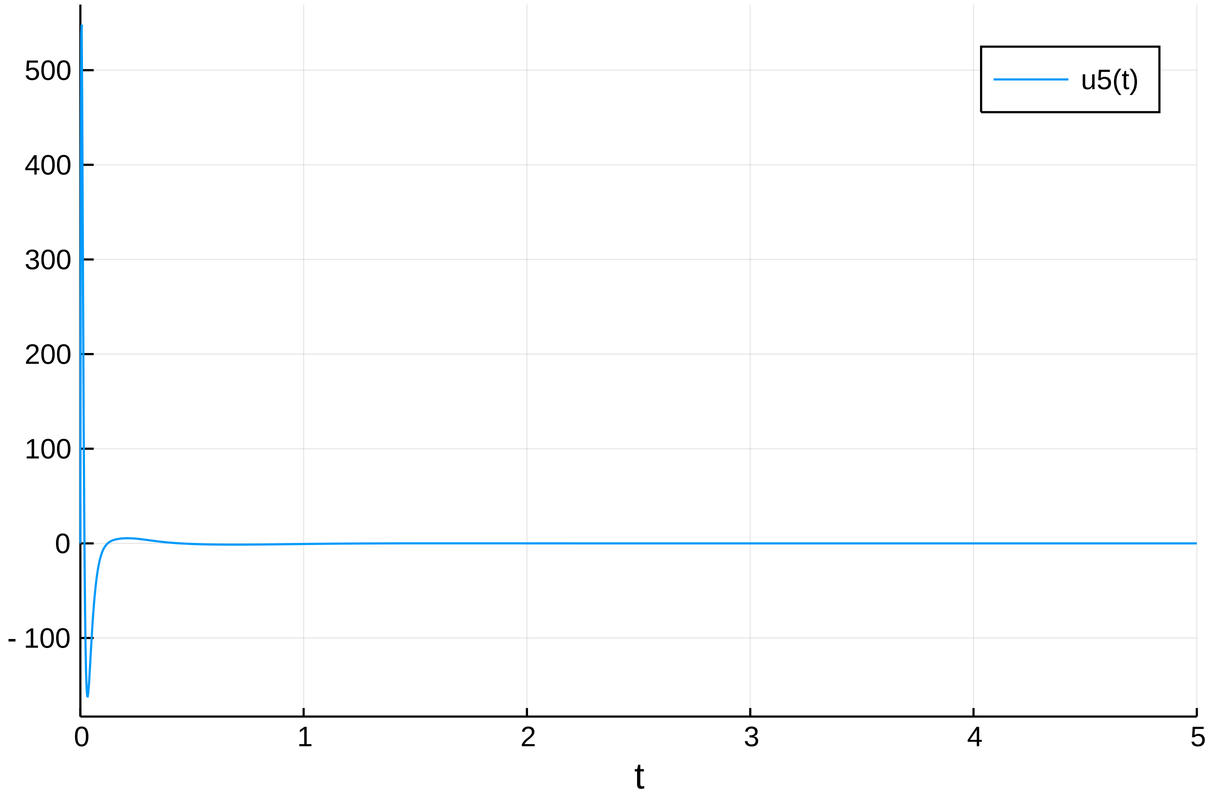
First, I'll plot x1

```
plot(sol, vars = (0,1))
```



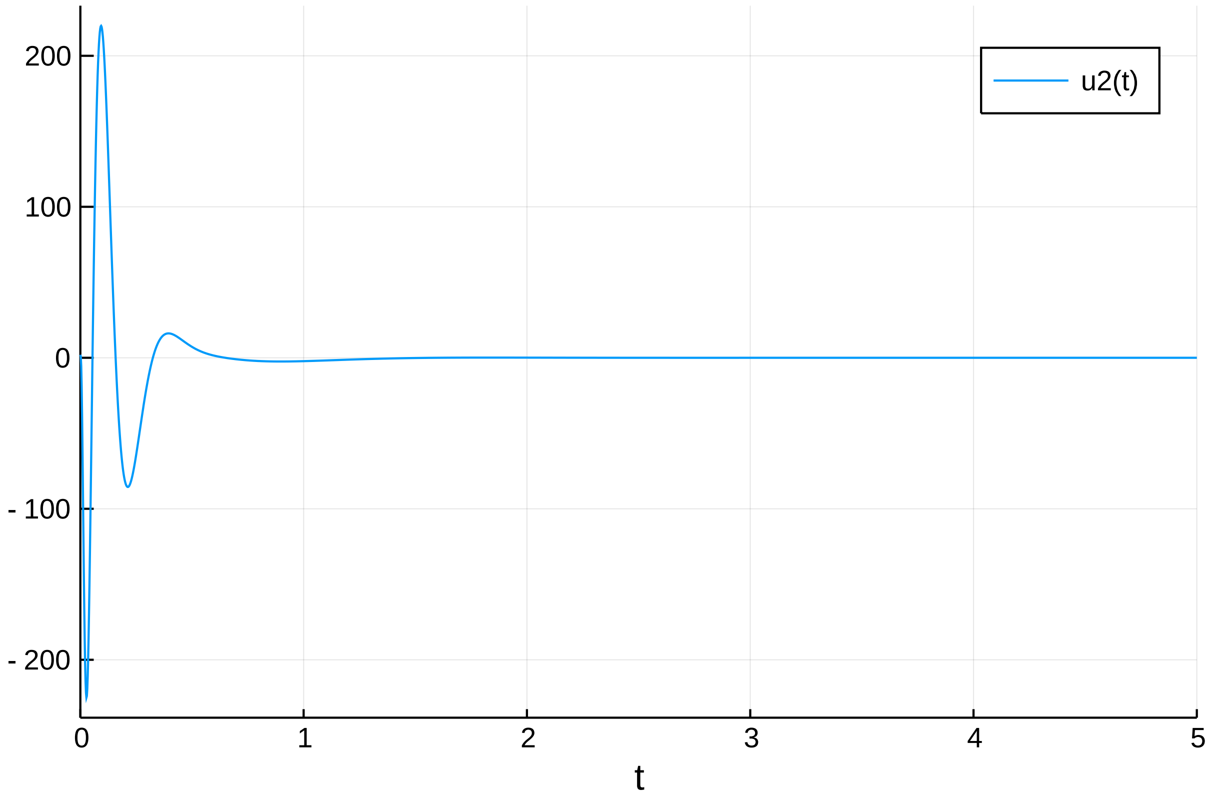
Now, I'll plot the Observer's estimate of x1

```
plot(sol, vars = (0,5))
```



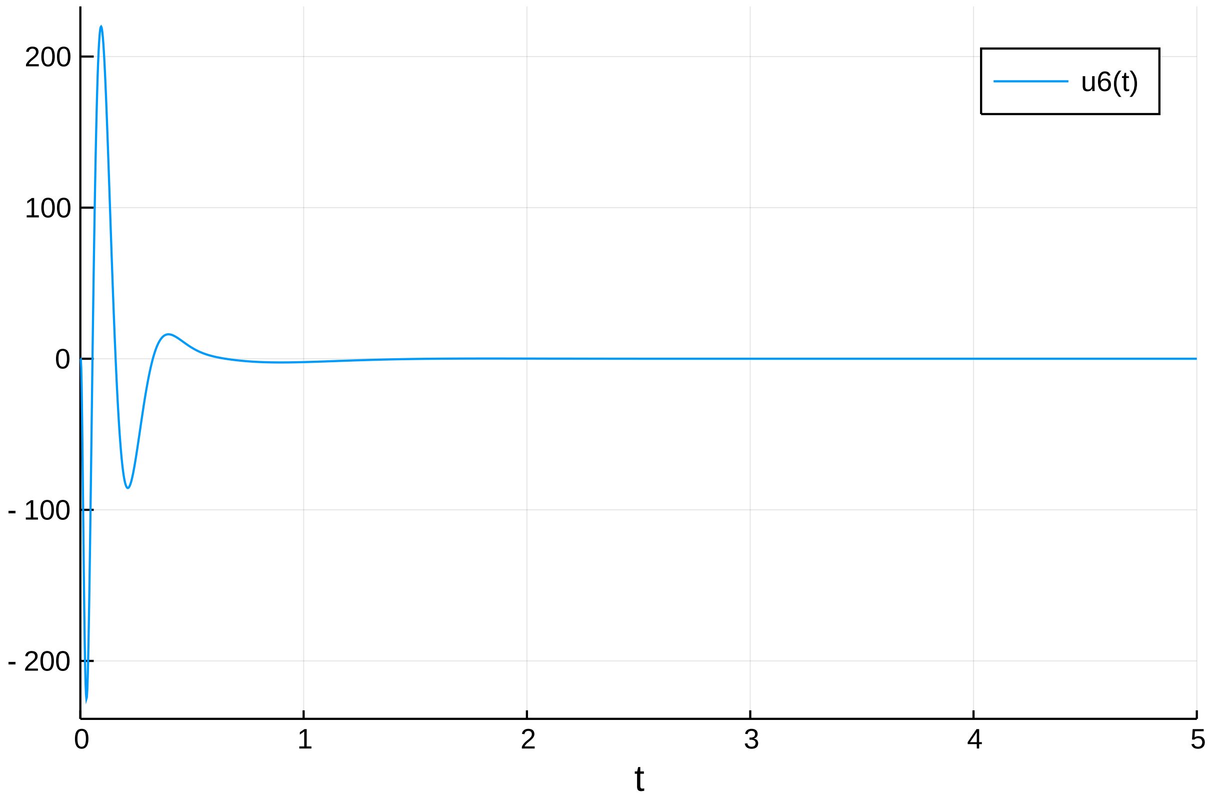
Next, I'll plot x2

```
plot(sol, vars = (0,2))
```



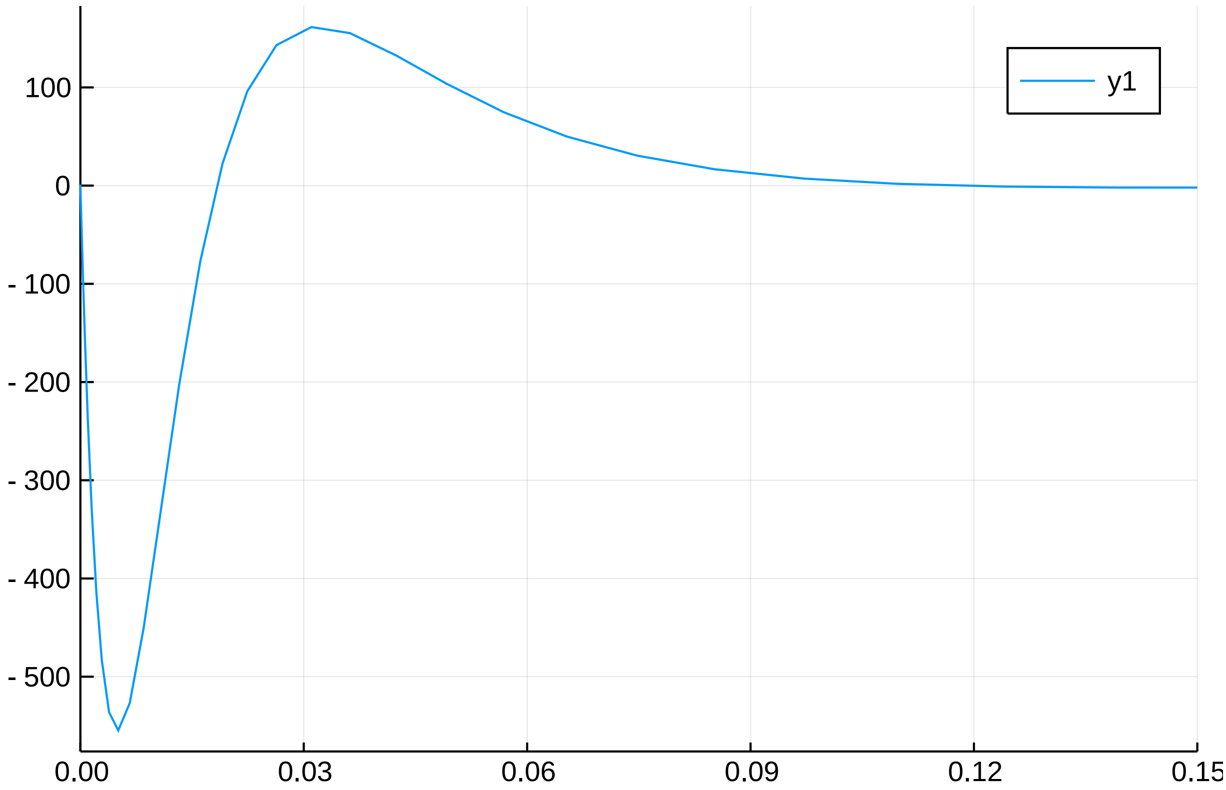
Now, I'll plot the Observer's estimate of x2

```
plot(sol, vars = (0,6))
```



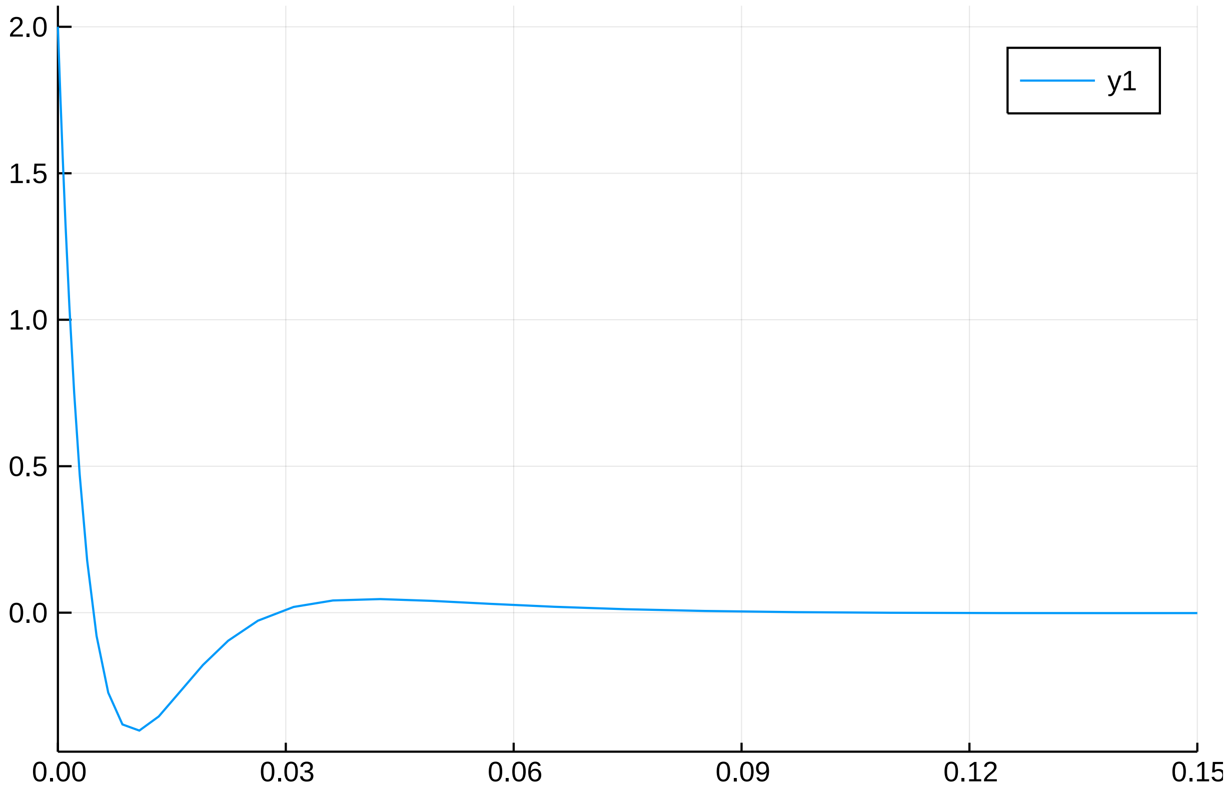
I'll plot the difference between the real time response of x1 and the observer's estimate of it

```
plot(sol.t,(sol[1,:]-sol[5,:]))
    xlims!(0,.15)
```



Finally, I'll plot the difference between the real time response of x2 and the observer's estimate of THAT

```
plot(sol.t,(sol[2,:]-sol[6,:]))
    xlims!(0,.15)
```



2.0
1.5
1.0
0.5
0.0
0.00
0.03
0.06
0.09
0.12
0.15
y1