test_updated

April 16, 2024

0.0.1 Part 2a - Install and Configure Julia Kernel

[ ]:
```
using Pkg
Pkg.status()
versioninfo()
```

```
Status `~/.julia/environments/v1.10/Project.toml`
  [336ed68f] CSV v0.10.14
  [a93c6f00] DataFrames v1.6.1
  [a98d9a8b] Interpolations v0.15.1
  [b6b21f68] Ipopt v1.6.2
  [4076af6c] JuMP v1.21.1
  [2fda8390] LsqFit v0.15.0
  [91a5bcdd] Plots v1.40.4
Julia Version 1.10.2
Commit bd47eca2c8a (2024-03-01 10:14 UTC)
Build Info:
  Official https://julialang.org/ release
Platform Info:
  OS: Linux (x86_64-linux-gnu)
  CPU: 8 × 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-15.0.7 (ORCJIT, tigerlake)
Threads: 1 default, 0 interactive, 1 GC (on 8 virtual cores)
Environment:
  LD_LIBRARY_PATH = /usr/lib/x86_64-linux-gnu/gazebo-11/plugins:/opt/ros/foxy/op
t/yaml_cpp_vendor/lib:/opt/ros/foxy/opt/rviz_ogre_vendor/lib:/opt/ros/foxy/lib/x
86_64-linux-gnu:/opt/ros/foxy/lib
  JULIA_NUM_THREADS =
```

0.0.2 Part 2b - Tire Force

[ ]:
```
using LsqFit
using CSV
using DataFrames
############## Here is a small example on how to use LsqFit ##############
# function model(input, par)
```

```julia
#     value = par[1] .* exp.( - input .* par[2])
#     return value
# end
# xdata = range(0, stop=10, length=20)
# ydata = model(xdata, [1.0 2.0]) + 0.01*randn(length(xdata))
# p0 = [0.5, 0.5]
# fit = curve_fit(model, xdata, ydata, p0)
# println("Your fitting value is: ", fit.param)
########## IMPORTANT COMMENT!!!: in function model(args), we add "." in front of mathematical operators to allow broadcasting (similar to Matlab) ##########


function magicFormula(input, par)
    # par = [B, C]
    # input = xdata
    #TODO Fill in the magic formula equation here
    alpha = input[:,1]
    Fz = input[:,2]
    mu = input[:,3]
    B = par[1]
    C = par[2]
    # Fy = mu .* Fz .* sin(C .* atan((B./mu).* alpha))

    Fy = mu .* Fz .* sin.(C .* atan.((B./mu) .* alpha))

    return Fy
end

TireForceDataFrame = CSV.read("TireForce.csv", DataFrame) # Load data in DataFrame mode, we recommend you to open csv to see the structure of data
TireForceMatrix = Matrix(TireForceDataFrame) # Change data format to matrix, it is formatted in the form of [alpha Fz mu Fy], each one is a N x 1 array

# TODO prepare xdata and ydata from TireForceMatrix
xdata = TireForceMatrix[:, 1:3]
ydata = TireForceMatrix[:, end]

p0      = [1.7, 9.5]; # Initial Guess of [B, C]

#TODO Fill in function similar to the above example
fit     = curve_fit(magicFormula,xdata,ydata,p0)

B       = round(fit.param[1]; digits = 4)
C       = round(fit.param[2]; digits = 3)
println("B coefficient is: " ,B, "  C Coefficient is: ", C)
```

```
B coefficient is: 5.68  C Coefficient is: 1.817
```

0.0.3 Part 2c - Vehicle Bicycle Model

[ ]:
```
function VehicleDynamics(states, control)
    la = 1.56
    lb = 1.64
    m = 2020
    g = 9.81
    Izz = 4095
    h = 0.6
    mu = 0.8
    x = states[1]
    y = states[2]
    v = states[3]
    r = states[4]
      = states[5]
    ux = states[6]
     f = states[7]
    ax = control[1]
    d f = control[2]

    Fzf =  m *g * (lb/(la + lb)) - (m*h)/(la+lb) * ax #TODO Front axle load
    Fzr =  m *g * (la/(la + lb)) + (m*h)/(la+lb) * ax #TODO Rear axle load

     f = d f - atan((v + la * r)/ux) #TODO Front slip angle
     r = -atan((v-lb*r)/ux) #TODO Rear slip angle

    Fyf = MagicFormula( f, Fzf, mu) #TODO Front lateral force
    Fyr = MagicFormula( r, Fzr, mu) #TODO Rear lateral force

    dx          = ux * cos( ) - v * sin( )#TODO
    dy          = ux * sin( ) + v * cos( )#TODO
    dv          = ((Fyf + Fyr)/m) - ux * r#TODO
    dr          = (Fyf * la - Fyr * lb)/Izz#TODO
    d           =  r #TODO
    dux         = ax #TODO
    d           = d f #TODO
    dstates     = [dx dy dv dr d  dux d ]
    return dstates
end


function MagicFormula(alpha, Fz, mu)
    B =     5.68 #TODO Input Q2b value here
    C =    1.817 #TODO Input Q2b value here
```

```
    Fy =  mu .* Fz .* sin.(C .* atan.((B./mu) .* alpha))  #TODO Lateral force␣
  ↪calculation
    return Fy
end


x0 = [-10.0 -5.0 0.5 0.1 0.1 10.0 0.1] # This is the initial state
ctrl = [1 0.1]  # One step control action
dstates = round.(VehicleDynamics(x0, ctrl); digits = 3) # Calculate states␣
  ↪derivative
println("The states derivative is: ", dstates)
```

```
The states derivative is: [9.9 1.496 -1.004 2.587 0.1 1.0 0.1]
```

[ ]:
```
transpose(dstates)
# dstates'
```

```
7×1 transpose(::Matrix{Float64}) with eltype Float64:
  9.9
  1.496
 -1.004
  2.587
  0.1
  1.0
  0.1
```

0.0.4 Part 2d - Vehicle Dynamics Propagation

[ ]:
```
# include("Q2c_VehicleDynamics.jl")

function Propagation(states, control,  T)
    #TODO Calculate states derivative using function
    #  VehicleDynamics(args).
    dstates =  VehicleDynamics(states, control)

    #TODO Calculate next Step
    # This is Explicit ForwardEuler
    statesNext = states + dstates .*  T
    return statesNext
end

#Testing purposes
x0 = [-10.0 -5.0 0.5 0.1 0.1 10.0 0.1] # This is the initial state
ctrl = [1 0.1]  # One step control action
 T = 0.01
# statesNextTemp = Propagation(x0,ctrl, T)
statesNextTemp = round.(Propagation(x0,ctrl, T); digits = 3)
```

```julia
println("The statesNextTemp  is: ", statesNextTemp)
```

```
The statesNextTemp  is: [-9.901 -4.985 0.49 0.126 0.101 10.01 0.101]
```

[ ]:
```julia
statesNext = StatesListFE02[1, :] .+ dstates .* dt1 #broadcast huh
```

```
7×7 Matrix{Float64}:
 -8.0  -10.0  -9.30233   -9.46313   -10.0  -9.8  -9.98
 -3.0   -5.0  -4.30233   -4.46313    -5.0  -4.8  -4.98
  2.0    0.0   0.697671   0.536874    0.0   0.2   0.02
  2.0    0.0   0.697671   0.536874    0.0   0.2   0.02
  2.0    0.0   0.697671   0.536874    0.0   0.2   0.02
 12.0   10.0  10.6977    10.5369     10.0  10.2  10.02
  2.0    0.0   0.697671   0.536874    0.0   0.2   0.02
```

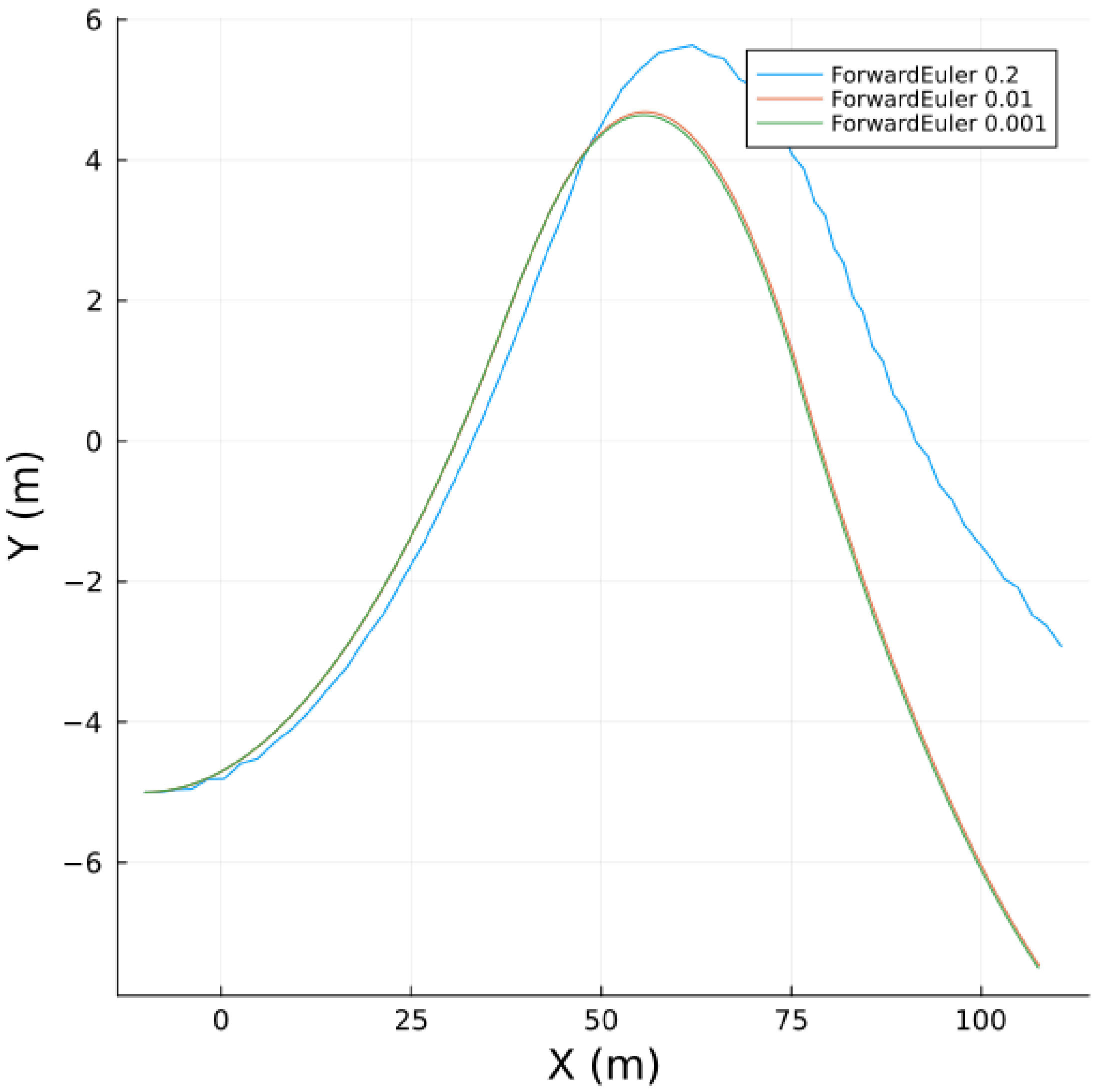
[ ]:
```julia
print(size(Propagation(reshape(StatesListFE02[1, :],(1,7)), control, dt1)))
```

```
(1, 7)
```

[ ]:
```julia
using Interpolations
using Plots
# include("Q2c_VehicleDynamics.jl")
# include("Q2d_StatesPropagator.jl")
x0 = [-10.0 -5.0 0.0 0.0 0.0 10.0 0.0]
ctrl = [1 0.1]
dstates = VehicleDynamics(x0, ctrl)


tc      = [0, 4, 8, 12]  # Key time step for control input
dfc    = [0, 0.02, -0.05, 0.02] # Key value for steering rate
axc     = [0, 1.0, -2.0, 1.0] # Key value for acceleration
dt1       = 0.2  # Simulation dt
t1        = 0:dt1:tc[end]
Interpolatedf = interpolate((tc ,), dfc, Gridded(Constant{Next}())) #
 ↪Interpolations
Interpolateax = interpolate((tc ,), axc, Gridded(Constant{Next}()))
df1    = Interpolatedf.(t1) # Get interpolated steering rate signal
ax1     = Interpolateax.(t1) # Get interpolated acceleration signal

StatesListFE02 = zeros(size(t1, 1), size(x0, 2)) # Initialize states list for 0.
 ↪2 update time
StatesListFE02[1, :] = x0 # Initial point

control = zeros(2,1) # Init control input

for i = 1:size(StatesListFE02, 1) - 1
```

```
    # TODO calculate the next states
    control[1] = ax1[i]
    control[2] = d f1[i]
    StatesListFE02[i + 1, :] = Propagation(reshape(StatesListFE02[i, :],(1,7)), control, dt1)
end

dt2      = 0.01 # Smaller time step
t2       = 0:dt2:tc[end]
d f2      = Interpolated f.(t2)
ax2       = Interpolateax.(t2)

StatesListFE001 = zeros(size(t2, 1), size(x0, 2)) # Initialize states list for 0.01 update time
StatesListFE001[1, :] = x0 # Initial point

for i = 1:size(StatesListFE001, 1) - 1

    # TODO calculate the next states
    control[1] = ax2[i]
    control[2] = d f2[i]
    StatesListFE001[i + 1, :] = Propagation(reshape(StatesListFE001[i, :],(1,7)), control, dt2)
end


dt3      = 0.001 # Smaller time step
t3       = 0:dt3:tc[end]
d f3      = Interpolated f.(t3)
ax3       = Interpolateax.(t3)

StatesListFE0001 = zeros(size(t3, 1), size(x0, 2)) # Initialize states list for 0.001 update time
StatesListFE0001[1, :] = x0 # Initial point

for i = 1:size(StatesListFE0001, 1) - 1

    # TODO calculate the next states
    control[1] = ax3[i]
    control[2] = d f3[i]
    StatesListFE0001[i + 1, :] = Propagation(reshape(StatesListFE0001[i, :],(1,7)), control, dt3)
end

p = plot(size = [600, 600])
```

```
plot!(p, StatesListFE02[:, 1], StatesListFE02[:, 2], label = "ForwardEuler " *␣
 ↪string(dt1), tickfontsize = 10, xlabel = "X (m)", ylabel = "Y␣
 ↪(m)",guidefont=15)
plot!(p, StatesListFE001[:, 1], StatesListFE001[:, 2], label = "ForwardEuler "␣
 ↪* string(dt2))
plot!(p, StatesListFE0001[:, 1], StatesListFE0001[:, 2], label = "ForwardEuler␣
 ↪" * string(dt3))
```



0.0.5 Part 2e - Optimal Control

Refer to : https://jump.dev/JuMP.jl/stable/tutorials/nonlinear/space_shuttle_reentry_trajectory/

[ ]:
```julia
# Objective: Maximize cross-range
@objective(model, Max,  [n])

set_silent(model)  # Hide solver's verbose output
optimize!(model)  # Solve for the control and state
@assert is_solved_and_feasible(model)

# Show final cross-range of the solution
println(
    "Final latitude   = ",
    round(objective_value(model) |> rad2deg; digits = 2),
    "°",
)
```

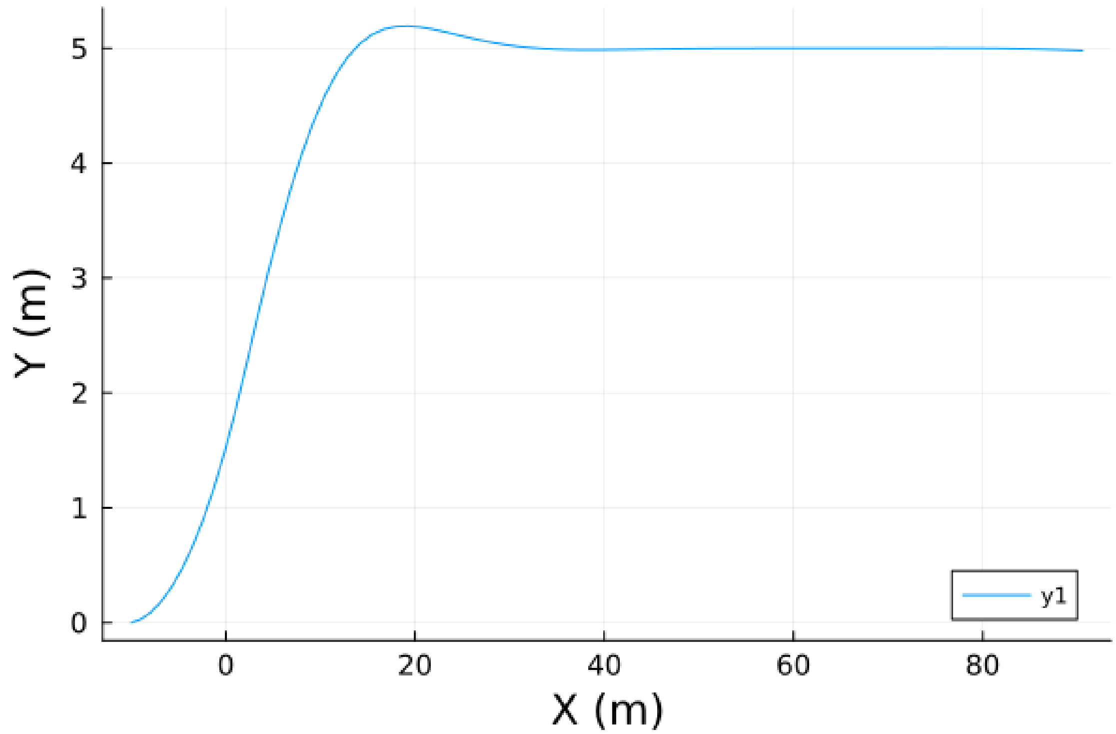
[ ]:
```julia
using JuMP
using Ipopt
using Plots
include("Q2c_VehicleDynamics.jl")

x0 = [-10.0, 0.0, 0.0, 0.0, 0.0, 10.0, 0.0] #TODO Initial Condition
XL = [-40, -20, -3, -pi/5,  -pi/2, 5.0, -pi/12] # States Lower Bound
XU = [300, 20, 3, pi/5, pi/2, 15.0, pi/12] #TODO States Upper Bound
CL = [-2.6, -0.1] #TODO Control Lower Bound
CU = [2.6, 0.1] #TODO Control Upper Bound

model = Model(optimizer_with_attributes(Ipopt.Optimizer)) # Initialize JuMP␣
 ↪model

numStates = 7 #TODO number of states
numControls = 2 #TODO number of control
PredictionHorizon = 8 #TODO Prediction Time
numColPoints = 81 #TODO
Δt = PredictionHorizon/(numColPoints - 1)# Time interval

@variables(model, begin
    # Set xst as a numColPoints x numStates matrix that is between the upper␣
 ↪and lower states bounds
    XL[i]   xst[j in 1:numColPoints, i in 1:numStates]    XU[i]
    #TODO Similarly, set u as a numColPoints x numControls matrix that is
    # between the upper and lower control bounds
    CL[i]   u[j in 1:numColPoints, i in 1:numControls]   CU[i]
end)

# Fix initial conditions
fix(xst[1, 1], x0[1]; force = true) # set the initial condition for x-position␣
 ↪value
#TODO Follow the same way, set the remaining initial conditions,
```

```
# set x0[2] to xst[1,2],... and so on.
fix(xst[1,2],x0[2];force = true)
fix(xst[1,3],x0[3];force = true)
fix(xst[1,4],x0[4];force= true)
fix(xst[1,5],x0[5];force= true)
fix(xst[1,6],x0[6];force= true)
fix(xst[1,7],x0[7];force= true)


# sa means steering angle, sr means steering rate
x = xst[:, 1]; y = xst[:, 2]; v = xst[:, 3]; r = xst[:, 4];   = xst[:, 5];
ux = xst[:, 6]; sa = xst[:, 7];
ax = u[:, 1]; # retract variable
sr = u[:, 2];

#  xst = Matrix{Any}(undef, numColPoints, numStates)
# write the states derivative for all states & controls
 xst = Matrix{Any}(undef, numColPoints, numStates)
for i = 1:1:numColPoints
    #  xst[i, :] = @expression(model, VehicleDynamics(xst[i, :], u[i, :]))
     xst[i, :] = @expression(model, VehicleDynamics(reshape(xst[i, :],(1,7)),
     reshape(u[i, :],(1,2))))
end

# add constraint to each state using backward Euler method
for j = 2:numColPoints
    for i = 1:numStates
        @constraint(model, xst[j, i] == xst[j - 1, i] +Δt *  xst[j, i])
    end
end


# TODO write the cost function for each term - Lane change
y_cost = @expression(model, sum((y[j] - 5)^2 * Δt for j= 1:1:numColPoints)) #global y position of C.G Cost
sr_cost = @expression(model, sum((sr[j])^2 * Δt for j= 1:1:numColPoints))
sa_cost = @expression(model, sum((sa[j])^2 * Δt for j= 1:1:numColPoints))
ux_cost = @expression(model, sum((ux[j] - 13)^2 * Δt for j= 1:1:numColPoints))
ax_cost = @expression( model, sum((ax[j])^2 * Δt for j=1:1:numColPoints)) # ax cost


#TODO define cost weight
w_y  = 0.05 # change later for 2f
w_sr = 2.0
w_ax = 0.2
w_ux = 0.2
```

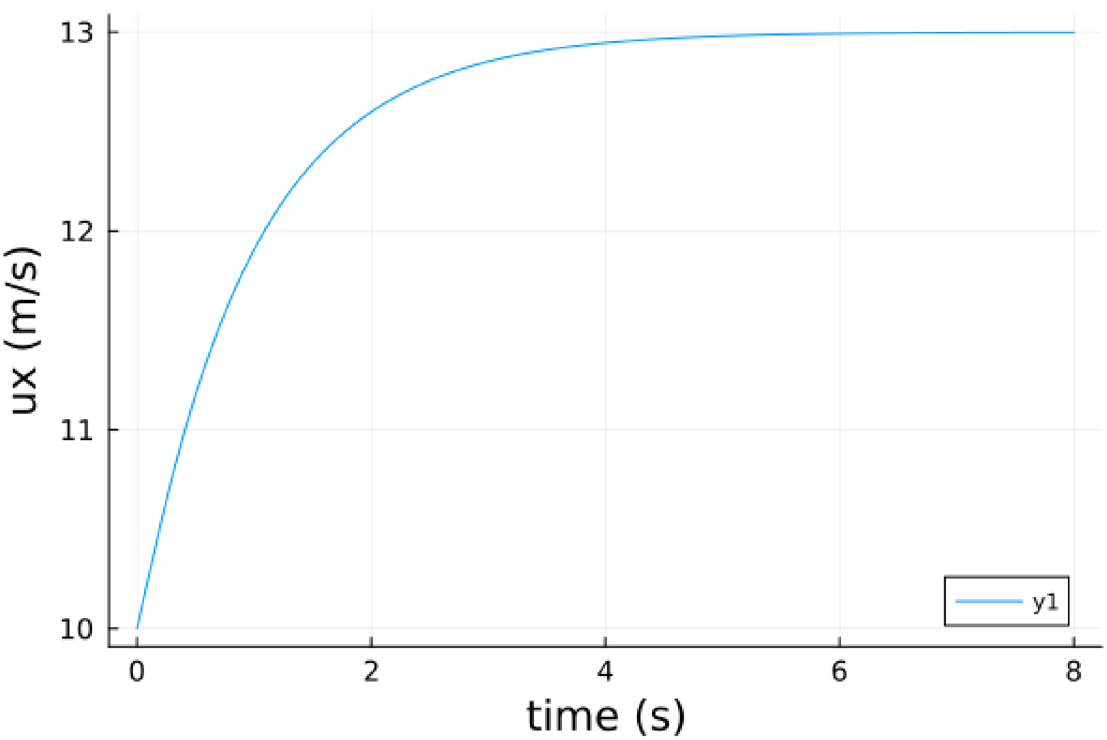
```
w_sa = 1.0

# Objective: Minimize cost function
@objective(model, Min, w_y * y_cost + w_sr * sr_cost + w_ax * ax_cost + w_ux *␣
 ↪ux_cost + w_sa * sa_cost) # objective value
optimize!(model) # optimize model
StatesHis = value.(model[:xst]) # retrieve data
if abs(objective_value(model) - 3.65) < 0.1 # check answer
    println("Congrats, your answer is correct")
else
    println("Something went wrong, please try again!")
end

println("Objective value model = ",objective_value(model))
println("Your y cost is:  ", round(value(y_cost); digits = 3))

#Plot
# plot(StatesHis[:, 1], StatesHis[:, 2], tickfontsize = 10, xlabel = "X (m)",␣
 ↪ylabel = "Y (m)",guidefont=15) # path plot
# plot(0:Δt:PredictionHorizon, StatesHis[:, 6], tickfontsize = 10, xlabel =␣
 ↪"time (s)", ylabel = "ux (m/s)",guidefont=15) # Speed plot

display(plot(StatesHis[:, 1], StatesHis[:, 2], tickfontsize = 10, xlabel = "X␣
 ↪(m)", ylabel = "Y (m)",guidefont=15)) # path plot
display(plot(0:Δt:PredictionHorizon, StatesHis[:, 6], tickfontsize = 10, xlabel␣
 ↪= "time (s)", ylabel = "ux (m/s)",guidefont=15)) # Speed plot)
```

```
The states derivative is: [9.9 1.496 -1.004 2.587 0.1 1.0 0.1]
This is Ipopt version 3.14.14, running with linear solver MUMPS 5.6.2.

Number of nonzeros in equality constraint Jacobian…:     2473
Number of nonzeros in inequality constraint Jacobian.:        0
Number of nonzeros in Lagrangian Hessian…:     3602

Total number of variables…:      722
                     variables with only lower bounds:        0
                variables with lower and upper bounds:      722
                     variables with only upper bounds:        0
Total number of equality constraints…:      560
Total number of inequality constraints…:        0
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  1.1142900e+02 9.50e+00 2.64e-01  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  1.1108982e+02 8.23e+00 6.83e+00  -1.0 3.13e+01    -  1.67e-02 1.33e-01f  1
```

```
   2  1.1024015e+02 6.92e+00 5.68e+00  -1.0 2.83e+01    -  6.67e-02 1.60e-01f  1
   3  1.0814632e+02 5.04e+00 4.04e+00  -1.0 2.49e+01    -  9.68e-02 2.71e-01f  1
   4  1.0460609e+02 3.07e+00 2.31e+00  -1.0 1.94e+01    -  1.38e-01 3.90e-01f  1
   5  9.8750701e+01 1.03e+00 1.23e+00  -1.0 1.29e+01    -  2.15e-01 6.66e-01f  1
   6  9.1032466e+01 3.43e-03 7.90e-01  -1.0 5.55e+00    -  4.76e-01 1.00e+00f  1
   7  7.6698955e+01 1.58e-03 8.00e-01  -1.0 4.29e+00    -  5.18e-01 1.00e+00f  1
   8  3.1555290e+01 1.12e-02 4.41e-01  -1.0 2.03e+01    -  4.57e-01 1.00e+00f  1
   9  1.0288882e+01 1.05e-02 1.09e-01  -1.0 1.49e+01    -  6.61e-01 1.00e+00f  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  10  4.9309175e+00 4.19e-03 2.62e-02  -1.0 7.88e+00    -  1.00e+00 1.00e+00f  1
  11  3.6599707e+00 2.29e-03 4.14e-02  -1.7 3.73e+00    -  1.00e+00 1.00e+00f  1
  12  3.2588409e+00 3.88e-03 2.72e-02  -2.5 1.92e+00    -  9.71e-01 1.00e+00f  1
  13  3.1493481e+00 5.07e-03 1.17e-02  -3.8 5.36e-01    -  8.85e-01 1.00e+00h  1
  14  3.1277009e+00 3.01e-03 1.78e-03  -3.8 1.91e-01    -  1.00e+00 1.00e+00h  1
  15  3.1242699e+00 7.74e-04 4.13e-04  -3.8 5.98e-02    -  1.00e+00 1.00e+00h  1
  16  3.1217627e+00 5.44e-04 4.68e-04  -5.7 4.73e-02    -  9.34e-01 1.00e+00h  1
  17  3.1214941e+00 2.08e-04 5.61e-05  -5.7 1.39e-02    -  1.00e+00 1.00e+00h  1
  18  3.1214421e+00 4.00e-05 5.14e-06  -5.7 5.09e-03    -  1.00e+00 1.00e+00h  1
  19  3.1214369e+00 2.89e-06 2.18e-07  -5.7 1.19e-03    -  1.00e+00 1.00e+00h  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  20  3.1214057e+00 2.33e-06 1.09e-06  -8.6 1.25e-03    -  9.97e-01 1.00e+00h  1
  21  3.1214054e+00 2.18e-08 1.44e-09  -8.6 9.73e-05    -  1.00e+00 1.00e+00h  1
  22  3.1214054e+00 7.13e-12 5.34e-13  -9.0 1.84e-06    -  1.00e+00 1.00e+00h  1

Number of Iterations…: 22

                                   (scaled)                 (unscaled)
Objective…:   3.1214054076720923e+00    3.1214054076720923e+00
Dual infeasibility…:   5.3362556413067902e-13    5.3362556413067902e-13
Constraint violation…:   7.1334049778215558e-12    7.1334049778215558e-12
Variable bound violation:   8.8477355164595650e-09    8.8477355164595650e-09
Complementarity…:   9.0957516544247136e-10    9.0957516544247136e-10
Overall NLP error…:   9.0957516544247136e-10    9.0957516544247136e-10


Number of objective function evaluations             = 23
Number of objective gradient evaluations             = 23
Number of equality constraint evaluations            = 23
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 23
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 22
Total seconds in IPOPT                               = 0.038

EXIT: Optimal Solution Found.
Something went wrong, please try again!
Objective value model = 3.1214054076720923
Your y cost is:  23.537
```



5
4
3
2
1
0
Y (m)
0
20
40
60
80
X (m)
y1



13
12
11
10
ux (m/s)
0
2
4
6
8
time (s)
y1

[ ]:

```
states = [-10.0 -5.0 0.5 0.1 0.1 10.0 0.1] # This is the initial state
ctrl = [1 0.1]  # One step control action

ux = StatesListFE02[:, 6]
r = StatesListFE02[:,4]
print(size(r))
print( 2 .- ux[6] * r[4])
```

```
(61,)0.08756238023141227
```