In [46]:
```julia
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
import FiniteDiff
import ForwardDiff as FD
import Convex as cvx
import ECOS
using LinearAlgebra
using Plots
using Random
using JLD2
using Test
using MeshCat
const mc = MeshCat
using StaticArrays
using Printf
```

```
  Activating environment at `~/Desktop/Work/Courses/OCRL/HW/HW4_S24/Project.t
oml`
```

In [47]:
```julia
include(joinpath(@__DIR__, "utils","ilc_visualizer.jl"))
```

Out[47]:
```
update_car_pose! (generic function with 1 method)
```

Q1: Iterative Learning Control (ILC) (40 pts)

In this problem, you will use ILC to generate a control trajectory for a Car as it swerves to avoid a moose, also known as "the moose test" ([wikipedia (https://en.wikipedia.org/wiki/Moose_test)](https://en.wikipedia.org/wiki/Moose_test), [video (https://www.youtube.com/watch?v=TZ2MYFInpMI)](https://www.youtube.com/watch?v=TZ2MYFInpMI)). We will model the dynamics of the car as with a simple nonlinear bicycle model, with the following state and control:

$$x = \begin{bmatrix} p_x \\ p_y \\ \theta \\ \delta \\ v \end{bmatrix}, \qquad u = \begin{bmatrix} a \\ \dot{\delta} \end{bmatrix}$$

where p_x and p_y describe the 2d position of the bike, θ is the orientation, δ is the steering angle, and v is the velocity. The controls for the bike are acceleration a, and steering angle rate $\dot{\delta}$.

In [48]:
```julia
function estimated_car_dynamics(model::NamedTuple, x::Vector, u::Vector)::Vector
    # nonlinear bicycle model continuous time dynamics
    px, py, θ, δ, v = x
    a, δdot = u

    β = atan(model.lr * δ, model.L)
    s,c = sincos(θ + β)
    ω = v*cos(β)*tan(δ) / model.L

    vx = v*c
    vy = v*s

    xdot = [
        vx,
        vy,
        ω,
        δdot,
        a
    ]

    return xdot
end
function rk4(model::NamedTuple, ode::Function, x::Vector, u::Vector, dt::Real)::Vector
    k1 = dt * ode(model, x,        u)
    k2 = dt * ode(model, x + k1/2, u)
    k3 = dt * ode(model, x + k2/2, u)
    k4 = dt * ode(model, x + k3,   u)
    return x + (1/6)*(k1 + 2*k2 + 2*k3 + k4)
end
```

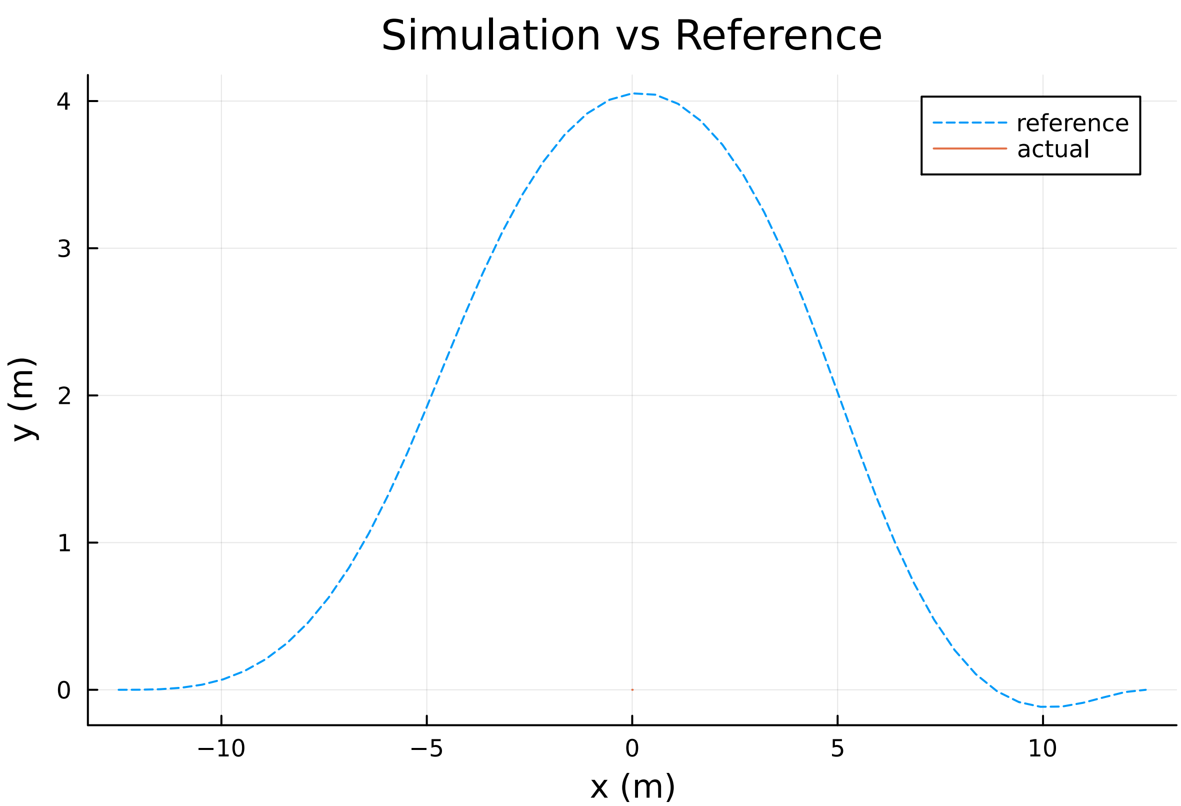
Out[48]:
```
rk4 (generic function with 1 method)
```

We have computed an optimal trajectory X_{ref} and U_{ref} for a moose test trajectory offline using this `estimated_car_dynamics` function. Unfortunately, this is a highly approximate dynamics model, and when we run U_{ref} on the car, we get a very different trajectory than we expect. This is caused by a significant sim to real gap. Here we will show what happens when we run these controls on the true dynamics:

In [49]:
```julia
function load_car_trajectory()
    # load in trajectory we computed offline
    path = joinpath(@__DIR__, "utils","init_control_car_ilc.jld2")
    F = jldopen(path)
    Xref = F["X"]
    Uref = F["U"]
    close(F)
    return Xref, Uref
end
function true_car_dynamics(model::NamedTuple, x::Vector, u::Vector)::Vector
    # true car dynamics
    px, py, θ, δ, v = x
    a, δdot = u

    # sluggish controls (not in the approximate version)
    a = 0.9*a - 0.1
    δdot = 0.9*δdot - .1*δ + .1

    β = atan(model.lr * δ, model.L)
    s,c = sincos(θ + β)
    ω = v*cos(β)*tan(δ) / model.L

    vx = v*c
    vy = v*s

    xdot = [
        vx,
        vy,
        ω,
        δdot,
        a
    ]

    return xdot
end

@testset "sim to real gap" begin
    # problem size
    nx = 5
    nu = 2
    dt = 0.1
    tf = 5.0
    t_vec = 0:dt:tf
    N = length(t_vec)
    model = (L = 2.8, lr = 1.6)

    # optimal trajectory computed offline with approximate model
    Xref, Uref = load_car_trajectory()

    # TODO: simulated Uref with the true car dynamics and store the states in
Xsim
    Xsim = [zeros(nx) for i = 1:N]


    # --------testing------------------
    @test norm(Xsim[1] - Xref[1]) == 0
```

```
    @test norm(Xsim[end] - [3.26801052, 15.0590156, 2.0482790, 0.39056168, 4.
5],Inf) < 1e-4

    # --------plotting/animation-------
    Xm= hcat(Xsim...)
    Xrefm = hcat(Xref...)
    plot(Xrefm[1,:], Xrefm[2,:], ls = :dash, label = "reference",
         xlabel = "x (m)", ylabel = "y (m)", title = "Simulation vs Referenc
e")
    display(plot!(Xm[1,:], Xm[2,:], label = "actual"))

end
```

```
sim to real gap: Test Failed at In[49]:55
  Expression: norm(Xsim[1] - Xref[1]) == 0
   Evaluated: 13.46291201783626 == 0
Stacktrace:
 [1] macro expansion
   @ In[49]:55 [inlined]
 [2] macro expansion
   @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/Te
st/src/Test.jl:1151 [inlined]
 [3] top-level scope
   @ In[49]:39
sim to real gap: Test Failed at In[49]:56
  Expression: norm(Xsim[end] - [3.26801052, 15.0590156, 2.048279, 0.39056168,
4.5], Inf) < 0.0001
   Evaluated: 15.0590156 < 0.0001
Stacktrace:
 [1] macro expansion
   @ In[49]:56 [inlined]
 [2] macro expansion
   @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/Te
st/src/Test.jl:1151 [inlined]
 [3] top-level scope
   @ In[49]:39
```




```
Test Summary:   | Fail  Total
sim to real gap |    2      2
```

```
Some tests did not pass: 0 passed, 2 failed, 0 errored, 0 broken.

Stacktrace:
 [1] finish(ts::Test.DefaultTestSet)
   @ Test /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/Test/src/Test.jl:913
 [2] macro expansion
   @ /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/Test/src/Test.jl:1161 [inlined]
 [3] top-level scope
   @ In[49]:39
```

In order to account for this, we are going to use ILC to iteratively correct our control until we converge.

To encourage the trajectory of the bike to follow the reference, the objective value for this problem is the following:

$$J(X,U) = \sum_{i=1}^{N-1}\left[\frac{1}{2}(x_i - x_{ref,i})^T Q(x_i - x_{ref,i}) + \frac{1}{2}(u_i - u_{ref,i})^T R(u_i - u_{ref,i})\right] + \frac{1}{2}(x_N - x_{ref,N})^T$$

Using ILC as described in [Lecture 18 (https://github.com/Optimal-Control-16-745/lecture-notebooks/blob/main/Lecture%2018/Lecture%2018.pdf)](https://github.com/Optimal-Control-16-745/lecture-notebooks/blob/main/Lecture%2018/Lecture%2018.pdf), we are to linearize our approximate dynamics model about X_{ref} and U_{ref} to get the following Jacobians:

$$A_k = \frac{\partial f}{\partial x}\bigg|_{x_{ref,k},u_{ref,k}}, \qquad B_k = \frac{\partial f}{\partial u}\bigg|_{x_{ref,k},u_{ref,k}}$$

where $f(x,u)$ is our **approximate discrete** dynamics model (`estimated_car_dynamics + rk4`). **You will form these Jacobians exactly once, using `Xref` and `Uref` .** Here is a summary of the notation:

- X_{ref} (`Xref`) - Optimal trajectory computed offline with approximate dynamics model.
- U_{ref} (`Uref`) - Optimal controls computed offline with approximate dynamics model.
- X_{sim} (`Xsim`) - Simulated trajectory with real dynamics model.
- $\bar{U}$ (`Ubar`) - Control we use for simulation with real dynamics model (this is what ILC updates).

In the second step of ILC, we solve the following optimization problem:

$$\begin{aligned}
\min_{\Delta x_{1:N},\Delta u_{1:N-1}} \quad & J(X_{sim} + \Delta X, \bar{U} + \Delta U) \\
\text{st} \quad & \Delta x_1 = 0 \\
& \Delta x_{k+1} = A_k \Delta x_k + B_k \Delta u_k \quad \text{for } k = 1,2,\ldots,N-1
\end{aligned}$$

We are going to initialize our $\bar{U}$ with U_{ref}, then the ILC algorithm will update $\bar{U} = \bar{U} + \Delta U$ at each iteration. It should only take 5-10 iterations to converge down to $\|\Delta U\| < 1 \cdot 10^{-2}$. You do not need to do any sort of linesearch between ILC updates.

In [ ]:

```julia
# feel free to use/not use any of these

# function trajectory_cost(Xsim::Vector{Vector{Float64}}, # simulated states
#                          Ubar::Vector{Vector{Float64}}, # simulated controls (ILC iterates this)
#                          Xref::Vector{Vector{Float64}}, # reference X's we want to track
#                          Uref::Vector{Vector{Float64}}, # reference U's we want to track
#                          Q::Matrix,                     # LQR tracking cost term
#                          R::Matrix,                     # LQR tracking cost term
#                          Qf::Matrix                     # LQR tracking cost term
#                          )::Float64                     # return cost J

#     J = 0
#     J += 0.5 * cvx.quadform(Xsim[end] - Xref[end], Qf)
#     # TODO: return trajectory cost J(Xsim, Ubar)
#     for i = 1:length(Xsim)-1
#         J += 0.5*cvx.quadform(Xsim[i] - Xref[i], Q) + 0.5*cvx.quadform(Ubar[i] - Uref[i], R)
#     end
#     return J
# end

function vec_from_mat(Xm::Matrix)::Vector{Vector{Float64}}
    # convert a matrix into a vector of vectors
    X = [Xm[:,i] for i = 1:size(Xm,2)]
    return X
end

function ilc_update(Xsim::Vector{Vector{Float64}}, # simulated states
                    Ubar::Vector{Vector{Float64}}, # simulated controls (ILC iterates this)
                    Xref::Vector{Vector{Float64}}, # reference X's we want to track
                    Uref::Vector{Vector{Float64}}, # reference U's we want to track
                    As::Vector{Matrix{Float64}},   # vector of A jacobians at each time step
                    Bs::Vector{Matrix{Float64}},   # vector of B jacobians at each time step
                    Q::Matrix,                     # LQR tracking cost term
                    R::Matrix,                     # LQR tracking cost term
                    Qf::Matrix                     # LQR tracking cost term
                    )::Vector{Vector{Float64}}     # return vector of ΔU's

    # solve optimization problem for ILC update
    N = length(Xsim)
    nx,nu = size(Bs[1])

    # create variables
    ΔX = cvx.Variable(nx, N)
    ΔU = cvx.Variable(nu, N-1)
```

```julia
    # TODO: cost function (tracking cost on Xref, Uref)
    cost = 0.5*cvx.quadform(ΔX[:,end] + Xsim[end] - Xref[end], Qf)
    for i = 1:N-1
        cost += 0.5*cvx.quadform(ΔX[:,i] + Xsim[i] - Xref[i], Q) + 0.5*cvx.qua
dform(ΔU[:,i] + Ubar[i] - Uref[i], R)
    end

    # problem instance
    prob = cvx.minimize(cost)

    # TODO: initial condition constraint
    prob.constraints += (ΔX[:,1] == zeros(nx))
    # TODO: dynamics constraints
    for i = 1:N-1
        prob.constraints += (ΔX[:,i+1] == As[i]*ΔX[:,i] + Bs[i]*ΔU[:,i])
    end

    cvx.solve!(prob, ECOS.Optimizer; silent_solver = true)

    # return ΔU
    ΔU = vec_from_mat(ΔU.value)

    return ΔU
end
```

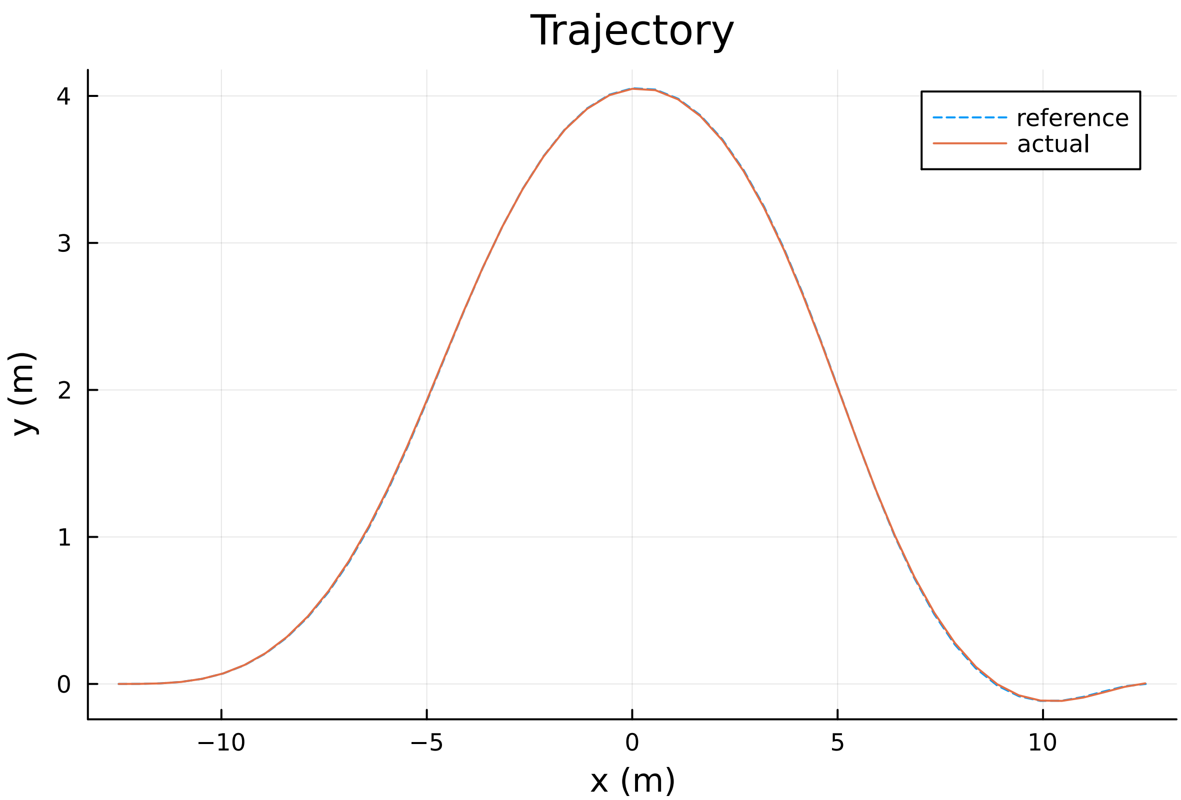
Out[ ]:
```
ilc_update (generic function with 1 method)
```

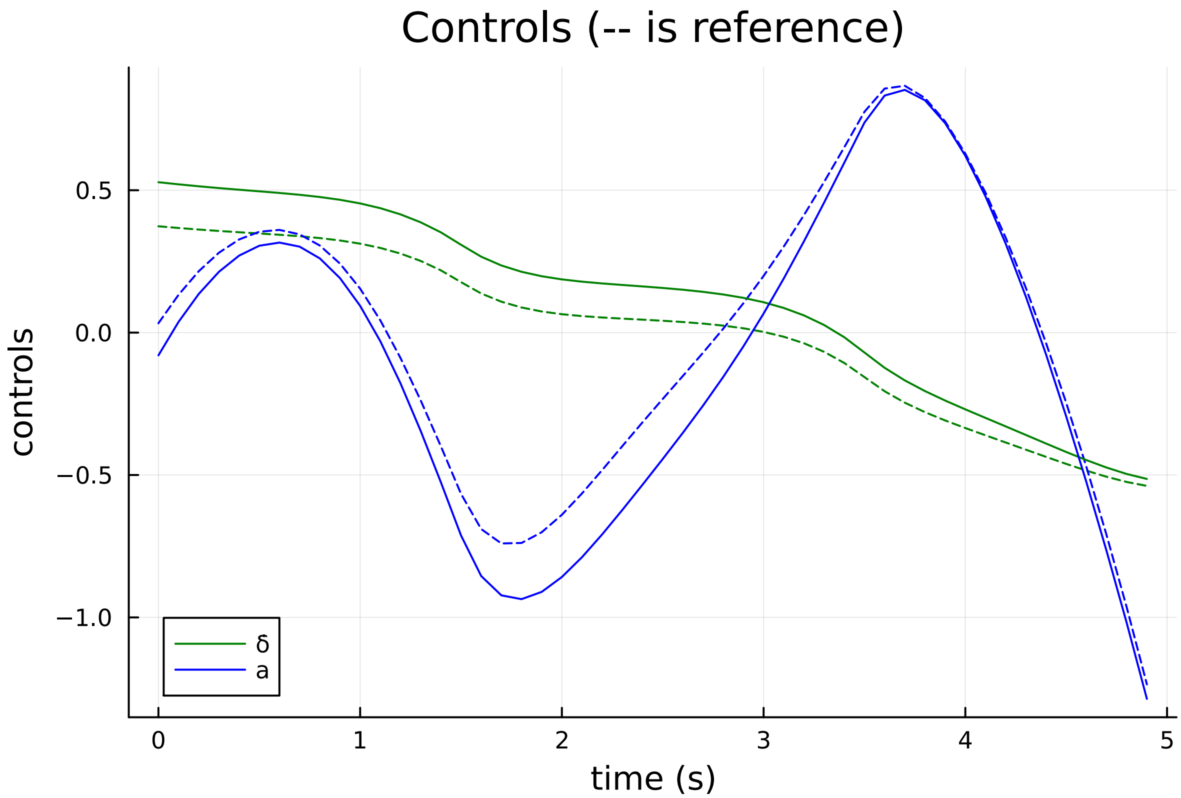
Here you will run your ILC algorithm. The resulting plots should show the simulated trajectory `Xsim` tracks `Xref` very closely, but there should be a significant difference between `Uref` and `Ubar`.

In [50]:
```
@testset "ILC" begin

    # problem size
    nx = 5
    nu = 2
    dt = 0.1
    tf = 5.0
    t_vec = 0:dt:tf
    N = length(t_vec)

    # optimal trajectory computed offline with approximate model
    Xref, Uref = load_car_trajectory()

    # initial and terminal conditions
    xic = Xref[1]
    xg = Xref[N]

    # LQR tracking cost to be used in ILC
    Q = diagm([1,1,.1,.1,.1])
    R = .1*diagm(ones(nu))
    Qf = 1*diagm(ones(nx))

    # load all useful things into params
    model = (L = 2.8, lr = 1.6)

    params = (Q = Q, R = R, Qf = Qf,xic = xic, xg = xg, Xref=Xref,Uref=Uref,
          dt = dt,
          N = N,
          model = model)


    # this holds the sim trajectory (with real dynamics)
    Xsim = [zeros(nx) for i = 1:N]

    # this is the feedforward control ILC is updating
    Ubar = [zeros(nu) for i = 1:(N-1)]
    Ubar .= Uref # initialize Ubar with Uref

    # TODO: calculate Jacobians
    A = [FD.jacobian(x -> rk4(model, estimated_car_dynamics, x, Uref[i], dt),
Xref[i]) for i = 1:N-1]
    B = [FD.jacobian(u -> rk4(model, estimated_car_dynamics, Xref[i], u, dt),
Uref[i]) for i = 1:N-1]

    # logging stuff
    @printf "iter     objv        |ΔU|     \n"
    @printf "------------------------------\n"

    for ilc_iter = 1:10 # it should not take more than 10 iterations to conver
ge
      Xsim[1] = xic
      # TODO: rollout
      for i = 1:N-1
          Xsim[i+1] = rk4(model, true_car_dynamics, Xsim[i], Ubar[i], dt)
      end
      # TODO: calculate objective val (trajectory_cost)
```

```
        obj_val = 0.5*Xsim[end]'*Qf*Xsim[end]
        for i = 1:N-1
            obj_val += 0.5*(Xsim[i] - Xref[i])'*Q*(Xsim[i] - Xref[i]) + 0.5*(Uba
r[i] - Uref[i])'*R*(Ubar[i] - Uref[i])
        end

        # solve optimization problem for update (ilc_update)
        ΔU = ilc_update(Xsim, Ubar, Xref, Uref, A, B, Q, R, Qf)

        # TODO: update the control
        Ubar = Ubar .+ ΔU

        # logging
        @printf("%3d   %10.3e  %10.3e  \n", ilc_iter, obj_val, sum(norm.(ΔU)))

    end

    # --------plotting/animation-------
    Xm= hcat(Xsim...)
    Um = hcat(Ubar...)
    Xrefm = hcat(Xref...)
    Urefm = hcat(Uref...)
    plot(Xrefm[1,:], Xrefm[2,:], ls = :dash, label = "reference",
         xlabel = "x (m)", ylabel = "y (m)", title = "Trajectory")
    display(plot!(Xm[1,:], Xm[2,:], label = "actual"))

    plot(t_vec[1:end-1], Urefm', ls = :dash, lc = [:green :blue],label = "",
         xlabel = "time (s)", ylabel = "controls", title = "Controls (-- is re
ference)")
    display(plot!(t_vec[1:end-1], Um', label = ["δ̇" "a"], lc = [:green :blu
e]))

    # animation
    vis = Visualizer()
    vis_traj!(vis, :traj, [[x[1],x[2],0.1] for x in Xsim]; R = 0.02)
    build_car!(vis[:car])
    anim = mc.Animation(floor(Int,1/dt))
    for k = 1:N
        mc.atframe(anim, k) do
            update_car_pose!(vis[:car], Xsim[k])
        end
    end
    mc.setanimation!(vis, anim)
    display(render(vis))

    # ----------testing-------------
    @test 0.1 <= sum(norm.(Xsim - Xref)) <= 1.0 # should be ~0.7
    @test 5 <=  sum(norm.(Ubar - Uref)) <= 10 # should be ~7.7

end
```

```
iter     objv          |ΔU|
------------------------------
  1    1.409e+03   6.307e+01
  2    1.193e+03   4.498e+01
  3    7.406e+02   9.266e+01
  4    1.082e+02   1.394e+01
  5    9.529e+01   1.959e+00
  6    9.094e+01   1.679e-01
  7    9.050e+01   1.649e-02
  8    9.045e+01   1.578e-03
  9    9.044e+01   1.911e-04
 10    9.044e+01   3.029e-05
```






```
[ Info: Listening on: 127.0.0.1:8700, thread id: 1
┌ Info: MeshCat server started. You can open the visualizer by visiting the f
ollowing URL in your browser:
└ http://127.0.0.1:8700
```

```
Test Summary: | Pass  Total
ILC           |    2      2
```

Out[50]:
```
Test.DefaultTestSet("ILC", Any[], 2, false, false)
```

In [1]:
```julia
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
import MathOptInterface as MOI
import Ipopt
import FiniteDiff
import ForwardDiff as FD
import Convex as cvx
import ECOS
using LinearAlgebra
using Plots
using Random
using JLD2
using Test
using MeshCat
const mc = MeshCat
using StaticArrays
using Printf
```

```
  Activating environment at `/home/sman/Work/CMU/Courses/OCRL/OCRL2024/HW/HW4_S24/Project.toml`
```

Julia note:

incorrect:

`x_l[idx.x[i]][2] = 0 # this does not change x_l`

correct:

`x_l[idx.x[i][2]] = 0 # this changes x_l`

It should always be `v[index] = new_val` if I want to update v with `new_val` at `index`.

In [2]:
```julia
let

    # vector we want to modify
    Z = randn(5)

    # original value of Z so we can check if we are changing it
    Z_original = 1 * Z

    # index range we are considering
    idx_x = 1:3

    # this does NOT change Z
    Z[idx_x][2] = 0

    # we can prove this
    @show norm(Z - Z_original)

    # this DOES change Z
    Z[idx_x[2]] = 0

    # we can prove this
    @show norm(Z - Z_original)


end
```

```
norm(Z - Z_original) = 0.0
norm(Z - Z_original) = 1.3770596629221186

1.3770596629221186
```

In [3]:
```julia
include(joinpath(@__DIR__, "utils","fmincon.jl"))
include(joinpath(@__DIR__, "utils","walker.jl"))
```

```
update_walker_pose! (generic function with 1 method)
```

(If nothing loads here, check out `walker.gif` in the repo)

NOTE: This question will have long outputs for each cell, remember you can use `cell -> all output -> toggle scrolling` to better see it all

Q2: Hybrid Trajectory Optimization (60 pts)

In this problem you'll use a direct method to optimize a walking trajectory for a simple biped model, using the hybrid dynamics formulation. You'll pre-specify a gait sequence and solve the problem using Ipopt. Your final solution should look like the video above.

The Dynamics

Our system is modeled as three point masses: one for the body and one for each foot. The state is defined as the x and y positions and velocities of these masses, for a total of 6 degrees of freedom and 12 states. We will label the position and velocity of each body with the following notation:

$$
\begin{aligned}
r^{(b)} &= \begin{bmatrix} p_x^{(b)} \\ p_y^{(b)} \end{bmatrix} \qquad & v^{(b)} &= \begin{bmatrix} v_x^{(b)} \\ v_y^{(b)} \end{bmatrix} \\
r^{(1)} &= \begin{bmatrix} p_x^{(1)} \\ p_y^{(1)} \end{bmatrix} \qquad & v^{(1)} &= \begin{bmatrix} v_x^{(1)} \\ v_y^{(1)} \end{bmatrix} \\
r^{(2)} &= \begin{bmatrix} p_x^{(2)} \\ p_y^{(2)} \end{bmatrix} \qquad & v^{(2)} &= \begin{bmatrix} v_x^{(2)} \\ v_y^{(2)} \end{bmatrix}
\end{aligned}
$$

Each leg is connected to the body with prismatic joints. The system has three control inputs: a force along each leg, and the torque between the legs.

The state and control vectors are ordered as follows:

$$
x = \begin{bmatrix} p_x^{(b)} \\ p_y^{(b)} \\ p_x^{(1)} \\ p_y^{(1)} \\ p_x^{(2)} \\ p_y^{(2)} \\ v_x^{(b)} \\ v_y^{(b)} \\ v_x^{(1)} \\ v_y^{(1)} \\ v_x^{(2)} \\ v_y^{(2)} \end{bmatrix} \qquad u = \begin{bmatrix} F^{(1)} \\ F^{(2)} \\ \tau \end{bmatrix}
$$

where e.g. $p_x^{(b)}$ is the x position of the body, $v_y^{(i)}$ is the y velocity of foot i, $F^{(i)}$ is the force along leg i, and τ is the torque between the legs.

The continuous time dynamics and jump maps for the two stances are shown below:

In [4]:
```julia
function stance1_dynamics(model::NamedTuple, x::Vector, u::Vector)
    # dynamics when foot 1 is in contact with the ground

    mb,mf = model.mb, model.mf
    g = model.g

    M = Diagonal([mb mb mf mf mf mf])

    rb  = x[1:2]   # position of the body
    rf1 = x[3:4]   # position of foot 1
    rf2 = x[5:6]   # position of foot 2
    v   = x[7:12]  # velocities


    ℓ1x = (rb[1]-rf1[1])/norm(rb-rf1)
    ℓ1y = (rb[2]-rf1[2])/norm(rb-rf1)
    ℓ2x = (rb[1]-rf2[1])/norm(rb-rf2)
    ℓ2y = (rb[2]-rf2[2])/norm(rb-rf2)

    B = [ℓ1x  ℓ2x  ℓ1y-ℓ2y;
         ℓ1y  ℓ2y  ℓ2x-ℓ1x;
          0    0      0;
          0    0      0;
          0   -ℓ2x  ℓ2y;
          0   -ℓ2y -ℓ2x]

    v̇ = [0; -g; 0; 0; 0; -g] + M\(B*u)

    ẋ = [v; v̇]

    return ẋ
end

function stance2_dynamics(model::NamedTuple, x::Vector, u::Vector)
    # dynamics when foot 2 is in contact with the ground

    mb,mf = model.mb, model.mf
    g = model.g
    M = Diagonal([mb mb mf mf mf mf])

    rb  = x[1:2]   # position of the body
    rf1 = x[3:4]   # position of foot 1
    rf2 = x[5:6]   # position of foot 2
    v   = x[7:12]  # velocities

    ℓ1x = (rb[1]-rf1[1])/norm(rb-rf1)
    ℓ1y = (rb[2]-rf1[2])/norm(rb-rf1)
    ℓ2x = (rb[1]-rf2[1])/norm(rb-rf2)
    ℓ2y = (rb[2]-rf2[2])/norm(rb-rf2)

    B = [ℓ1x  ℓ2x  ℓ1y-ℓ2y;
         ℓ1y  ℓ2y  ℓ2x-ℓ1x;
        -ℓ1x   0  -ℓ1y;
        -ℓ1y   0   ℓ1x;
          0    0    0;
          0    0    0]
```

```julia
    v̇ = [0; -g; 0; -g; 0; 0] + M\(B*u)

    ẋ = [v; v̇]

    return ẋ
end

function jump1_map(x)
    # foot 1 experiences inelastic collision
    xn = [x[1:8]; 0.0; 0.0; x[11:12]]
    return xn
end

function jump2_map(x)
    # foot 2 experiences inelastic collision
    xn = [x[1:10]; 0.0; 0.0]
    return xn
end

function rk4(model::NamedTuple, ode::Function, x::Vector, u::Vector, dt::Real)::Vector
    k1 = dt * ode(model, x,        u)
    k2 = dt * ode(model, x + k1/2, u)
    k3 = dt * ode(model, x + k2/2, u)
    k4 = dt * ode(model, x + k3,   u)
    return x + (1/6)*(k1 + 2*k2 + 2*k3 + k4)
end
```

```
rk4 (generic function with 1 method)
```

We are setting up this problem by scheduling out the contact sequence. To do this, we will define the following sets:

$$\mathcal{M}_1 = \{1{:}5, 11{:}15, 21{:}25, 31{:}35, 41{:}45\}$$
$$\mathcal{M}_2 = \{6{:}10, 16{:}20, 26{:}30, 36{:}40\}$$

where $\mathcal{M}_1$ contains the time steps when foot 1 is pinned to the ground (`stance1_dynamics`), and $\mathcal{M}_2$ contains the time steps when foot 2 is pinned to the ground (`stance2_dynamics`). The jump map sets $\mathcal{J}_1$ and $\mathcal{J}_2$ are the indices where the mode of the next time step is different than the current, i.e. $\mathcal{J}_i \equiv \{k+1 \notin \mathcal{M}_i \mid k \in \mathcal{M}_i\}$. We can write these out explicitly as the following:

$$\mathcal{J}_1 = \{5, 15, 25, 35\}$$
$$\mathcal{J}_2 = \{10, 20, 30, 40\}$$

Another term you will see is set subtraction, or $\mathcal{M}_i \setminus \mathcal{J}_i$. This just means that if $k \in \mathcal{M}_i \setminus \mathcal{J}_i$, then k is in $\mathcal{M}_i$ but not in $\mathcal{J}_i$.

We will make use of the following Julia code for determining which set an index belongs to:

In [5]:
```julia
let
    M1 = vcat([ (i-1)*10      .+ (1:5)   for i = 1:5]...) # stack the set into a vector
    M2 = vcat([((i-1)*10 + 5) .+ (1:5)   for i = 1:4]...) # stack the set into a vector
    J1 = [5,15,25,35]
    J2 = [10,20,30,40]

    @show  (5 in M1) # show if 5 is in M1
    @show  (5 in J1) # show if 5 is in J1
    @show !(5 in M1) # show is 5 is not in M1

    @show (5 in M1) && !(5 in J1) # 5 in M1 but not J1 (5 ∈ M_1 \ J1)

end
```

```
5 in M1 = true
5 in J1 = true
!(5 in M1) = false
5 in M1 && !(5 in J1) = false
```

```
false
```

We are now going to setup and solve a constrained nonlinear program. The optimization problem looks complicated but each piece should make sense and be relatively straightforward to implement. First we have the following LQR cost function that will track x_{ref} (`Xref`) and u_{ref} (`Uref`):

$$J(x_{1:N}, u_{1:N-1}) = \sum_{i=1}^{N-1} \left[\frac{1}{2}(x_i - x_{ref,i})^T Q (x_i - x_{ref,i}) + \frac{1}{2}(u_i - u_{ref,i})^T R (u_i - u_{ref,i}) \right] + \frac{1}{2}(x_N - x$$

Which goes into the following full optimization problem:

$$\begin{aligned}
\min_{x_{1:N}, u_{1:N-1}} \quad & J(x_{1:N}, u_{1:N-1}) & & \\
\text{st} \quad & x_1 = x_{ic} & & (1)\\
& x_N = x_g & & (2)\\
& x_{k+1} = f_1(x_k, u_k) & \text{for } k \in \mathcal{M}_1 \setminus \mathcal{J}_1 & \quad (3)\\
& x_{k+1} = f_2(x_k, u_k) & \text{for } k \in \mathcal{M}_2 \setminus \mathcal{J}_2 & \quad (4)\\
& x_{k+1} = g_2(f_1(x_k, u_k)) & \text{for } k \in \mathcal{J}_1 & \quad (5)\\
& x_{k+1} = g_1(f_2(x_k, u_k)) & \text{for } k \in \mathcal{J}_2 & \quad (6)\\
& x_k[4] = 0 & \text{for } k \in \mathcal{M}_1 & \quad (7)\\
& x_k[6] = 0 & \text{for } k \in \mathcal{M}_2 & \quad (8)\\
& 0.5 \leq \|r_k^{(b)} - r_k^{(1)}\|_2 \leq 1.5 & \text{for } k \in [1, N] & \quad (9)\\
& 0.5 \leq \|r_k^{(b)} - r_k^{(2)}\|_2 \leq 1.5 & \text{for } k \in [1, N] & \quad (10)\\
& x_k[2, 4, 6] \geq 0 & \text{for } k \in [1, N] & \quad (11)
\end{aligned}$$

Each constraint is now described, with the type of constraint for `fmincon` in parantheses:

1. Initial condition constraint **(equality constraint)**.
2. Terminal condition constraint **(equality constraint)**.
3. Stance 1 discrete dynamics **(equality constraint)**.
4. Stance 2 discrete dynamics **(equality constraint)**.
5. Discrete dynamics from stance 1 to stance 2 with jump 2 map **(equality constraint)**.
6. Discrete dynamics from stance 2 to stance 1 with jump 1 map **(equality constraint)**.
7. Make sure the foot 1 is pinned to the ground in stance 1 **(equality constraint)**.
8. Make sure the foot 2 is pinned to the ground in stance 2 **(equality constraint)**.
9. Length constraints between main body and foot 1 **(inequality constraint)**.
10. Length constraints between main body and foot 2 **(inequality constraint)**.
11. Keep the y position of all 3 bodies above ground **(primal bound)**.

And here we have the list of mathematical functions to the Julia function names:

- f_1 is `stance1_dynamics + rk4`
- f_2 is `stance2_dynamics + rk4`
- g_1 is `jump1_map`
- g_2 is `jump2_map`

For instance, $g_2(f_1(x_k, u_k))$ is `jump2_map(rk4(model, stance1_dynamics, xk, uk, dt))`

Remember that $r^{(b)}$ is defined above.

In [6]:
```
function reference_trajectory(model, xic, xg, dt, N)
    # creates a reference Xref and Uref for walker

    Uref = [[model.mb*model.g*0.5;model.mb*model.g*0.5;0] for i = 1:(N-1)]

    Xref = [zeros(12) for i = 1:N]

    horiz_v = (3/N)/dt
    xs = range(-1.5, 1.5, length = N)
    Xref[1] = 1*xic
    Xref[N] = 1*xg

    for i = 2:(N-1)
        Xref[i] = [xs[i],1,xs[i],0,xs[i],0,horiz_v,0,horiz_v,0,horiz_v,0]
    end

    return Xref, Uref
end
```

```
reference_trajectory (generic function with 1 method)
```

To solve this problem with Ipopt and `fmincon`, we are going to concatenate all of our x's and u's into one vector (same as HW3Q1):

$$Z = \begin{bmatrix} x_1 \\ u_1 \\ x_2 \\ u_2 \\ \vdots \\ x_{N-1} \\ u_{N-1} \\ x_N \end{bmatrix} \in \mathbb{R}^{N \cdot nx + (N-1) \cdot nu}$$

where $x \in \mathbb{R}^{nx}$ and $u \in \mathbb{R}^{nu}$. Below we will provide useful indexing guide in `create_idx` to help you deal with Z. Remember that the API for `fmincon` (that we used in HW3Q1) is the following:

$$\begin{aligned} \min_z \quad & \ell(z) && \text{cost function} \\ \text{st} \quad & c_{eq}(z) = 0 && \text{equality constraint} \\ & c_L \leq c_{ineq}(z) \leq c_U && \text{inequality constraint} \\ & z_L \leq z \leq z_U && \text{primal bound constraint} \end{aligned}$$

Template code has been given to solve this problem but you should feel free to do whatever is easiest for you, as long as you get the trajectory shown in the animation `walker.gif` and pass tests.

In [10]:
```julia
# feel free to solve this problem however you like, below is a template for a
# good way to start.

function create_idx(nx,nu,N)
    # create idx for indexing convenience
    # x_i = Z[idx.x[i]]
    # u_i = Z[idx.u[i]]
    # and stacked dynamics constraints of size nx are
    # c[idx.c[i]] = <dynamics constraint at time step i>
    #
    # feel free to use/not use this

    # our Z vector is [x0, u0, x1, u1, …, xN]
    nz = (N-1) * nu + N * nx # length of Z
    x = [(i - 1) * (nx + nu) .+ (1 : nx) for i = 1:N]
    u = [(i - 1) * (nx + nu) .+ ((nx + 1):(nx + nu)) for i = 1:(N - 1)]

    # constraint indexing for the (N-1) dynamics constraints when stacked up
    c = [(i - 1) * (nx) .+ (1 : nx) for i = 1:(N - 1)]
    nc = (N - 1) * nx # (N-1)*nx

    return (nx=nx,nu=nu,N=N,nz=nz,nc=nc,x= x,u = u,c = c)
end

function walker_cost(params::NamedTuple, Z::Vector)::Real
    # cost function
    idx, N, xg = params.idx, params.N, params.xg
    Q, R, Qf = params.Q, params.R, params.Qf
    Xref,Uref = params.Xref, params.Uref

    # TODO: input walker LQR cost
    J = 0
    for i = 1:(N-1)
        xi = Z[idx.x[i]]
        ui = Z[idx.u[i]]
        J += 0.5*(xi - Xref[i])'*Q*(xi - Xref[i]) + 0.5*(ui - Uref[i])'*R*(ui
- Uref[i])
    end
    xn = Z[idx.x[N]]
    J += 0.5*(xn - Xref[N])'*Qf*(xn - Xref[N])
    return J
end

function walker_dynamics_constraints(params::NamedTuple, Z::Vector)::Vector
    idx, N, dt = params.idx, params.N, params.dt
    M1, M2 = params.M1, params.M2
    J1, J2 = params.J1, params.J2
    model = params.model

    # create c in a ForwardDiff friendly way (check HW0)
    c = zeros(eltype(Z), idx.nc)

    # TODO: input walker dynamics constraints (constraints 3-6 in the opti pro
blem)
    for i = 1:(N-1)
        xi = Z[idx.x[i]]
```

```
        ui = Z[idx.u[i]]
        xi1 = Z[idx.x[i+1]]
        if (i in M1) && !(i in J1)
            c[idx.c[i]] = rk4(model, stance1_dynamics, xi, ui, dt) - xi1
        elseif (i in M2) && !(i in J2)
            c[idx.c[i]] = rk4(model, stance2_dynamics, xi, ui, dt) - xi1
        elseif i in J1
            c[idx.c[i]] = jump2_map(rk4(model, stance1_dynamics, xi, ui, dt))
- xi1
        elseif i in J2
            c[idx.c[i]] = jump1_map(rk4(model, stance2_dynamics, xi, ui, dt))
- xi1
        end
    end

    return c
end

function walker_stance_constraint(params::NamedTuple, Z::Vector)::Vector
    idx, N, dt = params.idx, params.N, params.dt
    M1, M2 = params.M1, params.M2
    J1, J2 = params.J1, params.J2

    model = params.model

    # create c in a ForwardDiff friendly way (check HW0)
    c = zeros(eltype(Z), N)

    # TODO: add walker stance constraints (constraints 7-8 in the opti proble
m)

    for i = 1:N
        xi = Z[idx.x[i]]
        if i in M1
            c[i] = xi[4]
        elseif i in M2
            c[i] = xi[6]
        end
    end

    return c
end

function walker_equality_constraint(params::NamedTuple, Z::Vector)::Vector
    N, idx, xic, xg = params.N, params.idx, params.xic, params.xg

    # TODO: stack up all of our equality constraints

    # should be length 2*nx + (N-1)*nx + N
    # inital condition constraint (nx)       (constraint 1)
    # terminal constraint         (nx)       (constraint 2)
    # dynamics constraints        (N-1)*nx   (constraint 3-6)
    # stance constraint           N          (constraint 7-8)
    c = Vector{eltype(Z)}()

    cic = Z[idx.x[1]] - xic
    append!(c, cic)
```

```
    cfc = Z[idx.x[N]] - xg
    append!(c, cfc)
    cdc = walker_dynamics_constraints(params, Z)
    append!(c, cdc)
    csc = walker_stance_constraint(params, Z)
    append!(c, csc)

    return c
end

function walker_inequality_constraint(params::NamedTuple, Z::Vector)::Vector
    idx, N, dt = params.idx, params.N, params.dt
    M1, M2 = params.M1, params.M2

    # create c in a ForwardDiff friendly way (check HW0)
    c = zeros(eltype(Z), 2*N)

    # TODO: add the length constraints shown in constraints (9-10)
    # there are 2*N constraints here

    for i = 1:N
        xi = Z[idx.x[i]]

        rb = xi[1:2]
        rf1 = xi[3:4]
        rf2 = xi[5:6]

        c[2*i-1] = norm(rb-rf1)^2
        c[2*i] = norm(rb-rf2)^2
    end

    return c
end
```

```
walker_inequality_constraint (generic function with 1 method)
```

In [13]:
```
@testset "walker trajectory optimization" begin

    # dynamics parameters
    model = (g = 9.81, mb= 5.0, mf = 1.0, ℓ_min = 0.5, ℓ_max = 1.5)

    # problem size
    nx = 12
    nu = 3
    tf = 4.4
    dt = 0.1
    t_vec = 0:dt:tf
    N = length(t_vec)

    # initial and goal states
    xic = [-1.5;1;-1.5;0;-1.5;0;0;0;0;0;0;0]
    xg =  [1.5;1;1.5;0;1.5;0;0;0;0;0;0;0]

    # index sets
    M1 = vcat([ (i-1)*10      .+ (1:5)   for i = 1:5]...)
    M2 = vcat([((i-1)*10 + 5) .+ (1:5)   for i = 1:4]...)
    J1 = [5,15,25,35]
    J2 = [10,20,30,40]

    # reference trajectory
    Xref, Uref = reference_trajectory(model, xic, xg, dt, N)

    # LQR cost function (tracking Xref, Uref)
    Q = diagm([1; 10; fill(1.0, 4); 1; 10; fill(1.0, 4)]);
    R = diagm(fill(1e-3,3))
    Qf = 1*Q;

    # create indexing utilities
    idx = create_idx(nx,nu,N)

    # put everything useful in params
    params = (
        model = model,
        nx = nx,
        nu = nu,
        tf = tf,
        dt = dt,
        t_vec = t_vec,
        N = N,
        M1 = M1,
        M2 = M2,
        J1 = J1,
        J2 = J2,
        xic = xic,
        xg = xg,
        idx = idx,
        Q = Q, R = R, Qf = Qf,
        Xref = Xref,
        Uref = Uref
    )

    # TODO: primal bounds (constraint 11)
```

```
    x_l = -Inf*ones(idx.nz) # update this
    x_u = Inf*ones(idx.nz) # update this

    for i = 1:N
        x_l[idx.x[i][2]] = 0
        x_l[idx.x[i][4]] = 0
        x_l[idx.x[i][6]] = 0
    end


    # TODO: inequality constraint bounds
    c_l = (0.5^2)*ones(2*N) # update this
    c_u = (1.5^2)*ones(2*N) # update this

    # TODO: initialize z0 with the reference Xref, Uref
    z0 = zeros(idx.nz) # update this
    for i = 1:(N-1)
        z0[idx.x[i]] = Xref[i]
        z0[idx.u[i]] = Uref[i]
    end
    z0[idx.x[N]] = Xref[N]

    # adding a little noise to the initial guess is a good idea
    z0 = z0 + (1e-6)*randn(idx.nz)

    diff_type = :auto

    Z = fmincon(walker_cost,walker_equality_constraint,walker_inequality_const
raint,
                x_l,x_u,c_l,c_u,z0,params, diff_type;
                tol = 1e-6, c_tol = 1e-6, max_iters = 10_000, verbose = true)

    # pull the X and U solutions out of Z
    X = [Z[idx.x[i]] for i = 1:N]
    U = [Z[idx.u[i]] for i = 1:(N-1)]

    # ------------plotting--------------
    Xm = hcat(X...)
    Um = hcat(U...)

    plot(Xm[1,:],Xm[2,:], label = "body")
    plot!(Xm[3,:],Xm[4,:], label = "leg 1")
    display(plot!(Xm[5,:],Xm[6,:], label = "leg 2",xlabel = "x (m)",
                  ylabel = "y (m)", title = "Body Positions"))

    display(plot(t_vec[1:end-1], Um',xlabel = "time (s)", ylabel = "U",
                 label = ["F1" "F2" "τ"], title = "Controls"))

    # ---------animation-------------
    vis = Visualizer()
    build_walker!(vis, model::NamedTuple)
    anim = mc.Animation(floor(Int,1/dt))
    for k = 1:N
        mc.atframe(anim, k) do
            update_walker_pose!(vis, model::NamedTuple, X[k])
        end
    end
```

```julia
    mc.setanimation!(vis, anim)
    display(render(vis))

    # --------testing----------------

    # initial and terminal states
    @test norm(X[1] - xic,Inf) <= 1e-3
    @test norm(X[end] - xg,Inf) <= 1e-3

    for x in X

        # distance between bodies
        rb = x[1:2]
        rf1 = x[3:4]
        rf2 = x[5:6]
        @test (0.5 - 1e-3) <= norm(rb-rf1) <= (1.5 + 1e-3)
        @test (0.5 - 1e-3) <= norm(rb-rf2) <= (1.5 + 1e-3)

        # no two feet moving at once
        v1 = x[9:10]
        v2 = x[11:12]
        @test min(norm(v1,Inf),norm(v2,Inf)) <= 1e-3

        # check everything above the surface
        @test x[2] >= (0 - 1e-3)
        @test x[4] >= (0 - 1e-3)
        @test x[6] >= (0 - 1e-3)

    end

end
```

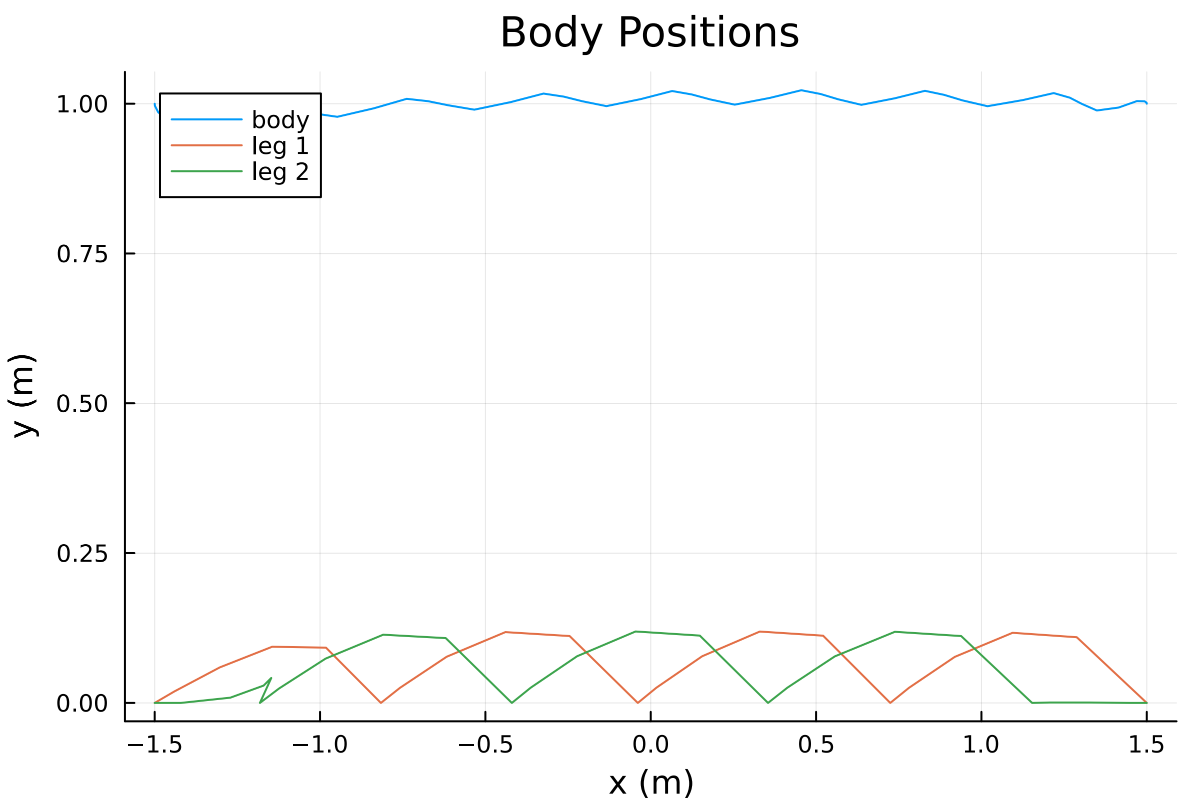
```
---------checking dimensions of everything----------
---------all dimensions good-----------------------
---------diff type set to :auto (ForwardDiff.jl)----
---------testing objective gradient-----------------
---------testing constraint Jacobian----------------
---------successfully compiled both derivatives-----
---------IPOPT beginning solve----------------------
5314e+02 1.83e+00 2.84e+01  -1.0 1.63e+00    -  1.00e+00 1.00e+00f  1
 161r 6.1961400e+02 1.83e+00 8.23e+00  -1.0 1.10e+00    -  1.00e+00 1.00e+00h
1
 162r 6.1951049e+02 1.83e+00 1.86e+00  -1.0 1.87e-01    -  1.00e+00 1.00e+00h
1
 163r 6.1975715e+02 1.83e+00 1.06e+01  -1.6 1.06e+00    -  8.53e-01 1.00e+00f
1
 164r 6.1469830e+02 1.82e+00 3.00e+01  -1.6 7.84e+01    -  1.23e-01 1.04e-01f
1
 165r 6.1456259e+02 1.83e+00 1.04e+02  -0.6 2.55e+01    -  2.36e-01 8.39e-02f
1
 166r 6.1445512e+02 1.83e+00 8.16e+01  -1.4 1.25e+01    -  1.64e-01 5.10e-02h
2
 167r 6.1388205e+02 1.83e+00 8.21e+01  -1.4 6.08e+00    -  3.18e-01 1.06e-01f
1
 168r 6.1348302e+02 1.83e+00 1.07e+02  -1.4 6.35e+00    -  6.55e-01 2.60e-01f
1
 169r 6.0818933e+02 1.83e+00 5.25e+01  -1.4 5.14e+00    -  6.92e-01 1.00e+00f
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
 170r 6.0846931e+02 1.83e+00 1.26e+02  -1.4 2.21e+00    -  1.00e+00 5.23e-01f
1
 171r 6.0967089e+02 1.82e+00 1.34e+01  -1.4 2.26e+00    -  1.00e+00 9.25e-01h
1
 172r 6.0885484e+02 1.83e+00 2.50e+01  -1.4 1.90e+00    -  1.00e+00 1.00e+00h
1
 173r 6.0937231e+02 1.83e+00 4.81e+00  -1.4 8.20e-01    -  1.00e+00 1.00e+00h
1
 174r 6.0929124e+02 1.83e+00 1.49e+01  -1.7 1.24e+00    -  9.95e-01 1.00e+00f
1
 175r 6.0514328e+02 1.83e+00 8.23e+01  -0.7 2.74e+01    -  5.08e-01 3.04e-01f
1
 176r 6.0663654e+02 1.83e+00 7.68e+01  -1.2 1.22e+01    -  1.07e-01 1.22e-01f
1
This is Ipopt version 3.14.4, running with linear solver MUMPS 5.4.1.

Number of nonzeros in equality constraint Jacobian...:   401184
Number of nonzeros in inequality constraint Jacobian.:    60480
Number of nonzeros in Lagrangian Hessian.............:        0

Total number of variables............................:      672
                     variables with only lower bounds:      135
                variables with lower and upper bounds:        0
                     variables with only upper bounds:        0
Total number of equality constraints.................:      597
Total number of inequality constraints...............:       90
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:       90
        inequality constraints with only upper bounds:        0
```

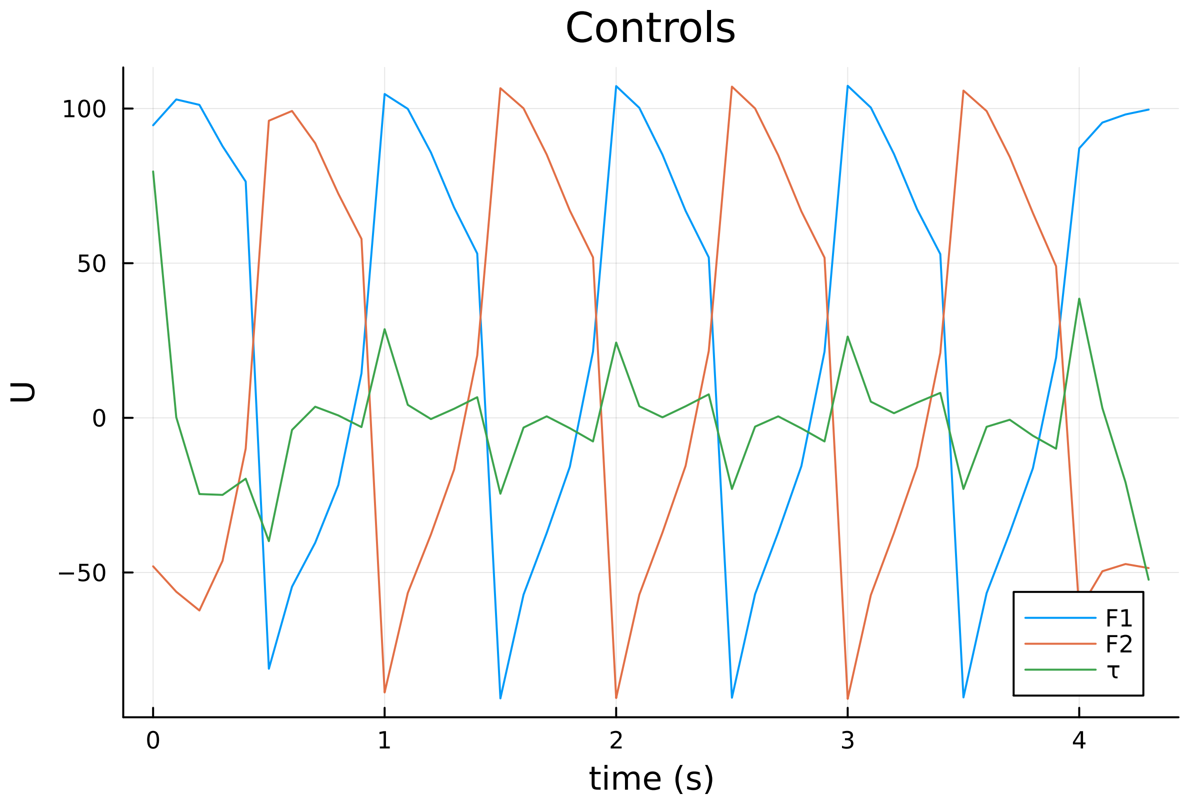
```
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
   0  4.4999916e-03 1.47e+00 1.00e+00   0.0 0.00e+00    -  0.00e+00 0.00e+00
0
   1  6.7558367e+01 1.06e+00 4.49e+03  -0.7 1.18e+02    -  4.10e-01 3.62e-01h
1
   2  2.5766535e+02 1.03e+00 5.53e+03   1.0 1.70e+02    -  9.95e-01 2.57e-01f
1
   3  4.4592238e+02 9.70e-01 1.83e+03   0.8 7.44e+01    -  7.98e-01 9.29e-01h
1
   4  4.5866581e+02 3.61e-01 1.03e+04   0.8 3.80e+01    -  2.01e-01 7.30e-01f
1
   5  4.3941631e+02 1.45e-01 9.60e+02   1.1 4.01e+01    -  9.81e-01 1.00e+00f
1
   6  3.7458943e+02 3.30e-02 2.24e+02   0.8 2.49e+01    -  1.00e+00 1.00e+00h
1
   7  3.1874607e+02 6.48e-02 1.49e+02   0.3 4.46e+01    -  9.80e-01 1.00e+00h
1
   8  2.9911782e+02 2.58e-02 1.17e+02   0.2 2.39e+01    -  9.82e-01 1.00e+00H
1
   9  2.7730885e+02 2.17e-03 2.49e+01  -0.1 1.04e+01    -  1.00e+00 1.00e+00H
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  10  2.6643551e+02 1.58e-04 2.41e+01  -0.6 1.32e+01    -  9.86e-01 1.00e+00H
1
  11  2.7404437e+02 1.83e-02 1.74e+01  -0.4 3.60e+01    -  1.00e+00 1.00e+00F
1
  12  2.9219967e+02 7.06e-03 1.21e+03  -0.5 2.11e+01    -  5.75e-01 1.00e+00H
1
  13  2.5270353e+02 5.06e-02 7.12e+00  -0.5 2.35e+01    -  1.00e+00 1.00e+00f
1
  14  2.5243773e+02 1.34e-02 1.61e+01  -1.2 8.92e+00    -  9.08e-01 1.00e+00h
1
  15  2.5343307e+02 1.95e-03 1.12e+01  -1.3 1.49e+01    -  1.00e+00 1.00e+00H
1
  16  2.5270286e+02 1.14e-03 1.45e+01  -1.6 6.40e+00    -  9.31e-01 1.00e+00H
1
  17  2.4977315e+02 4.43e-03 1.80e+00  -2.1 4.10e+00    -  1.00e+00 1.00e+00f
1
  18  2.4943444e+02 1.18e-03 8.00e-01  -2.9 3.16e+00    -  1.00e+00 1.00e+00f
1
  19  2.4909951e+02 1.18e-03 2.10e+00  -3.6 2.81e+00    -  1.00e+00 1.00e+00f
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  20  2.4902068e+02 4.12e-03 2.33e+00  -4.5 6.33e+00    -  1.00e+00 1.00e+00f
1
  21  2.4891931e+02 4.62e-03 1.28e+01  -4.8 7.72e+00    -  1.00e+00 5.07e-01f
1
  22  2.4820721e+02 1.30e-03 2.34e+00  -4.5 2.31e+00    -  1.00e+00 1.00e+00f
1
  23  2.4811148e+02 6.38e-04 1.16e+01  -4.0 1.82e+00    -  1.00e+00 5.23e-01h
1
  24  2.4810601e+02 6.27e-04 1.90e+01 -10.1 3.96e+00    -  5.81e-01 1.67e-02h
1
```

```
  25  2.4800523e+02 2.67e-03 1.22e+00  -5.2 7.73e+00    -  1.00e+00 1.00e+00f
1
  26  2.4796417e+02 2.56e-03 3.81e+01  -5.0 1.21e+01    -  1.00e+00 5.77e-02f
1
  27  2.4798754e+02 3.23e-05 1.05e+00  -5.3 3.78e+00    -  1.00e+00 1.00e+00H
1
  28  2.4781894e+02 9.23e-04 7.87e-01  -5.9 1.06e+00    -  1.00e+00 1.00e+00f
1
  29  2.4775256e+02 3.10e-05 2.05e-01  -7.1 3.78e-01    -  1.00e+00 1.00e+00h
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  30  2.4774536e+02 2.28e-05 1.24e-01  -6.3 6.15e-01    -  1.00e+00 1.00e+00h
1
  31  2.4774407e+02 5.25e-05 5.80e+01  -7.0 2.99e+00    -  1.00e+00 2.44e-01h
3
  32  2.4773482e+02 1.08e-04 5.63e-01  -8.2 1.68e+00    -  1.00e+00 1.00e+00h
1
  33  2.4773204e+02 9.38e-05 1.91e+02  -9.3 2.23e+00    -  1.00e+00 1.25e-01h
4
  34  2.4772959e+02 5.66e-06 8.25e-02 -10.2 2.22e-01    -  1.00e+00 1.00e+00h
1
  35  2.4773017e+02 2.32e-08 6.13e-02 -11.0 2.14e-01    -  1.00e+00 1.00e+00H
1
  36  2.4772924e+02 5.96e-06 4.81e-02 -11.0 2.62e-01    -  1.00e+00 1.00e+00f
1
  37  2.4772880e+02 5.68e-06 4.58e-02 -11.0 2.32e-01    -  1.00e+00 1.00e+00h
1
  38  2.4772808e+02 1.85e-06 3.12e-02 -11.0 1.32e-01    -  1.00e+00 1.00e+00h
1
  39  2.4772794e+02 3.52e-07 3.67e-02 -10.5 6.46e-02    -  1.00e+00 1.00e+00h
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  40  2.4772815e+02 1.00e-08 8.52e-02 -11.0 1.62e-01    -  1.00e+00 1.00e+00H
1
  41  2.4772813e+02 2.57e-06 5.62e+02 -11.0 4.02e-01    -  1.00e+00 5.00e-01f
2
  42  2.4772813e+02 3.73e-06 3.37e-02 -11.0 1.14e-01    -  1.00e+00 1.00e+00h
1
  43  2.4772768e+02 8.82e-07 3.59e-02 -11.0 6.57e-02    -  1.00e+00 1.00e+00h
1
  44  2.4772762e+02 2.94e-07 2.02e-02 -11.0 4.46e-02    -  1.00e+00 1.00e+00h
1
  45  2.4772756e+02 1.03e-07 7.18e-03 -11.0 2.39e-02    -  1.00e+00 1.00e+00h
1
  46  2.4772757e+02 8.34e-08 6.83e-03 -11.0 1.42e-02    -  1.00e+00 1.00e+00h
1
  47  2.4772756e+02 3.10e-08 1.59e-03 -11.0 8.23e-03    -  1.00e+00 1.00e+00h
1
  48  2.4772755e+02 1.00e-08 8.23e-04 -11.0 3.86e-03    -  1.00e+00 1.00e+00h
1
  49  2.4772758e+02 1.00e-08 1.06e-02 -11.0 5.67e-02    -  1.00e+00 1.00e+00H
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  50  2.4772760e+02 1.00e-08 1.04e-02 -11.0 1.73e-02    -  1.00e+00 1.00e+00H
```

```
1
  51  2.4772755e+02 8.09e-08 3.82e-03 -11.0 2.15e-02    -  1.00e+00 1.00e+00h
1
  52  2.4772755e+02 1.00e-08 2.20e-03 -11.0 2.70e-03    -  1.00e+00 1.00e+00h
1
  53  2.4772754e+02 1.00e-08 2.04e-04 -11.0 2.01e-03    -  1.00e+00 1.00e+00h
1
  54  2.4772754e+02 1.00e-08 1.76e-04 -11.0 5.82e-04    -  1.00e+00 1.00e+00h
1
  55  2.4772756e+02 1.00e-08 6.68e-03 -11.0 1.61e-02    -  1.00e+00 1.00e+00H
1
  56  2.4772754e+02 2.56e-08 2.29e-03 -11.0 9.25e-03    -  1.00e+00 1.00e+00h
1
  57  2.4772754e+02 1.00e-08 1.26e-03 -11.0 3.34e-03    -  1.00e+00 1.00e+00h
1
  58  2.4772754e+02 1.00e-08 7.41e-04 -11.0 1.39e-03    -  1.00e+00 1.00e+00h
1
  59  2.4772754e+02 1.00e-08 3.69e-04 -11.0 1.12e-03    -  1.00e+00 1.00e+00h
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  60  2.4772754e+02 1.00e-08 1.95e-04 -11.0 4.28e-04    -  1.00e+00 1.00e+00h
1
  61  2.4772754e+02 1.00e-08 1.09e-04 -11.0 2.04e-04    -  1.00e+00 1.00e+00h
1
  62  2.4772754e+02 1.00e-08 5.63e-05 -11.0 1.50e-04    -  1.00e+00 1.00e+00h
1
  63  2.4772754e+02 1.00e-08 4.61e-04 -11.0 1.93e-03    -  1.00e+00 1.00e+00H
1
  64  2.4772754e+02 1.00e-08 5.62e+02 -11.0 1.51e-03    -  1.00e+00 5.00e-01h
2
  65  2.4772754e+02 1.00e-08 3.24e-04 -11.0 1.12e-03    -  1.00e+00 1.00e+00h
1
  66  2.4772754e+02 1.00e-08 4.87e-04 -11.0 1.21e-03    -  1.00e+00 1.00e+00H
1
  67  2.4772754e+02 1.00e-08 2.11e-04 -11.0 1.10e-03    -  1.00e+00 1.00e+00h
1
  68  2.4772754e+02 1.00e-08 5.08e-04 -11.0 1.19e-03    -  1.00e+00 1.00e+00H
1
  69  2.4772754e+02 1.00e-08 1.32e-04 -11.0 9.88e-04    -  1.00e+00 1.00e+00h
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  70  2.4772754e+02 1.00e-08 2.71e-04 -11.0 6.51e-04    -  1.00e+00 1.00e+00h
1
  71  2.4772754e+02 1.00e-08 5.89e-05 -11.0 3.74e-04    -  1.00e+00 1.00e+00h
1
  72  2.4772754e+02 1.00e-08 9.16e-05 -11.0 2.45e-04    -  1.00e+00 1.00e+00h
1
  73  2.4772754e+02 1.00e-08 1.24e-05 -11.0 1.56e-04    -  1.00e+00 1.00e+00h
1

Number of Iterations....: 73

                                   (scaled)                 (unscaled)
Objective...............:   2.4772754375835552e+02    2.4772754375835552e+02
Dual infeasibility......:   1.2369357519331770e-05    1.2369357519331770e-05
```

```
Constraint violation....:   9.9999997617077813e-09    9.9999997617077813e-09
Variable bound violation:   9.9999997617077813e-09    9.9999997617077813e-09
Complementarity.........:   1.0000000028666082e-11    1.0000000028666082e-11
Overall NLP error.......:   2.8503568616139391e-07    1.2369357519331770e-05


Number of objective function evaluations             = 111
Number of objective gradient evaluations             = 74
Number of equality constraint evaluations            = 111
Number of inequality constraint evaluations          = 111
Number of equality constraint Jacobian evaluations   = 74
Number of inequality constraint Jacobian evaluations = 74
Number of Lagrangian Hessian evaluations             = 0
Total seconds in IPOPT                               = 21.971

EXIT: Optimal Solution Found.
```






```
┌ Info: Listening on: 127.0.0.1:8700, thread id: 1
└ @ HTTP.Servers /root/.julia/packages/HTTP/vnQzp/src/Servers.jl:382
┌ Info: MeshCat server started. You can open the visualizer by visiting the f
ollowing URL in your browser:
│ http://127.0.0.1:8700
└ @ MeshCat /root/.julia/packages/MeshCat/QXID5/src/visualizer.jl:64
```

```
Test Summary:                  | Pass  Total
walker trajectory optimization |  272    272
```

```
Test.DefaultTestSet("walker trajectory optimization", Any[], 272, false, fals
e)
```

In [ ]:

Q3 (5 pts)

Please fill out the following project form (one per group). This will primarily be for the TAs to use to understand what you are working on and hopefully be able to better assist you. If you haven't decided on certain aspects of the project, just include what you are currently thinking/what decisions you need to make.

(1) Write down your dynamics (handwritten, code, or latex). This can be continuous-time (include how you are discretizing your system) or discrete-time.

We're using MATLAB to automate the derivation process using the symbolic toolbox.

We have a symbolic matrix that represents the contrained lagrangian of the system when either foot is on the ground. It's 11x11 so I can't really paste it in here.

(2) What is your state (what does each variable represent)?

State = [x y t1 t2 t3 t4 t5 dx dy dt1 dt2 dt3 dt4 dt5]

They represent the x and y position of the mass, and joint angles of the different legs (and their time derivatives).

(3) What is your control (what does each variable represent)?

u = [tau1 tau2 tau3 tau4 tau5]

They represent joint torques at each joints

(4) Briefly describe your goal for the project. What are you trying to make the system do? Specify whether you are doing control, trajectory optimization, both, or something else.

The goal of the project is to develop a trajectory optimization strategy that enables a 5-link walker to ascend stairs. This involves creating a stable walking gait on flat ground and then adapting this gait to climb stairs.

(5) What are your costs?

Energy (minimizing torque applied) Tracking error from reference trajectory

(6) What are your constraints?

Assuming one foot in contact with ground Knee strike

(7) What solution methods are you going to try?

We will try to use hybrid direct collocation and NLP.

(8) What have you tried so far?

Implementing and simulating 3 links kneed passive walker down a slope in Julia with switching legs after the heel strike.

Modeling 5 link bipedal walker with a torso and two identical legs with knees.

(9) If applicable, what are you currently running into issues with?

Trying to make sure we have to correct dynamics and generating a reference trajectory

(10) If your system doesn't fit with some of the questions above or there are additional things you'd like to elaborate on, please explain/do that here.

Will Dircol only adhere to the reference trajectory or will it also optimize that to reduce cost? One question we have is hwo to reduce the amount of acutation efforts and take advantage of passive dynamics of the walker. Maybe the reference trajectory is not the most energy efficient and I wonder what set of costs and constraints will lead to the "most efficient control policy"?