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

```
  Activating environment at `~/Dropbox/My Mac (MacBook Pro (2))/Desktop/CMU/Optimal Control/HW4_S23/Project.toml`
┌ Warning: backend `PlotlyBase` is not installed.
└ @ Plots ~/.julia/packages/Plots/tDHxD/src/backends.jl:43
┌ Warning: backend `PlotlyKaleido` is not installed.
└ @ Plots ~/.julia/packages/Plots/tDHxD/src/backends.jl:43
```

In [2]:
```
include(joinpath(@__DIR__, "utils","ilc_visualizer.jl"))
```

Out[2]:
```
vis_traj! (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, video). We will model the dynamics of the car as with a simple nonlinear bicycle model, with the following state and control:

$$x = \begin{bmatrix} p_x \\ p_y \\ \theta \\ \delta \\ v \end{bmatrix}, \qquad u = \begin{bmatrix} a \\ \dot{\delta} \end{bmatrix} \tag{1}$$

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 [3]:
```
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
```

```julia
    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
```

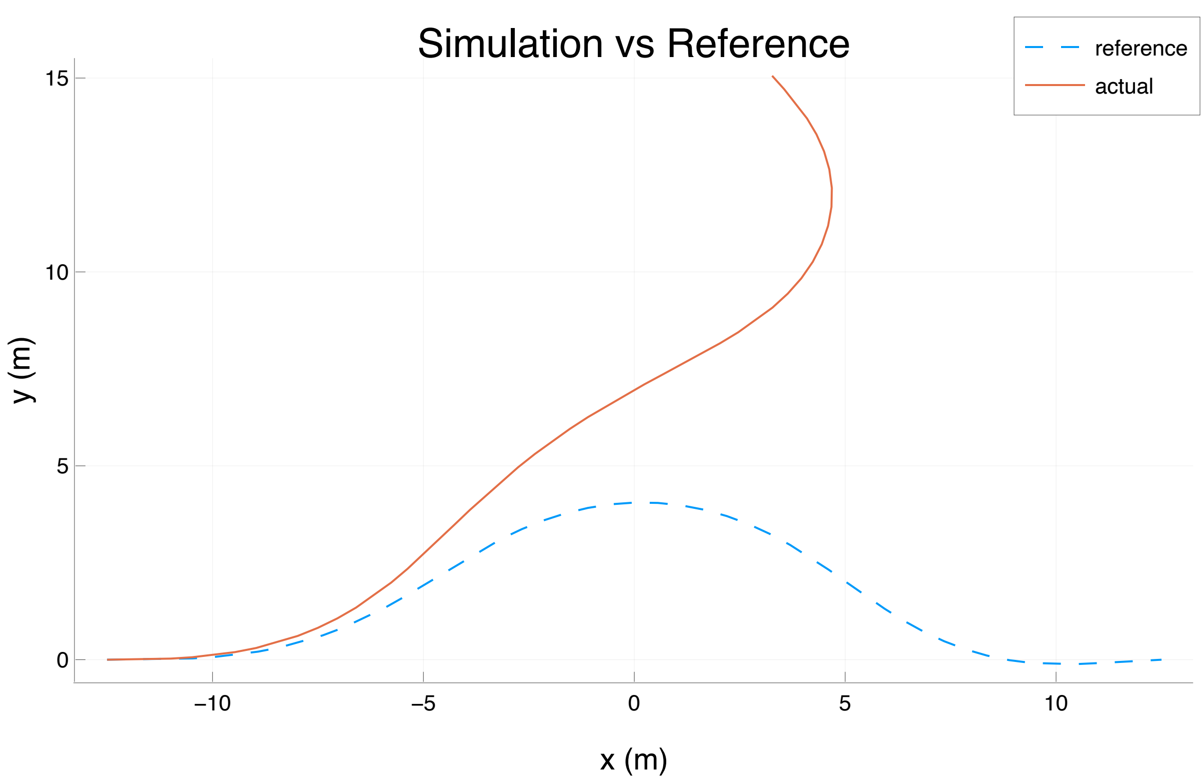
Out[3]: rk4 (generic function with 1 method)

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 [4]:

```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]
    Xsim[1] = Xref[1]
    for i = 1:(N-1)
        Xsim[i+1] = rk4(model, true_car_dynamics, Xsim[i], Uref[i], dt)
    end


    # --------testing------------------
    @test norm(Xsim[1] - Xref[1]) == 0
    @test norm(Xsim[end] - [3.26801052, 15.0590156, 2.0482790, 0.39056168, 4.5],Inf) < 1

    # --------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 Reference")
    display(plot!(Xm[1,:], Xm[2,:], label = "actual"))

end
```



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

Out[4]: `Test.DefaultTestSet("sim to real gap", Any[], 2, false, false)`

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, 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:

$$\min_{\Delta x_{1:N}, \Delta u_{1:N-1}} \quad J(X_{sim} + \Delta X, \bar{U} + \Delta U) \tag{2}$$
$$\text{st} \quad \Delta x_1 = 0 \tag{3}$$
$$\Delta x_{k+1} = A_k \Delta x_k + B_k \Delta u_k \quad \text{for } k = 1, 2, \ldots, N-1 \tag{4}$$

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 [5]:
```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 iterat
                         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
    # TODO: return trajectory cost J(Xsim, Ubar)
    N = length(Xsim);
    for i = 1:(N-1)
        X_tilde = Xsim[i] - Xref[i];
        U_tilde = Ubar[i] - Uref[i];
```

```julia
#         J += 0.5*cvx.quadform(X_tilde, Q)
#         J += 0.5*cvx.quadform(U_tilde, R)
        J += 0.5*X_tilde'*Q*X_tilde + 0.5*U_tilde'*R*U_tilde
    end
    Xf_tilde = Xsim[N] - Xref[N];
#     J += 0.5*cvx.quadform(Xf_tilde, Qf)
    J += 0.5*Xf_tilde'*Qf*Xf_tilde
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 th
                    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
                    Bs::Vector{Matrix{Float64}},   # vector of B jacobians at each time
                    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)

    # TODO: cost function (tracking cost on Xref, Uref)
    cost = 0.0
    for i = 1:(N-1)
        cost += 0.5*cvx.quadform((Xsim[i] + ΔX[:,i] - Xref[i]), Q)
        cost += 0.5*cvx.quadform((Ubar[i] + ΔU[:,i] - Uref[i]), R)
    end
    cost += 0.5*cvx.quadform((Xsim[N] + ΔX[:,N] - Xref[N]), Qf)

    # problem instance
    prob = cvx.minimize(cost)

    # TODO: initial condition constraint
    prob.constraints += (ΔX[:,1] == zeros(nx, 1))

    # 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
```

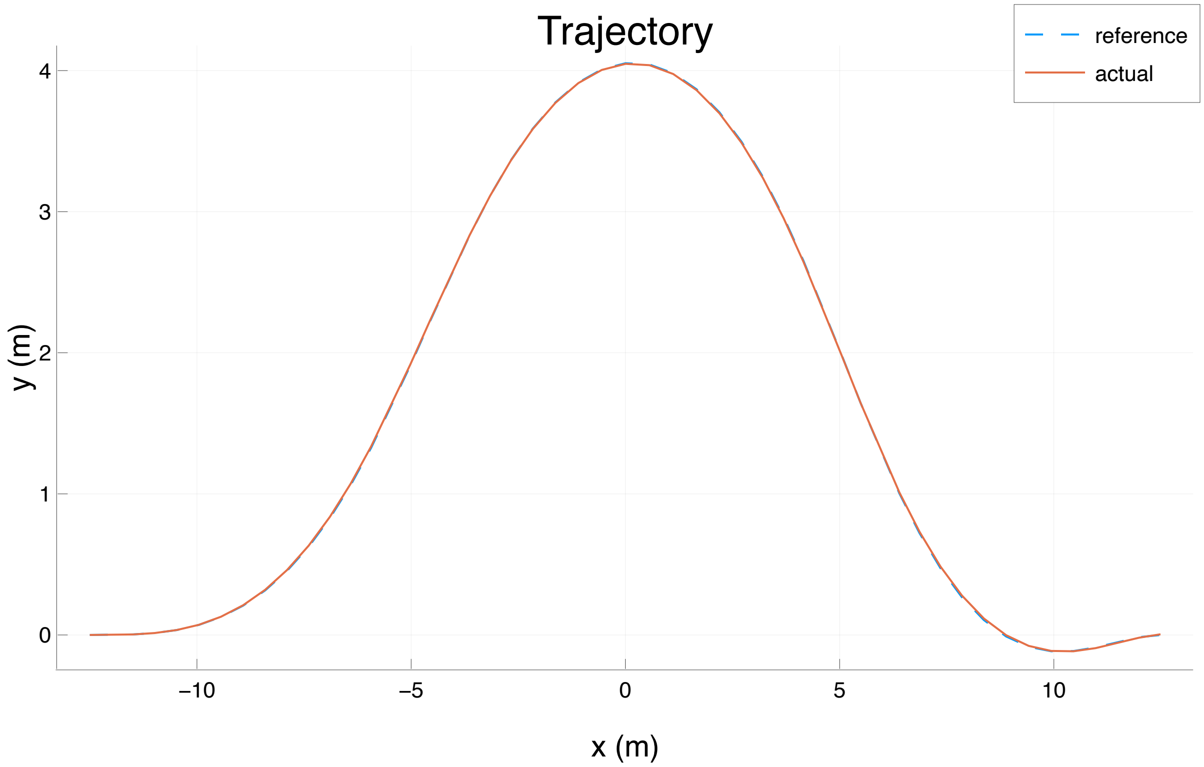
Out[5]: ilc_update (generic function with 1 method)

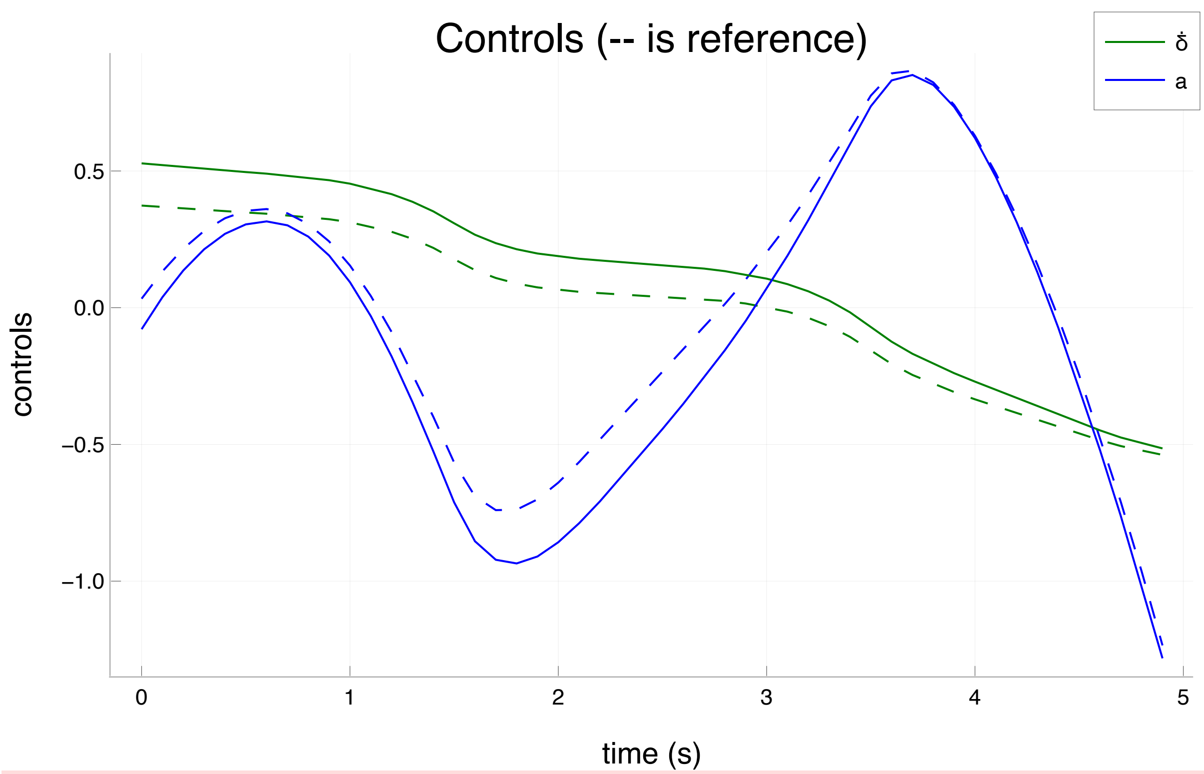
Here you will run your ILC algorithm. The resulting plots should show the simulated trajectory `Xsim` tracks `Xref` very closely, but there should be a significant difference between `Uref` and `Ubar`.

In [6]:
```
@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
    As = [zeros(nx, nx) for i = 1:(N-1)]
    Bs = [zeros(nx, nu) for i = 1:(N-1)]
    for i = 1:(N-1)
        As[i] = FD.jacobian(x -> rk4(model, true_car_dynamics, x, Uref[i], dt), Xref[i])
        Bs[i] = FD.jacobian(u -> rk4(model, true_car_dynamics, Xref[i], u, dt), Uref[i])
    end

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

    for ilc_iter = 1:10 # it should not take more than 10 iterations to converge

        # TODO: rollout
        Xsim[1] = Xref[1]
        for i = 1:(N-1)
```

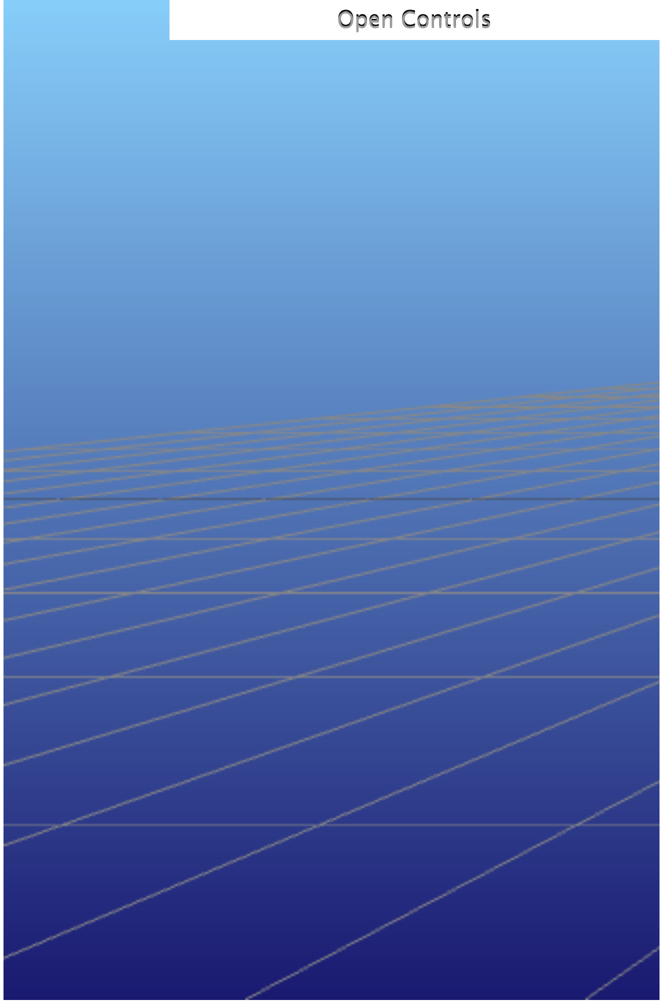
```julia
            Xsim[i+1] = rk4(model, true_car_dynamics, Xsim[i], Ubar[i], dt)
        end

        # TODO: calculate objective val (trajectory_cost)
        obj_val = trajectory_cost(Xsim, Ubar, Xref, Uref, Q, R, Qf)

        # solve optimization problem for update (ilc_update)
        ΔU = ilc_update(Xsim, Ubar, Xref, Uref, As, Bs, 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 reference)")
    display(plot!(t_vec[1:end-1], Um', label = ["δ̇" "a"], lc = [:green :blue]))

    # animation
    vis = Visualizer()
    X_vis = [[x[1],x[2],0.1] for x in Xsim]
    vis_traj!(vis, :traj, X_vis; R = 0.02)
    vis_model = TrajOptPlots.RobotZoo.BicycleModel()
    TrajOptPlots.set_mesh!(vis, vis_model)
    X  = [x[SA[1,2,3,4]] for x in Xsim]
    visualize!(vis, vis_model, tf, X)
    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    1.436e+03   6.701e+01
  2    8.969e+02   3.614e+01
  3    7.951e+02   4.016e+01
  4    4.823e+02   1.929e+01
  5    2.625e+02   3.530e+01
  6    7.354e+01   1.646e+01
  7    9.984e+00   9.419e+00
  8    2.809e-01   1.212e+00
  9    7.146e-02   2.535e-02
 10    7.142e-02   1.815e-04
```





┌ **Info:** MeshCat server started. You can open the visualizer by visiting the following UR
L in your browser:
└ http://127.0.0.1:8700




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

Out[6]: `Test.DefaultTestSet("ILC", Any[], 2, false, false)`

In [ ]:

In [ ]:

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; plotly()
using Random
using JLD2
using Test
using MeshCat
const mc = MeshCat
using TrajOptPlots
using StaticArrays
using Printf
```

```
  Activating environment at `~/Dropbox/My Mac (MacBook Pro (2))/Desktop/CMU/Optimal Control/HW4_S23/Project.toml`
[ Info: Precompiling PlotlyBase [a03496cd-edff-5a9b-9e67-9cda94a718b5]
[ Info: Precompiling PlotlyKaleido [f2990250-8cf9-495f-b13a-cce12b45703c]
┌ Warning: backend `PlotlyBase` is not installed.
└ @ Plots ~/.julia/packages/Plots/tDHxD/src/backends.jl:43
┌ Warning: backend `PlotlyKaleido` is not installed.
└ @ Plots ~/.julia/packages/Plots/tDHxD/src/backends.jl:43
```

In [2]:
```julia
include(joinpath(@__DIR__, "utils","fmincon.jl"))
include(joinpath(@__DIR__, "utils","walker.jl"))
```

Out[2]: `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{align} r^{(b)} &= \begin{bmatrix} p_x^{(b)} \\ p_y^{(b)} \end{bmatrix} & v^{(b)} &= \begin{bmatrix} v_x^{(b)} \\ v_y^{(b)} \end{bmatrix}\\ r^{(1)} &= \begin{bmatrix} p_x^{(1)} \\ p_y^{(1)} \end{bmatrix} & v^{(1)} &= \begin{bmatrix} v_x^{(1)}\\ v_y^{(1)} \end{bmatrix}\\ r^{(2)} &=

\begin{bmatrix} p_x^{(2)} \\ p_y^{(2)} \end{bmatrix} & v^{(2)} &= \begin{bmatrix} v_x^{(2)}\\ v_y^{(2)} \end{bmatrix} \end{align}$$ 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:

$$ x = \begin{bmatrix} p_x^{(b)} \\ p_y^{(b)} \\ p_x^{(1)} \\ p_y^{(1)} \\ p_x^{(2)} \\ p_y^{(2)} \\ v_x^{(b)} \\ v_y^{(b)} \\ v_x^{(1)} \\ v_y^{(1)} \\ v_x^{(2)} \\ v_y^{(2)} \\ \end{bmatrix} \quad u = \begin{bmatrix} F^{(1)} \\ F^{(2)} \\ \tau \end{bmatrix} $$
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 [3]:
```
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
                ]  # velocities
```

```julia
    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]

    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
```

Out[3]: `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:

$$ \begin{align} \mathcal{M}_1 &= \{1\text{:}5,11\text{:}15,21\text{:}25,31\text{:}35,41\text{:}45\} \\ \mathcal{M}_2 &= \{6\text{:}10,16\text{:}20,26\text{:}30,36\text{:}40\} \end{align}$$
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. $\mathcal{J}_i \equiv \{k+1 \notin \mathcal{M}_i \; | \; k \in \mathcal{M}_i\}$. We can write these out explicitly as the following:

$$ \begin{align} \mathcal{J}_1 &= \{5,15,25,35\} \\ \mathcal{J}_2 &= \{10,20,30,40\} \end{align}$$
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 [4]:
```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
```

Out[4]:
```
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`):

$$ J(x_{1:N},u_{1:N-1}) = \sum_{i=1}^{N-1} \bigg[\frac{1}{2} (x_i - x_{ref,i})^TQ(x_i - x_{ref,i}) + \frac{1}{2} (u_i-u_{ref,i})^TR(u_i-u_{ref,i}) \bigg] + \frac{1}{2}(x_N - x_{ref,N})^TQ_f(x_N - x_{ref,N})$$ Which goes into the following full optimization problem: $$ \begin{align} \min_{x_{1:N},u_{1:N-1}} \quad & J(x_{1:N},u_{1:N-1}) & \\ \text{st} \quad & x_1 = x_{ic} & \tag{1}\\ \quad & x_N = x_{g} &\tag{2}\\ & x_{k+1} = f_1(x_k,u_k) &\text{for } k \in \mathcal{M}_1 \setminus \mathcal{J}_1 \tag{3}\\ & x_{k+1} = f_2(x_k,u_k) &\text{for } k \in \mathcal{M}_2 \setminus \mathcal{J}_2 \tag{4}\\ & x_{k+1} = g_2(f_1(x_k,u_k)) &\text{for } k \in \mathcal{J}_1 \tag{5}\\ & x_{k+1} = g_1(f_2(x_k,u_k)) &\text{for } k \in \mathcal{J}_2 \tag{6}\\ & x_{k}[4] = 0 &\text{for } k \in \mathcal{M}_1 \tag{7}\\ & x_{k}[6] = 0 &\text{for } k \in \mathcal{M}_2 \tag{8}\\ & 0.5 \leq \| r^{(b)}_k - r^{(1)}_k\|_2 \leq 1.5 & \text{for } k \in [1, N]\tag{9} \\ & 0.5 \leq \| r^{(b)}_k - r^{(2)}_k\|_2 \leq 1.5 & \text{for } k \in [1, N]\tag{10} \\ & x_{k}[2,4,6] \geq 0 & \text{for } k \in [1, N]\tag{11} \end{align}$$

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 [5]:
```julia
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
```

Out[5]:
```
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: $$ \begin{align} \min_{z} \quad & \ell(z) & \text{cost function}\\ \text{st} \quad & c_{eq}(z) = 0 & \text{equality constraint}\\ & c_L \leq c_{ineq}(z) \leq c_U & \text{inequality constraint}\\ & z_L \leq z \leq z_U & \text{primal bound constraint} \end{align}$$

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 [6]:
```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_tilde = Z[idx.x[i]] - Xref[i]
        ui_tilde = Z[idx.u[i]] - Uref[i]
        J += 0.5*xi_tilde'*Q*xi_tilde + 0.5*ui_tilde'*R*ui_tilde
    end
    xN_tilde = Z[idx.x[N]] - Xref[N]
    J += 0.5*xN_tilde'*Qf*xN_tilde

    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 problem)
    for k = 1:(N-1)
        xk = Z[idx.x[k]]
        uk = Z[idx.u[k]]
        if (k in M1) && !(k in J1)
#             @show "Stance 1"
            c[idx.c[k]] = Z[idx.x[k+1]] - rk4(model, stance1_dynamics, xk, uk, dt)
        elseif (k in M2) && !(k in J2)
#             @show "Stance 2"
            c[idx.c[k]] = Z[idx.x[k+1]] - rk4(model, stance2_dynamics, xk, uk, dt)
        elseif (k in M1) && (k in J1)
#             @show "Transitioning from stance 1 to stance 2"
            c[idx.c[k]] = Z[idx.x[k+1]] - jump2_map(rk4(model, stance1_dynamics, xk, uk,
        elseif (k in M2) && (k in J2)
#             @show "Transitioning from stance 2 to stance 1"
            c[idx.c[k]] = Z[idx.x[k+1]] - jump1_map(rk4(model, stance2_dynamics, xk, uk,
        end
    end

    return c
```

```julia
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 problem)
    for k = 1:N
        xk = Z[idx.x[k]]
        if (k in M1)
            c[k] = xk[4]
        elseif (k in M2)
            c[k] = xk[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)
    initialCondition = Z[idx.x[1]] - xic
    terminalCondition = Z[idx.x[N]] - xg
    dynamicsConstraints = walker_dynamics_constraints(params, Z)
    stanceConstraints = walker_stance_constraint(params, Z)

    return [initialCondition; terminalCondition; dynamicsConstraints; stanceConstraints]
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-1)
        xi = Z[idx.x[i]]
        rbi = xi[1:2]
        r1i = xi[3:4]
        r2i = xi[5:6]
        c[2*i] = norm(rbi-r1i)
        c[2*i+1] = norm(rbi-r2i)
    end
```

```julia
    return c
end
```

Out[6]: walker_inequality_constraint (generic function with 1 method)

In [7]:
```julia
@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
    )

    # ...al bounds (constraint 11)
```

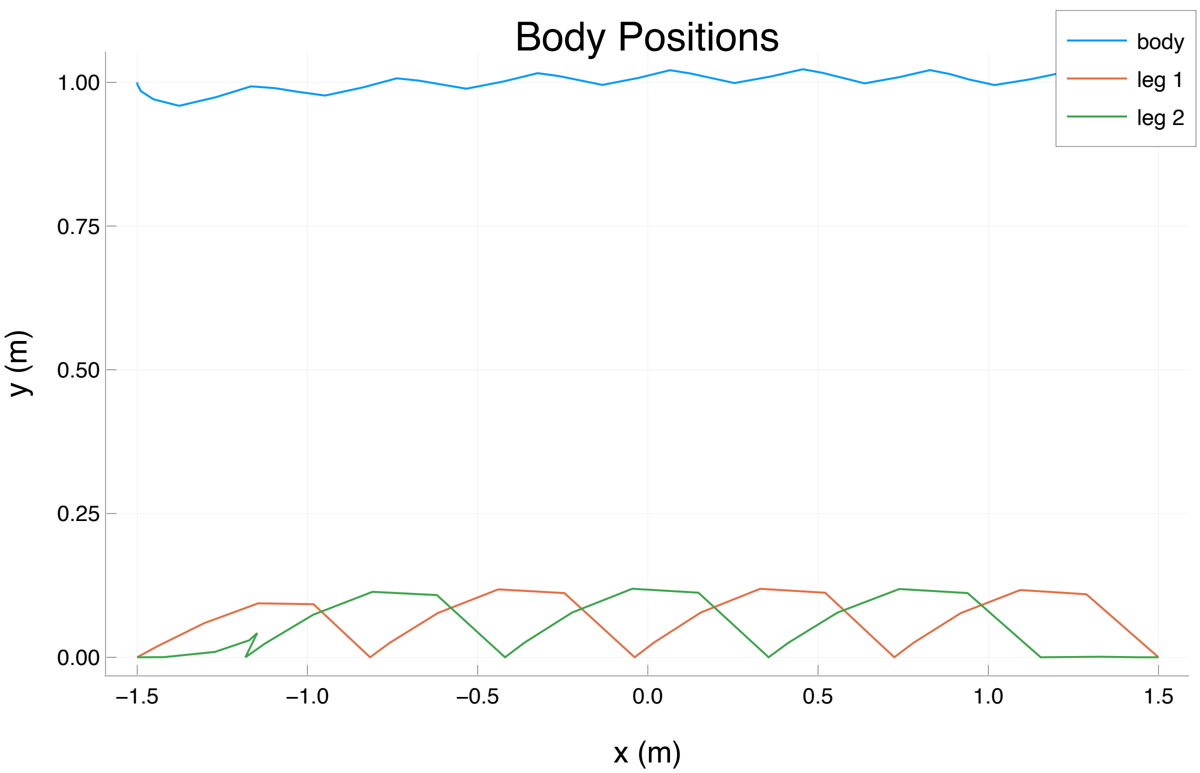
```
    x_l = -Inf*ones(idx.nz) # update this
    x_u =  Inf*ones(idx.nz) # update this
    for i = 1:N
        xi_l = -Inf*ones(idx.nx)
        xi_l[2] = 0
        xi_l[4] = 0
        xi_l[6] = 0
        x_l[idx.x[i]] = xi_l
    end
#     @show size(idx.x[1])
#     @show size(x_l)

    # TODO: inequality constraint bounds
    c_l = 0.5*ones(2*N) # update this
    c_u = 1.5*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]
#     @show size(z0)
#     @show size(Xref[1])
#     @show size(Xref)
#     @show size(Xref)*size(Xref[1])
#     @show size(Uref)
#     @show f
    # 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_constraint,
                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
            te_walker_pose!(vis, model::NamedTuple, X[k])
        end
```

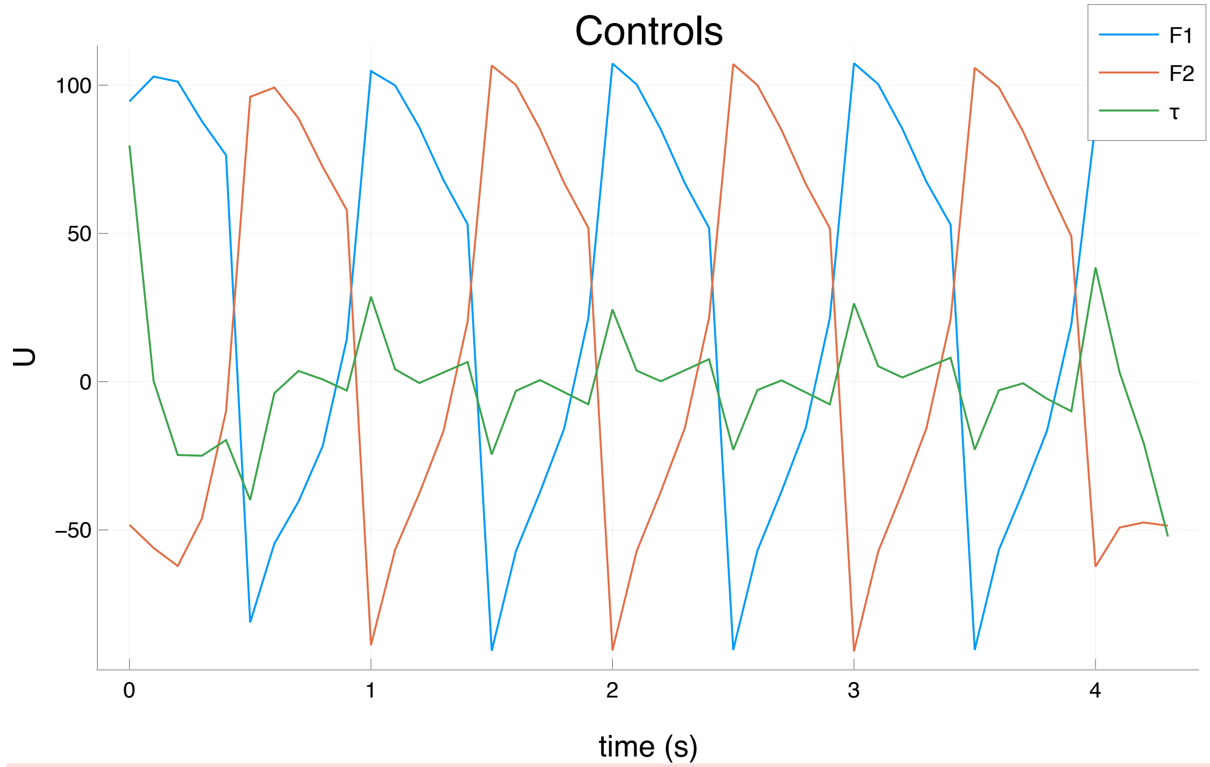
```julia
    end
    mc.setanimation!(vis, anim)
    display(render(vis))

    # --------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----------------------

******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit https://github.com/coin-or/Ipopt
******************************************************************************

This is Ipopt version 3.13.4, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt documentation).

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  2.2067459e-01 1.44e+00 2.20e+01  -0.7 1.18e+02    -  3.33e-01 1.94e-02h  1
   2  2.2269790e-01 1.44e+00 9.22e+04   0.1 1.72e+02    -  7.05e-01 1.98e-04h  1
   3  2.3692405e-01 1.44e+00 1.81e+07   0.2 1.13e+02    -  3.75e-01 1.96e-03h  1
   4  9.2299673e+01 8.40e-01 9.30e+06  -1.1 9.45e+01    -  5.22e-01 5.07e-01h  1
   5  4.6850264e+02 1.99e+00 6.38e+07  -0.4 1.32e+02    -  2.25e-01 9.90e-01h  1
   6  3.3334467e+02 1.19e+00 3.55e+07  -0.1 4.96e+01    -  1.00e+00 4.24e-01f  1
   7  2.8329193e+02 5.00e-01 3.28e+01   0.2 4.59e+01    -  1.00e+00 1.00e+00f  1
   8  2.8362447e+02 5.00e-01 1.27e+07   0.2 2.09e+01    -  7.14e-01 1.00e+00h  1
   9  2.6774338e+02 5.00e-01 2.32e+01  -0.1 1.35e+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  2.6185563e+02 5.00e-01 1.79e+06  -0.6 7.72e+00    -  9.66e-01 1.00e+00h  1
  11  2.6507564e+02 5.00e-01 1.24e+01  -0.8 1.13e+01    -  1.00e+00 1.00e+00H  1
  12  2.5600063e+02 5.00e-01 1.19e+06  -0.9 6.41e+00    -  9.85e-01 1.00e+00f  1
  13  2.5535597e+02 5.00e-01 2.29e+00  -1.5 4.16e+00    -  1.00e+00 1.00e+00h  1
  14  2.5391897e+02 5.00e-01 3.16e+05  -2.3 3.18e+00    -  9.97e-01 1.00e+00f  1
  15  2.5245044e+02 5.00e-01 6.35e+00  -3.0 1.05e+01    -  1.00e+00 1.00e+00f  1
  16  2.5140563e+02 5.00e-01 2.37e+08  -2.8 6.52e+01    -  1.00e+00 1.62e-01f  1
  17  2.6271830e+02 5.00e-01 1.20e+08  -2.8 2.25e+01    -  1.00e+00 7.22e-01H  1
  18  2.4920348e+02 5.00e-01 1.16e+08  -3.0 9.59e+00    -  2.48e-01 1.00e+00f  1
  19  2.4893816e+02 5.00e-01 1.50e+00  -3.6 1.95e+00    -  1.00e+00 1.00e+00h  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  20  2.4861374e+02 5.00e-01 1.52e+07  -3.9 2.48e+00    -  9.61e-01 1.00e+00h  1
  21  2.4885118e+02 5.00e-01 4.28e+07  -2.7 5.70e+00    -  9.55e-01 1.00e+00F  1
  22  2.4844193e+02 5.00e-01 2.70e+07  -2.2 5.90e+00    -  1.00e+00 8.89e-01f  1
  23  2.4840088e+02 5.00e-01 1.09e+00  -3.1 2.06e+00    -  1.00e+00 1.00e+00h  1
  24  2.4799459e+02 5.00e-01 1.22e+05  -3.1 1.31e+00    -  1.00e+00 1.00e+00h  1
  25  2.4795915e+02 5.00e-01 1.13e-01  -2.9 1.00e+00    -  1.00e+00 1.00e+00h  1
[illegible]+02 5.00e-01 2.75e+00  -2.6 1.49e+01    -  1.00e+00 1.00e+00H  1
  27  2.4810344e+02 5.00e-01 1.75e+08  -2.9 3.31e+01    -  9.48e-01 1.76e-01h  3
```

```
  28  2.4808595e+02 5.00e-01 2.13e+07  -2.9 8.57e+00    -  9.41e-01 1.00e+00H  1
  29  2.5010280e+02 5.00e-01 3.38e+08  -2.9 1.67e+01    -  6.90e-01 1.00e+00H  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  30  2.4813644e+02 5.00e-01 6.70e+07  -2.9 1.29e+01    -  9.41e-01 1.00e+00F  1
  31  2.4788602e+02 5.00e-01 2.10e+08  -2.9 1.95e+01    -  1.00e+00 2.15e-01f  1
  32  2.4913096e+02 5.00e-01 2.33e+00  -3.5 5.38e+00    -  1.00e+00 1.00e+00H  1
  33  2.4777080e+02 5.00e-01 1.78e-01  -3.6 5.12e+00    -  1.00e+00 1.00e+00f  1
  34  2.4773901e+02 5.00e-01 3.33e-01  -4.8 4.09e-01    -  1.00e+00 1.00e+00h  1
In iteration 34, 2 Slacks too small, adjusting variable bounds
  35  2.4773196e+02 5.00e-01 9.34e-02  -6.3 1.59e-01    -  1.00e+00 1.00e+00h  1
In iteration 35, 2 Slacks too small, adjusting variable bounds
  36  2.4773133e+02 5.00e-01 2.84e-02  -6.9 9.88e-02    -  1.00e+00 1.00e+00h  1
In iteration 36, 2 Slacks too small, adjusting variable bounds
  37  2.4773034e+02 5.00e-01 2.94e-02  -7.5 1.59e-01    -  1.00e+00 1.00e+00h  1
  38  2.4773016e+02 5.00e-01 2.08e+09  -5.5 1.14e+00    -  1.00e+00 4.31e-01h  1
  39  2.4773671e+02 5.00e-01 1.57e-01  -5.2 3.06e-01    -  1.00e+00 1.00e+00H  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  40  2.4772826e+02 5.00e-01 2.28e-02  -3.9 2.27e-01    -  1.00e+00 1.00e+00h  1
  41  2.4772792e+02 5.00e-01 1.23e-02  -5.2 6.58e-02    -  1.00e+00 1.00e+00h  1
  42  2.4772812e+02 5.00e-01 6.13e-02  -4.5 2.28e-01    -  1.00e+00 1.00e+00h  1
  43  2.4773487e+02 5.00e-01 1.71e-01  -4.6 5.51e-01    -  1.00e+00 1.00e+00H  1
  44  2.4772808e+02 5.00e-01 4.34e-02  -4.6 5.57e-01    -  1.00e+00 1.00e+00H  1
  45  2.4772802e+02 5.00e-01 5.32e+08  -4.6 5.11e-01    -  1.00e+00 2.50e-01h  3
  46  2.4772763e+02 5.00e-01 1.36e+08  -4.6 1.30e-01    -  1.00e+00 8.09e-01h  1
In iteration 46, 2 Slacks too small, adjusting variable bounds
  47  2.4772781e+02 5.00e-01 3.96e+07  -6.1 1.28e-01    -  1.00e+00 9.72e-01H  1
In iteration 47, 2 Slacks too small, adjusting variable bounds
  48  2.4772759e+02 5.00e-01 7.81e-03  -7.2 8.45e-02    -  1.00e+00 1.00e+00h  1
In iteration 48, 2 Slacks too small, adjusting variable bounds
  49  2.4772787e+02 5.00e-01 1.46e+08  -7.8 6.97e-02    -  1.00e+00 9.77e-01H  1
In iteration 49, 2 Slacks too small, adjusting variable bounds
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  50  2.4772755e+02 5.00e-01 1.78e-03  -7.9 6.33e-02    -  1.00e+00 1.00e+00h  1
In iteration 50, 2 Slacks too small, adjusting variable bounds
  51  2.4772755e+02 5.00e-01 2.09e-03  -8.7 3.37e-03    -  1.00e+00 1.00e+00h  1
In iteration 51, 2 Slacks too small, adjusting variable bounds
  52  2.4772755e+02 5.00e-01 1.33e-03  -9.4 2.08e-03    -  1.00e+00 1.00e+00h  1
In iteration 52, 2 Slacks too small, adjusting variable bounds
  53  2.4772755e+02 5.00e-01 7.56e-04  -9.5 2.23e-03    -  1.00e+00 1.00e+00h  1
In iteration 53, 2 Slacks too small, adjusting variable bounds
  54  2.4772754e+02 5.00e-01 1.72e-03  -9.5 4.71e-03    -  1.00e+00 1.00e+00h  1
In iteration 54, 2 Slacks too small, adjusting variable bounds
  55  2.4772754e+02 5.00e-01 1.10e+10  -9.5 5.68e-02    -  1.00e+00 6.25e-02h  5
In iteration 55, 2 Slacks too small, adjusting variable bounds
  56  2.4772754e+02 5.00e-01 1.02e-03  -9.5 7.32e-03    -  1.00e+00 1.00e+00h  1
In iteration 56, 2 Slacks too small, adjusting variable bounds
  57  2.4772754e+02 5.00e-01 1.03e+10  -9.5 1.97e-02    -  1.00e+00 1.25e-01h  4
In iteration 57, 2 Slacks too small, adjusting variable bounds
  58  2.4772754e+02 5.00e-01 7.03e-04  -9.5 1.48e-03    -  1.00e+00 1.00e+00h  1
In iteration 58, 2 Slacks too small, adjusting variable bounds
  59  2.4772754e+02 5.00e-01 9.33e-04  -9.5 1.36e-03    -  1.00e+00 1.00e+00h  1
In iteration 59, 2 Slacks too small, adjusting variable bounds
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  60  2.4772754e+02 5.00e-01 5.86e+09  -9.5 7.16e-04    -  1.00e+00 5.00e-01h  2
In iteration 60, 2 Slacks too small, adjusting variable bounds
  61  2.4772754e+02 5.00e-01 1.59e-04  -9.5 8.42e-04    -  1.00e+00 1.00e+00h  1
  62r 2.4772754e+02 5.00e-01 1.00e+03  -0.3 0.00e+00    -  0.00e+00 4.77e-07R 22
  63r 2.4772754e+02 5.00e-01 6.42e+00  -6.4 5.00e-04    -  9.90e-01 9.90e-01f  1
  64r 2.4772754e+02 5.00e-01 9.88e-01  -8.4 4.77e-04    -  8.46e-01 9.89e-01f  1
  65r 2.4772765e+02 5.00e-01 2.72e-02  -6.4 4.05e-02    -  9.73e-01 6.38e-01f  1
Loading [MathJax]/extensions/Safe.js +02 5.00e-01 1.22e-04  -9.0 6.71e-03    -  9.96e-01 8.92e-01f  1
  67r 2.4778141e+02 5.00e-01 6.26e+01  -6.7 2.72e+00    -  9.97e-01 8.77e-01f  1
```

```
  68r 2.4774202e+02 5.00e-01 6.07e-02  -7.0 2.40e+00    -  1.00e+00 9.98e-01h  1
  69r 2.4774193e+02 5.00e-01 7.41e+01  -6.0 1.28e+00    -  1.00e+00 3.89e-01h  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  70r 2.4773665e+02 5.00e-01 2.71e-04  -6.4 6.94e-01    -  1.00e+00 1.00e+00H  1
  71r 2.4773150e+02 5.00e-01 1.33e+01  -8.2 3.70e-01    -  1.00e+00 8.71e-01h  1
  72r 2.4773065e+02 5.00e-01 2.23e-02  -9.0 1.02e-01    -  9.92e-01 1.00e+00h  1

Number of Iterations....: 72

                                   (scaled)                 (unscaled)
Objective...............:   2.4773062020996909e+02    2.4773062020996909e+02
Dual infeasibility......:   3.8011748820607667e+00    3.8011748820607667e+00
Constraint violation....:   4.9999998999999989e-01    4.9999998999999989e-01
Complementarity.........:   5.0120786392302953e-09    5.0120786392302953e-09
Overall NLP error.......:   3.8011748820607667e+00    3.8011748820607667e+00


Number of objective function evaluations             = 133
Number of objective gradient evaluations             = 64
Number of equality constraint evaluations            = 133
Number of inequality constraint evaluations          = 133
Number of equality constraint Jacobian evaluations   = 75
Number of inequality constraint Jacobian evaluations = 75
Number of Lagrangian Hessian evaluations             = 0
Total CPU secs in IPOPT (w/o function evaluations)   =     50.288
Total CPU secs in NLP function evaluations           =     25.839

EXIT: Converged to a point of local infeasibility. Problem may be infeasible.
```






```
┌ Info: MeshCat server started. You can open the visualizer by visiting the following UR
L in your browser:
└ http://127.0.0.1:8701
```

Open Controls

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

Out[7]: `Test.DefaultTestSet("walker trajectory optimization", Any[], 272, false, false)`

In [ ]:

```

```

In [ ]:

```

```