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 TrajOptPlots
#using StaticArrays
using Printf
```

```
  Activating environment at `C:\Users\rdesa\OneDrive\Desktop\OCRL_HW4\HW4_S23
-main\Project.toml`
```

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:

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

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 [3]:
```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
```

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:

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

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 [4]:
```
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
```

In [5]:
```
J1 = [5,15,25,35]

print(5 in J1)
```

```
true
```

In [6]:
```
J1 = [5,15,25,35]

print(!(4 in J1))
```

```
true
```

In [ ]:

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 [7]:
```
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[7]:
```
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 [8]:

```
# 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 k = 1:(N-1)
        xk = Z[idx.x[k]]
        uk = Z[idx.u[k]]
        x_k = xk - Xref[k]
        u_k = uk - Uref[k]
        J += 0.5*x_k'*Q*x_k + 0.5*u_k'*R*u_k
    end
    xN = Z[idx.x[N]]
    x_N = xN - Xref[N]
    J += 0.5*x_N'*Qf*x_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)
    XWalk = [Z[idx.x[i]] for i = 1:N]

    for i = 1:(N-1)
        xi = Z[idx.x[i]]
        ui = Z[idx.u[i]]
        if i in M1 && !(i in J1)

            c[idx.c[i]] = rk4(model,stance1_dynamics,xi,ui,dt) - Z[idx.x[i+1]]

        elseif i in M2 && !(i in J2)

            c[idx.c[i]] = rk4(model,stance2_dynamics,xi,ui,dt) - Z[idx.x[i+1]]

        elseif i in J1

            c[idx.c[i]] = jump2_map(rk4(model,stance1_dynamics,xi,ui,dt)) - Z
[idx.x[i+1]]

        elseif i in J2

            c[idx.c[i]] = jump1_map(rk4(model,stance2_dynamics,xi,ui,dt)) - Z
[idx.x[i+1]]

        else
            print("\n Not Accounted For \n")
        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

        if i in M1

            c[i] = Z[idx.x[i][4]]
        elseif i in M2

            c[i] = Z[idx.x[i][6]]
        else
            print("Not Accounted For")
        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

    c = [
          Z[idx.x[1]] - xic;
          Z[idx.x[N]] - xg;
          walker_dynamics_constraints(params, Z);
          walker_stance_constraint(params, Z)
        ]

    return c
    # 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)

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
        x = Z[idx.x[i]]
        rb = x[1:2]
        rf1 = x[3:4]
        c[i] = norm(rb-rf1)^2
    end

    for i = 1:N
        x = Z[idx.x[i]]
        rb = x[1:2]
        rf2 = x[5:6]
        c[i+N] = norm(rb-rf2)^2
    end


    return c
end
```

Out[8]: walker_inequality_constraint (generic function with 1 method)

In [9]:
```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
    )

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

```julia
    x_l = -Inf*ones(idx.nz)

    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

    x_u = Inf*ones(idx.nz)

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

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

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

    diff_type = :auto

    print("\n Starting fmincon \n")
    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
```

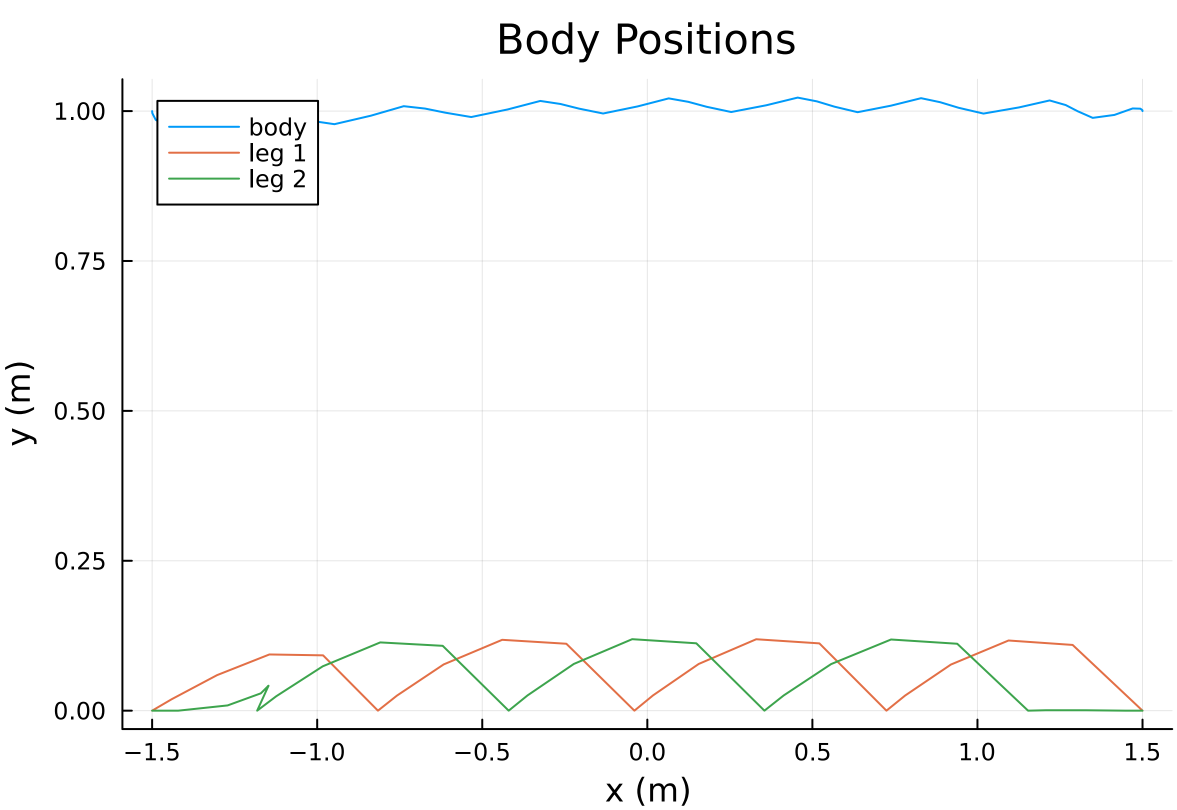
```
    #end
    #mc.setanimation!(vis, anim)
    #display(render(vis))

    # --------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
```

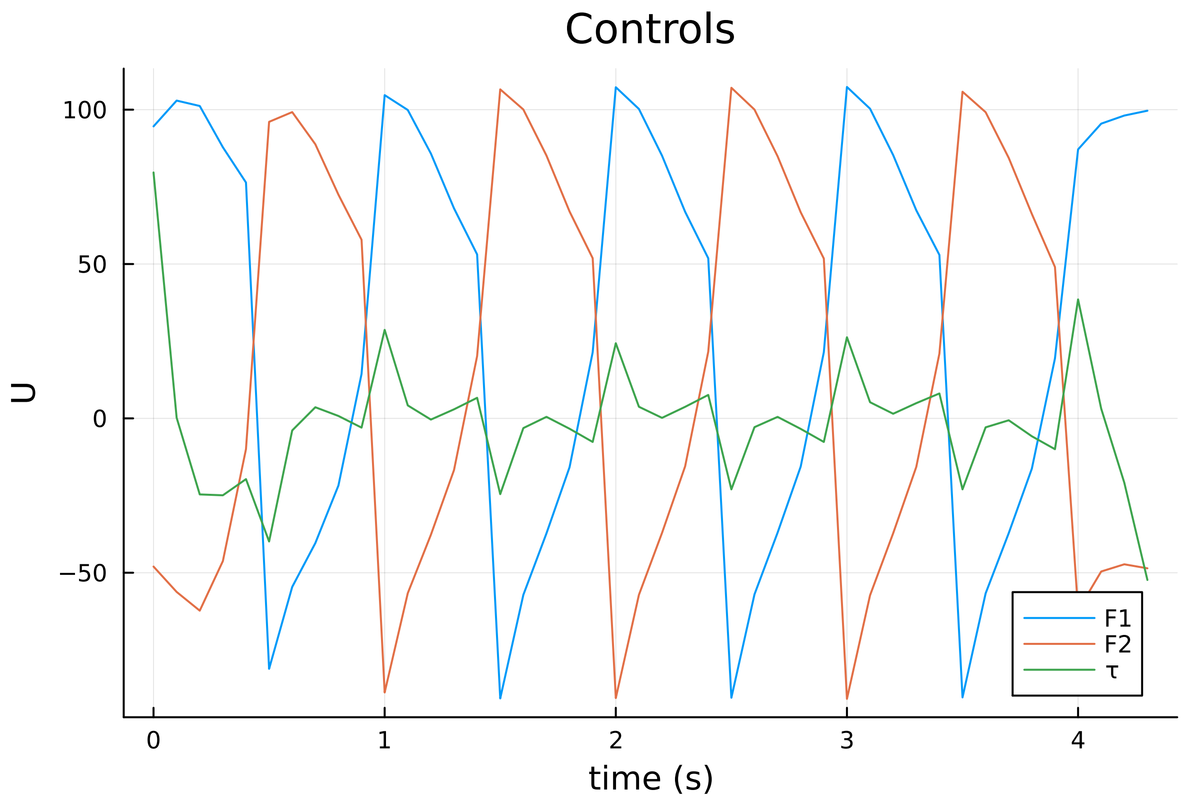
```
 Starting fmincon
---------checking dimensions of everything----------
---------all dimensions good------------------------
---------diff type set to :auto (ForwardDiff.jl)----
---------testing objective gradient-----------------
---------testing constraint Jacobian----------------
---------successfully compiled both derivatives-----
---------IPOPT beginning solve----------------------

******************************************************************************
*
This program contains Ipopt, a library for large-scale nonlinear optimizatio
n.
 Ipopt is released as open source code under the Eclipse Public License (EP
L).
         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  8.2799946e+00 1.50e+00 1.09e+01   0.0 0.00e+00    -  0.00e+00 0.00e+00
0
   1  8.1415358e+00 1.47e+00 1.30e+01  -0.1 1.19e+02    -  4.36e-02 1.94e-02h
1
   2  8.1404608e+00 1.47e+00 3.15e+04  -0.1 1.38e+02    -  8.24e-01 2.15e-04h
1
   3  8.1474975e+00 1.47e+00 2.45e+06   1.0 5.88e+02    -  5.07e-02 8.17e-04h
1
   4  4.4540316e+01 1.06e+00 1.22e+07  -0.5 9.83e+01    -  3.72e-01 3.40e-01h
1
   5  2.2526477e+02 8.19e-01 1.15e+07   0.5 2.29e+02    -  1.89e-01 4.15e-01h
1
   6  2.3653949e+02 7.69e-01 1.10e+07   2.1 1.57e+02    -  6.01e-01 6.32e-02h
1
   7  2.7346249e+02 5.03e-01 7.82e+06   2.6 5.20e+01    -  4.02e-02 3.45e-01f
1
   8  2.9455727e+02 3.64e-01 5.76e+06   2.6 4.43e+01    -  2.94e-01 2.98e-01f
1
   9  3.0571628e+02 2.88e-01 4.61e+06   2.6 2.88e+01    -  5.04e-02 2.09e-01f
```

```
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  10  3.2310534e+02 1.66e-01 2.72e+06   2.6 3.68e+01    -  4.83e-03 4.30e-01f
1
  11  3.2346875e+02 1.63e-01 2.67e+06   2.6 1.83e+01    -  5.17e-01 1.86e-02h
1
  12  3.3439395e+02 1.02e-01 1.70e+06   2.4 1.21e+01    -  1.00e+00 3.73e-01h
1
  13  3.5741202e+02 4.70e-02 1.34e+02   1.9 7.33e+00    -  1.00e+00 1.00e+00h
1
  14  3.4146333e+02 4.05e-03 6.25e+02  -4.1 1.10e+01    -  7.96e-01 9.42e-01f
1
  15  2.9914521e+02 9.46e-02 6.03e+03   0.7 4.25e+01    -  8.60e-01 1.00e+00f
1
  16  2.8100312e+02 9.16e-02 1.31e+01   0.3 2.40e+01    -  1.00e+00 1.00e+00h
1
  17  2.6259845e+02 3.16e-03 5.26e+01  -0.2 1.88e+01    -  9.44e-01 1.00e+00H
1
  18  2.7376753e+02 1.57e-03 2.32e+01  -0.3 3.28e+01    -  1.00e+00 1.00e+00H
1
  19  2.5173909e+02 1.07e-01 1.13e+01  -0.5 1.74e+01    -  9.67e-01 1.00e+00f
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  20  2.5177682e+02 1.35e-02 3.30e+00  -1.0 6.36e+00    -  9.99e-01 1.00e+00h
1
  21  2.5121455e+02 2.89e-03 3.37e+00  -1.2 5.98e+00    -  9.75e-01 1.00e+00h
1
  22  2.5197667e+02 5.15e-04 5.60e+00  -1.2 7.77e+00    -  9.75e-01 1.00e+00H
1
  23  2.4951503e+02 3.36e-03 3.11e+00  -1.5 4.38e+00    -  1.00e+00 1.00e+00f
1
  24  2.4933192e+02 1.49e-03 1.05e+00  -2.0 2.38e+00    -  9.94e-01 1.00e+00h
1
  25  2.4887481e+02 1.12e-03 1.38e+00  -2.3 2.98e+00    -  9.99e-01 1.00e+00f
1
  26  2.4871077e+02 3.98e-03 2.20e+00  -2.8 6.12e+00    -  1.00e+00 1.00e+00f
1
  27  2.4858211e+02 4.58e-03 3.97e+00  -3.0 6.50e+00    -  1.00e+00 6.74e-01f
1
  28  2.4832718e+02 2.33e-03 4.12e+00  -3.1 6.03e+00    -  1.00e+00 1.00e+00f
1
  29  2.4882073e+02 1.42e-04 1.68e+00  -3.3 2.63e+00    -  9.69e-01 1.00e+00H
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  30  2.4788312e+02 1.01e-03 3.20e-01  -3.6 1.91e+00    -  1.00e+00 1.00e+00f
1
  31  2.4785447e+02 7.39e-05 2.19e-01  -5.0 1.09e+00    -  1.00e+00 1.00e+00h
1
  32  2.4781515e+02 8.61e-04 1.49e+00  -6.0 7.16e+00    -  1.00e+00 1.00e+00f
1
  33  2.4780017e+02 6.27e-04 5.13e+01  -5.7 8.25e+00    -  1.00e+00 1.77e-01h
1
  34  2.4779151e+02 5.38e-04 8.11e-01  -4.8 1.82e+00    -  1.00e+00 9.99e-01f
1
```

```
  35  2.4774697e+02 2.40e-04 5.84e+00 -10.8 1.41e+00    -  8.17e-01 9.96e-01h
1
  36  2.4773775e+02 5.02e-05 1.84e-01  -6.7 1.41e+00    -  1.00e+00 1.00e+00h
1
  37  2.4779659e+02 7.29e-06 3.97e-01  -7.6 1.77e+00    -  1.00e+00 1.00e+00H
1
  38  2.4773401e+02 2.09e-04 1.16e-01  -8.3 1.01e+00    -  1.00e+00 1.00e+00f
1
  39  2.4773264e+02 1.16e-05 1.19e-01  -9.7 3.23e-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  2.4773083e+02 3.20e-06 5.97e-02 -11.0 2.39e-01    -  1.00e+00 1.00e+00h
1
  41  2.4773025e+02 1.44e-06 7.56e-02 -11.0 3.56e-01    -  1.00e+00 1.00e+00h
1
  42  2.4773296e+02 8.10e-08 2.08e-01  -9.4 8.19e-01    -  1.00e+00 1.00e+00H
1
  43  2.4772875e+02 2.07e-05 1.80e-01 -10.6 2.11e-01    -  1.00e+00 1.00e+00f
1
  44  2.4772835e+02 3.44e-06 6.59e-02 -11.0 1.62e-01    -  1.00e+00 1.00e+00h
1
  45  2.4772781e+02 3.28e-07 2.23e-02 -11.0 9.06e-02    -  1.00e+00 1.00e+00h
1
  46  2.4772777e+02 1.41e-07 1.41e-02 -11.0 2.05e-02    -  1.00e+00 1.00e+00h
1
  47  2.4772769e+02 2.92e-07 2.44e-02 -11.0 4.68e-02    -  1.00e+00 1.00e+00h
1
  48  2.4772847e+02 1.00e-08 1.08e-01 -11.0 1.83e-01    -  1.00e+00 1.00e+00H
1
  49  2.4772788e+02 3.41e-06 7.09e-02 -11.0 8.53e-02    -  1.00e+00 1.00e+00f
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  50  2.4772759e+02 9.94e-07 2.54e-02 -11.0 6.49e-02    -  1.00e+00 1.00e+00h
1
  51  2.4772796e+02 5.03e-07 3.51e-02 -11.0 4.19e-02    -  1.00e+00 1.00e+00h
1
  52  2.4772756e+02 2.91e-07 8.77e-04 -11.0 3.69e-02    -  1.00e+00 1.00e+00h
1
  53  2.4772756e+02 1.00e-08 9.44e-04 -11.0 1.75e-03    -  1.00e+00 1.00e+00h
1
  54  2.4772756e+02 1.00e-08 9.95e-03 -11.0 3.35e-02    -  1.00e+00 1.00e+00H
1
  55  2.4772774e+02 1.00e-08 2.34e-02 -11.0 9.97e-02    -  1.00e+00 1.00e+00H
1
  56  2.4772759e+02 5.46e-07 1.10e-02 -11.0 8.58e-02    -  1.00e+00 1.00e+00h
1
  57  2.4772790e+02 1.00e-08 2.69e-02 -11.0 9.53e-02    -  1.00e+00 1.00e+00H
1
  58  2.4772757e+02 6.94e-07 8.44e-03 -11.0 8.19e-02    -  1.00e+00 1.00e+00h
1
  59  2.4772760e+02 1.35e-07 1.57e-02 -11.0 3.15e-02    -  1.00e+00 1.00e+00h
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  60  2.4772756e+02 1.05e-07 5.67e-03 -11.0 2.64e-02    -  1.00e+00 1.00e+00h
```

```
1
  61  2.4772759e+02 1.00e-08 9.64e-03 -11.0 1.51e-02    -  1.00e+00 1.00e+00H
1
  62  2.4772754e+02 6.98e-08 9.15e-04 -11.0 1.21e-02    -  1.00e+00 1.00e+00h
1
  63  2.4772754e+02 1.00e-08 1.58e-04 -11.0 5.88e-04    -  1.00e+00 1.00e+00h
1

Number of Iterations....: 63

                                   (scaled)                 (unscaled)
Objective...............:   2.4772754442020545e+02    2.4772754442020545e+02
Dual infeasibility......:   1.5786008804968832e-04    1.5786008804968832e-04
Constraint violation....:   9.9999999930062278e-09    9.9999999930062278e-09
Complementarity.........:   1.0000000055174815e-11    1.0000000055174815e-11
Overall NLP error.......:   7.4932748834136459e-07    1.5786008804968832e-04


Number of objective function evaluations             = 81
Number of objective gradient evaluations             = 64
Number of equality constraint evaluations            = 81
Number of inequality constraint evaluations          = 81
Number of equality constraint Jacobian evaluations   = 64
Number of inequality constraint Jacobian evaluations = 64
Number of Lagrangian Hessian evaluations             = 0
Total CPU secs in IPOPT (w/o function evaluations)   =     15.542
Total CPU secs in NLP function evaluations           =     20.914

EXIT: Optimal Solution Found.
```






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

Out[9]:
```
Test.DefaultTestSet("walker trajectory optimization", Any[], 272, false, fals
e)
```

In [ ]:
```

```