In [68]:
```julia
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
import MathOptInterface as MOI
import Ipopt
import FiniteDiff
import ForwardDiff
import Convex as cvx
import ECOS
using LinearAlgebra
using Plots; plotly()
using Random
using JLD2
using Test
import MeshCat as mc
using Statistics
```

```
  Activating environment at `~/Dropbox/My Mac (MacBook Pro (2))/Desktop/CMU/Optimal Cont
rol/HW3_S23/Project.toml`
┌ Warning: backend `PlotlyBase` is not installed.
└ @ Plots ~/.julia/packages/Plots/B5j7d/src/backends.jl:43
┌ Warning: backend `PlotlyKaleido` is not installed.
└ @ Plots ~/.julia/packages/Plots/B5j7d/src/backends.jl:43
```

In [69]:
```julia
include(joinpath(@__DIR__, "utils","fmincon.jl"))
include(joinpath(@__DIR__, "utils","planar_quadrotor.jl"))
```

Out[69]:
```
check_dynamic_feasibility (generic function with 1 method)
```

Q3: Quadrotor Reorientation (40 pts)

In this problem, you will use the trajectory optimization tools you have demonstrated in questions one and two to solve for a collision free reorientation of three planar quadrotors. The planar quadrotor (as described in lecture 9) is described with the following state and dynamics:

$$x = \begin{bmatrix} p_x \\ p_z \\ \theta \\ v_x \\ v_z \\ \omega \end{bmatrix}, \qquad (1) \dot{x} =$$

where p_x and p_z are the horizontal and vertial positions, v_x and v_z are the corresponding velocities, θ for orientation, ω for angular velocity, ℓ for length of the quadrotor, m for mass, g for gravity acceleration in the $-z$ direction, and a moment of inertia of J.

You are free to use any solver/cost/constraint you would like to solve for three collision free, dynamically feasible trajectories for these quadrotors that looks something like the following:

(if an animation doesn't load here, check out `quadrotor_reorient.gif`.)

Here are the performance requirements that the resulting trajectories must meet:

- The three quadrotors must start at `x1ic`, `x2ic`, and `x2ic` as shown in the code (these are the initial conditions).

- The three quadrotors must finish their trajectories within **.2** meters of `x1g`, `x2g`, and `x2g` (these are the goal states).

- The three quadrotors must never be within **0.8** meters of one another (use $[p_x, p_z]$ for this).

There are two main ways of going about this:

1. **Cost Shaping**: Design cost functions for each quadrotor that motivates them to take paths that do not result in a collision. You can do something like designing a reference trajectory for each quadrotor to use in the cost. You can use iLQR or DIRCOL for this.

2. **Collision Constraints**: You can optimize over all three quadrotors at once by creating a new state $\tilde{x} = [x_1^T, x_2^T, x_3^T]^T$ and control $\tilde{u} = [u_1^T, u_2^T, u_3^T]^T$, and then directly include collision avoidance constraints. In order to use constraints, you must use DIRCOL (at least for now).

Hints

- You should not use `norm() >= R` in any constraints, instead you should square the constraint to be `norm()^2 >= R^2`. This second constraint is still non-convex, but it is differentiable everywhere.

- If you are using DIRCOL, you can initialize the solver with a "guess" solution by linearly interpolating between the initial and terminal conditions. Julia let's you create a length N linear interpolated vector of vectors between `a::Vector` and `b::Vector` like this: `range(a, b, length = N)` (experiment with this to see how it works).

You can use either RK4 (iLQR or DIRCOL) or Hermite-Simpson (DIRCOL) for your integration. The `dt = 0.2`, and `tf = 5.0` are given for you in the code (you may change these but only if you feel you really have to).

In [70]:
```julia
function single_quad_dynamics(params, x,u)
    # planar quadrotor dynamics for a single quadrotor

    # unpack state
    px,pz,θ,vx,vz,ω = x

    xdot = [
        vx,
        vz,
        ω,
        (1/params.mass)*(u[1] + u[2])*sin(θ),
        (1/params.mass)*(u[1] + u[2])*cos(θ) - params.g,
        (params.ℓ/(2*params.J))*(u[2]-u[1])
    ]

    return xdot
end
function combined_dynamics(params, x, u)
    # dynamics for three planar quadrotors, assuming the state is stacked
    # in the following manner: x = [x1;x2;x3]
```

```julia
    # NOTE: you would only need to use this if you chose option 2 where
    # you optimize over all three trajectories simultaneously

    # quadrotor 1
    x1 = x[1:6]
    u1 = u[1:2]
    xdot1 = single_quad_dynamics(params, x1, u1)

    # quadrotor 2
    x2 = x[(1:6) .+ 6]
    u2 = u[(1:2) .+ 2]
    xdot2 = single_quad_dynamics(params, x2, u2)

    # quadrotor 3
    x3 = x[(1:6) .+ 12]
    u3 = u[(1:2) .+ 4]
    xdot3 = single_quad_dynamics(params, x3, u3)

    # return stacked dynamics
    return [xdot1;xdot2;xdot3]
end

function hermite_simpson(params::NamedTuple, x1::Vector, x2::Vector, u, dt::Real)::Vecto
    # TODO: input hermite simpson implicit integrator residual
    dx1 = combined_dynamics(params, x1, u)
    dx2 = combined_dynamics(params, x2, u)
    x12 = (1/2)*(x1 + x2) + (dt/8)*(dx1 - dx2)
    dx12 = combined_dynamics(params, x12, u)
    f = (x1 + (dt/6)*(dx1 + 4*dx12 + dx2) - x2)
    return f

end
```

Out[70]: hermite_simpson (generic function with 1 method)

In [71]:

```julia
function create_idx(nx,nu,N)
    # This function creates some useful indexing tools for Z
    # x_i = Z[idx.x[i]]
    # u_i = Z[idx.u[i]]

    # Feel free to use/not use anything here.


    # 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

"""
    quadrotor_reorient

Function for returning collision free trajectories for 3 quadrotors.
```

```
Outputs:
    x1::Vector{Vector}  # state trajectory for quad 1
    x2::Vector{Vector}  # state trajectory for quad 2
    x3::Vector{Vector}  # state trajectory for quad 3
    u1::Vector{Vector}  # control trajectory for quad 1
    u2::Vector{Vector}  # control trajectory for quad 2
    u3::Vector{Vector}  # control trajectory for quad 3
    t_vec::Vector
    params::NamedTuple

The resulting trajectories should have dt=0.2, tf = 5.0, N = 26
where all the x's are length 26, and the u's are length 25.

Each trajectory for quad k should start at `xkic`, and should finish near
`xkg`. The distances between each quad should be greater than 0.8 meters at
every knot point in the trajectory.
"""
function quadrotor_cost(params::NamedTuple, Z::Vector)::Real
    idx, N = params.idx, params.N
    x1g, x2g, x3g = params.x1g, params.x2g, params.x3g
    Q, R, Qf = params.Q, params.R, params.Qf

    xg = [x1g; x2g; x3g]

    J = 0
    for i = 1:(N-1)
        xi = Z[idx.x[i]]
        ui = Z[idx.u[i]]
        xi_tilde = xi - xg

        J += 0.5*xi_tilde'*Q*xi_tilde + 0.5*ui'*R*ui
    end

    #Terminal cost
    xf = Z[idx.x[end]]
    J += 0.5*xf'*Qf*xf

    return J
end

function quadrotor_dynamics_constraints(params::NamedTuple, Z::Vector)::Vector
    idx, N, dt = params.idx, params.N, params.dt

    c = zeros(eltype(Z), idx.nc)
    for i = 1:(N-1)
        xi = Z[idx.x[i]]
        ui = Z[idx.u[i]]
        xip1 = Z[idx.x[i+1]]

        c[idx.c[i]] = hermite_simpson(params, xi, xip1, ui, dt)
    end
    return c
end

function quadrotor_collision_constraints(params::NamedTuple, Z::Vector)::Vector
    N, idx = params.N, params.idx

    c12 = zeros(eltype(Z), idx.nc)
    c13 = zeros(eltype(Z), idx.nc)
    c23 = zeros(eltype(Z), idx.nc)

    for i = 1:(N)
```

```julia
        xi = Z[idx.x[i]]
        x1i = xi[1:2]
        x2i = xi[7:8]
        x3i = xi[13:14]
        c12[i] = norm(x1i - x2i)^2
        c13[i] = norm(x1i - x3i)^2
        c23[i] = norm(x2i - x3i)^2
    end

    return [c12; c13; c23]

end

function quadrotor_equality_constraint(params::NamedTuple, Z::Vector)::Vector
    N, idx = params.N, params.idx
    x1ic, x2ic, x3ic = params.x1ic, params.x2ic, params.x3ic
    x1g, x2g, x3g = params.x1g, params.x2g, params.x3g

    c_dynamics = quadrotor_dynamics_constraints(params,Z) #dynamics constraints

    xic = [x1ic; x2ic; x3ic]
    c_ic = Z[idx.x[1]] - xic #initial position constraints

    xg = [x1g; x2g; x3g]
    c_g = Z[idx.x[N]] - xg #goal position constraints

    return [c_dynamics; c_ic; c_g] # 10 is an arbitrary number
end

function quadrotor_reorient(;verbose=true)

    # problem size
    nx = 18
    nu = 6
    dt = 0.2
    tf = 5.0
    t_vec = 0:dt:tf
    N = length(t_vec)

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

    # initial conditions and goal states
    lo = 0.5
    mid = 2
    hi = 3.5
    x1ic = [-2,lo,0,0,0,0]  # ic for quad 1
    x2ic = [-2,mid,0,0,0,0] # ic for quad 2
    x3ic = [-2,hi,0,0,0,0]  # ic for quad 3

    x1g = [2,mid,0,0,0,0]   # goal for quad 1
    x2g = [2,hi,0,0,0,0]    # goal for quad 2
    x3g = [2,lo,0,0,0,0]    # goal for quad 3

    # load all useful things into params
    # TODO: include anything you would need for a cost function (like a Q, R, Qf if you
    # LQR cost)
    Q1 = [1;1;1;1;1;1]
    Q2 = [1;1;1;1;1;1]
    Q3 = [1;1;1;1;1;1]
    Q = diagm([Q1; Q2; Q3])
```

```
    Qf = 10*diagm(ones(nx))

    R1 = [1;1]
    R2 = [1;1]
    R3 = [1;1]
    R = 0.1*diagm([R1; R2; R3])

    params = (x1ic=x1ic,
              x2ic=x2ic,
              x3ic=x3ic,
              x1g = x1g,
              x2g = x2g,
              x3g = x3g,
              dt = dt,
              N = N,
              idx = idx,
              Q = Q,
              Qf = Qf,
              R = R,
              mass = 1.0, # quadrotor mass
              g = 9.81,   # gravity
              ℓ = 0.3,    # quadrotor length
              J = .018)   # quadrotor moment of inertia

    # TODO: solve for the three collision free trajectories however you like
    # TODO: primal bounds
    x_l = -Inf*ones(idx.nz)
    x_u =  Inf*ones(idx.nz)

    # inequality constraint bounds (this is what we do when we have no inequality constr
    r = 0.9
    c_l = (r^2)*ones(3*idx.nc)
    c_u = Inf*ones(3*idx.nc)

    # initial guess
    x1_0 = range(x1ic, x1g, length = N)
    x2_0 = range(x2ic, x2g, length = N)
    x3_0 = range(x3ic, x3g, length = N)
    u0 = [ones(idx.nu) for i = 1:N-1]
    z0 = zeros(idx.nz)
    for i = 1:(N-1)
        z0[idx.x[i]] = [x1_0[i]; x2_0[i]; x3_0[i]]
        z0[idx.u[i]] = u0[i]
    end
    z0[idx.x[N]] = [x1_0[N]; x2_0[N]; x3_0[N]]

    # choose diff type (try :auto, then use :finite if :auto doesn't work)
    diff_type = :auto
#     diff_type = :finite


    Z = fmincon(quadrotor_cost,quadrotor_equality_constraint,quadrotor_collision_constra
                x_l,x_u,c_l,c_u,z0,params, diff_type;
                tol = 1e-6, c_tol = 1e-6, max_iters = 10_000, verbose = verbose)

    # 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)]
    # return the trajectories
    x1 = [X[i][1:6] for i = 1:N]
    x2 = [X[i][7:12] for i = 1:N]
    x3 = [X[i][13:18] for i = 1:N]
```

```julia
    u1 = [U[i][1:2] for i = 1:(N-1)]
    u2 = [U[i][3:4] for i = 1:(N-1)]
    u3 = [U[i][5:6] for i = 1:(N-1)]

    return x1, x2, x3, u1, u2, u3, t_vec, params
end
```

Out[71]: quadrotor_reorient (generic function with 1 method)

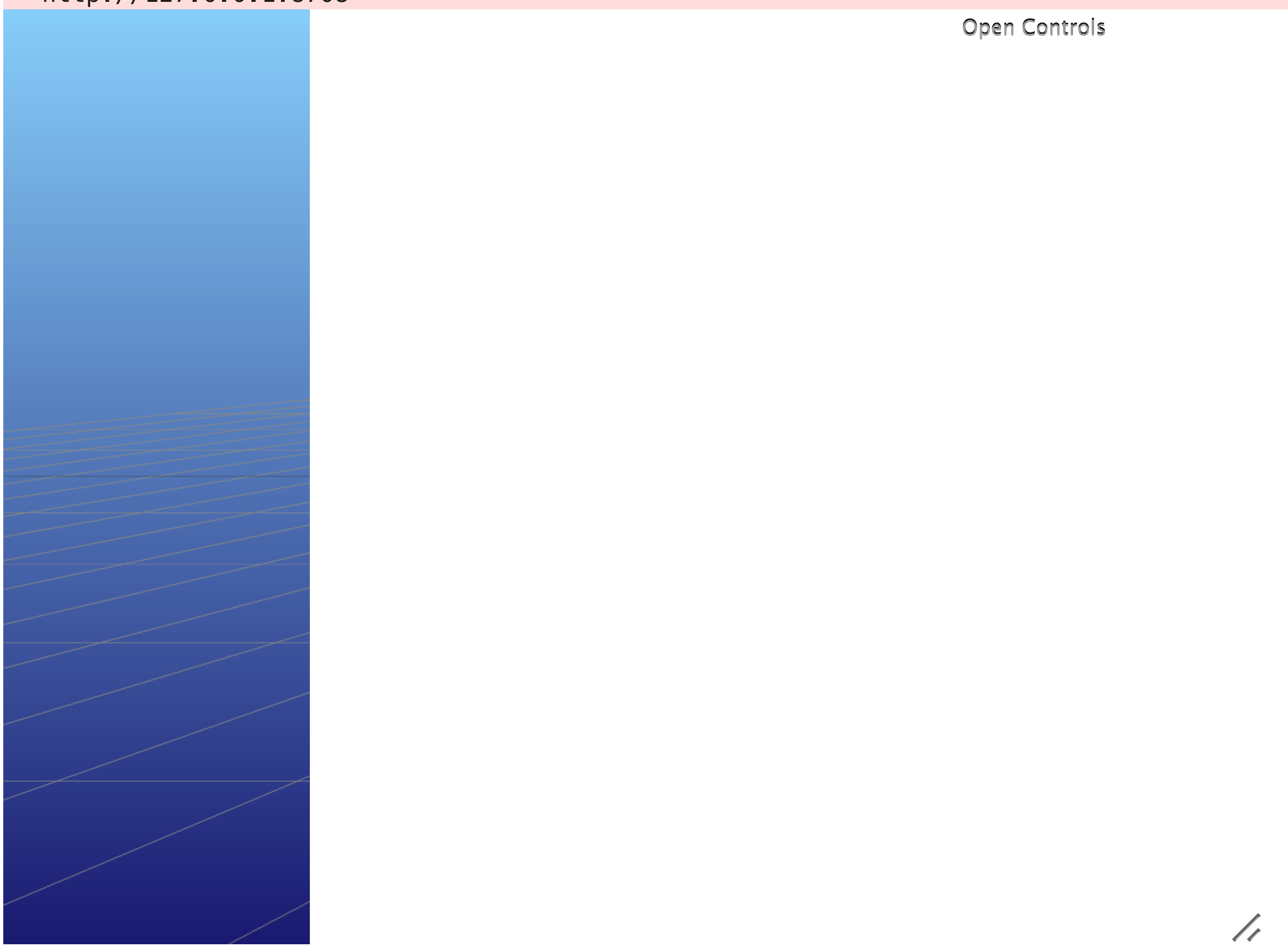
In [72]:

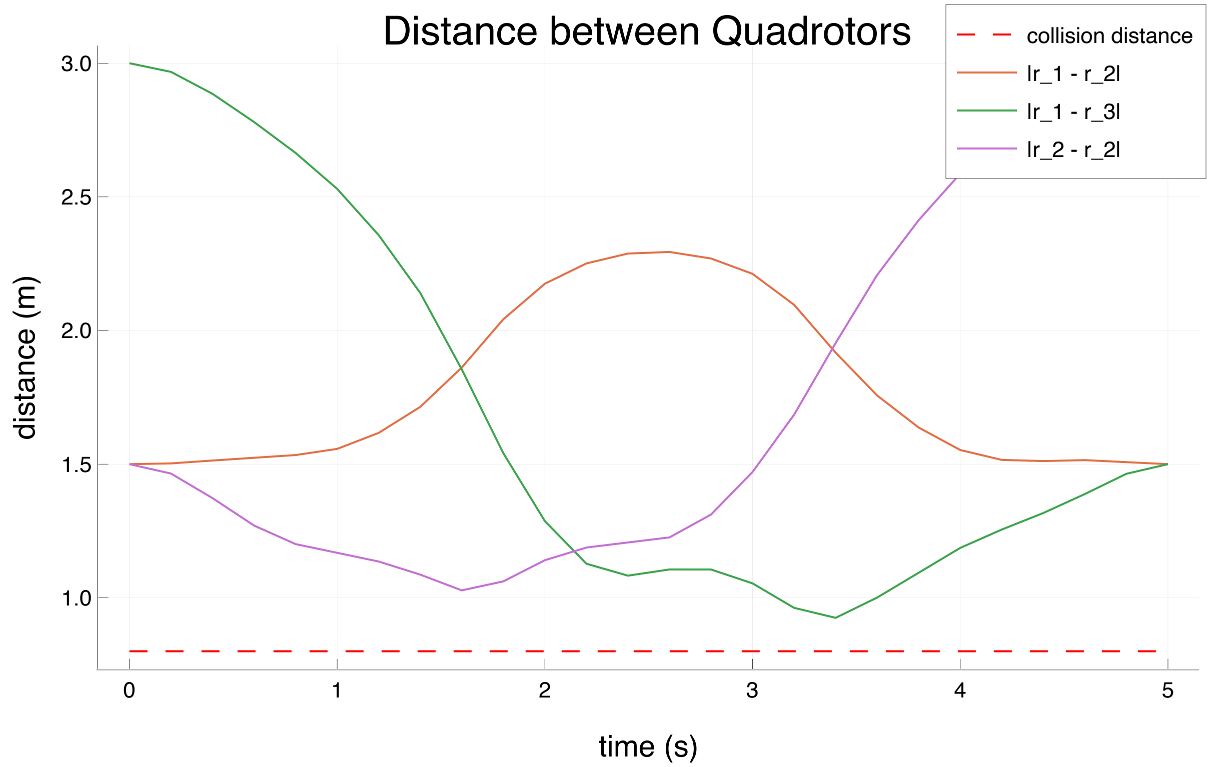
```julia
@testset "quadrotor reorient" begin

    X1, X2, X3, U1, U2, U3, t_vec, params  = quadrotor_reorient(verbose=true)


    #---------------testing-----------------
    # check lengths of everything
    @test length(X1) == length(X2) == length(X3)
    @test length(U1) == length(U2) == length(U3)
    @test length(X1) == params.N
    @test length(U1) == (params.N-1)

    # check for collisions
    distances = [distance_between_quads(x1[1:2],x2[1:2],x3[1:2]) for (x1,x2,x3) in zip(X
    @test minimum(minimum.(distances)) >= 0.799

    # check initial and final conditions
    @test norm(X1[1] - params.x1ic, Inf) <= 1e-3
    @test norm(X2[1] - params.x2ic, Inf) <= 1e-3
    @test norm(X3[1] - params.x3ic, Inf) <= 1e-3
    @test norm(X1[end] - params.x1g, Inf) <= 2e-1
    @test norm(X2[end] - params.x2g, Inf) <= 2e-1
    @test norm(X3[end] - params.x3g, Inf) <= 2e-1

    # check dynamic feasibility
    @test check_dynamic_feasibility(params,X1,U1)
    @test check_dynamic_feasibility(params,X2,U2)
    @test check_dynamic_feasibility(params,X3,U3)


    #---------------plotting/animation-------
    display(animate_planar_quadrotors(X1,X2,X3, params.dt))

    plot(t_vec, 0.8*ones(params.N),ls = :dash, color = :red, label = "collision distance
         xlabel = "time (s)", ylabel = "distance (m)", title = "Distance between Quadrot
    display(plot!(t_vec, hcat(distances...)', label = ["|r_1 - r_2|" "|r_1 - r_3|" "|r_2

    X1m = hcat(X1...)
    X2m = hcat(X2...)
    X3m = hcat(X3...)

    plot(X1m[1,:], X1m[2,:], color = :red,title = "Quadrotor Trajectories", label = "qua
    plot!(X2m[1,:], X2m[2,:], color = :green, label = "quad 2",xlabel = "p_x", ylabel = 
    display(plot!(X3m[1,:], X3m[2,:], color = :blue, label = "quad 3"))

    plot(t_vec, X1m[3,:], color = :red,title = "Quadrotor Orientations", label = "quad 1
    plot!(t_vec, X2m[3,:], color = :green, label = "quad 2",xlabel = "time (s)", ylabel 
    display(plot!(t_vec, X3m[3,:], color = :blue, label = "quad 3"))


end
```

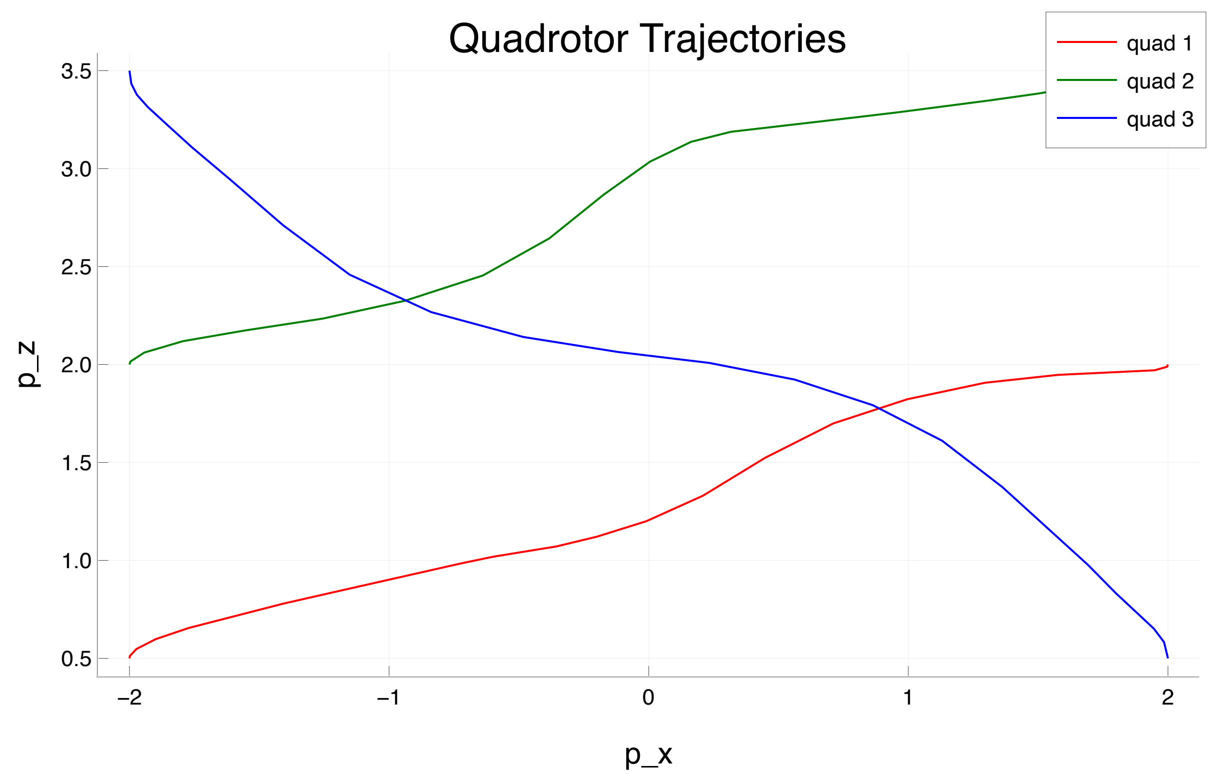
```
---------checking dimensions of everything----------
---------all dimensions good-----------------------
---------diff type set to :auto (ForwardDiff.jl)----
---------testing objective gradient-----------------
---------testing constraint Jacobian----------------
---------successfully compiled both derivatives-----
---------IPOPT beginning solve----------------------
This is Ipopt version 3.14.4, running with linear solver MUMPS 5.4.1.

Number of nonzeros in equality constraint Jacobian...:   300348
Number of nonzeros in inequality constraint Jacobian.:   834300
Number of nonzeros in Lagrangian Hessian.............:        0

Total number of variables............................:      618
                     variables with only lower bounds:        0
                variables with lower and upper bounds:        0
                     variables with only upper bounds:        0
Total number of equality constraints.................:      486
Total number of inequality constraints...............:     1350
        inequality constraints with only lower bounds:     1350
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  4.2183000e+02 1.56e+00 1.37e+00   0.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  4.2227964e+02 1.54e+00 3.29e+00  -7.0 5.32e+00    -  5.27e-02 1.21e-02h  1
   2  4.2223163e+02 1.54e+00 1.56e+02  -7.0 5.90e+01    -  3.71e-03 1.22e-04h  1
   3r 4.2223163e+02 1.54e+00 9.98e+02   0.5 0.00e+00    -  0.00e+00 3.06e-07R  3
   4r 4.7673785e+02 1.30e+00 9.90e+02   0.7 1.92e+02    -  3.61e-03 8.25e-03f  1
   5  4.7659642e+02 1.30e+00 1.56e+02  -7.0 1.79e+01    -  4.58e-02 2.91e-04h  1
   6r 4.7659642e+02 1.30e+00 9.99e+02   0.1 0.00e+00    -  0.00e+00 3.64e-07R  4
   7r 4.8127138e+02 1.27e+00 9.98e+02  -6.0 2.21e+02    -  5.25e-03 5.25e-04f  1
   8r 6.1149318e+02 8.10e-01 9.97e+02  -0.6 5.09e+03    -  8.43e-03 1.19e-03f  1
   9r 6.3350092e+02 8.10e-01 9.93e+02   0.8 2.15e+02    -  8.77e-03 3.58e-03f  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  10r 6.4768590e+02 8.10e-01 9.88e+02   1.0 1.20e+02    -  2.71e-03 5.09e-03f  1
  11r 6.6321942e+02 8.10e-01 9.79e+02   0.8 2.43e+01    -  2.97e-02 8.96e-03f  1
  12r 7.0442071e+02 8.10e-01 9.31e+02   0.3 1.04e+01    -  1.49e-01 4.88e-02f  1
  13r 7.0472240e+02 8.10e-01 7.83e+02  -0.6 2.09e+00    -  7.43e-01 1.61e-01f  1
  14r 6.8146185e+02 8.10e-01 1.39e+02  -0.7 2.86e+00    -  5.83e-01 8.22e-01f  1
  15r 6.7286667e+02 8.10e-01 3.64e+01  -6.5 1.89e+00    -  6.09e-01 8.97e-01f  1
  16r 6.7286667e+02 8.10e-01 9.93e+02   0.7 0.00e+00    -  0.00e+00 2.52e-07R  6
  17r 6.7408257e+02 8.10e-01 9.92e+02   1.1 2.28e+01    -  7.17e-02 7.76e-03f  2
  18r 6.7676817e+02 8.10e-01 9.11e+02   0.6 7.22e+00    -  1.18e-01 8.14e-02f  1
  19r 6.7674405e+02 8.10e-01 7.35e+02  -1.2 1.43e-01    -  9.26e-01 1.93e-01f  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  20r 6.7674781e+02 8.10e-01 5.11e+01  -5.7 2.36e-02    -  9.81e-01 9.30e-01f  1
  21r 6.7689803e+02 8.10e-01 3.27e-01  -2.9 1.02e-01    -  1.00e+00 9.94e-01f  1
  22r 6.7681683e+02 8.10e-01 1.25e-03  -4.6 2.84e-02    -  1.00e+00 9.98e-01f  1
  23r 6.7680617e+02 8.10e-01 2.60e-04  -9.0 1.60e-03    -  1.00e+00 1.00e+00f  1
  24r 6.7418961e+02 8.10e-01 6.82e-06  -9.0 4.45e-01    -  1.00e+00 1.00e+00h  1
  25r 6.7356763e+02 8.10e-01 4.41e-07  -8.7 8.52e-02    -  1.00e+00 1.00e+00h  1
  26r 6.7346544e+02 8.10e-01 3.29e-07  -9.0 4.98e-02    -  1.00e+00 1.00e+00h  1
  27r 6.7336336e+02 8.10e-01 1.35e-07  -9.0 2.54e-02    -  1.00e+00 1.00e+00h  1
  28r 6.7335440e+02 8.10e-01 1.36e-07  -9.0 2.94e-02    -  1.00e+00 1.00e+00h  1

Number of Iterations....: 28

                                   (scaled)                 (unscaled)
Objective...............:   6.7340024460610800e+02    6.7340024460610800e+02
Dual infeasibility......:   3.5000000000003375e+01    3.5000000000003375e+01
Constraint violation....:   8.0999999000000000e-01    8.0999999000000000e-01
```

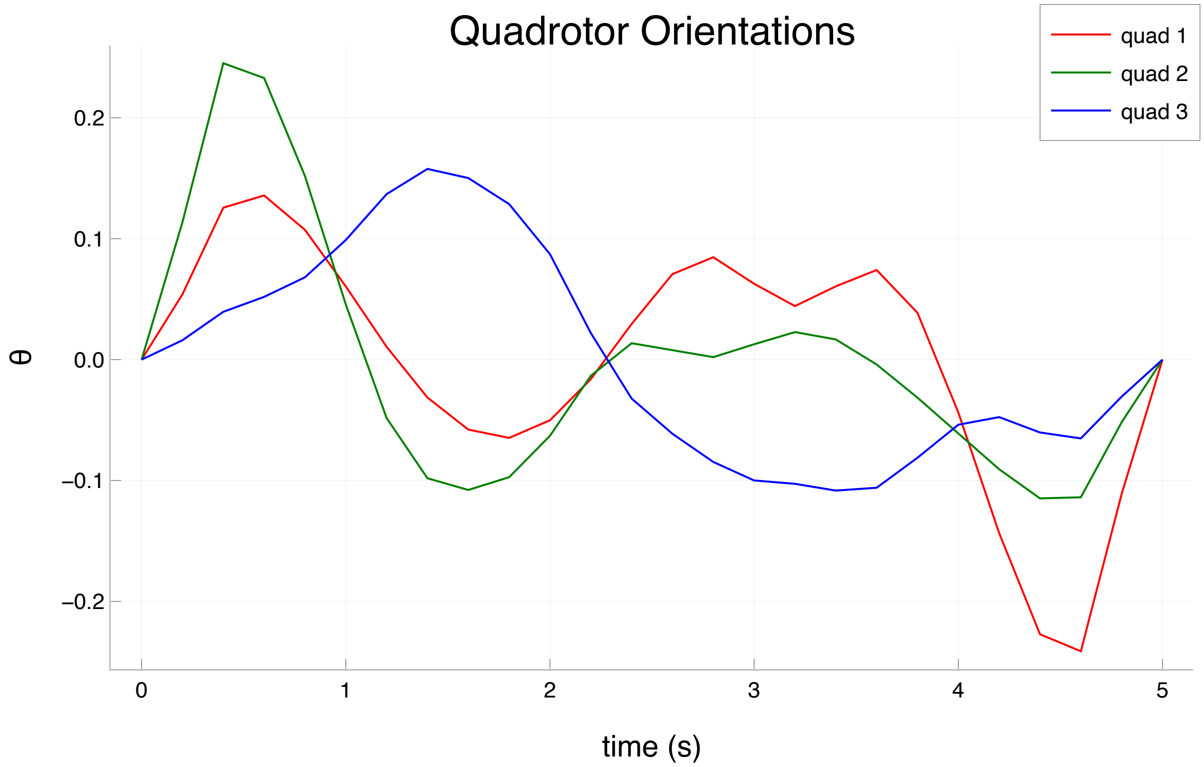
```
Variable bound violation:   0.0000000000000000e+00    0.0000000000000000e+00
Complementarity.........:   9.9999999999999986e-10    9.9999999999999986e-10
Overall NLP error.......:   4.3832547169003808e+00    3.5000000000003375e+01


Number of objective function evaluations             = 46
Number of objective gradient evaluations             = 10
Number of equality constraint evaluations            = 46
Number of inequality constraint evaluations          = 46
Number of equality constraint Jacobian evaluations   = 33
Number of inequality constraint Jacobian evaluations = 33
Number of Lagrangian Hessian evaluations             = 0
Total seconds in IPOPT                               = 18.648

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://127.0.0.1:8708
```




Distance between Quadrotors
collision distance
lr_1 - r_2l
lr_1 - r_3l
lr_2 - r_2l
distance (m)
time (s)


Quadrotor Trajectories
quad 1
quad 2
quad 3
p_z
p_x




```
Test Summary:      | Pass  Total
quadrotor reorient |   14     14
```

Out[72]: `Test.DefaultTestSet("quadrotor reorient", Any[], 14, false, false)`

In [ ]:

In [ ]: