In [1]:
```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
using Random
using JLD2
using Test
import MeshCat as mc
using Statistics
```

```
  Activating environment at `C:\Users\rdesa\OneDrive\Desktop\OCRL_HW3\HW3_S23-main\Project.toml`
```

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

Out[2]:
```
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:

$$x = \begin{bmatrix} p_x \\ p_z \\ \theta \\ v_x \\ v_z \\ \omega \end{bmatrix}, \qquad \dot{x} = \begin{bmatrix} v_x \\ v_z \\ \omega \\ \frac{1}{m}(u_1 + u_2)\sin\theta \\ \frac{1}{m}(u_1 + u_2)\cos\theta \\ \frac{\ell}{2J}(u_2 - u_1) \end{bmatrix}$$

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 [3]:
```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]

    # 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
```

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

In [4]:
```
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
```

Out[4]: create_idx (generic function with 1 method)

In [5]:
```
#Dircol
```

In [6]:
```
#integrator (rk4 or hs)
function hermite_simpson(params::NamedTuple, x1::Vector, x2::Vector, u, dt::Real)::Vector
    x1dot = single_quad_dynamics(params,x1,u)
    x2dot = single_quad_dynamics(params,x2,u)
    xk = (0.5*(x1+x2))+((dt/8).*(x1dot-x2dot))
    xkdot = single_quad_dynamics(params,xk,u)
    residuals = x1 + ((dt/6).*(x1dot+(4*xkdot)+x2dot)) - x2
end
```

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

In [7]:
```julia
#cost
function quadrotor_cost(params::NamedTuple, Z::Vector)::Real
    idx, N, xg = params.idx, params.N, params.xg
    Q, R, Qf = params.Q, params.R, params.Qf

    J = 0
    for i = 1:(N-1)
        xi = Z[idx.x[i]]
        ui = Z[idx.u[i]]
        x_d = (xi-xg)
        J += 0.5*(x_d'*Q*x_d) + 0.5*(ui'*R*ui)
    end

    x_T = (Z[idx.x[N]]-xg)
    J += 0.5*(x_T'*Qf*x_T)

    return J
end
```

Out[7]: quadrotor_cost (generic function with 1 method)

In [8]:
```julia
#dynamic constraint
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
```

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

In [9]:
```julia
#equality contraint
function quadrotor_equality_constraint(params::NamedTuple, Z::Vector)::Vector
    N, idx, xic, xg = params.N, params.idx, params.xic, params.xg
    c = quadrotor_dynamics_constraints(params, Z)
    ceq = Z[idx.x[1]] - xic
    ceq2 = Z[idx.x[N]] - xg
    return [ceq; ceq2; c]
end
```

Out[9]: quadrotor_equality_constraint (generic function with 1 method)

In [10]:
```
#solve
function solve_quadrotor_trajectory1(;verbose=true)

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

    # LQR cost
    Q = diagm(ones(nx))
    R = 0.1*diagm(ones(nu))
    Qf = 10*diagm(ones(nx))

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

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

    # TODO: primal bounds
    x_l = -Inf*ones(idx.nz)
    x_u =  Inf*ones(idx.nz)
```

```julia
    # inequality constraint bounds (this is what we do when we have no inequal
ity constraints)
    c_l = zeros(0)
    c_u = zeros(0)
    function inequality_constraint(params, Z)
        return zeros(eltype(Z), 0)
    end

    # initial guess
    z0 = 0.001*randn(idx.nz)

    # 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,inequality_constr
aint,
                x_l,x_u,c_l,c_u,z0,params, diff_type;
                tol = 1e-6, c_tol = 1e-6, max_iters = 10_000, verbose = verbos
e)

    # pull the X and U solutions out of Z
    x1 = [Z[idx.x[i]] for i = 1:N]
    u1 = [Z[idx.u[i]] for i = 1:(N-1)]

    return x1, u1, t_vec, params
end
```

Out[10]:
```
solve_quadrotor_trajectory1 (generic function with 1 method)
```

In [11]:

```
#solve
function solve_quadrotor_trajectory2(;verbose=true)

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

    # LQR cost
    Q = diagm(ones(nx))
    R = 0.1*diagm(ones(nu))
    Qf = 10*diagm(ones(nx))

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

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

    # 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 inequal
ity constraints)
    c_l = zeros(0)
    c_u = zeros(0)
    function inequality_constraint(params, Z)
        return zeros(eltype(Z), 0)
    end

    # initial guess
    z0 = 0.001*randn(idx.nz)

    # 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,inequality_constr
aint,
                x_l,x_u,c_l,c_u,z0,params, diff_type;
                tol = 1e-6, c_tol = 1e-6, max_iters = 10_000, verbose = verbos
e)

    # pull the X and U solutions out of Z
    x2 = [Z[idx.x[i]] for i = 1:N]
    u2 = [Z[idx.u[i]] for i = 1:(N-1)]

    return x2, u2, t_vec, params
end
```

Out[11]:

```
solve_quadrotor_trajectory2 (generic function with 1 method)
```

In [12]:
```
#solve
function solve_quadrotor_trajectory3(;verbose=true)

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

    # LQR cost
    Q = diagm(ones(nx))
    R = 0.1*diagm(ones(nu))
    Qf = 10*diagm(ones(nx))

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

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

    # 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 inequal
ity constraints)
    c_l = zeros(0)
    c_u = zeros(0)
    function inequality_constraint(params, Z)
        return zeros(eltype(Z), 0)
    end

    # initial guess
    z0 = 0.001*randn(idx.nz)

    # 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,inequality_constr
aint,
                x_l,x_u,c_l,c_u,z0,params, diff_type;
                tol = 1e-6, c_tol = 1e-6, max_iters = 10_000, verbose = verbos
e)

    # pull the X and U solutions out of Z
    x3 = [Z[idx.x[i]] for i = 1:N]
    u3 = [Z[idx.u[i]] for i = 1:(N-1)]

    return x3, u3, t_vec, params
end
```

Out[12]: solve_quadrotor_trajectory3 (generic function with 1 method)

In [13]:
```
#function create_reference(xic,xg,N, dt)
#    # create reference trajectory for quadrotor
#    R = 6
#    Xref = [ [R*cos(t);R*cos(t)*sin(t);1.2 + sin(t);zeros(9)] for t = range(-
pi/2,3*pi/2, length = N)]
#    for i = 1:(N-1)
#        Xref[i][4:6] = (Xref[i+1][1:3] - Xref[i][1:3])/dt
#    end
#    Xref[N][4:6] = Xref[N-1][4:6]
#    Uref = [(9.81*0.5/4)*ones(4) for i = 1:(N-1)]
#    return Xref, Uref
#end
```

In [14]:
```julia
"""
    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_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 were doing an
    # LQR cost)
    params = (x1ic=x1ic,
              x2ic=x2ic,
              x3ic=x3ic,
              x1g = x1g,
              x2g = x2g,
```

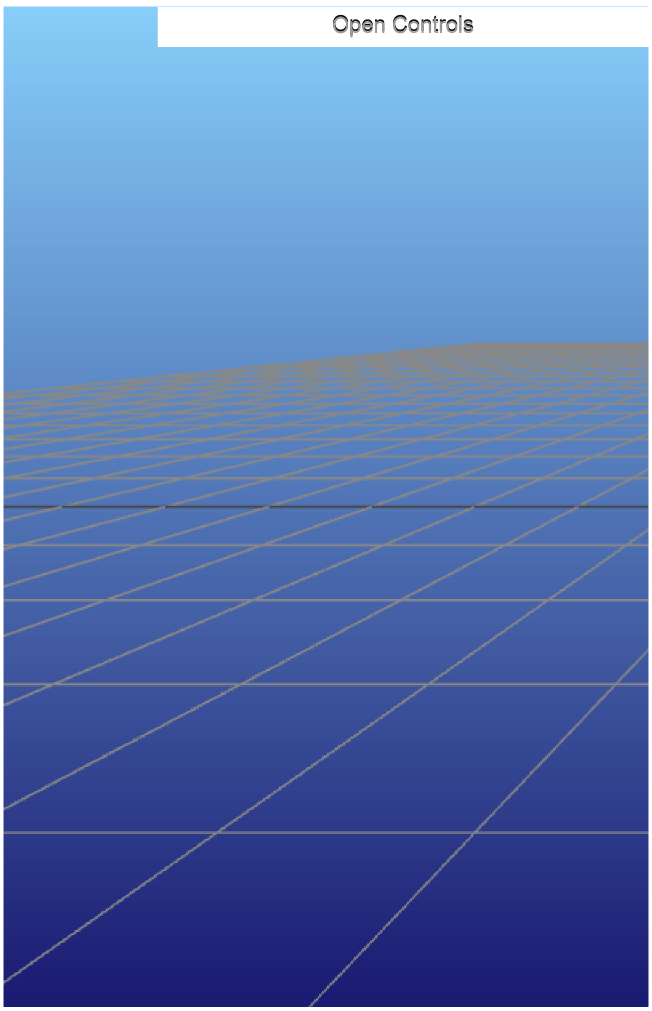
```julia
                x3g = x3g,
                dt = dt,
                N = N,
                idx = idx,
                mass = 1.0, # quadrotor mass
                g = 9.81,   # gravity
                ℓ = 0.3,    # quadrotor length
                J = .018)   # quadrotor moment of inertia

    # TODO: solve for the three collision free trajectories however you like
    #x1, u1, k1, t_vec, params =  solve_quadrotor_trajectory1(verbose = false)
    x1, u1, t_vec, params =  solve_quadrotor_trajectory1(verbose = false)
    x2, u2, t_vec, params =  solve_quadrotor_trajectory2(verbose = false)
    x3, u3, t_vec, params =  solve_quadrotor_trajectory3(verbose = false)

    # return the trajectories
    #x1 = [zeros(6)  for _ = 1:N]
    #x2 = [zeros(6)  for _ = 1:N]
    #x3 = [zeros(6)  for _ = 1:N]
    #u1 = [zeros(2)  for _ = 1:(N-1)]
    #u2 = [zeros(2)  for _ = 1:(N-1)]
    #u3 = [zeros(2)  for _ = 1:(N-1)]

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

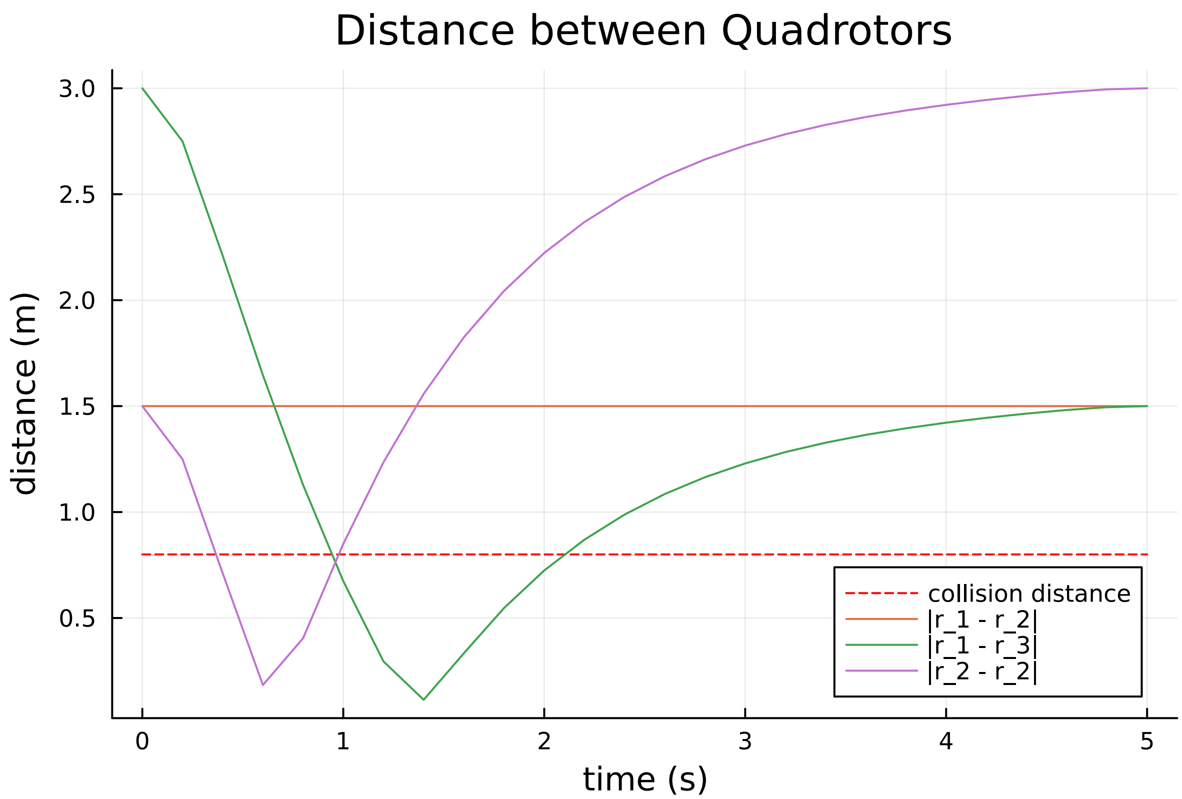
Out[14]: quadrotor_reorient

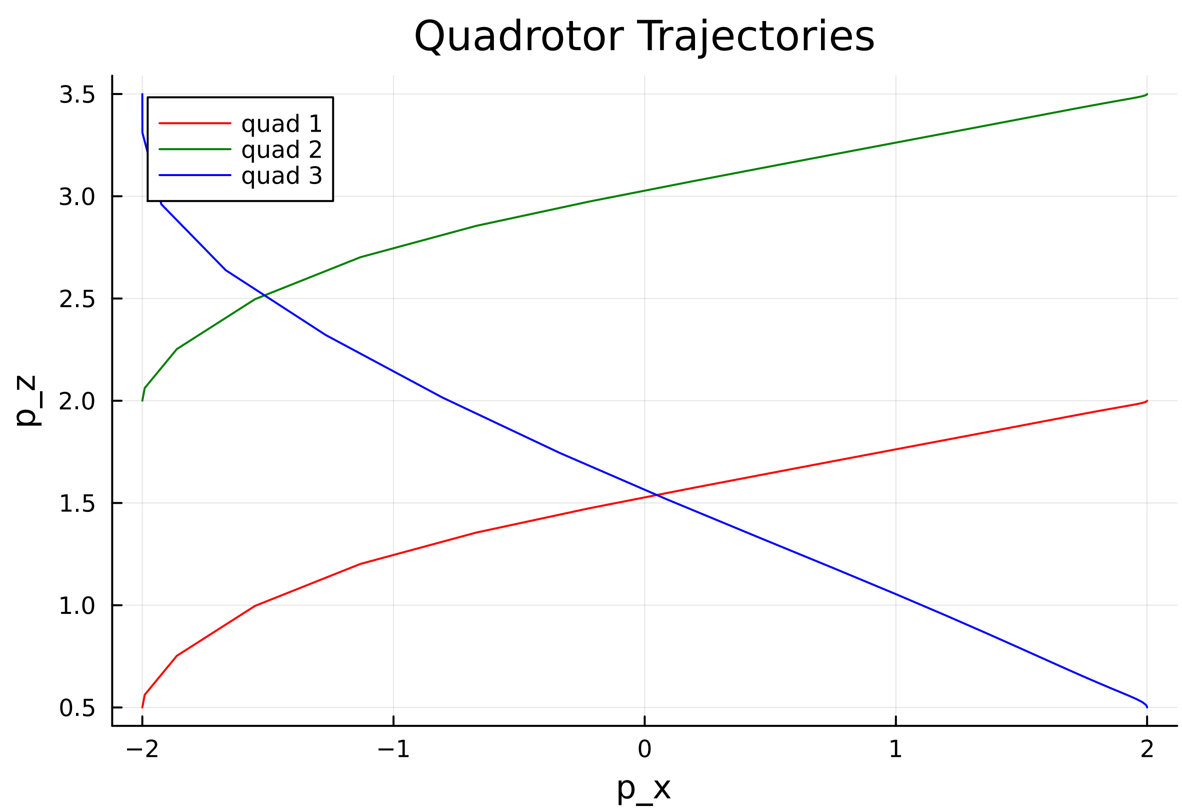
In [16]:
```
@testset "quadrotor reorient" begin

    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,x
3) in zip(X1,X2,X3)]
    @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 = "collisio
n distance",
         xlabel = "time (s)", ylabel = "distance (m)", title = "Distance betwe
en Quadrotors")
    display(plot!(t_vec, hcat(distances...)', label = ["|r_1 - r_2|" "|r_1 - r
_3|" "|r_2 - r_2|"]))

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

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

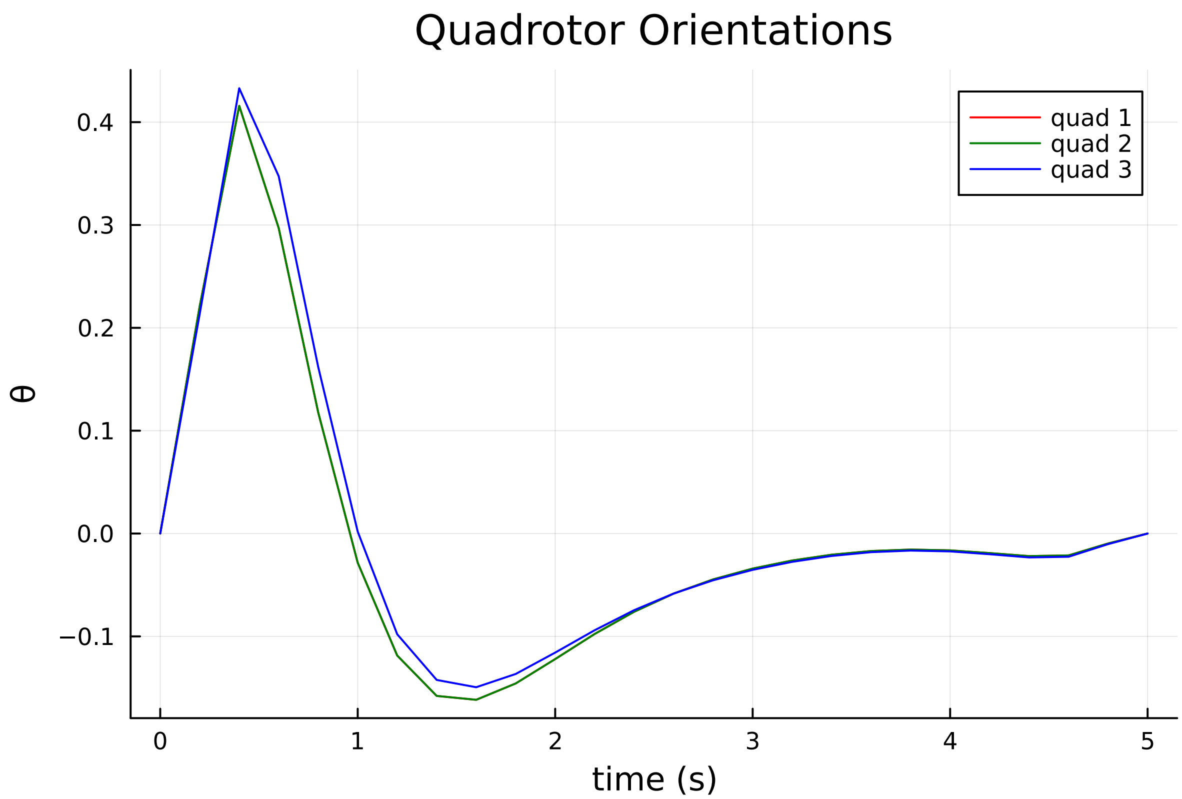
```
end
```

```
quadrotor reorient: Test Failed at In[16]:15
  Expression: minimum(minimum.(distances)) >= 0.799
   Evaluated: 0.11403010439325294 >= 0.799
Stacktrace:
 [1] macro expansion
   @ In[16]:15 [inlined]
 [2] macro expansion
   @ C:\buildbot\worker\package_win64\build\usr\share\julia\stdlib\v1.6\Test
\src\Test.jl:1151 [inlined]
 [3] top-level scope
   @ In[16]:3

┌ Info: MeshCat server started. You can open the visualizer by visiting the f
ollowing URL in your browser:
└ http://127.0.0.1:8707
```




Distance between Quadrotors
distance (m)
time (s)
0.5
1.0
1.5
2.0
2.5
3.0
0
1
2
3
4
5
collision distance
|r_1 - r_2|
|r_1 - r_3|
|r_2 - r_2|


Quadrotor Trajectories
p_z
p_x
0.5
1.0
1.5
2.0
2.5
3.0
3.5
−2
−1
0
1
2
quad 1
quad 2
quad 3




```
Test Summary:      | Pass  Fail  Total
quadrotor reorient |   13     1     14

Some tests did not pass: 13 passed, 1 failed, 0 errored, 0 broken.

Stacktrace:
 [1] finish(ts::Test.DefaultTestSet)
   @ Test C:\buildbot\worker\package_win64\build\usr\share\julia\stdlib\v1.6
\Test\src\Test.jl:913
 [2] macro expansion
   @ C:\buildbot\worker\package_win64\build\usr\share\julia\stdlib\v1.6\Test
\src\Test.jl:1161 [inlined]
 [3] top-level scope
   @ In[16]:3
```

In [ ]: