In [1]:
```
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
```

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

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

Out[2]:
```
animate_cartpole (generic function with 1 method)
```

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

Q1: Direct Collocation (DIRCOL) for a Cart Pole (30 pts)

We are now going to start working with the NonLinear Program (NLP) Solver IPOPT to solve some trajectory optimization problems. First we will demonstrate how this works for simple optimization problems (not trajectory optimization). The interface that we have setup for IPOPT is the following:

$$
\begin{aligned}
\min_{x} \quad & \ell(x) && \text{cost function} \\
\text{st} \quad & c_{eq}(x) = 0 && \text{equality constraint} \\
& c_L \leq c_{ineq}(x) \leq c_U && \text{inequality constraint} \\
& x_L \leq x \leq x_U && \text{primal bound constraint}
\end{aligned}
$$

where $\ell(x)$ is our objective function, $c_{eq}(x) = 0$ is our equality constraint, $c_L \leq c_{ineq}(x) \leq c_U$ is our bound inequality constraint, and $x_L \leq x \leq x_U$ is a bound constraint on our primal variable x.

Part A: Solve an LP with IPOPT (5 pts)

To demonstrate this, we are going to ask you to solve a simple Linear Program (LP):

$$\begin{aligned} \min_x \quad & q^T x \\ \text{st} \quad & Ax = b \\ & Gx \leq h \end{aligned}$$

Your job will be to transform this problem into the form shown above and solve it with IPOPT. To help you interface with IPOPT, we have created a function `fmincon` for you. Below is the docstring for this function that details all of the inputs.

In [3]:

```
"""
x = fmincon(cost,equality_constraint,inequality_constraint,x_l,x_u,c_l,c_u,x0,
params,diff_type)

This function uses IPOPT to minimize an objective function

`cost(params, x)`

With the following three constraints:

`equality_constraint(params, x) = 0`
`c_l <= inequality_constraint(params, x) <= c_u`
`x_l <= x <= x_u`

Note that the constraint functions should return vectors.

Problem specific parameters should be loaded into params::NamedTuple (things l
ike
cost weights, dynamics parameters, etc.).

args:
    cost::Function                      - objective function to be minimzed (ret
urns scalar)
    equality_constraint::Function       - c_eq(params, x) == 0
    inequality_constraint::Function     - c_l <= c_ineq(params, x) <= c_u
    x_l::Vector                         - x_l <= x <= x_u
    x_u::Vector                         - x_l <= x <= x_u
    c_l::Vector                         - c_l <= c_ineq(params, x) <= x_u
    c_u::Vector                         - c_l <= c_ineq(params, x) <= x_u
    x0::Vector                          - initial guess
    params::NamedTuple                  - problem parameters for use in costs/co
nstraints
    diff_type::Symbol                   - :auto for ForwardDiff, :finite for Fin
iteDiff
    verbose::Bool                       - true for IPOPT output, false for nothi
ng

optional args:
    tol                                 - optimality tolerance
    c_tol                               - constraint violation tolerance
    max_iters                           - max iterations
    verbose                             - verbosity of IPOPT

outputs:
    x::Vector                           - solution

You should try and use :auto for your `diff_type` first, and only use :finite
if you
absolutely cannot get ForwardDiff to work.

This function will run a few basic checks before sending the problem off to IP
OPT to
solve. The outputs of these checks will be reported as the following:

---------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----------------------

If you're getting stuck during the testing of one of the derivatives, try swit
ching
to FiniteDiff.jl by setting diff_type = :finite.
""";
```

In [4]:
```julia
@testset "solve LP with IPOPT" begin

    LP = jldopen(joinpath(@__DIR__,"utils","random_LP.jld2"))

    params = (q = LP["q"], A = LP["A"], b = LP["b"], G = LP["G"], h = LP["h"])

    # return a scalar
    function cost(params, x)::Real
        # TODO: create cost function with params and x
        cost = (params.q)'*x
        return cost
    end

    # return a vector
    function equality_constraint(params, x)::Vector
        # TODO: create equality constraint function with params and x
        ceq = params.A*x - params.b
        return ceq
    end

    # return a vector
    function inequality_constraint(params, x)::Vector
        # TODO: create inequality constraint function with params and x
        cineq = params.G*x - params.h
        return cineq
    end

    # TODO: primal bounds
    # you may use Inf, like Inf*ones(10) for a vector of positive infinity
    x_l = -Inf*ones(20)
    x_u = Inf*ones(20)

    # TODO: inequality constraint bounds
    c_l = -Inf*ones(20)
    c_u = 0*ones(20)

    # initial guess
    x0= randn(20)

    diff_type = :auto   # use ForwardDiff.jl
#     diff_type = :finite # use FiniteDiff.jl

    x = fmincon(cost, equality_constraint, inequality_constraint,
                x_l, x_u, c_l, c_u, x0, params, diff_type;
                tol = 1e-6, c_tol = 1e-6, max_iters = 10_000, verbose = true);


    @test isapprox(x, [-0.44289, 0, 0, 0.19214, 0, 0, -0.109095,
                       -0.43221, 0, 0, 0.44289, 0, 0, 0.192142,
                       0, 0, 0.10909, 0.432219, 0, 0], atol = 1e-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 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 3.14.4, running with linear solver MUMPS 5.4.1.

Number of nonzeros in equality constraint Jacobian...:       80
Number of nonzeros in inequality constraint Jacobian.:      400
Number of nonzeros in Lagrangian Hessian.............:        0

Total number of variables............................:       20
                     variables with only lower bounds:        0
                variables with lower and upper bounds:        0
                     variables with only upper bounds:        0
Total number of equality constraints.................:        4
Total number of inequality constraints...............:       20
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:       20

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
   0  4.8663662e+00 4.04e+00 3.33e-01   0.0 0.00e+00    -  0.00e+00 0.00e+00
0
   1  4.7616061e+00 5.41e-16 3.05e-01  -1.0 1.49e+00    -  7.48e-01 1.00e+00f
1
   2  2.5352950e+00 1.11e-16 4.43e-02  -1.2 1.58e+00    -  8.55e-01 8.49e-01f
1
   3  1.6857903e+00 5.55e-17 6.97e-08  -2.0 5.37e-01    -  1.00e+00 6.73e-01f
1
   4  1.3222773e+00 5.55e-17 3.05e-09  -3.4 1.77e-01    -  1.00e+00 7.17e-01f
1
   5  1.1791799e+00 1.11e-16 1.09e-09  -4.0 6.80e-02    -  8.81e-01 9.96e-01f
1
   6  1.1763521e+00 1.11e-16 3.21e-12  -9.8 9.35e-04    -  9.97e-01 9.99e-01f
1
   7  1.1763494e+00 2.22e-16 2.33e-15 -11.0 1.37e-06    -  1.00e+00 1.00e+00f
1

Number of Iterations....: 7

                                   (scaled)                 (unscaled)
Objective...............:   1.1763493513115122e+00    1.1763493513115122e+00
```

```
Dual infeasibility......:   2.3314683517128287e-15    2.3314683517128287e-15
Constraint violation....:   2.2204460492503131e-16    2.2204460492503131e-16
Variable bound violation:   0.0000000000000000e+00    0.0000000000000000e+00
Complementarity.........:   1.0901269946697252e-11    1.0901269946697252e-11
Overall NLP error.......:   1.0901269946697252e-11    1.0901269946697252e-11


Number of objective function evaluations             = 8
Number of objective gradient evaluations             = 8
Number of equality constraint evaluations            = 8
Number of inequality constraint evaluations          = 8
Number of equality constraint Jacobian evaluations   = 8
Number of inequality constraint Jacobian evaluations = 8
Number of Lagrangian Hessian evaluations             = 0
Total seconds in IPOPT                               = 2.935

EXIT: Optimal Solution Found.
Test Summary:       | Pass  Total
solve LP with IPOPT |    1      1
```

Out[4]:
```
Test.DefaultTestSet("solve LP with IPOPT", Any[], 1, false, false)
```

Part B: Cart Pole Swingup (20 pts)

We are now going to solve for a cartpole swingup. The state for the cartpole is the following:

$$x = [p, \theta, \dot{p}, \dot{\theta}]^T$$

Where p and θ can be seen in the graphic `cartpole.png` .

where we start with the pole in the down position ($\theta = 0$), and we want to use the horizontal force on the cart to drive the pole to the up position ($\theta = \pi$).

$$\begin{aligned}
\min_{x_{1:N}, u_{1:N-1}} \quad & \sum_{i=1}^{N-1} \left[\frac{1}{2}(x_i - x_{goal})^T Q (x_i - x_{goal}) + \frac{1}{2} u_i^T R u_i \right] + \frac{1}{2}(x_N - x_{goal})^T Q_f (x_N - x_{goal}) \\
\text{st} \quad & x_1 = x_{\text{IC}} \\
& x_N = x_{goal} \\
& f_{hs}(x_i, x_{i+1}, u_i, dt) = 0 \quad \text{for } i = 1, 2, \ldots, N-1 \\
& -10 \leq u_i \leq 10 \quad \text{for } i = 1, 2, \ldots, N-1
\end{aligned}$$

Where $x_{IC} = [0, 0, 0, 0]$, and $x_{goal} = [0, \pi, 0, 0]$, and $f_{hs}(x_i, x_{i+1}, u_i)$ is the implicit integrator residual for Hermite Simpson (see HW1Q1 to refresh on this). Note that while Zac used a first order hold (FOH) on the controls in class (meaning we linearly interpolate controls between time steps), we are using a zero-order hold (ZOH) in this assignment. This means that each control u_i is held constant for the entirety of the timestep.

In [5]:

```julia
# cartpole
function dynamics(params::NamedTuple, x::Vector, u)
    # cartpole ODE, parametrized by params.

    # cartpole physical parameters
    mc, mp, l = params.mc, params.mp, params.l
    g = 9.81

    q = x[1:2]
    qd = x[3:4]

    s = sin(q[2])
    c = cos(q[2])

    H = [mc+mp mp*l*c; mp*l*c mp*l^2]
    C = [0 -mp*qd[2]*l*s; 0 0]
    G = [0, mp*g*l*s]
    B = [1, 0]

    qdd = -H\(C*qd + G - B*u[1])
    xdot = [qd;qdd]
    return xdot

end
function hermite_simpson(params::NamedTuple, x1::Vector, x2::Vector, u, dt::Real)::Vector
    # TODO: input hermite simpson implicit integrator residual
#function hermite_simpson(params::NamedTuple, dynamics::Function, x1::Vector, x2::Vector, u, dt::Real)::Vector #u instead of dynamics
    x1dot = dynamics(params,x1,u)
    x2dot = dynamics(params,x2,u)
    xk = (0.5*(x1+x2))+((dt/8).*(x1dot-x2dot))
    xkdot = dynamics(params,xk,u)
    residuals = x1 + ((dt/6).*(x1dot+(4*xkdot)+x2dot)) - x2
end
```

Out[5]:

```
hermite_simpson (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:

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

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.

It is also worth noting that while there are inequality constraints present ($-10 \leq u_i \leq 10$), we do not need a specific `inequality_constraints` function as an input to `fmincon` since these are just bounds on the primal (Z) variable. You should use primal bounds in `fmincon` to capture these constraints.

In [6]:
```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

function cartpole_cost(params::NamedTuple, Z::Vector)::Real
    idx, N, xg = params.idx, params.N, params.xg
    Q, R, Qf = params.Q, params.R, params.Qf

    # TODO: input cartpole LQR cost
    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

    # dont forget terminal cost
    x_T = (Z[idx.x[N]]-xg)
    J += 0.5*(x_T'*Qf*x_T)

    return J
end

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

    # TODO: create dynamics constraints using hermite simpson

    # create c in a ForwardDiff friendly way (check HW0)
    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]]
        # TODO: hermite simpson
        c[idx.c[i]] = hermite_simpson(params, xi, xip1, ui, dt)
    end
    return c
end
```

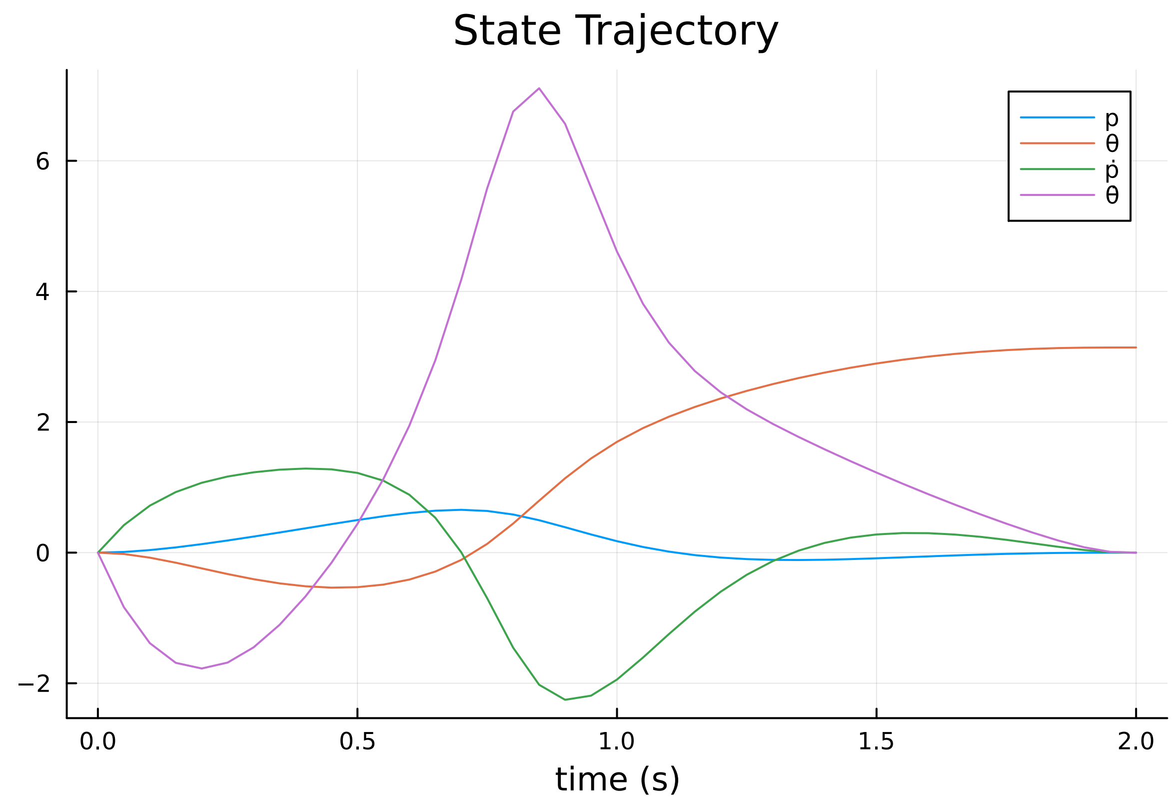
```julia
function cartpole_equality_constraint(params::NamedTuple, Z::Vector)::Vector
    N, idx, xic, xg = params.N, params.idx, params.xic, params.xg
    c = cartpole_dynamics_constraints(params, Z)
    # TODO: return all of the equality constraints
    ceq = Z[idx.x[1]] - xic
    ceq2 = Z[idx.x[N]] - xg
    return [ceq; ceq2; c]
end

function solve_cartpole_swingup(;verbose=true)

    # problem size
    nx = 4
    nu = 1
    dt = 0.05
    tf = 2.0
    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
    xic = [0, 0, 0, 0]
    xg = [0, pi, 0, 0]

    # load all useful things into params
    params = (Q = Q, R = R, Qf = Qf, xic = xic, xg = xg, dt = dt, N = N, idx =
idx,mc = 1.0, mp = 0.2, l = 0.5)

    # TODO: primal bounds
    x_l = -Inf*ones(204)
    x_u =  Inf*ones(204)



    # 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
```

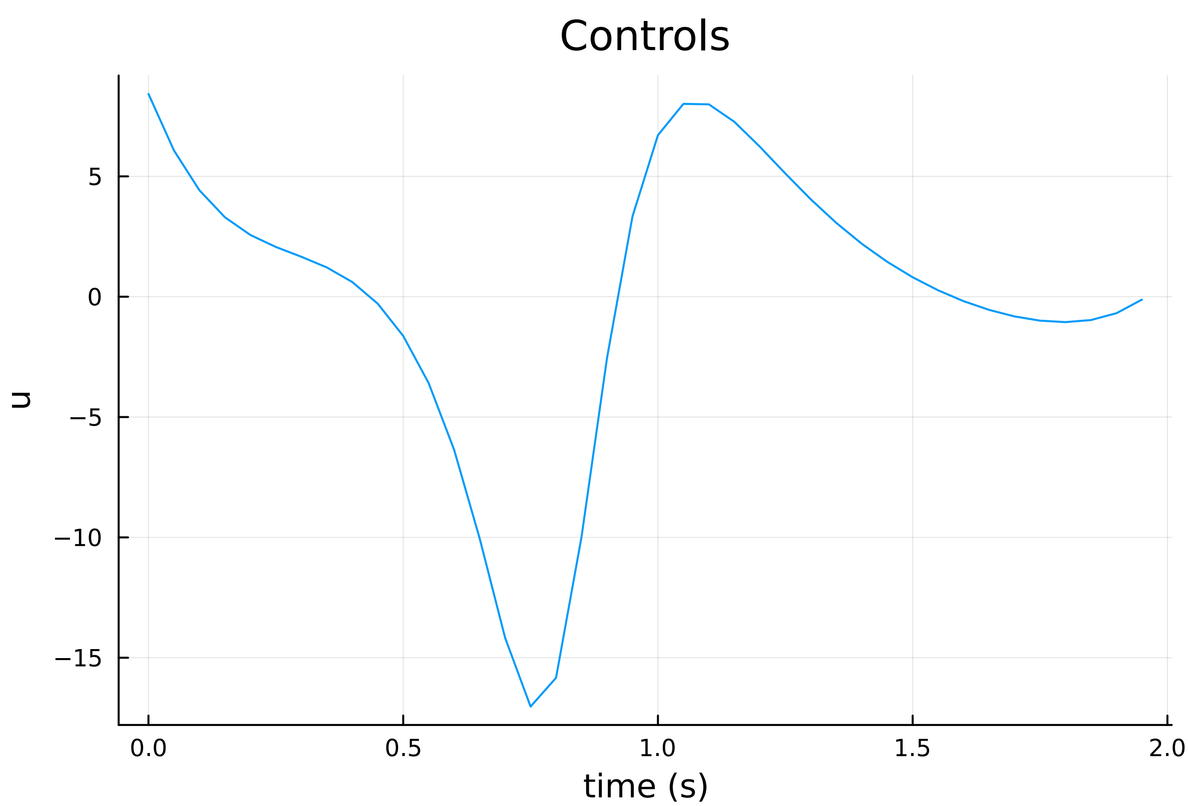
```
    Z = fmincon(cartpole_cost,cartpole_equality_constraint,inequality_constrai
nt,
                x_l,x_u,c_l,c_u,z0,params, diff_type;
                tol = 1e-6, c_tol = 1e-6, max_iters = 10_000, verbose = verbos
e)

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

    return X, U, t_vec, params
end

@testset "cartpole swingup" begin

    X, U, t_vec = solve_cartpole_swingup(verbose=true)


    # --------------testing------------------
    @test isapprox(X[1],zeros(4), atol = 1e-4)
    @test isapprox(X[end], [0,pi,0,0], atol = 1e-4)
    Xm = hcat(X...)
    Um = hcat(U...)

    # --------------plotting-----------------
    display(plot(t_vec, Xm', label = ["p" "θ" "ṗ" "θ̇"], xlabel = "time (s)", t
itle = "State Trajectory"))
    display(plot(t_vec[1:end-1],Um',label="",xlabel = "time (s)", ylabel =
"u",title = "Controls"))

    # meshcat animation
    display(animate_cartpole(X, 0.05))

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...:    34272
Number of nonzeros in inequality constraint Jacobian.:        0
Number of nonzeros in Lagrangian Hessian.............:        0

Total number of variables............................:      204
                     variables with only lower bounds:        0
                variables with lower and upper bounds:        0
                     variables with only upper bounds:        0
Total number of equality constraints.................:      168
Total number of inequality constraints...............:        0
        inequality constraints with only lower bounds:        0
   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  2.4674219e+02 3.14e+00 4.75e-04   0.0 0.00e+00    -  0.00e+00 0.00e+00
0
   1  2.7678226e+02 2.36e+00 7.82e+00 -11.0 1.27e+01    -  1.00e+00 2.50e-01h
3
   2  3.0916188e+02 2.06e+00 1.09e+01 -11.0 1.14e+01    -  1.00e+00 1.25e-01h
4
   3  3.4084546e+02 1.80e+00 1.50e+01 -11.0 1.55e+01    -  1.00e+00 1.25e-01h
4
   4  3.6416957e+02 1.58e+00 2.15e+01 -11.0 2.14e+01    -  1.00e+00 1.25e-01h
4
   5  3.9055035e+02 1.38e+00 2.66e+01 -11.0 2.15e+01    -  1.00e+00 1.25e-01h
4
   6  3.9949096e+02 1.29e+00 2.86e+01 -11.0 3.57e+01    -  1.00e+00 6.25e-02h
5
   7  4.1199378e+02 1.21e+00 3.05e+01 -11.0 2.09e+01    -  1.00e+00 6.25e-02h
5
   8  4.2167347e+02 1.14e+00 3.30e+01 -11.0 4.31e+01    -  1.00e+00 6.25e-02h
5
   9  4.3514581e+02 1.07e+00 3.40e+01 -11.0 2.55e+01    -  1.00e+00 6.25e-02h
5
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  10  4.4962297e+02 1.00e+00 3.48e+01 -11.0 2.81e+01    -  1.00e+00 6.25e-02h
5
  11  1.3226830e+03 6.08e+00 4.61e+01 -11.0 3.10e+01    -  1.00e+00 1.00e+00w
1
  12  6.4772710e+02 5.66e+00 5.99e+02 -11.0 4.26e+01    -  1.00e+00 1.00e+00w
1
  13  5.0443536e+03 9.11e+00 8.38e+01 -11.0 1.06e+02    -  1.00e+00 1.00e+00w
1
  14  4.6507831e+02 9.37e-01 3.55e+01 -11.0 8.53e+01    -  1.00e+00 6.25e-02h
4
```

```
  15  4.6442468e+02 8.79e-01 4.04e+01 -11.0 3.17e+01    -  1.00e+00 6.25e-02f
5
  16  4.2060153e+02 7.69e-01 5.40e+01 -11.0 1.15e+02    -  1.00e+00 1.25e-01f
4
  17  4.1487172e+02 7.21e-01 5.29e+01 -11.0 2.80e+01    -  1.00e+00 6.25e-02f
5
  18  4.2257935e+02 6.31e-01 4.53e+01 -11.0 3.20e+01    -  1.00e+00 1.25e-01h
4
  19  4.2245289e+02 5.91e-01 4.41e+01 -11.0 4.71e+01    -  1.00e+00 6.25e-02f
5
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  20  4.2192739e+02 5.73e-01 4.31e+01 -11.0 4.65e+01    -  1.00e+00 3.12e-02f
6
  21  4.2155381e+02 5.37e-01 4.12e+01 -11.0 3.99e+01    -  1.00e+00 6.25e-02f
5
  22  4.2179862e+02 5.04e-01 3.95e+01 -11.0 4.81e+01    -  1.00e+00 6.25e-02h
5
  23  4.2414843e+02 4.41e-01 4.19e+01 -11.0 2.60e+01    -  1.00e+00 1.25e-01h
4
  24  7.6507986e+02 2.87e+00 7.62e+01 -11.0 3.65e+01    -  1.00e+00 1.00e+00w
1
  25  5.3501820e+02 7.72e-01 8.98e+01 -11.0 2.65e+01    -  1.00e+00 1.00e+00w
1
  26  4.9265118e+02 1.04e+00 8.31e+01 -11.0 2.09e+01    -  1.00e+00 1.00e+00w
1
  27  4.3744255e+02 9.91e-02 6.50e+01 -11.0 7.07e+00    -  1.00e+00 1.00e+00h
1
  28  4.1964195e+02 8.11e-02 3.28e+01 -11.0 7.38e+00    -  1.00e+00 1.00e+00f
1
  29  4.1301246e+02 1.76e-01 1.65e+01 -11.0 6.69e+00    -  1.00e+00 1.00e+00f
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  30  4.0183064e+02 8.90e-02 1.90e+01 -11.0 9.28e+00    -  1.00e+00 1.00e+00f
1
  31  3.8239286e+02 1.21e-01 2.80e+01 -11.0 7.66e+00    -  1.00e+00 1.00e+00F
1
  32  3.8536136e+02 2.19e-02 1.90e+01 -11.0 5.82e+00    -  1.00e+00 1.00e+00h
1
  33  3.7804127e+02 3.73e-02 1.59e+01 -11.0 6.36e+00    -  1.00e+00 1.00e+00F
1
  34  3.7660585e+02 8.66e-03 1.30e+01 -11.0 3.86e+00    -  1.00e+00 1.00e+00F
1
  35  3.7450880e+02 8.95e-03 2.11e+01 -11.0 5.77e+00    -  1.00e+00 1.00e+00F
1
  36  3.6507143e+02 1.81e-01 3.22e+01 -11.0 1.17e+01    -  1.00e+00 1.00e+00F
1
  37  3.7242383e+02 1.03e-02 2.02e+01 -11.0 3.98e+00    -  1.00e+00 1.00e+00h
1
  38  3.6956593e+02 8.65e-03 6.59e+00 -11.0 1.70e+00    -  1.00e+00 1.00e+00f
1
  39  3.6998154e+02 4.03e-03 2.00e+00 -11.0 1.05e+00    -  1.00e+00 1.00e+00h
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  40  3.6893859e+02 6.26e-03 1.11e+01 -11.0 3.92e+00    -  1.00e+00 1.00e+00F
```

```
1
  41  3.5831207e+02 2.66e-01 2.28e+01 -11.0 8.71e+00    -  1.00e+00 1.00e+00F
1
  42  3.6213452e+02 1.24e-01 1.44e+01 -11.0 3.54e+00    -  1.00e+00 5.00e-01h
2
  43  3.6524750e+02 5.70e-02 6.49e+00 -11.0 2.33e+00    -  1.00e+00 5.00e-01h
2
  44  3.6958334e+02 9.49e-03 1.08e+00 -11.0 1.56e+00    -  1.00e+00 1.00e+00h
1
  45  3.6910420e+02 9.51e-06 3.14e-01 -11.0 9.60e-02    -  1.00e+00 1.00e+00f
1
  46  3.6910443e+02 3.24e-06 9.43e-02 -11.0 2.91e-02    -  1.00e+00 1.00e+00h
1
  47  3.6910426e+02 1.17e-06 6.69e-02 -11.0 2.29e-02    -  1.00e+00 1.00e+00h
1
  48  3.6910414e+02 2.12e-09 4.42e-04 -11.0 2.09e-02    -  1.00e+00 1.00e+00H
1
  49  3.6910414e+02 1.37e-11 6.56e-04 -11.0 2.69e-02    -  1.00e+00 1.00e+00H
1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  50  3.6910413e+02 8.85e-11 5.57e-04 -11.0 8.72e-03    -  1.00e+00 1.00e+00F
1
  51  3.6910413e+02 6.84e-08 1.36e-04 -11.0 7.25e-03    -  1.00e+00 1.00e+00h
1
  52  3.6910413e+02 1.07e-09 5.62e-05 -11.0 1.45e-03    -  1.00e+00 1.00e+00h
1
  53  3.6910413e+02 3.29e-10 8.10e-06 -11.0 4.74e-04    -  1.00e+00 1.00e+00h
1
  54  3.6910413e+02 8.58e-12 9.02e-06 -11.0 7.91e-05    -  1.00e+00 1.00e+00h
1
  55  3.6910413e+02 1.27e-12 6.65e-07 -11.0 5.38e-05    -  1.00e+00 1.00e+00h
1

Number of Iterations....: 55

                                   (scaled)                 (unscaled)
Objective...............:   3.6910412711158563e+02    3.6910412711158563e+02
Dual infeasibility......:   6.6454529645820770e-07    6.6454529645820770e-07
Constraint violation....:   1.2683187833317788e-12    1.2683187833317788e-12
Variable bound violation:   0.0000000000000000e+00    0.0000000000000000e+00
Complementarity.........:   0.0000000000000000e+00    0.0000000000000000e+00
Overall NLP error.......:   6.6454529645820770e-07    6.6454529645820770e-07


Number of objective function evaluations             = 162
Number of objective gradient evaluations             = 56
Number of equality constraint evaluations            = 162
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 56
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 0
Total seconds in IPOPT                               = 8.141

EXIT: Optimal Solution Found.
```






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

Open Controls

```
Test Summary:    | Pass  Total
cartpole swingup |    2      2
```

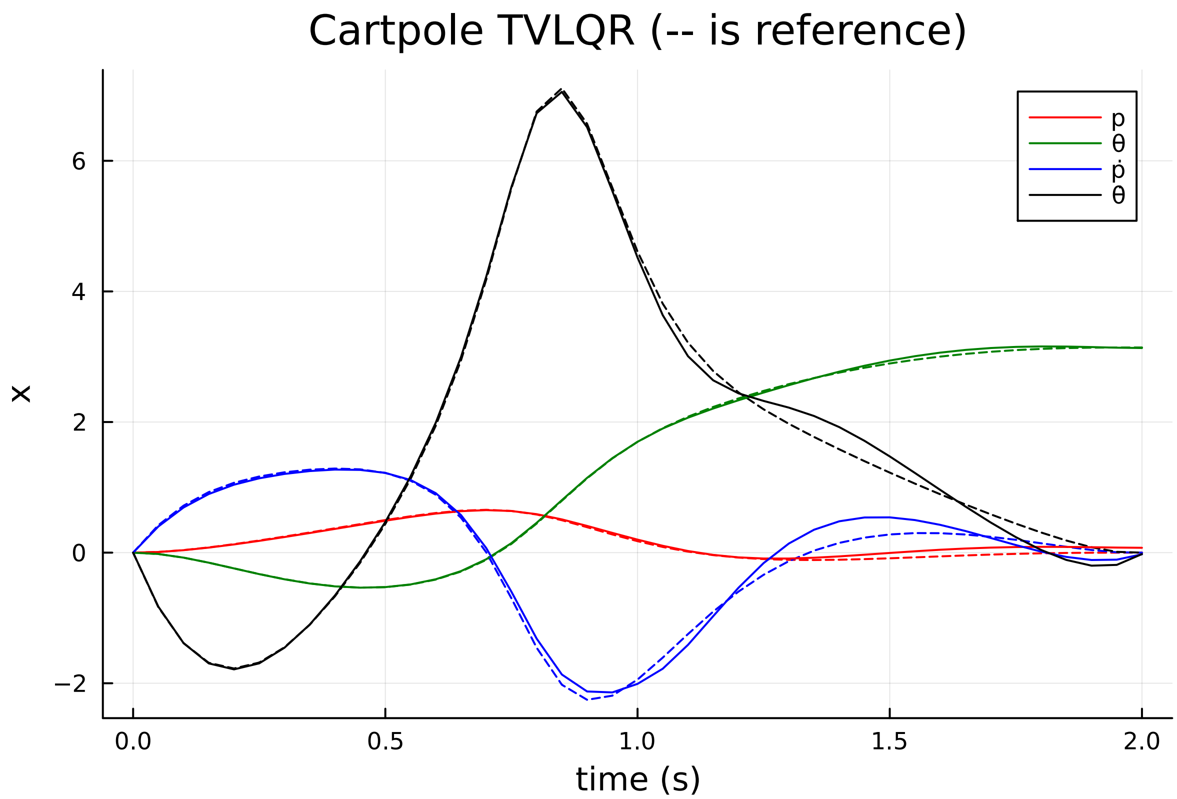
Out[6]: `Test.DefaultTestSet("cartpole swingup", Any[], 2, false, false)`

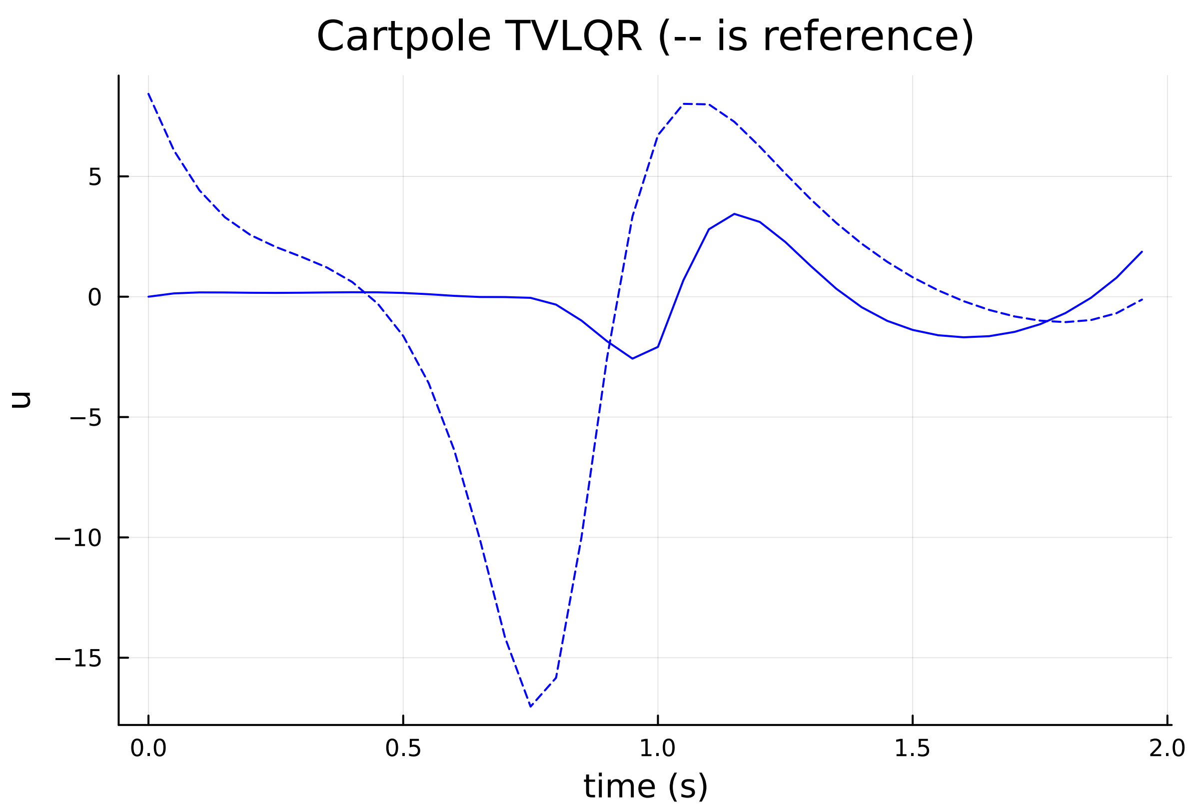
Part C: Track DIRCOL Solution (5 pts)

Now, similar to HW2 Q2 Part C, we are taking a solution X and U from DIRCOL, and we are going to track the trajectory with TVLQR to account for model mismatch. While we used hermite-simpson integration for the dynamics constraints in DIRCOL, we are going to use RK4 for this simulation. Remember to clamp your control to be within the control bounds.

In [7]:
```
function rk4(params::NamedTuple, x::Vector,u,dt::Float64)
    # vanilla RK4
    k1 = dt*dynamics(params, x, u)
    k2 = dt*dynamics(params, x + k1/2, u)
    k3 = dt*dynamics(params, x + k2/2, u)
    k4 = dt*dynamics(params, x + k3, u)
    x + (1/6)*(k1 + 2*k2 + 2*k3 + k4)
end

@testset "track cartpole swingup with TVLQR" begin

    X_dircol, U_dircol, t_vec, params_dircol = solve_cartpole_swingup(verbose
= false)

    N = length(X_dircol)
    dt = params_dircol.dt
    x0 = X_dircol[1]

    # TODO: use TVLQR to generate K's

    # use this for TVLQR tracking cost
    Q = diagm([1,1,.05,.1])
    Qf = 100*Q
    R = 0.01*diagm(ones(1))

    nx = 4
    nu = 1

    P =  [zeros(nx,nx) for i = 1:N] #(nx,nx)
    K =  [zeros(nu,nx) for i = 1:N-1] #(nu,nx)

    P[N] = deepcopy(Qf)

    U = [zeros(nu) for i = 1:N-1]

    P_k = P[N]
    for k = (N-1):-1:1
        #ForwardDiff
        #ForwardDiff
        A = ForwardDiff.jacobian(Dx -> rk4(params_dircol,Dx,U_dircol[k],dt),X_
dircol[k]) #Ubar and #Xbar
        B = ForwardDiff.jacobian(Du -> rk4(params_dircol,X_dircol[k],Du,dt),U_
dircol[k]) #Ubar and #Xbar

        K[k] = (R + B'*P_k*B)\(B'*P_k*A)
        K_k = K[k]
        P[k] = Q + A'*P_k*(A-B*(K_k))
        P_k = P[k]
    end

    # simulation
    Xsim = [zeros(4) for i = 1:N]
    Usim = [zeros(1) for i = 1:(N-1)]
    Xsim[1] = 1*x0

    # here are the real parameters (different than the one we used for DIRCOL)
```

```
    # this model mismatch is what's going to require the TVLQR controller to t
rack
    # the trajectory successfully.
    params_real = (mc = 1.05, mp = 0.21, l = 0.48)

    # TODO: simulate closed loop system
    for i = 1:(N-1)
        # TODO: add feeback control (right now it's just feedforward)

        Usim[i] = -K[i]*(Xsim[i]-X_dircol[i])
        #U_dircol[i] = clamp.(U_dircol[i], -10, 10)
        Usim[i] = clamp.(Usim[i], -10, 10)
        Xsim[i+1] = rk4(params_real, Xsim[i], (Usim[i]+U_dircol[i]), dt)
    end


    # -----------------testing----------------------
    xn = Xsim[N]
    @test norm(xn)>0
    @test 1e-6<norm(xn - X_dircol[end])<.8
    @test abs(abs(rad2deg(xn[2])) - 180) < 5 # within 5 degrees
    @test maximum(norm.(Usim,Inf)) <= (10 + 1e-3)

    # -----------------plotting---------------------
    Xm = hcat(Xsim...)
    Xbarm = hcat(X_dircol...)
    plot(t_vec,Xbarm',ls=:dash, label = "",lc = [:red :green :blue :black])
    display(plot!(t_vec,Xm',title = "Cartpole TVLQR (-- is reference)",
                 xlabel = "time (s)", ylabel = "x",
                 label = ["p" "θ" "ṗ" "θ̇"],lc = [:red :green :blue :black]))

    Um = hcat(Usim...)
    Ubarm = hcat(U_dircol...)
    plot(t_vec[1:end-1],Ubarm',ls=:dash,lc = :blue, label = "")
    display(plot!(t_vec[1:end-1],Um',title = "Cartpole TVLQR (-- is referenc
e)",
                 xlabel = "time (s)", ylabel = "u",lc = :blue, label = ""))

    # ----------------animate-----------------------
    display(animate_cartpole(Xsim, 0.05))

end
```






```
┌ Info: MeshCat server started. You can open the visualizer by visiting the f
ollowing URL in your browser:
└ http://127.0.0.1:8706
```

Open Controls

```
Test Summary:                    | Pass  Total
track cartpole swingup with TVLQR |    4      4
```

Out[7]:
```
Test.DefaultTestSet("track cartpole swingup with TVLQR", Any[], 4, false, false)
```

In [ ]:

In [1]:
```
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()

import MathOptInterface as MOI
import Ipopt
import ForwardDiff as FD
import Convex as cvx
import ECOS
using LinearAlgebra
using Plots
using Random
using JLD2
using Test
import MeshCat as mc
using Printf
```

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

Q2: iLQR (30 pts)

In this problem, we are going to use iLQR to solve a trajectory optimization for a 6DOF quadrotor. This problem we will use a cost function to motivate the quadrotor to follow a specified aerobatic manuever. The continuous time dynamics of the quadrotor are detailed in `quadrotor.jl` , with the state being the following:

$$x = [r, v, {}^N p^B, \omega]$$

where $r \in \mathbb{R}^3$ is the position of the quadrotor in the world frame (N), $v \in \mathbb{R}^3$ is the velocity of the quadrotor in the world frame (N), ${}^N p^B \in \mathbb{R}^3$ is the Modified Rodrigues Parameter (MRP) that is used to denote the attitude of the quadrotor, and $\omega \in \mathbb{R}^3$ is the angular velocity of the quadrotor expressed in the body frame (B). By denoting the attitude of the quadrotor with a MRP instead of a quaternion or rotation matrix, we have to be careful to avoid any scenarios where the MRP will approach it's singularity at 360 degrees of rotation. For the manuever planned in this problem, the MRP will be sufficient.

The dynamics of the quadrotor are discretized with `rk4` , resulting in the following discrete time dynamics function:

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

function discrete_dynamics(params::NamedTuple, x::Vector, u, k)
    # discrete dynamics
    # x - state
    # u - control
    # k - index of trajectory
    # dt comes from params.model.dt
    return rk4(params.model, quadrotor_dynamics, x, u, params.model.dt)
end
```

Out[2]:
```
discrete_dynamics (generic function with 1 method)
```

Part A: iLQR for a quadrotor (25 pts)

iLQR is used to solve optimal control problems of the following form:

$$\begin{aligned} \min_{x_{1:N},u_{1:N-1}} \quad & \left[\sum_{i=1}^{N-1} \ell(x_i,u_i)\right] + \ell_N(x_N) \\ \text{st} \quad & x_1 = x_{IC} \\ & x_{k+1} = f(x_k,u_k) \quad \text{for } i = 1,2,\ldots,N-1 \end{aligned}$$

where x_{IC} is the inital condition, $x_{k+1} = f(x_k,u_k)$ is the discrete dynamics function, $\ell(x_i,u_i)$ is the stage cost, and $\ell_N(x_N)$ is the terminal cost. Since this optimization problem can be non-convex, there is no guarantee of convergence to a global optimum, or even convergene rates to a local optimum, but in practice we will see that it can work very well.

For this problem, we are going to use a simple cost function consisting of the following stage cost:

$$\ell(x_i,u_i) = \frac{1}{2}(x_i - x_{ref,i})^T Q (x_i - x_{ref,i}) + \frac{1}{2}(u_i - u_{ref,i})^T R (u_i - u_{ref,i})$$

And the following terminal cost:

$$\ell_N(x_N) = \frac{1}{2}(x_N - x_{ref,N})^T Q_f (x_N - x_{ref,N})$$

This is how we will encourange our quadrotor to track a reference trajectory x_{ref}. In the following sections, you will implement `iLQR` and use it inside of a `solve_quadrotor_trajectory` function. Below we have included some starter code, but you are free to use/not use any of the provided functions so long as you pass the tests.

We will consider iLQR to have converged when $\Delta J < \text{atol}$ as calculated during the backwards pass.

In [3]:
```
function stage_cost(p::NamedTuple,x::Vector,u::Vector,k::Int)
    # TODO: return stage cost at time step k
    Q = p.Q
    x_k = x-p.Xref[k]
    R = p.R
    u_k = u-p.Uref[k]
    cost = 0.5*(x_k'*Q*x_k)+0.5*(u_k'*R*u_k)
    return cost
end
```

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

In [4]:
```
function term_cost(p::NamedTuple,x)
    # TODO: return terminal cost
    Qf = p.Qf
    x_f = x - p.Xref[p.N]
    cost = 0.5*(x_f'*Qf*x_f)
    return cost
end
```

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

In [5]:
```
function stage_cost_expansion(p::NamedTuple, x::Vector, u::Vector, k::Int)
    # TODO: return stage cost expansion
    # if the stage cost is J(x,u), you can return the following
    # ∇ₓ²J, ∇ₓJ, ∇ᵤ²J, ∇ᵤJ
    Q = p.Q
    x_k = x - p.Xref
    R = p.R
    u_k = u - p.Uref
    ∇ₓ²J = Q    #FD.hessian(Dx -> stage_cost(p, Dx, u, k),x_k)
    ∇ₓJ = Q*x_k  #FD.jacobian(Dx -> stage_cost(p, Dx, u, k),x_k)
    ∇ᵤ²J = R    #FD.hessian(Du -> stage_cost(p, x, Du, k),u_k)
    ∇ᵤJ = R*x_k #FD.jacobian(Du -> stage_cost(p, x, Du, k),u_k)
    return ∇ₓ²J, ∇ₓJ, ∇ᵤ²J, ∇ᵤJ
end
```

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

In [6]:
```
function term_cost_expansion(p::NamedTuple, x::Vector)
    # TODO: return terminal cost expansion
    # if the terminal cost is Jn(x,u), you can return the following
    # ∇ₓ²Jn, ∇ₓJn
    Qf = p.Qf
    x_f = x - p.Xref
    ∇ₓ²Jn = Qf*x_k  #FD.hessian(Dx -> term_cost(p,Dx),x_f)
    ∇ₓJn = Qf       #FD.jacobian(Dx -> term_cost(p,Dx),x_f)
    return ∇ₓ²Jn, ∇ₓJn
end
```

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

In [7]:
```
function backward_pass(params::NamedTuple,          # useful params
                      X::Vector{Vector{Float64}},  # state trajectory
                      U::Vector{Vector{Float64}})  # control trajectory
    # compute the iLQR backwards pass given a dynamically feasible trajectory
X and U
    # return d, K, ΔJ

    # outputs:
    #     d  - Vector{Vector} feedforward control
    #     K  - Vector{Matrix} feedback gains
    #     ΔJ - Float64        expected decrease in cost
    nx, nu, N = params.nx, params.nu, params.N

    # vectors of vectors/matrices for recursion
    P = [zeros(nx,nx) for i = 1:N]   # cost to go quadratic term
    p = [zeros(nx)    for i = 1:N]   # cost to go linear term
    d = [zeros(nu)    for i = 1:N-1] # feedforward control
    K = [zeros(nu,nx) for i = 1:N-1] # feedback gain

    # TODO: implement backwards pass and return d, K, ΔJ
    N = params.N
    ΔJ = 0.0

    #goal value
    xg = params.Xref[N]

    p[N] = params.Qf*(X[N]-xg)

    P[N] = params.Qf

    #Main Loop
    for k = (N-1):-1:1
        ######################################################################
#################
        #used to get dynamics for gx and gu
        q = params.Q*(X[k]-params.Xref[k])
        r = params.R*(U[k]-params.Uref[k])

        A = FD.jacobian(dx->discrete_dynamics(params,dx,U[k],k),X[k])
        B = FD.jacobian(du->discrete_dynamics(params,X[k],du,k),U[k])

        #obtain gx and gu
        gx = q + A'*p[k+1]
        gu = r + B'*p[k+1]

        #obtain Gxx, Guu, Gxu, Gux
        #regularize
        Gxx = params.Q + A'*P[k+1]*A
        Guu = params.R + B'*P[k+1]*B
        Gxu = A'*P[k+1]*B
        Gux = B'*P[k+1]*A

        #obtaining d delta_u = -d - K*delta_x
        d[k] = Guu\gu

        #obtaining K
```

```
        K[k] = Guu\Gux
        ######################################################################
###################

        p[k] = gx - K[k]'*gu + K[k]'*Guu*d[k] - Gxu*d[k]

        P[k] = Gxx + K[k]'*Guu*K[k] - Gxu*K[k] - K[k]'*Gux

        #should be okay
        ΔJ += gu'*d[k]
    end


    return d, K, ΔJ
end
```

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

In [8]:
```
function trajectory_cost(params::NamedTuple,          # useful params
                         X::Vector{Vector{Float64}},  # state trajectory
                         U::Vector{Vector{Float64}}) # control trajectory
    # compute the trajectory cost for trajectory X and U (assuming they are dy
namically feasible)
    J = 0
    N = params.N
    for k = 1:(N-1)
        J += stage_cost(params,X[k],U[k],k)

    end
    J += term_cost(params,X[N])

    # TODO: add trajectory cost
    return J
end
```

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

In [9]:
```
function forward_pass(params::NamedTuple,           # useful params
                      X::Vector{Vector{Float64}},   # state trajectory
                      U::Vector{Vector{Float64}},   # control trajectory
                      d::Vector{Vector{Float64}},   # feedforward controls
                      K::Vector{Matrix{Float64}};   # feedback gains
                      max_linesearch_iters = 20)    # max iters on linesearch
    # forward pass in iLQR with linesearch
    # use a line search where the trajectory cost simply has to decrease (no A
rmijo)

    # outputs:
    #     Xn::Vector{Vector}  updated state trajectory
    #     Un::Vector{Vector}  updated control trajectory
    #     J::Float64          updated cost
    #     α::Float64.         step length

    nx, nu, N = params.nx, params.nu, params.N

    Xn = [zeros(nx) for i = 1:N]      # new state history
    Un = [zeros(nu) for i = 1:N-1]    # new control history

    # initial condition
    Xn[1] = 1*X[1]

    # initial step length
    α = 1.0
    C = 0.5
    J = trajectory_cost(params,X,U)
    # TODO: add forward pass

    for i = 1:max_linesearch_iters
    ###################################################################

        for k = 1:(N-1)
            Un[k] = U[k] - α*d[k] - K[k]*(Xn[k]-X[k])
            Xn[k+1] = discrete_dynamics(params, Xn[k], Un[k], k)
        end

        Jn = trajectory_cost(params,Xn,Un)
        #print("\n")
        #print(Jn)
        #print("\n")
        if Jn < J
            J = Jn
            X = Xn
            U = Un
            return J, X, U, α
            break
        end

        α = C*α
    ###################################################################
    end

    error("forward pass failed")
end
```

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

In [10]:
```
function iLQR(params::NamedTuple,         # useful params for costs/dynamics/i
ndexing
              x0::Vector,                 # initial condition
              U::Vector{Vector{Float64}}; # initial controls
              atol=1e-3,                  # convergence criteria: ΔJ < atol
              max_iters = 250,            # max iLQR iterations
              verbose = true)             # print logging

    # iLQR solver given an initial condition x0, initial controls U, and a
    # dynamics function described by `discrete_dynamics`

    # return (X, U, K) where
    # outputs:
    #     X::Vector{Vector} - state trajectory
    #     U::Vector{Vector} - control trajectory
    #     K::Vector{Matrix} - feedback gains K

    # first check the sizes of everything
    @assert length(U) == params.N-1
    @assert length(U[1]) == params.nu
    @assert length(x0) == params.nx

    nx, nu, N = params.nx, params.nu, params.N

    # TODO: initial rollout
    X = [zeros(nx) for i = 1:N]

    X[1] = x0

    for k = 1:(N-1)
        X[k+1] = discrete_dynamics(params,X[k],U[k],k)
    end

    for ilqr_iter = 1:max_iters

        d, K, ΔJ = backward_pass(params,X,U)
        #print("\n backward pass complete \n")
        J, X, U, α = forward_pass(params, X, U, d, K, max_linesearch_iters = 2
0)

        # termination criteria
        if ΔJ < atol
            if verbose
                @info "iLQR converged"
            end
            return X, U, K
        end

        # ---------------logging -------------------
        if verbose
            dmax = maximum(norm.(d))
            if rem(ilqr_iter-1,10)==0
                @printf "iter     J           ΔJ        |d|         α         
\n"
                @printf "-------------------------------------------------\n"
            end
```

```
            @printf("%3d   %10.3e  %9.2e  %9.2e  %6.4f    \n",
              ilqr_iter, J, ΔJ, dmax, α)
        end

    end
    error("iLQR failed")
end
```

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

In [11]:
```
function create_reference(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(-p
i/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
function solve_quadrotor_trajectory(;verbose = true)

    # problem size
    nx = 12
    nu = 4
    dt = 0.05
    tf = 5
    t_vec = 0:dt:tf
    N = length(t_vec)

    # create reference trajectory
    Xref, Uref = create_reference(N, dt)

    # tracking cost function
    Q = 1*diagm([1*ones(3);.1*ones(3);1*ones(3);.1*ones(3)])
    R = .1*diagm(ones(nu))
    Qf = 10*Q

    # dynamics parameters (these are estimated)
    model = (mass=0.5,
            J=Diagonal([0.0023, 0.0023, 0.004]),
            gravity=[0,0,-9.81],
            L=0.1750,
            kf=1.0,
            km=0.0245,dt = dt)


    # the params needed by iLQR
    params = (
        N = N,
        nx = nx,
        nu = nu,
        Xref = Xref,
        Uref = Uref,
        Q = Q,
        R = R,
        Qf = Qf,
        model = model
    )

    # initial condition
    x0 = 1*Xref[1]

    # initial guess controls
```

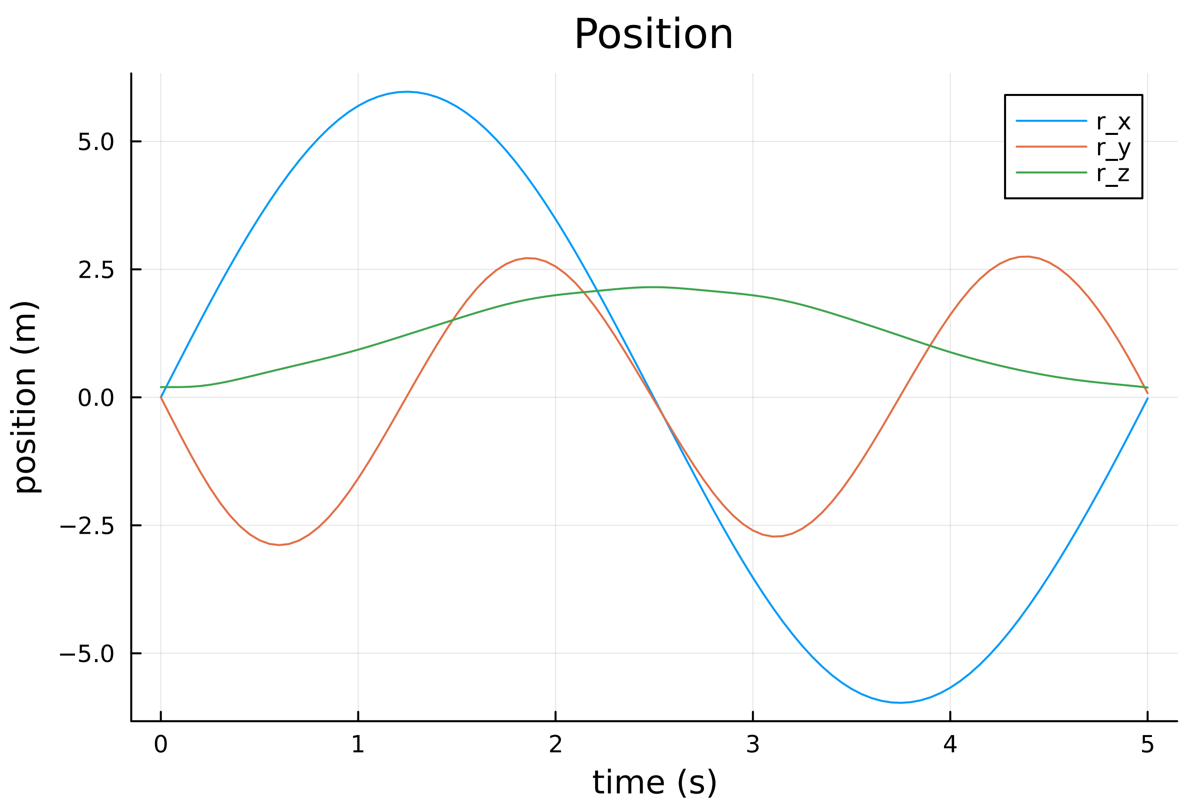
```
    U = [(uref + .0001*randn(nu)) for uref in Uref]

    # solve with iLQR
    X, U, K = iLQR(params,x0,U;atol=1e-4,max_iters = 250,verbose = verbose)

    return X, U, K, t_vec, params
end
```

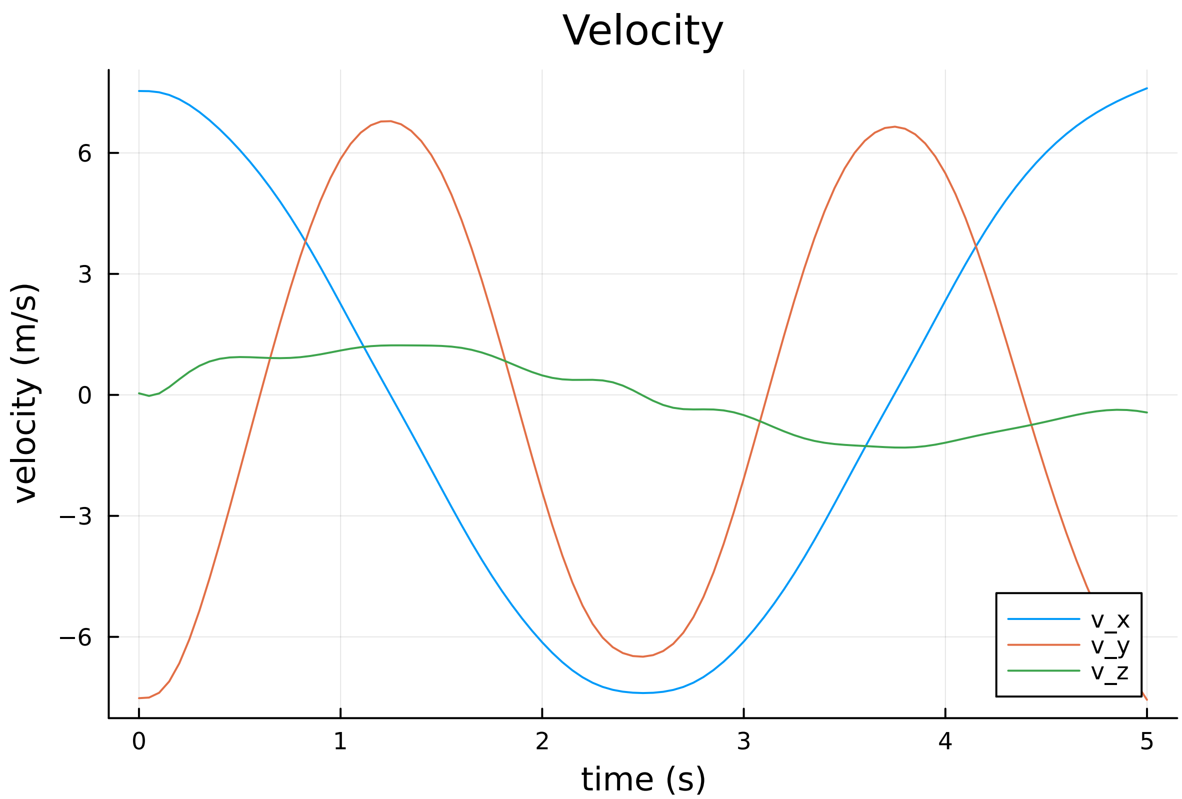
Out[11]:

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

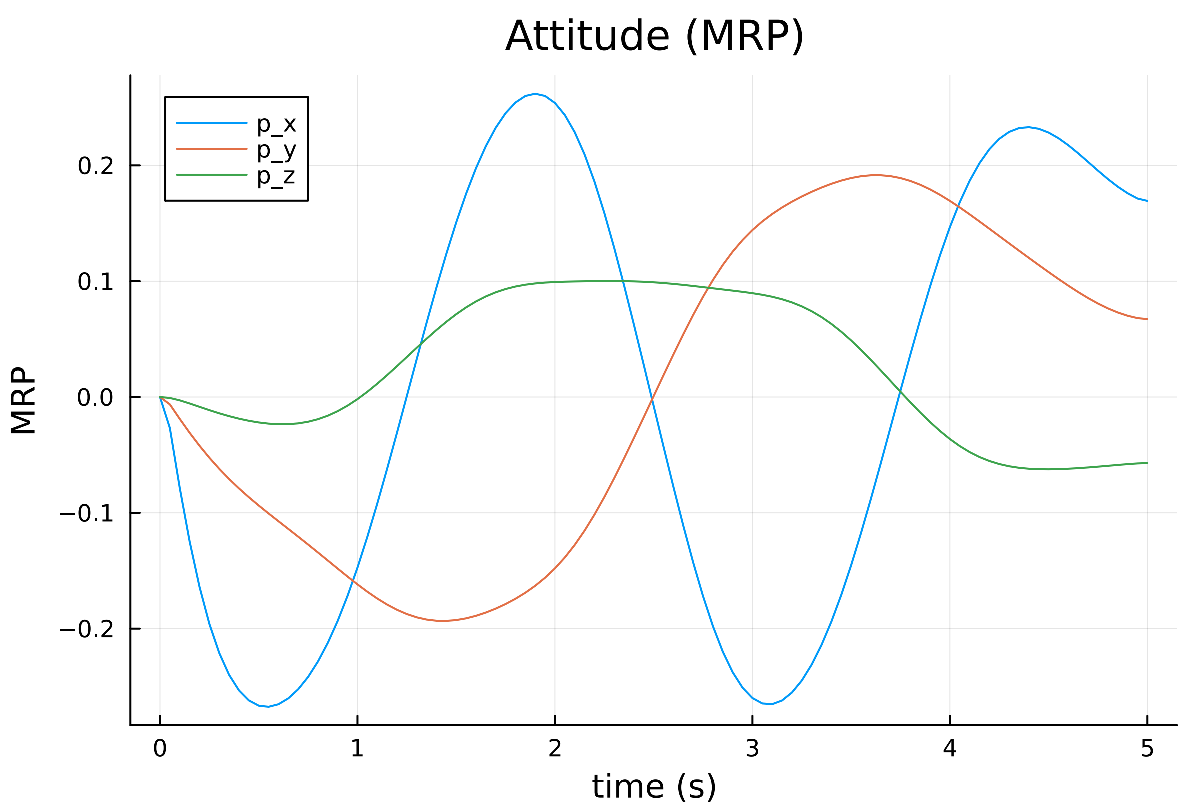
In [12]:
```
@testset "ilqr" begin
    # NOTE: set verbose to true here when you submit
    Xilqr, Uilqr, Kilqr, t_vec, params =  solve_quadrotor_trajectory(verbose =
true)

    # ---------------testing------------------
    Usol = load(joinpath(@__DIR__,"utils","ilqr_U.jld2"))["Usol"]
    @test maximum(norm.(Usol .- Uilqr,Inf)) <= 1e-2

    # ---------------plotting------------------
    Xm = hcat(Xilqr...)
    Um = hcat(Uilqr...)
    display(plot(t_vec, Xm[1:3,:]', xlabel = "time (s)", ylabel = "position
(m)",
                                   title = "Position", label = ["r_x" "r_y" "r
_z"]))
    display(plot(t_vec, Xm[4:6,:]', xlabel = "time (s)", ylabel = "velocity
(m/s)",
                                   title = "Velocity", label = ["v_x" "v_y" "v
_z"]))
    display(plot(t_vec, Xm[7:9,:]', xlabel = "time (s)", ylabel = "MRP",
                                   title = "Attitude (MRP)", label = ["p_x" "p
_y" "p_z"]))
    display(plot(t_vec, Xm[10:12,:]', xlabel = "time (s)", ylabel = "angular v
elocity (rad/s)",
                                   title = "Angular Velocity", label = ["ω_x"
"ω_y" "ω_z"]))
    display(plot(t_vec[1:end-1], Um', xlabel = "time (s)", ylabel = "rotor spe
eds (rad/s)",
                                   title = "Controls", label = ["u_1" "u_2" "u
_3" "u_4"]))
    display(animate_quadrotor(Xilqr, params.Xref, params.model.dt))
end
```

```
iter      J            ΔJ          |d|          α
-------------------------------------------------
  1    3.072e+02   1.32e+05   2.80e+01  1.0000
  2    1.096e+02   5.48e+02   1.34e+01  0.5000
  3    4.934e+01   1.37e+02   4.72e+00  1.0000
  4    4.431e+01   1.22e+01   2.44e+00  1.0000
  5    4.402e+01   8.57e-01   2.61e-01  1.0000
  6    4.398e+01   1.58e-01   9.19e-02  1.0000
  7    4.397e+01   4.22e-02   7.65e-02  1.0000
  8    4.396e+01   1.46e-02   4.02e-02  1.0000
  9    4.396e+01   5.80e-03   3.38e-02  1.0000
 10    4.396e+01   2.61e-03   2.08e-02  1.0000
iter      J            ΔJ          |d|          α
-------------------------------------------------
 11    4.396e+01   1.30e-03   1.71e-02  1.0000
 12    4.395e+01   7.05e-04   1.16e-02  1.0000
 13    4.395e+01   4.09e-04   9.48e-03  1.0000
 14    4.395e+01   2.50e-04   7.01e-03  1.0000
 15    4.395e+01   1.57e-04   5.71e-03  1.0000
 16    4.395e+01   1.01e-04   4.45e-03  1.0000

[ Info: iLQR converged
```

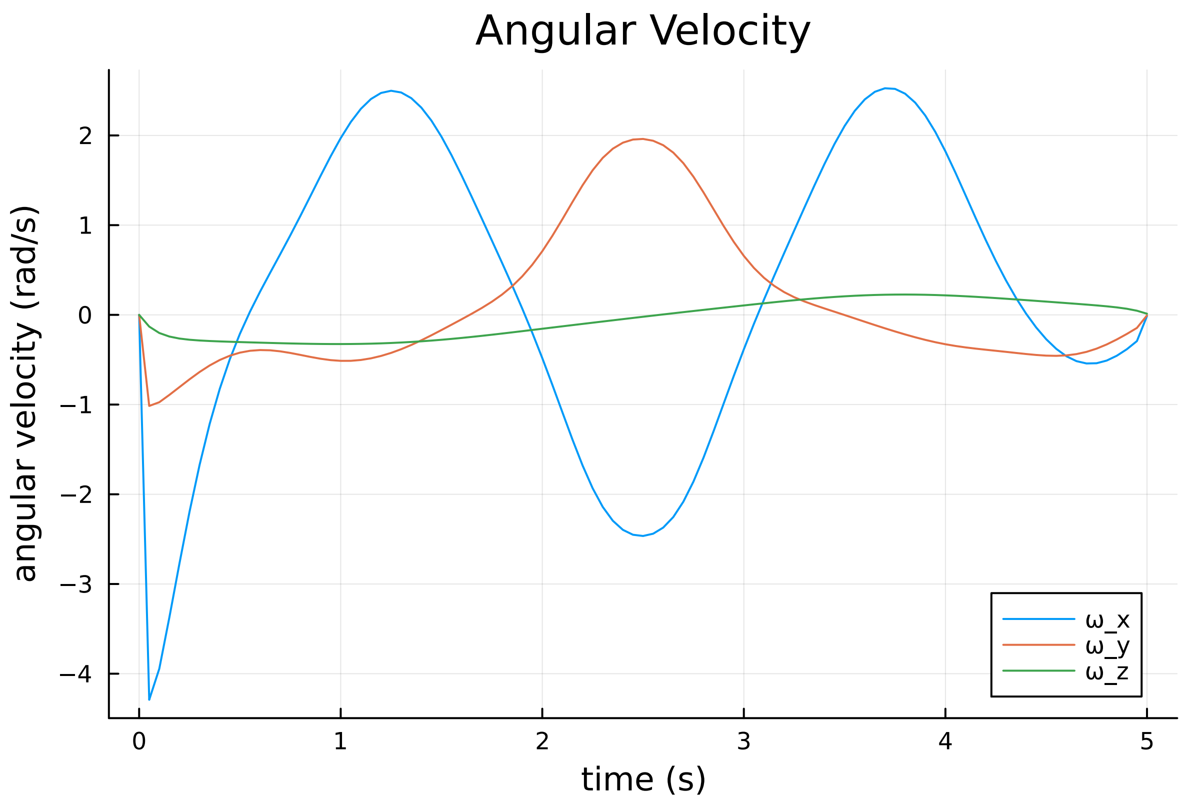


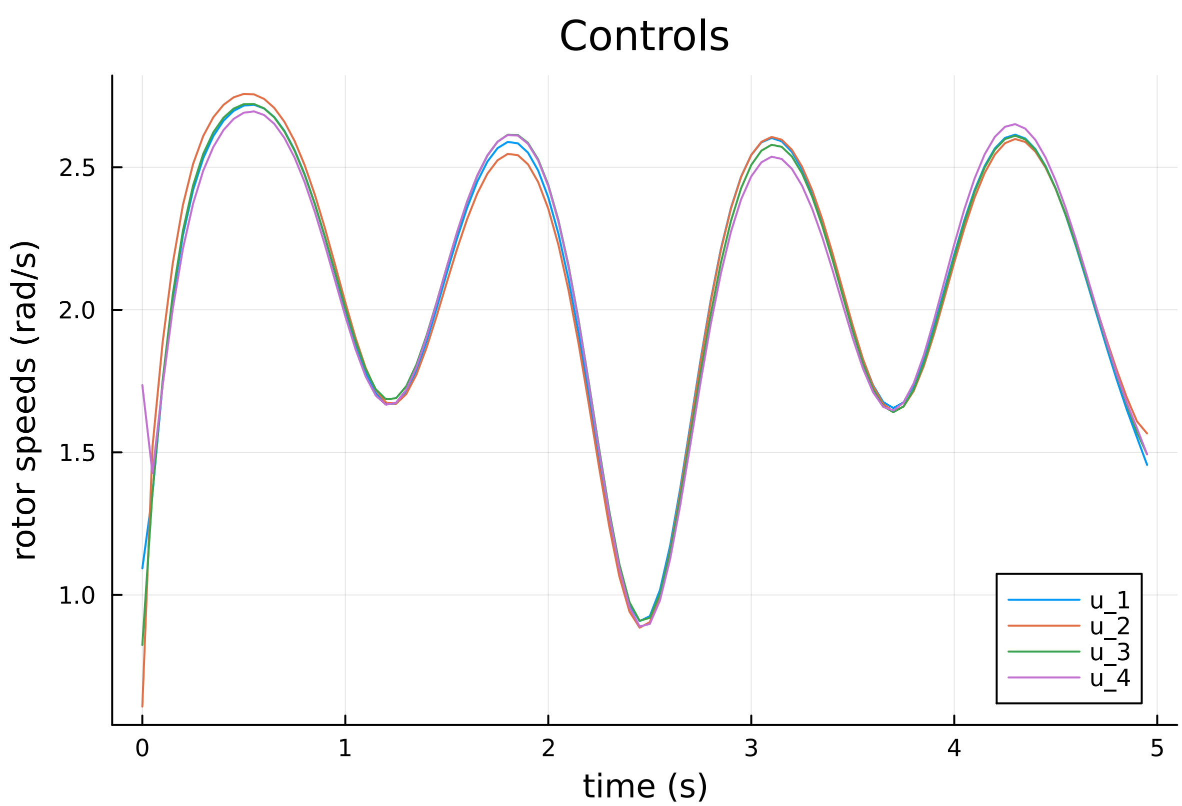


Velocity
6
3
0
−3
−6
velocity (m/s)
0
1
2
3
4
5
time (s)
v_x
v_y
v_z

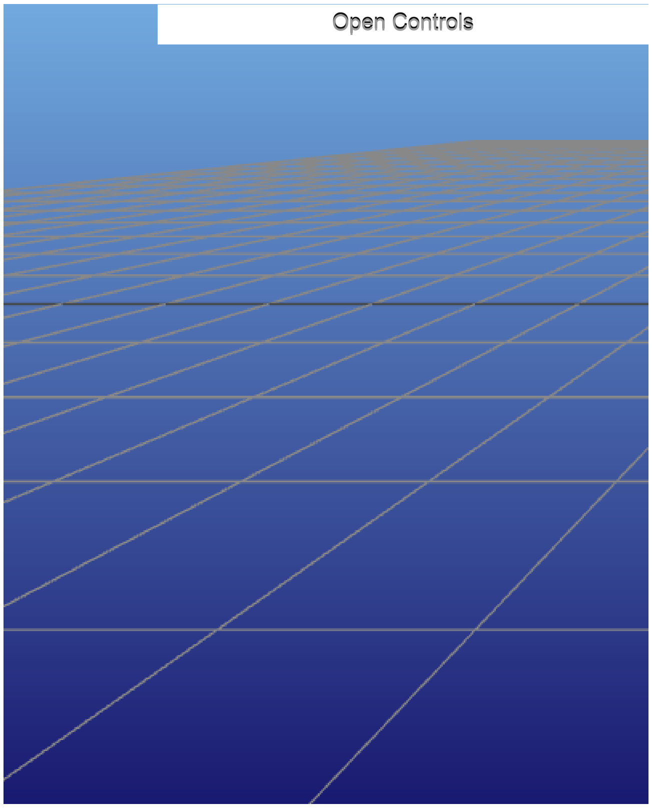


Attitude (MRP)
0.2
0.1
0.0
−0.1
−0.2
MRP
0
1
2
3
4
5
time (s)
p_x
p_y
p_z






```
┌ Info: MeshCat server started. You can open the visualizer by visiting the f
ollowing URL in your browser:
└ http://127.0.0.1:8700
```



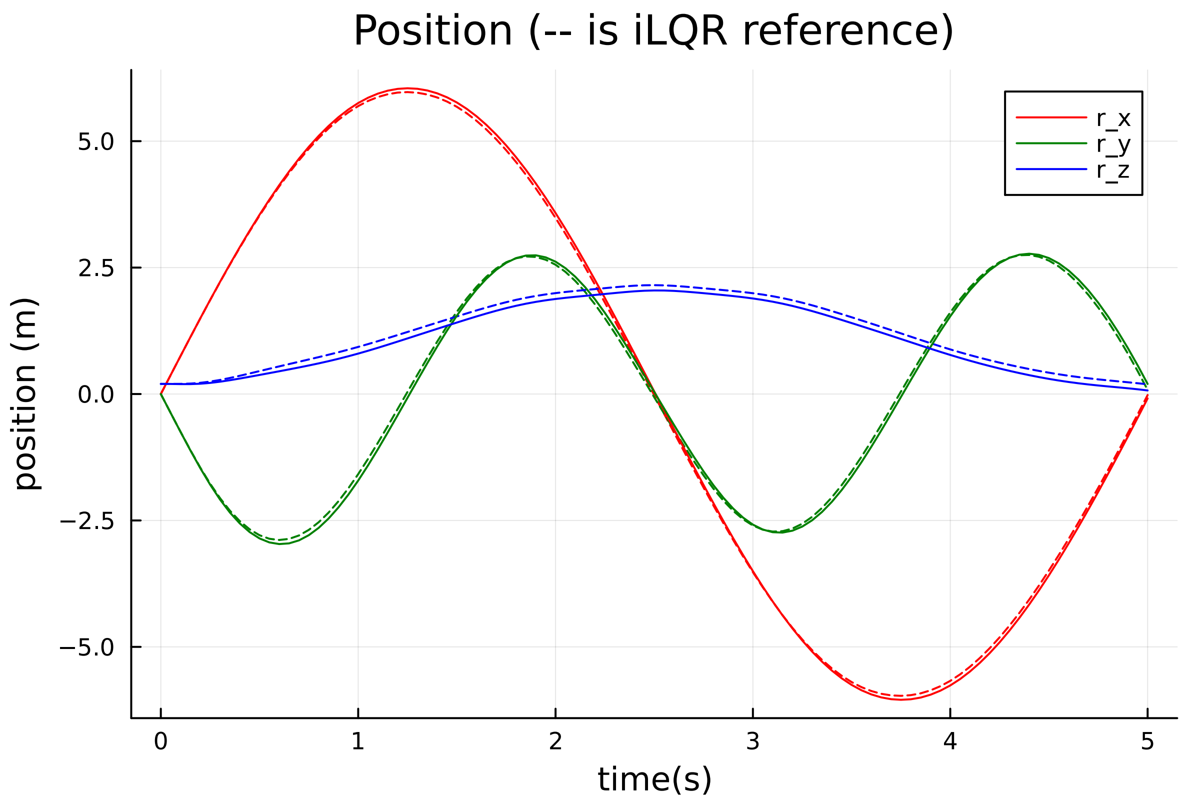

```
Test Summary: | Pass  Total
ilqr          |    1      1
```

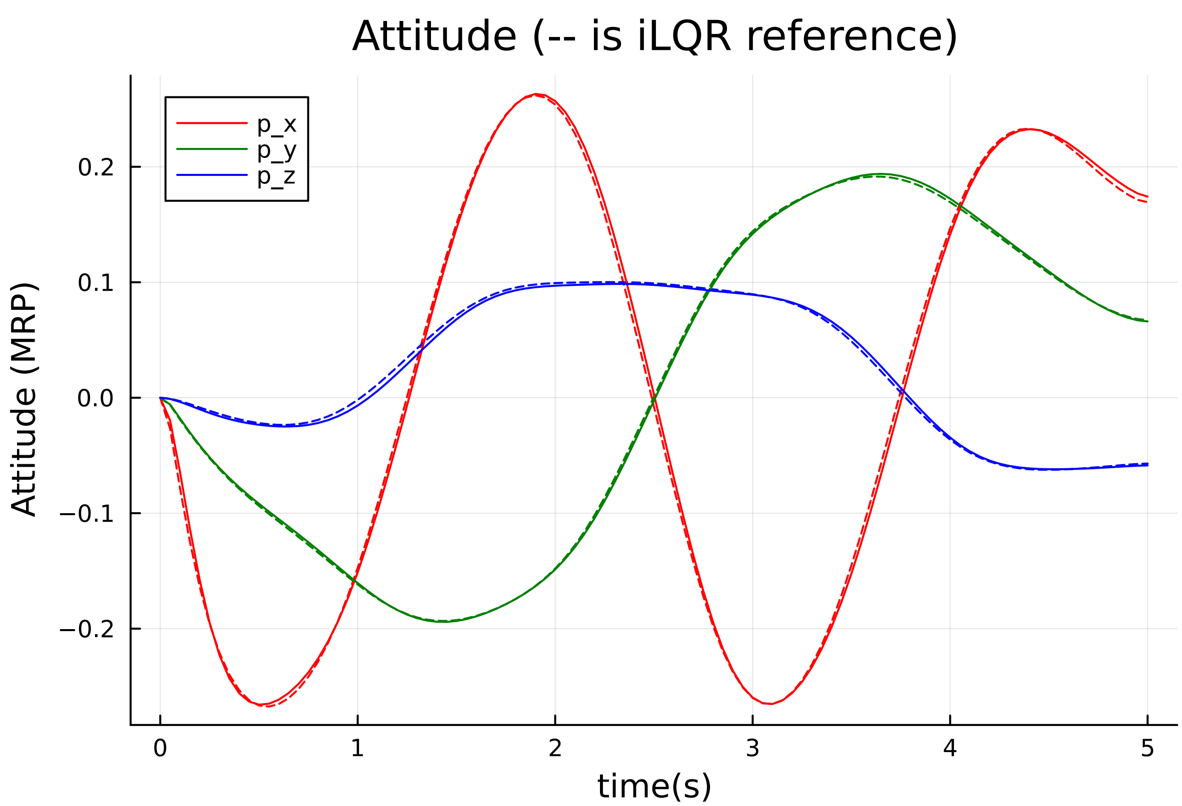
Out[12]: `Test.DefaultTestSet("ilqr", Any[], 1, false, false)`

Part B: Tracking solution with TVLQR (5 pts)

Here we will do the same thing we did in Q1 where we take a trajectory from a trajectory optimization solver, and track it with TVLQR to account for some model mismatch. In DIRCOL, we had to explicitly compute the TVLQR control gains, but in iLQR, we get these same gains out of the algorithmn as the K's. Use these to track the quadrotor through this manuever.

In [13]:
```
@testset "iLQR with model error" begin

    # set verbose to false when you submit
    Xilqr, Uilqr, Kilqr, t_vec, params =  solve_quadrotor_trajectory(verbose = 
false)

    # real model parameters for dynamics
    model_real = (mass=0.5,
            J=Diagonal([0.0025, 0.002, 0.0045]),
            gravity=[0,0,-9.81],
            L=0.1550,
            kf=0.9,
            km=0.0365,dt = 0.05)

    # simulate closed loop system
    nx, nu, N = params.nx, params.nu, params.N
    Xsim = [zeros(nx) for i = 1:N]
    Usim = [zeros(nx) for i = 1:(N-1)]

    # initial condition
    Xsim[1] = 1*Xilqr[1]

    # TODO: simulate with closed loop control
    for i = 1:(N-1)
        Usim[i] = -Kilqr[i]*(Xsim[i]-Xilqr[i])
        Xsim[i+1] = rk4(model_real, quadrotor_dynamics, Xsim[i], (Usim[i]+Uilq
r[i]), model_real.dt)
    end


    # -----------------testing--------------------
    @test 1e-6 <= norm(Xilqr[50] - Xsim[50],Inf) <= .3
    @test 1e-6 <= norm(Xilqr[end] - Xsim[end],Inf) <= .3

    # -----------------plotting--------------------
    Xm = hcat(Xsim...)
    Um = hcat(Usim...)
    Xilqrm = hcat(Xilqr...)
    Uilqrm = hcat(Uilqr...)
    plot(t_vec,Xilqrm[1:3,:]',ls=:dash, label = "",lc = [:red :green :blue])
    display(plot!(t_vec,Xm[1:3,:]',title = "Position (-- is iLQR reference)",
                 xlabel = "time(s)", ylabel = "position (m)",
                 label = ["r_x" "r_y" "r_z"],lc = [:red :green :blue]))

    plot(t_vec,Xilqrm[7:9,:]',ls=:dash, label = "",lc = [:red :green :blue])
    display(plot!(t_vec,Xm[7:9,:]',title = "Attitude (-- is iLQR reference)",
                 xlabel = "time(s)", ylabel = "Attitude (MRP)",
                 label = ["p_x" "p_y" "p_z"],lc = [:red :green :blue]))

    display(animate_quadrotor(Xilqr, params.Xref, params.model.dt))
end
```






```
┌ Info: MeshCat server started. You can open the visualizer by visiting the f
ollowing URL in your browser:
└ http://127.0.0.1:8702
```

Open Controls

```
Test Summary:         | Pass  Total
iLQR with model error |    2      2
```

Out[13]:
```
Test.DefaultTestSet("iLQR with model error", Any[], 2, false, false)
```

In [ ]:

In [1]:
```
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]:
```
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)
```

```
    # 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]:
```
"""
    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
```

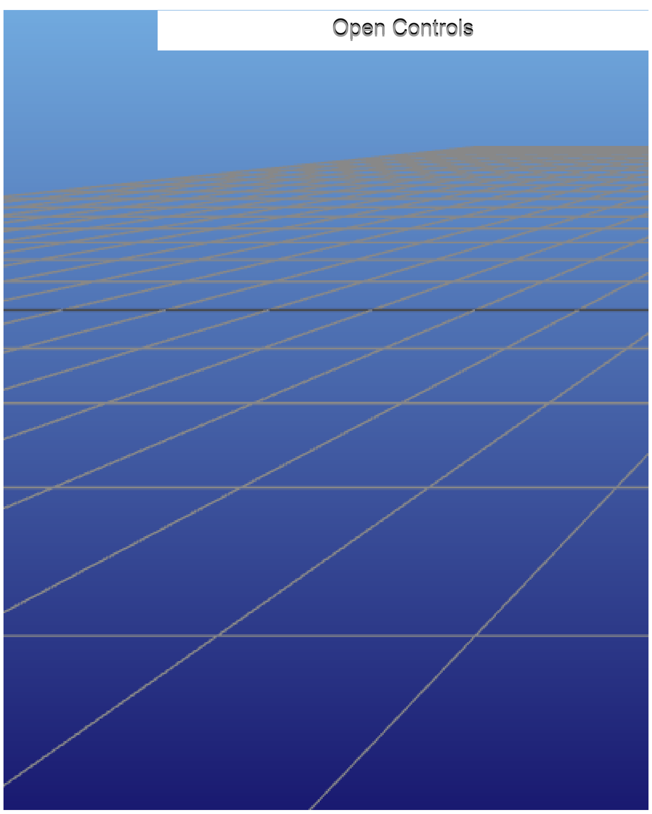
Out[14]: quadrotor_reorient

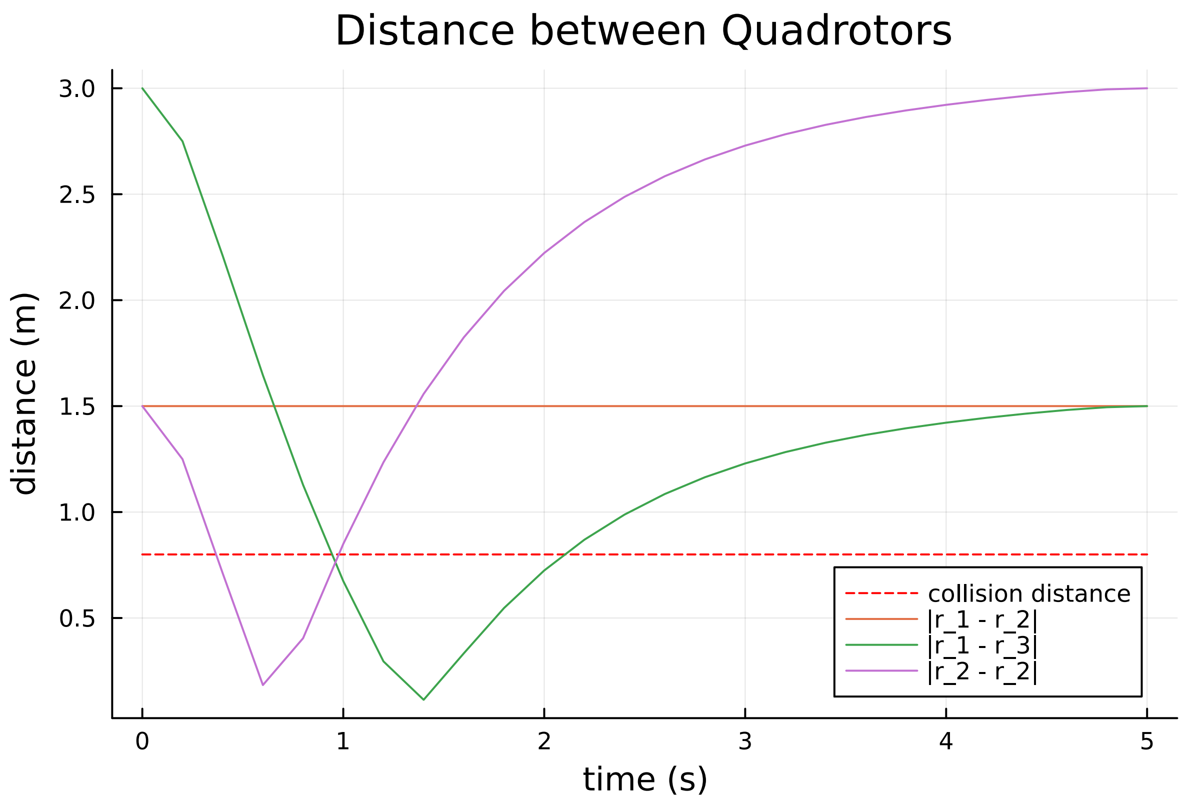
In [15]:

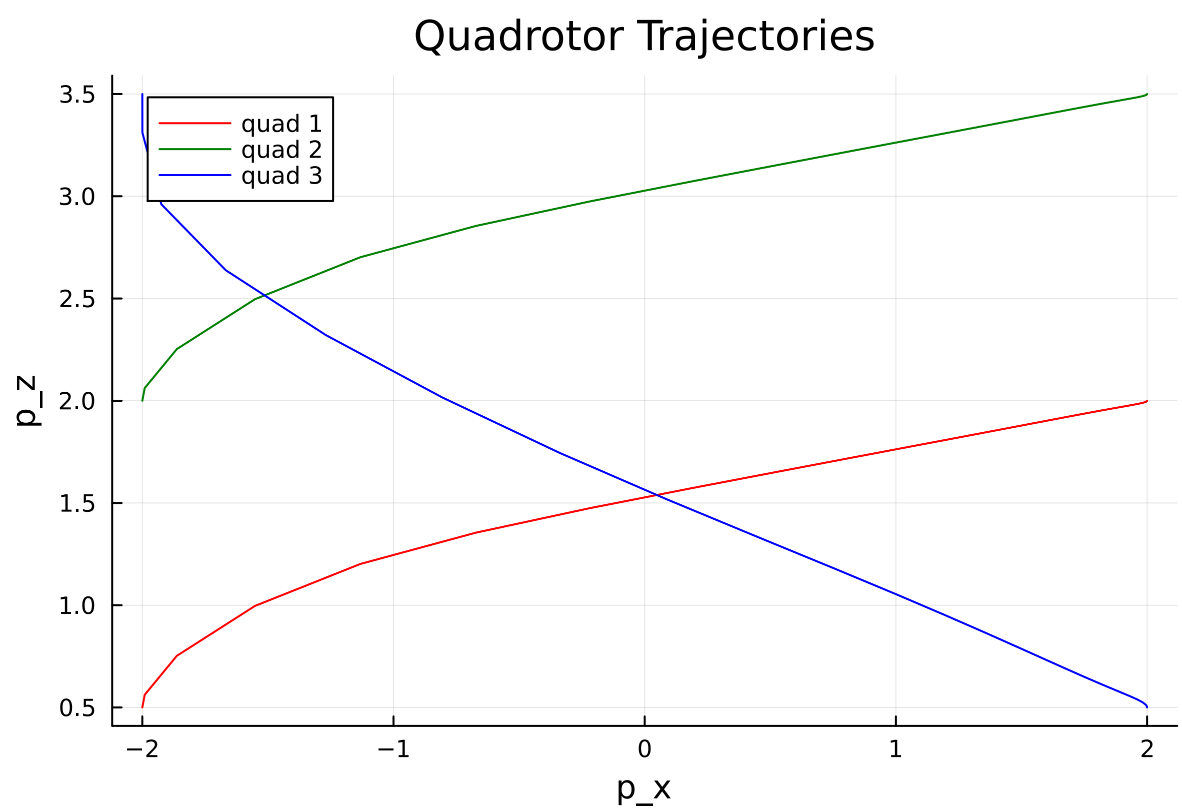
```
@testset "quadrotor reorient" begin

    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"))
```

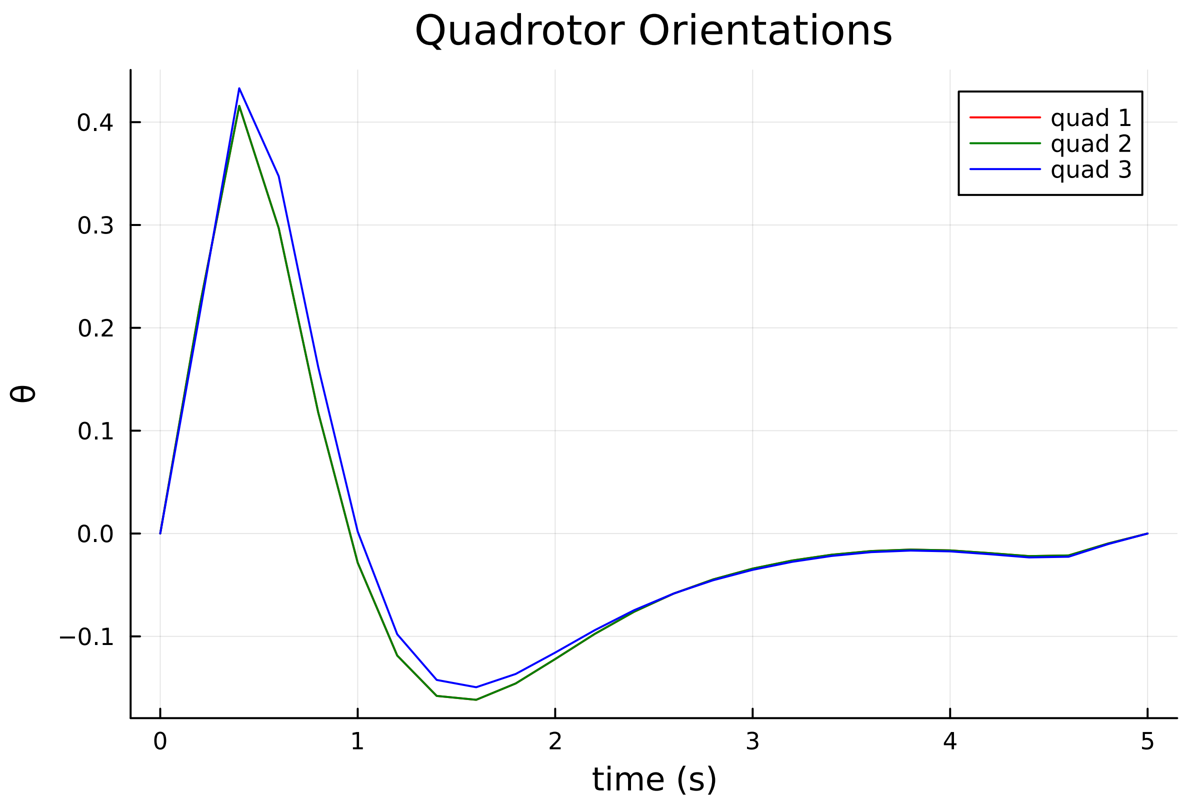
```
end
```

```
******************************************************************************
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
******************************************************************************

quadrotor reorient: Test Failed at In[15]:15
  Expression: minimum(minimum.(distances)) >= 0.799
   Evaluated: 0.11403010439325294 >= 0.799
Stacktrace:
 [1] macro expansion
   @ In[15]: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[15]:3

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




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


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




```
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[15]:3
```

In [ ]: