In [2]:
```
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
using LinearAlgebra, Plots
import ForwardDiff as FD
import MeshCat as mc
using Test
```

```
  Activating environment at `~/OCRL/HW1_S24/HW1/Project.toml`
  ✓ Observables
  ✓ Mustache
  ✓ FunctionalCollections
  ✓ AxisAlgorithms
  ✓ TranscodingStreams
  ✓ OffsetArrays
  ✓ Widgets
  ✓ FileIO
  ✓ Lazy
  ✓ WebIO
  ✓ Interpolations
  ✓ CSSUtil
  ✓ RigidBodyDynamics
  ✓ JSExpr
  ✓ MechanismGeometries
  ✓ Knockout
  ✓ Blink
  ✓ InteractBase
  ✓ MeshCat
  ✓ JLD2
  ✓ MeshCatMechanisms
  27 dependencies successfully precompiled in 11 seconds (163 already precom
piled)
  1 dependency precompiled but a different version is currently loaded. Rest
art julia to access the new version
```

Julia Warmup

Just like Python, Julia lets you do the following:

In [3]:
```
let
    x = [1,2,3]
    @show x
    y = x # NEVER DO THIS, EDITING ONE WILL NOW EDIT BOTH

    y[3] = 100 # this will now modify both y and x
    x[1] = 300 # this will now modify both y and x
```

```
    @show y
    @show x
end
```

```
x = [1, 2, 3]
y = [300, 2, 100]
x = [300, 2, 100]
```

Out[3]:
```
3-element Vector{Int64}:
 300
   2
 100
```

In [4]:
```
# to avoid this, here are two alternatives
let
    x = [1,2,3]
    @show x

    y1 = 1*x         # this is fine
    y2 = deepcopy(x) # this is also fine

    x[2] = 200 # only edits x
    y1[1] = 400 # only edits y1
    y2[3] = 100 # only edits y2

    @show x
    @show y1
    @show y2
end
```

```
x = [1, 2, 3]
x = [1, 200, 3]
y1 = [400, 2, 3]
y2 = [1, 2, 100]
```

Out[4]:
```
3-element Vector{Int64}:
   1
   2
 100
```

Optional function arguments

We can have optional keyword arguments for functions in Julia, like the following:

In [5]:
```
## optional arguments in functions

# we can have functions with optional arguments after a ; that have default
let
    function f1(a, b; c=4, d=5)
        @show a,b,c,d
    end

    f1(1,2)               # this means c and d will take on default value
    f1(1,2;c = 100,d = 2) # specify c and d
    f1(1,2;d = -30)       # or we can only specify one of them
end
```

```
(a, b, c, d) = (1, 2, 4, 5)
(a, b, c, d) = (1, 2, 100, 2)
(a, b, c, d) = (1, 2, 4, -30)
```

Out[5]: (1, 2, 4, -30)

Q1: Integration (25 pts)

In this question we are going to integrate the equations of motion for a double pendulum using multiple explicit and implicit integrators. We will write a generic simulation function for each of the two categories (explicit and implicit), and compare 6 different integrators.

The continuous time dynamics of the cartpole are written as a function:

$$\dot{x} = f(x)$$

In the code you will see `xdot = dynamics(params, x)` .

Part A (10 pts): Explicit Integration

Here we are going to implement the following explicit integrators:

- Forward Euler (explicit)
- Midpoint (explicit)
- RK4 (explicit)

In [6]:
```
# these two functions are given, no TODO's here
function double_pendulum_dynamics(params::NamedTuple, x::Vector)
    # continuous time dynamics for a double pendulum given state x,
    # also known as the "equations of motion".
    # returns the time derivative of the state, ẋ (dx/dt)

    # the state is the following:
    θ1,θ1,θ2,θ2 = x

    # system parameters
    m1, m2, L1, L2, g = params.m1, params.m2, params.L1, params.L2, params.g

    # dynamics
    c = cos(θ1-θ2)
    s = sin(θ1-θ2)

    ẋ = [
        θ1;
        ( m2*g*sin(θ2)*c - m2*s*(L1*c*θ1^2 + L2*θ2^2) - (m1+m2)*g*sin(θ1) )
        θ2;
        ((m1+m2)*(L1*θ1^2*s - g*sin(θ2) + g*sin(θ1)*c) + m2*L2*θ2^2*s*c) / (
        ]

    return ẋ
end
function double_pendulum_energy(params::NamedTuple, x::Vector)::Real
```

```
    # calculate the total energy (kinetic + potential) of a double pendulum


    # the state is the following:
    θ1,ω1,θ2,ω2 = x

    # system parameters
    m1, m2, L1, L2, g = params.m1, params.m2, params.L1, params.L2, params.g

    # cartesian positions/velocities of the masses
    r1 = [L1*sin(θ1), 0, -params.L1*cos(θ1) + 2]
    r2 = r1 + [params.L2*sin(θ2), 0, -params.L2*cos(θ2)]
    v1 = [L1*ω1*cos(θ1), 0, L1*ω1*sin(θ1)]
    v2 = v1 + [L2*ω2*cos(θ2), 0, L2*ω2*sin(θ2)]

    # energy calculation
    kinetic = 0.5*(m1*v1'*v1 + m2*v2'*v2)
    potential = m1*g*r1[3] + m2*g*r2[3]
    return kinetic + potential
end
```

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

Now we are going to simulate this double pendulum by integrating the equations of motion with the simplest explicit integrator, the Forward Euler method:

$$x_{k+1} = x_k + \Delta t \cdot f(x_k) \qquad \text{Forward Euler (explicit)}$$

In [7]:
```
"""
    x_{k+1} = forward_euler(params, dynamics, x_k, dt)

Given `ẋ = dynamics(params, x)`, take in the current state `x` and integrate
using Forward Euler method.
"""
function forward_euler(params::NamedTuple, dynamics::Function, x::Vector, dt
    ẋ = dynamics(params, x)

    # TODO: implement forward euler
    x_k = x + dt*dynamics(params,x)
    # error("forward euler not implemented")
    return x_k
end
```

Out[7]: forward_euler

In [8]:
```
include(joinpath(@__DIR__, "animation.jl"))

let

    # parameters for the simulation
    params = (
        m1 = 1.0,
        m2 = 1.0,
        L1 = 1.0,
```

```
        L2 = 1.0,
        g = 9.8
    )

    # initial condition
    x0 = [pi/1.6; 0; pi/1.8; 0]

    # time step size (s)
    dt = 0.01
    tf = 30.0
    t_vec = 0:dt:tf
    N = length(t_vec)

    # store the trajectory in a vector of vectors
    X = [zeros(4) for i = 1:N]
    X[1] = 1*x0

    # TODO: simulate the double pendulum with `forward_euler`
    # X[k] = `x_k`, so X[k+1] = forward_euler(params, double_pendulum_dynami
    for k = 1:N+1
        # X[k] = x_k
        X[k+1] = forward_euler(params, double_pendulum_dynamics(params,X[k])
    end

    # calculate energy
    E = [double_pendulum_energy(params,x) for x in X]

    @show @test norm(X[end]) > 1e-10   # make sure all X's were updated
    @show @test 2 < (E[end]/E[1]) < 3  # energy should be increasing

    # plot state history, energy history, and animate it
    display(plot(t_vec, hcat(X...)',xlabel = "time (s)", label = ["θ₁" "θi d
    display(plot(t_vec, E, xlabel = "time (s)", ylabel = "energy (J)"))
    meshcat_animate(params,X,dt,N)


end
```

```
MethodError: no method matching forward_euler(::NamedTuple{(:m1, :m2, :L1, :
L2, :g), NTuple{5, Float64}}, ::Vector{Float64}, ::Vector{Float64}, ::Float6
4)
Closest candidates are:
  forward_euler(::NamedTuple, ::Function, ::Vector{T} where T, ::Real) at In
[7]:7

Stacktrace:
 [1] top-level scope
   @ In[8]:31
```

Now let's implement the next two integrators:

Midpoint:

$$x_m = x_k + \frac{\Delta t}{2} \cdot f(x_k) \tag{1}$$

$$x_{k+1} = x_k + \Delta t \cdot f(x_m) \tag{2}$$

RK4:

$$k_1 = \Delta t \cdot f(x_k) \tag{3}$$
$$k_2 = \Delta t \cdot f(x_k + k_1/2) \tag{4}$$
$$k_3 = \Delta t \cdot f(x_k + k_2/2) \tag{5}$$
$$k_4 = \Delta t \cdot f(x_k + k_3) \tag{6}$$
$$x_{k+1} = x_k + (1/6) \cdot (k_1 + 2k_2 + 2k_3 + k_4) \tag{7}$$

In [9]:
```
function midpoint(params::NamedTuple, dynamics::Function, x::Vector, dt::Rea
    # TODO: implement explicit midpoint
    x_m = x + (dt/2)*dynamics(params,x)
    x_k = x + dt*dynamics(params,x_m)
    return x_k
end

function rk4(params::NamedTuple, dynamics::Function, x::Vector, dt::Real)::V
    # TODO: implement RK4
    k1 = dt*dynamics(params,x)
    k2 = dt*dynamics(params,(x+k1/2))
    k3 = dt*dynamics(params,(x+k2/2))
    k4 = dt*dynamics(params,(x+k3))
    x_rk = x + (1/6)*(k1 + 2*k2 + 2*k3 + k4)
    return x_rk
end
```

Out[9]:
```
rk4 (generic function with 1 method)
```

In [10]:
```
function simulate_explicit(params::NamedTuple,dynamics::Function,integrator:
    # TOOD: update this function to simulate dynamics forward
    # with the given explicit integrator


    # take in
    t_vec = 0:dt:tf
    N = length(t_vec)
    X = [zeros(length(x0)) for i = 1:N]
    X[1] = x0

    # TODO: simulate X forward
    for k = 1:N-1
        # X[k] = x_k
        X[k+1] = integrator(params, dynamics, X[k], dt)
    end

    # return state history X and energy E
    E = [double_pendulum_energy(params,x) for x in X]
    return X, E
end
```

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

In [11]:
```
# initial condition
const x0 = [pi/1.6; 0; pi/1.8; 0]
```

```
const params = (
    m1 = 1.0,
    m2 = 1.0,
    L1 = 1.0,
    L2 = 1.0,
    g = 9.8
)
```

Out[11]: (m1 = 1.0, m2 = 1.0, L1 = 1.0, L2 = 1.0, g = 9.8)

Part B (10 pts): Implicit Integrators

Explicit integrators work by calling a function with x_k and Δt as arguments, and returning x_{k+1} like this:

$$x_{k+1} = f_{explicit}(x_k, \Delta t)$$

Implicit integrators on the other hand have the following relationship between the state at x_k and x_{k+1}:

$$f_{implicit}(x_k, x_{k+1}, \Delta t) = 0$$

This means that if we want to get x_{k+1} from x_k, we have to solve for a x_{k+1} that satisfies the above equation. This is a rootfinding problem in x_{k+1} (our unknown), so we juse have to use Newton's method.

Here are the three implicit integrators we are looking at, the first being Backward Euler (1st order):

$$f(x_k, x_{k+1}, \Delta t) = x_k + \Delta t \cdot \dot{x}_{k+1} - x_{k+1} = 0 \qquad \text{Backward Euler}$$

Implicit Midpoint (2nd order)

$$x_{k+1/2} = \frac{1}{2}(x_k + x_{k+1}) \tag{8}$$

$$f(x_k, x_{k+1}, \Delta t) = x_k + \Delta t \cdot \dot{x}_{k+1/2} - x_{k+1} = 0 \qquad \text{Implicit Midpoint} \tag{9}$$

Hermite Simpson (3rd order)

$$x_{k+1/2} = \frac{1}{2}(x_k + x_{k+1}) + \frac{\Delta t}{8}(\dot{x}_k - \dot{x}_{k+1})$$

$$f(x_k, x_{k+1}, \Delta t) = x_k + \frac{\Delta t}{6} \cdot (\dot{x}_k + 4\dot{x}_{k+1/2} + \dot{x}_{k+1}) - x_{k+1} = 0 \qquad \text{Hermite-Sim}$$

When you implement these integrators, you will update the functions such that they take in a dynamics function, x_k and x_{k+1}, and return the residuals described above. We are NOT solving these yet, we are simply returning the residuals for each implicit integrator that we want to be 0.

In [12]:
```
# since these are explicit integrators, these function will return the resid
# NOTE: we are NOT solving anything here, simply return the residuals
function backward_euler(params::NamedTuple, dynamics::Function, x1::Vector,
    residual = x1 + dt*dynamics(params,x2) - x2
    return residual
end
function implicit_midpoint(params::NamedTuple, dynamics::Function, x1::Vecto
    residual_im = x1 +dt*dynamics(params,(0.5*(x1+x2))) - x2
    return residual_im
end
function hermite_simpson(params::NamedTuple, dynamics::Function, x1::Vector,
    x_dot1 = dynamics(params,x1)
    x_dot2 = dynamics(params,x2)

    x_k2 = 0.5*(x1+x2) + (dt/8)*( x_dot1 - x_dot2)
    x_dotk2 = dynamics(params,x_k2)

    residual_hms = x1 + (dt/6)*(x_dot1 + 4*x_dotk2 + x_dot2) - x2
    return residual_hms
end
```

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

In [13]:
```
# TODO
# this function takes in a dynamics function, implicit integrator function,
# and uses Newton's method to solve for an x2 that satsifies the implicit in
# that we wrote about in the functions above
function implicit_integrator_solve(params::NamedTuple, dynamics::Function, i

    # initialize guess
    x2 = 1*x1

    # TODO: use Newton's method to solve for x2 such that residual for the i
    # DO NOT USE A WHILE LOOP
    for i = 1:max_iters
        residual = implicit_integrator(params,dynamics,x1,x2,dt)

        if norm(residual) < tol
            return x2
        end
        Jacobian_fx = FD.jacobian(x -> implicit_integrator(params,dynamics,x
        del_x = - inv(Jacobian_fx) * residual #Backslash is used for the inv
        x2 += del_x
    end
    return x2
end
```

Out[13]: implicit_integrator_solve (generic function with 1 method)

In [14]:
```
@testset "implicit integrator check" begin

    dt = 1e-1
    x1 = [.1,.2,.3,.4]

    for integrator in [backward_euler, implicit_midpoint, hermite_simpson]
```

```
        println("-----testing $integrator ------")
        x2 = implicit_integrator_solve(params, double_pendulum_dynamics, int
        @test norm(integrator(params, double_pendulum_dynamics, x1, x2, dt))
    end

end
```

```
-----testing backward_euler ------
-----testing implicit_midpoint ------
-----testing hermite_simpson ------
Test Summary:              | Pass  Total
implicit integrator check |    3      3
```

Out[14]: Test.DefaultTestSet("implicit integrator check", Any[], 3, false, false)

Test.DefaultTestSet("implicit integrator check", Any[], 3, false, false)

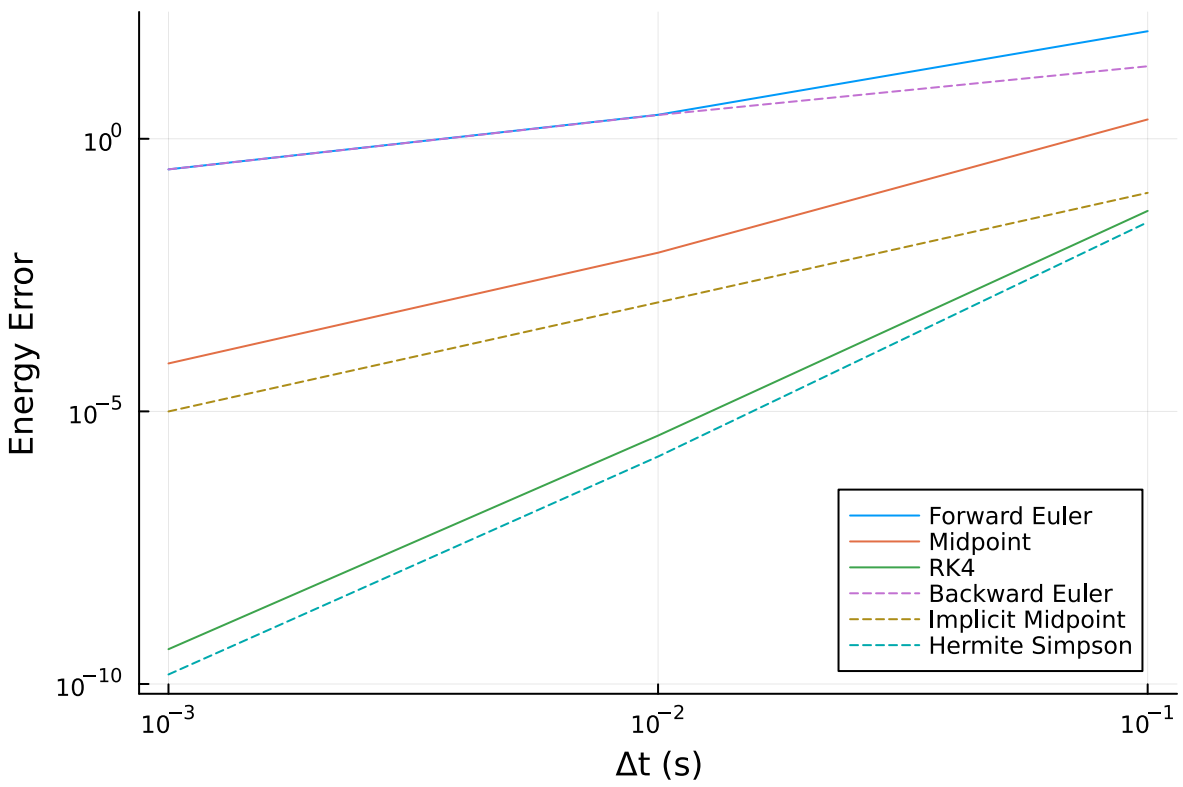
In [15]:
```
function simulate_implicit(params::NamedTuple,dynamics::Function,implicit_in
    t_vec = 0:dt:tf
    N = length(t_vec)
    X = [zeros(length(x0)) for i = 1:N]
    X[1] = x0

    # TODO: do a forward simulation with the selected implicit integrator
    # hint: use your `implicit_integrator_solve` function
    for k = 1:N-1
        X[k+1] = implicit_integrator_solve(params, dynamics, implicit_integr
    end
    E = [double_pendulum_energy(params,x) for x in X]
    @assert length(X)==N
    @assert length(E)==N
    return X, E
end
```

Out[15]: simulate_implicit (generic function with 1 method)

In [16]:
```
function max_err_E(E)
    E0 = E[1]
    err = abs.(E .- E0)
    return maximum(err)
end
function get_explicit_energy_error(integrator::Function, dts::Vector)
    [max_err_E(simulate_explicit(params,double_pendulum_dynamics,integrator,
end
function get_implicit_energy_error(integrator::Function, dts::Vector)
    [max_err_E(simulate_implicit(params,double_pendulum_dynamics,integrator,
end


const tf = 2.0
let
    # here we compare everything
    dts = [1e-3,1e-2,1e-1]
    explicit_integrators = [forward_euler, midpoint, rk4]
    implicit_integrators = [backward_euler, implicit_midpoint, hermite_simps

    explicit_data = [get_explicit_energy_error(integrator, dts) for integrat
    implicit_data = [get_implicit_energy_error(integrator, dts) for integrat
```

```
    plot(dts, hcat(explicit_data...),label = ["Forward Euler" "Midpoint" "RK
    plot!(dts, hcat(implicit_data...),ls = :dash, label = ["Backward Euler"
    plot!(legend=:bottomright)
end
```

Out[16]:

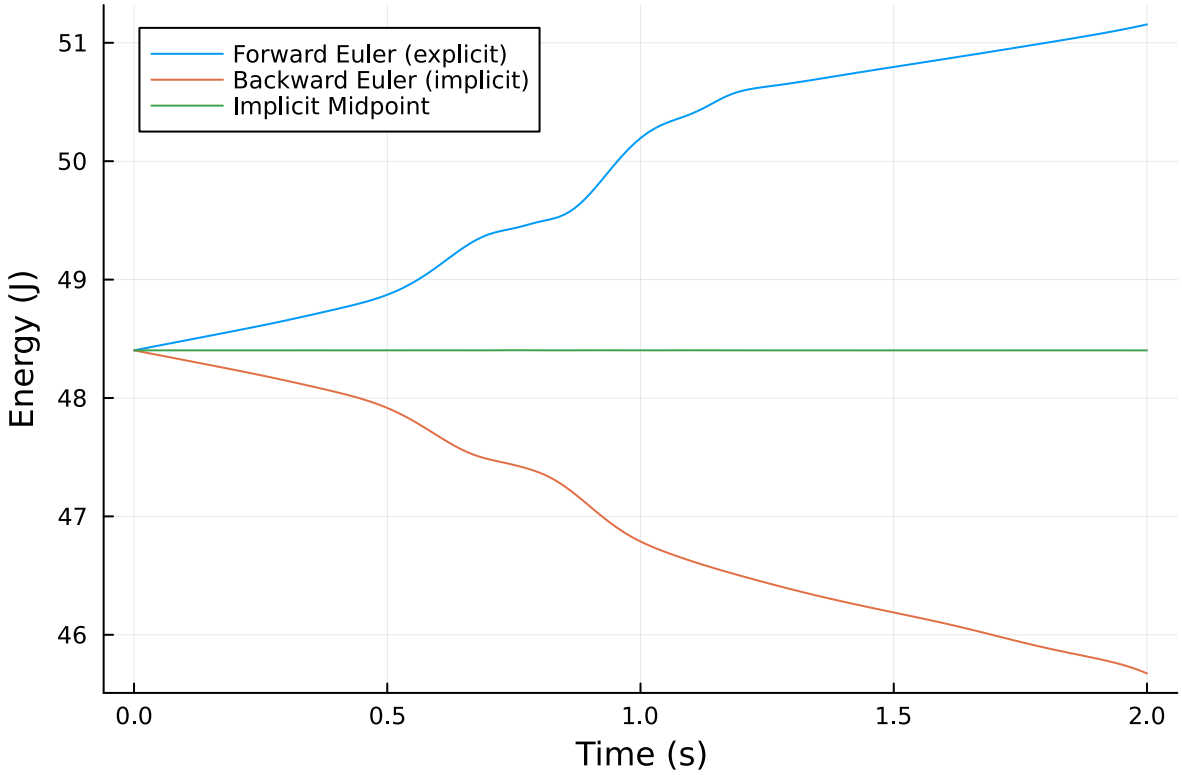


What we can see above is the maximum energy error for each of the integration methods. In general, the implicit methods of the same order are slightly better than the explicit ones.

In [17]:

```
@testset "energy behavior" begin

    # simulate with all integrators
    dt = 0.01
    t_vec = 0:dt:tf
    E1 = simulate_explicit(params,double_pendulum_dynamics,forward_euler,x0,
    E2 = simulate_implicit(params,double_pendulum_dynamics,backward_euler,x0
    E3 = simulate_implicit(params,double_pendulum_dynamics,implicit_midpoint
    E4 = simulate_implicit(params,double_pendulum_dynamics,hermite_simpson,x
    E5 = simulate_explicit(params,double_pendulum_dynamics,midpoint,x0,dt,tf
    E6 = simulate_explicit(params,double_pendulum_dynamics,rk4,x0,dt,tf)[2]

    # plot forward/backward euler and implicit midpoint
    plot(t_vec,E1, label = "Forward Euler (explicit)")
    plot!(t_vec,E2, label = "Backward Euler (implicit)")
    display(plot!(t_vec,E3, label = "Implicit Midpoint",xlabel = "Time (s)",

    # test energy behavior
    E0 = E1[1]

    @test 2.5  < (E1[end] - E0) < 3.0
```

```
    @test -3.0 < (E2[end] - E0) < -2.5
    @test abs(E3[end] - E0) < 1e-2
    @test abs(E0 - E4[end]) < 1e-4
    @test abs(E0 - E5[end]) < 1e-1
    @test abs(E0 - E6[end]) < 1e-4
end
```




```
Test Summary:    | Pass  Total
energy behavior |    6      6
```

Out[17]: `Test.DefaultTestSet("energy behavior", Any[], 6, false, false)`

Another important takeaway from these integrators is that explicit Euler results in unstable behavior (as shown here by the growing energy), and implicit Euler results in artificial damping (losing energy). Implicit midpoint however maintains the correct energy. Even though the solution from implicit midpoint will vary from the initial energy, it does not move secularly one way or the other.

Part C (5 pts): One sentence short answer

1. Describe the energy behavior of each integrator. Are there any that are clearly unstable?

Put ONE SENTENCE answer here

The most unstable would be the Forward Euler (explicit) as it is clearly uncontrolled growing energy, indicating that it is the most unstable.

In [ ]:

In [5]:
```
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
using LinearAlgebra, Plots
import ForwardDiff as FD
using MeshCat
using Test
using Plots
```

```
  Activating environment at `~/OCRL/HW1_S24/Project.toml`
```

Q2: Equality Constrained Optimization (25 pts)

In this problem, we are going to use Newton's method to solve some constrained optimization problems. We will start with a smaller problem where we can experiment with Full Newton vs Gauss-Newton, then we will use these methods to solve for the motor torques that make a quadruped balance on one leg.

Part A (10 pts)

Here we are going to solve some equality-constrained optimization problems with Newton's method. We are given a problem

$$\min_x \quad f(x) \tag{1}$$
$$\text{st} \quad c(x) = 0 \tag{2}$$

Which has the following Lagrangian:

$$\mathcal{L}(x, \lambda) = f(x) + \lambda^T c(x),$$

and the following KKT conditions for optimality:

$$\nabla_x \mathcal{L} = \nabla_x f(x) + \left[\frac{\partial c}{\partial x}\right]^T \lambda = 0 \tag{3}$$
$$c(x) = 0 \tag{4}$$

Which is just a root-finding problem. To solve this, we are going to solve for a $z = [x^T, \lambda]^T$ that satisfies these KKT conditions.

Newton's Method with a Linesearch

We use Newton's method to solve for when $r(z) = 0$. To do this, we specify `res_fx(z)` as $r(z)$, and `res_jac_fx(z)` as $\partial r / \partial z$. To calculate a Newton step, we do the following:

$$\Delta z = -\left[\frac{\partial r}{\partial z}\right]^{-1} r(z_k)$$

We then decide the step length with a linesearch that finds the largest $\alpha \leq 1$ such that the following is true:

$$\phi(z_k + \alpha \Delta z) < \phi(z_k)$$

Where ϕ is a "merit function", or `merit_fx(z)` in the code. In this assignment you will use a backtracking linesearch where α is initialized as $\alpha = 1.0$, and is divided by 2 until the above condition is satisfied.

NOTE: YOU DO NOT NEED TO (AND SHOULD NOT) USE A WHILE LOOP ANYWHERE IN THIS ASSIGNMENT.

In [8]:
```
function linesearch(z::Vector, Δz::Vector, merit_fx::Function;
                    max_ls_iters = 10)::Float64 # optional argument with a d

    # TODO: return maximum α≤1 such that merit_fx(z + α*Δz) < merit_fx(z)
    # with a backtracking linesearch (α = α/2 after each iteration)

    # NOTE: DO NOT USE A WHILE LOOP
    α = 1
    for i = 1:max_ls_iters
        # TODO: return α when merit_fx(z + α*Δz) < merit_fx(z)
        phi = merit_fx(z + α*Δz)
        if phi < merit_fx(z)
            return α
        else
            α = α/2
        end
    end
    # error("linesearch failed")
end

function newtons_method(z0::Vector, res_fx::Function, res_jac_fx::Function,
                        tol = 1e-10, max_iters = 50, verbose = false)::Vecto

    # TODO: implement Newton's method given the following inputs:
    # - z0, initial guess
    # - res_fx, residual function
    # - res_jac_fx, Jacobian of residual function wrt z
    # - merit_fx, merit function for use in linesearch

    # optional arguments
    # - tol, tolerance for convergence. Return when norm(residual)<tol
    # - max iter, max # of iterations
    # - verbose, bool telling the function to output information at each ite

    # return a vector of vectors containing the iterates
    # the last vector in this vector of vectors should be the approx. soluti
```

```
    # NOTE: DO NOT USE A WHILE LOOP ANYWHERE

    # return the history of guesses as a vector
    Z = [zeros(length(z0)) for i = 1:max_iters]
    Z[1] = z0

    for i = 1:(max_iters - 1)

        # NOTE: everything here is a suggestion, do whatever you want to

        # TODO: evaluate current residual
        current_residual = res_fx(Z[i])
        norm_r = norm(current_residual, Inf) # TODO: update this

        if verbose
            print("iter: $i    |r|: $norm_r   ")
        end

        # TODO: check convergence with norm of residual < tol
        # if converged, return Z[1:i]
        if norm_r < tol
            return Z[1:i]
        end

        # TODO: caculate Newton step (don't forget the negative sign)
        Δz = - inv(res_jac_fx(Z[i])) * current_residual #Backslash is used f

        # TODO: linesearch and update z
        α = linesearch(Z[i], Δz,merit_fx)
        Z[i+1] = Z[i] + α*Δz
        if verbose
            print("α: $α \n")
        end

    end
    error("Newton's method did not converge")
end
```

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

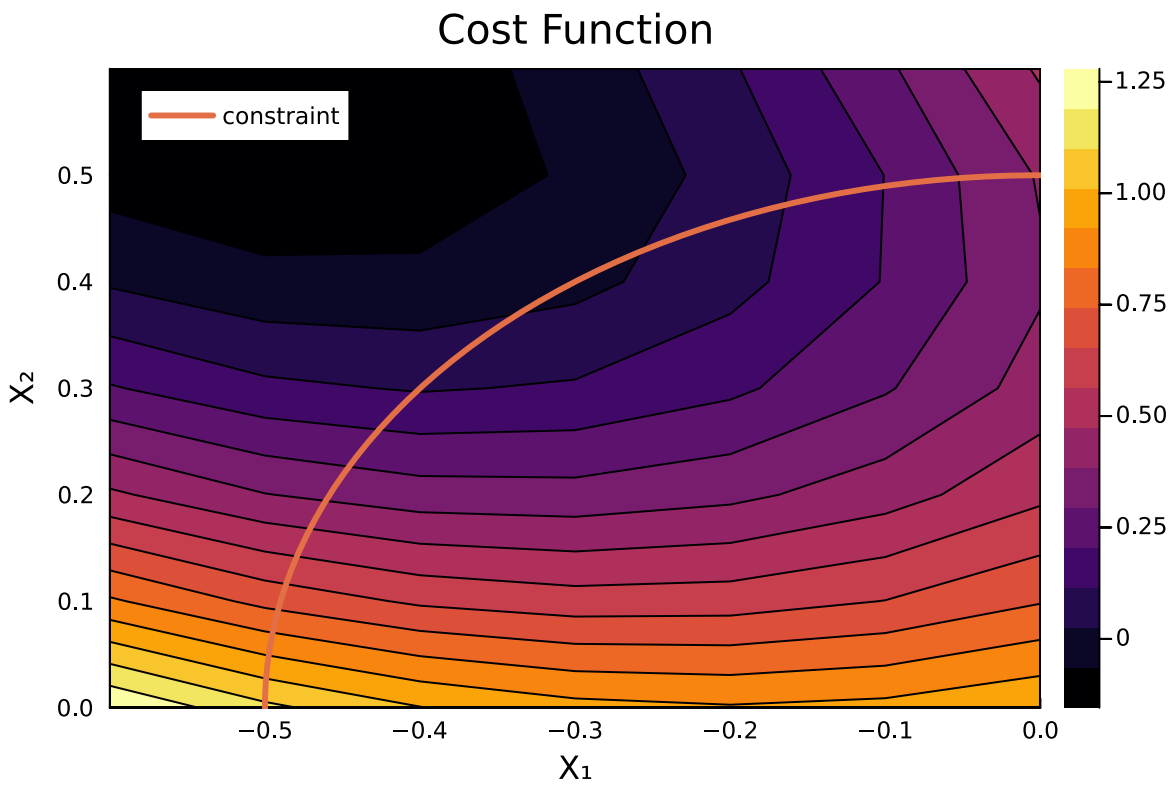
In [9]:

```
@testset "check Newton" begin

    f(_x) = [sin(_x[1]), cos(_x[2])]
    df(_x) = FD.jacobian(f, _x)
    merit(_x) = norm(f(_x))

    x0 = [-1.742410372590328, 1.4020334125022704]

    X = newtons_method(x0, f, df, merit; tol = 1e-10, max_iters = 50, verbos

    # check this took the correct number of iterations
    # if your linesearch isn't working, this will fail
    # you should see 1 iteration where α = 0.5
    @test length(X) == 6

    # check we actually converged
```

```
    @test norm(f(X[end])) < 1e-10

end
```

```
iter: 1    |r|: 0.9853104151741932   α: 1.0
iter: 2    |r|: 0.9421328488163083   α: 0.5
iter: 3    |r|: 0.17531541822305458   α: 1.0
iter: 4    |r|: 0.0018472215879181202   α: 1.0
iter: 5    |r|: 2.1010529101114835e-9   α: 1.0
iter: 6    |r|: 2.4492935982947064e-16   Test Summary: | Pass  Total
check Newton  |    2      2
Test.DefaultTestSet("check Newton", Any[], 2, false, false)
```

In [10]:
```
let
    function plotting_cost(x::Vector)
        Q = [1.65539  2.89376; 2.89376  6.51521];
        q = [2;-3]
        return 0.5*x'*Q*x + q'*x + exp(-1.3*x[1] + 0.3*x[2]^2)
    end
    contour(-.6:.1:0,0:.1:.6, (x1,x2)-> plotting_cost([x1;x2]),title = "Cost
            xlabel = "X₁", ylabel = "X₂",fill = true)
    xcirc = [.5*cos(θ) for θ in range(0, 2*pi, length = 200)]
    ycirc = [.5*sin(θ) for θ in range(0, 2*pi, length = 200)]
    plot!(xcirc,ycirc, lw = 3.0, xlim = (-.6, 0), ylim = (0, .6),label = "co
end
```



We will now use Newton's method to solve the following constrained optimization problem. We will write functions for the full Newton Jacobian, as well as the Gauss-Newton Jacobian.

In [49]:
```
# we will use Newton's method to solve the constrained optimization problem
```

```
function cost(x::Vector)
    Q = [1.65539  2.89376; 2.89376  6.51521];
    q = [2;-3]
    return 0.5*x'*Q*x + q'*x + exp(-1.3*x[1] + 0.3*x[2]^2)
end
function constraint(x::Vector)
    norm(x) - 0.5
end
# HINT: use this if you want to, but you don't have to
function constraint_jacobian(x::Vector)::Matrix
    # since `constraint` returns a scalar value, ForwardDiff
    # will only allow us to compute a gradient of this function
    # (instead of a Jacobian). This means we have two options for
    # computing the Jacobian: Option 1 is to just reshape the gradient
    # into a row vector

    # J = reshape(FD.gradient(constraint, x), 1, 2)

    # or we can just make the output of constraint an array,
    constraint_array(_x) = [constraint(_x)]
    J = FD.jacobian(constraint_array, x)

    # assert the jacobian has # rows = # outputs
    # and # columns = # inputs
    @assert size(J) == (length(constraint(x)), length(x))

    return J
end
function kkt_conditions(z::Vector)::Vector
    # TODO: return the KKT conditions

    x = z[1:2]
    λ = z[3:3]
    kkt_conditions = zeros(length(z))

    # TODO: return the stationarity condition for the cost function
    c = cost(x) # cost function
    ∇c = FD.gradient(cost,x)
    J = constraint_jacobian(x)
    kkt_conditions[1:2] = ∇c .+ J'*λ
    kkt_conditions[3] = constraint(x)
    # and the primal feasibility

    return kkt_conditions
end

function fn_kkt_jac(z::Vector)::Matrix
    # TODO: return full Newton Jacobian of kkt conditions wrt z
    x = z[1:2]
    λ = z[3]
    J_fn_kkt = zeros(2,2)
    reg = 1e-3
    # c = cost(dx)
    L(dx) = cost(dx) .+ λ'*constraint(dx)  #Lagrangian term
    J = constraint_jacobian(x)
    # TODO: return full Newton jacobian with a 1e-3 regularizer
```

```
    J_fn_kkt = [(FD.hessian(dx -> L(dx), x) + reg*I) J'; J -reg*I]
    return J_fn_kkt
end


function gn_kkt_jac(z::Vector)::Matrix
    # TODO: return Gauss-Newton Jacobian of kkt conditions wrt z
    x = z[1:2]
    λ = z[3]

    # TODO: return Gauss-Newton jacobian with a 1e-3 regularizer
    J_gn_kkt = zeros(2,2)
    reg = 1e-3
    # c = cost(dx)
    L(dx) = cost(dx) .+ λ'*constraint(dx)  #Lagrangian term
    J = constraint_jacobian(x)
    J_gn_kkt = [(FD.hessian(dx -> cost(dx), x) + reg*I) J'; J -reg*I]
    return J_gn_kkt
end
```

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

In [94]:

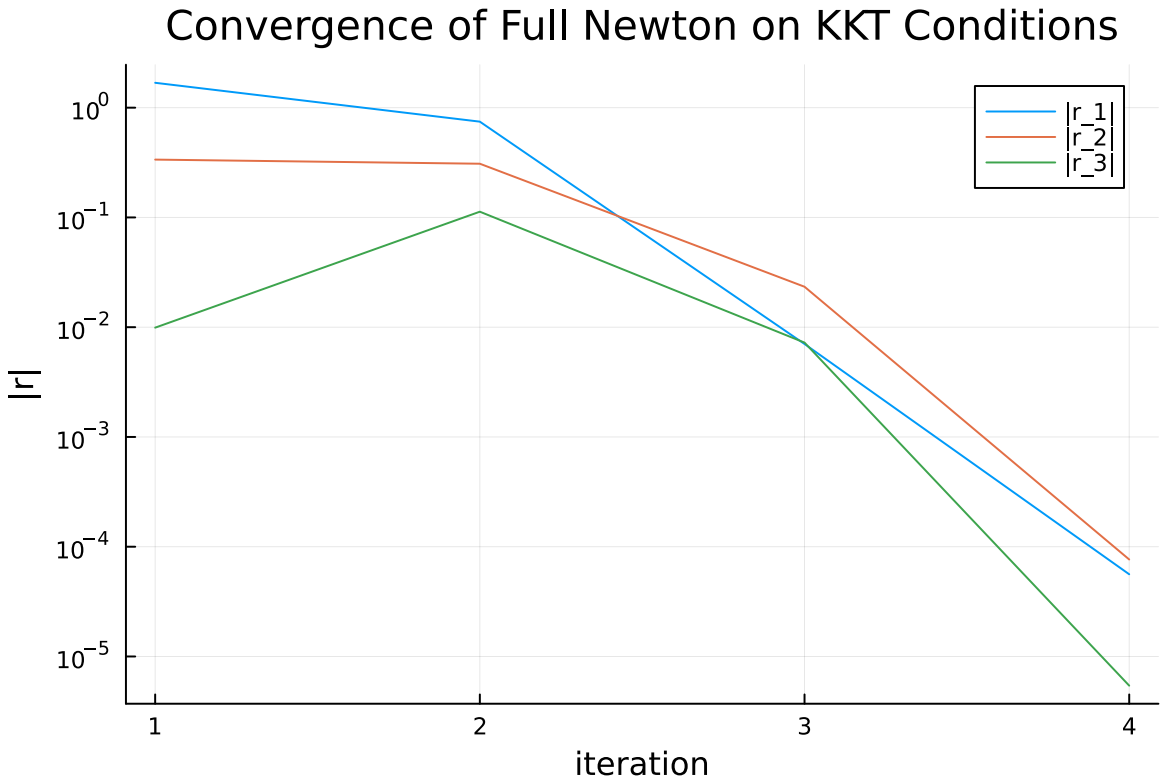
```
@testset "Test Jacobians" begin

    # first we check the regularizer
    z = randn(3)
    J_fn = fn_kkt_jac(z)
    J_gn = gn_kkt_jac(z)

    # check what should/shouldn't be the same between
    @test norm(J_fn[1:2,1:2] - J_gn[1:2,1:2]) > 1e-10
    @test abs(J_fn[3,3] + 1e-3) < 1e-10
    @test abs(J_gn[3,3] + 1e-3) < 1e-10
    @test norm(J_fn[1:2,3] - J_gn[1:2,3]) < 1e-10
    @test norm(J_fn[3,1:2] - J_gn[3,1:2]) < 1e-10
end
```

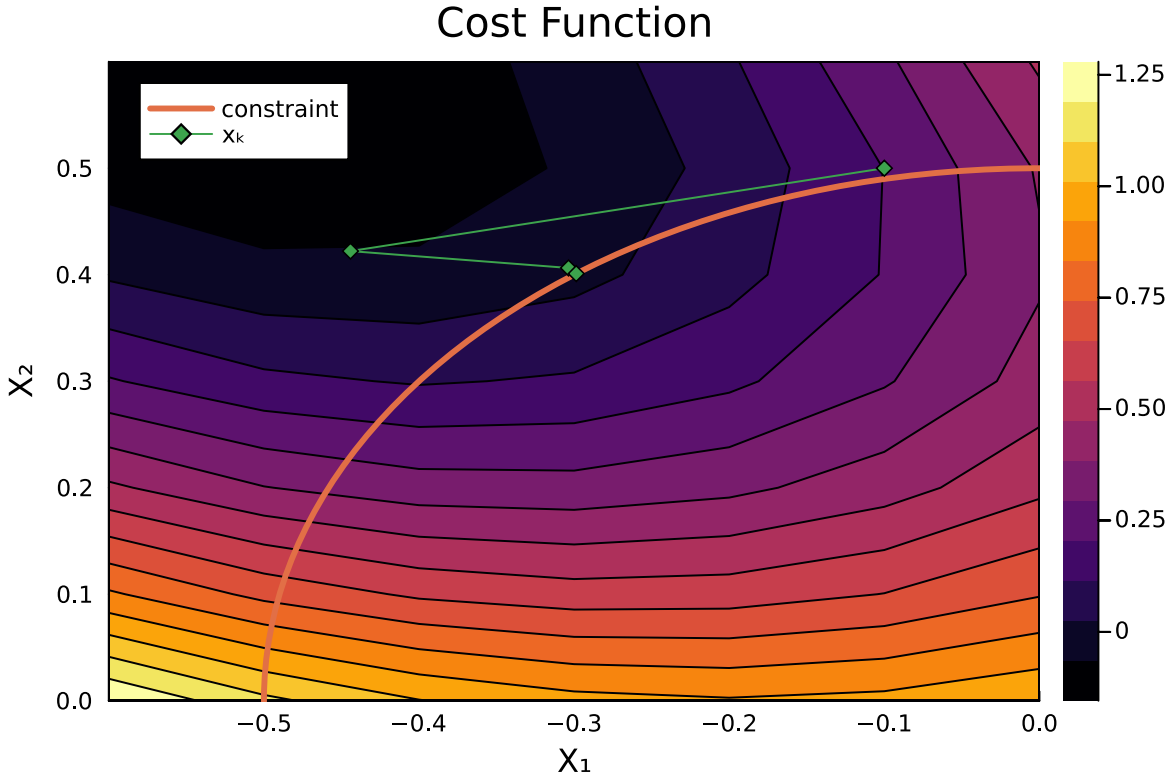
```
J_gn = [5.9906801663114475 3.560765760890812 -0.9025324581596084; 3.56076576
0890812 8.157657043286878 -0.4306218317368209; -0.9025324581596084 -0.430621
8317368209 -0.001]
Test Summary:  | Pass  Total
Test Jacobians |    5      5
Test.DefaultTestSet("Test Jacobians", Any[], 5, false, false)
```

In [28]:

```
@testset "Full Newton" begin


    z0 = [-.1, .5, 0] # initial guess
    merit_fx(_z) = norm(kkt_conditions(_z)) # simple merit function
    Z = newtons_method(z0, kkt_conditions, fn_kkt_jac, merit_fx; tol = 1e-4,
    R = kkt_conditions.(Z)

    # make sure we converged on a solution to the KKT conditions
    @test norm(kkt_conditions(Z[end])) < 1e-4
    @test length(R) < 6
```

```
    # ------------------------plotting stuff-----------------------
    Rp = [[abs(R[i][ii]) + 1e-15 for i = 1:length(R)] for ii = 1:length(R[1]
    
    plot(Rp[1],yaxis=:log,ylabel = "|r|",xlabel = "iteration",
         yticks= [1.0*10.0^(-x) for x = float(15:-1:-2)],
         title = "Convergence of Full Newton on KKT Conditions",label = "|r_
    plot!(Rp[2],label = "|r_2|")
    display(plot!(Rp[3],label = "|r_3|"))
    
    contour(-.6:.1:0,0:.1:.6, (x1,x2)-> cost([x1;x2]),title = "Cost Function
            xlabel = "X₁", ylabel = "X₂",fill = true)
    xcirc = [.5*cos(θ) for θ in range(0, 2*pi, length = 200)]
    ycirc = [.5*sin(θ) for θ in range(0, 2*pi, length = 200)]
    plot!(xcirc,ycirc, lw = 3.0, xlim = (-.6, 0), ylim = (0, .6),label = "co
    z1_hist = [z[1] for z in Z]
    z2_hist = [z[2] for z in Z]
    display(plot!(z1_hist, z2_hist, marker = :d, label = "xₖ"))
    # ------------------------plotting stuff-----------------------
end
```

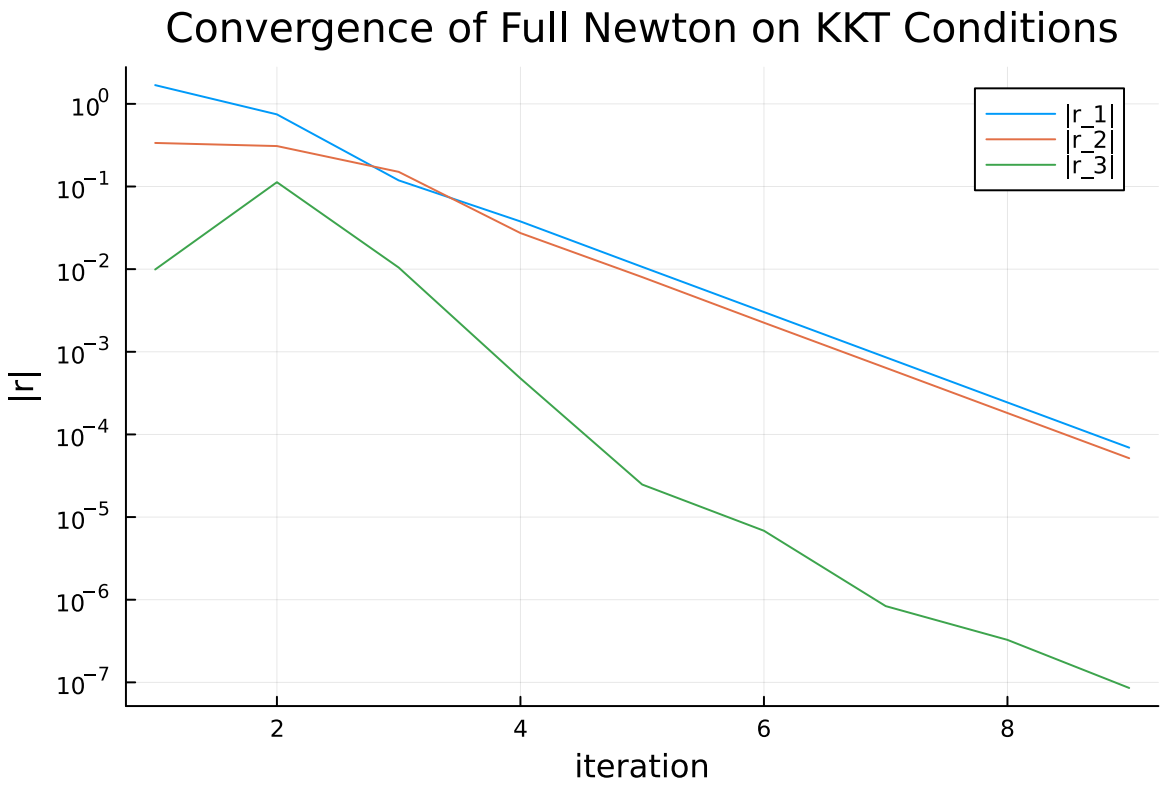


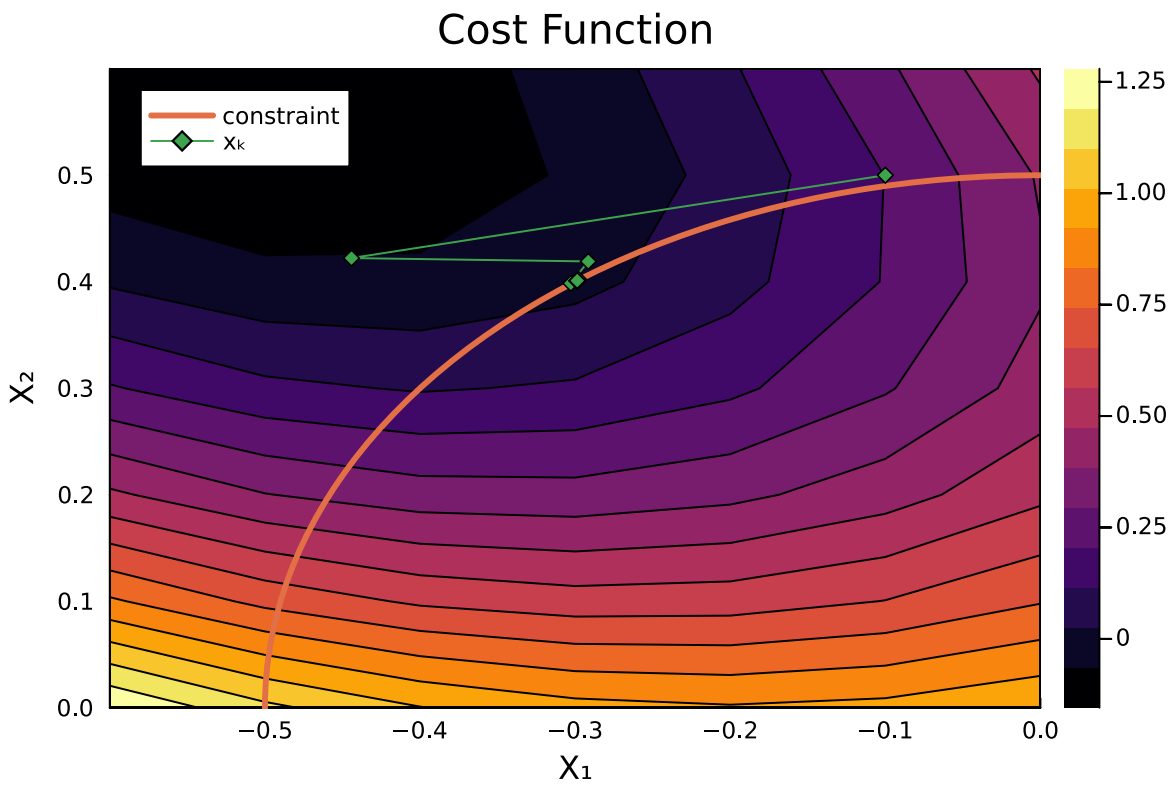



```
iter: 1    |r|: 1.6855584155478687   α: 1.0
iter: 2    |r|: 0.7459047551922087   α: 1.0
iter: 3    |r|: 0.023362325158016173   α: 1.0
iter: 4    |r|: 7.649879975402119e-5   Test Summary: |
Pass  Total
Full Newton   |    2      2
Test.DefaultTestSet("Full Newton", Any[], 2, false, false)
```

In [29]:
```
@testset "Gauss-Newton" begin


    z0 = [-.1, .5, 0] # initial guess
    merit_fx(_z) = norm(kkt_conditions(_z)) # simple merit function

    # the only difference in this block vs the previous is `gn_kkt_jac` inst
    Z = newtons_method(z0, kkt_conditions, gn_kkt_jac, merit_fx; tol = 1e-4,
    R = kkt_conditions.(Z)

    # make sure we converged on a solution to the KKT conditions
    @test norm(kkt_conditions(Z[end])) < 1e-4
    @test length(R) < 10


    # ------------------------plotting stuff-----------------------
    Rp = [[abs(R[i][ii]) + 1e-15 for i = 1:length(R)] for ii = 1:length(R[1]

    plot(Rp[1],yaxis=:log,ylabel = "|r|",xlabel = "iteration",
         yticks= [1.0*10.0^(-x) for x = float(15:-1:-2)],
         title = "Convergence of Full Newton on KKT Conditions",label = "|r_
    plot!(Rp[2],label = "|r_2|")
    display(plot!(Rp[3],label = "|r_3|"))
```

```
    contour(-.6:.1:0,0:.1:.6, (x1,x2)-> cost([x1;x2]),title = "Cost Function
            xlabel = "X₁", ylabel = "X₂",fill = true)
    xcirc = [.5*cos(θ) for θ in range(0, 2*pi, length = 200)]
    ycirc = [.5*sin(θ) for θ in range(0, 2*pi, length = 200)]
    plot!(xcirc,ycirc, lw = 3.0, xlim = (-.6, 0), ylim = (0, .6),label = "co
    z1_hist = [z[1] for z in Z]
    z2_hist = [z[2] for z in Z]
    display(plot!(z1_hist, z2_hist, marker = :d, label = "xₖ"))
    # ------------------------plotting stuff-----------------------
end
```






```
iter: 1    |r|: 1.6855584155478687   α: 1.0
iter: 2    |r|: 0.7459047551922087   α: 1.0
iter: 3    |r|: 0.1504394734838188   α: 1.0
iter: 4    |r|: 0.03778566208310541   α: 1.0
iter: 5    |r|: 0.010653168981697325   α: 1.0
iter: 6    |r|: 0.003027606542100325   α: 1.0
iter: 7    |r|: 0.0008588845232736508   α: 1.0
iter: 8    |r|: 0.00024377563414024195   α: 1.0
iter: 9    |r|: 6.918044121528855e-5   Test Summary: |
Pass  Total
Gauss-Newton  |    2      2
Test.DefaultTestSet("Gauss-Newton", Any[], 2, false, false)
```

Part B (10 pts): Balance a quadruped

Now we are going to solve for the control input $u \in \mathbb{R}^{12}$, and state $x \in \mathbb{R}^{30}$, such that the quadruped is balancing up on one leg at an equilibrium point. First, let's load in a dynamics model from `quadruped.jl`, where

$\dot{x} = f(x, u) =$ `dynamics(model, x, u)`

In [51]:
```
# include the functions from quadruped.jl
include(joinpath(@__DIR__, "quadruped.jl"))

# this loads in our continuous time dynamics function xdot = dynamics(model,
```

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

let's load in a model and display the rough "guess" configuration that we are going for:

In [52]:
```
# --------these three are global variables------------
model = UnitreeA1() # contains all the model properties for the quadruped
mvis = initialize_visualizer(model) # visualizer
const x_guess = initial_state(model) # our guess state for balancing
# ----------------------------------------------------

set_configuration!(mvis, x_guess[1:state_dim(model)/2])
render(mvis)
```

```
┌ Info: MeshCat server started. You can open the visualizer by visiting the
following URL in your browser:
│ http://127.0.0.1:8700
└ @ MeshCat /home/rsharde/.julia/packages/MeshCat/vWPbP/src/visualizer.jl:73
```

```
ArgumentError: invalid index: 1.0 of type Float64


Stacktrace:
  [1] to_index(i::Float64)
    @ Base ./indices.jl:300
  [2] to_index(A::Vector{Float64}, i::Float64)
    @ Base ./indices.jl:277
  [3] to_indices
    @ ./indices.jl:333 [inlined]
  [4] to_indices
    @ ./indices.jl:325 [inlined]
  [5] getindex
    @ ./abstractarray.jl:1170 [inlined]
  [6] macro expansion
    @ ./multidimensional.jl:866 [inlined]
  [7] macro expansion
    @ ./cartesian.jl:64 [inlined]
  [8] _unsafe_getindex!
    @ ./multidimensional.jl:861 [inlined]
  [9] _unsafe_getindex
    @ ./multidimensional.jl:852 [inlined]
 [10] _getindex
    @ ./multidimensional.jl:838 [inlined]
 [11] getindex(A::Vector{Float64}, I::StepRangeLen{Float64, Base.TwicePrecis
ion{Float64}, Base.TwicePrecision{Float64}})
    @ Base ./abstractarray.jl:1170
 [12] top-level scope
    @ ~/OCRL/HW1_S24/Q2.ipynb:7
```

Now, we are going to solve for the state and control that get us an equilibrium (balancing)

on just one leg. We are going to do this by solving the following optimization problem:

$$\min_{x,u} \quad \frac{1}{2}(x - x_{guess})^T(x - x_{guess}) + \frac{1}{2}10^{-3}u^T u \tag{5}$$

$$\text{st} \quad \dot{x} = f(x,u) = 0 \tag{6}$$

Where our primal variables are $x \in \mathbb{R}^{30}$ and $u \in \mathbb{R}^{12}$, that we can stack up in a new variable $y = [x^T, u^T]^T \in \mathbb{R}^{42}$. We have a constraint $\dot{x} = f(x,u) = 0$, which will ensure the resulting configuration is an equilibrium. This constraint is enforced with a dual variable $\lambda \in \mathbb{R}^{30}$. We are now ready to use Newton's method to solve this equality constrained optimization problem, where we will solve for a variable $z = [y^T, \lambda^T]^T \in \mathbb{R}^{72}$.

In this next section, you should fill out `quadruped_kkt(z)` with the KKT conditions for this optimization problem, given the constraint is that `dynamics(model, x, u) = zeros(30)` . When forming the Jacobian of the KKT conditions, use the Gauss-Newton approximation for the hessian of the Lagrangian (see example above if you're having trouble with this).

In [113…

```
# initial guess
const x_guess = initial_state(model)

# indexing stuff
const idx_x = 1:30
const idx_u = 31:42
const idx_c = 43:72

# I like stacking up all the primal variables in y, where y = [x;u]
# Newton's method will solve for z = [x;u;λ], or z = [y;λ]

function quadruped_cost(y::Vector)
    # cost function
    @assert length(y) == 42
    x = y[idx_x]
    u = y[idx_u]

    # TODO: return cost
    cost = 0.5*(x-x_guess)'*(x-x_guess)+0.5*((10^(-3))*u'*u)
    return cost
end
function quadruped_constraint(y::Vector)::Vector
    # constraint function
    @assert length(y) == 42
    x = y[idx_x]
    u = y[idx_u]

    # TODO: return constraint
    # error("quadruped constraint not implemented")
    # x_dot = FD.derivative(y,x)
    constraint = dynamics(model,x,u)
    return constraint
end
```

```
function quadruped_kkt(z::Vector)::Vector
    @assert length(z) == 72
    x = z[idx_x]
    u = z[idx_u]
    λ = z[idx_c]

    y = [x;u]
    # TODO: return the KKT conditions
    J = FD.jacobian(quadruped_constraint,y)
    kkt_cond = [FD.gradient(quadruped_cost,y) + J'*λ; quadruped_constraint(y
    return kkt_cond
end

function quadruped_kkt_jac(z::Vector)::Matrix
    @assert length(z) == 72
    x = z[idx_x]
    u = z[idx_u]
    λ = z[idx_c]

    y = [x;u]
    # L(dx) = cost(dx) .+ λ'*constraint(dx)  #Lagrangian term
    # J = constraint_jacobian(x)
    # J_gn_kkt = [(FD.hessian(dx -> cost(dx), x) + reg*I) J'; J -reg*I]

    # TODO: return Gauss-Newton Jacobian with a regularizer (try 1e-3,1e-4,1
    # and use whatever regularizer works for you
    # L(dy) = quadruped_cost(dy) .+ λ'*quadruped_constraint(dy)  #Lagrangian

    B = 1e-3
    J = FD.jacobian(quadruped_constraint,y)
    J_gn = [FD.hessian(quadruped_cost,y)+B*I J';J -B*I]
    # error("quadruped kkt jac not implemented")
    return J_gn
end
```

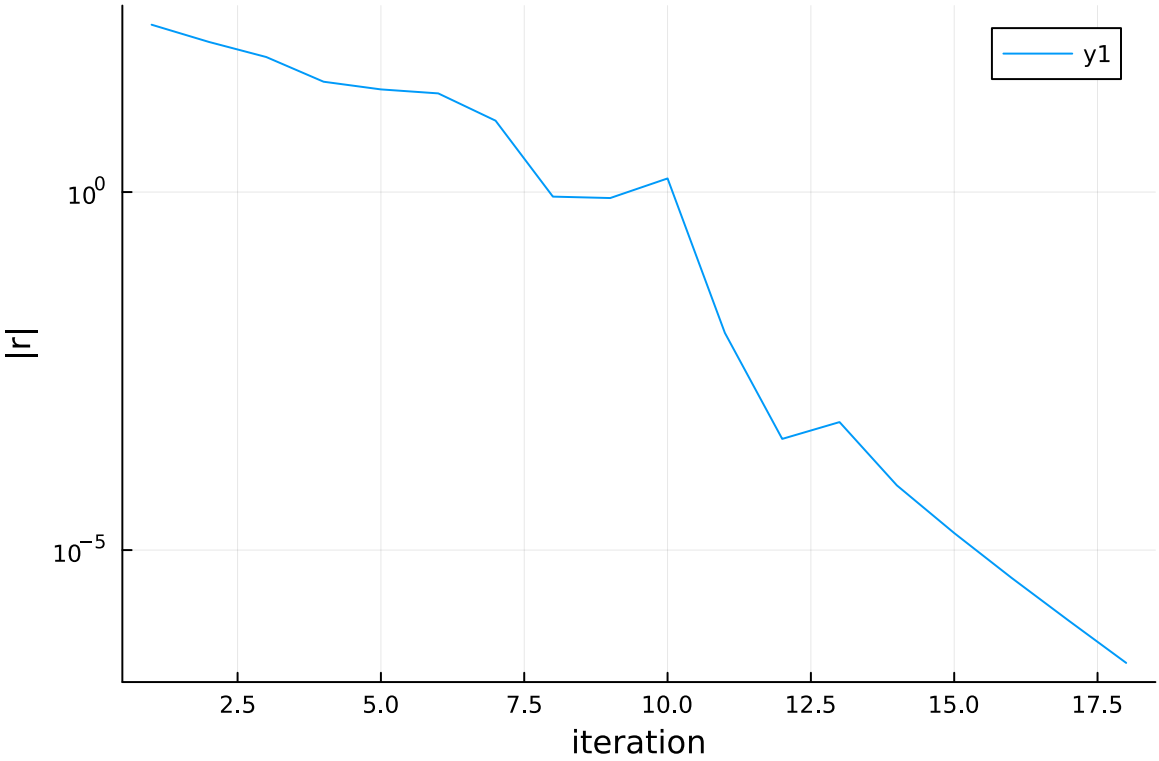
```
WARNING: redefinition of constant x_guess. This may fail, cause incorrect an
swers, or produce other errors.
quadruped_kkt_jac (generic function with 1 method)
```

In [114…

```
function quadruped_merit(z)
    # merit function for the quadruped problem

    @assert length(z) == 72
    r = quadruped_kkt(z)
    return norm(r[1:42]) + 1e4*norm(r[43:end])
end

@testset "quadruped standing" begin

    z0 = [x_guess; zeros(12); zeros(30)]
    Z = newtons_method(z0, quadruped_kkt, quadruped_kkt_jac, quadruped_merit

    set_configuration!(mvis, Z[end][1:state_dim(model)÷2])
    R = norm.(quadruped_kkt.(Z))

    display(plot(1:length(R), R, yaxis=:log,xlabel = "iteration", ylabel = "
```

```
    @test R[end] < 1e-6
    @test length(Z) < 25

    x,u = Z[end][idx_x], Z[end][idx_u]

    @test norm(dynamics(model, x, u)) < 1e-6

end
```

```
iter: 1    |r|: 172.1669185189028   α: 1.0
iter: 2    |r|: 102.34579254406071   α: 1.0
iter: 3    |r|: 34.3823956642315   α: 0.5
iter: 4    |r|: 16.106345867644208   α: 0.25
iter: 5    |r|: 12.891067357897528   α: 0.5
iter: 6    |r|: 13.611440847341227   α: 1.0
iter: 7    |r|: 7.6862149372807185   α: 1.0
iter: 8    |r|: 0.6262116041765683   α: 1.0
iter: 9    |r|: 0.805274572125095   α: 1.0
iter: 10    |r|: 1.537125876961492   α: 1.0
iter: 11    |r|: 0.010568630313613658   α: 1.0
iter: 12    |r|: 0.000327361161290618   α: 1.0
iter: 13    |r|: 0.000609291073245144   α: 1.0
iter: 14    |r|: 7.937680447911433e-5   α: 1.0
iter: 15    |r|: 1.7074286326468346e-5   α: 1.0
iter: 16    |r|: 4.034879679204462e-6   α: 1.0
iter: 17    |r|: 1.0134926500260377e-6   α: 1.0
iter: 18    |r|: 2.6115291112960293e-7
```




```
Test Summary:      | Pass  Total
quadruped standing |    3      3
Test.DefaultTestSet("quadruped standing", Any[], 3, false, false)
```

In [108…

```
let

    # let's visualize the balancing position we found

    z0 = [x_guess; zeros(12); zeros(30)]
    Z = newtons_method(z0, quadruped_kkt, quadruped_kkt_jac, quadruped_merit
    # visualizer
    mvis = initialize_visualizer(model)
    set_configuration!(mvis, Z[end][1:state_dim(model)÷2])
    render(mvis)


end
```

```
ArgumentError: number of columns of each array must match (got (1, 42))



Stacktrace:

 [1] _typed_vcat(#unused#::Type{Float64}, A::Tuple{Vector{Float64}, Matrix{F
loat64}})

   @ Base ./abstractarray.jl:1553

 [2] typed_vcat

   @ ./abstractarray.jl:1567 [inlined]

 [3] vcat(::Vector{Float64}, ::Matrix{Float64})

   @ SparseArrays /buildworker/worker/package_linux64/build/usr/share/julia/
stdlib/v1.6/SparseArrays/src/sparsevector.jl:1120

 [4] quadruped_kkt(z::Vector{Float64})

   @ Main ~/OCRL/HW1_S24/Q2.ipynb:43

 [5] newtons_method(z0::Vector{Float64}, res_fx::typeof(quadruped_kkt), res_
jac_fx::typeof(quadruped_kkt_jac), merit_fx::Function; tol::Float64, max_ite
rs::Int64, verbose::Bool)

   @ Main ~/OCRL/HW1_S24/Q2.ipynb:49

 [6] top-level scope

   @ ~/OCRL/HW1_S24/Q2.ipynb:6
```

Part C (5 pts): One sentence short answer

1. Why do we use a linesearch?

Linesearch is used to adjust the step size along a search direction and can help to converge faster.

2. Do we need a linesearch for both convex and nonconvex problems?

It can be used in both problems to get an efficient step size along the search direction and help converge faster.

3. Name one case where we absolutely do not need a linesearch.

A linesearch would not be needed in the case where the function is quadritic with a positive definite Hessian matrix, the optimal step size can be determined as the gradient descent direction is always the steepest direction.

In [2]:
```
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
using LinearAlgebra, Plots
import ForwardDiff as FD
using Printf
using JLD2
```

```
  Activating environment at `~/OCRL/H1_copy/HW1_S24/Project.toml`
    Updating registry at `~/.julia/registries/General`
   Installed ChainRulesCore ─ v1.21.1
```

```
  Updating `~/OCRL/H1_copy/HW1_S24/Project.toml`
[5ae59095] + Colors v0.12.10
[f6369f11] + ForwardDiff v0.10.36
[5c1252a2] + GeometryBasics v0.4.10
[033835bb] + JLD2 v0.4.45
[283c5d60] + MeshCat v0.15.1
[6ad125db] + MeshCatMechanisms v0.9.0
[91a5bcdd] + Plots v1.39.0
[366cf18f] + RigidBodyDynamics v2.3.2
[6038ab10] + Rotations v1.7.0
[90137ffa] + StaticArrays v1.9.2
  Updating `~/OCRL/H1_copy/HW1_S24/Manifest.toml`
[79e6a3ab] + Adapt v3.7.2
[bf4720bc] + AssetRegistry v0.1.0
[13072b0f] + AxisAlgorithms v1.0.1
[9e28174c] + BinDeps v1.0.2
[ad839575] + Blink v0.12.5
[70588ee8] + CSSUtil v0.1.1
[7057c7e9] + Cassette v0.3.13
[d360d2e6] + ChainRulesCore v1.21.1
[9e997f8a] + ChangesOfVariables v0.1.8
[35d6a980] + ColorSchemes v3.24.0
[3da002f7] + ColorTypes v0.11.4
[c3611d14] + ColorVectorSpace v0.9.10
[5ae59095] + Colors v0.12.10
[bbf7d656] + CommonSubexpressions v0.3.0
[34da2185] + Compat v4.12.0
[187b0558] + ConstructionBase v1.5.4
[d38c429a] + Contour v0.6.2
[150eb455] + CoordinateTransformations v0.6.3
[9a962f9c] + DataAPI v1.16.0
[864edb3b] + DataStructures v0.18.16
[e2d170a0] + DataValueInterfaces v1.0.0
[163ba53b] + DiffResults v1.1.0
[b552c78f] + DiffRules v1.15.1
[ffbed154] + DocStringExtensions v0.8.6
[411431e0] + Extents v0.1.2
[c87230d0] + FFMPEG v0.4.1
[5789e2e9] + FileIO v1.16.2
[53c48c17] + FixedPointNumbers v0.8.4
[59287772] + Formatting v0.4.2
[f6369f11] + ForwardDiff v0.10.36
[de31a74c] + FunctionalCollections v0.5.0
[46192b85] + GPUArraysCore v0.1.5
[28b8d3ca] + GR v0.72.8
[cf35fbd7] + GeoInterface v1.3.3
[5c1252a2] + GeometryBasics v0.4.10
[42e2da0e] + Grisu v1.0.2
[cd3eb016] + HTTP v0.8.19
[9fb69e20] + Hiccup v0.2.2
[83e8ac13] + IniFile v0.5.1
[d3863d7c] + InteractBase v0.10.10
[a98d9a8b] + Interpolations v0.14.7
[3587e190] + InverseFunctions v0.1.12
[92d709cd] + IrrationalConstants v0.2.2
[c8e1da08] + IterTools v1.4.0
```

```
[82899510] + IteratorInterfaceExtensions v1.0.0
[033835bb] + JLD2 v0.4.45
[1019f520] + JLFzf v0.1.7
[692b3bcd] + JLLWrappers v1.5.0
[97c1335a] + JSExpr v0.5.4
[682c06a0] + JSON v0.21.4
[bcebb21b] + Knockout v0.2.6
[b964fa9f] + LaTeXStrings v1.3.1
[23fbe1c1] + Latexify v0.16.1
[50d2b5c4] + Lazy v0.15.1
[9c8b4983] + LightXML v0.9.1
[2ab3a3ac] + LogExpFunctions v0.3.26
[39f5be34] + LoopThrottle v0.1.0
[1914dd2f] + MacroTools v0.5.13
[739be429] + MbedTLS v1.1.9
[442fdcdd] + Measures v0.3.2
[931e9471] + MechanismGeometries v0.7.1
[283c5d60] + MeshCat v0.15.1
[6ad125db] + MeshCatMechanisms v0.9.0
[e1d29d7a] + Missings v1.1.0
[99f44e22] + MsgPack v1.2.1
[ffc61752] + Mustache v1.0.19
[a975b10e] + Mux v0.7.6
[77ba4419] + NaNMath v1.0.2
[510215fc] + Observables v0.5.5
[6fe1bfb0] + OffsetArrays v1.13.0
[bac558e1] + OrderedCollections v1.6.3
[d96e819e] + Parameters v0.12.3
[69de0a69] + Parsers v2.8.1
[fa939f87] + Pidfile v1.3.0
[b98c9c47] + Pipe v1.3.0
[ccf2f8ad] + PlotThemes v3.1.0
[995b91a9] + PlotUtils v1.4.0
[91a5bcdd] + Plots v1.39.0
[aea7be01] + PrecompileTools v1.2.0
[21216c6a] + Preferences v1.4.1
[94ee1d12] + Quaternions v0.7.6
[c84ed2f1] + Ratios v0.4.5
[c1ae055f] + RealDot v0.1.0
[3cdcf5f2] + RecipesBase v1.3.4
[01d81517] + RecipesPipeline v0.6.12
[189a3867] + Reexport v1.2.2
[05181044] + RelocatableFolders v1.0.1
[ae029012] + Requires v1.3.0
[366cf18f] + RigidBodyDynamics v2.3.2
[6038ab10] + Rotations v1.7.0
[6c6a2e73] + Scratch v1.2.1
[992d4aef] + Showoff v1.0.3
[a2af1166] + SortingAlgorithms v1.2.1
[276daf66] + SpecialFunctions v2.3.1
[90137ffa] + StaticArrays v1.9.2
[1e83bf80] + StaticArraysCore v1.4.2
[82ae8749] + StatsAPI v1.7.0
[2913bbd2] + StatsBase v0.34.2
[09ab397b] + StructArrays v0.6.17
[3783bdb8] + TableTraits v1.0.1
```

```
[bd369af6] + Tables v1.11.1
[62fd8b95] + TensorCore v0.1.1
[3bb67fe8] + TranscodingStreams v0.10.3
[94a5cd58] + TypeSortedCollections v1.1.0
[30578b45] + URIParser v0.4.1
[3a884ed6] + UnPack v1.0.2
[1cfade01] + UnicodeFun v0.4.1
[1986cc42] + Unitful v1.19.0
[45397f5d] + UnitfulLatexify v1.6.3
[c4a57d5a] + UnsafeArrays v1.0.5
[41fe7b60] + Unzip v0.1.2
[0f1e0344] + WebIO v0.8.21
[104b5d7c] + WebSockets v1.5.9
[cc8bc4a8] + Widgets v0.6.6
[efce3f68] + WoodburyMatrices v0.5.6
[6e34b625] + Bzip2_jll v1.0.8+1
[83423d85] + Cairo_jll v1.16.1+1
[5ae413db] + EarCut_jll v2.2.4+0
[2702e6a9] + EpollShim_jll v0.0.20230411+0
[2e619515] + Expat_jll v2.5.0+0
[b22a6f82] + FFMPEG_jll v4.4.2+2
[a3f928ae] + Fontconfig_jll v2.13.93+0
[d7e528f0] + FreeType2_jll v2.13.1+0
[559328eb] + FriBidi_jll v1.0.10+0
[0656b61e] + GLFW_jll v3.3.9+0
[d2c73de3] + GR_jll v0.72.8+0
[78b55507] + Gettext_jll v0.21.0+0
[7746bdde] + Glib_jll v2.76.5+0
[3b182d85] + Graphite2_jll v1.3.14+0
[2e76f6c2] + HarfBuzz_jll v2.8.1+1
[aacddb02] + JpegTurbo_jll v3.0.1+0
[c1c5ebd0] + LAME_jll v3.100.1+0
[88015f11] + LERC_jll v3.0.0+1
[1d63c593] + LLVMOpenMP_jll v15.0.7+0
[dd4b983a] + LZO_jll v2.10.1+0
[e9f186c6] + Libffi_jll v3.2.2+1
[d4300ac3] + Libgcrypt_jll v1.8.7+0
[7e76a0d4] + Libglvnd_jll v1.6.0+0
[7add5ba3] + Libgpg_error_jll v1.42.0+0
[94ce4f54] + Libiconv_jll v1.17.0+0
[4b2f31a3] + Libmount_jll v2.35.0+0
[89763e89] + Libtiff_jll v4.4.0+0
[38a345b3] + Libuuid_jll v2.36.0+0
[e7412a2a] + Ogg_jll v1.3.5+1
[458c3c95] + OpenSSL_jll v1.1.23+0
[efe28fd5] + OpenSpecFun_jll v0.5.5+0
[91d4177d] + Opus_jll v1.3.2+0
[30392449] + Pixman_jll v0.42.2+0
[ea2cea3b] + Qt5Base_jll v5.15.3+2
[a2964d1f] + Wayland_jll v1.21.0+1
[2381bf8a] + Wayland_protocols_jll v1.31.0+0
[02c8fc9c] + XML2_jll v2.12.2+0
[aed1982a] + XSLT_jll v1.1.34+0
[4f6342f7] + Xorg_libX11_jll v1.8.6+0
[0c0b7dd1] + Xorg_libXau_jll v1.0.11+0
[935fb764] + Xorg_libXcursor_jll v1.2.0+4
```

```
[a3789734] + Xorg_libXdmcp_jll v1.1.4+0
[1082639a] + Xorg_libXext_jll v1.3.4+4
[d091e8ba] + Xorg_libXfixes_jll v5.0.3+4
[a51aa0fd] + Xorg_libXi_jll v1.7.10+4
[d1454406] + Xorg_libXinerama_jll v1.1.4+4
[ec84b674] + Xorg_libXrandr_jll v1.5.2+4
[ea2f1a96] + Xorg_libXrender_jll v0.9.10+4
[14d82f49] + Xorg_libpthread_stubs_jll v0.1.1+0
[c7cfdc94] + Xorg_libxcb_jll v1.15.0+0
[cc61e674] + Xorg_libxkbfile_jll v1.1.2+0
[12413925] + Xorg_xcb_util_image_jll v0.4.0+1
[2def613f] + Xorg_xcb_util_jll v0.4.0+1
[975044d2] + Xorg_xcb_util_keysyms_jll v0.4.0+1
[0d47668e] + Xorg_xcb_util_renderutil_jll v0.3.9+1
[c22f9ab0] + Xorg_xcb_util_wm_jll v0.4.1+1
[35661453] + Xorg_xkbcomp_jll v1.4.6+0
[33bec58e] + Xorg_xkeyboard_config_jll v2.39.0+0
[c5fb5394] + Xorg_xtrans_jll v1.5.0+0
[3161d3a3] + Zstd_jll v1.5.5+0
[214eeab7] + fzf_jll v0.43.0+0
[a4ae2306] + libaom_jll v3.4.0+0
[0ac62f75] + libass_jll v0.15.1+0
[f638f0a6] + libfdk_aac_jll v2.0.2+0
[b53b4c65] + libpng_jll v1.6.40+0
[f27f6e37] + libvorbis_jll v1.3.7+1
[1270edf5] + x264_jll v2021.5.5+0
[dfaa095f] + x265_jll v3.5.0+0
[d8fb68d0] + xkbcommon_jll v1.4.1+1
[0dad84c5] + ArgTools
[56f22d72] + Artifacts
[2a0f44e3] + Base64
[ade2ca70] + Dates
[8bb1440f] + DelimitedFiles
[8ba89e20] + Distributed
[f43a241f] + Downloads
```

```
[7b1f6079] + FileWatching
[b77e0a4c] + InteractiveUtils
[b27032c2] + LibCURL
[76f85450] + LibGit2
[8f399da3] + Libdl
[37e2e46d] + LinearAlgebra
[56ddb016] + Logging
[d6f4376e] + Markdown
[a63ad114] + Mmap
[ca575930] + NetworkOptions
[44cfe95a] + Pkg
[de0858da] + Printf
[3fa0cd96] + REPL
[9a3f8284] + Random
[ea8e919c] + SHA
[9e88b42a] + Serialization
[1a1011a3] + SharedArrays
[6462fe0b] + Sockets
[2f01184e] + SparseArrays
[10745b16] + Statistics
[fa267f1f] + TOML
[a4e569a6] + Tar
[8dfed614] + Test
[cf7118a7] + UUIDs
[4ec0a83e] + Unicode
[e66e0078] + CompilerSupportLibraries_jll
[deac9b47] + LibCURL_jll
[29816b5a] + LibSSH2_jll
[c8ffd9c3] + MbedTLS_jll
[14a3606d] + MozillaCACerts_jll
[05823500] + OpenLibm_jll
[efcefdf7] + PCRE2_jll
[83775a58] + Zlib_jll
[8e850ede] + nghttp2_jll
[3f19e933] + p7zip_jll
```

Q2 (30 pts): Augmented Lagrangian Quadratic Program Solver

Part (A): QP Solver (10 pts)

Here we are going to use the augmented lagrangian method described here in a video, with the corresponding pdf here to solve the following problem:

$$\min_{x} \quad \frac{1}{2}x^T Q x + q^T x \tag{1}$$
$$\text{s.t.} \quad Ax - b = 0 \tag{2}$$
$$Gx - h \leq 0 \tag{3}$$

where the cost function is described by $Q \in \mathbb{R}^{n \times n}$, $q \in \mathbb{R}^n$, an equality constraint is described by $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$, and an inequality constraint is described by

$G \in \mathbb{R}^{p \times n}$ and $h \in \mathbb{R}^p$.

By introducing a dual variable $\lambda \in \mathbb{R}^m$ for the equality constraint, and $\mu \in \mathbb{R}^p$ for the inequality constraint, we have the following KKT conditions for optimality:

$$
\begin{aligned}
Qx + q + A^T\lambda + G^T\mu = 0 \quad & \text{stationarity} && (4)\\
Ax - b = 0 \quad & \text{primal feasibility} && (5)\\
Gx - h \leq 0 \quad & \text{primal feasibility} && (6)\\
\mu \geq 0 \quad & \text{dual feasibility} && (7)\\
\mu \circ (Gx - h) = 0 \quad & \text{complementarity} && (8)
\end{aligned}
$$

where $\circ$ is element-wise multiplication.

In [9]:
```
# TODO: read below
# NOTE: DO NOT USE A WHILE LOOP ANYWHERE
"""
The data for the QP is stored in `qp` the following way:
    @load joinpath(@__DIR__, "qp_data.jld2") qp

which is a NamedTuple, where
    Q, q, A, b, G, h = qp.Q, qp.q, qp.A, qp.b, qp.G, qp.h

contains all of the problem data you will need for the QP.

Your job is to make the following function

    x, λ, μ = solve_qp(qp; verbose = true, max_iters = 100, tol = 1e-8)

You can use (or not use) any of the additional functions:
You can use (or not use) any of the additional functions:
You can use (or not use) any of the additional functions:
You can use (or not use) any of the additional functions:

as long as solve_qp works.
"""
function cost(qp::NamedTuple, x::Vector)::Real
    0.5*x'*qp.Q*x + dot(qp.q,x)
end
function c_eq(qp::NamedTuple, x::Vector)::Vector
    qp.A*x - qp.b
end
function h_ineq(qp::NamedTuple, x::Vector)::Vector
    qp.G*x - qp.h
end

function mask_matrix(qp::NamedTuple, x::Vector, μ::Vector, ρ::Real)::Matrix
    # error("not implemented")
    h_const = h_ineq(qp,x)
    m = length(h_const)
    I_p = zeros(m,m)
    for i=1:m
        if (h_const[i] < 0 && μ[i] == 0)
            I_p[i,i] = 0
```

```
        else
            I_p[i,i] = ρ
        end
    end
    return I_p
end

function augmented_lagrangian(qp::NamedTuple, x::Vector, λ::Vector, μ::Vecto
    # error("not implemented")
    c = c_eq(qp,x)
    h = h_ineq(qp,x)
    Lagrangian = cost(qp,x) + λ'*c + μ'*h
    rho_penalty = (ρ/2)*(c'*c)
    I_p = mask_matrix(qp,x,μ,ρ)
    ineq_term = 0.5*(h'*I_p*h)

    aug_Lagrangian = Lagrangian + rho_penalty + ineq_term
    return aug_Lagrangian
end

function logging(qp::NamedTuple, main_iter::Int, AL_gradient::Vector, x::Vec
    # TODO: stationarity norm
    c = c_eq(qp,x)
    h = h_ineq(qp,x)
    # stationarity = FD.gradient(dx -> cost(qp,dx), x) + FD.jacobian(dx -> c
    grad_L = FD.gradient(dx -> augmented_lagrangian(qp,dx,λ,μ,ρ),x)
    stationarity_norm = norm(grad_L,Inf)
    @printf("%3d  % 7.2e  % 7.2e  % 7.2e  % 7.2e  % 7.2e  %5.0e\n",
          main_iter, stationarity_norm, norm(AL_gradient), maximum(h_ineq(qp
          norm(c_eq(qp,x),Inf), abs(dot(μ,h_ineq(qp,x))), ρ)
end
function solve_qp(qp; verbose = true, max_iters = 100, tol = 1e-8)
    x = zeros(length(qp.q))
    λ = zeros(length(qp.b))
    μ = zeros(length(qp.h))
    φ = 10
    ρ = 1.0

    if verbose
        @printf "iter   |∇Lₓ|      |∇ALₓ|     max(h)     |c|        compl
        @printf "-------------------------------------------------------------
    end

    # TODO:
    for main_iter = 1:max_iters
        if verbose
            logging(qp, main_iter, zeros(1), x, λ, μ, 0.0)
        end

        # NOTE: when you do your dual update for μ, you should compute
        # your element-wise maximum with `max.(a,b)`, not `max(a,b)`

        # TODO: convergence criteria based on tol
        # Step 1: Solve min_x Lp(x,λ,μ,ρ)
        for i = 1:max_iters
            grad_L = FD.gradient(dx -> augmented_lagrangian(qp,dx, λ, μ, ρ),
```

```
            if norm(grad_L) < tol
                break
            end

            H = -FD.hessian(dx -> augmented_lagrangian(qp,dx, λ, μ, ρ),x)
            Δx = H\grad_L
            x += Δx
        end

        # Step 2: Update Dual variables
        c = c_eq(qp,x)
        h = h_ineq(qp,x)
        λ += ρ*c
        μ = max.(0,(μ+ρ*h))

        # Step 3: Update the ρ variable
        ρ = ρ*φ

        #Check convergence
        eq_violation = norm(c,Inf)
        ineq_violation = max.(0, maximum(ρ*h))

        if eq_violation < tol && ineq_violation <= tol
            return x, λ, μ
        end
    end
    error("qp solver did not converge")
end
let
    # example solving qp
    @load joinpath(@__DIR__, "qp_data.jld2") qp
    x, λ, μ = solve_qp(qp; verbose = true, tol = 1e-8)
end
```

```
iter    |∇Lₓ|      |∇ALₓ|     max(h)      |c|         compl      ρ
-----------------------------------------------------------------
  1   1.59e+01   0.00e+00   4.38e+00   6.49e+00   0.00e+00  0e+00
  2   2.66e-15   0.00e+00   5.51e-01   1.27e+00   4.59e-01  0e+00
  3   4.28e+00   0.00e+00   2.56e-02   3.07e-01   1.05e-02  0e+00
  4   3.49e-01   0.00e+00   6.84e-03   1.35e-02   7.94e-03  0e+00
  5   2.65e-12   0.00e+00   3.64e-05   1.62e-04   1.06e-04  0e+00
  6   1.30e-11   0.00e+00  -5.61e-09   2.05e-08   1.14e-08  0e+00
([-0.326230805713402, 0.24943797997188866, -0.43226766440507025, -1.41722469
71241351, -1.3994527400876546, 0.6099582408523686, -0.07312202122159463, 1.3
031477521999633, 0.5389034791065502, -0.7225813651685944], [-0.1283519512786
051, -2.8376241671761155, -0.832080449930223], [0.0363529426392617, 0.0, 0.0
, 1.059444495180323, 0.0])
```

QP Solver test

In [10]:
```
# 10 points
using Test
@testset "qp solver" begin
    @load joinpath(@__DIR__, "qp_data.jld2") qp
```

```
    x, λ, μ = solve_qp(qp; verbose = true, max_iters = 100, tol = 1e-6)

    @load joinpath(@__DIR__, "qp_solutions.jld2") qp_solutions
    @test norm(x - qp_solutions.x,Inf)<1e-3;
    @test norm(λ - qp_solutions.λ,Inf)<1e-3;
    @test norm(μ - qp_solutions.μ,Inf)<1e-3;
end
```

```
iter    |∇Lₓ|      |∇ALₓ|     max(h)      |c|         compl      ρ
-----------------------------------------------------------------
  1   1.59e+01   0.00e+00   4.38e+00   6.49e+00   0.00e+00  0e+00
  2   2.66e-15   0.00e+00   5.51e-01   1.27e+00   4.59e-01  0e+00
  3   4.28e+00   0.00e+00   2.56e-02   3.07e-01   1.05e-02  0e+00
  4   3.49e-01   0.00e+00   6.84e-03   1.35e-02   7.94e-03  0e+00
  5   2.65e-12   0.00e+00   3.64e-05   1.62e-04   1.06e-04  0e+00
Test Summary: |
Pass  Total
qp solver     |    3      3
Test.DefaultTestSet("qp solver", Any[], 3, false, false)
```

Simulating a Falling Brick with QPs

In this question we'll be simulating a brick falling and sliding on ice in 2D. You will show that this problem can be formulated as a QP, which you will solve using an Augmented Lagrangian method.

The Dynamics

The dynamics of the brick can be written in continuous time as

$$M\dot{v} + Mg = J^T\mu$$
$$\text{where } M = mI_{2\times 2},\ g = \begin{bmatrix} 0 \\ 9.81 \end{bmatrix},\ J = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

and $\mu \in \mathbb{R}$ is the normal force. The velocity $v \in \mathbb{R}^2$ and position $q \in \mathbb{R}^2$ are composed of the horizontal and vertical components.

We can discretize the dynamics with backward Euler:

$$\begin{bmatrix} v_{k+1} \\ q_{k+1} \end{bmatrix} = \begin{bmatrix} v_k \\ q_k \end{bmatrix} + \Delta t \cdot \begin{bmatrix} \frac{1}{m}J^T\mu_{k+1} - g \\ v_{k+1} \end{bmatrix}$$

We also have the following contact constraints:

$$Jq_{k+1} \geq 0 \qquad \text{(don't fall through the ice)} \tag{9}$$
$$\mu_{k+1} \geq 0 \qquad \text{(normal forces only push, not pull)} \tag{10}$$
$$\mu_{k+1}Jq_{k+1} = 0 \qquad \text{(no force at a distance)} \tag{11}$$

Part (B): QP formulation for Falling Brick (5 pts)

Show that these discrete-time dynamics are equivalent to the following QP by writing down the KKT conditions.

$$\begin{aligned} &\text{minimize}_{v_{k+1}} && \frac{1}{2} v_{k+1}^T M v_{k+1} + [M(\Delta t \cdot g - v_k)]^T v_{k+1} && (12) \\ &\text{subject to} && -J(q_k + \Delta t \cdot v_{k+1}) \leq 0 && (13) \end{aligned}$$

TASK: Write down the KKT conditions for the optimization problem above, and show that it's equivalent to the dynamics problem stated previously. Use LaTeX markdown.

1. **Stationarity condition**:

$$\nabla_{V_{k+1}} L = M v_{k+1} + M(\Delta t \cdot g - v_k) - \mu (J \cdot \Delta t)^\top$$

2. **Primal feasability**:

$$-J q_k - J \Delta t \cdot v_{k+1} \leq 0$$

3. **Dual feasability**:

$$\mu \geq 0$$

4. **Complimentarity**:

$$\mu \odot J(q_k + \Delta t \cdot v_{k+1}) = 0$$

Part (C): Brick Simulation (5 pts)

In [13]:
```
function brick_simulation_qp(q, v; mass = 1.0, Δt = 0.01)

    # TODO: fill in the QP problem data for a simulation step
    # fill in Q, q, G, h, but leave A, b the same
    # this is because there are no equality constraints in this qp
    g = [0; 9.81]
    Mass_matrix = [1.0 0.0; 0.0 1.0]
    qp = (
        # Q = zeros(2,2),
        # q = zeros(2),
        Q = Mass_matrix,
        q = Mass_matrix*(Δt*g-v),
        A = zeros(0,2), # don't edit this
        b = zeros(0),   # don't edit this
        # G = zeros(1,2),
        G = -[0 1]*Δt,
```

```
        h = [0 1]*q
        )

    return qp
end
```

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

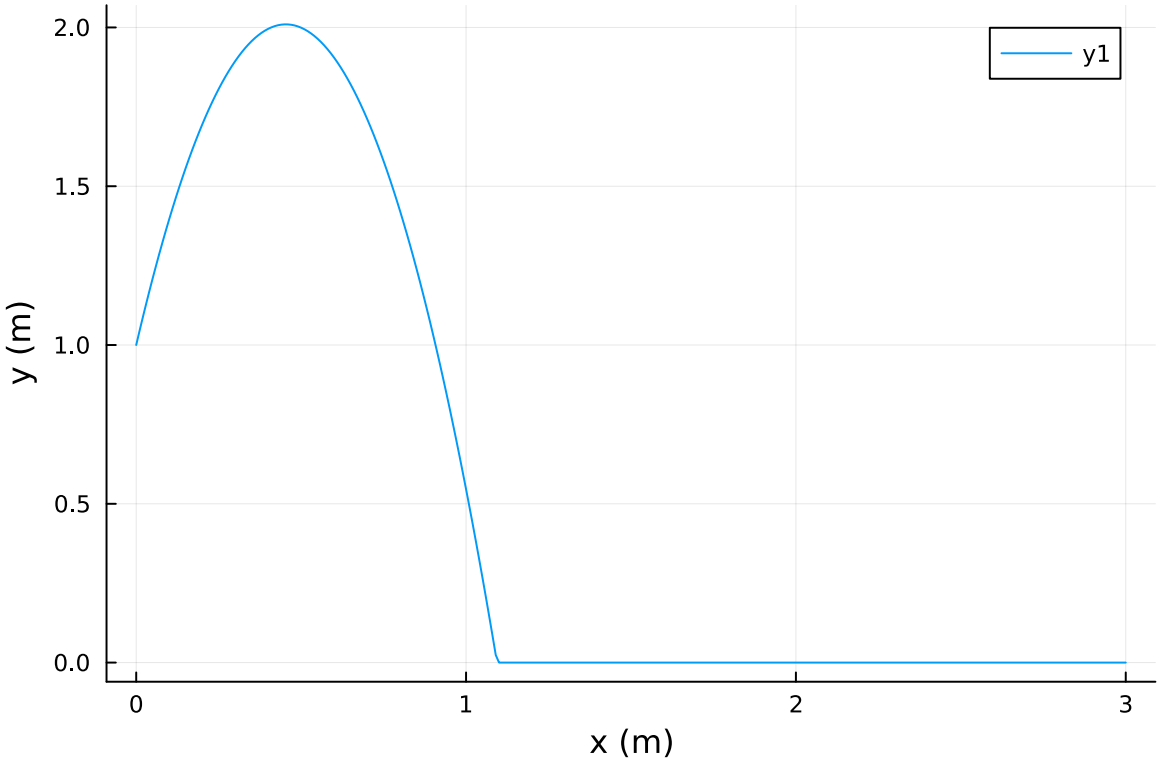
In [14]:
```
@testset "brick qp" begin

    q = [1,3.0]
    v = [2,-3.0]

    qp = brick_simulation_qp(q,v)

    # check all the types to make sure they're right
    qp.Q::Matrix{Float64}
    qp.q::Vector{Float64}
    qp.A::Matrix{Float64}
    qp.b::Vector{Float64}
    qp.G::Matrix{Float64}
    qp.h::Vector{Float64}

    @test size(qp.Q) == (2,2)
    @test size(qp.q) == (2,)
    @test size(qp.A) == (0,2)
    @test size(qp.b) == (0,)
    @test size(qp.G) == (1,2)
    @test size(qp.h) == (1,)

    @test abs(tr(qp.Q) - 2) < 1e-10
    @test norm(qp.q - [-2.0, 3.0981]) < 1e-10
    @test norm(qp.G - [0 -.01]) < 1e-10
    @test abs(qp.h[1] -3) < 1e-10

end
```

```
Test Summary: | Pass  Total
brick qp      |   10     10
Test.DefaultTestSet("brick qp", Any[], 10, false, false)
```

In [16]:
```
include(joinpath(@__DIR__, "animate_brick.jl"))
let

    dt = 0.01
    T = 3.0

    t_vec = 0:dt:T
    N = length(t_vec)

    qs = [zeros(2) for i = 1:N]
    vs = [zeros(2) for i = 1:N]

    qs[1] = [0, 1.0]
    vs[1] = [1, 4.5]

    # TODO: simulate the brick by forming and solving a qp
```

```
    # at each timestep. Your QP should solve for vs[k+1], and
    # you should use this to update qs[k+1]

    for k=1:(N-1)
        qp = brick_simulation_qp(qs[k], vs[k])
        vs[k+1],x,y=solve_qp(qp;verbose = false)
        qs[k+1] = qs[k]+vs[k+1]*dt
    end

    xs = [q[1] for q in qs]
    ys = [q[2] for q in qs]

    @show @test abs(maximum(ys)-2)<1e-1
    @show @test minimum(ys) > -1e-2
    @show @test abs(xs[end] - 3) < 1e-2

    xdot = diff(xs)/dt
    @show @test maximum(xdot) < 1.0001
    @show @test minimum(xdot) > 0.9999
    @show @test ys[110] > 1e-2
    @show @test abs(ys[111]) < 1e-2
    @show @test abs(ys[112]) < 1e-2

    display(plot(xs, ys, ylabel = "y (m)", xlabel = "x (m)"))

    animate_brick(qs)



end
```

```
#= /home/rsharde/OCRL/H1_copy/HW1_S24/Q3.ipynb:29 =# @test(abs(maximum(ys) -
2) < 0.1) = Test Passed
#= /home/rsharde/OCRL/H1_copy/HW1_S24/Q3.ipynb:30 =# @test(minimum(ys) > -0.
01) = Test Passed
#= /home/rsharde/OCRL/H1_copy/HW1_S24/Q3.ipynb:31 =# @test(abs(xs[end] - 3)
< 0.01) = Test Passed
#= /home/rsharde/OCRL/H1_copy/HW1_S24/Q3.ipynb:34 =# @test(maximum(xdot) <
1.0001) =
```




```
Test Passed
#= /home/rsharde/OCRL/H1_copy/HW1_S24/Q3.ipynb:35 =# @test(minimum(xdot) >
0.9999) = Test Passed
#= /home/rsharde/OCRL/H1_copy/HW1_S24/Q3.ipynb:36 =# @test(ys[110] > 0.01) =
Test Passed
#= /home/rsharde/OCRL/H1_copy/HW1_S24/Q3.ipynb:37 =# @test(abs(ys[111]) < 0.
01) = Test Passed
#= /home/rsharde/OCRL/H1_copy/HW1_S24/Q3.ipynb:38 =# @test(abs(ys[112]) < 0.
01) = Test Passed
```

```
┌ Info: MeshCat server started. You can open the visualizer by visiting the
following URL in your browser:
│ http://127.0.0.1:8701
└ @ MeshCat /home/rsharde/.julia/packages/MeshCat/vWPbP/src/visualizer.jl:73
```

Unable to connect

Firefox can’t establish a connection to the server at 127.0.0.1:8701.

- The site could be temporarily unavailable or too busy. Try again in a few moments.
- If you are unable to load any pages, check your computer’s network connection.
- If your computer or network is protected by a firewall or proxy, make sure that Firefox is permitted to access the web.

Try Again

Part D (5 pts): Solve a QP

Use your QP solver to solve the following optimization problem:

$$\begin{aligned}
\min_{y\in\mathbb{R}^2, a\in\mathbb{R}, b\in\mathbb{R}} \quad & \frac{1}{2} y^T \begin{bmatrix} 1 & .3 \\ .3 & 1 \end{bmatrix} y + a^2 + 2b^2 + \begin{bmatrix} -2 & 3.4 \end{bmatrix} y + 2a + 4b && (14)\\
\text{st} \quad & a + b = 1 && (15)\\
& \begin{bmatrix} -1 & 2.3 \end{bmatrix} y + a - 2b = 3 && (16)\\
& -0.5 \leq y \leq 1 && (17)\\
& -1 \leq a \leq 1 && (18)\\
& -1 \leq b \leq 1 && (19)
\end{aligned}$$

You should be able to put this into our standard QP form that we used above, and solve.

In [29]:
```
@testset "part D" begin
    Q = [1 0.3 0 0; 0.3 1 0 0; 0 0 0 0; 0 0 0 0]

    q = [-2; 3.4; 2; 4]

    A = [0 0 1 1; -1 2.3 1 -2]
    b = [1; 3]
```

```
    G = [1 0 0 0; 0 1 0 0; -1 0 0 0; 0 -1 0 0; 0 0 1 0; 0 0 0 1; 0 0 -1 0; 0
    h = [1; 1; 0.5; 0.5; 1; 1; 1; 1]

    x, λ, μ = solve_qp((Q = Q, q = q, A = A, b = b, G = G, h = h); verbose =

    # Assert that the solution satisfies the expected conditions
    @test norm(x[1:2] - [-0.080823; 0.834424]) < 1e-3
    @test abs(x[3] - 1) < 1e-3
    @test abs(x[4]) < 1e-3
end
```

```
iter    |∇Lₓ|      |∇ALₓ|     max(h)      |c|         compl      ρ
-----------------------------------------------------------------------
  1   4.00e+00   0.00e+00  -5.00e-01   3.00e+00   0.00e+00  0e+00
  2   7.55e-15   0.00e+00   2.26e+00   2.45e+00   7.74e+00  0e+00
  3   2.75e+00   0.00e+00   6.51e-01   4.13e-01   4.94e+00  0e+00
  4   3.23e+01   0.00e+00   3.19e-01   3.30e-01   1.25e+01  0e+00
  5   3.85e-13   0.00e+00  -3.16e-02   3.18e-02   2.39e-01  0e+00
  6   8.17e-12   0.00e+00  -5.64e-06   3.76e-06   4.22e-05  0e+00
Test Summary: | Pass  Total
part D        |    3      3
Test.DefaultTestSet("part D", Any[], 3, false, false)
```

Part E (5 pts): One sentence short answer

1. For our Augmented Lagrangian solver, if our initial guess for x is feasible (meaning it satisfies the constraints), will it stay feasible through each iteration?

Yes, it will because of the penalty term in the Augmented Lagrangian method.

2. Does the Augmented Lagrangian function for this problem always have continuous first derivatives?

Yes, because it comprises differentiable functions so usually it will have the continuous first derivatives.

3. Is the QP in part D always convex?

No, it is not always convex as that depends on the objective function and the constraints for each problem.

In [ ]: