In [1]:
```
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
using LinearAlgebra, Plots
import ForwardDiff as FD
import MeshCat as mc
using Test
```

```
  Activating project at `c:\CMU\SEM II\OCRL\HW1_S25`
```

# Julia Warmup

Just like Python, Julia lets you do the following:

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

In [3]:
```
# 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]
3-element Vector{Int64}:
   1
   2
 100
```

## Optional function arguments

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

In [4]:
```
## optional arguments in functions

# we can have functions with optional arguments after a ; that have default values
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)
```

```
(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 [5]:
```
# 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,ω1,θ2,ω2 = 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 = [
        ω1;
        ( m2*g*sin(θ2)*c - m2*s*(L1*c*ω1^2 + L2*ω2^2) - (m1+m2)*g*sin(θ1) ) / ( L1 *(m1+m2*s^2) );
        ω2;
        ((m1+m2)*(L1*ω1^2*s - g*sin(θ2) + g*sin(θ1)*c) + m2*L2*ω2^2*s*c) / (L2 * (m1 + m2*s^2));
        ]

    return ẋ
end
function double_pendulum_energy(params::NamedTuple, x::Vector)::Real
    # calculate the total energy (kinetic + potential) of a double pendulum given a state x


    # 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
```

```
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:

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

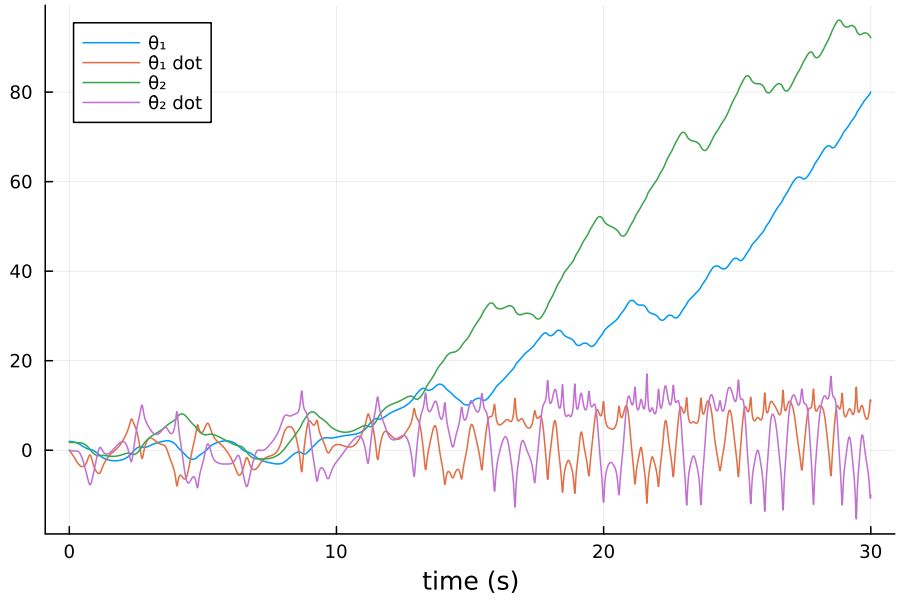
In [6]:
```
"""
    x_{k+1} = forward_euler(params, dynamics, x_k, dt)
```

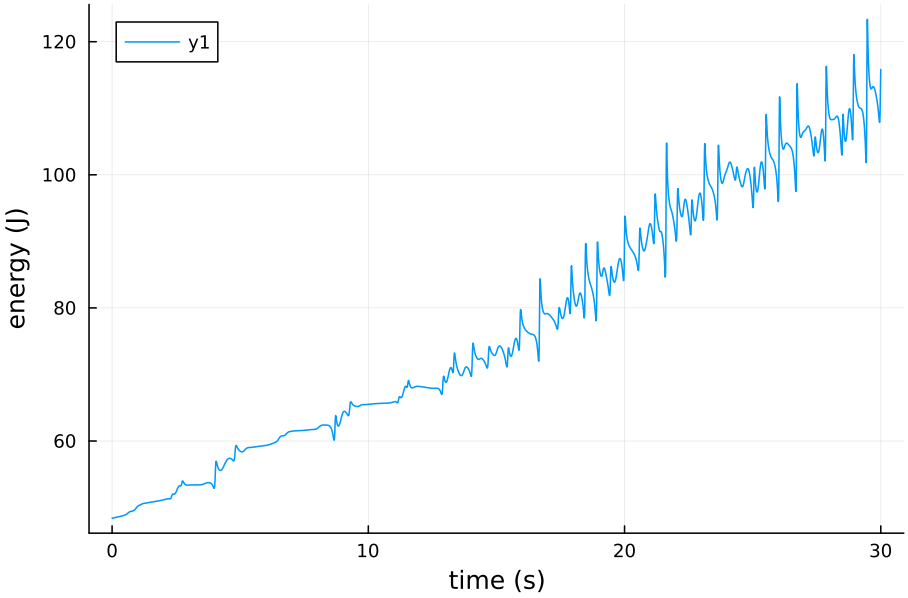
```
Given `ẋ = dynamics(params, x)`, take in the current state `x` and integrate it forward `dt`
using Forward Euler method.
"""
function forward_euler(params::NamedTuple, dynamics::Function, x::Vector, dt::Real)::Vector
    # ẋ = dynamics(params, x)
    # TODO: implement forward euler
    x_dot = dynamics(params, x)
    x_ = (x + dt*x_dot)

    return x_
end
```

```
forward_euler
```

In [7]:

```
include(joinpath(@__DIR__, "animation.jl"))

@testset "forward euler" begin

    # 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_dynamics, X[k], dt)
    for k in 1:N-1
        X[k+1] = forward_euler(params, double_pendulum_dynamics, X[k], dt)
    end

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

    @test norm(X[end]) > 1e-10   # make sure all X's were updated
    @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" "θ̇₁" "θ₂" "θ̇₂"]))
    display(plot(t_vec, E, xlabel = "time (s)", ylabel = "energy (J)"))
    meshcat_animate(params,X,dt,N)


end
```






```
┌ Info: Listening on: 127.0.0.1:8703, thread id: 1
└ @ HTTP.Servers C:\Users\barat\.julia\packages\HTTP\4AUPl\src\Servers.jl:382
┌ Info: MeshCat server started. You can open the visualizer by visiting the following URL in your browser:
│ http://127.0.0.1:8703
└ @ MeshCat C:\Users\barat\.julia\packages\MeshCat\9QrxD\src\visualizer.jl:43
```

**Test Summary:** | **Pass** **Total** **Time**
forward euler | 2 2 13.7s
Test.DefaultTestSet("forward euler", Any[], 2, false, false, true, 1.738890037406e9, 1.738890051077e9, false, "c
:\\CMU\\SEM II\\OCRL\\HW1_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X14sZmlsZQ==.jl")

Now let's implement the next two integrators:

**Midpoint:**

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

**RK4:**

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

In [8]:
```
function midpoint(params::NamedTuple, dynamics::Function, x::Vector, dt::Real)::Vector
    # TODO: implement explicit midpoint
    x_m = x + dt/2 * dynamics(params, x)
    x_ = x + dt * dynamics(params, x_m)

    return x_
end
function rk4(params::NamedTuple, dynamics::Function, x::Vector, dt::Real)::Vector
    # 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_ = x + (k1 + 2*k2 + 2*k3 + k4)/6

    return x_
end
```

```
rk4 (generic function with 1 method)
```

In [9]:
```
function simulate_explicit(params::NamedTuple,dynamics::Function,integrator::Function,x0::Vector,dt::Real,tf::R
    # 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 in 1:N-1
        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
```

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

In [10]:
```
# 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
)
```

```
(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 $\Delta t$ 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)

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

Hermite Simpson (3rd order)

$$x_{k+1/2} = \frac{1}{2}(x_k + x_{k+1}) + \frac{\Delta t}{8}(\dot{x}_k - \dot{x}_{k+1})$$
$$f(x_k, x_{k+1}, \Delta t) = x_k + \frac{\Delta t}{6} \cdot (\dot{x}_k + 4\dot{x}_{k+1/2} + \dot{x}_{k+1}) - x_{k+1} = 0 \qquad \text{Hermite-Simpson}$$

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 [11]:
```
# since these are explicit integrators, these function will return the residuals described above
# NOTE: we are NOT solving anything here, simply return the residuals
function backward_euler(params::NamedTuple, dynamics::Function, x1::Vector, x2::Vector, dt::Real)::Vector
    return x1 + dt*dynamics(params, x2) - x2
end
function implicit_midpoint(params::NamedTuple, dynamics::Function, x1::Vector, x2::Vector, dt::Real)::Vector
    x1_2 = 0.5*(x1 + x2)
    return x1 + dt*dynamics(params, x1_2) - x2
end
function hermite_simpson(params::NamedTuple, dynamics::Function, x1::Vector, x2::Vector, dt::Real)::Vector
    x1_2 = 0.5*(x1 + x2) + (dt/8)*(dynamics(params, x1) - dynamics(params, x2))
    return x1 + (dt/6)*(dynamics(params, x1) + 4*dynamics(params, x1_2) + dynamics(params, x2)) - x2
end
```

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

In [12]:
```
# TODO
# this function takes in a dynamics function, implicit integrator function, and x1
# and uses Newton's method to solve for an x2 that satsifies the implicit integration equations
# that we wrote about in the functions above
function implicit_integrator_solve(params::NamedTuple, dynamics::Function, implicit_integrator::Function, x1::V
```

```
    # initialize guess
    x2 = 1*x1

    # TODO: use Newton's method to solve for x2 such that residual for the integrator is 0
    # DO NOT USE A WHILE LOOP
    for i = 1:max_iters
        res = implicit_integrator(params, dynamics, x1, x2, dt)

        # TODO: return x2 when the norm of the residual is below tol
        if norm(res) < tol
            return x2
        end

        J = FD.jacobian(x -> implicit_integrator(params, dynamics, x1, x, dt), x2)
        Δx2 = - J \ res
        x2 += Δx2
    end
    error("implicit integrator solve failed")
end
```

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

In [13]:
```
@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, integrator, x1, dt)
        @test norm(integrator(params, double_pendulum_dynamics, x1, x2, dt)) < 1e-10
    end

end
```

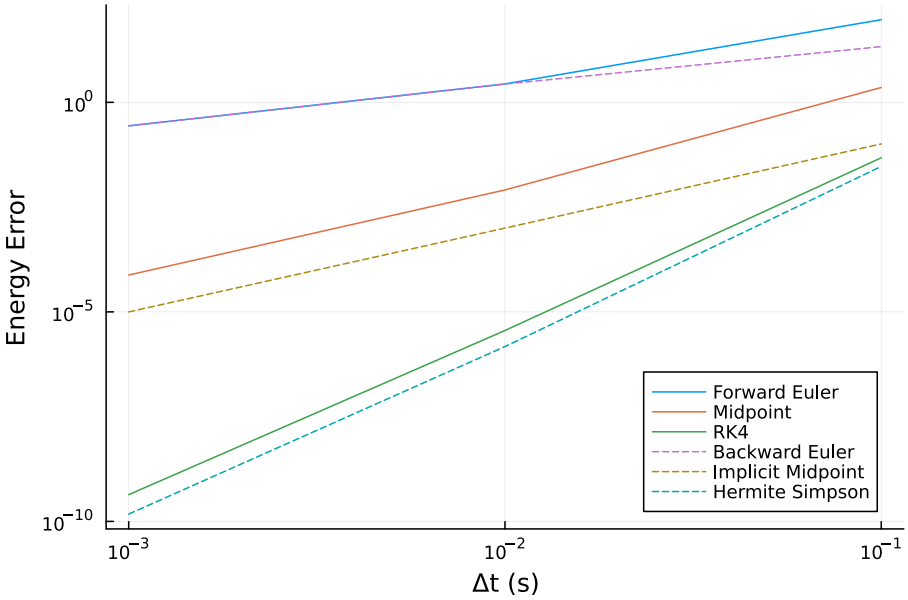
```
-----testing backward_euler ------
-----testing implicit_midpoint ------
-----testing hermite_simpson ------
Test Summary:              | Pass  Total  Time
implicit integrator check |    3      3  4.1s
Test.DefaultTestSet("implicit integrator check", Any[], 3, false, false, true, 1.738890051901e9, 1.738890056016e
9, false, "c:\\CMU\\SEM II\\OCRL\\HW1_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X25sZmlsZQ==.jl")
```

In [14]:
```
function simulate_implicit(params::NamedTuple,dynamics::Function,implicit_integrator::Function,x0::Vector,dt::R
    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 in 1:N-1
        X[k+1] = implicit_integrator_solve(params, dynamics, implicit_integrator, X[k], dt)
    end

    E = [double_pendulum_energy(params,x) for x in X]
    @assert length(X)==N
    @assert length(E)==N
    return X, E
end
```

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

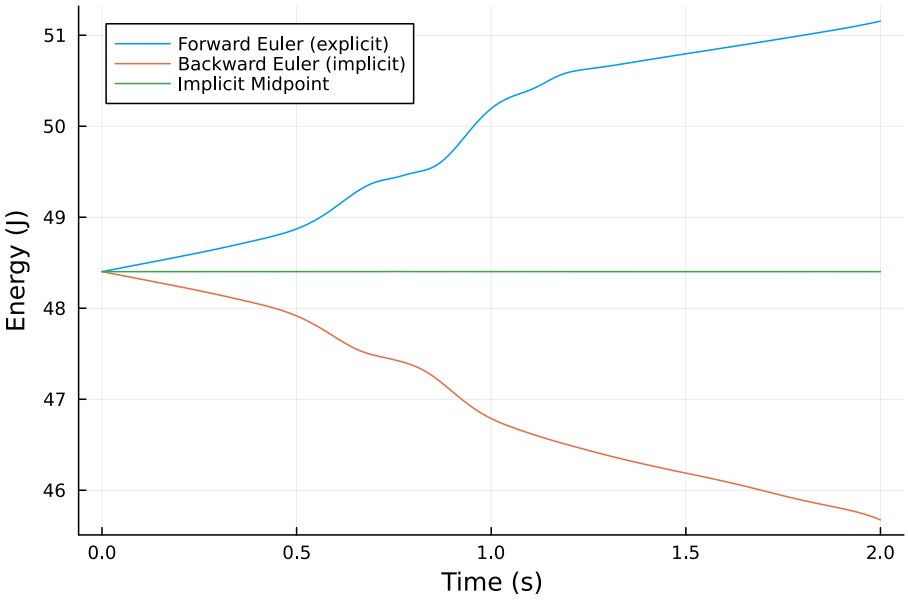
In [15]:
```
function max_err_E(E)
    E0 = E[1]
    err = abs.(E .- E0)
    return maximum(err)
end
function get_explicit_energy_error(integrator::Function, dts::Vector)
    [max_err_E(simulate_explicit(params,double_pendulum_dynamics,integrator,x0,dt,tf)[2]) for dt in dts]
end
function get_implicit_energy_error(integrator::Function, dts::Vector)
    [max_err_E(simulate_implicit(params,double_pendulum_dynamics,integrator,x0,dt,tf)[2]) for dt in dts]
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_simpson]

    explicit_data = [get_explicit_energy_error(integrator, dts) for integrator in explicit_integrators]
```

```
    implicit_data = [get_implicit_energy_error(integrator, dts) for integrator in implicit_integrators]

    plot(dts, hcat(explicit_data...),label = ["Forward Euler" "Midpoint" "RK4"],xaxis=:log10,yaxis=:log10, xlab
    plot!(dts, hcat(implicit_data...),ls = :dash, label = ["Backward Euler" "Implicit Midpoint" "Hermite Simpso
    plot!(legend=:bottomright)
end
```




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 [16]:
```
@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,dt,tf)[2]
    E2 = simulate_implicit(params,double_pendulum_dynamics,backward_euler,x0,dt,tf)[2]
    E3 = simulate_implicit(params,double_pendulum_dynamics,implicit_midpoint,x0,dt,tf)[2]
    E4 = simulate_implicit(params,double_pendulum_dynamics,hermite_simpson,x0,dt,tf)[2]
    E5 = simulate_explicit(params,double_pendulum_dynamics,midpoint,x0,dt,tf)[2]
    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)", ylabel="Energy (J)"))

    # 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  Time
energy behavior |    6      6  0.1s
Test.DefaultTestSet("energy behavior", Any[], 6, false, false, true, 1.738890057535e9, 1.738890057677e9, false,
"c:\\CMU\\SEM II\\OCRL\\HW1_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X32sZmlsZQ==.jl")
```

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.

In [1]:
```
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
using LinearAlgebra, Plots
import ForwardDiff as FD
using MeshCat
using Test
using Plots
```

```
 Activating project at `c:\CMU\SEM II\OCRL\HW1_S25`
```

# 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

$$\begin{aligned} \min_x \quad & f(x) \\ \text{st} \quad & c(x) = 0 \end{aligned}$$

Which has the following Lagrangian:

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

and the following KKT conditions for optimality:

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

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 $\phi$ is a "merit function", or `merit_fx(z)` in the code. In this assignment you will use a backtracking linesearch where $\alpha$ 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 [2]:
```
function linesearch(z::Vector, Δz::Vector, merit_fx::Function;
                    max_ls_iters = 10)::Float64 # optional argument with a default

    # 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)
    α = 1

    # NOTE: DO NOT USE A WHILE LOOP
    for i = 1:max_ls_iters

        # TODO: return α when merit_fx(z + α*Δz) < merit_fx(z)
        if merit_fx(z + α*Δz) < merit_fx(z)
            return α
        end
        α /= 2

    end
    error("linesearch failed")
end

function newtons_method(z0::Vector, res_fx::Function, res_jac_fx::Function, merit_fx::Function;
```

```
                        tol = 1e-10, max_iters = 50, verbose = false)::Vector{Vector{Float64}}

    # 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 iteration

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

    # 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
        res = res_fx(Z[i])

        norm_r = norm(res) # 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)
        J = res_jac_fx(Z[i])
        Δz = - J \ res

        α = linesearch(Z[i], Δz, merit_fx)

        # TODO: linesearch and update z
        Z[i+1] = Z[i] + α*Δz

        if verbose
            print("α: $α \n")
        end

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

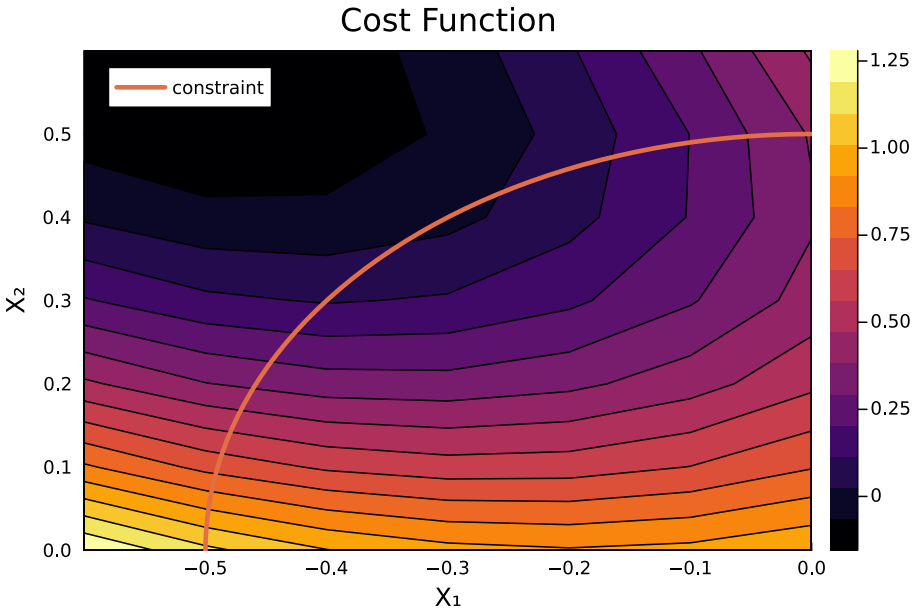
```
newtons_method (generic function with 1 method)
```

In [3]:

```
@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, verbose = true)

    # 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|: 0.9995239729818045   α: 1.0
iter: 2    |r|: 0.9421342427117169   α: 0.5
iter: 3    |r|: 0.1753172908866053   α: 1.0
iter: 4    |r|: 0.0018472215879181287   α: 1.0
iter: 5    |r|: 2.1010529101114843e-9   α: 1.0
iter: 6    |r|: 2.5246740534795566e-16   Test Summary: | Pass  Total  Time
check Newton  |    2      2  1.6s
Test.DefaultTestSet("check Newton", Any[], 2, false, false, true, 1.738890034587e9, 1.738890036199e9, false, "c:
\\CMU\\SEM II\\OCRL\\HW1_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_W5sZmlsZQ==.jl")
```

In [4]:
```
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 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 = "constraint")
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 [5]:
```
# we will use Newton's method to solve the constrained optimization problem shown above
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]

    # TODO: return the stationarity condition for the cost function
    # and the primal feasibility
    cond = [FD.gradient(x -> cost(x), x) + constraint_jacobian(x)' * λ; constraint(x)]

    return cond
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_x_x = FD.jacobian(x -> FD.gradient(x -> cost(x), x), x) + FD.jacobian(x -> FD.gradient(x -> constraint(x)
    c_x = reshape(FD.gradient(x -> constraint(x), x), 1, 2)

    jac = [J_x_x c_x'; c_x 0] - 1e-3*I

    # TODO: return full Newton jacobian with a 1e-3 regularizer
    return jac
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]

    J_x_x = FD.jacobian(x -> FD.gradient(x -> cost(x), x), x)
    c_x = reshape(FD.gradient(x -> constraint(x), x), 1, 2)

    jac = [J_x_x c_x'; c_x 0] - 1e-3*I
    print(jac[3, 3])

    # TODO: return Gauss-Newton jacobian with a 1e-3 regularizer
    return jac
end
```

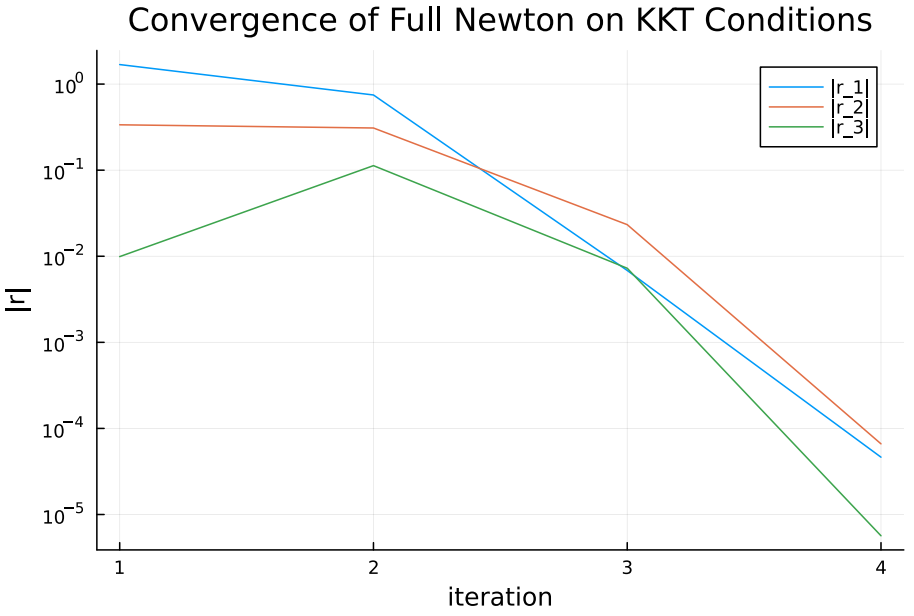
```
gn_kkt_jac (generic function with 1 method)
```

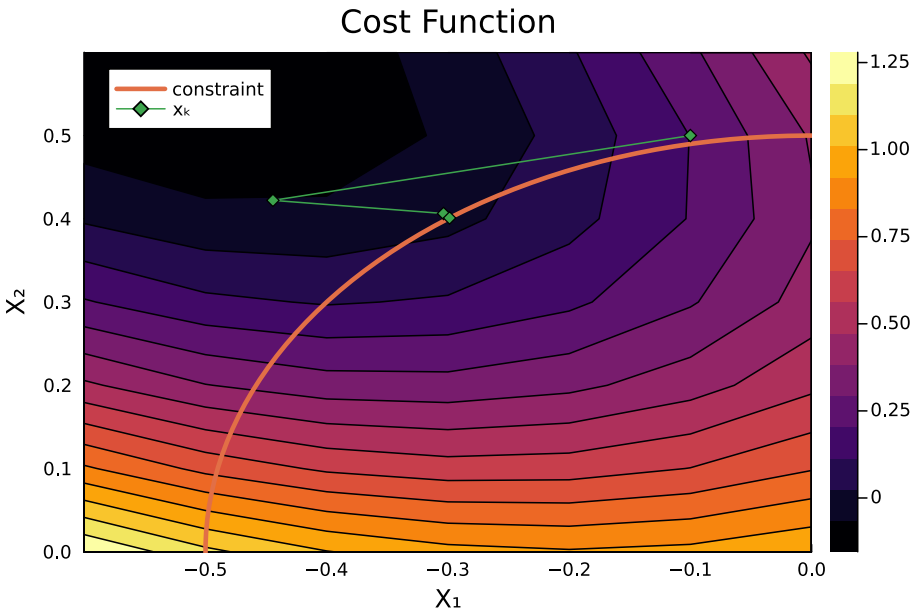
In [6]:
```
@testset "Test Jacobians" begin

    # 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
```

```
-0.001Test Summary:  | Pass  Total  Time
Test Jacobians |    5      5  7.1s
Test.DefaultTestSet("Test Jacobians", Any[], 5, false, false, true, 1.738890041398e9, 1.73889004853e9, false, "c
:\\CMU\\SEM II\\OCRL\\HW1_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X12sZmlsZQ==.jl")
```

In [7]:
```
@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, max_iters = 100, verbose = true)
    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])] # this gets abs of each term at

    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_1|")
    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 = "constraint")
    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
```



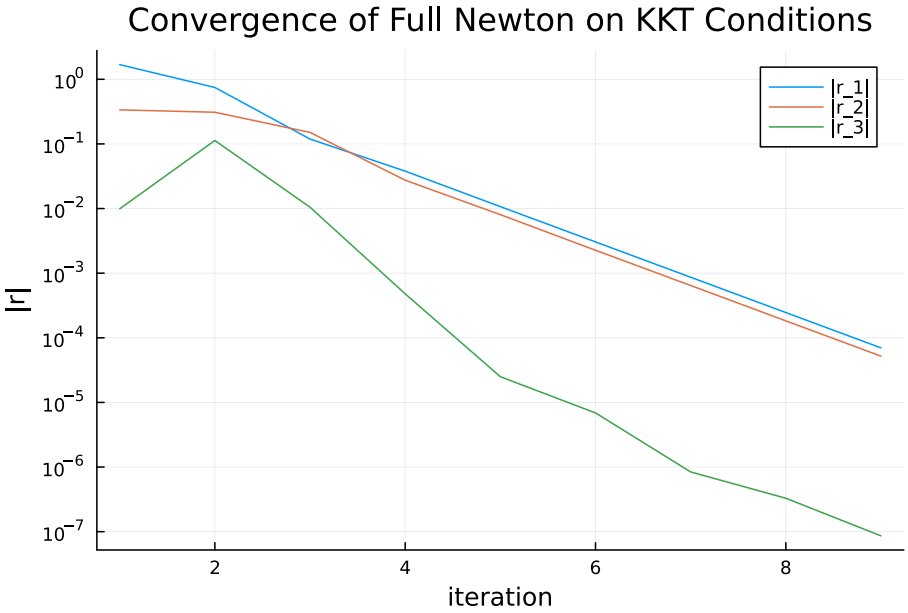



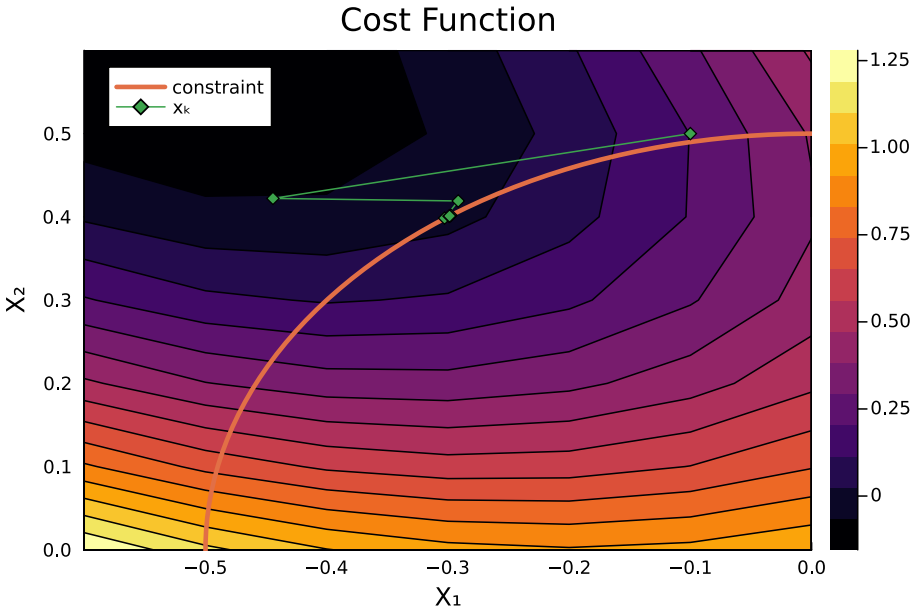

```
iter: 1    |r|: 1.7188450769812715   α: 1.0
iter: 2    |r|: 0.8163267519728127   α: 1.0
iter: 3    |r|: 0.025332347137044835   α: 1.0
iter: 4    |r|: 8.09925829123034e-5   Test Summary: | Pass  Total  Time
Full Newton   |    2      2  2.6s
Test.DefaultTestSet("Full Newton", Any[], 2, false, false, true, 1.738890048551e9, 1.738890051165e9, false, "c:\
\CMU\\SEM II\\OCRL\\HW1_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X13sZmlsZQ==.jl")
```

In [8]:

```
@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` instead of `fn_kkt_jac`
    Z = newtons_method(z0, kkt_conditions, gn_kkt_jac, merit_fx; tol = 1e-4, max_iters = 100, verbose = true)
    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])] # this gets abs of each term at
```

```
    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_1|")
    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 = "constraint")
    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|: 1.7188450769812715   -0.001α: 1.0
iter: 2    |r|: 0.8163267519728127   -0.001α: 1.0
iter: 3    |r|: 0.1922177677011686   -0.001α: 1.0
iter: 4    |r|: 0.04678866823071684   -0.001α: 1.0
iter: 5    |r|: 0.0133893914077011   -0.001α: 1.0
iter: 6    |r|: 0.003792680422273685   -0.001α: 1.0
iter: 7    |r|: 0.0010784701647844727   -0.001α: 1.0
iter: 8    |r|: 0.00030635243738783284   -0.001α: 1.0
iter: 9    |r|: 8.7049117069183e-5   Test Summary: | Pass  Total  Time
Gauss-Newton  |    2      2  0.4s
Test.DefaultTestSet("Gauss-Newton", Any[], 2, false, false, true, 1.738890051187e9, 1.738890051615e9, false, "c:
\\CMU\\SEM II\\OCRL\\HW1_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X14sZmlsZQ==.jl")
```

## Part B (10 pts): Balance a quadruped

Now we are going to solve for the control input $u \in \mathrm{R}^{12}$, and state $x \in \mathrm{R}^{30}$, 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 [9]:
```
# include the functions from quadruped.jl
include(joinpath(@__DIR__, "quadruped.jl"))

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

```
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 [10]:
```
# --------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: Listening on: 127.0.0.1:8704, thread id: 1
└ @ HTTP.Servers C:\Users\barat\.julia\packages\HTTP\4AUPl\src\Servers.jl:382
┌ Info: MeshCat server started. You can open the visualizer by visiting the following URL in your browser:
│ http://127.0.0.1:8704
└ @ MeshCat C:\Users\barat\.julia\packages\MeshCat\9QrxD\src\visualizer.jl:43
```

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:

$$\begin{aligned} \min_{x,u} \quad & \frac{1}{2}(x - x_{guess})^T (x - x_{guess}) + \frac{1}{2} 10^{-3} u^T u \\ \text{st} \quad & \dot{x} = f(x, u) = 0 \end{aligned}$$

Where our primal variables are $x \in \mathrm{R}^{30}$ and $u \in \mathrm{R}^{12}$, that we can stack up in a new variable $y = [x^T, u^T]^T \in \mathrm{R}^{42}$. 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 $\lambda \in \mathrm{R}^{30}$. We are now ready to use Newton's method to solve this equality constrained optimization problem, where we will solve for a variable $z = [y^T, \lambda^T]^T \in \mathrm{R}^{72}$.

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 [11]:
```
# 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]
```

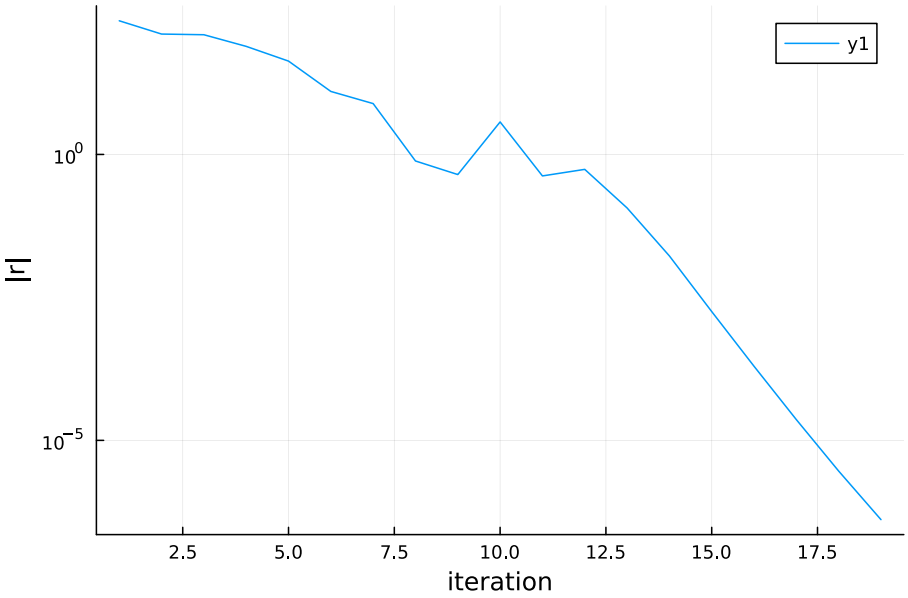
```
# Newton's method will solve for z = [x;u;λ], or z = [y;λ]

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 + 0.5e-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
    return vec(dynamics(model, x, u))
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
    kkt = [vec(FD.gradient(y -> quadruped_cost(y), y)) + FD.jacobian(y -> quadruped_constraint(y), y)' * λ; vec

    return kkt
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]

    # TODO: return Gauss-Newton Jacobian with a regularizer (try 1e-3,1e-4,1e-5,1e-6)
    # and use whatever regularizer works for you
    jac_x_x = FD.jacobian(y -> FD.gradient(y -> quadruped_cost(y), y), y)
    c_x = FD.jacobian(y -> quadruped_constraint(y), y)

    jac = [jac_x_x c_x'; c_x zeros(size(c_x, 1), size(c_x, 1))] + 1e-5*I

    return jac
end
```

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

In [12]:

```
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; tol = 1e-6, verbose = true, max_i
    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 = "|r|"))

    @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|: 217.37629250205396   α: 1.0
iter: 2    |r|: 127.37109282515635   α: 0.5
iter: 3    |r|: 123.94117266840766   α: 1.0
iter: 4    |r|: 77.59042148826394   α: 0.5
iter: 5    |r|: 42.87268187444911   α: 1.0
iter: 6    |r|: 12.637628850400517   α: 1.0
iter: 7    |r|: 7.747833509747416   α: 1.0
iter: 8    |r|: 0.7684073314045563   α: 1.0
iter: 9    |r|: 0.4447347898073554   α: 1.0
iter: 10    |r|: 3.6999226000545837   α: 1.0
iter: 11    |r|: 0.4209252502519447   α: 1.0
iter: 12    |r|: 0.5481167166535564   α: 1.0
iter: 13    |r|: 0.11598642462204954   α: 1.0
iter: 14    |r|: 0.016801076673492784   α: 1.0
iter: 15    |r|: 0.0017879300463144571   α: 1.0
iter: 16    |r|: 0.00019773674365391412   α: 1.0
iter: 17    |r|: 2.305016320890164e-5   α: 1.0
iter: 18    |r|: 2.906001540377103e-6   α: 1.0
iter: 19    |r|: 4.109649053260377e-7   Test Summary:      | Pass  Total   Time
quadruped standing |    3      3  25.0s
```




```
Test.DefaultTestSet("quadruped standing", Any[], 3, false, false, true, 1.738890067e9, 1.738890092049e9, false,
"c:\\CMU\\SEM II\\OCRL\\HW1_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X24sZmlsZQ==.jl")
```

In [13]:

```
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; tol = 1e-6, verbose = false, max_
    # visualizer
    mvis = initialize_visualizer(model)
    set_configuration!(mvis, Z[end][1:state_dim(model)÷2])
    render(mvis)


end
```

```
┌ Info: Listening on: 127.0.0.1:8705, thread id: 1
└ @ HTTP.Servers C:\Users\barat\.julia\packages\HTTP\4AUPl\src\Servers.jl:382
┌ Info: MeshCat server started. You can open the visualizer by visiting the following URL in your browser:
│ http://127.0.0.1:8705
└ @ MeshCat C:\Users\barat\.julia\packages\MeshCat\9QrxD\src\visualizer.jl:43
```

In [86]:
```
import Pkg
Pkg.add("FilePaths")
Pkg.activate(@__DIR__)
Pkg.instantiate()
using LinearAlgebra, Plots
import ForwardDiff as FD
using Printf
using JLD2
```

# Q3 (31 pts): Log-Domain Interior Point Quadratic Program Solver

Here we are going to use the log-domain interior point method described in Lecture 5 to create a QP solver for the following general problem:

$$\begin{aligned} \min_{x} \quad & \frac{1}{2} x^T Q x + q^T x \\ \text{s.t.} \quad & Ax - b = 0 \\ & Gx - h \geq 0 \end{aligned}$$

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

We'll first walk you through the steps to reformulate the problem into an interior point log-domain form that we can solve.

## Part (A): KKT Conditions (2 pts)

To reduce ambiguity (and make sure the test cases pass) for the KKT conditions, make sure that the stationarity condition term for the equality constraint is $(+A^T\mu)$ (not minus). The sign on $G^T\lambda$ is determined by the condition $\lambda \geq 0$.

**TASK**: Introduce Lagrange multipliers $\mu$ for the equality constraint, and $\lambda$ for the inequality constraint and fill in the following for the KKT conditions for the QP above. For complementarity use the $\circ$ symbol (i.e. $a \circ b = 0$).

Your KKT conditions should be in terms of problem data $(Q, q, A, b, G, h)$ and decision variables $(x, \mu \text{ and } \lambda)$

$$\begin{aligned} \mathrm{Qx + q + A'}\mu - G'\lambda &= 0 && \text{(stationarity)} \\ \mathrm{Ax - b} &= 0 && \text{(primal feasibility)} \\ \mathrm{Gx - h} &\geq 0 && \text{(primal feasibility)} \\ \lambda &\geq 0 && \text{(dual feasibility)} \\ \lambda \circ (Gx - h) &= 0 && \text{(complementarity)} \end{aligned}$$

## Part (B): Relaxed Complementarity (2 pts)

In order to apply the log-domain trick, we can introduce a slack variable to represent our inequality constraints $(s)$. This new variable lets us enforce the inequality constraint $(s \geq 0)$ by using a log-domain substitution which is always positive by construction.

We'll also relax the complementarity condition as shown in class.

**TASK**: Modify the primal feasibility and complementarity by doing the following

1. Modify the primal feasiblity condition for the inequalities by adding a slack variable to split it into $Gx - h \geq 0$ condition into $Gx - h = s$ and $s \geq 0$ (there are now two conditions)
2. Relax the complementarity condition so $\lambda \circ s = 0$ becomes $\lambda \circ s = 1^T\rho$ where $\rho$ will be some positive barrier parameter and $1$ is a vector of ones.

The rest of the KKT conditions should remain the same. Write down the KKT conditions (there should now be six) after you've done the above steps.

The decision variables are now $x, \mu, \lambda \text{ and } s$. $\rho$ is the barrier parameter, which is a hyperparameter that will be set by your solver (not a decision variable).

$$\begin{aligned} \mathrm{Qx + q + A'}\mu - G'\lambda &= 0 && \text{(stationarity)} \\ \mathrm{Ax - b} &= 0 && \text{(primal feasibility for equality constraints)} \\ \mathrm{Gx - h - s} &\geq 0 && \text{(primal feasibility for inequality constraints with slack)} \\ \mathrm{s} &\geq 0 && \text{(slack variable non-negativity)} \\ \lambda &\geq 0 && \text{(dual feasibility)} \\ \lambda \circ s &= 1'p && \text{(relaxed complementarity)} \end{aligned}$$

## Part (C): Log-domain Substitution (2 pts)

Finally, to enforce positivity on both $\lambda$ and $s$, we can perform a variable substitution. By using a particular substitution $\lambda = \sqrt{\rho}e^{-\sigma}$ and $s = \sqrt{\rho}e^{\sigma}$ we can also make sure that our relaxed complementarity condition $\lambda \circ s = 1^T\rho$ is always satisfied.

**TASK**: Finally do the following:

1. Define a new variable $\sigma$ and define $\lambda = \sqrt{\rho}e^{-\sigma}$ and $s = \sqrt{\rho}e^{\sigma}$.
2. Replace $\lambda$ and $s$ in your KKT conditions with the new definitions

After the steps above, the decision variables in your KKT conditions should be $x, \mu,$ and $\sigma$. QP solvers work by finding values for the decision variables that satisfy the KKT conditions if they exist.

With these new 6 KKT conditions, three of them are now always true for any value of $x, \mu$ and $\sigma$, meaning our solver doesn't need to consider them (think about properties of exponentials). Eliminate those 3 KKT conditions, leaving 3 conditions that are all equalities (can be written as $= 0$). Since our final conditions are equalities, we can use a root-finding method (specifically Newton's method) to solve for our decision variables.

Write the remaing 3 KKT conditions below (hint: they should all be $= 0$ and the only variables should be x, $\mu$, and $\sigma$).

$$\begin{aligned} \mathrm{Qx} + \mathrm{q} + \mathrm{A}'\mu - G'(\sqrt{\rho}e^{-\sigma}) &= 0 \\ \mathrm{Ax} - \mathrm{b} &= 0 \\ \mathrm{Gx} - \mathrm{h} - \sqrt{\rho}e^{\sigma} &= 0 \end{aligned}$$

## Part (D): Log-domain Interior Point Solver

We can now write our solver! You'll implement two residual functions (matching your residuals in Part A and C), and a function to solve the QP using Newton's method. The solver should work according to the following pseudocode where:

- $\rho$ is the barrier parameter
- kkt_conditions is the KKT conditions from part A
- ip_kkt_conditions is the KKT conditions from part C

```
rho = 0.1 (penalty parameter)
for max_iters
    calculate the Newton step using ip_kkt_conditions and ip_kkt_jac
    perform a linesearch (use the same condition as in Q2, with the norm of the
ip_kkt_conditions as the merit function)
    if norm(ip_kkt_conditions, Inf) < tol, update the barrier parameter
        rho = rho * 0.1
    end
    if norm(kkt_conditions, Inf) < tol
        exit
    end
end
```

In [87]:
```
# 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, xi, μi, σi = 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 functions where z = [x; μ; σ], λ = sqrt(ρ).*exp.(-σ), and s = sqrt(ρ).*exp.(σ

    kkt_res = kkt_conditions(qp, z, ρ)
    ip_res = ip_kkt_conditions(qp, z)
    ip_jac = ip_kkt_jacobian(qp, z)
    x, μ, λ = solve_qp(qp; verbose = true, max_iters = 100, tol = 1e-8)

"""

using JLD2
using FilePaths
using Printf
using LinearAlgebra


# Helper functions (you can use or not use these)
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

"""
    kkt_res = kkt_conditions(qp, z, ρ)

Return the KKT residual from part A as a vector (make sure to clamp the inequalities!)
In Julia, use the following for elementwise min.
elementwise_min = min.(a, b) # This is elementwise min
scalar_elementwise_min = min.(a, 0) # You can also take an elementwise min with a scalar
"""
function kkt_conditions(qp::NamedTuple, z::Vector, ρ::Float64)::Vector
    x, μ, σ = z[qp.xi], z[qp.μi], z[qp.σi]

    # TODO compute λ from σ and ρ
    λ = sqrt(ρ).*exp.(-σ)

    # TODO compute and return KKT conditions
    res = [qp.Q*x + qp.q + qp.A'*μ - qp.G'*λ; c_eq(qp, x); min.(h_ineq(qp, x), 0); λ.*h_ineq(qp, x)]
    return res
end

"""
    ip_res = ip_kkt_conditions(qp, z)

Return the interior point KKT residual from part C as a vector
"""
function ip_kkt_conditions(qp::NamedTuple, z::Vector, ρ::Float64)::Vector
    x, μ, σ = z[qp.xi], z[qp.μi], z[qp.σi]

    # TODO compute λ and s from σ and ρ
    λ = sqrt(ρ).*exp.(-σ)
    s = sqrt(ρ).*exp.(σ)

    # TODO compute and return IP KKT conditions
    res = [qp.Q*x + qp.q + qp.A'*μ - qp.G'*λ; c_eq(qp, x); h_ineq(qp, x) - s]
    return res
end

"""
    ip_jac = ip_jacobian(qp, z, ρ)

Return the full Newton jacobian of the interior point KKT conditions (part C) with respect to z
Construct it analytically (don't use auto differentiation)
"""
function ip_kkt_jac(qp::NamedTuple, z::Vector, ρ::Float64)::Matrix
    x, μ, σ = z[qp.xi], z[qp.μi], z[qp.σi]

    # TODO: return full Newton jacobian (don't use ForwardDiff)
    λ = sqrt(ρ).*exp.(-σ)
    s = sqrt(ρ).*exp.(σ)

    j11 = qp.Q
    j12 = qp.A'
    j13 = qp.G'*diagm(λ)

    j21 = qp.A
    j22 = zeros(length(qp.b), length(qp.b))
    j23 = zeros(length(qp.b), length(qp.h))

    j31 = qp.G
    j32 = zeros(length(qp.h), length(qp.b))
    j33 = diagm(-s)

    J = [j11 j12 j13;
         j21 j22 j23;
         j31 j32 j33] + 1e-6*I
    return J
end

function logging(qp::NamedTuple, main_iter::Int, z::Vector, ρ::Real, α::Real)
    x, μ, σ = z[qp.xi], z[qp.μi], z[qp.σi]

    # TODO: compute λ
    λ = sqrt(ρ).*exp.(-σ)

    # TODO: stationarity norm
    stationarity_norm = norm(kkt_conditions(qp, z, ρ), Inf)

    @printf("%3d  % 7.2e  % 7.2e  % 7.2e  % 7.2e  %5.0e  %5.0e\n",
          main_iter, stationarity_norm, minimum(h_ineq(qp,x)),
          norm(c_eq(qp,x),Inf), abs(dot(λ,h_ineq(qp,x))), ρ, α)
end
```

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

Solve the QP using the method defined in the pseudocode above, where z = [x; μ; σ], λ = sqrt(ρ).*exp.(-σ), and s
"""
function solve_qp(qp; verbose = true, max_iters = 100, tol = 1e-8)
    # Init solution vector z = [x; μ; σ]
    z = zeros(length(qp.q) + length(qp.b) + length(qp.h))

    if verbose
        @printf "iter   |∇Lₓ|      min(h)       |c|       compl     ρ      α\n"
        @printf "-------------------------------------------------------------\n"
    end

    ρ = 1.0e-1
    # TODO: implement your solver according to the above pseudocode
    for main_iter = 1:max_iters

        # Save the step length (α) from your linesearch for logging
        ip_kkt_res = ip_kkt_conditions(qp, z, ρ)
        J = ip_kkt_jac(qp, z, ρ)

        dz = -J\ip_kkt_res
        α = 1.0

        for i = 1:10
            z_new = z + α*dz
            if norm(ip_kkt_conditions(qp, z_new, ρ), Inf) < norm(ip_kkt_res, Inf)
                break
            end
            α *= 0.5
        end
        z += α*dz

        # Convergence criteria based on tol

        kkt_res = kkt_conditions(qp, z, ρ)
        if norm(kkt_res, Inf) < tol
            x, μ, σ = z[qp.xi], z[qp.μi], z[qp.σi]
            λ = sqrt(ρ).*exp.(-σ)
            return x, μ, λ
        end

        if norm(ip_kkt_res, Inf) < tol && ρ > 1e-6
            ρ *= 0.1
        end

        if verbose
            logging(qp, main_iter, z, ρ, α)
        end

    end
    error("qp solver did not converge")
end
```

```
solve_qp
```

## QP Solver test

In [88]:
```
# 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(kkt_conditions(qp, qp_solutions.z, qp_solutions.ρ))<1e-3;
    @test norm(ip_kkt_conditions(qp, qp_solutions.z, qp_solutions.ρ))<1e-3;
    @test norm(ip_kkt_jac(qp, qp_solutions.z, qp_solutions.ρ) - FD.jacobian(dz -> ip_kkt_conditions(qp, dz, qp_s
    @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
```

```
Test.DefaultTestSet("qp solver", Any[], 6, false, false, true, 1.73889041136e9, 1.738890412873e9, false, "c:\\CM
U\\SEM II\\OCRL\\HW1_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X11sZmlsZQ==.jl")
```

# 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 Interior Point 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 $\mu \in \mathrm{R}$ is the normal force. The velocity $v \in \mathrm{R}^2$ and position $q \in \mathrm{R}^2$ 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:

$$\begin{aligned} Jq_{k+1} &\geq 0 && \text{(don't fall through the ice)} \\ \mu_{k+1} &\geq 0 && \text{(normal forces only push, not pull)} \\ \mu_{k+1}Jq_{k+1} &= 0 && \text{(no force at a distance)} \end{aligned}$$

## Part (E): 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.

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

**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.

**PUT ANSWER HERE:**

$$\begin{aligned} Mv_{k+1} + M(\Delta t \cdot g - v_k) + \Delta t J^T\lambda &= 0 && \text{(stationarity)} \\ J(q_k + \Delta t \cdot v_{k+1}) &= 0 && \text{(primal feasibility)} \\ \lambda &\geq 0 && \text{(dual feasibility)} \\ \lambda^T J(q_k + \Delta t v_{k+1}) &= 0 && \text{(complementary slackness)} \end{aligned}$$

## Part (F): Brick Simulation (5 pts)

In [89]:
```
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
    M = [mass 0; 0 mass]
    g = [0; 9.81]
    J = [0 1.0]

    qp = (
        Q = M,
        q = M*(Δt*g - v),
        A = zeros(0,2), # don't edit this
        b = zeros(0),   # don't edit this
        G = J*Δt,
        h = -J*q,
        xi = 1:2,       # don't edit this
        μi = [],        # don't edit this
        σi = 3:3        # don't edit this
    )

    return qp
end
```

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

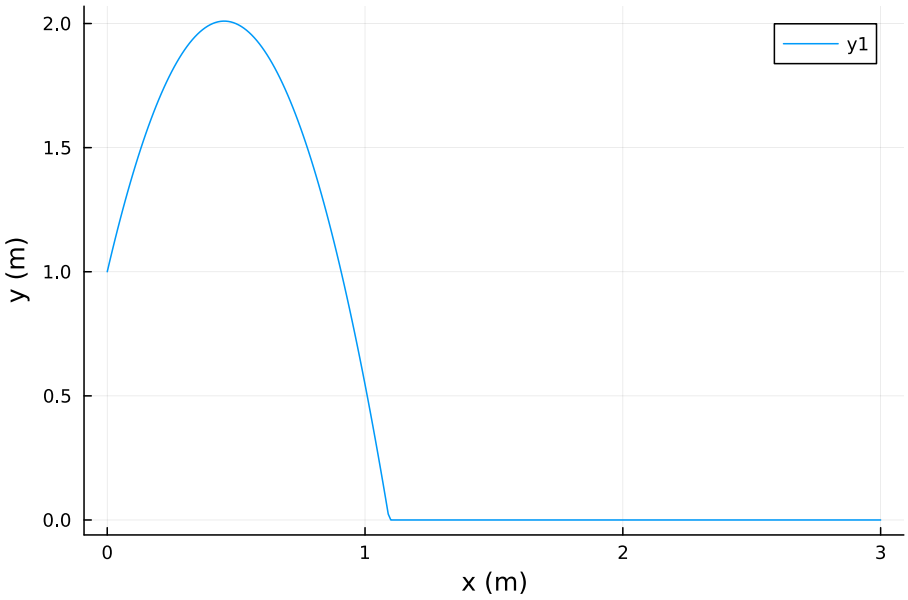
In [90]:
```
@testset "brick qp" begin

    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.DefaultTestSet("brick qp", Any[], 10, false, false, true, 1.738890412907e9, 1.738890412932e9, false, "c:\\CMU\\SEM II\\OCRL\\HW1_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X20sZmlsZQ==.jl")
```

In [91]:

```
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 in 1:N-1
        qp = brick_simulation_qp(qs[k], vs[k], Δt = dt)

        v_next, μ, λ = solve_qp(qp; verbose = true, max_iters = 200, tol = 1e-6)

        vs[k+1] = v_next
        qs[k+1] = qs[k] + dt*vs[k+1]
    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
```




# Part G (5 pts): Solve a QP

Use your QP solver to solve the following optimization problem:

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

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

In [92]:
```
@testset "part D" begin

    y = randn(2)
    a = randn()
    b = randn()

    #TODO: Create your qp and solve it. Don't forget the indices (xi, μi, and σi)
    function my_qp()

        Q = [1 0.3 0 0; 0.3 1 0 0; 0 0 2 0; 0 0 0 4]
```

```
        q = [-2; 3.4; 2; 4]
        A = [0 0 1 1; -1 2.3 1 -2]
        b = [1; 3]
        G = [-1.0  0.0    0.0   0.0;
            0.0  -1.0   0.0   0.0;
            1.0   0.0   0.0   0.0;
            0.0   1.0   0.0  0.0;
            0.0   0.0   1.0   0.0;
            0.0   0.0   -1.0   0.0;
            0.0   0.0   0.0   -1.0;
            0.0   0.0   0.0   1.0]
        h = [-1.0; -1.0; -0.5; -0.5; -1.0; -1.0; -1.0; -1.0;]


        qp = (
            Q  = Q,
            q  = q,
            A = A,
            b = b,
            G  = G,
            h  = h,
            xi = 1:4,
            μi = 5:6,
            σi = 7:14
        )

        return qp
    end

    qp = my_qp()

    x, μ, λ = solve_qp(qp; verbose = true, max_iters = 200, tol = 1e-6)

    y = x[1:2]
    a = x[3]
    b = x[4]

    @test norm(y - [-0.080823; 0.834424]) < 1e-3
    @test abs(a - 1) < 1e-3
    @test abs(b) < 1e-3
end
```

```
Test.DefaultTestSet("part D", Any[], 3, false, false, true, 1.73889041534e9, 1.73889041589e9, false, "c:\\CMU\\SEM II\\OCRL\\HW1_S25\\jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X23sZmlsZQ==.jl")
```