In [6]:
```
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
using LinearAlgebra, Plots
import ForwardDiff as FD
using Printf
using JLD2
```

```
  Activating environment at `~/ocrl_ws/16-745/HW1_S23/Project.toml`
```

Q2 (20 pts): Augmented Lagrangian Quadratic Program Solver

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:

$$\begin{aligned} \min_x \quad & \frac{1}{2}x^TQx + q^Tx && (1) \\ \text{s.t.} \quad & Ax - b = 0 && (2) \\ & Gx - h \leq 0 && (3) \end{aligned}$$

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:

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

where $\circ$ is element-wise multiplication.

In [22]:
```
# 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")

#      possible point of failure maybe think of some diff method  to calculate

    h = h_ineq(qp ,x)
    Iρ = zeros(length(h),length(h))

    for i = 1:length(h)
        if h[i] < 0 && μ[i] == 0
            Iρ[i,i] = 0
        else
            Iρ[i,i] = ρ
        end
    end

    return Iρ
end

function lagrangian(qp::NamedTuple, x::Vector, λ::Vector, μ::Vector, ρ::Real):
    cx = c_eq(qp ,x)
    hx = h_ineq(qp ,x)
    return cost(qp ,x)+ λ' * cx + μ' * hx
end

function augmented_lagrangian(qp::NamedTuple, x::Vector, λ::Vector, μ::Vector,
#
#     error("not implemented")

    cx = c_eq(qp ,x)
    hx = h_ineq(qp ,x)

    Iρ = mask_matrix(qp ,x , μ, ρ)

    lag = lagrangian(qp ,x , λ ,μ, ρ)

    return lag + 0.5 * ρ * cx' * cx + 0.5 * hx' * Iρ * hx
```

```
end

function augemented_lag_grad(qp::NamedTuple, x::Vector, λ::Vector, μ::Vector, 
    Iρ = mask_matrix(qp ,x , μ, ρ)
    return qp.Q * x + qp.q +  qp.A' * (λ + ρ * c_eq(qp ,x)) + qp.G' * (μ + Iρ 
end

function augemented_lag_hessian(qp::NamedTuple, x::Vector, λ::Vector, μ::Vecto
    Iρ = mask_matrix(qp ,x , μ, ρ)
    return qp.Q + ρ * qp.A' * qp.A + qp.G' * Iρ * qp.G
end

function stationarity(qp::NamedTuple, x::Vector, λ::Vector, μ::Vector, ρ::Real
    return FD.gradient(x -> augmented_lagrangian(qp ,x , λ ,μ, ρ), x)

#          δcx = FD.jacobian(x-> c_eq(qp, x),x)
#          δhx = FD.jacobian(x-> h_ineq(qp, x),x)
#          return FD.gradient(x-> cost(qp ,x),x) + δcx'*λ + δhx'μ
end

function logging(qp::NamedTuple, main_iter::Int, AL_gradient::Vector, x::Vecto
    # TODO: stationarity norm
    stationarity_norm = norm(stationarity(qp, x, λ, μ, ρ)) # fill this in
    @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,x
          norm(c_eq(qp,x),Inf), abs(dot(μ,h_ineq(qp,x))), ρ)
end

function kkt_cond(qp::NamedTuple, x::Vector, λ::Vector, μ::Vector, ρ::Real)
    cx = c_eq(qp ,x)
    hx = h_ineq(qp ,x)

#      return [stationarity(qp, x, λ, μ, ρ) ; cx; hx; μ; abs(dot(μ,h_ineq(qp,x))
    return stationarity(qp, x, λ, μ, ρ)
end

function newton_step(qp::NamedTuple, x::Vector, λ::Vector, μ::Vector, ρ::Real)
    kkt_jacobian = FD.hessian(x -> augmented_lagrangian(qp, x, λ, μ, ρ), x)
    return -kkt_jacobian \ kkt_cond(qp,x,λ,μ,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
    α = 1.0
    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, augemented_lag_grad(qp, x, λ, μ, ρ), x, λ, 
        end
#         return x, λ, μ
#       Use Newton method to calculate the change in x
        Δx = newton_step(qp,x,λ,μ,ρ)

#         skipping line search as it is specified that use α = 1

#         update x
        x = x + α.*Δx

#         update λ & μ
        λ = λ + ρ * c_eq(qp ,x)
        μ = max.(0, (μ + ρ * h_ineq(qp ,x) ) )

        ρ = ρ * ϕ

        # TODO: convergence criteria based on tol
        if norm(c_eq(qp,x), Inf) < tol && max(0,maximum(h_ineq(qp,x))) < 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   5.60e+01   5.60e+01   4.38e+00   6.49e+00   0.00e+00  1e+00
  2   7.50e+01   7.50e+01   1.55e+00   1.31e+00   2.64e+00  1e+01
  3   9.63e+01   9.63e+01   2.96e-02   3.04e-01   4.74e-02  1e+02
  4   4.21e+01   4.21e+01   6.37e-03   1.35e-02   7.39e-03  1e+03
  5   2.34e+03   2.34e+03   6.84e-02   1.55e-04   4.67e+00  1e+04
  6   2.12e+03   2.12e+03   2.12e-06   3.74e-06   2.71e-04  1e+05
  7   1.30e-01   1.30e-01  -1.94e-08   3.42e-08   2.18e-08  1e+06
```

Out[22]:
```
([-0.326230805713403, 0.24943797997177494, -0.43226766440520603, -1.4172246971
241924, -1.3994527400875825, 0.6099582408523401, -0.07312202122166526, 1.30314
7752200007, 0.5389034791065863, -0.7225813651685381], [-0.1283519512258231, -
2.837624169038528, -0.8320804497331935], [0.03635294279645507, 0.0, 0.0, 1.059
4444951620436, 0.0])
```

QP Solver test (10 pts)

In [23]:
```
# 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   5.60e+01   5.60e+01   4.38e+00   6.49e+00   0.00e+00  1e+00
  2   7.50e+01   7.50e+01   1.55e+00   1.31e+00   2.64e+00  1e+01
  3   9.63e+01   9.63e+01   2.96e-02   3.04e-01   4.74e-02  1e+02
  4   4.21e+01   4.21e+01   6.37e-03   1.35e-02   7.39e-03  1e+03
  5   2.34e+03   2.34e+03   6.84e-02   1.55e-04   4.67e+00  1e+04
  6   2.12e+03   2.12e+03   2.12e-06   3.74e-06   2.71e-04  1e+05
Test Summary: | Pass  Total
qp solver     |    3      3
```

Out[23]:
```
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

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

and $\lambda \in \mathbb{R}$ 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

$$\begin{bmatrix} \frac{1}{m}J^T\lambda_{k+1} - g \\ v_{k+1} \end{bmatrix}$$

$$

We also have the following contact constraints:

$$\begin{aligned} Jq_{k+1} &\geq 0 && \text{(don't fall through the ice)} && (9) \\ \lambda_{k+1} &\geq 0 && \text{(normal forces only push, not pull)} && (10) \\ \lambda_{k+1} J q_{k+1} &= 0 && \text{(no force at a distance)} && (11) \end{aligned}$$

Part (a): QP formulation (5 pts)

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

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

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.

PUT ANSWER HERE:

Comparing the above given cost function and constraints to the form

$$\begin{aligned} \min_x \quad & \frac{1}{2} x^T Q x + q^T x && (14) \\ \text{s.t.} \quad & Ax - b = 0 && (15) \\ & Gx - h \leq 0 && (16) \end{aligned}$$

we have,

$$Q = M = mI_{2\times 2}$$

$$x = v_{k+1}$$

$$q_k = M(\Delta t \cdot g - v_k)$$

$$G = -J\Delta t$$

$$h = Jq_k$$

Since, there i-s not equality constraint $Ax - b$ does not exist

The KKT conditions for the above are:

$$\begin{aligned} \frac{\delta L}{\delta x} = Qx + q + G^T \mu &= 0 && \text{stationarity} && (17) \\ & && && (18) \\ Gx - h &\leq 0 && \text{primal feasibility} && (19) \\ \mu &\geq 0 && \text{dual feasibility} && (20) \\ \mu \circ (Gx - h) &= 0 && \text{complementarity} && (21) \end{aligned}$$

Now we will expand these conditions to retrive the discrete-time dynamics:

1. Expanding the stationarity:

$$Qx + q + G^T\mu = 0$$

$$Mv_{k+1} + M(\Delta t \cdot g - v_k) + (-J\Delta t)^T\mu = 0$$

a. since Δt is a scalar $(-J\Delta t)^T = -J^T\Delta t$

b. setting $\mu = \lambda_{k+1}$ where $\lambda_{k+1} \geq 0$ (normal force only push, not pull) eq 11

$$Mv_{k+1} + M(\Delta t \cdot g - v_k) + -J^T\Delta t\lambda_{k+1} = 0$$

$$M(v_{k+1} + (\Delta t \cdot g - v_k)) + -J^T\Delta t\lambda_{k+1} = 0$$

$$M(v_{k+1} + (\Delta t \cdot g - v_k)) = J^T\Delta t\lambda_{k+1}$$

$$(v_{k+1} + (\Delta t \cdot g - v_k)) = M^{-1}J^T\Delta t\lambda_{k+1}$$

$$v_{k+1} = -(\Delta t \cdot g - v_k)) + M^{-1}J^T\Delta t\lambda_{k+1}$$

$$v_{k+1} = v_k + \Delta t \cdot (-g + M^{-1}J^T\lambda_{k+1})$$

Which is the dynamics equation 1

2. Expanding $Gx - h \leq 0$ - primal feasibility

$$Gx - h \leq 0$$

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

$$J\Delta t v_{k+1} + Jq_k \geq 0$$

$$J(\Delta t v_{k+1} + q_k) \geq 0$$

setting $q_{k+1} = (\Delta t v_{k+1} + q_k)$ from the dynamics eq2

$$Jq_{k+1} \geq 0$$

Which gives us the constraint - don't fall through the ice eq 9

3. Expanding $\mu \circ (Gx - h) = 0$ - complpementarity

$$\lambda_{k+1} \circ Jq_{k+1} = 0 \text{ - no force at a distance eq 11}$$

Thus we can see that the KKT conditions are equivalent to the dynamics problem stated proviously.

Brick Simulation (5 pts)

In [12]:
```
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]
    J = [0 1]
    M = mass .* [1.0 0.0 ; 0.0 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
    )

    return qp
end
```

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

In [13]:
```
@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
```

Out[13]: Test.DefaultTestSet("brick qp", Any[], 10, false, false)

In [46]:
```
include(joinpath(@__DIR__, "animate_brick.jl"))

function kkt_brick(qp::NamedTuple, x::Vector, μ::Real)
```

```julia
    return qp.Q * x + qp.q + qp.G' * μ
end

function brick_newton_step(qp::NamedTuple, x::Vector, μ::Real)
    kkt_func(x) = [kkt_brick(qp,x,μ)]
    @show kkt_brick(qp,x,μ)
    @show qp
    @show x
    @show μ
    kkt_jacobian = FD.jacobian(dx -> kkt_func(dx), x)
    @show kkt_jacobian(qp,x,μ)
    return -kkt_jacobian \ kkt_brick(qp,x,μ)
end


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]

    mass = 1.0
    tol = 1e-5
    g = [0;9.81]
    J = [0 1]
    λ = 9.81
    # 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
#       Use Newton method to calculate the change in x
        brick_qp = brick_simulation_qp(qs[k], vs[k])
        vs[k+1], _, λ = solve_qp(brick_qp; verbose = false, tol = 1e-8)
        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
```

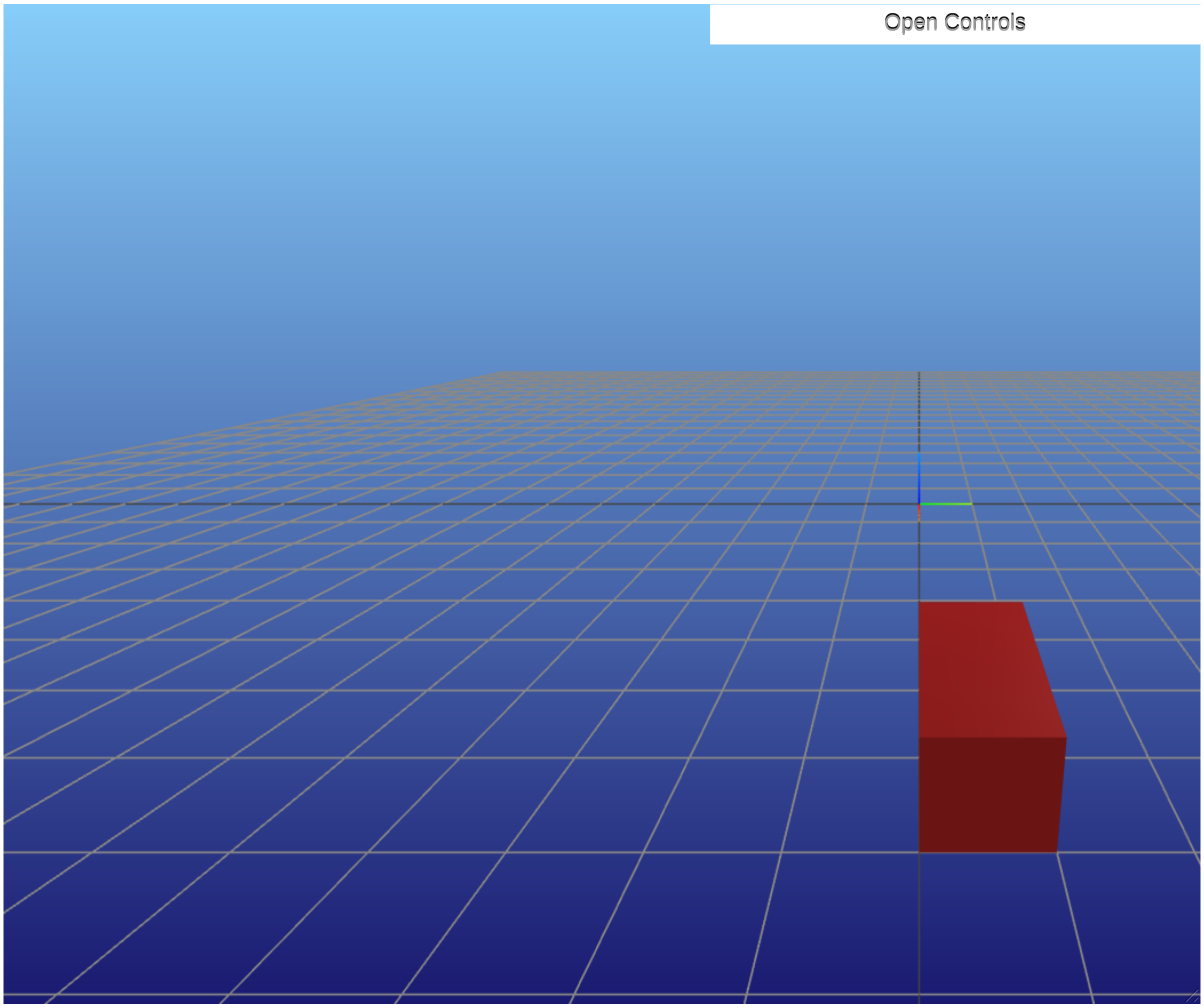
```
    @show @test abs(ys[112]) < 1e-2

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

    animate_brick(qs)

end
```

```
#= In[46]:52 =# @test(abs(maximum(ys) - 2) < 0.1) = Test Passed
#= In[46]:53 =# @test(minimum(ys) > -0.01) = Test Passed
#= In[46]:54 =# @test(abs(xs[end] - 3) < 0.01) = Test Passed
#= In[46]:57 =# @test(maximum(xdot) < 1.0001) = Test Passed
#= In[46]:58 =# @test(minimum(xdot) > 0.9999) = Test Passed
#= In[46]:59 =# @test(ys[110] > 0.01) = Test Passed
#= In[46]:60 =# @test(abs(ys[111]) < 0.01) = Test Passed
#= In[46]:61 =# @test(abs(ys[112]) < 0.01) = Test Passed
```

b'
\n'

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

Out[46]:



In [ ]:

In [ ]: