In [1]:
```julia
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
using LinearAlgebra, Plots
import ForwardDiff as FD
using Printf
using JLD2
```

```
  Activating project at `~/OCRL/HW1_S25`
```

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:

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

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 $h \in \mathbb{R}^p$.

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 μ for the equality constraint, and λ for the inequality constraint and fill in the following for the KKT conditions for the QP. For complementarity use the $\circ$ symbol (i.e. $a \circ b = 0$)

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

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 your KKT conditions by doing the following:

1. Add a slack variable to split the primal feasibility $Gx - h \geq 0$ condition into $Gx - h = s$ and $s \geq 0$
2. Relax the complementarity condition so $\lambda \circ s = 0$ becomes $\lambda \circ s = 1^T\rho$ where ρ will be some positive barrier parameter and 1 is a vector of ones.

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

$$
\begin{aligned}
\nabla_x L = Qx + q + A^T\mu - G^T\lambda = 0 \qquad & \text{(stationarity)} && (9)\\
Ax - b = 0 \qquad & \text{(primal feasibility)} && (10)\\
Gx - h = s \qquad & \text{(primal feasibility)} && (11)\\
s \geq 0 \qquad & \text{(primal feasibility)} && (12)\\
\lambda \geq 0 \qquad & \text{(dual feasibility)} && (13)\\
\lambda \circ s = 1^T\rho \qquad & \text{(complementarity)} && (14)
\end{aligned}
$$

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

Finally, to enforce positivity on both λ 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 σ and define $\lambda = \sqrt{\rho}e^{-\sigma}$ and $s = \sqrt{\rho}e^{\sigma}$.
2. Replace λ and s in your KKT conditions with the new definitions

Three of your KKT conditions from (B) should now be satisfied by construction. Write down the remaining 3 KKT conditions (hint: they should all be $= 0$ and the only variables should be x, μ, and σ).

$$
\begin{aligned}
\nabla_x L = Qx + q + A^T\mu - G^T\sqrt{\rho}e^{-\sigma} = 0 \qquad & \text{(stationarity)} && (15)\\
Ax - b = 0 \qquad & \text{(primal feasibility)} && (16)\\
Gx - h - \sqrt{\rho}e^{\sigma} = 0 \qquad & \text{(primal feasibility)} && (17)
\end{aligned}
$$

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:

- ρ 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 [2]:
```
# 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 =

    kkt_res = kkt_conditions(qp, z, ρ)
    ip_res = ip_kkt_conditions(qp, z)
```

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

"""

# 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!)
"""
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)
    s = sqrt(ρ) .* exp.(σ)
    # TODO compute and return KKT conditions
    kkt_res = [
        qp.Q*x + qp.q + qp.A' * μ - qp.G' * λ;
        c_eq(qp,x);
        min.(h_ineq(qp,x),0);
        min.(λ,0);
        λ' * h_ineq(qp,x)
        ]
    #error("kkt_conditions not implemented")
    return kkt_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
    ip_res = [
        qp.Q*x + qp.q + qp.A' * μ - qp.G' * λ;
        c_eq(qp,x);
        qp.G*x - qp.h - s
        ]

    #error("ip_kkt_conditions not implemented")
    return ip_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]

    λ = sqrt(ρ) .* exp.(-σ)
    s = sqrt(ρ) .* exp.(σ)

    n = length(qp.xi)
    m_eq = length(qp.μi)
    m_ineq = length(qp.σi)

    ip_jac = [
```

```julia
        qp.Q qp.A' -qp.G'*Diagonal(-λ);
        qp.A zeros(m_eq, m_eq) zeros(m_eq, m_ineq);
        qp.G zeros(m_ineq, m_eq) Diagonal(-s)
    ]

    # β = 1e-5
    # ip_jac += Diagonal([β*ones(length(x)); β*ones(length(μ)); -β*ones(length(σ))])
    return ip_jac
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(qp.Q*x + qp.q + qp.A' * μ - qp.G' * λ) # fill this in

    @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)
"""

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

    # TODO: implement your solver according to the above pseudocode
    ρ = 0.1
    for main_iter = 1:max_iters

        # TODO: make sure to save the step length (α) from your linesearch for logging

        r = ip_kkt_conditions(qp, z, ρ)
        Δz = -ip_kkt_jac(qp, z, ρ) \ r
        # line search
        alpha = 1
        for line = 1:max_iters
            if norm(ip_kkt_conditions(qp, z + alpha * Δz, ρ)) < norm(ip_kkt_conditions(qp, z, ρ
                break
            end
            alpha = alpha * 0.5
        end

        z = z + alpha * Δz

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

        # TODO: convergence criteria based on tol
        if norm(kkt_conditions(qp, z, ρ), Inf) < tol
            x = z[qp.xi]
            λ = sqrt(ρ) .* exp.(-z[qp.σi])
            μ = z[qp.μi]
            return x, μ, λ

        elseif norm(ip_kkt_conditions(qp, z, ρ), Inf) < tol
            ρ = ρ * 0.1
        end


    end
```

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

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

QP Solver test

In [3]:
```
# 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_condit
    @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|      min(h)       |c|        compl     ρ      α
----------------------------------------------------------------
  1   5.13e+00  -3.51e-01   2.22e-16   6.94e-01  1e-01  1e+00
  2   1.14e+00   6.08e-02   3.33e-16   3.80e-01  1e-01  1e+00
  3   1.16e-01   8.52e-02   8.88e-16   4.52e-01  1e-01  1e+00
  4   5.60e-03   9.01e-02   2.22e-16   4.90e-01  1e-01  1e+00
  5   4.43e-04   9.03e-02   6.66e-16   4.99e-01  1e-01  1e+00
  6   1.23e-06   9.03e-02   4.44e-16   5.00e-01  1e-01  1e+00
  7   7.41e-12   9.03e-02   4.44e-16   5.00e-01  1e-01  1e+00
  8   5.34e-01   3.03e-02   1.11e-16   9.05e-02  1e-02  5e-01
  9   2.25e-02   9.16e-03   2.22e-16   4.93e-02  1e-02  1e+00
 10   1.20e-04   9.28e-03   2.22e-16   5.00e-02  1e-02  1e+00
 11   1.08e-08   9.28e-03   2.22e-16   5.00e-02  1e-02  1e+00
 12   2.83e-01   2.98e-03   4.44e-16   8.59e-03  1e-03  5e-01
 13   9.83e-03   9.35e-04   1.78e-15   4.93e-03  1e-03  1e+00
 14   1.88e-05   9.40e-04   8.88e-16   5.00e-03  1e-03  1e+00
 15   2.75e-10   9.40e-04   0.00e+00   5.00e-03  1e-03  1e+00
 16   2.43e-01   2.99e-04   1.11e-16   8.89e-04  1e-04  5e-01
 17   1.02e-02   9.38e-05   0.00e+00   4.93e-04  1e-04  1e+00
 18   7.24e-05   9.43e-05   2.22e-16   5.00e-04  1e-04  1e+00
 19   1.87e-08   9.43e-05   8.88e-16   5.00e-04  1e-04  1e+00
 20   2.27e-01   2.99e-05   8.88e-16   9.10e-05  1e-05  5e-01
 21   8.54e-03   9.39e-06   8.88e-16   4.98e-05  1e-05  1e+00
 22   1.13e-05   9.44e-06   1.78e-15   5.00e-05  1e-05  1e+00
 23   4.11e-11   9.44e-06   2.22e-16   5.00e-05  1e-05  1e+00
 24   2.24e-01   2.99e-06   8.88e-16   9.11e-06  1e-06  5e-01
 25   8.36e-03   9.39e-07   3.33e-16   4.99e-06  1e-06  1e+00
 26   1.05e-05   9.44e-07   8.88e-16   5.00e-06  1e-06  1e+00
 27   1.68e-11   9.44e-07   2.22e-16   5.00e-06  1e-06  1e+00
 28   2.24e-01   2.98e-07   8.88e-16   9.11e-07  1e-07  5e-01
 29   8.34e-03   9.39e-08   8.88e-16   4.99e-07  1e-07  1e+00
 30   1.05e-05   9.44e-08   4.44e-16   5.00e-07  1e-07  1e+00
 31   1.66e-11   9.44e-08   2.22e-16   5.00e-07  1e-07  1e+00
Test Summary: | Pass  Total  Time
qp solver     |    6      6  5.7s
Test.DefaultTestSet("qp solver", Any[], 6, false, false, true, 1.738902786985502e9, 1.7389027926
63773e9, false, "/home/burger/OCRL/HW1_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X11
sZmlsZQ==.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 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:

$$
\begin{aligned}
Jq_{k+1} &\geq 0 && \text{(don't fall through the ice)} && (18) \\
\mu_{k+1} &\geq 0 && \text{(normal forces only push, not pull)} && (19) \\
\mu_{k+1}Jq_{k+1} &= 0 && \text{(no force at a distance)} && (20)
\end{aligned}
$$

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.

$$
\begin{aligned}
\text{minimize}_{v_{k+1}} \quad & \frac{1}{2}v_{k+1}^T M v_{k+1} + [M(\Delta t \cdot g - v_k)]^T v_{k+1} && (21) \\
\text{subject to} \quad & J(q_k + \Delta t \cdot v_{k+1}) \geq 0 && (22)
\end{aligned}
$$

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:

The KKT conditions are:

$$
\begin{aligned}
\nabla_{v_{k+1}} L = MV_{k+1} + M[\Delta t \cdot g - v_k] - \lambda^T \cdot \Delta t \cdot J^T = 0 && (23) \\
J(q_k + \Delta t \cdot v_{k+1}) \geq 0 && (24) \\
\lambda \geq 0 && (25) \\
\lambda \circ J(q_k + \Delta t \cdot v_{k+1}) = 0 && (26)
\end{aligned}
$$

Considering the dynamics of the block:
Using the discretised dynamics,

$$
\begin{aligned}
& v_{k+1} = v_k + \Delta t \cdot (1/m) \cdot J^T \cdot \mu_{k+1} - g \cdot \Delta t && (27) \\
& q_{k+1} = q_k + \Delta t \cdot v_{k+1} && (28)
\end{aligned}
$$

Equation (1) can be simplified to the following equation, which is the same as the first KKT condition if μ is replaced with λ

$$Mv_{k+1} + M(g\Delta t - v_k) - J^T \cdot \Delta t \cdot \mu_{k+1} = 0 \qquad (29)$$

Considering the constraints and simplifying,

$$J \cdot q_{k+1} \geq 0 \implies J \cdot (q_k + \Delta t \cdot v_{k+1}) \geq 0 \tag{30}$$
$$\mu_{k+1} \geq 0 \implies \lambda \geq 0 \tag{31}$$
$$\mu_{k+1} J q_{k+1} = 0 \implies \lambda \circ J(q_k + \Delta t \cdot v_{k+1}) = 0 \tag{32}$$

Hence, the given QP is equivalent to the discretized dynamics of the falling brick as the KKT conditions match.

Part (F): Brick Simulation (5 pts)

In [4]:
```
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
    J = [0 1]
    g = [0; 9.81]
    M = Matrix(mass * I(2))

    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
```

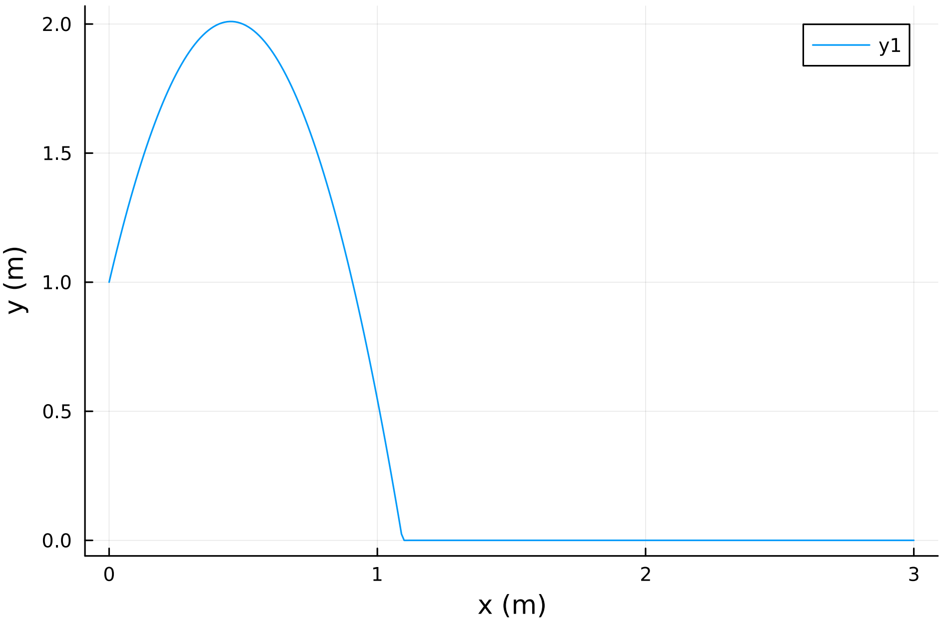
```
brick_simulation_qp (generic function with 1 method)
```

In [5]:
```
@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
```

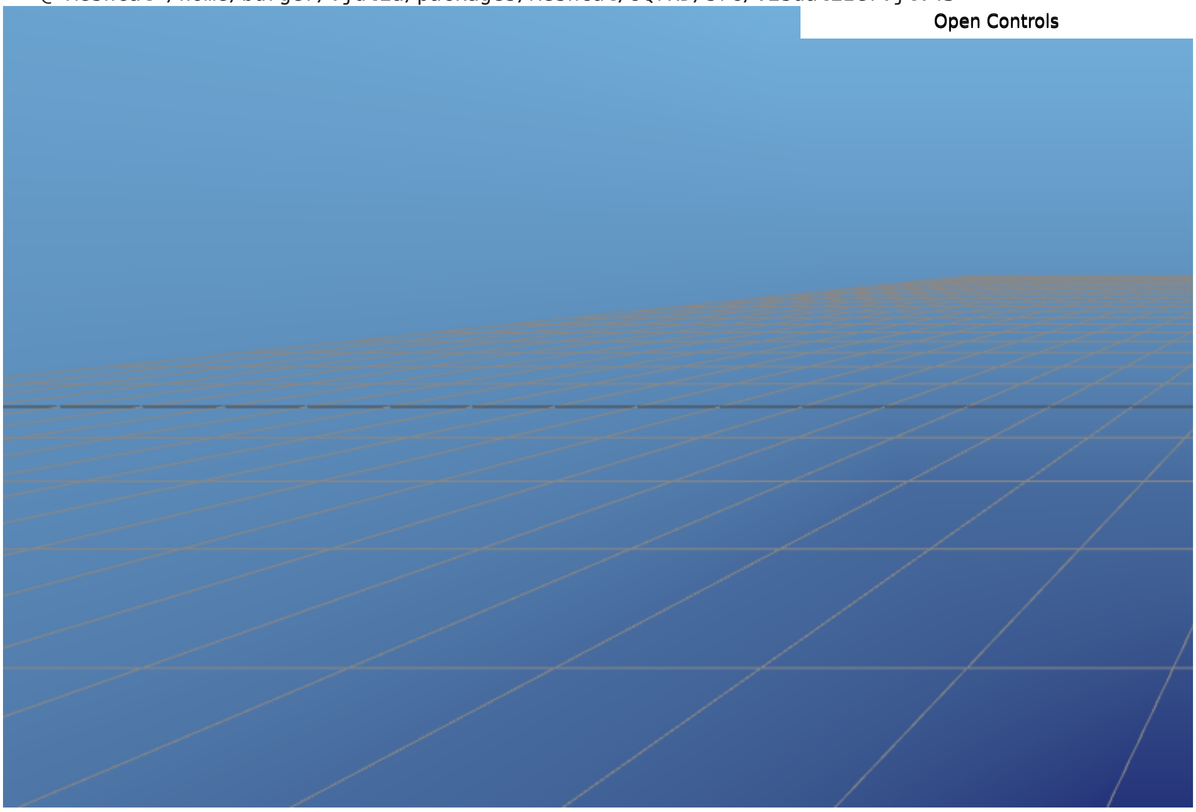
```
Test Summary: | Pass  Total  Time
brick qp      |   10     10  0.5s
Test.DefaultTestSet("brick qp", Any[], 10, false, false, true, 1.738902793233919e9, 1.7389027937
24835e9, false, "/home/burger/OCRL/HW1_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X20
sZmlsZQ==.jl")
```

In [6]:
```
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 i = 1:N-1
        quadratic_problem = brick_simulation_qp(qs[i], vs[i])
        vs[i+1], _ , _ = solve_qp(quadratic_problem, verbose = false, max_iters = 100, tol = 1e-
        qs[i+1] = qs[i] + dt * vs[i+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
```



```
#= /home/burger/OCRL/HW1_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X21sZmlsZQ==.jl:2
9 =# @test(abs(maximum(ys) - 2) < 0.1) = Test Passed
#= /home/burger/OCRL/HW1_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X21sZmlsZQ==.jl:3
0 =# @test(minimum(ys) > -0.01) = Test Passed
#= /home/burger/OCRL/HW1_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X21sZmlsZQ==.jl:3
1 =# @test(abs(xs[end] - 3) < 0.01) = Test Passed
#= /home/burger/OCRL/HW1_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X21sZmlsZQ==.jl:3
4 =# @test(maximum(xdot) < 1.0001) = Test Passed
#= /home/burger/OCRL/HW1_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X21sZmlsZQ==.jl:3
5 =# @test(minimum(xdot) > 0.9999) = Test Passed
#= /home/burger/OCRL/HW1_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X21sZmlsZQ==.jl:3
6 =# @test(ys[110] > 0.01) = Test Passed
#= /home/burger/OCRL/HW1_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X21sZmlsZQ==.jl:3
7 =# @test(abs(ys[111]) < 0.01) = Test Passed
#= /home/burger/OCRL/HW1_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X21sZmlsZQ==.jl:3
8 =# @test(abs(ys[112]) < 0.01) = Test Passed
┌ Info: Listening on: 127.0.0.1:8700, thread id: 1
└ @ HTTP.Servers /home/burger/.julia/packages/HTTP/4AUPl/src/Servers.jl:382
┌ Info: MeshCat server started. You can open the visualizer by visiting the following URL in you
r browser:
│ http://127.0.0.1:8700
└ @ MeshCat /home/burger/.julia/packages/MeshCat/9QrxD/src/visualizer.jl:43
```



Part G (5 pts): Solve a QP

Use your QP solver to solve the following optimization problem:

$$\min_{y\in\mathbb{R}^2, a\in\mathbb{R}, b\in\mathbb{R}} \quad \frac{1}{2}y^T\begin{bmatrix}1 & .3\\ .3 & 1\end{bmatrix}y + a^2 + 2b^2 + \begin{bmatrix}-2 & 3.4\end{bmatrix}y + 2a + 4b \tag{33}$$

$$\text{st} \quad a + b = 1 \tag{34}$$

$$\begin{bmatrix}-1 & 2.3\end{bmatrix}y + a - 2b = 3 \tag{35}$$

$$-0.5 \leq y \leq 1 \tag{36}$$

$$-1 \leq a \leq 1 \tag{37}$$

$$-1 \leq b \leq 1 \tag{38}$$

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

In [7]:
```
@testset "part D" begin

    y = randn(2)
```

```julia
    a = randn()
    b = randn()

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

    z = [y[1], y[2], a, b]

    partg_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; -1 0 0 0; 0 1 0 0; 0 -1 0 0; 0 0 1 0; 0 0 -1 0; 0 0 0 1; 0 0 0 -1],
        h = [-0.5, -1, -0.5, -1, -1, -1, -1, -1],
        xi = 1:4,
        μi = 5:6,
        σi = 7:14
    )


    zs, _ , _ = solve_qp(partg_qp, max_iters = 100, tol = 1e-3)


    y = [zs[1],zs[2]]
    a = zs[3]
    b = zs[4]
    @show y, a, b


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

```
iter   |∇Lₓ|      min(h)       |c|       compl     ρ      α
----------------------------------------------------------------
  1   4.44e+00   5.40e-01   2.25e+00   1.50e+00  1e-01  2e-01
  2   2.99e+00   1.12e-01   1.12e+00   1.16e+00  1e-01  5e-01
  3   2.52e+00  -2.86e-03   4.44e-16   5.75e-01  1e-01  1e+00
  4   2.89e-01   1.49e-02   1.11e-16   7.80e-01  1e-01  1e+00
  5   3.57e-03   1.58e-02   0.00e+00   7.99e-01  1e-01  1e+00
  6   1.19e-06   1.58e-02   0.00e+00   8.00e-01  1e-01  1e+00
  7   1.32e+00   6.45e-03   4.44e-16   1.02e-01  1e-02  5e-01
  8   1.68e-01   1.67e-03   4.44e-16   7.94e-02  1e-02  1e+00
  9   2.40e-03   1.79e-03   0.00e+00   8.00e-02  1e-02  1e+00
 10   5.12e-07   1.79e-03   2.22e-16   8.00e-02  1e-02  1e+00
 11   4.82e-01   5.85e-04   0.00e+00   1.04e-02  1e-03  5e-01
 12   2.22e-02   1.80e-04   0.00e+00   7.98e-03  1e-03  1e+00
 13   4.44e-05   1.82e-04   0.00e+00   8.00e-03  1e-03  1e+00
 14   3.83e-01   5.77e-05   2.22e-16   1.05e-03  1e-04  5e-01
 15   1.46e-02   1.81e-05   0.00e+00   7.98e-04  1e-04  1e+00
 16   1.94e-05   1.82e-05   1.11e-16   8.00e-04  1e-04  1e+00
(y, a, b) = ([-0.08090431656798606, 0.8344131759081898], 0.9999817929477258, 1.8207052274104906
e-5)
Test Summary: | Pass  Total  Time
part D        |    3      3  1.1s
Test.DefaultTestSet("part D", Any[], 3, false, false, true, 1.738902798915959e9, 1.7389027999801
1e9, false, "/home/burger/OCRL/HW1_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X23sZml
sZQ==.jl")
```