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

```
  Activating environment at `~/Dropbox/My Mac (MacBook Pro (2))/Desktop/CMU/Optimal Cont
rol/HW1_S23/Project.toml`
┌ Warning: backend `PlotlyBase` is not installed.
└ @ Plots ~/.julia/packages/Plots/io9zQ/src/backends.jl:43
┌ Warning: backend `PlotlyKaleido` is not installed.
└ @ Plots ~/.julia/packages/Plots/io9zQ/src/backends.jl:43
```

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 [2]:
```julia
# 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
    #Build Iρ
    h_ineq_eval = h_ineq(qp, x)
    len_h = length(qp.h)
    Iρ = zeros(eltype(x), len_h, len_h)
    for i in 1:len_h
        currVal = ρ
        if h_ineq_eval[i] < 0 && μ[i] == 0
            currVal = 0
        end
        Iρ[i, i] = currVal
    end
    return Iρ
end
function augmented_lagrangian(qp::NamedTuple, x::Vector, λ::Vector, μ::Vector, ρ::Real):

    c_eq_eval = c_eq(qp, x)
    h_ineq_eval = h_ineq(qp, x)
    L_eval = cost(qp, x) + transpose(λ)*c_eq_eval + transpose(μ)*h_ineq_eval
    Iρ = mask_matrix(qp, x, μ, ρ)

    return L_eval + (ρ/2)*transpose(c_eq_eval)*c_eq_eval + (1/2)*transpose(h_ineq_eval)*
end
function logging(qp::NamedTuple, main_iter::Int, AL_gradient::Vector, x::Vector, λ::Vect
    # TODO: stationarity norm
    L(x_) = cost(qp, x_) + transpose(λ)*c_eq(qp, x_) + transpose(μ)*h_ineq(qp, x_)
    stationarity_norm = norm(FD.gradient(L, x), Inf) # 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 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))

    Z = [x; λ; μ]
    α = 1
    ρ = 1
    ϕ = 5

    #Lagrange Equation
    L(x_) = cost(qp, x_) + transpose(λ)*c_eq(qp, x_) + transpose(μ)*h_ineq(qp, x_)

    #Constraint function
    constraint(x_) = vcat(reshape(c_eq(qp, x_), length(λ), 1), reshape(h_ineq(qp, x_), l
```

```julia
    #Constraint Jacobians
    c_jac(x_) = FD.jacobian(z -> c_eq(qp, z), x_)
    h_jac(x_) = FD.jacobian(z -> h_ineq(qp, z), x_)

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

    # TODO:
    for main_iter = 1:max_iters

        ##### Newton's method ######
        ∇f(X) = reshape(FD.gradient(x_ -> cost(qp, x_), X), 1, length(qp.q))
        Iρ = mask_matrix(qp, x, μ, ρ)
        ∇AL(x_) = vec(∇f(x_) + λ'*c_jac(x_) + ρ*transpose(c_eq(qp, x_))*c_jac(x_) + μ'*h

        if verbose
            logging(qp, main_iter, ∇AL(x), x, λ, μ, ρ)
        end

        #regularize with β = 1e-4
        β = 1e-4
        ∇2AL = FD.jacobian(∇AL, x) + β*I

        Δx = -∇2AL\∇AL(x)
        x = x + α*Δx

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

        #### Update Penalty ########
        ρ = ρ*ϕ

        # TODO: convergence criteria based on tol
        if norm(∇AL(x), Inf) < tol
            return x, λ, μ
        end

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

```
iter   |∇Lₓ|      |∇ALₓ|     max(h)     |c|        compl     ρ
----------------------------------------------------------------
  1   1.59e+01   5.60e+01   4.38e+00   6.49e+00   0.00e+00   1e+00
  2   3.61e+00   3.95e+01   1.55e+00   1.31e+00   2.64e+00   5e+00
  3   2.29e+00   3.53e+01   3.54e-02   4.25e-01   1.24e-01   2e+01
  4   3.32e-01   8.76e+00   1.69e-02   1.76e-02   1.95e-02   1e+02
  5   1.40e+01   1.63e+02   6.95e-02   1.19e-03   6.03e-01   6e+02
  6   2.44e-06   1.35e+02   3.61e-05   6.33e-05   5.66e-04   3e+03
  7   4.00e-07   1.30e-01  -1.24e-06   2.18e-06   1.40e-06   2e+04
  8   8.85e-11   7.06e-05  -1.40e-10   3.16e-10   1.54e-10   8e+04
```

Out[2]: ([-0.3262308057133945, 0.24943797997175585, -0.4322676644052281, -1.417224697124201, -1.3994527400875791, 0.6099582408523453, -0.07312202122168011, 1.303147752200024, 0.5389034791065955, -0.7225813651685231], [-0.12835195124116128, -2.8376241671707936, -0.8320804499224224], [0.03635294264803246, 0.0, 0.0, 1.059444495082744, 0.0])

QP Solver test (10 pts)

In [3]:
```julia
# 10 points
using Test
@testset "qp solver" begin
    @load joinpath(@__DIR__, "qp_data.jld2") qp
    x, λ, μ = solve_qp(qp; verbose = true, max_iters = 100, tol = 1e-6)

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

```
iter   |∇Lₓ|      |∇ALₓ|      max(h)      |c|        compl      ρ
----------------------------------------------------------------
  1   1.59e+01   5.60e+01   4.38e+00   6.49e+00   0.00e+00   1e+00
  2   3.61e+00   3.95e+01   1.55e+00   1.31e+00   2.64e+00   5e+00
  3   2.29e+00   3.53e+01   3.54e-02   4.25e-01   1.24e-01   2e+01
  4   3.32e-01   8.76e+00   1.69e-02   1.76e-02   1.95e-02   1e+02
  5   1.40e+01   1.63e+02   6.95e-02   1.19e-03   6.03e-01   6e+02
  6   2.44e-06   1.35e+02   3.61e-05   6.33e-05   5.66e-04   3e+03
  7   4.00e-07   1.30e-01  -1.24e-06   2.18e-06   1.40e-06   2e+04
  8   8.85e-11   7.06e-05  -1.40e-10   3.16e-10   1.54e-10   8e+04
Test Summary: | Pass  Total
qp solver     |    3      3
```

Out[3]: Test.DefaultTestSet("qp solver", Any[], 3, false, false)

Simulating a Falling Brick with QPs

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

The Dynamics

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

$$M\dot{v} + Mg = J^T\lambda$$

$$\text{where } M = mI_{2\times 2},\ g = \begin{bmatrix} 0 \\ 9.81 \end{bmatrix},\ J = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

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.

From the optimization problem above, we have the following KKT conditions:

$$\begin{aligned} v_{k+1}^T M + [M(\Delta t \cdot g - v_k)]^T && \text{(Stationarity)} && (14) \\ (-J\Delta t \cdot v_{k+1} - Jq_k) \leq 0 && \text{(Primal Feasibility)} && (15) \\ \mu \geq 0 && \text{(Dual Feasibility)} && (16) \\ \mu \circ (-J\Delta t \cdot v_{k+1} - Jq_k) = 0 && \text{(Complementarity)} && (17) \end{aligned}$$

Looking at the primal feasibility case, in order for the constraint to be valid i.e. $Gx - h \leq 0$ the following must hold true: $Jq_k \geq 0$. Thus the primal feasibility condition covers the not falling through the ice.

A similar thing occurs with the complementarity case where μ or in the case of the dynamics λ_{k+1} is multiplied by Jq_k which must be equal to zero. Thus the complementarity case covers the no force at a distance constraint.

Finally, since λ is equivalent to μ in the case of the KKT conditions, the dual feasibility covers the normal forces only pushing constraint.

Brick Simulation (5 pts)

In [4]:
```julia
function brick_simulation_qp(q, v; mass = 1, 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; 0.0 mass]
    J = [0 1.0]
    g = [0; 9.81]

    qp = (
        Q = 1*M,
        q = M*Δt*g - M*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[4]:
```
brick_simulation_qp (generic function with 1 method)
```

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

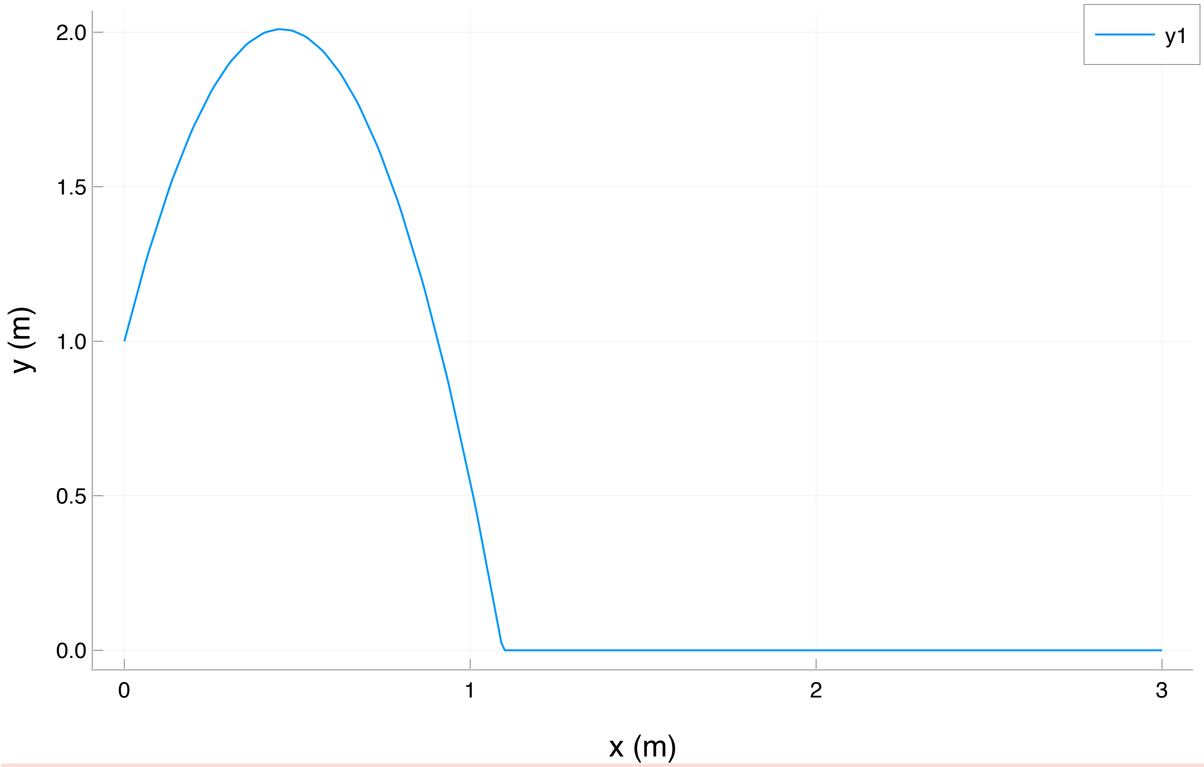
```
Test Summary: | Pass  Total
brick qp      |   10     10
```

Out[5]:
```
Test.DefaultTestSet("brick qp", Any[], 10, false, false)
```

In [6]:
```julia
include(joinpath(@__DIR__, "animate_brick.jl"))

let
```

```julia
    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 in 1:N-1
        qp = brick_simulation_qp(qs[i], vs[i], Δt = dt)
        v, λ, μ = solve_qp(qp, verbose = false)
        vs[i+1] = v
        qs[i+1] = qs[i] + dt*v
    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[6]:31 =# @test(abs(maximum(ys) - 2) < 0.1) = Test Passed
#= In[6]:32 =# @test(minimum(ys) > -0.01) = Test Passed
#= In[6]:33 =# @test(abs(xs[end] - 3) < 0.01) = Test Passed
#= In[6]:36 =# @test(maximum(xdot) < 1.0001) = Test Passed
#= In[6]:37 =# @test(minimum(xdot) > 0.9999) = Test Passed
#= In[6]:38 =# @test(ys[110] > 0.01) = Test Passed
#= In[6]:39 =# @test(abs(ys[111]) < 0.01) = Test Passed
#= In[6]:40 =# @test(abs(ys[112]) < 0.01) = Test Passed
```




```
┌ Info: MeshCat server started. You can open the visualizer by visiting the following UR
L in your browser:
└ http://127.0.0.1:8702
```

Out[6]: Open Controls

In [ ]: