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

```
[ Info: Precompiling IJuliaExt [2f4121a4-3b3a-5ce6-9c5e-1f2673ce168a]
[ Info: Precompiling ForwardDiff [f6369f11-7733-5829-9624-2563aa707210]
[ Info: Precompiling SpecialFunctionsExt [997ecda8-951a-5f50-90ea-61382e9770
4b]
[ Info: Precompiling MeshCat [283c5d60-a78f-5afe-a0af-af636b173e11]
[ Info: Precompiling ForwardDiffStaticArraysExt [b74fd6d0-9da7-541f-a07d-1b6
af30a262f]
[ Info: Precompiling GeometryBasicsExt [b238bd29-021f-5edc-8b0e-16b9cda5f63
a]
```

Julia Warmup

Just like Python, Julia lets you do the following:

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

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

    @show y
    @show x
end
```

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

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

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

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

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

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

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

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

Optional function arguments

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

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

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

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

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

Out[ ]:
```
(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 [ ]:
```julia
# 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, θ̇1, θ2, θ̇2 = 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 = [
        θ̇1
        (
            m2 * g * sin(θ2) * c - m2 * s * (L1 * c * θ̇1^2 + L2 * θ̇2^2) -
            (m1 + m2) * g * sin(θ1)
        ) / (L1 * (m1 + m2 * s^2))
        θ̇2
        (
            (m1 + m2) * (L1 * θ̇1^2 * s - g * sin(θ2) + g * sin(θ1) * c) +
            m2 * L2 * θ̇2^2 * 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


    # the state is the following:
    θ1, θ̇1, θ2, θ̇2 = 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 * θ̇1 * cos(θ1), 0, L1 * θ̇1 * sin(θ1)]
    v2 = v1 + [L2 * θ̇2 * cos(θ2), 0, L2 * θ̇2 * sin(θ2)]

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

Out[ ]: 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:

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

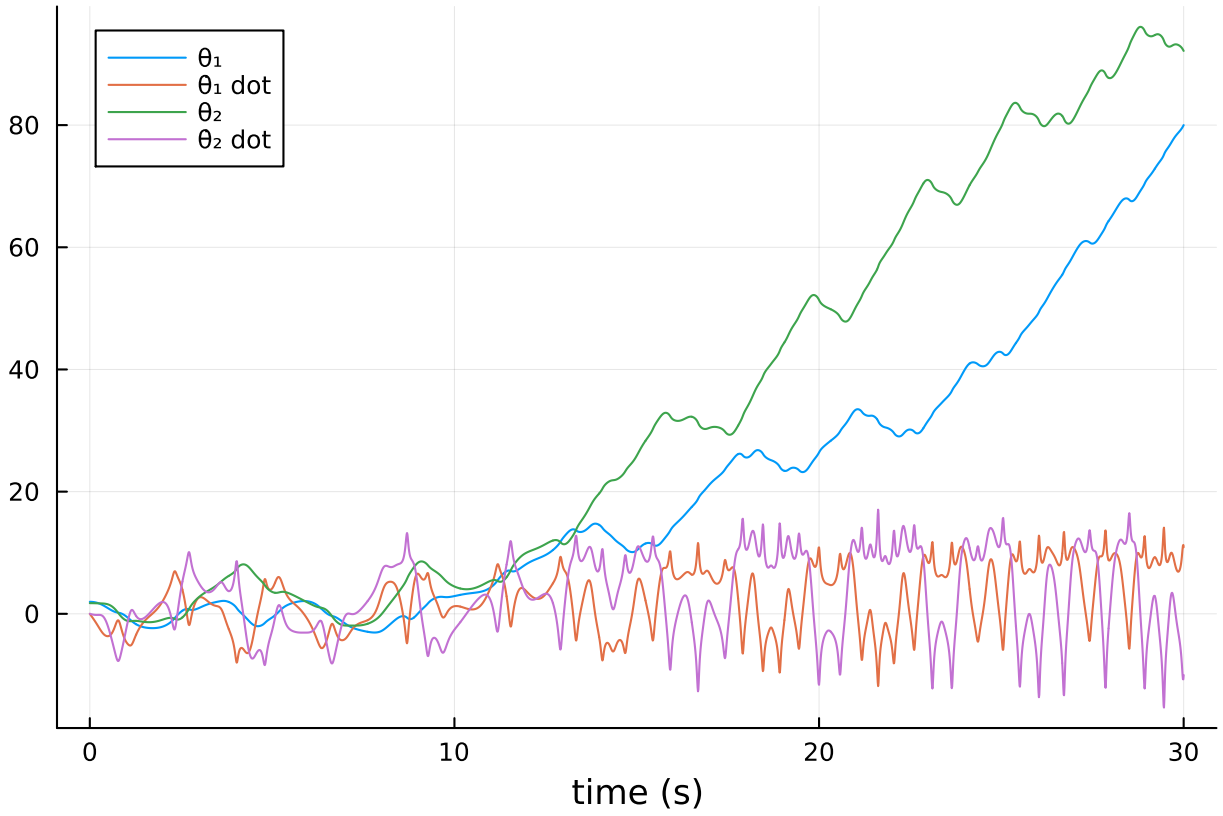
In [ ]:
```
"""
    x_{k+1} = forward_euler(params, dynamics, x_k, dt)

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

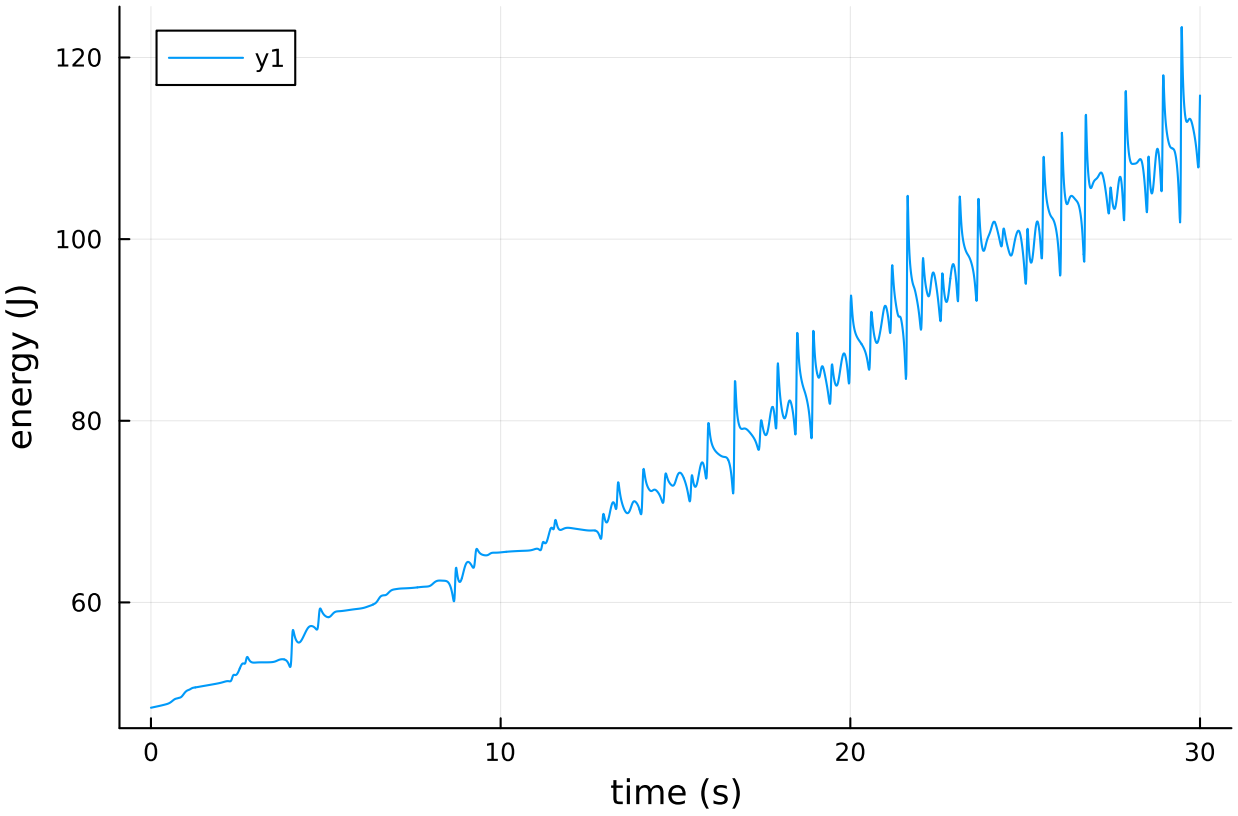
Out[ ]: forward_euler

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

let

    # parameters for the simulation
    params = (m1 = 1.0, m2 = 1.0, L1 = 1.0, L2 = 1.0, g = 9.8)

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

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

```
    N = length(t_vec)

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

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

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

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

    # plot state history, energy history, and animate it
    display(
        plot(
            t_vec,
            hcat(X...)',
            xlabel = "time (s)",
            label = ["θ₁" "θ̇₁ dot" "θ₂" "θ̇₂ dot"],
        ),
    )
    display(plot(t_vec, E, xlabel = "time (s)", ylabel = "energy (J)"))
    meshcat_animate(params, X, dt, N)


end
```

```
#= In[8]:36 =# @test(norm(X[end]) > 1.0e-10) = Test Passed
#= In[8]:37 =# @test(2 < E[end] / E[1] < 3) = Test Passed
```






```
[ Info: Listening on: 127.0.0.1:8700, thread id: 1
┌ Info: MeshCat server started. You can open the visualizer by visiting the
following URL in your browser:
└ http://127.0.0.1:8700
```

Out[ ]:

Now let's implement the next two integrators:

Midpoint:

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

RK4:

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

In [ ]:
```julia
function midpoint(params::NamedTuple, dynamics::Function, x::Vector, dt::Rea
    # TODO: implement explicit midpoint
    x_m = x + 0.5*dt*dynamics(params,x)
    return x + dt*dynamics(params,x_m)
```

```
end
function rk4(params::NamedTuple, dynamics::Function, x::Vector, dt::Real)::V
    # TODO: implement RK4
    k1 = dynamics(params,x) * dt
    k2 = dynamics(params,x + 0.5*k1) * dt
    k3 = dynamics(params,x + 0.5*k2) * dt
    k4 = dynamics(params,x + k3) * dt
    return x + (k1 + 2*k2 + 2*k3 + k4)/6
end
```

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

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


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

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

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

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

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

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

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

```
WARNING: both Plots and LinearAlgebra export "rotate!"; uses of it in module
Main must be qualified
```

Part B (10 pts): Implicit Integrators

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

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

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

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

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

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

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

Implicit Midpoint (2nd order)

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

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

Hermite Simpson (3rd order)

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

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

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 [ ]:
```julia
# since these are explicit integrators, these function will return the resid
# NOTE: we are NOT solving anything here, simply return the residuals
function backward_euler(
    params::NamedTuple,
    dynamics::Function,
    x1::Vector,
    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
    return x1 + dt * dynamics(params, 0.5 * (x1 + x2)) - x2
end
function hermite_simpson(
    params::NamedTuple,
    dynamics::Function,
    x1::Vector,
    x2::Vector,
    dt::Real,
)::Vector
    x_mid =
        0.5 * (x1 + x2) +
        0.125 * dt * (dynamics(params, x1) - dynamics(params, x2))
    return x1 +
           1 / 6 *
           dt *
           (
               dynamics(params, x1) +
               4 * dynamics(params, x_mid) +
               dynamics(params, x2)
           ) - x2
end
```

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

In [ ]:
```
# TODO
# this function takes in a dynamics function, implicit integrator function,
# and uses Newton's method to solve for an x2 that satsifies the implicit ir
# that we wrote about in the functions above
function implicit_integrator_solve(
    params::NamedTuple,
    dynamics::Function,
    implicit_integrator::Function,
    x1::Vector,
    dt::Real;
    tol = 1e-13,
    max_iters = 10,
)::Vector

    # initialize guess
    x2 = 1 * x1

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

```
            x2,
        )
        x2 = x2 - inv(J) * (implicit_integrator(params, dynamics, x1, x2, dt

        # TODO: return x2 when the norm of the residual is below tol
        if norm(implicit_integrator(params, dynamics, x1, x2, dt)) < tol
            return x2
        end


    end
    error("implicit integrator solve failed")
end
```

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

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

    dt = 1e-1
    x1 = [0.1, 0.2, 0.3, 0.4]

    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  1.6s
```

Out[ ]: Test.DefaultTestSet("implicit integrator check", Any[], 3, false, false, tr
ue, 1.706280499859626e9, 1.706280501501644e9, false)

In [ ]:
```
function simulate_implicit(
    params::NamedTuple,
    dynamics::Function,
    implicit_integrator::Function,
    x0::Vector,
    dt::Real,
    tf::Real;
    tol = 1e-13,
)
```

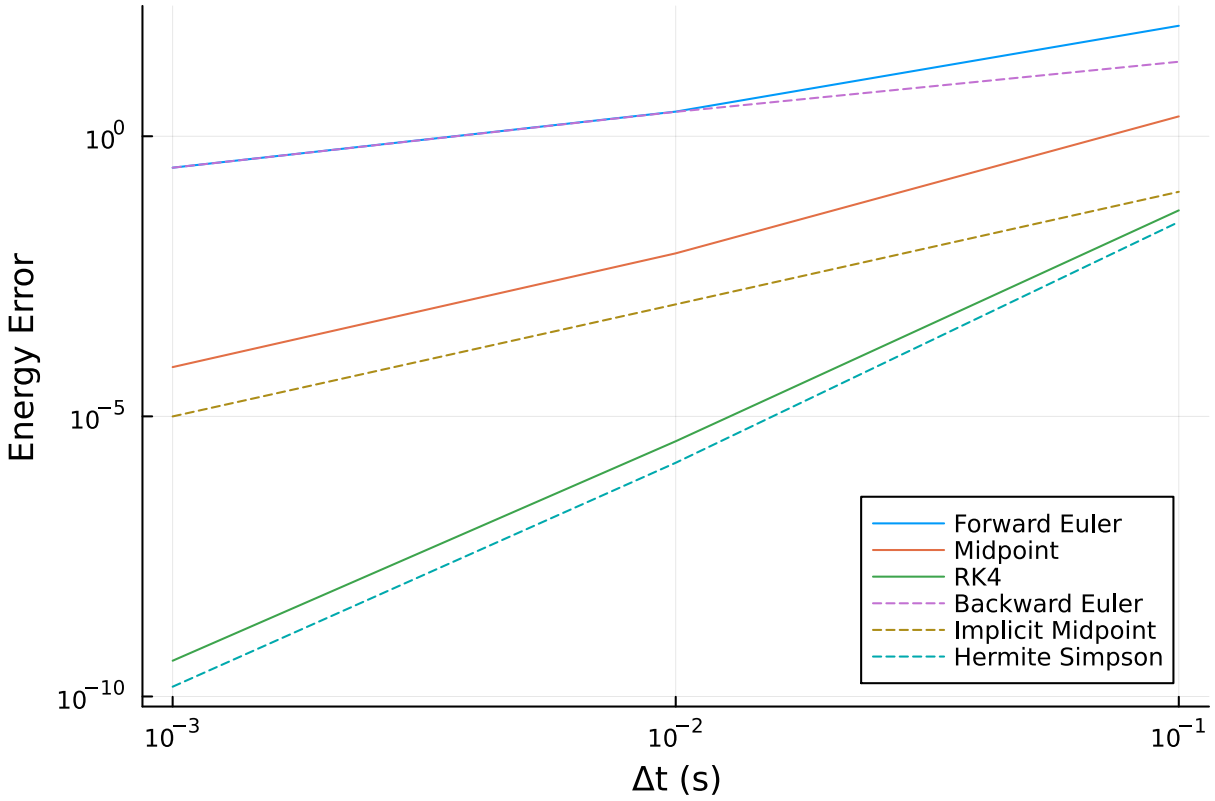
```
    t_vec = 0:dt:tf
    N = length(t_vec)
    X = [zeros(length(x0)) for i = 1:N]
    X[1] = x0

    # TODO: do a forward simulation with the selected implicit integrator
    # hint: use your `implicit_integrator_solve` function
    for k = 1:N-1
        X[k+1] = implicit_integrator_solve(
            params,
            dynamics,
            implicit_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
```

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

In [ ]:
```
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_simps

    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,
        xlabel = "Δt (s)",
        ylabel = "Energy Error",
    )
    plot!(
        dts,
        hcat(implicit_data...),
        ls = :dash,
        label = ["Backward Euler" "Implicit Midpoint" "Hermite Simpson"],
    )
    plot!(legend = :bottomright)
end
```

Out[ ]:

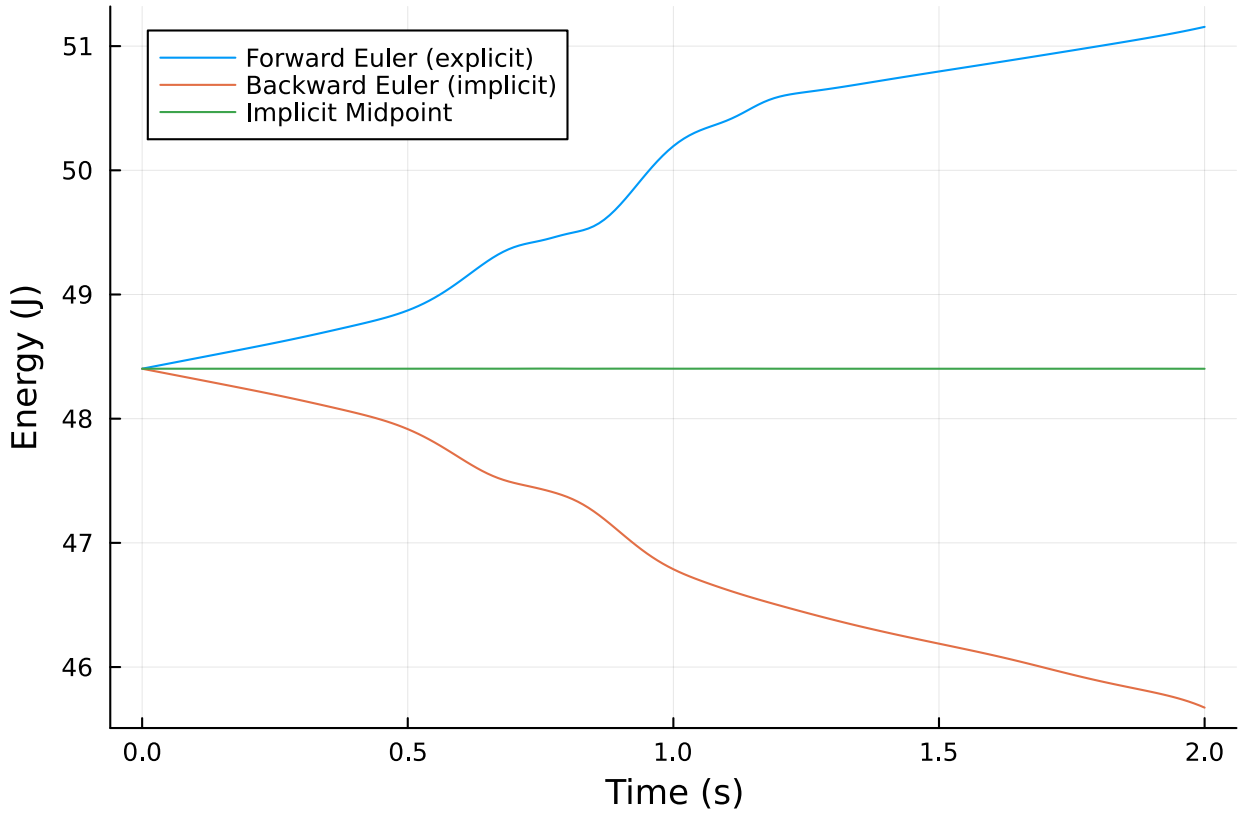


What we can see above is the maximum energy error for each of the integration methods. In general, the implicit methods of the same order are slightly better than the explicit ones.

In [ ]:

```
@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

    # 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
```

Out[ ]:
```
Test.DefaultTestSet("energy behavior", Any[], 6, false, false, true, 1.7062
80537683998e9, 1.706280537763158e9, false)
```

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

Part C (5 pts): One sentence short answer

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

Put ONE SENTENCE answer here

- Forward Euler: Energy grows unbounded, which implies that it is unstable.
- Backward Euler: Energy decreases unbounded. The system is artificially damped.
- Implicit Midpoint: Energy is bounded and stays close to the initial energy, which implies that it is stable.

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

Q2: Equality Constrained Optimization (25 pts)

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

Part A (10 pts)

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

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

Which has the following Lagrangian:

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

and the following KKT conditions for optimality:

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

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

Newton's Method with a Linesearch

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

do the following:

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

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

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

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

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

In [ ]:
```
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)

    alpha = 2.0

    # NOTE: DO NOT USE A WHILE LOOP
    for i = 1:max_ls_iters
        alpha = alpha / 2.0

        # TODO: return α when merit_fx(z + α*Δz) < merit_fx(z)

        if merit_fx(z + alpha * Δz) < merit_fx(z)
            return alpha
        end

    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 ite

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

    # NOTE: DO NOT USE A WHILE LOOP ANYWHERE

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


    for i = 1:(max_iters-1)

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

        # TODO: evaluate current residual
        r = res_fx(Z[i])

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

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

        # TODO: caculate Newton step (don't forget the negative sign)
        Δz = -res_jac_fx(Z[i]) \ r

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

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

    end
```

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

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

In [ ]:
```
@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  0.3s
```

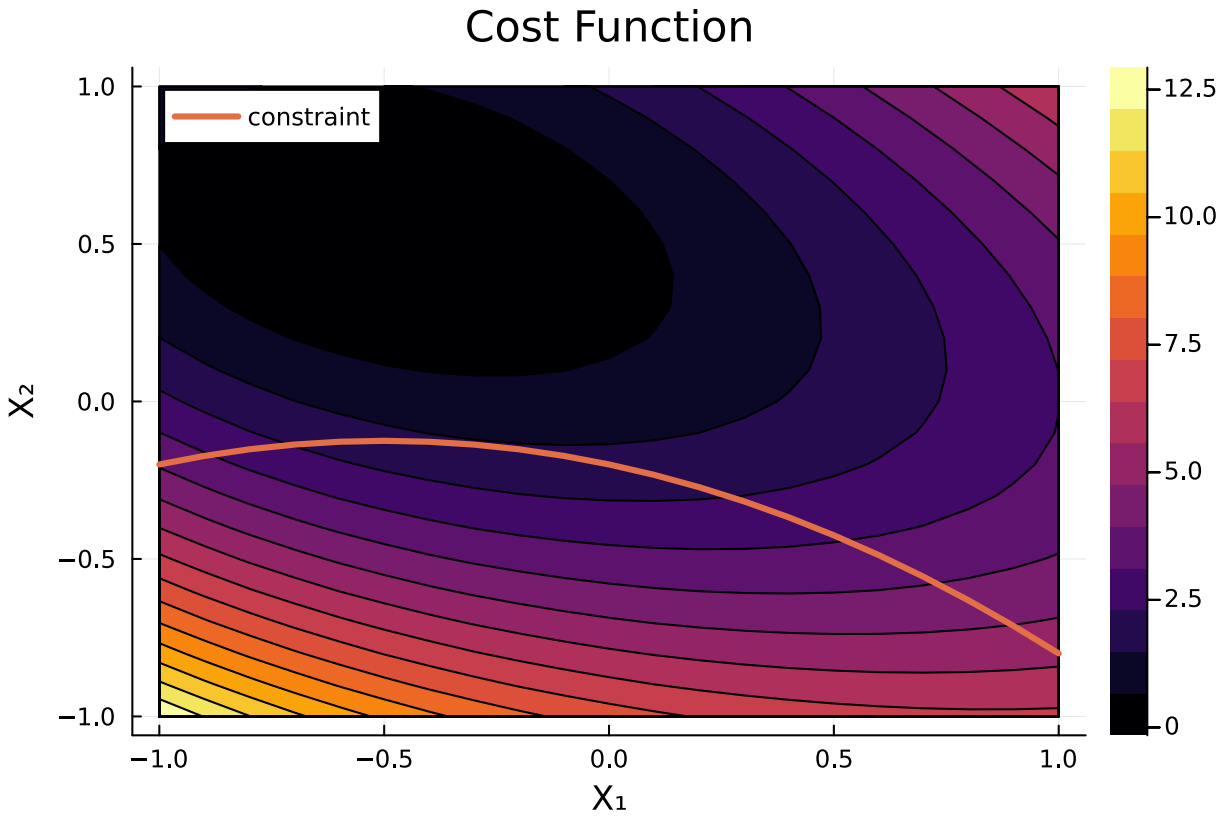
Out[ ]:
```
Test.DefaultTestSet("check Newton", Any[], 2, false, false, true, 1.7062833
59469679e9, 1.706283359760339e9, false)
```

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 [ ]:
```
let
    Q = [1.65539 2.89376; 2.89376 6.51521]
    q = [2; -3]
    cost(x) = 0.5 * x' * Q * x + q' * x + exp(-1.3 * x[1] + 0.3 * x[2]^2) #
    contour(
```

```
        -1:0.1:1,
        -1:0.1:1,
        (x1, x2) -> cost([x1; x2]),
        title = "Cost Function",
        xlabel = "X₁",
        ylabel = "X₂",
        fill = true,
    )
    plot!(
        -1:0.1:1,
        -0.3 * (-1:0.1:1) .^ 2 - 0.3 * (-1:0.1:1) .- 0.2,
        lw = 3,
        label = "constraint",
    )
end
```

Out[ ]:



In [ ]:

```
# we will use Newton's method to solve the constrained optimization problem
function cost(x::Vector)
    Q = [1.65539 2.89376; 2.89376 6.51521]
    q = [2; -3]
    return 0.5 * x' * Q * x + q' * x + exp(-1.3 * x[1] + 0.3 * x[2]^2)
end
function constraint(x::Vector)
    norm(x) - 0.5
end
# HINT: use this if you want to, but you don't have to
function constraint_jacobian(x::Vector)::Matrix
    # since `constraint` returns a scalar value, ForwardDiff
    # will only allow us to compute a gradient of this function
```

```
    # (instead of a Jacobian). This means we have two options for
    # computing the Jacobian: Option 1 is to just reshape the gradient
    # into a row vector

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

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

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

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

    x = z[1:2]
    λ = z[3:3]

    # TODO: return the stationarity condition for the cost function
    # and the primal feasibility
    cost_gradient = FD.gradient(cost, x) # (2,1)
    constrain_j = constraint_jacobian(x) # (1,2)
    station_condition = cost_gradient + constrain_j' * λ
    primal_feasibility = constraint(x)

    residue = [station_condition; primal_feasibility]

    return residue # (3, 1)
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]

    # TODO: return full Newton jacobian with a 1e-3 regularizer
    Lx(_x) = cost(_x) + λ * constraint(_x)
    Lxx = FD.hessian(Lx, x)
    Lxlam = constraint_jacobian(x)

    reg = 1e-3 * I(3)
    reg[3, 3] = -1e-3 # NOTE: lambda's eign value is negative
    kkt_jac = [Lxx Lxlam'; Lxlam zeros(1, 1)] + reg

    return kkt_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]

    # TODO: return Gauss-Newton jacobian with a 1e-3 regularizer
    Lx(_x) = cost(_x)
    Lxx = FD.hessian(Lx, x)
    Lxlam = constraint_jacobian(x)

    reg = 1e-3 * I(3)
    reg[3, 3] = -1e-3 # NOTE: lambda's eign value is negative
    kkt_jac = [Lxx Lxlam'; Lxlam zeros(1, 1)] + reg

    return kkt_jac
end
```

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

In [ ]:
```
@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
```

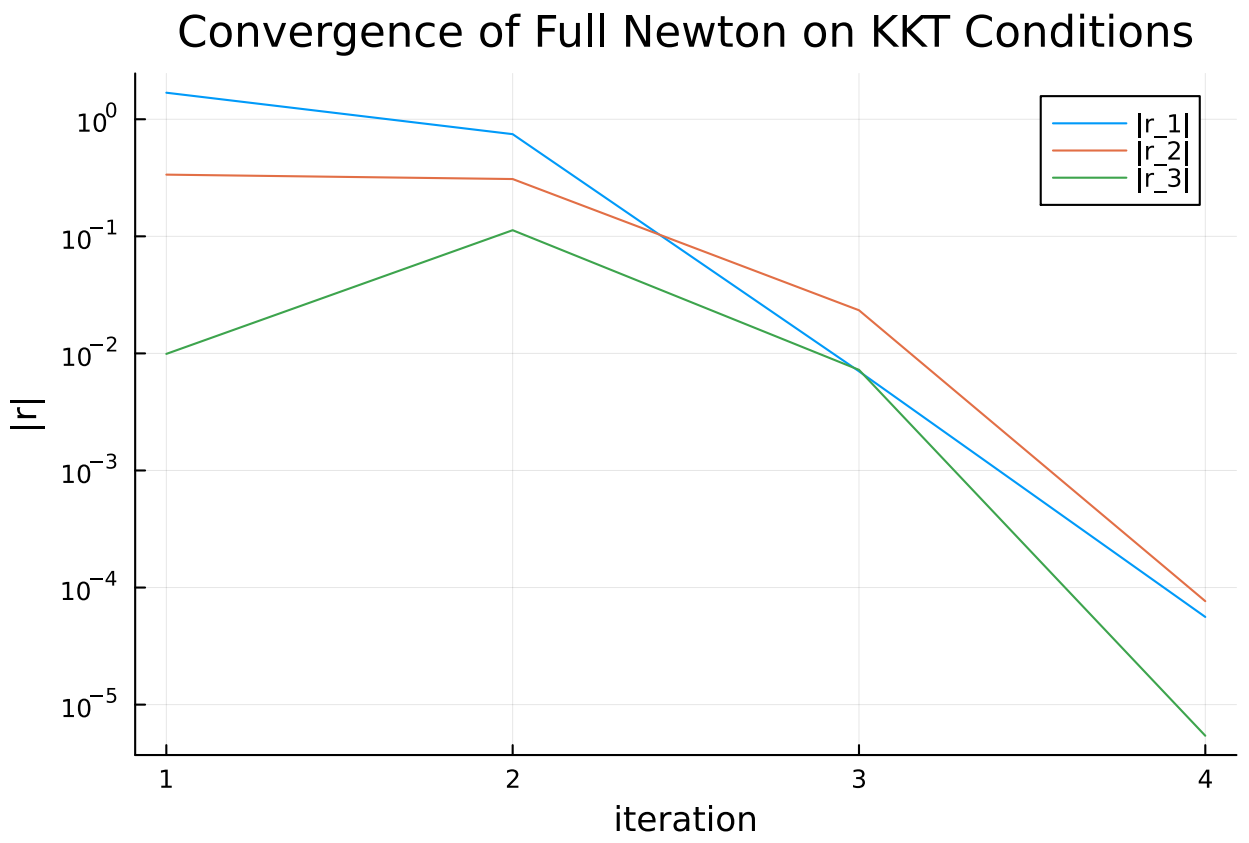
```
Test Summary:  | Pass  Total  Time
Test Jacobians |    5      5  1.9s
```

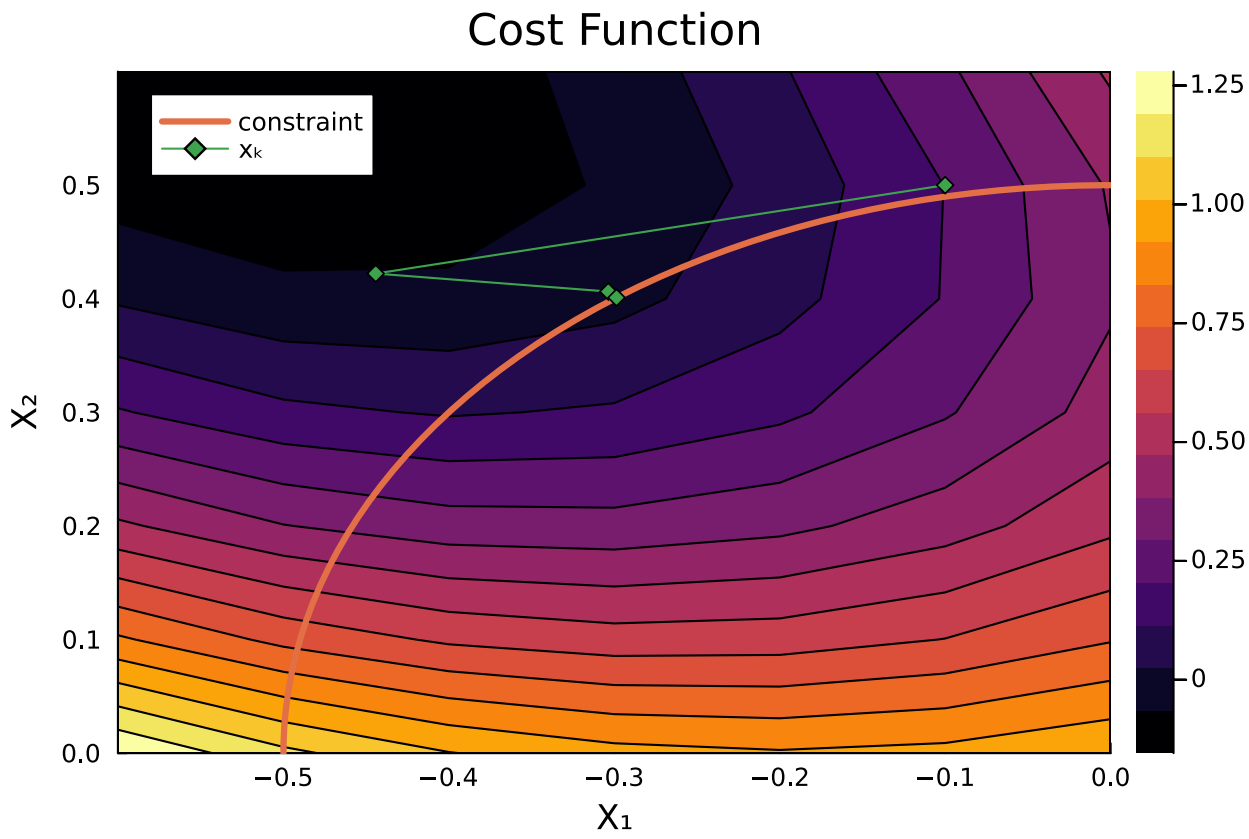
Out[ ]:
```
Test.DefaultTestSet("Test Jacobians", Any[], 5, false, false, true, 1.70629
1878488978e9, 1.706291880427015e9, false)
```

In [ ]:
```
@testset "Full Newton" begin


    z0 = [-0.1, 0.5, 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])

    plot(
        Rp[1],
        yaxis = :log,
        ylabel = "|r|",
        xlabel = "iteration",
        yticks = [1.0 * 10.0^(-x) for x in 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(
        -0.6:0.1:0,
        0:0.1:0.6,
        (x1, x2) -> cost([x1; x2]),
        title = "Cost Function",
        xlabel = "X₁",
        ylabel = "X₂",
        fill = true,
    )
    xcirc = [0.5 * cos(θ) for θ in range(0, 2 * pi, length = 200)]
    ycirc = [0.5 * sin(θ) for θ in range(0, 2 * pi, length = 200)]
    plot!(
        xcirc,
        ycirc,
        lw = 3.0,
        xlim = (-0.6, 0),
        ylim = (0, 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|: 0.815049596220325   α: 1.0
iter: 3    |r|: 0.025448943695826724   α: 1.0
iter: 4    |r|: 9.501514353541471e-5
```






```
Test Summary: | Pass  Total  Time
Full Newton   |    2      2  1.7s
```

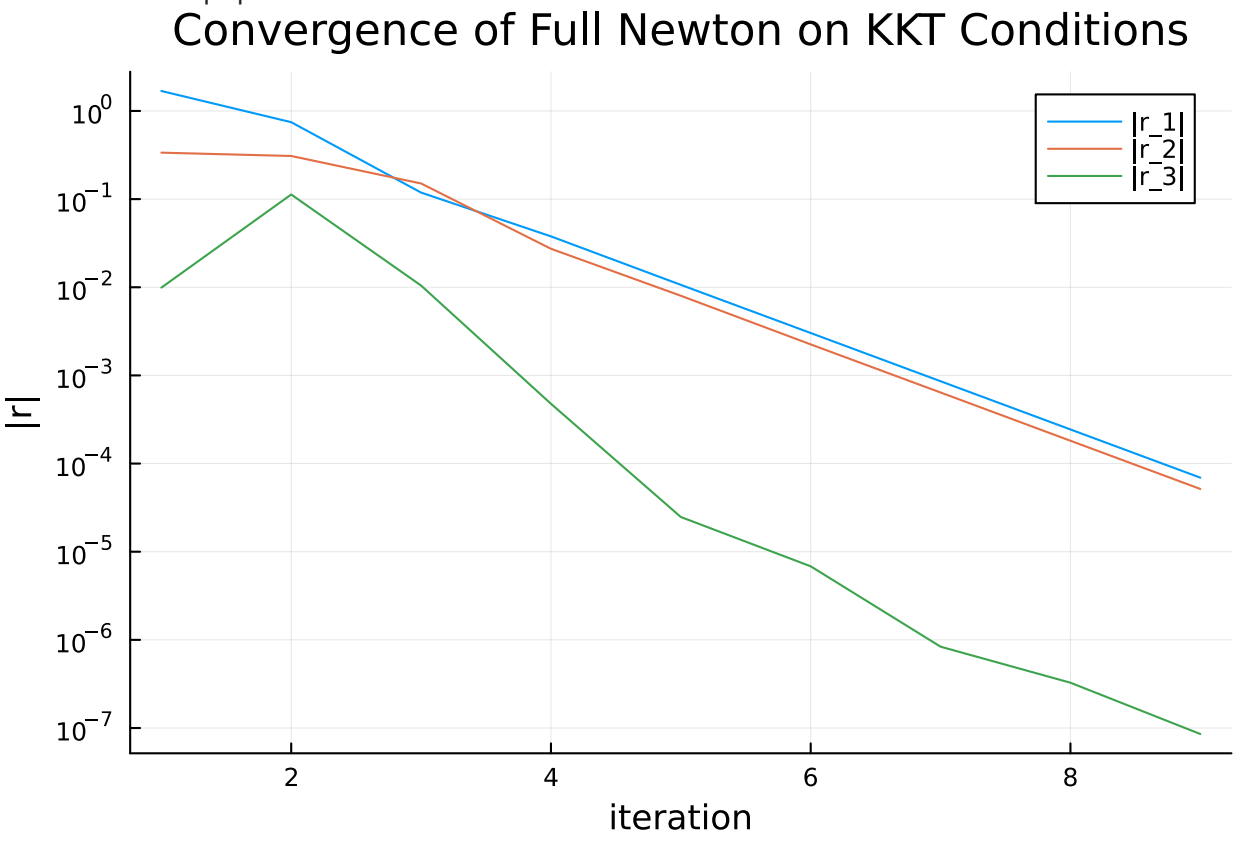
Out[ ]: Test.DefaultTestSet("Full Newton", Any[], 2, false, false, true, 1.706291888375631e9, 1.706291890070227e9, false)

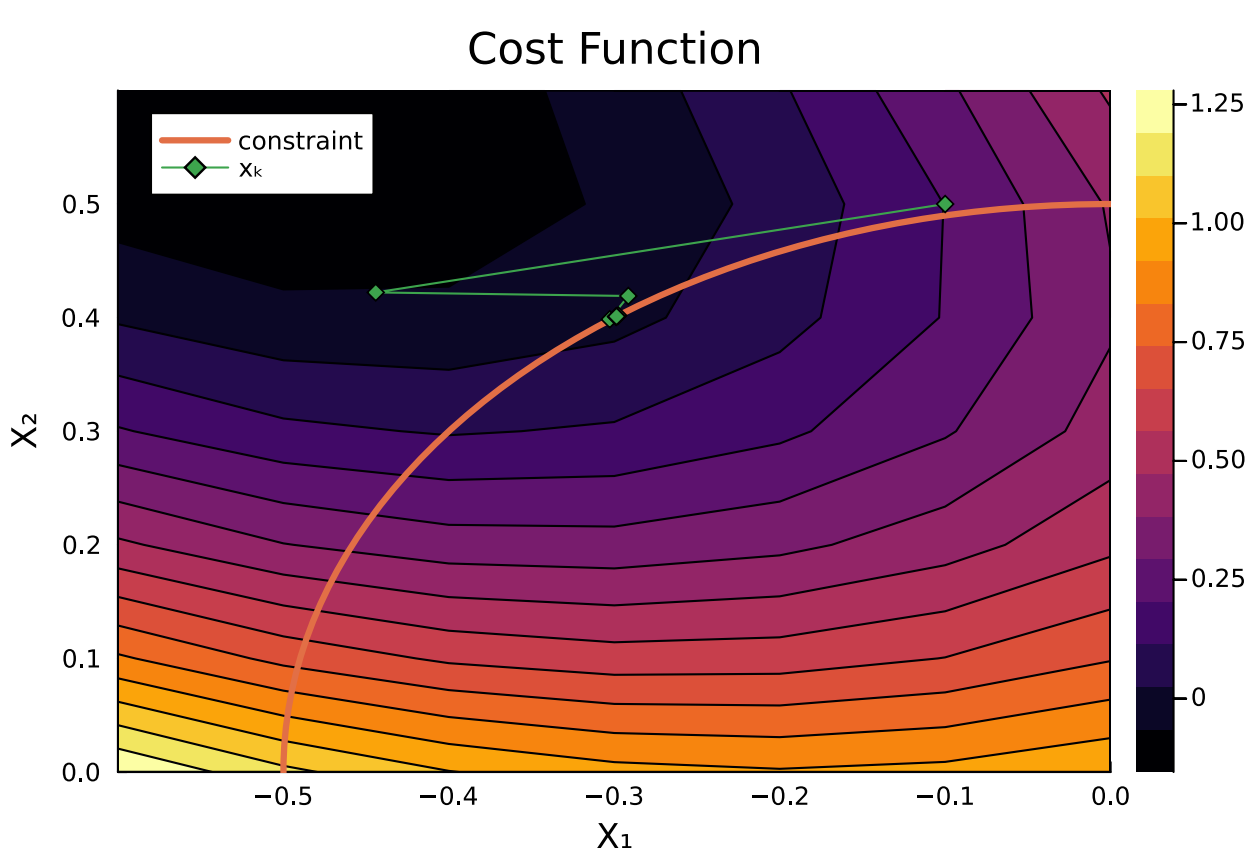
In [ ]:

```
@testset "Gauss-Newton" begin


    z0 = [-0.1, 0.5, 0] # initial guess
    merit_fx(_z) = norm(kkt_conditions(_z)) # simple merit function

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

    plot(
        Rp[1],
        yaxis = :log,
        ylabel = "|r|",
        xlabel = "iteration",
        yticks = [1.0 * 10.0^(-x) for x in 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(
        -0.6:0.1:0,
        0:0.1:0.6,
        (x1, x2) -> cost([x1; x2]),
        title = "Cost Function",
        xlabel = "X₁",
        ylabel = "X₂",
        fill = true,
    )
    xcirc = [0.5 * cos(θ) for θ in range(0, 2 * pi, length = 200)]
    ycirc = [0.5 * sin(θ) for θ in range(0, 2 * pi, length = 200)]
    plot!(
        xcirc,
        ycirc,
        lw = 3.0,
```

```
        xlim = (-0.6, 0),
        ylim = (0, 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|: 0.815049596220325   α: 1.0
iter: 3    |r|: 0.19186516708148585   α: 1.0
iter: 4    |r|: 0.04663490553083133   α: 1.0
iter: 5    |r|: 0.013329778429546028   α: 1.0
iter: 6    |r|: 0.0037714013578573355   α: 1.0
iter: 7    |r|: 0.001071165054782875   α: 1.0
iter: 8    |r|: 0.00030392210707413806   α: 1.0
iter: 9    |r|: 8.625764141582568e-5
```






```
Test Summary: | Pass  Total  Time
Gauss-Newton  |    2      2  0.2s
```

Out[ ]:
```
Test.DefaultTestSet("Gauss-Newton", Any[], 2, false, false, true, 1.706291898151176e9, 1.706291898333151e9, false)
```

Part B (10 pts): Balance a quadruped

Now we are going to solve for the control input $u \in \mathbb{R}^{12}$, and state $x \in \mathbb{R}^{30}$, such that the quadruped is balancing up on one leg. First, let's load in a model and display the rough "guess" configuration that we are going for:

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

# --------these three are global variables------------
model = UnitreeA1()
mvis = initialize_visualizer(model)
const x_guess = initial_state(model)
# ----------------------------------------------------

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

```
[ Info: Listening on: 127.0.0.1:8702, thread id: 1
┌ Info: MeshCat server started. You can open the visualizer by visiting the
following URL in your browser:
└ http://127.0.0.1:8702
WARNING: redefinition of constant x_guess. This may fail, cause incorrect an
swers, or produce other errors.
```

Out[ ]:

Now, we are going to solve for the state and control that get us a statically stable stance on just one leg. We are going to do this by solving the following optimization problem:

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

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

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

In this next section, you should fill out `quadruped_kkt(z)` with the KKT conditions for

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

In [ ]:
```
# 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
    x_error = x - x_guess[idx_x]

    return 0.5 * x_error' * x_error + 0.5 * 1e-3 * u' * u
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 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
    cost_gradient = FD.gradient(quadruped_cost, y)
    constrain_jacobian = FD.jacobian(quadruped_constraint, y)
    station_condition = cost_gradient + constrain_jacobian' * λ
    primal_feasibility = quadruped_constraint(y)

    return [station_condition; primal_feasibility]
end
```

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

    y = [x;u]

    # TODO: return Gauss-Newton Jacobian with a regularizer (try 1e-3,1e-4,1
    # and use whatever regularizer works for you
    Ly(_y) = quadruped_cost(_y)
    Lyy = FD.hessian(Ly, y)
    Lylam = FD.jacobian(quadruped_constraint, y)

    reg = diagm(0 => [ones(x_len+u_len); -ones(lam_len)]) * 1e-3

    # @show size(Lyy)
    # @show size(Lylam)
    # @show lam_len
    # @show size(reg)

    kkt_jac = [Lyy Lylam'; Lylam zeros(lam_len,lam_len)] + reg

    return kkt_jac
end
```

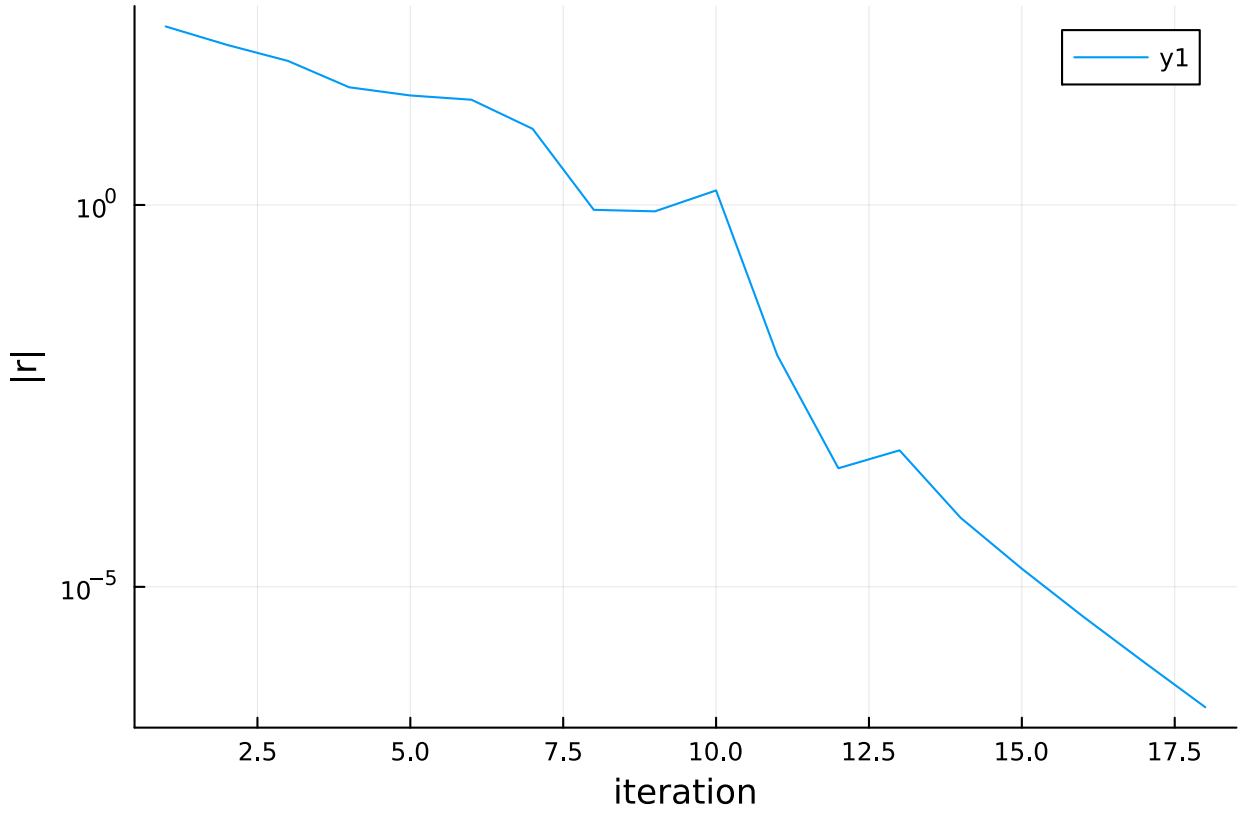
```
WARNING: redefinition of constant x_guess. This may fail, cause incorrect an
swers, or produce other errors.
```

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

In [ ]:
```
function quadruped_merit(z)
    # merit function for the quadruped problem
    @assert length(z) == 72
    r = quadruped_kkt(z)
    return norm(r[1:42]) + 1e4*norm(r[43:end])
end

@testset "quadruped standing" begin

    z0 = [x_guess; zeros(12); zeros(30)]
    Z = newtons_method(z0, quadruped_kkt, quadruped_kkt_jac, quadruped_merit
    set_configuration!(mvis, Z[end][1:state_dim(model)÷2])
    R = norm.(quadruped_kkt.(Z))

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

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

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

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

end
```

```
iter: 1    |r|: 217.37236872332247   α: 1.0
iter: 2    |r|: 124.92133581598108   α: 1.0
iter: 3    |r|: 76.87596686967947   α: 0.5
iter: 4    |r|: 34.75020218490922   α: 0.25
iter: 5    |r|: 27.139783671701174   α: 0.5
iter: 6    |r|: 23.87618772970579   α: 1.0
iter: 7    |r|: 9.928511516364996   α: 1.0
iter: 8    |r|: 0.8635831086148376   α: 1.0
iter: 9    |r|: 0.8252015646602422   α: 1.0
iter: 10    |r|: 1.549464041851805   α: 1.0
iter: 11    |r|: 0.010794824533036831   α: 1.0
iter: 12    |r|: 0.0003569664754826479   α: 1.0
iter: 13    |r|: 0.0006131222647310681   α: 1.0
iter: 14    |r|: 8.012756305099094e-5   α: 1.0
iter: 15    |r|: 1.7291193005033428e-5   α: 1.0
iter: 16    |r|: 4.0962955391749e-6   α: 1.0
iter: 17    |r|: 1.0301773198252933e-6   α: 1.0
iter: 18    |r|: 2.6560749183908207e-7
```




```
Test Summary:      | Pass  Total  Time
quadruped standing |    3      3  3.2s
```

Out[ ]:
```
Test.DefaultTestSet("quadruped standing", Any[], 3, false, false, true, 1.7
06292685189511e9, 1.706292688416884e9, false)
```

In [ ]:
```
let
```

```
    # let's visualize the balancing position we found

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


end
```

```
[ Info: Listening on: 127.0.0.1:8703, thread id: 1
┌ Info: MeshCat server started. You can open the visualizer by visiting the
following URL in your browser:
└ http://127.0.0.1:8703
```

Out[ ]:

Part C (5 pts): One sentence short answer

1. Why do we use a linesearch?

put ONE SENTENCE answer here

A: To aviod overshooting the minimum and boost the convergence rate by shrinking to a appropriate step size.

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

put ONE SENTENCE answer here

A:

For convex problem: we don't need a linesearch because Newton's method is guaranteed to converge to the minimum.

For nonconvex problem: we need a linesearch to avoid overshooting.

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

put ONE SENTENCE answer here

A: For standard quadratic programming problems, we don't need a linesearch since the objective function is convex.

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

```
  Activating project at `~/Desktop/2024Spring/CMU16745_OptimalControl/CMU16-745-Optimal-Control-HW/hw1`
```

Q2 (30 pts): Augmented Lagrangian Quadratic Program Solver

Part (A): QP Solver (10 pts)

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

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

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:

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

where $\circ$ is element-wise multiplication.

In [ ]:
```
# 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
    h_mask = h_ineq(qp, x) .< 0.0
    mu_mask = μ .== 0.0
    Irpo = I(length(μ)) * ρ
    zero_mask = .!(h_mask .* mu_mask)
    return Irpo .* zero_mask
end

function augmented_lagrangian(
    qp::NamedTuple,
    x::Vector,
    λ::Vector,
    μ::Vector,
    ρ::Real,
)::Real
    cost(qp, x) +
    dot(λ, c_eq(qp, x)) +
    dot(μ, h_ineq(qp, x)) +
    0.5 * ρ * c_eq(qp, x)' * c_eq(qp, x) +
    0.5 * h_ineq(qp, x)' * mask_matrix(qp, x, μ, ρ) * h_ineq(qp, x)
end

function logging(
    qp::NamedTuple,
    main_iter::Int,
    AL_gradient::Vector,
```

```
    x::Vector,
    λ::Vector,
    μ::Vector,
    ρ::Real,
)
    # TODO: stationarity norm
    stationarity_norm = norm(
        [qp.Q * x + qp.q + qp.A' * λ + qp.G' * μ]
    )
    @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))

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

    # TODO:
    rho = 1.0
    phi = 2.0
    for main_iter = 1:max_iters
        if verbose
            AL_gradient =
                FD.gradient(_x -> augmented_lagrangian(qp, _x, λ, μ, rho), x
            logging(qp, main_iter, AL_gradient, x, λ, μ, rho)
        end

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

        # update x
        for inner_iter = 1:max_iters
            L_gradient =
                FD.gradient(_x -> augmented_lagrangian(qp, _x, λ, μ, rho), x
            L_hessian =
                FD.hessian(_x -> augmented_lagrangian(qp, _x, λ, μ, rho), x)
            x = x - L_hessian \ L_gradient
            if norm(L_gradient) < tol
                break
            end
```

```
            if inner_iter == max_iters
                error("x did not converge")
            end
        end

        # update lambda, mu
        λ = λ + rho * c_eq(qp, x)
        μ = max.(0.0, μ + rho * h_ineq(qp, x))

        # update rho
        rho = phi * rho

        # TODO: convergence criteria based on tol   CHECK: if this is the co
        kkt_stationary = [qp.Q * x + qp.q + qp.A' * λ + qp.G' * μ]
        kkt_stationary_check = norm(kkt_stationary) < tol
        kkt_primal_eq = c_eq(qp, x)
        kkt_primal_eq_check = norm(kkt_primal_eq) < tol
        kkt_primal_ineq = h_ineq(qp, x)
        kkt_primal_ineq_check = all(kkt_primal_ineq .<= 0.0)
        kkt_complementarity = μ .* h_ineq(qp, x)
        kkt_complementarity_check = norm(kkt_complementarity) < tol
        if kkt_stationary_check &&
           kkt_primal_eq_check &&
           kkt_primal_ineq_check &&
           kkt_complementarity_check
            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   2.98e+01   5.60e+01   4.38e+00   6.49e+00   0.00e+00   1e+00
  2   4.70e-15   9.84e+00   5.51e-01   1.27e+00   4.59e-01   2e+00
  3   8.03e-01   6.82e+00   9.68e-02   6.03e-01   6.58e-02   4e+00
  4   2.06e-01   2.65e+00   7.41e-02   8.78e-02   7.71e-02   8e+00
  5   3.48e-14   3.28e-01   3.92e-03   5.39e-03   2.04e-03   2e+01
  6   3.23e-14   5.65e-02   2.86e-04   5.25e-04   2.36e-04   3e+01
  7   1.40e-13   5.24e-03   1.36e-05   2.70e-05   1.23e-05   6e+01
  8   1.75e-13   2.65e-04   3.55e-07   7.34e-07   3.32e-07   1e+02
  9   5.10e-13   7.31e-06   4.91e-09   1.04e-08   4.68e-09   3e+02
 10   5.76e-13   1.07e-07   3.53e-11   7.61e-11   3.42e-11   5e+02
 11   1.43e-12   8.20e-10   1.31e-13   2.86e-13   1.28e-13   1e+03
 12   4.44e-12   9.22e-12   8.88e-16   8.88e-16   3.26e-16   2e+03
 13   7.08e-12   2.36e-11   9.16e-16   0.00e+00   9.38e-16   4e+03
```

Out[ ]:
```
([-0.3262308057133928, 0.24943797997175304, -0.43226766440523, -1.417224697
1242028, -1.3994527400875778, 0.609958240852346, -0.07312202122168282, 1.30
31477522000245, 0.5389034791065969, -0.7225813651685227], [-0.1283519512351
2233, -2.8376241672109543, -0.8320804499649519], [0.0363529426381497, 0.0,
0.0, 1.059444495111564, 0.0])
```

QP Solver test

In [ ]:
```
# 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   2.98e+01   5.60e+01   4.38e+00   6.49e+00   0.00e+00  1e+00
  2   4.70e-15   9.84e+00   5.51e-01   1.27e+00   4.59e-01  2e+00
  3   8.03e-01   6.82e+00   9.68e-02   6.03e-01   6.58e-02  4e+00
  4   2.06e-01   2.65e+00   7.41e-02   8.78e-02   7.71e-02  8e+00
  5   3.48e-14   3.28e-01   3.92e-03   5.39e-03   2.04e-03  2e+01
  6   3.23e-14   5.65e-02   2.86e-04   5.25e-04   2.36e-04  3e+01
  7   1.40e-13   5.24e-03   1.36e-05   2.70e-05   1.23e-05  6e+01
  8   1.75e-13   2.65e-04   3.55e-07   7.34e-07   3.32e-07  1e+02
  9   5.10e-13   7.31e-06   4.91e-09   1.04e-08   4.68e-09  3e+02
 10   5.76e-13   1.07e-07   3.53e-11   7.61e-11   3.42e-11  5e+02
 11   9.58e-13   8.18e-10   1.31e-13   2.86e-13   1.28e-13  1e+03
 12   1.75e-12   5.22e-12   4.44e-16   6.66e-16   4.70e-16  2e+03
 13   5.73e-12   1.15e-11   4.44e-16   8.88e-16   1.61e-17  4e+03
 14   1.11e-11   1.97e-11   2.78e-17   4.44e-16   2.88e-18  8e+03
Test Summary: | Pass  Total  Time
qp solver     |    3      3  0.2s
```

Out[ ]:
```
Test.DefaultTestSet("qp solver", Any[], 3, false, false, true, 1.7074293975
51332e9, 1.70742939776971e9, false)
```

Simulating a Falling Brick with QPs

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

The Dynamics

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

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

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

We can discretize the dynamics with backward Euler:

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

We also have the following contact constraints:

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

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

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

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

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:

KKT conditions:

$$\text{stationarity} \quad Mv_{k+1} + M(\Delta t \cdot g - v_k) - (J^T \Delta t)\mu_{k+1} = 0 \tag{14}$$
$$\Rightarrow v_{k+1} = (\frac{1}{M} J^T \mu_{k+1} - g)\Delta t + v_k \tag{15}$$
$$\text{primal feasibility} \quad -J(q_k + \Delta t \cdot v_{k+1}) \leq 0 \tag{16}$$
$$\text{dual feasibility} \quad \mu_{k+1} \geq 0 \tag{17}$$
$$\text{complementarity} \quad \mu_{k+1} \circ (-J(q_k + \Delta t \cdot v_{k+1})) = 0 \tag{18}$$

Part (C): Brick Simulation (5 pts)

In [ ]:
```
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 = [0.0; 9.81]
    J = [0 1.0]

    qp = (
        Q=[mass 0.0; 0.0 mass],
        q=mass * (Δt * g - v),
        A=zeros(0, 2), # don't edit this
        b=zeros(0),   # don't edit this
        G=-[0.0 Δt],
        h=J * q,
    )

    return qp
end
```

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

In [ ]:
```
@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
    @show typeof(qp.q)
    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 -0.01]) < 1e-10
    @test abs(qp.h[1] - 3) < 1e-10
```

```
end
```

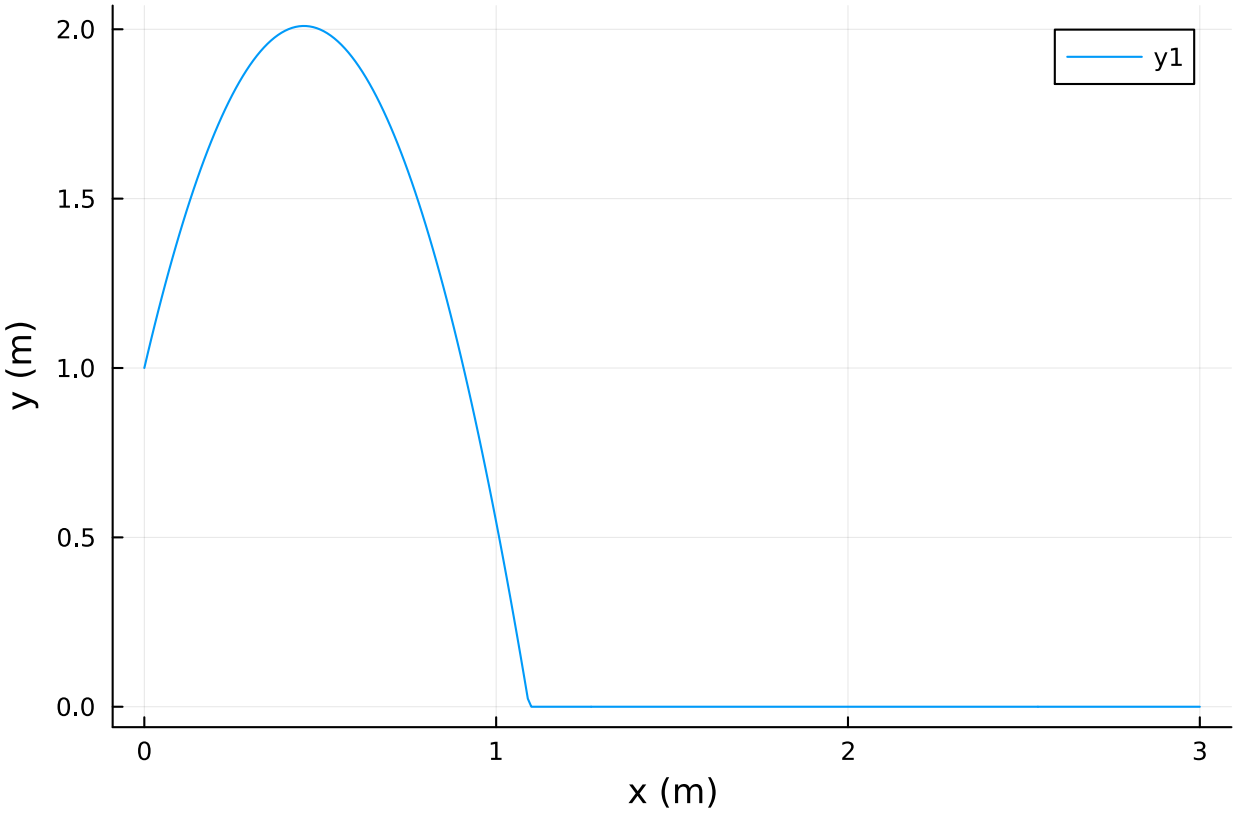
```
typeof(qp.q) = Vector{Float64}
Test Summary: | Pass  Total  Time
brick qp      |   10     10  0.2s
```

Out[ ]:
```
Test.DefaultTestSet("brick qp", Any[], 10, false, false, true, 1.7074293983
23972e9, 1.707429398549595e9, false)
```

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

    dt = 0.01
    T = 3.0

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

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

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

    # TODO: simulate the brick by forming and solving a qp
    # at each timestep. Your QP should solve for vs[k+1], and
    # you should use this to update qs[k+1]
    for k = 1:N-1
        qp = brick_simulation_qp(qs[k], vs[k])
        v, λ, μ = solve_qp(qp; verbose = false, max_iters = 100, tol = 1e-6)
        vs[k+1] = v
        qs[k+1] = qs[k] + 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)
```

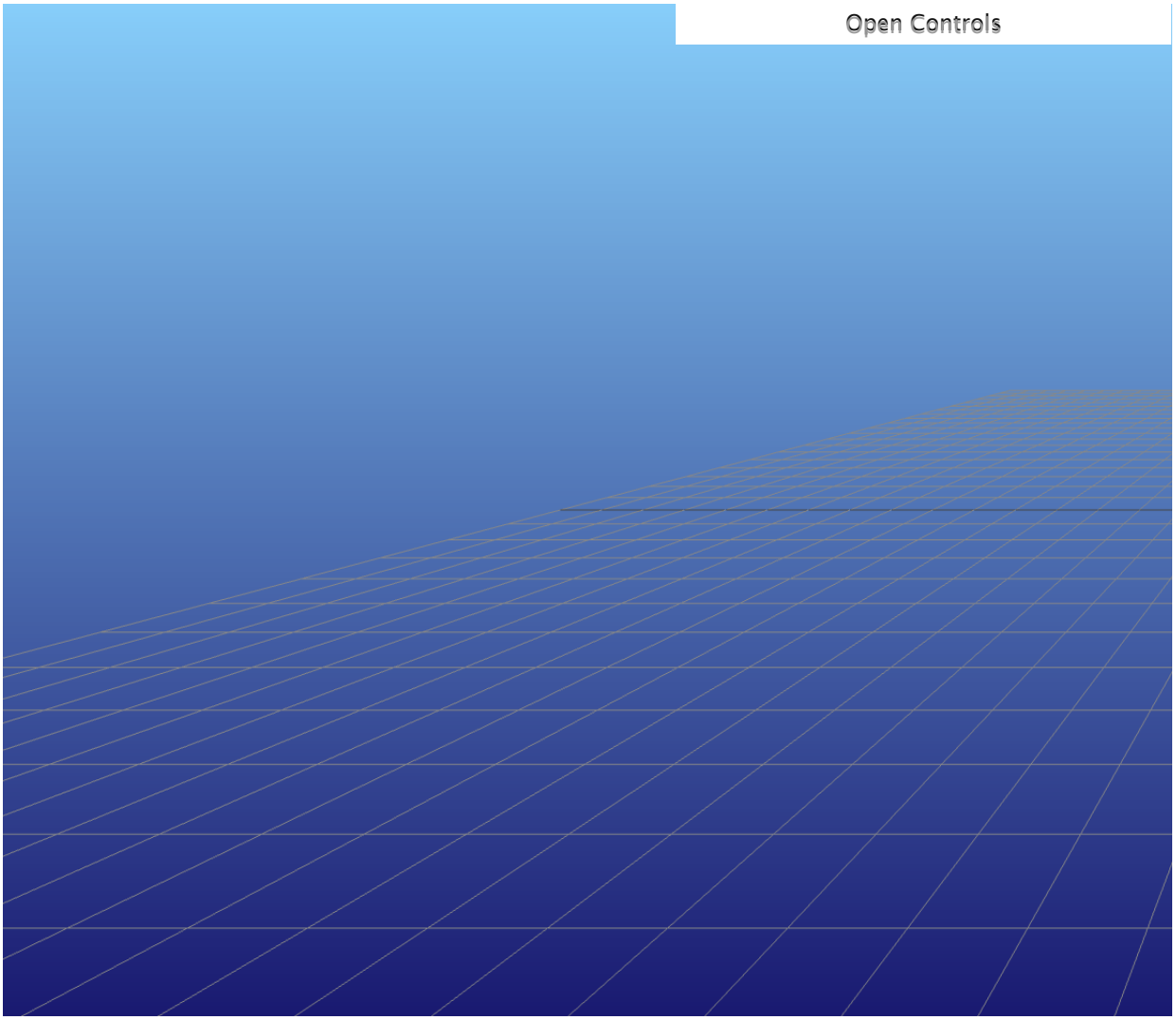
```
end
```

```
#= In[6]:30 =# @test(abs(maximum(ys) - 2) < 0.1) = Test Passed
#= In[6]:31 =# @test(minimum(ys) > -0.01) = Test Passed
#= In[6]:32 =# @test(abs(xs[end] - 3) < 0.01) = Test Passed
#= In[6]:35 =# @test(maximum(xdot) < 1.0001) = Test Passed
#= In[6]:36 =# @test(minimum(xdot) > 0.9999) = Test Passed
#= In[6]:37 =# @test(ys[110] > 0.01) = Test Passed
#= In[6]:38 =# @test(abs(ys[111]) < 0.01) = Test Passed
#= In[6]:39 =# @test(abs(ys[112]) < 0.01) = Test Passed
```




```
[ Info: Listening on: 127.0.0.1:8700, thread id: 1
┌ Info: MeshCat server started. You can open the visualizer by visiting the
following URL in your browser:
└ http://127.0.0.1:8700
```

Out[ ]:



Part D (5 pts): Solve a QP

Use your QP solver to solve the following optimization problem:

$$\begin{aligned}
\min_{y\in\mathbb{R}^2, a\in\mathbb{R}, b\in\mathbb{R}} \quad & \frac{1}{2} y^T \begin{bmatrix} 1 & .3 \\ .3 & 1 \end{bmatrix} y + a^2 + 2b^2 + \begin{bmatrix} -2 & 3.4 \end{bmatrix} y + 2a + 4b && (19) \\
\text{st} \quad & a + b = 1 && (20) \\
& \begin{bmatrix} -1 & 2.3 \end{bmatrix} y + a - 2b = 3 && (21) \\
& -0.5 \leq y \leq 1 && (22) \\
& -1 \leq a \leq 1 && (23) \\
& -1 \leq b \leq 1 && (24)
\end{aligned}$$

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

In [ ]:

```
@testset "part D" begin
    # define qp parameters
    qp = (
        Q = [
```

```
            1.0 0.3 0.0 0.0
            0.3 1.0 0.0 0.0
            0.0 0.0 1.0 0.0
            0.0 0.0 0.0 2.0
        ],
        q = [-2.0, 3.4, 2.0, 4.0],
        A = [0.0 0.0 1.0 1.0; -1.0 2.3 1.0 -2.0],
        b = [1.0, 3.0],
        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],
    )

    x, λ, μ = solve_qp(qp; verbose = false, max_iters = 100, 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 Summary: | Pass  Total  Time
part D        |    3      3  1.1s
```

Out[ ]:
```
Test.DefaultTestSet("part D", Any[], 3, false, false, true, 1.7074294201355
56e9, 1.707429421275817e9, false)
```

Part E (5 pts): One sentence short answer

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

put ONE SENTENCE answer here

A: No. If the initial guess is feasible, the constraint is not active, which implies we are doing unconstrained optimization. Consequently, during the update, the constrains might be violated.

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

put ONE SENTENCE answer here

A: Yes. Otherwise the Newton's method is not applicable since it requires the Hessian.

1. Is the QP in part D always convex?

put ONE SENTENCE answer here

A: Yes. The objective function is convex, and the constraints are affine.