In [1]:

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

```
  Activating environment at `~/villa/Studyroom/Sem_2_Assignments/16745A/Optimal-Control-16-745_HW1_S23/
Project.toml`
```

Julia Warnings

Just like Python, Julia lets you do the following:

In [2]:

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

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

    @show x y 1 x.^2
#     @show x
end
```

```
x = [1, 2, 3]
x = [300, 2, 100]
y = [300, 2, 100]
1 = 1
x .^ 2 = [90000, 4, 10000]
```

Out[2]:

```
3-element Vector{Int64}:
 90000
     4
 10000
```

In [3]:

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

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

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

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

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

Out[3]:

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

Optional function arguments

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

In [4]:

```
## optional arguments in functions

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

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

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

Out[4]:

```
(1, 2, 4, -30)
```

Q1: Integration (20 pts)

In this question we are going to integrate the equations of motion for a double pendulum using multiple explicit and implicit integrators. We will write a generic simulation function for each of the two categories (explicit and implicit), and compare 6 different integrators.

The continuous time dynamics of the cartpole are written as a function:

$$\dot{x} = f(x)$$

In the code you will see `xdot = dynamics(params, x)` .

Part A (10 pts): Explicit Integration

Here we are going to implement the following explicit integrators:

- Forward Euler (explicit)
- Midpoint (explicit)
- RK4 (explicit)

In [5]:

```
# these two functions are given, no TODO's here
function double_pendulum_dynamics(params::NamedTuple, x::Vector)
    # continuous time dynamics for a double pendulum given state x,
    # also known as the "equations of motion".
    # returns the time derivative of the state, ẋ (dx/dt)

    # the state is the following:
    θ1,θ1,θ2,θ2 = 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 = [
        θ1;
        ( m2*g*sin(θ2)*c - m2*s*(L1*c*θ1^2 + L2*θ2^2) - (m1+m2)*g*sin(θ1) ) / ( L1 *(m1+m2*s^2) );
        θ2;
        ((m1+m2)*(L1*θ1^2*s - g*sin(θ2) + g*sin(θ1)*c) + m2*L2*θ2^2*s*c) / (L2 * (m1 + m2*s^2));
        ]

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


    # the state is the following:
    θ1,θ1,θ2,θ2 = 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*θ1*cos(θ1), 0, L1*θ1*sin(θ1)]
    v2 = v1 + [L2*θ2*cos(θ2), 0, L2*θ2*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[5]:

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

Now we are going to simulate this double pendulum by integrating the equations of motion with the simplest explicit integrator, the Forward Euler method:

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

In [6]:

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

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

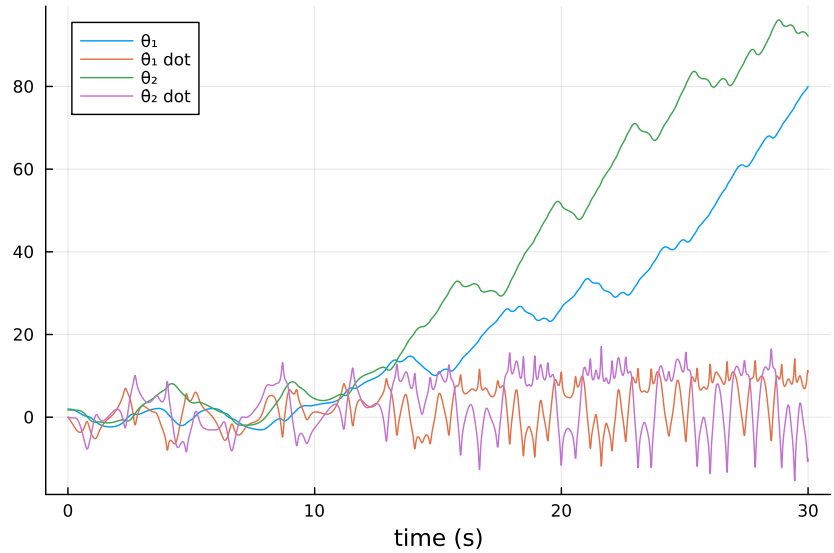
Out[6]:

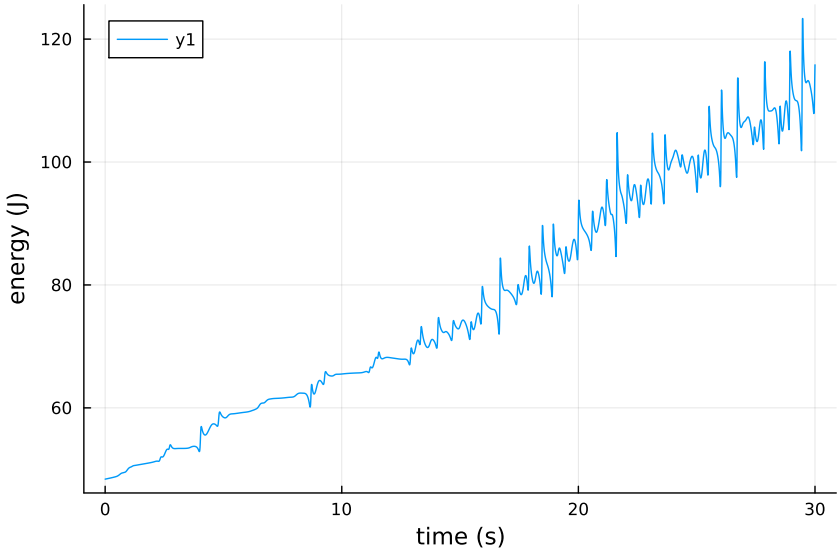
```
forward_euler
```

In [7]:

```
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
    @show X[1]

    # TODO: simulate the double pendulum with `forward_euler`
    # X[k] = `x_k`, so X[k+1] = forward_euler(params, double_pendulum_dynamics, X[k], dt)
    for i=2:length(X)
        X[i] .= forward_euler(params, double_pendulum_dynamics, X[i-1], 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
```

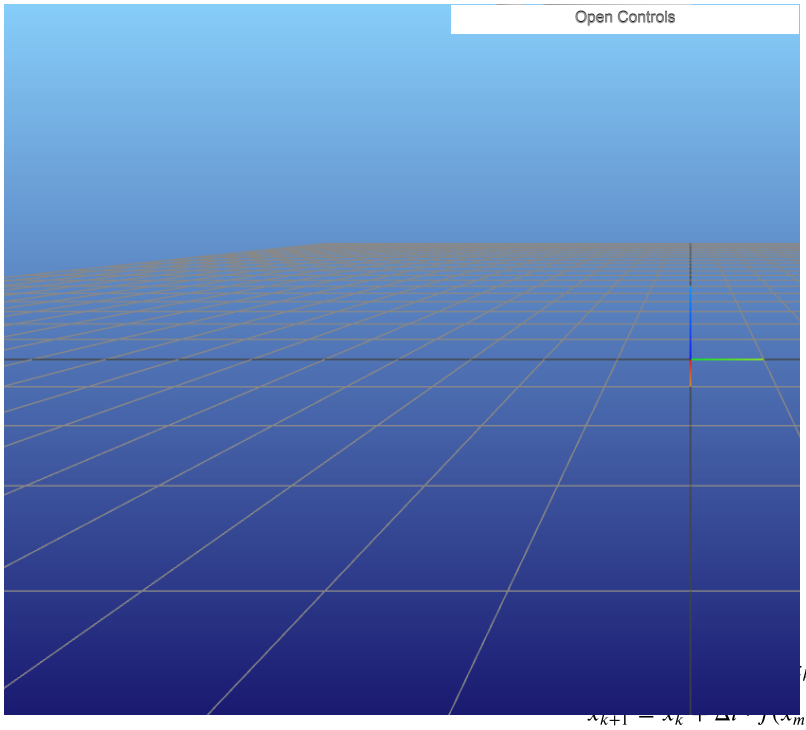
```
X[1] = [1.9634954084936207, 0.0, 1.7453292519943295, 0.0]
#= In[7]:37 =# @test(norm(X[end]) > 1.0e-10) = Test Passed
#= In[7]:38 =# @test(2 < E[end] / E[1] < 3) = Test Passed
```






```
┌ Info: MeshCat server started. You can open the visualizer by visiting the following URL in your brows
er:
└ http://127.0.0.1:8700
```

Out[7]:



RK4:

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

In [8]:

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

Out[8]:

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

In [9]:

```
function simulate_explicit(params::NamedTuple,dynamics::Function,integrator::Function,x0::Vector,dt::Real,tf::Real)
    # 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 i=2:length(X)
        X[i] .= integrator(params, dynamics, X[i-1], dt)
    end

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

Out[9]:

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

In [10]:

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

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

Out[10]:

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

Part B (10 pts): Implicit Integrators

Explicit integrators work by calling a function with x_k and 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)

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

Hermite Simpson (3rd order)

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

When you implement these integrators, you will update the functions such that they take in a dynamics function, x_k and x_{k+1}, and return the residuals described above. We are NOT solving these yet, we are simply returning the residuals for each implicit integrator that we want to be 0.

In [11]:

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

Out[11]:

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

In [12]:

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

    # initialize guess
    x2 = 1*x1

    # TODO: use Newton's method to solve for x2 such that residual for the integrator is 0
    # DO NOT USE A WHILE LOOP
    for i = 1:max_iters
        f = implicit_integrator(params, dynamics, x1, x2, dt)
        ∇f = FD.jacobian(x -> implicit_integrator(params, dynamics, x1, x, dt), x2)
        Δx = - ∇f\f

        # TODO: return x2 when the norm of the residual is below tol
        if norm(Δx)<tol
            return x2
        end
        x2 .= x2 .+ Δx
    end
    error("implicit integrator solve failed")
end
```

Out[12]:

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

In [13]:

```
@testset "implicit integrator check" begin
# let
    dt = 1e-1
    x1 = [.1,.2,.3,.4]

    for integrator in [backward_euler, implicit_midpoint, hermite_simpson]
        println("-----testing $integrator ------")
        x2 = implicit_integrator_solve(params, double_pendulum_dynamics, integrator, x1, dt)
        @test norm(integrator(params, double_pendulum_dynamics, x1, x2, dt)) < 1e-10
    end

end
```

```
-----testing backward_euler ------
-----testing implicit_midpoint ------
-----testing hermite_simpson ------
Test Summary:              | Pass  Total
implicit integrator check |    3      3
```

Out[13]:

```
Test.DefaultTestSet("implicit integrator check", Any[], 3, false, false)
```

In [14]:

```
function simulate_implicit(params::NamedTuple,dynamics::Function,implicit_integrator::Function,x0::Vector,dt::Real,tf:
    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 i=2:length(X)
# implicit_integrator_solve(params::NamedTuple, dynamics::Function, implicit_integrator::Function, x1::Vector, dt::Rea
        X[i] .= implicit_integrator_solve(params, dynamics, implicit_integrator, X[i-1], dt)
#         X[i] .= implicit_integrator(params, dynamics, X[i-1], dt)
    end


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

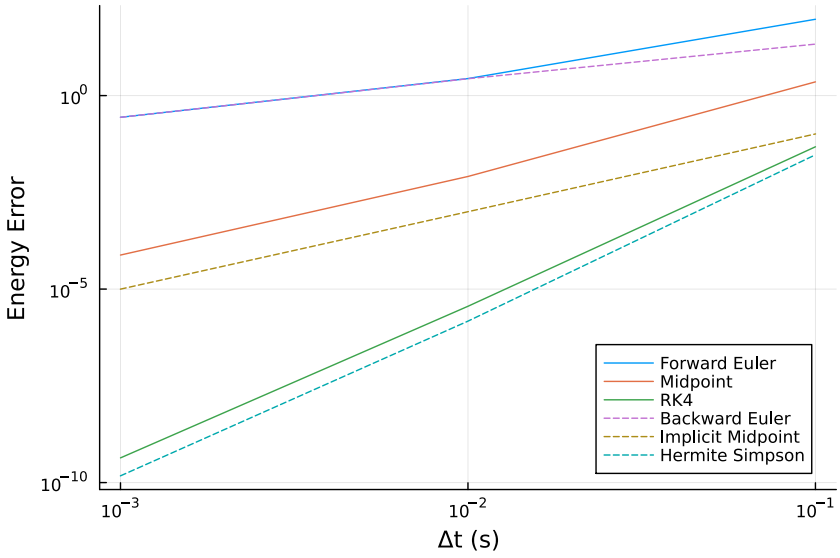
Out[14]:

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

In [15]:

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


const tf = 2.0
let
    # here we compare everything
    dts = [1e-3,1e-2,1e-1]
    explicit_integrators = [forward_euler, midpoint, rk4]
    implicit_integrators = [backward_euler, implicit_midpoint, hermite_simpson]

    explicit_data = [get_explicit_energy_error(integrator, dts) for integrator in explicit_integrators]
    implicit_data = [get_implicit_energy_error(integrator, dts) for integrator in implicit_integrators]

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

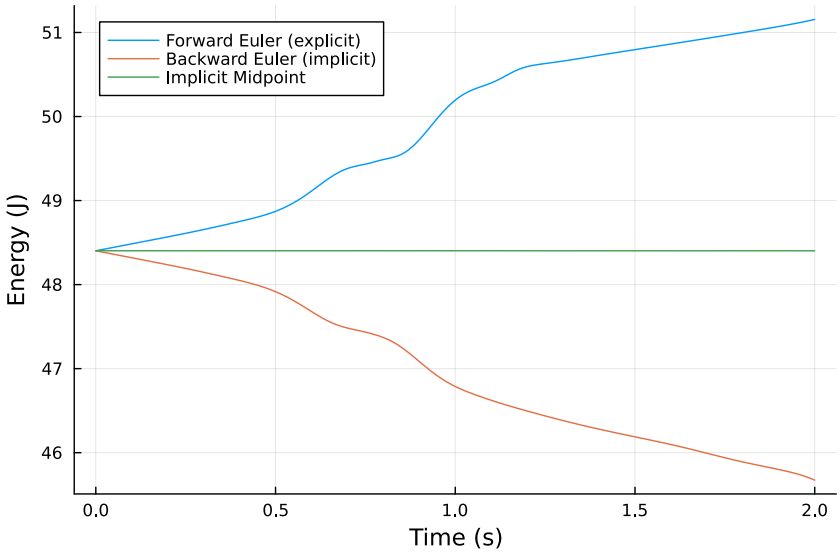
Out[15]:



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

In [16]:

```
@testset "energy behavior" begin

    # simulate with all integrators
    dt = 0.01
    t_vec = 0:dt:tf
    E1 = simulate_explicit(params,double_pendulum_dynamics,forward_euler,x0,dt,tf)[2]
    E2 = simulate_implicit(params,double_pendulum_dynamics,backward_euler,x0,dt,tf)[2]
    E3 = simulate_implicit(params,double_pendulum_dynamics,implicit_midpoint,x0,dt,tf)[2]
    E4 = simulate_implicit(params,double_pendulum_dynamics,hermite_simpson,x0,dt,tf)[2]
    E5 = simulate_explicit(params,double_pendulum_dynamics,midpoint,x0,dt,tf)[2]
    E6 = simulate_explicit(params,double_pendulum_dynamics,rk4,x0,dt,tf)[2]

    # plot forward/backward euler and implicit midpoint
    plot(t_vec,E1, label = "Forward Euler (explicit)")
    plot!(t_vec,E2, label = "Backward Euler (implicit)")
    display(plot!(t_vec,E3, label = "Implicit Midpoint",xlabel = "Time (s)", ylabel="Energy (J)"))

    # test energy behavior
    E0 = E1[1]

    @test 2.5  < (E1[end] - E0) < 3.0
    @test -3.0 < (E2[end] - E0) < -2.5
    @test abs(E3[end] - E0) < 1e-2
    @test abs(E0 - E4[end]) < 1e-4
    @test abs(E0 - E5[end]) < 1e-1
    @test abs(E0 - E6[end]) < 1e-4
end
```




```
Test Summary:   | Pass  Total
energy behavior |    6      6
```

Out[16]:

```
Test.DefaultTestSet("energy behavior", Any[], 6, false, 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.

In [1]:

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

```
  Activating environment at `~/villa/Studyroom/Sem_2_Assignments/16745A/Optimal-Control-16-745_HW1_S23/
Project.toml`
```

Q2: Equality Constrained Optimization (20 pts)

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

Part A (10 pts)

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

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

Which has the following Lagrangian:

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

and the following KKT conditions for optimality:

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

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

Newton's Method with a Linesearch

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

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

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

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

Where 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 [2]:

```
function linesearch(z::Vector, Δz::Vector, merit_fx::Function;
                    max_ls_iters = 10)::Float64 # optional argument with a default

    # TODO: return maximum α≤1 such that merit_fx(z + α*Δz) < merit_fx(z)
    # with a backtracking linesearch (α = α/2 after each iteration)
    α = 1

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

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

        α = α/2
    end
    error("linesearch failed")
end

function newtons_method(z0::Vector, res_fx::Function, res_jac_fx::Function, merit_fx::Function;
                        tol = 1e-10, max_iters = 50, verbose = false)::Vector{Vector{Float64}}

    # TODO: implement Newton's method given the following inputs:
    # - z0, initial guess
    # - res_fx, residual function
    # - res_jac_fx, Jacobian of residual function wrt z
    # - merit_fx, merit function for use in linesearch

    # optional arguments
    # - tol, tolerance for convergence. Return when norm(residual)<tol
    # - max iter, max # of iterations
    # - verbose, bool telling the function to output information at each iteration

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

    # NOTE: DO NOT USE A WHILE LOOP ANYWHERE

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


    for i = 1:(max_iters - 1)

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

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

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

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

        # TODO: caculate Newton step (don't forget the negative sign)
        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[2]:

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

In [3]:

```
@testset "check Newton" begin

    f(_x) = [sin(_x[1]), cos(_x[2])]
    df(_x) = FD.jacobian(f, _x)
    merit(_x) = norm(f(_x))

    x0 = [-1.742410372590328, 1.4020334125022704]

    X = newtons_method(x0, f, df, merit; tol = 1e-10, max_iters = 50, verbose = true)

    # check this took the correct number of iterations
    # if your linesearch isn't working, this will fail
    # you should see 1 iteration where α = 0.5
    @test length(X) == 6

    # check we actually converged
    @test norm(f(X[end])) < 1e-10

end
```

```
iter: 1    |r|: 0.9995239729818045   α: 1.0
iter: 2    |r|: 0.9421342427117169   α: 0.5
iter: 3    |r|: 0.1753172908866053   α: 1.0
iter: 4    |r|: 0.0018472215879181287   α: 1.0
iter: 5    |r|: 2.1010529101114843e-9   α: 1.0
iter: 6    |r|: 2.5246740534795566e-16   Test Summary: | Pass  Total
check Newton  |    2      2
```

Out[3]:

```
Test.DefaultTestSet("check Newton", Any[], 2, false, 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 [4]:

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

Out[4]:

In [5]:

```
# we will use Newton's method to solve the constrained optimization problem shown above
function cost(x::Vector)
    Q = [1.65539  2.89376; 2.89376  6.51521];
    q = [2;-3]
    return 0.5*x'*Q*x + q'*x + exp(-1.3*x[1] + 0.3*x[2]^2)
end
function constraint(x::Vector)
    norm(x) - 0.5
end
# HINT: use this if you want to, but you don't have to
function constraint_jacobian(x::Vector)::Matrix
    # since `constraint` returns a scalar value, ForwardDiff
    # will only allow us to compute a gradient of this function
    # (instead of a Jacobian). This means we have two options for
    # computing the Jacobian: Option 1 is to just reshape the gradient
    # into a row vector

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

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

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

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

    x = z[1:2]
    λ = z[3:3] # Or z[3]?????
    λ = z[3]
    @show z
    @show λ

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

    return kkt_conditions
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
    L(x,λ) = cost(x) + λ'*constraint(x)
    J = [
        FD.hessian(x_ -> L(x_,λ), x) constraint_jacobian(x)';
        constraint_jacobian(x) 0
    ]
    regularizer = 1e-3*Diagonal([1,1,-1])
    return J .+ regularizer
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
#     L(x,λ) = cost(x) + λ'*constraint(x)
    J = [
        FD.hessian(cost, x) constraint_jacobian(x)';
        constraint_jacobian(x) 0
    ]
    regularizer = 1e-3*Diagonal([1,1,-1])
    return J .+ regularizer
end
```

Out[5]:

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

In [6]:

```
@testset "Test Jacobians" begin

    # first we check the regularizer
    z = randn(3)
    J_fn = fn_kkt_jac(z)
    J_gn = gn_kkt_jac(z)

    # check what should/shouldn't be the same between
    @test norm(J_fn[1:2,1:2] - J_gn[1:2,1:2]) > 1e-10
    @test abs(J_fn[3,3] + 1e-3) < 1e-10
    @test abs(J_gn[3,3] + 1e-3) < 1e-10
    @test norm(J_fn[1:2,3] - J_gn[1:2,3]) < 1e-10
    @test norm(J_fn[3,1:2] - J_gn[3,1:2]) < 1e-10
end
```

```
Test Summary:  | Pass  Total
Test Jacobians |    5      5
```

Out[6]:

```
Test.DefaultTestSet("Test Jacobians", Any[], 5, false, false)
```

In [7]:

```
@testset "Full Newton" begin


    z0 = [-.1, .5, 0] # initial guess
    merit_fx(_z) = norm(kkt_conditions(_z)) # simple merit function
    Z = newtons_method(z0, kkt_conditions, fn_kkt_jac, merit_fx; tol = 1e-4, max_iters = 100, verbose = true)
    R = kkt_conditions.(Z)

    # make sure we converged on a solution to the KKT conditions
    @test norm(kkt_conditions(Z[end])) < 1e-4
    @test length(R) < 6


    # ------------------------plotting stuff------------------------
    Rp = [[abs(R[i][ii]) + 1e-15 for i = 1:length(R)] for ii = 1:length(R[1])] # this gets abs of each term at each it

    plot(Rp[1],yaxis=:log,ylabel = "|r|",xlabel = "iteration",
         yticks= [1.0*10.0^(-x) for x = float(15:-1:-2)],
         title = "Convergence of Full Newton on KKT Conditions",label = "|r_1|")
    plot!(Rp[2],label = "|r_2|")
    display(plot!(Rp[3],label = "|r_3|"))

    contour(-.6:.1:0,0:.1:.6, (x1,x2)-> cost([x1;x2]),title = "Cost Function",
            xlabel = "X₁", ylabel = "X₂",fill = true)
    xcirc = [.5*cos(θ) for θ in range(0, 2*pi, length = 200)]
    ycirc = [.5*sin(θ) for θ in range(0, 2*pi, length = 200)]
    plot!(xcirc,ycirc, lw = 3.0, xlim = (-.6, 0), ylim = (0, .6),label = "constraint")
    z1_hist = [z[1] for z in Z]
    z2_hist = [z[2] for z in Z]
    display(plot!(z1_hist, z2_hist, marker = :d, label = "xₖ"))
    # ------------------------plotting stuff------------------------
end
```

```
z = [-0.1, 0.5, 0.0]
λ = 0.0
iter: 1    |r|: 1.7188450769812715   z = [-0.44417180918862653, 0.4221780142709899, 1.088861051106566
8]
λ = 1.0888610511065668
z = [-0.1, 0.5, 0.0]
λ = 0.0
α: 1.0
z = [-0.44417180918862653, 0.4221780142709899, 1.0888610511065668]
λ = 1.0888610511065668
iter: 2    |r|: 0.8150495962203247   z = [-0.3036022848982124, 0.40634423680908655, 1.091429835837133]
λ = 1.091429835837133
z = [-0.44417180918862653, 0.4221780142709899, 1.0888610511065668]
λ = 1.0888610511065668
α: 1.0
z = [-0.3036022848982124, 0.40634423680908655, 1.091429835837133]
λ = 1.091429835837133
iter: 3    |r|: 0.025448943695826287   z = [-0.2986383705244941, 0.4010243719147633, 1.096225134913626
7]
```

In [8]:

```
@testset "Gauss-Newton" begin


    z0 = [-.1, .5, 0] # initial guess
    merit_fx(_z) = norm(kkt_conditions(_z)) # simple merit function

    # the only difference in this block vs the previous is `gn_kkt_jac` instead of `fn_kkt_jac`
    Z = newtons_method(z0, kkt_conditions, gn_kkt_jac, merit_fx; tol = 1e-4, max_iters = 100, verbose = true)
    R = kkt_conditions.(Z)

    # make sure we converged on a solution to the KKT conditions
    @test norm(kkt_conditions(Z[end])) < 1e-4
    @test length(R) < 10


    # ------------------------plotting stuff------------------------
    Rp = [[abs(R[i][ii]) + 1e-15 for i = 1:length(R)] for ii = 1:length(R[1])] # this gets abs of each term at each it

    plot(Rp[1],yaxis=:log,ylabel = "|r|",xlabel = "iteration",
         yticks= [1.0*10.0^(-x) for x = float(15:-1:-2)],
         title = "Convergence of Full Newton on KKT Conditions",label = "|r_1|")
    plot!(Rp[2],label = "|r_2|")
    display(plot!(Rp[3],label = "|r_3|"))

    contour(-.6:.1:0,0:.1:.6, (x1,x2)-> cost([x1;x2]),title = "Cost Function",
            xlabel = "X₁", ylabel = "X₂",fill = true)
    xcirc = [.5*cos(θ) for θ in range(0, 2*pi, length = 200)]
    ycirc = [.5*sin(θ) for θ in range(0, 2*pi, length = 200)]
    plot!(xcirc,ycirc, lw = 3.0, xlim = (-.6, 0), ylim = (0, .6),label = "constraint")
    z1_hist = [z[1] for z in Z]
    z2_hist = [z[2] for z in Z]
    display(plot!(z1_hist, z2_hist, marker = :d, label = "xₖ"))
    # ------------------------plotting stuff------------------------
end
```

```
z = [-0.1, 0.5, 0.0]
λ = 0.0
iter: 1    |r|: 1.7188450769812715   z = [-0.44417180918862653, 0.4221780142709899, 1.0888610511065668]
λ = 1.0888610511065668
z = [-0.1, 0.5, 0.0]
λ = 0.0
α: 1.0
z = [-0.44417180918862653, 0.4221780142709899, 1.0888610511065668]
λ = 1.0888610511065668
iter: 2    |r|: 0.8150495962203247   z = [-0.2914943321527874, 0.41904874452378005, 1.0678537003108388]
λ = 1.0678537003108388
z = [-0.44417180918862653, 0.4221780142709899, 1.0888610511065668]
λ = 1.0888610511065668
α: 1.0
z = [-0.2914943321527874, 0.41904874452378005, 1.0678537003108388]
λ = 1.0678537003108388
iter: 3    |r|: 0.19186516708148574   z = [-0.3027433547361709, 0.39852658733981405, 1.105780350254007
8]
λ = 1.1057803502540078
z = [-0.2914943321527874, 0.41904874452378005, 1.0678537003108388]
λ = 1.0678537003108388
α: 1.0
z = [-0.3027433547361709, 0.39852658733981405, 1.1057803502540078]
λ = 1.1057803502540078
iter: 4    |r|: 0.04663490553083029   z = [-0.2975752324082024, 0.4018379584327316, 1.0931088386832075]
λ = 1.0931088386832075
z = [-0.3027433547361709, 0.39852658733981405, 1.1057803502540078]
λ = 1.1057803502540078
α: 1.0
z = [-0.2975752324082024, 0.4018379584327316, 1.0931088386832075]
λ = 1.0931088386832075
iter: 5    |r|: 0.01332977842954523   z = [-0.29894380097445084, 0.40079849578706145, 1.096986163578871
1]
λ = 1.0969861635788711
z = [-0.2975752324082024, 0.4018379584327316, 1.0931088386832075]
λ = 1.0931088386832075
α: 1.0
z = [-0.29894380097445084, 0.40079849578706145, 1.0969861635788711]
λ = 1.0969861635788711
iter: 6    |r|: 0.0037714013578573355   z = [-0.29854706334262865, 0.40108454732144355, 1.0959101292574
474]
λ = 1.0959101292574474
z = [-0.29894380097445084, 0.40079849578706145, 1.0969861635788711]
λ = 1.0969861635788711
α: 1.0
z = [-0.29854706334262865, 0.40108454732144355, 1.0959101292574474]
λ = 1.0959101292574474
iter: 7    |r|: 0.001071165054782875   z = [-0.2986589895100171, 0.40100266156581094, 1.09621749573717
1]
λ = 1.096217495737171
z = [-0.29854706334262865, 0.40108454732144355, 1.0959101292574474]
λ = 1.0959101292574474
α: 1.0
z = [-0.2986589895100171, 0.40100266156581094, 1.096217495737171]
λ = 1.096217495737171
iter: 8    |r|: 0.00030392210707413806   z = [-0.2986271714359921, 0.40102584325727714, 1.0961304261917
06]
λ = 1.096130426191706
z = [-0.2986589895100171, 0.40100266156581094, 1.096217495737171]
λ = 1.096217495737171
α: 1.0
z = [-0.2986271714359921, 0.40102584325727714, 1.096130426191706]
λ = 1.096130426191706
iter: 9    |r|: 8.625764141582568e-5   z = [-0.1, 0.5, 0.0]
λ = 0.0
z = [-0.44417180918862653, 0.4221780142709899, 1.0888610511065668]
λ = 1.0888610511065668
z = [-0.2914943321527874, 0.41904874452378005, 1.0678537003108388]
λ = 1.0678537003108388
z = [-0.3027433547361709, 0.39852658733981405, 1.1057803502540078]
λ = 1.1057803502540078
z = [-0.2975752324082024, 0.4018379584327316, 1.0931088386832075]
λ = 1.0931088386832075
z = [-0.29894380097445084, 0.40079849578706145, 1.0969861635788711]
λ = 1.0969861635788711
z = [-0.29854706334262865, 0.40108454732144355, 1.0959101292574474]
λ = 1.0959101292574474
z = [-0.2986589895100171, 0.40100266156581094, 1.096217495737171]
λ = 1.096217495737171
z = [-0.2986271714359921, 0.40102584325727714, 1.096130426191706]
λ = 1.096130426191706
z = [-0.2986271714359921, 0.40102584325727714, 1.096130426191706]
λ = 1.096130426191706
```

Test Summary:	Pass	Total
Gauss-Newton	2	2

Out[8]:

```
Test.DefaultTestSet("Gauss-Newton", Any[], 2, false, 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 [9]:

```
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)
```

The WebIO Jupyter extension was not detected. See the [WebIO Jupyter integration documentation (https://juliagizmos.github.io/WebIO.jl/latest/providers/ijulia/)](https://juliagizmos.github.io/WebIO.jl/latest/providers/ijulia/) for more information.

```
┌ Info: MeshCat server started. You can open the visualizer by visiting the following URL in your brows
er:
└ http://127.0.0.1:8701
```

Out[9]:

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:

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

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 [10]:

```
# 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
    return 0.5*(x-x_guess)'*(x-x_guess) + 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]

    L(y, λ) = quadruped_cost(y) + λ'*quadruped_constraint(y)

    kkt_conditions = [
        FD.gradient(y_ -> L(y_,λ),y);
        FD.gradient(λ_ -> L(y,λ_),λ)
    ]

    # TODO: return the KKT conditions
    return kkt_conditions
end

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

    y = [x;u]

    # TODO: return Gauss-Newton Jacobian with a regularizer (try 1e-3,1e-4,1e-5,1e-6)
    J = [
        FD.hessian(quadruped_cost, y) FD.jacobian(quadruped_constraint, y)';
        FD.jacobian(quadruped_constraint, y) zeros(length(λ), length(λ))
    ]
    @show size(J)
    regularizer = 1e-3*cat(I(length(y)), -I(length(λ)), dims=(1,2))
    @show size(regularizer)
    return J .+ regularizer
end

# let
# #     quadruped_cost([x_guess;zeros(12)])
# #     quadruped_kkt(zeros(72))
#     quadruped_kkt_jac(zeros(72))
# end
```

```
WARNING: redefinition of constant x_guess. This may fail, cause incorrect answers, or produce other err
ors.
```

Out[10]:

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

In [11]:

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

@testset "quadruped standing" begin

    z0 = [x_guess; zeros(12); zeros(30)]
    Z = newtons_method(z0, quadruped_kkt, quadruped_kkt_jac, quadruped_merit; tol = 1e-6, verbose = true, max_iters =
    set_configuration!(mvis, Z[end][1:state_dim(model)÷2])
    R = norm.(quadruped_kkt.(Z))

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

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

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

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

end
```

```
iter: 1    |r|: 217.37236872332227   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 2    |r|: 124.92133581597675   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 3    |r|: 76.87596686964667   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 0.5
iter: 4    |r|: 34.7502021848973   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 0.25
iter: 5    |r|: 27.13978367169712   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 0.5
iter: 6    |r|: 23.876187729699637   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 7    |r|: 9.928511516366587   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 8    |r|: 0.8635831086124133   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 9    |r|: 0.8252015646633398   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 10    |r|: 1.5494640418601664   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 11    |r|: 0.01079482454404196   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 12    |r|: 0.00035696648511781296   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 13    |r|: 0.0006131222696283716   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 14    |r|: 8.01275653868689e-5   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 15    |r|: 1.729119398018798e-5   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 16    |r|: 4.096285441662522e-6   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 17    |r|: 1.0301881217122464e-6   size(J) = (72, 72)
size(regularizer) = (72, 72)
α: 1.0
iter: 18    |r|: 2.655991080263677e-7
```

Test Summary: | **Pass** **Total**
quadruped standing | 3 3

Out[11]:

```
Test.DefaultTestSet("quadruped standing", Any[], 3, false, false)
```

In [12]:

```
let

    # let's visualize the balancing position we found

    z0 = [x_guess; zeros(12); zeros(30)]
    Z = newtons_method(z0, quadruped_kkt, quadruped_kkt_jac, quadruped_merit; tol = 1e-6, verbose = false, max_iters =
    # visualizer
    mvis = initialize_visualizer(model)
    set_configuration!(mvis, Z[end][1:state_dim(model)÷2])
    render(mvis)


end
```

```
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)
size(J) = (72, 72)
size(regularizer) = (72, 72)

┌ Info: MeshCat server started. You can open the visualizer by visiting the following URL in your brows
er:
└ http://127.0.0.1:8702
```

Out[12]:

In [3]:

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

```
  Activating environment at `~/villa/Studyroom/Sem_2_Assignments/16745A/Optimal-Control-16-745_HW1_S23/
Project.toml`
```

Q2 (20 pts): Augmented Lagrangian Quadratic Program Solver

Here we are going to use the augmented lagrangian method described here in a video (https://www.youtube.com/watch?v=0x0JD5uO_ZQ), with the corresponding pdf here (https://github.com/Optimal-Control-16-745/lecture-notebooks-2022/blob/main/misc/AL_tutorial.pdf) to solve the following problem:

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

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:

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

where $\circ$ is element-wise multiplication.

In [14]:

```
## COPIED NEWTON'S METHOD FROM THE Q2.ipynb FILE

function linesearch(z::Vector, Δz::Vector, merit_fx::Function;
                    max_ls_iters = 10)::Float64 # optional argument with a default

    # TODO: return maximum α≤1 such that merit_fx(z + α*Δz) < merit_fx(z)
    # with a backtracking linesearch (α = α/2 after each iteration)
    α = 1

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

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

        α = α/2
    end
    error("linesearch failed")
end

function newtons_method(z0::Vector, res_fx::Function, res_jac_fx::Function, merit_fx::Function;
                        tol = 1e-10, max_iters = 50, verbose = false)::Vector{Vector{Float64}}

    # TODO: implement Newton's method given the following inputs:
    # - z0, initial guess
    # - res_fx, residual function
    # - res_jac_fx, Jacobian of residual function wrt z
    # - merit_fx, merit function for use in linesearch

    # optional arguments
    # - tol, tolerance for convergence. Return when norm(residual)<tol
    # - max iter, max # of iterations
    # - verbose, bool telling the function to output information at each iteration

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

    # NOTE: DO NOT USE A WHILE LOOP ANYWHERE

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


    for i = 1:(max_iters - 1)

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

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

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

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

        # TODO: caculate Newton step (don't forget the negative sign)
#         @show size(res_jac_fx(Z[i]))
#         @show size(r)
        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[14]:

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

In [20]:

```
# 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
    mask_matrix = ρ*I(length(μ))
    for i=1:length(μ)
        if h_ineq(qp, x)[i] < 0 && μ[i] == 0
            mask_matrix[i,i] = 0
        end
    end
    return mask_matrix
end

function augmented_lagrangian(qp::NamedTuple, x::Vector, λ::Vector, μ::Vector, ρ::Real)::Real
    L(x,λ,μ) = cost(qp,x) + λ'*c_eq(qp,x) + μ'*h_ineq(qp,x)
    return L(x,λ,μ) + (ρ/2)*c_eq(qp,x)'*c_eq(qp,x) + (1/2)*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
    L(x,λ,μ) = cost(qp,x) + λ'*c_eq(qp,x) + μ'*h_ineq(qp,x)
    ∇L_x = FD.gradient(x_ -> L(x_,λ,μ), x)
    stationarity_norm = norm(∇L_x) # 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))
    ρ = 1
#     ∇AL_x = zeros(length(x))

    L(x,λ,μ) = cost(qp,x) + λ'*c_eq(qp,x) + μ'*h_ineq(qp,x)
    L_ρ(x,λ,μ,ρ) = augmented_lagrangian(qp, x, λ, μ, ρ)

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

    # TODO:
    for main_iter = 1:max_iters
        ∇L_ρ(x) = FD.gradient(x_ -> L_ρ(x_,λ,μ,ρ), x)
        if verbose
            logging(qp, main_iter, ∇L_ρ(x), x, λ, μ, ρ)
        end

        # Minimizing L_ρ keeping λ,μ,ρ constant. So finding root x for ∇L_ρ = 0
        f(_x) = ∇L_ρ(_x)
        df(_x) = FD.jacobian(f, _x)
```

```
        merit(_x) = norm(f(_x))

        x .= newtons_method(x, f, df, merit, verbose=false)[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)`
        λ .= λ + ρ*c_eq(qp,x)
        μ .= max.(0, μ + ρ*h_ineq(qp,x))
        ρ = 10*ρ

        # TODO: convergence criteria based on tol
        ∇L_x = FD.gradient(x_ -> L(x_,λ,μ), x)
        if (maximum(h_ineq(qp,x)) < tol
                && norm(c_eq(qp,x),Inf) < tol
                && norm(∇L_x) < tol
                && all(μ.≥ 0))
            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-7)
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   1.10e-14   4.92e+01   5.51e-01   1.27e+00   4.59e-01  1e+01
  3   6.16e+00   8.87e+01   2.56e-02   3.07e-01   1.05e-02  1e+02
  4   5.52e-01   4.28e+01   6.84e-03   1.35e-02   7.94e-03  1e+03
  5   5.26e-12   5.30e+00   3.64e-05   1.62e-04   1.06e-04  1e+04
```

Out[20]:

```
([-0.32623080431873497, 0.24943798756566352, -0.4322676547111396, -1.4172246948129288, -1.3994527462892
89, 0.6099582436073466, -0.07312201788675664, 1.3031477492933288, 0.5389034765217046, -0.72258137076087
19], [-0.12835193069528705, -2.8376241686069887, -0.8320804891433029], [0.036352958372898314, 0.0, 0.0,
1.05944451240556, 0.0])
```

QP Solver test (10 pts)

In [21]:

```
# 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   1.10e-14   4.92e+01   5.51e-01   1.27e+00   4.59e-01  1e+01
  3   6.16e+00   8.87e+01   2.56e-02   3.07e-01   1.05e-02  1e+02
  4   5.52e-01   4.28e+01   6.84e-03   1.35e-02   7.94e-03  1e+03
  5   5.26e-12   5.30e+00   3.64e-05   1.62e-04   1.06e-04  1e+04
Test Summary: | Pass  Total
qp solver     |    3      3
```

Out[21]:

```
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:

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

Part (a): QP formulation (5 pts)

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

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

TASK: Write down the KKT conditions for the optimization problem above, and show that it's equivalent to the dynamics problem stated previously. Use LaTeX markdown.

Solution:

Lagrangian is given as:

$$L = \frac{1}{2} v_{k+1}^T M v_{k+1} + [M(\Delta t \cdot g - v_k)]^T v_{k+1} + \lambda(-J(q_k + \Delta t \cdot v_{k+1}))$$

Rearranging the Backward Euler equations:

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

$$\implies q_{k+1} = q_k + \Delta t \cdot v_{k+1}$$

So, the Lagrangian constraint is interchangeable as the following

$$L = \frac{1}{2} v_{k+1}^T M v_{k+1} + [M(\Delta t \cdot g - v_k)]^T v_{k+1} + \lambda(-J(q_{k+1}))$$

The KKT Conditions are:

$$\begin{aligned} \nabla_{v_{k+1}} L &= 0 && \text{(Stationarity)} \\ \implies v_{k+1} m + m(\Delta t.\, g - v_k) + (-J\Delta t)\lambda &= 0 && \text{(Eqv to velocity dynamics equation)} \\ \\ \nabla_\lambda L &\leq 0 && \text{(Primal feasibility)} \\ \implies Jq_{k+1} &\geq 0 \\ \\ \lambda &\geq 0 && \text{(Dual feasibility, same as condition on normal force)} \\ \\ \lambda.\, \nabla_\lambda L &= 0 && \text{(Complementarity)} \\ \implies \lambda_{k+1} Jq_{k+1} &= 0 \end{aligned}$$

Brick Simulation (5 pts)

In [40]:

```
function brick_simulation_qp(q, v; mass = 1.0, Δt = 0.01)

    # TODO: fill in the QP problem data for a simulation step
    # fill in Q, q, G, h, but leave A, b the same
    # this is because there are no equality constraints in this qp
    g = [0,9.81]
    M = mass*I(2)
    J = [0 1]

    qp = (
        Q = zeros(2,2),
        q = zeros(2),
        A = zeros(0,2), # don't edit this
        b = zeros(0),   # don't edit this
        G = zeros(1,2),
        h = zeros(1)
    )
    qp.Q .= M
    qp.q .= M*(Δt*g-v)
    qp.G .= -J*Δt
    qp.h .= J*q

    return qp
end
```

Out[40]:

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

In [41]:

```
@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[41]:

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

In [43]:

```
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
        qp = brick_simulation_qp(qs[i], vs[i]; Δt=dt)
        vs[i+1] .= solve_qp(qp)[1]
        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
```

```
iter   |∇Lₓ|      |∇ALₓ|     max(h)     |c|        compl     ρ
-----------------------------------------------------------------
  1   4.51e+00   4.51e+00  -1.00e+00   0.00e+00   0.00e+00  1e+00
iter   |∇Lₓ|      |∇ALₓ|     max(h)     |c|        compl     ρ
-----------------------------------------------------------------
  1   4.42e+00   4.42e+00  -1.04e+00   0.00e+00   0.00e+00  1e+00
iter   |∇Lₓ|      |∇ALₓ|     max(h)     |c|        compl     ρ
-----------------------------------------------------------------
  1   4.32e+00   4.32e+00  -1.09e+00   0.00e+00   0.00e+00  1e+00
iter   |∇Lₓ|      |∇ALₓ|     max(h)     |c|        compl     ρ
-----------------------------------------------------------------
  1   4.23e+00   4.23e+00  -1.13e+00   0.00e+00   0.00e+00  1e+00
iter   |∇Lₓ|      |∇ALₓ|     max(h)     |c|        compl     ρ
-----------------------------------------------------------------
  1   4.13e+00   4.13e+00  -1.17e+00   0.00e+00   0.00e+00  1e+00
iter   |∇Lₓ|      |∇ALₓ|     max(h)     |c|        compl     ρ
-----------------------------------------------------------------
  1   4.04e+00   4.04e+00  -1.21e+00   0.00e+00   0.00e+00  1e+00
iter   |∇Lₓ|      |∇ALₓ|     max(h)     |c|        compl     ρ
```

In [ ]: