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

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

Which has the following Lagrangian:

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

and the following KKT conditions for optimality:

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

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,
```

```julia
)::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
```

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

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

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

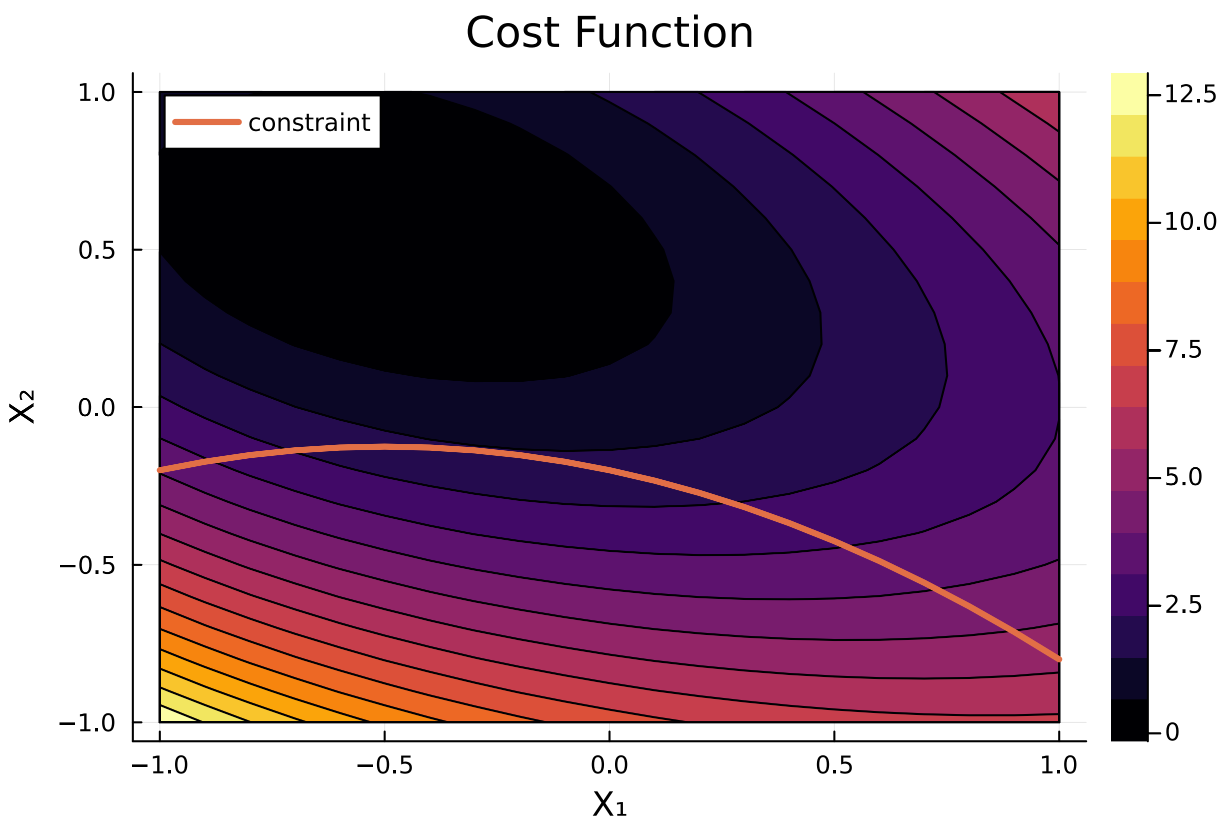
Out[ ]: Test.DefaultTestSet("check Newton", Any[], 2, false, false, true, 1.706283359469679e9, 1.706283359760339e9, false)

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 [ ]:
```julia
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
```

```julia
    # (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
```

```julia
    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 [ ]:
```julia
@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
```

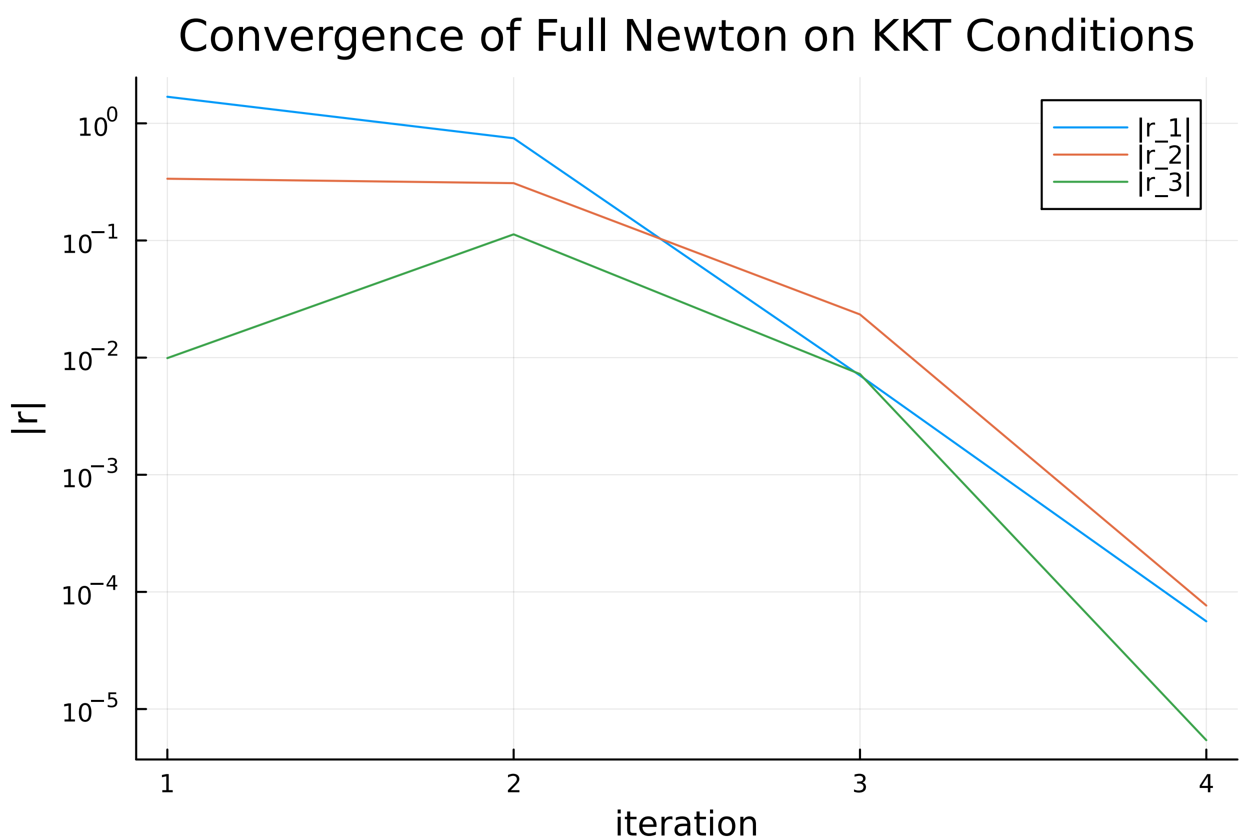
```
Test Summary:  | Pass  Total  Time
Test Jacobians |    5      5  1.9s
```

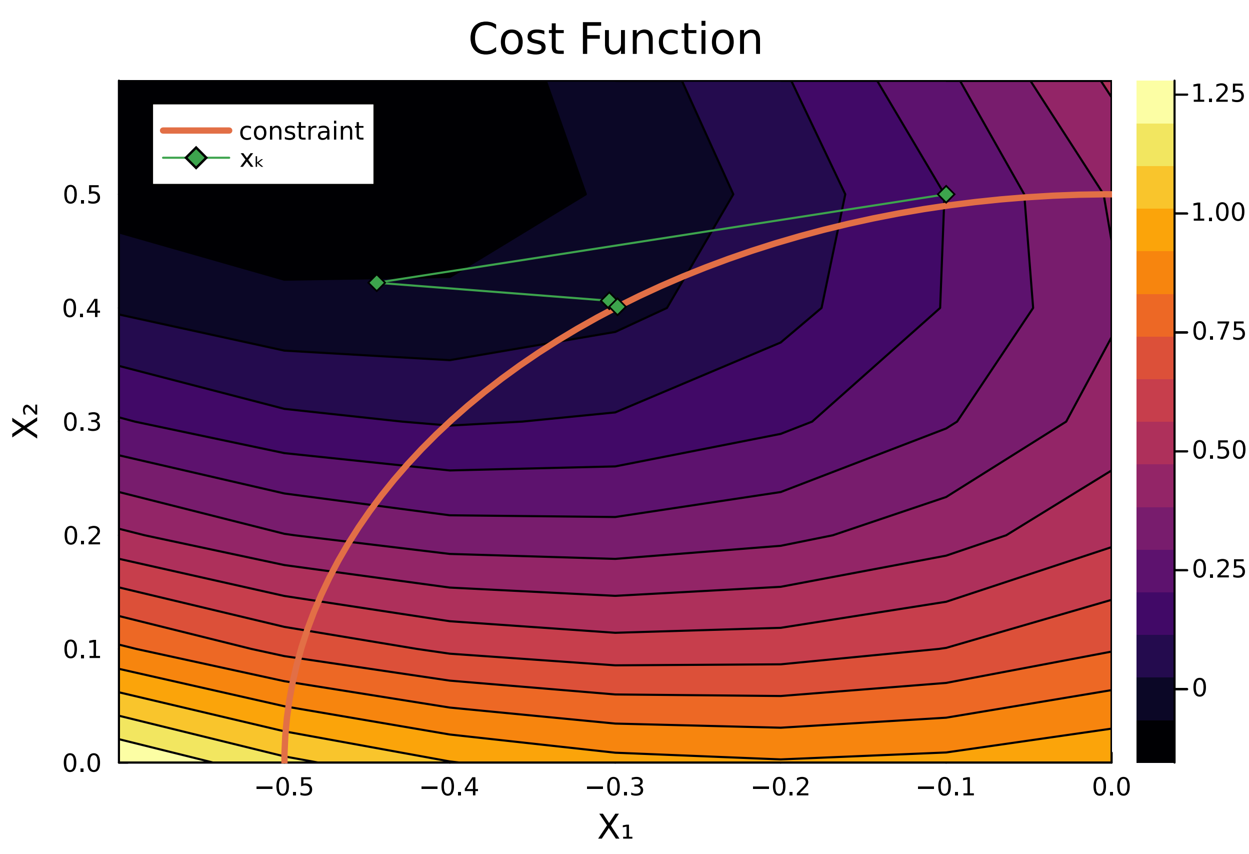
Out[ ]: Test.DefaultTestSet("Test Jacobians", Any[], 5, false, false, true, 1.706291878488978e9, 1.706291880427015e9, false)

In [ ]:
```julia
@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)
```

```julia
    # 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
```

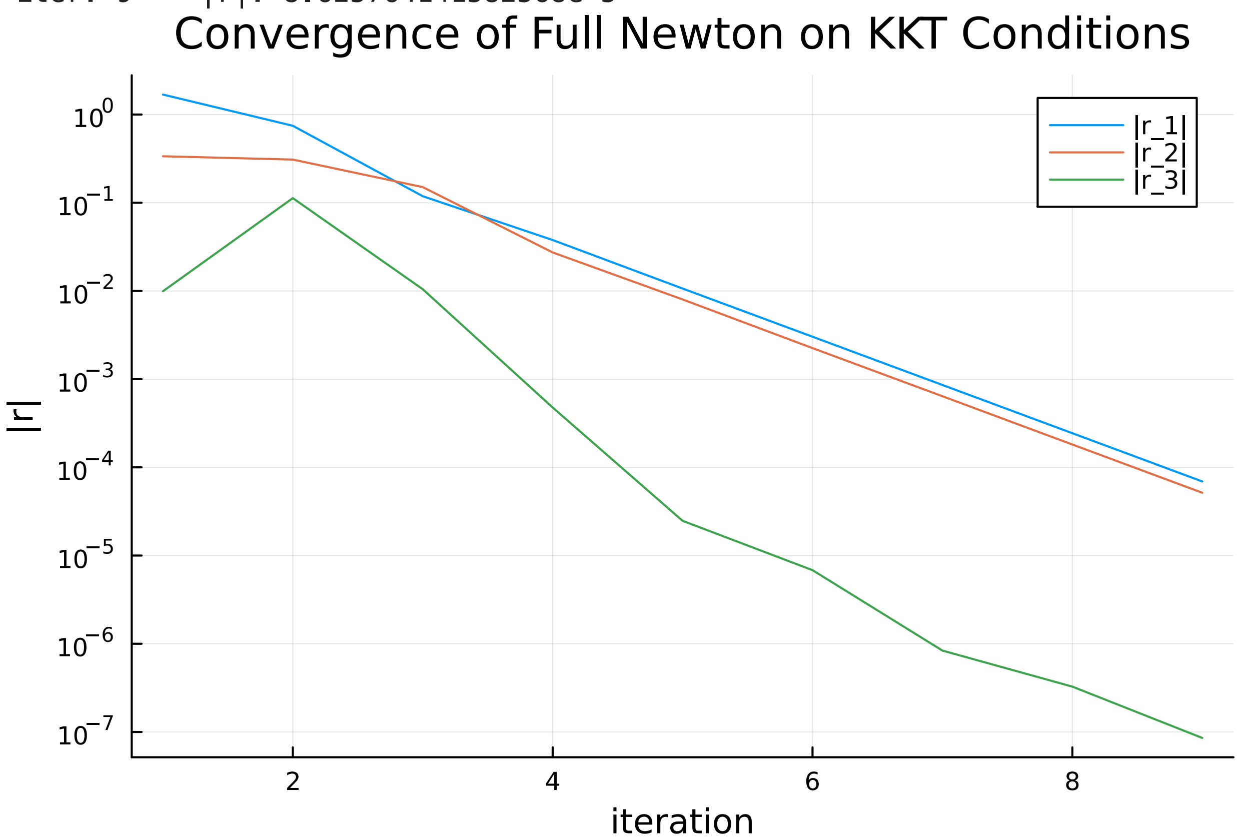
```
Out[ ]: Test.DefaultTestSet("Full Newton", Any[], 2, false, false, true, 1.70629188
8375631e9, 1.706291890070227e9, false)
```

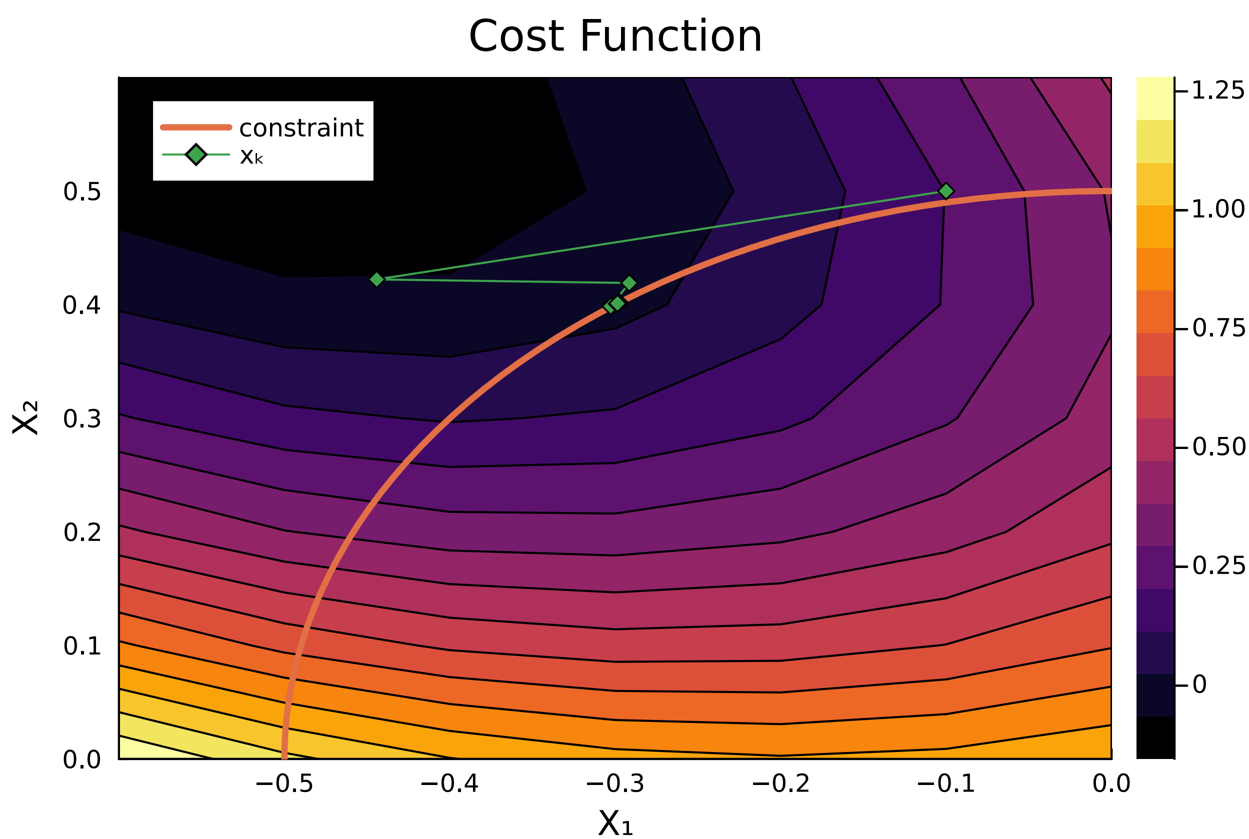
In [ ]:

```julia
@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, 1.7062918
98151176e9, 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 answers, 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:

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

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

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 [ ]:
```julia
# 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
```

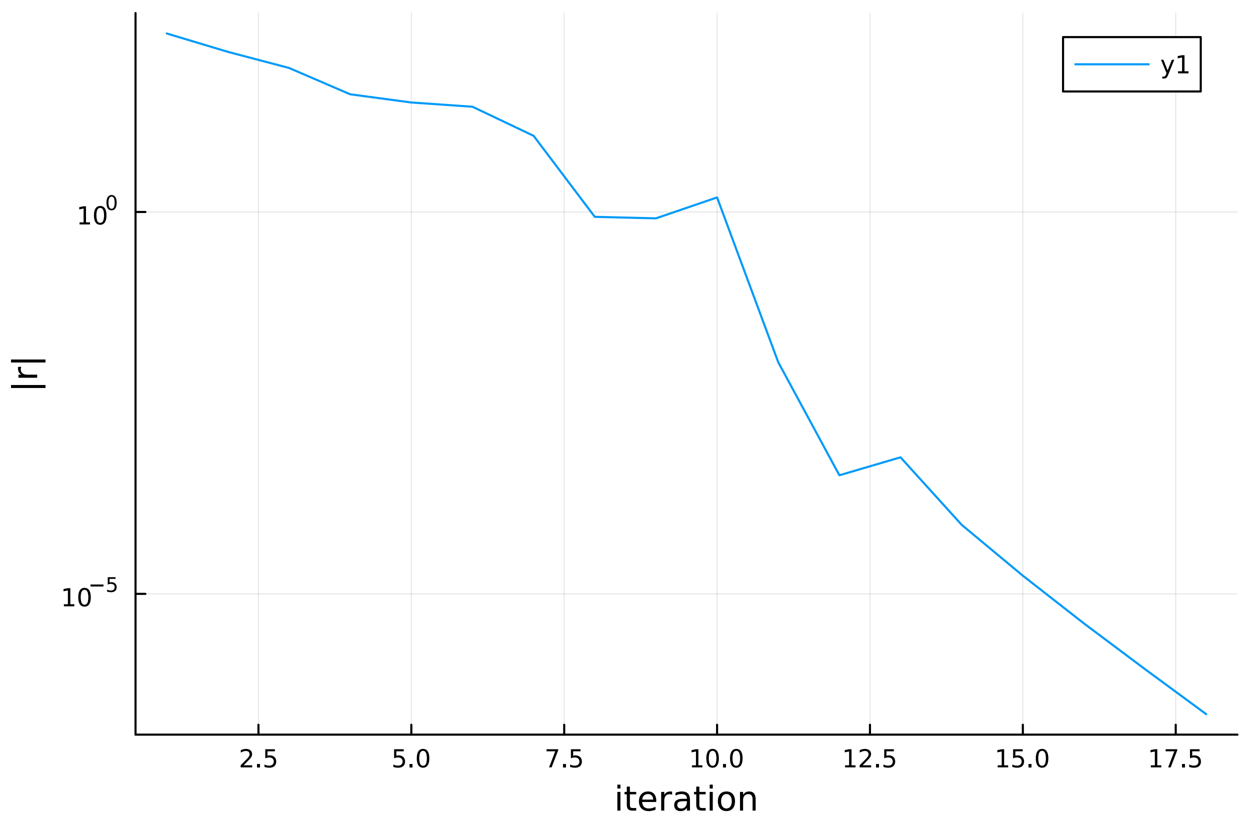
```
WARNING: redefinition of constant x_guess. This may fail, cause incorrect an
swers, or produce other errors.
```

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
```

```julia
    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, 1.706292685189511e9, 1.706292688416884e9, false)
```

In [ ]:
```julia
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.