In [1]:
```
# here is how we activate an environment in our current directory
import Pkg; Pkg.activate(@__DIR__)

# instantate this environment (download packages if you haven't)
Pkg.instantiate();

using Test, LinearAlgebra
import ForwardDiff as FD
import FiniteDiff as FD2
using Plots; plotly()
```

```
  Activating environment at `~/Dropbox/My Mac (MacBook Pro (2))/Desktop/CM
U/Optimal Control/HW0_S23/Project.toml`
┌ Warning: For saving to png with the `Plotly` backend `PlotlyBase` and `Pl
otlyKaleido` need to be installed.
│   err =
│    ArgumentError: Package PlotlyBase not found in current path:
│    - Run `import Pkg; Pkg.add("PlotlyBase")` to install the PlotlyBase pa
ckage.
│
└ @ Plots ~/.julia/packages/Plots/nuwp4/src/backends.jl:545
```

Out[1]:
```
Plots.PlotlyBackend()
```

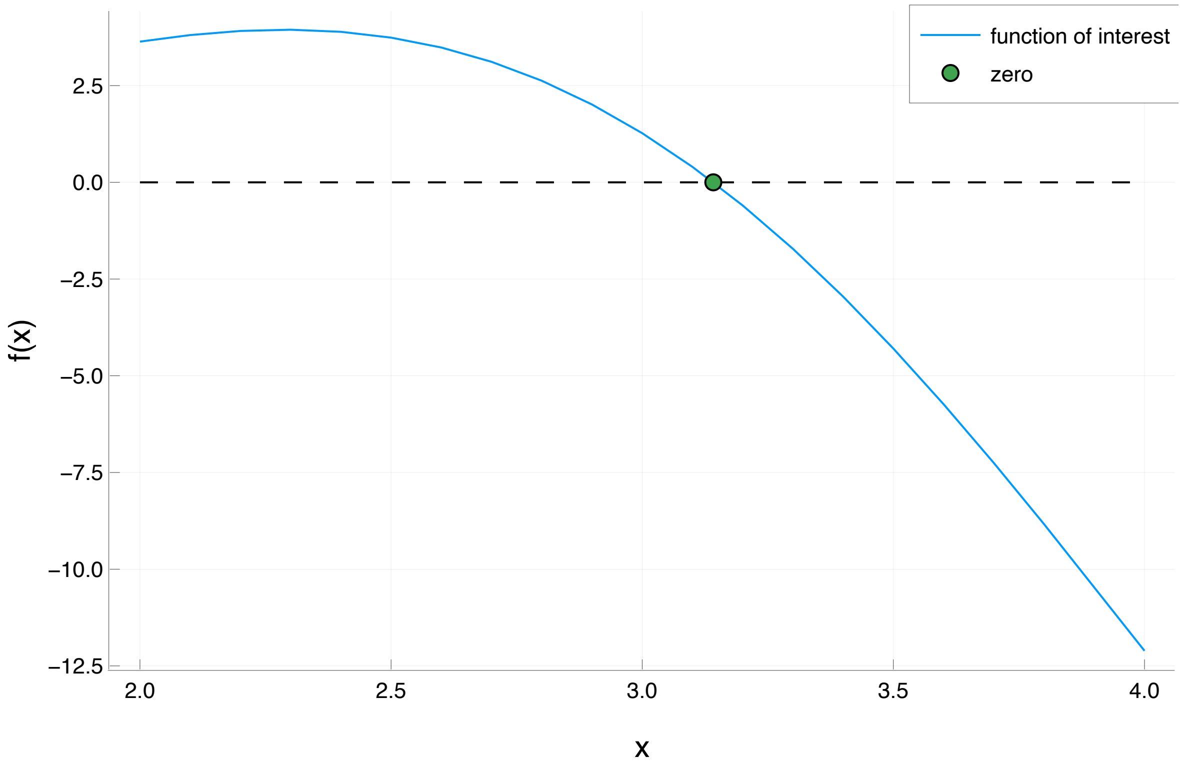
Q2: Newton's Method (20 pts)

Part (a): Newton's method in 1 dimension (8pts)

First let's look at a nonlinear function, and label where this function is equal to 0 (a root of the function).

In [2]:
```
let
    x = 2:0.1:4;
    y = sin.(x) .* x.^2
    plot(x,y,label = "function of interest")
    plot!(x,0*x,linestyle = :dash, color = :black,label = "")
    xlabel!("x")
    ylabel!("f(x)")
    scatter!([pi],[0],label = "zero")
end
```

Out[2]:



We are now going to use Newton's method to numerically evaluate the argument x where this function is equal to zero. To make this more general, let's define a residual function,

$$r(x) = \sin(x)x^2.$$

We want to drive this residual function to be zero (aka find a root to $r(x)$). To do this, we start with an initial guess at x_k, and approximate our residual function with a first-order Taylor expansion:

$$r(x_k + \Delta x) \approx r(x_k) + \left[\frac{\partial r}{\partial x}\bigg|_{x_k} \right] \Delta x.$$

We now want to find the root of this linear approximation. In other words, we want to find a Δx such that $r(x_k + \Delta x) = 0$. To do this, we simply re-arrange:

$$\Delta x = -\left[\frac{\partial r}{\partial x}\bigg|_{x_k} \right]^{-1} r(x_k).$$

We can now increment our estimate of the root with the following:

$$x_{k+1} = x_k + \Delta x$$

We have now described one step of Netwon's method. We started with an initial point, linearized the residual function, and solved for the Δx that drove this linear approximation to zero. We keep taking Newton steps until $r(x_k)$ is close enough to zero for our purposes (usually not hard to drive below 1e-10).

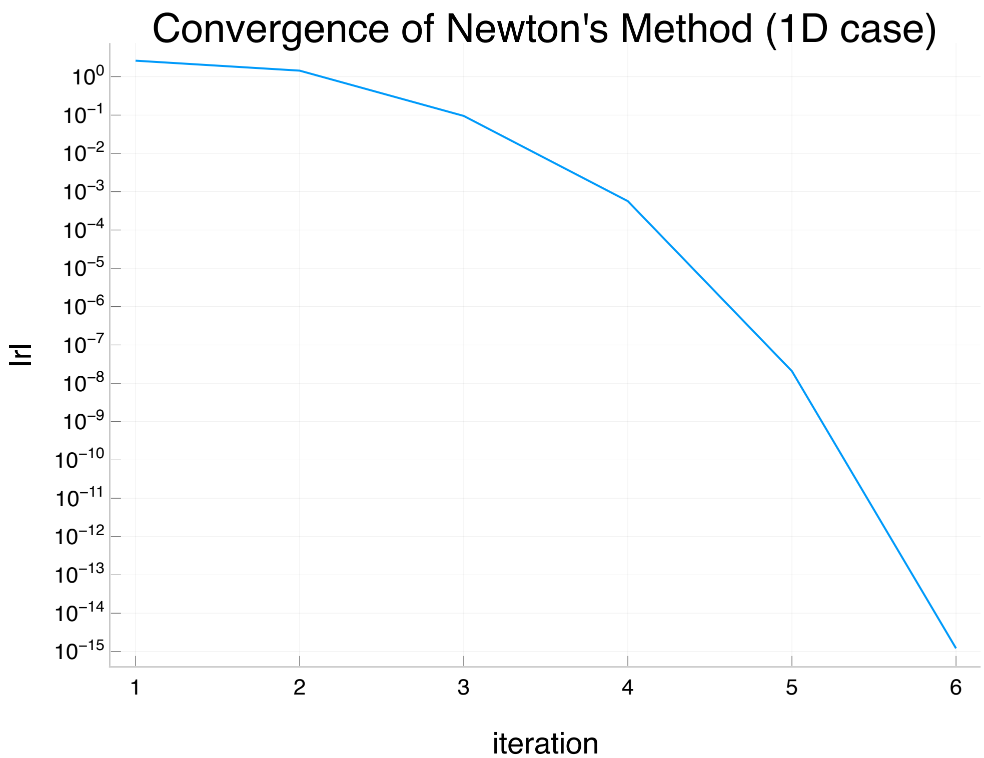
Julia tip: `x=A\b` solves linear systems of the form $Ax = b$ whether A is a matrix or a scalar.

In [3]:
```
"""
    X = newtons_method_1d(x0, residual_function; max_iters)

Given an initial guess x0::Float64, and `residual_function`,
use Newton's method to calculate the zero that makes
residual_function(x) ≈ 0. Store your iterates in a vector
X and return X[1:i]. (first element of the returned vector
should be x0, last element should be the solution)
"""

function newtons_method_1d(x0::Float64, residual_function::Function; max_ite
    # return the history of iterates as a 1d vector (Vector{Float64})
    # consider convergence to be when abs(residual_function(X[i])) < 1e-10
    # at this point, trim X to be X = X[1:i], and return X

    X = zeros(max_iters)
    X[1] = x0

    for i = 1:max_iters

        # TODO: Newton's method here
        ΔX = - FD.derivative(residual_function, X[i])\residual_function(X[i]
        X[i+1] = X[i] + ΔX

        # return the trimmed X[1:i] after you converges
        if abs(residual_function(X[i+1])) < 1e-10
            return X[1:i+1]
        elseif i == max_iters
            return X
        end

    end
    error("Newton did not converge")
end
```

Out[3]: newtons_method_1d (generic function with 1 method)

In [4]:
```
@testset "2a" begin
    # residual function
    residual_fx(_x) = sin(_x)*_x^2

    x0 = 2.8
    X = newtons_method_1d(x0, residual_fx; max_iters = 10)
    R = residual_fx.(X) # the . evaluates the function at each element of th

    @test abs(R[end]) < 1e-10

    # plotting
    display(plot(abs.(R),yaxis=:log,ylabel = "|r|",xlabel = "iteration",
         yticks= [1.0*10.0^(-x) for x = float(15:-1:-2)],
         title = "Convergence of Newton's Method (1D case)",label = ""))
```

```
end
```




```
Test Summary: | Pass  Total
2a            |    1      1
```

Out[4]:
```
Test.DefaultTestSet("2a", Any[], 1, false, false)
```

Part (b): Newton's method in multiple variables (8 pts)

We are now going to use Newton's method to solve for the zero of a multivariate function.

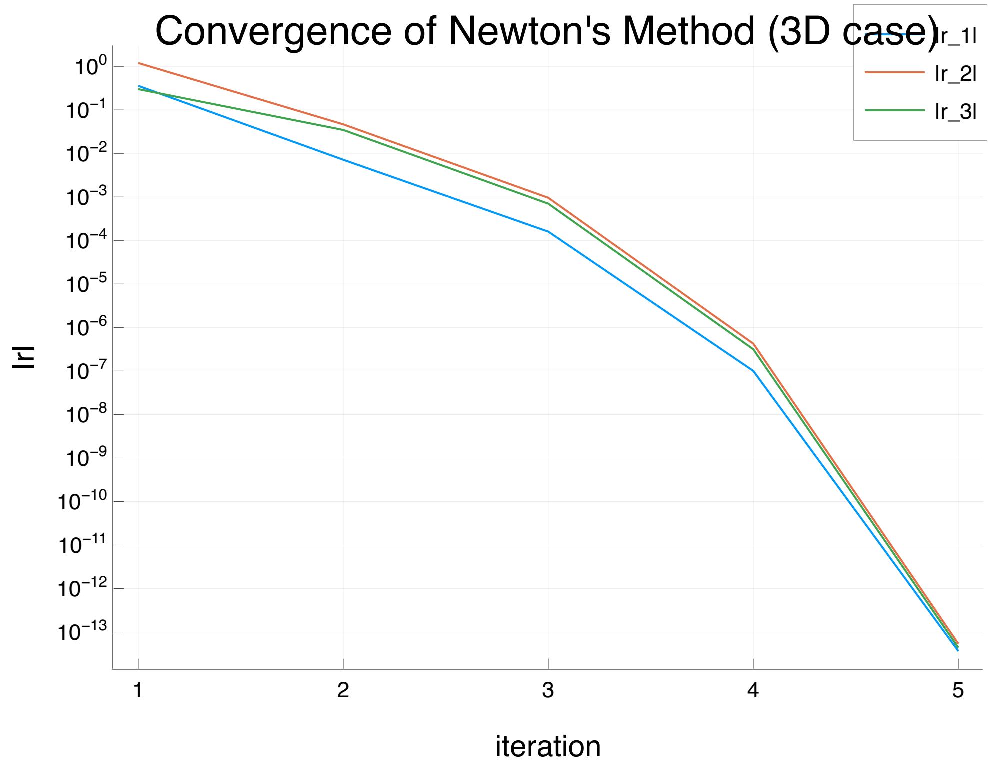
In [5]:
```
"""
    X = newtons_method(x0, residual_function; max_iters)

Given an initial guess x0::Vector{Float64}, and `residual_function`,
use Newton's method to calculate the zero that makes
norm(residual_function(x)) ≈ 0. Store your iterates in a vector
X and return X[1:i]. (first element of the returned vector
should be x0, last element should be the solution)
"""

function newtons_method(x0::Vector{Float64}, residual_function::Function; ma
    # return the history of iterates as a vector of vectors (Vector{Vector{F
    # consider convergence to be when norm(residual_function(X[i])) < 1e-10
    # at this point, trim X to be X = X[1:i], and return X

    X = [zeros(length(x0)) for i = 1:max_iters]
    X[1] = x0
```

```
    for i = 1:max_iters

        # TODO: Newton's method here
        ΔX = -FD.jacobian(residual_function, X[i])\residual_function(X[i])
        X[i+1] = X[i] + ΔX

        # return the trimmed X[1:i] after you converge
        if norm(residual_function(X[i+1])) < 1e-10
            return X[1:i+1]
        elseif i == max_iters
            return X
        end

    end
    error("Newton did not converge")
end
```

Out[5]: newtons_method (generic function with 1 method)

In [6]:

```
@testset "2b" begin
    # residual function
    r(x) = [sin(x[3] + 0.3)*cos(x[2]- 0.2) - 0.3*x[1];
            cos(x[1]) + sin(x[2]) + tan(x[3]);
            3*x[1] + 0.1*x[2]^3]

    x0 = [.1;.1;0.1]
    X = newtons_method(x0, r; max_iters = 10)
    R = r.(X) # the . evaluates the function at each element of the array

    Rp = [[abs(R[i][ii]) for i = 1:length(R)] for ii = 1:3] # this gets abs

    # tests
    @test norm(R[end])<1e-10

    # convergence plotting
    plot(Rp[1],yaxis=:log,ylabel = "|r|",xlabel = "iteration",
         yticks= [1.0*10.0^(-x) for x = float(15:-1:-2)],
         title = "Convergence of Newton's Method (3D case)",label = "|r_1|")
    plot!(Rp[2],label = "|r_2|")
    display(plot!(Rp[3],label = "|r_3|"))

end
```




```
Test Summary: | Pass  Total
2b            |    1      1
```

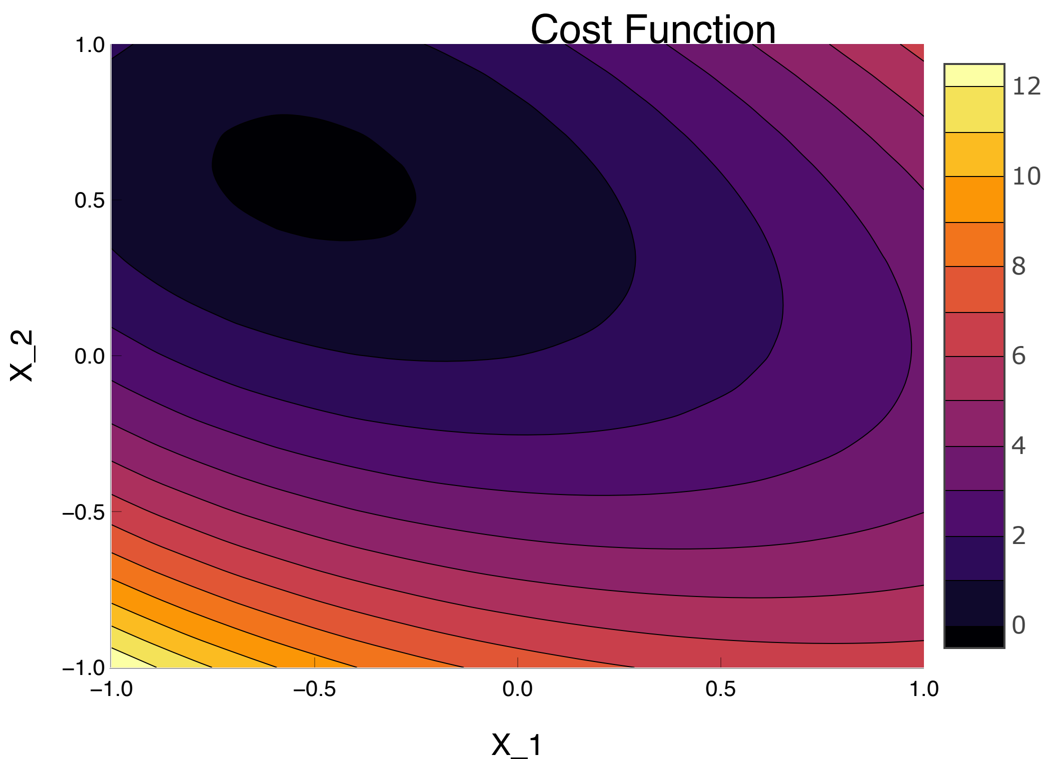
Out[6]: `Test.DefaultTestSet("2b", Any[], 1, false, false)`

Part (c): Newtons method in optimization (4 pt)

Now let's look at how we can use Newton's method in numerical optimization. Let's start by plotting a cost function $f(x)$, where $x \in \mathbb{R}^2$.

In [7]:
```
let
    Q = [1.65539  2.89376; 2.89376  6.51521];
    q = [2;-3]
    f(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)-> f([x1;x2]),title = "Cost Function",
            xlabel = "X_1", ylabel = "X_2",fill = true)
end
```

Out[7]:



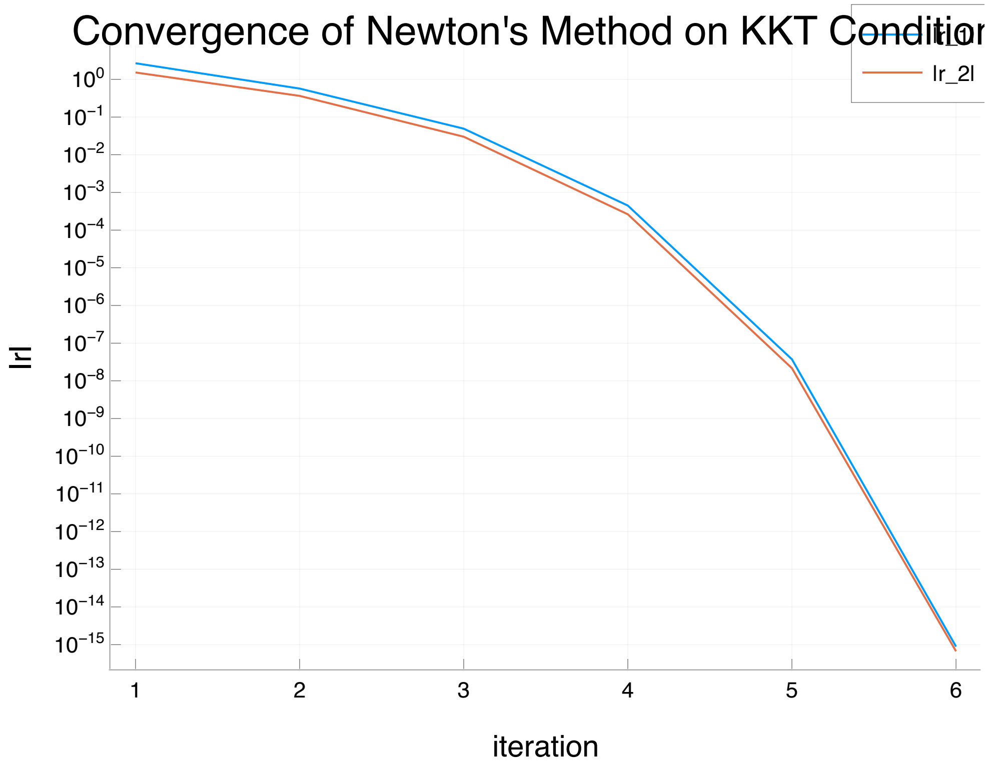
To find the minimum for this cost function $f(x)$, let's write the KKT conditions for optimality:

$$\nabla f(x) = 0 \qquad \text{stationarity},$$

which we see is just another rootfinding problem. We are now going to use Newton's method on the KKT conditions to find the x in which $\nabla f(x) = 0$.

In [8]:
```
@testset "2c" begin
    Q = [1.65539  2.89376; 2.89376  6.51521];
    q = [2;-3]
    f(x) = 0.5*x'*Q*x + q'*x + exp(-1.3*x[1] + 0.3*x[2]^2)

    function kkt_conditions(x)
        # TODO: return the stationarity condition for the cost function f (∇
        # hint: use forward diff
        return FD.gradient(dx -> f(dx), x)
    end

    residual_fx(_x) = kkt_conditions(_x)

    x0 = [-0.9512129986081451, 0.8061342694354091]
    X = newtons_method(x0, residual_fx; max_iters = 10)
    R = residual_fx.(X) # the . evaluates the function at each element of th

    Rp = [[abs(R[i][ii]) for i = 1:length(R)] for ii = 1:length(R[1])] # thi

    # tests
    @test norm(R[end])<1e-10;
```

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

end
```




```
Test Summary: | Pass  Total
2c            |    1      1
```

Out[8]: `Test.DefaultTestSet("2c", Any[], 1, false, false)`

Note on Newton's method for unconstrained optimization

To solve the above problem, we used Newton's method on the following equation:

$$\nabla f(x) = 0 \qquad \text{stationarity},$$

Which results in the following Newton steps:

$$\Delta x = -\left[\frac{\partial \nabla f(x)}{x}\right]^{-1} \nabla f(x_k).$$

The jacobian of the gradient of $f(x)$ is the same as the hessian of $f(x)$ (write this out and convince yourself). This means we can rewrite the Newton step as the equivalent expression:

$$\Delta x = -[\nabla^2 f(x)]^{-1} \nabla f(x_k)$$

What is the interpretation of this? Well, if we take a second order Taylor series of our cost function, and minimize this quadratic approximation of our cost function, we get the following optimization problem:

$$\min_{\Delta x} \qquad f(x_k) + [\nabla f(x_k)^T]\Delta x + \frac{1}{2}\Delta x^T[\nabla^2 f(x_k)]\Delta x$$

Where our optimality condition is the following:

$$\nabla f(x_k)^T + [\nabla^2 f(x_k)]\Delta x = 0$$

And we can solve for Δx with the following:

$$\Delta x = -[\nabla^2 f(x)]^{-1}\nabla f(x_k)$$

Which is our Newton step. This means that Newton's method on the stationary condition is the same as minimizing the quadratic approximation of the cost function at each iteration.

```
In [ ]:
```