In [ ]:
```julia
# 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();

# let's load LinearAlgebra in
using LinearAlgebra
using Test
```

```
  Activating project at `~/Desktop/2024Spring/CMU16745_OptimalControl/CMU16-745-Optimal-Control-HW/hw1`
Precompiling project...
  ✓ XSLT_jll
  ✓ Xorg_libxcb_jll
  ✓ OpenSSL
  ✓ Xorg_xcb_util_jll
  ✓ Xorg_libX11_jll
  ✓ Xorg_xcb_util_keysyms_jll
  ✓ Xorg_xcb_util_image_jll
  ✓ Xorg_xcb_util_wm_jll
  ✓ Xorg_libXext_jll
  ✓ Xorg_libXrender_jll
  ✓ Xorg_xcb_util_renderutil_jll
  ✓ Xorg_libxkbfile_jll
  ✓ Libglvnd_jll
  ✓ Xorg_libXrandr_jll
  ✓ Xorg_libXfixes_jll
  ✓ Xorg_libXinerama_jll
  ✓ Xorg_libXcursor_jll
  ✓ Xorg_libXi_jll
  ✓ Xorg_xkbcomp_jll
  ✓ Xorg_xcb_util_cursor_jll
  ✓ GLFW_jll
  ✓ Xorg_xkeyboard_config_jll
  ✓ xkbcommon_jll
  ✓ HTTP
  ✓ Vulkan_Loader_jll
  ? Cairo_jll
  ? Qt6Base_jll
  ✓ HarfBuzz_jll
  ✓ libass_jll
  ✓ FFMPEG_jll
  ✓ FFMPEG
  ✓ GR_jll
  ✓ GR
  ✓ Plots
  ✓ Plots → UnitfulExt
```

```
  33 dependencies successfully precompiled in 71 seconds. 130 already precom
piled.
  2 dependencies failed but may be precompilable after restarting julia
  2 dependencies had warnings during precompilation:
┌ Qt6Base_jll [c0090381-4147-56d7-9ebc-da0b1113ec56]
│  ┌ Warning: The call to compilecache failed to create a usable precompiled
cache file for Fontconfig_jll [a3f928ae-7b40-5064-980b-68af3947d34b]
│  │   exception = Required dependency Expat_jll [2e619515-83b5-522b-bb60-26
c02a35a201] failed to load from a cache file.
│  └ @ Base loading.jl:1818
└
┌ Cairo_jll [83423d85-b0ee-5818-9007-b63ccbeb887a]
│  ┌ Warning: The call to compilecache failed to create a usable precompiled
cache file for FreeType2_jll [d7e528f0-a631-5988-bf34-fe36492bcfd7]
│  │   exception = Required dependency Bzip2_jll [6e34b625-4abd-537c-b88f-47
1c36dfa7a0] failed to load from a cache file.
│  └ @ Base loading.jl:1818
└
```

Question 1: Differentiation in Julia (10 pts)

Julia has a fast and easy to use forward-mode automatic differentiation package called ForwardDiff.jl that we will make use of throughout this course. In general it is easy to use and very fast, but there are a few quirks that are detailed below. This notebook will start by walking through general usage for the following cases:

- functions with a single input
- functions with multiple inputs
- composite functions

as well as a guide on how to avoid the most common ForwardDiff.jl error caused by creating arrays inside the function being differentiated. First, let's look at the ForwardDiff.jl functions that we are going to use:

- `FD.derivative(f,x)` derivative of scalar or vector valued f wrt scalar x
- `FD.jacobian(f,x)` jacobian of vector valued f wrt vector x
- `FD.gradient(f,x)` gradient of scalar valued f wrt vector x
- `FD.hessian(f,x)` hessian of scalar valued f wrt vector x

Note on gradients:

For an arbitrary function $f(x) : \mathbb{R}^N \rightarrow \mathbb{R}^M$, the jacobian is the following:

$$\frac{\partial f(x)}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Now if we have a scalar valued function (like a cost function) $f(x) : \mathbb{R}^N \to \mathbb{R}$, the jacobian is the following row vector:

$$\frac{\partial f(x)}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \end{bmatrix}$$

The transpose of this jacobian for scalar valued functions is called the gradient:

$$\nabla f(x) = \left[\frac{\partial f(x)}{\partial x}\right]^T$$

TLDR:

- the jacobian of a scalar value function is a row vector
- the gradient is the transpose of this jacobian, making the gradient a column vector
- ForwardDiff.jl will give you an error if you try to take a jacobian of a scalar valued function, use the gradient function instead

Part (a): General usage (2 pts)

The API for functions with one input is detailed below:

In [ ]:
```
# NOTE: this block is a tutorial, you do not have to fill anything out.

#---------load the package-----------
# using ForwardDiff # this puts all exported functions into our namespace
# import ForwardDiff # this means we have to use ForwardDiff.<function name>
import ForwardDiff as FD # this let's us do FD.<function name>

function foo1(x)
    #scalar input, scalar output
    return sin(x)*cos(x)^2
end

function foo2(x)
    # vector input, scalar output
    return sin(x[1]) + cos(x[2])
end
function foo3(x)
```

```
    # vector input, vector output
    return [sin(x[1])*x[2];cos(x[2])*x[1]]
end


let # we just use this to avoid creating global variables

    # evaluate the derivative of foo1 at x1
    x1 = 5*randn();
    @show ∂foo1_∂x = FD.derivative(foo1, x1);

    # evaluate the gradient and hessian of foo2 at x2
    x2 = 5*randn(2);
    @show ∇foo2 = FD.gradient(foo2, x2);
    @show ∇²foo2 = FD.hessian(foo2, x2);

    # evluate the jacobian of foo3 at x2
    @show ∂foo3_∂x = FD.jacobian(foo3,x2);

end
```

```
∂foo1_∂x = FD.derivative(foo1, x1) = 0.2955143450569553
∇foo2 = FD.gradient(foo2, x2) = [0.9884304656377759, 0.7075456235888418]
∇²foo2 = FD.hessian(foo2, x2) = [-0.15167469993077098 0.0; 0.0 -0.7066676662
620677]
∂foo3_∂x = FD.jacobian(foo3, x2) = [-6.987416878362954 0.15167469993077098;
0.7066676662620677 0.10773256334998937]
```

Out[ ]:

```
2×2 Matrix{Float64}:
 -6.98742   0.151675
  0.706668  0.107733
```

In [ ]:

```
# here is our function of interest
function foo4(x)
    Q = diagm([1;2;3.0]) # this creates a diagonal matrix from a vector
    return 0.5*x'*Q*x/x[1] - log(x[1])*exp(x[2])^x[3]
end

function foo4_expansion(x)
    # TODO: this function should output the hessian H and gradient g of the

    # TODO: calculate the gradient of foo4 evaluated at x
    g = FD.gradient(foo4, x)

    # TODO: calculate the hessian of foo4 evaluated at x
    H = FD.hessian(foo4, x)

    return g, H
end
```

Out[ ]:

```
foo4_expansion (generic function with 1 method)
```

In [ ]:
```
@testset "1a" begin
    x = [.2;.4;.5]
    g,H = foo4_expansion(x)
    @test isapprox(g,[-18.98201379080085, 4.982885952667278, 8.286308762133
    @test norm(H -[164.2850689540042 -23.053506895400425 -39.94280551632033
                            -23.053506895400425 10.491442976333639 2.358926
                            -39.94280551632034 2.3589262864014673 15.314523
end
```

```
Test Summary: | Pass  Total  Time
1a            |    2      2  1.0s
```

Out[ ]:
```
Test.DefaultTestSet("1a", Any[], 2, false, false, true, 1.705675263052655e
9, 1.705675264092445e9, false)
```

Part (b): Derivatives for functions with multiple input arguments (2 pts)

In [ ]:
```
# NOTE: this block is a tutorial, you do not have to fill anything out.

# calculate derivatives for functions with multiple inputs
function dynamics(x,a,b,c)
    return [x[1]*a; b*c*x[2]*x[1]]
end

let
    x1 = randn(2)
    a = randn()
    b = randn()
    c = randn()

    # this evaluates the jacobian with respect to x, given a, b, and c
    A1 = FD.jacobian(dx -> dynamics(dx, a, b, c), x1)

    # it doesn't matter what we call the new variable
    A2 = FD.jacobian(_x -> dynamics(_x, a, b, c), x1)

    # alternatively we can do it like this using a closure
    dynamics_just_x(_x) = dynamics(_x, a, b, c)
    A3 = FD.jacobian(dynamics_just_x, x1)

    @test norm(A1 - A2) < 1e-13
    @test norm(A1 - A3) < 1e-13
end
```

Out[ ]: **Test Passed**

In [ ]:
```
function eulers(x,u,J)
```

```
    # dynamics when x is angular velocity and u is an input torque
    ẋ = J\(u - cross(x,J*x))
    return ẋ
end

function eulers_jacobians(x,u,J)
    # given x, u, and J, calculate the following two jacobians

    # TODO: fill in the following two jacobians

    # ∂ẋ/∂x
    A = FD.jacobian(_x -> eulers(_x,u,J), x)

    # ∂ẋ/∂u
    B = FD.jacobian(_u -> eulers(x,_u,J), u)

    return A, B
end
```

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

In [ ]:
```
@testset "1b" begin

    x = [.2;-7;.2]
    u = [.1;-.2;.343]
    J = diagm([1.03;4;3.45])

    A,B = eulers_jacobians(x,u,J)

    skew(v) = [0 -v[3] v[2]; v[3] 0 -v[1]; -v[2] v[1] 0]
    @test isapprox(A,-J\(skew(x)*J - skew(J*x)), atol = 1e-8)

    @test norm(B - inv(J)) < 1e-8

end
```

```
Test Summary: | Pass  Total  Time
1b            |    2      2  2.8s
```

Out[ ]: Test.DefaultTestSet("1b", Any[], 2, false, false, true, 1.705675404673835e9, 1.705675407477736e9, false)

Part (c): Derivatives of composite functions (1 pts)

In [ ]:
```
# NOTE: this block is a tutorial, you do not have to fill anything out.
function f(x)
    return x[1]*x[2]
end
function g(x)
```

```
    return [x[1]^2; x[2]^3]
end

let
    x1 = 2*randn(2)

    # using gradient of the composite function
    ∇f_1 = FD.gradient(dx -> f(g(dx)), x1)

    # using the chain rule
    J = FD.jacobian(g, x1)
    ∇f_2 = J'*FD.gradient(f, g(x1))

    @show norm(∇f_1 - ∇f_2)
end
```

```
norm(∇f_1 - ∇f_2) = 0.0
```

Out[ ]: 0.0

In [ ]:
```
function f2(x)
    return x*sin(x)/2
end
function g2(x)
    return cos(x)^2 - tan(x)^3
end

function composite_derivs(x)

    # TODO: return ∂y/∂x where y = g2(f2(x))
    # (hint: this is 1D input and 1D output, so it's ForwardDiff.derivative)
    return FD.derivative(_x -> g2(f2(_x)), x)
end
```

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

In [ ]:
```
@testset "1c" begin
    x = 1.34
    deriv = composite_derivs(x)

    @test isapprox(deriv,-2.390628273373545,atol = 1e-8)
end
```

```
Test Summary: | Pass  Total  Time
1c            |    1      1  0.1s
```

Out[ ]: Test.DefaultTestSet("1c", Any[], 1, false, false, true, 1.705675477356017e9, 1.705675477409722e9, false)

Part (d): Fixing the most common ForwardDiff error (2

pt)

First we will show an example of this error:

In [ ]:
```
# NOTE: this block is a tutorial, you do not have to fill anything out.
function f_zero_1(x)
    println("-------types of input x---------")
    @show typeof(x) # print out type of x
    @show eltype(x) # print out the element type of x


    xdot = zeros(length(x)) # this default creates zeros of type Float64
    println("-------types of output xdot---------")
    @show typeof(xdot)
    @show eltype(xdot)

    # these lines will error because i'm trying to put a ForwardDiff.dual
    # inside of a Vector{Float64}
    xdot[1] = x[1]*x[2]
    xdot[2] = x[2]^2

    return xdot
end

let
    # try and calculate the jacobian of f_zero_1 on x1
    x1 = randn(2)
    @info "this error is expected:"
    try
        FD.jacobian(f_zero_1,x1)
    catch e
        buf = IOBuffer()
        showerror(buf,e)
        message = String(take!(buf))
        Base.showerror(stdout,e)
    end
end
```

```
[ Info: this error is expected:
```

```
-------types of input x---------
typeof(x) = Vector{ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float6
4}, Float64, 2}}
eltype(x) = ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float64}, Flo
at64, 2}
-------types of output xdot---------
typeof(xdot) = Vector{Float64}
eltype(xdot) = Float64
MethodError: no method matching Float64(::ForwardDiff.Dual{ForwardDiff.Tag{t
ypeof(f_zero_1), Float64}, Float64, 2})

Closest candidates are:
  (::Type{T})(::Real, ::RoundingMode) where T<:AbstractFloat
   @ Base rounding.jl:207
  (::Type{T})(::T) where T<:Number
   @ Core boot.jl:792
  (::Type{T})(::AbstractChar) where T<:Union{AbstractChar, Number}
   @ Base char.jl:50
  ...
```

This is the most common ForwardDiff error that you will encounter. ForwardDiff works by pushing `ForwardDiff.Dual` variables through the function being differentiated. Normally this works without issue, but if you create a vector of `Float64` (like you would with `xdot = zeros(5)`, it is unable to fit the `ForwardDiff.Dual`'s in with the `Float64`'s. To get around this, you have two options:

Option 1

Our first option is just creating xdot directly, without creating an array of zeros to index into.

In [ ]:

```julia
# NOTE: this block is a tutorial, you do not have to fill anything out.
function f_zero_1(x)

    # let's create xdot directly, without first making a vector of zeros
    xdot = [x[1]*x[2], x[2]^2]

    # NOTE: the compiler figures out which type to make xdot, so when you ca
    # it's a Float64, and when it's being diffed, it's automatically promote

    println("-------types of input x---------")
    @show typeof(x) # print out type of x
    @show eltype(x) # print out the element type of x

    println("-------types of output xdot---------")
    @show typeof(xdot)
    @show eltype(xdot)
```

```
    return xdot
end

let
    # try and calculate the jacobian of f_zero_1 on x1
    x1 = randn(2)
    FD.jacobian(f_zero_1,x1) # this will work
end
```

```
-------types of input x---------
typeof(x) = Vector{ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float6
4}, Float64, 2}}
eltype(x) = ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float64}, Flo
at64, 2}
-------types of output xdot---------
typeof(xdot) = Vector{ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Flo
at64}, Float64, 2}}
eltype(xdot) = ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float64},
Float64, 2}
```

Out[ ]:
```
2×2 Matrix{Float64}:
 0.593995  -0.128867
 0.0        1.18799
```

Option 2

The second option is to create the array of zeros in a way that accounts for the input type. This can be done by replacing `zeros(length(x))` with `zeros(eltype(x),length(x))`. The first argument `eltype(x)` simply creates a vector of zeros that is the same type as the element type in vector x.

In [ ]:
```
# NOTE: this block is a tutorial, you do not have to fill anything out.
function f_zero_1(x)

    xdot = zeros(eltype(x), length(x))

    xdot[1] = x[1]*x[2]
    xdot[2] = x[2]^2

    println("-------types of input x---------")
    @show typeof(x) # print out type of x
    @show eltype(x) # print out the element type of x

    println("-------types of output xdot---------")
    @show typeof(xdot)
    @show eltype(xdot)

    return xdot
```

```
end

let
    # try and calculate the jacobian of f_zero_1 on x1
    x1 = randn(2)
    FD.jacobian(f_zero_1,x1) # this will fail!
end
```

```
-------types of input x---------
typeof(x) = Vector{ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float6
4}, Float64, 2}}
eltype(x) = ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float64}, Flo
at64, 2}
-------types of output xdot---------
typeof(xdot) = Vector{ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Flo
at64}, Float64, 2}}
eltype(xdot) = ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float64},
Float64, 2}
```

Out[ ]:
```
2×2 Matrix{Float64}:
 -2.16954   0.277068
 -0.0      -4.33909
```

Now you can show that you understand these two options by fixing two broken functions.

In [ ]:
```
# TODO: fix this error when trying to diff through this function
# hint: you can use promote_type(eltype(x),eltype(u)) to return the correct

function dynamics(x,u)
    # ẋ = zeros(length(x))
    # ẋ[1] = x[1]*sin(u[1])
    # ẋ[2] = x[2]*cos(u[2])
    ẋ = [x[1]*sin(u[1]); x[2]*cos(u[2])]
    return ẋ
end
```

Out[ ]:
```
dynamics (generic function with 2 methods)
```

In [ ]:
```
@testset "1d" begin
    x = [.1;.4]
    u = [.2;-.3]
    A = FD.jacobian(_x -> dynamics(_x,u),x)
    B = FD.jacobian(_u -> dynamics(x,_u),u)
    @test typeof(A) == Matrix{Float64}
    @test typeof(B) == Matrix{Float64}
end
```

```
Test Summary: | Pass  Total  Time
1d            |    2      2  0.5s
```

Out[ ]:
```
Test.DefaultTestSet("1d", Any[], 2, false, false, true, 1.70567574703161e9,
1.705675747501535e9, false)
```

Finite Difference Derivatives

If you ever have trouble working through a ForwardDiff error, you should always feel free to use the FiniteDiff.jl FiniteDiff.jl package instead. This computes derivatives through a finite difference method. This is slower and less accurate than ForwardDiff, but it will always work so long as the function works.

Before with ForwardDiff we had this:

- `FD.derivative(f,x)` derivative of scalar or vector valued f wrt scalar x
- `FD.jacobian(f,x)` jacobian of vector valued f wrt vector x
- `FD.gradient(f,x)` gradient of scalar valued f wrt vector x
- `FD.hessian(f,x)` hessian of scalar valued f wrt vector x

Now with FiniteDiff we have this:

- `FD2.finite_difference_derivative(f,x)` derivative of scalar or vector valued f wrt scalar x
- `FD2.finite_difference_jacobian(f,x)` jacobian of vector valued f wrt vector x
- `FD2.finite_difference_gradient(f,x)` gradient of scalar valued f wrt vector x
- `FD2.finite_difference_hessian(f,x)` hessian of scalar valued f wrt vector x

In [ ]:
```
# NOTE: this block is a tutorial, you do not have to fill anything out.

# load the package
import FiniteDiff as FD2

function foo1(x)
    #scalar input, scalar output
    return sin(x)*cos(x)^2
end

function foo2(x)
    # vector input, scalar output
    return sin(x[1]) + cos(x[2])
end
function foo3(x)
    # vector input, vector output
    return [sin(x[1])*x[2];cos(x[2])*x[1]]
```

```julia
end


let # we just use this to avoid creating global variables

    # evaluate the derivative of foo1 at x1
    x1 = 5*randn();
    @show ∂foo1_∂x = FD2.finite_difference_derivative(foo1, x1);

    # evaluate the gradient and hessian of foo2 at x2
    x2 = 5*randn(2);
    @show ∇foo2 = FD2.finite_difference_gradient(foo2, x2);
    @show ∇²foo2 = FD2.finite_difference_hessian(foo2, x2);

    # evluate the jacobian of foo3 at x2
    @show ∂foo3_∂x = FD2.finite_difference_jacobian(foo3,x2);

    @test norm(∂foo1_∂x - FD.derivative(foo1, x1)) < 1e-4
    @test norm(∇foo2 - FD.gradient(foo2, x2)) < 1e-4
    @test norm(∇²foo2 - FD.hessian(foo2, x2)) < 1e-4
    @test norm(∂foo3_∂x - FD.jacobian(foo3, x2)) < 1e-4


end
```

```
∂foo1_∂x = FD2.finite_difference_derivative(foo1, x1) = -0.603129667261091
∇foo2 = FD2.finite_difference_gradient(foo2, x2) = [-0.48903838176472963, -
0.9747960568683749]
∇²foo2 = FD2.finite_difference_hessian(foo2, x2) = [0.8722622617006804 0.0;
0.0 -0.2230978423327758]
∂foo3_∂x = FD2.finite_difference_jacobian(foo3, x2) = [-0.6581503169907323 -
0.8722622732606932; 0.2230978428169308 2.029314147719695]
```

Out[ ]: **Test Passed**

In [ ]:

In [ ]:
```
# 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
```

```
  Activating project at `~/Desktop/2024Spring/CMU16745_OptimalControl/CMU16-745-Optimal-Control-HW/hw1`
Precompiling project...
  ✓ OpenSpecFun_jll
  ✓ Cairo_jll
  ✓ Qt6Base_jll
  ✓ HarfBuzz_jll
  ✓ libass_jll
  ✓ SpecialFunctions
  ✓ DiffRules
  ✓ FFMPEG_jll
  ✓ ColorVectorSpace → SpecialFunctionsExt
  ✓ FFMPEG
  ✓ GR_jll
  ✓ ForwardDiff
  ✓ GR
  ✓ ColorSchemes
  ✓ PlotUtils
  ✓ PlotThemes
  ✓ RecipesPipeline
  ✓ Plots
  ✓ Plots → UnitfulExt
  19 dependencies successfully precompiled in 59 seconds. 146 already precompiled.
[ Info: Precompiling IJuliaExt [2f4121a4-3b3a-5ce6-9c5e-1f2673ce168a]
```

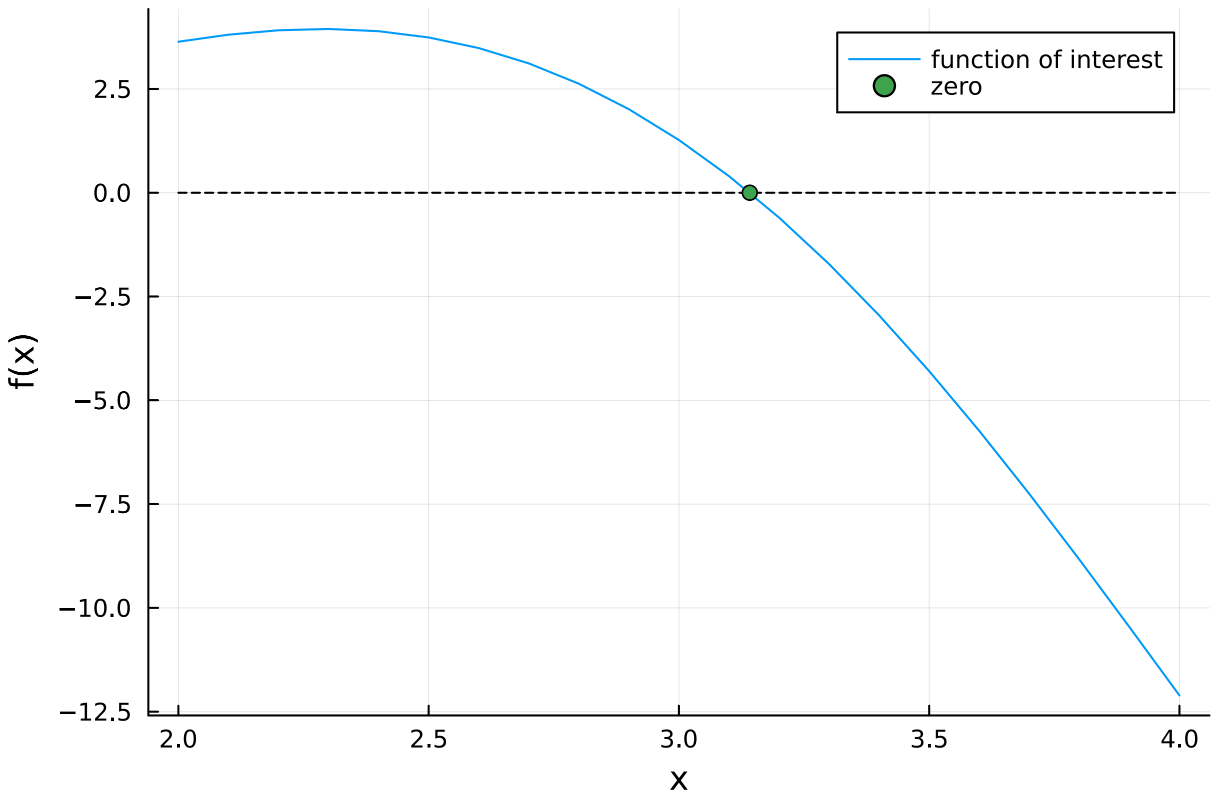
Q2: Newton's Method (20 pts)

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 [ ]:
```
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[ ]:



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.

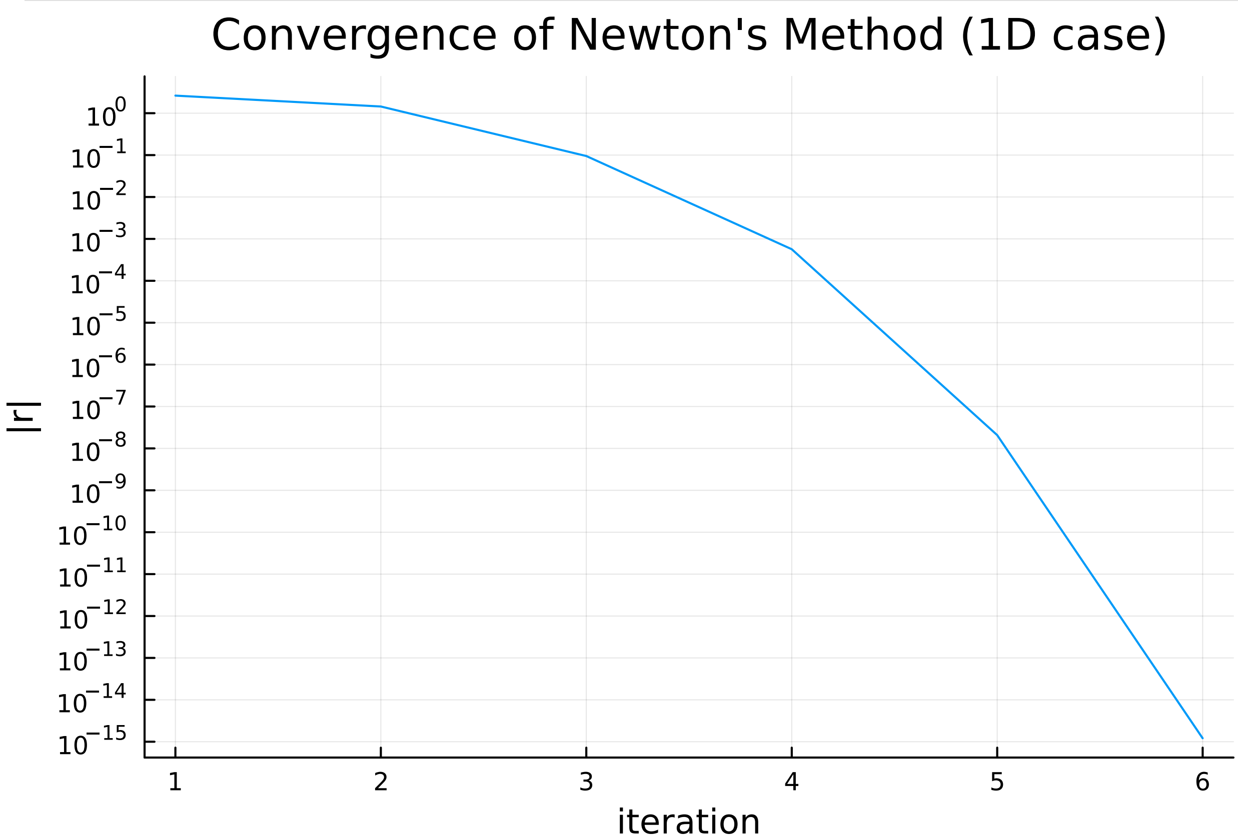
In [ ]:

```
"""
    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 = -residual_function(X[i]) / FD.derivative(residual_function,X[i]
        X[i+1] = X[i] + Δx

        # return the trimmed X[1:i] after you converge
        if abs(residual_function(X[i])) < 1e-10
            return X[1:i]
        end
    end
    error("Newton did not converge")
```

```
end
```

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

In [ ]:
```
@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  Time
2a            |    1      1  0.7s
```

Out[ ]: Test.DefaultTestSet("2a", Any[], 1, false, false, true, 1.705676686816917e9, 1.705676687555239e9, false)

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

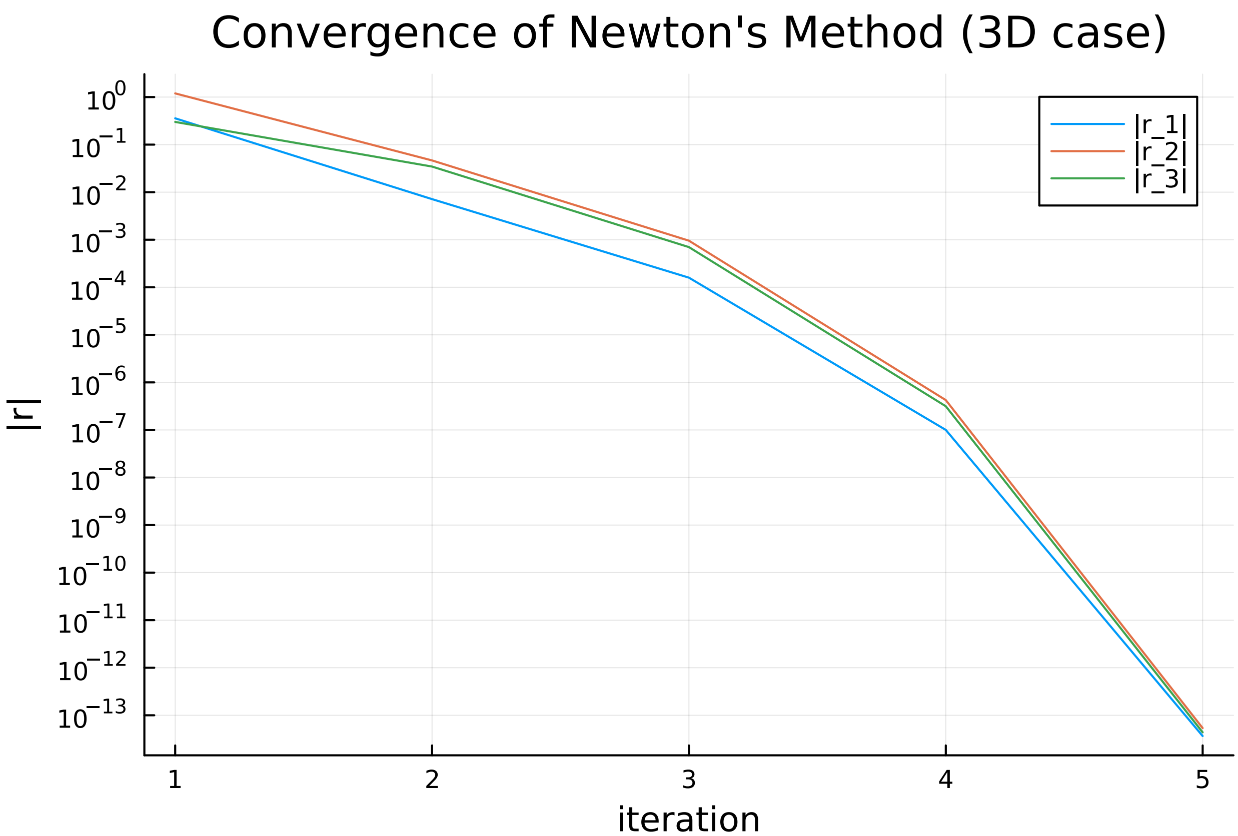
We are now going to use Newton's method to solve for the zero of a multivariate function.

In [ ]:
```
"""
    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])) < 1e-10
            return X[1:i]
        end

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

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

In [ ]:
```
@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  Time
2b            |    1      1  0.4s
```

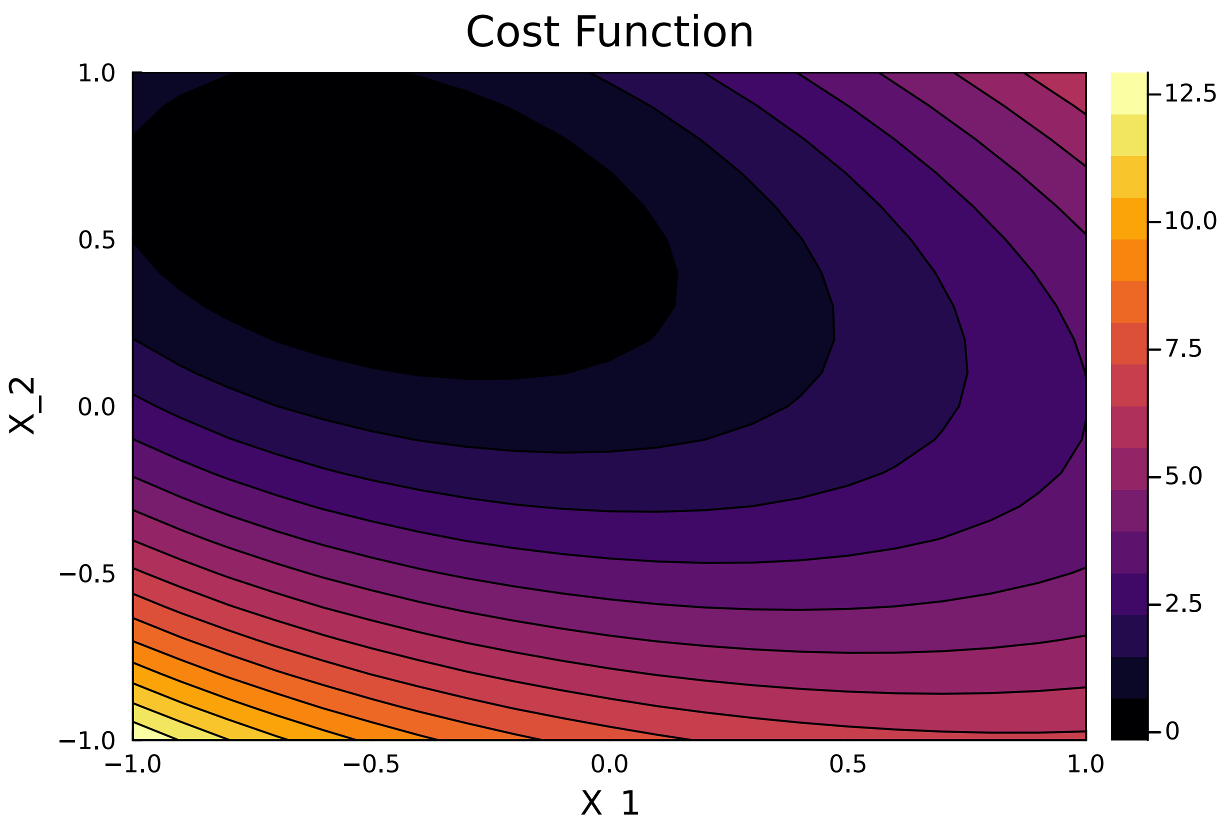
Out[ ]: `Test.DefaultTestSet("2b", Any[], 1, false, false, true, 1.705676896938623e9, 1.705676897360055e9, false)`

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 [ ]:
```
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[ ]:



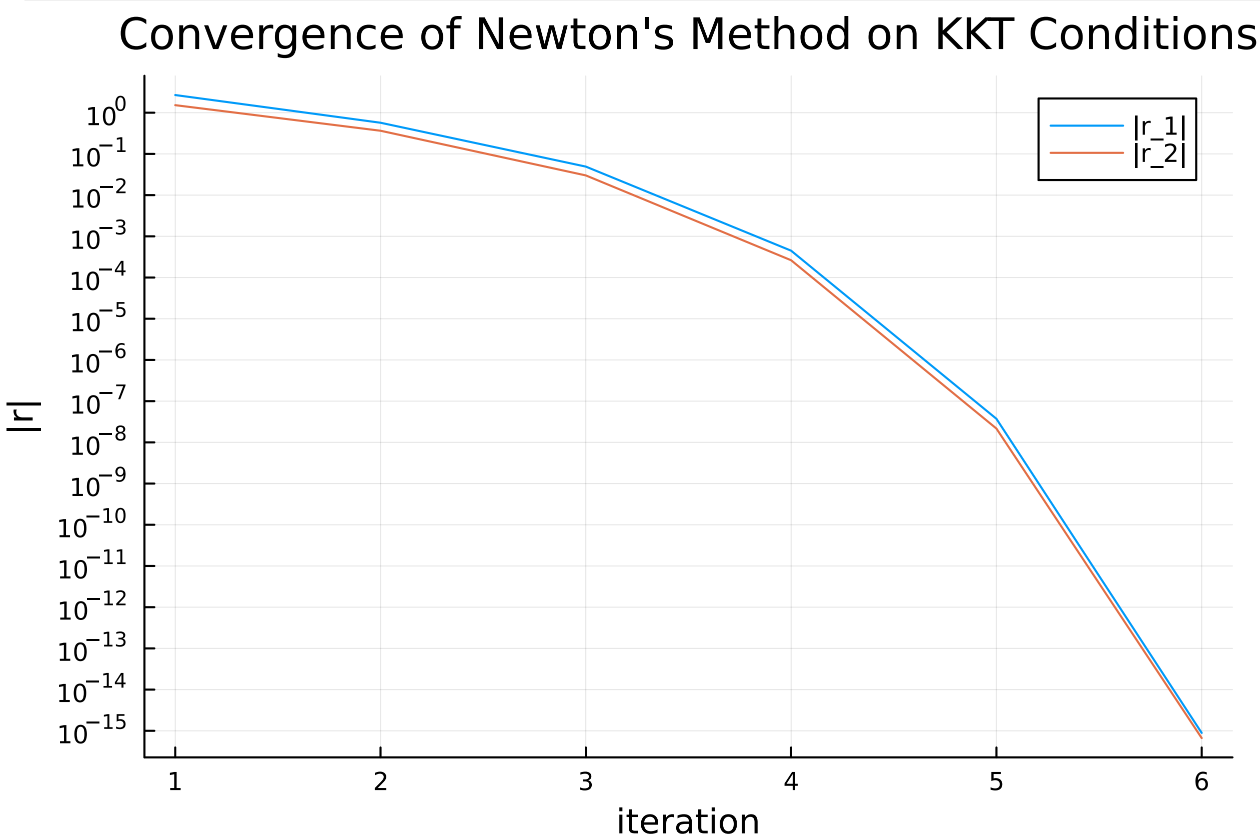
To find the minimum for this cost function $f(x)$, let's write the KKT conditions for optimality:

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

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 [ ]:
```
@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(f,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  Time
2c            |    1      1  0.9s
```

Out[ ]:
```
Test.DefaultTestSet("2c", Any[], 1, false, false, true, 1.7056770355387e9,
1.70567703646328e9, 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.