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/16-745 OCRL/16745-Optimal-Control-and-RL/HW0_S24-main`
```

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) $f(x) : \mathbb{R}^N \rightarrow \mathbb{R}$, 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 [ ]:
```julia
# NOTE: this block is a tutorial, you do not have to fill anything out.

# Derivative: the derivative of a scalar or vector-valued function with respect to a scalar

# Jacobian: the matrix of all first-order partial derivatives of a vector-valued function with respect to a vector

# Hessian: the matrix of all second-order partial derivatives of a scalar-valued function with respect to a vector
# Gradient: the vector of first-order partial derivatives of a scalar-valued function with respect to a vector


#---------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) = 0.6182826960085401
∇foo2 = FD.gradient(foo2, x2) = [0.9994908184848428, 0.3070853398657475]
∇²foo2 = FD.hessian(foo2, x2) = [-0.0319077383168893 0.0; 0.0 0.9516819815671296]
∂foo3_∂x = FD.jacobian(foo3, x2) = [-2.8280230709770855 0.0319077383168893; -0.9516819815671296 1.939274157553389]
```

Out[ ]:
```
2×2 Matrix{Float64}:
 -2.82802   0.0319077
 -0.951682  1.93927
```

In [ ]:
```julia
# 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 function foo4

    # TODO: calculate the gradient of foo4 evaluated at x
    g = zeros(length(x))
    @show ∇foo4 = FD.gradient(foo4, x);
    g = ∇foo4

    # TODO: calculate the hessian of foo4 evaluated at x
    H = zeros(length(x),length(x))
    @show ∇²foo4 = FD.hessian(foo4, x);
    H = ∇²foo4
    return g, H
end
```

Out[ ]:
```
foo4_expansion (generic function with 1 method)
```

In [ ]:
```julia
@testset "1a" begin
    x = [.2;.4;.5]
    g,H = foo4_expansion(x)
    @test isapprox(g,[-18.98201379080085, 4.982885952667278, 8.286308762133823],atol = 1e-8)
    @test norm(H -[164.2850689540042 -23.053506895400425 -39.942805516320334;
                         -23.053506895400425 10.491442976333639 2.3589262864014673;
                         -39.94280551632034 2.3589262864014673 15.314523504853529]) < 1e-8
end
```

```
∇foo4 = FD.gradient(foo4, x) = [-18.98201379080085, 4.982885952667278, 8.286308762133823]
∇²foo4 = FD.hessian(foo4, x) = [164.2850689540042 -23.053506895400425 -39.942805516320334; -23.053506895400425 10.491442976333639 2.3589262864014673; -39.94280551632034 2.3589262864014673 15.314523504853529]
Test Summary: | Pass  Total  Time
1a            |    2      2  0.1s
```

Out[ ]:
```
Test.DefaultTestSet("1a", Any[], 2, false, false, true, 1.722924719444092e9, 1.722924719525836e9, false, "In[10]")
```

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

In [ ]:
```julia
# 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 [ ]:
```julia
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 = zeros(3,3)
    A = FD.jacobian(_x -> eulers(_x, u, J), x)

    # ∂ẋ/∂u
    B = zeros(3,3)
    B = FD.jacobian(_u -> eulers(x, _u, J), u)

    return A, B
end
```

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

In [ ]:
```julia
@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  2.6s
```

Out[ ]:
```
Test.DefaultTestSet("1b", Any[], 2, false, false, true, 1.722924876426499e9, 1.722924879037479e9, false, "In[13]")
```

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

In [ ]:
```julia
# 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)
    @show x1;

    # 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
```

```
x1 = [1.9269792679885727, 2.1578483151147028]
norm(∇f_1 - ∇f_2) = 0.0
```

Out[ ]:
```
0.0
```

In [ ]:
```julia
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)

    ∂y_∂x = FD.derivative(dx -> g2(f2(dx)), x)

    return ∂y_∂x
end
```

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

In [ ]:
```julia
@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  0.0s
```

Out[ ]:
```
Test.DefaultTestSet("1c", Any[], 1, false, false, true, 1.722925239472583e9, 1.722925239511638e9, false, "In[26]")
```

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

First we will show an example of this error:

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

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

    # these lines will error because i'm trying to put a ForwardDiff.dual
    # inside of a Vector{Float64}
    println("Following Error caused by 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
```

```
-------types of input x---------
typeof(x) = Vector{ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float64}, Float64, 2}}
eltype(x) = ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float64}, Float64, 2}

-------types of output xdot---------
typeof(xdot) = Vector{Float64}
eltype(xdot) = Float64

Following Error caused by trying to put a ForwardDiff.Dual inside of a Vector{Float64}
MethodError: no method matching Float64(::ForwardDiff.Dual{ForwardDiff.Tag{typeof(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
  Float64(::IrrationalConstants.Invsqrt2π)
   @ IrrationalConstants ~/.julia/packages/IrrationalConstants/vp5v4/src/macro.jl:112
  ...
[ Info: this error is expected:
```

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 call the function normally
        # it's a Float64, and when it's being diffed, it's automatically promoted to a ForwardDiff.Dual type

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

        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), Float64}, Float64, 2}}
eltype(x) = ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float64}, Float64, 2}

-------types of output xdot---------
typeof(xdot) = Vector{ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float64}, Float64, 2}}
eltype(xdot) = ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float64}, Float64, 2}
```

Out[ ]:
```
2×2 Matrix{Float64}:
 -0.00737925   0.653191
 -0.0         -0.0147585
```

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 [ ]:
```julia
# 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()

    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), Float64}, Float64, 2}}
eltype(x) = ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float64}, Float64, 2}

-------types of output xdot---------
typeof(xdot) = Vector{ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float64}, Float64, 2}}
eltype(xdot) = ForwardDiff.Dual{ForwardDiff.Tag{typeof(f_zero_1), Float64}, Float64, 2}
```

Out[ ]:
```
2×2 Matrix{Float64}:
 0.458508  -0.738165
 0.0        0.917016
```

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

In [ ]:
```julia
# 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 type if either x or u is a ForwardDiff.Dual (option 1)

function dynamics(x, u)

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

        xdot[1] = x[1] * sin(u[1])
        xdot[2] = x[2] * cos(u[2])
        return xdot
end
```

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

In [ ]:
```julia
@testset "1d" begin
        x = [0.1; 0.4]
        u = [0.2; -0.3]
        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  0.2s
```

Out[ ]:
```
Test.DefaultTestSet("1d", Any[], 2, false, false, true, 1.722925577210449e9, 1.722925577414394e9, false, "In[51]")
```

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 [ ]:
```julia
# 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]]
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) = 0.3026746768191607
∇foo2 = FD2.finite_difference_gradient(foo2, x2) = [0.9105562858455398, 0.9250807888907285]
∇²foo2 = FD2.finite_difference_hessian(foo2, x2) = [-0.4133851230144501 3.6508587537099827e-10; 3.6508587537099827e-10 -0.37977035798508685]
∂foo3_∂x = FD2.finite_difference_jacobian(foo3, x2) = [4.645600825548172 0.4133851127806077; 0.3797703683376312 0.39424035083144604]
```

Out[ ]:
```
Test Passed
```