We define a data structure for dual numbers. The following code is poor Julia style (as it fixes data types unnecessarily), but it is more transparent. This part is modeled after the notes by David Sanders at https://mitmath.github.io/18337/lecture8/automatic_differentiation (https://mitmath.github.io/18337/lecture8/automatic_differentiation)

In [1]:

```julia
struct Dual
    val::Float64
    deriv::Float64
end
```

This is to create the variable \(x\). It’s derivative in \(x\) is 1

In [2]:

```julia
Var(x::Number) = Dual(x, 1.0)
```

Out[2]:

Var (generic function with 1 method)

Now, we overload the fundamental operations. For simplicity, I will only overload the ones we need for this example (and maybe a few others).

In [3]:

```julia
import Base: +, *, -, /, exp
```
In 

```python
+(f::Dual, g::Dual) = Dual(f.val + g.val, f.deriv + g.deriv)
+(f::Dual, x::Number) = Dual(f.val + x, f.deriv)
+(x::Number, f::Dual) = f + x
-(f::Dual, g::Dual) = Dual(f.val - g.val, f.deriv - g.deriv)
exp(f::Dual) = Dual(exp(f.val), exp(f.val)*f.deriv)
*(f::Dual, g::Dual) = Dual(f.val * g.val, f.val*g.deriv + f.deriv*g.val)
*(x::Number, f::Dual) = Dual(x * f.val, x* f.deriv)
/(f::Dual, g::Dual) = Dual(f.val / g.val, (f.deriv*g.val - f.val*g.deriv) / (g.val)^2)
```

Out[4]:
/ (generic function with 116 methods)

This is the function $2x$, evaluated at $x = 1$. Afterwards we define $3x$.

In [5]:

```python
x2 = 2*Var(1.0)
```

Out[5]:

Dual(2.0, 2.0)

In [6]:

```python
x3 = 3*Var(1.0)
```

Out[6]:

Dual(3.0, 3.0)

This should be $6x^2$, so its derivative should be $12x$. Recall that we are evaluating at $x = 1$.

In [8]:

```python
x2*x3
```

Out[8]:

Dual(6.0, 12.0)

Let's try the expression from class.
Here is the evaluation at \( a=2.0 \), both the value and the gradient.

To check if that gradient is correct, we can test by using a small \( \delta \). Following the analysis in the recommended reading, and the next section of this notebook, we try \( \delta \) around the square root of machine precision.

That is very close. Let's see in how many digits they agree.

You may be wondering which is more accurate: the code or the finite difference?

To check, we will use high precision floats.
In [12]:

A = BigFloat("2")
Del = BigFloat("1e-30")
(F(A+Del) - F(A)) / Del

Out[12]:
9.236320123663312784038034325734461510685622094702893119747800004273028992111742

In [13]:

(F(A+Del) - F(A)) / Del - dF.deriv

Out[13]:
1.107628385827214206421879469750952893119747800004273028992111741961458733212561e-15

We see that the forward differentiation gives the right answer.

**Finite Differencing**

The following is a demonstration of how the choice of δ effects the accuracy of derivative estimations.

Note that we can find out the machine precision:

In [21]:

u = eps(1.0)

Out[21]:
2.220446049250313e-16

In [22]:

1 - u

Out[22]:
0.9999999999999998

In [29]:

1 - u / 4 == 1

Out[29]:
true
Let's try a very simple function, whose derivative we know.

In [30]:

```python
f(x) = x^2 / 2
```

Out[30]:

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

In [31]:

```python
f(2)
```

Out[31]:

```
2.0
```

The derivative is just $x$ itself.

Let's see what differencing gives.

In [33]:

```python
δ = 1e-8
```

Out[33]:

```
1.0e-8
```

In [36]:

```python
(f(1+δ) - f(1)) / δ
```

Out[36]:

```
0.999999993922529
```

We now evaluate $f$ at an arbitrary $x$, and loop over various choices for $\delta$. This will enable us to see how the choice of $\delta$ impacts accuracy.

In [39]:

```python
x = 1.0
```

Out[39]:

```
1.0
```
In [40]:

```python
for i in 1:16:
    δ = 10.0^(-i)
    err = 1 - (f(x+δ) - f(x)) / δ
    println("At δ = $δ error is $(err)")
end
```

At δ = 0.1 error is -0.05000000000000093
At δ = 0.01 error is -0.00500000000003375
At δ = 0.001 error is -0.000499999998487326
At δ = 0.0001 error is -4.999999958599233e-5
At δ = 1.0e-5 error is -5.000006964905879e-6
At δ = 1.0e-6 error is -4.999621836532242e-7
At δ = 1.0e-7 error is -5.0543903284960834e-8
At δ = 1.0e-8 error is 6.07747097092215e-9
At δ = 1.0e-9 error is -8.274037099909037e-8
At δ = 1.0e-10 error is -8.274037099909037e-8
At δ = 1.0e-11 error is -8.274037099909037e-8
At δ = 1.0e-12 error is -8.890058234101161e-5
At δ = 1.0e-13 error is 0.0007992778373591136
At δ = 1.0e-14 error is 0.0007992778373591136
At δ = 1.0e-15 error is -0.11022302462515654
At δ = 1.0e-16 error is 1.0

So, we should choose δ = 1e − 8, which is the square root of machine precision.

Let's repeat with a better formula.

In [43]:

```python
δ = 1e-8
(f(x+δ) - f(x-δ)) / (2δ)
```

Out[43]:

0.999999996980866
In [44]:

```python
for i in 1:16:
    δ = 10.0^(-i)
    err = 1 - (f(x+δ) - f(x-δ)) / (2δ)
    println("At δ = $δ error is $(err)")
end
```

That was too good, because the function is quadratic. Let's try another for which we know the derivative, and a more interesting $x$.

In [46]:

```python
f2(x) = exp(x)
```

Out[46]:

f2 (generic function with 1 method)
In [48]:

```python
x = 3.14
for i in 1:16
    δ = 10.0^(-i)
    err = f2(x) - (f2(x+δ) - f2(x-δ)) / (2δ)
    println("At δ = $(δ) error is $(err)")
end
```

At δ = 0.1 error is -0.03852570257168253
At δ = 0.01 error is -0.00038506637248048037
At δ = 0.001 error is -3.8564976989619026e-8
At δ = 1.0e-5 error is -5.864642105279927e-10
At δ = 1.0e-6 error is -1.8299175508218468e-9
At δ = 1.0e-7 error is 3.9026289755383914e-8
At δ = 1.0e-8 error is 1.2784413172539644e-7
At δ = 1.0e-9 error is -2.18141975494929e-6
At δ = 1.0e-10 error is 1.3712939193055718e-6
At δ = 1.0e-11 error is -0.00014073725323271447
At δ = 1.0e-12 error is -0.002982908196273115
At δ = 1.0e-13 error is 0.011227946518928889
At δ = 1.0e-14 error is -0.3440434213611212
At δ = 1.0e-15 error is 1.7875847859191794
At δ = 1.0e-16 error is 23.103866858722185

So, the best choice of δ is around 1e-5, or the cube root of machine precision.

In [ ]: