Forward mode automatic differentiation

UCSD CSE 291 Differentiable Programming

Tzu-Mao Li

Defining derivatives

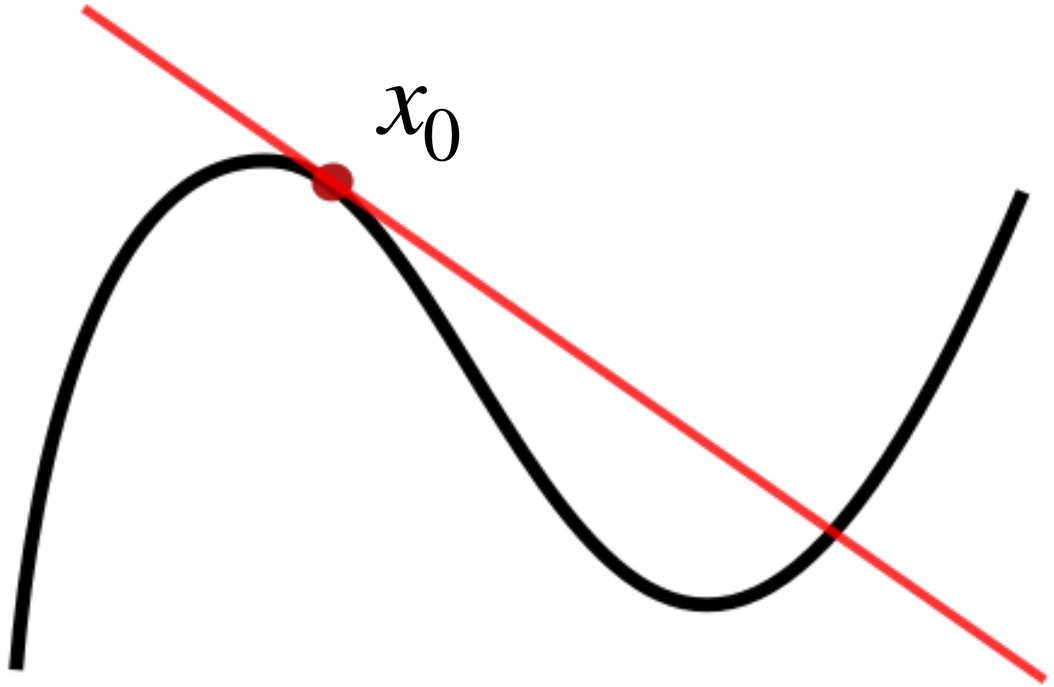
What is a derivative?

- def. 1: instantaneous rate of change (=slope)

$$f : \mathbb{R} \to \mathbb{R}$$

$$f'(x_0) = \lim_{dx \to 0} \frac{f(x_0 + dx) - f(x_0)}{dx}$$

how do we generalize this notion to higher-dimensional space?



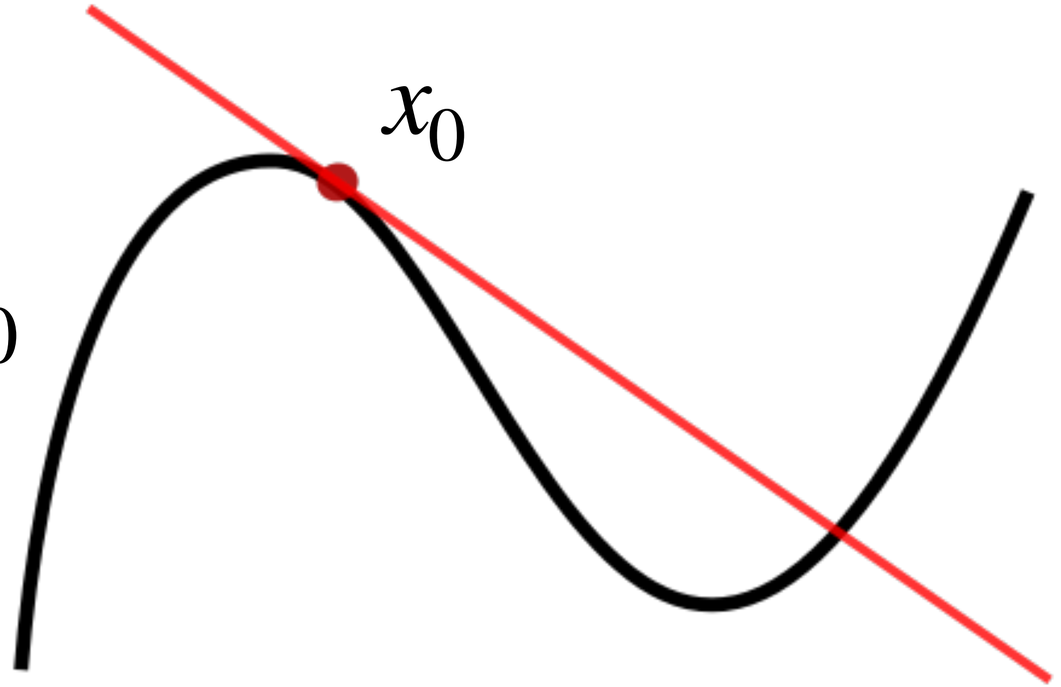
What is a derivative?

- def. 2: the *closest linear approximation*

$$f : \mathbb{R} \to \mathbb{R}$$

$$f(x_0 + dx) \approx f(x_0) + Df(x_0)(dx)$$

$$\lim_{dx \to 0} \frac{f(x_0 + dx) - f(x_0) - Df(x_0)(dx)}{dx} = 0$$



Spivak, *"Calculus on Manifolds"*

What is a derivative?

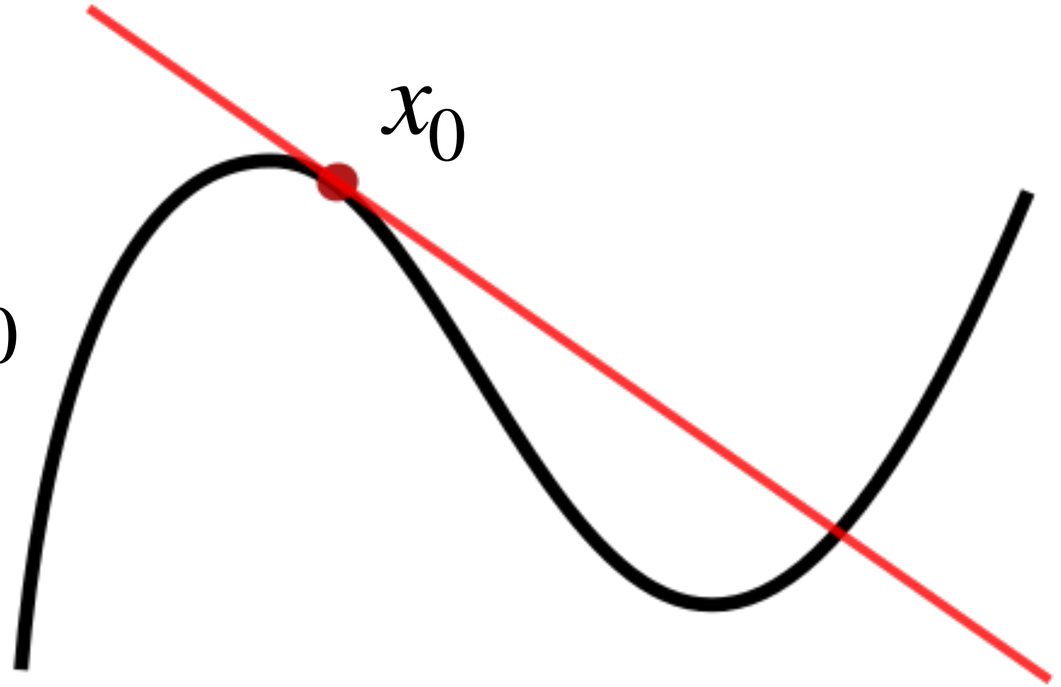
- def. 2: the *closest linear approximation*

$$f : \mathbb{R} \to \mathbb{R}$$

linear over dx

$$f(x_0 + dx) \approx f(x_0) + \boxed{Df(x_0)(dx)}$$

$$\lim_{dx \to 0} \frac{f(x_0 + dx) - f(x_0) - Df(x_0)(dx)}{dx} = 0$$



Spivak, *"Calculus on Manifolds"*

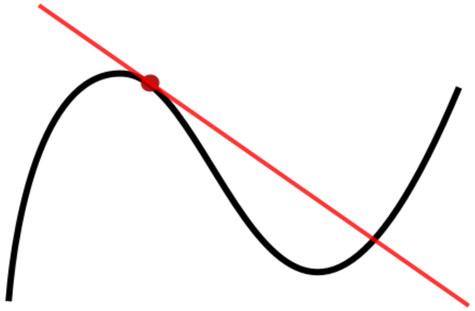
Differentiation:
generate best linear approximation at x0

$$f : \mathbb{R} \to \mathbb{R}$$

$$f(x_0 + dx) \approx f(x_0) + Df(x_0)(dx)$$

$$Df(x_0) : \mathbb{R} \to \mathbb{R} \text{ (linear)}$$

$$Df : \mathbb{R} \to (\mathbb{R} \to \mathbb{R})$$



Differentiation:
generate best linear approximation at x0

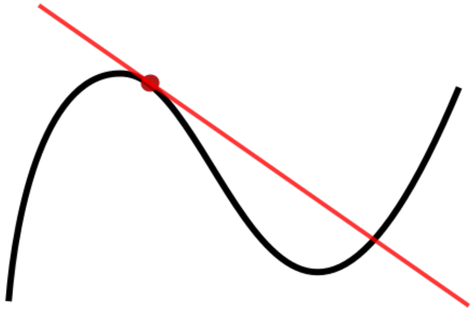
$$f : \mathbb{R} \to \mathbb{R}$$

$$f(x_0 + dx) \approx f(x_0) + Df(x_0)(dx)$$

$$Df(x_0) : \mathbb{R} \to \mathbb{R} \text{ (linear)}$$

$$f'(x_0) = Df(x_0)(1)$$

$$Df : \mathbb{R} \to (\mathbb{R} \to \mathbb{R})$$



Naturally generalize to multi-dimensional case

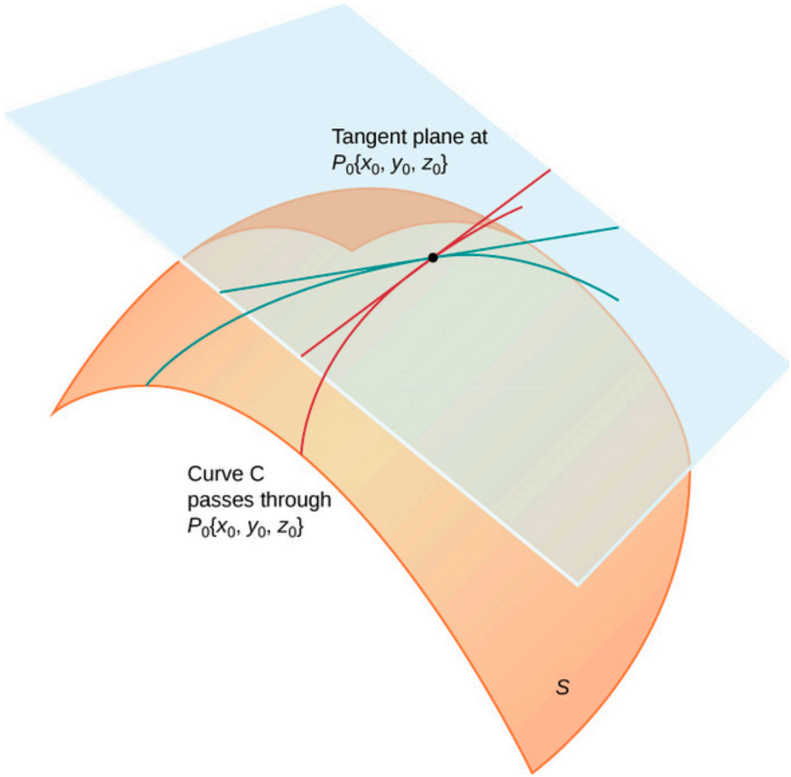
$$f : \mathbb{R}^n \to \mathbb{R}^m$$

linear over dx

$$f(x_0 + dx) \approx f(x_0) + Df(x_0)(dx)$$

$$\lim_{dx \to 0} \frac{\| f(x_0 + dx) - f(x_0) - Df(x_0)(dx) \|}{\| dx \|} = 0$$

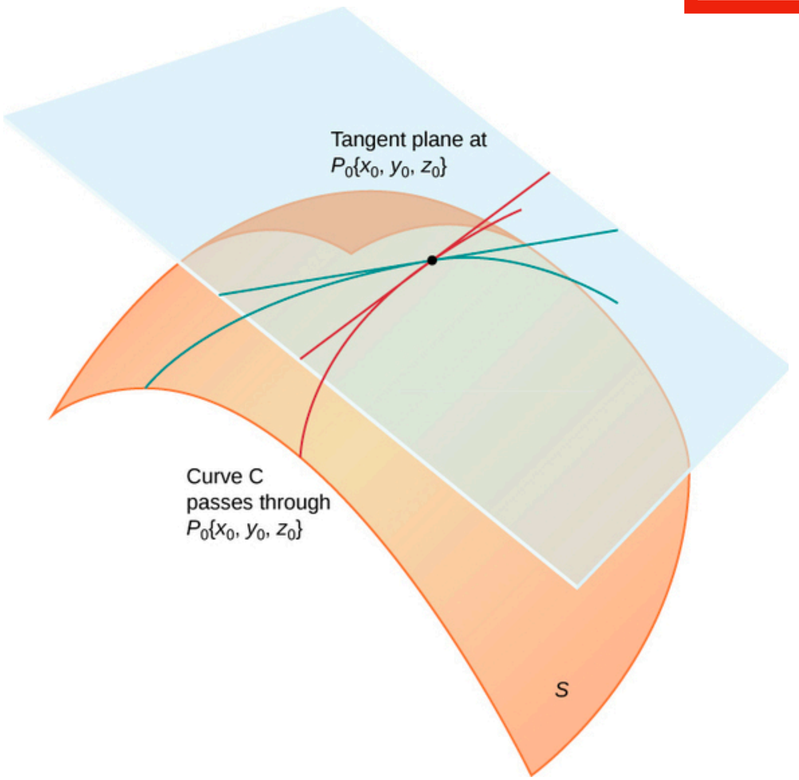
existence of limit <=> differentiable



Naturally generalize to multi-dimensional case

linear over dx

$$f(x_0 + dx) \approx f(x_0) + \boxed{Df(x_0)(dx)}$$



$$f : \mathbb{R}^n \to \mathbb{R}^m$$

$Df(x_0)$ = Jacobian

$Df(x_0)(dx)$ = total derivative

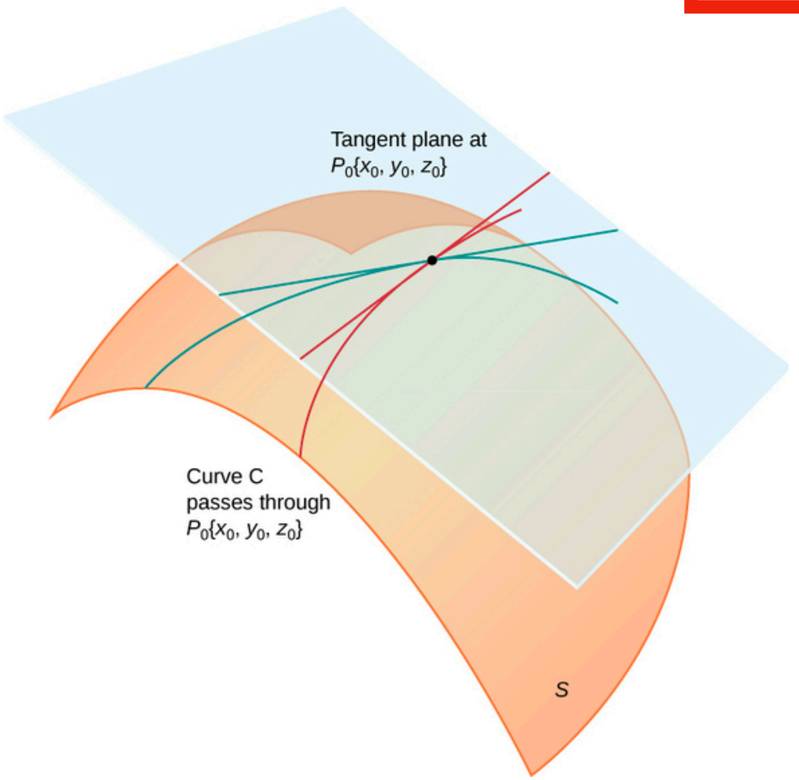
$$Df(x_0) : \mathbb{R}^n \to \mathbb{R}^m \text{ (linear)}$$

$$Df : \mathbb{R}^n \to \left(\mathbb{R}^n \to \mathbb{R}^m\right)$$

Naturally generalize to multi-dimensional case

linear over dx

$$f(x_0 + dx) \approx f(x_0) + \boxed{Df(x_0)(dx)}$$



$$f : \mathbb{R}^n \to \mathbb{R}^m$$

$$Df(x_0) = \text{Jacobian}$$

$$Df(x_0)(dx) = \text{total derivative}$$

$$Df(x_0) : \mathbb{R}^n \to \mathbb{R}^m \text{ (linear)}$$

$$Df : \mathbb{R}^n \to \left(\mathbb{R}^n \to \mathbb{R}^m\right)$$

total derivative = Jacobian vector product = “forward mode”

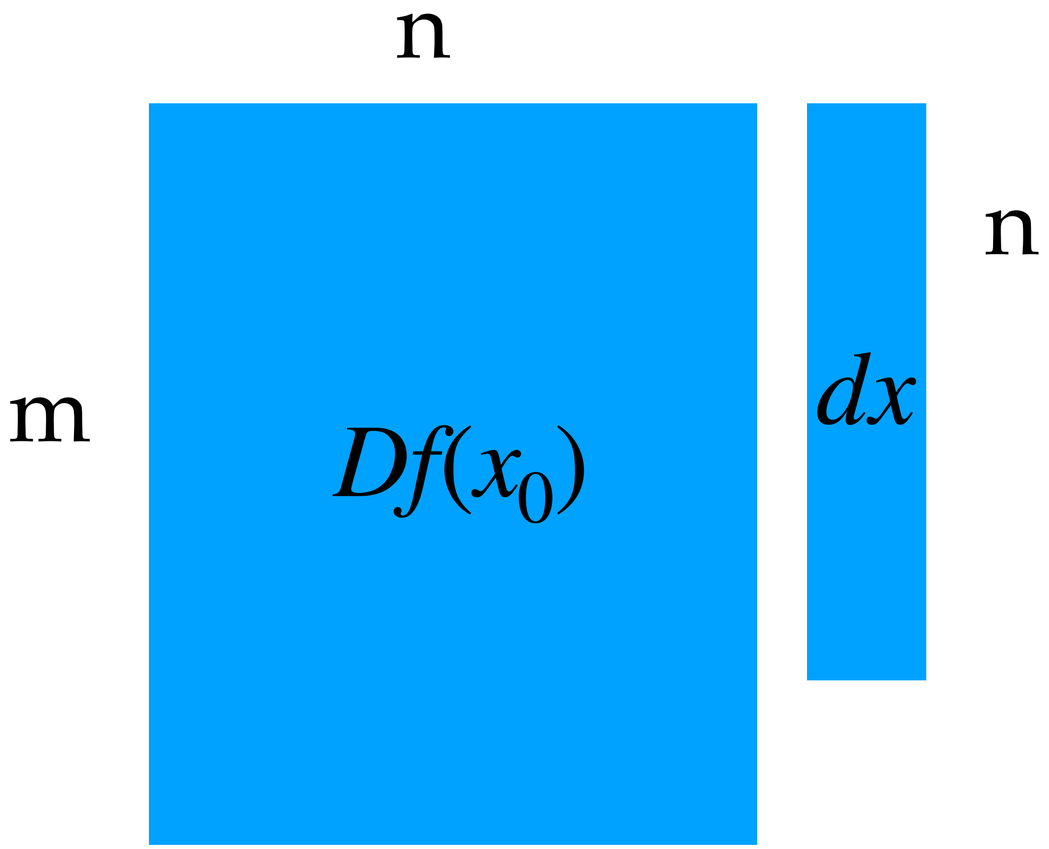
Total derivative = Jacobian-vector product

$f : \mathbb{R}^n \to \mathbb{R}^m$

$Df(x_0)$ = Jacobian

$Df(x_0) : \mathbb{R}^n \to \mathbb{R}^m$ (linear)

$Df(x_0)(dx)$ = total derivative

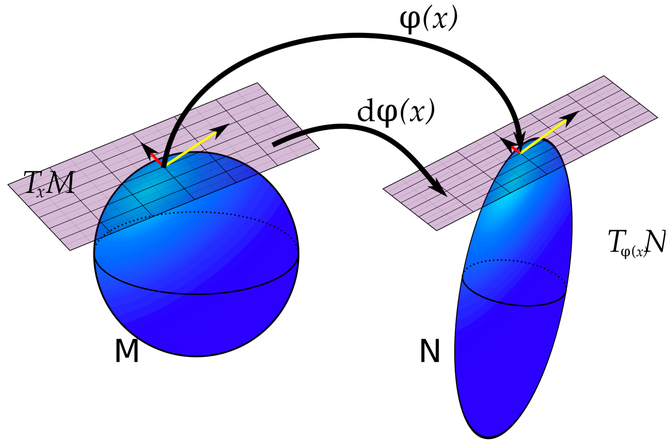


Chain rule = composition of total derivatives

can be seen as multiplying Jacobians, except we don't actually instantiate them

```
f(x, y):
  a = g(x, y)
  b = h(a)
  return b
```

```
Df(x, dx, y, dy):
  a, da = Dg(x, dx, y, dy)
  b, db = Dh(a, da)
  return b, db
```



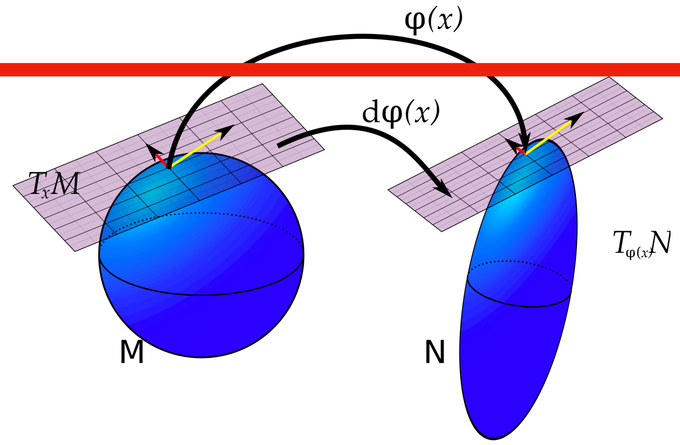
Chain rule = composition of total derivatives

can be seen as multiplying Jacobians, except we don't actually instantiate them

"forward-mode automatic differentiation"

```
f(x, y):
  a = g(x, y)
  b = h(a)
  return b
```

```
Df(x, dx, y, dy):
  a, da = Dg(x, dx, y, dy)
  b, db = Dh(a, da)
  return b, db
```



Let's see some examples

```
f(x, y):
    a = x + y
    b = -a
    return b
```

Let's see some examples

```
f(x, y):
   a = x + y
   b = -a
   return b
```

```
Df(x, y, dx, dy):
```

Let's see some examples

```
f(x, y):
    a = x + y
    b = -a
    return b
```

```
Df(x, y, dx, dy):
    a = x + y
    da = dx + dy
```

one way to see this:

let dx $= \frac{dx}{dt}$, dy $= \frac{dy}{dt}$, then:

$$\frac{da}{dt} = \frac{dx}{dt} + \frac{dy}{dt}$$

Let's see some examples

```
f(x, y):
   a = x + y
   b = -a
   return b
```

```
Df(x, y, dx, dy):
   a = x + y
   da = dx + dy
   b = -a
   db = -da
```

Let's see some examples

```
f(x, y):
   a = x + y
   b = -a
   return b
```

```
Df(x, y, dx, dy):
   a = x + y
   da = dx + dy
   b = -a
   db = -da
   return b, db
```

Let's see some more examples

```
f(x, y, z):
  a = x * y
  b = a / z
  return b
```

Let's see some more examples

```
f(x, y, z):
  a = x * y
  b = a / z
  return b
```

```
Df(x, dx, y, dy, z, dz):
```

Let's see some more examples

```
f(x, y, z):
  a = x * y
  b = a / z
  return b
```

```
Df(x, dx, y, dy, z, dz):
  a = x * y
  da = x * dy + dx * y
```

Let's see some more examples

```
f(x, y, z):
   a = x * y
   b = a / z
   return b
```

```
Df(x, dx, y, dy, z, dz):
   a = x * y
   da = x * dy + dx * y
   b = a / z
   db = da/z-dz*a/z^2
```

Let's see some more examples

```
f(x, y, z):
   a = x * y
   b = a / z
   return b
```

```
Df(x, dx, y, dy, z, dz):
   a = x * y
   da = x * dy + dx * y
   b = a / z
   db = da/z-dz*a/z^2
   return b, db
```

Let's see even more examples

```
f(x):
   t = sqrt(x)
   t = sin(t)
   t = cos(t)
   t = exp(t)
   return t
```

Let's see even more examples

```
f(x):
   t = sqrt(x)
   t = sin(t)
   t = cos(t)
   t = exp(t)
   return t
```

```
Df(x, dx):
```

Let's see even more examples

```
f(x):
   t = sqrt(x)
   t = sin(t)
   t = cos(t)
   t = exp(t)
   return t
```

```
Df(x, dx):
   t = sqrt(x)
   dt = 0.5*dx/sqrt(x)
```

what happens if x = 0?

Let's see even more examples

```
f(x):
   t = sqrt(x)
   t = sin(t)
   t = cos(t)
   t = exp(t)
   return t
```

```
Df(x, dx):
   t = sqrt(x)
   dt = 0.5*dx/sqrt(x)
   t = sin(t)
   dt = dt*cos(t)
```

Let's see even more examples

```
f(x):
   t = sqrt(x)
   t = sin(t)
   t = cos(t)
   t = exp(t)
   return t
```

```
Df(x, dx):
   t = sqrt(x)
   dt = 0.5*dx/sqrt(x)
   t = sin(t)
   dt = dt*cos(t)
   t = cos(t)
   dt = -dt*sin(t)
```

Let's see even more examples

```
f(x):
   t = sqrt(x)
   t = sin(t)
   t = cos(t)
   t = exp(t)
   return t
```

```
Df(x, dx):
   t = sqrt(x)
   dt = 0.5*dx/sqrt(x)
   t = sin(t)
   dt = dt*cos(t)
   t = cos(t)
   dt = -dt*sin(t)
   t = exp(t)
   dt = dt*t
   return t, dt
```

A key of automatic differentiation: efficiently handle intermediate variables

```
f(x):
  r = x
  for i = 1 to 8:
    r = exp(r)
  return r
```

A key of automatic differentiation: efficiently handle intermediate variables

```
f(x):
  r = x
  for i = 1 to 8:
    r = exp(r)
  return r
```

mathematica (37 exponents):

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = e^{x+e^{e^{e^{e^{e^{e^{e^{x}}}}}}}+e^{e^{e^{e^{e^{e^{x}}}}}}+e^{e^{e^{e^{e^{x}}}}}+e^{e^{e^{e^{x}}}}+e^{e^{e^{x}}}+e^{e^{x}}+e^{x}}$$

A key of automatic differentiation: efficiently handle intermediate variables

```
f(x):
  r = x
  for i = 1 to 8:
    r = exp(r)
  return r
```

mathematica (37 exponents):

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = e^{x+e^{e^{e^{e^{e^{e^{e^x}}}}}}+e^{e^{e^{e^{e^{e^x}}}}}+e^{e^{e^{e^{e^x}}}}+e^{e^{e^{e^x}}}+e^{e^{e^x}}+e^{e^x}+e^x}$$

forward-mode automatic differentiation (8 exponents):

```
Df(x, dx):
  r = x
  dr = dx
  for i = 1 to 8:
    r = exp(r)
    dr = dr * r
  return r, dr
```

Complexity analysis

forward mode always maintain the same time complexity as the primal code

```
f(x):
  r = x
  for i = 1 to 8:
    r = exp(r)
  return r
```

```
Df(x, dx):
  r = x
  dr = dx
  for i = 1 to 8:
    r = exp(r)
    dr = dr * r
  return r, dr
```

Complexity analysis

forward mode always maintain the same time complexity as the primal code

however, we only outputs $\mathbb{R}^m$ number for each evaluation,
to get the full Jacobian, we need to evaluate n times

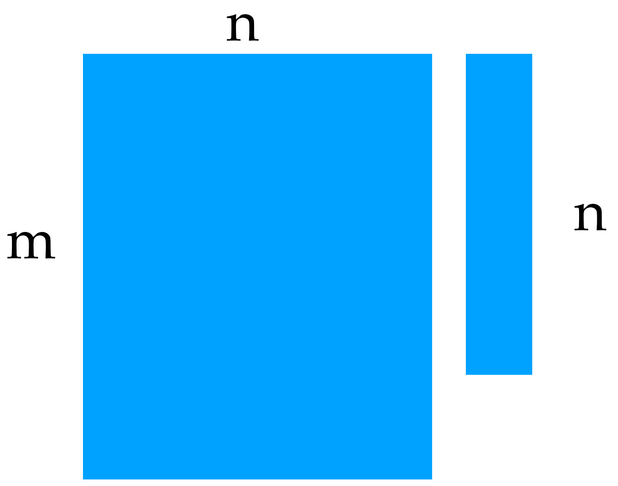
$$f : \mathbb{R}^n \to \mathbb{R}^m$$

$$Df(x_0) : \mathbb{R}^n \to \mathbb{R}^m \text{ (linear)}$$

$$Df(x_0) = \text{Jacobian}$$

$$Df : \mathbb{R}^n \to \left(\mathbb{R}^n \to \mathbb{R}^m\right)$$

$$Df(x_0)(dx) = \text{total derivative}$$



One way to implement forward mode: operator overloading

```
class dfloat:
  val : float
  dval : float
```

`float`

One way to implement forward mode: operator overloading

```
class dfloat:
  val : float
  dval : float
```

`float`

```
dfloat operator+(dfloat x, dfloat y):
  return dfloat(x.val+y.val, x.dval+y.dval)
```

One way to implement forward mode: operator overloading

```
class dfloat:
  val : float
  dval : float
```

`float` ⟶

```
dfloat operator+(dfloat x, dfloat y):
  return dfloat(x.val+y.val, x.dval+y.dval)

dfloat operator-(dfloat x, dfloat y):
  return dfloat(x.val-y.val, x.dval-y.dval)
```

One way to implement forward mode: operator overloading

`float` ⟶

```
class dfloat:
  val : float
  dval : float

dfloat operator+(dfloat x, dfloat y):
  return dfloat(x.val+y.val, x.dval+y.dval)

dfloat operator-(dfloat x, dfloat y):
  return dfloat(x.val-y.val, x.dval-y.dval)

dfloat operator*(dfloat x, dfloat y):
  return dfloat(x.val*y.val,
    x.dval*y.val+y.val*x.dval)
```

One way to implement forward mode: operator overloading

```
class dfloat:
  val : float
  dval : float
```

`float`

```
dfloat operator+(dfloat x, dfloat y):
  return dfloat(x.val+y.val, x.dval+y.dval)

dfloat operator-(dfloat x, dfloat y):
  return dfloat(x.val-y.val, x.dval-y.dval)

dfloat operator*(dfloat x, dfloat y):
  return dfloat(x.val*y.val,
    x.dval*y.val+y.val*x.dval)
```

you can then
change all float in your code into dfloat

Dual numbers implement forward mode

dual number

$$a + \epsilon b$$

$$\epsilon^2 = 0$$

Dual numbers implement forward mode

$$a + \epsilon b + c + \epsilon d = (a + c) + \epsilon \left(b + d\right)$$

dual number

$$a + \epsilon b$$

$$\epsilon^2 = 0$$

Dual numbers implement forward mode

$$a + \epsilon b + c + \epsilon d = (a + c) + \epsilon \left(b + d\right)$$

$$\left(a + \epsilon b\right)\left(c + \epsilon d\right) = ac + \epsilon \left(bc + ad\right)$$

dual number

$$a + \epsilon b$$

$$\epsilon^2 = 0$$

Dual numbers implement forward mode

dual number

$$a + \epsilon b$$

$$\epsilon^2 = 0$$

$$a + \epsilon b + c + \epsilon d = (a + c) + \epsilon \left(b + d\right)$$

$$\left(a + \epsilon b\right) \left(c + \epsilon d\right) = ac + \epsilon \left(bc + ad\right)$$

$$\frac{\left(a + \epsilon b\right)}{\left(c + \epsilon d\right)} = \frac{\left(a + \epsilon b\right) \left(a - \epsilon b\right)}{\left(c + \epsilon d\right) \left(c - \epsilon d\right)}$$

$$= \frac{a}{c} + \epsilon \frac{bc - ad}{c^2}$$