Reverse mode AD cont'd

The above process is called "forward-mode" AD.
It is accurate to machine epsilon, and easy to program.
But it is not optimally efficient.

Machine learning - training a neural network - involves minimizing a real-valued function called the "loss", which is a measure of how poorly the network is performing the desired task.

The network is characterized by a large set of parameters called "weights",

$w_0, w_1, w_2, \ldots, w_{1000000000}$.

loss $y = N(\vec{w})$. Want to minimize y with respect to $\vec{w}$.

large vector of weights

The standard method is to move repeatedly in the direction of steepest descent. gradient descent

That direction is $-\nabla_w N$

Because of the huge dimension of $\vec{w}$, we need to be as efficient as possible (as well as accurate).

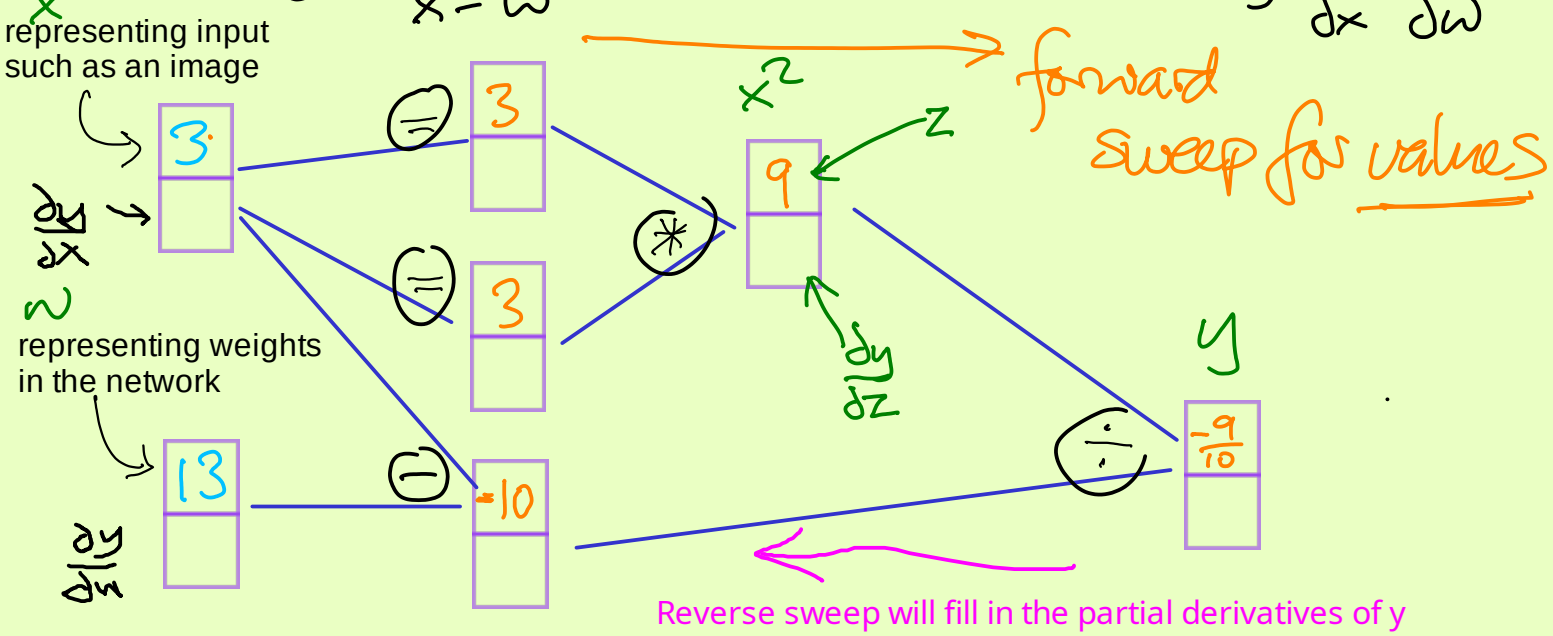
"Reverse-mode AD" or "back-propagation" is more efficient than forward-mode AD.

In the diagram below,
the upper slots are for the *values* of initial, intermediate, and final quantities in the calculation,
the lower slots are for the *derivative of the output value y with respect to the quantity*.

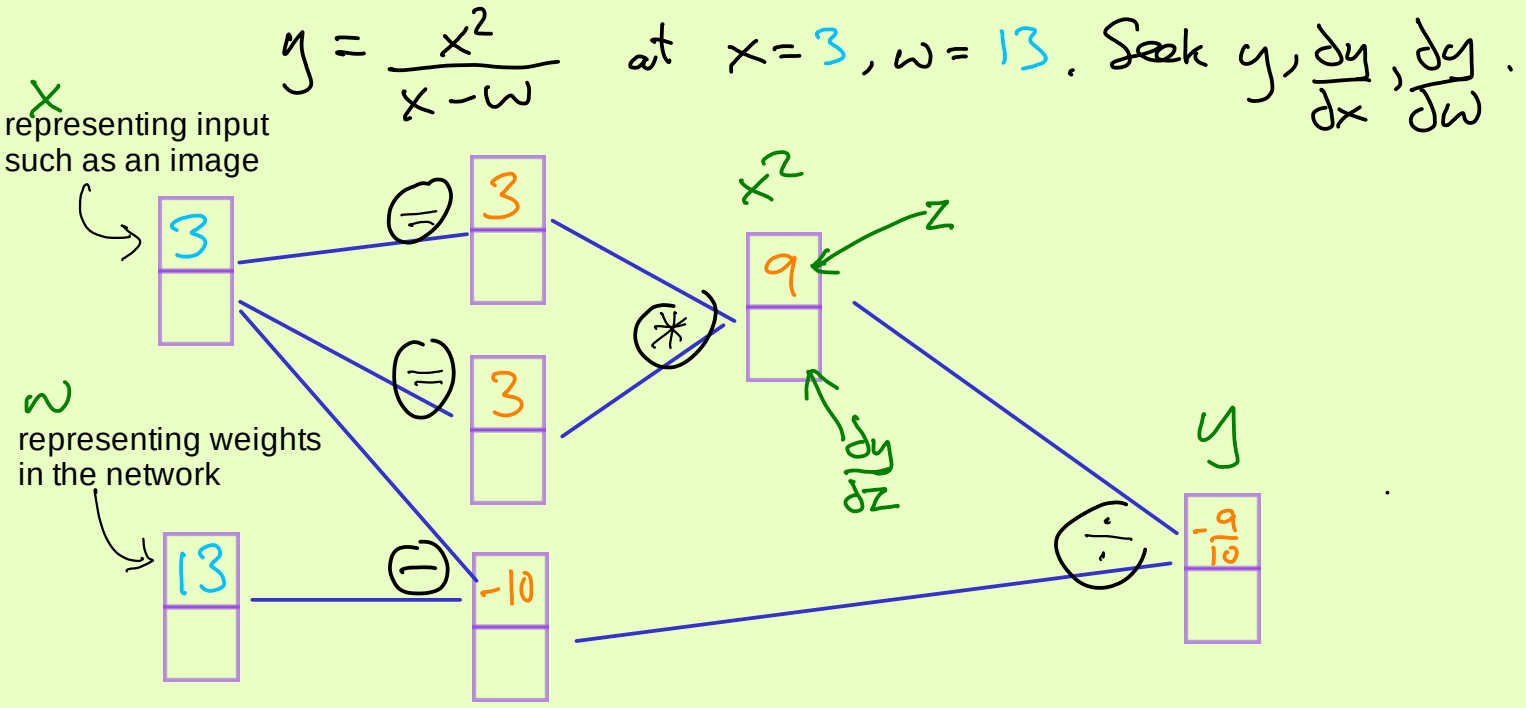
Our goal is to fill in all the slots in two sweeps:
(i) a forward sweep to fill in all the *values*
(ii) a reverse sweep (back propagation) to fill in all the partial derivatives,

The last numbers we compute in the reverse sweep will be the desired components of the gradient of y.

Example: $y = \frac{x*x}{x-w}$ at $x=3, w=13$. Seek $y, \frac{dy}{dx}, \frac{dy}{dw}$.

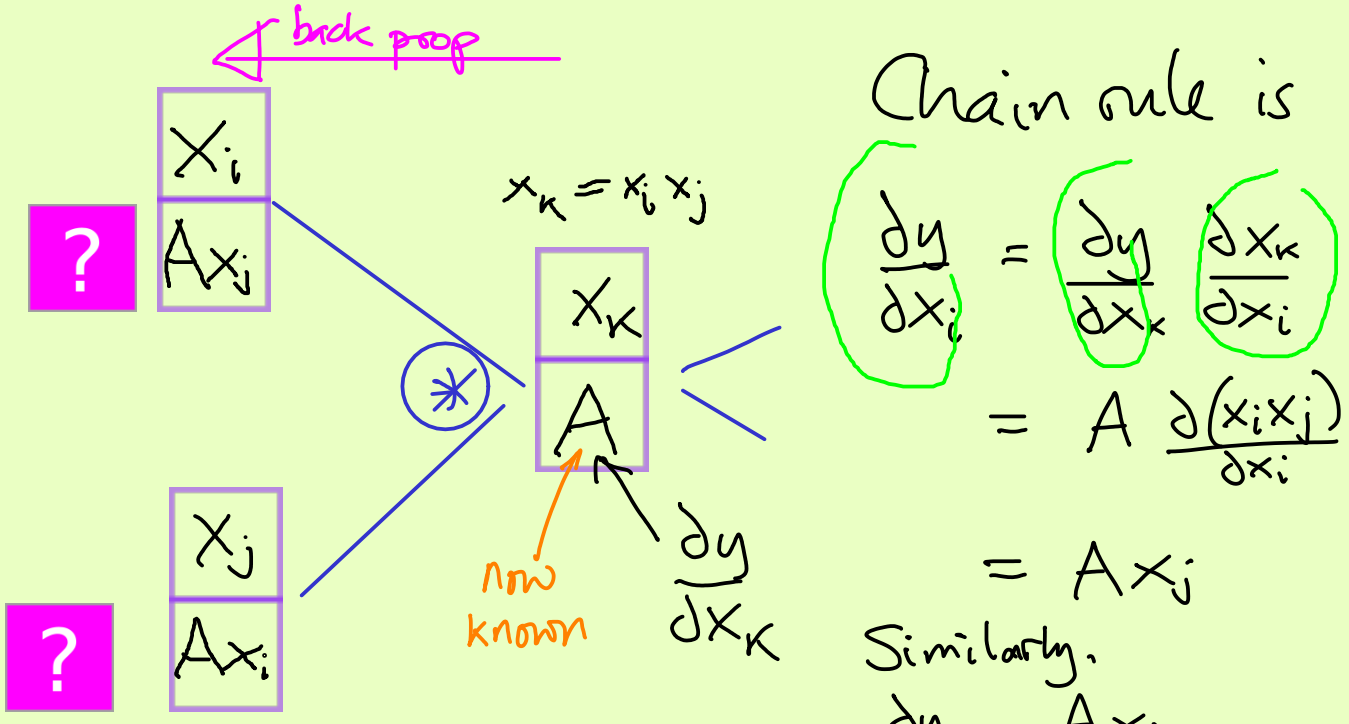


Forward sweep to fill in the values:



For the reverse sweep to fill in the partial derivatives of y, we need to develop some rules ...

Let's look at multiplication.



Chain rule is

$$\frac{dy}{dx_i} = \frac{dy}{dx_k}\frac{dx_k}{dx_i}$$

$$= A\frac{d(x_i x_j)}{dx_i}$$

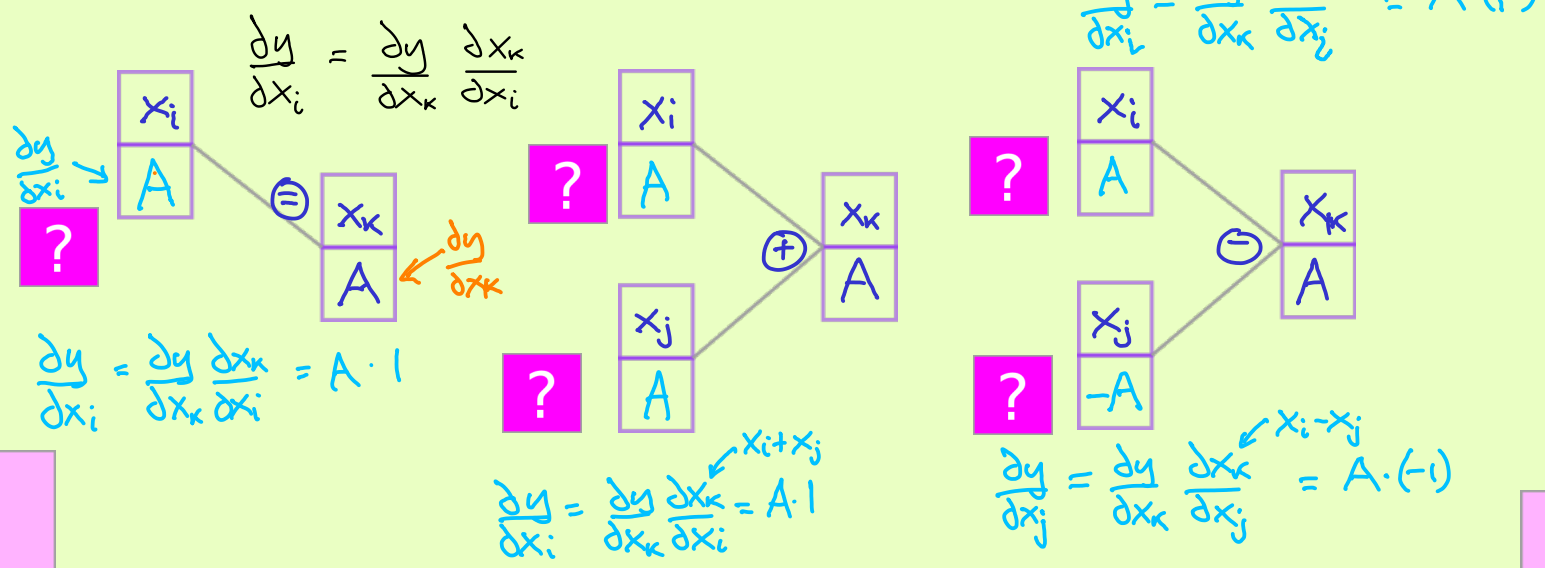
$$= Ax_j$$

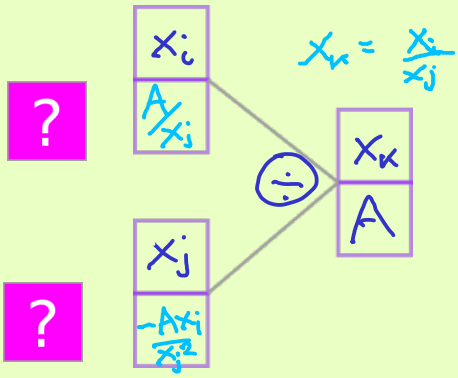
Similarly,

$$\frac{dy}{dx_j} = Ax_i$$

Exercise for you:

Find the back prop rules for the operations of $(=), (-), (\div)$ and branching.



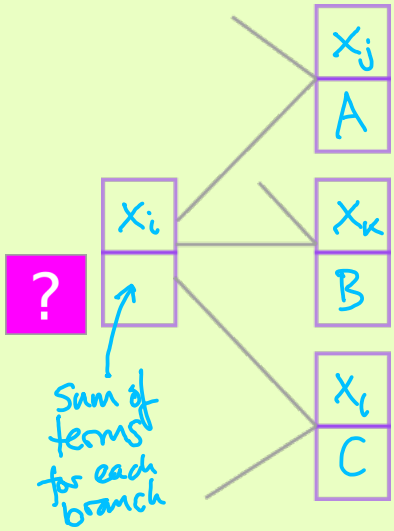


$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial x_k}\frac{\partial x_k}{\partial x_i} = A \cdot \frac{\partial}{\partial x_i}\left(\frac{x_i}{x_j}\right) = A \cdot \frac{1}{x_j}$$

$$\frac{\partial y}{\partial x_j} = \frac{\partial y}{\partial x_k}\frac{\partial x_k}{\partial x_j} = A \cdot \frac{\partial}{\partial x_j}\left(\frac{x_i}{x_j}\right) = A \cdot \left(-\frac{x_i}{x_j^2}\right)$$

$$\frac{\partial y_k}{\partial x}$$

Branching - y depends on x_i via multiple paths

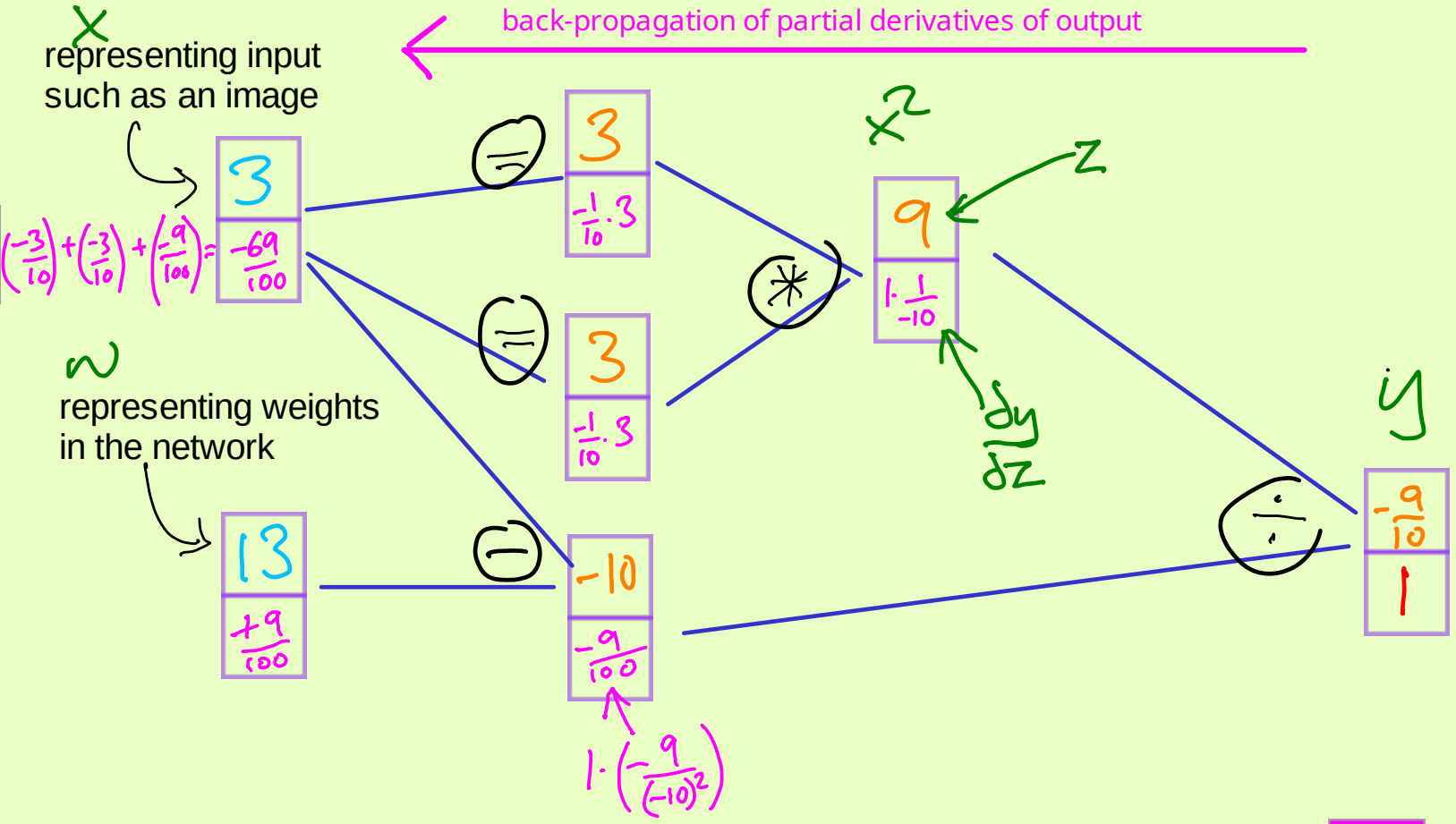


$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial x_j}\frac{\partial x_j}{\partial x_i} + \frac{\partial y}{\partial x_k}\frac{\partial x_k}{\partial x_i} + \frac{\partial y}{\partial x_l}\frac{\partial x_l}{\partial x_i}$$

$$= A\frac{\partial x_j}{\partial x_i} + B\frac{\partial x_k}{\partial x_i} + C\frac{\partial x_l}{\partial x_i}$$

Finally let's apply our back prop rules to compute the gradient of our example expression.

$y = \frac{x^2}{x-w}$ at $x=3, w=13$. Seek $y, \frac{dy}{dx}, \frac{dy}{dw}$.



?

Let's check our reverse-mode AD results with symbolic differentiation:

```
import sympy as sp
x,w = sp.symbols('x,w')
y = x**2/(x-w)
dydx = sp.diff(y,x)
dydw = sp.diff(y,w)
print('partial derivatives of y wrt x,w :')
display(dydx)
display(dydw)
values = {x:3,w:13}
print(f'value of y at x={values[x]},w={values[w]}: {y.subs(values)}\n')
print(f'partial derivatives at x={values[x]},w={values[w]}:',\
      f'{dydx.subs(values)}, {dydw.subs(values)}')
```

```
partial derivatives of y wrt x,w :
```

$$-\frac{x^2}{(-w+x)^2}+\frac{2x}{-w+x}$$

$$\frac{x^2}{(-w+x)^2}$$

```
value of y at x=3,w=13: -9/10

partial derivatives at x=3,w=13: -69/100 9/100
```

xk = sin(xi)

dy/dxi =
dy/dk * dxk/dxi
= "A" * cos(xi)

?

Quadrature

= numerical evaluation of definite integrals

The word quadrature comes from real-world estimating of

$$\int_R f\,dA$$

e.g. $\int_R 1\,dA$ = area of region R

Crude estimate:
count the "quadrats".

area of stone = 6 to 9

Start with 1D: $f:[a,b]\to\mathbb{R}$. To approximate $J(f) \equiv \int_a^b f = \int_a^b f(x)\,dx$,

Why not just find a formula F such that F' = f, and use J(f) = F(b) - F(a)?

We will when we can, but a closed formula for such an F frequently
(i) does not exist, or (ii) is a pain to obtain.

We will sample f at m+1 points $x_0, x_1, \cdots, x_m$,
and devise a "quadrature rule"

$Q(f) = q\left(f(x_0), f(x_1), \cdots, f(x_m)\right) \approx J(f)$ as accurately as possible.

We would really like Q(cf) = c Q(f) for any constant c
and Q(f+g) = Q(f)+Q(g)
to mirror the corresponding properties required of an integral.

This forces q to be a linear combination of its arguments (proof?):

$$Q(f) = \sum_{j=0}^{m} w_j f(x_j) \overset{\text{convenience}}{=} (b-a)\sum_{j=0}^{m} \alpha_j f(x_j)$$

(α_j: weights)

Factoring out (b-a) from the w_j's makes the α_j's independent of the interval length.

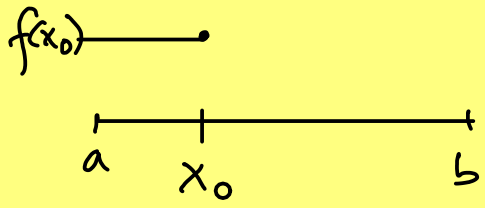
It remains to choose the $\{x_j\}$ and the $\{\alpha_j\}$ to get an accurate approximation of $\int_a^b f$.

Based on our experience with interpolation, it seems likely that some placements of the x_j will be better than others. But supposed we've decided on the {x_j}. Maybe uniformly space, maybe not.

Then we need to decide on the weights $\{w_j\}$ or equivalently $\{\alpha_j\}$.

What principle(s) should we use to constrain the weights?

Example: m=0, one sample of f:



$\{x_j\} = (x_0)$

$$Q(f) = w_0 f(x_0) = (b-a)\alpha_0 f(x_0).$$

What's a good choice for α_0 ?

Seems like it is highly desirable to get the integral of a constant function exactly correct:

$$Q(1) = (b-a)\alpha_0 \cdot 1 \overset{\text{want}}{=} J(1) = \int_a^b 1\,dx = (b-a)$$

This requires $\alpha_0 = 1$.

So our one-point quadrature rule is: $Q(f) = (b-a)\cdot 1 \cdot f(x_0)$.

What if we have more than 1 sample point?

What does $Q(1) = \int 1$ dictate?

$$Q(1) = (b-a)\sum_{j=0}^{m} \alpha_j f(x_j) \overset{f=1}{=} (b-a)\sum_{j=0}^{m} \alpha_j \cdot 1 \overset{\text{need}}{=} (b-a) = \int_a^b 1\,dx$$

That is, we need $\boxed{\sum_{j=0}^{m} \alpha_j = 1}$. That's 1 constraint on m+1 variables.

We need m additional constraints.

How about requiring the rule to integrate linear functions correctly?

$$Q(L) = \int L$$

That provides one more (linear) constraint on the weights.

Continuing the theme, we could require that Q gets the exact integral for all polynomials of degree m or less:

$$Q(p) = \int p \quad \forall\ p \in P_m.$$

This is m+1 (linear) constraints on the m+1 variables $\{a_j\}$.

Such a Q is said to have "polynomial degree m".

We can arrive at the same quadrature rule Q in another way ...

Let $p(f) \in P_m$ be the unique polynomial of degree at most m that interpolates the data $\{(x_j, f(x_j))\}$.

Then define $Q(f) = \int_a^b p(f)$. (This integral is easy to do symbolically because p is a polynomial.)

We could write p(f) in terms of the Lagrange polynomials:

$$Q(f) = \int_a^b p(f)(x)\,dx = \int_a^b \sum_{j=0}^{m} f(x_j)\,\ell_j(x)\,dx = \sum_{j=0}^{m} f(x_j) \underbrace{\int_a^b \ell_j(x)\,dx}_{=w_j}$$

We can show that the two definitions of Q are equivalent.

which can be computed symbolically exactly.

Another example:

(m+1)-point rule of polynomial degree m with $x_0 = a, x_m = b$ and the others uniformly spaced in between.

This is called the "closed Newton-Cotes rule" of degree m.

Again note that once the {x_j} are chosen, the requirement to be of polynomial degree m fixes the weights.

If the m+1 nodes are uniformly spaced with the endpoints a,b omitted, this is called the "open Newton-Cotes" rule of degree m.

Deriving the weights exactly using sympy

```
# derive weights for quadrature rule by integrating Lagrange form of interpolating polynomial
import numpy as np
xx = sp.symbols('x')
a,b = 0,2
H = b-a
m = 3
x = [sp.Rational(j*H,m) for j in range(m+1)]
display(x)

alpha = []
for j in range(m+1):
    Lj = 1
    for i in range(m+1):
        if i!=j: Lj *= (xx-x[i])/(x[j]-x[i])
    Lj = Lj.expand()
    display(Lj)
    alpha.append(sp.integrate(Lj,(xx,a,b))/H)
alpha
```

[a,b]

m+1 = 3+1 = 4 sample points $\{x_j\}$

$\{x_j\} =$ `[0, 2/3, 4/3, 2]`

$\ell_0(x) : -\frac{9x^3}{16} + \frac{9x^2}{4} - \frac{11x}{4} + 1$

$\ell_1(x) : \frac{27x^3}{16} - \frac{45x^2}{8} + \frac{9x}{2}$

$\ell_2(x) = -\frac{27x^3}{16} + \frac{9x^2}{2} - \frac{9x}{4}$

$\ell_3(x) = \frac{9x^3}{16} - \frac{9x^2}{8} + \frac{x}{2}$

Lagrange polynomials for this partition of this interval.

$\{x_j\} =$ `[1/8, 3/8, 3/8, 1/8]`

$$Q(f) = \frac{2}{3}\left\{\frac{1}{8}f(0) + \frac{3}{8}f\left(\frac{2}{3}\right) + \frac{3}{8}f\left(\frac{4}{3}\right) + \frac{1}{8}f(2)\right\}$$

$\frac{b-a}{m}$

Let's "funcify" the above code and look at some other concrete examples ...

- closed Newton-Cotes on several intervals
- open Newton-Cotes
- randomly chosen points

```
def linspace(a,b,n):  # exact rationals with sympy
    h = b-a
    return [a + sp.Rational(i,n-1)*h for i in range(n)]

def get_alphas(a,b,x):
    H = b-a
    alpha = []
    for j in range(m+1):
        Lj = 1
        for i in range(m+1):
            if i!=j: Lj *= (xx-x[i])/(x[j]-x[i])
        Lj = Lj.expand()
        #display(Lj)
        alpha.append(sp.integrate(Lj,(xx,a,b))/H)
    return alpha

m = 3

a,b = 0,2
x = linspace(a,b,m+1)
print( f'For the interval [{a},{b}] with the {m+1} points {x},\nthe alphas are',
       get_alphas(0,2,x) )

a,b = 0,100
x = linspace(a,b,m+1)
print( f'For the interval [{a},{b}] with the {m+1} points {x},\nthe alphas are',
       get_alphas(a,b,x) )
```

```
For the interval [0,2] with the 4 points [0, 2/3, 4/3, 2],
the alphas are [1/8, 3/8, 3/8, 1/8]
For the interval [0,100] with the 4 points [0, 100/3, 200/3, 100],
the alphas are [1/8, 3/8, 3/8, 1/8]
```

Copyable code:

```
def linspace(a,b,n):  # exact rationals with sympy
    h = b-a
    return [a + sp.Rational(i,n-1)*h for i in range(n)]

def get_alphas(a,b,x):
    H = b-a
    alpha = []
    for j in range(m+1):
        Lj = 1
        for i in range(m+1):
            if i!=j: Lj *= (xx-x[i])/(x[j]-x[i])
        Lj = Lj.expand()
        #display(Lj)
        alpha.append(sp.integrate(Lj,(xx,a,b))/H)
    return alpha

m = 3

a,b = 0,2
x = linspace(a,b,m+1)
print( f'For the interval [{a},{b}] with the {m+1} points {x},\nthe alphas
       get_alphas(0,2,x) )

a,b = 0,100
x = linspace(a,b,m+1)
print( f'For the interval [{a},{b}] with the {m+1} points {x},\nthe alphas
       get_alphas(a,b,x) )
```