The Derivative Rules

Introduction

In 1979 Ken Iverson wrote a paper on the derivative as part of the ACM APL Conference [0]. That paper included a section summarizing the rules for the derivatives of compound formulations such as `f×g`. His summary omits the derivation of these rules. This article aims to restate these rules in more modern notation and provide derivations for Iverson's results. A brief summary of the rules is included as Appendix A.

Why are the derivative rules useful?

Writing an operator to calculate a derivative numerically is not that difficult and it certainly works well for many functions, even quite elaborate ones. So why not use this operator for all derivative calculations? There are two reasons. Firstly, with arguments having many elements, it's tough to avoid excessive cpu calculation or memory usage. And secondly, there's the matter of accuracy. General purpose numeric differentiation is always an approximation.

For many problems it's possible to write the derivative analytically using the derivative rules. Doing so can often reduce the resources required as well as improve the accuracy of the result.

The APL environment

All of the text in the APL385 Unicode font is executable in APL. The particular APL used here is Dyalog APL 17.0 with:

```
⎕io←0
⎕pp←6
]boxing on
```

Dyalog APL is freely available for non-commercial use at www.dyalog.com.

Shape analysis

It's useful to take an expression and work though its functional happenings to determine the shape of its result. Included here is an informal technique to record the steps in this process. It requires a convention.

> *If text is in red, it should be read (in APL) as "an object of shape ...".*

For example:

```
    3×2 3 4
2 3 4
```

A scalar multiplying a rank 3 object results in a rank 3 object of the same shape.

```
    (2 3 4⍴6){⍺∘.×⍵}⍤¯2⊢2 3 5 6⍴7
»   2 3 4{⍺∘.×⍵}⍤¯2⊢2 3 5 6
»   2 3,(4∘.×5 6)
2 3 4 5 6
```

Outer products in a rank 2 frame.

Supporting Operators and Functions

This article assumes that a number of operators and functions are already in place. Those that can be described briefly are summarized here. Those requiring more detail are described in Appendix B.

UTILITIES

`min←{⌊/,⍵}`	Minimum
`max←{⌈/,⍵}`	Maximum
`num←{×/⍴⍵}`	Number
`sum←{+/,⍵}`	Sum
`mean←{(sum÷num)⍵}`	Mean
`sop←{+/,⍺×⍵}`	Sum of product
`ssq←{sop⍨⍵}`	Sum of squares
`rnd←{⍺×⌊0.5+⍵÷⍺}`	Round
`image←{28 28⍴(⊂⍵>127)⌷'' *''}'`	Display an image sample
`timer←{∆∆j←1⌷⎕ai ⋄ ∆∆k←⍎⍵ ⋄ 0.001×∆∆j-⍨1⌷⎕ai}`	Execution time in seconds

CELLS OF AN ARRAY

The function `cells` boxes the cells of a specified rank from an array.

```
cells←{⊂⍤⍺⊢⍵}
```

DISPLAY OF HIGHER RANK ARRAYS

Displaying higher rank results can be a bit awkward. It is more manageable if we box things up a bit and take advantage of `]boxing on`, with:

```
disp←{⊂⍤2⊢⍵}
```

COMPARING VALUES

At times we'd like to compare two arrays which, in theory, are identical. However, as one has (or both have) been calculated by numerical approximation, they may differ slightly. To check how close two values are, we use a simple statistic based on the ratio of corresponding elements being within 1% of each other.

```
comp←{a b←∊¨⍵ ⋄ (⍴a)≢⍴b:0 ⋄ mean 0=0.99 1.01⍸a÷b+2×a×b=0}
```

For example:

```
      a←?2 3 4⍴9
      comp¨ (a a) (a(a+100)) (a(1.009×a)) (a(0.991×a)) ((5⍴0)(5⍴0)) ((5⍴1)(5⍴0))
1 0 1 1 1 0
```

The Derivative Operator

The derivative operator

The derivative operator `∆` (introduced by Iverson in [0] and is discussed in [1]) acts on a monadic function `f` of rank `s` to produce a derived function `df`. As Iverson points out in [0], `df` has the same rank as `f`. (This comes directly from the definition of the derivative.)

```
      df←f ∆
```

What should we expect as the shape of the result of `df`? It's a function, just like any other, so, depending on the rank of its argument `x`, the shape of its result could have contributions from both a frame and an individual result. As the rank of `df` is the same as that of `f`, both produce results with frames of the same shape `fr`. This is:

```
      fr←(-s)↓⍴x
```

The individual result for `f` comes from its application to an `s`-cell of x. It has shape `sir`:

```
      c←⊃s cells x
      sir←⍴f c
```

A representative cell.

The individual result for `df` is a little more elaborate. It produces the sensitivity of each individual result relative to its corresponding cell `c`. The shape of `df`'s result is:

```
      fr,sir,⍴c
»     ((-s)↓⍴x),(⍴f c),⍴c
```

Avoiding surplus zeros

It's very important to observe that we need to know the rank of the function `f` to apply its derivative correctly. (In the J language this can be discovered with the adverb `b.0`. Unfortunately, Dyalog APL does not yet provide the means to get this information.) Below is a definition for a derivative operator. It assumes that the rank of its argument does not exceed the rank of the function – i.e. no frame is produced. If we mistakenly use this operator with an argument of too great a rank, the result will be sparse, filled with many surplus zeros. For example:

```
      f←{⍵*2}                      A rank 0 function.
      ⍴a←f ∆⍤0⊢x←3 4 5⍴⍳60         ∆ used correctly.
3 4 5
      ⍴b←f ∆ x                     ∆ making a bad assumption about the rank of
3 4 5 3 4 5                        the function f.
```

We can readily see the surplus zeros as follows:

```
      sum a=0
0                                  No zeros.
      sum b=0
3540                               All but 60 elements are zero.
```

Note also that all the values in `a` do appear in `b`, just interspersed with many zeros:

```
      (,a)≡(,b)~0
1
```

Definition

Here is a definition of a derivative operator `∆`.

```
r←(f ∆)x;c;p;q;dx;d;j;n;sf;sx;t

 ⍝ Derivative of function f at x.
 ⍝ Assumes that the application of f to x does not produce a frame.
 ⍝ Coding comments:
 ⍝   Uses a loop to reduce memory usage.
 ⍝   Careless regard by f for the locals used here could be fatal.

 sf←⍴p←f x ⋄ sx←⍴x ⋄ dx←0.000001×{⍵+⍵=0}x←,x
 r←(×/sx,sf)⍴p ⋄ c←0 ⋄ j←0
 :While c<⍴dx
     q←x ⋄ d←c⌷dx ⋄ (c⌷q)+←d
     n←⍴t←,((f sx⍴q)-p)÷d ⋄ r[j+⍳n]←t
     c+←1 ⋄ j+←n
 :EndWhile
 r←(sx,sf)⍴r ⋄ r←((⍴sf)⌽⍳⍴⍴r)⍉r
```

This definition of the derivative matches that given by Iverson in [0] with one caveat:

> *If the rank* `s` *of the function* `f` *is less than that of the argument* `x`*, then the derivative* `f ∆` *must be applied with rank* `s`.

Note that:

- The function argument `f` is executed by `∆`. If `f` relies on global variables, then the values used may be those localized within `∆`. This is likely to be incorrect. The problem can be pushed out of sight by prefixing every name used within the definition of `∆` with a very uncommon prefix (such as two underbar characters). For reasons of readability, the definition of `∆` above leaves out those prefix characters – although they should be present in the workspace definition.
- This definition works by looping through every element of the argument once. This could have been coded without a loop but doing so requires more memory.

Iverson's Rules for Scalar and Vector Functions

Notation

Iverson chose glyphs to represent a number of new operators. In line with the technology available at the time (the IBM Selectric type ball), some are overstruck combinations of two APL characters. Here's a summary:

```
Composition    F¨G  ↔  F{αα ωω ω}G

Sum            F+̄G  ↔  F{(αα ω)+ωω ω}G
Product        F×̄G  ↔  F{(αα ω)×ωω ω}G
Transpose      F⍉̄G  ↔  F{(αα ω)⍉ωω ω}G

Inner Product  F⊕G  ↔  F{(αα ω)+.×ωω ω}G
Outer Product  F⊗G  ↔  F{(αα ω)∘.×ωω ω}G
```

Function rank

Iverson chose to keep two integers for the definition of rank. He said:

> "We will characterize the <u>rank</u> of a monadic function by a two-element vector, whose last element specifies the minimum rank of the argument on which the function is normally defined, and whose first specifies the rank of the result when the function is applied to an argument of minimum rank."

In the discussion below, we'll follow the definition of rank for a monadic function as it is used today – a single integer corresponding to the last element of Iverson's definition. Where we need to refer to the first element of Iverson's definition we'll refer to the "result rank".

Iverson's statement of the rules

Iverson stated the rules as follows:

```
"These operators allow us to express the extension
to vector functions (of rank 1 1) of the usual sum,
product, composition, and other rules for the derivatives
of scalar functions:

Sum           F+̄GΔ ←→ (FΔ)+̄(GΔ)
              F⊗GΔ ←→ F⊗(GΔ)⊕(0 2 1 ⍉̄(FΔ⊗G))
Product       F×̄GΔ ←→ 0 0 1¨⍉¨(F⊗GΔ)
Composition   F¨GΔ ←→ FΔ¨G⊕(GΔ)
Inverse      F⍣¯1Δ ←→ ÷¨(FΔ¨(F⍣¯1))
              F⊕GΔ ←→ F⊗GΔ⊕0 1

It may be helpful to compare these identities with the
simpler identities for scalar functions, noting, in particular
that for a scalar function G the expressions G⊗(GΔ) and
G×̄(GΔ) are equivalent:

Sum           F+̄GΔ ←→ (FΔ)+̄(GΔ)
Product       F×̄GΔ ←→ (FΔ×̄G)+̄(GΔ×̄F)
Composition   F¨GΔ ←→ FΔ¨G×̄(GΔ)
Inverse      F⍣¯1Δ ←→ ÷¨(FΔ¨(F⍣¯1))
"
```

Each of the rules uses an operator to produce a derived function which is supplied as the left argument to the derivative operator `∆`. The operators are all dyadic, three (plus-overbar, times-overbar and dieresis) taking functions `F` and `G` as the left and right arguments respectively and the last (power) taking the function `F` as its left argument and `¯1` as its right argument.

A useful starting point is to restate these identities taking advantage of today's APL operators and functions.

Scalar functions

The second table gives the rules for scalar functions.

For this table, `F` and `G` are scalar functions (of rank `0`) having an individual cell of shape `sic←θ` and an individual result of `sir←θ`. We can expect these functions to produce derivatives, which when applied to an argument `x`, produce a frame of shape `fr←ρx` and an empty vector for the shape of both a cell and its corresponding result (`sic←sir←θ`). The shape of the final result is just `fr,sir,sic`, in other words `ρx`.

At the time Iverson wrote, the concept of function trains was little understood. In fact, all four of the identities can be rewritten with trains, avoiding the introduction of new operators. Here's how an updated table of scalar function derivatives could look:

```
Sum           F+̄GΔ ←→ (FΔ)+̄(GΔ)           (f+g)∆ ←→ f ∆+g ∆
Product       F×̄GΔ ←→ (FΔ×̄G)+̄(GΔ×̄F)      (f×g)∆ ←→ (f ∆×g)+(g ∆×f)
Composition   F¨GΔ ←→ FΔ¨G×̄(GΔ)           (f g)∆ ←→ (f ∆ g)×g ∆
Inverse      F⍣¯1Δ ←→ ÷¨(FΔ¨(F⍣¯1))       f⍣¯1 ∆ ←→ ÷(f ∆ f⍣¯1)
```

Note that:

- `(f+g)` and `(f×g)` are monadic forks. `(f g)` is monadic atop.
- `f ∆+(g ∆)` is a monadic fork evaluated as `{(f ∆ ⍵)+g ∆ ⍵}`.
- `(f ∆×g)+g ∆×f` is a monadic fork comprising of the functions `(f ∆×g)`, `+` and `g ∆×f`. Further `(f ∆×g)` and `(g ∆×f)` are themselves monadic forks. The result is evaluated as `{((f ∆ ⍵)×g ⍵)+(g ∆ ⍵)×f ⍵}`.
- `(f ∆ g)×g ∆` is a monadic fork comprising of the functions `(f ∆ g)`, `×` and `g ∆`. Further `(f ∆ g)` itself is a train, an example of monadic atop. The result is evaluated as `{(f ∆ g ⍵)×g ∆ ⍵}`.
- `÷(f ∆ f⍣¯1)` is the reciprocal of `f`'s derivative atop its inverse. It is evaluated as `{÷(f ∆ f⍣¯1)⍵}`.

Sum

```
      f←{⍵*2} ⋄ g←{1○⍵}
      x←3 1 2 7
      dfplusg←{(2×⍵)+2○⍵}
```

The exact derivative of `(f+g)`.

```
      dfplusg x
5.01001 2.5403 3.58385 14.7539
      (f+g)∆⍤0⊢x
5.01001 2.5403 3.58385 14.7539
      rule←(f ∆)+g ∆
      rule⍤0⊢x
5.01001 2.5403 3.58385 14.7539
```

Product

```
      dftimesg←{(2×⍵×1○⍵)+(⍵*2)×2○⍵}
```

The exact derivative of `(f×g)`.

```
      dftimesg x
¯8.06321 2.22324 1.9726 46.139
      rule←(f ∆×g)+g ∆×f
      rule⍤0⊢x
¯8.06321 2.22324 1.9726 46.1389
```

Composition

```
      dfatopg←{2×(1○⍵)×2○⍵}
```

The exact derivative of `(f g)`.

```
      dfatopg x
¯0.279415 0.909297 ¯0.756802 0.990607
      rule←(f ∆ g)×g ∆
      rule⍤0⊢x
¯0.279416 0.909297 ¯0.756805 0.990605
```

INVERSE

```
      f←*∘2                          as dfns are not yet valid arguments to ⍣.
      fi←f⍣¯1                        The inverse of f.
      dfi←{0.5×⍵*¯0.5}               The exact derivative of the inverse of f.
      f fi x
3 1 2 7
      fi f x
3 1 2 7
      dfi x
0.288675 0.5 0.353553 0.18898
      rule←÷(f Δ fi)
      rule⍤0⊢x
0.288675 0.5 0.353553 0.188982
```

Vector functions

The first table gives the rules for vector functions `f` and `g` (rank `1` producing vector results). As these are the arguments to the operators used for sum, product, composition and inverse, the derived functions so produced are also vector functions. There are two other identities included, inner and outer product. The inner product is of rank `1` producing a scalar result (what Iverson calls a scalar function of a vector). The outer product is also of rank `1`, acting on vectors, but produces a result of rank `2`. The six identities can be restated in more modern notation as follows:

```
Sum           F+̄GΔ ←→ (FΔ)+̄(GΔ)                     (f+g)Δ ←→ f Δ+g Δ
              F⊗GΔ ←→ F⊗(GΔ)⊕(0 2 1 ⍉̄(FΔ⊗G))     (f∘.×g)Δ ←→ (f∘.×g Δ)+0 2 1⍉f Δ∘.×g
Product       F×̄GΔ ←→ 0 0 1¨⍉¨(F⊗GΔ)                 (f×g)Δ ←→ 0 0 1⍉(f∘.×g)Δ
Composition   F¨GΔ ←→ FΔ¨G⊕(GΔ)                      (f g)Δ ←→ (f Δ g)+.×g Δ
Inverse     F⍣¯1Δ ←→ ÷¨(FΔ¨(F⍣¯1))                  f⍣¯1 Δ ←→ ⌹f Δ f⍣¯1
              F⊕GΔ ←→ F⊗GΔ⊕0 1                     (f+.×g)Δ ←→ 0 1 tc⍨(f∘.×g)Δ
```

Note that there are probably two typos in Iverson's table:

- The expression for the derivative of the outer product should have two terms, added together with the operator plus overbar not with the inner product operator ⊕.
- The expression for the derivative of an inverse uses the reciprocal function `÷`. This should probably be the matrix inverse `⌹`. (In many quite ordinary situations we can expect to have zeros appear in the result of `f Δ f⍣¯1`. These give errors with `÷` rather than the correct result which is obtained with `⌹`.)

Here are some examples showing the application of the derivative of a vector function to a higher rank array.

SUM

```
      f←{×\ω}⍤1 ⋄ g←{1○ω}⍤1
      ⍴a←(f+g)∆⍤1⊢x←2 3⍴⍳6
2 3 3
      disp a
┌───────────────────┬─────────────────────────────┐
│2 0        0       │ 0.0100073  0        0       │
│1 0.540302  0      │ 4          2.34636  0       │
│2 0        ¯0.416148│20         15       12.2837  │
└───────────────────┴─────────────────────────────┘
      rule←f ∆+g ∆
      ⍴b←rule⍤1⊢x
2 3 3
      comp a b
1
```

PRODUCT

```
      ⍴a←(f×g)∆⍤1⊢x
2 3 3
      disp a
┌──────────────┬──────────────────────────┐
│1.00000E¯6 0 0│ ¯2.82886   0       0     │
│8.41471E¯1 0 0│ ¯3.02721 ¯10.1141 0      │
│1.81859E0  0 0│¯19.1785  ¯14.3839 5.5128 │
└──────────────┴──────────────────────────┘
      rule←0 0 1⍉(f∘.×g)∆
      ⍴b←rule⍤1⊢x
2 3 3
      comp a b
1
```

COMPOSITION

```
      ⍴a←(f g)∆⍤1⊢x
2 3 3
      disp a
┌────────────┬──────────────────────────────────┐
│1        0 0│¯0.989993  0          0           │
│0.841471 0 0│ 0.749229 ¯0.092242   0           │
│0.765147 0 0│¯0.718454  0.0884531 ¯0.0302954   │
└────────────┴──────────────────────────────────┘
      rule←(f ∆ g)+.×g ∆
      ⍴b←rule⍤1⊢x
2 3 3
      comp a b
1
```

INVERSE

```
      f←*∘2
      ρa←(f⍣¯1)∆⍤1⊢x+1
2 3 3
      disp a
┌────────────────────┬─────────────────────┐
│0.5 0        0      │0.25 0        0      │
│0   0.353553 0      │0    0.223607 0      │
│0   0        0.288675│0    0        0.204124│
└────────────────────┴─────────────────────┘
      rule←⌹(f ∆ f⍣¯1)
      ρb←rule⍤1⊢x+1
      comp a b
1
```

x+1 to avoid zeros in the argument.

OUTER PRODUCT

```
      f←{ω*2}⍤1 ⋄ g←{1○ω}⍤1
      ρa←(f∘.×g)∆⍤1⊢x
2 3 3 3
      disp a
┌─────────────────────────┬──────────────────────────┬───────────────────────────┐
│1.00000E¯12 0 0          │1 0        0              │4 0       0                │
│8.41471E¯7  0 0          │0 2.22325  0              │0 2.16121 3.36589          │
│9.09297E¯7  0 0          │0 1.8186   ¯0.416148      │0 0       1.9726           │
├─────────────────────────┼──────────────────────────┼───────────────────────────┤
│¯8.06323  0        0     │¯15.8399   1.12896 0      │¯24.7498   0        1.4112 │
│¯4.54082 ¯5.88278 0      │  0      ¯16.5127  0      │  0      ¯16.3411 ¯7.56803 │
│¯5.75355  0        2.55298│  0       ¯7.6714  4.53863│  0        0       ¯2.49762│
└─────────────────────────┴──────────────────────────┴───────────────────────────┘
      rule←(f∘.×g ∆)+0 2 1⍉f ∆∘.×g
      ρb←rule⍤1⊢x
2 3 3 3
      comp a b
0.981481
```

INNER PRODUCT

```
      x←2 3ρ3 1 4 1 5 9
      ρa←(f+.×g)∆⍤1⊢x
2 3
      a
¯8.06323  2.22325 ¯16.5127
 2.22325 ¯2.49762 ¯66.3837
      rule←0 1 tc⍨(f∘.×g)∆
      ρb←rule⍤1⊢x
2 3
      comp a b
1
```

Taylor expansion

The establishment of the derivative rules relies on the Taylor expansion of a function `f` about a value `x`. The complete expansion has terms in all the derivatives but, for our purposes, we are only interested in the first derivative. We're looking for a function that will allow us to write something like:

```
   f x+dx ←→ (f x)+dx{...}f ∆ x
```

Here's an example of the sort of function `f` we might be interested in:

```
      f←{(ssq,mean,max)⍵}⍤2
      ⎕rl←16807
      disp x←?2 5 4⍴9
```

This is a rank `2` function.

3 2 3 0 4 2 1 1 4 8 7 3 1 1 5 7 0 8 8 8	0 5 5 8 1 5 1 6 3 5 2 5 6 0 7 5 3 0 2 4

```
      dx←0.000001×x
      ⎕pp←10
      f x
450 3.8  8
379 3.65 8
```

It is helpful to be able to see more precision.

```
      f x+dx
450.0009   3.8000038  8.000008
379.000758 3.65000365 8.000008
```

So far we have a function `f` which calculates three simple statistics of an array argument. It is of rank `2` and when applied to `x`, which is of rank `3`, produces a result of shape `2 3`.

The rank of the derived function produced as the derivative of `f` is `2`, the same as `f`. With a rank `3` array argument there will be a frame as part of the derivative's result. Therefore, to obtain the correct result, we should apply the derived function `f ∆` as follows:

```
      ⍴a←f ∆⍤2⊢x
2 3 5 4
```

This corresponds to a frame of shape `2`, an individual result of shape `3` and an individual cell of shape `5 4`:

```
      disp 0.01 rnd a
```

6 4 6 0 8 4 2 2 8 16 14 6 2 2 10 14 0 16 16 16	0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1
0 10 10 16 2 10 2 12 6 10 4 10 12 0 14 10 6 0 4 8	0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

The first column has two matrices. These correspond to the variation of the `ssq` value with slight adjustments to the argument `x`. The centre column shows how the mean changes. With `20` values in each matrix, this tells us that a unit change in one element of `x` will produce a `÷20` change in the mean. The last column shows that a slight adjustment in `x` causes a unit change for a maximum value, otherwise zero (i.e. `1`s indicating where the largest values occur).

We've been careful so far to specify the rank for `f ∆`. What does the result look like if we leave this out? This produces a rank `5` array and we need to take a little more care with its display.

```
      ⍴b←f ∆ x
2 3 2 5 4
      disp disp 0.01 rnd b
┌───────────────────────────────────┬──────────────────────────────────┐
│┌───────────────────────┬─────────┐│┌─────────┬───────────────────────┐│
││6   4   6   0          │0 0 0 0  │││0 0 0 0  │ 0  10 10  16          ││
││8   4   2   2          │0 0 0 0  │││0 0 0 0  │ 2  10  2  12          ││
││8  16  14   6          │0 0 0 0  │││0 0 0 0  │ 6  10  4  10          ││
││2   2  10  14          │0 0 0 0  │││0 0 0 0  │12   0 14  10          ││
││0  16  16  16          │0 0 0 0  │││0 0 0 0  │ 6   0  4   8          ││
│├───────────────────────┼─────────┤│├─────────┼───────────────────────┤│
││0.05 0.05 0.05 0.05    │0 0 0 0  │││0 0 0 0  │0.05 0.05 0.05 0.05    ││
││0.05 0.05 0.05 0.05    │0 0 0 0  │││0 0 0 0  │0.05 0.05 0.05 0.05    ││
││0.05 0.05 0.05 0.05    │0 0 0 0  │││0 0 0 0  │0.05 0.05 0.05 0.05    ││
││0.05 0.05 0.05 0.05    │0 0 0 0  │││0 0 0 0  │0.05 0.05 0.05 0.05    ││
││0.05 0.05 0.05 0.05    │0 0 0 0  │││0 0 0 0  │0.05 0.05 0.05 0.05    ││
│├───────────────────────┼─────────┤│├─────────┼───────────────────────┤│
││0 0 0 0                │0 0 0 0  │││0 0 0 0  │0 0 0 1                ││
││0 0 0 0                │0 0 0 0  │││0 0 0 0  │0 0 0 0                ││
││0 1 0 0                │0 0 0 0  │││0 0 0 0  │0 0 0 0                ││
││0 0 0 0                │0 0 0 0  │││0 0 0 0  │0 0 0 0                ││
││0 1 1 1                │0 0 0 0  │││0 0 0 0  │0 0 0 0                ││
│└───────────────────────┴─────────┘│└─────────┴───────────────────────┘│
└───────────────────────────────────┴──────────────────────────────────┘
```

This shows twelve (i.e. `×/¯2↓⍴b`) `5` by `4` matrices. The six matrices highlighted in yellow are the same as those created by `a←f ∆⍤2⊢x`; the other six are entirely zero. The zero matrices come about when `∆` tries to find the sensitivity of the result of one cell to variations in a different cell. In a sense they are wasted as they produce no new information and are always zero.

How do we produce an expression for a first order Taylor expansion for the example introduced above? Bear in mind that we are looking for something of the form:

```
     f x+dx ←→ (f x)+dx{...}f ∆⍤2⊢x
```

Note that:

- An incremental amount `dx{...}f ∆⍤2⊢x` is to be added to the result of `f x`. This is of shape `2 3`.
- The differential amount `dx` is of shape `2 5 4`.
- The derivative `f ∆⍤2`, evaluated at x, is of shape `2 3 5 4`.

A suitable expression is not too difficult to find with an appropriate use of the rank operator. The Taylor expansion to first order is then:

```
      k←2-⍴⍴x
      (f x)+dx tip⍤k⊢f ∆⍤2⊢x
```

```
450.0009   3.8000038  8.000032
379.000758 3.65000365 8.000008
```

which compares with:

```
      f x+dx
450.0009   3.8000038  8.000008
379.000758 3.65000365 8.000008
```

So, for this example where `(ρρx)>k`, the expression for the Taylor expansion of `f` about `x` is:

```
      f x+dx ↔ (f x)+dx tip⍤(k-ρρx)⊢f ∆⍤k⊢x
```

and where `(ρρx)≤k`, the expression is:

```
      f x+dx ↔ (f x)+dx tip f ∆ x
```

Fortunately, it is possible to combine these two rules into one, by using the rank operator to limit the rank of the data values. So, if `k` is the rank of the function `f`:

```
      f x+dx ↔ (f x)+dx{⍺ tip f ∆ ⍵}⍤k⊢x
```

The example now becomes:

```
      (f x)+dx {⍺ tip f ∆ ⍵}⍤2⊢x
450.0009   3.8000038  8.000032
379.000758 3.65000365 8.000008
      comp (f x+dx) ((f x)+dx{⍺ tip f ∆ ⍵}⍤2⊢x)
1
      ⎕pp←6
```

Derivation of the Rules

Strict conformance

APL has rules about how the arguments to functions must conform. For example, if we wish to add two arrays together, they need to be of the same shape. Of course that's the "strict" rule and there is an exception. In most cases, if one of the arguments `x` is a single value (`1≡≢x`), it is repeated as necessary to match the other argument. So, we can write:

```
      3+⍳5
3 4 5 6 7
      (1 1 1 1⍴3)+⍳5
3 4 5 6 7
      (4 9⍴⍳36)+.×2
72 234 396 558
```

This is very handy but complicates considerably the discussion of the derivative rules. We won't allow it, instead assuming strict conformance.

Sum (and difference) rule

For two functions `f` and `g`, of rank `s` and `t` respectively, consider the difference in the value of the function `(f+g)` at two points infinitesimally close together, `x` and `x+dx`. The train `f+g` is of unbounded rank, simply passing its argument `x` to both `f` and `g`. In order to conform, the shapes of the results of `f` and g must match.

```
      ((f+g)x+dx)-(f+g)x
»     ((f x+dx)+g x+dx)-(f x)+g x
```

As shorthand, let's write `fx←f x`, `gx←g x`, `dfx←dx{α tip f ∆ ⍵}⍤s⊢x` and `dgx←dx{α tip g ∆ ⍵}⍤t⊢x`.

```
»     fx+gx+dfx+dgx-fx+gx
```

Using the first order Taylor expansions for both `f` and `g` about `x`.

```
»     (dx{α tip f ∆ ⍵}⍤s⊢x)+dx{α tip g ∆ ⍵}⍤t⊢x
```

This general expression cannot be simplified further unless we require that `s≡t` and that both `s` and `t` are not less than the rank of the argument `x`. When this is so, we have:

```
»     (dx tip f ∆ x)+dx tip g ∆ x
»     dx tip(f ∆ x)+g ∆ x
```

As `(a tip b)+a tip c ←→ a tip b+c`

```
»     dx tip(f ∆+g ∆)x
```

As this is true for arbitrary `x` and `dx`:

```
(f+g)∆ ←→ (f ∆+g ∆)
s≥⍴⍴x
(f-g)∆ ←→ (f ∆-g ∆)
```

with `s≡t` and

For example:

```
      x←2 3 4⍴3 1 4 2 7 1
      f←{⍵*2}⍤1 ⋄ g←{⍵*3}⍤1
      ⍴a←(f+g)∆⍤1⊢x
2 3 4 4
      rule←f ∆+g ∆
      ⍴b←rule⍤1⊢x
2 3 4 4
      comp a b
1
```

Product rule

For two functions `f` and `g`, of rank `s` and `t` respectively, consider the difference in the value of the function `(f×g)` at two points infinitesimally close together, `x` and `x+dx`. The train `f×g` is of unbounded rank, simply passing its argument `x` to both `f` and `g`. In order to conform, the shapes of the results of `f` and g must match.

```
      ((f×g)x+dx)-(f×g)x
»     ((f x+dx)×g x+dx)-(f x)×g x
```

As shorthand, let's write `fx←f x`, `gx←g x`, `dfx←dx{⍺ tip f ∆ ⍵}⍤s⊢x` and `dgx←dx{⍺ tip g ∆ ⍵}⍤t⊢x`.

`» ((fx+dfx)×gx+dgx)-fx×gx`	Using the first order Taylor expansion for both `f` and `g` about `x`.
`» (fx×gx)+(fx×dgx)+(dfx×gx)+(dfx×dgx)-fx×gx`	Expanding the product of two sums.
`» (fx×dgx)+dfx×gx`	Cancelling the first and last terms and ignoring the 2nd order term in `dx`.
`» ((f x)×dx{⍺ tip g ∆ ⍵}⍤t⊢x)` `+(g x)×dx{⍺ tip f ∆ ⍵}⍤s⊢x`	Exchanging the arguments to `×` in the second term.

This general expression cannot be simplified further unless we require that `s≡t` and that both `s` and `t` are not less than the rank of the argument `x`. When this is so, we have:

`» ((f x)×dx tip g ∆ x)+(g x)×dx tip f ∆ x`	
`» (dx tip (f x) xp g ∆ x)+dx tip (g x) xp f ∆ x`	As `a×b tip c ←→ b tip a xp c`
`» dx tip((f x) xp g ∆ x)+(g x) xp f ∆ x`	Combining terms with: `(a tip b)+a tip c ←→ a tip b+c`
`» dx tip((f×g ∆)+g×f ∆)x`	

As this is true for arbitrary `x` and `dx`:

> `(f×g)∆ ←→ (f xp g ∆)+g xp f ∆` with `s≡t` and `s≥⍴⍴x`

If `s<⍴⍴x`, `f` and `g` both create frames and we must apply the rule with an explicit rank: `((f xp g ∆)+g xp f ∆)⍤s`. We can test this with the sample values for `f`, `g` and `x` from above:

```
      ρa←(f×g)∆⍤1⊢x
2 3 4 4
      rule←(f xp g ∆)+g xp f ∆
      ρb←rule⍤1⊢x
2 3 4 4
      comp a b
1
```

Quotient rule

For two functions `f` and `g`, of rank `s` and `t` respectively, consider the difference in the value of the function `(f÷g)` at two points infinitesimally close together, `x` and `x+dx`. The train `f÷g` is of unbounded rank, simply passing its argument `x` to both `f` and `g`. In order to conform, the shapes of the results of `f` and g must match.

	`((f÷g)x+dx)-(f÷g)x`	
»	`((f x+dx)÷g x+dx)-(f x)÷g x`	As `(f÷g)x ↔ (f x)÷g x`

As shorthand, let's write `fx←f x`, `gx←g x`, `dfx←dx{α tip f ∆ ω}⍤s⊢x` and `dgx←dx{α tip g ∆ ω}⍤t⊢x`.

»	`((fx+dfx)÷gx+dgx)-fx÷gx`	Using the first order Taylor expansion for both `f` and `g` about `x`.
»	`((fx+dfx)÷gx×1+dgx÷gx)-fx÷gx`	
»	`(((fx÷gx)+dfx÷gx)÷1+dgx÷gx)-fx÷gx`	
»	`(((fx÷gx)+dfx÷gx)×1-dgx÷gx)-fx÷gx`	As `{÷1+ω} ↔ {1-ω}` for small arguments.
»	`((fx÷gx)+(dfx÷gx)-((fx÷gx)×dgx÷gx)+dfx×dgx÷gx×gx)-fx÷gx`	
»	`((dfx÷gx)-(fx÷gx)×dgx÷gx)+dfx×dgx÷gx×gx`	Cancelling the first and last terms.
»	`(dfx÷gx)-(fx÷gx)×dgx÷gx`	Ignoring the 2nd order term in `dx`.
»	`((dx{α tip f ∆ ω}⍤s⊢x)÷gx)-(fx÷gx)×(dx{α tip g ∆ ω}⍤t⊢x)÷gx`	

This general expression cannot be simplified further unless we require that `s≡t` and that both `s` and `t` are not less than the rank of the argument `x`. When this is so, we have:

»	`((dx tip f ∆ x)÷gx)-(fx÷gx)×(dx tip g ∆ x)÷gx`	
»	`(dx tip (f ∆ x)xp÷gx)-(fx÷gx)×dx tip (g ∆ x)xp÷gx`	
»	`(dx tip (f ∆ x)xp÷gx)-dx tip (fx÷gx)xp(g ∆ x)xp÷gx`	As `a×b tip c ↔ b tip a xp c`
»	`dx tip ((f ∆ x)xp÷gx)-(fx÷gx)xp(g ∆ x)xp÷gx`	As `(a tip b)-a tip c ↔ a tip b-c`
»	`dx tip ((f ∆ x)-(fx÷gx)xp g ∆ x)xp÷gx`	
»	`dx tip (((f ∆)-(f÷g)xp g ∆)xp(÷g))x`	

As this is true for arbitrary `x` and `dx`:

`(f÷g)∆ ↔ ((f ∆)-(f÷g)xp g ∆)xp(÷g)`	with `s≡t` and `s≥ρρx`

For example:

```
      f←{ω+2} ⋄ g←{ω*3}
      (f÷g)∆⍤0⊢3 4 5
¯0.148148 ¯0.0546874 ¯0.0256
      rule←((f ∆)-(f÷g)xp g ∆)xp(÷g)
      rule⍤0⊢3 4 5
¯0.148148 ¯0.0546874 ¯0.0256
```

Outer product rule

The outer product `(f∘.×g)` is defined for monadic functions `f` and `g` of rank `s` and `t` respectively. It is of unbounded rank, producing a result without a frame for an argument of any rank. Consider the difference in the value of the function `(f∘.×g)` at two points infinitesimally close together, `x` and `x+dx`.

```
      ((f∘.×g)x+dx)-(f∘.×g)x
»     ((f x+dx)∘.×g x+dx)-(f x)∘.×g x
```

As shorthand, let's write `fx←f x`, `gx←g x`, `dfx←dx{α tip f ∆ ω}⍤s⊢x` and `dgx←dx{α tip g ∆ ω}⍤t⊢x`.

»	`((fx+dfx)∘.×gx+dgx)-fx∘.×gx`	Approximating `f x+dx` and `g x+dx` as first order Taylor expansions about `x`.
»	`(fx∘.×gx)+(fx∘.×dgx)+(dfx∘.×gx)+(dfx∘.×dgx)` `-fx∘.×gx`	Expanding `(a+b)∘.×c+d` as `(a∘.×c)+(a∘.×d)+(b∘.×c)+b∘.×d`
»	`(fx∘.×dgx)+dfx∘.×gx`	Cancelling the first and last terms and eliminating the second order term in `dx`.

At this point it's useful to analyze the shapes of the two terms in this expression. Let's first define some variables to use for the various shapes we'll encounter.

Object	of shape	where

`fx←f x`	`ff,rf`	`sx←⍴x`
		`ff←(-s)↓sx`
		`rf←⍴f⊃s cells x`

`dfx←f ∆⍤s⊢x`	`ff,rf,cf`	`cf←(-s)↑sx`
`gx←g x`	`fg,rg`	`fg←(-t)↓sx` `rg←⍴g⊃t cells x`
`dgx←g ∆⍤t⊢x`	`fg,rg,cg`	`cg←(-t)↑sx`

Here are the two terms that make up the current expression:

`fx∘.×dgx`	`dfx∘.×gx`
`fx∘.×dx{α tip g ∆ ω}⍤t⊢x`	`(dx{α tip f ∆ ω}⍤s⊢x)∘.×gx`
`ff,rf∘.×sx{α tip g ∆ ω}⍤t⊢sx`	`(sx{α tip f ∆ ω}⍤s⊢sx)∘.×fg,rg`
`ff,rf∘.×fg,(cg tip g ∆ cg)`	`(ff,(cf tip f ∆ cf))∘.×fg,rg`
`ff,rf∘.×fg,(cg tip rg,cg)`	`(ff,(cf tip rf,cf))∘.×fg,rg`
`ff,rf∘.×fg,rg`	`ff,rf∘.×fg,rg`
`ff,rf,fg,rg`	`ff,rf,fg,rg`

As we'd expect, both produce objects of the same shape which can be added together.

In order to derive a rule for the derivative of the outer product, we'd like to combine these two terms into one of the form `dx tip{rule ω}`. If we can construct this, we'll be able to assert that as `dx` and `x` are quite general, `rule` must hold for the derivative of the outer product. Unfortunately, this is not possible in general. We need to impose the restrictions seen in several of the other derivative rules The functions `f` and `g` need to be of the same rank (`s=t`) and they should only be applied where they do not produce a frame. This means that the variables we're using for the shapes of objects need to be redefined.

Object	of shape	where

`fx←f x`	`rf`	`sx←⍴x`
		`ff←θ`
		`rf←⍴f x`

`dfx←f ∆ x`	`rf,sx`	`cf←sx`
`gx←g x`	`rg`	`fg←θ` `rg←⍴g x`
`dgx←g ∆x`	`rg,sx`	`cg←sx`

The expression for the derivative now becomes:

```
»    (fx∘.×dx tip g ∆ x)+(dx tip f ∆ x)∘.×gx
      rf    sx      rg,sx    sx      rf,sx    rg
```

We can put the first term in the desired form as `a∘.×b tip c ↔ b tip a∘.×c`:

```
»    (dx tip fx∘.×g ∆ x)+(dx tip f ∆ x)∘.×gx
      sx      rf    rg,sx    sx      rf,sx    rg
```

The first term is `rf,rg`. Unfortunately, a rearrangement of the second term to `dx tip (f ∆ x)∘.×gx` is no good as it calls for `sx tip rf,sx,rg` which fails in general. Luckily this can be fixed by including a transpose to get the axes in the right order, as follows:

```
     order←{⍋⍋((⍴⍴f ω),(⍴⍴ω),⍴⍴g ω)/0 2 1}
»    dx tip fx∘.×g ∆ x)+(order x)⍉(f ∆ x)∘.×gx
```

```
»    dx tip((f∘.×g ∆)+order⍉f ∆∘.×g)x
```

As this is true for arbitrary `dx` we have the following rule for the derivative of an outer product where the ranks of `f` and `g` are the same and neither `f` or `g` produces a frame with the supplied argument:

`(f∘.×g)∆ ←→ (f∘.×g ∆)+order⍉f ∆∘.×g`	with `s≡t` and `s≥⍴⍴x`

Note that:

- If `f` and `g` are vector functions and `x` is a vector, the rule simplifies to `(f∘.×g ∆)+0 2 1⍉f ∆∘.×g` (as per Iverson [0]).
- Although the rule can only be used where its execution does not produce a frame, higher rank arguments can still be handled by applying the rule with `⍤` operator.

Here are two examples:

Both `f` and `g` have the same rank which is equal to `⍴⍴x`.

```
      x←?6 7⍴9
      f←{(ssq,mean,max)⍵}⍤2
      g←{+/,⍵}⍤2
      ⍴a←(f∘.×g)∆ x
3 6 7
      rule←(f∘.×g ∆)+order⍉f ∆∘.×g
      ⍴b←rule x
3 6 7
      comp a b
1
```

Both `f` and `g` have the same rank which is less than `⍴⍴x`.

```
      x←?5 6 7⍴9

      ⍴a←(f∘.×g)∆⍤2⊢x
5 3 6 7
      ⍴b←rule⍤2⊢x
5 3 6 7
      comp a b
1
```

Composition – the chain rule

The rank of a composition

The composition `(f g)` is, by definition, of unbounded rank. It accepts arguments of any rank, simply passing them unchanged to `g`. It does not produce a frame and its result is just `f g x`. Naturally, along the way, `f` and `g` may treat the values they receive as cells within frames but that just gives structure to the result and does not influence the rank of the composition.

Of course, the composition can be assigned a specified rank as `(f g)⍤k` for example. In that case, the result may be made up of a frame of individual results.

THE SHAPES OF A COMPOSITION AND ITS DERIVATIVE

Consider a composition of functions `f` and `g` having ranks `s` and `t` respectively.

The application of `g` to an argument `x` of shape `sx` produces a result having a shape made up of two parts. One part has the shape of the result obtained by applying `g` to an individual `t`-cell of `x`. Let's call that `rg`. The other part is the frame `fg` left after the final `t` axes are dropped from `sx`. The final shape of `g`'s result is `fg,rg`.

The derivative `g ∆⍤t⊢x` has the same frame `fg` but the shape of its individual result is `rg,cg` where `cg` is the shape of a `t`-cell of `x`.

`f` operates in much the same way producing a result with a frame of shape `ff` and an individual result of shape `rf`. However, the individual result comes about by the application of `f` to an `s`-cell of the result of `g x`. The frame `ff` is just the axes left after the final `s` are dropped from the shape of `g x`. The final shape of `f`'s result is `ff,rf`.

The derivative `f ∆⍤s⊢g x` has the same frame `ff` but the shape of its individual result is `rf,cf` where `cf` is the shape of an `s`-cell of `g x`.

The shape of the result of `(f g)` at `x` is `ff,rf`.

The derivative of the composition `(f g) ∆ x` is unbounded. It does not produce a frame. Its result is of shape `ff,rf,sx`.

The following table summarizes the definitions:

Object	of shape	where

`g x`	`fg,rg`	`sx←⍴x`
		`fg←(-t)↓sx`
		`rg←⍴g⊃t cells x`

`g ∆⍤t⊢x`	`fg,rg,cg`	`cg←⍴⊃t cells x`
`f g x`	`ff,rf`	`ff←(-s)↓fg,rg` `rf←⍴f⊃s cells g x`
`f∆⍤s⊢g x`	`ff,rf,cf`	`cf←⍴⊃s cells g x`
`(f g)∆ x`	`ff,rf,sx`	

THE RULE

We need to start with the difference in the value of the composition `(f g)` at two points infinitesimally close together, `x` and `x+dx`:

```
(f g x+dx)-f g x
```

The first step is to express `g x+dx` as a first order Taylor expansion of `g` about `x`. Writing `fx←f x`, `gx←g x`, `dfx←dx{α tip f ∆ ω}⍤s⊢x` and `dgx←dx{α tip g ∆ ω}⍤t⊢x` we have:

```
»   (f gx+dgx)-f g x
```

The next step is to express `f gx+dgx` as a first order Taylor expansion of `f` about `gx`.

```
»    (f gx)+(dgx{α tip f ∆ ω}⍤s⊢gx)-f gx
»    dgx{α tip f ∆ ω}⍤s⊢gx
```
Cancelling first and last terms.

We have now arrived at something of a challenging expression. Before we go further, let's do a shape analysis.

```
(dx{α tip g ∆ ω}⍤t⊢x){α tip f ∆ ω}⍤s⊢g x
(sx{α tip g ∆ ω}⍤t⊢sx) {α tip f ∆ ω}⍤s⊢g x
(fg,(cg tip g ∆ cg)){α tip f ∆ ω}⍤s⊢g x
(fg,(cg tip rg,cg){α tip f ∆ ω}⍤s⊢g x
fg,rg{α tip f ∆ ω}⍤s⊢fg,rg
ff,rf
```

This checks out – we started with `(f g x+dx)-f g x` which is also `ff,rf`.

In order to find a simplifying rule of the form `dx tip {...} x` we need to avoid the frames potentially produced by `{α tip g ∆ ω}⍤t` and `{α tip f ∆ ω}⍤s`. We can do this by imposing the restrictions `s≥⍴⍴g x` and `t≥⍴⍴x`. Then we have:

```
»    (dx tip g ∆ x)tip f ∆ gx
»    (f ∆ gx)tip dx tip g ∆ x
```
Commuting the arguments to `tip`.

Writing `n←⍴⍴g x`, this becomes:

```
»    dx tip (f ∆ g x)(n mp)g ∆ x
```

As this is true for arbitrary `dx` we have the following rule for the derivative of a composition:

`(f g)∆ ←→ (f ∆ g)(n mp)g ∆`	`n←⍴⍴g x`
`←→ (f ∆ g)+.×g ∆`	for a vector function `g`
`←→ (f ∆ g)×g ∆`	for a scalar function `g`

Scalar functions

Here's an extreme example where there will be frames and plenty of surplus zeros. `f` and `g` are rank `0` functions returning scalars as their results.

```
    f←{ω×3} ⋄ g←{ω*2}
    dx←0.000001×x←?5 6 7⍴9
```

In the evaluation of the composition, `g` acts on cells of rank `0` from `x` producing a rank `3` frame of scalars. `f` acts on that result, producing a further rank `3` frame of scalars. The individual result for the composition is of shape `5 6 7` and the derivative becomes of shape `5 6 7 5 6 7`, as follows:

```
    ⍴a←(f g)∆ x
5 6 7 5 6 7
```

This array has `44100` elements, most of which are surplus zeros:

```
      ×/⍴a
44100
      mean a=0
0.995238
      sum a≠0
210
```

Surplus zeros fill 209 of the 210 sub-arrays of shape 5 6 7.

If we wish to avoid frames and the surplus zeros, we need to apply the derivative with rank 0.

```
      ⍴b←(f g)∆⍤0⊢x
5 6 7
```

This produces the compact value for the derivative. Note that b is identical to a with the zeros omitted:

```
      (,b)≡(,a)~0
1
```

As long as we are applying the derivative with rank 0, we can make use of the simpler rule for the derivative of a composition, as follows:

```
      rule←(f ∆ g)×g ∆
      ⍴c←rule⍤0⊢x
5 6 7
      (b≡c),comp b c
0 1
      {dx tip ⍵}¨b c
0.02682 0.02682
```

as, for scalars, 0 mp ←→ ×

Not identical, but close.

Vector functions

```
      f←{⍵*3}⍤1 ⋄ g←{⍵+.×d}⍤1
      dx←0.000001×x←?5 6 3⍴9
      d←?3 8⍴9
```

Here, g acts on cells which are vectors (of length 3, to conform with d) producing a vector of length 8 for each. These are assembled in a frame of shape 5 6 and the result is passed to f. f acts on vectors ultimately producing an individual result of shape 5 6 8. For the derivative, the result has no frame and is of shape 5 6 8 5 6 3.

```
      ⍴(f g)∆ x
5 6 8 5 6 3
```

In order to produce a denser derivative, we should apply the derivative with rank 1.

```
      ⍴a←(f g)∆⍤1⊢x
5 6 8 3
      rule←(f ∆ g)+.×g ∆
      ⍴b←rule⍤1⊢x
5 6 8 3
      (a≡b),comp a b
0 0.998611
```

as +.× ←→ 1 mp

Not identical, but very close.

Matrix multiplication

Iverson [0] shows the derivative of the matrix product between vector functions as a tensor contraction of the derivative of the outer product:

```
(f+.×g)∆ ←→ 0 1 tc⍨(f∘.×g)∆
```

In this case, `f` and `g` both produce vector results and the appropriate contraction is on the first two axes. It's possible to generalize this for higher rank results from `f` and g but we won't pursue that here. Instead we'll go back to first principles and derive a rule for the derivative of a matrix product.

Consider a matrix product of functions `f` and `g` having ranks `s` and `t` respectively. The function `(f+.×g)` is itself of unbounded rank. It merely passes its argument to both `f` and `g`.

The application of `g` to an argument `x` of shape `sx` produces a result having a shape made up of two parts. One part has shape `rg` of the result obtained by applying `g` to an individual `t`-cell of `x`. The other part is the frame `fg` left after the final `t` axes are dropped from `sx`. The final shape of `g`'s result is `fg,rg`. The derivative `g ∆⍤t⊢x` has the same frame `fg` but the shape of its individual result is `rg,cg` where `cg` is the shape of a `t`-cell of `x`.

`f` operates in much the same way producing a result with a frame of shape `ff` and an individual result of shape `rf`. The derivative `f ∆⍤s⊢g x` has the same frame `ff` but the shape of its individual result is `rf,cf` where `cf` is the shape of an `s`-cell of `x`.

The following table summarizes these relationships:

Object	of shape	where

`g x`	`fg,rg`	`sx←⍴x`
		`fg←(-t)↓sx`
		`rg←⍴g⊃t cells x`

`g ∆⍤t⊢x`	`fg,rg,cg`	`cg←⍴⊃t cells x`
`f x`	`ff,rf`	`ff←(-s)↓sx` `rf←⍴f⊃s cells x`
`f∆⍤s⊢g x`	`ff,rf,cf`	`cf←⍴⊃s cells x`
`(f+.×g)x`	`(¯1↓ff,rf),1↓fg,rg`	`(¯1↑ff,rf)≡1↑fg,rg`
`(f+.×g)∆ x`	`(¯1↓ff,rf),(1↓fg,rg),sx`	

The matrix product `(f+.×g)` requires that the last axis of `f x` match the first axis of `g x`. The shape of the result is `(¯1↓ff,rf),1↓fg,rg`.

THE RULE

Starting with the difference in the value of `(f+.×g)` at two points infinitesimally close together, `x` and `x+dx`, we have:

```
    ((f+.×g)x+dx)-(f+.×g)x
»   ((f x+dx)+.×g x+dx)-(f x)+.×g x
```

Applying the definition of `f+.×g`.

As shorthand, let's write `fx←f x`, `gx←g x`, `dfx←dx{α tip f ∆ ω}⍤s⊢x` and `dgx←dx{α tip g ∆ ω}⍤t⊢x`.

```
»   ((fx+dfx)+.×gx+dgx)-fx+.×gx
```

Using the first order Taylor expansions of `f` and `g` about `x`.

```
»   (fx+.×gx)+(fx+.×dgx)+(dfx+.×gx)+(dfx+.×dgx)
    -fx+.×gx
```

Expanding terms, using: `(a+b)+.×c+d ←→ (a+.×c)+(b+.×c)+(a+.×d)+(b+.×d)`

```
»   (fx+.×dgx)+dfx+.×gx
```

Cancelling first and last terms and ignoring the second order term in `dx`.

Here's an analysis of the shapes of the two terms in the line above:

`fx+.×dgx`
`fx+.×dx{α tip g ∆ ω}⍤t⊢x`
`ff,rf+.×sx{α tip g ∆ ω}⍤t⊢sx`
`ff,rf+.×fg,(cg tip g ∆ cg)`
`ff,rf+.×fg,rg`
`(¯1↓ff,rf),1↓fg,rg`
	`dfx+.×gx`
	`(dx{α tip f ∆ ω}⍤s⊢x)+.×gx`
	`(sx{α tip f ∆ ω}⍤s⊢sx)+.×fg,rg`
	`(ff,(cf tip f ∆ cf))+.×fg,rg`
	`ff,rf+.×fg,rg`
	`(¯1↓ff,rf),1↓fg,rg`

If we are to find a rule of the form `dx tip ...` the first term needs to become something like `dx tip fx+.×{g ∆ ω}⍤t⊢x`. Here's a shape analysis for this possibility:

`dx tip fx+.×{g ∆ ω}⍤t⊢x`
`sx tip fx+.×fg,(g ∆ cg)`
`sx tip ff,rf+.×fg,rg,cg`

The only way this can produce `ff,rf+.×fg,rg` is if `cg≡sx`. This means that `g` produces no frame, `fg` is empty and the entire argument is a single cell, i.e. `t≥⍴⍴x`. Accepting this restriction, the expression then becomes:

```
»   (dx tip fx+.×g ∆ x)+(dx{α tip f ∆ ω}⍤s⊢x)+.×gx
```

The second term also needs to be transformed into something of the form `dx tip` ... but this is not as simple as just removing the parentheses. That would only work if `tip` and `+.×` were associative. The way forward can be seen by looking at the shapes of the following possible expression:

```
dx{α tip (f ∆ ω)+.×gx}⍤s⊢x
sx{α tip (f ∆ ω)+.×gx}⍤s⊢sx
ff,(cf tip (f ∆ cf)+.×rg)
(ff,(cf tip rf,cf+.×rg)
```

As it stands, this expression does not work. The problem comes with trying to form the inner product `rf,cf+.×rg`. In general, `cf` and `rg` do not conform. Fortunately, there is a way out. We can rearrange the order of the calculations by first shifting the axes of `f ∆ cf` to be `cf,rf` which then conforms with `g x`. Lastly we need to undo that shift for the `tip` evaluation with `dx`. The degree of the shift is just `s`. Let's redo the shape analysis for an improved expression.

```
dx{α tip (-s)sh(s sh f ∆ ω)+.×gx}⍤s⊢x
sx{α tip (-s)sh(s sh f ∆ ω)+.×gx}⍤s⊢sx
ff,(cf tip (-s)sh(s sh f ∆ cf)+.×rg)
ff,(cf tip (-s)sh(s sh rf,cf)+.×rg)
ff,(cf tip (-s)sh(cf,rf)+.×rg)
ff,(cf tip (-s)sh cf,(¯1↓rf),1↓rg)
ff,(cf tip ((¯1↓rf),1↓rg),cf)
ff,(¯1↓rf),1↓rg
```

We now have:

```
»    (dx tip(f x)+.×g ∆ x) + dx{α tip(-s)sh(s sh f ∆ ω)+.×g x}⍤s⊢x
```

One last step remains. In order to be able to combine these two terms we need to avoid the application of the rank operator in the second term. We need to impose the further restriction that `s≥⍴⍴x`. Finally we have:

```
»    dx tip(fx+.×g ∆ x)+(-⍴⍴x)sh((⍴⍴x)sh f ∆ x)+.×gx
```

Combining terms with:
`(a tip b)+a tip c ↔ a tip b+c`

The rule for the derivative of a matrix product between functions whose rank is greater than or equal to that of the argument is then:

```
(f+.×g)∆  ↔  (f+.×g ∆)+{(-⍴⍴ω)sh((⍴⍴ω)sh f ∆ ω)+.×g ω}
```

Here's an example:

```
      f←{ω∘.*2 3 4}⍤1
      g←{6 7 8∘.-ω,5}⍤1
```

As both `f` and `g` are rank `1` functions, there is no difficulty getting the derivative with a vector argument:

```
      ⍴a←(f+.×g)∆ x←?7⍴9
7 8 7
```

However, with a matrix argument, we can't even evaluate the train or its derivative:

```
      (f+.×g)x←?2 4⍴9                                      LENGTH ERROR
```

Fortunately, we can deal with this by applying the derivative with the rank operator:

```
      ⍴a←(f+.×g)∆⍤1⊢x
2 4 5 4
```

Let's now check whether the rule produces the correct result:

```
      rule←(f+.×g ∆)+{(-⍴⍴⍵)sh((⍴⍴⍵)sh f ∆ ⍵)+.×g ⍵}
      ⍴b←rule⍤1⊢x
2 4 5 4
      comp a b
1
```

Inverse

Let’s assume that we have a function `f` with an inverse `fi`. By this we mean that the compositions (`f fi`) and (`fi f`) both act as the identity function simply returning their argument unchanged.

We'll confine ourselves to applications of the functions `f` and `fi` to arguments of small enough rank that they do not produce frames and we'll assume that `f x` and `fi x` produce results of the same shape `sx←⍴x`. So what should we expect for the derivatives of `f` and `fi`?

As there are no frames to consider, the results of `f ∆ x` and `g ∆ x` will both be of shape `sx,sx`. For example:

```
      f←{(1⌽⍵)*2}⍤1 ⋄ fi←{¯1⌽⍵*0.5}⍤1
      x←3 1 5 2 6 8
      x≡f fi x
1
      x≡fi f x
1
      ⍴a←f ∆ x
6 6
      disp a
┌─────────────────┐
│0 2  0 0  0  0   │
│0 0 10 0  0  0   │
│0 0  0 4  0  0   │
│0 0  0 0 12  0   │
│0 0  0 0  0 16   │
│6 0  0 0  0  0   │
└─────────────────┘
   ⍴b←fi ∆ x
6 6
   disp b
┌──────────────────────────────────────────────────┐
│0        0   0        0        0        0.176777  │
│0.288675 0   0        0        0        0         │
│0        0.5 0        0        0        0         │
│0        0   0.223607 0        0        0         │
│0        0   0        0.353553 0        0         │
│0        0   0        0        0.204124 0         │
└──────────────────────────────────────────────────┘
```

Let's start by considering the derivative of (`f fi`):

```
(f fi)∆ x ←→ {ω}∆ x ←→ id sx
```

Alternatively, we know from the result for the derivative of a composition that:

```
(f fi)∆ x ←→ (f ∆ fi x)(n mp)fi ∆ x                    where n←ρsx
```

Equating these, we have:

```
id sx ←→ (f ∆ fi p)(n mp)fi ∆ x
```

This expression holds for any array `x` as long as its rank is not greater than `k`. However, continuing to find an expression for `fi ∆` is only possible if `f` is of rank `0` and `x` is a scalar or `f` is of rank `1` and `x` is a scalar or a vector. When these conditions hold, we can transform the equivalence by applying the matrix inverse of `f ∆ fi x` to both sides. Then,

```
(⌹f ∆ fi x)+.×id sx ←→ (⌹f ∆ fi x)+.×(f ∆ fi x)+.× fi ∆ x
```

We can simplify the left hand side to `⌹f ∆ fi x` as `id sx` is an identity array for the inner product with an array of shape `sx,sx`; and the right hand side can be simplified to `fi ∆ x` as `+.×` is associative. This gives the following rule for the derivative of the inverse of scalar and vector functions:

```
fi ∆  ←→  ⌹(f ∆ fi)
```

This equation is good for both vector and scalar functions as matrix inverse `⌹` is equivalent to reciprocal `÷` for scalars. For functions of higher rank, we need an "extended inverse" `xinv` to replace the `⌹` used above. This function needs to satisfy:

```
id ρp ←→ (xinv f ∆ fi x)((ρρx)ip)f ∆ fi x
```

Unfortunately, I'm not aware of a definition for `xinv`. Submissions, of course, are welcome.

Appendix A
Summary of the derivative rules

Name	Definition	Rule	Note
Taylor expansion	`f x+dx`	`(f x)+dx{α tip f ∆ ω}⍤rf⊢x`	`rf` is the rank of `f`
Sum	`(f+g)∆`	`f ∆ + g ∆`	
Difference	`(f-g)∆`	`f ∆ - g ∆`	
Product	`(f×g)∆`	`(f xp g ∆)+g xp f ∆`	
Quotient	`(f÷g)∆`	`((f ∆)-(f÷g)xp g ∆)xp(÷g)`	
Outer Product	`(f∘.×g)∆`	`(f∘.×g ∆)+order⍉f ∆∘.×g`	
Composition	`(f g)∆`	`(f ∆ g)(n mp)g ∆`	`m←⍴⍴g x`
Inverse	`fi ∆`	`⌹(f ∆ fi)`	
Matrix multiplication	`(f+.×g)∆`	`(f+.×g ∆)+(-n)sh(n sh f ∆)+.×g`	`n←⍴⍴x`

Appendix B
Supporting Operators and Functions

Shuffle

```
sh←{(⍺⌽⍳⍴⍴⍵)⍉⍵}
```

sh uses a dyadic transpose to shuffle the specified number of axes to the right. For example:

```
      a←3 4 5 6 7⍴9
      ⍴2 sh a
6 7 3 4 5
      ⍴¯2 sh a
5 6 7 3 4
```

Diagonality and the idem function

Diagonal matrices should be familiar. For example, the following 3 by 3 matrix is diagonal:

```
   3 0 0
   0 1 0
   0 0 4
```

It has values on the diagonal but is zero elsewhere. Note that it is necessarily square and of even rank.

It's possible to extend this concept to arrays of other ranks. Scalars are always diagonal but vectors never (because they are of odd rank). Square arrays m of higher rank are diagonal if the only non-zero values (i,j)⌷m occur at indices i,j with i≡j. Of particular interest are diagonal arrays whose diagonal elements are all 1. These can be generated as follows:

```
      id←{(⍵,⍵)⍴1,(×/⍵)⍴0}
      id 3
1 0 0
0 1 0
0 0 1
      ⍴id 2 3
2 3 2 3
      disp id 2 3
┌─────┬─────┬─────┐
│1 0 0│0 1 0│0 0 1│
│0 0 0│0 0 0│0 0 0│
├─────┼─────┼─────┤
│0 0 0│0 0 0│0 0 0│
│1 0 0│0 1 0│0 0 1│
└─────┴─────┴─────┘
      ,⍸id 2 3
┌───────┬───────┬───────┬───────┬───────┬───────┐
│0 0 0 0│0 1 0 1│0 2 0 2│1 0 1 0│1 1 1 1│1 2 1 2│
└───────┴───────┴───────┴───────┴───────┴───────┘
```

The positions of 1s in the 2 3 2 3 array

Tensor contraction

`tc` is the tensor contraction function (see [0] and [1]). Its left argument is an array and its right argument a vector of pairs of axes. The result is the array contracted on all pairs of axes specified. The definition is:

```
tc←{m←⍴⍴⍺ ⋄ p←0.5×n←⍴⍵ ⋄ q←m-n ⋄ r←⍳m ⋄ ((⊂r~⍵)⌷r)←⍳m-n ⋄ ((⊂⍵)⌷r)←2/q+⍳p
    {+/,⍵}⍤p⊢r⍉⍺}
```

Extended product

The extended product `xp` acts by repeating the argument of lesser rank on the right to match the argument of greater rank. The result has the greater rank and is produced by a multiplication without contraction. A useful way to think of this is as scalar extension for higher rank arrays. So if:

```
xp←{⍺×⍤(-(⍴⍴⍺)⌊⍴⍴⍵)⊢⍵}
```

```
      a←3 1 4 1 ⋄ b←4 7⍴5 9 2 6 5
      a xp b
15 27  6 18 15 15 27
 2  6  5  5  9  2  6
20 20 36  8 24 20 20
 9  2  6  5  5  9  2
```

`xp` commutes, is associative and linear in its arguments:

```
      a xp b ←→ b xp a
 a xp b xp c ←→ (a xp b)xp c
    a xp b+c ←→ (a xp b)+a xp c
```

Note that if the arguments to `xp` are of the same rank, `xp` is equivalent to `×`.

Extended matrix product

The usual matrix product `+.×` acts on vectors, one from the last axis of the left argument and the other from first axis of the right argument. A useful extension to this is to define a function which behaves in the same way but applies to subarrays of the arguments having rank greater than `1`. Here is the definition of an operator which generates this family of functions:

```
mp←{(,⍤⍺⍺⊢⍺)+.×,[⍺⍺↑⍳⍴⍴⍵]⍵}
```

`mp` is a monadic operator producing a dyadic derived function. The left argument to `mp` specifies a number of axes `k`. The derived function `k mp` is dyadic and applies the function `+.×` between modified arguments. The left argument is modified so that its trailing `k` axes are ravelled. The right argument is modified so that its leading `k` axes are ravelled.

For example:

```
      ⍴(?4 5 6 3 2⍴9) (2 mp) ?3 2 7 8⍴9
4 5 6 7 8
```

or

```
      ρ(?4 5 6ρ9) (1 mp) ?6 7ρ9
4 5 7
```
Regular matrix product `+.×`.

The following relationships for `mp` hold, with comments below:

`a(k mp)b`	↔	`⍉(⍉b)(k mp)⍉a`	
`a(j mp)b(k mp)c`	↔	`(a(j mp)b)(k mp)c`	if `(j+k)≤ρρb`
`a(k mp)b+c`	↔	`(a(k mp)b)+a(k mp)c`	
`(a+b)(k mp)c`	↔	`(a(k mp)c)+b(k mp)c`	

Commutivity

In general `mp` does not commute. `a(k mp)b` is not the same as `b(k mp)a`. This agrees with how the inner product `+.×` behaves. However, we do know that there is a relationship between the inner product and its commuted partner. It goes like this:

```
a+.×b  ↔ ⍉(⍉b)+.×⍉a
```

Is there a similar relationship for `mp`? Yes, there is:

```
a(k mp)b ↔ ⍉(⍉b)(k mp)⍉a
```

Here is an example:

```
      a←?3 2 4ρ9 ⋄ b←?2 4 6 7ρ9
      ρx←a(2 mp)b
3 6 7
      ρy←⍉(⍉b)(2 mp)⍉a
3 6 7
      x≡y
1
```

Associativity

In general `mp` is not associative. Here are two examples, one that is associative and one that is not:

```
      a←?2 3ρ9 ⋄ b←?3 4ρ9 ⋄ c←?4 7ρ9
      ρa(1 mp)b(1 mp)c
2 7
      (a(1 mp)b(1 mp)c)≡(a(1 mp)b)(1 mp)c
1
```
Associative.

```
      a←?6 3 2 7ρ9 ⋄ b←?3 2 4 5ρ9 ⋄ c←?4 5 7ρ9
      ρa(3 mp)b(2 mp)c
6
      (a(3 mp)b(2 mp)c)≡(a(3 mp)b)(2 mp)c
```
Fails with a `LENGTH ERROR`.

The second example illustrates why this happens. In the expression `a(3 mp)b(2 mp)c` the leftmost extended inner product `(3 mp)` requires arguments of at least rank `3`. Its left argument `a` has a shape with trailing elements `3 2 7`. No problem there. The right argument that it receives, however, only has two of these values (`3 2`) that come from `b`; the final element (`7`) comes from `c`. So, when we consider the expression `(a(3 mp)b)(2 mp)c`, the evaluation of `(a(3 mp)b)` has to proceed without the involvement of `c`. And that's a problem.

Associativity of the expression `a(j mp)b(k mp)c ↔ (a(j mp)b)(k mp)c` is only assured if `(j+k)≤ρρb`.

Total inner product

The total inner product is similar to the extended matrix product `mp` but performs contractions using trailing axes.

```
tip←{α sop⍤((ρρα)⌊ρρω)⊢ω}
```

The rank of the result produced is the difference of the ranks of the arguments. For example, if the left argument has shape `2 7 4 3 5` and the right argument shape `3 5`, the total inner product will be of shape `2 7 4` with values produced by multiplying matrices of shape `3 5` together and summing all their elements. Here's an example:

```
      ρ(?2 7 4 3 5ρ9) tip ?3 5ρ9
2 7 4
```

Note that if we wish to contract on leading axes, we may do so with `{⍉(⍉α)tip⍉ω}`.

`tip` commutes and is linear in its arguments

```
  a tip b ↔ b tip a
a tip b+c ↔ (a tip b)+a tip c
```

but is not associative. Consider the shapes of `a`, `b` and `c` in the expression `a tip b tip c`. In order to produce a result of shape `r`, they can take one of four forms:

	Shape of a	Shape of b	Shape of c
A	r,s	s,t	t
B	r,s	t	s,t
C	s	r,s,t	t
D	s	t	r,s,t

With these patterns, we cannot in general form the term `a tip b` and must conclude that `tip` is not associative.

Useful relationships

```
a tip b tip c  ↔  b tip a∘.×c          Forms A & C
               ↔  c tip a∘.×b          Forms B & D
               ↔  c tip a(n mp)b       Form A with n←(ρρb)-ρρc
```

- For `a×b tip c`

  Let's assume that `c` has greater rank than `b`.Then `c` must be of shape `(⍴a),⍴b` in order to produce an array that matches `a` in shape. Each element of `b tip c` is formed as the sum of a product between `b` and a cell of matching rank from `c`. Finally, each element of the result is produced by multiplying an element from `a` with an element from `b tip c`. We can produce the same result by first factoring the cells of `c` with their matching elements from `a`. That's exactly what `a xp c` does.

- For `(⍉b)+.×a`

  Let's start with an example for this equivalence:

```
      a←?2 4⍴15 ⋄ b←?2 3⍴15 ⋄ a b
┌──────────┬──────┐
│5  8  1  5│10 6 3│
│1 13 13 13│ 5 9 2│
└──────────┴──────┘
      a tip 2 sh b+.×id 3 4
55 145  75 115
39 165 123 147
17  50  29  41
      (⍉b)+.×a
55 145  75 115
39 165 123 147
17  50  29  41
```

Let's break this down a bit. The right argument to `tip` is a `3` by `4` array of twelve matrices:

```
      disp 2 sh b+.×id 3 4
┌────────┬────────┬────────┬────────┐
│10 0 0 0│0 10 0 0│0 0 10 0│0 0 0 10│
│ 5 0 0 0│0  5 0 0│0 0  5 0│0 0 0  5│
├────────┼────────┼────────┼────────┤
│6 0 0 0 │0 6 0 0 │0 0 6 0 │0 0 0 6 │
│9 0 0 0 │0 9 0 0 │0 0 9 0 │0 0 0 9 │
├────────┼────────┼────────┼────────┤
│3 0 0 0 │0 3 0 0 │0 0 3 0 │0 0 0 3 │
│2 0 0 0 │0 2 0 0 │0 0 2 0 │0 0 0 2 │
└────────┴────────┴────────┴────────┘
```

The execution of `tip` multiplies together each of these twelve matrices with the left argument matrix `a`, summing all the elements of each multiplication. Because of the zeros introduced by `id`, we can see that each of the twelve values in the result comes about as the result of a matrix product between a column of `a` and a column of `b`. All twelve results can be brought together into an inner product with `b` if we use `⍉a` as the left argument.

- For `a∘.×b tip c`

For example:

```
      a←?3 4⍴9 ⋄ b←?2 5⍴9 ⋄ c←?6 2 5⍴9
      ⍴a∘.×b tip c
3 4 6
      (a∘.×b tip c)≡b tip a∘.×c
1
```

References

[0] *“The Derivative Operator”* K.E. Iverson, Proceedings of APL79: ACM 0-89791-005-2/79/0500-0347.

[1] *“The Derivative Revisited”* M. Powell, https://aplwiki.com/images/f/f9/1_The_Derivative_Revisited.pdf

Mike Powell
mdpowell@gmail.com
April 2020