The Derivative Revisited

Introduction

In 1979 Ken Iverson wrote a paper on the derivative as part of the ACM APL Conference [0]. In 8 pages he covered a lot of ground. His discussion of the derivative goes way beyond *dy/dx*. It deals with derivatives of the most general type. This article revisits much of that paper and attempts to bring the notation up to date. It also provides an APL model of the derivative operator.

(Ken Iverson did himself revisit this topic in 2002. "*Calculus*" is written in J and may be found at [6].)

Many of Iverson's suggestions for extensions to the language proposed in the 1970s and 1980s have now been implemented but with some changes to the notation. For example, Iverson uses a scalar plus operator with the combined symbol "plus overbar" (not yet in the Unicode standard). This operator produces a derived function which is the sum of two functions:

```
plusbar←{(αα ω)+ωω ω}
```

Of course, nowadays we recognize this as a train and we can write `plusbar←{(αα+ωω)ω}` or just `fplusg←f+g`.

Similarly, Iverson defines operators for inner and outer products as follows:

```
f⊕g x ↔ (f x)+.×g x          f⊗g x ↔ (f x)∘.×g x
```

Again, trains can deal with this in a quite general way:

```
f⊕g x ↔ (f+.×g)x             f⊗g x ↔ (f∘.×g)x
```

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.)

Supporting operators and functions

This article assumes that a number of operators and functions are already in place. For convenience their definitions are collected here, with minimal comment.

```
min←{⌊/,ω}
max←{⌈/,ω}
num←{×/⍴ω}
sum←{+/,ω}
mean←{(sum÷num)ω}
sop←{+/,α×ω}
ssq←{sop⍨ω}
rnd←{α×⌊0.5+ω÷α}
```

Minimum
Maximum
Number
Sum
Mean
Sum of product
Sum of squares
Round

```
timer←{ΔΔj←1⎕⎕ai ⋄ ΔΔk←±ω ⋄ 0.001×ΔΔj-¯1⎕⎕ai}
cells←{⊂⍤⍺⊢ω}
disp←{⊂⍤2⊢ω}
```

Execution time in seconds
Cells of an array
Display higher rank array

Function rank

An essential concept is function rank. Iverson [0] describes this at p. 347 as:

> "We will characterize the rank 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."

Iverson associates two rank integers with a function. The second of these is the rank of the argument that can be used with the function without producing a frame. The first is the rank of the result so produced. Nowadays, we usually only concern ourselves with the second of these. Herein, when we use a single integer or refer to a "rank `k`" function, we are referring to just the second integer, the function argument rank.

Iverson is well aware that functions can be usefully applied to values that have rank greater than the function argument rank. There's no need to look further than the scheme for scalar extension already present in APL from day one. This allows a simple function, perhaps something like `{ω*2}`, which is a rank `0` function defined for scalars, to be extended to arrays of any rank. That's why the following does something useful rather than producing an error:

```
     {ω*2} 2 3⍴⍳6                    Scalar extension at work
0  1  4
9 16 25
```

Let's assume that a function `f` has rank `s`. This means that if `f`'s argument is an array of greater rank then it will be treated as a frame of cells. Each cell will be of rank `s` and the frame will have shape of whatever is left over. By convention, the cells use trailing elements of the shape; the frame uses leading elements. We'll assume the application of `f` to an array of rank `s` produces a result of shape `sir`. The result is of shape `fr,sir`.

For example, consider the function `{+/ω}`, which is of rank `1`. When it is applied to an array of higher order, say rank `k`, the result will be of the form `fr,sir` where `fr` is the shape of the frame and `sir` is the shape of an individual result. `sir` is found by considering the application of `f` to an array of rank `1`, a vector. This individual result is the sum of a vector, which is a scalar. It's shape, `sir`, is the empty vector `⍬`. As the rank of this function is `1`, `fr` is `k-1`. Combining these together gives the shape of the complete result as `k-1`.

Now let's consider the function `{+/+/ω}⍤3` applied to an array of shape `5 9 3 7 4 2`. This function uses cells of rank `3` (i.e. of shape `7 4 2`) producing results of rank `1` (`sir←,7`). As this is a rank `3` function, the frame will be of shape `fr←5 9 3`. Combining these together produces a result of shape `fr,sir` which is `5 9 3 7`.

```
     ⍴3 cells x←5 9 3 7 4 2⍴3
5 9 3
     ⍴⊃3 cells x
7 4 2
     ⍴{+/+/ω}⍤3⊢x
5 9 3 7
```

The Derivative Operator

The derivative operator `∆` applied to a monadic function `f` of arbitrary rank `s` produces a derived function `df` of the same rank as `f` and the concepts of frame, cells and the shape of a result all apply to `df` exactly as they do to any other function.

The shape of the result

This means that the result of applying `df` to an array `x` should be expected to produce a result of shape `fr,sir`. Of course, if `s≥ρρx`, the frame `fr` is empty. So what is `sir`, the shape of the individual result for a derivative? To answer that question, we need to look at Iverson's definition for the derivative operator:

> "If I and J are vector indices of of F A and A, respectively, then (since the result rank of F∆ is +/R), I,J is a vector index of F∆ A. We now define F∆ by specifying the value of each element (I,J)[]F∆ A as follows:
>
> (I,J)[]F∆ A ←→ Limit (S→0) of (÷S)×I[](F A+S×J∧.=DтDρι×/D←ρA)-F A
>
> In other words, element I,J of F∆ A is the derivative of 'the Ith component function of F' with respect to element J of of the argument, and is referred to as the 'partial derivative (of the component function) with respect to the Jth variable'."

This makes it clear that `sir` is made up of two parts: the first part corresponds to `ρf a`, where `a` is an `s`-cell of `x`; the second part is just the shape of the cell `a`. In other words, the complete set of partial derivatives requires calculating the variation of the result value (which may be an array) for changes in each element of its individual cell (which may also be an array).

In summary, the shape of the result produced by the derivative of a function `f` with rank `s` applied to an argument `x` is made up of three parts:

- The shape of an individual cell. This is a cell of the minimum rank `s` that `f` can handle. The shape of this is `sic←(-s⌊ρρx)↑ρx`.
- A frame of the shape elements remaining after the shape of an individual cell is removed. The shape of this is `fr←(-s)↓ρx`.
- The shape of the individual result produced by the application of `f` to a representative `s`-cell of `x`. This is `sir←ρf⊃s cells x`.

The shape of the final result is obtained by combining these three parts together as `fr,sir,sic`.

The tables below show how this works out for two representative functions:

Argument of shape

{ω*2} applied with rank	⍬	4	4 5	4 5 6	4 5 6 7
0	⍬	4	4 5	4 5 6	4 5 6 7
1	⍬	4 4	4 5 5	4 5 6 6	4 5 6 7 7
2	⍬	4 4	4 5 4 5	4 5 6 5 6	4 5 6 7 6 7
3	⍬	4 4	4 5 4 5	4 5 6 4 5 6	4 5 6 7 5 6 7

Argument of shape

{(max,min,mean)ω} applied with rank	⍬	4	4 5	4 5 6	4 5 6 7
0	3	4 3	4 5 3	4 5 6 3	4 5 6 7 3
1	3	3 4	4 3 5	4 5 3 6	4 5 6 3 7
2	3	3 4	3 4 5	4 3 5 6	4 5 3 6 7
3	3	3 4	3 4 5	3 4 5 6	4 3 5 6 7

Iverson's definition

Recall Iverson's definition from [0] of the derivative operator:

```
(I,J)⌸FΔ A ←→ Limit (S→0) of (÷S)×I⌸(F A+S×J∘.=DTDρι×/D←ρA)-F A
```

We can define an APL operator to emulate this as:

```
Δij←{s←1e¯7 ⋄ i j←⍺ ⋄ (÷s)×i⌸(⍺⍺ ⍵+s×j∘.=dTdρι×/d←ρ⍵)-⍺⍺ ⍵}
```

The first thing to notice is that this definition does not take a limit as `s` tends to `0`. It simply makes a numeric approximation with `s←0.0000001`.

`Δij` is a monadic operator with a left argument of a function. The result is a dyadic derived function. Its left argument is two elements, both vectors, the first a vector index into `f x`, the second a vector index into `x`. The right argument `x` is the array at which the derivative is to be evaluated. So, for example:

```
f←{+/⍵*2}                    A rank 1 function
f x←3 1 7 2
63
```

For this function `f`, the derivative at `x` has no frame (`fr←0`), a scalar individual result (`sir←0`) and cells which are vectors (`sic←1`). The result is therefore a vector and we can use `Δij` to pick out the various elements of the result:

```
     df←f Δij
     (0 0 df x),(0 1 df x),(0 2 df x),0 3 df x
6 2 14 4
```

As the result of applying`{+/⍵*2}` to a vector is a scalar, the value for the `i` index has to be the empty vector (to conform with the index selection function `[]`). The values for the `j` index range through `0` to `3` (that is `⍳⍴x`).

Let's see what happens with a different function. We'll use `{⍵*3}` and force it to have a rank of `1`.

```
     f←{⍵*3}⍤1
     f x
27 1 343 8
```

For this function `f`, the derivative at `x` has no frame (`fr←0`), a vector individual result (`sir←1`) and cells which are vectors (`sic←1`). The result is therefore of rank `2`. Now when we use the `Δij` operator, both the `i` and `j` arguments can range from `0` to `3`.

```
     df←f Δij
     (0 0 df x),(0 1 df x),(0 2 df x),0 3 df x
27 0 0 0
     (1 0 df x),(1 1 df x),(1 2 df x),1 3 df x
0 3 0 0
     (2 0 df x),(2 1 df x),(2 2 df x),2 3 df x
0 0 147 0
     (3 0 df x),(3 1 df x),(3 2 df x),3 3 df a
0 0 0 12
```

Here's the explanation of the values in this result. The first row shows how the first element of the function's result varies when each element of the argument is changed. This has a non-zero value only in the first position. The second row does the same for the second element of the result; and so on.

In traditional mathematical notation, if we call the elements of the result f_0, f_1, f_2 and f_3 and the elements of the argument x_0, x_1, x_2 and x_3, this matrix is:

$$\begin{array}{cccc} \partial f_0/\partial x_0 & \partial f_0/\partial x_1 & \partial f_0/\partial x_2 & \partial f_0/\partial x_3 \\ \partial f_1/\partial x_0 & \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \partial f_1/\partial x_3 \\ \partial f_2/\partial x_0 & \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \partial f_2/\partial x_3 \\ \partial f_3/\partial x_0 & \partial f_3/\partial x_1 & \partial f_3/\partial x_2 & \partial f_3/\partial x_3 \end{array}$$

An operator for the derivative

Let's define an operator `Δ` to produce the full set of partial derivatives.

In order to correctly model the derivative operator, we need to know the rank of the function to which it is to be applied. This is necessary so that the shape of the data argument can be broken up into its frame and cells. In J, this can be discovered with the `f b. 0` verb. Unfortunately, this sort of information is not available in Dyalog APL (yet) and we must do something else.

The first option is simply to require that the function rank be specified as the left argument to the derived function. This gives correct results but is unnecessary when frames are not involved. A less cumbersome approach is to define `Δ` including the assumption there is no frame involved but with the caveat:

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

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

```
r←(f Δ)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←⍴⍴←f x ⋄ sx←⍴x ⋄ dx←0.000001×{⍵+⍵=0}x←,x
 r←(×/sx,sf)⍴⍬ ⋄ 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
```

Note that:

- The function argument `f` is executed by `Δ`. If `f` relies on global variables, then the values used may be those localized within `Δ`. This is likely to be incorrect. The problem can be pushed out of sight by prefixing every name used within the definition of `Δ` with a very uncommon prefix (such as two underbar characters). For reasons of readability, the definition of `Δ` 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.

Here are two examples. The first was introduced earlier and is just restated using `Δ` rather than `Δij`:

```
     f←{⍵*3}⍤1 ⋄ x←3 1 7 2
     f Δ x                                (or equivalently f Δ⍤1⊢x)
27 0   0  0
 0 3   0  0
 0 0 147  0
 0 0   0 12
```

The second is a little more elaborate. The derivative under examination is for the function `{(max,min,mean)⍵}⍤2` evaluated with a rank `3` data argument.

```
      ⎕rl←16807                          For repeatability.
      f←{(max,min,mean)⍵}⍤2 ⋄ x←?4 5 6⍴9
      df←f ∆
      ⍴z←df⍤2⊢x
4 3 5 6
```

Here's what happens. The function `f` is of rank `2`, transforming matrices into `3` element vectors. The derived functionative `df` is applied with rank `2`. This means that there is a frame (`fr←4`), individual cells are matrices (`sic←5 6`) and each individual result is a vector (`sir←3`). The result is of shape `4 3 5 6`.

```
      disp x
```

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

```
      disp 0.001 rnd z
```

0	0	0	0	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	0	1	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	0	0	0	1	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
1	1	0	0	0	1	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	0	0	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	0	0	1	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	0	0	1	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	0	1	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	1	0	0	1	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	1	0	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	0	0	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	0	0	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	0	1	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	0	0	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
1	0	0	0	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	1	1	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	0	0	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	0	0	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
1	0	0	1	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						
0	0	0	0	0	0	0	0	0	0	0	0	0.033	0.033	0.033	0.033	0.033	0.033						

The display of the result `z` shows `12` matrices, each of shape `5` by `6`, arranged in a `4` by `3` grid. The first column of matrices corresponds to variations for the `max` function. Ones in these matrices correspond to locations of the largest value in each matrix in `x`. The second column is similar but for the `min` function. The last column corresponds to variations for the `mean` function all of which have the same value, `1/30`.

Tensor Contraction; Inner and Outer product

The APL inner product is equivalent to doing an outer product followed by a contraction (as is used in tensor analysis) on the last axis of the left argument and the first axis of the right argument. In tensor notation this would be written as $A_{ij}B_{jk}$. A convenient way to do this is to use dyadic transpose to form a trace at the final axis and apply `+/` to that last axis. For example:

```
      a←4 3⍴⍳12 ⋄ b←3 7⍴⍳21
      ⍴a+.×b
4 7
      a+.×b
 35  38  41  44  47  50  53
 98 110 122 134 146 158 170
161 182 203 224 245 266 287
224 254 284 314 344 374 404

      ⍴a∘.×b
4 3 3 7
      ⍴c←0 2 2 1⍉a∘.×b
4 7 3
      +/c
 35  38  41  44  47  50  53
 98 110 122 134 146 158 170
161 182 203 224 245 266 287
224 254 284 314 344 374 404
```

More general contractions are possible based on dyadic transpose. Iverson suggests an operator which has the axes to be contracted specified in pairs. This is how this would work. Suppose we have a rank `6` array and we wish to contract on axes `0` and `5` and then `1` and `3`:

```
      a←3 5 2 5 4 3⍴3 1 4 1 5 9 2
      ⍴b←2 3 0 3 1 2⍉a
2 4 3 5
      +/+/b
61 50 55 59
44 61 50 55
```

Axes to be contracted are of same length.
First produce the double trace.

Then do the double sum.

An alternative way to do this which avoids the multiple reductions is as follows:

```
      {+/,⍵}⍤2⊢b
61 50 55 59
44 61 50 55
```

We can use this technique to define a general contraction function `tc` which contracts its left argument according to pairs of axes specified in the right argument:

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

Total Inner Product

Iverson introduces an operator which he names the total inner product. This is similar to the regular inner product `+.×` but does more than one contraction. A slight paraphrase of his description goes like this:

> "The inner product between M and N is equivalent to their outer product followed by a contraction which pairs the last axis contributed by M with the first axis contributed by N. If instead we contract by pairing the last axis contributed by M with the last contributed by N, the penultimate axes contributed by M and N, and so on over all possible pairs, the overall process is called the total inner product."

Iverson proposes an operator (plus-with-circle-overbar) with function arguments to produce `M` and `N`. With the advent of trains, it's simpler just to have a function `tip` acting on `M` and `N` directly. If desired, `M` and `N` can be produced with a train such as `(f tip g)x`.

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

```
     a←2 7 3 5⍴⍳11 ⋄ b←3 5⍴⍳15
     ⍴a∘.×b
2 7 3 5 3 5
     (a∘.×b)⍤2 4 3 5
465 511 579 460 550 519 466
600 481 483 661 465 511 579
```

We could define `tip` in terms of `⍤` but it's simpler to employ the rank operator `⍤⍤` as follows:

```
     tip←{⍺ +.×⍤⍤((⍴⍴⍺)⌊⍴⍴⍵)⊢⍵}
     a tip b
465 511 579 460 550 519 466
600 481 483 661 465 511 579
```

`tip` is commutative and linear in its arguments but is not associative.

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

So,

```
     a←4 3⍴⍳12 ⋄ b←4 3⍴3 1 4 1  ⋄ c←2 7 1 ⋄ d←8 2 8
     a b c d
```

0 1 2 3 4 5 6 7 8 9 10 11	3 1 4 1 3 1 4 1 3 1 4 1	2 7 1	8 2 8

```
     a tip c
9 39 69 99
     (a tip c) ≡ c tip a
1
     (a tip c+d) ≡ (a tip c)+a tip d
1
     ((a+b) tip c) ≡ (a tip c)+b tip c
1
     ((5×a)tip c) ≡ 5×a tip c
1
```

The Derivative Rules

The rules for the derivatives of compound formulations (such as sum `f+g` and product `f×g`) are normally stated for simple cases. Here's an example, in both conventional and APL notation, of the product rule:

$y = x^2 \sin x$	`y←(x*2)×1○x`
$dy/dx = x^2 \cos x + 2x \sin x$	`y ∆ x ↔ ((x*2)×2○x)+2×x×1○x`

The variable *x* is a scalar and so is the result *y*. The derived function *dy/dx* takes a scalar as its argument and produces a scalar result.

Of course, we don't have to look far to find something a little more complex. A common situation encountered is where we have two functions of a vector of coordinates, both returning a scalar result. The coordinates might represent locations on a map. At every two dimensional point `pt←x,y` there might be values known for the population density `d` and the energy consumption per person `e`. At every point we can then calculate the total energy consumption as `(d×e)pt`. Perhaps we need to know the rate of change of total energy consumption across the map. This is a vector with a value in both the `x` and `y` directions. For this case, the rule is:

```
(d×e)∆ pt    ↔    ((d×e ∆)+d ∆×e)pt
```

Iverson includes two tables of the common derivative rules, one for the case of vector functions and one for scalar functions. These have been updated, extended somewhat and are explained in much greater detail in [5]. Here are Iverson's tables with their translations to more modernAPL.

For scalar functions:

Sum	`F⍥G∆ ↔ (F∆)⍥(G∆)`	`(f+g)∆ ↔ f ∆+g ∆`
Product	`F⍤G∆ ↔ (F∆⍤G)⍥(G∆⍤F)`	`(f×g)∆ ↔ (f ∆×g)+(g ∆×f)`
Composition	`F⍨G∆ ↔ F∆⍨G⍤(G∆)`	`(f g)∆ ↔ (f ∆ g)×g ∆`
Inverse	`F⍣¯1∆ ↔ ÷⍨(F∆⍨(F⍣¯1))`	`f⍣¯1 ∆ ↔ ÷(f ∆ f⍣¯1)`

For vector functions:

Sum	`F⍥G∆ ↔ (F∆)⍥(G∆)`	`(f+g)∆ ↔ f ∆+g ∆`
	`F⊗G∆ ↔ F⊗(G∆)⊕(0 2 1 ⍉(F∆⊗G))`	`(f∘.×g)∆ ↔ (f∘.×g ∆)+0 2 1⍉f ∆∘.×g`
Product	`F⍤G∆ ↔ 0 0 1⍨⍨(F⊗G∆)`	`(f×g)∆ ↔ 0 0 1⍉(f∘.×g)∆`
Composition	`F⍨G∆ ↔ F∆⍨G⊕(G∆)`	`(f g)∆ ↔ (f ∆ g)+.×g ∆`
Inverse	`F⍣¯1∆ ↔ ÷⍨(F∆⍨(F⍣¯1))`	`f⍣¯1 ∆ ↔ ⌹f ∆ f⍣¯1`
	`F⊕G∆ ↔ F⊗G∆⊕0 1`	`(f+.×g)∆ ↔ 0 1 tc⍨(f∘.×g)∆`

Symmetry and the Canonical Skew Cube

Iverson introduces symmetric and skew-symmetric arrays with definitions for a pair of axes:

> "An array A is said to be symmetric in axes I and j if the result of interchanging axes I and J equals A, and skew-symmetric in I and J if the result equals -A. An array is said to be symmetric (skew-symmetric) if it is symmetric (skew-symmetric) in all axis pairs."

A helpful way to look at this in APL is to remember that the interchange of axes calls for a dyadic transpose. So, if we start with an array `a`, the interchange of a pair of axes could produce a result which is unchanged, the negation `-a`, or even something else. For example:

```
xchg←{t←⍳⍴⍴⍵ ⋄ i j←⍺ ⋄ (⍺[t+(i-j)×(t=j)-t=i])⍉⍵}
ρ1 2 xchg 2 3 4 5 6ρ3
2 4 3 5 6
sym←{⍵≡⍺ xchg ⍵}
skew←{⍵≡-⍺ xchg ⍵}
```

Symmetric with respect to axes `⍺`
Skew with respect to axes `⍺`

Iverson then looks at the results of transposition with permutation vectors:

> "If A is symmetric and P is a permutation vector of shape ⍴⍴A, then P⍉A equals A. If A is skew, then P⍉A equals A if the permutation is even (i.e., the permutation is equivalent to an even number of pairwise transpositions) and -A if it is odd."

And this leads to a definition for the sign of a vector:

> "We now define the sign of a vector as 1 if the vector is an even permutation, ¯1 if it is odd, and 0 if it is not a permutation."

Iverson provides a function for the sign of a vector. It is written (as an `⍺ ⍵` direct definition) to accept a matrix as an argument producing one element in the result for each row of the argument:

```
S:1:0≠D←¯1↑⍴⍵:(R<D)×(¯1*R×D-1)×S(0=⍳⍴⍴⍵)↓¯1+(R←+/∧\⍵≠0)⍋⍵
S m←3 4ρ0 1 2 3,0 1 3 2,0 1 1 2
1 ¯1 0
```

As the sign is defined for a vector, it seems more natural to define a function of rank `1`, which can then be applied to arrays of any rank. So:

```
sign←{0=d←⍴⍵:1 ⋄ r←+/∧\⍵≠0 ⋄ ⍬⍴(r<d)×(¯1*r×d-1)×∇ ¯1+1↓r⍋⍵}⍤1
sign m
1 ¯1 0
```

Next up for Iverson's consideration is the canonical skew cube. This rather formidable sounding name is actually not too difficult to understand. We start with a cube of order `n`. This is nothing more than a rank `n` array with all dimensions being `n`. So an order `3` cube would be of shape `3 3 3` and an order `4` array would be of shape `4 4 4 4`.

Skew cubes are cubes with elements whose position, expressed as an index, is used to calculate the `sign` for that element. Skew cubes can be scalar multiples of one special skew cube – the canonical skew cube. The rank `2` version of this is `2 2⍴0 1 ¯1 0`. The first element of this array is at index `0,0`. As this is not a permutation the skew cube has a `0` in this position. The next element is at index `0,1` which is an even permutation producing a `1` for this element. The third element has index `1,0` which is an odd permutation producing a `¯1`. Lastly is an element at position `1,1` which gives a `0` as it is not a permutation.

Iverson's direct definition for the canonical skew cube is:

```
CSC:S⍵Dτ⍵Dρ⍳×/D←NρN←ρ⍵
```

This takes a vector as an argument. The length of this vector is the order of the cube. This can be simplified a little by using the `sign` function defined above:

```
csc←{signt⍳nρn←ρ⍵}
```

Using this definition, we have:

```
    csc 6 3
 0 1
¯1 0
     disp csc 3 4 5
┌─────┬─────┬─────┐
│0  0 0│0 0 ¯1│ 0 1 0│
│0  0 1│0 0  0│¯1 0 0│
│0 ¯1 0│1 0  0│ 0 0 0│
└─────┴─────┴─────┘
```

By examining the order `4` example below, we can see that the canonical skew cube is mainly zeros. `1`s and `¯1`s appear in pairs, but only when their indices correspond to permutations.

```
    disp csc 1 9 4 9
┌─────────┬─────────┬─────────┬─────────┐
│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│0  0 1 0│
│0 0 0 0 │0 0  0 1│0 0 0  0│0 ¯1 0 0│
│0 0 0 0 │0 0 ¯1 0│0 1 0  0│0  0 0 0│
├─────────┼─────────┼─────────┼─────────┤
│0 0 0  0│0 0 0 0 │ 0 0 0 1│0 0 ¯1 0│
│0 0 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│1 0  0 0│
│0 0 1  0│0 0 0 0 │¯1 0 0 0│0 0  0 0│
├─────────┼─────────┼─────────┼─────────┤
│0  0 0 0│0 0 0 ¯1│0 0 0 0 │ 0 1 0 0│
│0  0 0 1│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 ¯1 0 0│1 0 0  0│0 0 0 0 │ 0 0 0 0│
├─────────┼─────────┼─────────┼─────────┤
│0 0  0 0│ 0 0 1 0│0 ¯1 0 0│0 0 0 0 │
│0 0 ¯1 0│ 0 0 0 0│1  0 0 0│0 0 0 0 │
│0 1  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 │
└─────────┴─────────┴─────────┴─────────┘
```

The canonical skew cube (of order `4`) is the ε_{ijkl} pseudotensor used in tensor analysis. It is known by several names and Wikipedia has a useful article [2] for the "Levi-Cevita symbol".

Iverson defines the normal operator as:

> "The total inner product of a function F with the canonical skew function CSC is an operator on F which will be denoted by F⊕∘ and will be called the normal operator."

This can be defined in a straightforward manner, with a train:

```
norm←{(αα tip csc)ω}
```

Iverson continues (with `tip` in place of the untypable plus-circle-overbar character):

> "If F is a vector function, then F⊕∘ is orthogonal (or normal) to F in the sense that F⊕∘ tip F (or, equivalently F⊕∘+.×F) is zero.
>
> More generally, if F and G are vector functions, then both are orthogonal to the normal of their outer product, that is, to F⊗G⊕∘."

We can test this with:

```
     f←{ω*2}⍤1
     g←{1○ω*÷3}⍤1
     (f norm tip f)x←1 9 4 9
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
     (f norm +.× f)x
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
     (f tip (f∘.×g)norm)x
0 ¯1.13687E¯13 0 2.27374E¯13
     (g tip (f∘.×g)norm)x
0 ¯1.06581E¯14 0 1.42109E¯14
```

Pretty close to zero.

ditto.

Mathematical Physics

Iverson deals briefly with six common differential forms used in mathematical physics. He defines these for the most common case: the field is a function of a vector position. (`F01` indicates a rank `0 1` function. `F11` indicates a rank `1 1` function.) These are in the following table, which has an extra column for the APL `dfn` equivalents:

		Resulting Function Rank	dfn equivalent
Gradient	F01Δ	1 1	{αα Δ ω}
Matrix of partials	F11Δ	2 1	{αα Δ ω}
Jacobian	-.×¨(F11Δ)	0 1	{det αα Δ ω}
Divergence	F11Δ⊕0 1	0 1	{(αα Δ ω)tc 0 1}
Laplacian	F01ΔΔ⊕0 1	0 1	{(αα Δ Δ ω)tc 0 1}
Curl	-¨F11Δ⊕∘	1 1	{-αα Δ norm ω}

Determinant

Iverson uses monadic `-.×` for the determinant function. Unfortunately, this is not (yet) defined in Dyalog APL. In its place we'll use the following definition (copied from the `dfns` workspace supplied with Dyalog APL):

```
det←{⍺←1
      0=n←≢⍵:⍺
      i j←(⍴⍵)⊤{⍵⍳⌈/⍵}⌊,⍵ ⋄ k←⍳n
      (⍺×⍵[i;j]×¯1*i+j)∇ ⍵[k~i;k~j]-⍵[k~i;j]∘.×⍵[i;k~j]÷⍵[i;j]}
```

Gradient

Let's start out with electrostatics. According to Maxwell's laws, the electric field at a point in three dimensional space is the gradient of the electric potential at that point:

$$E = -grad\ \phi$$

Although this might look like the application of a function to a value, it is not. ϕ is a function of position, returning a scalar at each point. Iverson describes this as a rank `0 1` function: i.e. a function of a vector argument producing a scalar result. And *grad* is an operator, not a function. It produces a derived function with rank `1 1`. The derived function takes a vector as an argument and produces a vector as a result. In APL terms, this is:

```
phi←{⍬⍴ … }⍤1          A scalar function of vector coordinates
grad←{αα Δ ω}
e←-phi grad            e is a vector function
```

The electric field is now seen as a function of position. Every physicist knew that all along but now it's clearly stated. Here's an example:

```
phi←{÷(⍵+.×⍵)*0.5}⍤1          The field of a unit point charge
e←-phi grad
e 2 3 4
0.0128066 0.01920990.0256131
```

Matrix of partial derivatives

The matrix of partial derivatives is usually defined for a rank `1` function with a vector data argument. This produces a rank `2` result which is expected to be square. In these circumstances, the matrix of partials is just the result produced by the application of the gradient.

```
mop←{αα ∆ ω}
```

For example, for a function f of a vector argument with elements x_0, x_1, x_2 and x_3, producing a result with elements f_0, f_1, f_2 and f_3 the matrix of partials would be of shape `4` by `4`:

$$\begin{array}{cccc} \partial f_0/\partial x_0 & \partial f_0/\partial x_1 & \partial f_0/\partial x_2 & \partial f_0/\partial x_3 \\ \partial f_1/\partial x_0 & \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \partial f_1/\partial x_3 \\ \partial f_2/\partial x_0 & \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \partial f_2/\partial x_3 \\ \partial f_3/\partial x_0 & \partial f_3/\partial x_1 & \partial f_3/\partial x_2 & \partial f_3/\partial x_3 \end{array}$$

Jacobian

The Jacobian is just the determinant of the matrix of partial derivatives:

```
J←{det αα mop ω}
```

Here's an example:

```
f←{x y←ω ⋄ (y×x*2),(5×x)+1○y}⍤1
```

The argument to this vector function is expected to be of length `2`. The matrix of partials is of shape `2 2` with elements corresponding to:

```
y×2×x          x*2
5              2○y
```

The Jacobian, determined exactly, is then `(y×2×x×2○y)-5×x*2`. Evaluating this at the point `2 3`, we get a matrix of partials:

```
12  4
 5 ¯0.989992
```

and a determinant of `¯31.8799`.

For comparison, the equivalent values for the matrix of partials and the Jacobian using the `∆` operator are:

```
    f mop 2 3
12  4
 5 ¯0.989993
    f J 2 3
¯31.8799
```

Note that this Jacobian operator works with functions of vectors of any length. Jacobians come up often in the calculations for the transformations from one coordinate system to another – and this could easily be in four dimensions, as is required in special and general relativity.

Divergence

The divergence is an operator applied to a vector function. It produces a derived function of rank `1` which generates a scalar result. For example, the function might be the electric field at every point in three dimensional space. The divergence of that field at a point is related to the charge density (a scalar value) by:

$$div\ E = 4\pi\varrho$$

Referring to the matrix of partials in traditional notation example above, the divergence is:

$$\partial f_0/\partial x_0 + \partial f_1/\partial x_1 + \partial f_2/\partial x_2 + \partial f_3/\partial x_3$$

The corresponding APL definition of the divergence is then `div←{+/0 0⍉⍺⍺ mop ⍵}` or equivalently

```
div←{(⍺⍺ ∆ ⍵)tc 0 1}
```

Continuing with the example of the electric field of a unit charge, we have:

```
     phi←{÷(⍵+.×⍵)*0.5}⍤1
     e←-phi grad
     e div 2 3 4
¯0.0000149545
```

Warning: An often encountered definition of the divergence is:

$$div\ F = \nabla \cdot F = (\partial/\partial x, \partial/\partial y, \partial/\partial z) \cdot (F_x, F_y, F_z) = \partial F_x/\partial x + \partial F_y/\partial y + \partial F_z/\partial z$$

Wikipedia has the following to say about this notation:

> "The common notation for the divergence ∇ · F is a convenient mnemonic, where the dot denotes an operation reminiscent of the dot product: take the components of the ∇ operator (see del), apply them to the corresponding components of F, and sum the results. Because applying an operator is different from multiplying the components, this is considered an abuse of notation."

So don't bother trying to find ∇ or this sort of dot operator in APL.

Laplacian

The Laplacian of a scalar field ϕ, a function of coordinates x, y and z can be written as:

$$\nabla^2 \phi = \partial^2\phi/\partial x^2 + \partial^2\phi/\partial y^2 + \partial^2\phi/\partial z^2$$

and can be defined as the divergence of the gradient of the field ϕ:

$$\nabla^2 \phi = div\ grad\ \phi$$

Written in standard notation, this has the order of execution for operators as right to left, just the same as for functions. APL insists on a distinction and the definition of the Laplacian then is either `{⍺⍺ ∆ div ⍵}` or equivalently:

```
L←{⍺⍺ grad div ⍵}
```

`⍺⍺` is the scalar field function;
`⍵` is the point

For the electric field we used earlier, we have:

```
     phi←{÷(⍵+.×⍵)*0.5}⍤1
     phi L 2 3 4                    Result is a scalar
0.0000149545
```

Curl

A fairly straightforward definition of the curl operator is given in Einstein tensor notation (e.g. Synge & Child, [1] p. 135). If f is a vector field in 3 dimensions (i.e. a rank `1 1` function), then:

$$(curl\ f)^i = \varepsilon^{ijk}\ \partial f_j / \partial x_k$$

This calls for the formation of the matrix of partial derivatives (i.e. all the combinations of the derivative of f_j with respect to x_k), its outer product with the canonical skew cube of order 3 and the contraction of the result on axis pairs j and k. This goes over into APL directly as:

```
curl←{-αα ∆ norm ω}
```

Note that as the APL definition does not make use of indices, `curl` requires no modification to be used in four or more dimensions.

Here's an example which appears in the Wikipedia article on curl [3]. Consider a three dimensional vector field:

```
f←{x y z←ω ⋄ y,(-x),0}          Field f at a point x,y,z
```

As the `z` value is unused in the definition of `f`, this field is cylindrical: at each point there is a 3-vector oriented at right angles to a line joining the point to the `z` axis perpendicularly. The magnitude of this vector is proportional to the point's distance from the `z` axis. Here's a graph of the field in the x-y plane.

.

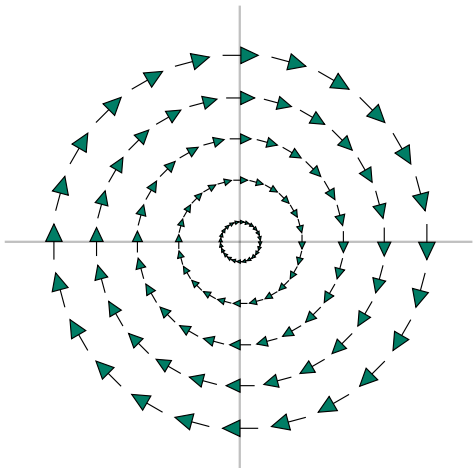

Figure A

Using the component expression for curl in standard notation we get:

$$curl\ f = (0, 0, \partial f_0/\partial x_1 - \partial f_1/\partial x_0)$$
$$= (0, 0, -2)$$

This value is constant throughout the whole vector field. The value given by `curl` at a representative point is:

```
      f curl 1 1 1
0 0 ¯2
```

The Predator & Prey Problem

An interesting example of a vector field is the predator-prey problem (Wikipedia [4]). This has two coupled non-linear differential equations describing the dynamics of the populations of predators and prey. Typical situations involve foxes and rabbits or cheetahs and baboons. The equations take this form:

$$\begin{aligned} dx/dt &= ax - bxy \\ dy/dt &= cxy - dy \end{aligned}$$

where x is the population of prey, y is the population of predators and a, b, c and d are constants.

The prey survive by reproducing well. With an abundant supply of food and no predators, their population `prey` grows exponentially. Assume they have a constant growth rate `a`, meaning that in a small time interval, their population grows by a number `a×prey`. However, there are predators around and a number of prey will be eaten. We'd expect the number lost to be proportional to the populations of both predators and prey, i.e. `b×predator×prey`. Of course, some prey will die through old age but this is accounted for using a smaller value for the parameter `a`. Predators face their own challenges. Their reproduction is tied to their population and to the supply of prey: `c×predator×prey`. They have no predators but die either from starvation or old age: `d×predator`.

For example:

```
a b c d←2 4 3 3÷3
```

At any point in time, the two populations are changing. If we draw a graph with axes of prey population and predator population, there is a two element vector associated with each point which has elements of the time rate of change of the two populations. That's a vector field imposed on the state diagram. Here is a rough picture of this field:

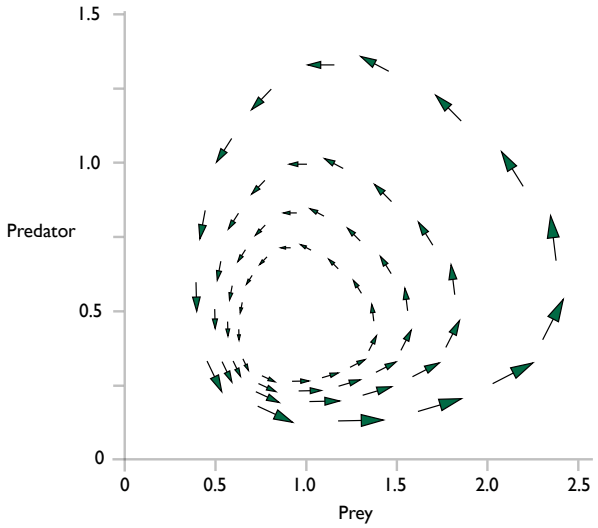

Figure B

We can express the time derivative of the two populations as a two element vector:

```
    growth←{prey predator←⍵ ⋄ ((a×prey)-b×prey×predator),(c×prey×predator)-d×predator}
»   growth←{prey predator←⍵ ⋄ (prey,predator)×(a,-d)-(b,-c)×predator,prey}
```

And if we choose our parameters in pairs:

```
    p←a,-d ⋄ q←b,-c
    growth←{⍵×p-q×⌽⍵}
```

The Wikipedia article explains that these growth vectors form closed loops.

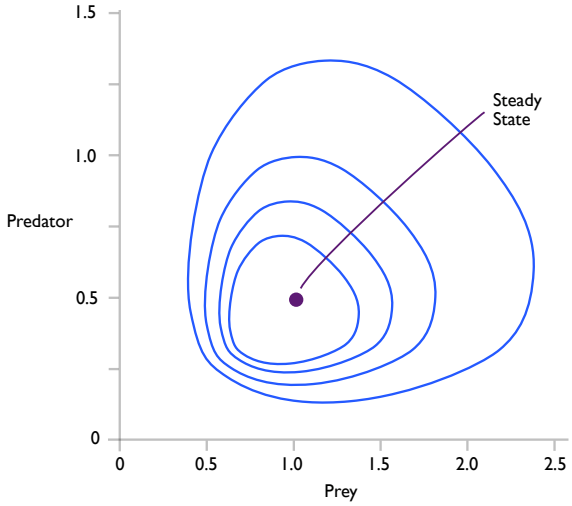

Figure C

In order to see this numerically, we need to select a starting point, calculate the growth vector, move to the new point and repeat until we get back to the beginning. Here's a function to do this:

```
    repeat←{⍺=0:⍵ ⋄ ⍵,⍨{⍵+0.01×growth ⍵}¯1 2↑⍵}(⍺-1)∇ ⍵}
```

So, if we pick a starting point of `1 0.9`, we can see the first few moves:

```
    5 repeat 1 2⍴1 0.9
1        0.9
0.994667 0.9
0.989362 0.899952
0.984086 0.899856
0.978839 0.899713
0.973623 0.899523
```

Our starting point is at the top of a loop and we're moving due West. We can explore the entire loop with more iterations and pick out the other extreme points:

```
    ⍴loop←1000 repeat 1 2⍴1 0.9
1001 2
```

```
{(⊂ω+⍳10)⍣loop}¨186 436 638
```

The left, bottom and right edges

0.55571 0.509689	0.983757 0.238443	1.64358 0.487316
0.555638 0.507424	0.987188 0.238405	1.64386 0.490453
0.555583 0.505169	0.990631 0.238374	1.64407 0.49361
0.555545 0.502924	0.994087 0.238352	1.64421 0.49679
0.555523 0.500689	0.997555 0.238338	1.64428 0.49999
0.555518 0.498464	1.00104 0.238332	1.64428 0.503211
0.555529 0.496248	1.00453 0.238334	1.64421 0.506453
0.555557 0.494042	1.00803 0.238345	1.64406 0.509716
0.555601 0.491847	1.01155 0.238364	1.64385 0.512999
0.555662 0.489661	1.01508 0.238392	1.64357 0.516302

If we follow the loop further we can return to the top which is pretty close to the starting point:

```
{(⊂ω+⍳10)⍣loop}¨788
```

1.02484 0.912039
1.01921 0.912266
1.01361 0.912441
1.00803 0.912565
1.00249 0.912639
0.996971 0.912661
0.991486 0.912634
0.986031 0.912556
0.980607 0.912429
0.975215 0.912252

The growth diagram in Figure B suggests that there is an equilibrium point `pt` at which the growth vector is `0 0`. It's not difficult to see this from the definition of the `growth` function: `{ω×p-q×⌽ω}`. In fact, there are two possibilities. One is when both populations are zero; the other is when `p=q×⌽pt` or `pt←⌽p÷q`.

```
     ⌽p÷q
1 0.5
     growth⍤1⊢2 2⍴ 0 0,⌽p÷q
0 0
0 0
```

Intuitively we can guess from the growth diagram in Figure B that this vector field has both divergence and curl. Let's evaluate the divergence for a grid of suitable points:

```
     h←0.4+0.2×⍳9
     grid←h∘.,0.5×h
     1⍕{growth div ω}¨grid
 ¯0.2 ¯0.3 ¯0.5 ¯0.6 ¯0.7 ¯0.9 ¯1.0 ¯1.1 ¯1.3
  0.0 ¯0.1 ¯0.3 ¯0.4 ¯0.5 ¯0.7 ¯0.8 ¯0.9 ¯1.1
  0.2  0.1 ¯0.1 ¯0.2 ¯0.3 ¯0.5 ¯0.6 ¯0.7 ¯0.9
  0.4  0.3  0.1  0.0 ¯0.1 ¯0.3 ¯0.4 ¯0.5 ¯0.7
  0.6  0.5  0.3  0.2  0.1 ¯0.1 ¯0.2 ¯0.3 ¯0.5
  0.8  0.7  0.5  0.4  0.3  0.1  0.0 ¯0.1 ¯0.3
  1.0  0.9  0.7  0.6  0.5  0.3  0.2  0.1 ¯0.1
  1.2  1.1  0.9  0.8  0.7  0.5  0.4  0.3  0.1
  1.4  1.3  1.1  1.0  0.9  0.7  0.6  0.5  0.3
```

This reveals a pattern. The values of smaller magnitude lie along the principal diagonal and represent something like laminar flow at those points. The values of larger magnitude are on the opposite diagonal (divergent if positive and convergent if negative):

```
     1⍤{growth div ⍵}¨0 0⍉grid
 ¯0.2 ¯0.1 ¯0.1 0.0 0.1 0.1 0.2 0.3 0.3

     1⍤{growth div ⍵}¨0 0⍉⌽grid
¯1.3 ¯0.9 ¯0.6 ¯0.3 0.1 0.4 0.7 1.1 1.4
```

The curl of this field demonstrates a different pattern:

```
     1⍤{growth curl ⍵}¨grid
 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5
 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8
 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1
 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3
 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6
 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9
 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1
 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4
 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7
```

It appears that there is a circulation at every point but it is most pronounced for larger values of the populations.

Conclusion

Ken Iverson’s description of the derivative operator and some of its uses is a model of clarity. Bringing the notation up to date has been straightforward.

Iverson's paper was written before the concept of trains was clearly understood. This has meant that in a few places (e.g. `tip` and `tc`) he resorted to an operator where today he could just as effectively have used a function.

Although this paper makes use of function rank throughout, there is no mention of the rank operator ⍤ or the concepts of cell and frame in [0]. These parts of the language saw the light of day a little later. If they had been available, the text could have more easily dealt with argument values which produce frames.

References

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

[1] *"Tensor Calculus"* J.L. Synge & A. Schild, Dover Publications 1949.

[2] https://en.wikipedia.org/wiki/Levi-Civita_symbol

[3] https://en.wikipedia.org/wiki/Curl_(mathematics)

[4] https://en.wikipedia.org/wiki/Lotka–Volterra_equations

[5] "The Derivative Rules", M. Powell, https://aplwiki.com/images/f/f8/2_The_Derivative_Rules.pdf

[6] "*Calculus*" K.E. Iverson, https://www.jsoftware.com/books/pdf/calculus.pdf

Mike Powell, mdpowell@gmail.com

April 2020