APL AND PARTITIONED DATA

by Jonathan Barman

Introduction

APL arrays provide a natural way of partitioning data. A matrix can be viewed as a set of vectors; each row of a matrix of numbers could be vectors of costs incurred by each department in a business. Adding up the total costs incurred by each department is then a simple matter of applying plus reduction along the last dimension. Reduction and Scan operators allow the application of any scalar function along any axis of an array, and provide powerful tools for creating functions which work on partitioned data. There are, however, limitations in some applications. In the example of department costs, some departments may incur many cost items, and have long vectors, while others may have only one or two cost items. APL arrays have to be rectangular, so holding the costs as a matrix means that all the rows have to have the same length; short vectors have to be padded out with zeros to match the largest number of cost items. The amount of padding required can be so large that it becomes difficult to manipulate the matrix without workspace full messages, although the amount of data is relatively small. If one department out of 100 departments had 5000 cost items and all the other departments averaged 5 items apiece, then the matrix has to be 100 rows by 5000 columns taking up 2,000,000 bytes, of which only 40,000 bytes is data.

Another difficulty is where the data is normally manipulated as a vector and it is inconvenient to form it into a matrix so that reductions and scans can be applied, and then reformat it as a vector. For example, text typed by the user of a system may need to be manipulated and re-displayed, and it is convenient to keep the text as a vector throughout the processing.

This article explores the ways in which partitioned data can be processed in a more natural way, without looping. The techniques are well known and have been in use for many years. The Working Memorandum on Boolean Techniques by Robert A. Smith was published by STSC in 1975, and sets out the fundamental ideas and lists an extensive set of functions. The FinnAPL Idiom list contains examples of manipulating partitioned data. The APL*Plus and Sharp timesharing services both provide workspaces of partition functions.

Before plunging into detail, the general principles will be illustrated with a simple example. The principles will then be analysed in more detail and illustrated with more examples.

Taking the department cost example, assume that each department has a unique code and that the costs and codes are held in two numeric vectors COSTS and DEPTS. The costs and department codes were entered from invoices, so that each cost has a corresponding department code.

There are three basic ways of adding up the costs for each department; by looping through each department code, by forming the data into a matrix and using plus reduction, or by using partition techniques. The looping method could be implemented as follows:

```
      ∇R←DEPTS ADDUP COSTS;A;B;⎕IO
[1]   ⍝ ADD UP <COSTS> FOR EACH CODE IN <DEPTS>
[2]   ⍝ <R[;1]> IS DEPT CODES, <R[;2] IS TOTAL COSTS.
[3]   ⍝ LOOPING METHOD.
[4]    ⎕IO←1
[5]    R←0 2⍴0
[6]   L1:→(0∈⍴DEPTS)/0
[7]    A←1↑DEPTS
[8]    B←A=DEPTS
[9]    R←R,[1]A,+/B/COSTS
[10]   B←~B
[11]   DEPTS←B/DEPTS
[12]   COSTS←B/COSTS
[13]   →L1
       ∇

       1 4 4 3 1 ADDUP 10 20 30 40 50
  1 60
  4 50
  3 40
```

This method is inefficient if large amounts of data are involved. Lines 9 to 12 reassign the variables, so data is being moved in memory for every unique department found.

Forming the data into a matrix is more efficient than the looping method, but, as explained above, there may be workspace full problems:

```
      ∇R←DEPTS ADDUP COSTS;A;P;⎕IO
[1]   ⍝ ADD UP <COSTS> FOR EACH CODE IN <DEPTS>
[2]   ⍝ <R[;1]> IS DEPT CODES, <R[;2] IS TOTAL COSTS.
[3]   ⍝ MATRIX METHOD.
[4]    ⎕IO←1
[5]    ⍝ SORT INTO DEPT CODE SEQUENCE
[6]    A←⍋DEPTS
[7]    DEPTS←DEPTS[A]
[8]    COSTS←COSTS[A]
[9]    ⍝ FIND WHERE CODES CHANGE
[10]   P←DEPTS≠1↓DEPTS,0
[11]   P[(0≠⍴P)/⍴P]←1
[12]   ⍝ FIND NUMBER OF CODES FOR EACH DEPT.
[13]   A←P/⍳⍴P
[14]   A←A-¯1↓0,A
[15]   ⍝ FORM EXPANSION VECTOR.
[16]   A←A∘.≥⍳⌈/0,A
[17]   ⍝ MAKE COSTS INTO A MATRIX.
[18]   R←(⍴A)⍴(,A)\COSTS
[19]   ⍝ ADD UP, AND APPEND DEPT CODES.
[20]   R←(P/DEPTS),[1.5]+/R
       ∇

       1 4 4 3 1 ADDUP 10 20 30 40 50
1 60
3 40
4 50
```

Forming the data into a matrix requires a technique which is constantly being used when dealing with partitioned data. Line 10 is a "not-equals positive difference operation", and line 14 is a "minus negative difference operation".

Lines 10 and 11 generate a "partition vector". Line 11 is necessary because it cannot be guaranteed that a department code of zero does not exist. If the rotation method is used:

```
P←DEPTS≠1⌽DEPTS
```

then line 11 is required in case there is only one department code in the data.

Care has been taken that empty arguments do not cause an error. Line 11 checks for an empty vector. Line 14 could have been written as:

```
A←A-0,¯1↓A
```

which would have caused a length error if A was empty. Line 16 has a 0 catenated to A in case it is empty. It is good practice to ensure that all code will work properly on empty vectors, but it is sometimes simpler to branch out on empty at the beginning of the function rather than having to include special processing as in line 11.

The partitioned data approach is as follows:

```
      ∇R←DEPTS ADDUP COSTS;A;P;⎕IO
[1]   ⍝ ADD UP <COSTS> FOR EACH CODE IN <DEPTS>
[2]   ⍝ <R[;1]> IS DEPT CODES, <R[;2] IS TOTAL COSTS.
[3]   ⍝ PARTITION METHOD.
[4]    ⎕IO←1
[5]   ⍝ SORT INTO DEPT CODE SEQUENCE
[6]    A←⍋DEPTS
[7]    DEPTS←DEPTS[A]
[8]    COSTS←COSTS[A]
[9]   ⍝ FIND WHERE CODES CHANGE
[10]   P←DEPTS≠1↓DEPTS,0
[11]   P[(0≠⍴P)/⍴P]←1
[12]  ⍝ CUMULATIVE SUM FOR EACH DEPT.
[13]   R←P/+\COSTS
[14]  ⍝ CONVERT TO INDIVIDUAL SUMS.
[15]   R←R-¯1↓0,R
[16]  ⍝ APPEND DEPT CODES
[17]   R←(P/DEPTS),[1.5]R
       ∇

       1 4 4 3 1 ADDUP 10 20 30 40 50
1 60
3 40
4 50
```

The steps down to line 11 are identical to the matrix method. Line 13 gets the overall cumulative sum for each department, and line 15 does the "minus negative difference

operation" which converts the cumulative sums back to individual sums. This relationship between scan and negative difference operation is another important technique which will be explored more fully later.

The ADDUP function is really carrying out three processes: sort the data, set up a partition vector, and carry out a partitioned plus reduction. The processes are needed in many varied circumstances, so it is convenient, and better programming practice, to have separate functions. Lines 10 and 11 generated a trailing partition vector as it was needed in this form on line 13. A trailing partition vector is one where a 1 flags the end of each partition:

```
      A←2 2 2 6 6 7 7 7
      A≠1↓A,0
0 0 1 0 1 0 0 1
```

A leading partition vector is one where a 1 flags the start of each partition:

```
      A≠¯1↓0,A
1 0 0 1 0 1 0 0
```

All partition functions need a partition vector as an argument, and it is necessary to standardise on either leading or trailing partitions. As the literature on partition functions always uses leading partitions, we will do likewise. The first function to be defined is one to create a partition vector:

```
      ∇R←CREATE∆PARTITION A
[1]   ⍝ <R> IS A LEADING PARTITION VECTOR WITH 1'S WHERE'
[2]   ⍝ <A> CHANGES.
[3]    R←A≠¯1⌽A
[4]    →(0∈⍴R)/0
[5]    R[⎕IO]←1
       ∇

      CREATE∆PARTITION 1 1 1 1 8 8 50 50 50
1 0 0 0 1 0 1 0 0
```

Line 3 uses the rotation method to allow for the data being either character or numeric, line 4 branches on empty, and line 5 guarantees the first element is a 1.

```
      ∇R←P P∆PLUS∆RED A
[1]   ⍝ <P> IS A LEADING PARTITION VECTOR,
[2]   ⍝ <A> IS A NUMERIC ARRAY.
[3]   ⍝ <R> IS A PARTITIONED PLUS REDUCTION ON THE
[4]   ⍝ FIRST DIMENSION OF <A>.
[5]    R←(1⌽P)⌿+⍀A
[6]    R←R-(⍴R)⍴0,[⎕IO]R
       ∇
```

Lines 5 and 6 of the partitioned plus reduction function are generalisations of lines 13 and 15 of the last ADDUP example. The rotate of the leading partitions on line 5 changes them into trailing partitions, and the plus scan is carried out along the first dimension so that the data can

be a matrix. Line 6 carries out the "minus negative difference operation" along the first dimension of the array.

```
      P
1 0 0 0 1 0 1 0 0
      A
 6  9  1
 1  6  7
 1  4  1
 5  7  6
10  9  6
 1  7  5
 8 10  8
 3  1  8
 4  7  8
      P PΔPLUSΔRED A
13 26 15
11 16 11
15 18 24
```

A more general accumulation function can be written in place of the ADDUP function:

```
      ∇R←ACCUMULATE A;P;⎕IO
[1]   ⍝ <A[;1]> IS A SET OF CODES.  REMAINING COLUMNS
[2]   ⍝ OF <A> IS DATA.  <R> IS THE UNIQUE SET OF
[3]   ⍝ CODES IN COUMN 1, WITH THE TOTALS OF THE DATA
[4]   ⍝ IN THE REMAINING COLUMNS.
[5]    ⎕IO←1
[6]    →(0∈ρR←A)/0
[7]    R←R[⍋R[;1];]
[8]    P←CREATEΔPARTITION R[;1]
[9]    R←(P/R[;1]),P PΔPLUSΔRED 0 1↓R
       ∇

    A
5 37 25 99
4 76 66  8
4 89 28 44
4 48 24 28
2 17 49 90
5  7 91 51
   ACCUMULATE A
2  17  49  90
4 213 118  80
5  44 116 150
```

Difference Operations

A list of the boolean difference operations are given in the appendix. At first sight they tend to look similar, and it is difficult to appreciate which ones are going to be useful. Rather than go through them all, the following functions show how the more popular difference operations are used in practice.

The greater-than negative difference operation keeps the first one in each series of ones, and sets the remaining elements to zero:

```
      A←0 1 1 0 0 1 1 1 1 0
      A>¯1↓0,A
0 1 0 0 0 1 0 0 0 0
```

This difference operation is very useful when analysing text typed by the user. For example, when checking fullscreen data input it is usually necessary to count how many words or numbers have been entered in each screen field:

```
      ∇R←WORD∆COUNT A
[1]   ⍝ <A> IS A CHARACTER MATRIX.   <R> IS THE NUMBER
[2]   ⍝ OF WORDS OR NUMBERS ON EACH ROW OF <A>
[3]    R←A≠' '
[4]    R←R>(⍴R)↑0,R
[5]    R←+/R
      ∇

      A
 ONE   TWO
12 34 6
     589
      WORD∆COUNT A
2 3 1
```

When errors are found in the text typed by the user, an error message has to be displayed describing what has gone wrong. It is nice to be able to point out the exact location of the trouble:

```
      ∇R←ERR REPORT∆ERROR TEXT;A;CR
[1]   ⍝ <ERR> IS A BOOLEAN ERROR INDICATOR WITH AN ELEMENT
[2]   ⍝ FOR EACH WORD IN THE CHARACTER VECTOR <TEXT>.
[3]   ⍝ <R> IS AN ERROR MESSAGE.
[4]    A←TEXT≠' '
[5]    A←A>¯1↓0,A
[6]   ⍝ VSAPL CARRIAGE RETURN CHARACTER.
[7]    CR←0 1 0/⎕TC
[8]    R←'INVALID ITEM:',CR,TEXT,CR,A\ERR\'^'
      ∇

      0 0 1 0 REPORT∆ERROR ' ONE    TWO THRE   FOUR'
INVALID ITEM:
  ONE    TWO THRE   FOUR
           ^
```

Lines 4 and 5 flag the first letter of each word in the TEXT, which is used to position the caret on line 8.

Two algorithms for checking numbers were published in Quote Quad, and they both exhibit the use of difference operations. Algorithm 139 by Gerald Bamberger in the March 1980 issue of Quote Quad (Vol 10 No 3) verifies numeric input, and is similar to the Quad function available on Sharp APL and APL*Plus APL.

```
      ∇R←VI A
[1]   ⍝ VERIFY NUMERIC INPUT.
[2]   ⍝ <A> IS A CHARACTER VECTOR CONTAINING GROUPS OF
[3]   ⍝ CHARACTERS DELIMITED BY ONE OR MORE SPACES.
[4]   ⍝ <R> IS A BOOLEAN VECTOR WITH A 1 WHERE THE
[5]   ⍝ CHARACTER GROUP IS A VALID NUMBER.  NUMBERS
[6]   ⍝ OUTSIDE RANGE  (⌈/⍳0)<A<⌊/⍳0  COUNTED AS VALID.
[7]    R←' 1111111111345'[' 0123456789.¯E'⍳'0 ',A]
[8]    R←1↓≠((R∊'234')∨R≠¯1↓' ',R)/R
[9]    R←R∊(8 3⍴0 41 431)+1 12 121 21 31 312 3121 321∘.×1 100
       1000
       ∇

      VI'123 1.3  3¯4  3.4 3E4  ¯3E¯34  E3'
1 1 0 1 1 1 0
```

A slightly simpler version, excluding E notation values, may be preferred:

```
      ∇R←ΔVI A
[1]   ⍝ VERIFY NUMERIC INPUT.
[2]   ⍝ <A> IS A CHARACTER VECTOR CONTAINING GROUPS OF
[3]   ⍝ CHARACTERS DELIMITED BY ONE OR MORE SPACES.
[4]   ⍝ <R> IS A BOOLEAN VECTOR WITH A 1 WHERE THE
[5]   ⍝ CHARACTER GROUP IS A VALID NUMBER.  NUMBERS
[6]   ⍝ OUTSIDE RANGE  (⌈/⍳0)<A<⌊/⍳0  COUNTED AS VALID.
[7]    R←' 1111111111234'[' 0123456789.¯'⍳'0 ',A]
[8]    R←1↓≠((R∊'23')∨R≠¯1↓' ',R)/R
[9]    R←R∊1 12 121 21 31 312 3121 321
       ∇
```

Line 8 of the function uses the not-equal negative difference operation to remove duplicates.

Algorithm Number 146 by Jeffrey Multack, in the September 1980 issue of Quote Quad (Vol 11 No 1) converts numeric input:

```
      ∇R←FI A;M
[1]   ⍝ VERIFY AND CONVERT NUMERIC INPUT.
[2]   ⍝ <A> IS A VECTOR CONTAINING GROUPS OF CHARACTERS
[3]   ⍝ DELIMITED BY ONE OR MORE SPACES.   <R> IS A
[4]   ⍝ NUMERIC VECTOR WITH VALID NUMBERS IN <A> OR
[5]   ⍝ ZERO FOR ANY GROUP WHICH IS NOT A NUMBER.
[6]    R←VI' ',A
[7]    →(∨/R)↓0
[8]   ⍝ FORM MASK FOR VALID CHARACTER GROUPS.
[9]    M←A≠' '
[10]   M←M>¯1↓0,M
[11]   M←≠\M\R≠¯1↓0,R
[12]   R[R/⍳⍴R]←⊂M/A
       ∇

      FI'123 1.3  3¯4  3.4 3E4  ¯3E¯34 E3'
123 1.3 0 3.4 30000 ¯3E¯34 0
```

Lines 10 and 11 have been altered slightly so that they are in the same form as the difference operations already given. Line 10 is the greater-than difference operation which flags the first character in each group. Line 11 then extends the ones for each character group that is valid.

```
      A←'1234  1..23    460'
      ⎕←M←A≠' '
1 1 1 1 0 0 1 1 1 1 1 0 0 0 0 1 1 1
      ⎕←M←M>¯1↓0,M
1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0
      ⎕←R←VI A
1 0 1
      ⎕←M←≠\M\R≠¯1↓0,R
1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1
```

Line 11 can be broken down into 3 stages, a not-equal difference operations:

```
      R≠¯1↓0,R
1 1 1
```

an expansion:

```
      M\R≠¯1↓0,R
1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0
```

and a not-equal scan:

```
      ≠\M\R≠¯1↓0,R
1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1
```

Not equals scan has the property of switching from 1 to 0 and from 0 to 1 every time a 1 is encountered in the vector. The not-equals difference operation does the opposite, so the three stages are: apply a transformation, expand, then put it back to what it was. The partitioned plus reduction does a similar task:

```
R←R-¯1↓0,R←P/+\A
```

Apply a transformation (plus scan), compress, then put it back using the minus negative difference operation. The conversion back to the original form is possible because the minus negative difference operation is the inverse of plus scan.

```
      ⎕←A←+\2 4 1 5
2 6 7 12
      A-¯1↓0,A
2 4 1 5
```

Also, the not-equals difference operation is the inverse of not-equals scan:

```
      □←A←≠\1 1 0 0 1
1 0 0 0 1
      A≠¯1↓0,A
1 1 0 0 1
```

Line 11 of FI is so useful that it should be defined as a function:

```
      ∇R←P PΔMASK A
[1]   ⍝ <P> IS A LEADING PARTITION VECTOR. <A> IS A BOOLEAN
[2]   ⍝ VECTOR WHERE ρA IS IDENTICAL TO +/P.
[3]   ⍝ <R> IS A BOOLEAN VECTOR WITH 1 IN EACH PARTITION
[4]   ⍝ WHERE <A> IS 1.
[5]    R←≠\P\A≠¯1↓0,A
      ∇

      1 0 0 0 1 0 1 0 0 PΔMASK 1 0 1
1 1 1 1 0 0 1 1 1
```

Having seen that the basic process is difference, expand, scan, with not-equals, an equivalent function can be written using the same basic process, but with minus and plus:

```
      ∇R←P PΔREPLICATE A
[1]   ⍝ <P> IS A LEADING PARTITION VECTOR.  <A> IS A
[2]   ⍝ NUMERIC VECTOR WHERE ρA IS IDENTICAL TO +/P.
[3]   ⍝ <R> IS A NUMERIC VECTOR WITH EACH ELEMENT OF
[4]   ⍝ <A> REPLICATED IN EACH PARTITION.
[5]    R←+\P\A-¯1↓0,A
      ∇

      1 0 0 0 1 0 1 0 0 PΔREPLICATE 2 6 3.2
2 2 2 2 6 6 3.2 3.2 3.2
```

Using the partitioned plus reduction technique of difference, compress, scan, but with not-equals in place of minus and plus, yields a partitioned not-equals reduction function:

```
      ∇R←P PΔNEΔRED A
[1]   ⍝ <P> IS A LEADING PARTITION VECTOR.   <A> IS A
[2]   ⍝ BOOLEAN VECTOR.  <R> IS ≠/ FOR EACH PARTITION.
[3]    R←(1⌽P)/≠\A
[4]    R←R≠¯1↓0,R
      ∇
```

This function can be used to flag partitions with an uneven number of occurrences.

Another 'tool-box' function that illustrates a difference operation is one for removing surplus spaces:

```
      ∇R←SQUEEZE A;B
[1]   ⍝ REMOVE LEADING, TRAILING AND DUPLICATE SPACES
[2]   ⍝ FROM CHARACTER VECTOR <A>
[3]    R←A,' '
[4]    B←' '≠R
[5]    B←B∨¯1↓0,B
[6]    R←¯1↓B/R
       ∇
```

Line 5 has an 'or' negative difference operation which adds a 1 after each group of ones. Line 3 guarantees a trailing space which is then removed on line 6.

The following function is one of a set of functions to help formatting numeric data in VS APL:

```
      ∇R←BRACKETS A;B;C;D
[1]   ⍝ <A> IS A CHARACTER ARRAY OF FORMATTED NUMBERS
[2]   ⍝ WITH AT LEAST ONE SPACE BEFORE EACH NUMBER.
[3]   ⍝ <R> HAS THE NUMBERS MOVED ONE SPACE TO THE LEFT
[4]   ⍝ AND HAS BRACKETS IN PLACE OF NEGATIVE SIGNS.
[5]    R←1⌽,A
[6]    B←R≠' '
[7]    C←B>¯1↓0,B
[8]    D←'¯'=C/R
[9]    R[(C\D)/⍳⍴R]←'('
[10]   C←B<¯1↓0,B
[11]   R[(C\D)/⍳⍴R]←')'
[12]   R←(⍴A)⍴R
       ∇

      NUMS
 .00   368.55    ¯.35   ¯40.10    ¯.34
 .00     .00    6.43   536.58    ¯.04
 .00     .00     .00  ¯761.24     .42

      BRACKETS NUMS
.00   368.55   (.35)   (40.10)   (.34)
.00     .00   6.43    536.58    (.04)
.00     .00    .00   (761.24)    .42
```

Line 7 is a greater-than negative difference which flags the beginning of each group of numbers. Line 10 is a less-than negative difference which flags the beginning of each group of spaces. The negative sign is therefore only replaced by both a left and right parenthesis.

The function was created to help develop a generalised formatter for VS APL. In practice, a formatting function would have the format specification available to indicate where the right parenthesis should be placed.

Partition Functions

Partition functions have been given for plus reduction and not-equal reduction, but partition functions are needed to carry out the equivalent of all the reduction and scan operations. The working Memorandum on Boolean Techniques gives a very comprehensive list, but here are two that are most frequently used:

```
      ∇R←P PΔORΔRED A
[1]   ⍝ <P> IS A LEADING PARTITION VECTOR.  <A> IS A BOOLEAN
[2]   ⍝ VECTOR.  <R> IS ∨/ FOR EACH PARTITION OF <A>.
[3]    R←(P∨A)/P
[4]    R←(P/A)≥R/1⌽R
       ∇

      ∇R←P PΔANDΔRED A
[1]   ⍝ <P> IS A LEADING PARTITION VECTOR.  <A> IS A BOOLEAN
[2]   ⍝ VECTOR.  <R> IS ∧/ FOR EACH PARTITION OF <A>.
[3]    R←(P≥A)/P
[4]    R←(P/A)∧R/1⌽R
       ∇
```

These functions are also published in the FinnAPL Idiom Library numbers 491 and 492.

An example of their use is taken from a set of functions to carry out formatting under VS APL:

```
      ∇R←BLANKΔIFΔZERO A;B;C;P
[1]   ⍝ <A> IS A CHARACTER ARRAY OF FORMATTED NUMBERS.
[2]   ⍝ <R> HAS ALL ZERO NUMBERS SET TO SPACES.
[3]    R←A
[4]    →(0∈B←⍴R)/0
[5]    R←,R
[6]    P←R=' '
[7]    P←P>¯1↓0,P
[8]    P[⎕IO]←1
[9]    C←P PΔMASK~P PΔANDΔRED R∈' 0.'
[10]   R←B⍴C\C/R
       ∇

      NUMS
 .00    368.55     ¯.35    ¯40.10     ¯.34
 .00       .00     6.43     536.58     ¯.04
 .00       .00      .00    ¯761.24      .42

      BLANKΔIFΔZERO NUMS
        368.55     ¯.35    ¯40.10     ¯.34
                   6.43     536.58     ¯.04
                           ¯761.24      .42
```

Lines 6 to 8 set up a partition vector, and line 9 creates a mask for those partitions that do not have a space, zero or decimal point. Of course, if the format specification is available the partition vector can be set up without searching the data.

Finally, an example of using partition functions to eliminate looping. In the last issue of the APL User Group News Letter Dick Bowman gave a very interesting problem of calculating geometric means of sets of data.

The solution published calculated the geometric mean of each set of data in a loop, which would be inefficient if large amounts of data were involved and Dick ends his article with 'There surely must be a better way'. The partitioned data approach would be to create a partitioned geometric mean function.

```
      ∇R←P PΔGEOM A;B;C;D
[1]   ⍝ <P> IS A PARTITION VECTOR, <A> IS A NUMERIC VECTOR.
[2]   ⍝ <R> IS THE GEOMETRIC MEAN IN EACH PARTITION WHERE
[3]   ⍝ ALL NUMBERS ARE GREATER THAN ZERO, OTHERWISE ¯1.
[4]    B←P PΔANDΔRED A>0
[5]    C←P PΔMASK B
[6]    D←C/P
[7]    R←*(D PΔPLUSΔRED⍤C/A)÷PΔSHAPE D
[8]    R←(B\R)-~B
       ∇

      ∇R←PΔSHAPE P
[1]   ⍝ <P> IS A LEADING PARTITION VECTOR.
[2]   ⍝ <R> IS THE NUMBER OF ELEMENTS IN EACH PARTITION.
[3]    R←(1⌽P)/⍳⍴P
[4]    R←R-¯1↓0,R
       ∇

      1 0 0 0 1 0 1 0 0 PΔGEOM 3 6 8 1 0 2 3 2 1
3.464101615 ¯1 1.817120593
```

Line 4 finds the partitions that need to be processed, and line 5 creates a mask. Line 7 then calculates the geometric mean for each valid partition, and line 8 sets invalid partitions to negative one. This function can then be used to replace the loop in the original function, after having created a partition vector.

Conclusion

Difference operation and partition functions provide a useful set of tools which help to solve the programming problems where the data does not fit in with rectangular nature of APL arrays.

The generalised array facilities in APL2, NARS and Sharp APL developments provide much more powerful tools for manipulating non rectangular data. The partitioned data approach is much easier and more direct with generalised arrays; the partitioned plus reduction is merely a plus reduction for each element of a vector of vectors, and a proper notation is provided for its application. Roll on generalised arrays — but in the meantime we can make do quite successfully with partitioned functions!

APPENDIX

Boolean Difference Operations

Less-than Negative Difference. The first of each group of zeros is set to one, all other elements are set to zero.

```
      □←A←0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
0 0 1 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A<¯1↓0,A
0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0
```

Less-than Positive Difference. The last of each group of zeros is set to one, all other elements are set to zero.

```
      A
0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A<1↓A,0
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0
```

Less-than-or-equal Negative Difference. The first of each group of ones is set to zero, all other elements are set to one.

```
      A
0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A≤¯1↓1,A
1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0
```

Less-than-or-equal Positive Difference. The last of each group of ones is set to zero, all other elements are set to one.

```
      A
0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A≤1↓A,1
1 1 1 1 1 0 1 1 1 1 0 1 1 1 0 1 1
```

Equal Negative Difference. The first of each group of zeros and the element to the right of each group of zeros is set to zero, all other elements are set to one.

```
      A
0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A=¯1↓1,A
0 1 0 1 1 1 0 1 0 1 1 0 1 1 0 0 0
```

Equal Positive Difference. The element to the left of each group of zeros and the last of each group of zeros is set to zero, all other elements are set to one.

```
      A
0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A=1↓A,1
1 0 1 1 1 0 1 0 1 1 0 1 1 0 0 0 1
```

Greater-than-or-equal Negative Difference. The first of each group of zeros is set to zero, all other elements are set to one.

```
      A
0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A≥¯1↓1,A
0 1 1 1 1 1 0 1 1 1 1 0 1 1 1 0 1
```

Greater-than-or-equal Positive Difference. The last of each group of zeros is set to zero, all other elements are set to one.

```
      A
0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A≥1↓A,1
1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 1
```

Greater-than Negative Difference. The first of each group of one is set to one, all other elements are set to zero.

```
      A
0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A>¯1↓0,A
0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1
```

Greater-than Positive Difference. The last of each group of ones is set to one, all other elements are set to zeros.

```
      A
0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A>1↓A,0
0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 1
```

Not-equal Negative Difference. The first of each group of ones and the element to the right of each group of ones is set to one, all other elements are set to zero.

```
      A
0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A≠¯1↓0,A
0 0 1 0 0 0 1 0 1 0 0 1 0 0 1 1 1
```

Not-equal Positive Difference. The element to the left of each group of ones and the last of each group of ones is set to one, all other elements are set to zero.

```
      A
0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A≠1↓A,0
0 1 0 0 0 1 0 1 0 0 1 0 0 1 1 1 1
```

Or Negative Difference. The element to the right of each group of ones is set to one, all other elements are unaltered.

```
      A
0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A∨¯1↓0,A
0 0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 1
```

Or Positive Difference. The element to the left of each group of ones is set to one, all other elements are unaltered.

```
      A
0 0 1 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A∨1↓A,0
0 1 1 1 1 1 1 0 1 1 1 1 0 0 1 1 1 1
```

And Negative Difference. The element to the right of each group of zeros is set to zero, all other elements are unaltered.

```
      A
0 0 1 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A∧¯1↓1,A
0 0 0 1 1 1 1 0 0 0 1 1 0 0 0 0 0 0
```

And Positive Difference. The element to the left of each group of zeros is set to zero, all other elements are unaltered.

```
      A
0 0 1 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
      A∧1↓A,1
0 0 1 1 1 1 0 0 0 1 1 0 0 0 0 0 0 1
```