Redistribution of Totals Through Hierarchical Data

An Application of Benkard's Distributed Round

Adrian Smith
Causeway Graphical Systems Ltd.
adrian@causeway.co.uk

Abstract

The early days of APL are full of examples of spreadsheet-style applications which allow the user to maintain aggregated values and quickly redistribute the changes across base data. Generally these use a simple pro-rata approach which always leads to rounding problems where the adjusted values no longer sum to the required total.

The example discussed in this paper was developed for the Strategic Planning manager at Nestlé-Rowntree and a key requirement was to permit rebalancing of adjusted totals throughout an arbitrarily deep tree-structure. In this case a simple pro-rata leads to endless imbalances, and the trick is to use Benkard's distributed rounding which ensures that when changes to aggregates are redistributed, the detail always adds exactly to the required total.

The concept is illustrated using the *VideoSoft Flexgrid OCX control*. The Flexgrid was used here because it has a very simple and powerful approach to managing gridded data where one of the dimensions has an outline structure.

Why Use a Cumulative Round?

In some ways, this paper exposes my academic background as a physicist, rather than a mathematician. Physicists generally see the world in discrete chunks – if you look closely enough at anything, you can count it! The output of the biggest Hydro dam in the world could be measured as a (very) big integer – the number of electrons per unit time it pushes down the cables. This attitude was reinforced by 20 years in the chocolate industry, where everything is measured in simple whole numbers. Production quantities might be in cases, production times might be in shifts, people are 'full-day equivalents' and so on. You can't schedule an arbitrary amount of anything in this sort of world – either you run a machine for a shift or you don't. You staff it with 5 people or 4, and at the end of the day it will have produced an integer number of Aero bars.

Now consider the dilemma faced by your production planner who has a total demand for 14 units (of something), spread over 4 slots. This might be demand across weeks, or load across machines or whatever. The problem is the same:

```
      vv←4 3 2 5
      +/vv
14
```

He is told to cut the total to 13 units, so he gets out his trusty APL interpreter and tries:

```
      ∇ vec←vec pr tgt
[1]     ⍝ Simple 'pro-rata' approach
[2]       vec←tgt×vec÷+/vec
      ∇

      vv pr 13
3.714285714 2.785714286 1.857142857 4.64
2857143
      ⌊.5+vv pr 13
4 3 2 5
      +/⌊.5+vv pr 13
14
```

...and comes straight back to the number he first thought of! This kind of problem has plagued planning and budgeting software for decades, and usually leads to accountants claiming that "the computer can't add up" if the rounded figures are shown on screen next to the (correct) total. The problem is compounded as soon as you try to modify the base data – change that apparent '3' to 4 and you actually changed the internal value of 3.714... to 4, but the visible total stayed the same. Change the visible number back to 3 and suddenly the total becomes 12 as you now have an internal value of 3.00... in memory – confused? You should be!

My feeling is that *what is shown on the screen should exactly reflect what is held in memory* at all times, and that in an integer world that means integers everywhere. Consequently we need to solve the general problem of how to manage the pro-rata process such that the result is always integer, and that it always adds back *exactly* to the number the user asked for. As you will see when we explore the more complex case of rebalancing totals up and down a tree-structure, maintaining an exact correspondence becomes essential if we are to avoid the possibility of an unbounded loop where our algorithm may never terminate.

Fortunately, the trick is remarkably easy, and was mentioned almost in passing in Phil Benkard's excellent Stanford paper "Dance of the Rounds" [2]. If you first cumulate the numbers:

```
      vc←+\vv
      vc
4 7 9 14
```

...then do your pro-rata

```
      vc × 13 ÷ 14
3.714285714 6.5 8.357142857 13
```

You will notice immediately that the last number is exactly correct, as it would be. Now we just need the first difference of the rounded intermediate:

```
      ⌊.5+vc × 13 ÷ 14
4 7 8 13
      ¯2-/0,⌊.5+vc × 13 ÷ 14
4 3 1 5
```

This method has the obvious merits of simplicity and speed, and you can probably see that it extends naturally to higher dimensions. It can also adapt to handle an arbitrary scattering of 'held' values, which is also very necessary in many planning models. Maybe the first few weeks of production are 'locked in', or maybe you promised Big Bob next Friday as holiday and he is likely to hit you if you change the plan now. Here is a simple extension of the method for the vector case, but with the addition of a matching vector of held items.

This also adds the small refinement of a 'lot size' which would normally be left as 1 (for the integer case) but could be .01 (figures to the nearest cent) or 12 (chocolate bars are usually packed in dozens) or whatever. Also it handles the 'empty' case when the original data is all zeros.

```
∇ new←larg prh ttl;holds;old;
    cum;held;lot
   ⍝ Integer vector redistribution
    with optional held items.
   ⍝ Result is 'integer' at the
    specified lotsize

    :If 1<|≡larg
      (old holds lot)←3↑larg,1
    :Else
      (old holds lot)←larg 0 1
    :End

    :If 0∊holds  ⍝ we can proceed
      held←holds×old
      cum←+\((-holds)×old)÷lot
      :If 0=¯1↑cum
 ⍝ Even split if old is all zero
        cum←+\-holds
      :End
     ⍝ End-stop required if the
      only unheld cell is zero!
      :If 0=¯1↑cum ⋄ new←old
           ⋄ :Return ⋄ :End
      new←¯2-/0,⌊0.5+cum×
         ((ttl-+/held)÷lot)÷¯1↑cum
      new←held+lot×new
    :Else  ⍝ all cells locked in
      new←old
    :End
∇
```

Some simple examples should make the effect clear:

```
      vv prh 17
5 4 2 6
      vv (1 0 0 0) prh 17
              ⍝ first item held
4 4 3 6
      vv (1 1 0 0) prh 17
              ⍝ 2 items held
4 3 3 7
      vv (1 1 1 0) prh 17
              ⍝ 3 items held
4 3 2 8
      vv (1 1 1 1) prh 17
              ⍝ All items held
4 3 2 5
```

As you can see, the effect of holding items is to 'push' more and more of the adjustment into the remaining values. Sales plans very often develop just such an end-of-year bulge as an unrealistic total is forced upon increasingly unco-operative data. Now for some lot-sized examples:

```
      vv 0 1 prh 24
                ⍝ Default case
7 5 3 9
      vv 0 2 prh 24
                ⍝ Even numbers required
6 6 4 8
      vv 0 12 prh 24
                ⍝ Dozens please
12 0 0 12
      vv 0 .01 prh 24
                ⍝ Cents allowed
6.86 5.14 3.43 8.57
      vv (1 0 1 0) .01 prh 24
                ⍝ ... with holds
4 6.75 2 11.25
```

Which completes the basic toolkit. Just to give an idea of speed:

```
      bigv←?60000⍴12
      +/bigv
389973
      +/bigv prh 500000
500000
      +/bigr←bigv 0 10 prh 500000
500000
      12↑bigr
10 0 10 10 10 0 20 10 10 0 10 10
```

The redistribution step runs in just under 0.4s on a typical APL machine (Pentium-II, running Dyalog APL) which does tend to impress spreadsheet users, who expect these things to take several minutes, if they ever complete at all.

Extending the Algorithm to a Tree Structure

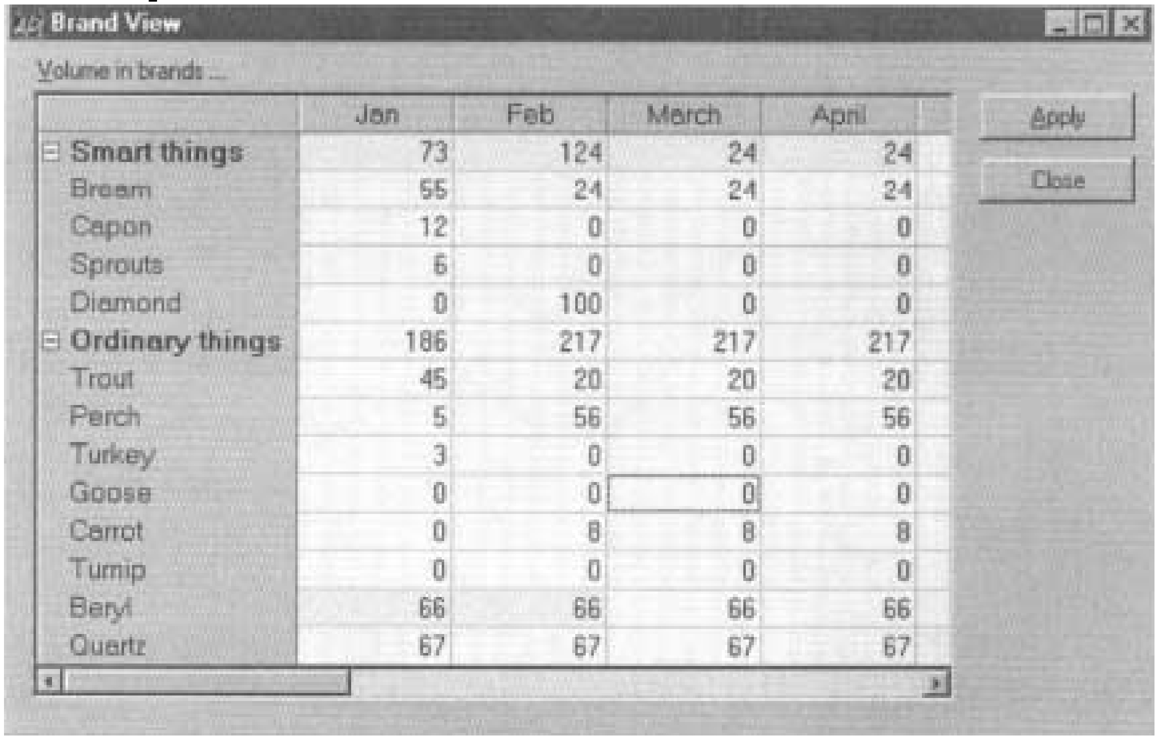
The previous example (carefully wrapped in a Causeway 'class') was implemented as a Gui component in the distribution planning software that was built during 1997 for Nestlé UK Ltd., as a replacement for a long-surviving (and much loved) mainframe APL program which did many of the same things. An interesting new problem was posed early in 1999 by the strategic planning manager, who wanted to explore the idea of planning either on the basis of brand, or on a completely new 'category' structure.

Traditionally, Rowntree organised its marketing effort around well-defined 'brands' such as Kit Kat, After Eight, Polo and so on. Brands were run by brand-managers, who reported to 'product group' managers and so on. Typically the product-group manager would be aiming to meet a revenue target for the year, and would split this down among his subordinates. They in turn would look at the various components (Kit Kat comes in 2-finger and 4-finger and now 'chunky' sizes, some of which may be packaged in collections of 6 or 8 bars) and try to decide where best to direct the marketing effort, and hence the promotional and advertising budget.

His problem was that the big retailers (an increasing power in the land) saw things very differently. They would see a 4-finger Kit Kat positioned alongside Polo Mints as a 'stick it near the checkout' impulse buy. On the other hand the multiple pack of 2-finger bars was a grocery item, probably sold to mothers looking to make up a school lunch box. Here we have a classic problem – at the lowest level there are only the individual items (called SKUs in the trade) which we want to assemble into a tree structure in two very different ways. It is then desirable to play with the data at an aggregate level by brand (say total 'Boxed Chocolates') or by retailer category. Both trees can in principle be quite deep (4 or 5 levels) and have no structural elements in common. Arbitrary items may be held at any level in the tree, and of course what is a constituent item of a subtree may itself be a total item from a lower level.

To help the management see what was a practical and sensible approach, a series of simple concept models were developed in Dyalog APL using the distributed round, and the VideoSoft FlexGrid as the primary Gui component. The interface to the Flexgrid is actually marginally easier under APL+Win [1] but at the time it lacked the other Gui design tools which I needed to build something quickly. The main reason for using FlexGrid was the very simple way it manages an outline structure. This can be done with the Dyalog Grid control, but not in such a simple and efficient way.

Some Sample Data



	Jan	Feb	March	April
Smart things	73	124	24	24
Bream	55	24	24	24
Capon	12	0	0	0
Sprouts	6	0	0	0
Diamond	0	100	0	0
Ordinary things	186	217	217	217
Trout	45	20	20	20
Perch	5	56	56	56
Turkey	3	0	0	0
Goose	0	0	0	0
Carrot	0	8	8	8
Turnip	0	0	0	0
Beryl	66	66	66	66
Quartz	67	67	67	67

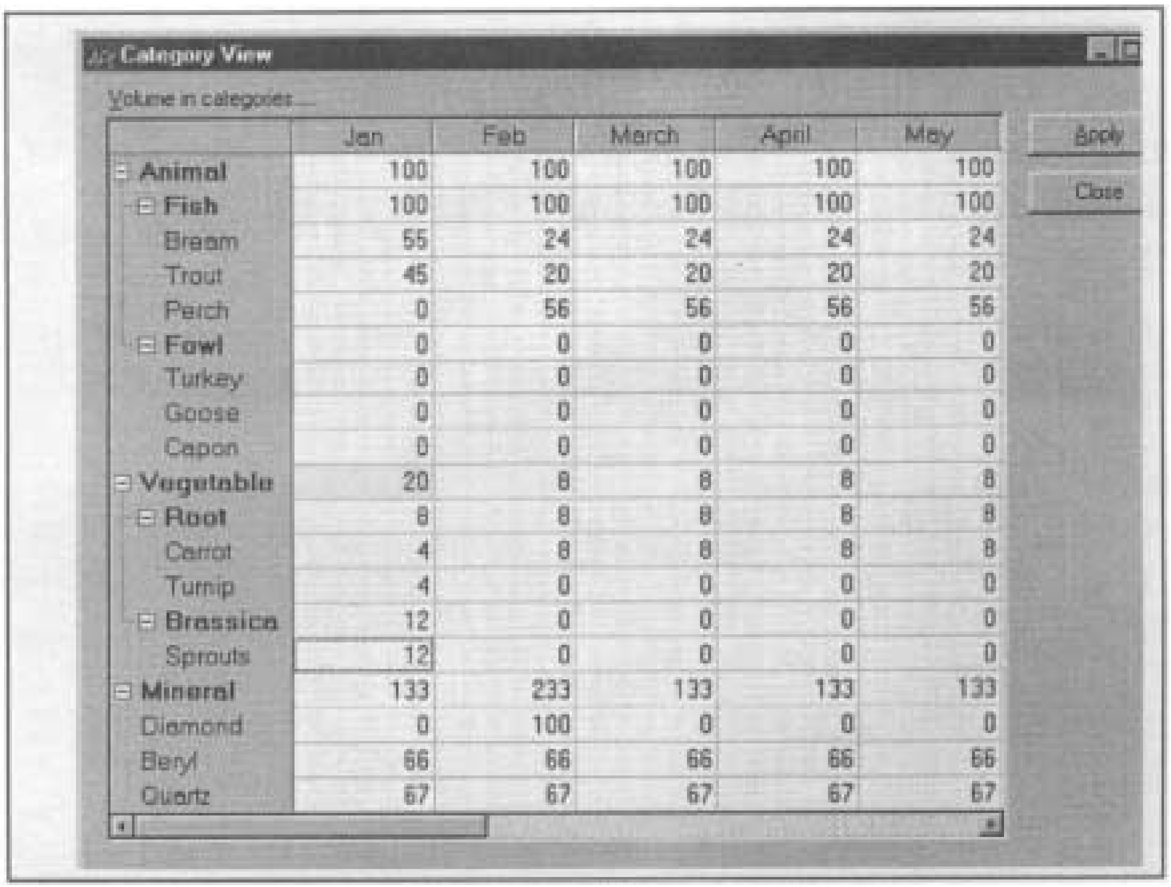
This is an intentionally simple structure, but it is enough to show the three basic things that may happen when a number is changed. The cells on a coloured background are 'held', and may only be changed by the user. In this case we have fixed the brand budget for 'Smart things' for the first 4 months, and the 'Beryl' SKU is locked in for 2 months. The possibilities are:

1. A total may be changed. The change is redistributed over the (unheld) constituent items. If all the items are held, we must ensure that the change is 'bounced' by adding up what we did and resetting the original cell.
2. A constituent of an **unheld** total may be changed. This is the easy one – we just recalculate the total.
3. A constituent of a **held** total may be changed. We must share out the balance between the unheld siblings. Again we should recover if all our siblings are held by resetting the original cell.

You can play some mental games with the example above, or download this toy workspace from the Causeway web site at www.causeway.co.uk/freestuf to see how it actually works. Of course in a more complex tree, the changes may ripple out in both directions, and a total of one subtree may become the sibling of some other totals, which are themselves part of a held (higher-level) total. So as well as rippling up, changes may get reflected back down!

Here are the same basic items, but shown in a more complex 'category' tree:

Here I have held the January total for Vegetable – if I change Sprouts from 12 to 10, we operate rule-2 and set the total for Brassica to 10. However we cannot ripple up any further so we set Root to 10 (rule-3), which ripples down to be shared between Carrot and Turnip (rule-1). If I hold Carrot, then Turnip must take the entire adjustment; if both are held the attempted change must bounce all the way back and reset the originally altered cell back to 12! This is one of those problems which you start off by viewing as a set of *If-Then* clauses and rapidly realise that you are in an infinite tree of possibilities. Instead you must simply apply the three rules as you hit them, marking as 'todo' any cell that you touch. Then the entire propagation algorithm becomes 'loop while there are any cells to do' and (being a strict tree rather than a network) eventually the fires die out and you cannot loop for ever.


Category View

Volume in categories...

	Jan	Feb	March	April	May
⊟ **Animal**	100	100	100	100	100
⊟ **Fish**	100	100	100	100	100
Bream	55	24	24	24	24
Trout	45	20	20	20	20
Perch	0	56	56	56	56
⊟ **Fowl**	0	0	0	0	0
Turkey	0	0	0	0	0
Goose	0	0	0	0	0
Capon	0	0	0	0	0
⊟ **Vegetable**	20	8	8	8	8
⊟ **Root**	8	8	8	8	8
Carrot	4	8	8	8	8
Turnip	4	0	0	0	0
⊟ **Brassica**	12	0	0	0	0
Sprouts	12	0	0	0	0
⊟ **Mineral**	133	233	133	133	133
Diamond	0	100	0	0	0
Beryl	66	66	66	66	66
Quartz	67	67	67	67	67

Managing the Propagation Rules

I would not claim that the code which follows is 'production quality' – remember that this was always meant to be a concept project, not a rock-solid application. However it is worth quoting in full, as it also illustrates quite nicely some of the techniques for interacting with the FlexGrid object. Our 'AfterEdit' event handler is simply given the row/col index of the changed cell – first it must grab the new (numeric) cell content:

```
⍝ Good ol'VB has a very tolerant execute
⍎
 val←this ⎕WG('ValueMatrix'cellfmt r c)
⍝ Propagate the new value and update the
 changed cells
 this Set r c val
```

All the hard work is done by the `Set` function, but first it is worth a look at that formatting utility:

```
     ∇ txt←kwd cellfmt iv
       ⍝ Format cell range
         txt←1↓∊';',¨⍕¨iv
         txt←kwd,'[',txt,']'
     ∇

               'Hello' cellfmt 3 4
Hello[3;4]
```

Dyalog have taken the approach of making the indexed properties look as much like APL expressions as possible. When you are just mucking around, it is quite natural to type expressions like `obj ⎕WG 'ValueMatrix[12;5]'`, but generating these index sets as character vectors becomes something of a nuisance when you are given the indices as numbers in a callback. This is one place where the APL+Win syntax `val←⎕WI 'ValueMatrix' 12 5` is possibly less pretty, but is much easier (and more efficient) to work with.

```
     ∇ {r}←this Set arg;row;col;value;inx;leaf;todo;tgt;subtree;sum;old;hld;PRH;CF;FM
       ⍝ Update array at [cell] with new value and ripple up or down as required.
       ⍝ <level> <holds> and <array> are global to the instance of the Gui object
         (row col value)←arg ⋄ r←1
       ⍝ Copy the functions we need and work in the instance from here on
         PRH←prh ⋄ CF←cellfmt ⋄ FM←fmt

```

```
:With this
  leaf←1↓(0,level)≥level,0
  todo←(⍴array)⍴0
  array[row;col]←value ⋄ todo[row;col]←2
 ⍝ Keep the visuals in line (this value may get changed later)
  ⎕WS('TextMatrix'CF row col)(FM value)

  :While todo[;col]∨.>0
    inx←(todo[;col]>0)⍳1
   ⍝ Is it a total item?
    :If 0=inx⊃leaf ⍝ Distribute over children
    :AndIf 2=inx⊃todo[;col]
      subtree←inx+⍳+/∧\(inx↓level)>inx⊃level
     ⍝ Direct descendents get the distributed total
      tgt←(level[subtree]=1+inx⊃level)/subtree
      old←array[tgt;col] ⋄ hld←0<holds[tgt;col]
      array[tgt;col]←0⌈old hld PRH array[inx;col] ⍝ No negatives!
      array[inx;col]←+/array[tgt;col]
      ⎕WS('TextMatrix'CF inx col)(FM array[inx;col])
      tgt←(old≠array[tgt;col])/tgt
      todo[tgt;col]←2    ⍝ Add changed cells to our 'todo' list
      :For r :In tgt     ⍝ Repaint all of them
        ⎕WS('TextMatrix'CF r col)(FM array[r;col])
      :End
    :End

   ⍝ Ripple upwards fixing parent totals
    :If (inx⊃level)>0
      tgt←inx-(⌽(inx-1)↑level)⍳¯1+inx⊃level
      subtree←tgt+⍳+/∧\(tgt↓level)>tgt⊃level
     ⍝ Sum all items in the subtree at the next level only
      subtree←(level[subtree]=1+tgt⊃level)/subtree
      sum←+/array[subtree;col]
      :If sum≠array[tgt;col] ⍝ ... if it changed
        :If 0=holds[tgt;col]
          array[tgt;col]←sum ⋄ todo[tgt;col]←1   ⍝ Ripple up
          ⎕WS('TextMatrix'CF tgt col)(FM sum)
        :Else ⍝ Distribute total between siblings with ourself held
          old←array[subtree;col] ⋄ hld←0<holds[subtree;col]
          hld[subtree⍳inx]←0
         ⍝ Not more than the total less held siblings!
          :If old[subtree⍳inx]>array[tgt;col]-hld+.×old
            old[subtree⍳inx]←array[tgt;col]-hld+.×old
            todo[inx;col]←102  ⍝ Redo myself again again!
          :End
          hld[subtree⍳inx]←1   ⍝ Hold self
          array[subtree;col]←old hld PRH array[tgt;col]
          sum←+/array[subtree;col]   ⍝ May not add up
          subtree←(old≠array[subtree;col])/subtree
          todo[subtree;col]←2 ⍝ Ripple down from changed cells
          :For r :In subtree  ⍝ Paint each cell
            ⎕WS('TextMatrix'CF r col)(FM array[r;col])
          :End
          :If sum≠array[tgt;col]  ⍝ Some problem, so ripple up
```

```
                 array[tgt;col]←sum ⋄ todo[tgt;col]←1
                 ⎕WS('TextMatrix'CF tgt col)(FM sum)
               :End
             :End
           :End
         :End
         todo[inx;col]←0⌈todo[inx;col]-100
       :EndWhile

      ⍝ Set the final outcome in the target cell
       ⎕WS('TextMatrix'CF row col)(FM array[row;col])
     :EndWith
   ∇
```

I think the only subfunction not detailed is `fmt`, which simple formats any data, ravels the result and replaces the high-bar with a conventional minus.

It is interesting to speculate on the way this function would have looked before we had control-structures in APL! Probably it would have been split over several smaller functions, but perhaps it would never have been written at all? There are some tasks which are intrinsically iterative and scalar, and for which APL can only borrow its tools of thought from more conventional programming languages.

Summary

The real core of this paper is the simple insight that redistribution of changes to totals is best carried out on cumulated data. You can probably imagine what might happen in that **Set** function if a total was ever allowed to differ from the sum of its parts. We can be confident that the algorithm will terminate because we have a guarantee that when we propagate changes through the tree we will always be in a position where everything remains balanced, and all summations are sure to be valid.

References

[1] Jonathan Barman. VideoSoft FlexGrid Pro with APL+Win. Vector Vol. 15 No.3, 77.

[2] Phil Benkard. Dance of the Rounds. APL91Conference Proceedings.