CSM-212

Round-Pound Arithmetic

Iain MacCallum,

macci@essex.ac.uk
Department of Computer Science
University of Essex.
Colchester CO4 3SQ
United Kingdom
Tel: +44 1206 872791

October 1994

Abstract

Rounded Arithmetic is the traditional way of presenting previous year accounts. The method of showing rounded sums of vectors in which the arithmetic still “adds up” and the errors in the display of the components are bounded by twice the maximum rounding error have been well known for at least 30 years in spite of there being little or nothing in the literature and no attempt to implement this in currently available spread-sheets. This paper extends the method from vectors to accumulator trees, and suggests a heuristic algorithm for the more general accumulator graphs in which nodes accumulate into more than one accumulator. The special case of cross-tabulations, in which all nodes accumulate to 2 accumulators represents is considered by both recursive and non-recursive methods.

1 Introduction

Rounded Arithmetic is the traditional way of presenting previous year accounts. The method of showing rounded sums in which the arithmetic still “adds up” and the errors in the display of the components are bounded by twice the maximum rounding error have been well known for at least 30 years in spite of there being little or nothing in the literature. In the academic sphere, use of spread-sheets for financial reporting appears to have led to a tacit acceptance that the least significant digit in any column of numbers is likely to be incorrect. In practice, where correct arithmetic is deemed important, the final rounding of multi-level statements appears to be generally done by hand.

Lotus 1-2-3 offers rounding either as a function of the display in which case the arithmetic may appear incorrect, or as function applied to internal values in which case the arithmetic will be correct but the totals shown will include the effects of rounding. Lotus 1-2-3 does not offer correctly summing arithmetic on rounded values with bounded errors. Brigham and Knechel [1] present 8 sample listings in their text book containing rounded displays which do not sum correctly. In several cases the error represents £1,000. In another, the error has crept up to 6 in the final decimal place.

The need for round-pound arithmetic is shown by the following sum.

$$\begin{array}{ll} 1.49 & \\ \underline{1.35} & 2.84 \end{array}$$

To round a column to the nearest unit, it is usually a requirement that the total be displayed to within 0.5, but that the displayed components may deviate by up to twice that tolerance, namely 1.0. Thus, acceptable representations of the above are

1		or	2		but not	1		nor	1	
$\underline{2}$	3		$\underline{1}$	3		$\underline{1}$	3		$\underline{1}$	2

We call the acceptable representations *round-pound correct*. The algorithm, based on carrying forward the rounding error, is straightforward and it is easily proved that the errors are within the limits stated above.

The analysis and program below are presented in terms of rounding to the nearest unit, but apply equally to rounding to any degree of accuracy, for example, £000s.

Let the column of the full-accuracy components be the vector $v[1..n]$ and the rounded values for display be $r[1..n]$. Let the sum of the elements of v and r be *sumv* and *sumr* respectively. Let the rounding function be defined to be

```
round x  = trunc (x + 0.5),         if x >= 0
         = trunc (x - 0.4999...),   otherwise
```

The *classical round-pound algorithm* applicable to a vector $v[1..n]$ may be expressed in C as:

```
carry[0] = 0;
for (i = 1; i <= n; i++) {
    r[i] = round (v[i] + carry[i-1]);
    carry[i] = (v[i] + carry[i-1]) - r[i];
}
sumr = sumv - carry[n];
```

It is easy to see that since $carry[i]$ is of the form $x - round(x)$, $-0.5 \le carry[i] < 0.5$, and therefore that $|v[i] - r[i]| = |carry[i] - carry[i-1]| \le 1.00$. The bounds on the carry forward also ensure that an integer will never be rounded up or down to an adjacent integer. This is an important feature in the presentation of financial information, where a zero may mean something other than, say, some value between –50p and +50p.

That $sumr = sumv - carry[n]$ follows from

$$\Sigma\, carry[i] = \Sigma\, (v[i] + carry[i-1]) - \Sigma\, r[i]$$

In practice, a scalar suffices in the algorithm for *carry*.

2 Accumulator trees

The presentation of accounts is more than the display of isolated columns (or vectors) of numbers that sum correctly. In general, a component of a vector of numbers will itself be a sub-total, the sum of another vector of numbers. For example,

A:	1.49		
B:	1.39	2.88	
C:	5.20		
–D:	0.65	4.55	
E:		0.40	7.83

The structure is known as an *n-ary tree* (see Figure 1), for the lengths of the vectors (i.e. the degree of branching at any node) in such a structure will vary. Furthermore, nodes must be tagged since the values at some nodes may be added to their accumulator whilst others, e.g. D, may be subtracted.

When it is realised that

(i) accumulators form a tree, and

(ii) in a simple vector addition, rounding errors in the components are double those of the sum, i.e. the level above

it would appear that the propagation of errors in multi-level accumulators could be exponential! However, it is shown below that a simple verifiable algorithm exists whereby an accumulator tree may be rounded with the error at the root not greater than 0.50 and the error at all other nodes not greater than 1.00. In other words, errors are not propagated through the tree as one might expect.

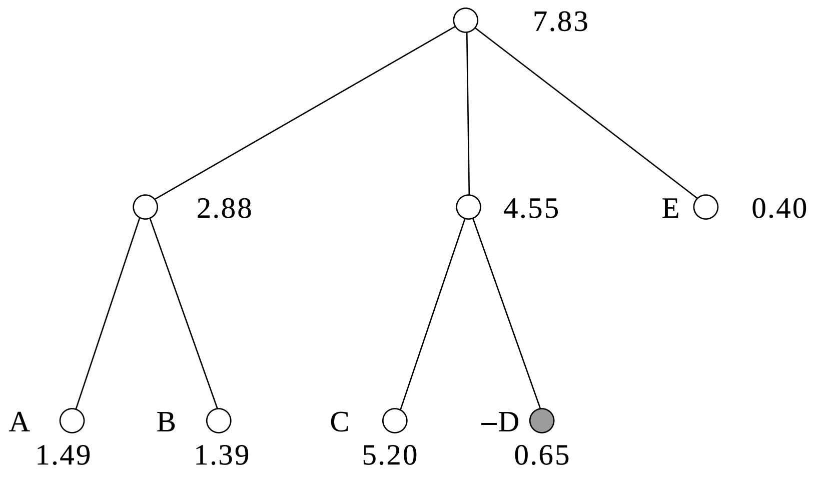

Figure 1. An accumulator tree

In the example above, applying the classical algorithm to the vectors [A, B] and [C, D], we get

A:	1.49	→	1			C	5.20	→	5	
B:	1.39	→	2	3		D	–0.65	→	0	5

Applying the algorithm to the vector [[A, B], [C, D], E] produces the rounded sum

[A, B]	2.88	→	3	
[C, D]	4.55	→	4	
E:	0.40	→	1	8

We now see an inconsistency at the sum of the vector [C, D]: its true rounded value is 5 yet, in the rounding for the grand total, it is shown as 4.

Setting aside the problems of rounding for a moment, we note that tree structures are usually best processed by recursion. Suppose the nodes of an accumulator tree are defined by the declaration (in ANSI C):

```
typedef struct acctype {
    int acc;
    int Tag;                        /* +/- to parent */
    struct acctype *parent;
} *AccTreePtr;
```

A C function for accumulating a full-accuracy amount *x* at a leaf node *t* and at all accumulators between it and the root is simply:

```
void UpdateAcc (AccPtr t, int x)
{
    t->acc = t->acc + x;

    if (t->parent != NULL) {
         if (t->Tag == minus) {
             x = -x;
         }
         UpdateAcc(t->parent, x);
    }
}
```

3 Round-Pound Accumulator Trees

In a simple accounting system, each node may contain descriptive information, a full-accuracy accumulator for this year's value and a round-pound accumulator for last year's value. In this section consider an accumulator tree node with a round-pound accumulator consisting of 3 fields, *pound*, *error* and the sign *Tag*:

```
typedef struct acctype {
    int pound;
    int error;
    int Tag;                        /* +/- to parent */
    struct acctype *parent;
} *AccTreePtr;
```

where the full-accuracy value is always

```
pound*100 + error.
```

It is obvious that for any full-accuracy value, there are many representations in this data structure. For example:

£1.49 can be represented as (0, 149), (1, 49), (2, –51), (3, –151) etc.

We say that if $-50 \le error < 50$, the accumulator is *normalised*, and that $-100 \le error < 100$, the accumulator is *acceptable*. For a given full-accuracy value there is one and only one normalised round-pound representation, but two acceptable representations, given by the application of the *floor* and *ceiling* functions respectively.

The rounding technique described here guarantees a normalised root, acceptable representations elsewhere and round-pound correct sums throughout.

4 The Algorithm

The only preconditions on the tree before normalisation are

(i) the root is normalised

(ii) the full-accuracy values sum correctly. (This is guaranteed if all updates are applied to leaf nodes only by a recursive function similar to *UpdateAcc*.)

4.1 Phase 1.

In order to sum across all sub-accumulators of a given accumulator, we introduce 2 additional fields, *firstchild* and *nextsibling*:

```
typedef struct acctype {
    int pound;
    int error;
    int Tag;                        /* +/- to parent */
    struct acctype *parent;
    struct acctype *firstchild;
    struct acctype *nextsibling;
} *AccTreePtr;
```

Starting at the root, the classical round-pound algorithm (see section 1) is applied by depth-first recursion to the vectors summing at all non-leaf nodes. The C function is

```
void classicRP (AccTreePtr t)

{
    int actual, carry, x, rp;
    AccTreePtr p;

    if (t->firstchild != NULL) {
         carry = 0;
         for (p = t->firstchild; p != NULL; p = p->nextsibling) {
             actual = 100*p->pound + p->error;
             x = actual + carry;
             rp = round(x);
             p->pound = rp;
             p->error = actual - rp*100;
             carry = x - rp*100;
         }
         for (p = t->firstchild; p != NULL; p = p->nextsibling) {
             classicRP(p);
         }
    }
}
```

This ensures that all nodes of the tree are acceptable, but does not guarantee correct round-pound sums. However at this stage, it can be shown that if there is a summing error, it must be ±1.00

To prove this, we observe that if there is a summing error then, considered as a component of the accumulator at the level above, the rounded value A is acceptable, i.e. within 1.00 of T, the true value. But when considered as the sum of the vector below, the rounded value N is normalised, i.e. within 0.50 of T. The following diagram shows the 2 cases of how A and N relate to T and that they must be adjacent integers.

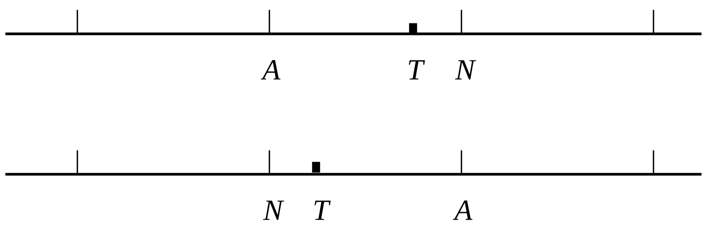

Figure 2. Acceptable and Normalised representations

4.2 Phase 2

Where there is a summing error, redistribution of ±1 in the pound field is always possible in such a way that the summing vector remains acceptable. It is this that prevents the errors from increasing as the algorithm proceeds from root to leaf. The function for this, determines recursively for every internal node of the tree whether there is a summing error. If one is detected, an adjustment is made.

In the example given earlier, the node [C, D] is converted from 4.55 to 4 in order to keep the sum into its accumulator correct. The adjustment to be carried out is therefore to subtract 1.00 from the rounded value of one of the nodes of the vector [C, D], in such a way that it remains acceptable. In this case it is easy to see that the representation of the value at the *subtracting* node D is *increased* from (0, 65) to (1, –35). (The reduction of the magnitude of the *error* in this example is merely incidental.)

To prove that this is *always* possible there are again 2 cases illustrated by figure 2.

Let the vector which sums to T and is displayed as N be v. If any of the contributing accumulators $v[i]$ subtract into T, reverse the sign and regard them as adding.

Case 1: $A = N - 1$.

From the diagram, $N.error < 0$

Therefore $\Sigma\ v[i].error < 0$

Therefore at least one of $v[i].error < 0$. Any contributor with negative *error* would do, but since it involves almost no additional effort, rounding errors are further reduced by adjusting the most negative, say $v[k]$.

$v[k].pound = v[k].pound - 1$

$v[k].error = v[k].error + 100$

If $v[k]$ is a subtracting contributor, its sign is then reversed.

Thus $|v[k].error| \leq 100$ which is the desired condition.

Case 2: $N = A - 1$ is similar.

The functions for detecting summing errors and making the appropriate adjustment, like the function *classicRP*, are recursive.

5 Multi-dimensional Accumulator Structures

In the foregoing discussion it was assumed that the accumulators form a tree; in other words, any value is accumulated into one and only one parent accumulator. In general, spreadsheets do not conform to this restriction. For example, in a cross-tabulation every value in the body of the table accumulates into its row sum *and* into its column sum. If the rounding algorithm above is applied to the tree which represents, say, only column sums, then there is no reason to suppose that these rounded values will satisfy the round-pound arithmetic requirements of the tree representing the row sums. In a cross-tabulation every entry occurs in two tree structures giving the problem a 2-dimensional characteristic. An accounting system in which just two entries are shown with an alternative analyses is also 2-dimensional. If entries have 3 different analyses then the problem is 3-dimensional, and so on.

One solution to the round-pound problem, applicable in any number of dimensions is simply to enumerate all possible roundings. Under the 2 times 0.5 tolerance used in the single dimensional case above, a value can round in exactly 2 ways: up or down to the next integer. Thus if there are n accumulators in an accumulator structure, there are 2^n possible candidates for a solution. A solution is one in which all additions are correct, and the grand totals are to within 0.5.

There appears to be no tractable deterministic algorithmic solution to the multi-dimensional problem. Random trials with two-dimensional cross-tabulations up to 100×100, suggest that the rounding problem might be always solvable in 2 dimensions.

The multi-dimensional case gives rise to accumulator *graphs* comprised of overlapping trees.

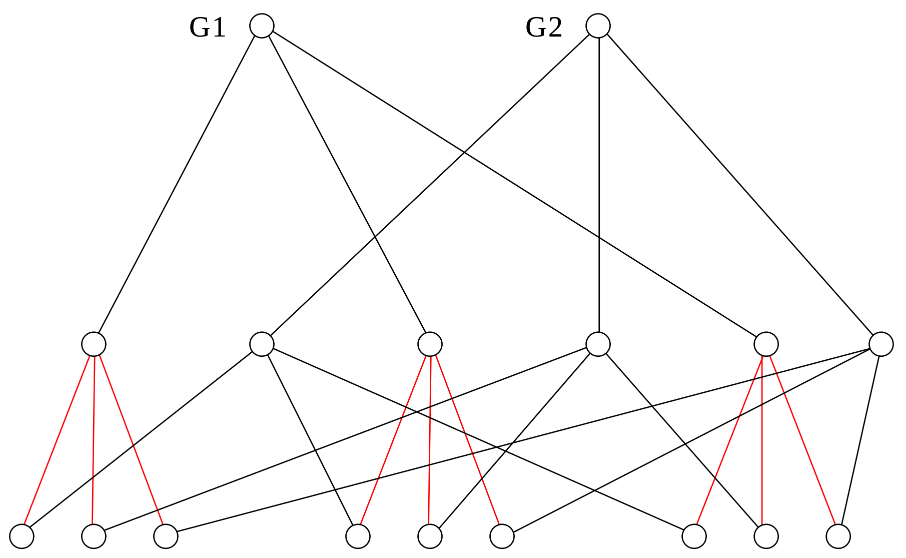

Figure 3. Accumulator graph for a 3×3 cross-tabulation

The example in figure 3 shows the two trees rooted at G1 and G2 which represent the row-column and column-row summations of a 3×3 cross-tabulation. The 9 leaves are common to both trees. The resulting directed graph is of an unusual kind in so far as from each root (i.e. node without predecessor) only a tree structure is accessible. Also there is symmetry between roots and leaves: reverse the sense of the arcs, and leaves become roots and vice versa. Whereas top to bottom processing of this structure uses well-known operations on trees, some operations have to recognise that nodes are generally multi-parented and exist in more than one tree.

A heuristic solution to the multi-dimensional case has been found (Table 1), based on hill-climbing and “taboo-list” search which is now described.

5.1 Preparation

The first stage is to search for all roots in the accumulator structure and apply the single tree round-pound method of section 4 If there are no multi-parent nodes, this is the solution. If the structure is only weakly multi-dimensional, it will bring it close to a solution and in practice one or two iterations of the general algorithm described below will complete the rounding.

5.2 Steepest ascent

The concept of a steepest ascent, in a problem domain which is merely seeking a feasible solution rather than an optimum solution is not strong. However, since the objective is that every sum should be round-pound correct, a steepest ascent adjustment at a node P is defined to be an adjustment which leaves the node round-pound correct with respect to P’s children, and which leaves all *parents* of the adjusted node round-pound correct. In other words, the adjustment results in an increase in the number of round-pound correct parents in the structure. The algorithm performs a breadth-first search of all trees. At each node it considers all the children of P until a child meets this condition.

5.3 Local improvement

A steepest ascent improvement is not always possible. It has been found experimentally that making periodic local round-pound adjustments, regardless of the consequences for the adjusted node’s parents, makes a considerable improvement to the performance of the algorithm. It is equivalent to making an ascent in only one dimension. If this is applied on a 1 in m basis recursively to all trees, a solution is generally found within $2m$ iterations.

5.4 Nudge

Although the solution space appears to be large, the two tactics above can lead to local maxima traps. Using the principle of a taboo-list, the least recently altered child of each node is changed (rounded up if it was down and vice-versa) recursively through all trees. Much experimentation with the frequency of application of these nudges has shown that the interval is best chosen to be co-prime with m and between $2m$ and $4m$.

5.5 Performance

Ideally, the algorithm should be run over a randomly generated structures as well as randomly generated data. However, as the difficulty of maintaining round-pound correctness clearly increases as the number of constraints increases, the hardest problem is that in which every node has the maximum number of parents. In 2 dimensions, this is the familiar cross-tabulation table. Every node has 2 parents. For each of the cross-tabulation sizes in Table 2, random entries for 1000 tables were generated and rounded on a NeXTstation. The number of iterations is the value of the variable *attempt* at the termination of the function *DoRound*. *MAXATTEMPTS* was set at 1000. In the case, all 6 of the cases on which the algorithm failed were easily rounded by inspection. Failure was due to the shortness of the taboo list. A simple enumeration of all 64 possible roundings of the matrix would have yielded a solution in very little time. In a small accounting system with a total of some 300

```
int DoRound()

    /* Purpose:   The strategy for achieving Round-pound cor-
rectness.
           Parameters: none
           Errors:    none
           Side effects:adjustments to the accumulators
           Returns:   1 if correct rounding is achieved; 0 if it is
not.
    */

{
    AccTreePtr root;

    for each root {
         classicRP(root);
         method2(root);              /* local improvement */
    }
    while (!DoCheck() && attempt < MAXATTEMPTS) {
         for each root {
             if (attempt % 17 == 5) {
                  method3(root);    /* nudge the least recently
                                              changed nodes*/
             }
             else {
                  if (attempt % 5 == 4) {
                      method2(root);/* local improvement */
                  }
                  else {
                      method1(root);/* steepest ascent */
                  }
             }
         }
         attempt++;
    }
    return (attempt < MAXATTEMPTS);
}
```

Table 1

nodes in 3 trees of maximum depth 12, the method rarely enters the body of the `while` iteration.

6 Cross-tabulations, the maximal 2-dimensional case.

Whilst the algorithm of the previous section has been tested mainly on cross-tabulation data, it is general enough to be applied to any accumulator graph comprised of overlapping trees. The regular form of a cross-tabulation table, in which every entry has exactly 2 parents (one as its row sum and the other its column sum) suggests there may be more efficient methods to handle this special case. If the rounding convention above is applied to a cross-tabulation table, then the elements of the table, the row sums and the column sums will be allowed an error of up to 1.0 whilst the grand total must be within 0.5.

matrix	average	worst case	av. time	failures
3×3	1.19	41	0.00068	0/1000
4×4	3.57	144	0.00187	0/1000
4×6	3.04	43	0.00244	0/1000
4×20	6.91	70	0.01552	0/1000
20×20	62.92	596	0.39201	0/1000
$2 \times 2 \times 2$	5.85	178	0.00541	6/1000
$3 \times 3 \times 3$	19.82	586	0.04486	0/1000

Table 2 Performance of *DoRound*

6.1 A recursive solution?

A tempting approach to rounding such an array is to look for a method which progresses row by row in a manner analogous to the classical algorithm's progress through vectors. Unfortunately the following example shows that this is not always possible. Consider the 2×1 array

$$\begin{array}{cc|c} 0.3 & 0.3 & 0.6 \\ \hline 0.3 & 0.3 & 0.6 \end{array} \qquad \text{rounds to} \qquad \begin{array}{cc|c} 0 & 1 & 1 \\ \hline 0 & 1 & 1 \end{array}$$

However the rounding of the 2×2 array:

$$\begin{array}{cc|c} 0.3 & 0.3 & 0.6 \\ 0.8 & 0.0 & 0.8 \\ \hline 1.1 & 0.3 & 1.4 \end{array} \qquad\qquad \begin{array}{cc|c} 0 & 1 & 1 \\ x & x & 0 \\ \hline 1 & 0 & 1 \end{array}$$

cannot be completed, given the values already determined for the first row. Had the first row been rounded to [1, 0] rather than [0, 1] then the second row [0.8, 0.0] could be rounded to [0, 0] and all would be well. But in this case, no proper rounding is possible for a second row consisting of [0.0, 0.8].

Since there is no recursive algorithm, there is no proof by induction that all 2 dimensional cross-tabulation tables can be properly rounded. This is indeed unfortunate, for a simple recursive algorithm exists which is remarkably effective and linear in time of execution with the number of elements in the table.

Without loss of generality, consider the cross-tabulation x to be an $m \times n$ matrix of values in the range $0 \le x_{ij} < 1.0$. Rounding of this table will be represented by b, an $m \times n$ matrix of binary values; $b_{ij} = 0$ represents x_{ij} rounded down, 1 represents x_{ij} rounded up.

Assume the upper $m \times l$ matrix of a cross-tabulation table x has already been rounded consistently with respect to row sums r_j and column sums $c_{i(l\text{-}1)}$, the sum of the first l elements of column i.

For all $0 \le i < m, 0 \le j < l$	$\lvert b_{ij} - x_{ij} \rvert < 1.0$	(by definition)
$r_j = \sum (0 \le i < m)\ x_{ij}$	for all $0 \le j < l$	
$c_{i(l\text{-}1)} = \sum (0 \le j < l)\ \ x_{ij}$	for all $0 \le i < m$	
$r_j.pound = \sum (0 \le i < m)\ b_{ij}$	for all $0 \le j < l$	
$c_{i(l\text{-}1)}.pound = \sum (0 \le j < l)\ b_{ij}$	for all $0 \le i < m$	

and that the vector $r[0..l–1]$ has been rounded according to the classical algorithm (section 1)

The algorithm allocates 1's in row l of b as follows.

(i) $r_l = \sum (0 \le i < m)\ x_{il}$

(ii) the vector $r[0..l]$ is rounded according to the classical algorithm. $r_l.pound$ is the number of 1's to be allocated in row l of b.

(iii) Allocate $r_l.pound$ 1's to row b_{*l} in descending order of the value of the deficits defined by

$$c_{il} - \sum (0 \le j < l)\ b_{ij} \qquad \text{for } (0 \le i < m)$$

For example suppose $m = 3$, $l = 2$ (i.e. the upper 3×2 matrix has been rounded) and x is

				r.pound	*r.error*
x_{*0}	0.8	0.2	0.1	1	0.1
x_{*1}	0.5	0.9	0.3	2	–0.3
x_{*2}	0.8	0.1	0.5	1	0.4
c_{*2}	2.1	1.2	0.9	4	0.2

and b is

b_{*0}	1	0	0
b_{*1}	0	1	1

The *deficits* for row 2 are

1.1 0.2 –0.1

and the number of 1's to be allocated to row b_{*2} is $r_2.pound = 1$. Allocating in decreasing order of the deficit makes row b_{*2}

b_{*2}	1	0	0

and the column sums of b are

2 1 1

resulting in column sum errors of

$$0.1 \qquad 0.2 \qquad -0.1$$

all of which are of magnitude less than 1.0.

Table 3 shows the performance of this algorithm on two kinds of randomly generated data in a variety of matrix sizes. It can be seen from these results that, except for the case where $m > n$, the success of the algorithm deteriorates as the density of integers increases There is perhaps reason to suppose that the recursive algorithm may be valid for any cross-tabulations $m \times n$ where $m > n$. This has not been proved.

matrix	av. time	failures (10% integers)	failures (90% integers)
3×3	0.00031	0/1000	0/1000
4×4	0.00058	0/1000	0/1000
4×6	0.00101	0/1000	2/1000
4×20	0.00710	2/1000	5/1000
20×4	0.00435	0/1000	0/1000
20×20	0.04330	0/1000	7/1000

Table 3 Performance of Recursive Algorithm

6.2 A non-recursive approach

This approach proceeds with the same initialisation of row and column sums as in section 6.1. In particular, it applies *classicRP* to row and column sums.

The algorithm *roundmatrix* (Table 4) uses the same auxiliary binary matrix *b* to represent rounding up (1) and down (0), and initially allocates 1's such that the number of 1's in each row is correct, and within each row the 1's are initially left justified. Throughout the algorithm, integer values are not subjected to any change. Row sum integrity is maintained: only column sums are changed.

A function *slidematrix* progresses column by column to the right, attempting to slide ones to the right, leaving column sums correct as it proceeds. In general this satisfies most but not all column sums. Remaining errors in column sums will occur in high-low pairs. An attempt is then made to make adjustments (functions *adjustmatrix* and *adjustcolumntotals*) to high-low pairs within the body of the matrix *b*, whilst maintaining the row sum integrity.

Adjustmatrix restricts its activity to the matrix *b*. It searches for a high-low pair and

(i) searches for a row with 1 in the high and 0 in the low column. If it finds such a row, it transfers the 1.

(ii) Failing that, it searches for a 1 in the high column that was rounded up and a 0 in the low column that was rounded down. If it finds such a pair it reverses the rounding. *Adjustcolumntotals* attempts to revise the column totals which were initialised by *classicRP*. It looks for a high column to which *classicRP* has given a positive error; and a low-column to which it has been given a positive error. If it finds such a pair, it reverses the rounding of the

```
void roundmatrix (int m, int n)
{
          iteration = 1;
          if (!slidematrix(m, n)) {
              iteration++;
              while (adjustmatrix(m, n)) {}
              while (adjustcolumntotals(m, n)) {}

              while (!checkbin(m, n) && iteration < 15) {
                   if (forcechangeup(m, n)) {}
                   iteration++;
                   while (adjustmatrix(m, n)) {}
                   while (adjustcolumntotals(m, n)) {}

                   if (!checkbin(m, n)) {
                       if (forcechangedown(m, n)) {}
                       iteration++;
                       while (adjustmatrix(m, n)) {}
                       while (adjustcolumntotals(m, n)) {}
                   }
              }
          }
}
```

Table 4 A Non-recursive Algorithm

column sums.In about one third of the trials of 20×20 matrices with 90% integer values, and very small proportion of other trials (0.01%), while loops of the measures above did not lead to a solution. A taboo-list approach would be probably be best here, but the function *forcechangeup*, which performs a simple search for any high column with an element to round down in a row with an element to round up, (and its converse *forcechangedown*) provided a sufficient disturbance to clear all but one of the local minima encountered.

Table 5 shows the performance of this algorithm on the same matrices as before.

matrix	average	worst	av. time	failures (10% integers)	failures (90% integers)
3×3	1.13	3	0.00156	0/1000	0/1000
4×4	1.22	3	0.00175	0/1000	0/1000
4×6	1.30	2	0.00106	0/1000	0/1000
4×20	1.78	2	0.00316	0/1000	0/1000
20×4	1.36	3	0.00292	0/1000	0/1000
20×20	1.88	2	0.02075	0/1000	1/1000

Table 5 Performance of Non-Recursive Algorithm

7 Conclusion

In several million trials, no accumulator graph was found which failed to round within the constraints of round-pound arithmetic. A proof that this is true for all graphs, or a contradictory example have yet to be found. However, there is no reason why round-pound arithmetic should not be incorporated into spread-sheets as a user option. The inter-cell dependencies needed for calculation and re-calculation are explicit and effectively define accumulator graphs. Lotus 1-2-3 detects any circular references [2], and warns the user. In this case, the processes described in section 4 do not terminate, the theory breaks down and a spread-sheet would therefore inform the user that round-pound arithmetic and cyclic dependencies may not be combined.

Demonstrative programs in C are available from the author.

References

[1] Brigham and Knechel, *Financial Accounting with Lotus 1-2-3* Harcourt, Brace, Jovanovich (1990)

[2] Lotus Development Corporation *Lotus 1-2-3 Reference Manual, Release 2.2* (1989)