Mean and Variance in One Pass over the Data

A hand-held calculator with limited storage capacity can compute the *Mean* and *Variance* of an arbitrarily long sequence of real numbers $x_1, x_2, x_3, \ldots, x_{n-1}, x_n, \ldots$ without storing all of them. As each datum x_n is entered into the calculator, it can compute the mean m_n and variance v_n of all the data entered so far:

$$m_n := \sum_1^n x_k/n \quad \text{and} \quad v_n := \sum_1^n (x_k - m_n)^2/n \,.$$

(The *Unbiased Sample-Variance* is $n \cdot v_n/(n-1)$.)

To accomplish these computations the calculator requires two schemes:

- Formulas to update m_n to m_{n+1} and v_n to v_{n+1} as soon as x_{n+1} becomes available.
- Tricks to keep roundoff from ruining the formulas' accuracy when v_n stays very small.

These schemes are described below together with tests of their efficacy.

In the absence of roundoff the following recurrences would update m_n and v_n :

$$m_{n+1} := m_n + (x_{n+1} - m_n)/(n+1) \,, \quad \text{starting with } m_1 := x_1 \,.$$

$$v_{n+1} := v_n - \big(v_n - n \cdot (x_{n+1} - m_n)^2/(n+1) \big)/(n+1) \,, \quad \text{starting with } v_1 := 0 \,.$$

These require only one pass over the sequence of data x_k . When n gets big, roundoff degrades these formulas noticeably unless m_n and v_n are computed extra-precisely or else *Compensated Summation* is invoked to attenuate rounding errors that occur at " $m_n + \ldots$ " and " $v_n - \ldots$ ".

Shuffling the order of sequence x_k shouldn't change m_n nor v_n , but roundoff can violate this.

To test the recurrences their results should be compared with the definitions above that require two passes over the data for each n . Those definitions' formulas also require precautions to attenuate roundoff unless the arithmetic's precision extravagantly exceeds the data's.

Below two MATLAB programs `meanvrc1` and `meanvrc2` compare results from the one-pass and two-pass formulas to see whether they disagree significantly. The second program runs correctly *ONLY* on 386-MATLAB 3.5, 68040-Macintosh MATLAB 5.2, and on PC-Windows MATLAB 6.5 after invocation of its command

" `system_dependent('setprecision', 64)` "

because only these versions of MATLAB perform extra-precise accumulation of matrix products necessary to obtain adequate accuracy from the two-pass formulas. A third program `meanvrc3` takes advantage of MATLAB's vectorization to achieve much higher speed though at the cost of slightly diminished accuracy despite efforts to exploit extra-precise scalar products.

MATLAB's arithmetic carries almost 16 sig.dec., exceeding the precision of practically all data extravagantly. Consequently only extravagant test data can reveal how different programs differ in their sensitivities to roundoff. A more practical test compares the one-pass program `meanvrc1` with a simpler version `meanvrc4`, different only in its lack of Compensated Summation, when performed in 4-byte wide `float` arithmetic upon `float` data instead of MATLAB's 8-byte `double`. This test data is more realistic though artificial; its correct values m_n and v_n are known exactly.

```
function  [m, v] = meanvrc1(x)
%  [m, v] = meanvrc1(x) = mean  and  variance  of array  x
%           computed in one pass over the data  x(:) .
%                                  W. Kahan,  19 July 2013
x = x(:) ;  %... assumed real
L = length(x) ;
if (L<1),  m = [] ;  v = [] ;  return, end
m = x(1) ;  v = 0 ;  dm = 0 ;  dv = 0 ;
for  n = 2:L
    xn = x(n) ;  oldm = m ;  oldv = v ;
    d = ( (xn - m) - dm )/n ;
    t = d + dm ;  m = oldm + t ;  %... rounded
    dm = (oldm - m) + t ;  %... compensates summation for  m
    d = ( ( d*d*(n*(n-1)) - v ) - dv )/n ;
    t = d + dv ;  v = oldv + t ;  %... rounded
    dv = (oldv - v) + t ;  %... compensates summation for  v
  end
% . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .


function  [m, v] = meanvrc2(x)
%  [m, v] = meanvrc2(x) = mean  and  variance  of array  x
%           computed in two passes over the data  x(:) .
%                                  W. Kahan,  19 July 2013
x = x(:) ;  %... assumed real
L = length(x) ;
if (L<1),  m = [] ;  v = [] ;  return, end
sn = 0 ;  ds = 0 ;
for  n = 1:L
    t = x(n) + ds ;  s = sn ;  sn = s + t ;
    ds = (s - sn) + t ;  %... compensates summation of  sn
  end
m = sn/L ;  %... almost current mean value
dm = ([sn, m, ds]*[1; -L; 1])/L ;  %... extra-precisely
v = 0 ;  dv = 0 ;
for  k = 1:L  %... second pass
    t = (x(k) - m) - dm ;  t = t*t + dv ;
    s = v ;  v = s + t ;
    dv = (s - v) + t ;  %... compensates summation of  v
  end
v = v/L ;
% . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .


function  [m, v] = meanvrc3(x)
%  [m, v] = meanvrc3(x) = mean  and  variance  of array  x
%           computed in fast vectorized passes over the data  x(:) .
%                                  W. Kahan,  19 July 2013
x = x(:) ;  %... assumed real
L = length(x) ;
if (L<1),  m = [] ;  v = [] ;  return, end
u = ones(L,1) ;
sn = x'*u ;  %... sum rounded to  53  sig.bits
ds = [sn, x']*[-1; u] ;  %... extracts up to  10  more sig.bits
m = sn/L ;  %... almost current mean value
dm = ([sn, m, ds]*[1; -L; 1])/L ;  %... extracts a few more sig.bits
u = [x, u, u]*[1; -m; -dm] ;  %... = x - m - dm  extra-precisely
v = (u'*u)/L ;
% . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

```
function  [m, v] = meanvrc4(x)
%  [m, v] = meanvrc4(x) = mean  and  variance  of array  x  computed
%           in one pass over the data  x(:) without compensated summation.
%                                  W. Kahan,  27 Aug. 2013
x = x(:) ;  %... assumed real
L = length(x) ;  if (L<1),  m = [] ;  v = [] ;  return, end
m = x(1) ;  v = 0 ;
for  n = 2:L
    d = (x(n) - m)/n ;  m = m + d ;  %... rounded
    v = v + ( d*d*(n-1) - v/n ) ;  %... rounded
  end
% . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

Results from several example sequences x_k are exhibited hereunder. The second is the array

`x = 100000000 + randn(1, 16384)`

consisting of 16384 entries each differing from 100000000 by a *Normally* distributed pseudo-random variate with mean 0 and variance 1 . The first three programs above agree to 15 sig. dec.; any difference from the fourth is due to its uncompensated roundoff. That these programs agree so closely corroborates that they are intended to compute the same mean m and variance v , though using four different methods affected differently by roundoff.

```
» Results from  386-PC-Matlab 3.5  on an old  80386-based  Intel 302

» x = randn(1, 32768) ;  %...  32768 “normal” random no’s,  m = 0 ,  v = 1
» [m1,v1] = meanvrc1(x) ;  [m2,v2] = meanvrc2(x) ;
» [m3,v3] = meanvrc3(x) ;  [m4,v4] = meanvrc4(x) ;
» res1 = [m1,v1; m2,v2; m3,v3; m4,v4]

res1 =      m                         v
    -1.422220178371109e-3     1.012960974376284   %...  from  meanvrc1
    -1.422220178371108e-3     1.012960974376284   %...  from  meanvrc2
    -1.422220178371108e-3     1.012960974376284   %...  from  meanvrc3
    -1.422220178371098e-3     1.012960974376278   %...  from  meanvrc4

» x = 100000000 + randn(1, 16384) ;  %...  m = 100000000 ,  v = 1
» [m1,v1] = meanvrc1(x) ;  [m2,v2] = meanvrc2(x) ;
» [m3,v3] = meanvrc3(x) ;  [m4,v4] = meanvrc4(x) ;
» res2 = [m1,v1; m2,v2; m3,v3; m4,v4]

res2 =       m                    v
     100000000.0049386    0.9951514108806957   %...  from  meanvrc1
     100000000.0049386    0.9951514108806957   %...  from  meanvrc2
     100000000.0049386    0.9951514108806957   %...  from  meanvrc3
     100000000.0049381    0.9951514086945294   %...  from  meanvrc4
```

As expected, `meanvrc4`’s variance v is most vulnerable to roundoff when $\sqrt{v}/m$ is small.

The third example is the non-random array

`z = 2^52 - 12345678 + [1:30000]`

whose 30000 elements have easily calculated mean `m = 4503599615039818.5` and variance `v = 7499999.916666…` . But `meanvrc3` will lose half the sig.dec. of v because its mean m is too small by almost 0.5 , too little to show up in binary-decimal conversion to only 16 sig.dec.

```
» x0 = 2^52 - 12345678 ;  %... =  4503599615024818
» n = 30000 ;  x = x0 + [1:n] ;  %...  Length(x) = n
» m = x0 + (n+1)/2 ;  v = (n-1)*(n+1)/12 ;  %...  if computed exactly
»  ...  etc.  ...
resA =      m                   v
     4503599615039818.     74999999.91666667   %...   m & v  correctly rounded
     4503599615039818.     74999999.91666667   %...  from  meanvrc1
     4503599615039818.     74999999.91666667   %...  from  meanvrc2
     4503599615039818.     75000000.10300279   %...  from  meanvrc3
     4503599615039818.     74999999.91666667   %...  from  meanvrc4

» x = reshape([x(1:n/2); x(1+n/2:n)], n,1) ;  %... shuffle preserves  m & v ?
»  ...  etc.  ...
resB =      m                   v
     4503599615039818.     74999999.91666667   %...   m & v  correctly rounded
     4503599615039818.     74999999.91666667   %...  from  meanvrc1
     4503599615039818.     74999999.91666667   %...  from  meanvrc2
     4503599615039818.     74999999.91666673   %...  from  meanvrc3
     4503599615039819.     74996802.16918673   %...  from  meanvrc4
```

Apparently meanvrc4's simple uncompensated summation leaves v too vulnerable to roundoff.

Adding a constant to x should add that constant to m but leave v unaltered except by roundoff.

```
» x0 = 4650607080901020 ;
n = 30000 ;  x = x0 + [1:n] ;  %...  Length(x) = n
m = x0 + (n+1)/2 ;  v = (n-1)*(n+1)/12 ;
»  ...  etc.  ...
res3 =       m                  v
     4650607080916020.     74999999.91666667   %...   m & v  correctly rounded
     4650607080916020.     74999999.91666667   %...  from  meanvrc1
     4650607080916020.     74999999.91666667   %...  from  meanvrc2
     4650607080916020.     75000000.10470454   %...  from  meanvrc3
     4650607080916014.     75069536.20090015   %...  from  meanvrc4

» x = reshape([x(1:n/2); x(1+n/2:n)], n,1) ;  %... shuffle preserves  m & v ?
»  ...  etc.  ...
res4 =       m                  v
     4650607080916020.     74999999.91666667   %...   m & v  correctly rounded
     4650607080916020.     74999999.91666667   %...  from  meanvrc1
     4650607080916020.     74999999.91666667   %...  from  meanvrc2
     4650607080916020.     74999999.91666691   %...  from  meanvrc3
     4650607080916016.     68971878.98776774   %...  from  meanvrc4
```

Again an error in vectorized meanvrc3's m was too tiny for binary-decimal conversion to show, but it affected v. And meanvrc4's simple uncompensated summation left v utterly wrong.

So far, the only consistently exemplary test results have come from one-pass program meanvrc1 and two-pass meanvrc2, both with compensated summation, and the latter with a tricky extra-precise computation of dm. How well do these programs compute the variance v of an array x whose elements $x_k = x_0 \pm 1$ fluctuate only in their last two bits?

```
» x0 = 4650607080901020 ;
» n = 30001 ;  x = x0 + (-1).^[1:n] ;  %...  Length(x) = n
» m = x0 - 1/n ;  v = 1 - 1/n^2 ;
»  ...  etc.  ...
```

```
res5 =        m                     v
      4650607080901020.     0.9999999988889630   %...   m & v  correctly rounded
      4650607080901020.     0.9999999988889630   %...  from  meanvrc1
      4650607080901020.     1.000000016665556    %...  from  meanvrc2
      4650607080901020.     1.000000016665556    %...  from  meanvrc3
      4650607080901020.     0.9996871364846397   %...  from  meanvrc4
```

Only one-pass meanvrc1 has passed all tests unblemished. Perhaps they are too stringent. All four programs will probably serve satisfactorily when applied to more realistic data, conveyed in 4-byte float variables deserving float results, provided the programs continue to perform all arithmetic in 8-byte double.

If converted to perform only float arithmetic, only meanvrc1 will work fully reliably.

The next three tests pit meanvrc1 with compensated summation against meanvrc4 without. Except for the correctly rounded m and v, all data, variables and arithmetic are in float.

```
» x0 = 8470605 ;  n = 30000 ;
» x = x0 + [1:n] ;  %...  Length(x) = n
» m = x0 + (n+1)/2 ;  v = (n-1)*(n+1)/12 ;
» precision = precn(24);  [m1,v1] = meanvrc1(x) ;  [m4,v4] = meanvrc4(x) ;
» precision = precn(64);  res6 = [m,v; m1,v1; m4,v4]

res5 =    m              v
     8485605.5      74999999.91666667   %...   m  and  v  correctly rounded
     8485606.0      75000000.          %...  from  meanvrc1
     8485606.0      74999368.          %...  from  meanvrc4

» x = reshape([x(1:n/2); x(1+n/2:n)], n,1) ;  %... shuffle preserves  m & v ?
» ...   etc.   ...
res6 =    m              v
     8485605.5      74999999.91666667   %...   m  and  v  correctly rounded
     8485606.0      75000000.          %...  from  meanvrc1
     8485606.0      68956288.          %...  from  meanvrc4


» x0 = 8470605 ;  n = 30001 ;
» x = x0 + (-1).^[1:n] ;  %...  Length(x) = n
» m = x0 - 1/n ;  v = 1 - 1/n^2 ;
» ...   etc.   ...
res7 =    m                      v
     8470604.999966668     0.999999998888963   %...   m & v  correctly rounded
     8470605.0             1.00000000         %...  from  meanvrc1
     8470605.0             0.999477327        %...  from  meanvrc4
```

Conclusion: To program even a simple computation satisfactorily for *all* instead of merely *most* mathematically unexceptionable data may require more knowledge about error-analysis than can be learned from one course in Numerical Analysis. An almost always adequate alternative is to perform all arithmetic and carry all intermediate variables in extravagantly more precision, more than twice as much, as is trusted in the initial data and desired in the final result. Doing so may run too slow. If you need both speed and accuracy, you may have to learn more about Numerical Analysis and its rounding errors than you had intended. See my web page's WrongR.pdf too.