Windowed Incremental Online Statistics

Extracting Models from Data, One Observation at a Time

Brian Beckman

29 March 2019

We calculate descriptive statistics—mean, unbiased variance, windowed versions of the same—of sequences of numerical data. *Fold* and its variants decouple iteration and accumulation from data access. Exactly the same stateless accumulator functions work over data distributed over space in memory or in tables, or over time in asynchronous streams. Foldable accumulator functions can be tested independently of data source, say over ground-truth test data. The same functions can be deployed in harsh asynchronous environments without even being recompiled.

Prelude: Running Count

For ground truth, we want a fixed but random collection of data. The following is a sample of length ten from the standard normal distribution, mean zero, standard deviation one:

In[1]:=

```
zs = {-0.178654, 0.828305, 0.0592247, -0.0121089,
   -1.48014, -0.315044, -0.324796, -0.676357, 0.16301, -0.858164};
```

We begin with focusing attention on *Fold* rather than on any particular accumulator function. **Fold** has three arguments: an accumulator function, an initial state, and an abstract sequence of data. All variations of **Fold**—over asynchronous streams, on-demand (lazy) streams, tables, lists, arrays—have exactly the same signature.

In[2]:=

```
ClearAll[cume];
cume[n_, z_] := n + 1;
Fold[cume, 0, zs]
```

Out[4]=

```
10
```

The first argument of the accumulator function, `cume`, is a *circulating state*, kept externally to the function. The caller must feed in the *prior* value n of the state and a new observation z. The caller receives the *posterior* value of the state, $n + 1$.

To see all intermediate results, use **FoldList**:

In[5]:=

```
FoldList[cume, 0, zs]
```

Out[5]=

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

LEFT FOLD

Wolfram’s documentation for **Fold** includes an example that explains this operator symbolically:

In[6]:=

```
Fold[f, z, {a, b, c, d}]
```

Out[6]=

```
f[f[f[f[z, a], b], c], d]
```

Fold first computes $f[z, a]$, then feeds the result—the posterior value of the circulating state—back into f, computing $f[f[z, a], b]$, and so on. More concisely, **Fold** iteratively applies the binary function f to the sequence $\{a, b, c, d\}$. **Fold** places no restriction on its accumulator-function argument f other than it accept an argument of the type T_z of z in its first position, an argument of the type T_a of any of a, b, c, d, in its second position, and produces a result of type T_a.

RIGHT FOLD

We use only the left fold in this document. Contrast it to the right fold, presented for conceptual completeness (the following is an expression of right fold in terms of left fold):

In[ ]:=

```
Fold[{x, y} ↦ f[y, x], z, {a, b, c, d}]
```

Out[7]=

```
f[d, f[c, f[b, f[a, z]]]]
```

Operators of the same signature as **Fold** are broadly useful for separating computation from data access. Wolfram’s **Fold** works only on lists in memory, but similar fold operators with exactly the same signatures are easy to write over asynchronous and lazy streams.

Running Mean

Write x for the mean-so-far, and write a new accumulator function that computes the running mean. Its circulating state includes the running mean x, the running count n, and the running sum. The new accumulator function computes the new values of all three state variables in the obvious way. Note that all three state variables are necessary to accomplish the task of tracking the running mean. We relieve that situation shortly below.

In[12]:=

```
ClearAll[cume];
cume[{x_, n_, sum_}, z_] :=
  {(sum + z)/(n + 1), n + 1, sum + z};
Fold[cume, {0, 0, 0}, zs]
```

Out[14]=

```
{-0.279472, 10, -2.79472}
```

Check against *Mathematica* built-in.

In[15]:=

```
Mean[zs]
```

Out[15]=

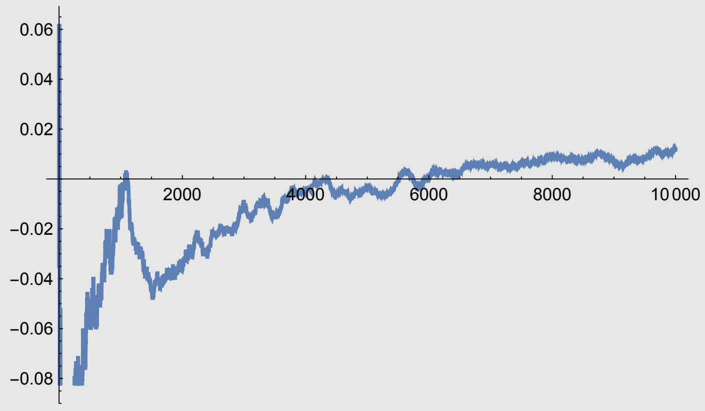
```
-0.279472
```

Notice how the mean improves and stabilizes as data accumulate, here over a longer and random sample:

In[18]:=

```
ListLinePlot[
 #⟦1⟧ & /@ FoldList[cume, {0, 0, 0}, RandomVariate[NormalDistribution[], 10 000]]]
```

Out[18]=



Here is a preferable form with one fewer quantity to track in the circulating state: compute the new value as sum of the old value and a *correction* that depends only on old values and the single observation z. The right-hand side of the *recurrence* $z \quad \overline{\ } \leftarrow \overline{z} + K \times (z - \overline{z})$, $\overline{z}$ here standing for the running mean, depends only on old values and the new observation z. The recurrence is not an equation, but it's shorthand *for* the following equation, just without the subscripts:

$$\overline{z}_{n+1} = \overline{z}_n + K \times (z_{n+1} - \overline{z}_n) = x + K \times (z_{n+1} - x). \quad (1)$$

In[∘]:=

```
ClearAll[cume];
cume[{x_, n_}, z_] :=
  With[{K = 1/(n + 1)},
   {x + K (z - x), n + 1}];
Fold[cume, {0, 0}, zs]
```

Out[∘]=

```
{-0.279472, 10}
```

We prefer this form because (1) it separates old quantities on the right (except for the new datum, z) from new quantities on the left, and (2) it is easy to memorize because it is an *affine* update:

the old estimate plus a *correction* written as a *gain* K times the *residual* $z - \overline{z}$: the difference between the new observation and the old mean-so-far.

To prove this formula $\overline{z} + K \times (z - \overline{z})$ correct, we need only show that it equals $\frac{\text{sum} + z}{n+1}$, which is obviously the new mean. Write $\overline{z}$ as $\frac{\overline{z}(n+1)}{n+1}$ and write $\overline{z} + K \times (z - \overline{z}) = \frac{n\,\overline{z} + \overline{z}}{n+1} + \frac{z - \overline{z}}{n+1} = \frac{\text{sum} + z}{n+1}$, noting that the running sum is $n\,\overline{z}$. I think this is beautiful.

Equation 1 is formally identical to the Kalman update equation. This document builds up to the Kalman filter through more elementary calculations that have similar forms.

The following numerical check proves that we have a correct formula.

In[•]:=

```
ClearAll[cume];
cume[{x_, n_}, z_] :=
  With[{K = 1/(n + 1)},
    {K (z + n x), n + 1}];
Fold[cume, {0, 0}, zs]
```

Out[•]=

```
{-0.279472, 10}
```

For another twist on the proof above, the Kalman-like form $x + K(z - x)$ is equal to the obvious form $(n\,x + z)/(n+1)$ when $n + 1 \neq 0$.

In[•]:=

```
Solve[x + K (z - x) == (n x + z)/(n + 1), K]
```

Out[•]=

```
{{K → 1/(1 + n)}}
```

Running Variance

Definitions

Variance is the sum of squared residuals against the running mean, divided by the count less one. The “count less one” is Bessel's correction.

In[•]:=

```
Variance[zs]
```

Out[•]=

```
0.395183
```

Variance is also standard deviation squared.

In[∘]:=
```
StandardDeviation[zs]^2 == Variance[zs]
```
Out[∘]=
```
True
```

The sum of squared residuals,

$$\Sigma_n \overset{\text{def}}{=} \sum_{i=1}^{n} (z_i - \overline{z}_n)^2 \qquad (2)$$

is the variance times the length-less-one:

In[∘]:=
```
Variance[zs] * (Length[zs] - 1)
```
Out[∘]=
```
3.55665
```

In[∘]:=
```
With[{μ = Mean[zs]},
 Plus @@ ((# - μ)^2 & /@ zs)]
```
Out[∘]=
```
3.55665
```

The *covariance*, for scalar data, is the same as the Variance. For vector data, the covariance is a matrix.

In[∘]:=
```
Covariance[zs] * (Length[zs] - 1)
```
Out[∘]=
```
3.55665
```

Requirement

We require a running variance that does not refer to future observations (even indirectly, through the eventual mean $z\overline{}_n$) and does not store the entire past, only a summary of constant size.

Given the definition, which explicitly refers to the past history, it seems, at first glance, impossible to compute the variance without keeping the entire history. But it is, and we'll get there in baby steps.

Brute-Force Variance (doesn't meet requirements)

The following accumulates variance directly from the definition, useful for numerical checks. However, it assumes we know length and mean of the entire sequence `zs` ahead of time, so it does not meet the requirement of being incremental. It also needs computer memory for the entire sequence `zs`, which is of variable or unknown length, so does not meet the requirement of constant memory.

Re-use the variable `zs`, now for a shorter sequence of ground-truth data.

In[19]:=
```
ClearAll[zs]; zs = {55, 89, 144};
```

In[20]:=
```
Variance[zs] // N
```
Out[20]=
```
2017.
```

Hand-calculated mean:

In[21]:=
```
(55 + 89 + 144) / 3.
```
Out[21]=
```
96.
```

Hand-calculated variance = SSR (sum of squared residuals) / $(n-1)$:

In[22]:=
```
((55 - 96.) ^2 + (89 - 96.) ^2 + (144 - 96.) ^2) / 2.
```
Out[22]=
```
2017.
```

Now, we'll write a truly nasty accumulator function to produce the final variance. It's nasty because it refers to the entire sequence through the variable `zs`.

In[ ]:=
```
ClearAll[cume];
cume[var_, z_] := var + (z - Mean[zs])^2 / (Length[zs] - 1);
FoldList[cume, 0., zs]
```
Out[ ]=
```
{0., 840.5, 865., 2017.}
```

It's even nastier because the intermediate variances are not valid. They do not refer to the "mean-so-far"—the incremental running mean, but to the final mean.

School Variance (meets requirements)

The following is much better, exploiting the "school formula," proved by expanding the square. Letting $\bar{z}_n = x$, incremental running mean, the following yields the sum of squared residuals, `ssq`:

$$\sum_{i=1}^{n}(z_i-\bar{z}_n)^2=\left(\sum_{i=1}^{n}z_i^2\right)-n\,\bar{z}_n^2 \qquad (3)$$

This is very nice because we can maintain $\sum_{i=1}^{n} z_i^{\,2}$ incrementally, just as we maintain the running sum, $\sum_{i=1}^{n} z_i$, one observation at a time.

The Variance is `ssq` divided by $n-1$ when $n>1$. When $n=1$, variance must be zero because there is no dispersion when there is only one datum. Add the variance and `ssq` to the circulating state, and write:

In[∘]:=

```
ClearAll[cume];
cume[{var_, ssq_, x_, n_}, z_] :=
  With[{n2 = n + 1},
   With[{K = 1/n2},
    With[{x2 = x + K (z - x), ssq2 = ssq + z^2},
     {(ssq2 - n2 x2^2)/Max[1, n], ssq2, x2, n2}]]];
FoldList[cume, {0, 0, 0, 0}, zs] // MatrixForm
```

Out[∘]//MatrixForm=

```
(  0      0    0  0 )
(  0    3025   55  1 )
( 578  10946   72  2 )
( 2017 31682   96  3 )
```

The variance computed this way is incremental and runs in constant memory, though it is not a recurrence, i.e., old value plus a correction.

This school variance tracks three auxiliary quantities: the sum of squares `ssq`, the running mean *x*, and the count *n*. Note, for the future, that `ssq` grows quickly, inviting numerical issues. We'll mitigate that shortly.

Recurrent Variance (meets requirements)

Though it is incremental, the school variance isn't a recurrence. A recurrence, in our preferred form, expresses the new variance as the old variance plus a correction, where the correction depends only on old values and the new datum *z*. Start with a recurrence for the sum of squared residuals, **ssr**, $\Sigma_n \overset{\text{def}}{=} \sum_{i=1}^{n}(z_i-\bar{z}_n)^2$:

$$\Sigma\leftarrow\Sigma+\frac{n}{n+1}(z-\bar{z})^2 \qquad (4)$$

The form in equation 4 is an abbreviation for equation 5. The abbreviation is precise under the rule that all accumulated quantities, namely $\bar{z}$ and Σ, on the right of the recurrence arrow, are old—have implicit subscript *n*; only the datum *z* is new—with subscript $n+1$, on the right of the recurrence arrow.

$$\Sigma_{n+1} = \Sigma_n + \frac{n}{n+1}\left(z_{n+1} - \overline{z}_n\right)^2 \tag{5}$$

Proof

Here’s the old ssq from the School formula:

$$S_n = \sum_{i=1}^{n} {z_i}^2 - n\,\overline{z}_n^2 \tag{6}$$

Here’s the new ssq via the same formula:

$$S_{n+1} = \sum_{i=1}^{n+1} {z_i}^2 - (n+1)\,\overline{z}_{n+1}^2 \tag{7}$$

Let $sz\,{}_n^2$ = `sz2[n]` be $\sum_{i=1}^{n} {z_i}^2$. Note that $sz\,{}_{n+1}^2 = sz_n^2 + z^2$ and that $\overline{z}_{n+1} = \frac{1}{n+1}\left(n\,\overline{z}_n + z\right)$. Those obvious equalities are enough to effect the proof.

Transcribe Equation 6.

```
In[◦]:=
ClearAll[s, zb, zbar];
s[n_] := sz2[n] - n zbar[n]^2
```

Transcribe the “obvious equalities” above.

```
In[◦]:=
rulez$ = {sz2[1 + n] - sz2[n] → z^2, zbar[1 + n] → n/(n + 1) zbar[n] + z/(n + 1)};
```

Note that the rules don’t do much to the definition of `s[n]`.

```
In[◦]:=
s[n] //. rulez$

Out[◦]=
sz2[n] - n zbar[n]^2
```

The rules instantly yield the correction terms in Recurrence 4 and Equation 5.

```
In[◦]:=
s[n + 1] - s[n] //. rulez$ // FullSimplify

Out[◦]=
n (z - zbar[n])^2
─────────────────
      1 + n
```

By hand, here are **`rulez$`**:

$$sz_{n+1}^2 = sz_n^2 + z^2$$

$$(n+1)\,\overline{z}_{n+1} = n\,\overline{z}_n + z$$

Expand out the definition of S_{n+1}:

$$S_{n+1} = sz^2_{n+1} - (n+1)\,\bar{z}^2_{n+1} = [sz^2_n + z^2] - \frac{1}{n+1}[(n+1)\,\bar{z}_{n+1}]^2 \quad = \quad [sz^2_n + z^2] - \frac{1}{n+1}[n\,\bar{z}_n + z]^2$$

Expand the second term:

$$S_{n+1} = sz^2_n + z^2 - \frac{n^2}{n+1}\bar{z}^2_n - \frac{2n}{n+1} z\,\bar{z}_n - \frac{1}{n+1} z^2$$

Add and subtract a term to isolate the prior value of S_n and prepare other terms for cancelation:

$$= [sz^2_n - n\,\bar{z}^2_n] + n\frac{n+1}{n+1}\bar{z}^2_n + \frac{n+1}{n+1} z^2 - \frac{n^2}{n+1}\bar{z}^2_n - \frac{2n}{n+1} z\,\bar{z}_n - \frac{1}{n+1} z^2$$

Cancel terms and complete the square:

$$= [sz^2_n - n\,\bar{z}^2_n] + \frac{n}{n+1}\bar{z}^2_n - \frac{2n}{n+1} z\,\bar{z}_n + \frac{n}{n+1} z^2$$

$$S_{n+1} = S_n + \frac{n}{n+1}(z - \bar{z}_n)^2$$

Equivalence to Welford’s

Welford ’s famous recurrence is

$$\Sigma \leftarrow \Sigma + (z - \bar{z}_n)(z - \bar{z}_{n+1}) \tag{8}$$

where $\bar{z}_n$ is the last estimate of the mean and $\bar{z}_{n+1}$ is the new or current estimate. The correction term on the right of Recurrence 8 is exactly equal to the correction term on the right of recurrence 4.

In[ ]:=

```
(z - zbar[n]) (z - zbar[n + 1] //. rulez$ // FullSimplify)
```

Out[ ]=

```
n (z - zbar[n])^2
-----------------
      1 + n
```

We won’t write out the algebraic manipulations, but it’s good practice.

Welford 's is easier to memorize than is Recurrence 4, but its right-hand side depends on both the old mean $\bar{z}_n$ and the new mean $\bar{z}_{n+1}$, whereas we always want right-hand sides that depend only on old values (except for new datum z) because such right-hand sides require fewer intermediate variables. Notice the following, using Welford’s, has three levels of With because ssr2 depends on two means and we must precompute the new mean x2:

```
ClearAll[cume];
cume[{var_, x_, n_}, z_] :=
  With[{K = 1/(n + 1)},
   With[{x2 = x + K (z - x)},
    With[{ssr2 = (n - 1) var + (z - x) (z - x2)},
     {ssr2/Max[1, n], x2, n + 1}]]];
FoldList[cume, {0, 0, 0}, zs] // MatrixForm
```

```
(  0    0  0 )
(  0   55  1 )
( 578  72  2 )
( 2017 96  3 )
```

Folding It

Recurrence 4 lets us get rid of one level of `With`, one level of dependency:

```
ClearAll[cume];
cume[{var_, x_, n_}, z_] :=
  With[{K = 1/(n + 1)},
   With[{x2 = x + K (z - x),
     ssr2 = (n - 1) var + K n (z - x)^2},
    {ssr2/Max[1, n], x2, n + 1}]];
FoldList[cume, {0, 0, 0}, zs] // MatrixForm
```

```
(  0    0  0 )
(  0   55  1 )
( 578  72  2 )
( 2017 96  3 )
```

▫ To LaTeX:

In[ ]:=

```
ClearAll[cume];
cume[{var_, x_, n_}, z_] :=
  With[{K = 1/(n + 1)},
   With[{x2 = x + K (z - x),
     ssr2 = (n - 1) var + K n (z - x)^2},
    {ssr2/Max[1, n], x2, n + 1}]];
FoldList[cume, {0, 0, 0}, zs] // TeXForm
```

Out[ ]//TeXForm=

```
\left(
\begin{array}{ccc}
 0 & 0 & 0 \\
 0 & 55 & 1 \\
 578 & 72 & 2 \\
 2017 & 96 & 3 \\
\end{array}
\right)
```

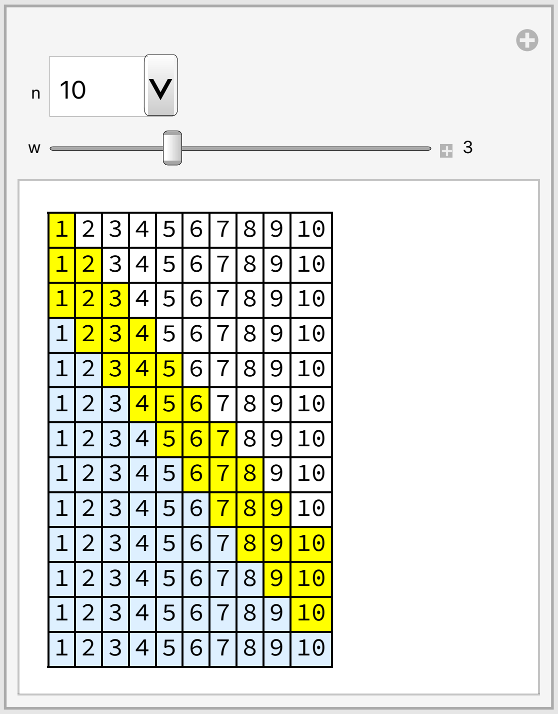
Windowed Statistics

Sometimes, we want the mean and variance of a limited amount of the past, say over a window of length *w*.

In[ ]:=

```
Manipulate[
 Grid[Table[With[{l = i - w + 1}, ⎡ Item[j, Background → LightBlue]  j < l
                                   ⎨ Item[j, Background → Yellow]     j ≤ i],
                                   ⎣ j                                True
    {i, n + w}, {j, n}], Frame → All],
 {{n, 10}, 5 + Range[100]},
 {{w, 3}, Range[n + 1] - 1, Appearance → "Labeled"}]
```

Out[ ]=



n 10

w 3

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

In[23]:=

```
ClearAll[buttonize];
buttonize[fnRandom_, fnResultFromRandom_, lbl_ : "RANDOMIZE!"] :=
  DynamicModule[{val = fnRandom[]},
Manipulate[fnResultFromRandom[val],
    Button[lbl, val = fnRandom[]]]];
```

In[25]:=

```
buttonize[RandomReal, Identity]
```

Out[25]=

```
RANDOMIZE !

0.263041
```

Data from the C program

```
On[Assert];
```

```
ClearAll[runningStatsFoldable];
Module[{u, j, mnp1, mjp1, mw, vnp1, vjp1,
    vw, cbuf, zj, checkMw, checkVw, checkMjp1, checkVjp1},
  runningStatsFoldable[w_][

    {_, n_, _, _, _, mn_, mj_, _, _, _, vn_, vj_, _, _, _}, z_] := (

    cbuf = ConstantArray[0, w];

    (* there are five recurrent inputs and two actionable outputs *)
    (*     in the circulating-state vector. The rest of the items *)
    (*     are auxilaries for verifying and debugging. *)

    (* window starts overlapping data by 1 cell, then by 2 cells *)
    (* eventually window becomes flush left, then advances into *)
    (* the data, leaving evicted data to its left *)
    (* datum at the right of the window -- datum to be evicted *)
    zj = cbuf⟦w⟧;
    (* rotate zj to the left of the window *)
    cbuf = RotateRight[cbuf, 1];
    (* replace zj with z *)
    cbuf⟦1⟧ = z;
    (* window starts containing one datum when n is 0 *)
    (* eventually, when n is w-1, window is flush left *)
    (* j = elements to left of window, 0 when flush left *)
    (*     then > 0 thereafter if w < N *)
    j = Max[0, n - w + 1];
    (* u = number of data in the window *)
    (* starts at 1, becomes 2, ..., w *)
    (* becomes w when j is flush left and stays w *)
    u = n - j + 1; Assert[u ≤ w];
    (* mn = prior mean of all data before z -- recurrent input *)
    (* mnp1 = posterior mean of all data including z *)
    (* this is our normal gain times residual calc *)
    mnp1 = mn + (z - mn)/(n + 1);
    (* mj = prior mean of data left of window -- recurrent input*)
    (* mjp1 = posterior mean of data left of the window *)
    mjp1 = If[j > 0, mj + (zj - mj)/j, 0];
    (* mw = posterior mean of data in window -- output *)
    mw = ((n + 1) mnp1 - j mjp1)/u;
    (* vn = prior variance of all data -- recurrent input *)
    (* posterior variance of all data via Recurrence 4 *)
    vnp1 = ((n - 1) vn + n/(n+1) (z - mn)^2)/Max[1, n];
    (* vj = prior variance of data left of window -- recurrent input *)
    (* vjp1 = posterior variance of data to left of window *)
    (* increase by scaled residual of the evicted datum vj *)
    (* will be zero until window moves off of flush left *)
    (* so it is safe to multiply it later by a negative n-w *)
    vjp1 = If[j > 1, (j - 2)/(j - 1) vj + 1/j (zj - mj)^2, 0];
    (* vw = posterior variance of data in window -- output *)
    (* three applications of the school formula *)
    (* n_Bessel v_n+1 = Σ_i=1^n+1 (z_i)^2-(n+1) z̄_n+1 ; ditto for j *)
    (* The first two terms in the numerator combine to equal *)
    (*   the sum of squared data (not residuals) in the window *)
    (* the third term in the numerator is the school correction *)
    (* for the window TODO make it Welford's or Recurrence 4 *);
    vw = ((n vnp1 + (n + 1) mnp1^2) - ((n - w) vjp1 + j mjp1^2) - u mw^2)/Max[1, u - 1];
    checkMjp1 = If[j > 0, Mean[data⟦1 ;; j⟧], 0];
    Assert[0 === Chop[checkMjp1 - mjp1]];
    checkMw = Mean[data⟦j + 1 ;; n + 1⟧];
    Assert[0 === Chop[checkMw - mw]];
    checkVjp1 = If[j > 1, Variance[data⟦1 ;; j⟧], 0];
    Assert[0 === Chop[checkVjp1 - vjp1]];
    checkVw = If[u > 1, Variance[data⟦j + 1 ;; n + 1⟧], 0];
    Assert[0 === Chop[checkVw - vw]];
```

```
    (* RETURN VALUE *)
    {j,
     n + 1,
     u,
     z,
     zj,
     mnp1,
     mjp1,
     checkMjp1,
     mw,
     checkMw,
     vnp1,
     vjp1,
     checkVjp1,
     vw,
     checkVw
    })];

With[{data = {0.857454, 0.312454, 0.705325, 0.839363,
    1.63781, 0.699257, -0.340016, -0.213596, -0.0418609, 0.054705}},
 Grid[
  Prepend[FoldList[runningStatsFoldable[3],
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, data],
   {"j", "n+1", "u", "z", "zj", "μ'", "μj'", "chk μj'", "μw'",
    "chk μw'", "υ'", "υj'", "chk υj'", "υw'", "chk υw'"}], Frame → All]]
```

```
Part : Cannot take positions 1 through 1 in data.

Assert : Assertion 0 === Chop[checkMw$55504 – mw$55504] failed.

Part : Cannot take positions 1 through 2 in data.

Assert : Assertion 0 === Chop[checkMw$55504 – mw$55504] failed.

Part : Cannot take positions 1 through 2 in data.

General : Further output of Part::take will be suppressed during this calculation.

Assert : Assertion 0 === Chop[checkVw$55504 – vw$55504] failed.

General : Further output of Assert::asrtf will be suppressed during this calculation.
```

Out[77]=

j	n+1	u	z	z_j	μ'	μ'_j	chk μ'_j	μ'_w	chk μ'_w	υ'	υ'_j	chk υ'_j	υ'_w	chk υ'_w
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0	1	1	0.857454	0	0.857454	0	0	0.857454	Mean[data⟦1;;1⟧]	0.	0	0	0.	0
0	2	2	0.312454	0	0.584954	0	0	0.584954	Mean[data⟦1;;2⟧]	0.148513	0	0	0.148513	Variance[data⟦1;;2⟧]
0	3	3	0.705325	0	0.625078	0	0	0.625078	Mean[data⟦1;;3⟧]	0.079086	0	0	0.079086	Variance[data⟦1;;3⟧]
1	4	3	0.839363	0	0.678649	0	Mean[data⟦1;;1⟧]	0.904865	Mean[data⟦2;;4⟧]	0.0642035	0	0	-0.210738	Variance[data⟦2;;4⟧]
2	5	3	1.63781	0	0.870481	0	Mean[data⟦1;;2⟧]	1.4508	Mean[data⟦3;;5⟧]	0.232151	0	Variance[data⟦1;;2⟧]	-0.798595	Variance[data⟦3;;5⟧]
3	6	3	0.699257	0	0.841944	0	Mean[data⟦1;;3⟧]	1.68389	Mean[data⟦4;;6⟧]	0.190607	0	Variance[data⟦1;;3⟧]	-1.65009	Variance[data⟦4;;6⟧]
4	7	3	-0.340016	0	0.673092	0	Mean[data⟦1;;4⟧]	1.57055	Mean[data⟦5;;7⟧]	0.358415	0	Variance[data⟦1;;4⟧]	-1.03901	Variance[data⟦5;;7⟧]
5	8	3	-0.213596	0	0.562256	0	Mean[data⟦1;;5⟧]	1.49935	Mean[data⟦6;;8⟧]	0.40549	0	Variance[data⟦1;;5⟧]	-0.688335	Variance[data⟦6;;8⟧]

6	9	3	-0.0418609	0	0.495132	0	Mean[data⟦1 ;; 6⟧]	1.4854	Mean[data⟦7 ;; 9⟧]	0.395354	0	Variance[data⟦1 ;; 6⟧]	-0.624987	Variance[data⟦7 ;; 9⟧]
7	10	3	0.054705	0	0.45109	0	Mean[data⟦1 ;; 7⟧]	1.50363	Mean[data⟦8 ;; 10⟧]	0.370824	0	Variance[data⟦1 ;; 7⟧]	-0.705248	Variance[data⟦8 ;; 10⟧]

In[78]:=

```
Module[{w = 3, u, j, mnp1, mjp1, mw, vnp1, vjp1, vw,
   cbuf, zj, checkMw, checkVw, checkMjp1, checkVjp1, vw1, cume,
   data = {0.857454, 0.312454, 0.705325, 0.839363,
     1.63781, 0.699257, -0.340016, -0.213596, -0.0418609, 0.054705}},
  cbuf = ConstantArray[0, w];
  cume[{_, n_, _, _, _, mn_, mj_, _, _, _, vn_, vj_, _, _, _}, z_] := (

    (* there are five recurrent inputs and two actionable outputs *)
    (*     in the circulating-state vector. The rest of the items *)
    (*     are auxilaries for verifying and debugging. *)

    (* window starts overlapping data by 1 cell, then by 2 cells *)
    (* eventually window becomes flush left, then advances into *)
    (* the data, leaving evicted data to its left *)
    (* datum at the right of the window -- datum to be evicted *)
    zj = cbuf⟦w⟧;
    (* rotate zj to the left of the window *)
    cbuf = RotateRight[cbuf, 1];
    (* replace zj with z *)
    cbuf⟦1⟧ = z;
    (* window starts containing one datum when n is 0 *)
    (* eventually, when n is w-1, window is flush left *)
    (* j = elements to left of window, 0 when flush left *)
    (*     then > 0 thereafter if w < N *)
    j = Max[0, n - w + 1];
    (* u = number of data in the window *)
    (* starts at 1, becomes 2, ..., w *)
    (* becomes w when j is flush left and stays w *)
    u = n - j + 1; Assert[u ≤ w];
```

```
(* mn = prior mean of all data before z -- recurrent input *)
(* mnp1 = posterior mean of all data including z *)
(* this is our normal gain times residual calc *)
mnp1 = mn + (z - mn)/(n + 1);
(* mj = prior mean of data left of window -- recurrent input*)
(* mjp1 = posterior mean of data left of the window *)
mjp1 = If[j > 0, mj + (zj - mj)/j, 0];
(* mw = posterior mean of data in window -- output *)
mw = ((n + 1) mnp1 - j mjp1)/u;
(* vn = prior variance of all data -- recurrent input *)
(* posterior variance of all data via Recurrence 4 *)
vnp1 = ((n - 1) vn + n/(n+1) (z - mn)^2)/Max[1, n];
(* vj = prior variance of data left of window -- recurrent input *)
(* vjp1 = posterior variance of data to left of window *)
(* increase by scaled residual of the evicted datum vj *)
(* will be zero until window moves off of flush left *)
(* so it is safe to multiply it later by a negative n-w *)
vjp1 = If[j > 1, (j - 2)/(j - 1) vj + 1/j (zj - mj)^2, 0];
(* vw = posterior variance of data in window -- output *)
(* three applications of the school formula *)
(* n_Bessel v_n+1 = Σ_i=1^n+1 (z_i)^2-(n+1) z̄_n+1 ; ditto for j *)
(* The first two terms in the numerator combine to equal *)
(*   the sum of squared data (not residuals) in the window *)
(* the third term in the numerator is the school correction *)
(* for the window *);
vw = ((n vnp1 + (n + 1) mnp1^2) - ((n - w) vjp1 + j mjp1^2) - u mw^2)/Max[1, u - 1];
checkMjp1 = If[j > 0, Mean[data[[1 ;; j]]], 0];
Assert[0 === Chop[checkMjp1 - mjp1]];
checkMw = Mean[data[[j + 1 ;; n + 1]]];
Assert[0 === Chop[checkMw - mw]];
checkVjp1 = If[j > 1, Variance[data[[1 ;; j]]], 0];
Assert[0 === Chop[checkVjp1 - vjp1]];
checkVw = If[u > 1, Variance[data[[j + 1 ;; n + 1]]], 0];
Assert[0 === Chop[checkVw - vw]];
(* RETURN VALUE *)
{j,
      n + 1,
      u,
      z,
      zj,
      mnp1,
      mjp1,
      checkMjp1,
      mw,
      checkMw,
      vnp1,
      vjp1,
      checkVjp1,
      vw,
      checkVw
     });
  Grid[
   Prepend[FoldList[cume, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, data],
    {"j", "n+1", "u", "z", "zⱼ", "μ'", "μⱼ́", "chk μⱼ́", "μ_ẃ",
     "chk μ_ẃ", "υ'", "υⱼ́", "chk υⱼ́", "υ_ẃ", "chk υ_ẃ"}], Frame → All]]
```

Out[78]=

j	n+1	u	z	z_j	μ'	μ'_j	chk μ'_j	μ'_w	chk μ'_w	υ'	υ'_j	chk υ'_j	υ'_w	chk υ'_w
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0.857454	0	0.857454	0	0	0.857454	0.857454	0.	0	0	0.	0
0	2	2	0.312454	0	0.584954	0	0	0.584954	0.584954	0.148513	0	0	0.148513	0.148513
0	3	3	0.705325	0	0.625078	0	0	0.625078	0.625078	0.079086	0	0	0.079086	0.079086
1	4	3	0.839363	0.857454	0.678649	0.857454	0.857454	0.619047	0.619047	0.0642035	0	0	0.0749912	0.0749912
2	5	3	1.63781	0.312454	0.870481	0.584954	0.584954	1.06083	1.06083	0.232151	0.148513	0.148513	0.254169	0.254169
3	6	3	0.699257	0.705325	0.841944	0.625078	0.625078	1.05881	1.05881	0.190607	0.079086	0.079086	0.256338	0.256338
4	7	3	-0.340016	0.839363	0.673092	0.678649	0.678649	0.665684	0.665684	0.358415	0.0642035	0.0642035	0.978794	0.978794
5	8	3	-0.213596	1.63781	0.562256	0.870481	0.870481	0.0485483	0.0485483	0.40549	0.232151	0.232151	0.321562	0.321562
6	9	3	-0.0418609	0.699257	0.495132	0.841944	0.841944	-0.198491	-0.198491	0.395354	0.190607	0.190607	0.0223952	0.0223952
7	10	3	0.054705	-0.340016	0.45109	0.673092	0.673092	-0.0669173	-0.0669173	0.370824	0.358415	0.358415	0.0184672	0.0184672

Window 6

```
Module[{w = 6, u, j, mnp1, mjp1, mw, vnp1, vjp1, vw, cbuf, zj,
   data = {0.857454, 0.312454, 0.705325, 0.839363, 1.63781,
     0.699257, -0.340016, -0.213596, -0.0418609, 0.054705, 1.10464,
     -0.387322, -0.00175018, 1.12034, 0.280948, -1.07877}, cume},
  cbuf = ConstantArray[0, w];
  cume[{_, n_, _, _, _, mn_, mj_, _, _, _, vn_, vj_, _, _, _}, z_] := (
    zj = cbuf[[w]];
    cbuf = RotateRight[cbuf, 1];
    cbuf[[1]] = z;
    j = Max[0, n - w + 1];    u = n - j + 1;
    mnp1 = mn + (z - mn)/(n + 1);
    mjp1 = If[j > 0, mj + (zj - mj)/j, 0];
    mw = ((n + 1) mnp1 - j mjp1)/u;
    vnp1 = ((n - 1) vn + n/(n+1) (z - mn)^2)/Max[1, n];
    vjp1 = If[j > 1, (j - 2)/(j - 1) vj + 1/j (zj - mj)^2, 0];
    vw = (n vnp1 - (n - w) vjp1 + (n + 1) mnp1^2 - j mjp1^2 - u mw^2)/Max[1, u - 1];
    {j, n + 1, u, z, zj, (* THIS IS THE RETURN VALUE *)
     mnp1, mjp1, If[j > 0, Mean[data[[1 ;; j]]], 0],
     mw, Mean[data[[j + 1 ;; n + 1]]],
     vnp1, vjp1, If[j > 1, Variance[data[[1 ;; j]]], 0],
     vw, If[u > 1, Variance[data[[j + 1 ;; n + 1]]], 0]});
  Grid[
   Prepend[FoldList[cume, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, data],
    {"j", "n+1", "u", "z", "zj", "μn+1", "μj",
     "μ́j", "μw", "μ́w", "υn+1", "υj", "ύj", "υw", "ύw"}], Frame → All]]
```

Out[ ]=

j	n+1	u	z	z_j	μ_{n+1}	μ_j	$\acute{\mu}_j$	μ_w	$\acute{\mu}_w$	υ_{n+1}	υ_j	$\acute{\upsilon}_j$	υ_w	$\acute{\upsilon}_w$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0.85⋱ 7454	0	0.85⋱ 7454	0	0	0.85⋱ 7454	0.85⋱ 7454	0.	0	0	0.	0

0	2	2	0.31⋱2454	0	0.58⋱4954	0	0	0.58⋱4954	0.58⋱4954	0.14⋱8513	0	0	0.14⋱8513	0.14⋱8513
0	3	3	0.70⋱5325	0	0.62⋱5078	0	0	0.62⋱5078	0.62⋱5078	0.07⋱9086	0	0	0.07⋱9086	0.07⋱9086
0	4	4	0.83⋱9363	0	0.67⋱8649	0	0	0.67⋱8649	0.67⋱8649	0.06⋱420⋱35	0	0	0.06⋱420⋱35	0.06⋱420⋱35
0	5	5	1.63⋱781	0	0.87⋱0481	0	0	0.87⋱0481	0.87⋱0481	0.23⋱2151	0	0	0.23⋱2151	0.23⋱2151
0	6	6	0.69⋱9257	0	0.84⋱1944	0	0	0.84⋱1944	0.84⋱1944	0.19⋱0607	0	0	0.19⋱0607	0.19⋱0607
1	7	6	−0.⋱3⋱4⋱0⋱0⋱1⋱6	0.85⋱7454	0.67⋱3092	0.85⋱7454	0.85⋱7454	0.64⋱2365	0.64⋱2366	0.35⋱8415	0	0	0.42⋱2167	0.42⋱2167
2	8	6	−0.⋱2⋱1⋱3⋱5⋱9⋱6	0.31⋱2454	0.56⋱2256	0.58⋱4954	0.58⋱4954	0.55⋱4691	0.55⋱4691	0.40⋱549	0.14⋱8513	0.14⋱8513	0.53⋱7708	0.53⋱7708
3	9	6	−0.⋱0⋱4⋱1⋱8⋱6⋱0⋱9	0.70⋱5325	0.49⋱5132	0.62⋱5078	0.62⋱5078	0.43⋱016	0.43⋱016	0.39⋱5354	0.07⋱9086	0.07⋱9086	0.58⋱5735	0.58⋱5735
4	10	6	0.05⋱4705	0.83⋱9363	0.45⋱109	0.67⋱8649	0.67⋱8649	0.29⋱9383	0.29⋱9383	0.37⋱0824	0.06⋱420⋱35	0.06⋱420⋱35	0.55⋱9916	0.55⋱9916
5	11	6	1.10⋱464	1.63⋱781	0.51⋱0503	0.87⋱0481	0.87⋱0481	0.21⋱0522	0.21⋱0522	0.37⋱2571	0.23⋱2151	0.23⋱2151	0.32⋱1851	0.32⋱1851

6	12	6	-0.3873222	0.699257	0.435684	0.841944	0.841944	0.029425	0.029425	0.405875	0.190607	0.190607	0.306206	0.306206
7	13	6	-0.00175018	-0.340016	0.402036	0.673092	0.673092	0.0858027	0.0858027	0.386771	0.358415	0.358415	0.275289	0.275289
8	14	6	1.12034	-0.213596	0.453343	0.562256	0.562256	0.308125	0.308125	0.393874	0.40549	0.40549	0.412102	0.412102
9	15	6	0.280948	-0.0418609	0.44185	0.495132	0.495132	0.361927	0.361927	0.367722	0.395354	0.395354	0.384278	0.384278
10	16	6	-1.07877	0.054705	0.346811	0.45109	0.45109	0.173014	0.173014	0.487725	0.370824	0.370824	0.737697	0.737697

Window 3, Random Data

```
Module[{w = 3, u, j, mnp1, mjp1, mw, vnp1, vjp1, vw, cbuf, zj,
  data = N@RandomInteger[{-100, 100}, 16], cume},
 cbuf = ConstantArray[0, w];
 cume[{_, n_, _, _, _, mn_, mj_, _, _, _, vn_, vj_, _, _, _}, z_] := (

   zj = cbuf[[w]];
   cbuf = RotateRight[cbuf, 1];
   cbuf[[1]] = z;
   j = Max[0, n - w + 1];
   u = n - j + 1;
   mnp1 = mn + (z - mn)/(n + 1);
   mjp1 = If[j > 0, mj + (zj - mj)/j, 0];
   mw = ((n + 1) mnp1 - j mjp1)/u;
   vnp1 = ((n - 1) vn + n/(n+1) (z - mn)^2)/Max[1, n];
   vjp1 = If[j > 1, (j - 2)/(j - 1) vj + 1/j (zj - mj)^2, 0];
   vw = (n vnp1 - (n - w) vjp1 + (n + 1) mnp1^2 - j mjp1^2 - u mw^2)/Max[1, u - 1];
   {j, n + 1, u, z, zj, (* THIS IS THE RETURN VALUE *)
    mnp1, mjp1, If[j > 0, Mean[data[[1 ;; j]]], 0],
    mw, Mean[data[[j + 1 ;; n + 1]]],
    vnp1, vjp1, If[j > 1, Variance[data[[1 ;; j]]], 0],
    vw, If[u > 1, Variance[data[[j + 1 ;; n + 1]]], 0]});

 Grid[
  Prepend[FoldList[cume, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, data],
   {"j", "n+1", "u", "z", "z_j", "μ_n+1", "μ_j",
    "μ'_j", "μ_w", "μ'_w", "υ_n+1", "υ_j", "υ'_j", "υ_w", "υ'_w"}], Frame → All]]
```

Out[ ]=

j	n+1	u	z	z_j	μ_{n+1}	μ_j	μ'_j	μ_w	μ'_w	υ_{n+1}	υ_j	υ'_j	υ_w	υ'_w
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	-20.	0	-20.	0	0	-20.	-20.	0.	0	0	0.	0
0	2	2	45.	0	12.5	0	0	12.5	12.5	2112.5	0	0	2112.5	2112.5
0	3	3	-94.	0	-23.	0	0	-23.	-23.	4837.	0	0	4837.	4837.

1	4	3	-63.	-20.	-33.	-20.	-20.	-37.3333	-37.3333	3624.67	0	0	5324.33	5324.33
2	5	3	82.	45.	-10.	12.5	12.5	-25.	-25.	5363.5	2112.5	2112.5	8827.	8827.
3	6	3	64.	-94.	2.33333	-23.	-23.	27.6667	27.6667	5203.47	4837.	4837.	6246.33	6246.33
4	7	3	10.	-63.	3.42857	-33.	-33.	52.	52.	4344.62	3624.67	3624.67	1404.	1404.
5	8	3	-76.	82.	-6.5	-10.	-10.	-0.666667	-0.666667	4512.57	5363.5	5363.5	4985.33	4985.33
6	9	3	-21.	64.	-8.11111	2.33333	2.33333	-29.	-29.	3971.86	5203.47	5203.47	1897.	1897.
7	10	3	-94.	10.	-16.7	3.42857	3.42857	-63.6667	-63.6667	4268.23	4344.62	4344.62	1446.33	1446.33
8	11	3	-53.	-76.	-20.	-6.5	-6.5	-56.	-56.	3961.2	4512.57	4512.57	1339.	1339.
9	12	3	35.	-21.	-15.4167	-8.11111	-8.11111	-37.3333	-37.3333	3853.17	3971.86	3971.86	4344.33	4344.33
10	13	3	82.	-94.	-7.92308	-16.7	-16.7	21.3333	21.3333	4262.08	4268.23	4268.23	4696.33	4696.33
11	14	3	10.	-53.	-6.64286	-20.	-20.	42.3333	42.3333	3957.17	3961.2	3961.2	1336.33	1336.33
12	15	3	94.	35.	0.0666667	-15.4167	-15.4167	62.	62.	4349.78	3853.17	3853.17	2064.	2064.
13	16	3	27.	82.	1.75	-7.92308	-7.92308	43.6667	43.6667	4105.13	4262.08	4262.08	1972.33	1972.33

In[ ]:=

```
DynamicModule[{w = 3, u, l, mn, ml, mw, υn,
  υl, υw, data = N@RandomInteger[{-100, 100}, 10], cume},
cume[{_, n_, _, _, x_, xw_, _, _, v_, vw_, _, _}, z_] := (

  (* n : how many points seen before this one. *)
  (* z : current data point, at index n + 1. *)
  (* l :
   index of first point in window at least 1 wide and no more than w wide. *)
  (* w : width of window. *)
  (* u : number of points including z in the running window;
  converges to w. *)
  (* mn : the mean of all points through z_{n+1}. *)
  (* ml : the mean of all points through index l-1 = n+1-w,
  lagging w behind mn. *)
  (* mw : the mean of all points within the window. *)
  (* ʋn : variance of all points through z_{n+1}. *)
  (* ʋl : variance of all points through index l-1 = n+1-w,
  lagging w behind ʋn *)
  (* ʋw : variance of all points within the window. *)

  l = Max[1, n - w + 2];
  mn = x + (z - x)/(n + 1);
  ml = If[n + 1 > w, xw + (data[[n + 1 - w]] - xw)/(n + 1 - w), 0];
  mw = ((n + 1) mn - (n + 1 - w) ml)/w;
  ʋn = ((n - 1) v + n/(n+1) (z - x)^2)/Max[1, n];
  ʋl = If[n + 1 > w, ((n - w - 1) ʋl + (n-w)/(n-w+1) (data[[n + 1 - w]] - xw)^2)/Max[1, n - w], 0];
  ʋw = (n ʋn + (n + 1) mn^2 - (n - w) ʋl - (n + 1 - w) ml^2 - w mw^2)/(w - 1);
  {l, n + 1, n + 2 - l, z,
   mn, ml, mw, Mean[data[[l ;; n + 1]]],
   ʋn, ʋl, ʋw, Quiet@Variance[data[[l ;; n + 1]]]}
  );

Manipulate[Grid[
  Prepend[FoldList[cume, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, data],
   {"l", "n+1", "u", "z", "μ_{n+1}", "μ_l", "μ_w", "μ´_w", "ʋ_{n+1}", "ʋ_l", "ʋ_w", "ʋ´_w"}],
  Frame → All],
 Button["RANDOMIZE!", data = N@RandomInteger[{-100, 100}, 10]]]
```

Out[•]=

RANDOMIZE !

l	n+1	u	z	μ_{n+1}	μ_l	μ_w	μ'_w	υ_{n+1}	υ_l	υ_w	υ'_w
0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	-22.	-22.	0	-7.33333	-22.	0.	0	161.333	Variance[{-22.}]
1	2	2	-4.	-13.	0	-8.66667	-13.	162.	0	137.333	162.
1	3	3	15.	-3.66667	0	-3.66667	-3.66667	342.333	0	342.333	342.333
2	4	3	46.	8.75	-22.	19.	19.	844.917	0.	637.	637.
3	5	3	-74.	-7.8	-13.	-4.33333	-4.33333	2003.2	162.	3880.33	3880.33
4	6	3	81.	7.	-3.66667	17.6667	17.6667	2916.8	342.333	6608.33	6608.33
5	7	3	-50.	-1.14286	8.75	-14.3333	-14.3333	2894.81	844.917	6960.33	6960.33
6	8	3	91.	10.375	-7.8	40.6667	40.6667	3542.55	2003.2	6190.33	6190.33
7	9	3	28.	12.3333	7.	23.	23.	3134.25	2916.8	4989.	4989.
8	10	3	-66.	4.5	-1.14286	17.6667	17.6667	3399.61	2894.81	6242.33	6242.33