Windowed Incremental Online Statistics

Extracting Models from Data, One Observation at a Time

Brian Beckman
11 May 2024

We calculate descriptive statistics—count, mean, unbiased variance—both overall and in a sliding window, of sequences of numerical data. The statistics calculator is written as a ***foldable accumulator function***, decoupling calculation from iteration. Exactly the same stateless accumulator function works over data distributed over space in memory or in tables, or over time in asynchronous or lazy streams. Foldable accumulator functions can be tested independently of data source, say over ground-truth test data, then later deployed in harsh asynchronous environments with high assurance, often without even being recompiled.

The solution at the end of this notebook has been implemented in C, Python, and Clojure and deployed in mission-critical web applications and embedded applications. It runs in $O(w)$ space, where w is the width of the window, and in $O(1)$ time. Contrast to more obvious solutions that run in $O(w)$ time.

Prelude: Running Count

For ground truth, we want a fixed but random sequence of data. The following is a sample of length 10 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};
```

`Fold` has three arguments: an accumulator function, an initial state, and a sequence of data. All variations of `Fold`—over asynchronous streams, over on-demand (lazy) streams, over tables, lists, arrays—have exactly the same signature as `Fold`.

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. Above, the circulating state is n. The caller must feed in two arguments to `cume`: the ***prior***

value *n* of the circulating 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. The following should be self-explanatory:

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*}. That's why we say *Fold decouples calculation from iteration*. **Fold** places no restriction on its accumulator-function argument *f* other than that 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

It is important to know that there is a dual: the right fold. Left and Right folds can behave differently on lazy lists, so be on the lookout! We use only left fold in this document.

In[7]:=
```
Fold[{x, y} ↦ f[y, x], z, {a, b, c, d}]
```

Out[7]=
```
f[d, f[c, f[b, f[a, z]]]]
```

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*. This 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 track the running mean calculated this way. We relieve that situation shortly below.

In[8]:=
```
ClearAll[cume];
cume[{x_, n_, sum_}, z_] :=
  {(sum + z)/(n + 1), n + 1, sum + z};
Fold[cume, {0, 0, 0}, zs]
```

Out[10]=
```
{-0.279472, 10, -2.79472}
```

Check against *Mathematica* built-in.

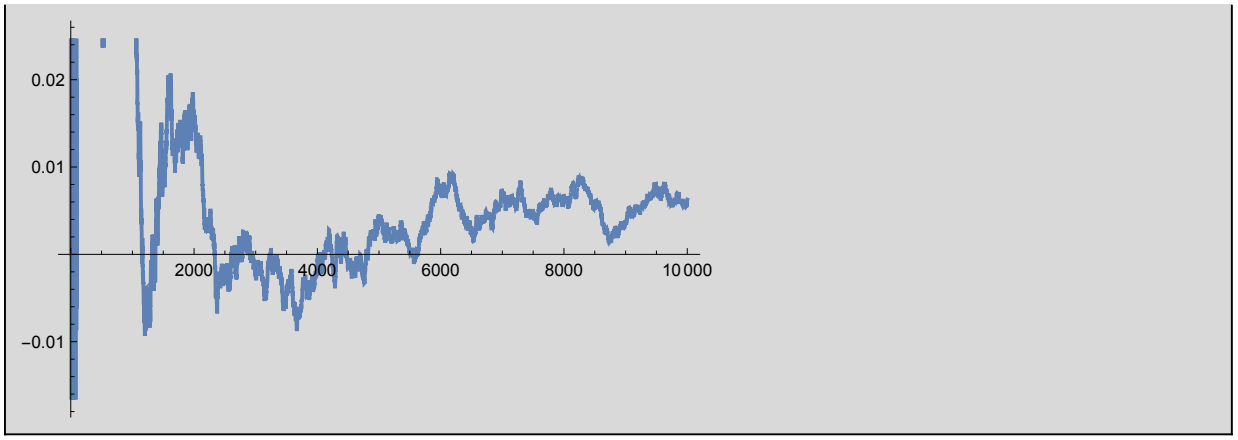
In[11]:=
```
Mean[zs]
```

Out[11]=
```
-0.279472
```

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

In[12]:=
```
ListLinePlot[
 #[[1]] & /@ FoldList[cume, {0, 0, 0}, RandomVariate[NormalDistribution[], 10 000]]]
```

Out[12]=



Preferable Form

Here is a preferable form with one fewer quantity in the circulating state. This form is so powerful that we state it as a mantra or slogan:

compute the new value as sum of the old value and a ***correction*** that depends only on old values and the single observation *z*.

We will strive for this form in everything that follows. It is the essential feature of all recurrent filter algorithms.

The right-hand side of the ***recurrence*** $\overline{z} \leftarrow \overline{z} + K \times (z - \overline{z})$, $\overline{z}$ 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 the following equation:

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

In[13]:=
```
ClearAll[cume];
cume[{x_, n_}, z_] :=
  With[{K = 1/(n + 1)},
   {x + K (z - x), n + 1}];
Fold[cume, {0, 0}, zs]
```

Out[15]=
```
{-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 residual is 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 $\overline{z}$, the old mean, as $\frac{\overline{z}(n+1)}{n+1} = \frac{n\overline{z}+\overline{z}}{n+1}$ 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 *sum* is $n\overline{z}$. I think this is beautiful. The same reasoning will help us derive many more sophisticated formulas, up-to-and-including the Kalman update.

For another twist on the proof above, we can let *Mathematica* do the work. 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[16]:=
```
Solve[x + K (z - x) == (n x + z)/(n + 1), K]
```

Out[16]=
```
{{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[17]:=
```
Variance[zs]
```
Out[17]=
```
0.395183
```

Variance is also the square of standard deviation.

In[18]:=
```
StandardDeviation[zs]^2 == Variance[zs]
```
Out[18]=
```
True
```

The ***sum of squared residuals***,

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

is the variance times the length-less-one:

In[19]:=
```
Variance[zs] * (Length[zs] - 1)
```
Out[19]=
```
3.55665
```

The sum of squared residuals will be a cornerstone quantity for all future calculations. Here is another look at it.

In[20]:=
```
With[{μ = Mean[zs]},
 Plus @@ ((# - μ)^2 & /@ zs)]
```
Out[20]=
```
3.55665
```

Requirement

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

The definition explicitly refers to the past history. It seems, at first glance, impossible to compute the variance without keeping the entire history. But it possible, 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, unknown, or infinite length, so does not meet the requirement of constant memory.

- *Re-use the variable zs, now for a shorter sequence of ground-truth data, convenient for checking arithmetic in one's head.*

In[21]:=
```
ClearAll[zs]; zs = {55, 89, 144};
```

In[22]:=
```
Variance[zs] // N
```

Out[22]=
```
2017.
```

Hand-calculated mean:

In[23]:=
```
(55 + 89 + 144) / 3.
```

Out[23]=
```
96.
```

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

In[24]:=
```
((55 - 96.) ^ 2 + (89 - 96.) ^ 2 + (144 - 96.) ^ 2) / 2.
```

Out[24]=
```
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[25]:=
```
ClearAll[cume];
cume[var_, z_] := var + (z - Mean[zs])^2 / (Length[zs] - 1);
FoldList[cume, 0., zs]
```

Out[27]=
```
{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*:

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

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. Concatenate the variance and *ssq* to the circulating state, and write:

In[28]:=
```
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[30]//MatrixForm=

0	0	0	0
0	3025	55	1
578	10 946	72	2
2017	31 682	96	3

The variance computed this way is incremental and runs in constant memory, though it is *not yet 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, *ssq*, $\Sigma_n \overset{\text{def}}{=} \sum_{i=1}^{n}(z_i-\overline{z}_n)^2$:

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

The form in Recurrence 4 is an abbreviation for the upcoming Equation 5. The abbreviation is precise under the rule that all accumulated quantities, namely $\overline{z}$ 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.

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

Symbolic 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 the sum of squares—not the sum of squared residual—$sz_n^2 =$ `sz2[n]` be $\sum_{i=1}^{n} z_i^{2}$. Note that $sz_{n+1}^2 = sz_n^2 + z^2$, and also that $(n+1)\,\overline{z}_{n+1} = n\,\overline{z}_n + z$. Those obvious (but beautiful) equalities are enough to effect the proof.

Transcribe Equation 6.

```
In[31]:= ClearAll[s, zb, zbar];
         s[n_] := sz2[n] - n zbar[n]^2
```

Transcribe the “obvious equalities” above.

```
In[33]:= 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]`.

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

```
In[34]:= s[n + 1] - s[n] //. rulez$ // FullSimplify
```

```
Out[34]= n (z - zbar[n])^2
         -----------------
               1 + n
```

Proof by Hand

Remember the “obvious equalities.”

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

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

Expand out the definition of S_{n+1} and employ the "obvious equalities":

$$S_{n+1} = \left[s z_{n+1}^2 - (n+1)\,\bar{z}_{n+1}^2\right] = \left[s z_n^2 + z^2\right] - \frac{1}{n+1}\left[(n+1)\,\bar{z}_{n+1}\right]^2 \quad = \quad \left[s z_n^2 + z^2\right] - \frac{1}{n+1}\left[n\,\bar{z}_n + z\right]^2$$

Expand the second term:

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

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

$$= \left[s z_n^2 - n\,\bar{z}_n^2\right] + n\,\frac{n+1}{n+1}\,\bar{z}_n^2 + \frac{n+1}{n+1}\,z^2 - \frac{n^2}{n+1}\,\bar{z}_n^2 - \frac{2\,n}{n+1}\,z\,\bar{z}_n - \frac{1}{n+1}\,z^2$$

Cancel terms and complete the square:

$$= \left[s z_n^2 - n\,\bar{z}_n^2\right] + \frac{n}{n+1}\,\bar{z}_n^2 - \frac{2\,n}{n+1}\,z\,\bar{z}_n + \frac{n}{n+1}\,z^2$$

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

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[35]:=
```
(z - zbar[n]) (z - zbar[n + 1] //. rulez$ // FullSimplify)
```

Out[35]=

$$\frac{n\,(z - \text{zbar}[n])^2}{1 + n}$$

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`:

In[36]:=
```
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
```

Out[38]//MatrixForm=

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 55 & 1 \\ 578 & 72 & 2 \\ 2017 & 96 & 3 \end{pmatrix}$$

Folding It

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

In[39]:=
```
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
```

Out[41]//MatrixForm=

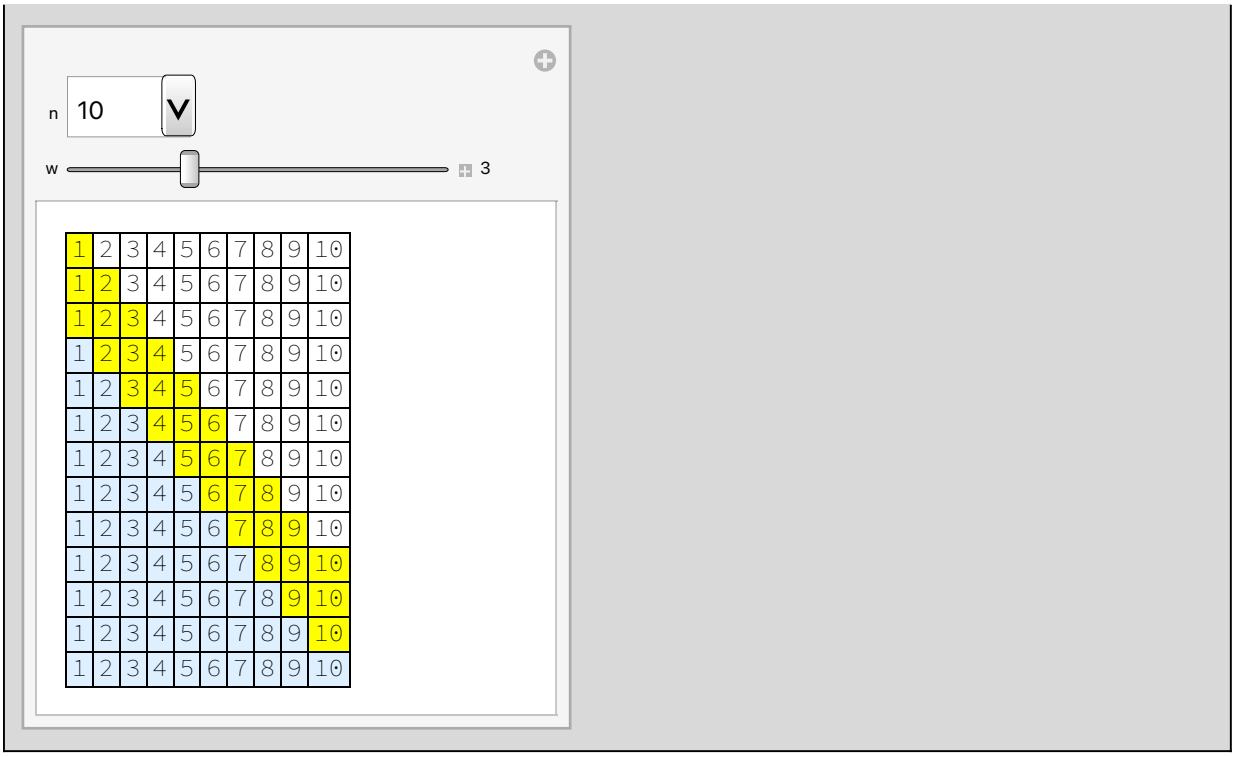
$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 55 & 1 \\ 578 & 72 & 2 \\ 2017 & 96 & 3 \end{pmatrix}$$

Windowed Statistics

Sometimes, we want the mean and variance of a limited amount of the past, say over a window of length $w \geq 0$.

In[42]:=
```
Manipulate[
 Grid[Table[With[{l = i - w + 1},
   {i, n + w}, {j, n}], Frame → All],
 {{n, 10}, 5 + Range[100]},
 {{w, 3}, Range[n + 1] - 1, Appearance → "Labeled"}]
```

Out[42]=



Foldable

`runningStatsFoldable` is a a foldable accumulator function that effects windowed statistics. Its circulating state carries extra information for pedagogy and convenience. It's possible to make the interface much tighter. The derivation of its formulae are written in comments. The specific example data in this section are taken from other deployments of this code in C and Python. In the next section, we add some checks of the results against Mathematica's built-ins.

In[43]:=
```
On[Assert];
```

In[44]:=
```
ClearAll[runningStatsFoldable];
Module[{u, j, mnp1, mjp1, mw, vnp1, vjp1, vw, cbuf, zj},
```

```
runningStatsFoldable[w_][{
   _, (* j, count of elements left of the window, j≥0 *)
   n_, (* index of the current datum; start at 0 *)

   _, (* u, elements in the window, 0≤u≤w *)
   _, (* z, new datum circulated for convenience *)
   _, (* zⱼ, datum evicted from the window *)

   mn_, (* mean of all data *)
   mj_, (* mean of data to the left of the window *)
   _, (* mean of data in the window *)

   vn_, (* variance of all data *)
   vj_, (* variance of data to the left of the window *)
   _}, (* variance of data in the window *)
  z_] := (

 cbuf = ConstantArray[0, w];

 (* there are five recurrent inputs n, mn, mj, vn, vj *)
 (*     and two actionable outputs mw and vw *)
 (*     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 *)

(* by 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];
    (* RETURN VALUE *)
    {j, n + 1, u, z, zj,
     mnp1, mjp1, mw,
     vnp1, vjp1, vw
    }]];
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}, data],
   {"j", "n+1", "u", "z", "zⱼ", "μ'", "μⱼ́", "μẃ", "υ'", "υⱼ́", "υẃ"}], Frame → All]]
```

Out[46]=

j	n+1	u	z	z_j	μ'	$\acute{\mu_j}$	$\acute{\mu_w}$	υ'	$\acute{\upsilon_j}$	$\acute{\upsilon_w}$
0	0	0	0	0	0	0	0	0	0	0
0	1	1	0.857454	0	0.857454	0	0.857454	0.	0	0.
0	2	2	0.312454	0	0.584954	0	0.584954	0.148513	0	0.148513
0	3	3	0.705325	0	0.625078	0	0.625078	0.079086	0	0.079086
1	4	3	0.839363	0	0.678649	0	0.904865	0.0642035	0	-0.210738
2	5	3	1.63781	0	0.870481	0	1.4508	0.232151	0	-0.798595
3	6	3	0.699257	0	0.841944	0	1.68389	0.190607	0	-1.65009
4	7	3	-0.340016	0	0.673092	0	1.57055	0.358415	0	-1.03901
5	8	3	-0.213596	0	0.562256	0	1.49935	0.40549	0	-0.688335
6	9	3	-0.0418609	0	0.495132	0	1.4854	0.395354	0	-0.624987
7	10	3	0.054705	0	0.45109	0	1.50363	0.370824	0	-0.705248

Random Data, Checked

The solution does not work when the window size is zero, but is robust even in the nonsensical case of the window's being longer than the data

.

In[47]:=

```
DynamicModule[{u, j, mnp1, mjp1, mw, vnp1, vjp1, vw, cbuf, zj,
  ndata = 16, w = 6, data, cume},
 data = N@RandomReal[{-100, 100}, ndata];
 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 + 1) mnp1^2) - (j mjp1^2 + (n - w) vjp1) - u mw^2)/Max[1, u - 1];
   Assert[0 === Chop[mjp1 - If[j > 0, Mean[data[[1 ;; j]]], 0]]];
   Assert[0 === Chop[mw - Mean[data[[j + 1 ;; n + 1]]]]];
   Assert[0 === Chop[vjp1 - If[j > 1, Variance[data[[1 ;; j]]], 0]]];
   Assert[0 === Chop[vw - If[u > 1, Variance[data[[j + 1 ;; n + 1]]], 0]]];
   {j, n + 1, u, z, zj,
    mnp1, mjp1, mw,
    vnp1, vjp1, vw});
 Manipulate[
  (ndata = dataLen; w = windowLen; cbuf = ConstantArray[0, w]; Grid[
    Prepend[FoldList[cume, {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_w", "ν_{n+1}", "ν_j", "ν_w"}],
    Frame → All]),
  {dataLen, {16, Sequence @@ Range[10, 40]}},
  {windowLen, {6, Sequence @@ Range[1, 40]}},
  Button["RANDOMIZE!",
   data = N@RandomReal[{-100, 100}, ndata]]
 ]]
```

Out[47]=

dataLen 16

windowLen 6

RANDOMIZE !

j	n+1	u	z	z_j	μ_{n+1}	μ_j	μ_w	υ_{n+1}	υ_j	υ_w
0	0	0	0	0	0	0	0	0	0	0
0	1	1	-11.16`:58	0	-11.16`:58	0	-11.16`:58	0.	0	0.
0	2	2	-6.4896	0	-8.8277	0	-8.8277	10.9335	0	10.9335
0	3	3	87.268	0	23.2042	0	23.2042	3083.6	0	3083.6
0	4	4	14.2296	0	20.9606	0	20.9606	2075.87	0	2075.87
0	5	5	91.344	0	35.0373	0	35.0373	2547.67	0	2547.67
0	6	6	-98.69`:58	0	12.7484	0	12.7484	5018.89	0	5018.89
1	7	6	64.987	-11.16`:58	20.2111	-11.16`:58	25.4405	4572.25	0	5256.98
2	8	6	93.5184	-6.4896	29.3745	-8.8277	42.1085	4590.81	10.9335	5646.6
3	9	6	-98.57`:61	87.268	15.1577	23.2042	11.1345	5836.	3083.6	8045.89
4	10	6	17.0321	14.2296	15.3452	20.9606	11.6016	5187.91	2075.87	8050.67
5	11	6	9.29872	91.344	14.7955	35.0373	-2.072`:62	4672.44	2547.67	6555.58
6	12	6	74.8767	-98.69`:58	19.8023	12.7484	26.8561	4548.49	5018.89	4868.36
7	13	6	5.58482	64.987	18.7086	20.2111	16.9558	4184.99	4572.25	4550.44
8	14	6	61.4841	93.5184	21.764	29.3745	11.6167	3993.77	4590.81	3740.43
9	15	6	-49.88`:79	-98.57`:61	16.9872	15.1577	19.7314	4050.76	5836.	1989.48
10	16	6	-67.39`:15	17.0321	11.7135	15.3452	5.66082	4225.7	5187.91	3268.52