# Composable Statistics

Brian Beckman

May 21, 2019

## Contents

# 1 COMPOSABLE STATISTICS

## 1.1 TODO TANGLE, NOWEB

This will become a fully literate program via org-babel `tangle` and `noweb`. At present, it is an org file converted directly from an old iPython (Jupyter) notebook. Jupyter proved not to scale well.

## 1.2 TODO GRAPHICS

We still have to work out how to embed graphics in this file.

## 1.3 CLOJURE

We prefer Clojure to Python for this exercise due to Clojure's concurrency primitives, especially atoms and core.async. Python is growing and improving rapidly, so we may return to it someday.

### 1.3.1 TODO PROJECT.CLJ

At present, the critical file `project.clj` is external to this document. It will be one of the first to tangle.

The best sites for learning Clojure by example are clojuredocs.org and 4clojure.org. A recommended book is Clojure for the Brave and True.

## 1.4 TODO HOW TO USE THIS DOCUMENT

Explain how to run Clojure code inside an org-mode buffer, how to tangle and weave, etc.

1. Install leiningen `https://leiningen.org/` (this is all you need for Clojure)

# 2 INTRODUCTION

We want to compute descriptive statistics in constant memory. We want exactly the same code to run over sequences distributed in space as runs over sequences distributed in time. Sequences distributed in space are vectors, lists, arrays, lazy or not. Sequences distributed over time are asynchronous streams. Descriptive statistics range from `count`, `mean`, `max`, `min`, and `variance` to Kalman filters and Gaussian processes. We decouple computation from data delivery by packaging computation in composable functions.

Some sample scalar data:

```
(def zs [-0.178654, 0.828305, 0.0592247, -0.0121089, -1.48014,
         -0.315044, -0.324796, -0.676357, 0.16301, -0.858164])
```

## 2.1 TODO: GENERATE NEW RANDOM DATA

# 3 RUNNING COUNT

The traditional and obvious way with `reduce` and `reductions` (`https://clojuredocs.org/clojure.core/reduce`). *Reduce* takes three arguments: a binary function, an initial value, and a space-sequence of inputs.

```
(reduce
    (fn [count datum] (inc count)) ; binary function
    0                              ; initial value
    zs)                            ; space sequence
```

```
10
```

... with all intermediate results:

```
(reductions (fn [c z] (inc c)) 0 zs)
```

0 1 2 3 4 5 6 7 8 9 10

## 3.1 THREAD-SAFE

Overkill for sequences in space, but safe for multiple threads from asynchronous streams. It also shows (1) *let-over-lambda* (LOL): closing over mutable state variables, and (2) transactional mutation, i.e., *atomic updates*. LOL is sematically equivalent to data encapsulation in OOP, and transactions are easier to verify than is OOP with locks and mutexes.

The following has a defect: we need `initial-count` both to initialize the atom and to initialize the `reduce` call. This defect must be traded off against the generalizable form or *functional type* of the reducible, namely (estimate, measurement) → estimate. We get rid of this defect later.

```
(let [initial-count 0] ; Must use this twice below.
    (reduce
        ; Let-over-lambda (anonymous "object") follows.
        ; "Atom" is a transactional (thread-safe) type in Clojure.
        (let [running-count (atom initial-count)]
            ; That was the "let" of "LOL." Here comes the lambda:
            ; Reducible closure over "running-count."
            (fn [c z] ; Here's the "lambda" of "LOL"
                (swap! running-count inc) ; transactional update
                @running-count))
                ; safe "read" of the atom ~~> new value for c
        initial-count
        zs))
```

```
10
```

Showing all intermediate results:

```
(let [initial-count 0]
    (reductions ; <-- this is the only difference to above
        (let [running-count (atom initial-count)]
            (fn [c z]
                (swap! running-count inc)
                @running-count)) ; ~~> new value for c
        initial-count
        zs))
```

0 1 2 3 4 5 6 7 8 9 10

## 3.2 AVOIDING *REDUCE*

`Reduce` only works in space, not in time. Avoiding `reduce` decouples the statistics code ("business logic") from the space environment ("plumbing"). That spaces environment delivers data from vectors, lists, etc.). We want to be able to switch out an environment that delivers data from space for an environment that delivers data points $z$ from time.

The following is a thread-safe LOL, without `reduce`. We *map* the LOL over a space-sequence in memory to produce exactly the same result as with `reduce`. The mappable LOL does not need an accumulator argument for `count`.

Below, we map *exactly* the same mappable LOL over asynchronous streams.

A subtle defect: the output is still coupled to the computing environment through `print`. We get rid of that, too, below.

```
(dorun ; <-- Discard 'nil's produced by "print."
    (map
        (let [running-count (atom 0)]
            (fn [z] ; <-- one fewer argument
                (swap! running-count inc)
                (print (str @running-count " "))))
        zs))
```

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

# 4 RUNNING MEAN

Consider the following general scheme for recurrence: ***a new statistic is an old statistic plus a correction***.

The *correction* is a *gain* times a *residual*. For running mean, the residual is the difference between the new measurement $z$ and the old mean $x$. The gain is $1/(n+1)$, where $n$ is *count-so-far*. $n$ is a statistic, too, so it is an *old* value, computed and saved before the current observation $z$ arrived.

/The correction therefore depends only on the new input $z$ and on old statistics $x$ and $n$. The correction does not depend on new statistics/.

Mathematically, write the general recurrence idea without subscripts as

$$x \leftarrow x + K\,(z - x)$$

or, with Lamport's notation, wherein new versions of old values get a prime, as an equation

$$x' = x + K\,(z - x)$$

($z$ does not have a prime; it is the only exception to the rule that new versions of old quantities have primes).

Contrast the noisy traditional form, which introduces another variable, the index $n$. This traditional form is objectively more complicated than either of the two above:

$$x_{n+1} = x_n + K(n)\,(z_{n+1} - x_n)$$

```
(dorun
    (map
        (let [running-stats (atom {:count 0, :mean 0})]
            (fn [z]
                (let [{x :mean, n :count} @running-stats
                      n+1 (inc n) ; cool variable name!
                      K   (/ 1.0 n+1)]
                    (swap! running-stats conj
                           [:count n+1]
                           [:mean (+ x (* K (- z x)))])))
```

```
                (println @running-stats)))
        zs))
```

```
{:count 1, :mean -0.178654}
{:count 2, :mean 0.3248255}
{:count 3, :mean 0.2362919}
{:count 4, :mean 0.1741917}
{:count 5, :mean -0.15667464000000003}
{:count 6, :mean -0.18306953333333337}
{:count 7, :mean -0.20331617142857145}
{:count 8, :mean -0.262446275}
{:count 9, :mean -0.21517335555555556}
{:count 10, :mean -0.27947242}
```

The `swap` above calls `conj` on the current contents of the atom `running-stats` and on the rest of the arguments, namely `[:count n+1, :mean ...]`. `conj` is the idiom for "updating" a hashmap, the hashmap in the atom, the hashmap that starts off as `{:count 0, :mean 0}`.

## 4.1 REMOVING OUTPUT COUPLING

Remove `println` from inside the LOL function of $z$. Now the LOL function of $z$ is completely decoupled from its environment. Also, abstract a "factory" method for the LOL, *make-running-stats-mapper*, to clean up the line that does the printing.

### 4.1.1 MAKE-RUNNING-STATS-MAPPER

```
(defn make-running-stats-mapper []
    (let [running-stats (atom {:count 0 :mean 0 :datum 0})]
        (fn [z]
            (let [{x :mean, n :count, _ :datum} @running-stats
                  n+1 (inc n)
                  K   (/ 1.0 n+1)]
                (swap! running-stats conj
                       [:count n+1]
                       [:mean (+ x (* K (- z x)))]
                       [:datum z]))
            @running-stats)))

(clojure.pprint/pprint (map (make-running-stats-mapper) zs))
```

```
({:count 1, :mean -0.178654, :datum -0.178654}
 {:count 2, :mean 0.3248255, :datum 0.828305}
 {:count 3, :mean 0.2362919, :datum 0.0592247}
 {:count 4, :mean 0.1741917, :datum -0.0121089}
 {:count 5, :mean -0.15667464000000003, :datum -1.48014}
 {:count 6, :mean -0.18306953333333337, :datum -0.315044}
 {:count 7, :mean -0.20331617142857145, :datum -0.324796}
 {:count 8, :mean -0.262446275, :datum -0.676357}
 {:count 9, :mean -0.21517335555555556, :datum 0.16301}
 {:count 10, :mean -0.27947242, :datum -0.858164})
```

## 4.2 NUMERICAL CHECK

The last value of the running mean is $-0.279...42$. Check that against an independent calculation.

1. DEFN MEAN

```
(defn mean [zs] (/ (reduce + zs) (count zs)))
(println (mean zs))
```

```
-0.27947242
```

# 5 CORE.ASYNC

For data distributed over time, we'll use Clojure's core.async. Core.async has some subtleties that we analyze below.

```
(require
    '[clojure.core.async
      :refer
      [sliding-buffer dropping-buffer buffer
       <!!, <!, >!, >!!,
       go chan onto-chan close!
       thread alts! alts!! timeout]])
```

## 5.1 SHALLOW TUTORIAL

`https://github.com/clojure/core.async/blob/master/examples/walkthrough.clj`

## 5.2 DEEP TUTORIAL

The asynchronous, singleton `go` thread is loaded with very lightweight *pseudothreads* (my terminology, not standard; most things you will read or see about Clojure.async does not carefully distinguish between threads and pseudothreads, and I think that's not helpful).

Pseudothreads are lightweight state machines that pick up where they left off. It is feasible to have thousands, even millions of them. Pseudothreads don't block, they *park*. *Parking* and *unparking* are very fast. We can write clean code with pseudothreads because our code looks like it's blocked waiting for input or blocked waiting for buffer space. Code with blocking I/O is easy to write and to understand. Code in `go` forms doesn't actually block, just looks like it.

Some details are tricky and definitely not easy to divine from the documentation. Hickey's video from InfoQ 2013 (`https://www.infoq.com/presentations/core-async-clojure`) is more helpful, but you can only appreciate the fine points after you've stumbled a bit. I stumbled over the fact that buffered and unbuffered channels have different synchronization semantics. Syntactically, they look the same, but you cannot, in general, run the same code over an unbuffered channel that works on a buffered channel. Hickey says this, but doesn't nail it to the mast; doesn't emphasize it with an example, as I do here in this deep tutorial. He motivates the entire library with the benefits of first-class queues, but fails to emphasize that, by default, a channel is not a queue but a blocking rendezvous. He does mention it, but one cannot fully appreciate the ramifications from a passing glance.

### 5.2.1 COMMUNICATING BETWEEN THREADS AND PSEUDOTHREADS

Write output to unbuffered channel `c` via `>!` on the asynchronous `go` real-thread and read input from the same channel `c` via `<!!` on the UI/REPL `println` real-thread. We'll see later that writing via `>!!` to an unbuffered channel blocks the UI real-thread, so we can't write before reading unbuffered on the UI/REPL real-thread. However, we can write before reading on a non-blocking pseudothread, and no buffer space is needed.

```
(let [c (chan)]         ;; unbuffered chan
    (go (>! c 42))      ;; parks if no space in chan
    (println (<!! c))   ;; blocks UI/REPL until data on c
    (close! c))         ;; idiom; may be harmless overkill
```

```
42
```

In general, single-bang forms work on `go` pseudothreads, and double-bang forms work on real, heavy-weight, Java threads like the UI/REPL thread behind this notebook. In the rest of this notebook, "thread" means "real thread" and we write "pseudothread" explicitly when that's what we mean.

I don't address thread leakage carefully in this tutorial, mostly because I don't yet understand it well. I may overkill by closing channels redundantly.

#### 5.2.2 CHANNEL VOODOO FIRST

Writing before reading seems very reasonable, but it does not work on unbuffered channels, as we see below. Before going there, however, let's understand more corners of the example above.

The `go` form itself returns a channel:

```
(clojure.repl/doc go)
```

```
-------------------------
clojure.core.async/go
([& body])
Macro
  Asynchronously executes the body, returning immediately to the
  calling thread. Additionally, any visible calls to <!, >! and alt!/alts!
  channel operations within the body will block (if necessary) by
  'parking' the calling thread rather than tying up an OS thread (or
  the only JS thread when in ClojureScript). Upon completion of the
  operation, the body will be resumed.

  Returns a channel which will receive the result of the body when
  completed
```

I believe "the calling thread" above refers to a pseudothread inside the `go` real-thread, but I am not sure because of the ambiguities in the official documentation between "blocking" and "parking" and between "thread" and "well, we don't have a name for them, but Brian calls them 'pseudothreads'."

Is the channel returned by `go` the same channel as `c`?

```
(let [c (chan)]
    (println {:c-channel c})
    (println {:go-channel (go (>! c 42))})
    (println {:c-coughs-up (<!! c)})
    (println {:close-c (close! c)}))
```

```
{:c-channel #object[clojure.core.async.impl.channels.ManyToManyChannel 0x3cf6790b clojure.c
{:go-channel #object[clojure.core.async.impl.channels.ManyToManyChannel 0x319e369a clojure
{:c-coughs-up 42}
{:close-c nil}
```

No, `c` is a different channel from the one returned by `go`. Consult the documentation for `go` once more:

```
(clojure.repl/doc go)
```

```
-------------------------
clojure.core.async/go
([& body])
Macro
  Asynchronously executes the body, returning immediately to the
  calling thread. Additionally, any visible calls to <!, >! and alt!/alts!
  channel operations within the body will block (if necessary) by
```

```
  'parking' the calling thread rather than tying up an OS thread (or
  the only JS thread when in ClojureScript). Upon completion of the
  operation, the body will be resumed.

  Returns a channel which will receive the result of the body when
  completed
```

We should be able to read from the channel returned by `go`; call it `d`:

```
(let [c (chan)
      d (go (>! c 42))] ;; 'let' in Clojure is sequential,
                        ;; like 'let*' in Scheme or Common Lisp,
                        ;; so 'd' has a value, here.
    (println {:c-coughs-up (<!! c),  ;; won't block
              :d-coughs-up (<!! d)}) ;; won't block
    (close! c)
    (close! d))
```

```
{:c-coughs-up 42, :d-coughs-up true}
```

`d`'s coughing up `true` means that the body of the `go`, namely `(>!  c 42)` must have returned `true`, because `d` coughs up "the result of the body when completed." Let's see whether our deduction matches documentation for `>!`:

```
(clojure.repl/doc >!)
```

```
-------------------------
clojure.core.async/>!
([port val])
  puts a val into port. nil values are not allowed. Must be called
  inside a (go ...) block. Will park if no buffer space is available.
  Returns true unless port is already closed.
```

Sure enough. But something important is true and not obvious from this documentation. Writing to `c` inside the `go` block parks the pseudothread because no buffer space is available: `c` was created with a call to `chan` with no arguments, so no buffer space is allocated. Only when reading from `c` does the pseudothread unpark. How? There is no buffer space. Reading on the UI thread manages to short-circuit any need for a buffer and unpark the pseudothread. Such short-circuiting is called a *rendezvous* in the ancient literature of concurrency. Would the pseudothread unpark if we read inside a `go` block and not on the UI thread?

```
(let [c (chan)
      d (go (>! c 42))
      e (go (<! c))]
    (clojure.pprint/pprint {
      :c-channel c, :d-channel d, :e-channel e,
      :e-coughs-up (<!! e),  ;; won't block
      :d-coughs-up (<!! d)}) ;; won't block
    (close! c)
    (close! d)
    (close! e))
```

```
{:c-channel
 #object[clojure.core.async.impl.channels.ManyToManyChannel 0x1646c8b9 "clojure.core.async
 :d-channel
 #object[clojure.core.async.impl.channels.ManyToManyChannel 0x2572d097 "clojure.core.async
 :e-channel
 #object[clojure.core.async.impl.channels.ManyToManyChannel 0x668b86b1 "clojure.core.async
 :e-coughs-up 42,
 :d-coughs-up true}
```

Yes, the pseudothread that parked when 42 is put on `c` via `>!` unparks when 42 is taken off via `<!`. Channel `d` represents the parking step and channel `e` represents the unparking step. All three channels are different.

So now we know how to short-circuit or rendezvous unbuffered channels. In fact, the order of reading and writing (taking and putting) does not matter in the nebulous, asynchronous world of pseudothreads. How Einsteinian is that? The following takes (reads) from `c` on `e` before puting (writing) to `c` on `d`. That's the same as above, only in the opposite order.

```
(let [c (chan)
      e (go (<! c))
      d (go (>! c 42))]
    (clojure.pprint/pprint {
      :c-channel c, :d-channel d, :e-channel e,
      :e-coughs-up (<!! e),  ;; won't block
      :d-coughs-up (<!! d)}) ;; won't block
    (close! c)
    (close! d)
    (close! e))
```

```
{:c-channel
 #object[clojure.core.async.impl.channels.ManyToManyChannel 0x5f6a8425 "clojure.core.async
 :d-channel
 #object[clojure.core.async.impl.channels.ManyToManyChannel 0x4de19338 "clojure.core.async
 :e-channel
 #object[clojure.core.async.impl.channels.ManyToManyChannel 0x56b8f382 "clojure.core.async
 :e-coughs-up 42,
 :d-coughs-up true}
```

#### 5.2.3 PUTS BEFORE TAKES CONSIDERED RISKY

`>!!`, by default, blocks if called too early on an unbuffered real thread. We saw above that parked pseudothreads don't block: you can read and write to channels in `go` blocks in any order. However, that's not true with threads that actually block. The documentation is obscure, though not incorrect, about this fact.

```
(clojure.repl/doc >!!)
```

```
-------------------------
clojure.core.async/>!!
([port val])
  puts a val into port. nil values are not allowed. Will block if no
  buffer space is available. Returns true unless port is already closed.
```

When is "no buffer space available?" It turns out that the default channel constructor makes a channel with no buffer space allocated by default.

```
(clojure.repl/doc chan)
```

```
-------------------------
clojure.core.async/chan
([] [buf-or-n] [buf-or-n xform] [buf-or-n xform ex-handler])
  Creates a channel with an optional buffer, an optional transducer
  (like (map f), (filter p) etc or a composition thereof), and an
  optional exception-handler.  If buf-or-n is a number, will create
  and use a fixed buffer of that size. If a transducer is supplied a
  buffer must be specified. ex-handler must be a fn of one argument -
  if an exception occurs during transformation it will be called with
  the Throwable as an argument, and any non-nil return value will be
  placed in the channel.
```

We can test the blocking-on-unbuffered case as follows. The following code will block at the line `(>!! c 42)`, as you'll find if you uncomment the code (remove `#_` at the beginning) and run it. You'll have to interrupt the Kernel using the "Kernel" menu at the top of the notebook, and you might have to restart the Kernel, but you should try it once.

```
#_(let [c (chan)]
    (>!! c 42)
    (println (<!! c))
    (close! c))
```

The following variation works fine because we made "buffer space" before writing to the channel. The only difference to the above is the 1 argument to the call of `chan`.

```
(let [c (chan 1)]
    (>!! c 42)
    (println (<!! c))
    (close! c))
```

```
42
```

The difference between the semantics of the prior two examples is not subtle: one hangs the kernel and the other does not. However, the difference in the syntax is subtle and easy to miss.

We can read on the asynchronous `go` pool from the buffered channel `c` because the buffered write `(>!! c)` on the UI thread doesn't block:

```
(let [c (chan 1)]
    (>!! c 42)
    (println {:go-channel-coughs-up (<!! (go (<! c)))})
    (close! c))
```

```
{:go-channel-coughs-up 42}
```

1. ORDER DOESN'T MATTER, SOMETIMES

   We can do things backwards, reading before writing, even without a buffer. Read from channel `(<! c)` on the async `go` thread "before" writing to `(>!! c 42)` on the REPL / UI thread. "Before," here, of course, means syntactically or lexically "before," not temporally.

   ```
   (let [c (chan) ;; NO BUFFER!
         d (go (<! c)) ;; park a pseudothread to read c
         e (>!! c 42)] ;; blocking write unparks c's pseudothread
       (println {:c-hangs '(<!! c),
                 :d-coughs-up (<!! d),
                 :what's-e    e})
       (close! c) (close! d))
   ```

   ```
   {:c-hangs (<!! c), :d-coughs-up 42, :what's-e true}
   ```

   Why did `>!!` produce `true`? Look at docs again:

   ```
   (clojure.repl/doc >!!)
   ```

   ```
   -------------------------
   clojure.core.async/>!!
   ([port val])
     puts a val into port. nil values are not allowed. Will block if no
     buffer space is available. Returns true unless port is already closed.
   ```

Ok, now I fault the documentation. `>!!` will block if there is no buffer space available *and* if there is no *rendezvous* available, that is, no pseudothread parked waiting for `<!`. I have an open question in the Google group for Clojure about this issue with the documentation.

To get the value written in into `c`, we must read `d`. If we tried to read it from `c`, we would block forever because `>!!` blocks when there is no buffer space, and `c` never has buffer space. We get the value out of the `go` nebula by short-circuiting the buffer, by a rendezvous, as explained above.

`e`'s being true means that `c` wasn't closed. `(>!! c 42)` should hang.

```
(let [c (chan) ;; NO BUFFER!
      d (go (<! c)) ;; park a pseudothread to read c
      e (>!! c 42)  ;; blocking write unparks c's pseudothread
      f '(hangs (>!! c 43))] ;; is `c` closed?
    (println {:c-coughs-up '(hangs (<!! c)),
              :d-coughs-up (<!! d),
              :what's-e    e,
              :what's-f    f})
    (close! c) (close! d))
```

```
{:c-coughs-up (hangs (<!! c)), :d-coughs-up 42, :what's-e true, :what's-f (hangs (>!!
```

StackOverflow reveals a way to find out whether a channel is closed by peeking under the covers (`https://stackoverflow.com/questions/24912971`):

```
(let [c (chan) ;; NO BUFFER!
      d (go (<! c)) ;; park a pseudothread to read c
      e (>!! c 42)  ;; blocking write unparks c's pseudothread
      f (clojure.core.async.impl.protocols/closed? c)]
    (println {:c-coughs-up '(hangs (<!! c)),
              :d-coughs-up (<!! d),
              :c-is-open-at-e?  e,
              :c-is-open-at-f?  f})
    (close! c) (close! d))
```

```
{:c-coughs-up (hangs (<!! c)), :d-coughs-up 42, :c-is-open-at-e? true, :c-is-open-at-f
```

2. ORDER DOES MATTER, SOMETIMES

   Order does matter this time: Writing blocks the UI thread without a buffer and no parked read (rendezvous) in the `go` nebula beforehand. I hope you can predict that the following will block even before you run it. To be sure, run it, but you'll have to interrupt the kernel as before.

```
#_(let [c (chan)
      e (>!! c 42) ;; blocks forever
      d (go (<! c))]
    (println {:c-coughs-up '(this will hang (<!! c)),
              :d-coughs-up (<!! d),
              :what's-e    e})
    (close! c) (close! d))
```

### 5.2.4 TIMEOUTS: DON'T BLOCK FOREVER

In all cases, blocking calls like `>!!` to unbuffered channels without timeout must appear *last* on the UI, non-`go`, thread, and then only if there is some parked pseudothread that's waiting to read the channel by short-circuit (rendezvous). If we block too early, we won't get to the line that launches the async `go` nebula and parks the short-cicuitable pseudothread—parks the rendezvous.

The UI thread won't block forever if we add a timeout. `alts!!` is a way to do that. The documentation and examples are difficult, but, loosely quoting (emphasis and edits are mine, major ones in square brackets):

```
(alts!!  ports & {:as opts})
```

This destructures all keyword options into `opts`. We don't need `opts` or the `:as` keyword below.

> Completes at most one of several channel operations. [/Not for use inside a (go ...) block./] **ports is a vector of channel endpoints**, [A channel endpoint is] either a channel to take from or a vector of `[channel-to-put-to val-to-put]` pairs, in any combination. Takes will be made as if by `<!!`, and puts will be made as if by `>!!`. If more than one port operation is ready, a non-deterministic choice will be made unless the `:priority` option is true. If no operation is ready and a `:default` value is supplied, [=default-val :default=] will be returned, otherwise `alts!!` will [/block/ xxxxpark ?] until the first operation to become ready completes. **Returns `[val port]` of the completed operation**, where `val` is the value taken for takes, and a boolean (`true` unless already closed, as per `put!`) for puts. `opts` are passed as `:key val` ... Supported options: `:default val` - the value to use if none of the operations are immediately ready `:priority true` - (default `nil`) when `true`, the operations will be tried in order. Note: there is no guarantee that the port exps or val exprs will be used, nor in what order should they be, so they should not be depended upon for side effects.

`(alts!!  ...)` returns a `[val port]` 2-vector.

`(second (alts!!  ...))` is a wrapper of channel `c` We can't write to the resulting `timeout` channel because we didn't give it a name.

That's a lot of stuff, but we can divine an idiom: pair a channel `c` that *might* block with a fresh `timeout` channel in an `alts!!`. At most one will complete. If `c` blocks, the `timeout` will cough up. If `c` coughs up before the `timeout` expires, the `timeout` quietly dies (question, is it closed? Will it be left open and leak?)

For a first example, let's make a buffered thread that won't block and pair it with a long timeout. You will see that it's OK to write 43 into this channel (the `[c 43]` term is an implied write; that's clear from the documentation). `c` won't block because it's buffered, it returns immediately, long before the `timeout` could expire.

```
(let [c (chan 1)
      a (alts!! ; outputs a [val port] pair; throw away the val
                ; here are the two channels for `alts!!`
         [[c 43] (timeout 2500)])]
    (clojure.pprint/pprint {:c c, :a a})
    (let [d (go (<! c))]
        (println {:d-returns (<!! d)}))
    (close! c))
```

```
{:c
 #object[clojure.core.async.impl.channels.ManyToManyChannel 0x7fde4694 "clojure.core.async
 :a
 [true
  #object[clojure.core.async.impl.channels.ManyToManyChannel 0x7fde4694 "clojure.core.async
{:d-returns 43}
```

But, if we take away the buffer, the `timeout` channel wins. The only difference to the above is that instead of creating `c` via `(chan 1)`, that is, with a buffer of length 1, we create it with no buffer (and we quoted out the blocking read of `d` with a tick mark).

```
(let [c (chan)
      a (alts!! ; outputs a [val port] pair; throw away the val
                ; here are the two channels for `alts!!`
         [[c 43] (timeout 2500)])]
```

```
    (clojure.pprint/pprint {:c c, :a a})
    (let [d (go (<! c))]
        (println {:d-is d})
        '(println {:d-returns (<!! d)})) ;; blocks
    (close! c))
```

```
{:c
 #object[clojure.core.async.impl.channels.ManyToManyChannel 0x45064aec "clojure.core.async
 :a
 [nil
  #object[clojure.core.async.impl.channels.ManyToManyChannel 0x7d133eee "clojure.core.asyn
{:d-is #object[clojure.core.async.impl.channels.ManyToManyChannel 0x11b15f44 clojure.core.a
```

# 6 ASYNC DATA STREAMS

The following writes at random times (`>!`) to a parking channel `echo-chan` on an async `go` fast pseudothread. The UI thread block-reads (`<!!`) some data from `echo-chan`. The UI thread leaves values in the channel and thus leaks the channel according to the documentation for `close!` here `https://clojure.github.io/core.async/api-index.html#C`. To prevent the leak permanently, we close the channel explicitly.

```
(def echo-chan (chan))

(doseq   [z zs] (go (Thread/sleep (rand 100)) (>! echo-chan z)))
(dotimes [_ 3] (println (<!! echo-chan)))

(println {:echo-chan-closed?
          (clojure.core.async.impl.protocols/closed? echo-chan)})
(close! echo-chan)
(println {:echo-chan-closed?
          (clojure.core.async.impl.protocols/closed? echo-chan)})
```

```
0.0592247
-1.48014
-0.315044
{:echo-chan-closed? false}
{:echo-chan-closed? true}
```

We can chain channels, again with leaks that we explicitly close. Also, we must not `>!` (send) a nil to `repl-chan`, and `<!` can produce nil from `echo-chan` after the timeout and we close `echo-chan`.

```
(clojure.repl/doc <!)
```

```
-------------------------
clojure.core.async/<!
([port])
  takes a val from port. Must be called inside a (go ...) block. Will
  return nil if closed. Will park if nothing is available.
```

Every time you run the block of code below, you will probably get a different result, by design.

```
(def echo-chan (chan))
(def repl-chan (chan))

;; >! chokes on nulls. <! echo-chan can cough up nil if we time out
;; and close the channel. The following line will throw an exception
```

```
;; unless we don’t close the channel at the end of this code-block.

;; (dotimes [_ 10] (go (>! repl-chan (<! echo-chan))))

;; Instead of throwing an exception, just put a random character
;; like \? down the pipe after the echo-chan is closed:

(dotimes [_ 10] (go (>! repl-chan (or (<! echo-chan) \?))))

(doseq   [z zs] (go (Thread/sleep (rand 100)) (>! echo-chan z)))

(dotimes [_ 3]
    (println (<!! (second (alts!! [repl-chan
                                   (timeout 500)])))))

;; Alternatively, we can avoid the exception by NOT closing echo-chan.
;; Not closing echo chan will leak it, and that’s a lousy idea.

(close! echo-chan)

(close! repl-chan)
```

```
-0.315044
-0.0121089
0.828305
```

Reading from `echo-chan` may hang the UI thread because the UI thread races the internal `go` thread that reads `echo-chan`, but the timeout trick works here as above.

```
(def echo-chan (chan))
(def repl-chan (chan))

(dotimes [_ 10] (go (>! repl-chan (or (<! echo-chan) \?))))
(doseq   [z zs] (go (Thread/sleep (rand 100)) (>! echo-chan z)))
(dotimes [_ 3]
    (println (<!! (second (alts!! [echo-chan
                                   (timeout 500)])))))

(close! echo-chan)
(close! repl-chan)
```

```
nil
nil
nil
```

`println` on a `go` pseudoprocess works if we wait long enough. This, of course, is bad practice or “code smell.”

```
(def echo-chan (chan))

(doseq   [z zs] (go (Thread/sleep (rand 100)) (>! echo-chan z)))
(dotimes [_ 3]  (go (println (<! echo-chan))))

(Thread/sleep 500) ; no visible output if you remove this line.
(close! echo-chan)
```

```
-0.676357
-0.0121089
-0.178654
```

## 6.1 ASYNC RUNNING MEAN

### 6.1.1 DEFN ASYNC-RANDOMIZED-SCAN

We want `running-stats` called at random times and with data in random order. A *transducer*, `(map mapper)`, lets us collect items off the buffer. The size of the buffer does not matter, but we must specify it. Notice that the side-effector `effector` is passed in, so `async-randomized-scan` remains decoupled from its environment.

In this style of programming, the asynchronous stream might sometimes be called a *functor*, which is anything that's mappable, anything you can `map` over.

```
(defn async-randomized-scan [zs mapper effector]
    (let [transducer (map mapper)
          ; give buffer length if there is a transducer
          echo-chan (chan (buffer 1) transducer)]
        (doseq [z zs]
            (go (Thread/sleep (rand 100)) (>! echo-chan z)))
        (dotimes [_ (count zs)] (effector (<!! echo-chan)))
        (close! echo-chan)))

(async-randomized-scan zs (make-running-stats-mapper) println)
```

```
{:count 1, :mean 0.0592247, :datum 0.0592247}
{:count 2, :mean -0.05971465, :datum -0.178654}
{:count 3, :mean -0.04384606666666667, :datum -0.0121089}
{:count 4, :mean 0.007867949999999999, :datum 0.16301}
{:count 5, :mean -0.058664839999999996, :datum -0.324796}
{:count 6, :mean -0.2955773666666667, :datum -1.48014}
{:count 7, :mean -0.3759468857142857, :datum -0.858164}
{:count 8, :mean -0.368334025, :datum -0.315044}
{:count 9, :mean -0.23537413333333335, :datum 0.828305}
{:count 10, :mean -0.27947242, :datum -0.676357}
```

We don't need to explicitly say `buffer`, but I prefer to do.

### 6.1.2 DEFN MAKE SOW REAP

The `effector` above just prints to the console. Suppose we want to save the data?

The following is a version of Wolfram's `Sow` and `Reap` that does not include tags. It uses `atom` for an effectful store because a `let` variable like `result` is not a `var` and `alter-var-root` won't work on `(let [result []] ..)`. An atom might be overkill.

`make-sow-reap` returns a message dispatcher in the style of *The Little Schemer*. It responds to namespaced keywords `::sow` and `::reap`. In the case of `::sow`, it returns an `effector` function that `conj`'s its input to the internal result atomically. In the case of `::reap`, it returns the value of the result accumulated so-far.

```
(do (defn make-sow-reap []
        (let [result (atom [])]
            (fn [msg]
                (cond
                    (identical? msg ::sow)
                    (fn [x] (swap! result #(conj % x)))
                    (identical? msg ::reap)
                    @result))))

    (let [accumulator (make-sow-reap)]
        (async-randomized-scan zs
```

```
                                (make-running-stats-mapper)
                                (accumulator ::sow))
        (last (accumulator ::reap)))   )
```

:count 10 :mean -0.27947242 :datum -1.48014

Occasionally, there is some floating-point noise in the very low digits of the mean because async-randomized-scan scrambles the order of the inputs. The mean should always be almost equal to $-0.27947242$.

### 6.1.3 DEFN ASYNC NON RANDOM SCAN

Of course, the `mean` of any permutation of the data `zs` is the same, so the order in which data arrive does not change the final result, except for some occasional floating-point noise as mentioned above.

```
(do (defn async-non-random-scan [zs mapper effector]
        (let [transducer (map mapper)
              echo-chan (chan (buffer 1) transducer)]
            (go (doseq [z zs] (>! echo-chan z)))
            (dotimes [_ (count zs)] (effector (<!! echo-chan)))
            (close! echo-chan)))

    (let [accumulator (make-sow-reap)]
        (async-non-random-scan zs (make-running-stats-mapper)
                                (accumulator ::sow))
        (last (accumulator ::reap)))   )
```

:count 10 :mean -0.27947242 :datum -0.858164

### 6.1.4 DEFN SYNC SCAN: WITH TRANSDUCER

Here is the modern way, with `transduce`, to reduce over a sequence of data, in order. It's equivalent to the non-random async version above. The documentation for transduce writes its parameters as `xform f coll`, and then says

> reduce with a transformation of `f (xf)`. If `init` is not supplied, `(f)` will be called to produce it.

Our `xform` is `transducer`, or `(map mapper)`, and our `f` is `conj`, so this is an idiom for mapping because `(conj)`, with no arguments, returns `[]`, an appropriate `init`.

```
(do (defn sync-scan [zs mapper]
        (let [transducer (map mapper)]
            (transduce transducer conj zs)))

    (last (sync-scan zs (make-running-stats-mapper)))   )
```

:count 10 :mean -0.27947242 :datum -0.858164

We now have complete symmetry between space and time, space represented by the vector `zs` and time represented by values on `echo-chan` in random and in non-random order.

# 7 RUNNING STDDEV

## 7.1 BRUTE-FORCE (SCALAR VERSION)

The definition of variance is the following, for $N > 1$:

$$\frac{1}{N-1}\sum_{i=1}^{N}(z_i - \bar{z}_N)^2$$

The sum is the *sum of squared residuals*. Each residual is the difference between the \$i\$-th datum $z_i$ and the mean $\bar{z}_N$ of all N data in the sample. The outer constant, $1/(N-1)$ is Bessel's correction.

### 7.1.1 DEFN SSR: SUM OF SQUARED RESIDUALS

The following is *brute-force* in the sense that it requires all data up-front so that it can calculate the mean.

```
(do (defn ssr [sequ]
        (let [m (mean sequ)]
            (reduce #(+ %1 (* (- %2 m) (- %2 m)))
                    0 sequ)))
    (ssr zs)   )
```

```
3.5566483654807355
```

### 7.1.2 DEFN VARIANCE

Call `ssr` to compute variance:

```
(do
(defn variance [sequ]
    (let [n (count sequ)]
        (case n
            0 0
            1 (first sequ)
            #_default (/ (ssr sequ) (- n 1.0)))))
(variance zs)   )
```

```
0.3951831517200817
```

## 7.2 DEF Z2S: SMALLER EXAMPLE

Let's do a smaller example:

```
(do (def z2s [55. 89. 144.])
    (variance z2s)   )
```

```
2017.0
```

## 7.3 REALLY DUMB RECURRENCE

Remember our general form for recurrences, $x \leftarrow x + K \times (z - x)$?

We can squeeze running variance into this form in a really dumb way. The following is really dumb because:

1. it requires the whole sequence up front, so it doesn't run in constant memory
2. the intermediate values are meaningless because they refer to the final mean and count, not to the intermediate ones

But, the final value is correct.

```
(do (reductions
        (let [m (mean z2s) ; uh-oh, we refer to _all_ the data ??
              c (count z2s)]
            (fn [var z] (+ var (let [r (- z m)] ; residual
                                    (/ (* r r) (- c 1.0))))))
        0 z2s)   )
```

That was so dumb that we won't bother with a thread-safe, stateful, or asynchronous form.

## 7.4 SCHOOL VARIANCE

For an easy, school-level exercise, prove the following equation:

$$\frac{1}{N-1}\sum_{i=1}^{N}\left(z_i-\bar{z}_N\right)^2=\frac{1}{N-1}\left(\sum_{i=1}^{N}\left(z_i^2\right)-N\,\bar{z}_N^2\right)$$

Instead of the sum of squared residuals, $ssr$, accumulate the sum of squares, $ssq$.

*School variance* is exposed to *catastrophic cancellation* because $ssq$ grows quickly. We fix that defect below.

We see that something is not best with this form because we don't use the old variance to compute the new variance. We do better below.

Of course, the same mapper works synchronously and asynchronously.

## 7.5 DEFN MAKE SCHOOL STATS MAPPER

and test it both synchronously and asynchronously, randomized and not:

```
(defn make-school-stats-mapper []
    (let [running-stats (atom {:count 0, :mean 0,
                               :variance 0, :ssq 0})]
        (fn [z]
            (let [{x :mean, n :count, s :ssq} @running-stats
                  n+1 (inc n)
                  K   (/ 1.0 n+1)
                  r   (- z x)
                  x'  (+ x (* K r)) ;; Isn't prime notation nice?
                  s'  (+ s (* z z))]
                (swap! running-stats conj
                       [:count    n+1]
                       [:mean     x' ]
                       [:ssq      s']
                       [:variance (/ (- s' (* n+1 x' x')) (max 1 n))]))
            @running-stats)))

(clojure.pprint/pprint (sync-scan z2s (make-school-stats-mapper)))

(async-randomized-scan z2s (make-school-stats-mapper) println)

(async-non-random-scan z2s (make-school-stats-mapper) println)
```

```
[{:count 1, :mean 55.0, :variance 0.0, :ssq 3025.0}
 {:count 2, :mean 72.0, :variance 578.0, :ssq 10946.0}
 {:count 3, :mean 96.0, :variance 2017.0, :ssq 31682.0}]
{:count 1, :mean 55.0, :variance 0.0, :ssq 3025.0}
{:count 2, :mean 99.5, :variance 3960.5, :ssq 23761.0}
{:count 3, :mean 96.0, :variance 2017.0, :ssq 31682.0}
{:count 1, :mean 55.0, :variance 0.0, :ssq 3025.0}
```

```
{:count 2, :mean 72.0, :variance 578.0, :ssq 10946.0}
{:count 3, :mean 96.0, :variance 2017.0, :ssq 31682.0}
```

## 7.6 DEFN MAKE RECURRENT STATS MAPPER

We already know the recurrence for the mean:

$$x \leftarrow x + K \cdot (z - x) = x + \frac{1}{n+1}(z - x)$$

We want a recurrence with a similar form for the variance. It takes a little work to prove, but it's still a school-level exercise. $K$ remains $1/(n+1)$, the value needed for the new mean. We could define a pair of gains, one for the mean and one for the variance, but it would be less pretty.

$$v \leftarrow \frac{(n-1)\,v + K\,n\,(z-x)^2}{\max(1, n)}$$

```
(defn make-recurrent-stats-mapper []
    (let [running-stats (atom {:count 0, :mean 0,
                               :variance 0})]
        (fn [z]
            (let [{x :mean, n :count, v :variance} @running-stats
                  n+1 (inc n)
                  K   (/ 1.0 (inc n))
                  r   (- z x)
                  x'  (+ x (* K r))
                  ssr (+ (* (- n 1) v) ; old ssr is (* (- n 1) v)
                         (* K n r r))]
                (swap! running-stats conj
                       [:count    n+1]
                       [:mean     x' ]
                       [:variance (/ ssr  (max 1 n))]))
            @running-stats)))

(async-non-random-scan z2s (make-recurrent-stats-mapper) println)
```

```
{:count 1, :mean 55.0, :variance 0.0}
{:count 2, :mean 72.0, :variance 578.0}
{:count 3, :mean 96.0, :variance 2017.0}
```

## 7.7 DEFN MAKE WELFORD'S STATS MAPPER

The above is equivalent, algebraically and numerically, to Welford's famous recurrence for the sum of squared residuals $S$. In recurrences, we want everything on the right-hand sides of equations or left arrows to be be old, *prior* statistics, except for the new observation / measurement / input $z$. Welford's requires the new, *posterior* mean on the right-hand side, so it's not as elegant as our recurrence above. However, it is easier to remember!

$$S \leftarrow S + (z - x_N)\,(z - x_{N+1}) = S + (z - x)\,(z - (x + K\,(z - x)))$$

```
(do (defn make-welfords-stats-mapper []
        (let [running-stats (atom {:count 0, :mean 0, :variance 0})]
            (fn [z]
                (let [{x :mean, n :count, v :variance} @running-stats
                      n+1 (inc n)
                      K   (/ 1.0 n+1)
                      r   (- z x)
```

```
                         x'   (+ x (* K r))
                         ssr (+ (* (- n 1) v)
                                ;; only difference to recurrent variance:
                                (* (- z x) (- z x'))))]
                       (swap! running-stats conj
                              [:count    n+1]
                              [:mean     x' ]
                              [:variance (/ ssr  (max 1 n))]))
                  @running-stats)))

   (async-non-random-scan
     z2s (make-welfords-stats-mapper) println)   )
```

```
{:count 1, :mean 55.0, :variance 0.0}
{:count 2, :mean 72.0, :variance 578.0}
{:count 3, :mean 96.0, :variance 2017.0}
```

# 8 WINDOWED STATISTICS

Suppose we want running statistics over a history of fixed, finite length. For example, suppose we have $N = 10$ data and we want the statitics in a window of length $w = 3$ behind the current value, inclusively. When the first datum arrives, the window and the total include one datum. The window overhangs the left until the third datum. When the fourth datum arrives, the window contains three data and the total contains four data. After the tenth datum, we may consider three more steps marching the window "off the cliff" to the right. The following figure illustrates (the first row corresponds to $n = 0$, not to $n = 1$):

We won't derive the following formulas, but rather say that they have been vetted at least twice independently (in a C program and in a Mathematica program). The following table shows a unit test that we reproduce. The notation is explained after the table.

Denote prior statistics by plain variables like $m$ and corresponding posteriors by the same variables with primes like $m'$. The posteriors $j$ and $u$ do not have a prime.

| variable | description |
|---|---|
| $n$ | prior count of data points; equals $0$ when considering the first point |
| $z$ | current data point |
| $w$ | fixed, constant, maximum width of window; $w \geqslant 1$ |
| $j$ | posterior number of points left of the window; $j \geqslant 0$ |
| $u$ | posterior number of points including $z$ in the running window; $1 \leqslant u \leqslant w$ |
| $m$ | prior mean of all points, not including $z$ |
| $m'$ | posterior mean of all points including $z$ |
| $m_j$ | prior mean of points left of the window, lagging $w$ behind $m$ |
| $m'_j$ | posterior mean of points left of the window |
| $m'_w$ | posterior mean of points in the window, including the current point $z$ |
| $v$ | prior variance, not including $z$ |
| $v'$ | posterior variance of all points including $z$ |
| $v_j$ | prior variance of points left of the window, lagging $w$ behind $u_n$ |
| $v'_j$ | posterior variance of points left of the window |
| $v'_w$ | posterior variance of points within the window |

The recurrences for $m$, $v$, $m_j$, and $v_j$ have only priors (no primes) on their right-hand sides. The values of $m_w$ and $v_w$ are not recurrences because the non-primed versions do not appear on the right-hand sides of equations 10 and 13. Those equations are simply transformations of the posteriors (values with primes) $m'$, $m'_j$, $v'$, and $v'_j$.

$$
\begin{aligned}
j &= \max(0, n+1-w) \\
u &= n-j+1 \\
m' &= m + \frac{z-m}{n+1} \\
m'_j &= \begin{cases} m_j + \frac{z_j - m_j}{j} & j > 0 \\ 0 & \text{otherwise} \end{cases} \\
m'_w &= \frac{(n+1)\, m' - j\, m'_j}{u} \\
v' &= \frac{(n-1)\, v + \frac{n}{n+1}\, (z-m)^2}{\max(1, n)} \\
v'_j &= \begin{cases} \frac{j-2}{j-1}\, v_j + \frac{1}{j}\, (z_j - m_j)^2 & j > 1 \\ 0 & \text{otherwise} \end{cases} \\
v'_w &= \frac{n\, v' + (n-w)\, v'_j + (n+1)\, m'^2 - j\, m'^2_j - u\, m'_w{}^2}{\max(1, u-1)}
\end{aligned}
$$

Here is sample data we can compare with the unit test above.

## 8.1 DEF Z3S: MORE SAMPLE DATA

```
(def z3s [0.857454,  0.312454,  0.705325,  0.8393630, 1.637810,
          0.699257, -0.340016, -0.213596, -0.0418609, 0.054705])
```

The best algorithm we have found for tracking historical data is to keep a FIFO queue in a Clojure *vector* of length $w$. This is still constant memory because it depends only on the length $w$ of the window, not on the length of the data stream.

### 8.1.1 DEFN PUSH TO BACK

```
(defn push-to-back [item vek]
    (conj (vec (drop 1 vek)) item))
```

## 8.2 DEFN MAKE SLIDING STATS MAPPER

```
(defn make-sliding-stats-mapper [w]
    (let [running-stats (atom {:n 0, :m 0, :v 0,
                               :win (vec (repeat w 0)),
                               :mw 0, :vw 0,
                               :mj 0, :vj 0})]
        (fn [z]
            (let [{:keys [m n v win mj vj]} @running-stats
                  zj   (first win)
                  win' (push-to-back z win)
                  n+1  (double (inc n))
                  n-1  (double (dec n))
                  K    (/ 1.0 n+1)
                  Kv   (* n K)
                  r    (- z m)
                  j    (max 0, (- n+1 w))
                  u    (- n+1 j)
                  m'   (+ m (* K r))
```

```
                  rj   (- zj mj)
                  mj'  (if (> j 0), (+ mj (/ rj j)), 0)
                  mw'  (/ (- (* n+1 m') (* j mj')) u)
                  v'   (/  (+ (* n-1 v) (* Kv r r))
                           (max 1 n))
                  vj'  (if (> j 1)
                           (let [j21 (/ (- j 2.0)
                                        (- j 1.0))]
                               (+ (* j21 vj)
                                  (/ (* rj rj) j)))
                           0)
                  vw'  (let [t1 (- (* n v')
                                   (* (- n w) vj'))
                             t2 (- (* n+1 m' m')
                                   (* j mj' mj'))
                             t3 (- (* u mw' mw'))]
                           (/  (+ t1 t2 t3)
                               (max 1 (- u 1))))
                  ]
                (swap! running-stats conj
                       [:n    n+1 ]
                       [:m    m'  ]
                       [:v    v'  ]
                       [:mj   mj' ]
                       [:vj   vj' ]
                       [:mw   mw' ]
                       [:vw   vw' ]
                       [:win  win']))
            @running-stats)))

(clojure.pprint/print-table
    [:n :mw :vw]
    (sync-scan z3s (make-sliding-stats-mapper 3)))
```

| :n | :mw | :vw |
|---:|---:|---:|
| 1.0 | 0.857454 | 0.0 |
| 2.0 | 0.584954 | 0.14851250000000005 |
| 3.0 | 0.6250776666666666 | 0.07908597588033339 |
| 4.0 | 0.6190473333333332 | 0.07499115039433346 |
| 5.0 | 1.0608326666666668 | 0.2541686787463333 |
| 6.0 | 1.05881 | 0.25633817280899995 |
| 7.0 | 0.6656836666666668 | 0.9787942981023336 |
| 8.0 | 0.04854833333333334 | 0.3215618307563336 |
| 9.0 | -0.19849096666666663 | 0.022395237438003604 |
| 10.0 | -0.06691730000000007 | 0.01846722403596973 |

… passing the unit test.

# 9 KALMAN FILTER

## 9.1 BASIC LINEAR ALGEBRA

Go for high performance with CUDA or Intel KML later.

Add the following lines to `project.clj` in the directory that contains this org file:

### 9.1.1 TODO: FULLY LITERATE: TANGLE PROJECT.CLJ

```
[net.mikera/core.matrix "0.62.0"]
[net.mikera/vectorz-clj "0.48.0"]
[org.clojure/algo.generic "0.1.2"]
```

Smoke test:

```
(require '[clojure.core.matrix :as ccm])
(ccm/set-current-implementation :vectorz)
```

```
(ccm/shape
    (ccm/array [[1 2 3]
                [1 3 8]
                [2 7 4]]))
```

3 3

Bits and pieces we will need:

```
(ccm/transpose
    (ccm/array [[1 2 3]
                [1 3 8]
                [2 7 4]]))
```

```
#vectorz/matrix [[1.0,1.0,2.0],
[2.0,3.0,7.0],
[3.0,8.0,4.0]]
```

`mmul` is multiadic (takes more than two arguments). This is possible because matrix multiplication is associative.

```
(let [A (ccm/array [[1 2 3]
                    [1 3 8]
                    [2 7 4]])]
    (ccm/mmul (ccm/transpose A) A (ccm/inverse A)))
```

```
#vectorz/matrix [[1.000000000000003,1.0,2.0000000000000004],
[2.0000000000000093,3.000000000000001,6.999999999999998],
[3.000000000000006,8.0,3.999999999999999]]
```

### 9.1.2 DEFN LINSPACE

```
(defn linspace
  "A sequence of $n$ equally spaced points in the doubly closed
 interval $[a,b]$, that is, inclusive of both ends."
  [a b n]
  (let [d (/ (- b a) (dec n))]
    (map (fn [x] (+ a (* x d))) (range n))))
```

```
(clojure.pprint/pprint (linspace 2 3. 3))
```

```
(2.0 2.5 3.0)
```

## 9.2 DEFN SYMMETRIC PART

```
(do (defn symmetric-part [M]
        (ccm/div (ccm/add M (ccm/transpose M)) 2.0))
    (symmetric-part [[1 2 3]
                     [1 3 8]
                     [2 7 4]])   )
```

$$\begin{matrix} 1.0 & 1.5 & 2.5 \\ 1.5 & 3.0 & 7.5 \\ 2.5 & 7.5 & 4.0 \end{matrix}$$

## 9.3 DEFN ANTI-SYMMETRIC PART

```
(do (defn anti-symmetric-part [M]
        (ccm/div (ccm/sub M (ccm/transpose M)) 2.0))
    (anti-symmetric-part [[1 2 3]
                          [1 3 8]
                          [2 7 4]])   )
```

$$\begin{matrix} 0.0 & 0.5 & 0.5 \\ -0.5 & 0.0 & 0.5 \\ -0.5 & -0.5 & 0.0 \end{matrix}$$

```
(let [M [[1 2 3]
         [1 3 8]
         [2 7 4]]]
    (ccm/sub (ccm/add (symmetric-part M)
                      (anti-symmetric-part M))
             M))
```

$$\begin{matrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{matrix}$$

### 9.3.1 DEFN MATRIX ALMOST =

```
(require '[clojure.algo.generic.math-functions :as gmf])
```

The following isn't the best solution: neither relative nor absolute differences are robust. Units in Last Place (ULP) are a better criterion, however, this will unblock us for now.

```
(do  (defn matrix-almost=
        ([m1 m2 eps]
         "Checks for near equality against a given absolute difference."
        (mapv (fn [row1 row2]
                  (mapv (fn [e1 e2] (gmf/approx= e1 e2 eps))
                        row1 row2))
             m1 m2))
        ([m1 m2]
         "Checks for near equality against a default absolute difference of 1.0e-9"
         (matrix-almost= m1 m2 1.0e-9)))

    (let [M [[1 2 3]
             [1 3 8]
             [2 7 4]]]
        (matrix-almost= (ccm/add (symmetric-part M)
                                 (anti-symmetric-part M))
                        M))   )
```

| | | |
|---|---|---|
| true | true | true |
| true | true | true |
| true | true | true |

### 9.3.2 DEFN SIMILARITY TRANSFORM

```
(defn similarity-transform [A M]
    (ccm/mmul A M (ccm/transpose A)))
```

### 9.3.3 VECTORS, ROW VECTORS, COLUMN VECTORS

The library (like many others) is loose about matrices times vectors.

```
(ccm/mmul
    (ccm/matrix [[1 2 3]
                 [1 3 8]
                 [2 7 4]])
    (ccm/array [22 23 42]))
```

```
#vectorz/vector [194.0,427.0,373.0]
```

Pedantically, a matrix should only be allowed to left-multiply a column vector, i.e., a $1 \times 3$ matrix. The Clojure library handles this case.

```
(ccm/mmul
    (ccm/matrix [[1 2 3]
                 [1 3 8]
                 [2 7 4]])
    (ccm/array [[22] [23] [42]]))
```

```
#vectorz/matrix [[194.0],
[427.0],
[373.0]]
```

Non-pedantic multiplication of a vector on the right by a matrix:

```
(ccm/mmul
    (ccm/array [22 23 42])
    (ccm/matrix [[1 2 3]
                 [1 3 8]
                 [2 7 4]]))
```

```
#vectorz/vector [129.0,407.0,418.0]
```

Pedantic multiplication of a row vector on the right by a matrix:

```
(ccm/mmul
    (ccm/array [[22 23 42]])
    (ccm/matrix [[1 2 3]
                 [1 3 8]
                 [2 7 4]]))
```

```
#vectorz/matrix [[129.0,407.0,418.0]]
```

### 9.3.4 SOLVING INSTEAD OF INVERTING

Textbooks will tell you that, if you have $\mathbf{A}\mathbf{x} = \mathbf{b}$ and you want $\mathbf{x}$, you should compute $\mathbf{A}^{-1}\mathbf{b}$. Don't do this; the inverse is numerically risky and almost never needed:

```
(ccm/mmul
    (ccm/inverse
        (ccm/array [[1 2 3]
                    [1 3 8]
                    [2 7 4]]))
    (ccm/array [22 23 42]))
```

```
#vectorz/vector [22.05882352941177,-0.4705882352941142,0.2941176470588234]
```

Instead, use a linear solver. Almost everywhere that you see $\mathbf{A}^{-1}\mathbf{b}$, visualize solve$(\mathbf{A}, \mathbf{b})$. You will get a more stable answer. Notice the difference in the low-significance digits below. The following is a more reliable answer:

```
(require '[clojure.core.matrix.linear :as ccml])
```

```
(ccml/solve
    (ccm/array [[1 2 3]
                [1 3 8]
                [2 7 4]])
    (ccm/array [22 23 42]))
```

```
(ccml/solve
    (ccm/matrix [[1 2 3]
                 [1 3 8]
                 [2 7 4]])
    (ccm/matrix [22 23 42]))
```

```
#vectorz/vector [22.058823529411764,-0.4705882352941176,0.2941176470588236]
```

```
(ccm/shape (ccm/matrix [[22] [23] [42]]))
```

3 1

### 9.3.5 DEFN SOLVE MATRIX

We need `solve` to work on matrices:

```
(defn solve-matrix
  "The 'solve' routine in clojure.core.matrix only works on Matrix times Vector.
  We need it to work on Matrix times Matrix. The equation to solve is

  Ann * Xnm = Bnm

  Think of the right-hand side matrix Bnm as a sequence of columns. Iterate over
  its transpose, treating each column as a row, then converting that row to a
  vector, to get the transpose of the solution X."
  [Ann Bnm]
  (ccm/transpose (mapv (partial ccml/solve Ann) (ccm/transpose Bnm))))
```

```
(solve-matrix
    (ccm/matrix [[1 2 3]
                 [1 3 8]
                 [2 7 4]])
    (ccm/matrix [[22] [23] [42]]   ))
```

$$\begin{matrix} 22.058823529411764 \\ -0.4705882352941176 \\ 0.2941176470588236 \end{matrix}$$

```
(solve-matrix
    (ccm/matrix [[1 2 3]
                 [1 3 8]
                 [2 7 4]])
    (ccm/matrix [[22 44]
                 [23 46]
                 [42 84]]))
```

$$\begin{matrix} 22.058823529411764 & 44.11764705882353 \\ -0.4705882352941176 & -0.9411764705882352 \\ 0.2941176470588236 & 0.5882352941176472 \end{matrix}$$

## 9.4 DEFN KALMAN UPDATE: GENERAL EXTENDED KALMAN FILTER

Use Clojure's destructuring to write the Kalman filter as a binary function. See `http://vixra.org/abs/1606.0348`

`xn1` denotes a vector $x$ with dimension $n \times 1$, that is, a column vector of height $n$. `Pnn` denotes a covariance matrix of dimension $n \times n$, and So on.

The math is as follows (notice step 6 has the same form as all earlier statistics calculations in this document):

Letting inputs:

- $\mathbf{x}_{n,1}$ be the current, best estimate of the \$n\$-dimensional state of a system
- $\mathbf{P}_{n,n}$ be the current, best estimate of the $n \times n$ covariance of state $\mathbf{x}_{n,1}$
- $\mathbf{z}_{m,1}$ be the current, \$m\$-dimensional observation
- $\mathbf{H}_{m,n}$ be linearized observation model to be inverted: $\mathbf{z}_{m,1} = \mathbf{H}_{m,n} \cdot \mathbf{x}_{n,1}$
- $\mathbf{A}_{n,n}$ be linearized dynamics
- $\mathbf{Q}_{n,n}$ be process noise (covariance) accounting for uncertainty in $\mathbf{A}_{n,n}$
- $\mathbf{R}_{m,m}$ be observation noise (covariance) accounting for uncertainty in $\mathbf{z}_{m,1}$

and intermediates and outputs:

- $\mathbf{x}'_{n,1}$ (intermediate; *update*) be the estimate of the state after enduring one time step of linearized dynamics
- $\mathbf{x}''_{n,1}$ (output; *prediction*) be the estimate of the state after dynamics and after information from the observation $\mathbf{z}_{m,1}$
- $\mathbf{P}'_{n,n}$ (intermediate; *update*) be the current, best estimate of the $n \times n$ covariance of state $\mathbf{x}_{n,1}$ after dynamics
- $\mathbf{P}''_{n,n}$ (output; *prediction*) be the current, best estimate of the $n \times n$ covariance of state $\mathbf{x}_{n,1}$ after dynamics and oservation $\mathbf{z}_{m,1}$

The steps are:

1. *Update state estimate*: $\mathbf{x}'_{n,1} = \mathbf{A}_{n,n}\ \mathbf{x}_{n,1}$
2. *Update state covariance*: $\mathbf{P}'_{n,n} = \mathbf{Q}_{n,n} + (\mathbf{A}_{n,n}\ \mathbf{P}_{n,n}\ \mathbf{A}^{\intercal}_{n,n})$
3. *Covariance-update scaling matrix*: $\mathbf{D}_{m,m} = \mathbf{R}_{m,m} + \left(\mathbf{H}_{m,n}\ \mathbf{P}'_{n,n}\ \mathbf{H}^{\intercal}_{m,n}\right)$

4. *Kalman gain*: $\mathbf{K}_{n,m} = \mathbf{P}_{n,n}\ \mathbf{H}^{\intercal}_{m,n}\ \mathbf{D}^{-1}_{m,m}$
   (a) written as $\mathbf{K}^{\intercal}_{n,m} = \text{solve}\left(\mathbf{D}^{\intercal}_{m,m}, \mathbf{H}_{m,n}\ \mathbf{P}^{\intercal}_{n,n}\right)$
5. *Innovation: predicted observation residual*: $\mathbf{r}_{m,1} = \mathbf{z}_{m,1} - \mathbf{H}_{m,n}\ \mathbf{x}'_{n,1}$
6. *State prediction*: $\mathbf{x}''_{n,1} = \mathbf{x}'_{n,1} + \mathbf{K}_{n,m}\ \mathbf{r}_{m,1}$
7. *Covariance reduction matrix*: $\mathbf{L}_{n,n} = \mathbf{I}_{n,n} - \mathbf{K}_{n,m}\ \mathbf{H}_{m,n}$
8. *Covariance prediction*: $\mathbf{P}''_{n,n} = \mathbf{L}_{n,n}\ \mathbf{P}'_{n,n}$

```
(defn kalman-update [{:keys [xn1 Pnn]} {:keys [zm1 Hmn Ann Qnn Rmm]}]
  (let [x'n1   (ccm/mmul Ann xn1)                        ; Predict state
        P'nn   (ccm/add
                Qnn (similarity-transform Ann Pnn))  ; Predict covariance
        Dmm    (ccm/add
                Rmm (similarity-transform Hmn P'nn)) ; Gain precursor
        DTmm   (ccm/transpose Dmm)                   ; Support for "solve"
        HP'Tmn (ccm/mmul Hmn (ccm/transpose P'nn))   ; Support for "solve"
        ; Eqn 3 of http://vixra.org/abs/1606.0328:
        KTmn   (solve-matrix DTmm HP'Tmn)
        Knm    (ccm/transpose KTmn)                  ; Kalman gain
        ; innovation = predicted obn residual
        rm1    (ccm/sub zm1  (ccm/mmul Hmn x'n1))
        x''n1  (ccm/add x'n1 (ccm/mmul Knm rm1))     ; final corrected estimate
        n      (ccm/dimension-count xn1 0)
        ; new covariance ? catastrophic cancellation ?
        Lnn    (ccm/sub (ccm/identity-matrix n)
                        (ccm/mmul Knm Hmn))
        P''nn  (ccm/mmul Lnn P'nn)]                  ; New covariance

      {:xn1 x''n1, :Pnn P''nn}))
```

### 9.4.1 UNIT TEST

Let the measurement model be a cubic:

```
(defn Hmn-t [t]
  (ccm/matrix [[(* t t t) (* t t) t 1]]))
```

Ground truth state, constant with time in this unit test:

```
(def true-x
    (ccm/array [-5 -4 9 -3]))
```

```
(require '[clojure.core.matrix.random :as ccmr])
```

```
(defn fake [n]
  (let [times    (range -2.0 2.0 (/ 2.0 n))
        Hmns     (mapv Hmn-t times)
        true-zs  (mapv #(ccm/mmul % true-x) Hmns)
        zm1s     (mapv #(ccm/add
                         % (ccm/array
                            [[(ccmr/rand-gaussian)]]))
                       true-zs)]
    {:times times, :Hmns Hmns, :true-zs true-zs, :zm1s zm1s}))
```

```
(def test-data (fake 7))
```

A state cluster is a vector of $\mathbf{x}$ and $\mathbf{P}$:

```
(def state-cluster-prior
  {:xn1 (ccm/array [[0.0] [0.0] [0.0] [0.0]])
   :Pnn (ccm/mul 1000.0 (ccm/identity-matrix 4))})
```

An obn-cluster is a vector of $\mathbf{z}$, $\mathbf{H}$, $\mathbf{A}$, $\mathbf{Q}$, and $\mathbf{R}$. *Obn* is short for *observation*.

```
(def obn-clusters
  (let [c (count (:times test-data))]
    (mapv (fn [zm1 Hmn Ann Qnn Rmm]
             {:zm1 zm1, :Hmn Hmn, :Ann Ann, :Qnn Qnn, :Rmm Rmm})
           (:zm1s test-data)
           (:Hmns test-data)
           (repeat c (ccm/identity-matrix 4))
           (repeat c (ccm/zero-matrix 4 4))
           (repeat c (ccm/identity-matrix 1))
           )))
```

```
(clojure.pprint/pprint (reduce kalman-update state-cluster-prior obn-clusters))
```

```
{:xn1 #vectorz/matrix [[-4.951034346887338],
[-4.385574755471751],
[8.535893780887523],
[-2.872500567323339]],
 :Pnn
 #vectorz/matrix [[0.03208215055213958,-5.478256737134757E-15,-0.0874691388122202,-8.77076
[-2.3568386825489895E-15,0.03637145347999561,-5.2632377622874316E-14,-0.05541947257604415],
[-0.08746913881223455,-2.570860191397628E-14,0.2822249372573019,-1.1334683192032458E-14],
[4.6455894686658894E-15,-0.05541947257607027,-6.734196533741965E-15,0.15110531309503664]]}
```

Notice how close the estimate $x_{n\times 1}$ is to the ground truth, $[-5, -4, 9, -3]$ for $\mathbf{x}$. A chi-squared test would be appropriate to complete the verification (TODO).

## 9.5 DEFN MAKE-KALMAN-MAPPER

Just as we did before, we can convert a *foldable* into a *mappable* transducer and bang on an asynchronous stream of data. This only needs error handling to be deployable at scale. Not to minimize error handling: it's a big but separable engineering task.

```
(do (defn make-kalman-mapper [{:keys [xn1 Pnn]}]
        ;; let-over-lambda (LOL); here are the Bayesian priors
        (let [estimate-and-covariance (atom {:xn1 xn1, ;; prior-estimate
                                             :Pnn Pnn, ;; prior-covariance
                                             })]
            ;; here is the mapper (mappable)
            (fn [{:keys [zm1 Hmn Ann Qnn Rmm]}]
                (let [{xn1 :xn1, Pnn :Pnn} @estimate-and-covariance]
                    (let [ ;; out-dented so we don't go crazy reading it
        x'n1   (ccm/mmul Ann xn1)                    ; Predict state
        P'nn   (ccm/add
                Qnn (similarity-transform Ann Pnn))  ; Predict covariance
        Dmm    (ccm/add
                Rmm (similarity-transform Hmn P'nn)) ; Gain precursor
        DTmm   (ccm/transpose Dmm)                   ; Support for "solve"
```

```
        HP'Tmn (ccm/mmul Hmn (ccm/transpose P'nn))   ; Support for "solve"
        ; Eqn 3 of http://vixra.org/abs/1606.0328
        KTmn   (solve-matrix DTmm HP'Tmn)
        Knm    (ccm/transpose KTmn)                   ; Kalman gain
        ; innovation = predicted obn residual
        rm1    (ccm/sub zm1  (ccm/mmul Hmn x'n1))
        x''n1  (ccm/add x'n1 (ccm/mmul Knm rm1))     ; final corrected estimate
        n      (ccm/dimension-count xn1 0)
        ; new covariance ? catastrophic cancellation ?
        Lnn    (ccm/sub (ccm/identity-matrix n)
                        (ccm/mmul Knm Hmn))
        P''nn  (ccm/mmul Lnn P'nn)]
                        (swap! estimate-and-covariance conj
                               [:xn1 x''n1]
                               [:Pnn P''nn])   )   )
                @estimate-and-covariance)   ))

    ;; The following line maps over a fixed sequence in memory
    #_(clojure.pprint/pprint (last
                               (map (make-kalman-mapper state-cluster-prior)
                               obn-clusters)))

    #_(async-randomized-scan obn-clusters
                            (make-kalman-mapper state-cluster-prior)
                            clojure.pprint/pprint)

    (let [accumulator (make-sow-reap)]
        (async-randomized-scan obn-clusters
                               (make-kalman-mapper state-cluster-prior)
                               (accumulator ::sow))
        (last (accumulator ::reap)))   )
```

```
'(:xn1 #vectorz/matrix ((-4.951034346887342)
(-4.385574755471752)
(8.535893780887571)
(-2.872500567323389))  :Pnn #vectorz/matrix ((0.03208215055213757 -1.8726339923169633E-15 -
(-2.447694824603275E-15 0.03637145347997802 -4.1104272763270444E-15 -0.05541947257607147)
(-0.08746913881222558 -6.7480743215497796E-15 0.28222493725723435 3.563122019656362E-15)
(-1.1102230246251565E-16 -0.0554194725760703 4.808653475407709E-15 0.15110531309504036)))
```

# 10 OZ FOR VISUALIZATION

From `https://github.com/metasoarous/oz/blob/master/examples/clojupyter-example.ipynb`

```
(require '[clojupyter.misc.helper :as helper])
(helper/add-dependencies '[metasoarous/oz "1.6.0-alpha2"])
(require '[oz.notebook.clojupyter :as oz])
```

## 10.1 DEFN PLAY DATA

```
(do (defn play-data [& names]
      (for [n names
            i (range 20)]
        {:time i :item n :quantity (+ (Math/pow (* i (count n)) 0.8) (rand-int (count n)))
```

```
    (def stacked-bar
      {:data {:values (play-data "munchkin" "witch" "dog" "lion" "tiger" "bear")}
       :mark "bar"
       :encoding {:x {:field "time"}
                  :y {:aggregate "sum"
                      :field "quantity"
                      :type "quantitative"}
                  :color {:field "item"}}})
    (oz/view! stacked-bar)   )
```

```
#object[oz.notebook.clojupyter$view_BANG_$reify__21517 0x7e5dfc87 "oz.notebook.clojupyter$v
```

```
;; Create spec, then visualize
(def spec
  {:data {:url "https://gist.githubusercontent.com/metasoarous/4e6f781d353322a44b9cd3e4597c
   :mark "point"
   :encoding {
     :x {:field "Horsepower", :type "quantitative"}
     :y {:field "Miles_per_Gallon", :type "quantitative"}
     :color {:field "Origin", :type "nominal"}}})
(oz/view! spec)
```

```
#'composable-statistics.core/spec#object[oz.notebook.clojupyter$view_BANG_$reify__21517 0x2
```

```
(oz/view!
  [:div
   [:h1 "A little hiccup example"]
   [:p "Try drinking a glass of water with your head upside down"]
   [:div {:style {:display "flex" :flex-direction "row"}}
    [:vega-lite spec]
    [:vega-lite stacked-bar]]])
```

```
#object[oz.notebook.clojupyter$view_BANG_$reify__21517 0x7e0a0c2f "oz.notebook.clojupyter$v
```

# 11 GAUSSIAN PROCESSES

The Extended Kalman Filter above is a generalization of linear regression.

## 11.1 RECURRENT LINEAR REGRESSION

Emacs 26.2 of 2019-04-12, org version: 9.2.2