Kalman Folding 9: in C (WORKING DRAFT)

Extracting Models from Data, One Observation at a Time

Brian Beckman

<2016-05-17 Tue>

Contents

1 Literate Code **1**
1.1 Recurrence for the Mean 1
1.2 Step 2: Getting LAPACK to work 5

1 Literate Code

1.1 Recurrence for the Mean

1.1.1 Business Code

C code is usually much longer than the corresponding Wolfram code (or Lisp, or Haskell, or OCaml, etc.). The extra length, overall, is due mostly to the need for manual memory management in C. However, we also don't have pattern matching for unpacking inputs and we don't have literal list / array notation for expressing some inputs and for packing outputs.

The goal of our C code is to make the final result as clean and as close to the original design in Wolfram as possible. For instance, the original source for *Recurrence for the Mean* in paper 1[1] is

```
cume[{x_, n_}, z_] := (* pattern matching for unpacking inputs *)
  With[{K = 1/(n + 1)},
   {x + K (z - x), n + 1}]; (* literal notation for packing outputs *)
Fold[cume, {0, 0}, {55, 89, 144}] (* literal notation for some inputs *)
~~> {96, 3}
```

The equivalent "business" code in C is still longer, but can see that the code for computing the gain and for the resulting *Accumulation* is virtually identical (we must use an explicit multiplication operator `*`, whereas a space suffices in Wolfram for scalar multiplication). At least the memory management is hidden in the types *Accumulation* and *Observation*, articulated further below.

```
{   Accumulator cume = ^(Accumulation a, Observation z)
        {   /* unpack inputs */
```

[1] B. Beckman, *Kalman Folding, Part 1*, to appear.

```
        T x = a.elements[0];
        T n = a.elements[1];
        /* compute gain */
        T K = 1.0 / (1.0 + n);
        /* busines logic, and packing results */
        Accumulation r;
        r.elements[0] = x + K * (z - x);
        r.elements[1] = 1.0 + n;

        return r;   };

    Accumulation x0 = zeroAccumulation ();

    Observation tmp[3] = {55, 89, 144};
    Observations zs = createObservations(3, tmp);

    Accumulation result = fold (cume, x0, zs);

    printAccumulation (result);
    return 0;   }
```

1.1.2 Accumulation Type

The code for the Accumulation type is a bit elaborate, but the extra abstractions will serve us well when we get to the Kalman filter.

The Accumulation structure presumes that all values are copied around on every use, and that's safe, and also means that we don't need alloc & free routines for this type. These accumulation types are usually small, so the time needed to copy them around may be acceptable. More sophisticated memory management for them entails more code, so we opt for keeping the code small at the cost of some copying that could be optimized away.

Also, in the interest of saving space, specifically, staircases of closing curly braces on lines by themselves, we adopt the *Pico*[2] style for bracing.

```
typedef double T;

const size_t Accumulation_size = 2;
typedef struct s_Accumulation
{   T elements[Accumulation_size];   } Accumulation;

Accumulation zeroAccumulation (void)
{   Accumulation r;
    memset ((void *)r.elements, 0, Accumulation_size * sizeof (T));
    return r;   }

void printAccumulation (Accumulation a)
```

[2] http://tinyurl.com/gku2k74

```
{    printf ("{%lf, %lf}\n", a.elements[0], a.elements[1]);   }
```

1.1.3 Observation Type

Here, because we don't statically know the number of observations, we must use dynamic memory allocation. In an embedded application, we would use arena memory (fixed-length circular buffer pools of fixed-length structs) or stack allocation (*calloc*). Here, for brevity and because this is a testing deployment, we use heap memory (stdlib's *malloc* and *free*). These are unacceptable in embedded applications because of fragmentation and unbounded execution times.

When we get to lazy streams, we won't need these at all. They're only for arrays of observations all in memory at one time.

The primary helper type is a bounded array of *Observations* type that includes length information and a handy iterator-like *current* index. Most of the code for this type concerns explicit memory management for this helper type.

We also include an *Observation* type, for asbstraction hygiene.

```
typedef T Observation, * pObservation;
typedef struct s_BoundedArray_Observations
{   int count;
    int current;
    pObservation observations;   } Observations;

/*private*/pObservation allocObservationArray (int count_)
{   /* Don't use malloc & free in embedded apps. Use arena or stack memory. */
    pObservation po = (pObservation) malloc (count_ * sizeof (Observation));
    if (NULL == po)
    {   printf ("Failed to alloc %d observations\n", count_);
        exit (-1);   }
    return po;   }

Observations createObservations (int count_, pObservation pObservations)
{   pObservation po = allocObservationArray (count_);
    memcpy ((void *)po, (void *)pObservations, sizeof (Observation) * count_);
    Observations result;
    result.count   = count_;
    result.current = 0;
    result.observations    = po;
    return result;   }

void freeObservations (Observations o)
{   /* Don't use malloc & free in embedded apps. Use arena or stack memory. */
    free ((void *)o.observations);   }
```

1.1.4 Accumulator Type

Our last type definition is for the *Accumulator* function. Here we cheat a bit and use an extension to the C language called *Blocks*,[3] which implements full closures. We could explicitly implement enough of closures for our purposes, but this extension is widely available with clang and llvm on Apple computers and Linux, and it's too convenient to pass up. With compilers for bare-metal processors in embedded systems, we might not have it and have to do more work by hand. With this extension, the *Accumulator* type, defined with the hat syntax `^`, behaves just like a function pointer, which would be defined with the ordinary pointer syntax, `*`.

```
typedef Accumulation (^Accumulator) (Accumulation a, Observation b);
```

1.1.5 The Fold Over Observations

The final piece is the *fold* operator. This particular one knows details of the *Observations* type, so is specific to it. We have another fold over lazy streams, articulated below, just as with Wolfram.

```
Accumulation fold (Accumulator f, Accumulation x0, Observations zs)
{   for (zs.current = 0; zs.current < zs.count; ++zs.current)
    {   x0 = f (x0, zs.observations[zs.current]);   }
    return x0;   }
```

1.1.6 Pulling it All Together

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <Block.h>
<<c-accumulation-type>>
<<c-observation-types>>
<<c-accumulator-type>>
<<c-fold-over-observations>>
int main (int argc, char ** argv)
<<c-business-logic>>
```

Tangle this code out to a C file by executing 'org-babel-tangle' while visiting this literate source code in emacs. For those who prefer vim, we may suggest *Spacemacs*,[4] which is a package for emacs with near-perfect vim emulation. A distribution of emacs for the Mac with adequate org-babel support is maintained by Vincent Goulet at the University of Laval.[5]

Compile and run the code as follows:

```
gcc -Wall -Werror recurrenceForTheMean.c -o recurrenceForTheMean
./recurrenceForTheMean
```

{96.000000 3.000000}

producing results all-but-identical to those from the Wolfram language.

[3] http://tinyurl.com/bgwfkyc
[4] http://www.spacemacs.org
[5] http://vgoulet.act.ulaval.ca/en/emacs/

1.2 Step 2: Getting LAPACK to work

Make sure you have `gfortran` installed:

```
gfortran --version
```

Get LAPACK.[6] This builds BLAS as a side effect.

```
pushd ~/Documents/lapack-3.6.0
cmake .
make
make test
```

```
pushd ~/Documents/lapack-3.6.0
make install
```

1.2.1 Make LAPACKE

This is the C interface to LAPACK. The following mercilessly hacks around a couple of problems in the build of `examples`, but it's enough to get the example working.

```
pushd ~/Documents/lapack-3.6.0
cp make.inc.example make.inc
cd LAPACKE
make lapacke
```

```
pushd ~/Documents/lapack-3.6.0
find . -name "*.a"
```

```
pushd ~/Documents/lapack-3.6.0
cd LAPACKE
cp ./include/lapacke*.h /usr/local/include
cd example
cp ../../liblapacke.a /usr/local/lib
cp ../../lib/*.a ../..
cp ../../libblas.a ../../librefblas.a
make
```

Emacs 24.5.1 (Org mode 8.3.4)

[6] `http://www.netlib.org/lapack/`