Kalman Folding 9: in C (WORKING DRAFT)

Extracting Models from Data, One Observation at a Time

Brian Beckman

<2016-05-17 Tue>

Contents

1 Abstract

In this literate program, we show one way to implement the basic Kalman fold in C. The background for Kalman folding appears in *Kalman Folding, Part 1*,[1] where we highlight the the unique advantages of functional folding for deploying test-hardened code verbatim in harsh, mission-critical environments. In that and in other papers in this series, the emphasis is on mathematical clarity and conciseness. Here, the emphasis is on embedded systems and on the primary language for implementing such systems, C.

2 Literate Code

A *literate program*[2] is a single text document that contains both

- LaTeX typesetting instructions and

[1]B. Beckman, *Kalman Folding, Part 1*, to appear.
[2]`https://en.wikipedia.org/wiki/Literate_programming`

- source code and scripts for an application.

A literate program can be edited in any text editor and controlled with text-based versioning tools like git, but requires other tools to extract the typeset document and to extract the source code. My chosen tools are emacs and org-babel.[3] In addition to extraction, called *tangling*, these tools allow interactive evaluation of code in the original text document. A reader may reproduce the results here by interacting with the source document.[4] If you wish to follow along interactively, you will need emacs and the text source for this document. The text source for this document is found here ***TODO***. A distribution of emacs for the Mac with adequate org-babel support is maintained by Vincent Goulet at the University of Laval.[5] Emacs for Linux and Windows is easy to find, but I do not discuss it further. *Spacemacs*[6] offers *vim* emulation with the tools needed for literate programming for those who prefer *vim*.

3 This Document is Literate

3.1 Get C to Work

First, make sure the following works[7] in org-babel and org-babel tangle. If it does work, you have a correctly installed C compiler.

```
int a=7;
int b=6;
printf("%d\n", a*b);
// ~~> produces
```

```
42
```

3.2 Recurrence for the Mean

3.2.1 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}
```

[3] http://orgmode.org/worg/org-contrib/babel/

[4] https://www.coursera.org/learn/reproducible-research

[5] http://vgoulet.act.ulaval.ca/en/emacs/

[6] http://www.spacemacs.org

[7] Make sure the first example from http://tinyurl.com/kz2lz7m works

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 */
            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;   }
```

Figure 1: Business logic in C

3.2.2 Primary Numerical Type

We abstract out the primary numerical type into the traditional `T` to make it easier to change.

```
typedef double T;
```

Figure 2: Primary numerical type

3.2.3 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*[8] style for bracing.

```
const size_t Accumulation_size = 3;
typedef struct s_Accumulation
{   T elements[Accumulation_size];   } Accumulation, * pAccumulation;

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

void printAccumulation (Accumulation a)
{   printf ("{");
    for (size_t i = 0; i < Accumulation_size; ++i)
    {   printf ("%lf", a.elements[i]);
        if (i < Accumulation_size - 1)
        {   printf (", ");   }   }
    printf ("}\n");   }
```

Figure 3: Accumulation type

We have harmlessly used 3 for the accumulation size because we want to reuse this code later. We could make it variable at the cost of more unilluminating code.

3.2.4 Observation Types

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 the length 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;
```

[8] `http://tinyurl.com/gku2k74`

```
/*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);   }
```

Figure 4: Observation types

3.2.5 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*,[9] 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);
```

Figure 5: Accumulator type

3.2.6 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)
```

[9] `http://tinyurl.com/bgwfkyc`

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

Figure 6: Fold over observations in bounded arrays

3.2.7 Pulling it All Together

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

Figure 7: Recurrence for the mean: entire program

Tangle this code out to a C file by executing 'org-babel-tangle' while visiting this literate source code in emacs.

Compile and run the code as follows:

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

Figure 8: Build and execute script for recurrence-for-the-mean

Table 1: Output of recurrence-for-the-mean

```
{96.000000    3.0    0.000000}
```

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

3.3 FoldList and Recurrence for the Variance

The original paper introduced Wolfram's *FoldList* along with the recurrence for the variance. We do likewise here, implementing our own *foldList* in C.

3.3.1 Bounded Array for Accumulations

FoldList produces a list of accumulations, one for the initial accumulation and another for each observation. With lists of observations all in memory, we could calculate the length of the output and preallocate a list of accumlations of the correct size, but we are not able to do that with lazy streams of observations or asynchronous observables of observations. We opt, then, for on-demand, dynamic memory management for the output accumulations. "On-demand," here, means growing the output array as new accumulations arrive. We use the common trick of doubling the capacity of the output array every time the capacity is exceeded. This trick is a reasonable compromise of space and time efficiency.

We emulate the *bounded-array* interface created for observations, and add three more functions to the usual *create*, *free*, and *print*.

lastAccumulations returns the last accumulation in a bounded array; needed for *foldList*

appendAccumulations appends a new accumulation to a bounded array of accumulations, growing the capacity if needed

foldList takes an accumulator f, an initial accumulation a_0, a bounded array of observations zs, and produces a bounded array of accumulations.

```
typedef struct s_BoundedArray_Accumulations
{   int count;
    int max;
    pAccumulation accumulations ;   } Accumulations;

Accumulation lastAccumulations (Accumulations as)
{   if (0 == as.count)
    {   printf ("Attempt to pull non-existent element\n");
        exit (-4);   }
    return as.accumulations[as.count - 1];   }

Accumulations appendAccumulations (Accumulations as, Accumulation a)
{   Accumulations result = as;
    if (result.count + 1 > result.max)
    {   /* Double the storage. */
        int new_max = 2 * result.max;
        /* Don't use malloc & free in embdded apps. Use arena or stack memory. */
        pAccumulation new = (pAccumulation)
          malloc (sizeof (Accumulation) * new_max);
        if (NULL == new)
        {   printf ("Failed to alloc %d Accumulations\n", new_max);
            exit (-2);   }
        if (result.count != result.max)
        {   printf ("Internal bugcheck\n");
            exit (-3);   }
        memset ((void *)new, 0, new_max * sizeof (Accumulation));
        memcpy ((void *)new, (void *)result.accumulations,
          (sizeof (Accumulation) * result.max));
        free ((void *) result.accumulations);
        result.accumulations = new;
        result.max = new_max;   }
    result.accumulations[result.count] = a;
    ++ result.count;
    return result;   }

Accumulations createAccumulations (void)
{   Accumulations result;
    const int init_size = 4;
    result.max = init_size;
    result.count = 0;
    result.accumulations = (pAccumulation)
      malloc (sizeof (Accumulation) * init_size);
    memset ((void *)result.accumulations, 0,
      sizeof (Accumulation) * init_size);
    return result;   }

void freeAccumulations (Accumulations as)
{   memset ((void *) as.accumulations, 0,
      (sizeof (Accumulation) * as.count));
    free ((void *) as.accumulations);   }

void printAccumulations (Accumulations as)
{   for (int j = 0; j < as.count; ++j )
    {   printAccumulation (as.accumulations[j]);   }   }

Accumulations foldList (Accumulator f, Accumulation a0, Observations zs)
{   Accumulations result = createAccumulations ();
    result = appendAccumulations (result, a0);
    for (zs.current = 0; zs.current < zs.count; ++zs.current)
    {   result = appendAccumulations (
          result,
          f(lastAccumulations(result),
          zs.observations[zs.current]));   }
        return result;   }
```

Figure 9: Bounded array for accumulations

3.3.2 Pulling Together Recurrence for the Variance

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <Block.h>
<<c-numerical-type>>
<<c-accumulation-type>>
<<c-observation-types>>
<<c-accumulator-type>>
```

```
  <<c-fold-over-observations>>
  <<c-bounded-array-for-accumulations>>
  int main (int argc, char ** argv)
{   Observation tmp[3] = {55, 89, 144};
    Observations zs = createObservations(3, tmp);
    Accumulation x0 = zeroAccumulation ();
    Accumulator cume = ^(Accumulation a, Observation z)
        {   T var = a.elements[0];
            T x   = a.elements[1];
            T n   = a.elements[2];

            T K = 1.0 / (1.0 + n);
            T x2 = x + K * (z - x);
            T ssr2 = (n - 1.0) * var + K * n * (z - x) * (z - x);

            Accumulation r;
            r.elements[0] = ssr2 / (n > 1.0 ? n : 1.0);
            r.elements[1] = x2;
            r.elements[2] = n + 1.0;
            return r;   };

    Accumulations results = foldList (cume, x0, zs);
    printAccumulations (results);

    freeAccumulations (results);
    freeObservations (zs);
    return 0;   }
```

Figure 10: Recurrence for the variance: entire program

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

Figure 11: Build and execute script for recurrence-for-the-variance

Table 2: Output of recurrence-for-the-variance

{0.000000	0.0	0.000000}
{0.000000	55.0	1.000000}
{578.000000	72.0	2.000000}
{2017.000000	96.0	3.000000}

This result is semantically identical to that produced by the following Wolfram code:

```
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}]];
Fold[cume, {0, 0, 0}, zs]
~~> {2017, 96, 3}
```

Figure 12: Wolfram code for recurrence for the variance

4 Basic Kalman Folding

4.1 Avoiding the Inverse

In the first paper in this series, we wrote one version of the static Kalman filter, when there are no system dynamics,[10] as follows.

$$\text{cume}\,(\mathbf{Z})\,(\{\mathbf{x},\mathbf{P}\},\{\mathbf{A},\mathbf{z}\}) = \{\mathbf{x}+\mathbf{K}\,(\mathbf{z}-\mathbf{A}\,\mathbf{x})\,,\mathbf{P}-\mathbf{K}\,\mathbf{A}\,\mathbf{P}\} \tag{1}$$

where

$$\mathbf{K} = \mathbf{P}\,\mathbf{A}^\intercal\,\mathbf{D}^{-1} \tag{2}$$

$$\mathbf{D} = \mathbf{Z} + \mathbf{A}\,\mathbf{P}\,\mathbf{A}^\intercal \tag{3}$$

and all quantities are matrices, and

- $\mathbf{Z}$ = $b \times b$ covariance of observation noise
- $\mathbf{x}$ = $n \times 1$ model states
- $\mathbf{P}$ = $n \times n$ theoretical covariance of $\mathbf{x}$
- $\mathbf{A}$ = $b \times n$ *observation partials*
- $\mathbf{z}$ = $b \times 1$ multidimensional, decorrelated observations
- $\mathbf{K}$ = $n \times b$ *Kalman gain*
- $\mathbf{D}$ = $b \times b$ the Kalman denominator

Adding physical dimensions, if the physical and matrix dimensions of $\mathbf{x}$ are $[[\mathbf{x}]] \overset{\text{def}}{=} (\mathcal{X}, n \times 1)$ and of $\mathbf{z}$ are $[[\mathbf{z}]] \overset{\text{def}}{=} (\mathcal{Z}, b \times 1)$, then

$$\begin{array}{lcclcl}
[[\mathbf{Z}]] & = & (& \mathcal{Z}^2 & b \times b &) \\
[[\mathbf{P}]] & = & (& \mathcal{X}^2 & n \times n &) \\
[[\mathbf{A}]] & = & (& \mathcal{Z}/\mathcal{X} & b \times n &) \\
[[\mathbf{A}\,\mathbf{P}\,\mathbf{A}^\intercal]] & = & (& \mathcal{Z}^2 & b \times b &) \\
[[\mathbf{D}]] & = & (& \mathcal{Z}^2 & b \times b &) \\
[[\mathbf{P}\,\mathbf{A}^\intercal]] & = & (& \mathcal{X}\,\mathcal{Z} & n \times b &) \\
[[\mathbf{K}]] & = & (& \mathcal{X}/\mathcal{Z} & n \times b &)
\end{array}$$

[10] B. Beckman, *Kalman Folding 2: Tracking and System Dynamics*, to appear.

While an expression with an explicit inverse is mathematically acceptable, inverses are numerically risky, expensive in storage, slow to compute, and usually not necessary.[11] LAPACK can solve linear systems very efficiently, much more efficiently than it can invert matrices. Therefore, we rewrite the basic filter to avoid computing $\mathbf{D}^{-1}$.

If DiRes $= \mathbf{D}^{-1}\,(z - \mathbf{A}\,x)$ is the solution of the linear equation $\mathbf{D} \times$ DiRes $= (z - \mathbf{A}\,x)$, and if $\mathbf{K} = \mathbf{P}\,\mathbf{A}^\intercal\,\mathbf{D}^{-1}$, then $\mathbf{K}\,(z - \mathbf{A}\,x) = \mathbf{P}\,\mathbf{A}^\intercal$ DiRes and the Kalman state-update is $x \leftarrow x + \mathbf{P}\,\mathbf{A}^\intercal$ DiRes. Likewise, if DiAP $= \mathbf{D}^{-1}\,\mathbf{A}\,\mathbf{P}$ is the solution of the linear equation $\mathbf{D} \times$ DiAP $= \mathbf{A}\,\mathbf{P}$, then $\mathbf{K}\,\mathbf{A}\,\mathbf{P} = \mathbf{P}\,\mathbf{A}^\intercal$ DiAP and the Kalman covariance update is $\mathbf{P} \leftarrow \mathbf{P} - \mathbf{P}\,\mathbf{A}^\intercal$ DiAP.

In Wolfram, our original, foldable Kalman filter was

```
kalman[Z_][{x_, P_}, {A_, z_}] :=
  Module[{D, K},
   D = Z + A.P.Transpose[A];
   K = P.Transpose[A].Inverse[D];
   {x + K.(z - A.x), P - K.A.P}];
```

and our new minimal, foldable filter is

```
noInverseKalman[Z_][{x_, P_}, {A_, z_}] :=
  Module[{PAT, D, KRes, KAP},
   PAT = P.Transpose[A];
   D = Z + A.PAT;
   KRes = PAT.LinearSolve[D, z - A.x];
   KAP = PAT.LinearSolve[D, A.P];
   {x + KRes, P - KAP}];
```

This reads almost as easily as the original if one reads `LinearSolve` as *invert-first-argument-and-matrix-multiply*.

Notice we do not compute the Kalman gain explicitly, but only use it in combination with other matrices. This produces results indistinguishable from the original, up to floating-point issues, when folded over any source of data.

LAPACK offers a function, `dposv`,[12, 13] that solves this linear system when $\mathbf{D}$ is symmetric and positive definite. Because $\mathbf{D}$ is the sum of a diagonal matrix $\mathbf{Z}$ and a symmetric, positive-definite matrix $\mathbf{A}\,\mathbf{P}\,\mathbf{A}^\intercal$, it should also be symmetric and positive definite. Therefore, we transcribe the code above into C as follows

4.2 Fortran and C

We need matrix operations, and we choose CBLAS[14] and LAPACKE.[15]

[11]Cook, John D. *Don't invert that matrix* `http://tinyurl.com/ya4q2kv`
[12]`http://www.nag.com/numeric/FL/manual/pdf/F07/f07faf.pdf`
[13]`http://www.nag.com/numeric/CL/CLdocumentation.asp`
[14]`http://www.netlib.org/blas/`
[15]`http://www.netlib.org/lapack/lapacke.html`

4.3 LAPACK and LAPACKE

4.3.1 Full Least-Squares Without Fold

```
const int    m = 5;
const int    n = 4;

double A[m * n] = { 1,  0.,  0.,  0.,
                    1,  1.,  1.,  1.,
                    1, -1.,  1., -1.,
                    1, -2.,  4., -8.,
                    1,  2.,  4.,  8. };

// Compute Transpose[A].A; A is not disturbed.

double AtA[n * n];
memset (AtA, 0, n * n * sizeof (double));

cblas_dgemm (CblasRowMajor, CblasTrans, CblasNoTrans,
             n, n, m, 1,
             A, n, // LDA is pre-transpose
             A, n,
             0,
             AtA, n);

for (int r = 0; r < n; ++r)
{   for (int c = 0; c < n; ++c)
    {   printf ("%g ", AtA[c + r * n]);   }
    printf ("\n");   }
printf ("\n");

// Compute Transpose[A].z; neither A nor z is disturbed. Results are deposited
// into Atz.

double z[m] = {-2.28442, -4.83168, -10.4601, 1.40488, -40.8079};

double Atz[n];

cblas_dgemv (CblasRowMajor, CblasTrans,
             m, n, 1,
             A, n,
             z, 1, 0,
             Atz, 1);

for (int i = 0; i < n; ++i)
{   printf ("%g ", Atz[i]);   }
printf ("\n");
```

```
// Solve At.A.x = At.z = Atz. Unlike the CBLAS routines, the input storage
// locations are modified. The Cholesky decomposition of AtA is deposited into
// AtA, in-place, and the solution is deposited into Atz. To preserve these
// matrices, it’s necessary to copy them first.

// The documentation for LAPACKE_dposv has an apparent error (see
// http://tinyurl.com/htvod3e). It states that the leading dimension of B must
// be >= max(1, N), but we suspect it should say >= max(1, NRHS). The results
// are definitely wrong if N is used as LDB.

// The results of this computation are identical to those from Mathematica. This
// is not surprising because Mathematica probably uses LAPACK internally.

lapack_int LAPACKE_dposv( int matrix_layout, char uplo, lapack_int n,
                          lapack_int nrhs, double* a, lapack_int lda, double* b,
                          lapack_int ldb );

lapack_int result = LAPACKE_dposv (LAPACK_ROW_MAJOR, ’U’, n, 1, AtA, n, Atz, 1);

printf ("%d\n\n", result);

for (int i = 0; i < n; ++i)
{   printf ("%g ", Atz[i]);   }
printf ("\n");
```

```
       5        0       10        0
       0       10        0       34
      10        0       34        0
       0       34        0      130

-56.9792 -78.7971 -172.904 -332.074
       0

-2.97507  7.27001 -4.21039  -4.4558
```

4.3.2 Foldable Kalman Without Inverse

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cblas.h>
#include <lapacke.h>

void printm (char * nym, double * m, int rows, int cols)
{   printf ("%s\n", nym);
    for (int r = 0; r < rows; ++r)
    {   for (int c = 0; c < cols; ++c)
        {   printf ("%g ", m[c + r * cols]);   }
        printf ("\n");   }
    printf ("\n");   }

void kalman (int b,        /* # rows, cols, in Z; # rows in z */
             int n,        /* # rows, cols, in P; # rows in x */
             double * IdN, /* n x n identity matrix */
             double * Z,   /* b x b observation covariance */
             double * x,   /* n x 1, current state */
             double * P,   /* n x n, current covariance */
             double * A,   /* b x n, current observation partials */
             double * z    /* b x 1, current observation vector */
             ) {

     /* Transcribe the following Wolfram code (the intermediate matrices are not
      * necessary in Wolfram, but we need them in C).
      *
      * noInverseKalman[Z_][{x_, P_}, {A_, z_}] :=
      *   Module[{PAT, D, DiRes, DiAP, KRes, KAP},
      *    PAT = P.Transpose[A];               (* n x b *)
      *    D = Z + A.PAT;                      (* b x b *)
      *    DiRes = LinearSolve[D, z - A.x];    (* b x 1 *)
      *    KRes = PAT.DiRes;                   (* n x 1 *)
      *    DiAP = LinearSolve[D, A.P];         (* b x n *)
      *    KAP = PAT.DiAP;                     (* n x n *)
      *    {x + KRes, P - KAP}];
      */


     /* Use dgemm for P.A^T because dsymm doesn't offer a way to transpose the
        right-hand multiplicand. */

    /*
     *      PAT                 P           AT
     *       b                  n           b
     *    / * * \         / * * * * \   / * * \
     *  n | * * |  <--  n | * * * * | n | * * |
     *    | * * |         | * * * * |   | * * |
     *    \ * * /         \ * * * * /   \ * * /
     *
     */

    double PAT[n * b];
    /* dgemm: http://tinyurl.com/j24npm4 */
    /* C <-- alpha * A * B + beta * C */
```

```
cblas_dgemm (CblasRowMajor, CblasNoTrans, CblasTrans,
             n,          /* m (n),    # rows of A (P) */
             b,          /* n (b),    # cols of B (AT) (post-transpose) */
             n,          /* k (n),    # cols of A (P) == rows of B (AT post-tranpos
             1, P, n,    /* alpha, A, # cols A (P,  pre-transpose)*/
             A, n,       /*        B, # cols B (AT, pre-transpose)*/
             0, PAT, b); /* beta,  C, # cols C */
printm ("P.AT", PAT, n, b);

/*
 *       D                 A          PAT           Z
 *       b                 n           b            b
 *  b / * * \  <--  b / * * * * \ n / * * \  + b / * * \
 *    \ * * /         \ * * * * /   | * * |      \ * * /
 *                                  | * * |
 *                                  \ * * /
 *
 */

double D[b * b];
/* D <- A.PAT + Z (copy Z to D first) */
cblas_dcopy (b * b, Z, 1, D, 1);
/* dgemm: http://tinyurl.com/j24npm4 */
/* C <-- alpha * A * B + beta * C */
cblas_dgemm (CblasRowMajor, CblasNoTrans, CblasNoTrans,
             b,          /* m (b),          # rows of A (A) */
             b,          /* n (b),          # cols of B (PAT) */
             n,          /* k (n),          # cols of A (A) == rows of B (PAT) */
             1, A, n,    /* alpha, A (A),   # cols A (A) */
             PAT, b,     /*        B (PAT), # cols B (PAT)*/
             1, D, b);   /* beta,  C (Z),   # cols C (D) */
printm ("D", D, b, b);

/*
 *     Res                       A          x                 z
 *      1                        n          1                 1
 *  b / * \  <--  alpha * b / * * * * \ n / * \  + beta * b / * \
 *    \ * /                 \ * * * * /   | * |             \ * /
 *                                        | * |
 *                                        \ * /
 *
 */
double Res[b * 1];
/* Res <- (-A.x) + z (copy z to Res first)  */
cblas_dcopy (b * 1, z, 1, Res, 1);
/* dgemm: http://tinyurl.com/j24npm4 */
/* C <-- alpha * A * B + beta * C */
cblas_dgemm (CblasRowMajor, CblasNoTrans, CblasNoTrans,
             b,           /* m (b),        # rows of A (A) */
             1,           /* n (1),        # cols of B (x) */
             n,           /* k (n),        # cols of A (A) == rows of B (x) */
             -1, A, n,    /* alpha, A (A), # cols A (A) */
             x, 1,        /*        B (x), # cols B (x) */
             1, Res, 1); /* beta,  C (z), # cols C (Res) */
printm ("Res", Res, b, 1);

/*
 *    DiRes        Di = D^-1   Res
 *      1              b        1
 *  b / * \  <--  b / * * \ b / * \
 *    \ * /         \ * * /   \ * /
 *
 */

double DiRes[b * 1];
double DCholesky[b * b];
/* DiRes = LinearSolve[D, z - A.x];    (* b x 1 *) */
/* copy Res to DiRes, first. */
/* copy D to DCholesky first. */
/* dposv: http://goo.gl/O7gUH8 */
cblas_dcopy (b * 1, Res, 1, DiRes,     1);
cblas_dcopy (b * b, D,   1, DCholesky, 1);
int result = LAPACKE_dposv (LAPACK_ROW_MAJOR, 'U',
                            b,          /* NEQS: # rows of D */
                            1,          /* NRHS: # columns of z - A.x == Res */
                            DCholesky, /* DCholesky starts as D */
                            b,          /* PDA D */
                            DiRes,      /* output buffer */
                            b);         /* PDA DiRes */
printf ("DPOSV DiRes result %d\n\n", result);
printm ("DiRes",     DiRes,     b, 1);
printm ("DCholesky", DCholesky, b, b);

/*
 *     KRes           PAT     DiRes
 *      1              b        1
 *  n / * \       n / * * \ b / * \
 *    | * |  <--    | * * |   \ * /
 *    | * |         | * * |
 *    \ * /         \ * * /
 *
 */
```

```
double KRes[n * 1];
/* KRes <-- PAT.DiRes */
/* dgemm: http://tinyurl.com/j24npm4 */
/* C <-- alpha * A * B + beta * C */
cblas_dgemm (CblasRowMajor, CblasNoTrans, CblasNoTrans,
             n,           /* m (n),            # rows of A (PAT) */
             1,           /* n (1),            # cols of B (DiRes) */
             b,           /* k (b),            # cols of A (PAT) == # rows of B (Di
             1, PAT, b,   /* alpha, A (PAT),   # cols A (PAT) */
             DiRes, 1,    /*        B (DiRes), # cols B (DiRes) */
             0, KRes, 1); /* beta,  C (KRes),  # cols C (KRes) */
printm ("KRes", KRes, n, 1);

/*
 *         AP                   A             P
 *         n                    n             n
 *  b / * * * * \  <--  b / * * * * \ n / * * * * \
 *    \ * * * * /         \ * * * * /   | * * * * |
 *                                      | * * * * |
 *                                      \ * * * * /
 *
 */

double AP[b * n];
/* AP <-- A.P */
/* dgemm: http://tinyurl.com/j24npm4 */
/* C <-- alpha * A * B + beta * C */
cblas_dgemm (CblasRowMajor, CblasNoTrans, CblasNoTrans,
             b,           /* m (b),          # rows of A (A) */
             n,           /* n (n),          # cols of B (P) */
             n,           /* k (n),          # cols of A (A) == # rows of B (P) */
             1, A, n,     /* alpha, A (A),   # cols A (PAT) */
             P, n,        /*        B (P),   # cols B (DiRes) */
             0, AP, n);   /* beta,  C (AP),  # cols C (KRes) */
printm ("AP", AP, b, n);

/*
 *   Di = D^-1       A             P                  DiAP
 *       b           n             n                   n
 *  b / * * \ b / * * * * \ n / * * * * \  -->  b / * * * * \
 *    \ * * /   \ * * * * /   | * * * * |         \ * * * * /
 *                            | * * * * |
 *                            \ * * * * /
 *
 */
```

```
double DiAP[b * n];
/* DiAP = LinearSolve[D, AP];    (* b x n *) */
/* copy AP to DiAP, first. */
/* copy D to DCholesky first. */
/* dposv: http://goo.gl/O7gUH8 */
cblas_dcopy (b * n, AP, 1, DiAP,      1);
cblas_dcopy (b * b, D,  1, DCholesky, 1);
result = LAPACKE_dposv (LAPACK_ROW_MAJOR, 'U',
                        b,         /* NEQS: # rows of D */
                        n,         /* NRHS: # columns of z - A.x == Res */
                        DCholesky, /* DCholesky starts as D */
                        b,         /* PDA D */
                        DiAP,      /* output buffer */
                        n);        /* PDA DiRes */
printf ("DPOSV DiAP result %d\n\n", result);
printm ("DiAP",      DiAP,      b, n);
printm ("DCholesky", DCholesky, b, b);

/*
 *        KAP              PAT         DiAP
 *         n                b           n
 *  n / * * * * \  <--  / * * \ b / * * * * \
 *    | * * * * |     n | * * |   \ * * * * /
 *    | * * * * |       | * * |
 *    \ * * * * /       \ * * /
 *
 */

double KAP[n * n];
/* KAP <-- PAT.DiAP */
/* dgemm: http://tinyurl.com/j24npm4 */
/* C <-- alpha * A * B + beta * C */
cblas_dgemm (CblasRowMajor, CblasNoTrans, CblasNoTrans,
             n,             /* m (n),           # rows of A (PAT) */
             n,             /* n (n),           # cols of B (DiAP) */
             b,             /* k (b),           # cols of A (PAT) == # rows of B (D
             1, PAT, b,     /* alpha, A (PAT),  # cols A (PAT) */
             DiAP, n,       /*        B (Diap), # cols B (DiRes) */
             0, KAP, n);    /* beta,  C (KAP),  # cols C (KAP) */
printm ("KAP", KAP, n, n);

/*
 *      x                       Id          x                  KRes
 *      1                        n          1                   1
 *  n / * \  <--  alpha * n / * * * * \ n / * \  +  beta * n / * \
     *    | * |                 | * * * * |   | * |              | * |
     *    | * |                 | * * * * |   | * |              | * |
     *    \ * /                 \ * * * * /   \ * /              \ * /
     *
     */

    /* x <-- alpha * IdN[n] * KRes + beta * x */
    /* dgemm: http://tinyurl.com/j24npm4 */
    /* C <-- alpha * A * B + beta * C */
    cblas_dgemm (CblasRowMajor, CblasNoTrans, CblasNoTrans,
                 n,           /* m (n),           # rows of A (Id) */
                 1,           /* n (1),           # cols of B (x) */
                 n,           /* k (n),           # cols of A (Id) == rows of B (x) */
                 1, IdN, n,   /* alpha, A (Id),   # cols A */
                 x, 1,        /*        B (x),    # cols B */
                 1, KRes, 1); /* beta,  C (Kres), # cols C (new x) */
    cblas_dcopy (n * 1, KRes, 1, x, 1);
    printm ("x", x, n, 1);

    /*
     *         P                          Id            KAP                        P
     *         n                          n             n                          n
     *  n / * * * * \  <--  alpha * n / * * * * \ n / * * * * \  +  beta * n / * * * *
     *    | * * * * |                 | * * * * |   | * * * * |              | * * * *
     *    | * * * * |                 | * * * * |   | * * * * |              | * * * *
     *    \ * * * * /                 \ * * * * /   \ * * * * /              \ * * * *
     *
     */

    /* P <-- P - KAP == - IdN[n] * KAP  + P */
    /* dgemm: http://tinyurl.com/j24npm4 */
    /* C <-- alpha * A * B + beta * C */
    cblas_dgemm (CblasRowMajor, CblasNoTrans, CblasNoTrans,
                 n,           /* m (n),         # rows of A (Id) */
                 n,           /* n (n),         # cols of B (KAP) */
                 n,           /* k (n),         # cols of A (Id) == rows of B (KAP) */
                 -1, IdN, n,  /* alpha, A (Id), # cols A */
                 KAP, n,      /*        B (x),  # cols B */
                 1, P, n);    /* beta,  C (P),  # cols C (new P) */
    printm ("P", P, n, n); }

int main (int argc, char ** argv)
{   const int    b = 1;
    const int    n = 4;

    double IdN[n * n] = { 1., 0., 0., 0.,
                        0., 1., 0., 0.,
                        0., 0., 1., 0.,
                        0., 0., 0., 1. };


double Z[b * b] = {1.};

double x[n * 1] = {0., 0., 0., 0};
double P[n * n] = {1000.,    0.,    0.,    0.,
                        0., 1000.,    0.,    0.,
                        0.,    0., 1000.,    0.,
                        0.,    0.,    0., 1000. };

double A[b * n] = {1., 0., 0., 0};
double z[b] = {-2.28442};

kalman (b, n, IdN, Z, x, P, A, z);

A[0] = 1;
A[1] = 1;
A[2] = 1;
A[3] = 1;

z[0] = -4.83168;

kalman (b, n, IdN, Z, x, P, A, z);

A[0] = 1;
A[1] = -1;
A[2] = 1;
A[3] = -1;

z[0] = -10.4601;

kalman (b, n, IdN, Z, x, P, A, z);

A[0] = 1;
A[1] = -2;
A[2] = 4;
A[3] = -8;

z[0] = 1.40488;

kalman (b, n, IdN, Z, x, P, A, z);

A[0] = 1;
```

```
    A[1] = 2;
    A[2] = 4;
    A[3] = 8;

    z[0] = -40.8079;

    kalman (b, n, IdN, Z, x, P, A, z);

    return 0;   }
```

```
gcc qux.c -lcblas -llapacke -llapack
./a.out
```

```
const int    b = 1;
const int    n = 4;
double x[n * 1] = {0., 0., 0., 0};
double Z[b * b] = {1.};
double P[n * n] = {1000.,    0.,    0.,    0.,
                      0., 1000.,    0.,    0.,
                      0.,    0., 1000.,    0.,
                      0.,    0.,    0., 1000. };

double A[b * n] = {1., 0., 0., 0};
double z[b] = {-2.28442};

/* Transcribe the following Wolfram code (the intermediate matrices are not
 * necessary in Wolfram, but we need them in C).
 *
 * noInverseKalman[Z_][{x_, P_}, {A_, z_}] :=
 *   Module[{PAT, D, DiRes, DiAP, KRes, KAP},
 *    PAT = P.Transpose[A];               (* n x b *)
 *    D = Z + A.PAT;                      (* b x b *)
 *    DiRes = LinearSolve[D, z - A.x];    (* b x 1 *)
 *    KRes = PAT.DiRes;                   (* n x 1 *)
 *    DiAP = LinearSolve[D, A.P];         (* b x n *)
 *    KAP = PAT.DiAP;                     (* n x n *)
 *    {x + KRes, P - KAP}];
 */

/* Using dgemm for P.A^T because dsymm doesn't offer a way to transpose the
   right-hand multiplicand. */

/*
 *         A           x             z           Res
 *         n           1             1            1
 *  b / * * * * \   / * \  -->  b / * \ ,   b / * \
 *    \ * * * * / n | * |         \ * /       \ * /
 *                  | * |
 *                  \ * /
 *
 *         P           AT             PAT
 *         n           b               b
 *    / * * * * \   / * * \  -->    / * * \
 *  n | * * * * | n | * * |       n | * * |
 *    | * * * * |   | * * |         | * * |
 *    \ * * * * /   \ * * /         \ * * /
 *
 *       Z
 *       b
 *  b / * * \
 *    \ * * /
 *
 *         A             P           AT           APAT
 *         n             n           b              b
 *  b / * * * * \   / * * * * \   / * * \  --> b / * * \
 *    \ * * * * / n | * * * * | n | * * |        \ * * /
 *                  | * * * * |   | * * |
 *                  \ * * * * /   \ * * /
 *
 *       D         Di = D^-1   Res          DiRes
 *       b             b        1             1
 *  b / * * \ ,   b / * * \ b / * \  -->  b / * \
 *    \ * * /       \ * * /   \ * /         \ * /
 *
 *      PAT     DiRes          KRes
 *       b        1             1
 *    / * * \ b / * \  -->    / * \
 *  n | * * |   \ * /       n | * |
 *    | * * |                 | * |
 *    \ * * /                 \ * /
 *
 *         A
 *         n
 *  b / * * * * \
 *    \ * * * * /
 */

double PAT[n * b];
/* dgemm: http://tinyurl.com/j24npm4 */
cblas_dgemm (CblasRowMajor, CblasNoTrans, CblasTrans,
             n, b, n,    /* m, n, k       */
             1, P, n,    /* alpha, A, pda */
             A, n,       /*        B, pdb */
             0, PAT, n); /* beta,  C, pdc */

printf ("P.AT\n");
for (int r = 0; r < n; ++r)
{   for (int c = 0; c < b; ++c)
    {   printf ("%g ", PAT[c + r * b /* ncols */]);   }   }
printf ("\n\n");

double D[b * b];
/* D <- A.PAT + Z (copy Z to D first) */
cblas_dcopy (b * b, Z, 1, D, 1);
cblas_dgemm (CblasRowMajor, CblasNoTrans, CblasNoTrans,
             b, b, n,    /* m, n, k       */
             1, A, n,    /* alpha, A, pda */
             PAT, b,     /*        B, pdb */
             1, D, b);   /* beta,  C, pdc */

printf ("D\n");
for (int r = 0; r < b; ++r)
{   for (int c = 0; c < b; ++c)
    {   printf ("%g ", D[c + r * b]);   }   }
printf ("\n\n");

double Res[b * 1];
/* Res <- (-A.x) + z (copy z to Res first)  */
cblas_dcopy (b * 1, z, 1, Res, 1);
cblas_dgemm (CblasRowMajor, CblasNoTrans, CblasNoTrans,
             b, 1, n,    /* m, n, k       */
             -1, A, n,   /* alpha, A, pda */
             x, 1,       /*        B, pdb */
             1, Res, 1); /* beta,  C, pdc */

printf ("Res\n");
for (int r = 0; r < b; ++r)
{   for (int c = 0; c < 1; ++c)
    {   printf ("%g ", Res[c + r * 1]);   }   }
printf ("\n\n");

double DiRes[b * 1];
double DCholesky[b * b];
/*    DiRes = LinearSolve[D, z - A.x];    (* b x 1 *) */
/*    copy Res to DiRes, first. */
/*    copy D to DCholesky first. */
/* dposv: http://goo.gl/O7gUH8 */
cblas_dcopy (b * 1, Res, 1, DiRes,     1);
cblas_dcopy (b * b, D,   1, DCholesky, 1);
```

```
int result = LAPACKE_dposv (
, 'U',
b, /* n rows of D */
1, /* n columns of Res */
D,
b,
DiRes,
b);

printf ("DPOSV result %d\n\n", result);

printf ("DiRes\n");
for (int r = 0; r < b; ++r)
{   for (int c = 0; c < 1; ++c)
    {   printf ("%g ", DiRes[c + r * 1]);   }   }
printf ("\n\n");

printf ("DCholesky\n");
for (int r = 0; r < b; ++r)
{   for (int c = 0; c < b; ++c)
    {   printf ("%g ", DCholesky[c + r * b]);   }   }
printf ("\n\n");

double KRes[n * 1];
cblas_dgemm (CblasRowMajor, CblasNoTrans, CblasNoTrans,
             n, 1, b,     /* m, n, k       */
             1, PAT, 1,   /* alpha, A, pda */
             DiRes, 1,    /*        B, pdb */
             0, KRes, 1); /* beta,  C, pdc */

printf ("KRes\n");
for (int r = 0; r < n; ++r)
{   for (int c = 0; c < 1; ++c)
    {   printf ("%g ", KRes[c + r * 1]);   }   }
printf ("\n\n");

/*  AP              = (b x n).(n x n)  -->  (b x n)
 *  DiAP            = (b x b).(b x n)  -->  (b x n)
 *  KAP  = PAT.DiAP = (n x b).(b x n)  -->  (n x n)
 */

double AP[b * n];
cblas_dgemm (CblasRowMajor, CblasNoTrans, CblasNoTrans,
             b, n, n,     /* m, n, k       */
             1, A, n,     /* alpha, A, pda */
             P, n,        /*        B, pdb */
             0, AP, n);   /* beta,  C, pdc */

printf ("AP\n");
for (int r = 0; r < b; ++r)
{   for (int c = 0; c < n; ++c)
    {   printf ("%g ", AP[c + r * n]);   }   }
printf ("\n\n");

double DiAP[b * n];
/* DiAP = LinearSolve[D, AP];    (* b x n *) */
/* copy AP to DiAP, first. */
/* copy D to DCholesky first. */
/* dposv: http://goo.gl/O7gUH8 */
cblas_dcopy (b * n, AP, 1, DiAP,      1);
cblas_dcopy (b * b, D,  1, DCholesky, 1);
int result = LAPACKE_dposv (LAPACK_ROW_MAJOR, 'U', b, b, D, b, DiRes, b);

double KAP[n * n];
```

Emacs 24.5.1 (Org mode 8.3.4)