The Kalman Filter and the Gradient of the Log Likelihood

Andrew M. Bradley   Last updated: May 28, 2014

This report shows how to efficiently compute the gradient of the log likelihood function for a linear quadratic estimator implemented by the Kalman filter.

Computing the maximum likelihood estimator requires an optimization algorithm. Gradient-based optimization algorithms generally greatly outperform non-gradient methods when a problem is differentiable: even gradient calculations that are expensive relative to the objective evaluation are justified. Techniques such as using multiple initial points can increase robustness to poor local optimizers. More generally, a global optimization method—gradient-based or not—can be used to get close to a good solution, and then a gradient-based optimization method can find the associated local optimizer.

The gradient can be computed by finite differences, in which case the forward computation must be done for each parameter that is varied, or analytically, in which case the work involved in the calculation must be considered.

In this report we use the *adjoint method* to compute the gradient using work that is a small factor—1 to 2—times the work to run the Kalman filter once if issues related to memory are not considered. Then we show how to use a *dynamic checkpointing* algorithm to compute the gradient and the RTS smoother efficiently given a fixed maximum amount of storage. A small increase in the number of computations is traded for a large decrease in the maximum memory size.

1 The Kalman filter

We write the Kalman filter as follows. Time indices k are supressed whereever possible; a simple subscript $-$ indicates the subscripted quantity is from the previous time step, $+$ from the next. It is customary and useful to divide the operations into the prediction step:

$$\begin{aligned} x^p &= F x^f_- \\ P^p &= F P^f_- F^T + Q \end{aligned}$$

and the update or filter step:

$$\begin{aligned} z &= y - H x^p \\ S &= R + H P^p H^T \\ K &= P^p H S^{-1} \\ x^f &= x^p + K z \\ P^f &= (I - KH) P^p. \end{aligned}$$

The data log likelihood is a function of $\log\det S$ and $z^T S^{-1} z$; the usual log likelihood is

$$p \equiv \log \tilde{p}(y|Y_-) = -\frac{1}{2}(N_o \log 2\pi + \log\det S + z^T S^{-1} z), \tag{1}$$

where Y_- are the data up to the previous time index, $y \in \mathbb{R}^{N_o}$, and $\tilde{p}$ is the probability density function.

1.1 Computations

We implement a square-root Kalman filter using the QR factorization. The prediction step is straightforward and uses a Cholesky factorization. The arguments are F, Q, x^f_-, and $(P^f_-)_c$, which is the Cholesky factorization of P^f_-. Let N_s be the number of states and N_o the number of observations. The prediction step requires $O(N_s^3)$ work.

```
function [xp Ppc] = kf_qrsc_predict (F, Q, xf, Pfc)
% On input,
%   F is the state transition matrix.
%   Q is the process covariance.
%     xf = x_k-1|k-1
%     Pfc = chol(P_k-1|k-1).
% On output,
%     xp = x_k|k-1
%     Ppc = chol(P_k|k-1).
   xp = F*xf;
   PfcFp = Pfc*F';
   % PfcFp'*PfcFp is assuredly p.d., so a simple chol works.
   Pp = PfcFp'*PfcFp + Q;
   Ppc = chol(Pp);
end
```

For

$$A = \begin{pmatrix} R_c & 0 \\ P_c^p H^T & P_c^p \end{pmatrix},$$

the Schur complement of $(A^T A)(1{:}N_o,1{:}N_o)$ in $A^T A$ is $P^f = P^p - P^f H^T S^{-1} H P^p$, where $S = HP^pH^T + R$. Symmetric positive definiteness of the covariance matrices, and so numerical stability, is maintained because the QR factorization of A yields a factor A_c such that $P_c^f \equiv A_c(N_o+1{:}N_o+N_s,N_o+1{:}N_o+N_s)$, where $P^f = (P_c^f)^T P_c^f$, and $S_c \equiv A_c(1{:}N_o,1{:}N_o)$, where $S = S_c^T S_c$. The QR factorization requires $O((N_o+N_s)^3)$ work.

```
function [xf Pfc z Sc] = kf_qrsc_update (H, Rc, y, xp, Ppc)
% On input,
%     H is the state -> observation matrix.
%     Rc is chol(R), where R is the observation covariance matrix.
%     y is the vector of observations.
%     xp = x_k|k-1
%     Ppc = chol(P_k|k-1).
% On output,
%     xf = x_k|k
%     Pfc = chol(P_k|k)
%     z = y - H*xp
%     Sc = chol(H P_k|k-1 H' + R) [if requested].
   [n m] = size(H);
   A = [full(Rc) zeros(n,m); Ppc*H' Ppc];
   [~,Pfc] = qr(A,0);
   % If requested, extract chol(S).
   if (nargout > 3)
      Sc = Pfc(1:n,1:n);
      mask = diag(Sc < 0);
      Sc(mask,:) = -Sc(mask,:);
   end
   % Extract the part of chol(A) that is chol(P_k|k), ie, the
   % factorization of the Schur complement of interest.
   Pfc = Pfc(n+1:n+m,n+1:n+m);
   mask = diag(Pfc < 0);
   Pfc(mask,:) = -Pfc(mask,:);
   % Innovation
   z = y - H*xp;
   % Filtered state
   xf = xp + Pfc'*(Pfc*(H'*(Rc \ (Rc' \ z))));
end
```

2 The gradient

The gradient is obtained using Lagrange multipliers, one set for each Kalman filter relation at each time index. Workers studying ODE- and PDE-constrained optimization and automatic differentiation call this approach the *adjoint method* because in those settings it entails solving linear systems involving the adjoints of the equations obtained from a linearization of the forward problem. The Lagrange multipliers are the solution to the *ajdoint problem*, which evolves in the reverse time order of the forward problem.

The gradient is a total derivative. We write the total derivative of f with respect to the vector of parameters a as $d_a f$ and the partial derivative as f_a.

2.1 Small and large matrices

We shall need to calculate partial derivatives of matrix functions with respect to matrices, and so we need to decide on a means to organize the operations.

A *small* matrix is one of the original matrices like F or Q.

The *vectorization* operation $\overline{F}$ (a line on top of F) forms a vector that stacks the columns of F. The *unvectorization* operation $\underline{v}$ does the opposite: it maps v to a square matrix, filling the columns sequentially; $\underline{v}$ is undefined if $v \in \mathbb{R}^n$ and n is not a square number, but of course we avoid such a case here. In MATLAB, these operations are `A(:)` and `n = sqrt(numel(A)); reshape(A,n,n)`.

A partial derivative of a scalar quantity with respect to multiple parameters forms a row vector. The partial derivative of $\overline{F}$ with respect to $\overline{Q}$ is a *large* matrix in which row i corresponds to the partial derivatives $(\overline{F}_i)_{\overline{Q}}$, a row vector.

As an example of these operations, for $A \in \mathbb{R}^{N \times N}$, $\overline{A} \in \mathbb{R}^{N^2 \times 1}$ and $\overline{A}_{\overline{A}} = I \in \mathbb{R}^{N^2 \times N^2}$.

Rewriting $(\log \det C)_C = C^{-1}$ using the vectorization operation yields $(\log \det C)_{\overline{C}} = \overline{C^{-1}}^T$. Similarly, $(z^T C^{-1} z)_{\overline{C}} = -\overline{qq^T}^T$ for $q \equiv C^{-1} z$.

An alternative convention is to use index notation. For example, later we shall write

$$\bar{\tilde{\eta}}^T \overline{f}_{\overline{P^P}} = \bar{\tilde{\eta}}^T X \overline{f}_{\overline{P^P}} = \frac{1}{2} \overline{(\tilde{\eta} + \tilde{\eta}^T)} \overline{f}_{\overline{P^P}} = \overline{\eta}^T \overline{f}_{\overline{P^P}}.$$

We can write this using index notation as

$$\tilde{\eta}_{ij} (f_{ij})_{P^p_{mn}} = \tilde{\eta}_{ij} X_{ijop} (f_{op})_{P^p_{mn}} = \frac{1}{2} (\tilde{\eta}_{ij} + \tilde{\eta}_{ji}) (f_{ij})_{P^p_{mn}} = \eta_{ij} (f_{ij})_{P^p_{mn}}.$$

Which form is preferable is a matter of taste. We prefer the first for two reasons. First, formulas are not cluttered with indices. Second, operations map directly to BLAS level 2 and 3 routines. A formula using index notation must eventually be transformed, at least implicitly, to one using vectorization notation when it is programmed.

2.2 Symmetrization

Certain of the Lagrange multipliers can inherit the symmetry of the covariance matrices if the derivatives are suitably arranged. Symmetry in these multipliers is very beneficial for numerical stability because analytical symmetry allows us to enforce numerical symmetry, which stabilizes finite-precision computations; a square-root Kalman filter uses the same idea as part of enforcing positive definiteness. The algebraic manipulations necessary to obtain symmetry can be pushed to the final part of each calculation. We make some preliminary observations for later use.

Let $S \in \times R^{N \times N}$ be a symmetric matrix, and consider the total derivative $d_a \overline{S}$. S has $K = N(N+1)/2$ independent entries rather than N^2. Let these independent degrees of freedom be $\sigma \in \mathbb{R}^{K \times 1}$, and write $S(\sigma)$. Then $d_a \overline{S} = \overline{S}_\sigma \; d_a \sigma$.

We can decompose $\overline{S}_\sigma$ into the product of two matrices $X \in \mathbb{R}^{N^2 \times N^2}$, $Y^\sigma \in \mathbb{R}^{N^2 \times K}$: $\overline{S}_\sigma = X Y^\sigma$, where for square A, the symmetric large matrix X operates on A so that $\underline{X\overline{A}} = \frac{1}{2}(A + A^T)$.

X can be written as $\frac{1}{2}(I + T)$, where $\overline{A^T} = T\overline{A}$. T is symmetric. For if element (i, j) of $\underline{T\overline{A}}$ is element (m, n) of A, then element (m, n) of $\underline{T\overline{A}}$ is element (i, j) of A.

Y^σ is the same as the matrix $\overline{S}_\sigma$ except that any row corresponding to S_{ij} for $i > j$ (it could be defined with the opposite inequality) is all zero.

Let f be a scalar function of S. Then

$$d_a f(S) = f_{\overline{S}} \; \overline{S}_\sigma \; d_a \sigma = f_{\overline{S}} \; X \; Y^\sigma \; d_a \sigma = \frac{1}{2} \overline{(\underline{f_{\overline{S}}} + \underline{f_{\overline{S}}}^T)} \; Y^\sigma \; d_a \sigma.$$

If every column of a large matrix M is a symmetric small matrix when vectorized, then $M = XM$ since X is idempotent when acting on a vectorized symmetric matrix.

A property of the Kronecker product is that $(A \otimes B)\overline{V} = \overline{BVA^T}$.

If $A = A^T$, then $(A \otimes A)X = X(A \otimes A)$. Proof: Let $v \equiv \overline{V}$ and similarly for u, and $u \equiv Xv (= \frac{1}{2}\overline{(U + \overline{U}^T})$. First, $(A \otimes A)Xv = (A \otimes A)u = \overline{AUA}$. Second, $X(A \otimes A)v = X\overline{AVA} = \frac{1}{2}(\overline{AVA + (AVA)^T}) = \frac{1}{2}\overline{A(V + V^T)A} = \overline{AUA}$, as in the first case.

2.3 The Lagrangian

Let us consider a simple problem to illustrate the overall structure of the calculation. Suppose we have the scalar function $f(x,a)$, where $x \in \mathbb{R}^{N\times 1}$ and a is a vector of parameters, subject to $g(x,a)=0$ and we want $d_a f$. Define the Lagrangian $L \equiv f(x,a) + \lambda^T g(x,a)$. As $g(x,a)=0$, the total derivative $d_a f = d_a L = f_a + f_x d_a x + \lambda^T(g_a + g_x d_a x) = f_a + \lambda^T g_a + (f_x + \lambda^T g_x) d_a x$. If λ is chosen so that $f_x + \lambda^T g_x = 0$, then $d_a f = f_a + \lambda^T g_a$. λ is found by solving the *adjoint equation* $g_x^T \lambda = -f_x^T$.

2.3.1 The initial setup

The Lagrangian for the log likelihood is obtained by associated a vector or matrix of Lagrange multipliers with each relation in the Kalman filter for each time index. First we encode the Kalman filter in a set of equations:

$$\begin{aligned}
b &\equiv x^p - F x^f_- = 0\\
g &\equiv P^p - F P^f_- F^T - Q = 0\\
s &\equiv S - R - H P^p H^T = 0\\
c &\equiv x^f - x^p - Kz = 0\\
f &\equiv P^f - (I - KH)P^p = 0.
\end{aligned}$$

Then we define the Lagrangian; here it's useful to make the T time indices k explicit:

$$L \equiv \sum_{k=1}^{T} p_k + \overline{\tilde{\lambda}}_k^T \overline{s}_k + \overline{\tilde{\mu}}_k^T \overline{g}_k + \overline{\tilde{\eta}}_k^T \overline{f}_k + \beta_k^T b_k + \gamma_k^T c_k.$$

The tildes on $\tilde{\lambda}$, $\tilde{\mu}$, and $\tilde{\eta}$ are meant to indicate that these multipliers will be replaced by undecorated versions shortly when we symmetrize the problem. Next we obtain the total derivative with respect to a vector of parameters a whose role is as yet unspecified. In what follows we suppress all time-index subscripts for the current time index:

$$\begin{aligned}
d_a L = \sum_{k=1}^{T} & p_{\overline{S}} d_a \overline{S} + p_{x^p} d_a x^p + \\
& \overline{\tilde{\lambda}}^T (\overline{s}_a + s_{\overline{S}} d_a \overline{S} + s_{\overline{P^p}} d_a \overline{P^p}) + \\
& \overline{\tilde{\mu}}^T (\overline{g}_a + \overline{g}_{\overline{P^p}} d_a \overline{P^p} + \overline{g}_{\overline{P^f}_{k-1}} d_a \overline{P^f}_{k-1}) + \\
& \overline{\tilde{\eta}}^T (\overline{f}_a + \overline{f}_{\overline{P^p}} d_a \overline{P^p} + \overline{f}_{\overline{P^f}} d_a \overline{P^f} + \overline{f}_{\overline{S}} d_a \overline{S}) + \\
& \beta^T (b_a + b_{x^p} d_a x^p + b_{x^f_{k-1}} d_a x^f_{k-1}) + \\
& \gamma^T (c_a + c_{x^p} d_a x^p + c_{x^f} d_a x^f + c_{\overline{P^p}} d_a \overline{P^p} + c_{\overline{S}} d_a \overline{S}),
\end{aligned}$$

where setting $P^f_0 = 0$ and $x^f_0 = 0$ allows all iterations to appear the same. Then we factor out the total derivatives:

$$\begin{aligned}
d_a L = \sum_{k=1}^{T} & \overline{\tilde{\lambda}}^T \overline{s}_a + \overline{\tilde{\mu}}^T \overline{g}_a + \overline{\tilde{\eta}}^T \overline{f}_a + \beta^T b_a + \gamma^T c_a + \\
& (p_{\overline{S}} + \overline{\tilde{\lambda}}^T \overline{s}_{\overline{S}} + \overline{\tilde{\eta}}^T \overline{f}_{\overline{S}} + \gamma^T c_{\overline{S}})\ d_a \overline{S} + \\
& (\overline{\tilde{\lambda}}^T \overline{s}_{\overline{P^p}} + \overline{\tilde{\mu}}^T \overline{g}_{\overline{P^p}} + \overline{\tilde{\eta}}^T \overline{f}_{\overline{P^p}} + \gamma^T c_{\overline{P^p}})\ d_a \overline{P^p} + \\
& (\overline{\tilde{\eta}}^T \overline{f}_{\overline{P^f}} + \overline{\tilde{\mu}}_{k+1}^T (g_{k+1})_{\overline{P^f}})\ d_a \overline{P^f} + \\
& (p_{x^p} + \beta^T b_{x^p} + \gamma^T c_{x^p})\ d_a x^p + \\
& (\beta_{k+1}^T (b_{k+1})_{x^f} + \gamma^T c_{x^f})\ d_a x^f,
\end{aligned}$$

where $\tilde{\mu}_{T+1} = 0$ and $\beta_{T+1} = 0$.

2.3.2 Symmetrization

At this point we want to modify the problem so that the Lagrange multipliers λ, μ, and η are symmetric matrices.

Later we shall show how to compute each partial derivative. For now we take it as given that each large matrix with which the tilded multipliers are multiplied have one of the following three properties.

1. The large matrix is I. This is true of, for example, $s_{\overline{S}}$. Then $\overline{\tilde{\lambda}}^T s_{\overline{S}} X = \overline{\tilde{\lambda}}^T IX = \overline{\tilde{\lambda}}^T X = \overline{\lambda}^T$.
2. It has columns that are all vectorizations of small symmetric matrices. For example, in $\overline{\tilde{\eta}}^T \overline{f}_{\overline{P^p}}$, each column of $\overline{f}_{\overline{P^p}}$ is a vectorized small symmetric matrix. Therefore, $\overline{f}_{\overline{P^p}} = X\overline{f}_{\overline{P^p}}$, and similarly for the others. Let $\tilde{\lambda} = X\overline{\tilde{\lambda}}$ and similarly for μ and η. Then, for example, $\overline{\tilde{\eta}}^T \overline{f}_{\overline{P^p}} = \overline{\tilde{\eta}}^T X\overline{f}_{\overline{P^p}} = \overline{\eta}^T \overline{f}_{\overline{P^p}}$.
3. If A is symmetric, then $(A \otimes A)X = X(A \otimes A)$, as we showed earlier. Therefore, in the partial derivatives of matrix-valued functions that use this operation, X can be shifted left.

Furthermore, we can assume quite sensibly that each of s_a, f_a, and g_a are small symmetric matrices, as otherwise varying the parameter a would cause a covariance matrix to become unsymmetric. Hence, for example, $\overline{s}_a = X\overline{s}_a$.

Next, recall that, for example, $d_a\overline{S} = S_\sigma XY^\sigma d_a\sigma$ for σ a vector of independent elements of the symmetric matrix S. Corresponding to σ for S, we introduce ρ for P^p and ζ for P^f. Because S, P^p, and P^f are of the same size, the same matrices X and Y^σ relate the total derivative of the matrix to the total derivative of its vector of independent elements.

Putting these ideas together, we rewrite the gradient as

$$\begin{aligned} d_aL = \sum_{k=1}^{T} & \overline{\lambda}^T \overline{s}_a + \overline{\mu}^T \overline{g}_a + \overline{\eta}^T \overline{f}_a + \beta^T b_a + \gamma^T c_a + \\ & (p_{\overline{S}} + \overline{\lambda}^T + \overline{\eta}^T \overline{f}_{\overline{S}} + \gamma^T c_{\overline{S}})\ XY^\sigma d_a\sigma + \\ & (\overline{\lambda}^T \overline{s}_{\overline{P^p}} + \overline{\mu}^T + \overline{\eta}^T \overline{f}_{\overline{P^p}} + \gamma^T c_{\overline{P^p}})\ XY^\rho d_a\rho + \\ & (\overline{\eta}^T + \overline{\mu}_{k+1}^T (g_{k+1})_{\overline{P^f}})\ XY^\zeta d_a\zeta + \\ & (p_{x^p} + \beta^T b_{x^p} + \gamma^T c_{x^p})\ d_a x^p + \\ & (\beta_{k+1}^T (b_{k+1})_{x^f} + \gamma^T c_{x^f})\ d_a x^f. \end{aligned}$$

The following algebraic manipulations were used:

1. In the first line, $\overline{\tilde{\lambda}}^T \overline{s}_a = \overline{\tilde{\lambda}}^T X\overline{s}_a = \overline{\lambda}^T \overline{s}_a$, and similarly for $\overline{\mu}^T \overline{g}_a$ and $\overline{\eta}^T \overline{f}_a$.
2. In each of the first three parenthesized expressions, $\overline{\tilde{\lambda}}^T s_{\overline{S}} X = \overline{\tilde{\lambda}}^T IX = \overline{\tilde{\lambda}}^T X = \overline{\lambda}^T$ and similarly for $\overline{\mu}^T$ and $\overline{\eta}^T$ by property 1 above.
3. In these same expressions, $\overline{\tilde{\eta}}^T \overline{f}_{\overline{P^p}} = \overline{\tilde{\eta}}^T X\overline{f}_{\overline{P^p}} = \overline{\eta}^T \overline{f}_{\overline{P^p}}$ by property 2 above.
4. In these same expressions, $\overline{\lambda}^T X = \overline{\lambda}^T$ and similarly for $\overline{\mu}^T$ and $\overline{\eta}^T$, which allows us to keep X factored out by property 2 above.
5. In these same expressions, $\overline{\tilde{\eta}}^T \overline{f}_{\overline{S}} = \overline{\tilde{\eta}}^T X\overline{f}_{\overline{S}} = \overline{\eta}^T \overline{f}_{\overline{S}}$ and similarly for $\overline{\lambda}^T \overline{s}_{\overline{P^p}}$ and $\overline{\mu}_{k+1}^T (g_{k+1})_{\overline{P^f}}$ by property 3 above.

If we choose the Lagrange multipliers so that every expression in parentheses is 0, then the gradient

$$g \equiv d_a \sum_{k=1}^{T} q = d_aL = \sum_{k=1}^{T} \overline{\lambda}^T \overline{s}_a + \overline{\mu}^T \overline{g}_a + \overline{\eta}^T \overline{f}_a + \beta^T b_a + \gamma^T c_a. \tag{2}$$

The steps to calculate the gradient are as follows:

1. Set $\mu_{T+1} = 0$ and $\beta_{T+1} = 0$.
2. For k from T down to 1:
   (a) Solve for η_k: $\overline{\eta}^T \overline{f}_{\overline{P^f}} + \overline{\mu}_{k+1}^T (g_{k+1})_{\overline{P^f}} X = 0$
   and for γ_k: $\beta_{k+1}^T (b_{k+1})_{x^f} + \gamma^T c_{x^f} = 0$.
   (b) Solve for λ_k: $p_{\overline{S}} + \overline{\lambda}^T \overline{s}_{\overline{S}} + \overline{\eta}^T \overline{f}_{\overline{S}} + \gamma^T c_{\overline{S}} = 0$.
   (c) Solve for μ_k: $\overline{\lambda}^T \overline{s}_{\overline{P^p}} + \overline{\mu}^T \overline{g}_{\overline{P^p}} + \overline{\eta}^T \overline{f}_{\overline{P^p}} + \gamma^T c_{\overline{P^p}} X = 0$
   and for β_k: $p_{x^p} + \beta^T b_{x^p} + \gamma^T c_{x^p} = 0$.

2.4 Computations

To be efficient, the gradient calculation must take at most a small constant of about 1 times the time to run the Kalman filter. Recall that each step of the QR-based square-root Kalman filter requires $O((N_s + N_o)^3)$ work. We describe each computation necessary to calculate the gradient and find that the work per time step is $O(N_s^2 N_o + N_s N_o^2)$ and so is efficient.

The simple partial derivatives are these:

$$\begin{aligned}
\overline{s}_{\overline{S}} &= \overline{g}_{\overline{P^p}} = \overline{f}_{\overline{P^f}} = I \\
b_{x^p} &= c_{x^f} = I \\
(b_k)_{x^f_{k-1}} &= -F_k \\
c_{x^p} &= -I + KH.
\end{aligned}$$

Partial derivatives in the parameter a of course depend of the particular model.

Two operations necessary to compute certain partial derivatives are important to implement correctly. They involve matrix-vector products with large matrices that either are numerically dense but have low-rank structure or have many structural zeros. The first computes $y^T = \overline{\nu}^T (\overline{ABA^T})_{\overline{A}}$ for $B = B^T$, $\nu = \nu^T$. If the small matrices $A, B \in \mathbb{R}^{N \times N}$, then the large matrix $(\overline{ABA^T})_{\overline{A}} \in \mathbb{R}^{N^2 \times N^2}$. However, this large matrix has many structural zeros. Taking advantage of these, the matrix-vector product can be implemented as $y = 2\overline{\nu AB}$, which is an $O(m^2 n + mn^2)$ operation for $A \in \mathbb{R}^{m \times n}$. As $B = B^T$, $(ABA^T)_{A_{ij}}$ is symmetric. Hence each column of $(ABA^T)_A$ is the vectorization of a symmetric small matrix, and so $(ABA^T)_A = X = (ABA^T)_A$.

The second operation computes $y^T = \overline{\nu}^T (A \otimes A)$, where $\otimes$ is the Kronecker product. If the small matrix A is completely dense, then so is the large matrix $A \otimes A$. Fortunately, the large matrix has low-rank structure that again makes the matrix-vector product cubic rather than quartic in the size of A. The operation can be implemented as $y = \overline{A^T \nu A}$, which is also an $O(m^2 n + mn^2)$ operation. We have already shown that if A is symmetric, then $X(A \otimes A) = (A \otimes A)X$, a property necessary for symmetrization. We shall use the operation $y^T = \overline{\nu}^T (A \otimes A)$ frequently, and so we write $A_{\otimes}(\nu, A) \equiv \overline{\nu}^T (A \otimes A)$.

Each term in which the Lagrange multiplier is a small matrix involves either the operation $A_{\otimes}$ for symmetric A or a partial derivative of the form $(ABA^T)_A$. Hence the symmetrization of the terms in the Lagrangian can be implemented in the manner we discussed.

Now we complete the calculations:

$$\begin{aligned}
(z^T C^{-1} z)_C &= -qq^T, \quad \text{where } q = C^{-1} z \\
\overline{\nu}^T \overline{C^{-1}}_{\overline{C}} &= -A_{\otimes}(\nu, C^{-1}) \\
\overline{\nu}^T (\overline{ABA^T})_{\overline{B}} &= A_{\otimes}(\nu, A) \\
\overline{\nu}^T (\overline{AB^{-1}A^T})_{\overline{B}} &= \overline{\nu}^T (\overline{AB^{-1}A^T})_{\overline{B^{-1}}} \overline{B^{-1}}_{\overline{B}} = -A_{\otimes}(A_{\otimes}(\nu, A), B^{-1}).
\end{aligned}$$

For vectors v and b,

$$\begin{aligned}
(v^T A b)_A &= vb^T \\
\overline{(v^T A^{-1} b)_A} &= \overline{(v^T A^{-1} b)_{A^{-1}}}^T \overline{A^{-1}}_{\overline{A}} = -A_{\otimes}(vb^T, A^{-1}).
\end{aligned}$$

Finally, we uses these calculations to implement the remaining partial derivatives:

$$
\begin{aligned}
\overline{\mu}_{k+1}^T(\overline{g}_{k+1})_{\overline{P_k^f}} &= -\overline{\mu}_{k+1}^T(\overline{F_{k+1}P_k^fF_{k+1}^T})_{\overline{P}_k^f} = -A_{\otimes}(\mu_{k+1}, F_{k+1}) \\
\beta_{k+1}^T(b_{k+1})_{x_k^f} &= -\beta_{k+1}^T F \\
\overline{\eta}^T\overline{f}_{\overline{S}} &= \overline{\eta}^T(\overline{P^pH^TS^{-1}HP^p})_{\overline{S}} = -A_{\otimes}(A_{\otimes}(\eta, P^pH^T), S^{-1}) \\
\gamma^T c_{\overline{S}} &= -\gamma^T P^pH^TS^{-1}z = A_{\otimes}((\gamma^T P^pH^T)^T z^T, S^{-1}) \\
\overline{\eta}^T\overline{f}_{\overline{P^p}} &= \overline{\eta}^T(-I + (\overline{P^pH^TS^{-1}HP^p})_{\overline{P^p}} = -\overline{\eta}^T + 2\overline{\eta P^pV^TV}, \quad V = S_c^{-T}H \\
\overline{\lambda}^T\overline{s}_{\overline{P^p}} &= -\overline{\lambda}^T(\overline{HP^pH^T})_{\overline{P}^p} = A_{\otimes}(\lambda, H) \\
\gamma^T c_{\overline{P^p}} &= -\gamma^T(\overline{P^pH^TS^{-1}z})_{\overline{P^p}} = -\gamma H^TS^{-1}z \\
\gamma^T c_{x^p} &= -\gamma^T + \gamma^T P^pV^TV \\
p_z z_{x^p} &= -p_z H.
\end{aligned}
$$

If p is the log likelihood of the normal distribution and $q = S^{-1}z$,

$$p_{\overline{S}} = -\frac{1}{2}((\log\det S)_{\overline{S}} + (z^TS^{-1}z)_{\overline{S}}) = -\frac{1}{2}(\overline{S^{-1}}^T - \overline{qq^T}^T) \tag{3}$$

$$p_z = -z^TS^{-1}. \tag{4}$$

2.5 Code

We provide a Matlab implementation of these calculations. First, the individual computations follow.

```
function p = Calc_vt_kronAA (v, A, vissym)
% For v = v' if vissym, general v otherwise, compute
%     p = v(:)'*kron(A,A);
% efficiently. This straightforward version is O(N^2 M^2) for [M N] =
% size(A). The efficient version that follows is O(m^2 n + n^2 m).
  [m n] = size(A);
  if (vissym)
    v = reshape(v, m, m);
    v = v - diag(diag(v))/2;
    p = A'*triu(v)*A;
    p = p + p';
  else
    p = A'*(reshape(v, m, m)*A);
  end
  p = p(:)';
end

function v = Calc_vt_invC_C (v, Ci)
% For C = R' R, return
%     z(:)' inv(C)_C.
  v = -Calc_vt_kronAA(v, Ci, false);
end

function v = Calc_vt_ABAt_B (v, A)
% For B = B', v = v', return
%     v(:)' (A B A')_B.
  v = Calc_vt_kronAA(v(:), A, true);
end

function v = Calc_vtAb_A (v, b)
% Return (v' A b)_A.
  v = v(:)*b(:)';
end

function v = Calc_vtinvAb_A (v, Ai, b)
% Return (v' inv(A) b)_A for A = A'.
  n = size(Ai, 1);
  v = Calc_vtAb_A(v(:), b);
  v = reshape(Calc_vt_invC_C(v(:), Ai), n, n);
end

function v = Calc_vt_AinvBAt_B (v, A, Bi)
% For B = B', v = v', Bi = inv(B), return
%   v(:)' (A inv(B) A')_B.

  % v' (A inv(B) A')_B = v' [(A inv(B) A')_inv(B)] [inv(B)_B]
  v = Calc_vt_kronAA(v, A, true);
  v = -Calc_vt_kronAA(v, Bi, true);
end

function p = Calc_vt_ABAt_A (v, A, B)
% For B = B', v = v', return
%      v(:)' (A B A')_A.
% This is an O(m^2 n + m n^2) operation for [m n] = size(A).
  m = size(A, 1);
  v = reshape(v, m, m);
  % If v were unsymmetric, we would use this:
  %      p = (v' + v)*A*B;
  p = 2*v*A*B;
  p = p(:)';
end

function A = Sym (A)
% Return (A + A')/2.
  n = sqrt(numel(A));
  A = reshape(A, n, n);
  A = (A + A')/2;
  A = A(:)';
end
```

Putting these together, one step of the adjoint calculation is implemented as follows.

```
function [lambda mu eta beta gamma] = kf_grad(...
    Fp1,H,Sc,Si,p_S,p_z,z,Pp,mup1,betap1)
% [lambda mu eta beta gamma] = kf_grad(F,Fp1,H,Sc,Si,p_S,p_z,z,Pp,mup1,betap1)
% Carry out one time step of computing the gradient of p(S) wrt to the
% hyperparameters.
%   p(S) is something like a log likelihood.
%   S = Sc'*Sc, where Sc is from kf_qrsc_update.
%   Si = inv(S). Call Si = kf_grad_Calc_invC(Sc) to get Si.
%   F, Fp1, H are the usual Kalman filter matrices. Fp1 is F at the next time
% index.
%   p_S is the partial derivative of p wrt S. You will likely use
%      Si = kf_grad_Calc_invC(Sc)
%      kf_grad_Calc_LogDetC_C(Si)
%      kf_grad_Calc_ztinvCz_C(z,Sc)
% to compute p_S, where z = y - H*xp. For example, if p is just the usual log
% likelihood so that, neglecting a constant term,
%      p = - 1/2 log(det(S)) - 1/2 z' inv(S) z,
% then
%      p_S = -0.5*kf_grad_Calc_LogDetC_C(Si) - 0.5*kf_grad_Calc_ztinvCz_C(Sc,z).
%   p_z is the partial derivative of p wrt z = y - H*xp. For p as above,
%      p_z = -Sc \ (z' / Sc)' %= inv(S) z.
%   mup1, betap1 are mu and beta from step k+1, where this is currently step
% k. Set mu = zeros(N^2,1) and beta = zeros(N,1) for the first call to this
% function.
%   Pp is the prediction-step covariance matrix.
%   The outputs lambda and mu are used to calculate the gradient. The total
% derivative wrt hyperparameter a is
%      sum_{k = 1}^N -lambda_k' R_a(:) - mu_k' Q_a(:),
% where R_a, Q_a are partial derivatives of R and Q. In many cases, R_a and Q_a
% are almost entirely zero and the nonzero structure is known; it's important to
% take advantage of these facts in order to make the gradient computation
% efficient.
  [no ns] = size(H);
  ns2 = ns^2;
  no2 = no^2;
  vec = @(x) x(:);
  % Solve
  %      eta_k ' f_{Pf_k} + mu_{k+1}' g_{Pf_k} = 0
  % for eta.
  eta = Calc_vt_ABAt_B(mup1,Fp1); % -mu_{k+1}' g_Pf
  % Not needed because Calc_vt_ABAt_B takes care of it implicitly:
  %     eta = Sym(eta);
  % Solve
  %      beta_{k+1}' (b_{k+1})_{xf_k} + gamma_k' c_{xf_k} = 0
  % for gamma.
  gamma = betap1(:)'*Fp1; % -beta_{k+1}' b_xf
  % Solve
  %      p_{S_k} + lambda_k' s_{S_k} + eta_k ' f_{S_k} + gamma_k' c_{S_k} = 0
  % for lambda.
  lambda = -p_S(:)' +...
           -Calc_vt_AinvBAt_B(eta, Pp*H', Si) +...       % -eta ' f_S
           vec(Calc_vtinvAb_A((gamma*Pp)*H', Si, z))'; % -gamma' c_S
  lambda = Sym(lambda);
  % Solve
  %      lambda_k' s_{Pp_k} + mu_k' g_{Pp_k} + eta_k ' f_{Pp_k} +
  %        gamma_k' c_{Pp_k} = 0
  % for mu.
  V = Sc' \ H;
  Siz = Sc \ (z' / Sc)';
  mu = eta - Calc_vt_ABAt_A(eta, Pp, V'*V) +... % -eta ' f_Pp
       Calc_vt_ABAt_B(lambda, H) +...             % -lambda ' s_Pp
       vec(Calc_vtAb_A(gamma, H'*Siz))';          % -gamma c_Pp
  mu = Sym(mu);
  % Solve
  %      p_{xf_k} + beta_k' b_{xp_k} + gamma_k' c_{xp_k} = 0
  % for beta.
  beta = gamma - ((gamma*Pp)*V')*V +... % -gamma' c_xp
         p_z(:)'*H;                     % -p_z z_xp
end
```

3 Storage associated with the adjoint method

A practical loss of speed could result from having apparently to store at least the factorization of each P^p_k; for large problems, such storage makes disk I/O necessary.

One might think that an alternative is to reverse the per-step computations and thereby have only to store a small amount of data independent of the number of time steps. However, reversing the Kalman filter is numerically unstable; moreover, numerical tests show loss of stability occurs quickly, so it seems unlikely a simple method of stabilizing the computation (e.g., adding a factor times identity to a covariance matrix that has lost definiteness) will work in general. The key observation is that the forward covariance computations can be written as

$$\begin{aligned} P^p &= F P^f_- F^T + Q \\ (P^f)^{-1} &= (P^p)^{-1} + H^T R^{-1} H. \end{aligned}$$

Hence the forward computations include summing two positive definite (pd) matrices and taking the inverse of the sum of the inverses of two pd matrices. These operations are inherently stable because in both cases two pd matrices are summed; regardless of their values, as long as they are pd, the outputs are pd, and so the computations are stable. Reversing these operations requires something like these operations:

$$\begin{aligned} (P^p)^{-1} &= (P^f)^{-1} - H^T R^{-1} H \\ P^f_- &= F^{-1}(P^p - Q) F^{-T}. \end{aligned}$$

In both steps a pd matrix is subtracted from another. Indefiniteness can result due to finite precision arithmetic even if the true result is pd. Moreover, if R, say, is sufficiently large relative to $(P^f)^{-1}$, then

$(P^p)^{-1}$ actually is indefinite in exact arithmetic. Hence these computations on their own need not produce pd outputs: even in exact arithmetic, external knowledge must be provided to prove that the output matrices are pd.

A better alternative is to use a checkpointing procedure; the problem of finding the gradient of the likelihood computed by a Kalman filter fulfills the conditions that make Griewank's `revolve` provably optimal. As an example, if one has 3650 time steps and memory to hold only 100 covariance matrices, `revolve` requires extra forward computations that produce a slowdown of less than a factor of 2; with just 10 matrices, less than 5. This factor should be multiplied with the factor associated with the adjoint computation. Hence it is quite likely that we can in practice compute the gradient in between 4 and 10 times as long as running the basic Kalman filter with no storage.

We can use this same procedure when running the RTS smoother.

Greiwank's program `revolve` was downloaded from TOMS and a mex interface written. We compared two versions of the gradient calculation. Our experience is that in general the non-checkpointing version with disk I/O is faster than the checkpointing version using main memory but number of checkpoints less than the number of time steps. Hence we believe the only reasons to use the checkpointing version are (1) the amount of disk storage necessary is greater one has or (2) I/O is particularly slow relative to FLOPS and memory access.

4 An example

The package `kfgs` contains these Matlab routines and routines to compute the Kalman filter, smoother, and likelihood gradient. We show an example of usage. First, create a random problem using this routine:

```
function p = RandProb (m, n, nt)
% m states, n observations, nt time steps.
  p.F = randn(m);
  p.H = randn(n, m);
  p.Pp0 = randn(m); p.Pp0 = p.Pp0'*p.Pp0; p.Pp0c = chol(p.Pp0);
  p.x0 = randn(m, 1);
  p.Y = randn(n, nt);
  p.Q = randn(m); p.Q = p.Q'*p.Q;
  p.R = randn(n); p.R = p.R'*p.R; p.Rc = chol(p.R);
end
```

Create a problem instance:

```
% Set up a random problem. p holds F, H, Q, R, etc.
Ns = 10;  % number of states
No = 5;   % number of observations
Nt = 100; % number of time steps
Nstack = 20; % number of checkpoint slots
p = RandProb(Ns, No, Nt);
```

In this example problem, we will consider two cases: first, the gradient of the log-likelihood function with respect to the diagonal elements of R and Q; second, the gradient with respect to a scalar multiple of R and another of Q. In this second case of only two optimization variables, a finite-difference gradient approximation is just as efficient or more efficient than `kfgs`, but it is meant only as illustration. In the first case, for most values of `Ns` and `No`, `kfgs` is far more efficient than a finite-difference gradient. In this example, control this choice as follows:

```
% 1 or 2. Test different types of parameterizations.
p.grad_test = 2;
```

We use checkpointing with checkpoints stored in memory. Initialize the memory buffer:

```
r = kf_rcd('init', 'mem');
```

The core routines are `kf_loglik_grad` and `kf_loglik_grad_cp`, where the first stores data for all time steps and the second uses checkpointing. Each requires the memory buffer, a function handle that we describe in a moment, and problem size data. The checkpointing version additionally requires the maximum permitted buffer size. (In Matlab, type `help function-name` for all these routines for usage details.)

```
[f g] = kf_loglik_grad_cp(...
    r, @(varargin)kl_fn(p, varargin{:}), Ns, Nt, ...
    Nstack*(Ns+1)*Ns*8);
```

An optional function handle that is called at each time step allows the client access to all the data, e.g., for recording.

`kl_fn` implements the client's state-space model. It returns data at the request of `dfgs`. `key` indicates the type of data. `tidx` is the time index (starting at 1). `varargin` contains input data associated with `key`.

```
function varargout = kl_fn (p, key, tidx, varargin)
  switch (key)
    case 'i'
      % Initial conditions.
      varargout(1:2) = {p.x0 p.Pp0c};

    case 'f'
      % State-space transition matrix.
      varargout{1} = p.F;

    case 'fq'
      % State-space transition matrix and process covariance.
      varargout(1:2) = {p.F p.Q};

    case 'h'
      % Observation matrix.
      varargout{1} = p.H;

    case 'hry'
      % Observation matrix, chol(observation covariance), observations at
      % time index tidx.
      varargout(1:3) = {p.H p.Rc p.Y(:, tidx)};

    case 'll'
      % Contribution to log-likelihood at time index tidx.
      [Sc z] = deal(varargin{1:2});
      q = z' / Sc;
      varargout{1} = -sum(log(diag(Sc))) - 0.5*q*q';

    case 'p'
      % Partial derivatives of the log-likelihood term at time index tidx
      % with respect to S = R + H Pp H' and z = y - H xp.
      [Sc Si z] = deal(varargin{1:3});
      p_S = -0.5*(kf_grad_Calc_LogDetC_C(Si) + kf_grad_Calc_ztinvCz_C(z, Sc));
      p_z = -Sc \ (z' / Sc)';
      varargout = {p_S p_z};

    case 'g'
      % Contribution of the term for time index tidx to the gradient of the
      % log-likelihood function.
      %   If tidx == 1, there was no prediction and so no Q. However, there was
      % a filter ('update').
      [lambda mu] = deal(varargin{1:2});
      switch (p.grad_test)
        case 1
          % Gradient wrt to diagonal elements of R and Q.
          [m n] = size(p.H);
          if (tidx > 1) mu1 = diag(reshape(mu, n, n));
          else          mu1 = zeros(n, 1); end
          varargout{1} = -[mu1; diag(reshape(lambda, m, m))].';
        case 2
          % Gradient wrt a scalar multiple of R and Q.
          if (tidx > 1) mu1 = mu(:)'*p.Q(:);
          else          mu1 = 0; end
          varargout{1} = -[mu1, lambda(:)'*p.R(:)];
      end
  end
end
```

Keys `i`, `f`, `fq`, `h`, and `hry` return data for the state-space model. Key `ll` implements (1) and key `p` implements (3) and (4). The routines `kf_grad_Calc_LogDetC_C` and `kf_grad_Calc_ztinvCz_C` are available in `kfgs` for convenience in this and similar calculations. Finally, key `g` implements (2). This calculation is the most complicated. In most problems, parameters (sometimes called hyperparameter) are used only in Q and R; hence only Lagrange multipliers λ and μ are used. Equations $s = 0$ (associated with λ) and $g = 0$ (from Section 2.3.1) respectively involve $-R$ and $-Q$. Therefore, for a parameter a, the partial derivatives $s_a = -R_a$ and $g_a = -Q_a$ in (2), and the other terms are 0. The form of R_a depends on how R depends on a; the example shows two cases. In the second, $R \equiv a\hat{R}$, where $\hat{R}$ is constant, and so $R_a = \hat{R}$, and similarly for Q. In the first case, each diagonal element is a parameter.