Efficient Kalman filter in Julia
25th International Conference Computing in Economics and Finance

Michel Juillard¹

Ottawa, June 29, 2019

The views expressed herein are ours and do not necessarily represent the views of Bank of France.

¹Banque de France, michel.juillard@mjui.fr
Introduction

- Estimation of unobservable components models and linearized DSGE models: log likelihood computed with Kalman filter
- Called repeatedly during estimation
- Most time consuming single algorithm
- Julia provides elegant ways to implement optimized code
- What is the scope for parallel computing?
Kalman filter

State space model:

\[ y_t = Z \alpha_t + \epsilon_t, \quad \epsilon_t \sim N(0, H) \]
\[ \alpha_{t+1} = T \alpha_t + R \eta_t, \quad \eta_t \sim N(0, Q) \]
\[ \alpha_1 \sim N(a_1, P_1) \]

Kalman filter recursion

\[ \nu_t = y_t - Z a_t \]
\[ F_t = Z P_{t|t-1} Z' + H \]
\[ a_{t+1|t} = T \left( a_{t|t-1} + P_t Z' F_t^{-1} \nu_t \right) \]
\[ P_{t+1|t} = T \left( P_{t|t-1} - P_{t|t-1} Z' F_t^{-1} Z P_{t|t-1} \right) T' + R Q R' \]
Optimization strategy

- In place computations
- Factorizing: never compute the same thing twice even if it occupies memory
- Avoid temporary matrix allocation (cf. $A \cdot B \cdot C$ needs $T_1 = B \cdot C$, $T_2 = A \cdot T_1$)
- BLAS and LAPACK are your friends
  - Matrix multiplication in place, $AB + C \rightarrow C$: gemm!()
  - Solving linear system with positive definite matrix using Choleksy decomposition: potrf!() and potrs()
Matrix multiplication in place

- $\alpha AB + \beta C \rightarrow C$:
  
  ```
gemm!(tA, tB, alpha, A, B, beta, C)
```

- Examples
  - $AB + 0.5C \rightarrow C$
    ```
gemm!(’N’, ’N’, 1.0, A, B, 0.5, C)
```
  - $2AB’ + 0.5C \rightarrow C$
    ```
gemm!(’N’, ’T’, 2.0, A, B, 0.5, C)
```
  - $2A’B’ \rightarrow C$
    ```
gemm!(’T’, ’T’, 2.0, A, B, 0.0, C)
```
Cholesky decomposition

- F is a symmetric matrix
- Cholesky decomposition: triangular matrix U such that $U'U = F$ (for uplo='U')
- LAPACK.potrf!(uplo, F) returns upper/lower triangular matrix in F
- Solution of $FX = A$: solves two triangular systems: $U'Y = A$, $UX = Y$
- LAPACK.potrs!(uplo, F, A) returns the solution of $FX = A$ using the Choleski decomposition of $F$ returned by LAPACK.potrf!()
- The determinant of matrix $F$ is

$$|F| = \sum_{i=1,...,n} U^2_{i,i}$$
Kalman filter recursion in Julia (I)

1. $\nu_t = y_t - Z a_{t|t-1}$
   - copy!(\(\nu\), \(y\))
   - gemv!(’N’, -1.0, \(Z\), \(a\), 1.0, \(\nu\))

2. $M_t = Z P_{t|t-1}$
   - mul!(\(M\), \(Z\), \(P\))

3. $F_t = M_t Z' + H$
   - copy!(\(F\), \(H\))
   - gemm!(’N’, ’T’, 1.0, \(M\), \(Z\), 1.0, \(F\))

4. $iF\nu_t = F_t^{-1} \nu_t$
   - potrf!(’U’, \(F\))
   - potrs!(’U’, \(F\), \(\nu\))

5. $a_{t|t} = a_{t|t-1} + M_t' iF\nu_t$
   - copy(a1, \(a\))
   - gemv!(’T’, 1.0, \(M\), \(\nu\), 1.0, \(a1\))
Kalman filter recursion in Julia (II)

6. $iFM_t = F_t^{-1}M_t$
   
   ```julia
   potrs!(’U’, F, M)
   ```

7. $P_{t|t} = P_{t|t-1} - M_t' iFM_t$
   
   ```julia
   copy!(P1, P)
   gemm!(’N’, ’T’, -1.0, M, iFM, 1.0, P1)
   ```

8. $a_{t+1|t} = T a_{t|t}$
   
   ```julia
   mul!(a, T, a1)
   ```

9. $QQ = RQR'$
   
   ```julia
   mul!(R1, R, Q)
   mul!(QQ, R1, Q)
   ```

10. $P_{t+1|t} = TP_{t|t}T' + QQ$
    
    ```julia
    copy!(P1, P)
    mul!(P2, T, P1)
    copy!(P, QQ)
    gemm!(’N’, ’T’, 1.0, P2, T, P)
    ```
Log likelihood

\[ LIK = \sum_{t=1}^{Nobs} -\frac{1}{2} \left( n \ln 2\pi + \ln |F_t| + \nu_t' F_t^{-1} \nu_t \right) \]

Julia implementation

```julia
lik[t] = log(det_from_cholesky(F))
+ LinearAlgebra.dot(ν, iFν)

lik_cst = nobs*n*log(2*pi)
LIK = -0.5*(lik_cst + sum(lik))
```
struct KalmanLikelihoodWs{T, U}  
    R1::Matrix{T}  
    QQ::Matrix{T}  
    ν::Vector{T}  
    M::Matrix{T}  
    F::Matrix{T}  
    iFν::Vector{T}  
    iFM::Matrix{T}  
    a1::Vector{T}  
    P1::Matrix{T}  
    P2::Matrix{T}  
    lik::Vector{T}  

function KalmanLikelihoodWs(ny::U, ns::U, np::U, nobs::U)
    R1::Matrix{T}(undef, ns, np)
    QQ::Matrix{T}(undef, ns, ns)
    ν::Vector{T}(undef, ny)
    M::Matrix{T}(undef, ny, ns)
    F::Matrix{T}(undef, ny, ny)
    iFν::Vector{T}(undef, ny)
    iFM::Matrix{T}(undef, ny, ns)
    a1::Vector{T}(undef, ns)
    P1::Matrix{T}(undef, ns, ns)
    P2::Matrix{T}(undef, ns, ns)
    lik::Vector{T}(undef, nobs)
    new(R1, QQ, ν, M, F, iFν, iFM, a1, P1, P2, lik)
end
end
Parallel computing

- Kalman filter algorithm is serial by nature: no scope for parallel computing
- Matrix multiplication can be computed in parallel
- On a computer with several cores, OpenBlas may split matrix multiplication in too many threads
- Example for a model with 60 state variables and 12 observables
  - 1 thread: 27.171 ms
  - 2 threads: 26.905 ms
  - 3 threads: 26.151 ms
  - 4 threads: 33.104 ms
  - 5 threads: 32.258 ms
  - 6 threads: 38.128 ms
  - 7 threads: 34.826 ms
  - 8 threads: 46.085 ms
- Use `BLAS.set_num_threads()`
- If there is scope to run several Kalman filters in parallel, allocate only one thread to OpenBlas
Additional functionalities

- Z as a selection matrix
- Missing observations
- Monitoring for steady state of the filter
- State filtering and forecasting
- Unobserved variables and innovations smoothing
- Chandrasekhar-Herbst fast recursion
- Durbin Koopman diffuse filter for non-stationary models
- Time varying state space models
- Derivative of log likelihood with respect with elements of system matrices
Package available on Gitlab

https://gitlab.com/MichelJuilllard/KalmanFilterTools
Conclusion

- For critical code, optimizing Julia code brings about significant benefits
- Pre-allocate workspace
- Use low level in place functions
- Try to reuse past computations
- Find the best setting for BLAS.set_num_threads()