MATH 8090: State-Space Models II

Whitney Huang, Clemson University

11/16/2026

Contents

Local Level Model . . . 2
Simulate data from local level models . . . 2
Compare observed series with the latent state . . . 2
Carrying out the Kalman filter . . . 3
Kalman filter: forecasting . . . 5
Forecast uncertainty intervals . . . 5
Nile River annual minima: missing observations . . . 6
Kalman smoothing . . . 9
Kalman smoothing: local-level model . . . 10
Parameter Estimation . . . 11
Generate Data . . . 11
Initial Estimates . . . 12
Function to Evaluate the Likelihood . . . 12
Estimation . . . 13
Global Temperature Example . . . 14
EM Algorithm Example . . . 16
Generate Data (same as Example 6.6) . . . 16
Initial Estimates . . . 16
Standard Errors . . . 18
Evaluate Likelihood at Estimates . . . 18
Display Summary of Estimation . . . 18
Bayesian Estimation for the Local-Level Model . . . 18
Generate Data . . . 19
Set up the Gibbs Sampler . . . 20
Progress Bar . . . 20
Summarize Posterior State Draws . . . 21
Plot the Results . . . 21
References . . . 22

Local Level Model

Simulate data from local level models

$$Y_t = X_t + W_t, \qquad W_t \sim \mathrm{N}(0, \sigma_W^2),$$
$$X_t = X_{t-1} + V_t, \qquad V_t \sim \mathrm{N}(0, \sigma_V^2).$$

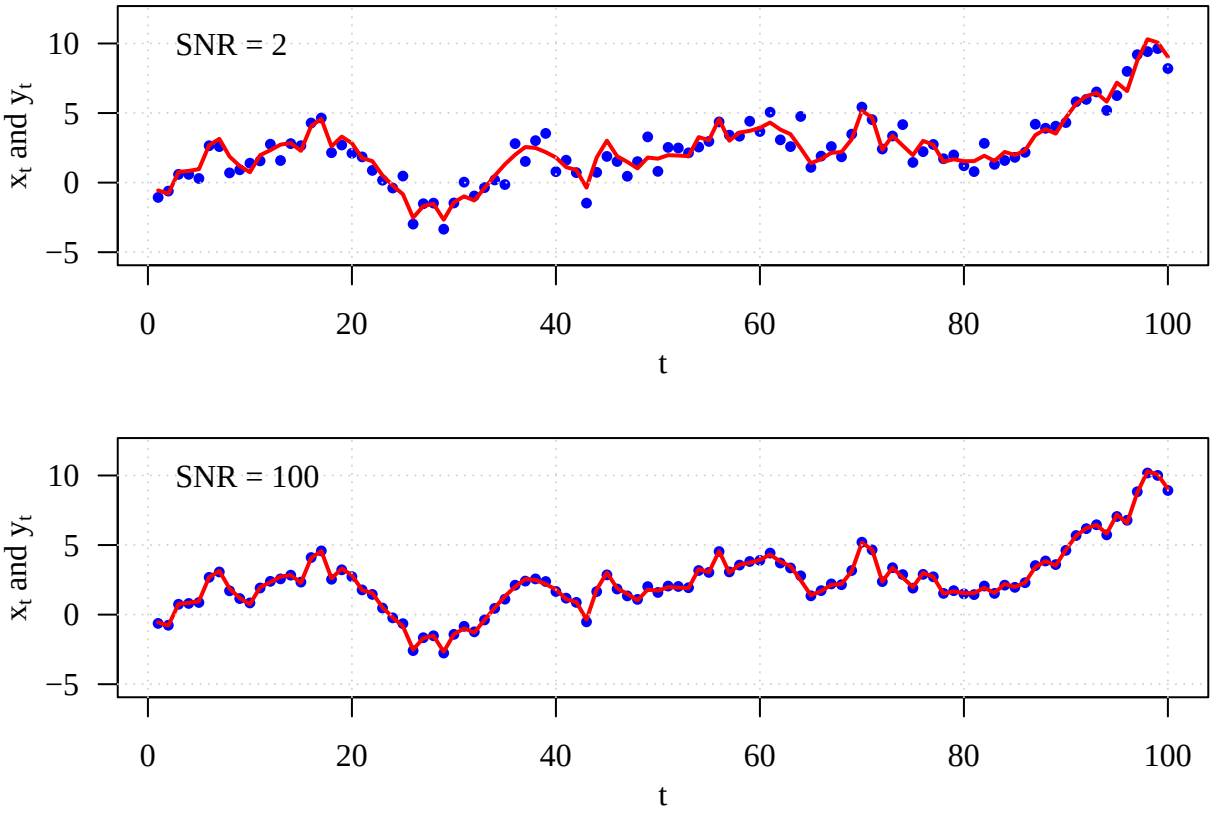
Here $\mu_0 = 0$, $\sigma_0^2 = 1$, $\sigma_V^2 = 1$, $\sigma_W^2 = \sigma_V^2/\mathrm{SNR}$.

```
set.seed(123)
# Same latent path; change only observation-noise variance
mu0 <- 0; sig0 <- 1; sig2.V <- 1
# First plotted state is X_1, with prior mean mu0 and variance sig0
X.1 <- rnorm(1, mean = mu0, sd = sqrt(sig0))
X <- cumsum(c(X.1, rnorm(99, sd = sqrt(sig2.V))))
W <- rnorm(100)
SNR <- 2; Y.2 <- X + W * sqrt(sig2.V / SNR)
SNR <- 100; Y.100 <- X + W * sqrt(sig2.V / SNR)
```

Compare observed series with the latent state

The red curve is the unobserved state. Blue points are measurements; a larger SNR produces observations closer to the state.

```
par(las = 1, mgp = c(2, 1, 0), mar = c(3.5, 3.5, 1, 0.6), mfrow = c(2, 1),
    family = "serif")
plot(Y.2, col = "blue", pch = 16, cex = 0.75, xlab = "t",
     ylab = expression(paste(x[t], " and ", y[t])), main = "", ylim = c(-5.25, 12))
lines(X, col = "red", lwd = 2)
grid()
legend("topleft", legend = "SNR = 2", bty = "n")
plot(Y.100, col = "blue", pch = 16, cex = 0.75, xlab = "t",
     ylab = expression(paste(x[t], " and ", y[t])), main = "", ylim = c(-5.25, 12))
lines(X, col = "red", lwd = 2)
grid()
legend("topleft", legend = "SNR = 100", bty = "n")
```



Carrying out the Kalman filter

The following implementation follows the six scalar Kalman recursions in the lecture. It also skips the measurement update when an observation is missing. The code is adapted from Donald B. Percival's UW Stat 519 materials.

1. Compute innovation:

$$U_t = Y_t - \hat{Y}_{t|t-1} = Y_t - \mu_t^f.$$

2. Compute `MSE` for $\hat{Y}_{t|t-1}$:

$$\Sigma_t^f + \sigma_W^2 = F_t.$$

3. Compute the new filtered value:

$$\mu_t^a = \mu_t^f + K_t U_t,$$

where $K_t = \Sigma_t^f / F_t$ is the `Kalman gain`.

4. Compute MSE for new filtered value:

$$\Sigma_t^a = \Sigma_t^f(1 - K_t).$$

5. Compute new forecast:

$$\mu_{t+1}^f = \mu_t^f + K_t U_t = \mu_t^a.$$

6. Compute MSE for new forecast:

$$\Sigma_{t+1}^f = \Sigma_t^f(1 - K_t) + \sigma_V^2 = \Sigma_t^a + \sigma_V^2$$

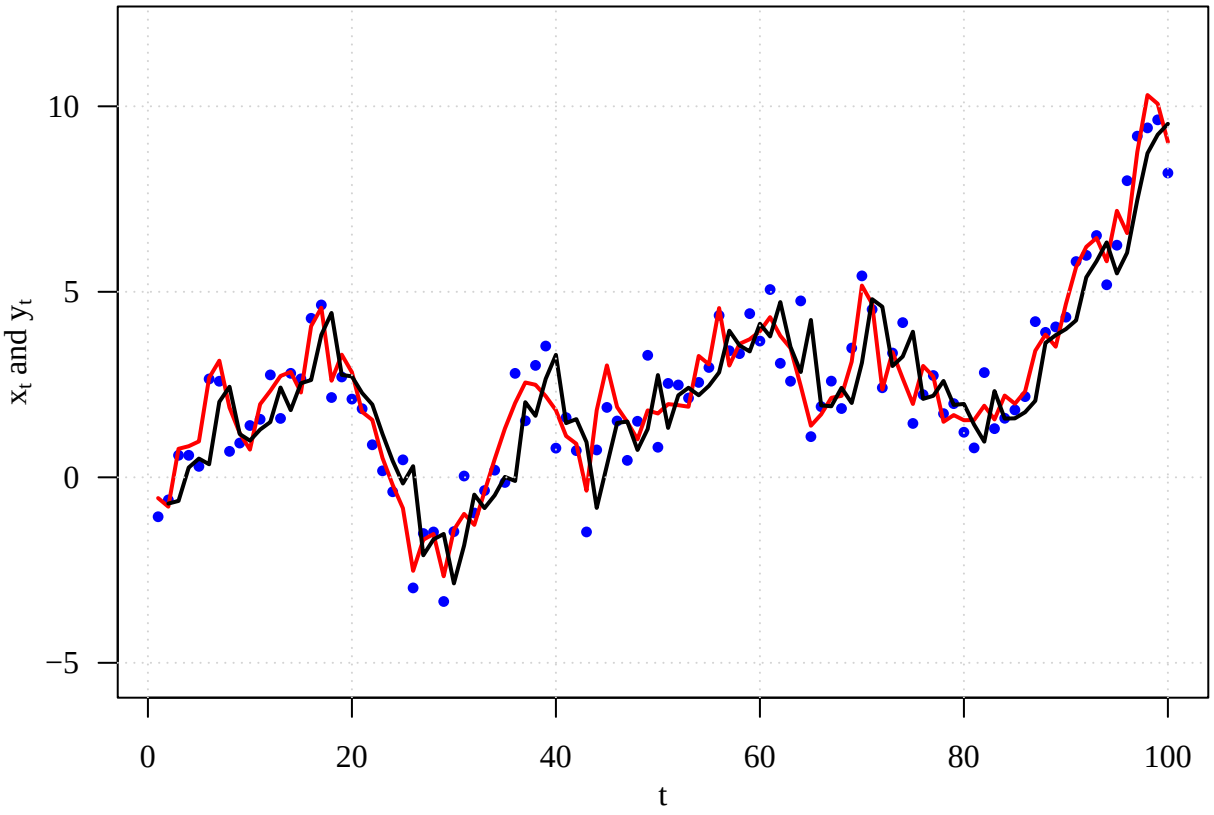
```
# One observation: forecast -> innovation -> filtering -> next forecast
KF.one.step.local.level <- function(X.t.tm1, P.t.tm1, Y.t, sig2.W, sig2.V){
  U.t <- if(is.na(Y.t)) NA else Y.t - X.t.tm1  # Missing Y: no innovation
  F.t <- P.t.tm1 + sig2.W
  K.t <- if(is.na(Y.t)) 0 else P.t.tm1 / F.t
  X.t.t <- X.t.tm1 + if(is.na(Y.t)) 0 else K.t * U.t
  P.t.t <- P.t.tm1 * (1 - K.t)
  X.tp1.t <- X.t.t
  P.tp1.t <- P.t.t + sig2.V
  structure(list(filter = X.t.t, forecast = X.tp1.t, filter.var = P.t.t,
                 forecast.var = P.tp1.t, innovation = U.t, innovation.var = F.t,
                 gain = K.t))
}

# Apply the one-step update sequentially to the full series
KF.n.steps.local.level <- function(ts, m.1 = 0, P.1 = 1, sig2.W = 1, sig2.V = 1){
  n <- length(ts)
  filter.ts <- forecast.ts <- filter.var.ts <- innovations.ts <- rep(0, n)
  forecast.var.ts <- innovations.var.ts <- gain.ts <- rep(0, n)
  X.forecast.in <- m.1; X.forecast.var.in <- P.1
  forecast.ts[1] <- X.forecast.in; forecast.var.ts[1] <- X.forecast.var.in
  Y.in <- ts[1]
  for(t in 1:n){
    temp <- KF.one.step.local.level(X.forecast.in, X.forecast.var.in, Y.in,
                                    sig2.W, sig2.V)
    filter.ts[t] <- temp$filter; filter.var.ts[t] <- temp$filter.var
    innovations.ts[t] <- temp$innovation; innovations.var.ts[t]  <- temp$innovation.var
    gain.ts[t] <- temp$gain
    if(t < n){
      forecast.ts[t + 1] <- temp$forecast
      forecast.var.ts[t + 1] <- temp$forecast.var
      X.forecast.in <- temp$forecast
      X.forecast.var.in <- temp$forecast.var
      Y.in <- ts[t + 1]
    }
  }
  structure(list(filter.ts = filter.ts, forecast.ts = forecast.ts,
                 filter.var.ts = filter.var.ts, forecast.var.ts = forecast.var.ts,
                 innovations.ts = innovations.ts,
                 innovations.var.ts = innovations.var.ts,
```

```
                gain.ts = gain.ts))
}
```

Kalman filter: forecasting

The black curve gives one-step-ahead forecasts based only on observations available before each time point.

```
Y.2.KF <- KF.n.steps.local.level(Y.2, sig2.W = 0.5)

par(las = 1, mar = c(3.5, 3.5, 1, 0.6), family = "serif", mgp = c(2, 1, 0))
plot(Y.2, col = "blue", pch = 16, cex = 0.75, xlab = "t",
     ylab = expression(paste(x[t], " and ", y[t])), main = "", ylim = c(-5.25, 12))
lines(X, col = "red", lwd = 2)
lines(2:100, Y.2.KF$forecast.ts[-1], lwd = 2)
grid()
```

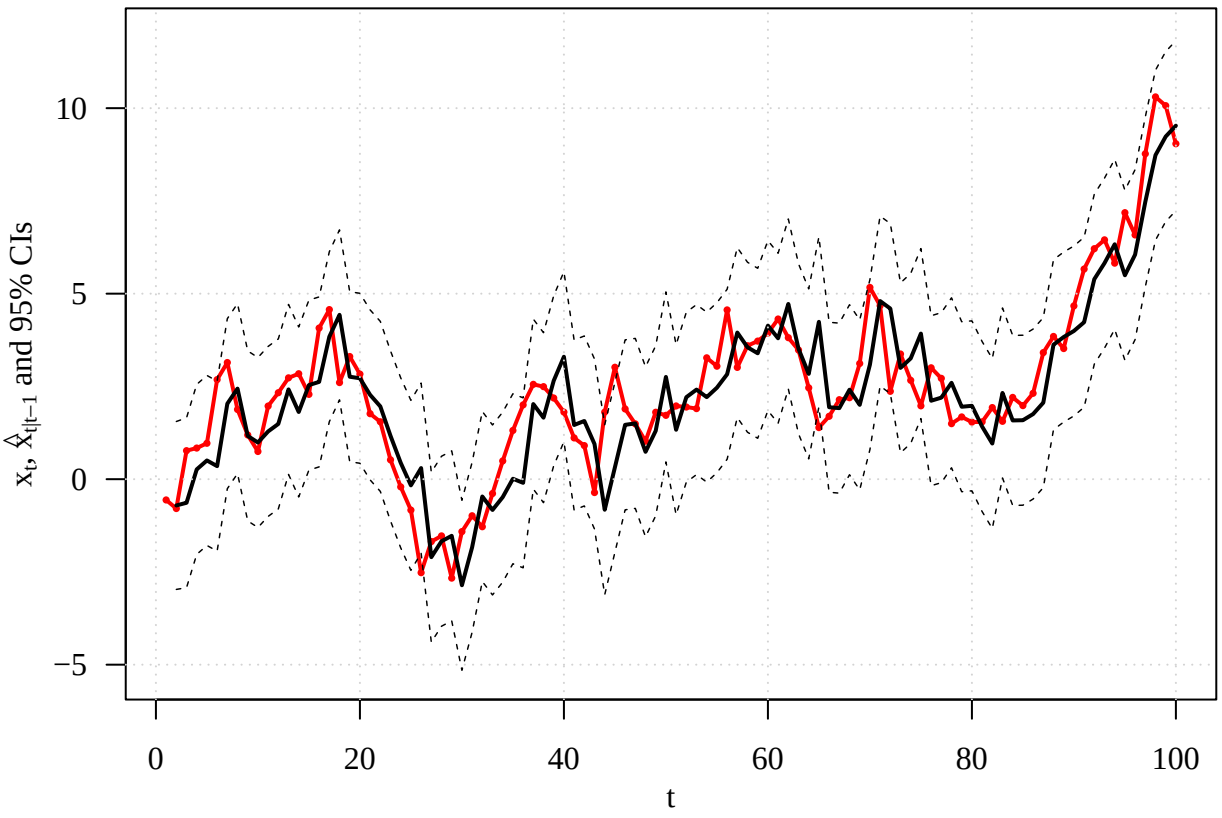


Forecast uncertainty intervals

The dashed curves are pointwise 95% intervals for the latent state, based on the forecast MSE.

```
par(las = 1, mgp = c(2, 1, 0), mar = c(3.5, 3.5, 1, 0.6), family = "serif")
plot(1:100, X, col = "red", pch = 16, cex = 0.5, xlab = "t",
     ylab = expression(paste(x[t], ", ", hat(x)[paste(t, "|", t-1)]," and 95% CIs")),
     main = "", ylim = c(-5.25, 12))
lines(X, col = "red", lwd = 2)
```

```r
lines(2:100, Y.2.KF$forecast.ts[-1], lwd = 2)
grid()
ME <- qnorm(0.975) * sqrt(Y.2.KF$forecast.var.ts[-1])
lines(2:100, Y.2.KF$forecast.ts[-1] - ME, lwd = .75, lty = 2)
lines(2:100, Y.2.KF$forecast.ts[-1] + ME, lwd = .75, lty = 2)
```

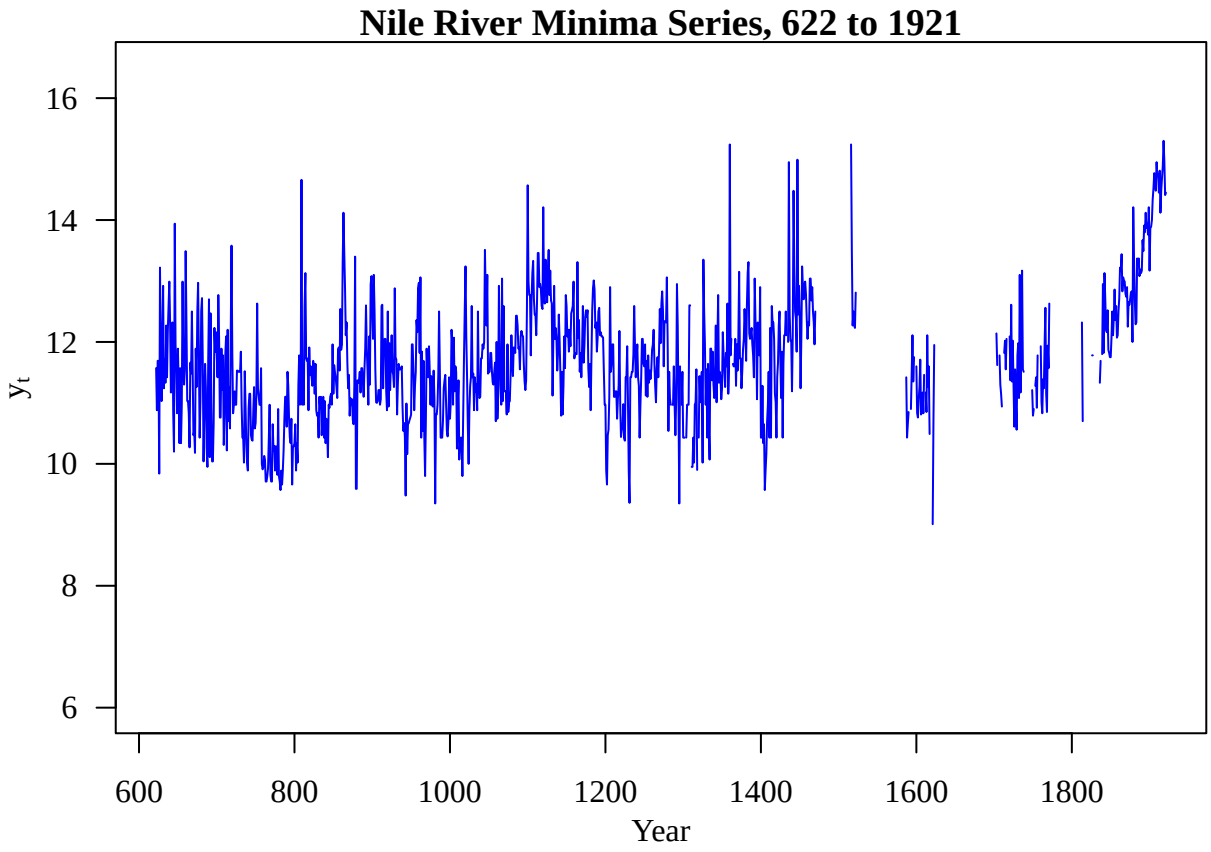


Nile River annual minima: missing observations

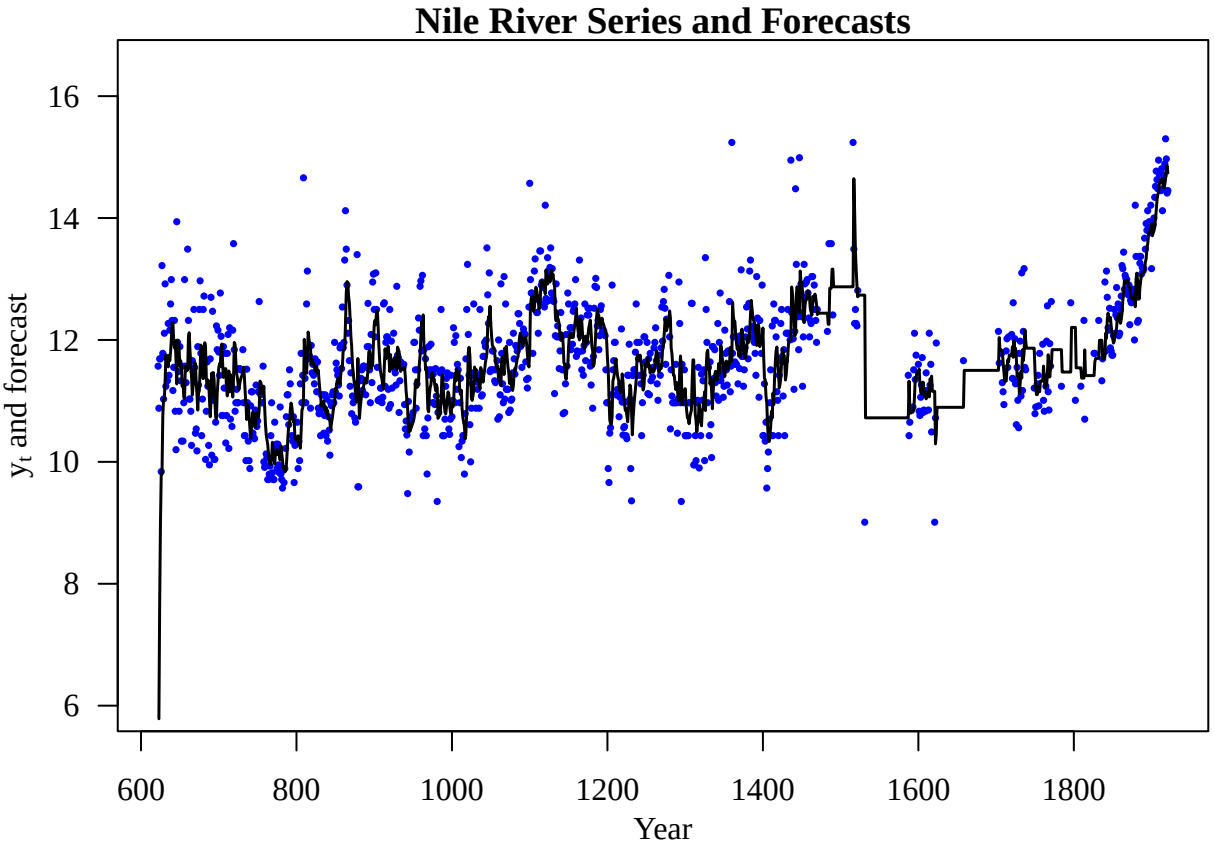
We now apply the same filter to an observed series with gaps. During missing periods, the state forecast continues, but its uncertainty increases without new measurements.

The analysis below is adapted from Dr. `Donald B. Percival`'s UW Stat 519 R codes.

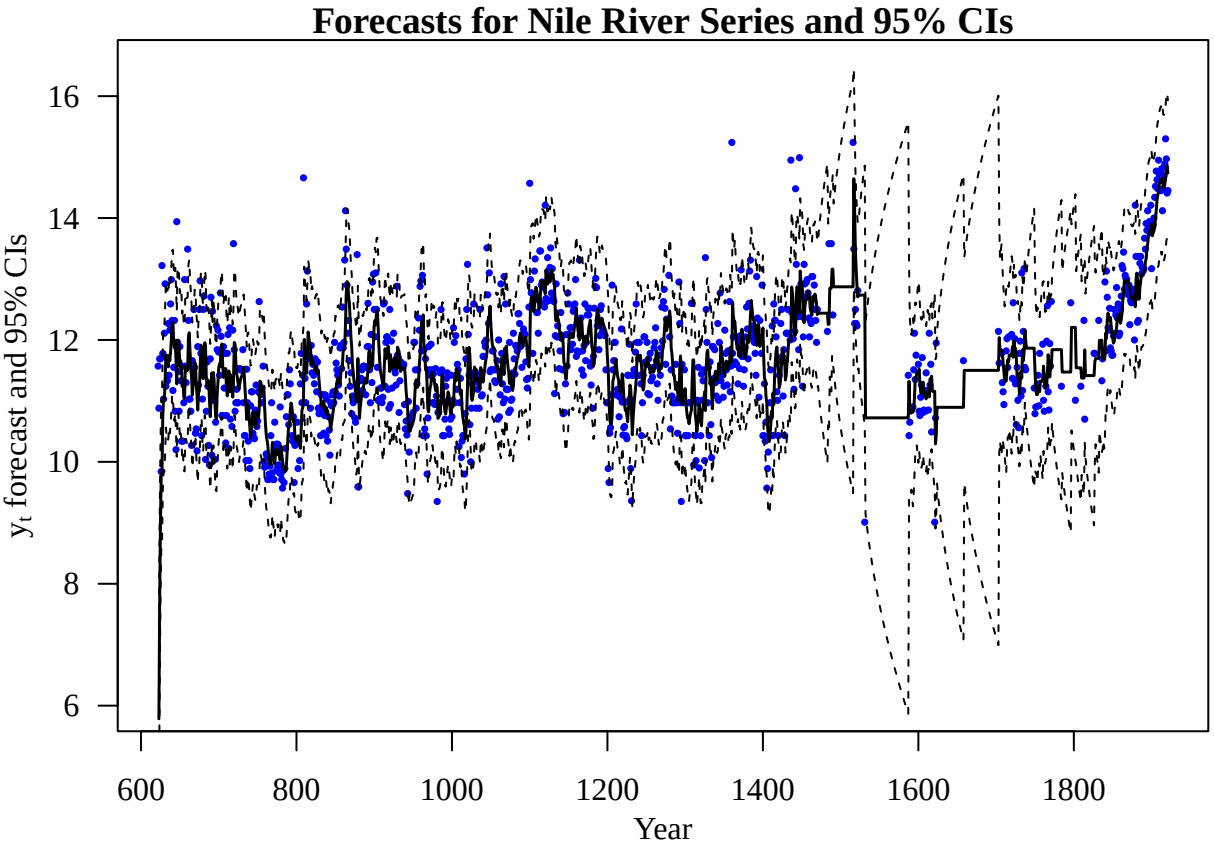
```r
nile <- scan("http://faculty.washington.edu/dbp/s519/Data/Nile-622-1921.txt")
nile.years <- 622:1921

par(las = 1, mgp = c(2, 1, 0), mar = c(3.5, 3.5, 1, 0.6), family = "serif")
plot(nile.years, nile, ylim = c(6, 16.5), type = "l", col = "blue",
     xlab = "Year", ylab = expression(y[t]),
     main = "Nile River Minima Series, 622 to 1921")
```




```
# Forecast across missing observations
nile.KF <- KF.n.steps.local.level(nile, P.1 = 1, sig2.W = 1, sig2.V = 0.1)
par(las = 1, mgp = c(2, 1, 0), mar = c(3.5, 3.5, 1, 0.6), family = "serif")
plot(nile.years, nile, pch = 16, cex = 0.5, col = "blue", ylim = c(6, 16.5),
     xlab = "Year", ylab = expression(paste(y[t]," and forecast")),
     main = "Nile River Series and Forecasts")
lines(nile.years[-1], nile.KF$forecast.ts[-1], lwd = 1.5)
```




```
# Pointwise forecast intervals for the latent state
par(las = 1, mgp = c(2, 1, 0), mar = c(3.5, 3.5, 1, 0.6), family = "serif")
plot(nile.years, nile, pch = 16, cex = 0.5, col = "blue", ylim = c(6, 16.5),
     xlab = "Year", ylab = expression(paste(y[t]," forecast and 95% CIs")),
     main = "Forecasts for Nile River Series and 95% CIs")
ME <- qnorm(0.975) * sqrt(nile.KF$forecast.var.ts[-1])
lines(nile.years[-1], nile.KF$forecast.ts[-1], lwd = 1.5)
lines(nile.years[-1], nile.KF$forecast.ts[-1] - ME, lty = 2)
lines(nile.years[-1], nile.KF$forecast.ts[-1] + ME, lty = 2)
```



Kalman smoothing

The smoother uses the entire observed series, including later observations, to refine past state estimates. The backward recursion starts from the final time point.

The function below is adapted from Dr. `Donald B. Percival`'s UW Stat 519 R codes.

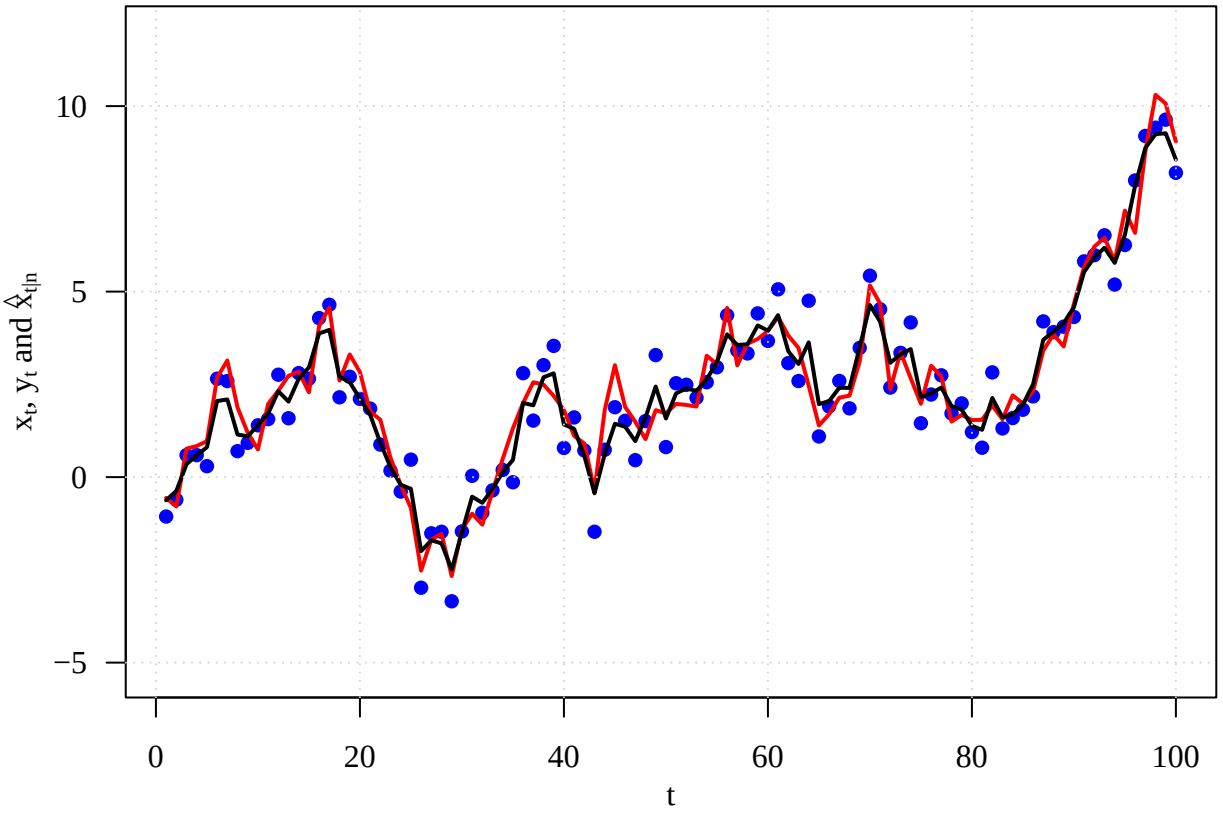
```
# Backward disturbance-smoothing recursion (scalar local-level model)
KS.local.level <- function(KF.results){
  n <- length(KF.results$filter.ts)
  L.t.ts <- 1 - KF.results$gain.ts
  r.ts <- N.t.ts <- rep(0, n + 1)
  innov <- KF.results$innovations.ts
  F <- KF.results$innovations.var.ts
  observed <- !is.na(innov)

  # Missing observations contribute no innovation information
  for(t in n:1){
    r.ts[t] <- (if(observed[t]) innov[t] / F[t] else 0) +
      L.t.ts[t] * r.ts[t + 1]
    N.t.ts[t] <- (if(observed[t]) 1 / F[t] else 0) +
      L.t.ts[t]^2 * N.t.ts[t + 1]
  }

  # Smoothed means and MSEs from the forward forecasts
  smooth.ts <- KF.results$forecast.ts +
```

```
    KF.results$forecast.var.ts * r.ts[1:n]
  smooth.var.ts <- KF.results$forecast.var.ts -
    KF.results$forecast.var.ts^2 * N.t.ts[1:n]

  list(L.t.ts = L.t.ts, r.ts = r.ts, smooth.ts = smooth.ts,
       N.t.ts = N.t.ts, smooth.var.ts = smooth.var.ts)
}
```

Kalman smoothing: local-level model

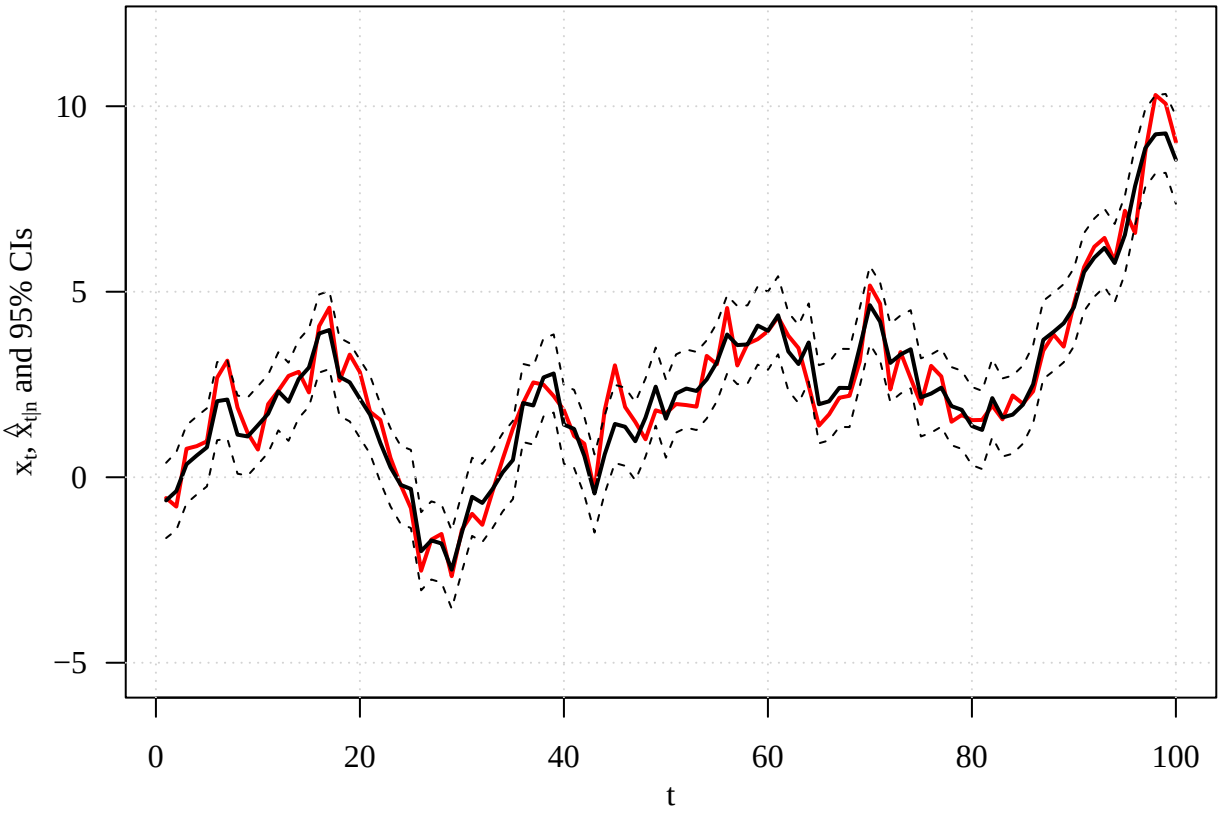
Compare the smoothed trajectory with the true state. Smoothing generally reduces conditional uncertainty relative to filtering.

```
Y.2.KS <- KS.local.level(Y.2.KF)

par(las = 1, mgp = c(2, 1, 0), mar = c(3.5, 3.5, 1, 0.6), family = "serif")
plot(Y.2, col = "blue", pch = 16, xlab = "t",
     ylab = expression(paste(x[t],", ",y[t]," and ", hat(x)[paste(t, "|", n)])), main = "",
     ylim = c(-5.25, 12))
lines(X, col = "red", lwd = 2)
lines(Y.2.KS$smooth.ts, lwd = 2)
grid()
```




```
par(las = 1, mgp = c(2, 1, 0), mar = c(3.5, 3.5, 1, 0.6))
plot(X, col = "red", typ = "l", lwd = 2, xlab = "t",
     ylab = expression(paste(x[t],", ",hat(x)[paste(t, "|", n)]," and 95% CIs")),
```

```
     main = "", ylim = c(-5.25, 12))
lines(Y.2.KS$smooth.ts, lwd = 2)
lines(Y.2.KS$smooth.ts - 1.96 * sqrt(Y.2.KS$smooth.var.ts), lty = 2)
lines(Y.2.KS$smooth.ts + 1.96 * sqrt(Y.2.KS$smooth.var.ts), lty = 2)
grid()
```



Parameter Estimation

This example follows Shumway and Stoffer (2017), Example 6.6. We estimate the state persistence and both noise scales using the innovations likelihood.

Generate Data

```
library(astsa)
set.seed(123)
# Simulate a latent AR(1) state process
num = 100
N = num + 1
x <- sarima.sim(n = N, ar = .8)
# Observations: latent state + measurement noise
y <- ts(x[-1] + rnorm(num, 0, 1))
```

The corresponding state-space model is

$$x_t = 0.8x_{t-1} + w_t, \quad w_t \sim N(0,1),$$
$$y_t = x_t + v_t, \qquad\quad v_t \sim N(0,1).$$

Here, $x[-1]$ removes the first simulated state, leaving 100 latent states aligned with the 100 observations. The state process has temporal dependence, while the observation noise is independent.

This provides a simple example for introducing Kalman filtering and state-space parameter estimation.

Initial Estimates

Use sample autocovariances to obtain initial parameter estimates for numerical maximum likelihood estimation. For the stationary AR(1) state-space model, the theoretical autocovariances satisfy

$$\gamma_Y(0) = \frac{\sigma_W^2}{1-\phi^2} + \sigma_V^2, \qquad \gamma_Y(k) = \frac{\sigma_W^2}{1-\phi^2}\phi^k, \quad k \geq 1.$$

Matching the theoretical and sample autocovariances gives

$$\phi^{(0)} = \frac{\widehat{\gamma}_Y(2)}{\widehat{\gamma}_Y(1)},$$

$$\sigma_W^{2(0)} = \frac{1-(\phi^{(0)})^2}{\phi^{(0)}}\widehat{\gamma}_Y(1),$$

$$\sigma_V^{2(0)} = \widehat{\gamma}_Y(0) - \frac{\sigma_W^{2(0)}}{1-(\phi^{(0)})^2}.$$

These moment-based estimates provide convenient starting values, but sampling variability may produce estimates outside the admissible parameter space, requiring adjustment before optimization.

```
# Sample covariances at lags 0, 1, and 2
u = ts.intersect(y, lag(y, -1), lag(y, -2))
varu = var(u)
coru = cor(u)

# Moment-based estimates of state-space parameters
phi = coru[1, 3] / coru[1, 2]
q = (1 - phi^2) * varu[1, 2] / phi
r = varu[1, 1] - q / (1 - phi^2)

# Initial values: AR coefficient and noise standard deviations
(init.par = c(phi, sqrt(q), sqrt(r)))
```

```
## [1] 0.7614651 1.0020091 0.8744762
```

Function to Evaluate the Likelihood

For each parameter candidate, the Kalman filter supplies the Gaussian innovations likelihood.

```
# Negative log-likelihood for the AR(1) state-space model
Linn <- function(para){
  phi <- para[1]; sigw <- para[2]; sigv <- para[3]
  # Stationary variance of the initial latent state
  Sigma0 <- (sigw^2) / (1 - phi^2); Sigma0[Sigma0 < 0] = 0
  # Kalman filter evaluates the likelihood
  kf = Kfilter(y, 1, mu0 = 0, Sigma0, phi, sigw, sigv)
  return(kf$like)
}
```

The function evaluates the negative Gaussian log-likelihood using the Kalman filter, with parameters $(\phi, \sigma_W^2, \sigma_V^2)$. The initial state variance assumes stationarity, requiring $|\phi| < 1$.

Estimation

Optimize the negative log-likelihood and use the Hessian for approximate standard errors, provided the optimum is interior and the Hessian is well-conditioned.

```
# Maximum likelihood estimation using BFGS optimization
(est = optim(init.par, Linn, gr = NULL, method = "BFGS", hessian = TRUE,
             control = list(trace = 1, REPORT = 1)))
```

```
## initial  value 84.170842
## iter   2 value 84.102702
## iter   3 value 83.916203
## iter   4 value 83.915653
## iter   5 value 83.889723
## iter   6 value 83.885783
## iter   7 value 83.885762
## iter   7 value 83.885762
## iter   7 value 83.885762
## final  value 83.885762
## converged
```

```
## $par
## [1] 0.8213276 0.8308274 0.9691287
##
## $value
## [1] 83.88576
##
## $counts
## function gradient
##       29        7
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##            [,1]     [,2]      [,3]
## [1,] 263.738652 74.14214 -9.936399
## [2,]  74.142142 69.77014 44.355806
## [3,]  -9.936399 44.35581 85.616367
```

```
# Approximate standard errors from the inverse Hessian
SE = sqrt(diag(solve(est$hessian)))
# Report parameter estimates and standard errors
cbind(estimate = c(phi = est$par[1], sigw = est$par[2], sigv = est$par[3]), SE)
```

```
##       estimate         SE
```

```
## phi  0.8213276 0.08831157
## sigw 0.8308274 0.20920610
## sigv 0.9691287 0.15849779
```

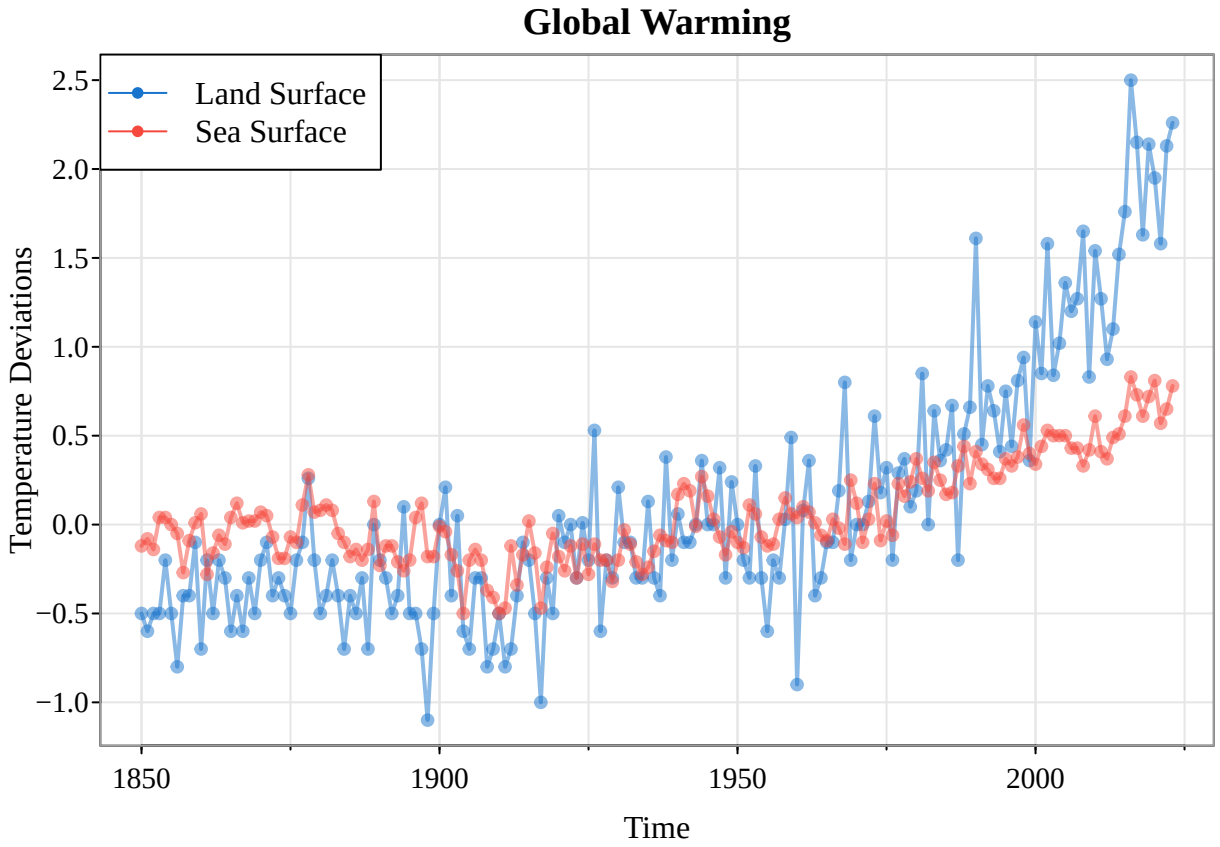
The BFGS algorithm minimizes the negative log-likelihood computed by the Kalman filter. Standard errors are approximated using the inverse observed information matrix (the Hessian of the negative log-likelihood), evaluated at the MLE.

Global Temperature Example

We illustrate state smoothing for a real temperature-anomaly series using a local-level specification.

This example is taken from Shumway and Stoffer (2017) example 6.7

```
par(las = 1, mgp = c(2, 1, 0), mar = c(3.5, 3.5, 1, 0.6), family = "serif")
tsplot(cbind(gtemp_land, gtemp_ocean), spaghetti = TRUE,
       lwd = 2, pch = 20, type = "o", col=astsa.col(c(4,2),.5),
       ylab = "Temperature Deviations", main = "Global Warming")
legend("topleft", legend = c("Land Surface", "Sea Surface"), lty = 1,
       pch = 20, col = c(4, 2), bg = "white")
```

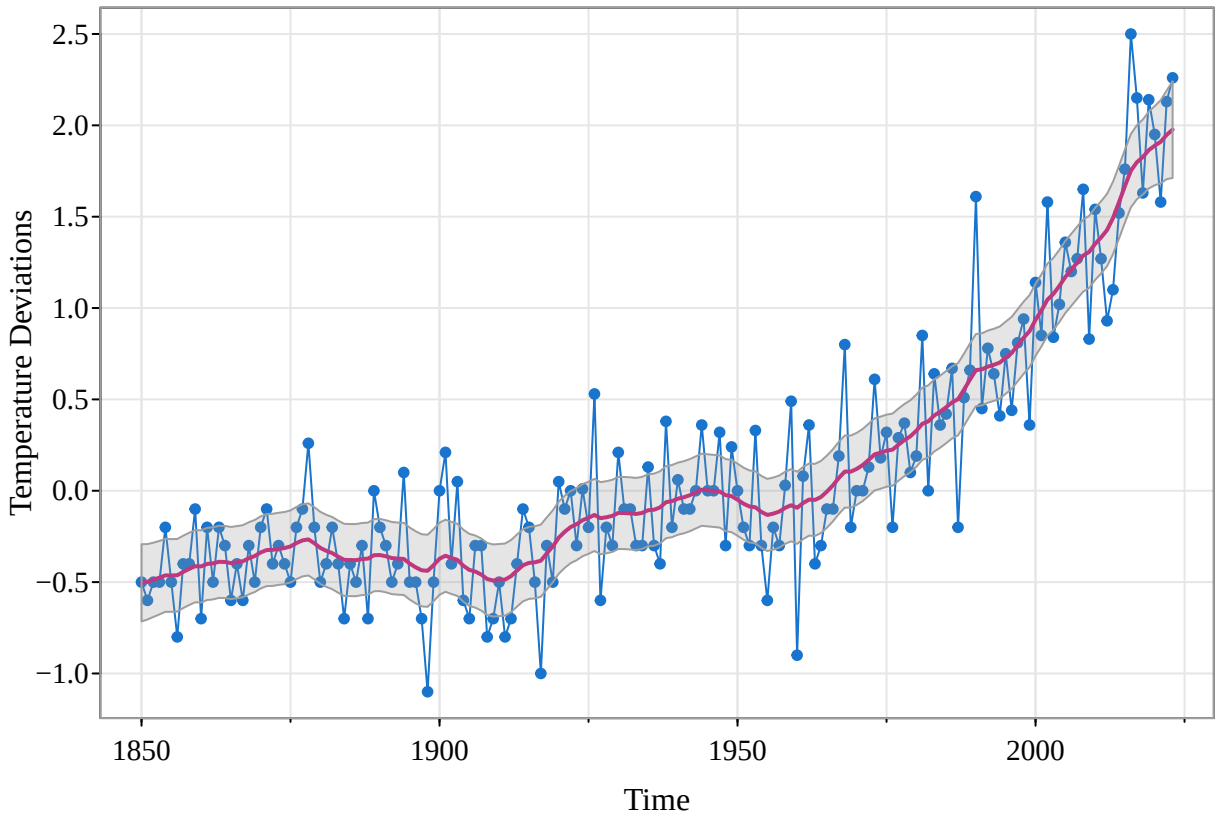


We illustrate state smoothing under a local-level model. The smoothed state estimates provide an estimate of the underlying temperature trend, together with uncertainty bands.

```
# Fit a local-level state-space model
u = ssm(gtemp_land, A = 1, phi = 1, alpha = .01, sigw = .01, sigv = .1, fixphi = TRUE)
```

```
## initial  value 548.927004
## iter   2 value 426.219829
## iter   3 value 425.534188
## iter   4 value 421.056500
## iter   5 value 420.929087
## iter   6 value 420.769864
## iter   7 value 419.824779
## iter   8 value 417.481820
## iter   9 value 308.066621
## iter  10 value 227.358420
## iter  11 value 91.340031
## iter  12 value -7.648163
## iter  13 value -52.574167
## iter  14 value -56.368335
## iter  15 value -59.350322
## iter  16 value -67.455015
## iter  17 value -70.682140
## iter  18 value -70.772274
## iter  19 value -79.846982
## iter  20 value -99.576830
## iter  21 value -100.354988
## iter  22 value -101.698806
## iter  23 value -104.630208
## iter  24 value -104.863636
## iter  25 value -105.139492
## iter  26 value -105.983246
## iter  27 value -106.008918
## iter  28 value -106.010596
## iter  29 value -106.010668
## iter  29 value -106.010668
## final  value -106.010668
## converged
##         estimate          SE
## alpha 0.01429012 0.005140273
## sigw  0.06643629 0.013370144
## sigv  0.29494320 0.017369955
```

```r
# Plot observed temperature anomalies
par(las = 1, mgp = c(2, 1, 0), mar = c(3.5, 3.5, 1, 0.6), family = "serif")
tsplot(gtemp_land, col = 4, type = "o", pch = 20, ylab = "Temperature Deviations")
# Add the smoothed latent state
lines(u$Xs, col = 6, lwd = 2)
# Construct approximate 95% pointwise uncertainty bands
xx = c(time(u$Xs), rev(time(u$Xs)))
yy = c(u$Xs - 2*sqrt(u$Ps), rev(u$Xs + 2*sqrt(u$Ps)))
# Shade the uncertainty region around the smoothed state
polygon(xx, yy, border = 8, col = gray(.6, alpha = .25))
```



The smoothed state represents the estimated underlying temperature signal after accounting for measurement noise. The shaded region shows approximately 95% pointwise uncertainty intervals, based on the smoothing variances.

EM Algorithm Example

Generate Data (same as Example 6.6)

```
library(nlme)
set.seed(123); num = 100; N = num + 1
x = sarima.sim(ar = .8, n = N)
y = ts(x[-1] + rnorm(num, 0, 1))
```

Initial Estimates

Initialize the EM algorithm with covariance-based estimates before alternating between state smoothing and parameter updates.

```
# Moment-based initial estimates from sample autocovariances
u = ts.intersect(y,lag(y, -1), lag(y, -2))
varu = var(u); coru = cor(u)

phi = coru[1,3] / coru[1,2]
q = (1 - phi^2) * varu[1, 2] / phi
r = varu[1, 1] - q/(1 - phi^2)
```

```
# Initial noise standard deviations and state distribution
cr = sqrt(r); cq = sqrt(q); mu0 = 0; Sigma0 = 2.8

# EM estimation: maximum 75 iterations, tolerance 1e-5
(em = EM(y, 1, mu0, Sigma0, phi, cq, cr, 75, .00001))
```

```
## iteration    -loglikelihood
##     1          84.36778
##     2          83.97942
##     3          83.82139
##     4          83.74255
##     5          83.69475
##     6          83.66085
##     7          83.63427
##     8          83.61222
##     9          83.59335
##     10          83.57691
##     11          83.56242
##     12          83.54955
##     13          83.53808
##     14          83.52781
##     15          83.51859
##     16          83.5103
```

```
## $Phi
## [1] 0.8106963
##
## $Q
## [1] 0.7752158
##
## $R
##           [,1]
## [1,] 0.8704274
##
## $mu0
##           [,1]
## [1,] 0.7842457
##
## $Sigma0
##           [,1]
## [1,] 0.1469216
##
## $like
##  [1] 84.36778 83.97942 83.82139 83.74255 83.69475 83.66085 83.63427 83.61222
##  [9] 83.59335 83.57691 83.56242 83.54955 83.53808 83.52781 83.51859 83.51030
##
## $niter
## [1] 16
##
## $cvg
## [1] 9.921766e-05
```

The EM algorithm alternates between estimating the latent states using Kalman smoothing (E-step) and

updating the model parameters (M-step). Moment-based estimates provide starting values, and iterations continue until convergence or the maximum number of iterations is reached.

Standard Errors

```
# Extract EM estimates of the state transition and noise standard deviations
phi = em$Phi; cq = chol(em$Q); cr = chol(em$R)

# Extract estimated initial state mean and variance
mu0 = em$mu0; Sigma0 = em$Sigma0

# Collect parameter estimates for subsequent Kalman filtering
para = c(phi, cq, cr)
```

Extract the parameter estimates from the EM algorithm. The Cholesky factors of the estimated noise covariance matrices provide the standard deviations required by Kfilter. The estimated initial state distribution is also retained for subsequent filtering or smoothing.

Evaluate Likelihood at Estimates

```
# Negative log-likelihood using EM parameter estimates
Linn = function(para){
  kf = Kfilter(y, 1, mu0, Sigma0, para[1], para[2], para[3])
  return(kf$like)
}
# Numerically approximate the Hessian at the EM estimates
emhess = fdHess(para, function(para) Linn(para))
# Approximate standard errors from the inverse Hessian
SE = sqrt(diag(solve(emhess$Hessian)))
```

Display Summary of Estimation

```
estimate = c(para, em$mu0, em$Sigma0); SE = c(SE, NA, NA)
u = cbind(estimate, SE)
rownames(u) = c("phi", "sigw", "sigv", "mu0", "Sigma0")
u
```

```
##         estimate         SE
## phi    0.8106963 0.09836856
## sigw   0.8804634 0.23235380
## sigv   0.9329670 0.17421057
## mu0    0.7842457         NA
## Sigma0 0.1469216         NA
```

Bayesian Estimation for the Local-Level Model

Finally, a Gibbs sampler alternates between sampling latent states and sampling the state and observation noise variances. Posterior draws provide uncertainty intervals that also reflect parameter uncertainty.

This example is taken from Shumway and Stoffer (2017) example 6.7

Generate Data

We simulate a Gaussian local-level model,

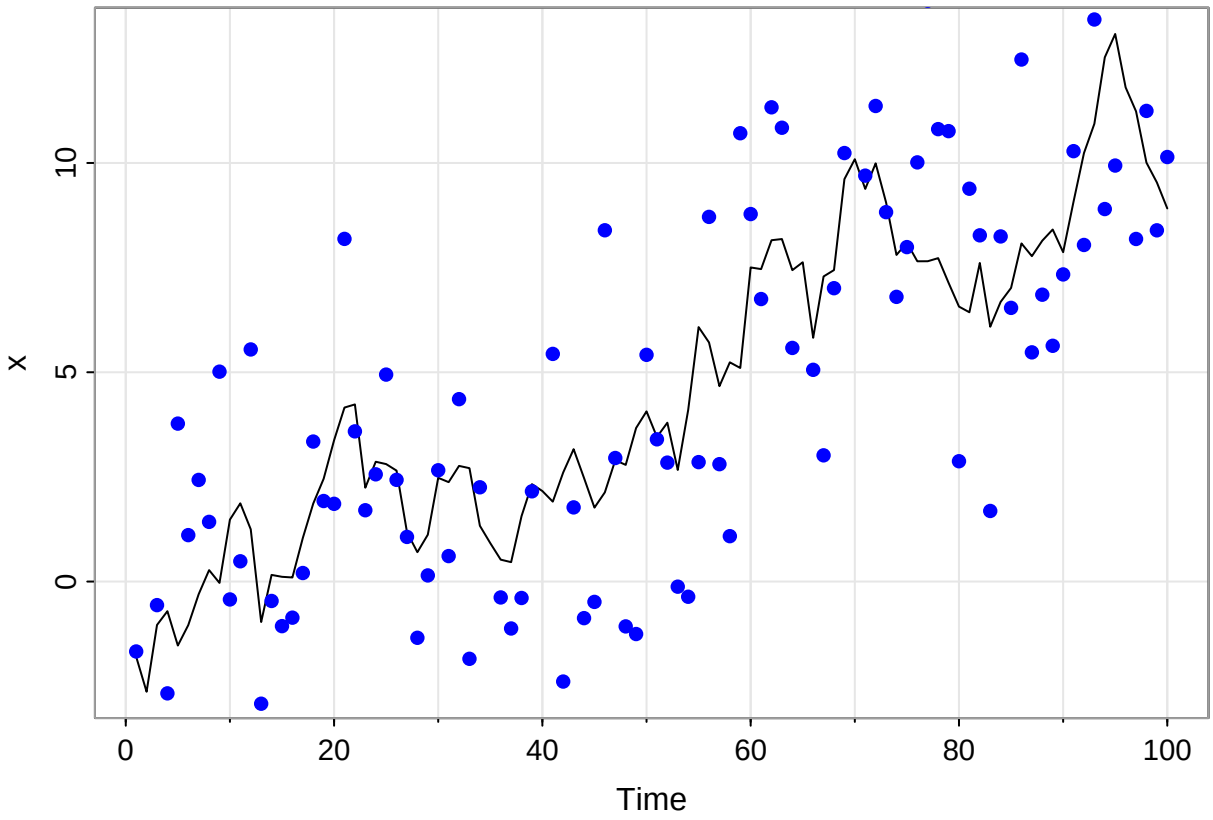
$$X_t = X_{t-1} + W_t, \quad W_t \sim N(0, Q),$$
$$Y_t = X_t + V_t, \qquad V_t \sim N(0, R),$$

with $Q = 1$, $R = 9$, and the prior $X_0 \sim N(0, 10)$.

The latent state follows a random walk, while observations contain additional measurement noise. This simulation provides a starting point for Bayesian inference on the unobserved states and model parameters.

Note that this code simulates data from a model with a prior on the initial state; it does not yet perform Bayesian posterior estimation.

```
set.seed(1)
# Draw the initial latent state from its prior distribution
sQ = 1; sR = 3; n = 100  # sQ, sR are standard deviations
mu0 = 0; Sigma0 = 10; x0 = rnorm(1, mu0, sqrt(Sigma0))  # Sigma0 is a variance

# Generate independent state and observation innovations
w = rnorm(n); v = rnorm(n)

# Initialize the first state and observation
x = c(x0 + sQ * w[1])  # initialize states
y = c(x[1] + sR * v[1])  # initialize obs
for (t in 2:n){
  x[t] = x[t - 1] + sQ * w[t]
  y[t] = x[t] + sR * v[t]
}

# Compare the latent states with noisy observations
tsplot(x, pch = 16)
points(1:100, y, pch = 16, col = "blue")
```



Set up the Gibbs Sampler

Use inverse-gamma priors for the two noise variances and discard an initial burn-in period.

```
# Set burn-in and number of retained MCMC iterations
burn = 50; n.iter = 1000
niter = burn + n.iter
draws = c()

# Inverse-Gamma priors for observation variance R and state variance Q
a = 2; b = 2; c = 2; d = 1

# (1) Initialize noise standard deviations from their priors
sR = sqrt(1 / rgamma(1, a, b)); sQ = sqrt(1 / rgamma(1, c, d))
```

Progress Bar

```
# Initialize progress bar for MCMC iterations
pb = txtProgressBar(min = 0, max = niter, initial = 0, style = 3)
```

```
##   |                                                                      |
```

```
# Gibbs sampler: alternate between states and variance parameters
for (iter in 1:niter){
```

```
  # (2) Sample latent states using forward filtering, backward sampling
  run = ffbs(y, 1, 0, 10, 1, sQ, sR)  # ffbs(y,A,mu0,Sigma0,Phi,Ups,Gam,sQ,sR,input)
  # (1) Sample noise variances from their conditional posteriors
  Xs = as.matrix(run$Xs)
  R = 1 / rgamma(1, a + n / 2, b + sum((y - Xs)^2) / 2)
  sR = sqrt(R)
  Q = 1 / rgamma(1,c + (n - 1) / 2, d + sum(diff(Xs)^2) / 2)
  sQ = sqrt(Q)
  # Store sampled parameters and latent states
  draws = rbind(draws, c(sQ, sR, Xs))
  setTxtProgressBar(pb,iter)
}
```

```
##   |                                                                      |
```

```
close(pb)
```

This Gibbs sampler alternates between two conditional sampling steps: 1. State sampling: Given the current noise parameters, forward filtering–backward sampling (FFBS) generates a draw of the entire latent state trajectory. 2. Parameter sampling: Given the sampled states, the inverse-Gamma conjugate posteriors provide new draws of the observation and state innovation variances. After discarding the burn-in iterations, the retained draws approximate the joint posterior distribution of the latent states and noise parameters.

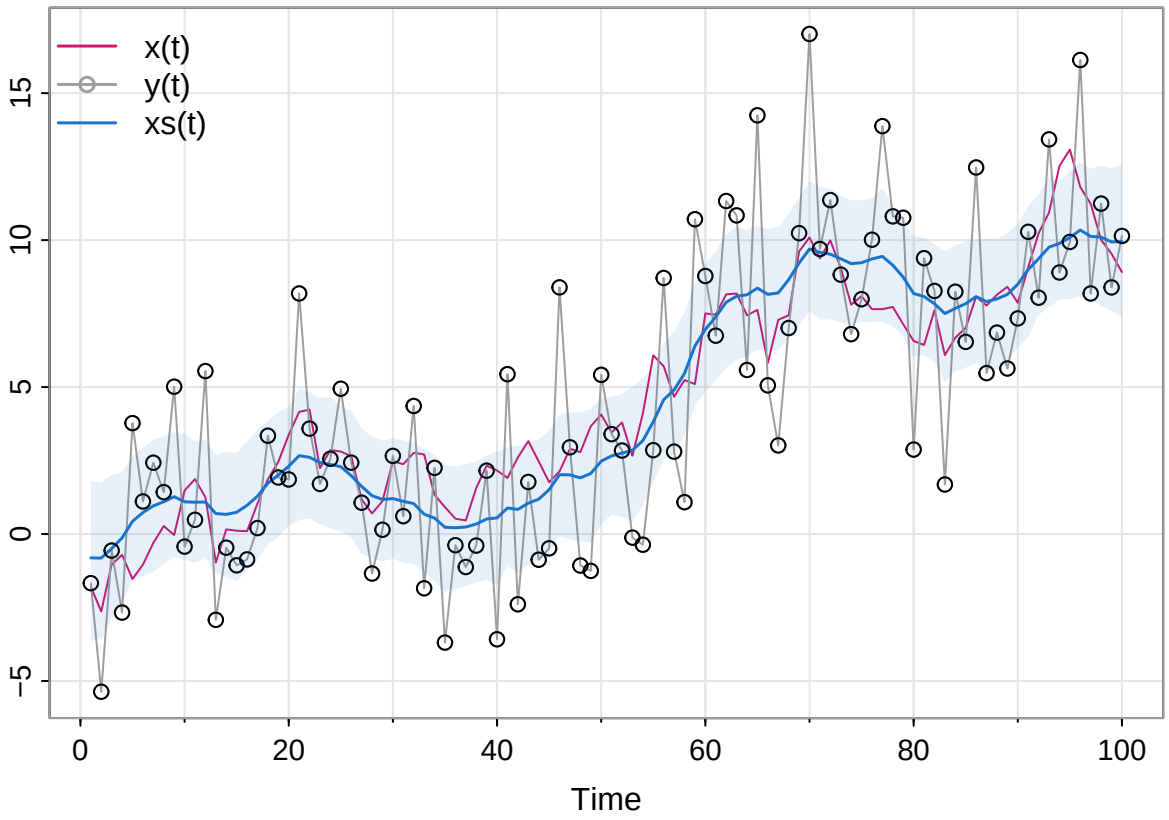
Summarize Posterior State Draws

Compute posterior means and pointwise 95% credible intervals for the latent state.

```
draws = draws[(burn + 1):(niter),]
q025 = function(x){quantile(x, 0.025)}
q975 = function(x){quantile(x, 0.975)}
xs = draws[, 3:(n + 2)]
lx = apply(xs, 2, q025)
mx = apply(xs, 2, mean)
ux = apply(xs, 2, q975)
```

Plot the Results

```
tsplot(cbind(x, y, mx), spag = TRUE, ylab='', col = c(6, 8, 4),
       lwd = c(1, 1, 1.5), type = 'o', pch = c(NA, 1, NA))
legend('topleft', legend = c("x(t)", "y(t)", "xs(t)"),
       lty = 1, col = c(6, 8, 4), lwd = 1.5, bty = "n", pch = c(NA, 1, NA))
points(y)
xx = c(1:100, 100:1)
yy = c(lx, rev(ux))
polygon(xx, yy, border = NA, col = astsa.col(4, .1))
```



References

Shumway, Robert H, and David S Stoffer. 2017. *Time Series Analysis and Its Applications*. 4th ed. Springer.