ETF Pairs Trading: Unscented Kalman Filter versus Regular Kalman Filter

David Dallaire

February 9, 2016

Introduction

One[1] approach to pairs trading in statistical arbitrage is the application of the Kalman filter. This earlier filter assumes a linear relationship via a time varying beta; alternatively, the Unscented Kalman filter is designed for non-linear cases. These two are estimated here and analyzed to see if the Unscented version catches more of a signal, implying non-linearity in the time series.

The two algorithms are estimated in R using Global Optimization by Differential Evolution (DEoptim). The regular Kalman filter uses the R package FKF. The Unscented Kalman filter is coded in both R and RcppArmadillo. Optimizing the R version is slow but Rcpp Armadillo version is much faster. (RcppArmadillo is a R link to the C++ Armidillo library and is faster).

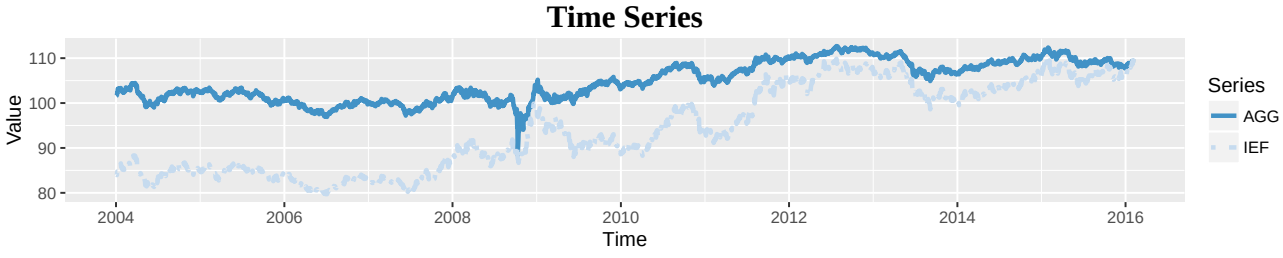
Based on the comparison of estimate to actual, beta and signal-to-noise ratio, both methods give almost identical results. Note that this is based on the levels of the ETFs and not on the returns. The next step would be to run the analysis on returns or on different pairs.

[1] `https://ca.linkedin.com/in/ddallaire`

Figure 1: Sections

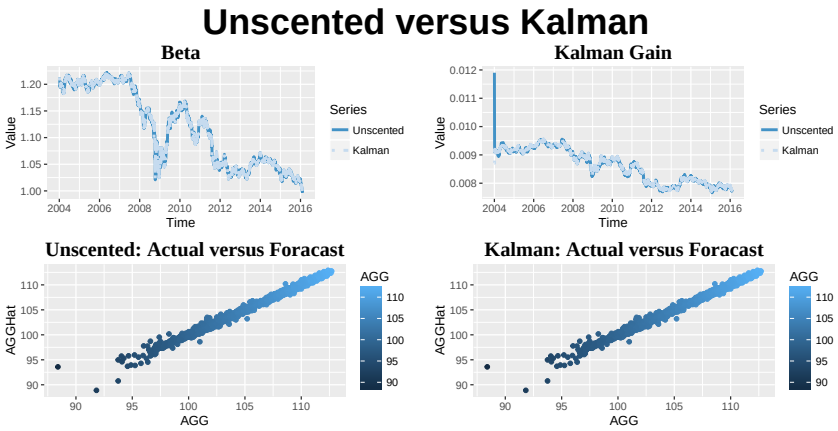
ETF Time Series

The two methods are applied here to the ETF pairs of AGG (iShares Core U.S. Aggregate Bond ETF) and IEF (iShares 7-10 Year Treasury Bond ETF).



Results

Viewing the plots created from the two estimated filters, the results look very similar: the scater plots the estimate vs actual looks identical; the betas track the same; the kalman gain looks identical except for the beginning. Note, the kalman gain–which is the proportion of the signal that the filter uses–is slowly decreasing over time. The significance here is that the filters are degrading over time.



The graph of the simulated version of signal-to-noise ratio shows a marginal benefit to the Unscented filter. This ratio shows proportion of the signal used in filtering: the higher the ratio, the better the signal. The optimization version of the signal-to-noise ratio was the similar for both with 0.000382 for Unscented and 0.00384 for the regular Kalman filter. These ratios are look to be reasonable. In general, it is expected that the Unscented version would at the least get a better signal.

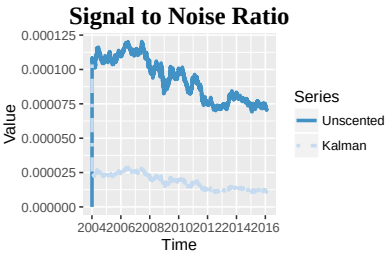


Figure 2: Signal-to-Noise Ratio

Conclusion

In the case analyzed here, the Unscented version of the Kalman filter does not provide much of an advantage over the linear version of the filter. Further research would be to try the filters using returns in addition to the level. Also, could put the filters through a back test, but the feeling here is that it could not make much difference. Finally, could try different new pairs.

Note on Kalman Filter Variables

To clarify, the symbols used in the code will follow that used by Harvey. The following are the state-space equations of the Kalman Filter[2]:

[2] The Z variable in equation 1 holds the dependent variable time series.

$$y_t = Z_t a_t + d_t + \epsilon_t, t = 1, ..., T \tag{1}$$

$$E(\epsilon) = 0 \text{ and } \sigma_\epsilon = H_t$$

$$a_t = T_t a_{t-1} + d_t + R_t n_t \tag{2}$$

$$E(n) = 0 \text{ and } \sigma_n = Q_t$$

Code for regular Kalman Filter and Optimizing

```
# use harvey notation y = Z * a + d + eps
# iid(0,H) a = Ta(t-1) + c + Rn iid(0,Q)
# bivariate

# This runs Kalman bi-variate regression with
# no moving alpha or no alpha at all

KalmanBetaFKFBivariateNoAlpha <- function(y, x,
    Q, H, beta0, sigma0) {
    n = length(y)
    y.mat <- as.matrix(y)
    x.mat <- as.matrix(x)

    # put Z,T, H and Q matrix in three dimensional
    # array that is need by FKF
    Z = array(0, dim = c(1, 1, n))
    Z[, 1, ] <- x
    Tt <- array(c(1), dim = c(1, 1, n))
    dt <- matrix(0, nrow = 1, ncol = 1)
    Ht <- array(H, dim = c(1, 1, n))
    Qt <- array(Q, dim = c(1, 1, n))

    mydt <- matrix(0, nrow = 1, ncol = 1)
    ans <- fkf(a0 = as.vector(beta0), P0 = sigma0,
        dt = mydt, ct = matrix(0), Tt = Tt, Zt = Z,
        HHt = Ht, GGt = Qt, yt = t(y.mat))

    yHat <- x * t(ans$at)[1:(dim(x)[1]), ]
```

```
    ans <- c(ans, list(yHat = yHat, y = y))
    ans
}

# optimization
KalmanOptimFKFBivariateNoAlpha <- function(y,
    x, H, Q, mu0, Sigma0, opt) {
    n = length(y)
    y.mat <- as.matrix(y)
    x.mat <- as.matrix(x)

    if (opt == 1) {
        est.kf <- RcppDE::DEoptim(LikFKF, lower = c(0,
            0), upper = c(1, 1), y.mat = y.mat,
            x.mat = x.mat, mu0 = mu0, Sigma0 = Sigma0,
            DEoptim.control(trace = FALSE, itermax = 5))
        param <- est.kf$optim$bestmem
    }
    if (opt == 2) {
        # tried using the partical swarm but would
        # need to research it some more
        os <- psoptim(rep(NA, 2), LikFKF, lower = c(1e-08,
            1e-09), upper = c(10, 10), y.mat = y.mat,
            x.mat = x.mat)
        param <- os$par
        # param <- 2
    }
    param
}

LikFKF <- function(param, y.mat, x.mat, mu0, Sigma0) {
    n = length(y.mat)
    H <- param[1]
    Q <- param[2]

    # put Z,T, H and Q matrix in three dimensional
    # array that is need by FKF
    Z = array(0, dim = c(1, 1, n))
    Z[, 1, ] <- x.mat
    myTt <- array(c(1), dim = c(1, 1, n))
    myHHt <- array(H, dim = c(1, 1, n))
    myGGt <- array(Q, dim = c(1, 1, n))
    # set dt to zero since not using it
```

```
    mydt <- matrix(0, nrow = 1, ncol = 1)

    ans <- fkf(a0 = as.vector(mu0), P0 = Sigma0,
        dt = mydt, ct = matrix(0), Tt = myTt,
        Zt = Z, HHt = myHHt, GGt = myGGt, yt = t(y.mat))
    Lik <- -1 * ans$logLik
}
```

Kalman Driver Code

```
library(FKF)
library(Quandl)
library(DEoptim)
library(RcppDE)
library(pso)

# iShares Barclays 7-10 Year Trasry Bnd Fd
# (IEF)
IEF <- Quandl("GOOG/AMEX_IEF", authtoken = "oPvBdLZBxGf4kteaV8Hi",
    type = "xts")

# iShares Barclays Aggregate Bond Fund (AGG)
AGG <- Quandl("GOOG/AMEX_AGG", authtoken = "oPvBdLZBxGf4kteaV8Hi",
    type = "xts")

IEF1 <- IEF["2004::", "Close"]
AGG1 <- AGG["2004::", "Close"]

colnames(IEF1) <- "IEF"
colnames(AGG1) <- "AGG"

n = length(AGG1)

# initial value of Beta
beta0 <- as.vector(AGG1[1, ]/IEF1[1, ])

# initial value of process/signal/beta
# volatility
sigma0 <- matrix(0)

# put dependent variable in a three
# dimensional array that is needed by FKF
Z = array(0, dim = c(1, 1, n))
Z[, 1, ] <- IEF1
```

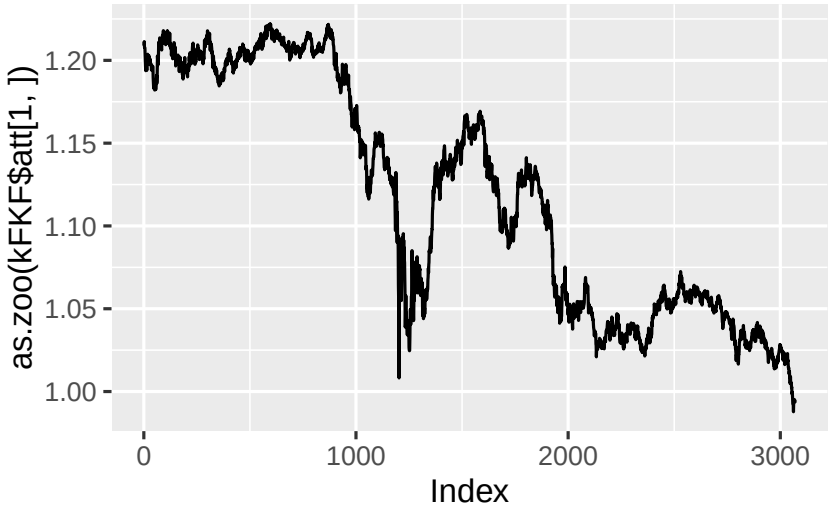
```
# initialize H and Q to zero
H <- 0  #noise
Q <- 0  #signal
HHt <- array(H, dim = c(1, 1, n))
GGt <- array(Q, dim = c(1, 1, n))

optFKF <- KalmanOptimFKFBivariateNoAlpha(AGG1,
    Z, Q, H, beta0, sigma0, 1)

Qhat <- optFKF["par1"]
Hhat <- optFKF["par2"]
signalToNoise <- Qhat/Hhat

kFKF <- KalmanBetaFKFBivariateNoAlpha(AGG1, IEF1,
    Qhat, Hhat, beta0, sigma0)

library(zoo)
library(ggplot2)
autoplot(as.zoo(kFKF$att[1, ]))
```



Unscented Kalman Filter Coded in R

```
UnscentedKalman <-function(y,x,beta0,P0,H,Q,T){
#########################################################################
#
#              Unscented Kalman Filter
#              Version 3.0.0
```

```
#              David Dallaire, February 11, 2016
#
#              INPUTS (Using Harvey notation)
#              y - time series of dependant variable
#              x - time series of independant variable. Z in Harvey
#              beta0 - initial value of unobserved state variable.
#                      This is the signal variable. It is alpha/a in
#                      Harvey.
#              P0 - initial value of the covariance.
#              H - variance of the measurement equation y.
#              Q - variance of the process equation alpha/a.
##############################################################################
#
#              OUTPUTS
#              xPredict - the predicted state of the process. alpha/a.
#              xUpdate - the filtered values of the state of the process.
#                        alpha/a.
#              like1 and like2 - two calculations of the likelihood value
##############################################################################
  N <-dim(y)[2]
  nT <-dim(y)[1]
  m <-dim(x)[2]
  #calculate the sigma points. Hard coded inputs for this version.
  sigmas <- MerweScaledSigmaPoints(m,.1,2,1,beta0,P0)
  nSigmas <- dim(sigmas$sigmas)[1]
  nColSigmas <- dim(sigmas$sigmas)[2]
  #initialize a new value
  X <- beta0
  P <- P0

  #initialize output containers
  xPredict <- matrix(0,nrow=nT,ncol=m)
  xUpdate <- matrix(0,nrow=nT,ncol=m)
  like1 <- 0
  like2 <- 0

  #loop throught all times series points
  for(n in 1:nT){
    sigmasQ <- MerweScaledSigmaPoints(m,.1,2,1,X,P)

    #PREDICT STEP
    ut <- UnscentedTransformation(sigmasQ$sigmas,sigmasQ$Wm,sigmasQ$Wc,Q)
    X <- ut$x
    P <- ut$P
```

```
    xPredict[n,] <- X

    #UPDATE STEP
    #initialize
    sigmasH <- matrix(c(0),nrow=nSigmas,ncol=1)
    Z <- x[n,]

    for (i in 1:nSigmas)
      sigmasH[i] <- t(Z) %*% sigmasQ$sigmas[i,]

    zt <- UnscentedTransformation(sigmasH,sigmas$Wm,sigmas$Wc,H)
    #initialize covariance of process and measurement.
    Pxz = matrix(0,nrow=m,ncol=N)

    for (i in 1:nSigmas){
      dx = sigmasQ$sigmas[i,] - X
      dz = sigmasH[i] - zt$x
      Pxz <- Pxz +  sigmasQ$Wc[i,] * (as.vector(dx) %o% as.vector(dz))
    }
    #zt$P is the F matrix in Harvey - F = ZPZ + H
    #K in Harvey notation K = P * Z * inv(F)
    K <- Pxz %*% ginv(zt$P) #Kalman gain
    #Forecast Error
    e <- y[n,] - zt$x
    #Update/filter state
    X <- X + K %*% e

    xUpdate[n,] <- X
    $Update/filtered Covariance
    P <- P - (K %*% zt$P %*% t(K))
    like1 <- like1 + log(det(zt$P)) + t(e)%*%ginv(zt$P)%*%e
    like2 <- like2 + dmvnorm(as.numeric(y[i,]),as.numeric(zt$x),as.matrix(zt$P),log=TRUE)
  }
  like1 <- 0.5 * like1
  list(xPredict=xPredict,xUpdate=xUpdate,like1=like1,like2=like2)
}

UnscentedTransformation <- function(sigmas,Wm,Wc,noise)
{
  x<- t(Wm) %*% sigmas
  nRowSigmas <- dim(sigmas)[1]
  nColSigmas <- dim(sigmas)[2]
  y <- sigmas - t(matrix(t(x),nrow=nColSigmas,ncol=nRowSigmas))
  PP <- t(y) %*% diag(as.numeric(Wc)) %*% (y)
```

```
  PP <- PP + noise
  list(x=t(x),PP=PP)
}
```

Sigma Points in R

```
MerweScaledSigmaPoints <- function(n, alpha, beta,
    kappa, x, P) ############################################################################ Unscented Kalman
############################################################################ Dallaire, February 11, 2016 INPUTS
############################################################################ dimensions of the process equation
############################################################################ alpha/a) or (rows of square matrix
############################################################################ - spread of points around the mean
############################################################################ distribution scaling kappa - secon
############################################################################ scaling x - state vector (alpha/a)
############################################################################ covariance of state vector note -
############################################################################ lambdas as z values that map to th
############################################################################ distribution of data points - like
############################################################################ semi-transparent dart board on top
############################################################################ scatter of points
{
    if (n != nrow(x)) {
        stop("n not equal nrow(x).")
    }
    # Weights
    lambda <- alpha^2 * (n + kappa) - n
    c = 0.5/(n + lambda)
    numberOfWeights <- 2 * n + 1
    Wc <- matrix(c, nrow = numberOfWeights, ncol = 1)
    Wc[1] <- (lambda/(n + lambda) + (1 - alpha^2 +
        beta))
    Wm <- Wc
    Wm[1] <- lambda/(n + lambda)

    # sigma points
    U <- chol((lambda + n) * P)
    sigmas <- matrix(0, nrow = 2 * n + 1, ncol = n)

    # first row is actual mean
    sigmas[1, ] <- x

    for (i in 1:n) {
        sigmas[i + 1, ] <- t(x + U[i, ])
        sigmas[n + i + 1, ] <- t(x - U[i, ])
    }
```

```
    list(sigmas = sigmas, Wc = Wc, Wm = Wm)
}
```

Unscented Kalman Filter and Sigma Point using C++ package RcppArmadillo

include

include

include <boost/math/distributions/normal.hpp> // for normal_distribution

using boost::math::normal; // typedef provides default type is double. #include using std::setw; using std::setprecision;
const double pi = boost::math::constants::pi();
// [[Rcpp::depends(RcppArmadillo)]]
using namespace std; using namespace Rcpp;
List MerweScaledSigmaPointsRCPP(Int32 n,double alpha,double beta, double kappa,const arma::vec & x, const arma::mat & P) {
double lambda, c; lambda = alpha * alpha * (n + kappa) - n; c = 0.5/ (n + lambda); int numberOfWeights; numberOfWeights = 2 * n + 1; arma::vec Wc(numberOfWeights); Wc.fill(c); Wc[0] = (lambda/(n + lambda) + (1 - alpha*alpha + beta)); arma::vec Wm = Wc; Wm[0] = lambda/(n + lambda); arma::mat U = arma::chol((lambda + n)*P); arma::mat sigmas = arma::zeros(numberOfWeights,n); sigmas.row(0) = x.t();
for(int i=0;i < n;i++){ sigmas.row(i+1) = x.t() + U.row(i); sigmas.row(n+i+1) = x.t() - U.row(i); }
return Rcpp::List::create(Rcpp::Named("sigmas") = sigmas, Rcpp::Named("Wc") = Wc, Rcpp::Named("Wm") = Wm) ;
}
List UnscentedTransform(const arma::mat & sigmas,const arma::vec & Wm, const arma::vec & Wc, const arma::mat & noise){ arma::mat x = Wm.t() * sigmas; arma::mat y = sigmas; y.each_row() -= x; arma::mat P = (y.t() * diagmat(Wc) * y) + noise; return Rcpp::List::create(Rcpp::Named("x") = x.t(), Rcpp::Named("P") = P) ; }
// [[Rcpp::export]] //List List UnscentedKalmanRCPP(const arma::vec & y,const arma::mat & x, const arma::mat & x0, const arma::mat & P0, const arma::mat & H, const arma::mat & Q, const arma::mat & T) { int nT = y.n_rows; int m = x.n_cols;
List sigmasList; List ut; List zt; sigmasList = MerweScaledSigmaPointsRCPP(m,0.1,2,1,x0,P0); arma::mat sigmas; arma::vec Wc; arma::vec Wm; sigmas = as(sigmasList[0]); Wc = as(sigmasList[1]); Wm = as(sigmasList[2]);
arma::mat xPredict(nT,m); arma::mat KalmanGain(nT,m); arma::mat

F(nT,m);
arma::mat xUpdate(nT,m); arma::mat yHat(nT,m);
arma::vec X = x0; arma::mat P = P0; arma::mat Z; arma::mat Pxz(m,1,arma::fill::zeros); arma::mat dx; arma::mat dz; arma::vec ztX; arma::mat zP; arma::mat K; arma::mat e; arma::cube pUpdate(m,m,nT,arma::fill::zeros);
arma::mat sigmas_h(sigmas.n_rows,1);
arma::mat like1(1,1,arma::fill::zeros); arma::mat like2(1,1,arma::fill::zeros);
int d = 1; //This is number of y variables like2 -= (nT * d) * log(sqrt(M_PI));
for(int n = 0; n < nT;n++){ sigmasList = MerweScaledSigmaPointsRCPP(m,0.1,2,1,X,P); sigmas = as(sigmasList[0]); Wc = as(sigmasList[1]); Wm = as(sigmasList[2]);

```
ut = UnscentedTransform(sigmas,Wm,Wc,Q);
X = as<arma::vec>(ut[0]);
P = as<arma::mat>(ut[1]);

xPredict.row(n) = X.t();
Z = x.row(n);

for (int i = 0;i<sigmas_h.n_rows;i++){
  sigmas_h.row(i) = Z * sigmas.row(i).t() ;
}

zt = UnscentedTransform(sigmas_h,Wm,Wc,H);
ztX = as<arma::vec>(zt[0]);
zP = as<arma::mat>(zt[1]);

Pxz.fill(0.0);
for (int i = 0;i<sigmas_h.n_rows;i++){
  dx = sigmas.row(i) - X.t();
  dz = sigmas_h.row(i) - ztX.t();
  Pxz = Pxz +  (dx.t() * dz) * Wc(i);
}

K = Pxz * inv(zP);
F.row(n) = zP;
KalmanGain.row(n) = K;
e = y.row(n) - ztX;
yHat.row(n) = ztX;
X = X + (K * e);
xUpdate.row(n) = X.t();
P = P - K * zP * K.t();
```

```
pUpdate.slice(n) = P;

like1 += log(det(zP)) + e.t() * inv(zP) * e;
like2 -= 0.5 * (log(det(zP)) + (e.t() * inv(zP) * e));
//cout << "  log(det(zP)) " << log(det(zP)) << endl;
//cout << " e.t() * inv(zP) * e " << e.t() * inv(zP) * e << endl;
//cout << " lile " << like1 << endl;
```

} like1 *= 0.5;
return Rcpp::List::create(Rcpp::Named("y") = y, Rcpp::Named("yHat")
= yHat, Rcpp::Named("xPredict") = xPredict, Rcpp::Named("xUpdate")
= xUpdate, Rcpp::Named("pUpdate") = pUpdate, Rcpp::Named("KalmanGain")
= KalmanGain, Rcpp::Named("F") = F, Rcpp::Named("like1") = like1,
Rcpp::Named("like2") = like2) ;
//return 10.0; }

Unscented Kalman Filter Drive

```
library(Quandl)
library(MASS)
library(mvtnorm)
library(RcppDE)

#iShares Barclays 7-10 Year Trasry Bnd Fd (IEF)
IEF <- Quandl("GOOG/AMEX_IEF", authtoken="oPvBdLZBxGf4kteaV8Hi",type="xts")

#iShares Barclays Aggregate Bond Fund (AGG)
AGG <- Quandl("GOOG/AMEX_AGG", authtoken="oPvBdLZBxGf4kteaV8Hi",type="xts")

IEF1 <- IEF["2004::","Close"]
AGG1 <- AGG["2004::","Close"]
colnames(IEF1) <- "IEF"
colnames(AGG1) <- "AGG"

beta0 <- AGG1[1,]/IEF1[1,]


Q <- matrix(c(0.1),nrow=1,ncol=1)
H <- matrix(c(1),nrow=1,ncol=1)
T <- matrix(c(1.0),nrow=1,ncol=1)
P0 <- matrix(c(10),nrow=1,ncol=1)
beta0 <- matrix(beta0,nrow=1,ncol=1)

unscent <- UnscentedKalman(AGG1,IEF1,beta0,P0,H,Q,T)
```

```
unscent$like1
#10384.72

#This version of the unscented optimization takes very long
u_like1 <- function(param,y,x,x0,p0,T){
  H <- matrix(param[1],nrow=1,ncol=1)
  Q <- matrix(param[2],nrow=1,ncol=1)
  unOpt <- UnscentedKalman(AGG1,IEF1,x0,P0,H,Q,T)
  like <- unOpt$like1
  like
}

start.time <- Sys.time()
est.kf <- RcppDE::DEoptim(u_like1,lower=c(0,0),upper=c(1,1),y=AGG1,x=IEF1,x0=beta0,p0=p0,T=T,
                          DEoptim.control(trace=FALSE,itermax=100))
end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken

est.kf$optim$bestmem

unscentRCPP <- UnscentedKalmanRCPP(AGG1,IEF1,beta0,P0,H,Q,T)
unscentRCPP$like1

#Likelihood from C++/RcppArmadillo implementation of Unscented Kalman Filter
u_likeRCPP <- function(param,y,x,x0,p0,T){
  H <- matrix(param[1],nrow=1,ncol=1)
  Q <- matrix(param[2],nrow=1,ncol=1)
  uRCPP <- UnscentedKalmanRCPP(AGG1,IEF1,x0,P0,H,Q,T)
  like <- uRCPP$like1
  like
}

Q <- matrix(c(0.0),nrow=1,ncol=1)
H <- matrix(c(0),nrow=1,ncol=1)
T <- matrix(c(1.0),nrow=1,ncol=1)
P0 <- matrix(c(10),nrow=1,ncol=1)
beta0 <- matrix(beta0,nrow=1,ncol=1)


est.uRCPP <- RcppDE::DEoptim(u_likeRCPP,lower=c(0,0),upper=c(1,1),y=AGG1,x=IEF1,x0=beta0,p0=P0,T=T)
                             DEoptim.control(trace=FALSE,itermax=200))

param <- est.uRCPP$optim$bestmem
```

```
param["par2"]/param["par1"]
param["par2"]
param["par1"]

QHat <- matrix(param["par2"],nrow=1,ncol=1)
HHat <- matrix(param["par1"],nrow=1,ncol=1)

unscentRCPP2 <- UnscentedKalmanRCPP(AGG1,IEF1,beta0,P0,HHat,QHat,T)
library(zoo)
library(ggplot2)
autoplot(as.zoo(unscentRCPP2$xUpdate[,1]))
```