함수추정 및 실습 PROJECT

김보창

```
set.seed(123)
```

실행할때마다 동일한 결과가 나오도록 set.seed를 통해 시드를 설정해준다.

```
data <- read.table("EXDATA.txt", header = T)
```

데이터를 사용하기전에,

$X_i = i, i = 1, 2...263$으로 설정되어 있으므로, 이에 알맞게 date에 따라 X를 설정해줄 필요가 있다.

```
data$X <- seq(1,263)
```

위와같이 설정해 주었다.

Q1

RSS를 구해주는 함수를 만들고, 이 함수를 이용해서 각 method에 해당하는 RSS를 구하자.

그리고, estimated curve를 출력해주는 함수도 만들자.

인자로 받는 estimate_func은 xdata, ydata, smoothing parameter를 받아서 xdata의 위치에 해당하는 estimated된 y값들을 출력해야한다.

즉, 다음과 같은 형태를 가진다.

estimate_func = function(xdata, ydata, smtparam, estim_x = NULL)

estim_x 가 NULL이면, xdata의 위치에서 추정한 값을 리턴한다. 그렇지 않으면, estim_x에 있는 데이터의 위치에 해당하는 추정값들을 리턴하게 된다.

```
get_RSS <- function(estimate_func, xdata, ydata, smtparam)
{
    n <- length(xdata)
```

```
    estimated_y <- estimate_func(xdata, ydata, smtparam)
    rss <- sum((ydata - estimated_y)**2)/n
    return(rss)
}

plot_graph <- function(estimate_func, xdata, ydata, smtparam, description = "")
{
    par(mfrow=c(1,1))
    estimated_y <- estimate_func(xdata, ydata, smtparam)
    plot(xdata, ydata , xlab = "X", ylab = "Y", type = "p", main = description, sub = sprintf("using smtparam : %
f", smtparam))
    lines(xdata, estimated_y, col = "blue")
}
```

plot graph함수는 원래 데이터를 점으로, estimate된 결과를 파란 선으로 그려준다.

이제, 각 방법론에 대해서 estimate_func를 짜면 위 두 함수를 이용하여 RSS와 그래프를 그릴 수 있다.

chapter 3에서 사용했던 방법론으로는 nadaraya-watson estimator, local polynomial regression, natural spline (using binomial filter), smoothing spline, LOESS, supersmoother, LOWESS가 있다.

각각의 방법에 대해, RSS와 그래프를 그려보고 그중 최적의 RSS를 가지는 방법론을 찾아보도록 하겠다.

이때, 각각의 방법론은 대부분 smoothing parameter에 따라 결과가 달라지게 되므로, 각 방법론 내부에서도 최적의 smoothing parameter를 찾을 필요가 있다.

따라서, Cross Validation 값을 이용해서, 이 값이 가장 작은 smoothing parameter를 사용할것이고,

이러한 과정을 편하게 진행하기 위해 다음과 같은 함수들을 미리 짜놓고 이용할 것이다.

cross validation 값의 정의는 아래와 같고,

$$CV[\hat{m}(x)] = \frac{\sum_{k=1}^{n}\left(Y_k - \hat{m}^{-k}\left(X_k\right)\right)^2}{n}$$

Least-sqaure estimator들의 경우

leave-one-out-cross-validation 값이 다음과 같이 간편하게 구해진다.

$$CV[\hat{m}(x)] = \frac{\sum_{k=1}^{n}\left(Y_k - \hat{m}\left(X_k\right)\right)^2}{n(1-H_{ii})^2}$$

loess와 같은 경우에는 위의 cross validation값이 참값이 아니게 되지만, 데이터가 많고, 각 데이터에 대해 정의대로 일일히 추정하는것은 너무 오랜 시간이 걸리므로, 위와같이 CV를 사용하도록 하겠다.

```
get_cross_validation <- function(estimate_func, x1, y1, smtparam_array)
{
#   (1)
    getcv <- function(smtparam,estimate_func, x1 ,y1)
    {
#   (2)
        nd <- length(x1)
        estim_y <- estimate_func(x1, y1, smtparam)
        res <- y1 - estim_y
#   (3)
        dhat1 <- function(estimate_func, x2, smtparam)
        {
            nd2 <- length(x2)
            diag1 <- diag(nd2)
            dhat <- rep(0, length = nd2)
#   (4)
            for(jj in 1.:nd2) {
                y2 <- diag1[, jj]
                estim_y2 <- estimate_func(x2, y2, smtparam)
                dhat[jj] <- estim_y2[jj]
            }
            return(dhat)
        }
#   (5)
        dhat <- dhat1(estimate_func, x1, smtparam)
#   (6)
        cv <- sum((res/(1. - dhat))^2.)/nd
#   (7)
        return(cv)
```

```
#   (8)
    cvlst <- lapply(as.list(smtparam_array),getcv,  estimate_func = estimate_func, x1 = x1, y1 = y1)
    cvlst <- unlist(cvlst)
    return(cvlst)
}
```

위 함수는, x,y,smoothing parameter를 가지고 추정값을 리턴해주는 estimate_func과, smoothing parameter들 목록을 받아 cross validation 값들을 리턴해준다.

```
get_best_smtparam <- function(estimate_func, x1, y1, smtparam_array)
{
    cvarray <- get_cross_validation(estimate_func, x1, y1, smtparam_array)
    smt_idx <- which(cvarray == min(cvarray, na.rm = TRUE))
    min_smt_cv <- smtparam_array[smt_idx]

    return(min_smt_cv)
}
```

또한, 위 함수는 위를 통해 생성된 cv값들중 가장 낮은 cv값을 가지는 parameter를 찾아준다.

```
plot_cv <- function(smtparam_array, cv_array, description = "")
{
    par(mfrow = c(1, 1))
    plot(smtparam_array, cv_array, type = "n",
     xlab = "smt_param", ylab = "CV", main = description)
    points(smtparam_array, cv_array, pch = 1, cex = 0.5)
    #lines(smtparam_array, cv_array, lwd = 1)
    pcvmin <- seq(along = cv_array)[cv_array == min(cv_array, na.rm = TRUE)]
    spancv <- smtparam_array[pcvmin]
    cvmin <- cv_array[pcvmin]
    points(spancv, cvmin, cex = 1, pch = 15, col = "red")
}
```

위 함수는 위를 통해 생성된 cv값들을 그래프로 출력해준다.

빨간점은 최소 cv를 나타낸다

마지막으로, 각각의 함수에 대해 그래프를 출력해주고 RSS값을 리턴해주는 통합 함수를 만들자.

```
full_test <- function(estimate_func, xdata, ydata, smtparam_array, description = "", using_CV = TRUE)
{
    if(using_CV){
        cv_array <- get_cross_validation(estimate_func, xdata, ydata, smtparam_array)

        plot_cv(smtparam_array, cv_array, description)

        smt_idx <- which(cv_array == min(cv_array, na.rm = TRUE))
        if(length(smt_idx) > 1)
        {
            smt_idx <- smt_idx[1]
        }
        best_smt <- smtparam_array[smt_idx]
    }
    else
    {
        best_smt <- smtparam_array[1]
    }
    plot_graph(estimate_func, xdata, ydata, best_smt, description = description)

    rss <- get_RSS(estimate_func, xdata, ydata, best_smt)

    return(rss)
}
```

그리고 rss값을 저장할 벡터를 만들자.

```
rss_GBP <- vector()
rss_CHF <- vector()
rss_CAD <- vector()
```

이제 모든 준비가 끝났다.

nadaraya-watson estimator는 R의 함수중 ksmooth 함수를 사용하면 구할 수 있다.

$$\hat{m}(x) = \frac{\sum_{i=1}^{n} K\left(\frac{x-x_i}{h}\right) Y_i}{\sum_{i=1}^{n} K\left(\frac{x-X_i}{h}\right)} = \sum_{i=1}^{n} W_i(x) Y_i$$

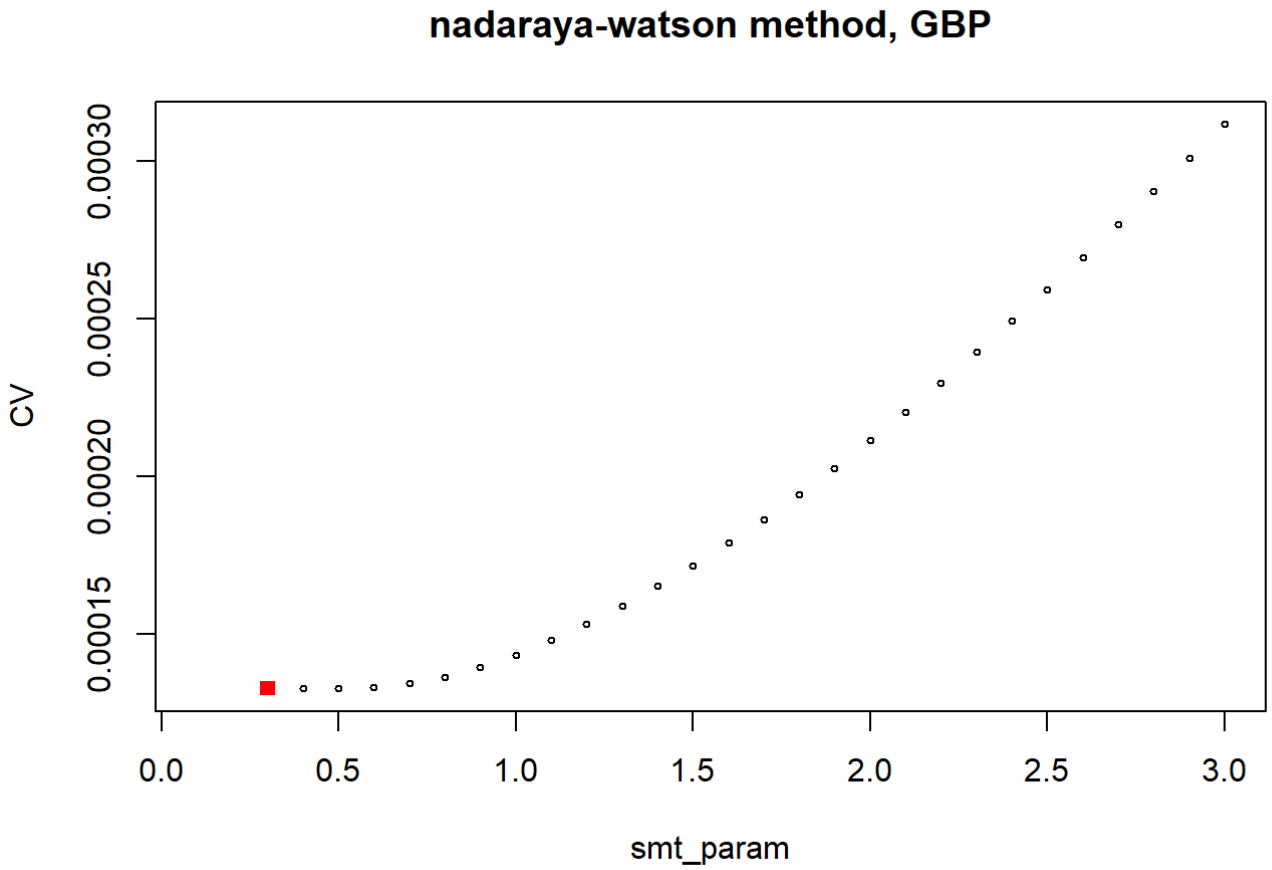
위와 같은 방법으로 추정값을 구하는데, 여기서 사용할 Kernel함수에 따라, 또한 bandwidth로 무엇을 사용하느냐에 따라 추정값이 달라지게 된다.

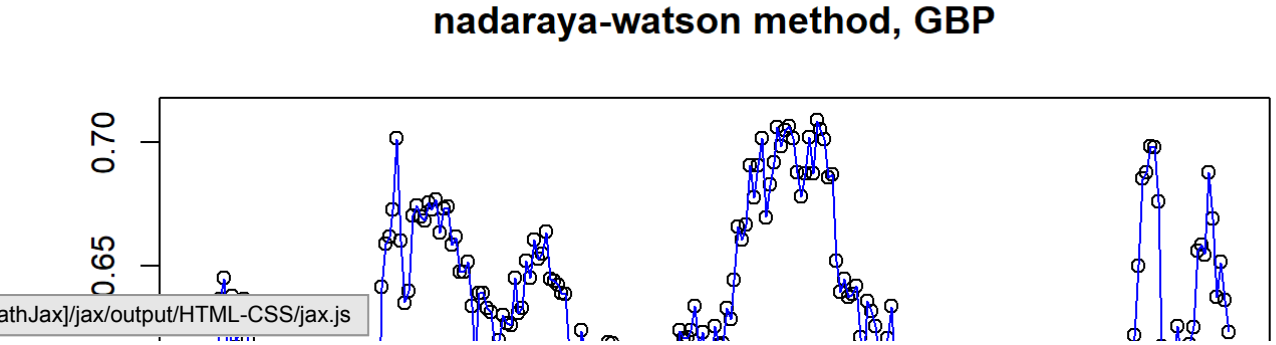
여기서는 kernel로 gaussian을, bandwidth는 여러 bandwidth를 사용해서 Cross validation값이 가장 작은 값을 구하고, Cross validation을 가장 작게 만드는 값을 사용해서 bandwidth를 구하도록 하겠다.

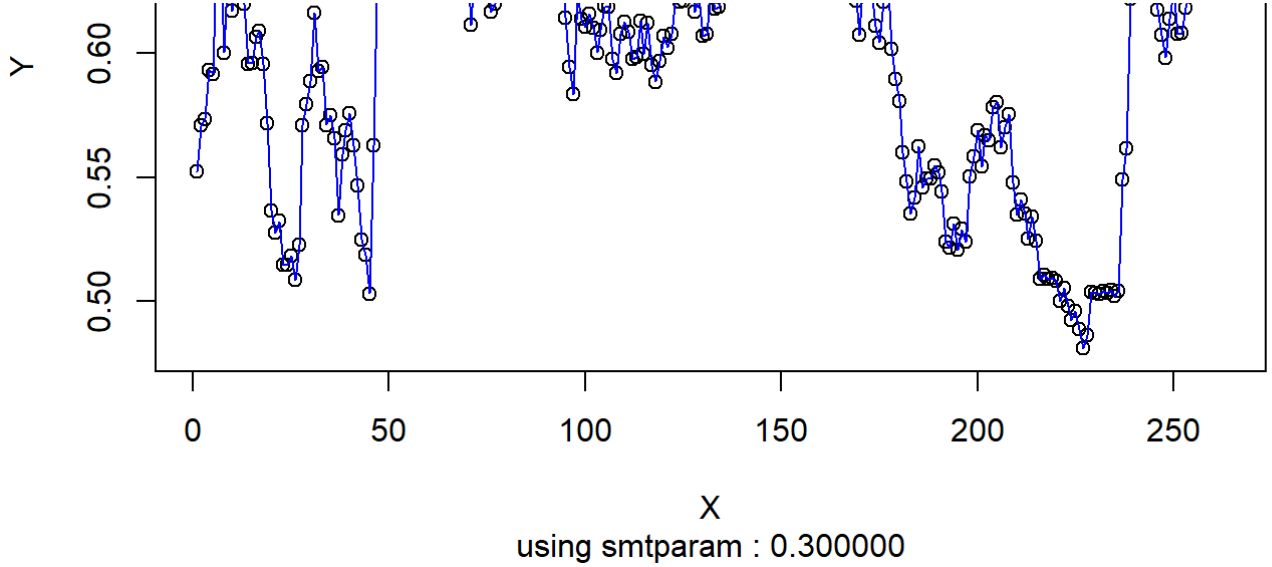
```
estimate_nadaraya <- function(xdata, ydata, bw, estim_x = NULL)
{
    bwsplus <- bw/0.3708159
    if(is.null(estim_x))
    {
        fit.ks <- ksmooth(xdata, ydata, "normal", bandwidth = bwsplus, x.points = xdata)
    }
    else
    {
        fit.ks <- ksmooth(xdata, ydata, "normal", bandwidth = bwsplus, x.points = estim_x)
    }
    return(fit.ks$y)
}
```

bandwidth를 0.1~3까지, 0.1단위로 실험해봤을때, 다음과 같이 가장 CV를 작게 만드는 RSS값을 찾을 수 있다.

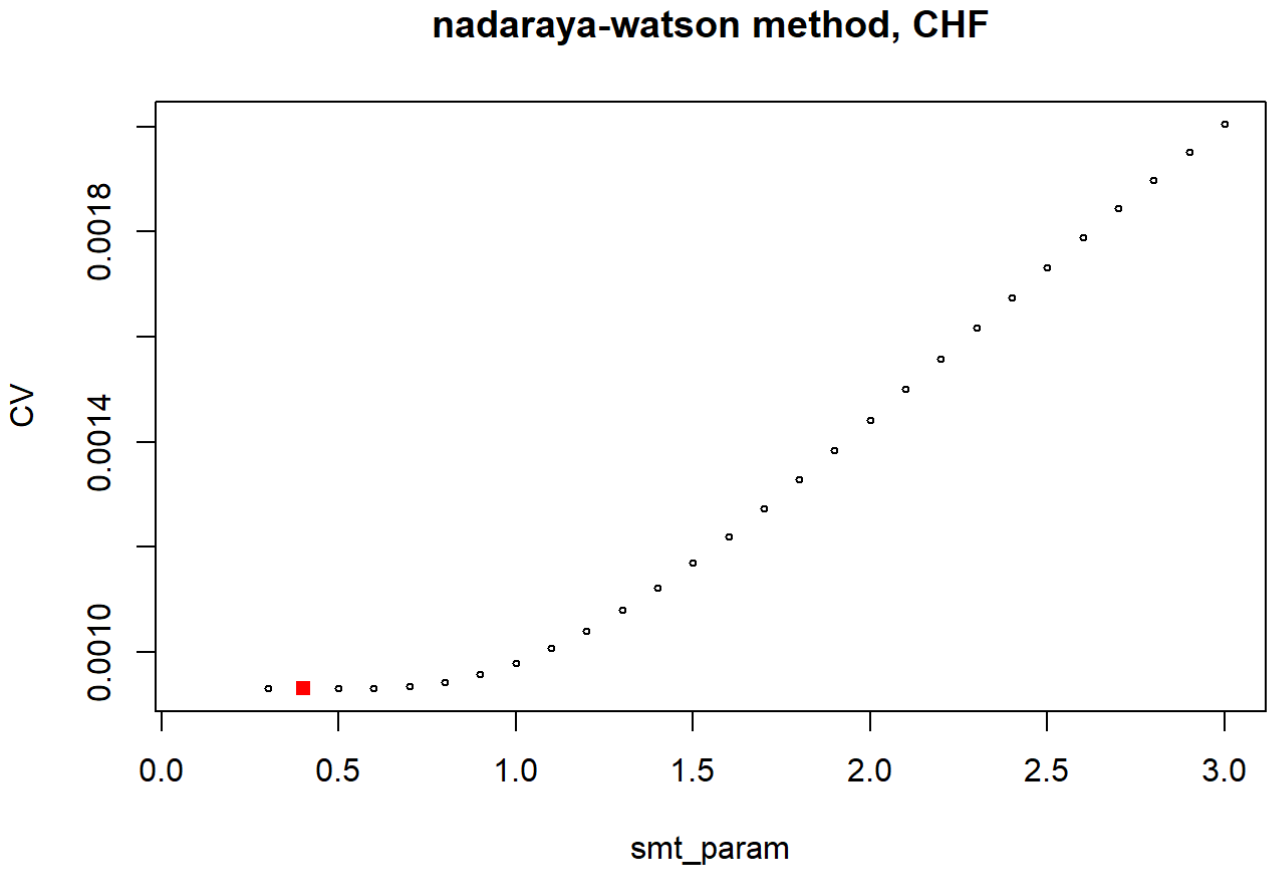
```
rss_GBP[1] <- full_test(estimate_nadaraya, data$X, data$GBP, seq(0.1,3,by = 0.1), description = "nadaraya-watson
 method, GBP")
```

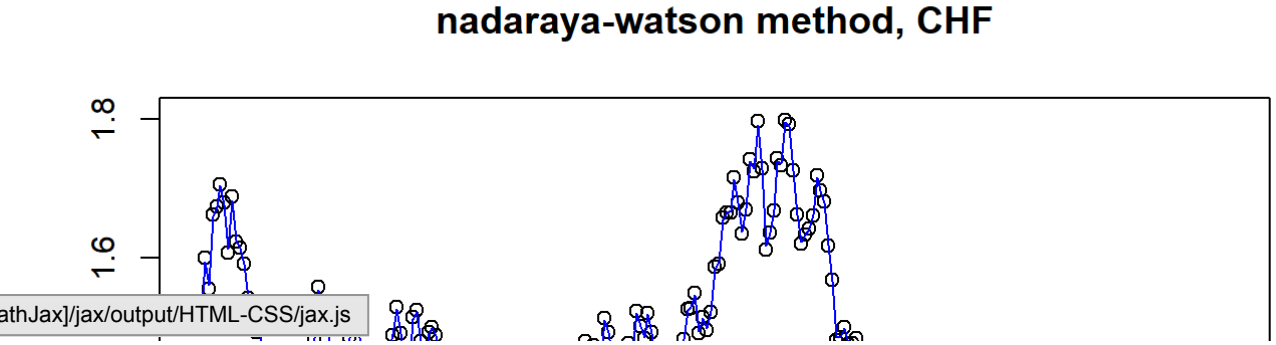


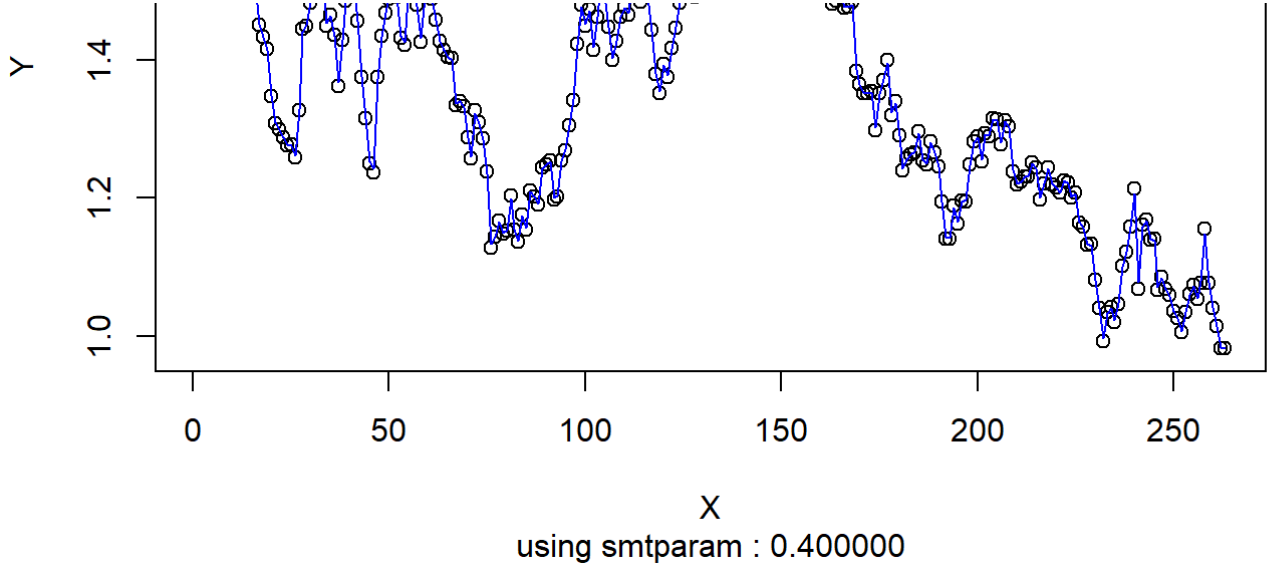




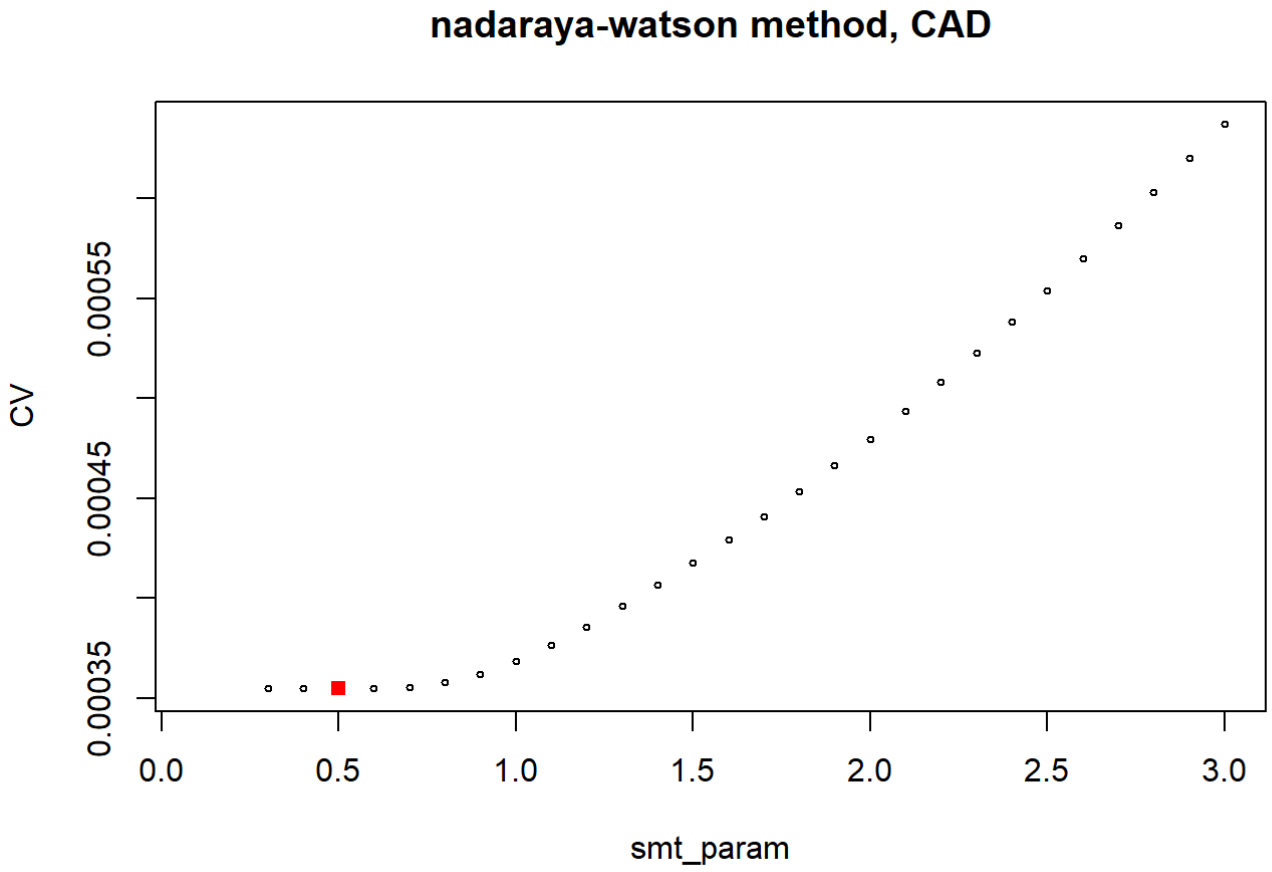

```
rss_CHF[1] <- full_test(estimate_nadaraya, data$X, data$CHF, seq(0.1,3,by = 0.1), description = "nadaraya-watson
 method, CHF")
```

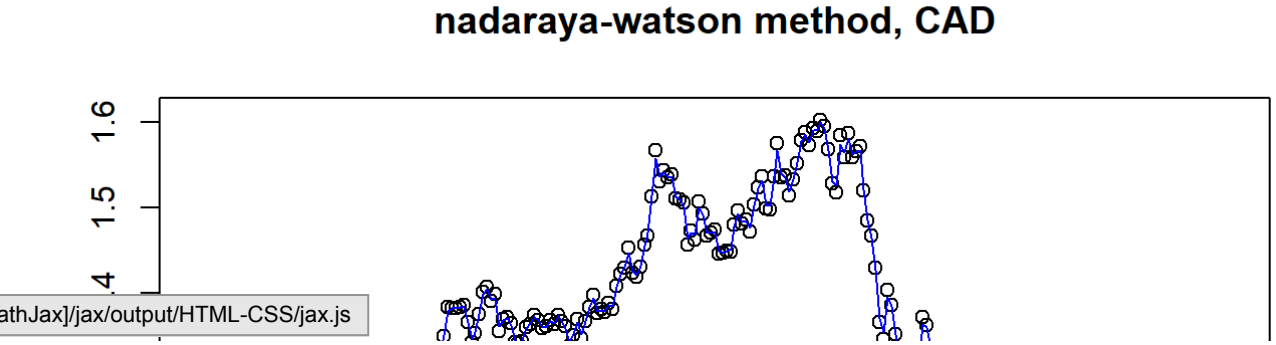


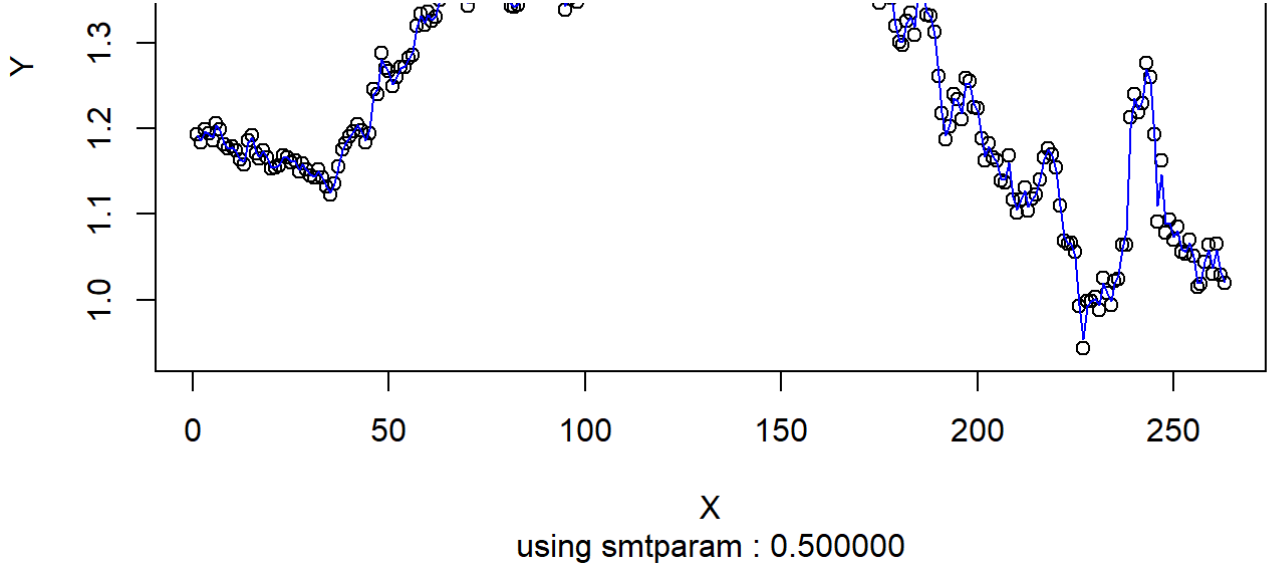





```
rss_CAD[1] <- full_test(estimate_nadaraya, data$X, data$CAD, seq(0.1,3,by = 0.1), description = "nadaraya-watson
 method, CAD")
```







결과는 위와 같다.

단순히 RSS만 최소화 하려면, smoothing parameter를 아주 작게 만들면 데이터에 overfit이 되므로 RSS를 최소화 할 수 있다.

local polynomial

local polynomial estimator는 R의 함수중 lm함수를 weighted 옵션을 주면 구현할 수 있다.

정확히는, 각 x^*에 대해

$$E_{\text{local}}\left(x^*\right) = \sum_{i=1}^{n} w\left(\frac{X_i - x}{h}\right)\left(Y_i - m\left(X_i, x^*\right)\right)^2$$

$$= \sum_{i=1}^{n} w\left(\frac{X_i - x^*}{h}\right)\left(Y_i - \left(a_0\left(x^*\right) + \sum_{j=1}^{p} a_j\left(x^*\right)\left(x - x^*\right)^j\right)\right)^2$$

을 최소화하는 $\hat{a}_j(x^*)$들을 구하고, 그중에 $\hat{a}_0(x^*)$을 $\hat{m}(x^*)$의 값으로 사용하게 된다.

이때의 kernel은 다양한 선택이 가능하지만, 여기서는 gaussian만을 사용하도록 하겠다. 즉, $w(\frac{X_i - x^*}{h}) = exp(-\frac{1}{2}(\frac{X_i - x^*}{h})^2)$

또한, polynomial의 차수도 다양하게 선택이 가능하지만, 여기서는 1차 polynomial에 대해서만 구해보겠다. (2차 이상의 polynomial regression을 하기에는 너무 많은 시간이 필요하다.)

여기서, 책에 있는 방법으로 각 점에대해 local polynomial regression을 하면

점이 260개정도이므로 각 데이터마다 대략 260번씩 regression을 해야 하므로 시간이 너무 오래걸린다.

거기에 cross validation을 구하기 위해서는 각 bandwidth에 대해 H를 구해주는 작업이 필요한데, 이러한 H를 구하는 작업도 260개의 데이터 (우리 알고리즘은 $e_1, e_2 \ldots e_n$에 대해 estimate해본 결과에 따라 H를 구한다.)에 대해 각각 추정을 해야되므로 끔찍하게 긴 시간이 걸릴 것이다.

즉, 아래와 같이 구현한 함수는 사용하지 않는다.

```
local_linear_not_use <- function(x_star, xdata, ydata, band)
{
    modi_x <- xdata - x_star
    wts <- exp((-0.5 * modi_x^2)/band^2)

    df <- data.frame(x = modi_x, y = ydata, www = wts)

    fit.lm <- lm(y ~ x, data = df, weights = www)

    est <- fit.lm$coef[1.]
    names(est) <- NULL

    return(est)
}

estimate_local_linear_not_use <- function(xdata, ydata, bw, estim_x = NULL)
{
    if(is.null(estim_x))
    {
        ey <- sapply(as.list(xdata), local_linear_not_use, xdata = xdata, ydata = ydata, band = bw)
    }
    else
    {
```

```
        ey <- sapply(as.list(estim_x), local_linear_not_use, xdata = xdata, ydata = ydata, band = bw)
    }

    return(ey)
}
```

따라서, 이러한 시간적 문제를 해결하기 위해 외부 패키지를 사용할 것이다.

locpol library의 locpol 함수를 사용하도록 하겠다.

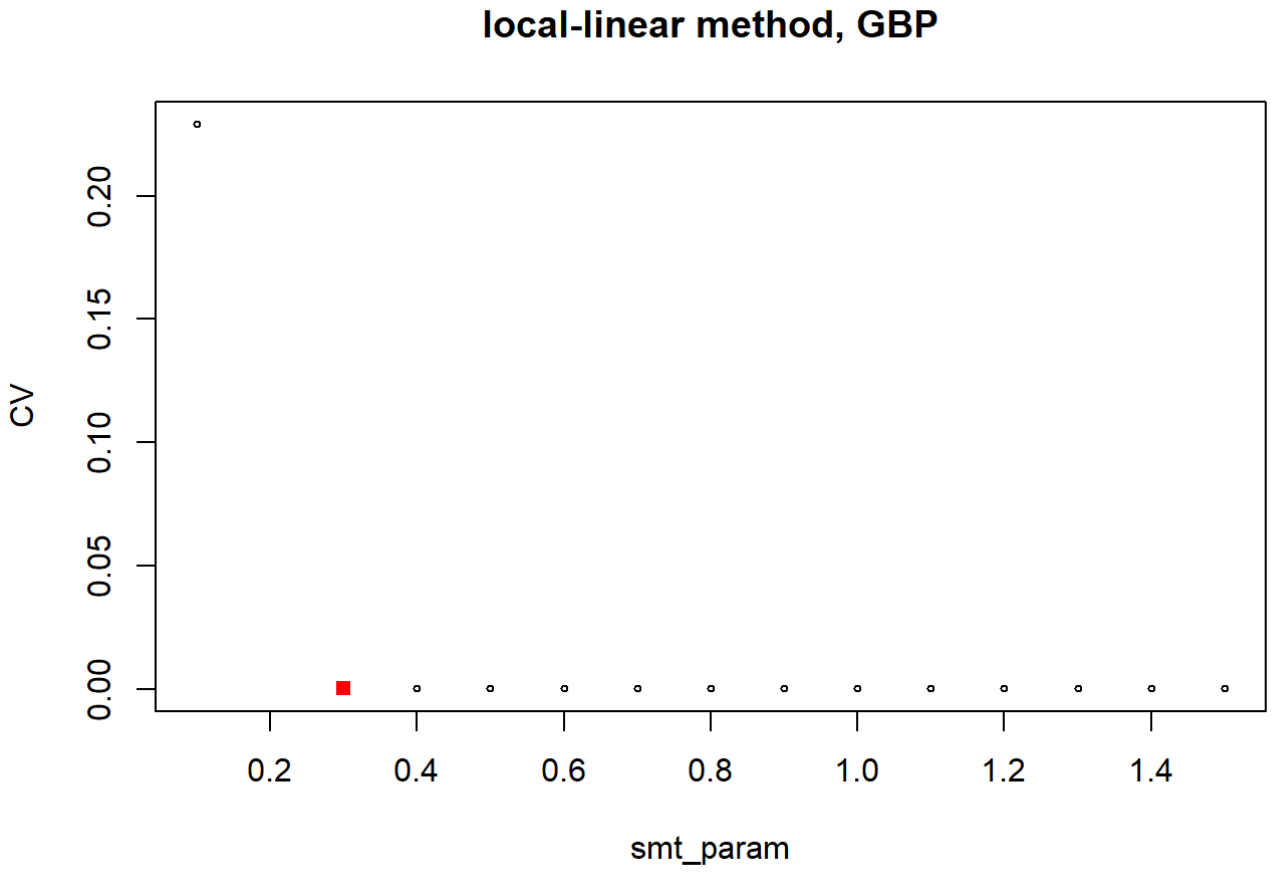
kernel로는 gaussian 함수를 사용하고, linear polynomial case에 대해 구한다.

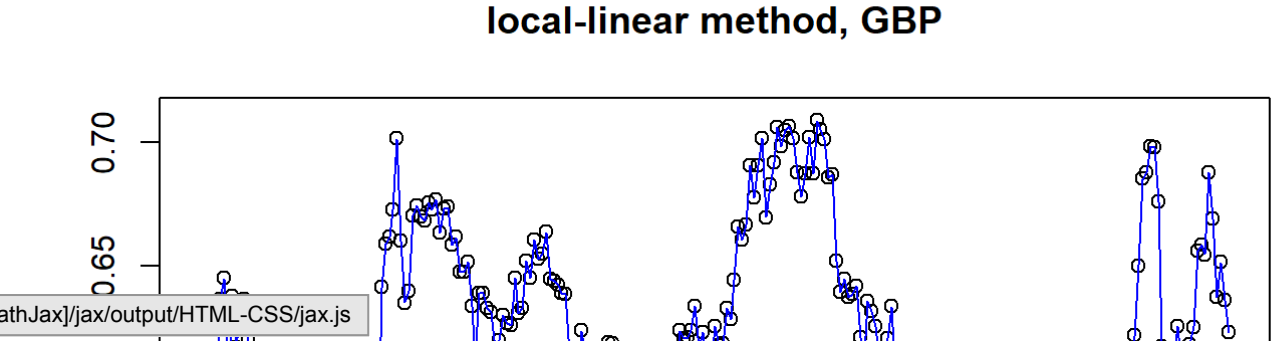
bandwidth로는 bw를 사용한다.

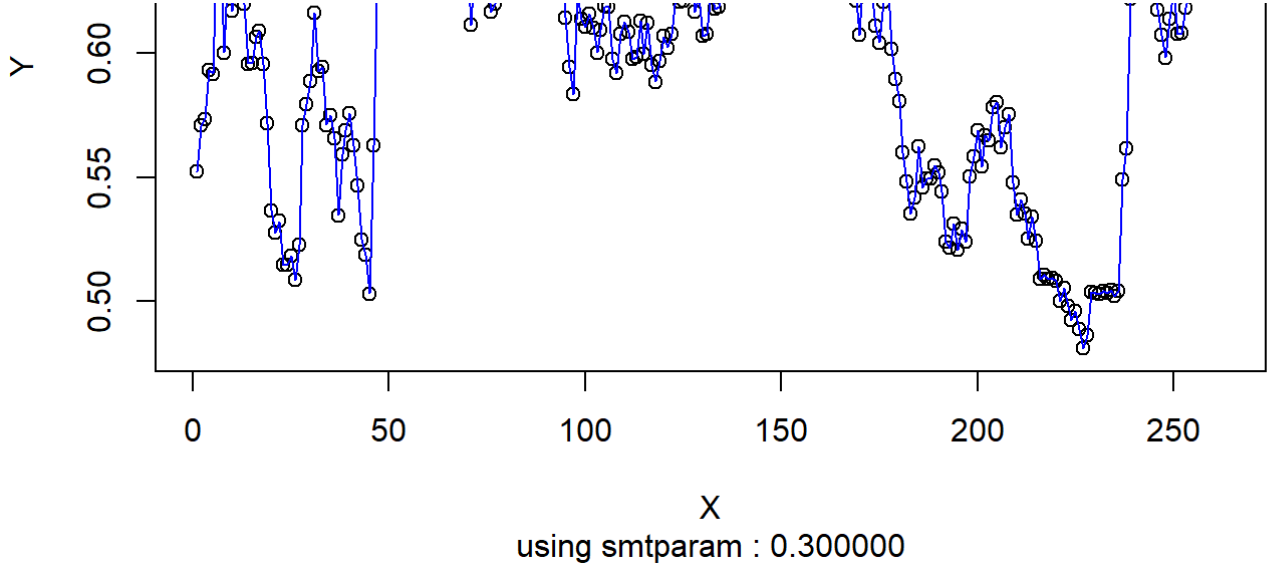
```
library(locpol) #install.packages("locpol")

estimate_local_linear <- function(xdata, ydata, bw, estim_x = NULL)
{
    df <- data.frame(x = xdata, y = ydata)

    if(is.null(estim_x))
    {
        lp.fit <- locpol(y ~ x,df, bw = bw, kernel = gaussK, deg = 1, xeval = df$x)
    }
    else
    {
        lp.fit <- locpol(y ~ x,df, bw = bw, kernel = gaussK, deg = 1, xeval = estim_x)
    }
    ey <- fitted(lp.fit)

    return(ey)
}
```

bandwidth를 0.1~1.5까지 주고, 0.1 간격으로 각각에 대해 test를 해보자. 다음과 같이 가장 CV를 작게 만드는 RSS값을 찾을 수 있다.

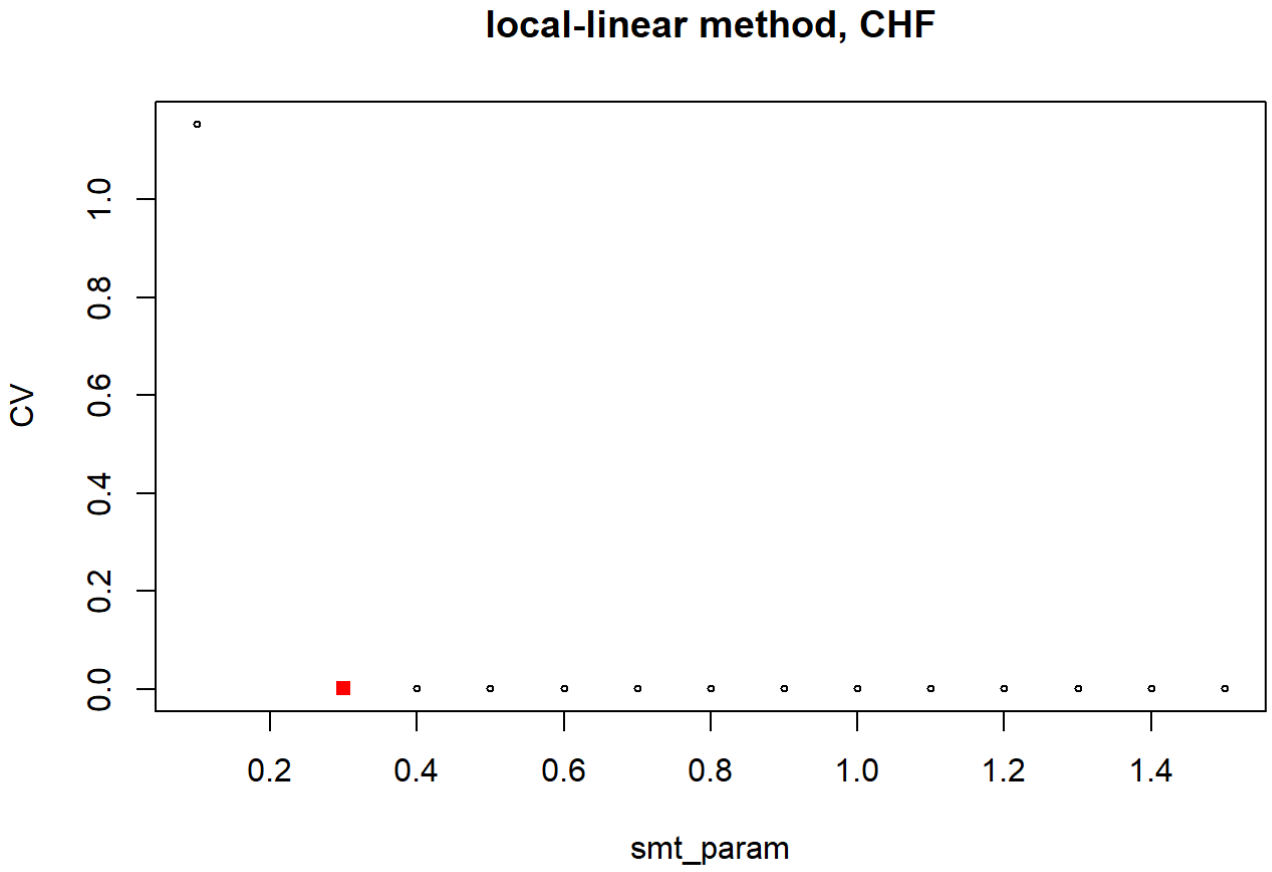
```
rss_GBP[2] <- full_test(estimate_local_linear, data$X, data$GBP, seq(0.1,1.5,by = 0.1), description = "local-line
ar method, GBP")
```

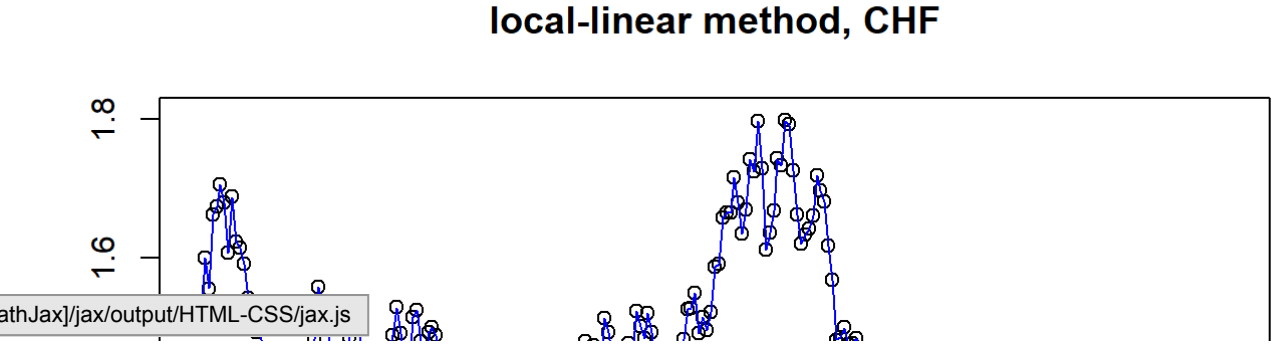


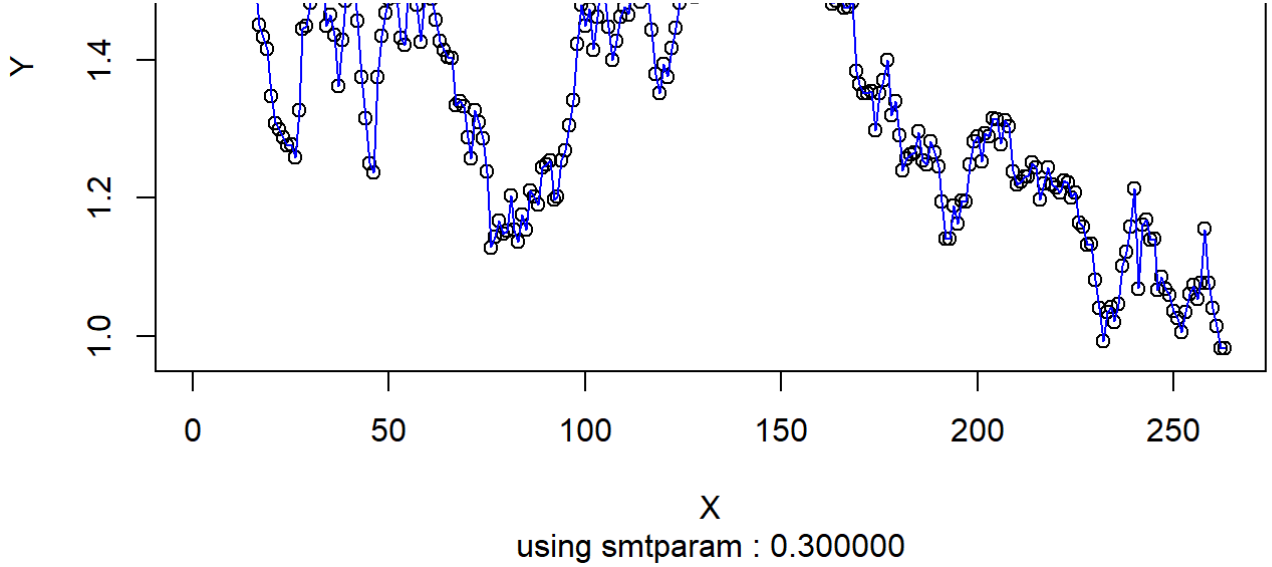




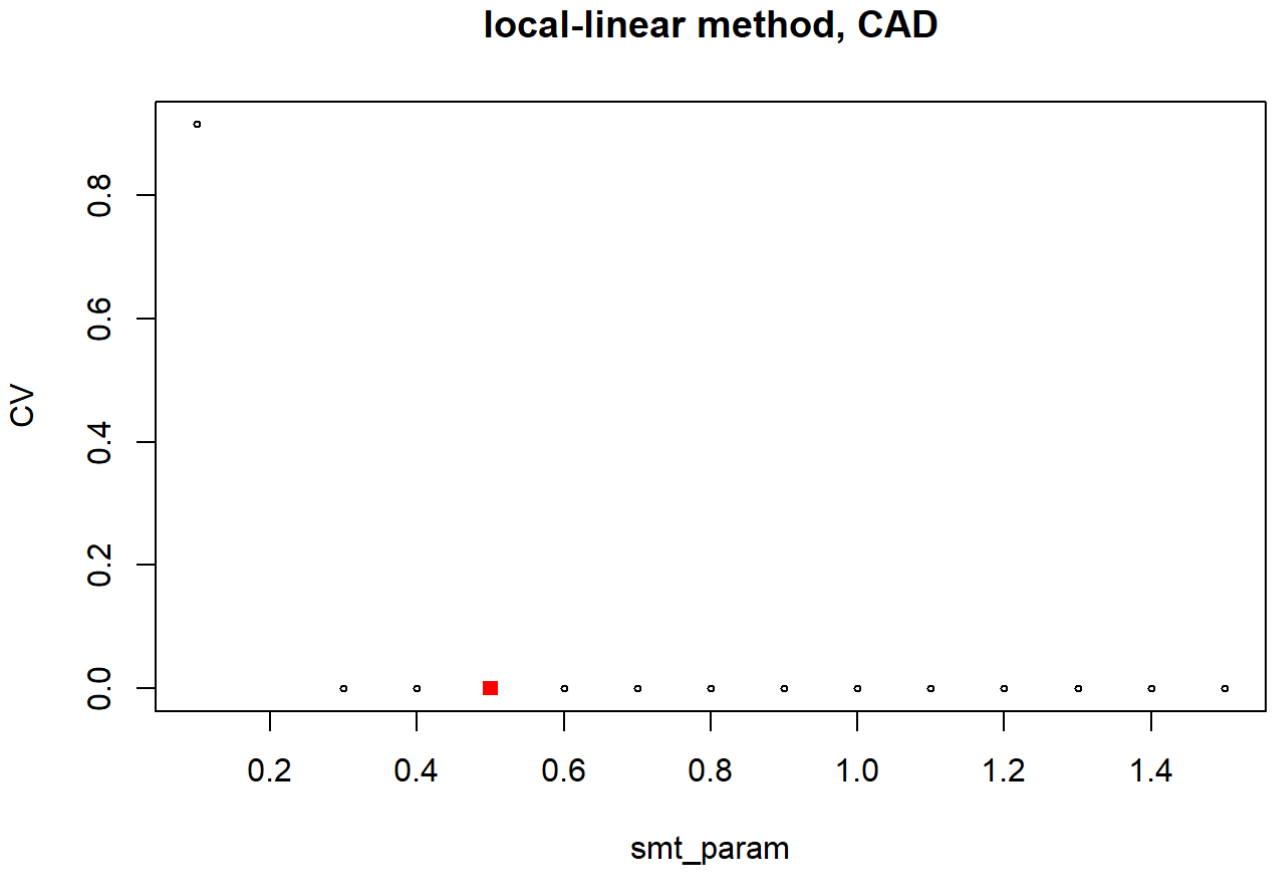

```
rss_CHF[2] <- full_test(estimate_local_linear, data$X, data$CHF, seq(0.1,1.5,by = 0.1), description = "local-line
ar method, CHF")
```

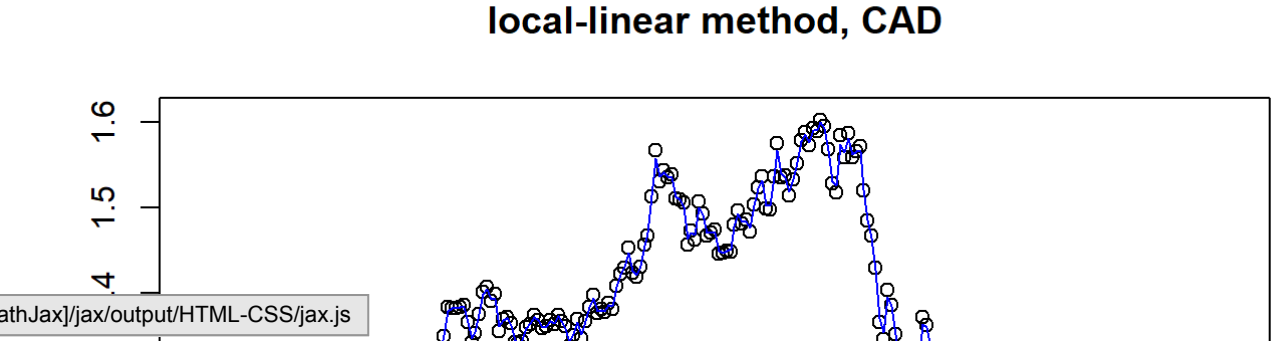


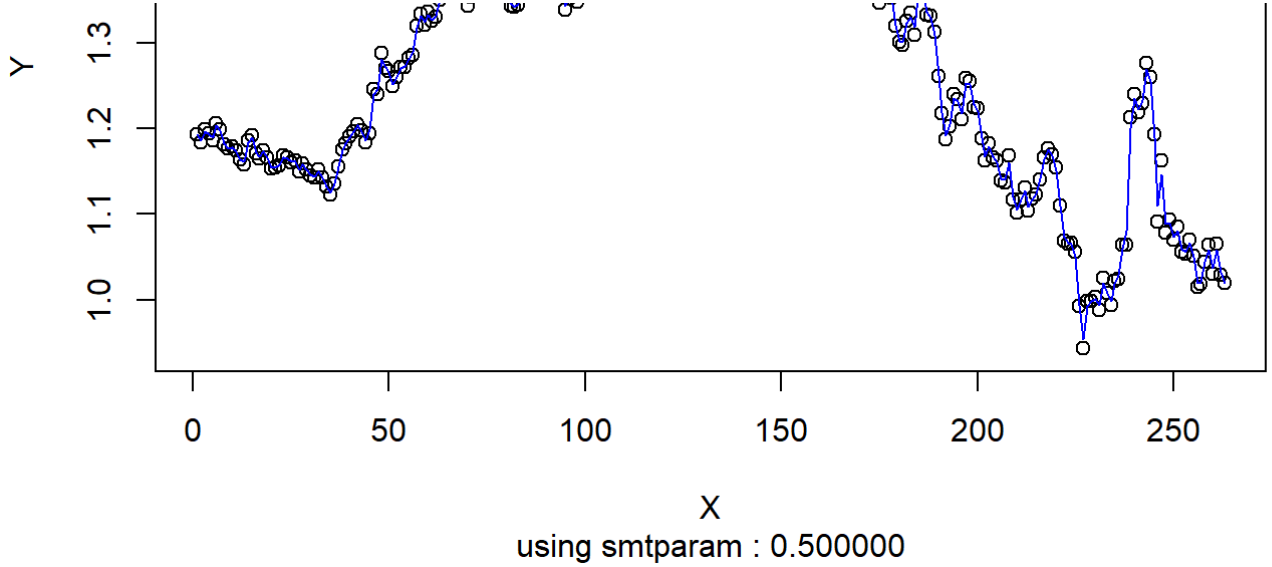





```
rss_CAD[2] <- full_test(estimate_local_linear, data$X, data$CAD, seq(0.1,1.5,by = 0.1), description = "local-line
ar method, CAD")
```







smoothing spline

smoothing spline은,

$$E_{ss} = \sum_{i=1}^{n}\left(Y_i - m\left(X_i\right)\right)^2 + \lambda\int_{-\infty}^{\infty}\left(m^{(2)}(x)\right)^2 dx$$

$$E_{ss} = \left(Y - Rb - Q^t a\right)^t\left(Y - Rb - Q^t a\right) + \lambda b^t R b$$

위 식을 최소화하는 cubic spline을 구하고,

$$s_{finite}(x) = a_0 + a_1 x + \frac{1}{12}\sum_{j=1}^{n} b_j |x - X_j|^3$$

($X_1 \le x \le X_n$의 범위에서 위와 같은 형태)

위 spline을 통해 x값을 추정하는것이다.

여기서

$$\mathbf{Q} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ X_1 & X_2 & X_3 & \dots & X_n \end{pmatrix}$$

,

$$\mathbf{R} = \begin{pmatrix} 0 & \frac{|X_1 - X_2|^3}{12} & \frac{|X_1 - X_3|^3}{12} & \cdots & \frac{|X_1 - X_n|^3}{12} \\ \frac{|X_2 - X_1|^3}{12} & 0 & \frac{|X_2 - X_3|^3}{12} & \cdots & \frac{|X_2 - X_n|^3}{12} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{|X_n - X_1|^3}{12} & \frac{|X_n - X_2|^3}{12} & \frac{|X_n - X_3|^3}{12} & \cdots & 0 \end{pmatrix}$$

이다.

위 식을 최소화하는 계수행렬 (b, a)는 미분을 통해 구할 수 있음을 안다.

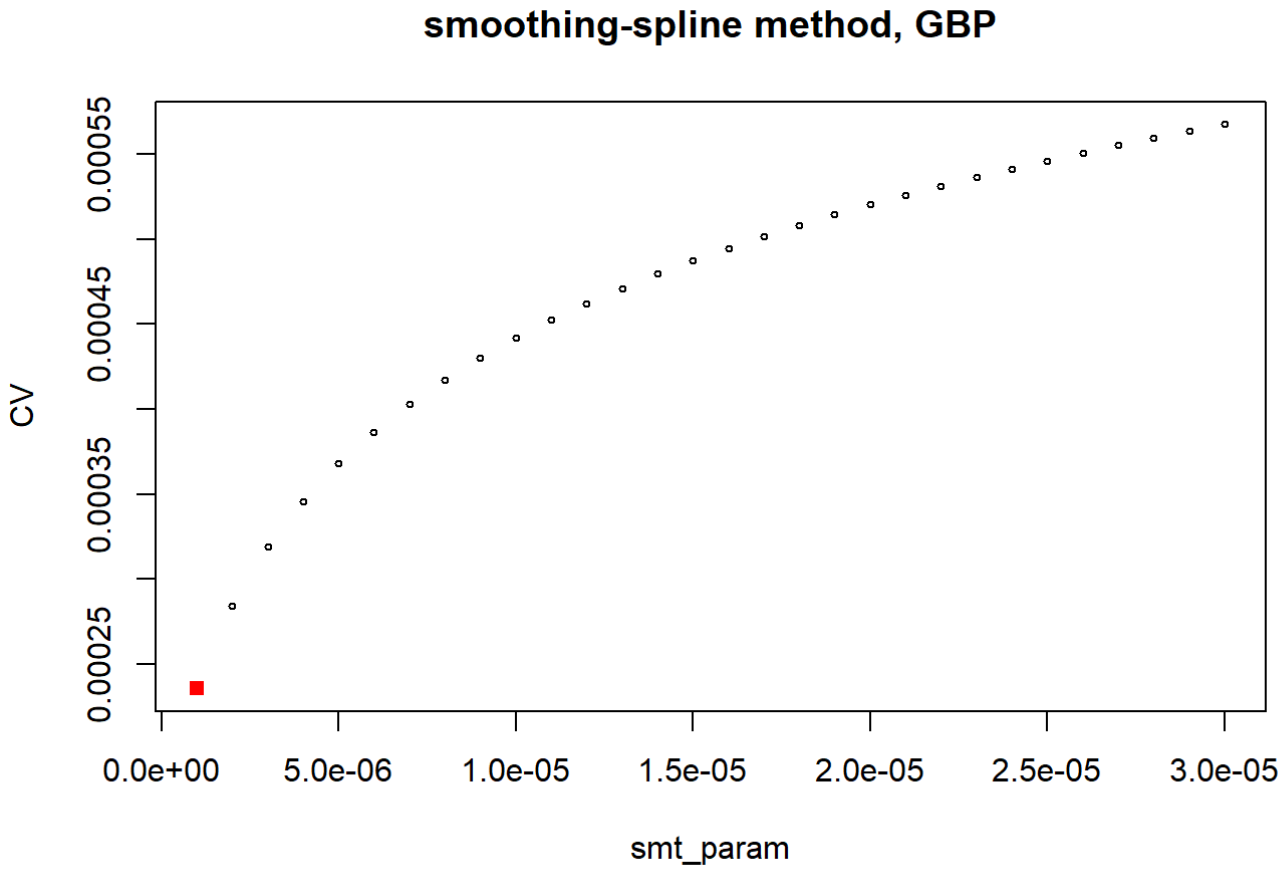
여기서의 smoothing parameter는 λ이고, 이 smoothing spline은 R에서 smooth.spline 함수를 통해 지원되므로, 이를 이용하도록 하겠다.

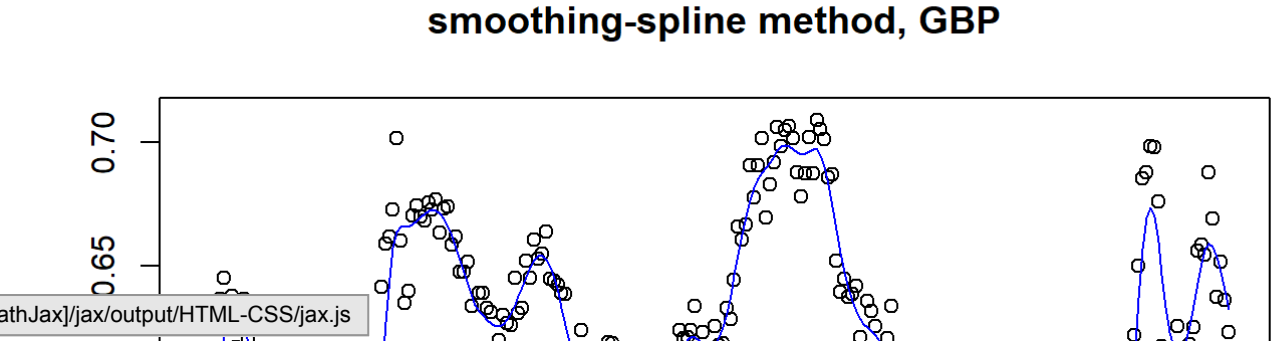
```
estimate_smoothing_spline <- function(xdata, ydata, lamb, estim_x = NULL)
{

    if(is.null(estim_x))
    {

        fit.sp <- smooth.spline(xdata, ydata, lambda = lamb, all.knots = TRUE)

        pred <- predict(fit.sp, xdata)
    }
    else
```

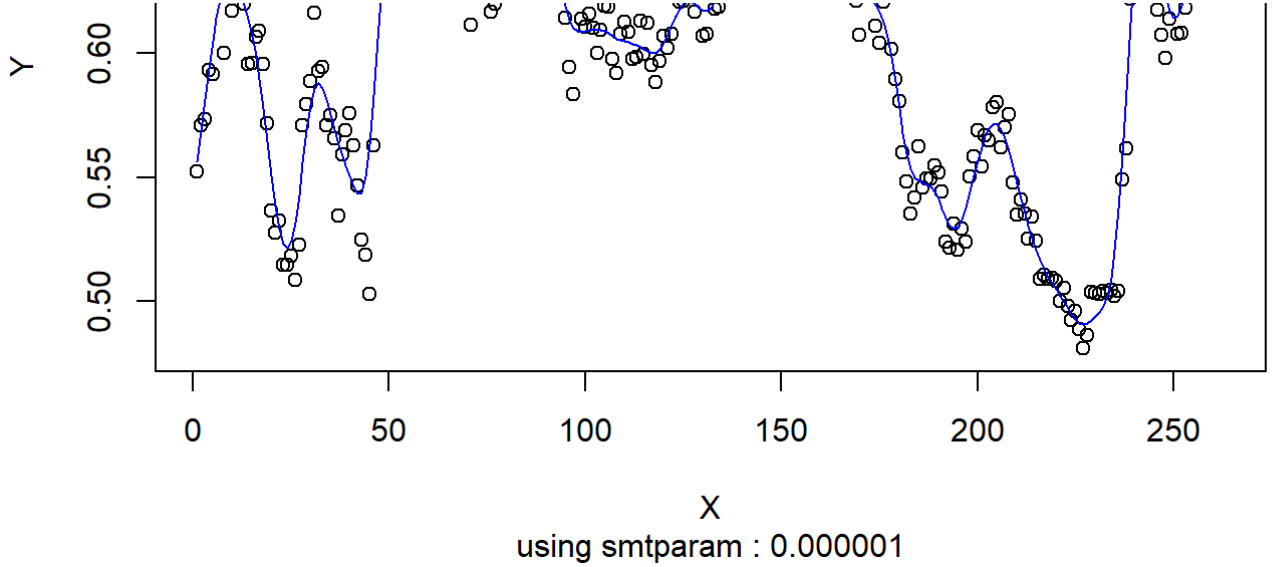
```
        fit.sp <- smooth.spline(xdata, ydata, lambda = lamb, all.knots = TRUE)

        pred <- predict(fit.sp, estim_x)
    }

    return(pred$y)
}
```

아래는 lambda = lambda로 0.000001~0.00003까지, 0.000001 간격으로 사용했을때, 가장 CV를 작게 만들어주는 lambda를 이용하여 추정한 값이다.

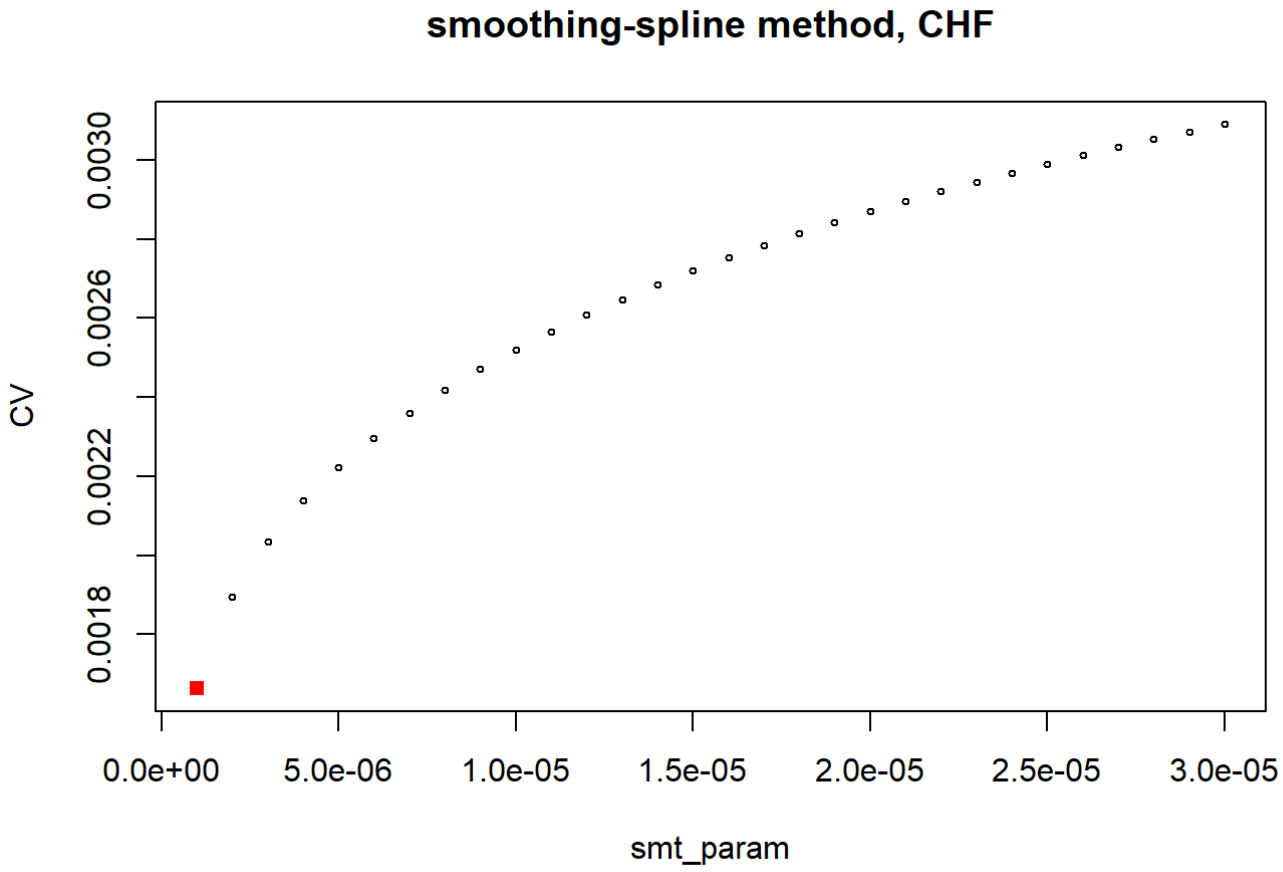
```
rss_GBP[3] <- full_test(estimate_smoothing_spline, data$X, data$GBP, seq(0.000001,0.00003,by = 0.000001), descrip
tion = "smoothing-spline method, GBP")
```

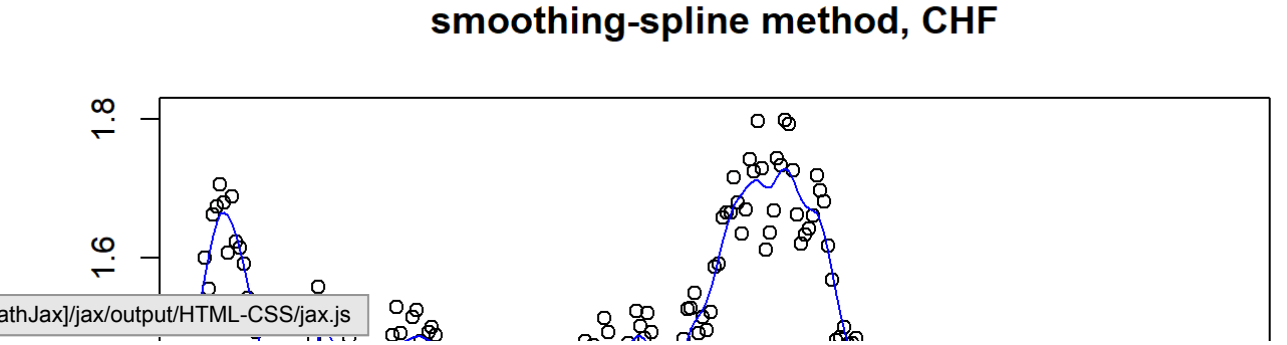


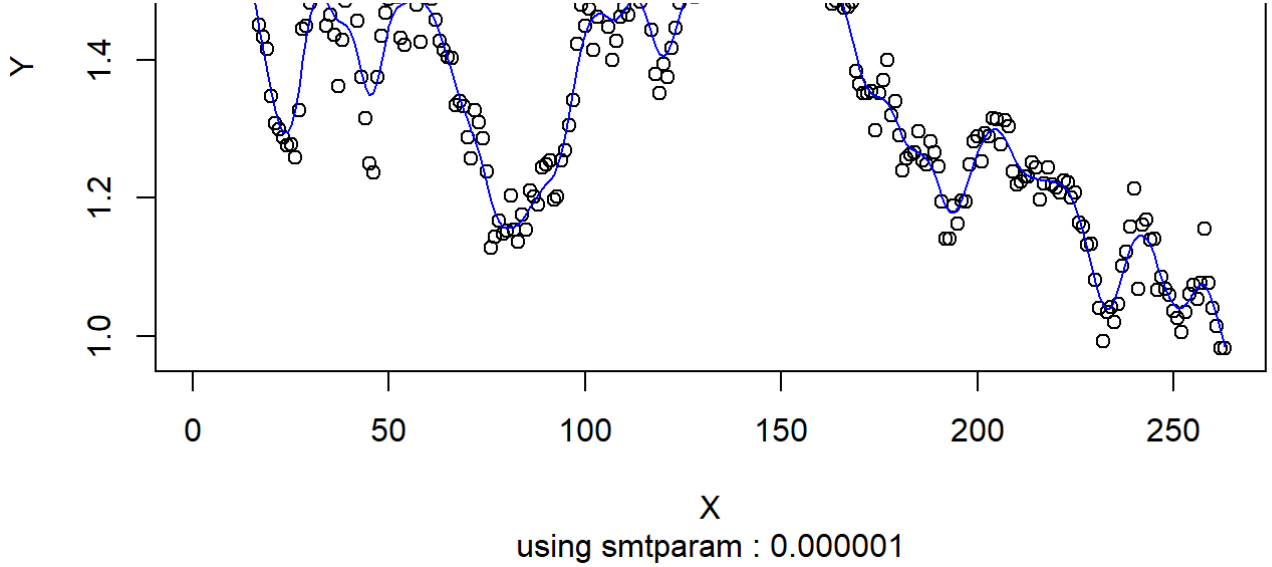




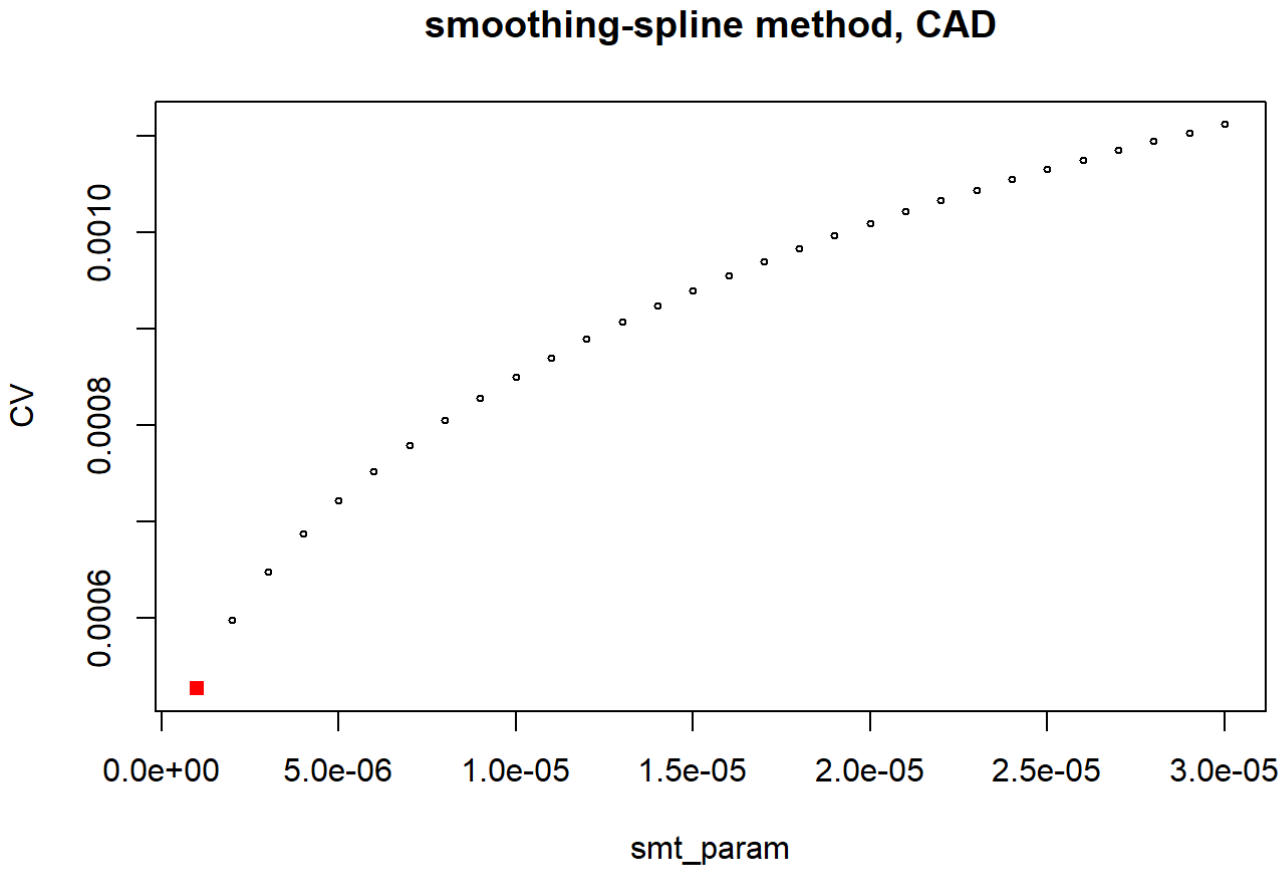

```
rss_CHF[3] <- full_test(estimate_smoothing_spline, data$X, data$CHF, seq(0.000001,0.00003,by = 0.000001), descrip
tion = "smoothing-spline method, CHF")
```

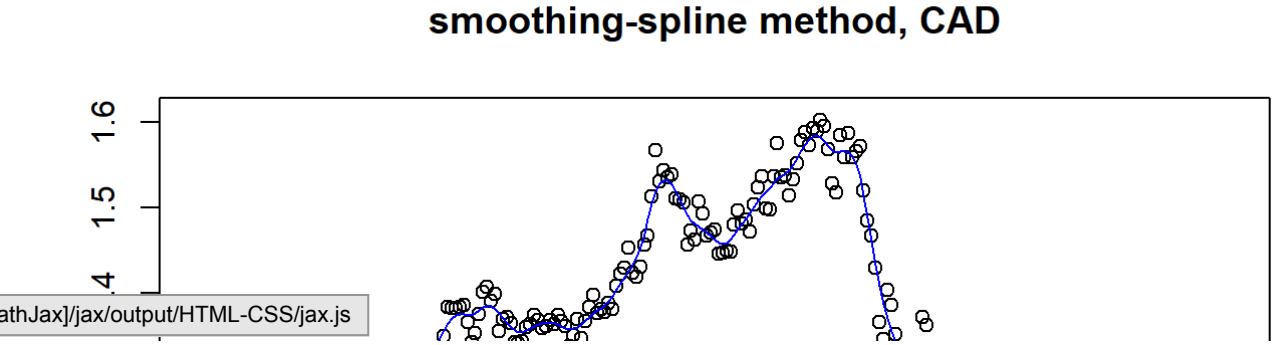


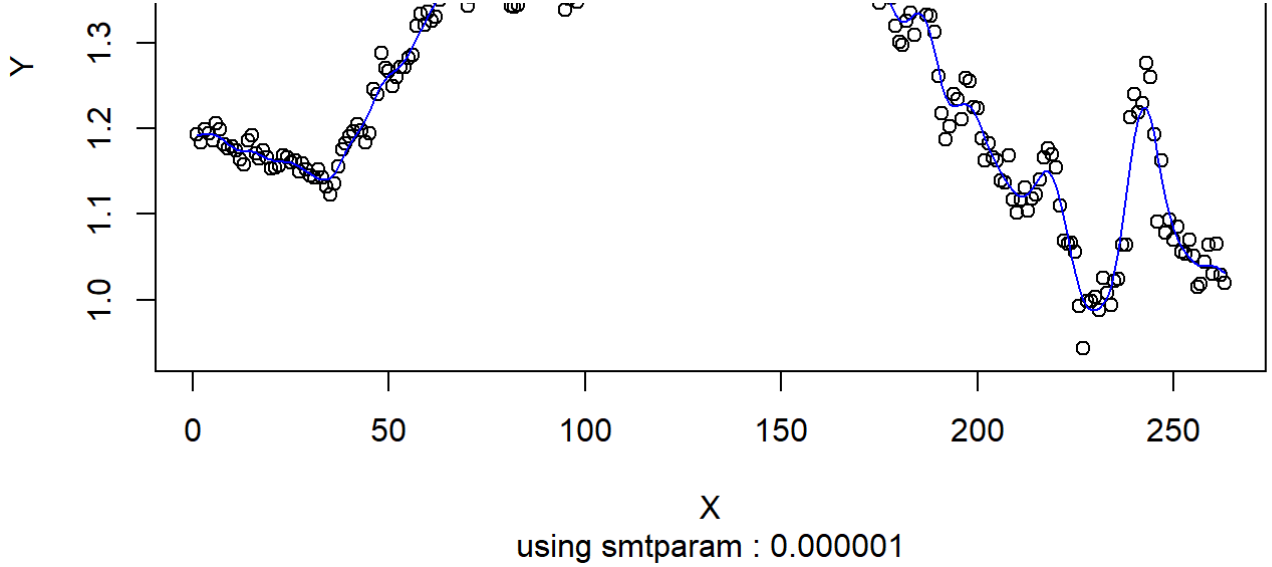





```
rss_CAD[3] <- full_test(estimate_smoothing_spline, data$X, data$CAD, seq(0.000001,0.00003,by = 0.000001), descrip
tion = "smoothing-spline method, CAD")
```







번외. natural spline

smoothing spline에서 lambda = 0으로 주게되면 이는 natural spline이 되고, natural spline으로 추정된 그래프는 datapoint를 interpolation 하는 그래프가 된다.

즉, data point로 X_i, Y_i가 주어지면, $\hat{m}(X_i) = Y_i$가 된다.

따라서, 앞에서 정의된 RSS의 정의상, 이렇게 유도된 그래프는 RSS의 값을 0으로 만들고, 따라서 RSS만을 최적화 하는것이 목적이면, data point를 interpolate 하는 그래프를 사용하면 목표를 이룰 수 있다.

R에서 natural spline은 spline 함수를 사용하여 구할 수 있으므로, estimate_func을 다음과 같이 짜자.

lambda는 사용할 필요가 없지만, 함수 형태를 맞추기 위해 들어가는 dummy variable이다.

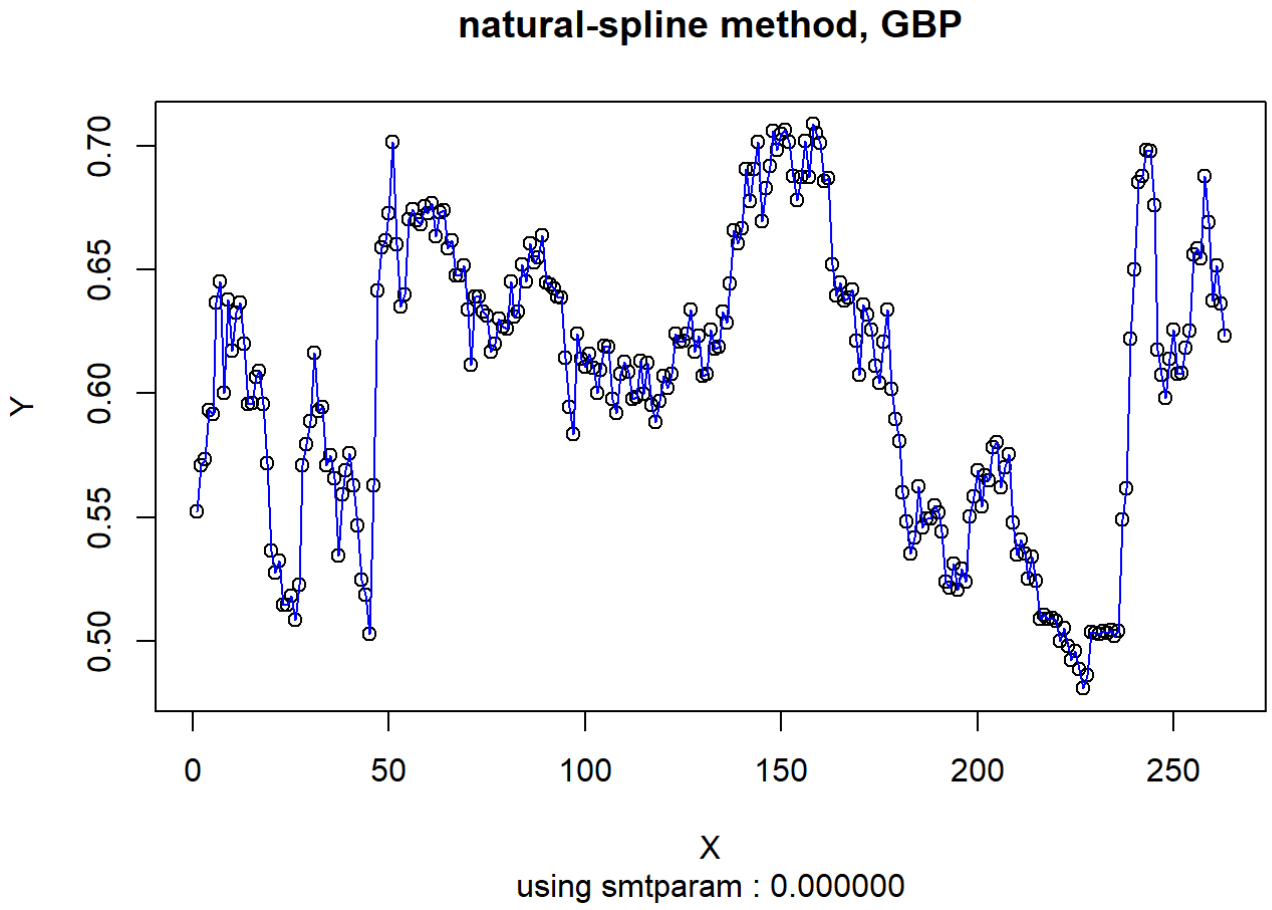
```
estimate_natural_spline <- function(xdata, ydata, lamb = 0, estim_x = NULL)
{
    if(is.null(estim_x))
    {
        exy <- spline(xdata, ydata, method = 'natural', xout = xdata)
    }
    else
```

```
    {
        exy <- spline(xdata, ydata, method = 'natural', xout = estim_x)
    }

    return(exy$y)
}
```

이를 이용하여, 앞에서와 같이 test를 해보자.

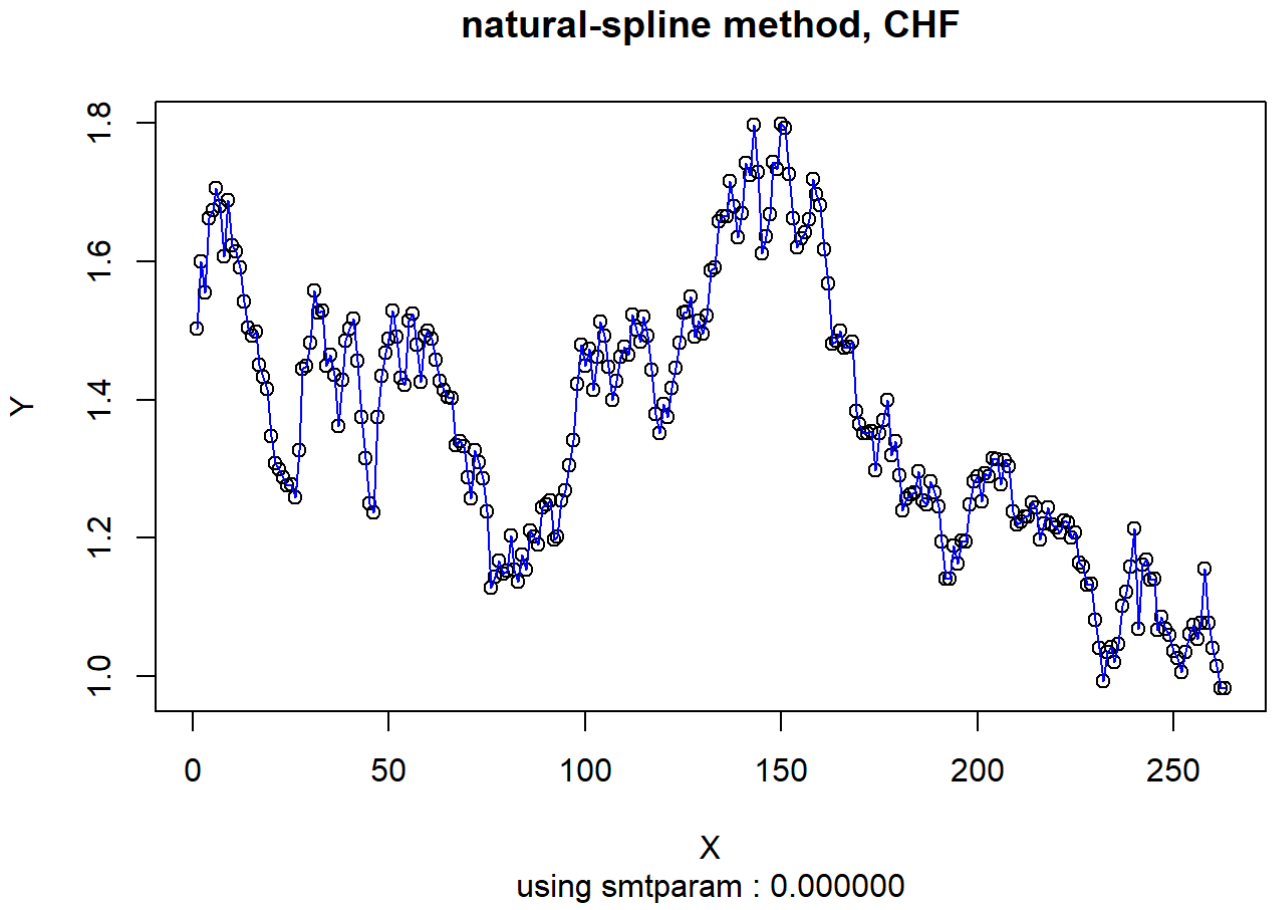
natural spline은 data point를 interpolation하므로, Cross validation의 값은 H_{ii}가 모두 1에서, 앞에서와 같이 Cross validation을 구하면 0으로 나눗셈이 되어 오류가 발생할 것이므로, cross validation의 값을 구하지 않고 넘어가겠다.

```
full_test(estimate_natural_spline, data$X, data$GBP, c(0), description = "natural-spline method, GBP", using_CV =
 FALSE)
```



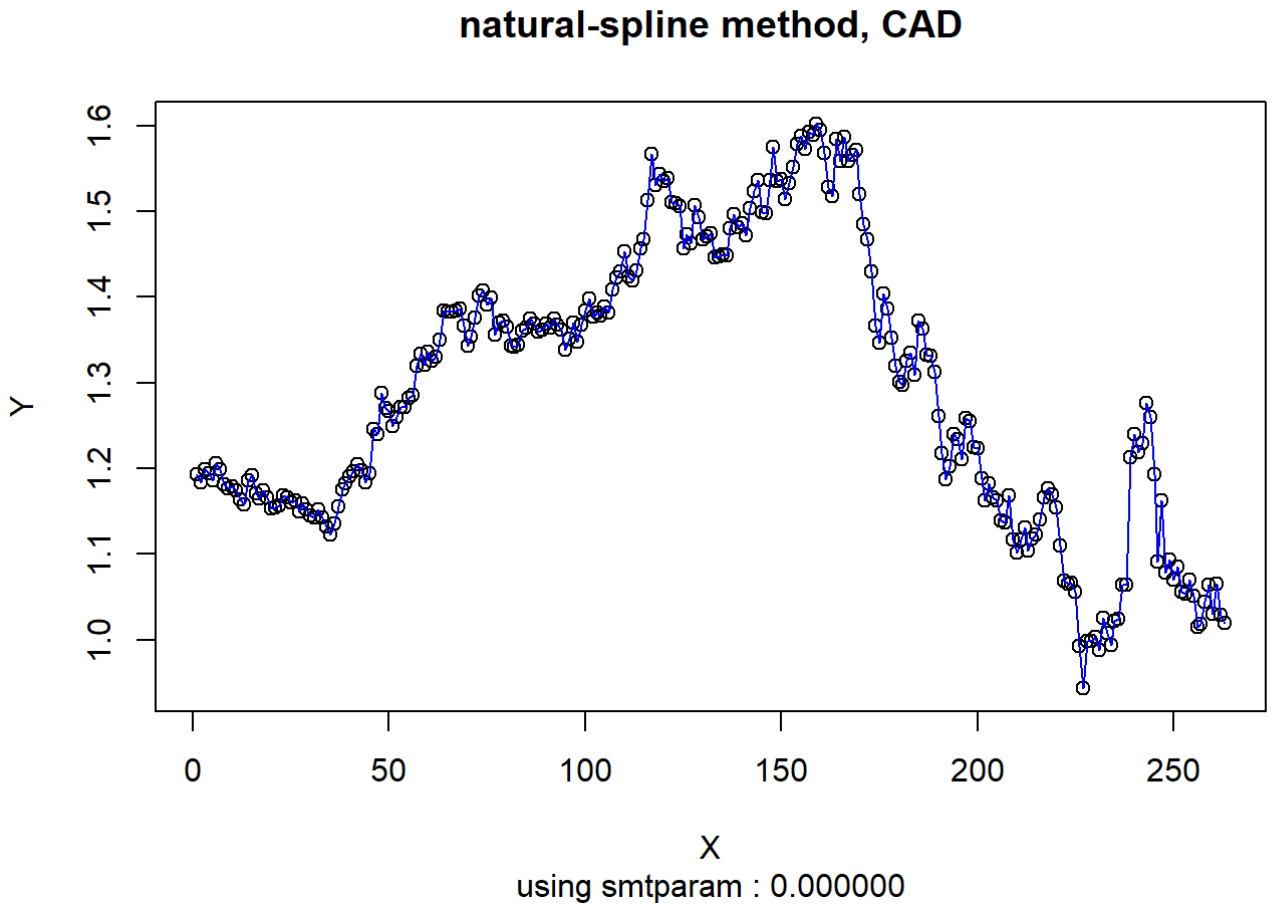

```
## [1] 0
```

```
full_test(estimate_natural_spline, data$X, data$CHF, c(0), description = "natural-spline method, CHF", using_CV =
 FALSE)
```




```
## [1] 0
```

```
full_test(estimate_natural_spline, data$X, data$CAD, c(0), description = "natural-spline method, CAD", using_CV =
 FALSE)
```




```
## [1] 0
```

각각 RSS의 값들이 모두 0인것을 알 수 있다. 즉, smoothing을 하지 않고, data point를 interpolate하는 method를 사용하는것이 RSS의 값을 가장 낮추는것을 알 수 있다.

natural spline using binomial filter

데이터가 equispaced 이므로, binomial filter를 사용해서 각 데이터에 대해 smoothing이 가능하다.

이렇게 binomial filter를 이용해 smooting된 데이터를 사용해서 natural spline을 구함으로써, 데이터를 smooting하는것이 가능하다.

책의 chapter 2에서 사용했던 코드를 그대로 이용해서, binomial filterling을 할 수 있고, 이렇게 binomial filtering된 데이터를 이용하여 natural spline을 구하는 방식으로 smoothing을 진행하자.

binomial filter의 smooting parameter는 m이므로, 다음과 같이 estimate_func를 짜겠다.

```
binom1<-function(yy, mm)
{
#(1)
  nd <- length(yy)
#(2)
  mm2 <- mm * 0.5
#(3)
  yywl <- yy
  yyw2 <- yy
  rlim <- mm2
  yyr <- NULL
  count <- 0
  while(rlim > nd) {
    yyw1 <- rev(yyw1)
    yyr <- c(yyr, yyw1)
    rlim <- rlim - nd
    count <- count + 1
  }
  switch(count %% 2 + 1,
         yyr <- c(yyr, yy[nd:(nd - rlim + 1)]),
         yyr <- c(yyr, yy[1:rlim]))
  llim <- mm2
  yyl <- NULL
  while (llim > nd) {
    yyw2 <- rev(yyw2)
    yyl <- c(yyw2, yyl)
    llim <- llim - nd
  }
  switch(count %% 2 + 1,
         yyl <- c(yy[llim:1], yyl),
         nd - llim + 1):nd], yyl))
```

```
  y2 <- matrix(c(yyl, yy, yyr), ncol = 1)
#(4)
  ww <- matrix(0, ncol = nd + mm, nrow = nd + mm)
#(5)
  imat <- row(ww)
  jmat <- col(ww)
#(6)
  check <- 0 <= (mm2 + imat - jmat) & (mm2 + imat - jmat) <= mm
#(7)
  ww[check] <- exp(lgamma(mm +1) -
                     lgamma(mm2 + imat[check] - jmat[check] + 1) -
                     lgamma(mm2 - imat[check] + jmat[check] + 1) -
                     mm * logb(2))
#(8)
  ey <- ww %*% y2
  ey <- as.vector(ey[(mm2 + 1):(nd + mm2)])
#(9)
  return(ey)
}
```

먼저, binomial filter를 해주는 binom1 함수를 위와같이 정의한다.

위 함수는 equispaced인 data에 대해, smoothing parameter m과 y값들을 이용해서

binomial filter를 이용해서 추정한 $\hat{m}(X_i)$를 리턴해준다.

우리의 데이터는 equispaced data이므로, 위 함수를 이용하여 binomial filtering이 가능하다.

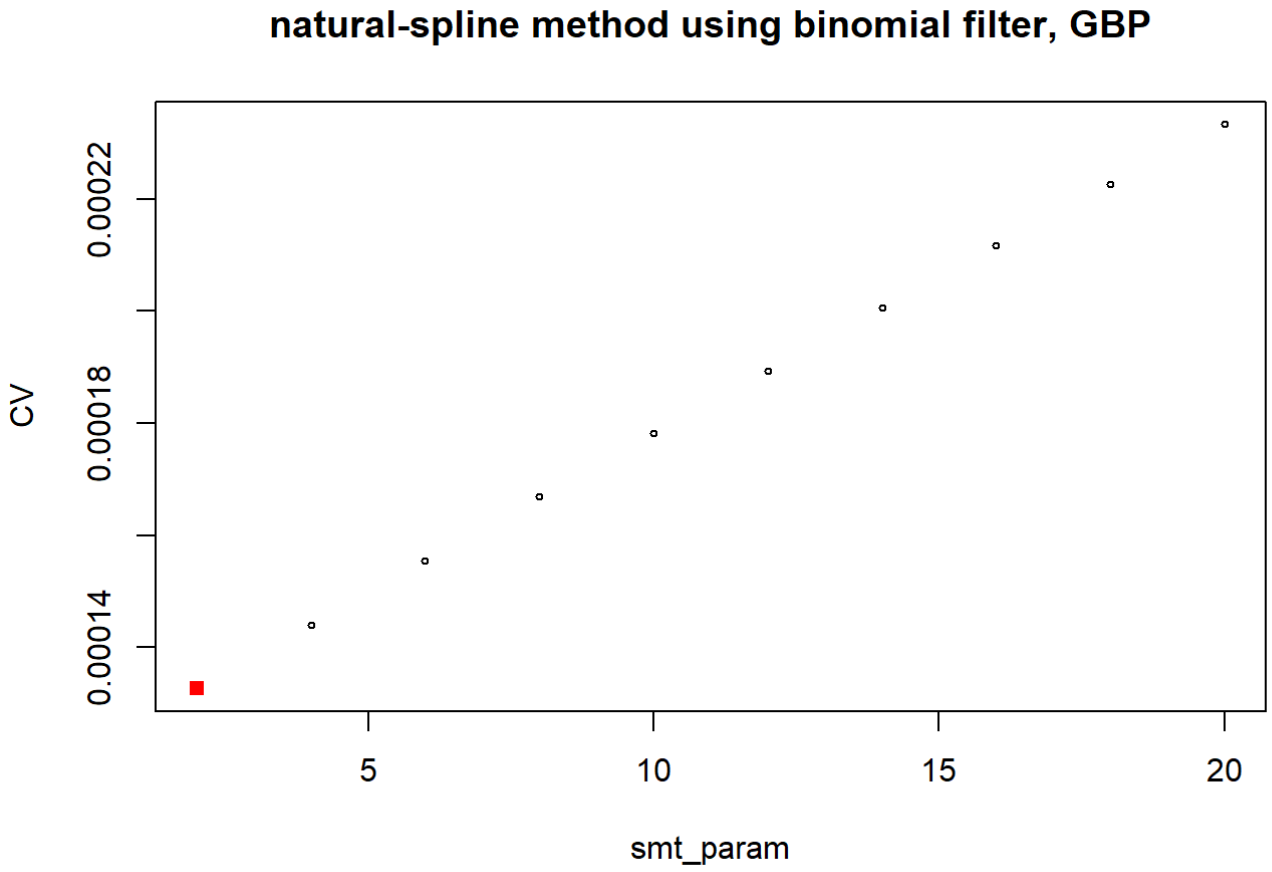
이제, 앞과 같이 natural spline을 이용하여 smoothing을 해보자.

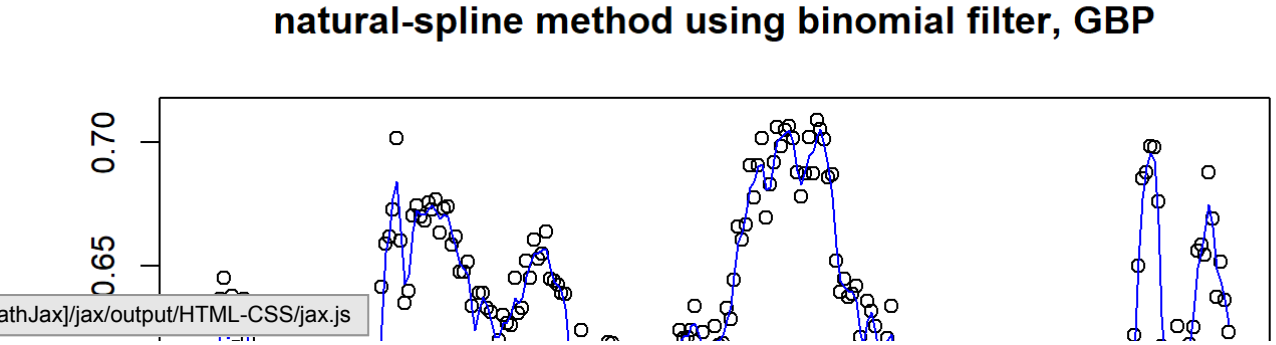
```
estimate_natural_spline_binomfilter <- function(xdata, ydata, m, estim_x = NULL)
{
    biny <- binom1(ydata, m)

    if(is.null(estim_x))
    {
        exy <- spline(xdata, biny, method = 'natural', xout = xdata)
```

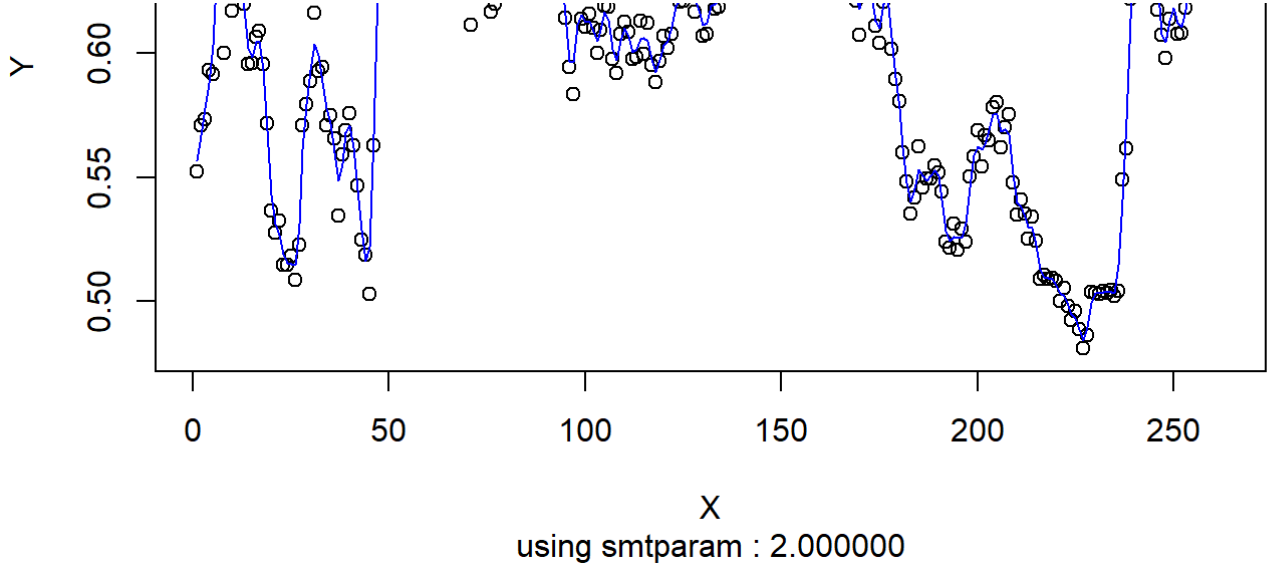
```
    else
    {
        exy <- spline(xdata, biny, method = 'natural', xout = estim_x)
    }

    return(exy$y)
}
```

이제, 앞에서와 같이 이를 이용하여 testing을 해보자. m은 짝수여야하므로, 2,4,…20까지의 값을 사용하도록 하겠다.

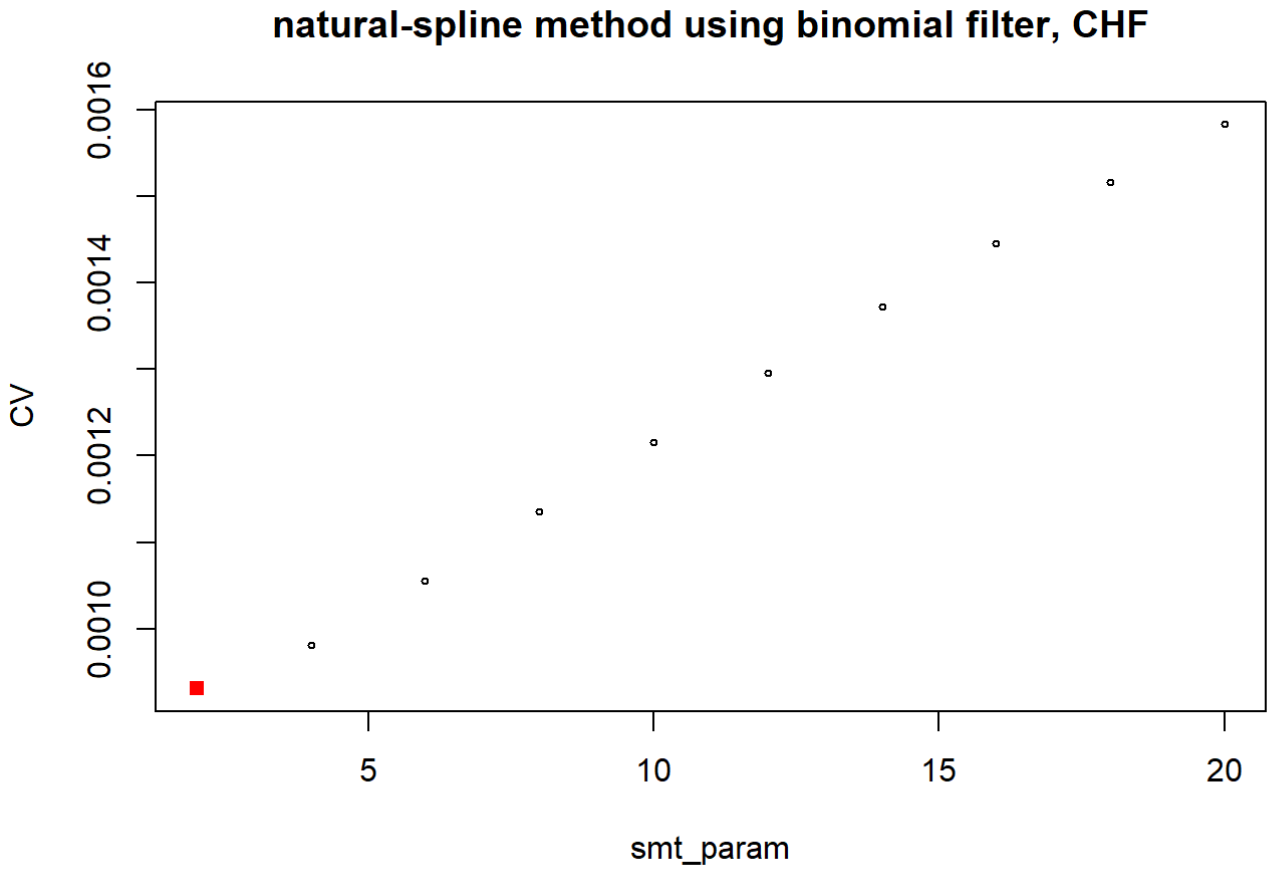
```
rss_GBP[4] <- full_test(estimate_natural_spline_binomfilter, data$X, data$GBP, seq(from = 2, to = 20, by = 2), de
scription = "natural-spline method using binomial filter, GBP")
```

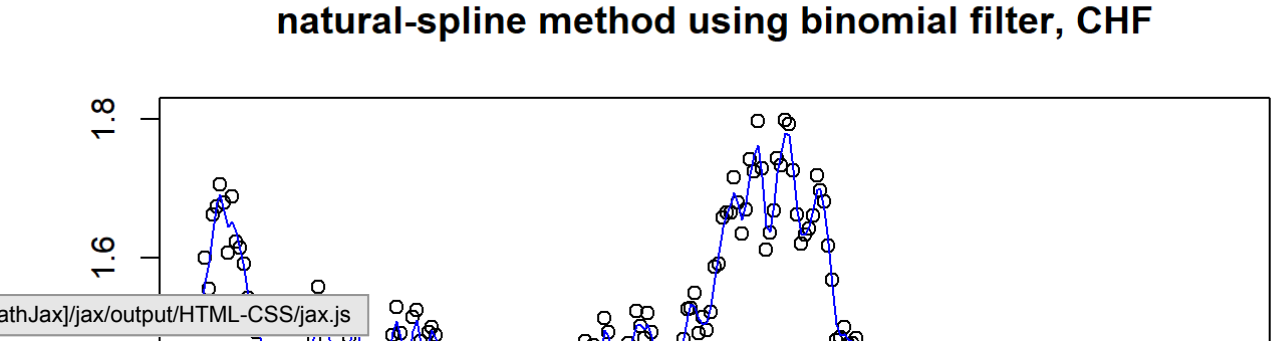


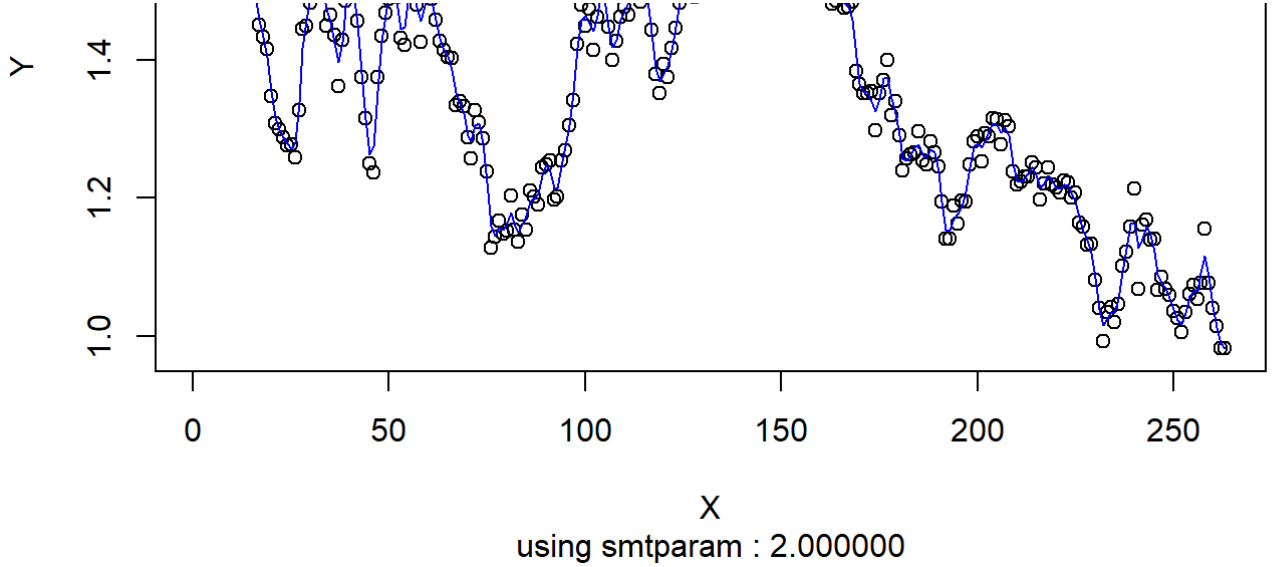




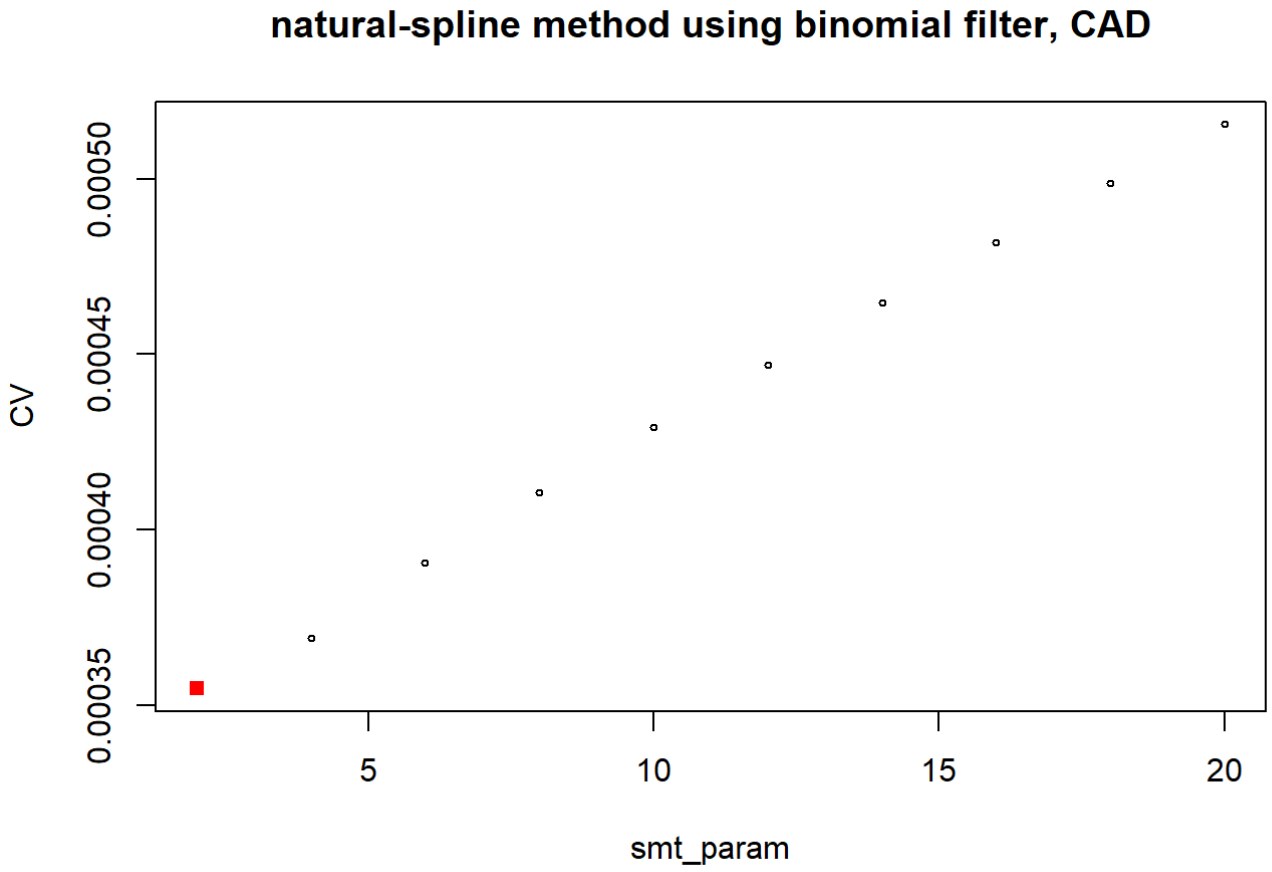

```
rss_CHF[4] <- full_test(estimate_natural_spline_binomfilter, data$X, data$CHF, seq(from = 2, to = 20, by = 2), de
scription = "natural-spline method using binomial filter, CHF")
```

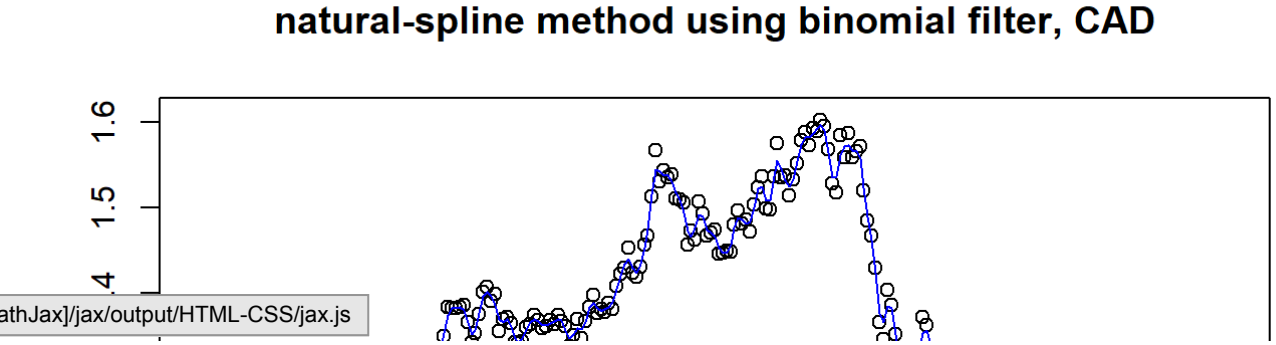


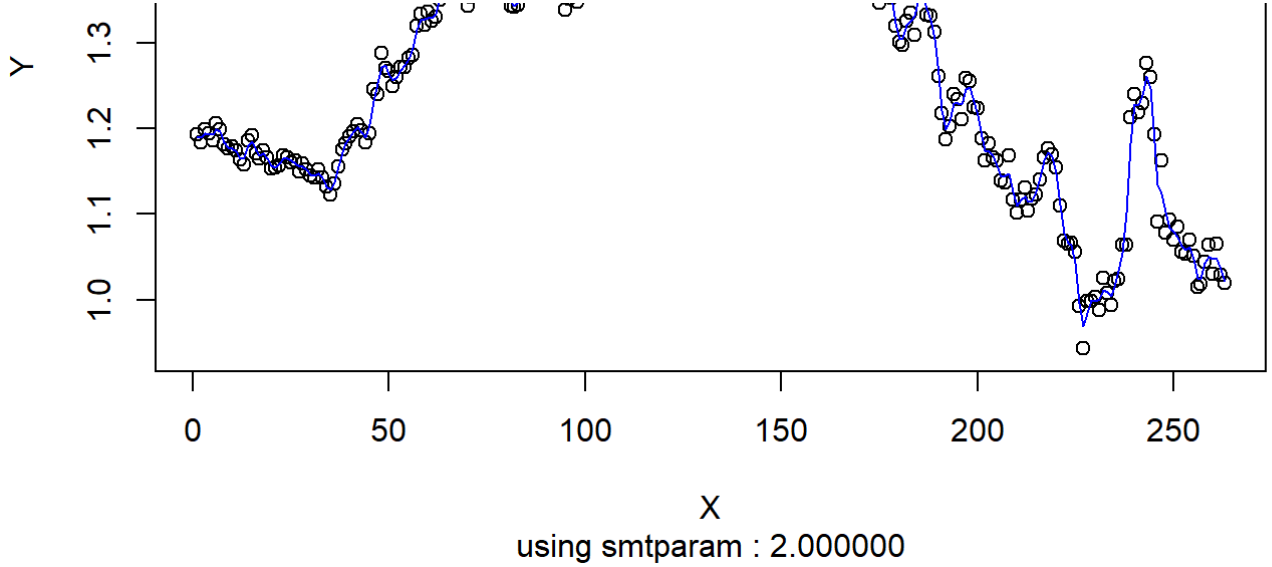





```
rss_CAD[4] <- full_test(estimate_natural_spline_binomfilter, data$X, data$CAD, seq(from = 2, to = 20, by = 2), de
scription = "natural-spline method using binomial filter, CAD")
```







위와 같다.

LOESS

LOESS는 local polynomial regression의 개선된 버전으로, 각 data point에서의 bandwidth를 데이터가 dense한 부분에서는 작게, 데이터가 sparse한 부분에서는 크게 가져가는 방법이다.

local polynomial regression과 비슷하게,

다음과 같은 식을 최소화 하는 $a(x^*)$을 구한 다음 그중 $a_0(x^*)$을 $\hat{m}(x^*)$로 사용하게 된다.

$$E_{LOESS}\left(x^*\right) = \sum_{i=1}^{n}\left(w\left(\frac{X_i - x^*}{h_k\left(x^*\right)}\right)\left(a_0\left(x^*\right) + \sum_{j=1}^{p} a_i\left(x^*\right)\left(X_i - x^*\right)^j - Y_i\right)^2\right)$$

이때, local polynomial regression과의 차이점으로는 $h_k(x^*)$이 고정된 값이 아니라, k와 x^*에 대한 함수가 된다는 것인데, 여기서 k는 span이라는 smoothing parameter에 비례하는 값으로써, k번째로 가까운 데이터까지의 거리를 bandwidth로 사용하게 된다.

span을 s라 하면, $k = [n \cdot s]$로 계산된다.

예를들어, s를 0.05로 주면 $h_k(x^*)$값으로, x^*에서
전체데이터의 5%번째만큼 떨어진 값까지의 거리를 사용하게 된다.

여하튼, 이러한 loess는 R에서 loess 함수를 이용해서 사용할 수 있다.

local polynomial regression의 경우에서와 같이, kernel 함수로는 다양한 함수를 사용할 수 있지만, 여기서는 gaussian kernel을 사용할 것이다.

또한, 사용할 다항식의 차수도 정할 수 있는데, 여기서는 linear한 경우와 2차식인 경우를 사용하도록 하겠다.

따라서, 다음과 같이 estimate function을 정의하자.

```
estimate_loess_linear <- function(xdata, ydata, sp, estim_x = NULL)
{
    df <- data.frame(x = xdata, y = ydata)
    fit.lo <- loess(y ~ x, data = df, span = sp, family = "gaussian", degree = 1, surface = "direct")
    if(is.null(estim_x))
    {
        ey <- predict(fit.lo)
    }
    else
    {
        ey <- predict(fit.lo, estim_x)
    }
    return(ey)
}

estimate_loess_quad <- function(xdata, ydata, sp,  estim_x = NULL)
{
    df <- data.frame(x = xdata, y = ydata)
    fit.lo <- loess(y ~ x, data = df, span = sp, family = "gaussian", degree = 2, surface = "direct")

    if(is.null(estim_x))
    {
        ey <- predict(fit.lo)
    }
    else
    {
                                fit.lo, estim_x)
```

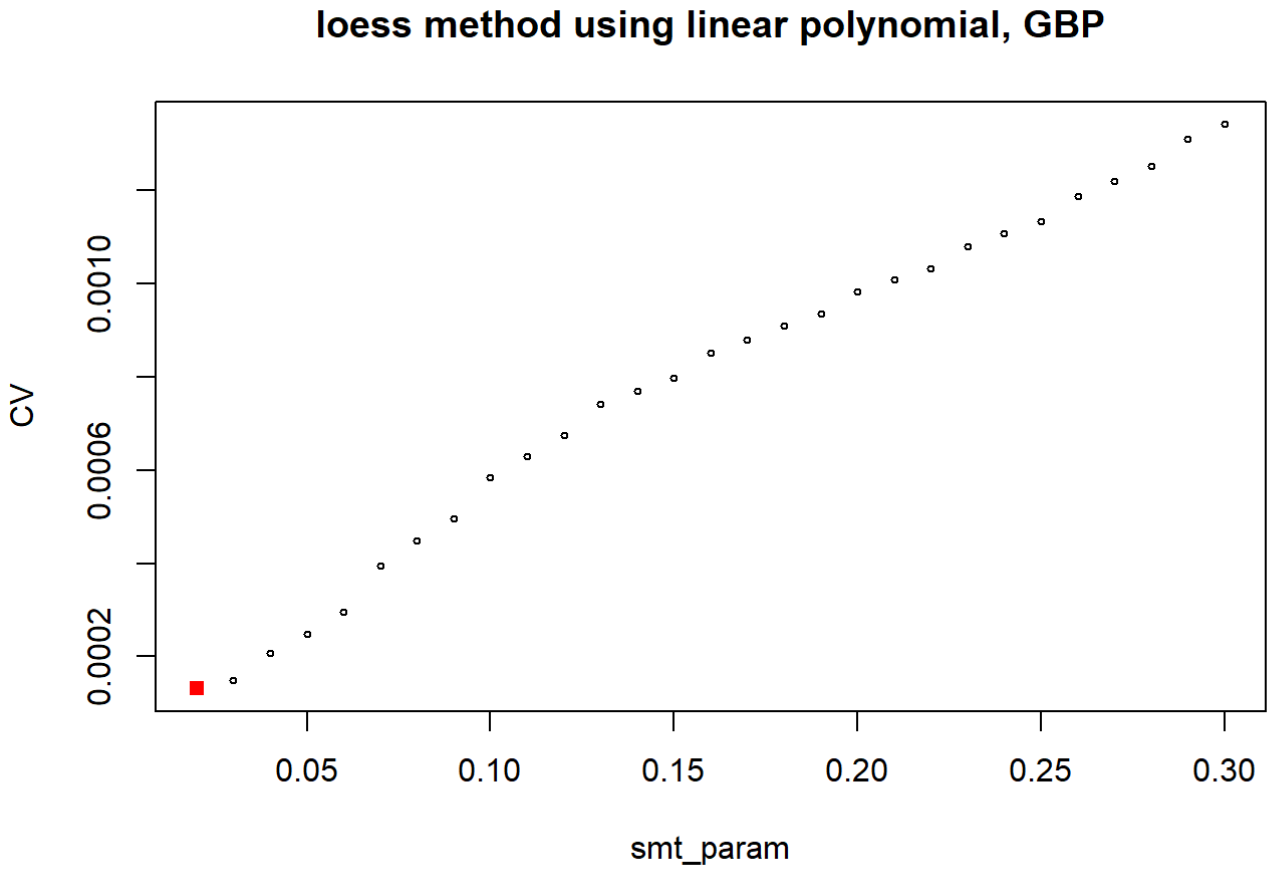
```
    }
    return(ey)
}
```

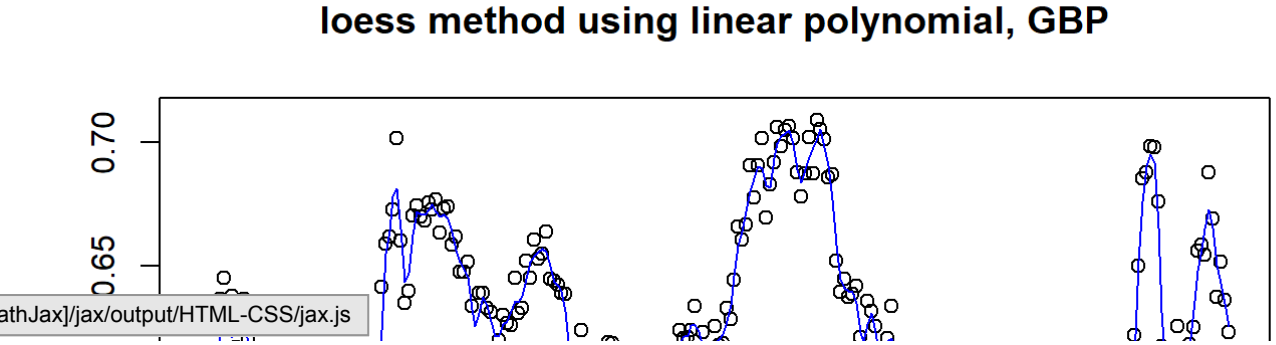
이때, 우리가 사용하는 데이터는 equispaced data이므로, local polynomial의 경우와 별 차이가 없을것이라는 사실을 유추할 수 있다.

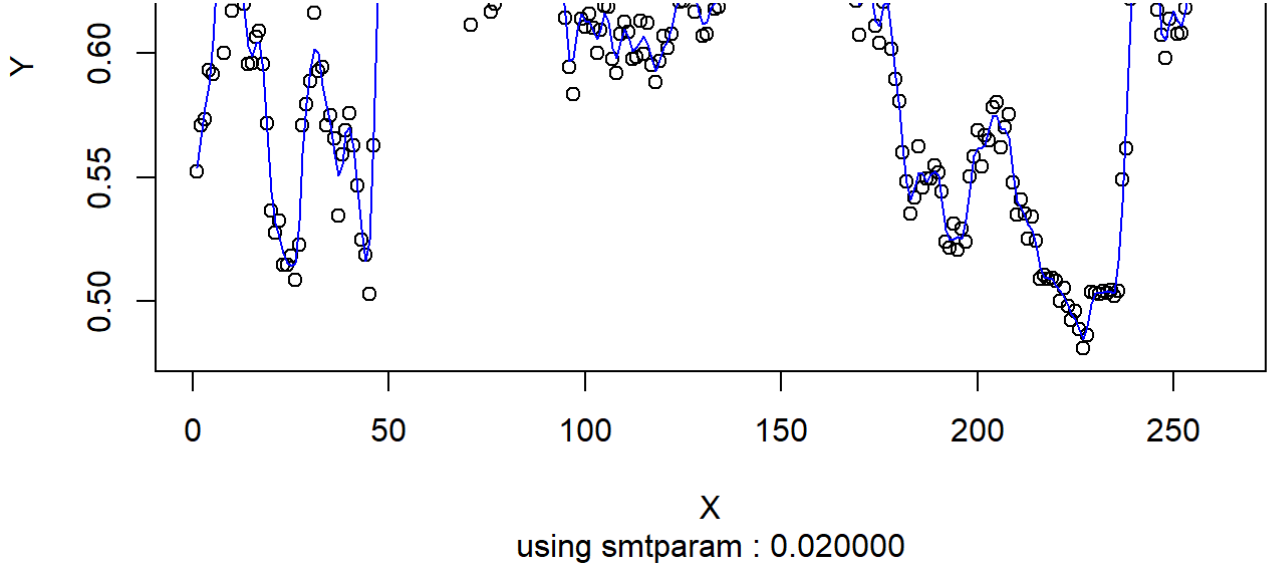
앞에서와 같이, 같은 방법으로 smoothing을 해보자. span이 너무 작으면 오류가 발생하므로, 0.02~0.3까지, 0.01간격으로 span을 이용할 것이다.

먼저, linear case의 경우이다.

```
full_test(estimate_loess_linear, data$X, data$GBP, seq(from = 0.02, to = 0.3, by = 0.01), description = "loess me
thod using linear polynomial, GBP")
```

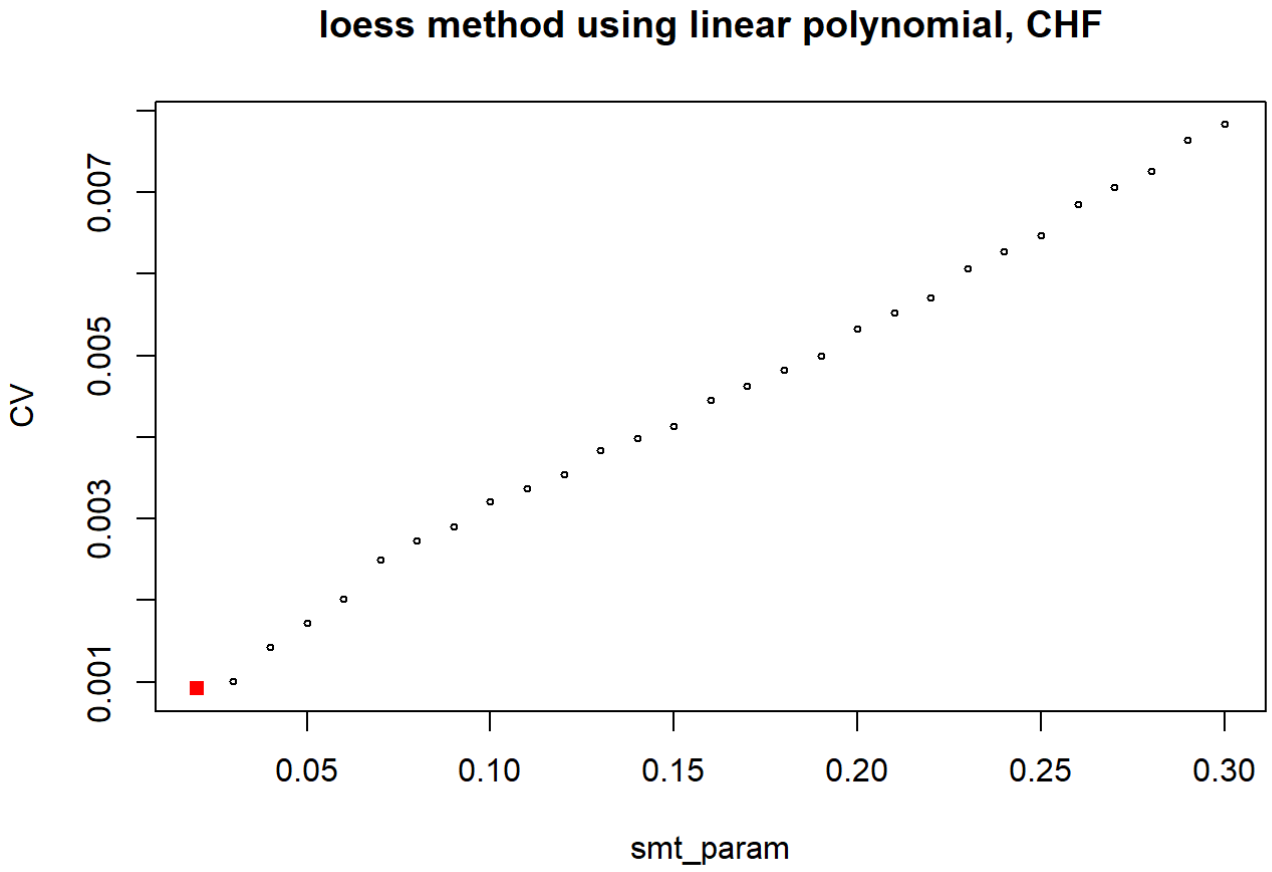


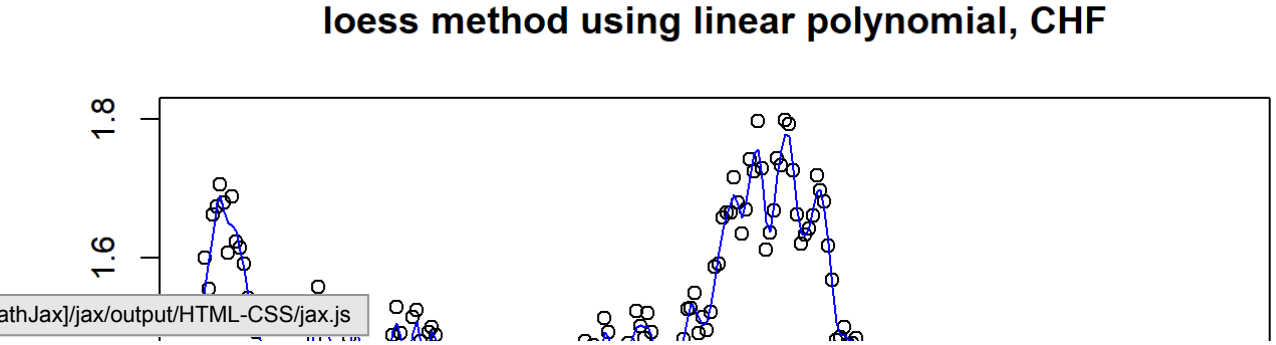


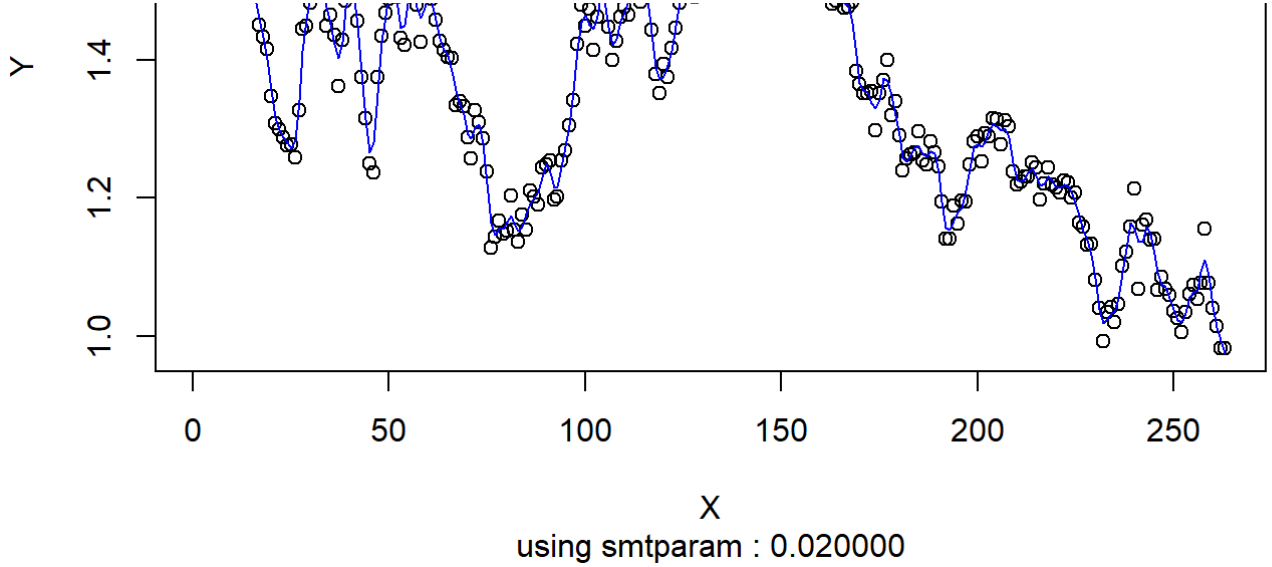



```
## [1] 4.29159e-05
```

```
full_test(estimate_loess_linear, data$X, data$CHF, seq(from = 0.02, to = 0.3, by = 0.01), description = "loess me
thod using linear polynomial, CHF")
```

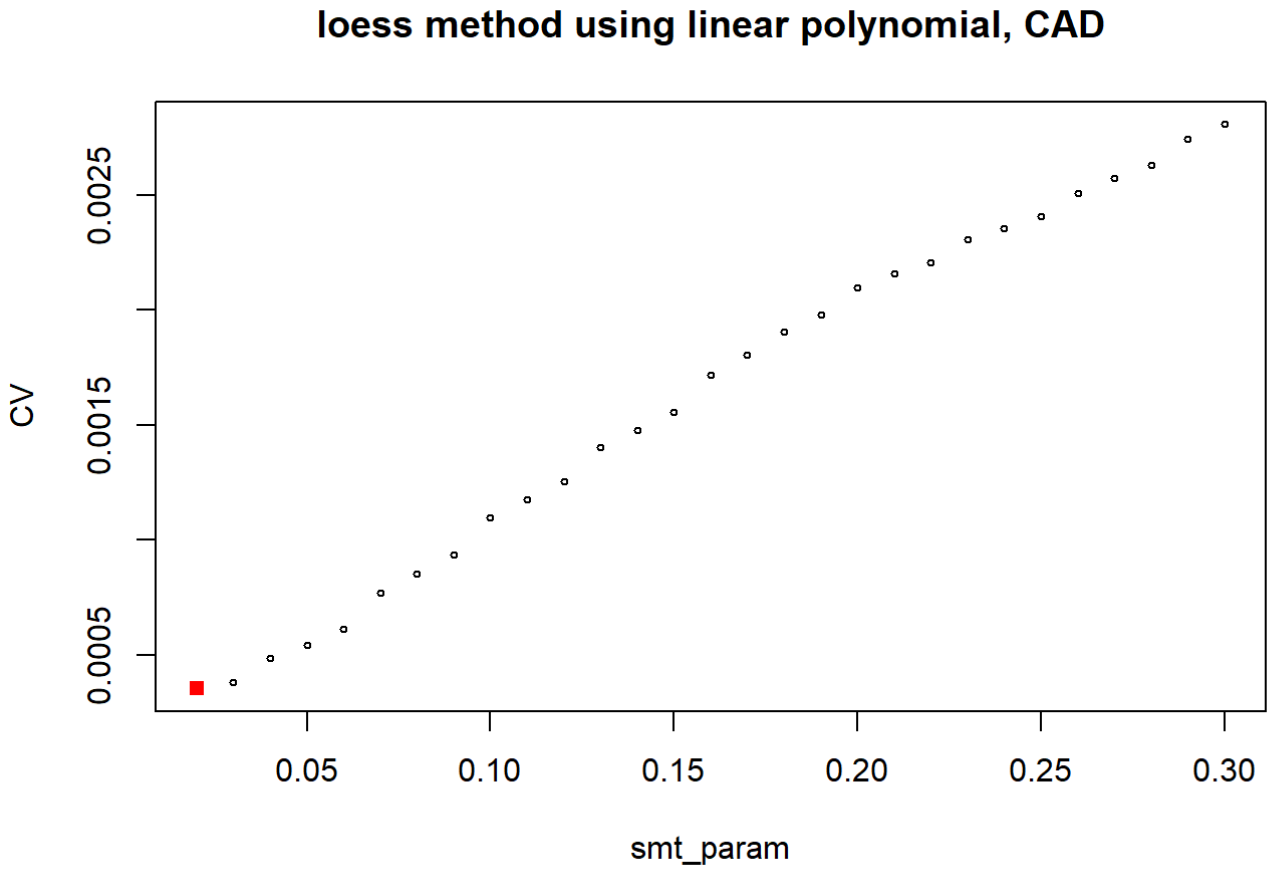


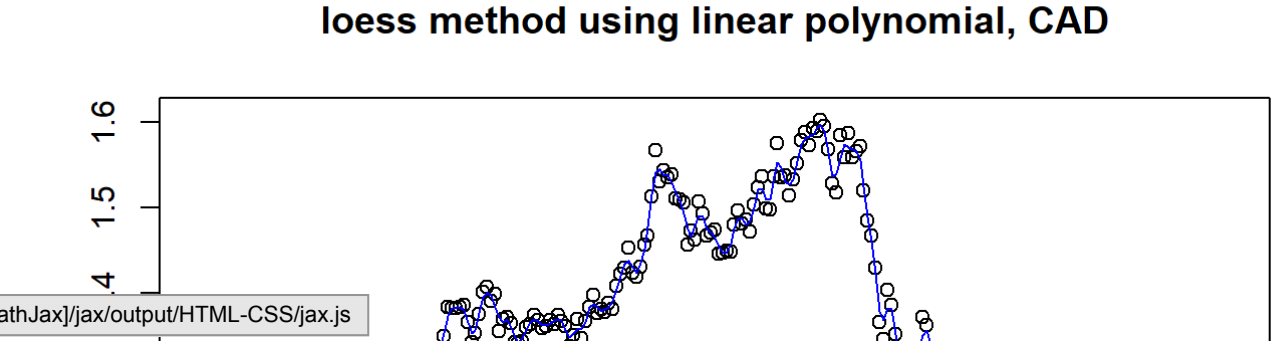


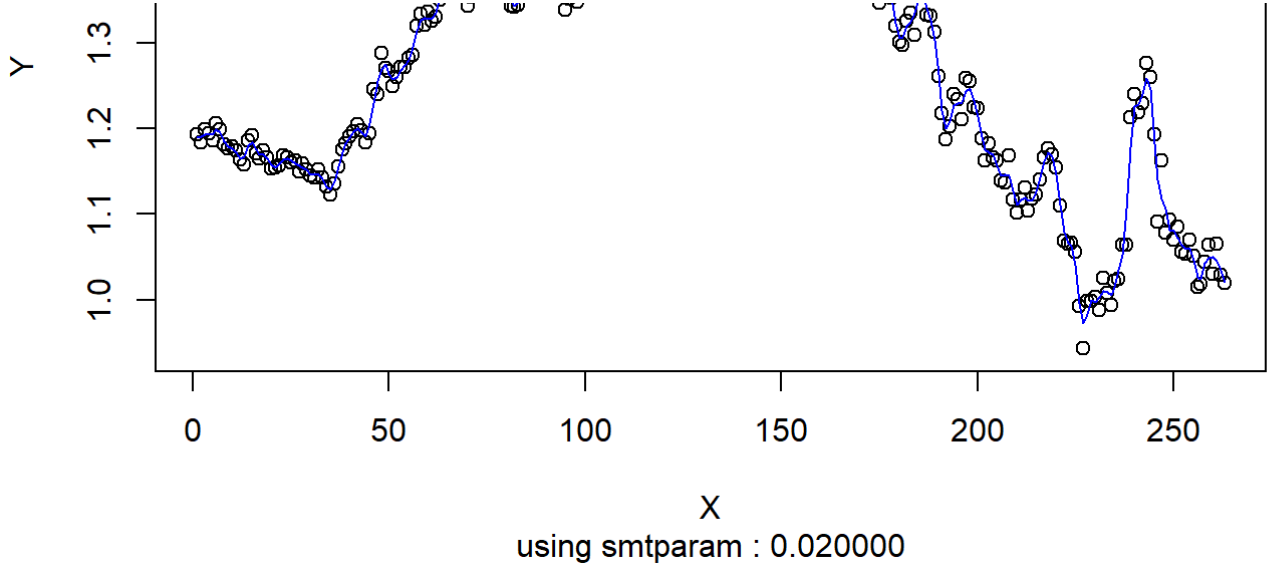



```
## [1] 0.0002952386
```

```
full_test(estimate_loess_linear, data$X, data$CAD, seq(from = 0.02, to = 0.3, by = 0.01), description = "loess me
thod using linear polynomial, CAD")
```



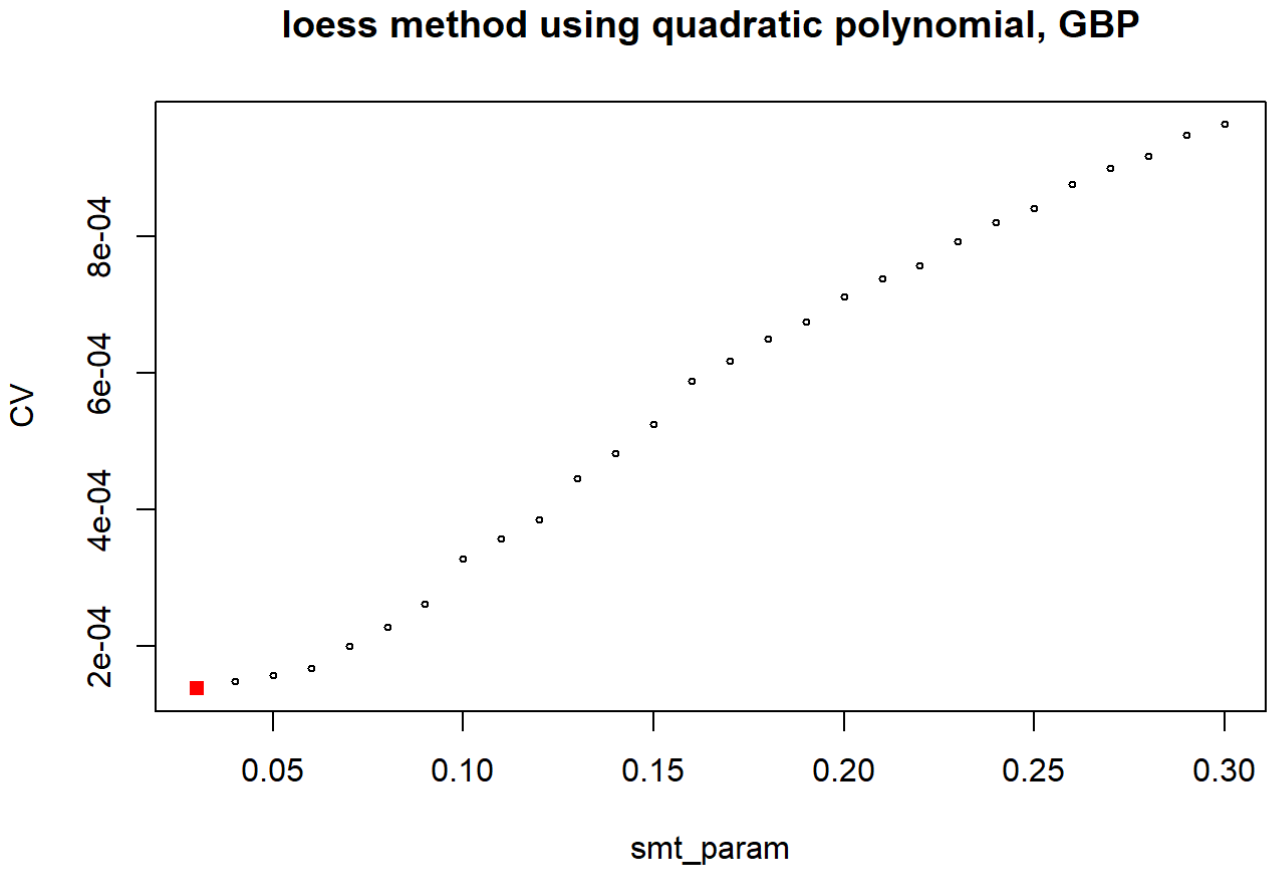


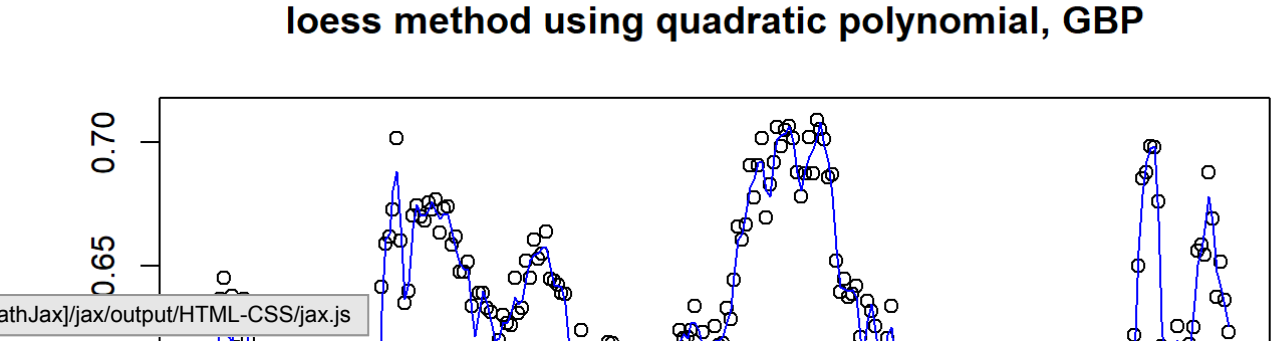


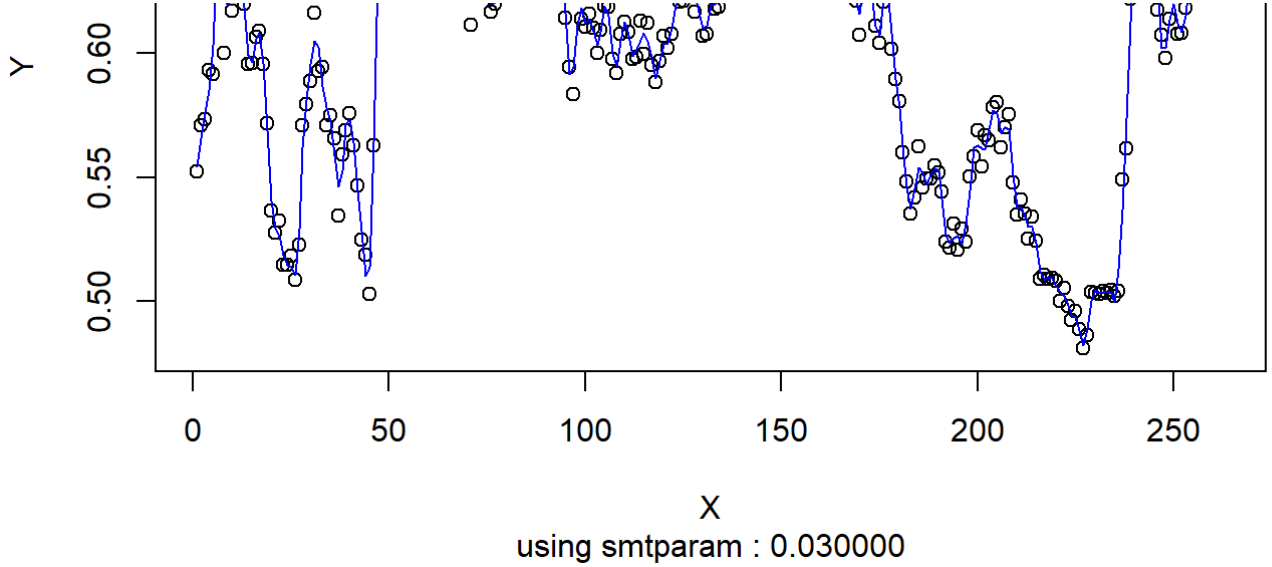

```
## [1] 0.0001160333
```

다음은, 2차 함수를 사용했을 경우이다. 역시, span이 너무 작으면 estimate를 하는데 사용할 데이터 포인트의 개수가 너무 작아지므로, 0.03~0.3까지, 0.01간격으로 span을 넣고 실험해보았다.

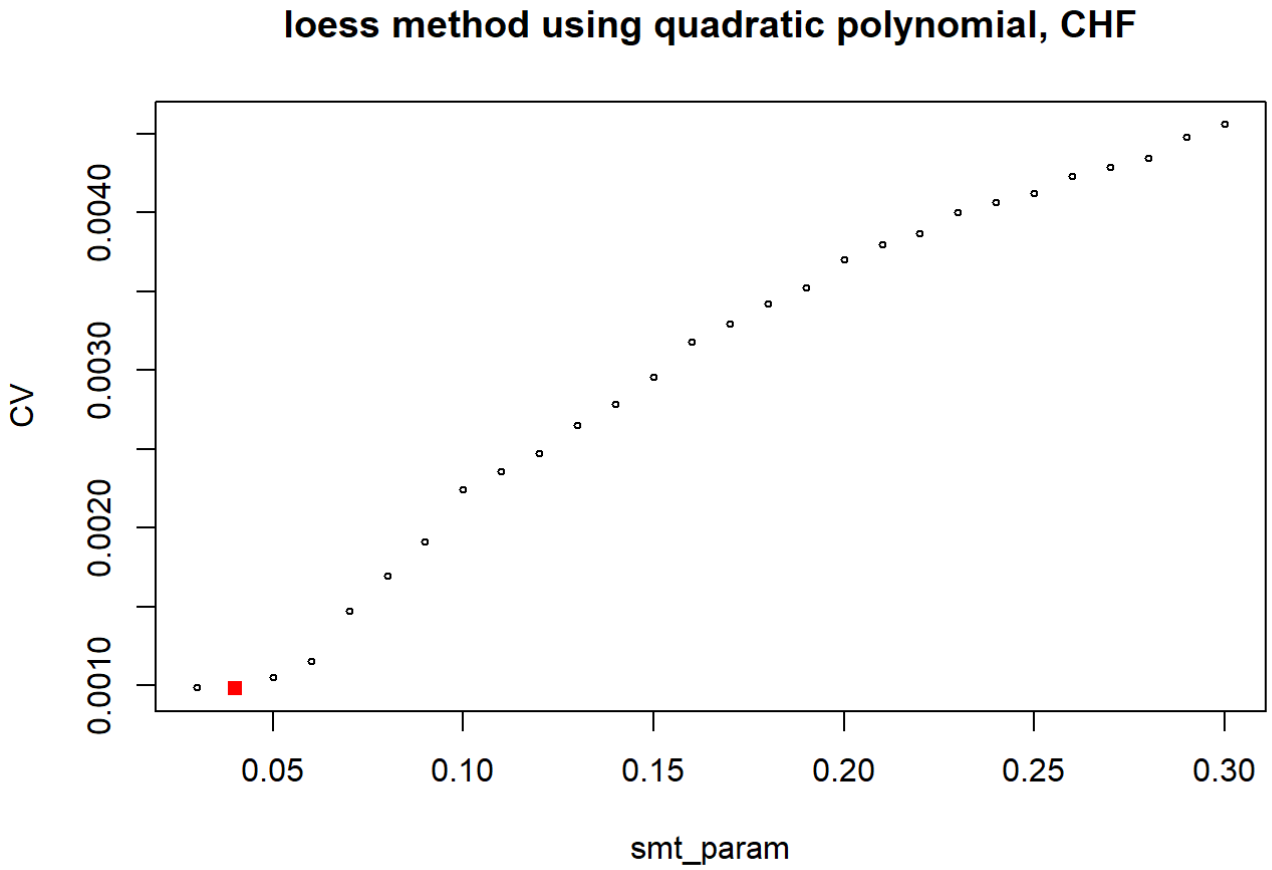
```
rss_GBP[5] <- full_test(estimate_loess_quad, data$X, data$GBP, seq(from = 0.03, to = 0.3, by = 0.01), description
 = "loess method using quadratic polynomial, GBP")
```

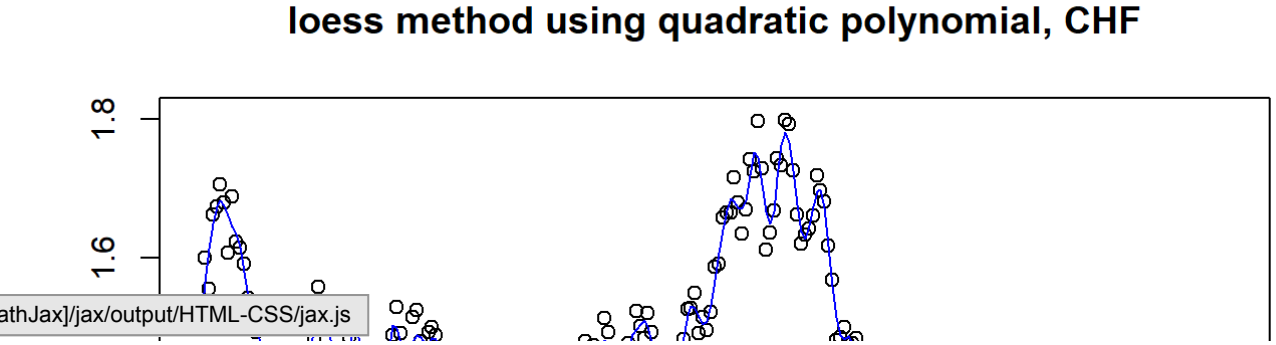


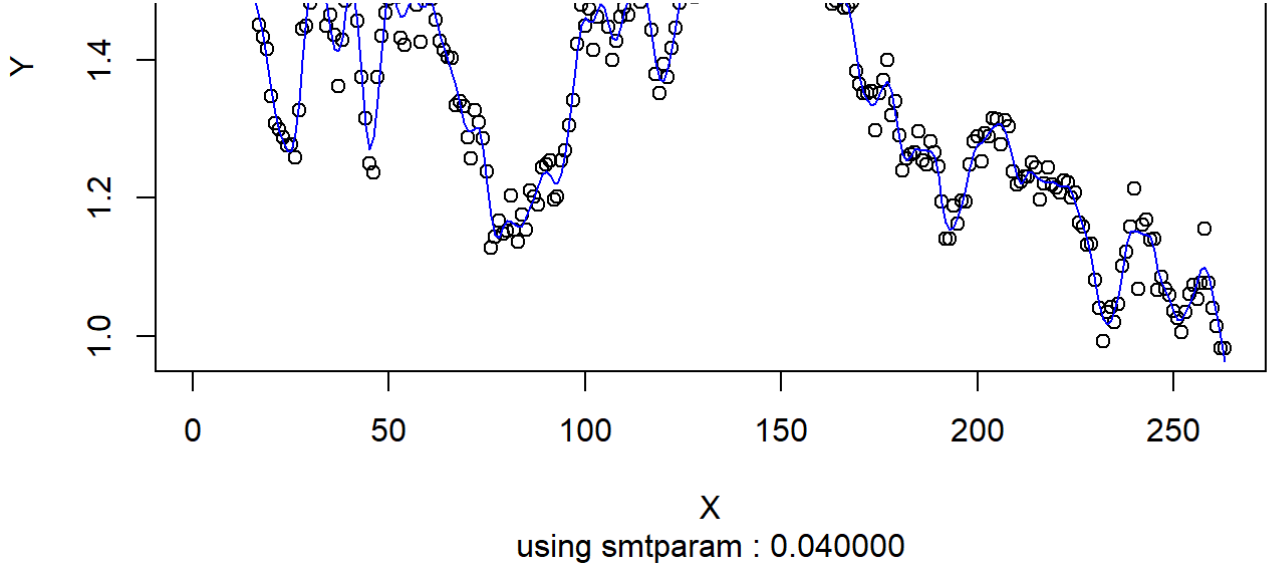




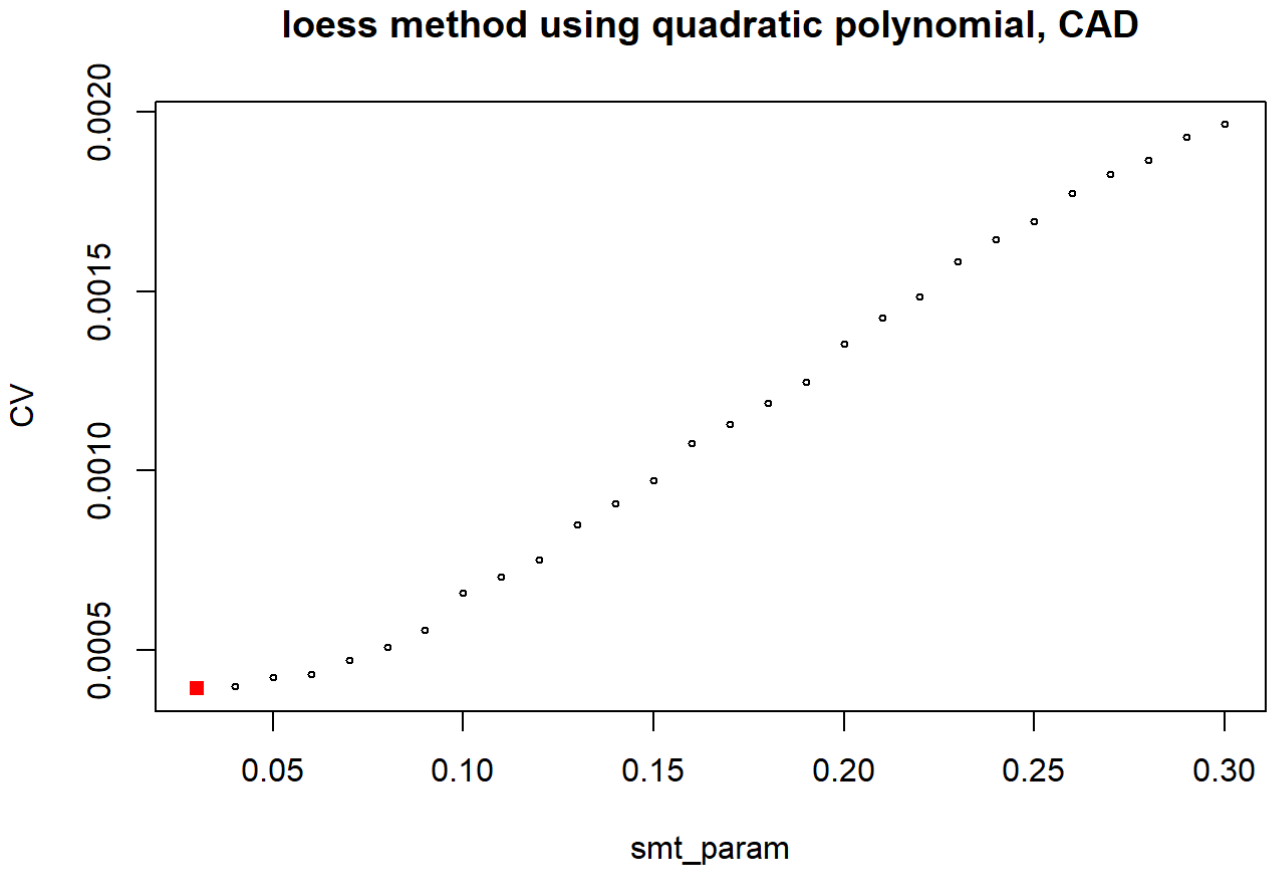

```
rss_CHF[5] <- full_test(estimate_loess_quad, data$X, data$CHF, seq(from = 0.03, to = 0.3, by = 0.01), description
 = "loess method using quadratic polynomial, CHF")
```

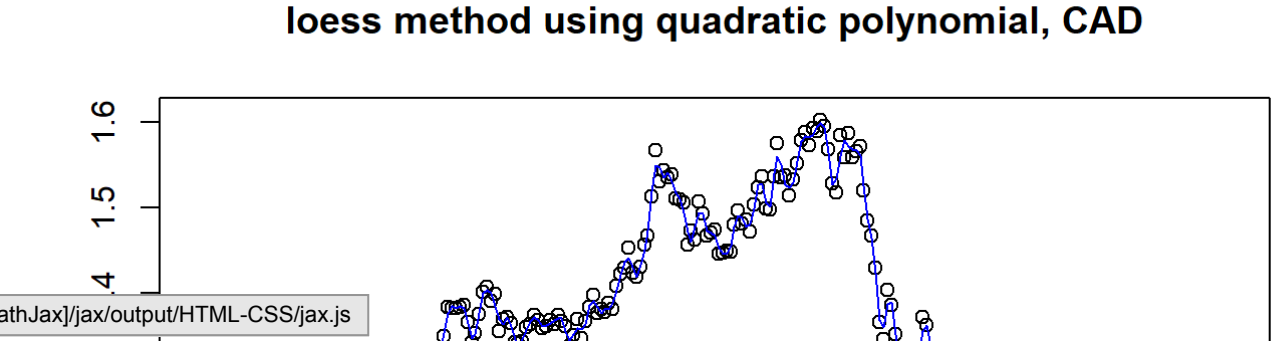


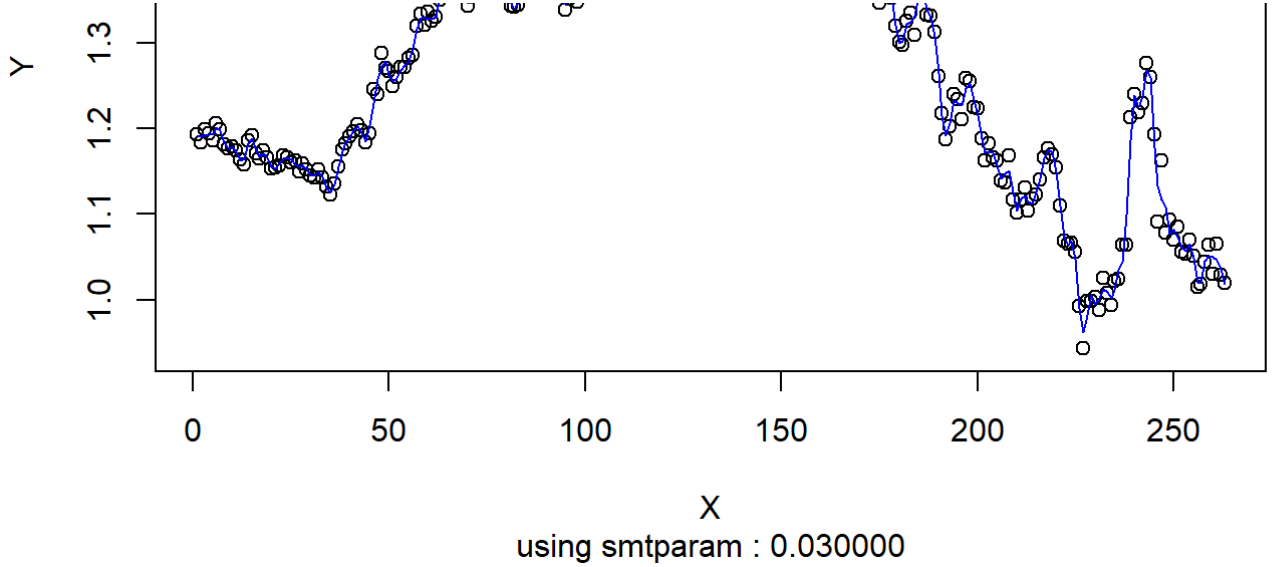





```
rss_CAD[5] <- full_test(estimate_loess_quad, data$X, data$CAD, seq(from = 0.03, to = 0.3, by = 0.01), description
 = "loess method using quadratic polynomial, CAD")
```







supersmoother

super smoother의 알고리즘을 생각해보면, k값으로 0.05n, 0.2n, 0.5n의 값을 사용하고, 1차 polynomial LOESS를 span = k/n의 값으로 각 k에 대해 진행한 다음,

$$r_{(i)}(k) = \frac{Y_i - \hat{m}_k(X_i)}{1 - h_{ii}}$$

(여기서 h_{ii}는 local linear regression의 hat matrix에서 얻어짐)

위 residual의 크기를 최소화하는 k를 이용하여 X_i에서의 값을 추정한다음,

그 추정 값을 $k^{(*)}(X_i)$라 할때,

이 추정값들을 이용해서 $k_2 = 0.2n$을 이용하여 다시 smoothing이 된 결과를 최종 추정값으로 내놓게 된다.

이러한 과정은 R의 supsmu 함수를 이용하여 가능하다.

supsmu함수는 span인자를 지정하지 않으면 위와 같은 알고리즘으로 smoothing을 하게 되고, span 인자를 지정해주면 span에 해당하는 single smoother만 사용해서 smoothing을 하게 된다.

 어진 datapoint에서만의 값을 리턴할 수 있고, 따라서 주어진 datapoint 바깥에서의 prediction은 불가능하다.

따라서, 지금까지와 달리 estim_x 가 NULL일때만 작동해야한다.

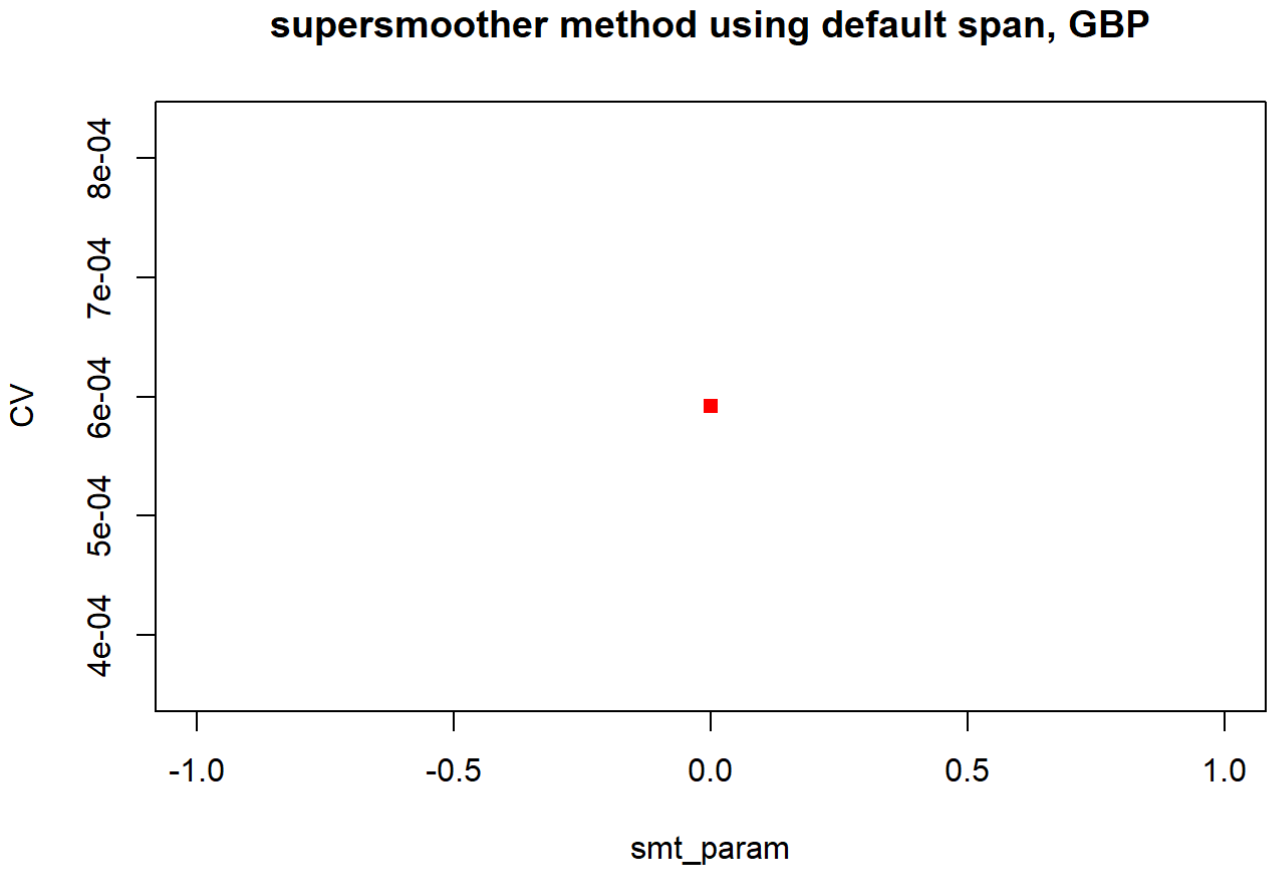
span인자에 따라 다음과 같은 두가지의 버전을 만들자.

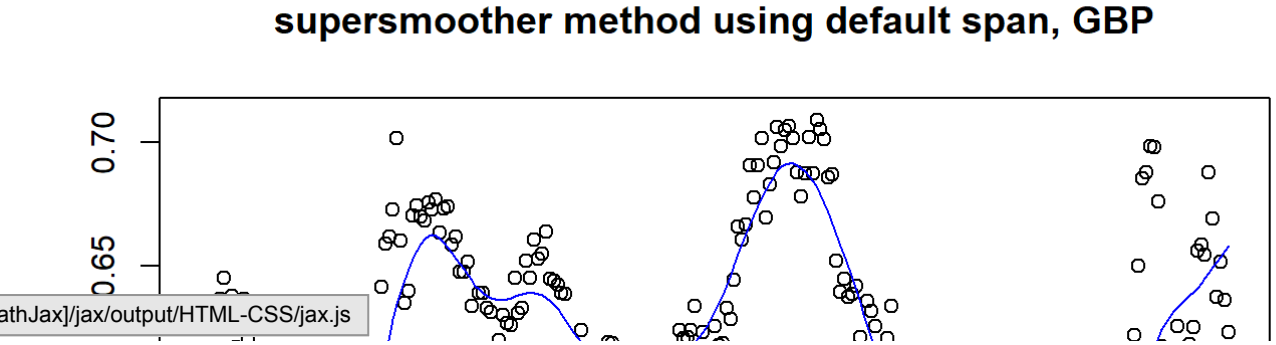
```
estimate_super_smoother_default <- function(xdata, ydata, sp = 0, estim_x = NULL)
{
    fit.su <- supsmu(xdata, ydata, span = "cv")

    if(is.null(estim_x))
    {
        ey <- fit.su$y
    }
    else
    {
        stop("super smoother can't predict!")
    }
    return(ey)
}


estimate_super_smoother <- function(xdata, ydata, sp, estim_x = NULL)
{
    fit.su <- supsmu(xdata, ydata, span = sp)

    if(is.null(estim_x))
    {
        ey <- fit.su$y
    }
    else
    {
        stop("super smoother can't predict!")
    }
    return(ey)
}
```

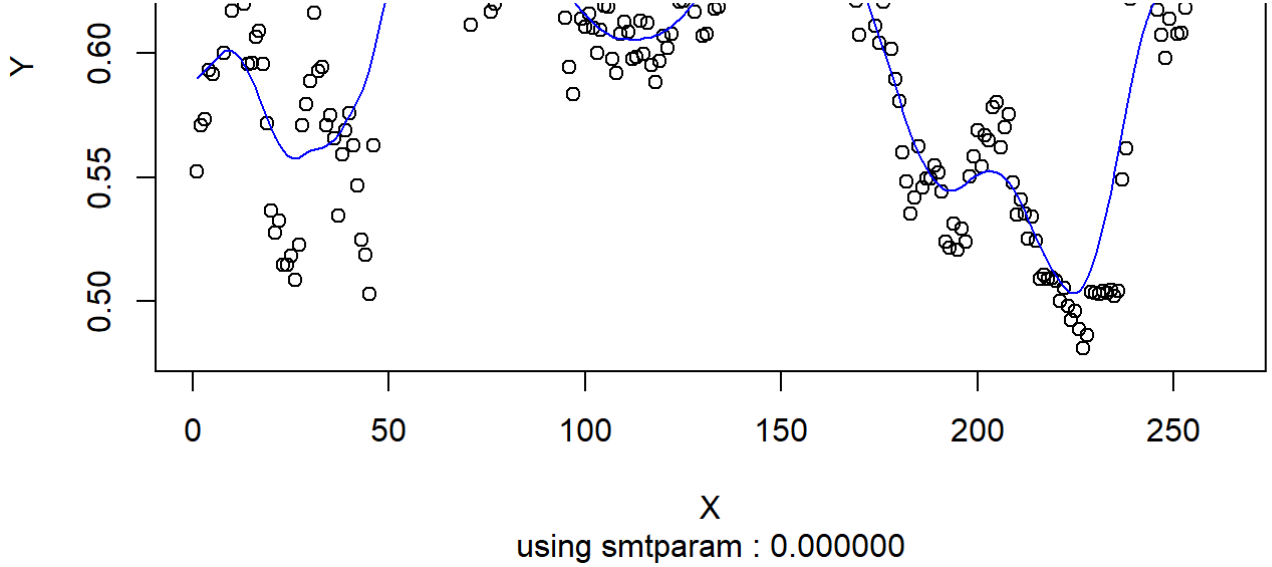
이제, 위 함수를 이용해서 test를 진행할 수 있다.

default값을 사용한 케이스와, 그렇지 않은 케이스에 대해 각각 test를 진행해보자.

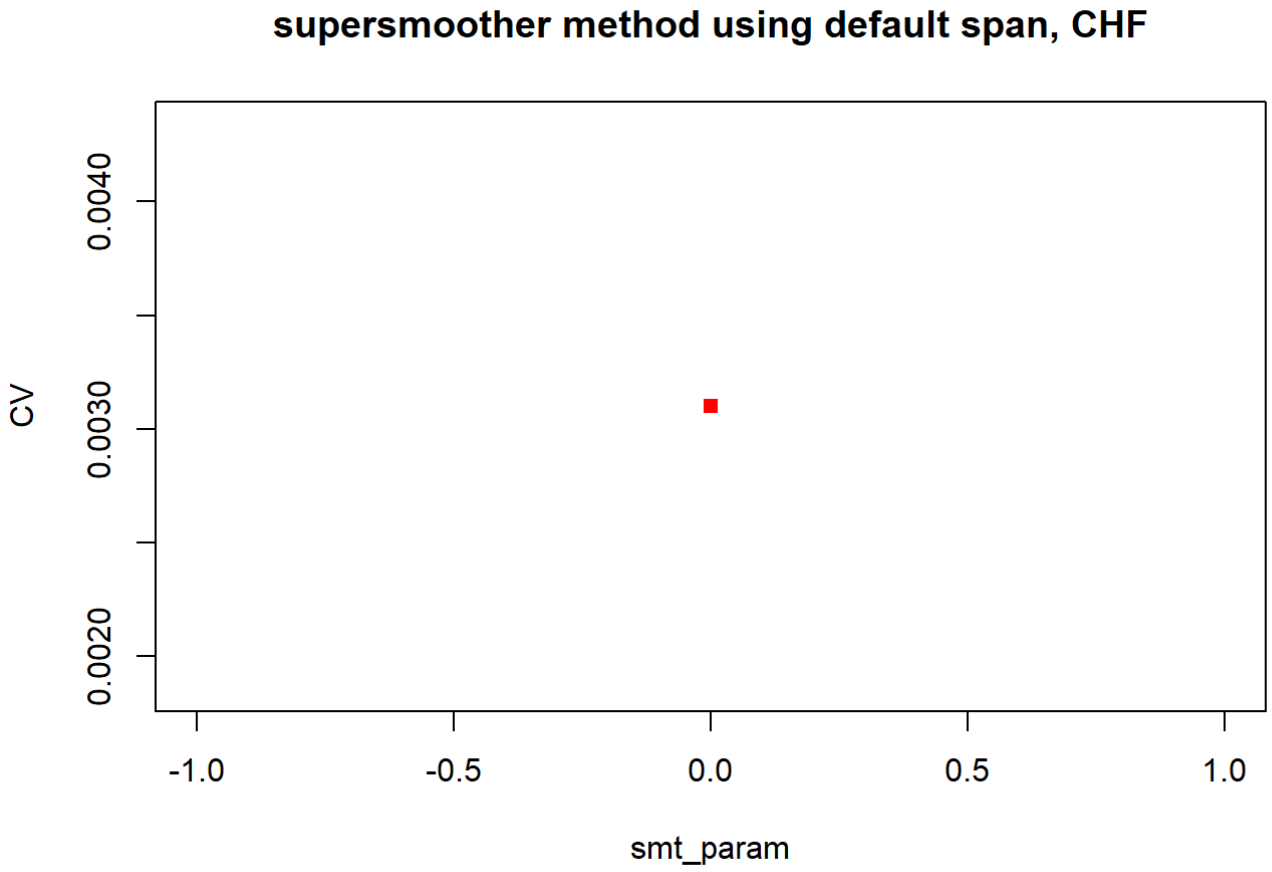
```
rss_GBP[6] <- full_test(estimate_super_smoother_default, data$X, data$GBP, c(0), description = "supersmoother met
hod using default span, GBP")
```

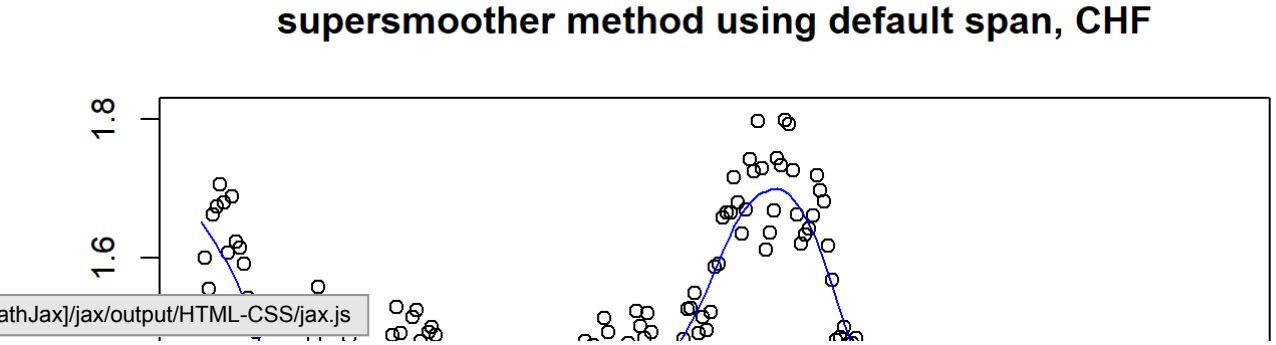


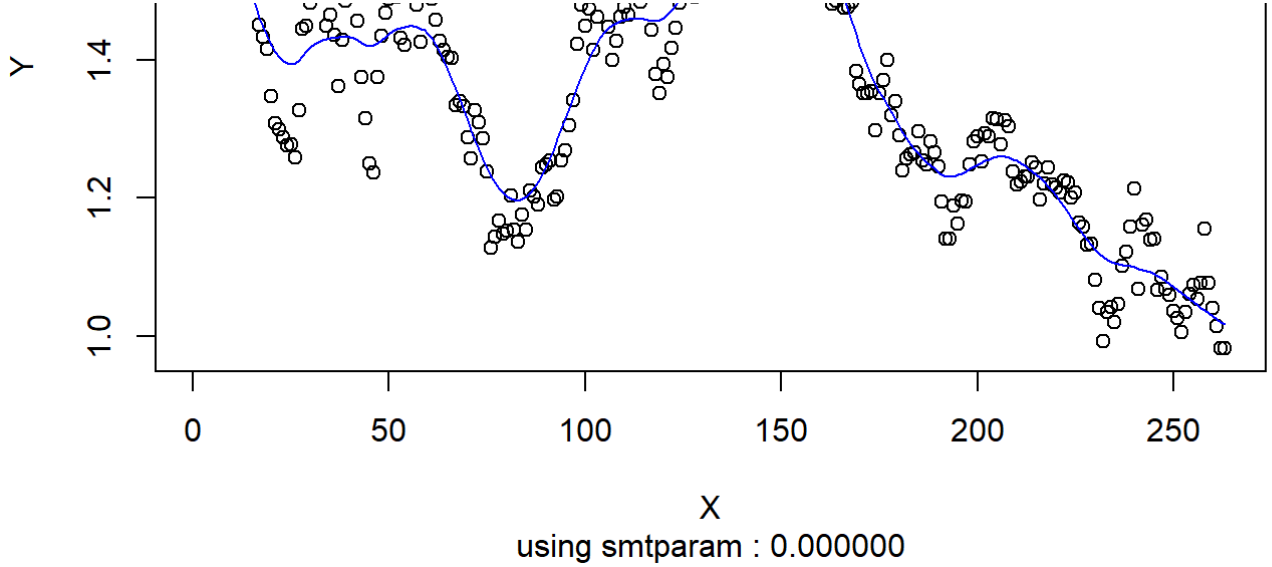




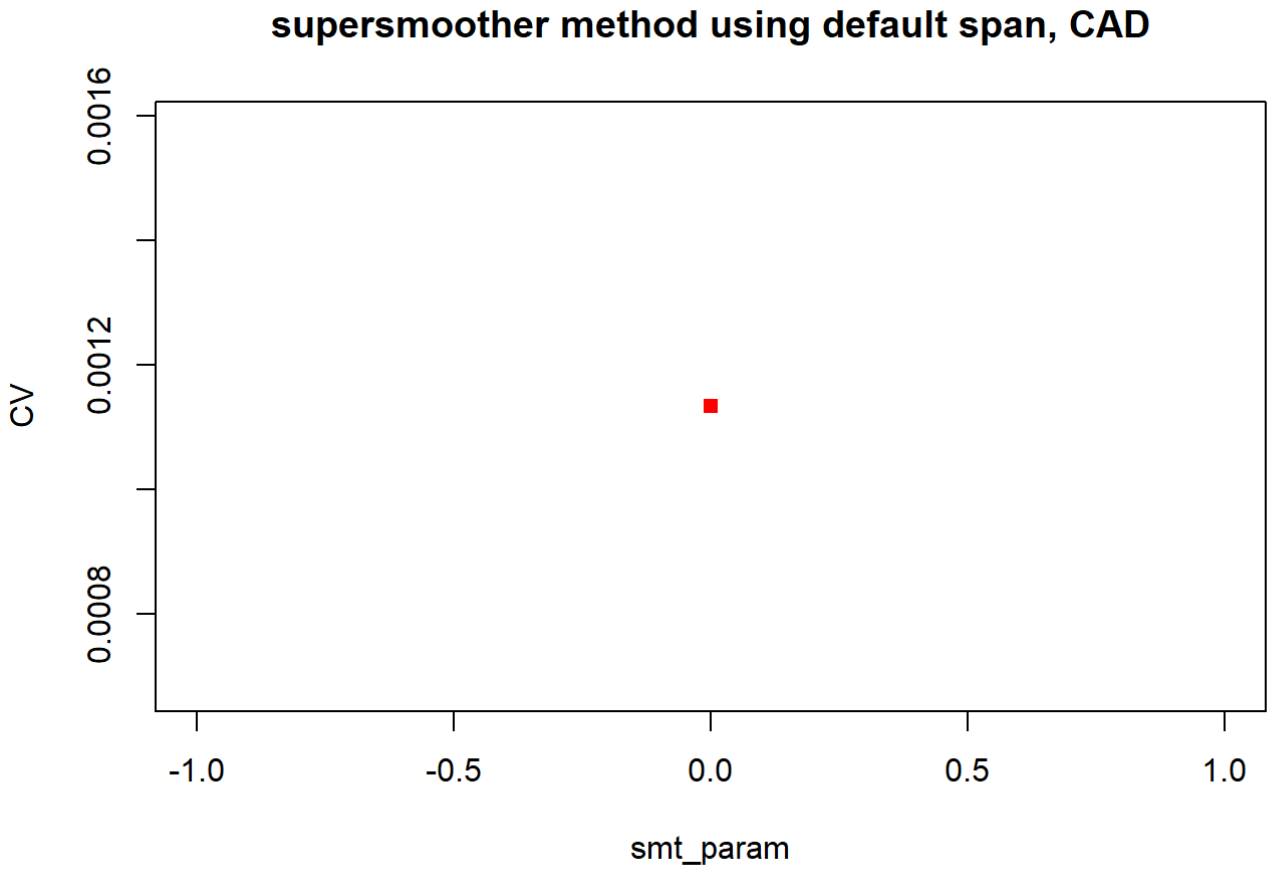

```
rss_CHF[6] <- full_test(estimate_super_smoother_default, data$X, data$CHF, c(0), description = "supersmoother met
hod using default span, CHF")
```

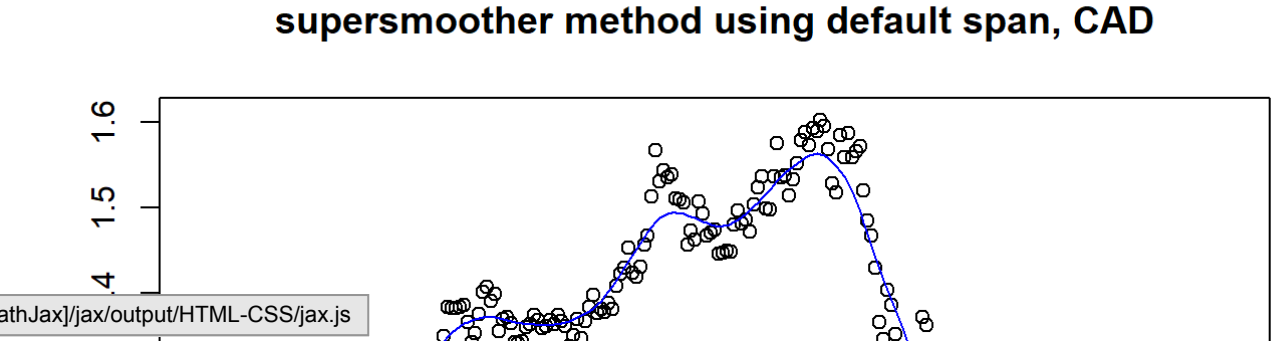


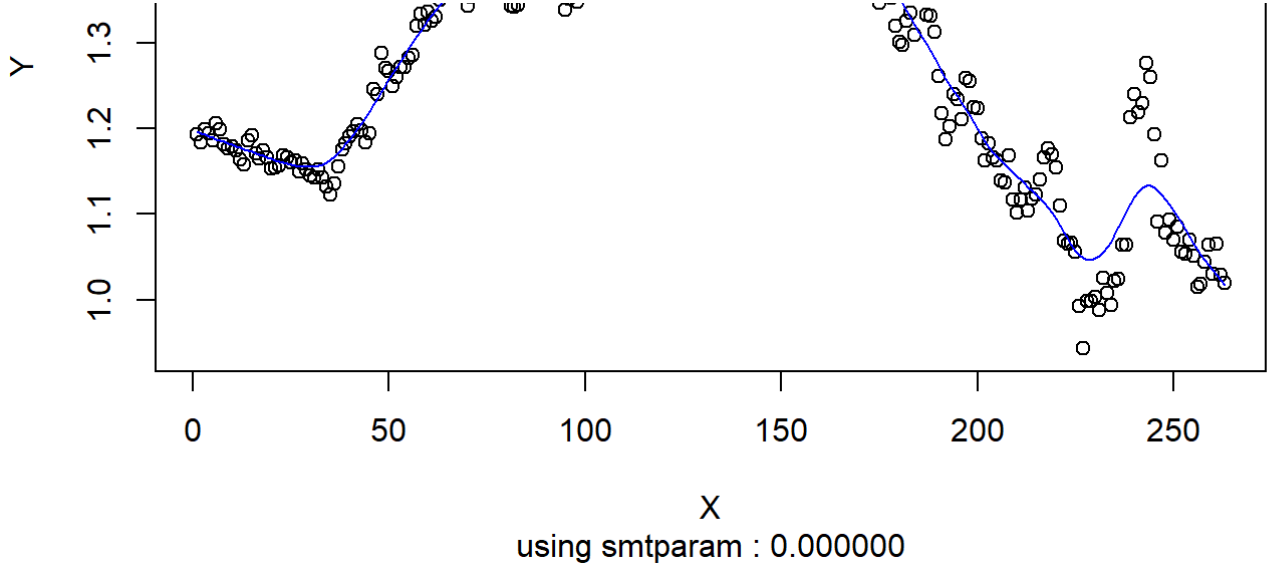





```
rss_CAD[6] <- full_test(estimate_super_smoother_default, data$X, data$CAD, c(0), description = "supersmoother met
hod using default span, CAD")
```



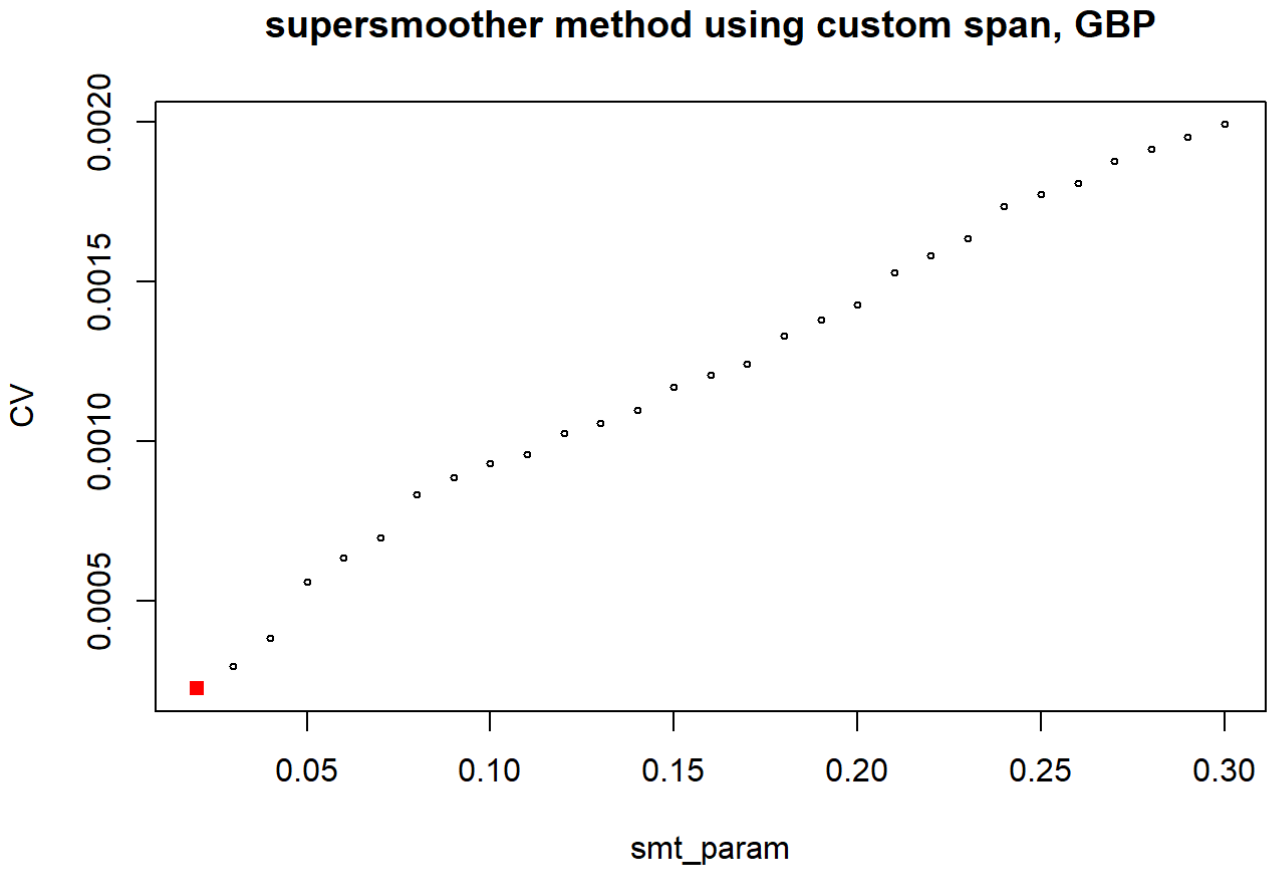


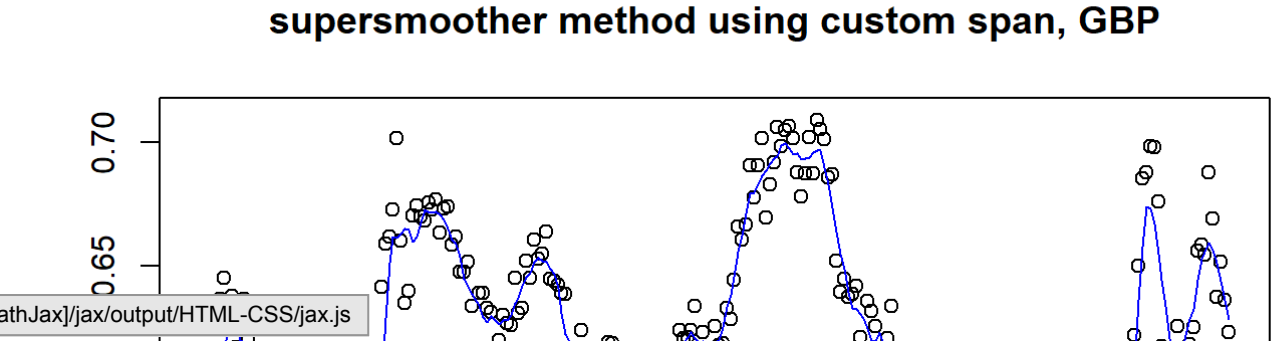


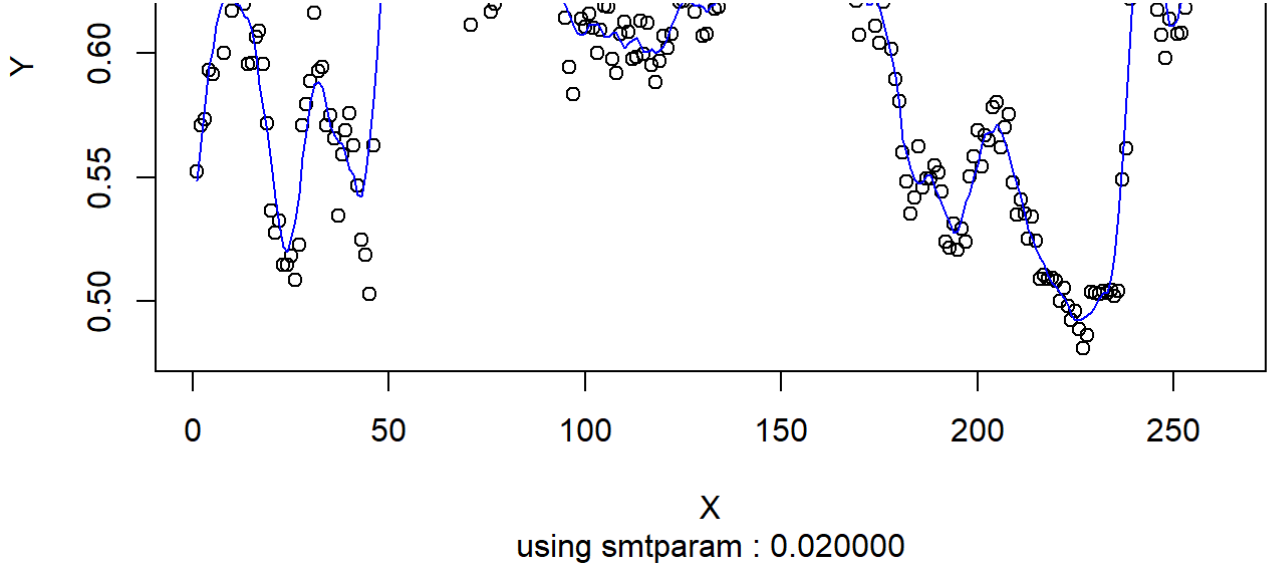
결과는 위와 같다.

이제, span의 값으로 0.02~0.3까지, 0.01간격을 두고 실행하여 test를 진행해보자.

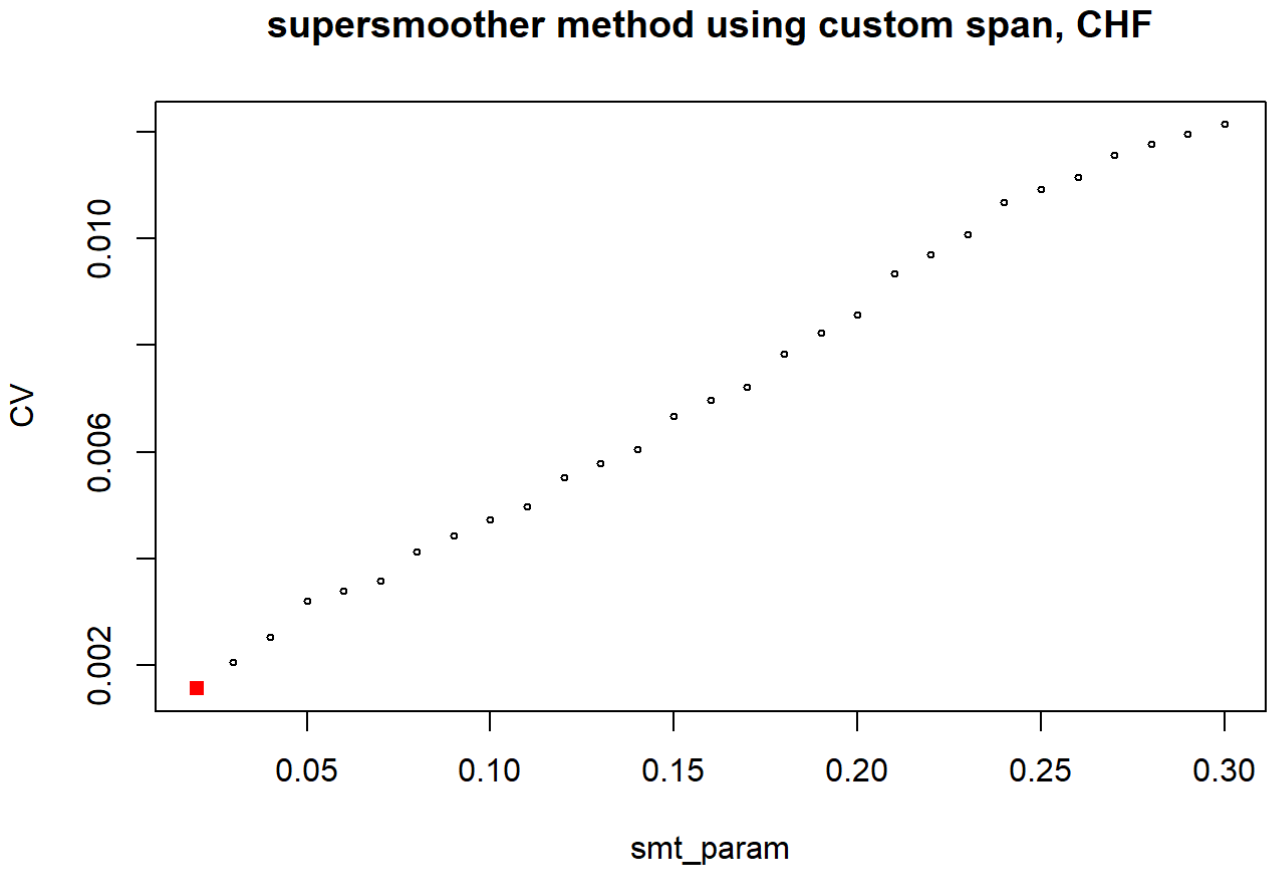
```
rss_GBP[7] <- full_test(estimate_super_smoother, data$X, data$GBP, seq(from = 0.02, to = 0.3, by = 0.01), descrip
tion = "supersmoother method using custom span, GBP")
```

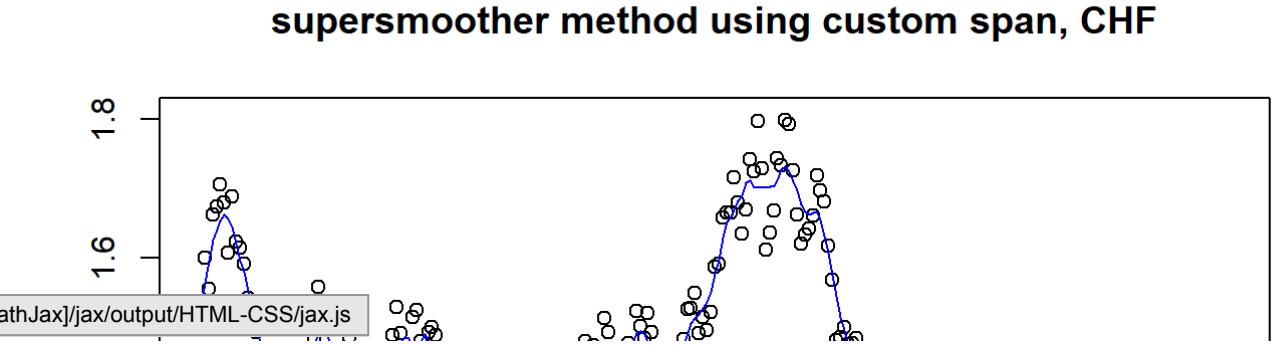


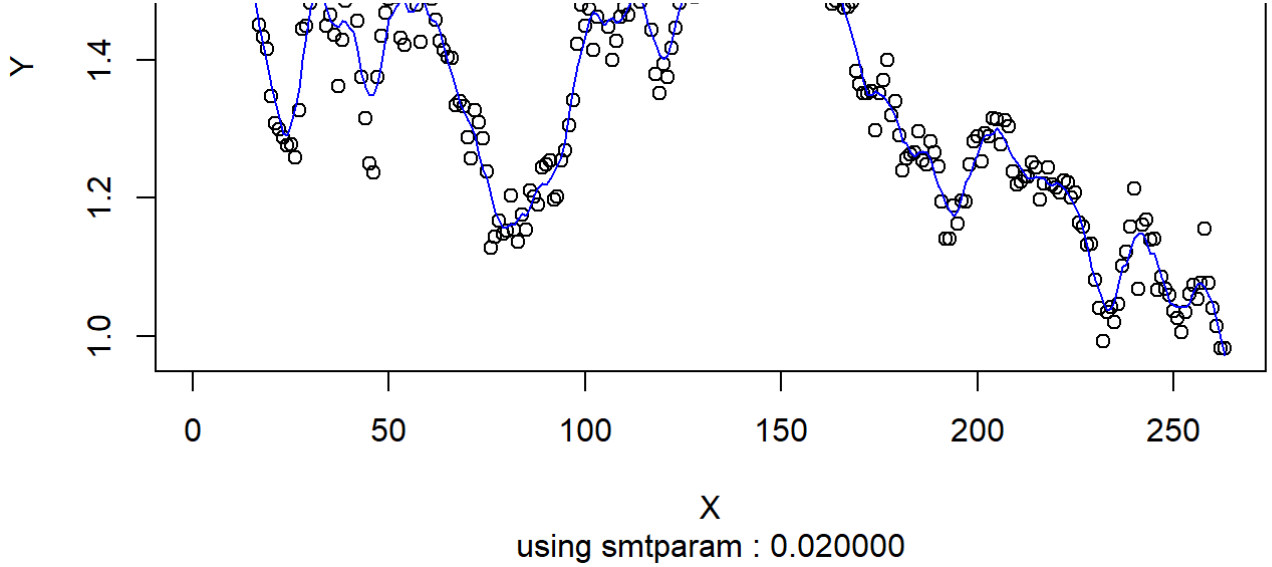




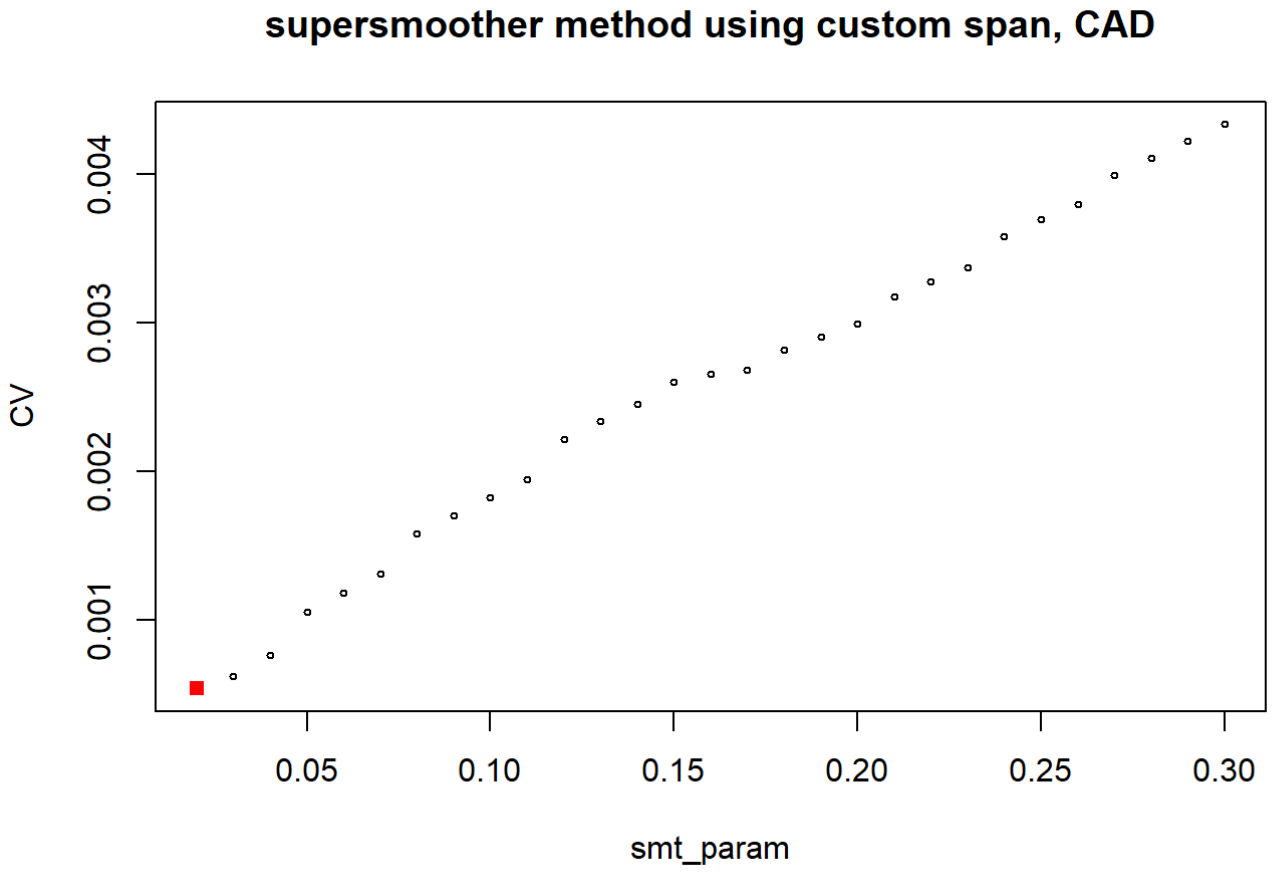

```
rss_CHF[7] <- full_test(estimate_super_smoother, data$X, data$CHF, seq(from = 0.02, to = 0.3, by = 0.01), descrip
tion = "supersmoother method using custom span, CHF")
```

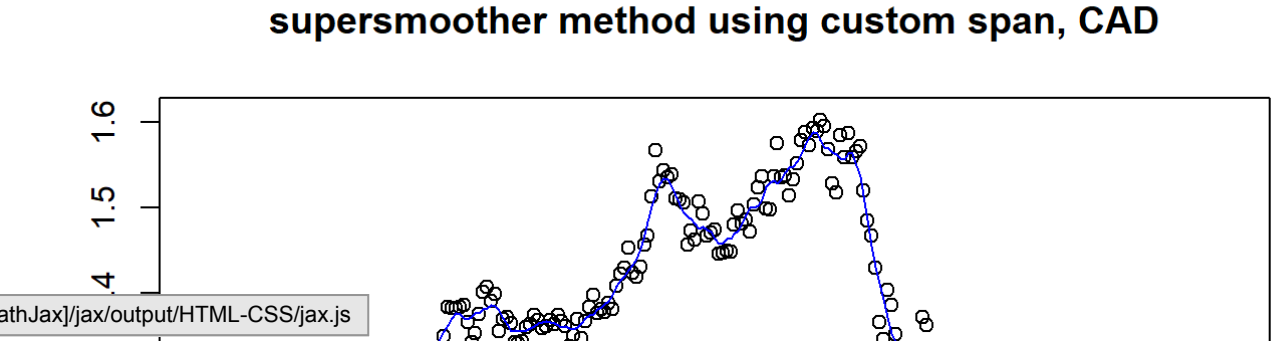


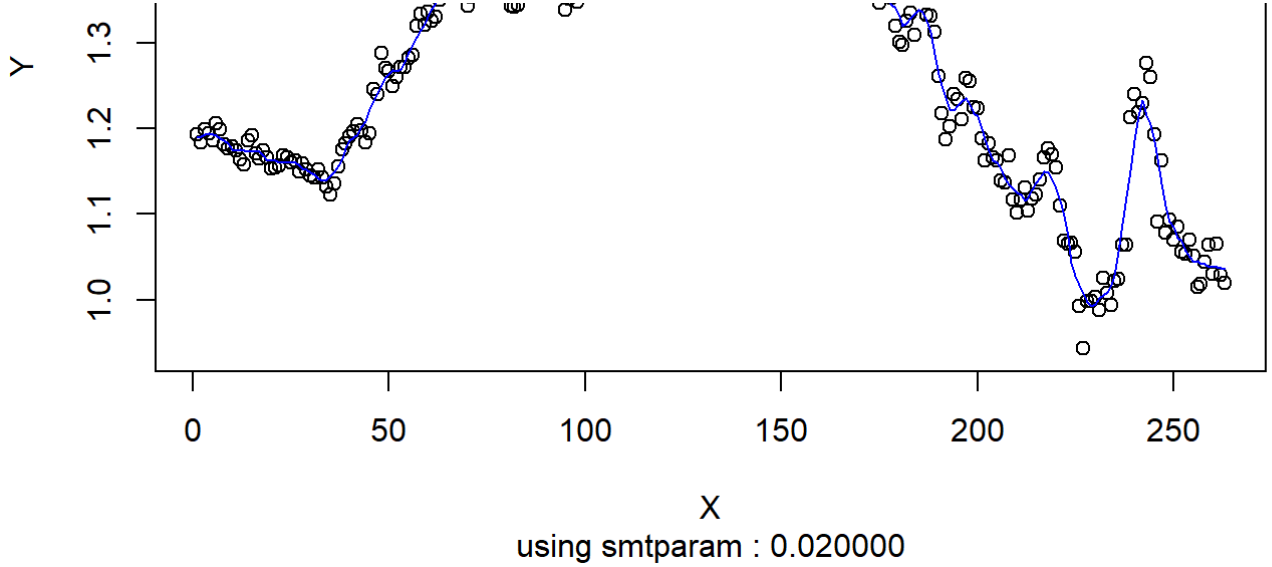





```
rss_CAD[7] <- full_test(estimate_super_smoother, data$X, data$CAD, seq(from = 0.02, to = 0.3, by = 0.01), descrip
tion = "supersmoother method using custom span, CAD")
```







결과는 위와 같다.

LOWESS

LOWESS는 LOESS의 개선된 버전으로, LOESS에서 outlier가 있을때 생기는 문제를 해결하기 위해 제안되었다.

즉, LOESS의 robust버전으로, 다음과 같은 알고리즘을 사용하여 smoothing을 진행한다.

1. LOESS를 일단 해서, $\hat{Y_i}$들을 구한다. 그리고, residual들을 계산한다. $r_i = Y_i - \hat{Y_i}$
2. robustness weight $\{\sigma_i = B(r_i/(6\hat{s})\}$를 구한다. 이때, $B(u)$는 bisquare weight function이고, $\hat{s} = med(\{r_i\})$가 된다.
3. LOESS에서, weight를 $\sigma_i w(\frac{X_i - x^*}{h_k(x^*)})$ 로 바꿔서 진행한다음, residual을 다시 구한다.
4. 2,3을 3번 더 반복한다.

R에서 LOWESS는 lowess 함수를 이용하면 가능하고,

이 함수 역시 prediction을 하는 기능은 제공하지 않으므로

supersmoother를 구현할때와 같이 estim_x를 사용하는 기능은 막아놓을것이다.

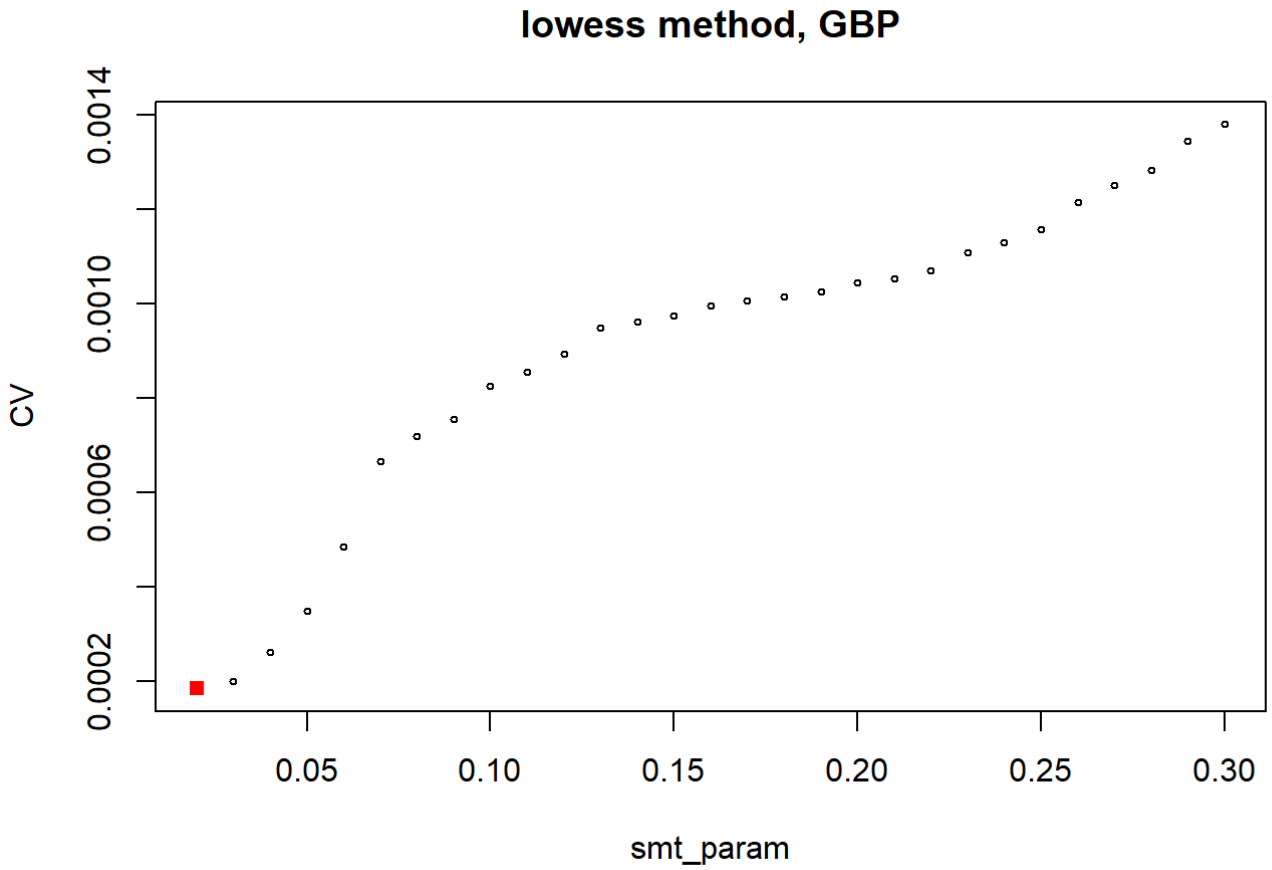
```
estimate_lowess <- function(xdata, ydata, sp, estim_x = NULL)
{
    fit.low <- lowess(xdata, ydata, f = sp)

    if(is.null(estim_x))
    {
        ey <- fit.low$y
    }
    else
    {
        stop("lowess can't predict!")
    }
    return(ey)
}
```

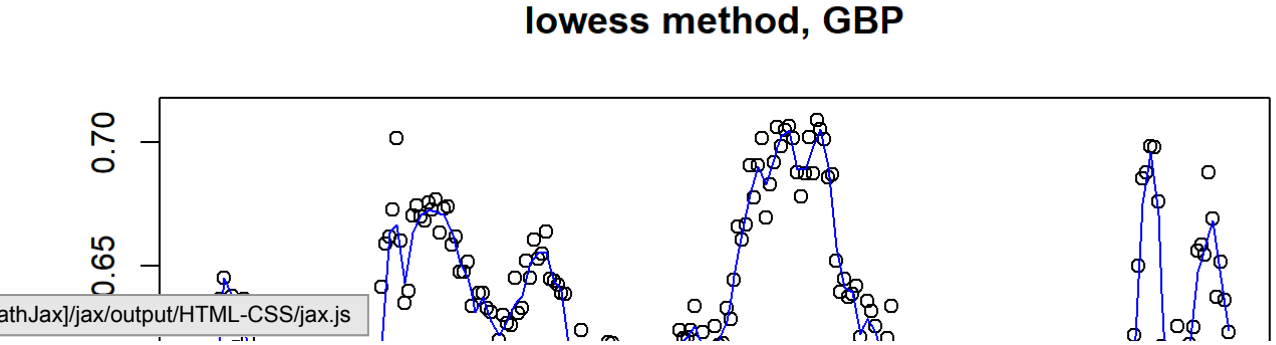
위와 같이 estimate_function을 구현한다.

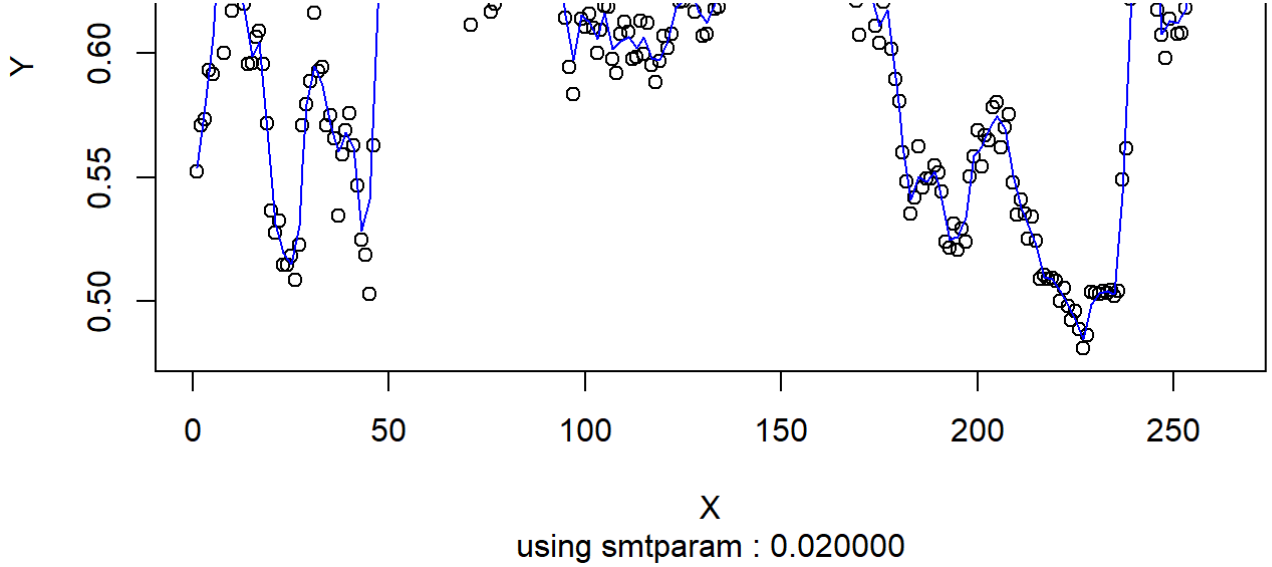
이제, 위 함수를 이용해서 test를 진행할 수 있다.

span의 값으로 0.02~0.3까지, 0.01간격을 두고 실행하여 test를 진행해보자.

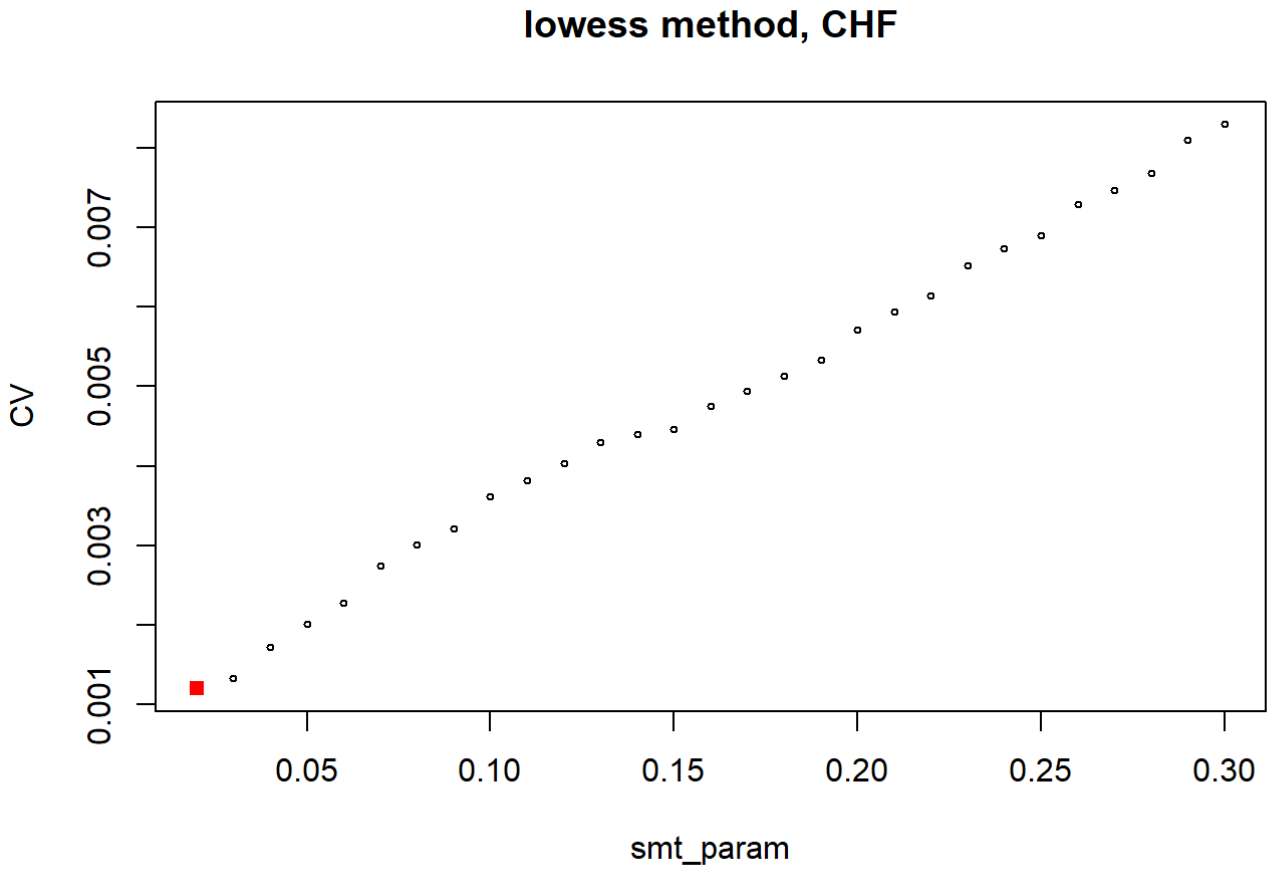
```
rss_GBP[8] <- full_test(estimate_lowess, data$X, data$GBP, seq(from = 0.02, to = 0.3, by = 0.01), description =
"lowess method, GBP")
```

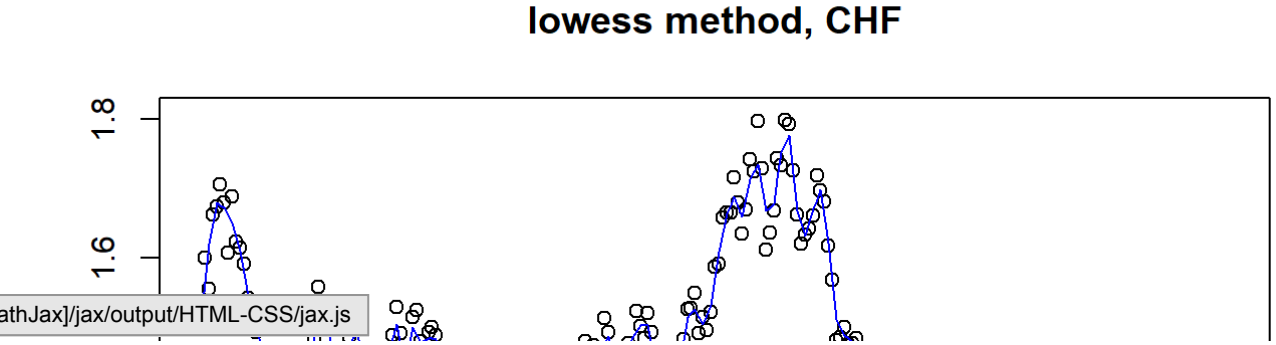


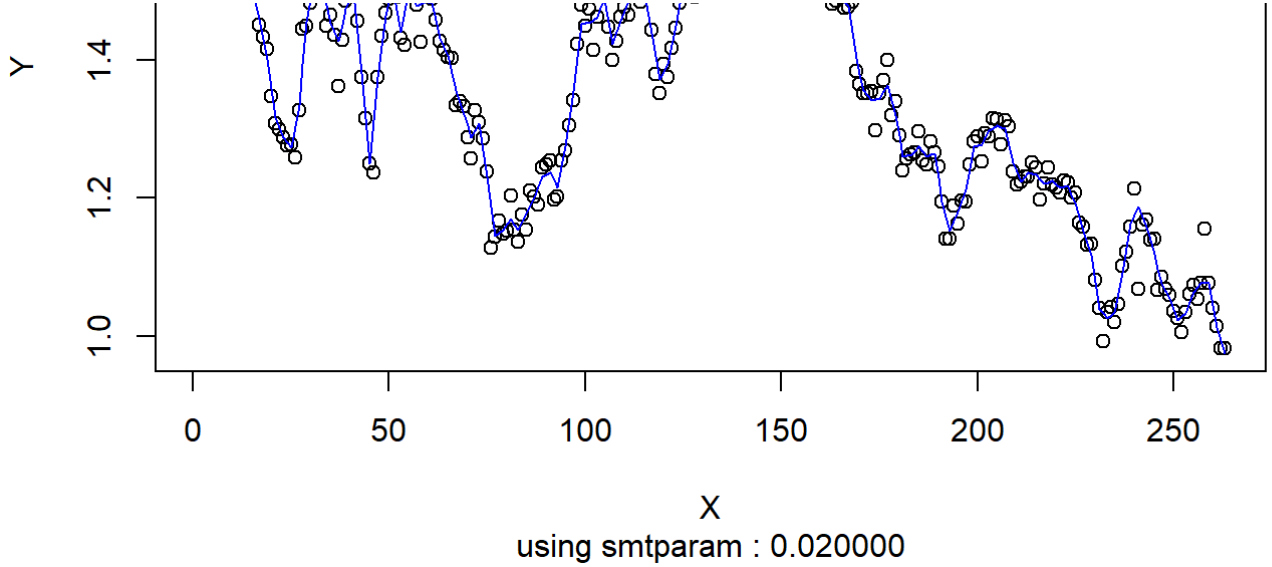




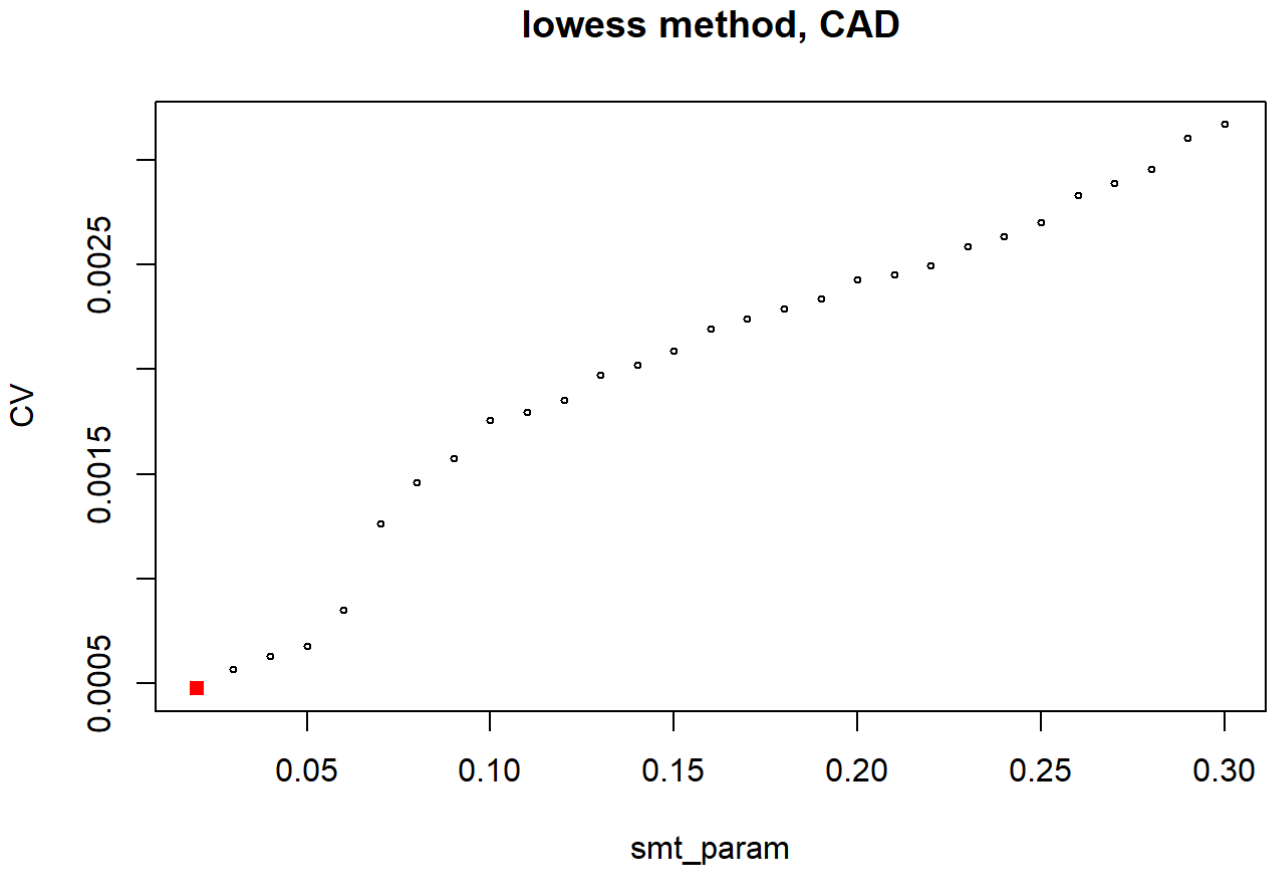

```
rss_CHF[8] <- full_test(estimate_lowess, data$X, data$CHF, seq(from = 0.02, to = 0.3, by = 0.01), description =
"lowess method, CHF")
```

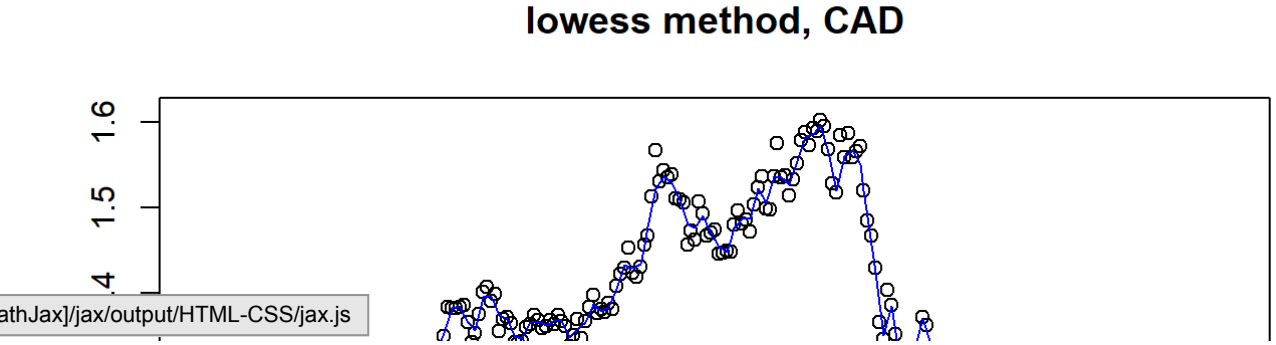


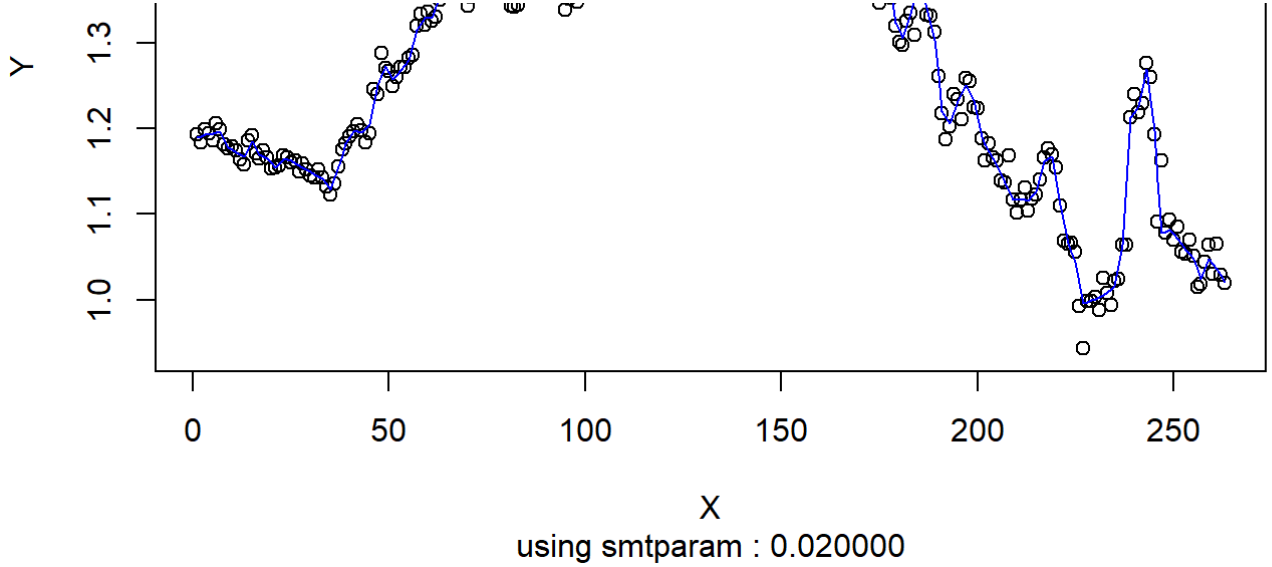





```
rss_CAD[8] <- full_test(estimate_lowess, data$X, data$CAD, seq(from = 0.02, to = 0.3, by = 0.01), description =
"lowess method, CAD")
```







이제, 가장 좋은 method를 찾기위해 앞에서 만든 rss 벡터들의 값중 어떤 값이 가장 작은지를 볼것이다.

1 : nadaraya-watson 2 : local polynomial (linear) 3 : natural spline with binomial filter 4 : smooting spline 5 : loess (linear) 6 : supersmoother using default span 7 : supersmoother using custom span 8 : lowess

```
which(rss_GBP == min(rss_GBP, na.rm = TRUE))
```

```
## [1] 1
```

```
which(rss_CHF == min(rss_CHF, na.rm = TRUE))
```

```
## [1] 2
```

```
which(rss_CAD == min(rss_CAD, na.rm = TRUE))
```

```
## [1] 1
```

각각 GBP, CHF, CAD에 대해 어떤것이 best method인지를 위와 같이 알아볼 수 있었다.

Q2

지금까지의 경향을 살펴보면, 대부분의 경우 smoothing parameter의 값을 가능한 한 작게 유지했을때 cross validation값이 작게 유지됨을 알 수 있다. 이는, cross validation의 경우 전체 dataset $D = (X_i, Y_i)_{i=1,2...n}$에서 하나의 데이터만 제외한 데이터를 이용해서 $(D^{-k} = (X_i, Y_i)_{i \neq k})$, 제외한 데이터의 자리에 대한 값을 추정했을때의 값이 최대한 원래 데이터의 값과 비슷할때에 앞에서 구한 cross validation의 값이 작아지게 된다.

$$CV[\hat{m}(x)] = \frac{\sum_{k=1}^{n}\left(Y_k - \hat{m}\left(X_k\right)\right)^2}{n(1 - H_{ii})^2}$$

하지만, 우리의 데이터는 시계열 데이터이다. 즉, 시간에 따라 연속적으로 변하는 데이터이고, 우리의 데이터는 더군다나 환율이기 때문에 (X_i, Y_i)는 이 값 주변의 값들, 즉, 시간대가 비슷한 데이터들과 큰 연관이 있을수밖에 없다.

(예를 들어, 전날 환율이 1달러에 1000원일때와 1달러에 2000원이었을때, 다음날 환율이 1달러에 1050원일 확률은 당연히 전자가 높을수밖에 없다. 즉, 시간적으로 인접한 데이터에 영향을 받으므로, 독립 가정이 불가능하다.)

따라서 앞에서처럼 leave-one-out cross validation을 한다고 하더라도, data point의 개수가 대략 250개정도나 되고, 제외된 데이터의 주변 데이터는 제외된 데이터와 비슷한 값을 가지고 있을 것이므로, smoothing parameter를 매우 작게해도 비슷한 값을 추정할 수 있을것이고, 따라서 leave-one-out cross validation을 통해 best smoothing parameter를 구하는 방법은 그렇게 좋은 방법이라고 할 수 없다.

왜냐하면 smoothing parameter를 더욱 작게 가져갈수록, Y_i가 $\hat{m}(X_i)$에 비슷해질것이고, 이에 따라 대부분의 경우 CV의 값이 계속해서 작아질 것이므로, 대부분의 경우 overfit이 발생하고, 따라서 새로운 데이터를 predict하는데는 별로 도움이 되지 않을것이기 때문이다.

따라서 다른 기준을 이용해서 그래프의 prediction power를 측정할 필요가 있다.

결국, 제외한 데이터의 인접한 데이터를 사용하기 때문에 overfit 현상이 발생하는것이 문제가 되므로, 데이터를 하나만 제외하고 cross validation을 구할것이 아니라,

충분히 많은 데이터를 제외하면서 추정을 해보고, 제외된 datapoint에서 추정된 값과, 실제 데이터 값을 이용하여 cross validation 값을 구할것이다.

즉, 전체 데이터를 k개의 그룹으로 나눈 뒤, cross validation을 구하는,

k-fold cross validation을 사용할 것이다.

앞의 leave-one out cross validation은 전체의 그룹을 n개로 나눈, n-fold cross validation이라 할 수 있으므로,

이를 일반화해서 다음과 같이 k-fold cross validation을 구하자. 이 값을 $CVERR_{(k)}$로 정의한다.

$$CVERR_{(k)} = \frac{1}{k}\sum_{j=1}^{k} CV_j$$

전체의 데이터 $D = (X_i, Y_i)_{1 \le i \le n}$를 k개의 그룹으로 나눠서,

j번째 그룹의 데이터를 제외한 데이터를 $D^{-j} = (X_i, Y_i)_{i \in S - S_j}$ 라 하자.

이때, $S = \{1, 2, \ldots n\}$, $\cup_{j=1}^{k} S_j = S$, $S_i \cap S_j = \varnothing, (i \neq j)$ 가 성립한다.

예를들어, 100개의 데이터를 10-fold cross validation 한다고 하면,

$S_1 = \{1, 2..10\}, S_2 = \{11, 12...20\} \ldots S_{10} = \{91, \ldots 100\}$이 될것이다.

$$CV_j = \frac{1}{n_j}\sum_{i \in S_j} (Y_i - \hat{m^{-j}}(X_i))^2$$

,

n_j는 S_j의 원소의 개수,

m^{-j}는 $D^{-j} = (X_i, Y_i)_{i \in S - S_j}$ 를 사용하여 추정한 함수이다.

이제, 위에서 처럼 xdata, ydata, smoothing parameter를 받아 estimate된 값을 출력하는 함수를 estimate_func라 할때, 이를 이용해서 k-fold CV의 값을 구해주는 함수를 만들것이다.

먼저, 아래는 전체 n개의 데이터를 k개의 집합으로 나눠주는 함수이다.

리턴하는 값은 list로, 각 list에는 S_j에 속하는 index들이 담겨있다.

각 집합에는 $[n/k]$혹은 $[n/k]+1$개의 데이터가 들어간다. $[x]$는 x를 넘지 않는 가장 큰 정수이다.

```
get_fold_seq <- function(k, n)
{
    remainder <- n %% k
    q <- n %/% k
```

```
    start_point = 1

    idx_lst <- list()
    for(i in seq_len(k))
    {
        if(remainder > 0)
        {
            idx_lst[[i]] = seq(from = start_point, by = 1, length = q + 1)
            start_point <- start_point + q + 1
            remainder <- remainder - 1
        }
        else
        {
            idx_lst[[i]] = seq(from = start_point, by = 1, length = q)
            start_point <- start_point + q
        }
    }

    return(idx_lst)
}
```

```
get_kfold_cross_validation <- function(estimate_func, x1, y1, smtparam_array, k)
{
#   (1)
    get_kfold_cv <- function(smtparam,estimate_func, x1 ,y1, k)
    {
#   (2)
        cv <- 0
        nd <- length(x1)
        idx_lst <- get_fold_seq(k, nd)
        for(i in seq_len(k)) # get CV_(j) and sum to cv
        {
            xdata <- x1[-idx_lst[[i]]]
            ydata <- y1[-idx_lst[[i]]]
            remain_xdata <- x1[idx_lst[[i]]]
            remain_ydata <- y1[idx_lst[[i]]]
```

```
            estim_y <- estimate_func(xdata, ydata, smtparam, remain_xdata)
            cv <- cv + sum((remain_ydata - estim_y)**2)/length(remain_xdata)
        }

        cv <- cv / k
#   (7)
        return(cv)
    }
#   (8)
    cvlst <- lapply(as.list(smtparam_array),get_kfold_cv,  estimate_func = estimate_func, x1 = x1, y1 = y1, k =
 k)
    cvlst <- unlist(cvlst)
    return(cvlst)
}
```

위 함수는, x,y,smoothing parameter를 가지고 추정값을 리턴해주는 estimate_func과, smoothing parameter들 목록, k를 받아 각 smoothing parameter에 해당하는 k-fold cross validation 값들을 리턴해준다.

또한, $\sum_{i=252}^{263}(Y_i - \hat{Y}_i)^2$를 구해주는 함수는 다음과 같다.

```
get_predict_err <- function(estimate_func, xdata, ydata, smtparam, remain_x, remain_y)
{
    pred_y <- estimate_func(xdata, ydata, smtparam, remain_x)
    # get predicted value using xdata, ydata at remain_x

    res <- sum((pred_y-remain_y)**2)

    return(res)
}
```

또한, 다음은 xdata, ydata를 이용해 추정한 그래프를, remain_x값에 해당하는 위치까지 확장해서 그림을 그려준다.

```
plot_graph_remain <- function(estimate_func, xdata, ydata, smtparam, remain_x, remain_y , description = "")
{
    whole_x <- c(xdata, remain_x)
, remain_y)
```

```
    par(mfrow=c(1,1))
    estimated_y <- estimate_func(xdata, ydata, smtparam, whole_x)

    plot(whole_x, whole_y , xlab = "X", ylab = "Y", type = "p", main = description, sub = sprintf("using smtparam
 : %f", smtparam))
    lines(whole_x, estimated_y, col = "blue")
}
```

이를 이용해서, 각 data를 이용해서 각 smoothing parameter에 대해 k-fold cross validation 값이 가장 작은 smoothing parameter를 구하고, 그 값에 대해 smoothing을 진행한 결과를 그래프로 plot하고, 마지막으로 $i = 252 \sim 263$에 대해 predict error를 출력하는 함수는 다음과 같다.

```
full_test_kfold <- function(estimate_func, xdata, ydata, smtparam_array, k, description = "", using_CV = TRUE)
{
    remain_x <- xdata[252:263]
    remain_y <- ydata[252:263]
    xdata_excepted <- xdata[1:251]
    ydata_excepted <- ydata[1:251]


    if(using_CV){
        cv_array <- get_kfold_cross_validation(estimate_func, xdata_excepted, ydata_excepted, smtparam_array, k)

        plot_cv(smtparam_array, cv_array, description)

        smt_idx <- which(cv_array == min(cv_array, na.rm = TRUE))
        if(length(smt_idx) > 1)
        {
            smt_idx <- smt_idx[1]
        }
        best_smt <- smtparam_array[smt_idx]
    }
    else
    {
        best_smt <- smtparam_array[1]
    }
    plot_graph_remain(estimate_func, xdata_excepted, ydata_excepted, best_smt, remain_x, remain_y,description = d
```

```
    rss <- get_predict_err(estimate_func, xdata_excepted, ydata_excepted, best_smt, remain_x, remain_y)

    return(rss)
}
```

이제 앞에서 구현했던 함수들을 이용하면,

다음과 같이 지금까지 구현했던 모든 method를 이용해서 (super smoother와 lowess를 제외하고) 간편하게 k-fold test가 가능하다.

모든 경우에 대해 test를 해보고, 가장 prediction error가 낮은 method를 사용하는것이 타당하다고 생각된다.

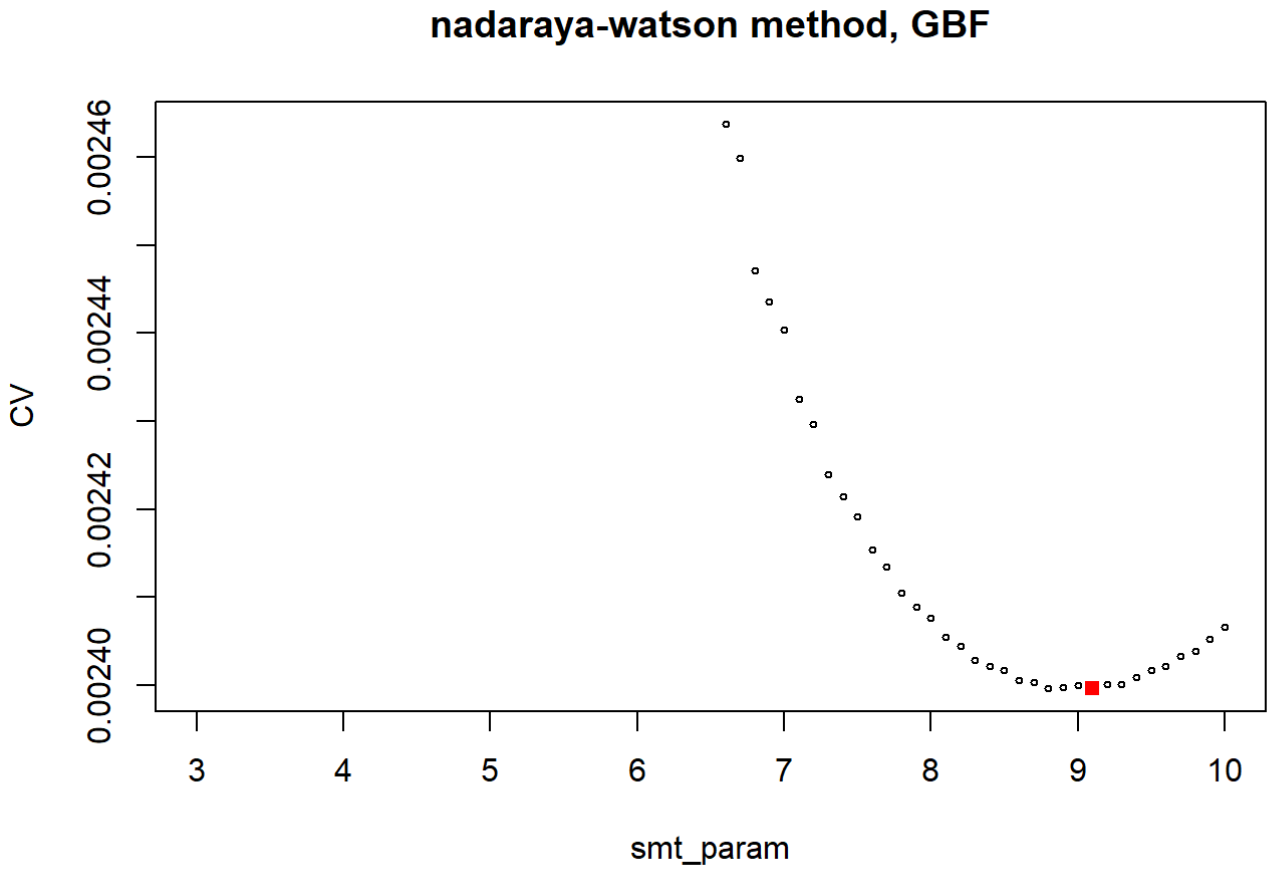
k-fold cross-validation에서, k로는 10을 사용하도록 하겠다.

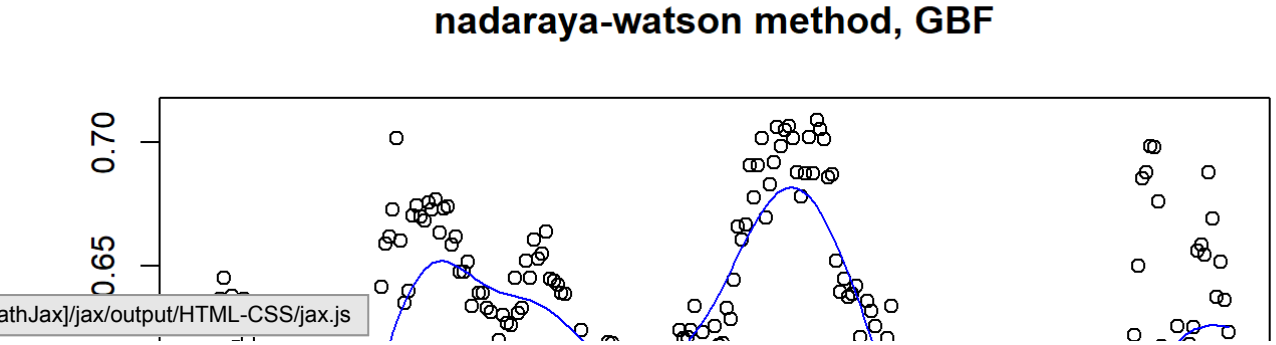
이 결과들에서 prediction error는 다음과 같은 벡터에 저장될것이다.

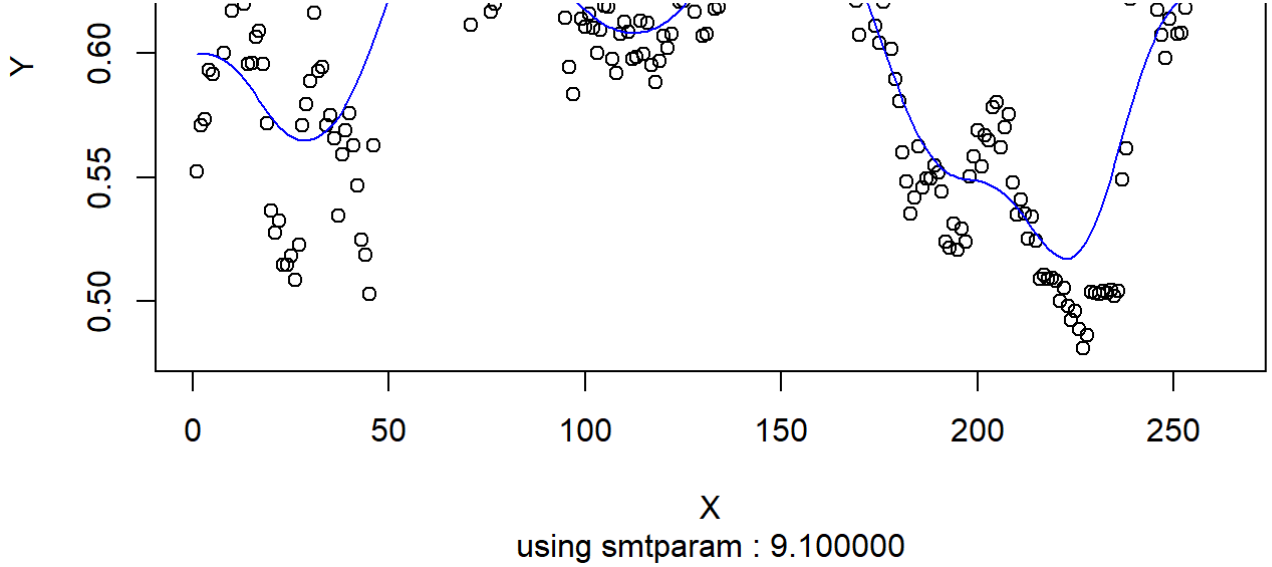
```
pred_err_GBP <- vector()
pred_err_CHF <- vector()
pred_err_CAD <- vector()
```

먼저, nadaraya-watson method를 사용했을때다.

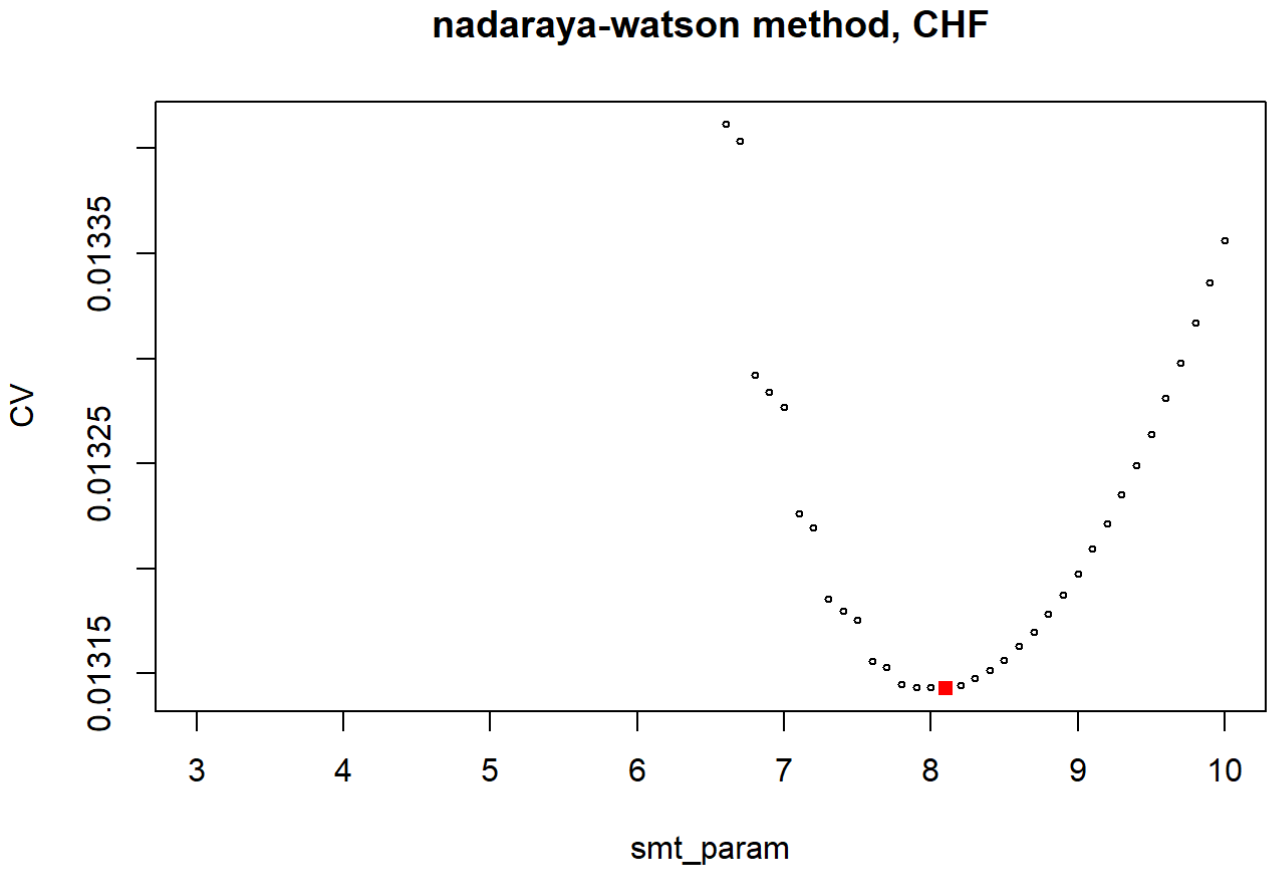
```
pred_err_GBP[1] <- full_test_kfold(estimate_nadaraya, data$X, data$GBP, seq(3,10,by = 0.1), k = 10, description =
 "nadaraya-watson method, GBF")
```

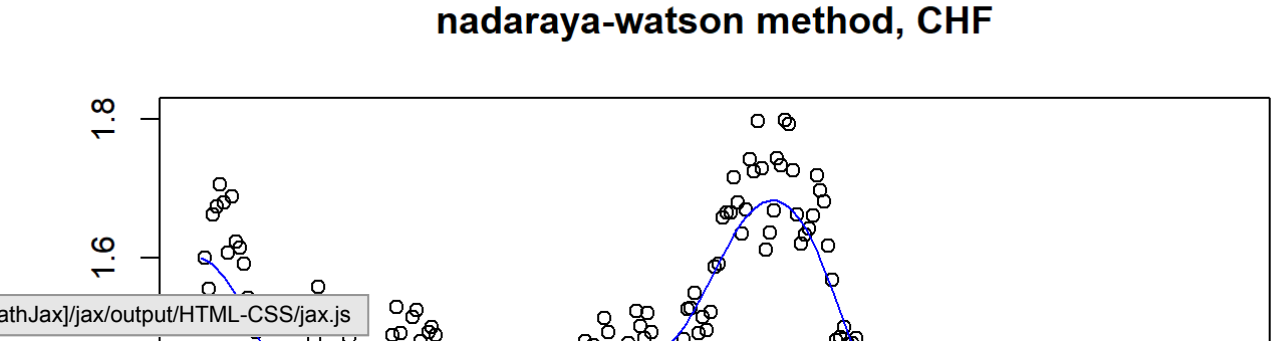


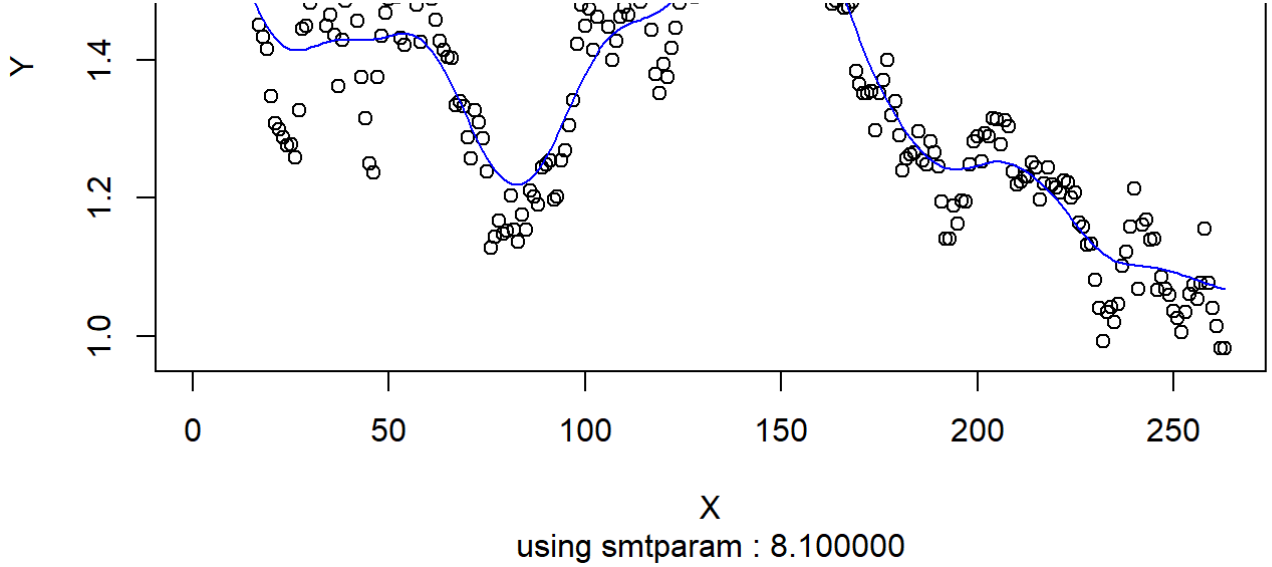




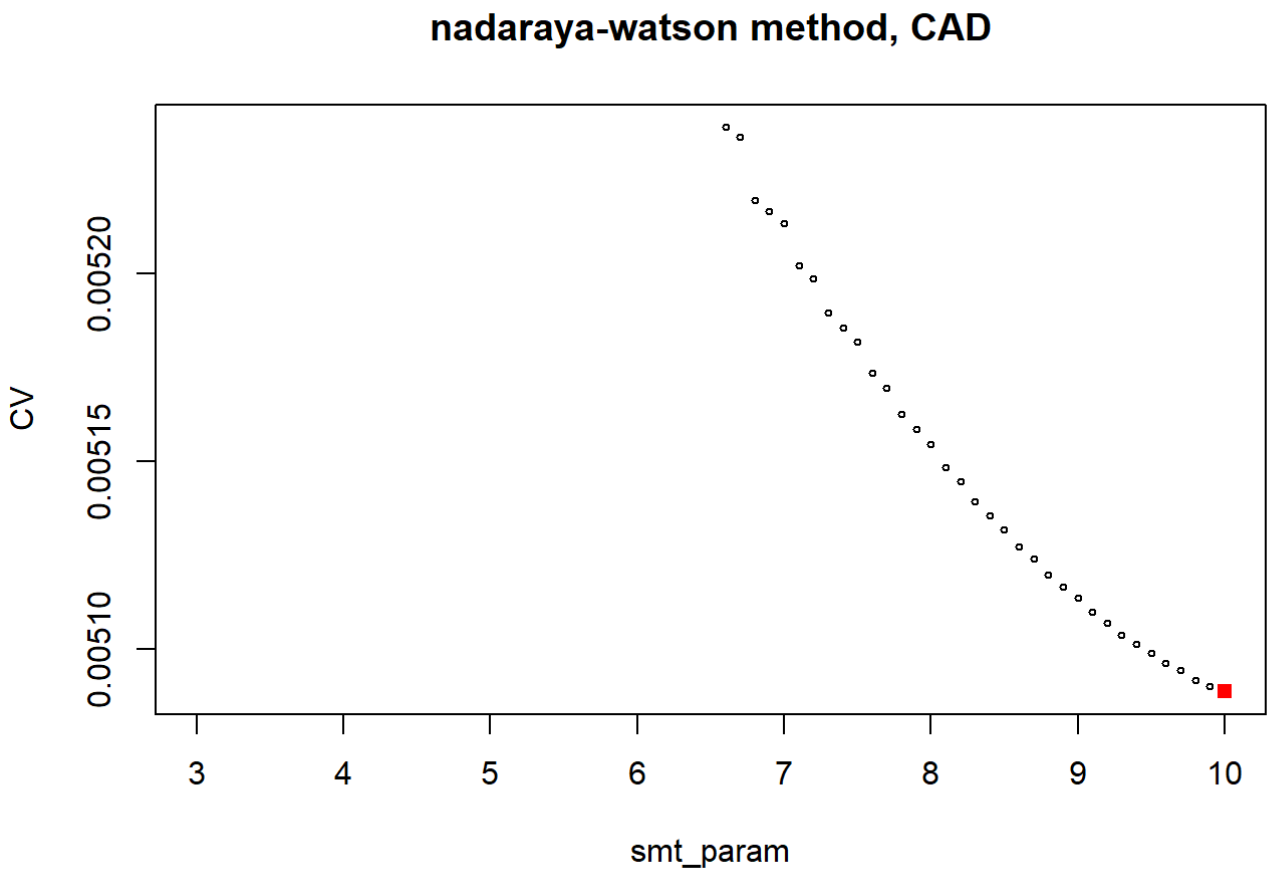

```
pred_err_CHF[1] <- full_test_kfold(estimate_nadaraya, data$X, data$CHF, seq(3,10,by = 0.1), k = 10, description =
 "nadaraya-watson method, CHF")
```

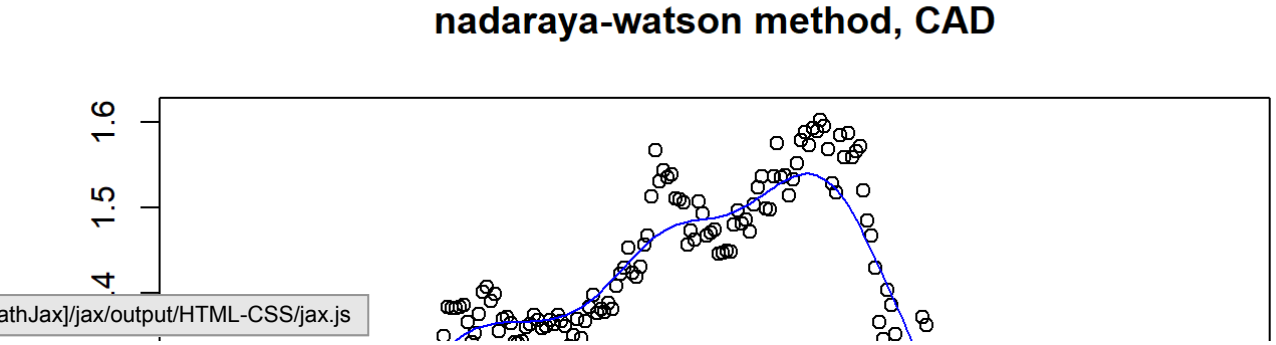


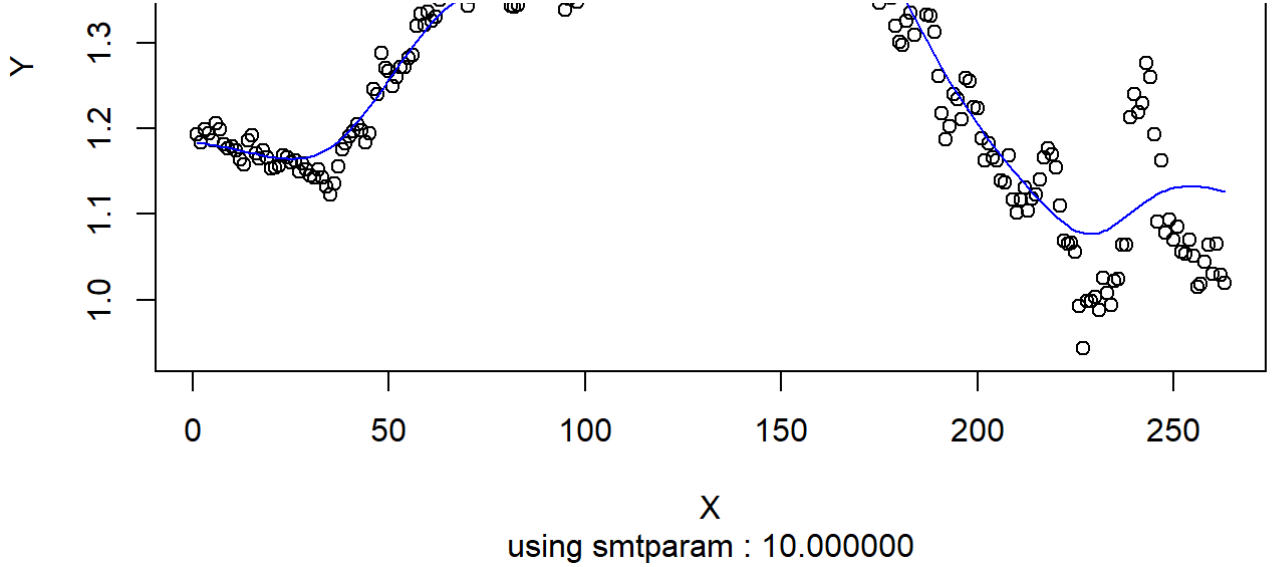





```
pred_err_CAD[1] <- full_test_kfold(estimate_nadaraya, data$X, data$CAD, seq(3,10,by = 0.1), k = 10, description =
 "nadaraya-watson method, CAD")
```

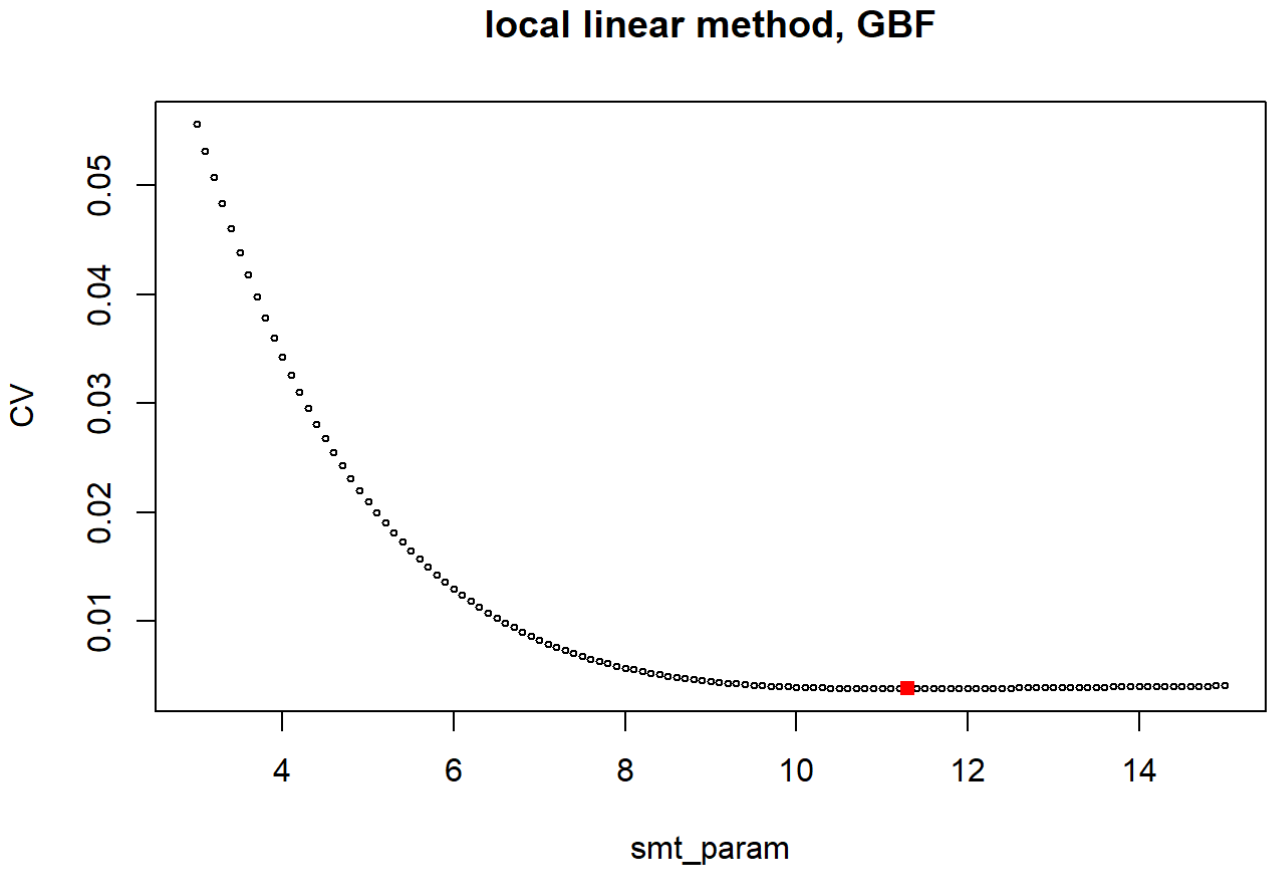


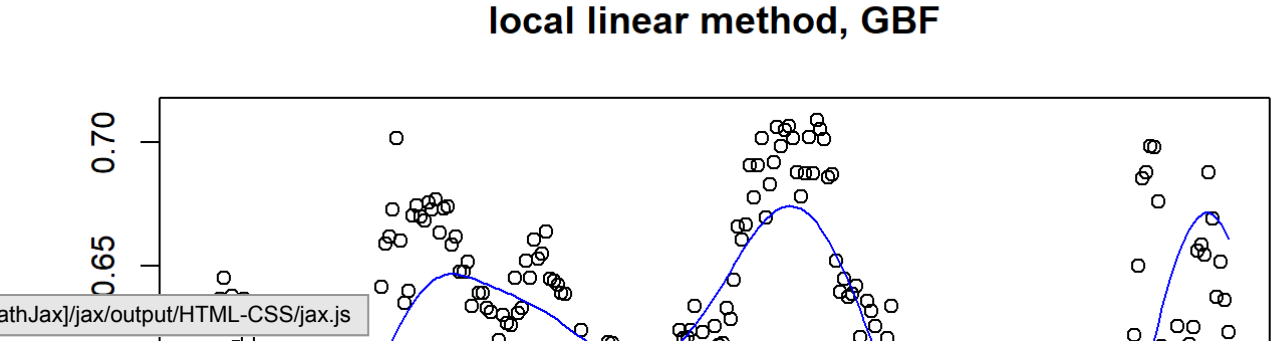


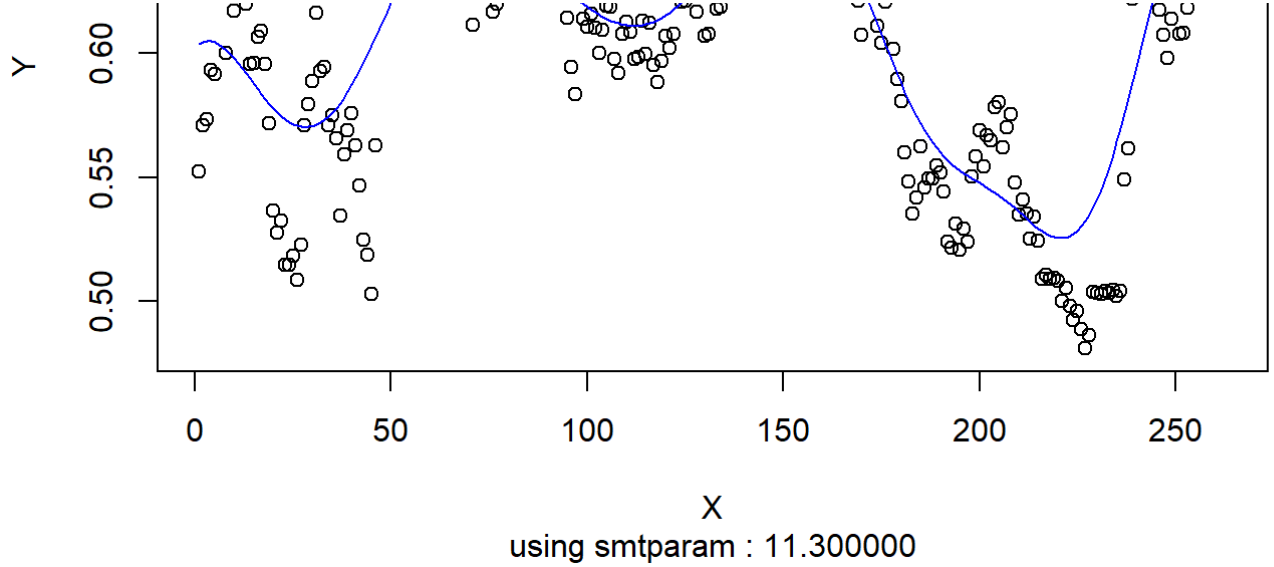


다음은 local linear method를 사용했을 때다. bandwidth로 3~15, 0.1간격으로 사용하도록 하겠다.

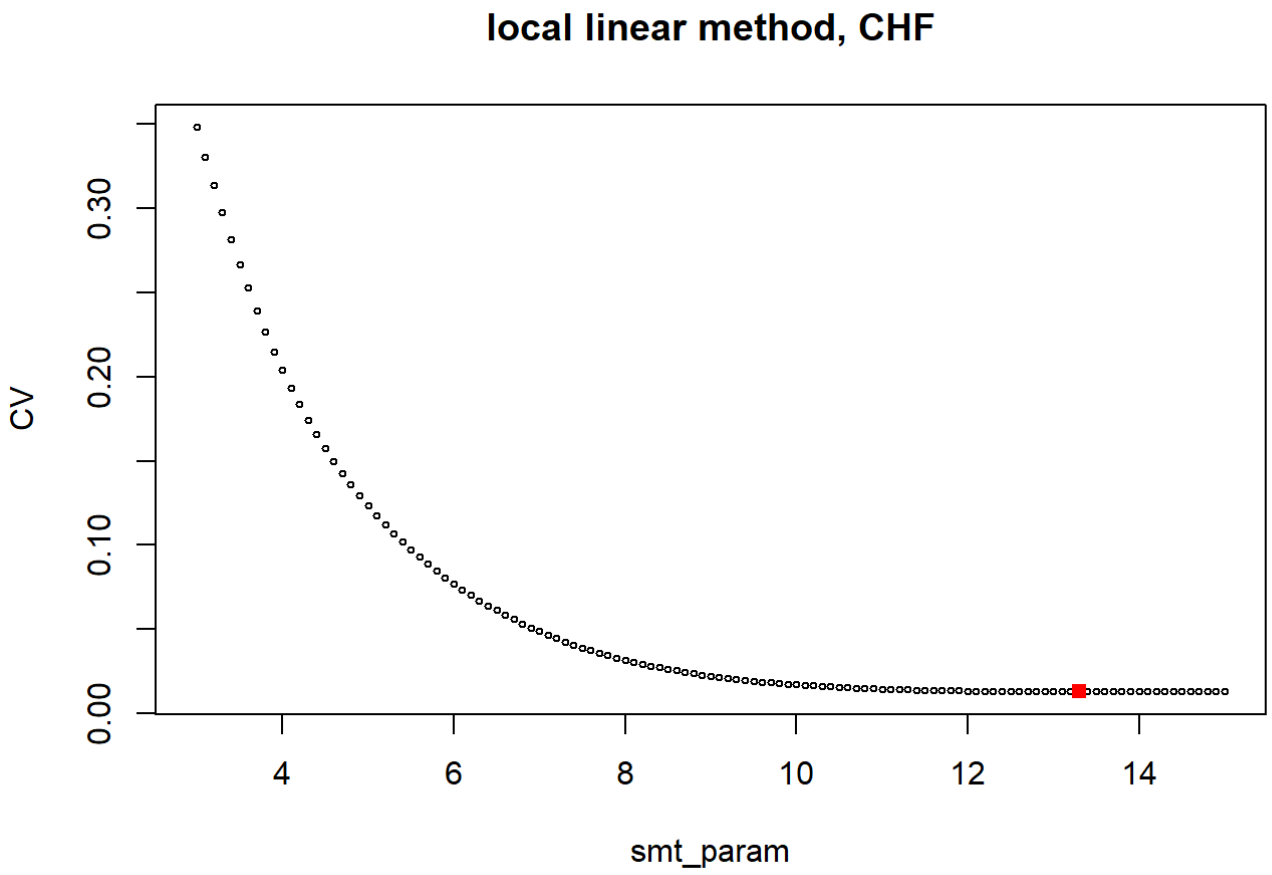
```
pred_err_GBP[2] <- full_test_kfold(estimate_local_linear, data$X, data$GBP, seq(3,15,by = 0.1), k = 10, descripti
on = "local linear method, GBF")
```

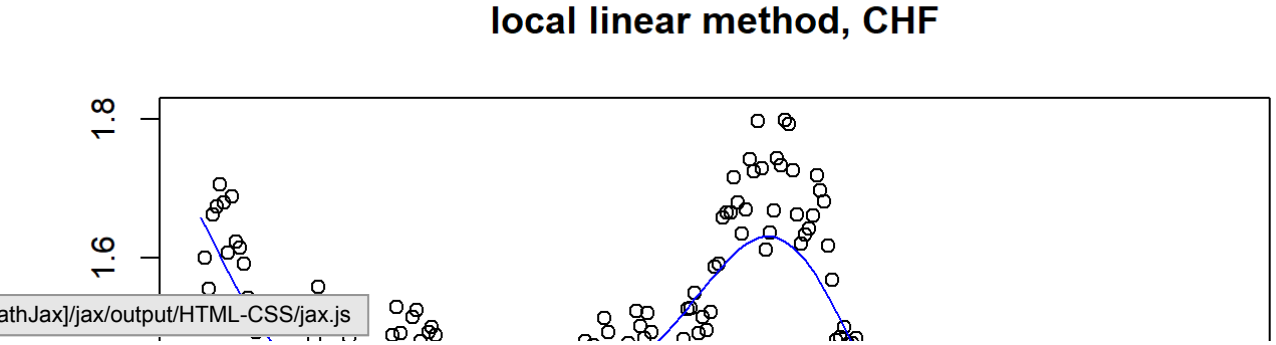


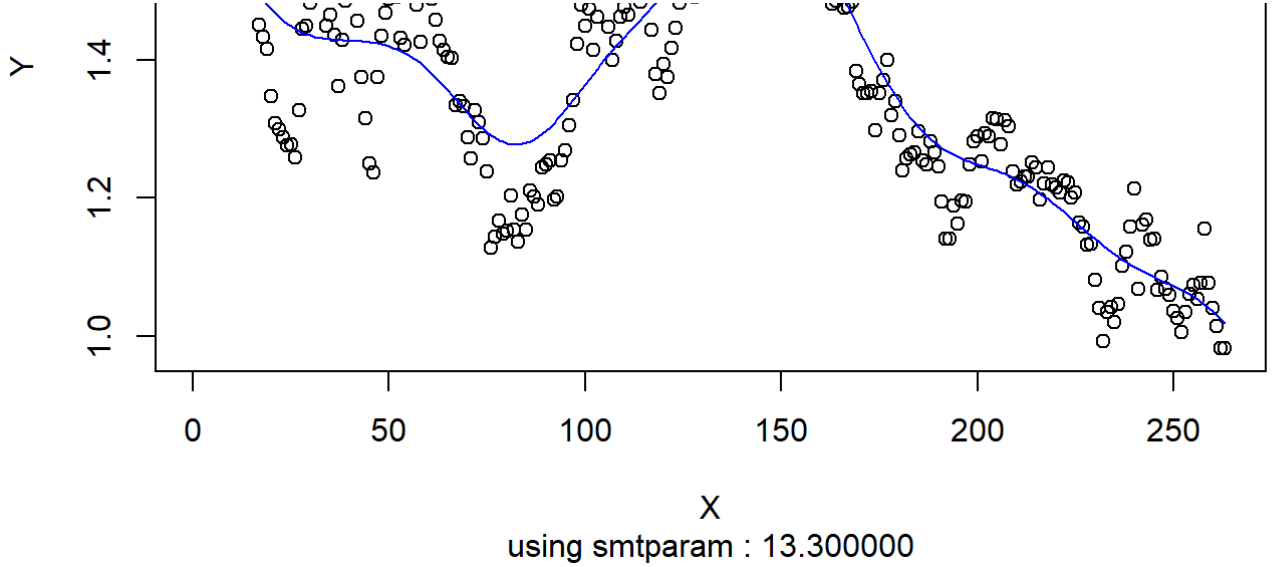




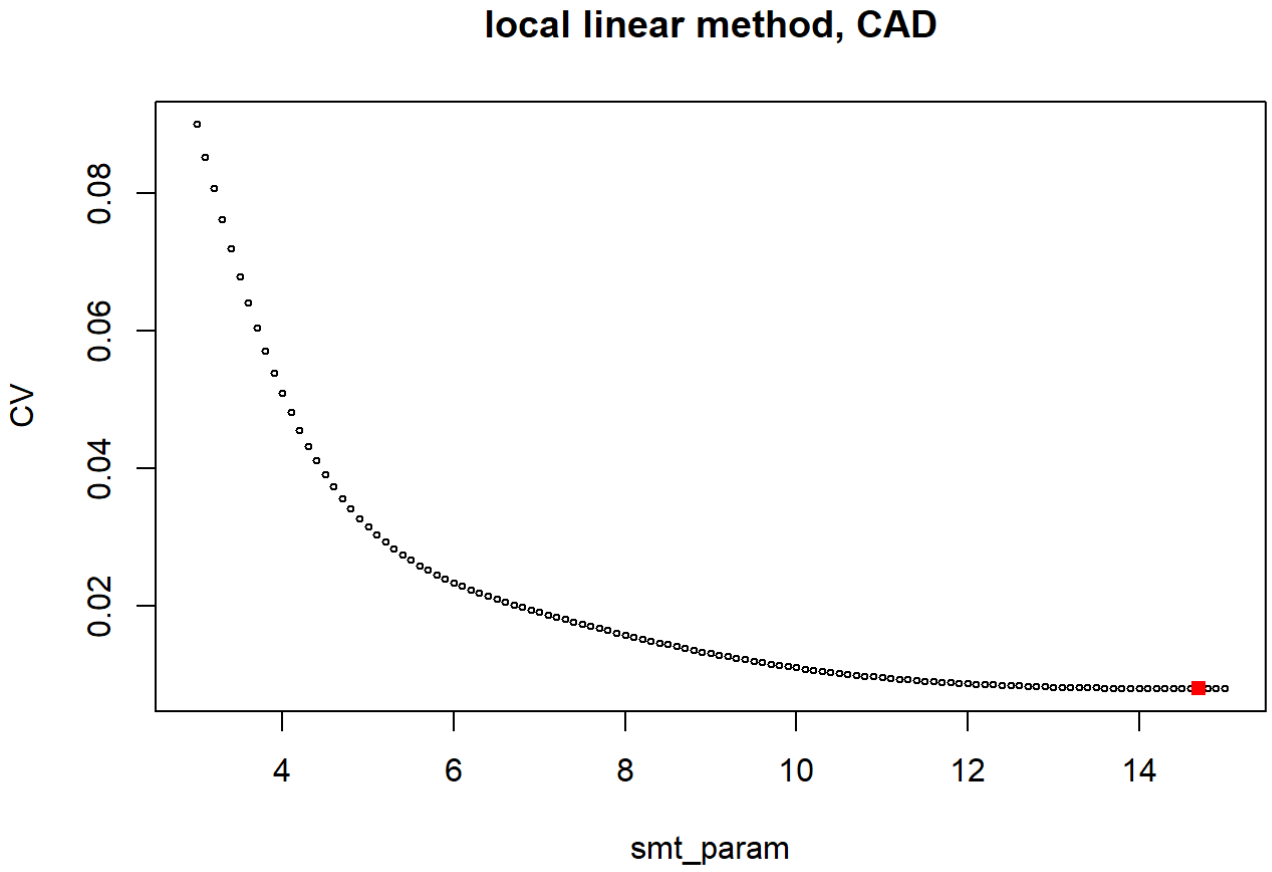

```
pred_err_CHF[2] <- full_test_kfold(estimate_local_linear, data$X, data$CHF, seq(3,15,by = 0.1), k = 10, descripti
on = "local linear method, CHF")
```

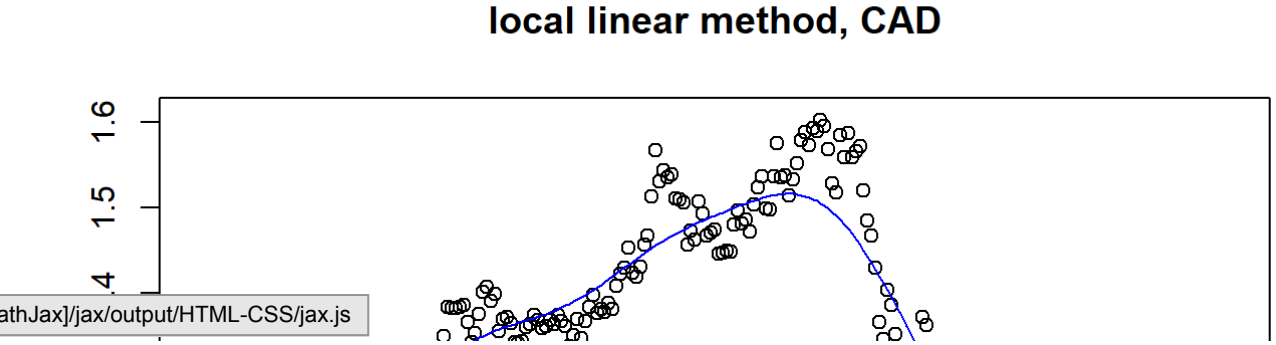


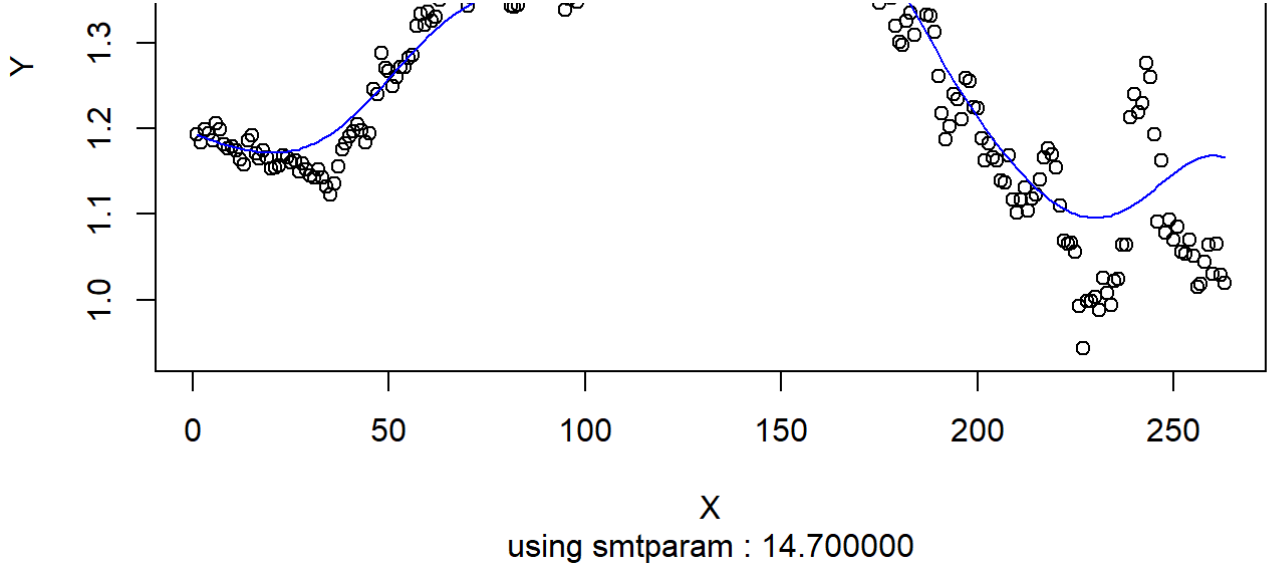





```
pred_err_CAD[2] <- full_test_kfold(estimate_local_linear, data$X, data$CAD, seq(3,15,by = 0.1), k = 10, descripti
on = "local linear method, CAD")
```

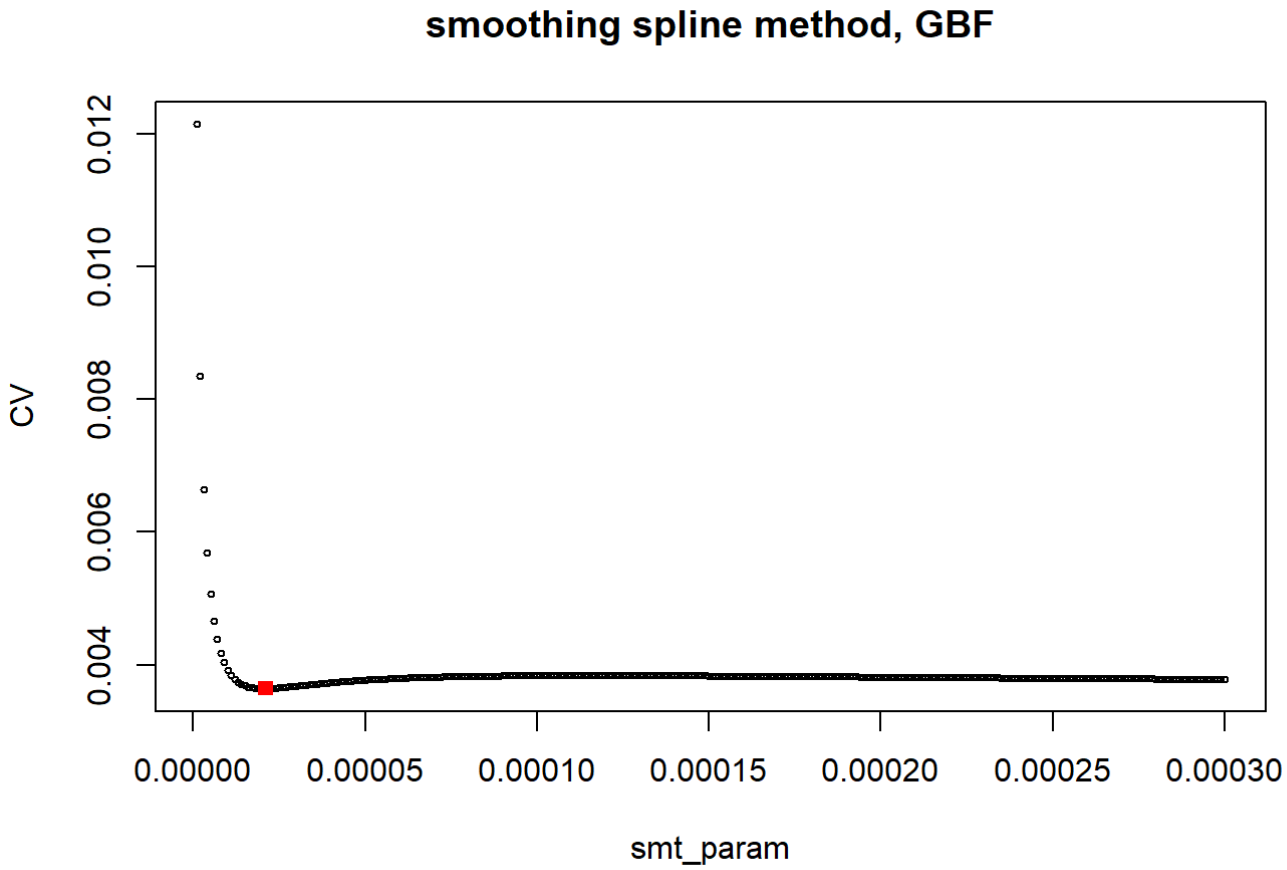


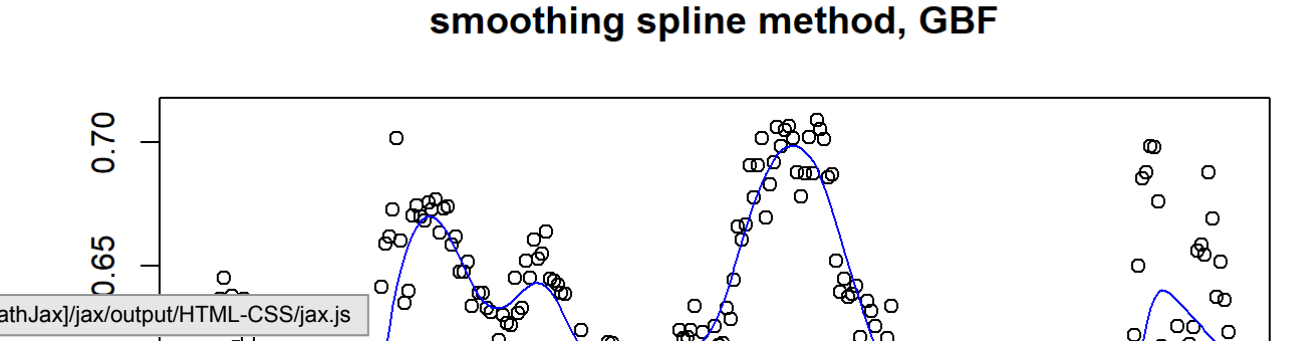


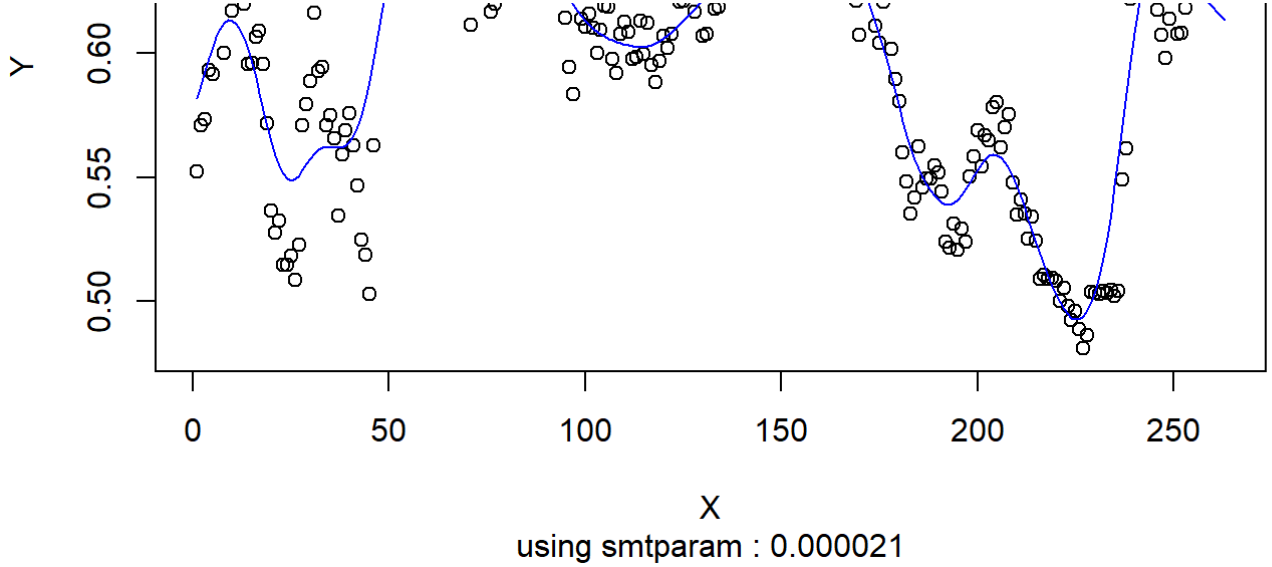


다음은 smoothing spline을 사용했을때다, lambda로 0.000001~0.0003까지 0.000001간격으로 테스트를 해보겠다.

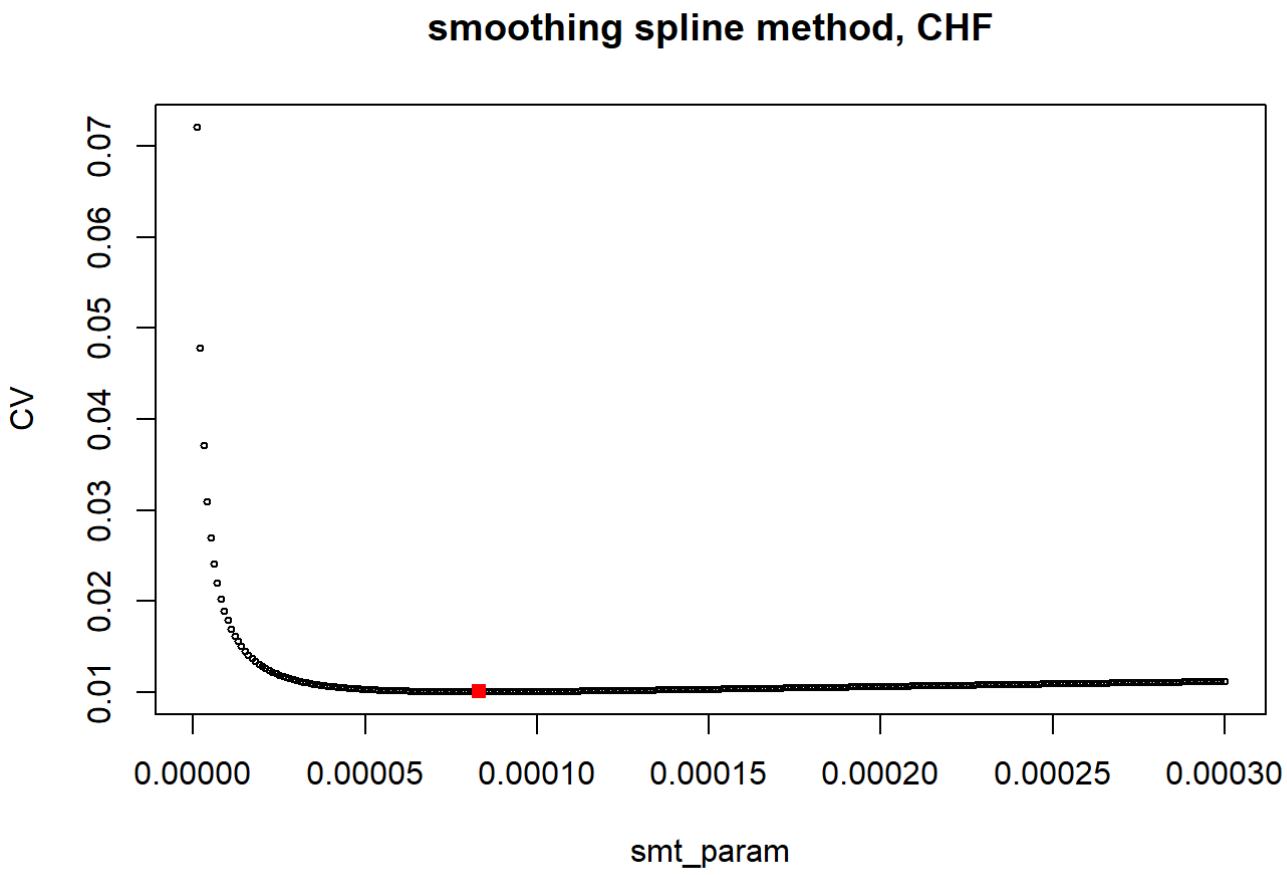
```
pred_err_GBP[3] <- full_test_kfold(estimate_smoothing_spline, data$X, data$GBP, seq(0.000001,0.0003,by = 0.000001
), k = 10, description = "smoothing spline method, GBF")
```

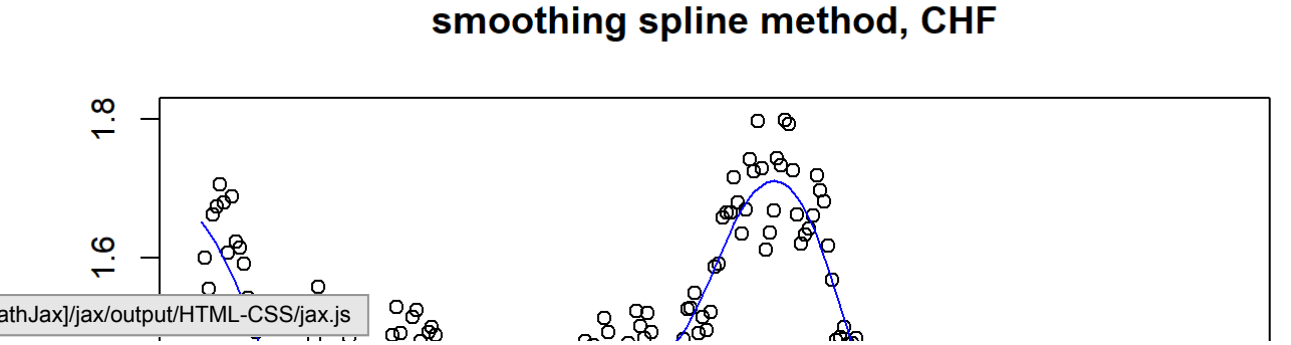


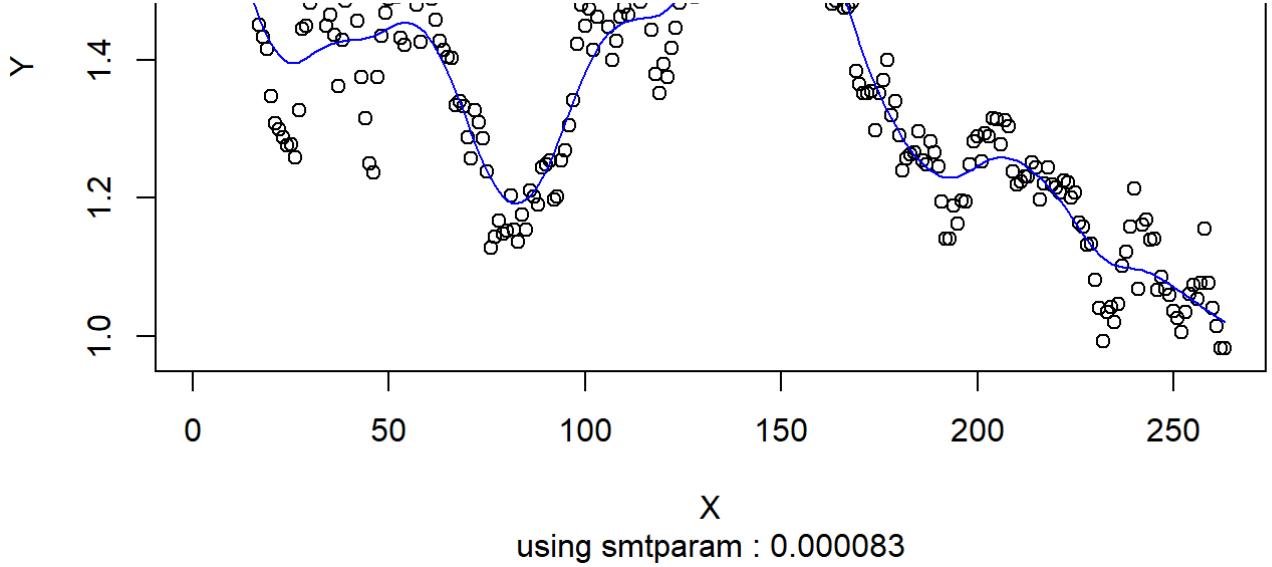




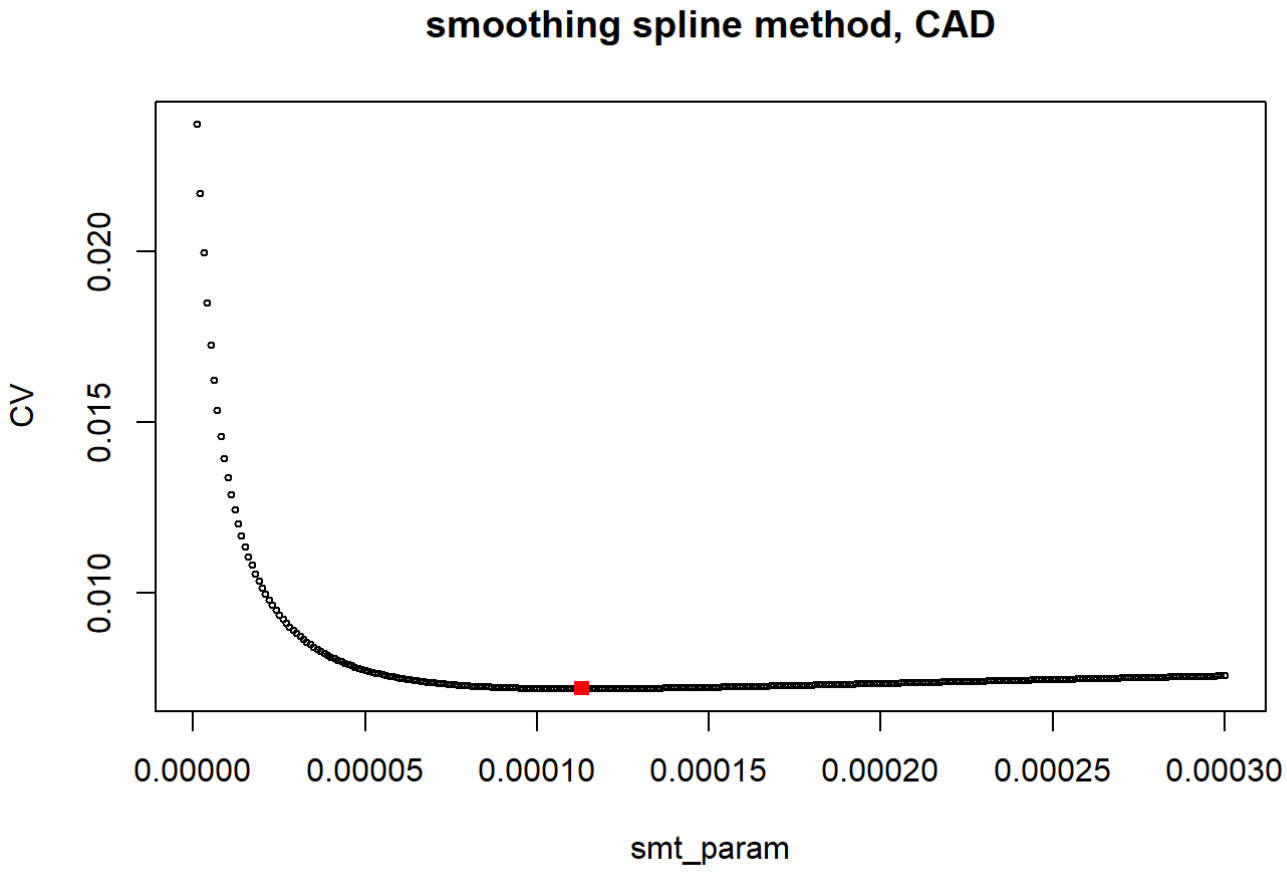

```
pred_err_CHF[3] <- full_test_kfold(estimate_smoothing_spline, data$X, data$CHF, seq(0.000001,0.0003,by = 0.000001
), k = 10, description = "smoothing spline method, CHF")
```

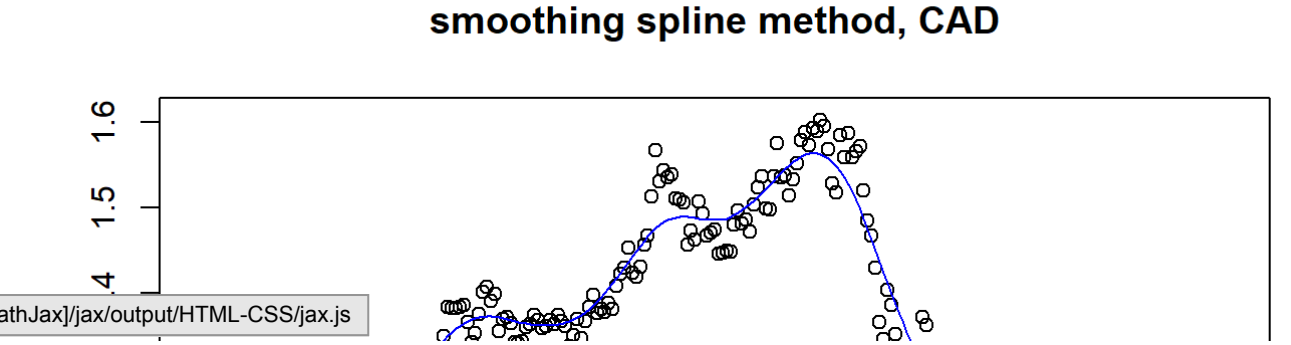


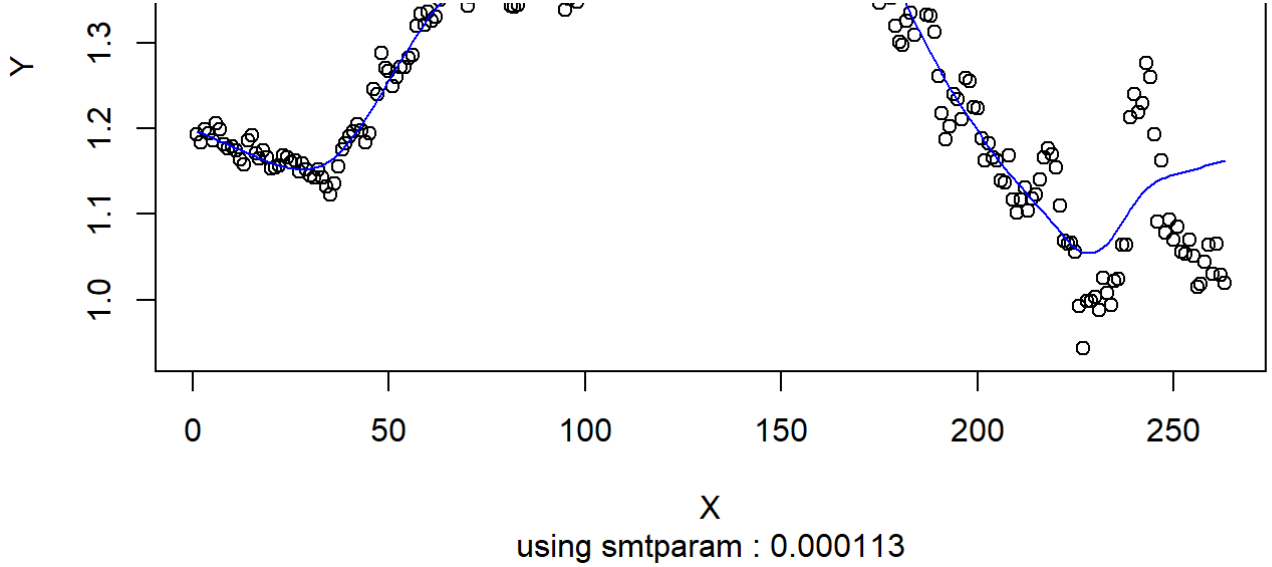





```
pred_err_CAD[3] <- full_test_kfold(estimate_smoothing_spline, data$X, data$CAD, seq(0.000001,0.0003,by = 0.000001
), k = 10, description = "smoothing spline method, CAD")
```

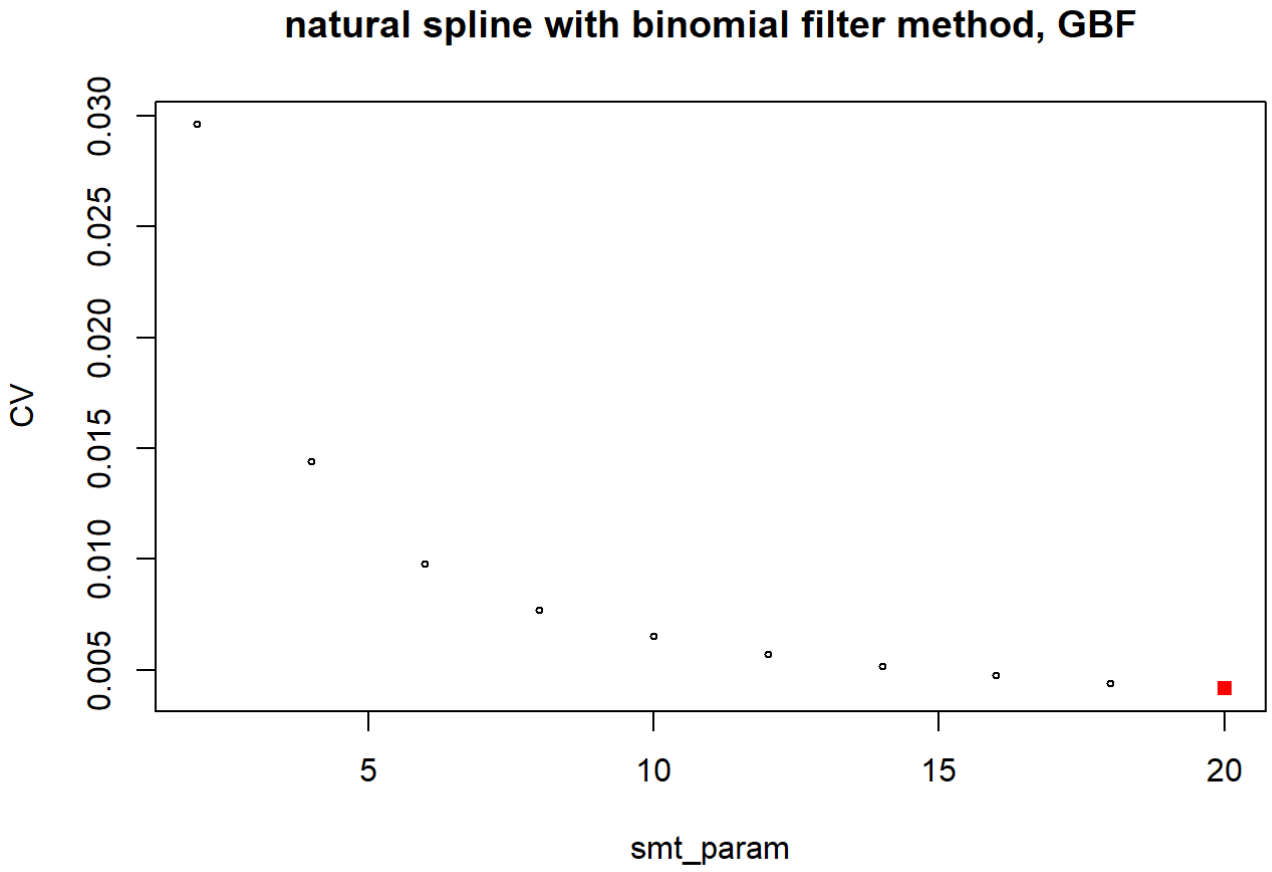


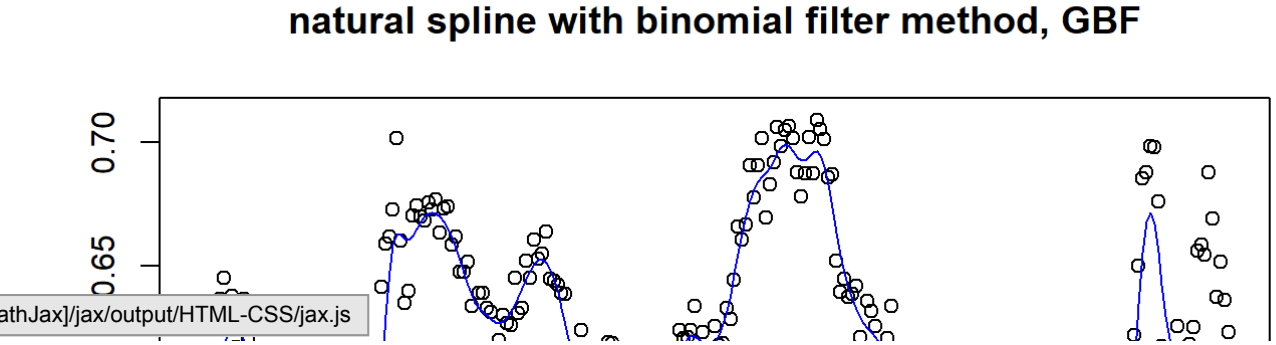


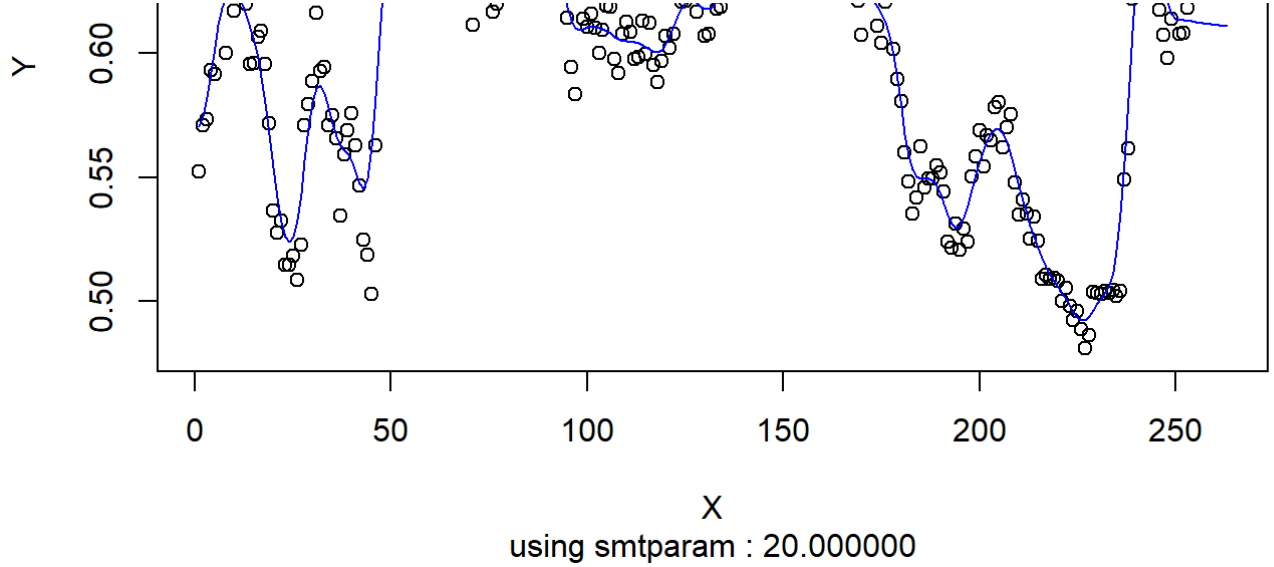


다음은 natural spline (using binomial filter)를 사용했을때다. m으로 2~20까지 2간격으로 테스트를 해보겠다.

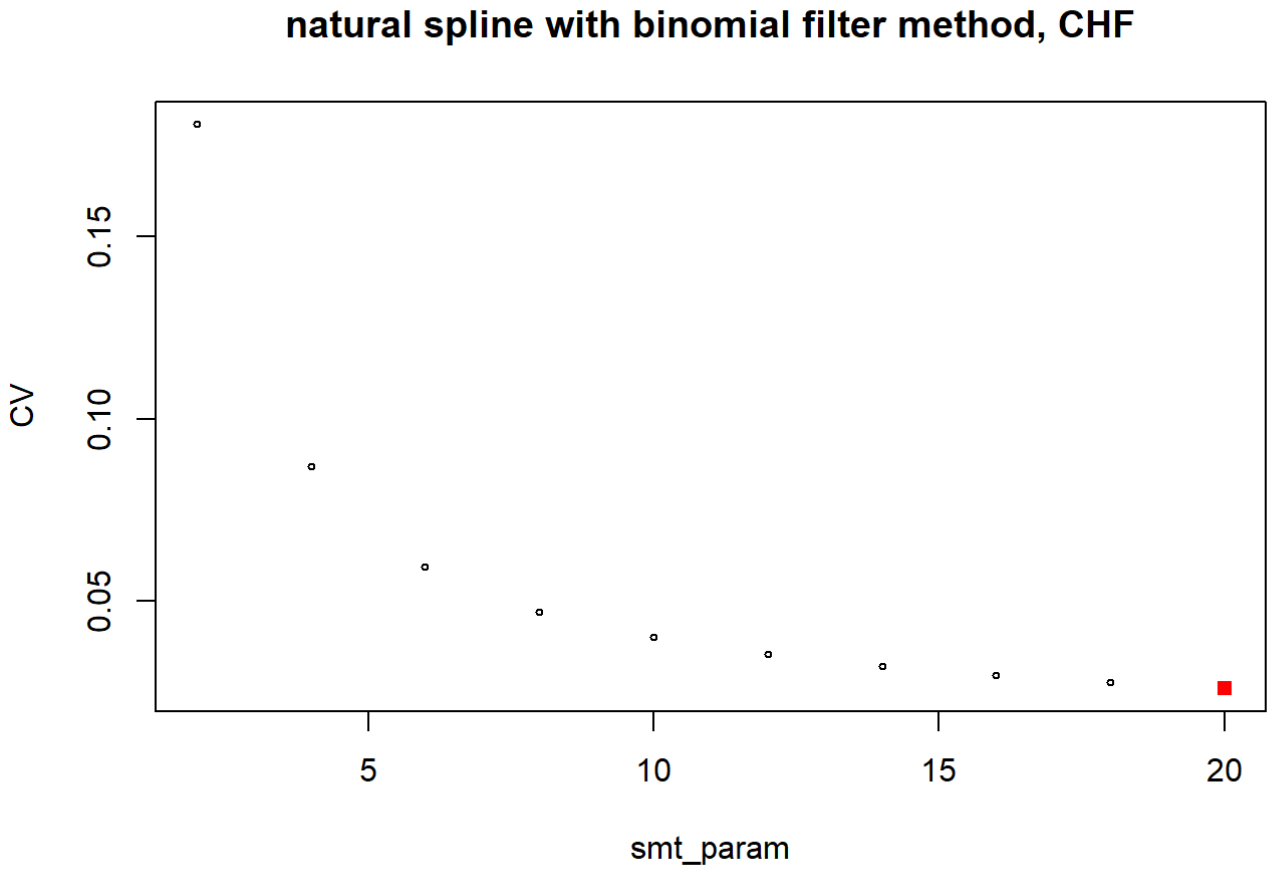
```
pred_err_GBP[4] <- full_test_kfold(estimate_natural_spline_binomfilter, data$X, data$GBP, seq(2,20,by = 2), k = 10, description = "natural spline with binomial filter method, GBF")
```

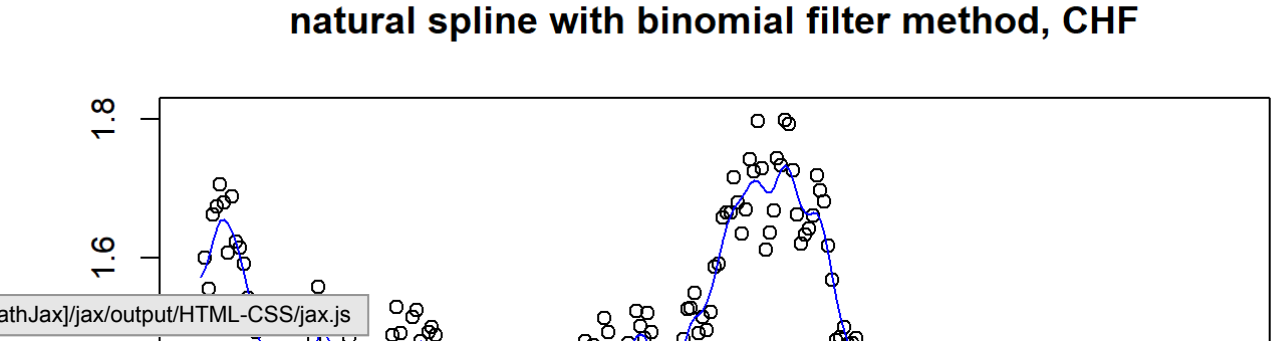


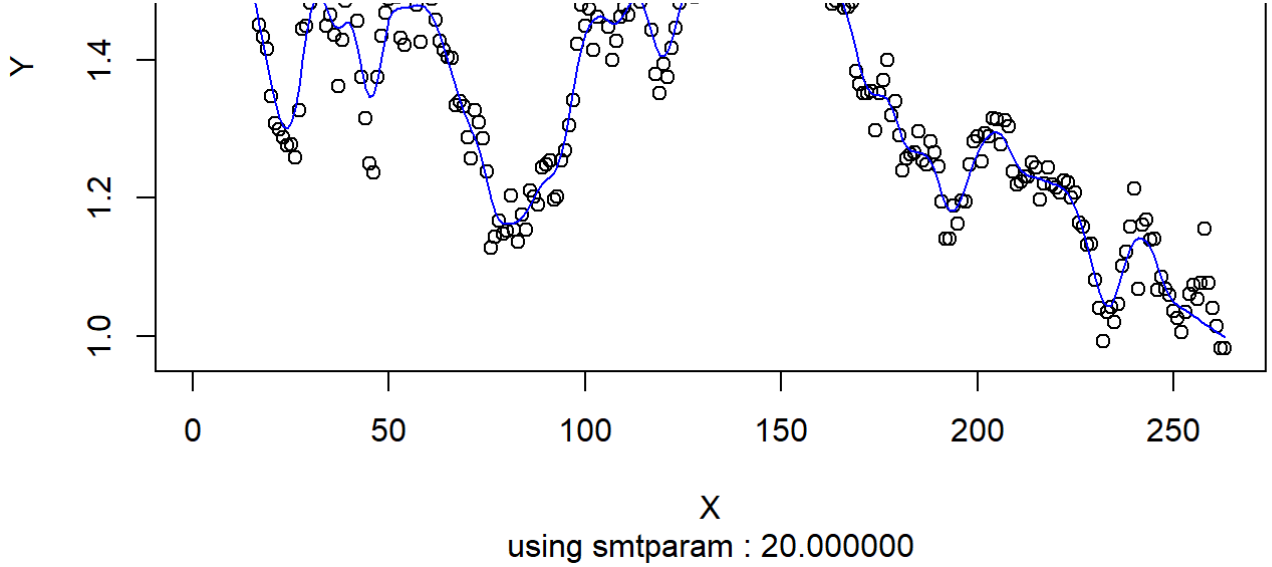




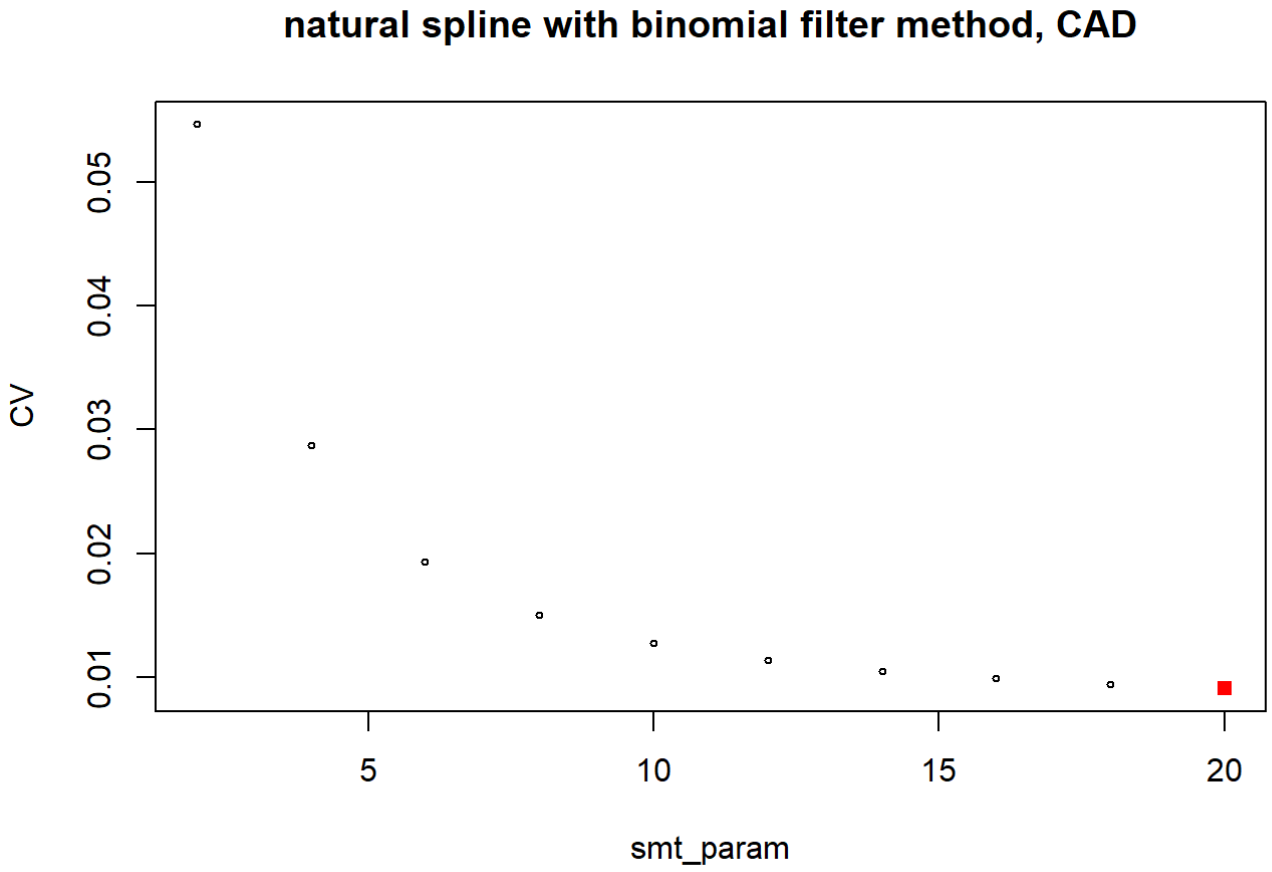

```
pred_err_CHF[4] <- full_test_kfold(estimate_natural_spline_binomfilter, data$X, data$CHF, seq(2,20,by = 2), k = 10, description = "natural spline with binomial filter method, CHF")
```

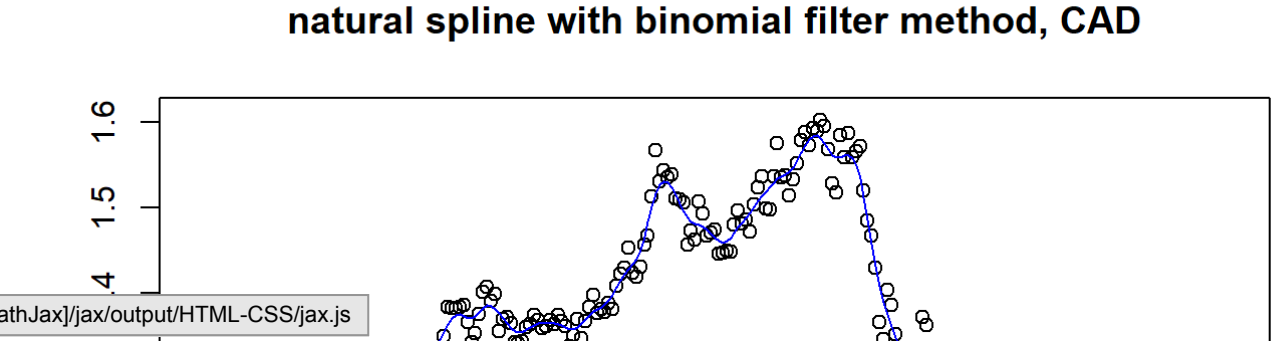


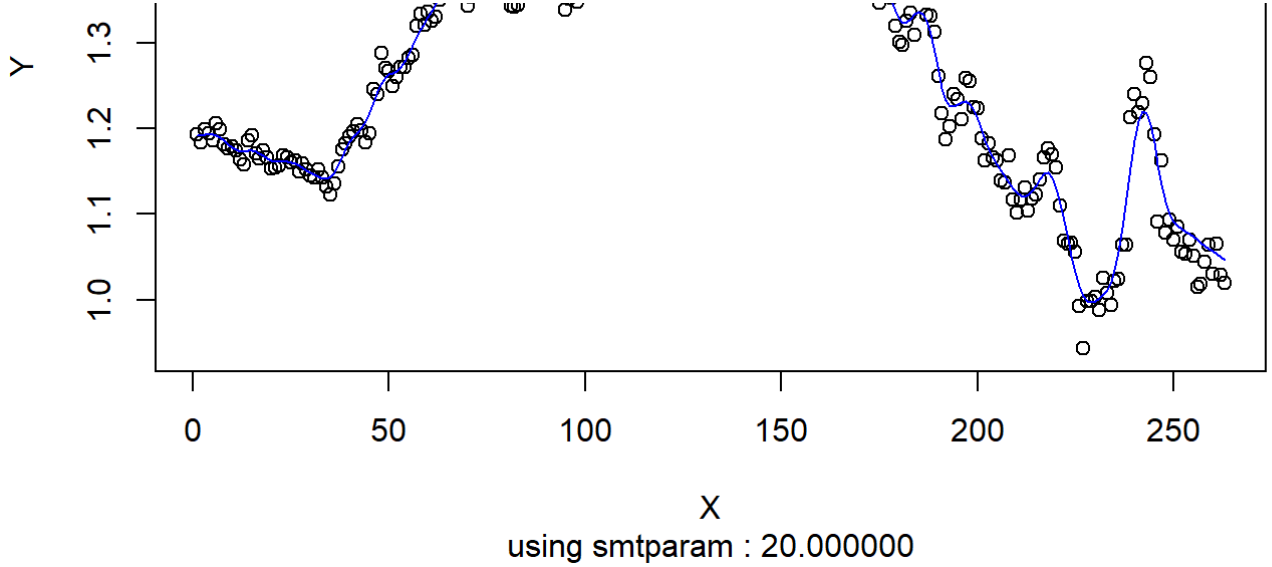





```
pred_err_CAD[4] <- full_test_kfold(estimate_natural_spline_binomfilter, data$X, data$CAD, seq(2,20,by = 2), k = 10, description = "natural spline with binomial filter method, CAD")
```



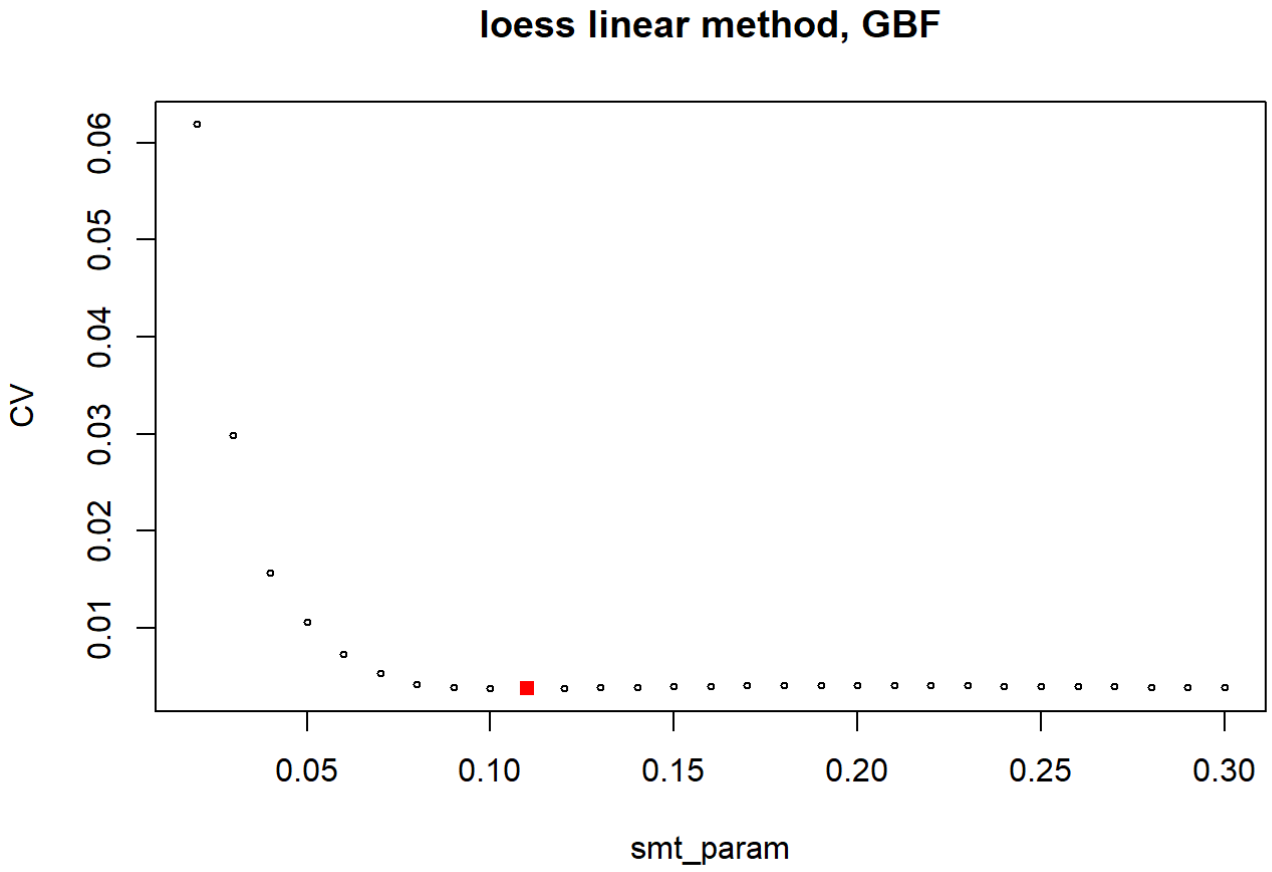


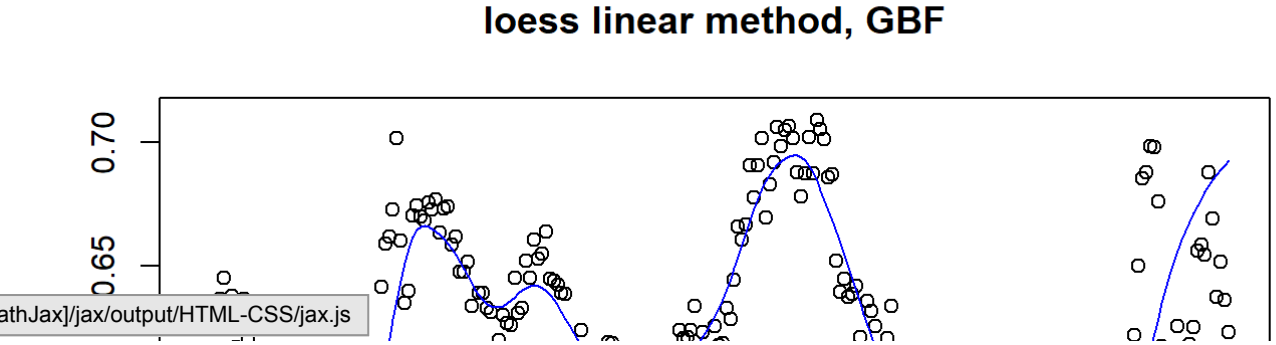


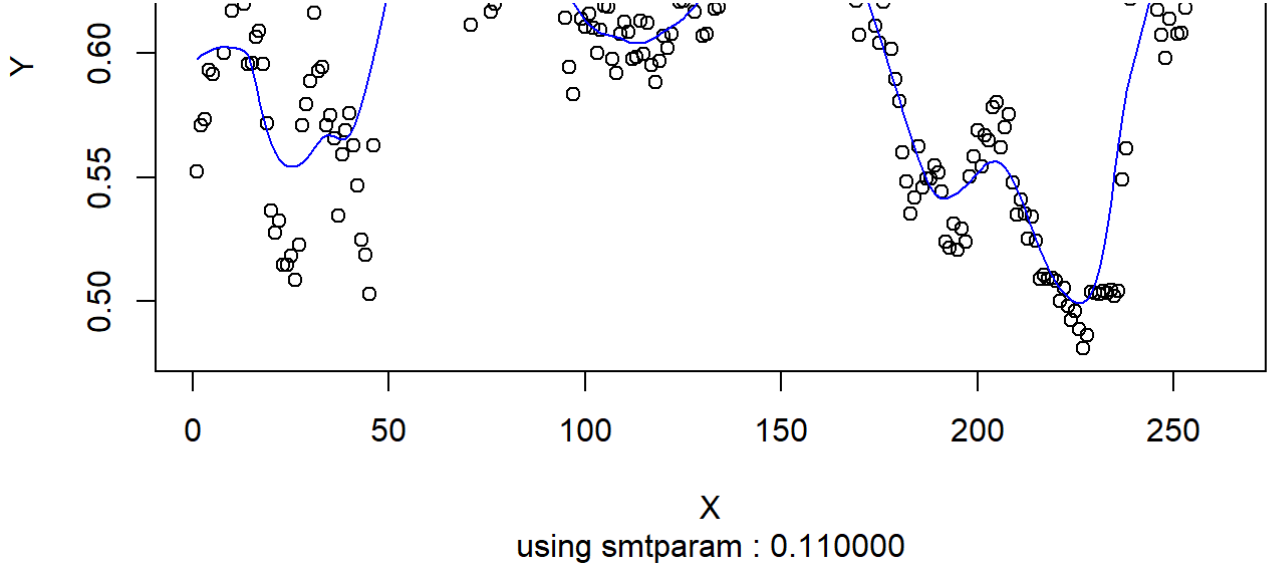
다음은 loess를 사용했을때다. span으로 0.02~0.3까지 0.01간격으로 테스트를 해보겠다.

먼저 linear case이다.

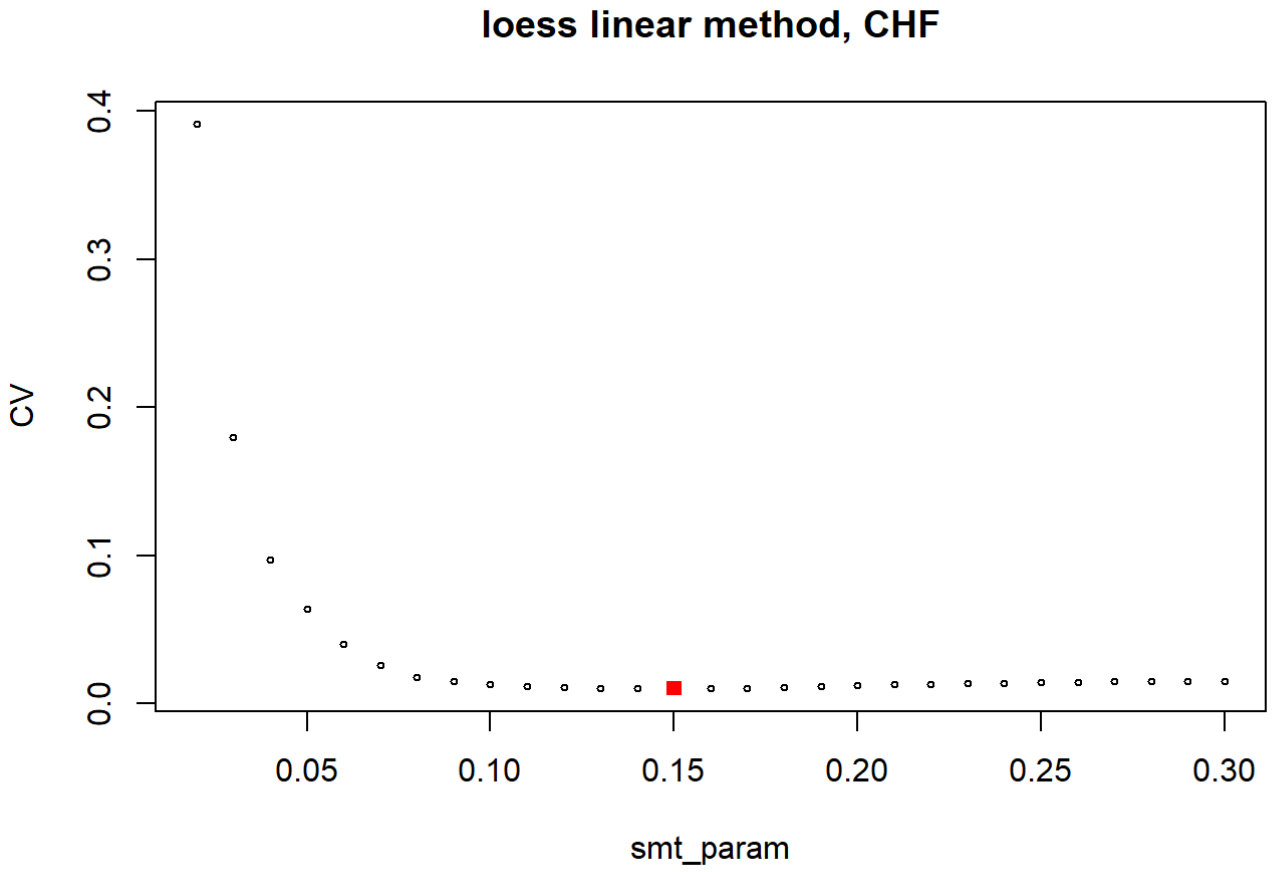
```
pred_err_GBP[5] <- full_test_kfold(estimate_loess_linear , data$X, data$GBP, seq(0.02,0.3,by = 0.01), k = 10, des
cription = "loess linear method, GBF")
```

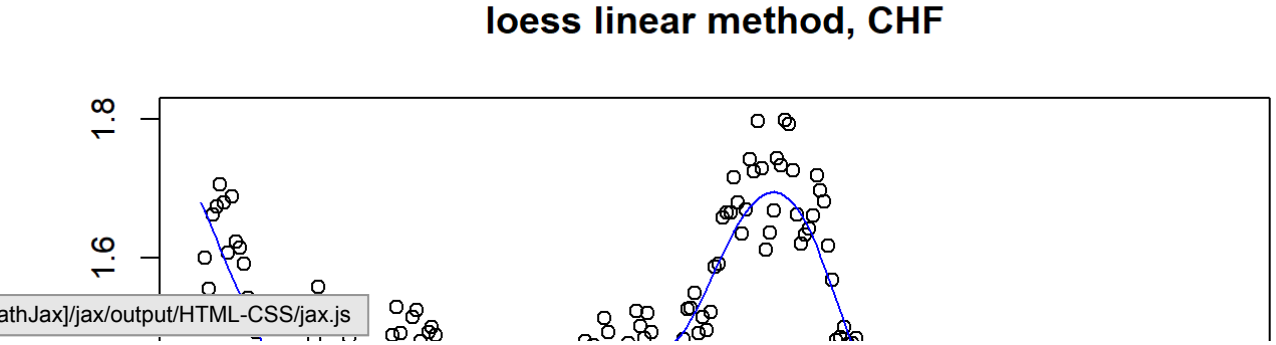


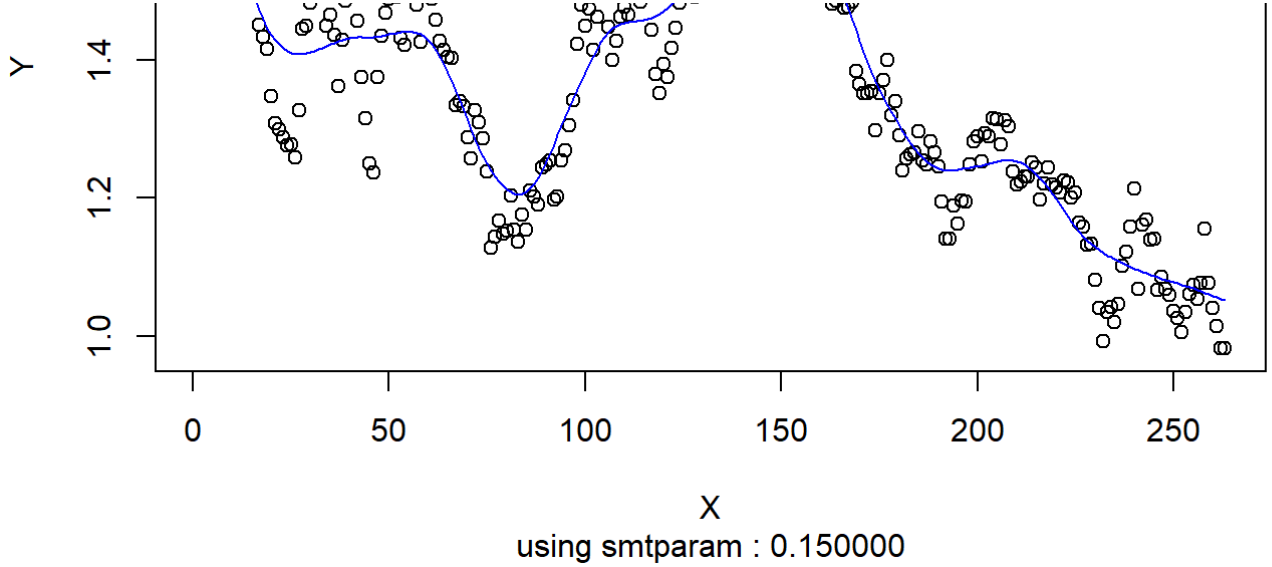




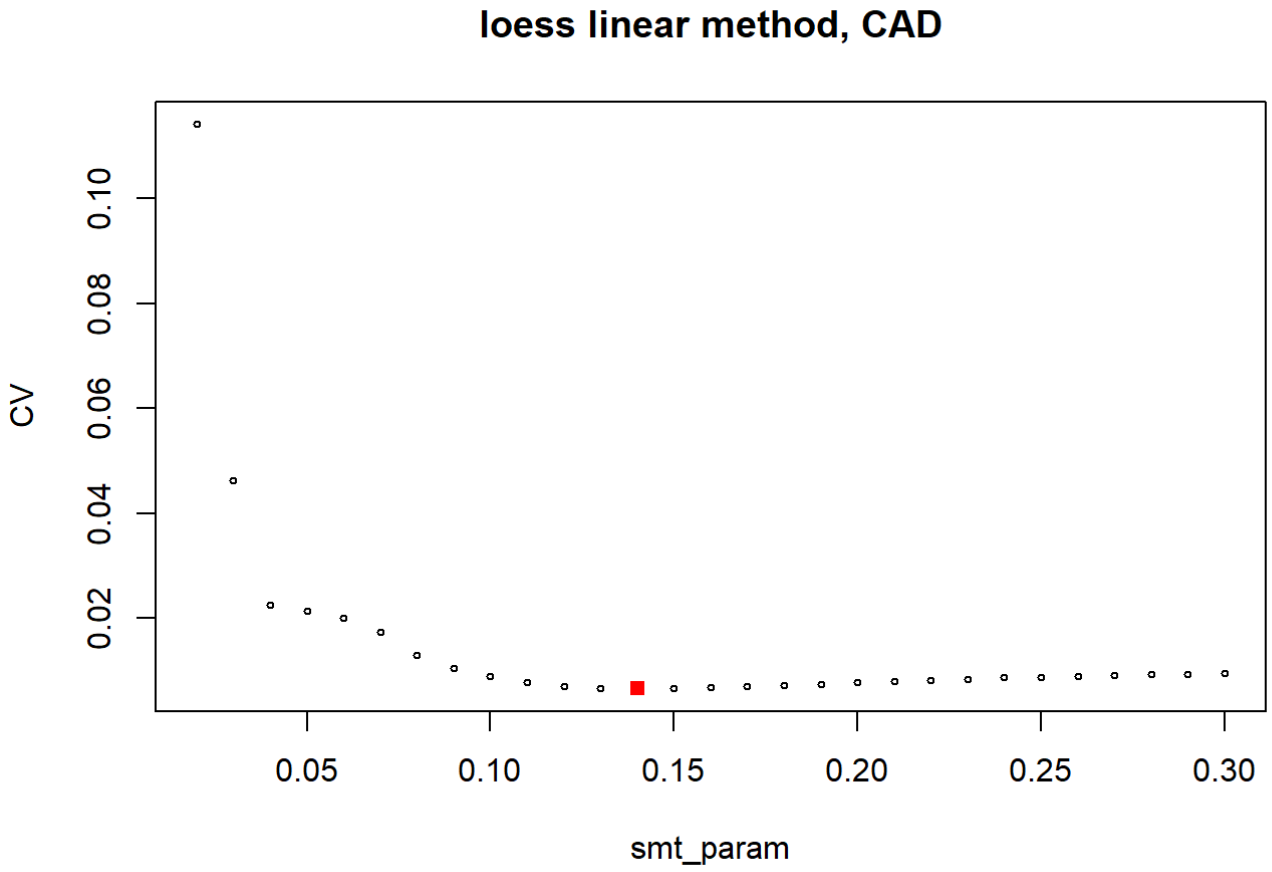

```
pred_err_CHF[5] <- full_test_kfold(estimate_loess_linear, data$X, data$CHF, seq(0.02,0.3,by = 0.01), k = 10, desc
ription = "loess linear method, CHF")
```

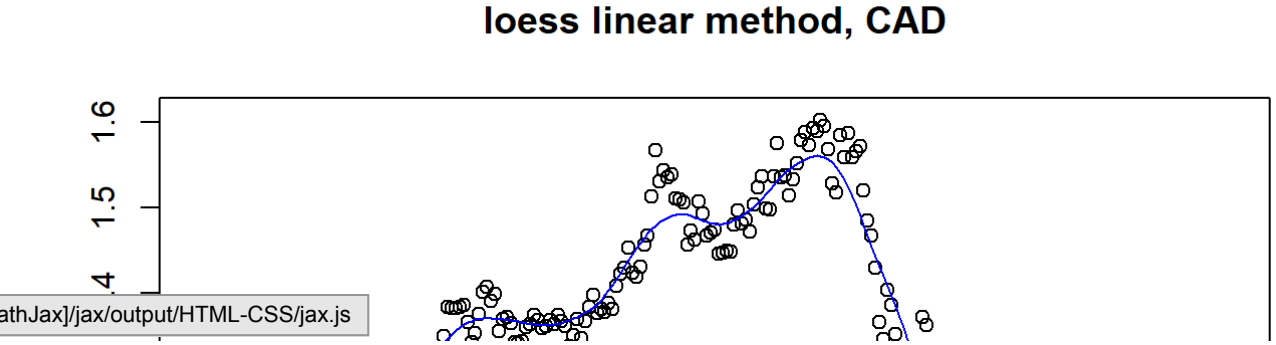


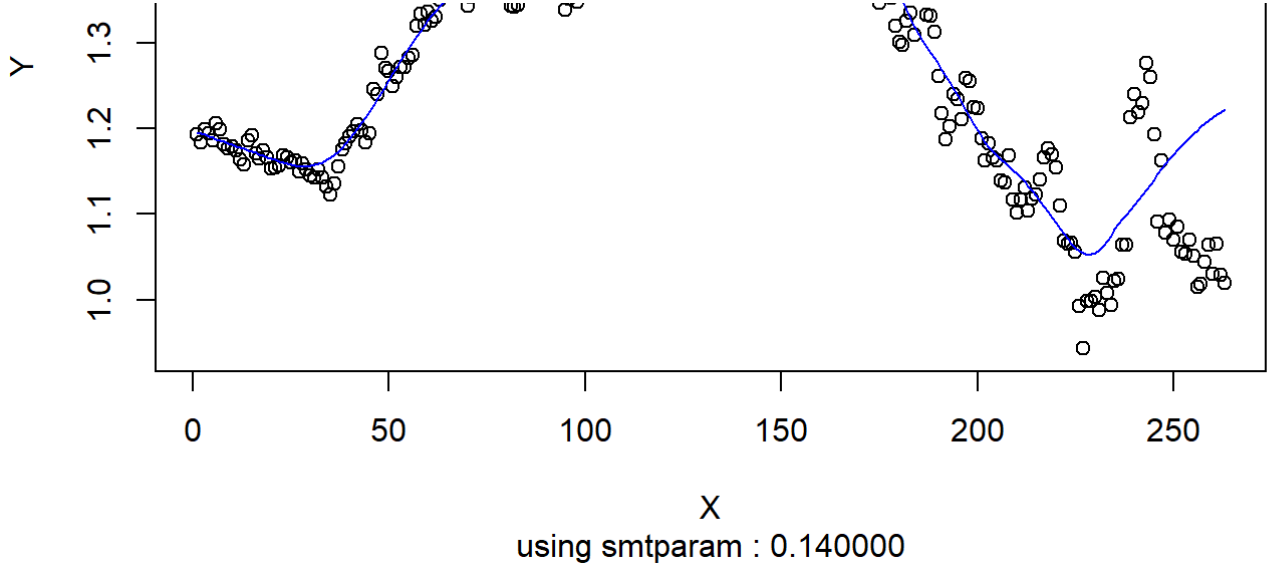





```
pred_err_CAD[5] <- full_test_kfold(estimate_loess_linear, data$X, data$CAD, seq(0.02,0.3,by = 0.01), k = 10, desc
ription = "loess linear method, CAD")
```

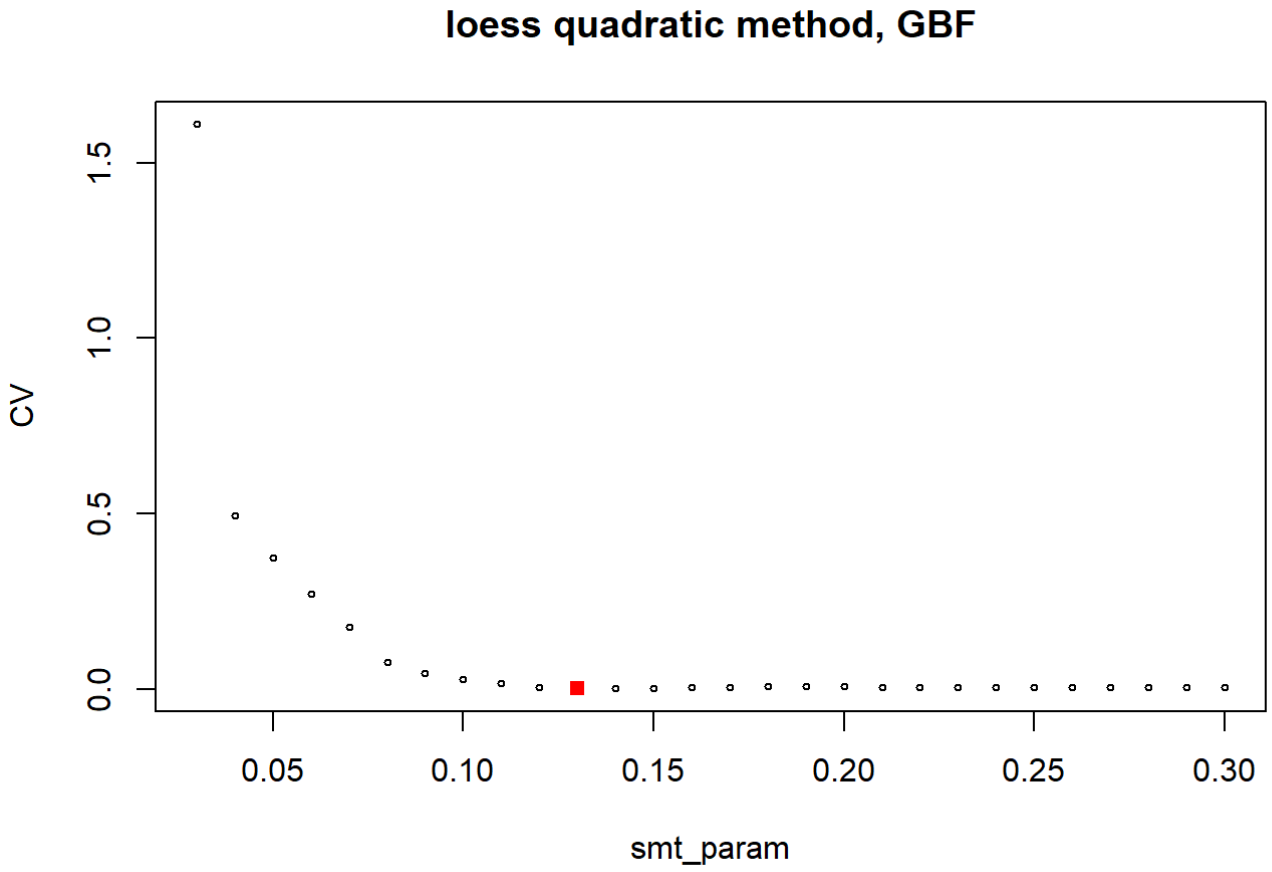


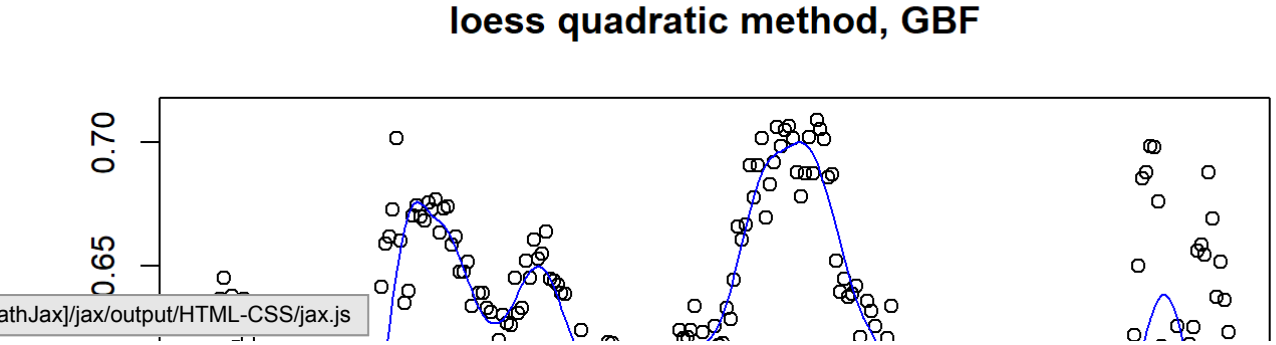


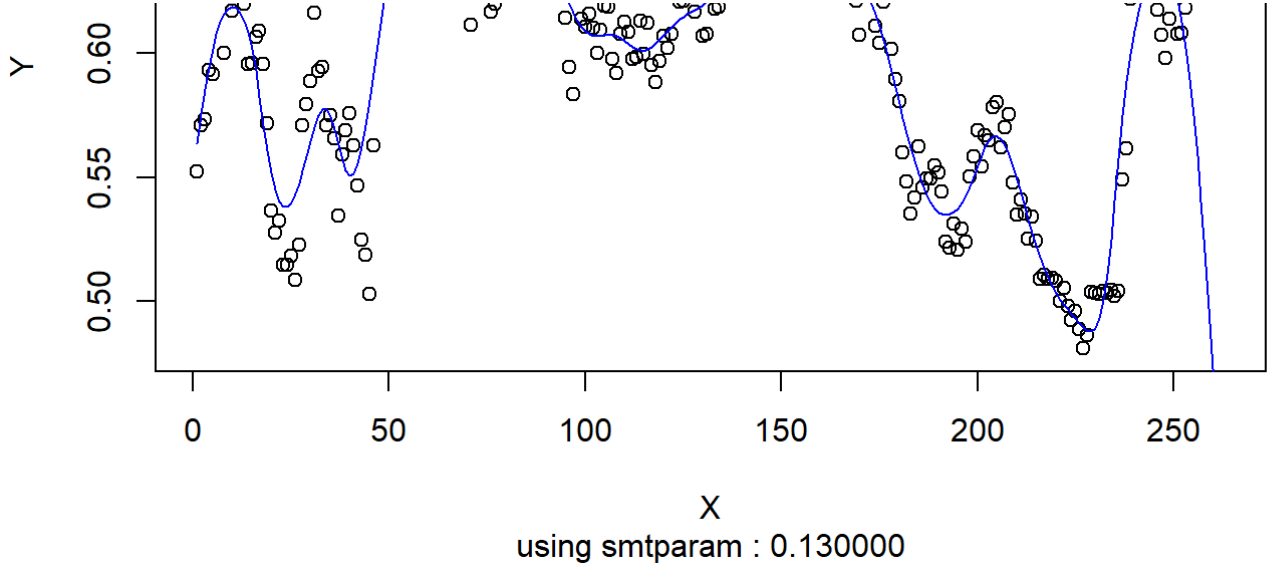


다음은 quadratic case이다. span으로 0.03~0.3까지 0.01간격으로 테스트를 해보겠다.

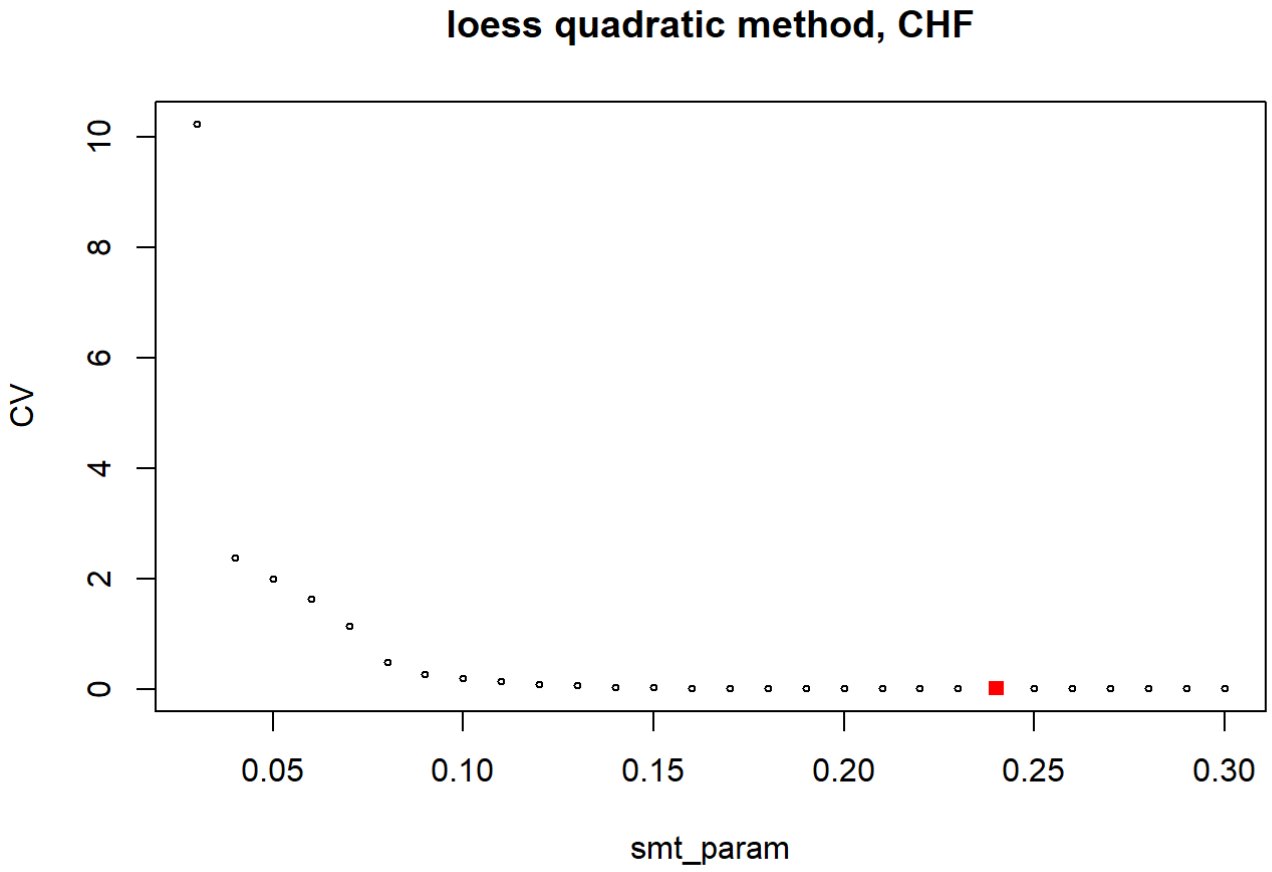
```
pred_err_GBP[6] <- full_test_kfold(estimate_loess_quad , data$X, data$GBP, seq(0.03,0.3,by = 0.01), k = 10, descr
iption = "loess quadratic method, GBF")
```

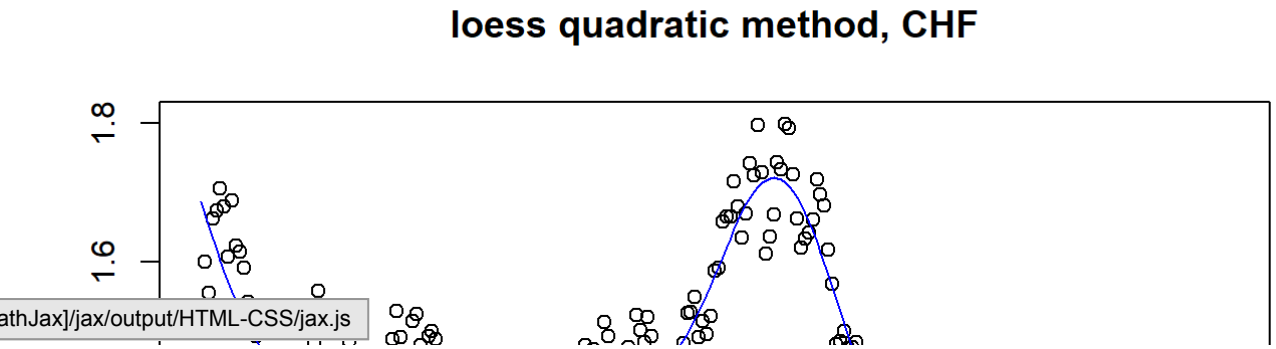


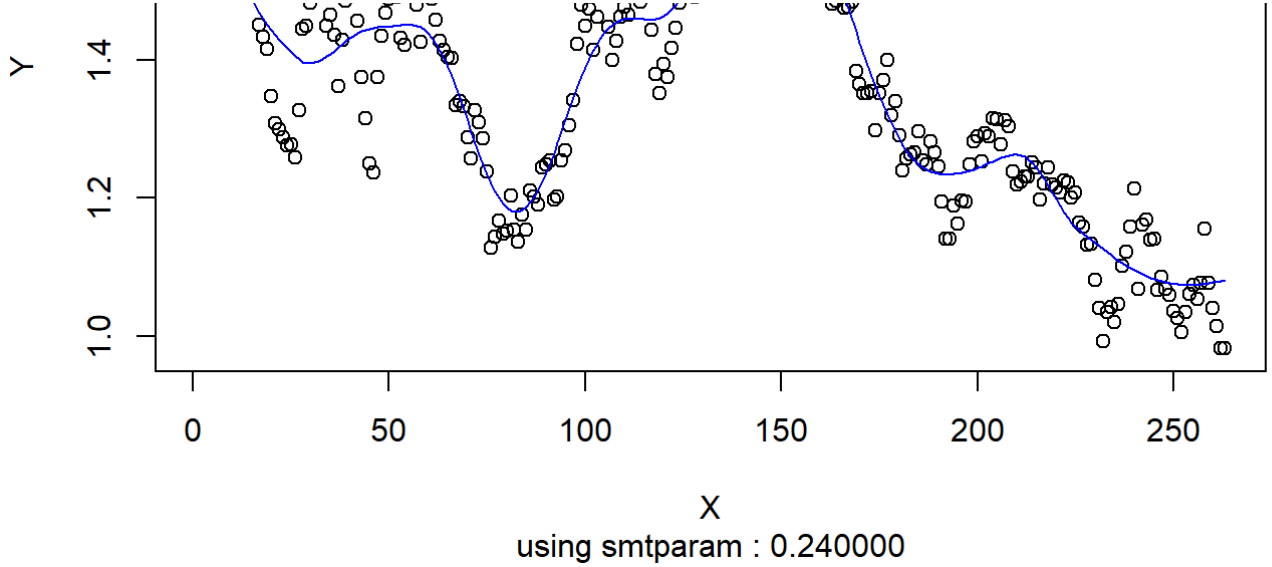




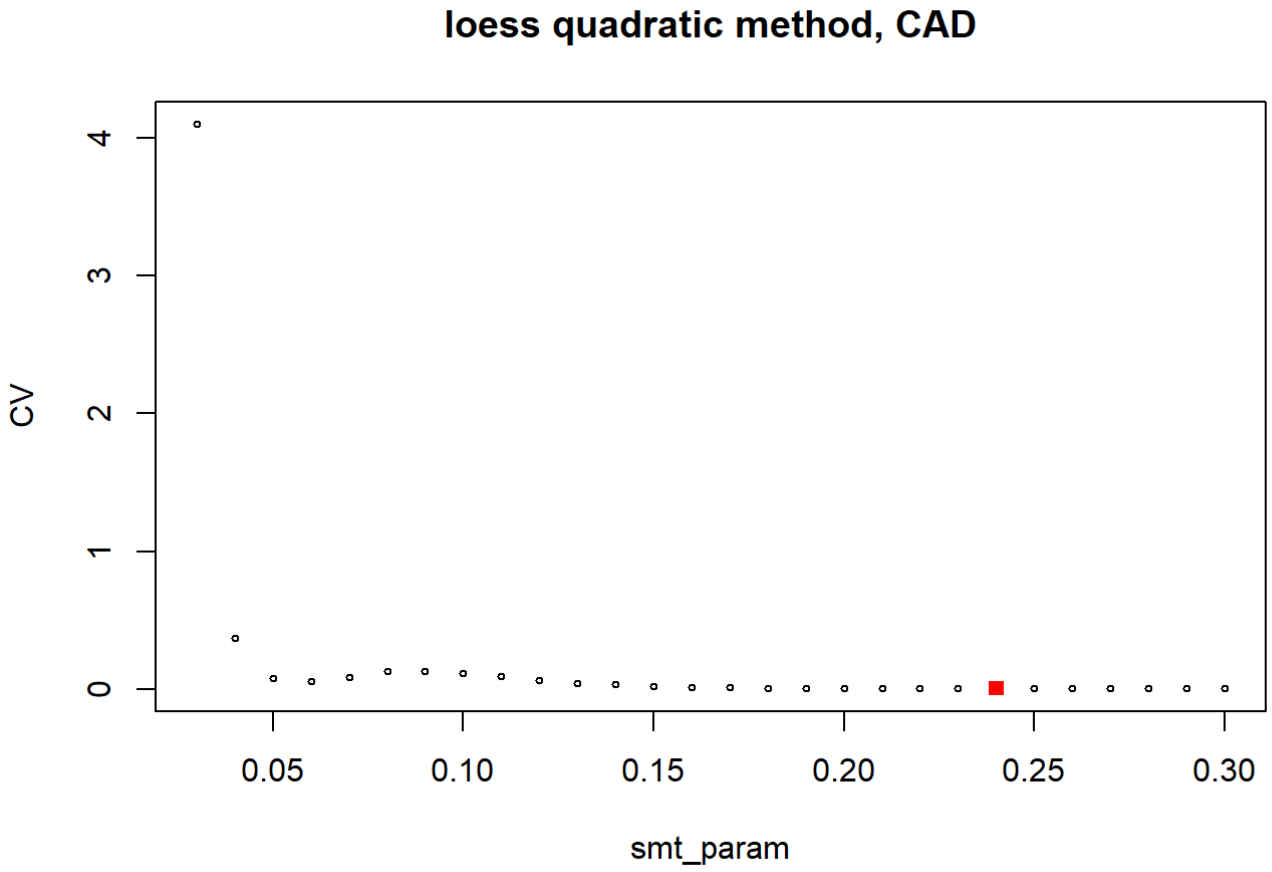

```
pred_err_CHF[6] <- full_test_kfold(estimate_loess_quad, data$X, data$CHF, seq(0.03,0.3,by = 0.01), k = 10, descri
ption = "loess quadratic method, CHF")
```

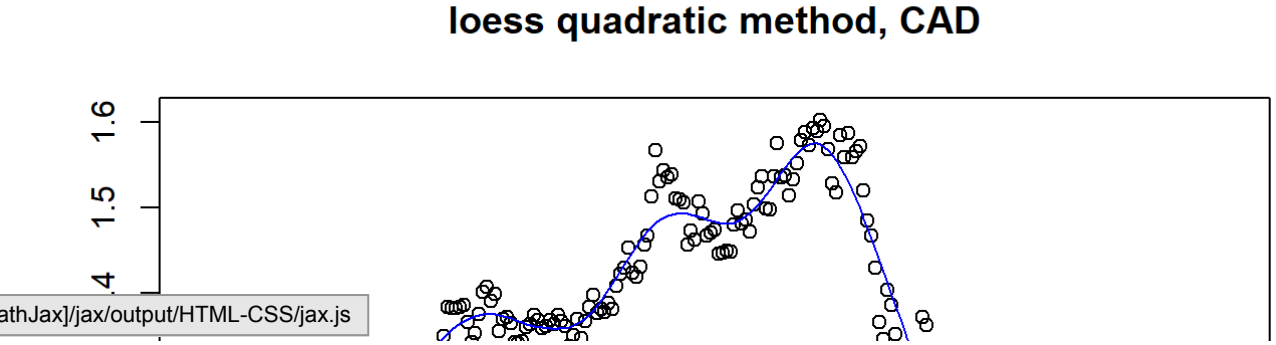


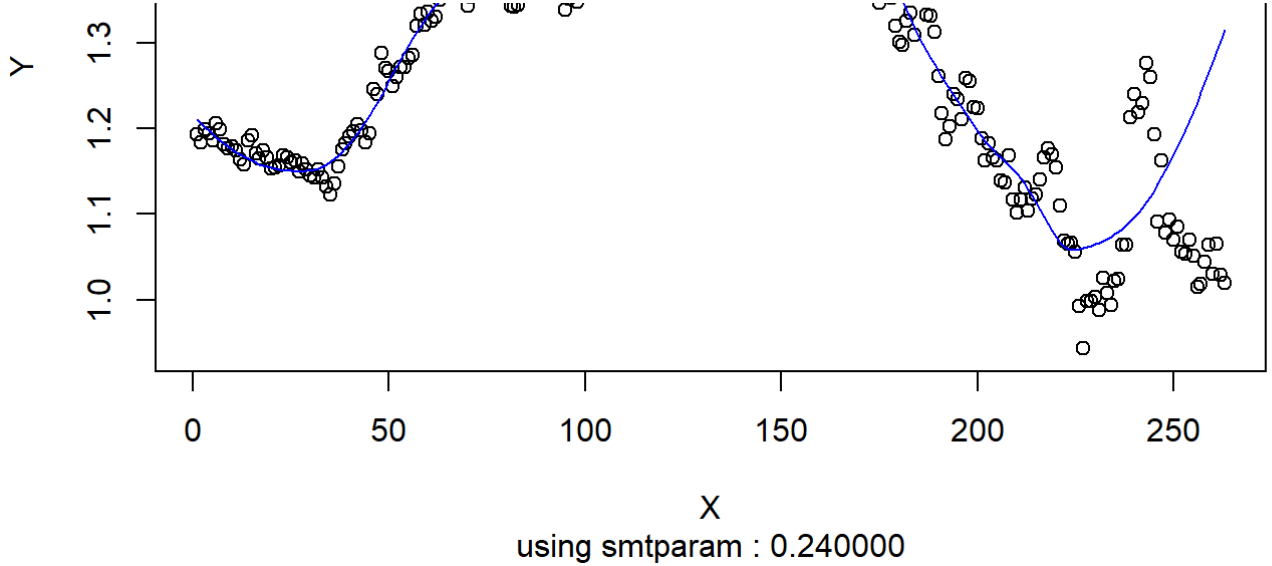





```
pred_err_CAD[6] <- full_test_kfold(estimate_loess_quad, data$X, data$CAD, seq(0.03,0.3,by = 0.01), k = 10, descri
ption = "loess quadratic method, CAD")
```







이제, 가장 좋은 method를 찾기위해 앞에서 만든 pred_err 벡터들의 값중 어떤 값이 가장 작은지를 볼것이다.

1 : nadaraya-watson 2 : local polynomial (linear) 3 : smooting spline 4 : natural spline with binomial filter 5 : loess (linear) 6 : loess (quadratic)

```
which(pred_err_GBP == min(pred_err_GBP, na.rm = TRUE))
```

```
## [1] 1
```

```
which(pred_err_CHF == min(pred_err_CHF, na.rm = TRUE))
```

```
## [1] 2
```

```
which(pred_err_CAD == min(pred_err_CAD, na.rm = TRUE))
```

```
## [1] 4
```

각각 GBP, CHF, CAD에 대해 어떤것이 best method인지를 위와 같이 알아볼 수 있었다.

즉, 10-fold cross validation을 실행했을때, prediction err가 가장 작은 method는 각각 위와 같다는것을 알 수 있었다.