# 함수추정 및 실습 HW2

김보창

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

실행할때마다 동일한 결과가 나오도록 set.seed를 통해 시드를 설정해준다.

# Q1

## 1-(a)

smoothing spline을 해주고, 그래프까지 출력해주는 함수를 만든다.

r의 spline 패키지에 있는 smooth.spline함수를 이용하면, spline 할 수 있다.

```
my_func <- function(x)
{
    return(sin(0.1 * pi * x))
}

# n is data set's number, l is lambda
# form of func is func(x), it's return value is function value of x.
# err_sd is sd of error.
draw_spline <- function(x, true_func, err_sd = 0.5, n = 40, lamb = 1e-10)
{
    y <- matrix(nrow = length(x), ncol = n)
    ey <- matrix(nrow = length(x), ncol = n)

    for(i in seq_len(n)) # save ey to columns
    {
        e <- rnorm(length(x), mean = 0, sd = err_sd)
        y[,i] <- true_func(x) + e

        fit.sp <- smooth.spline(x, y[,i], lambda = lamb, all.knots = T)

        ey[,i] <- fitted(fit.sp)
    }

    mean_ey <- rowMeans(ey)
    mean_true <- true_func(x)

    bias_ey <- mean_ey - mean_true

    var_ey <- rowSums((ey - mean_ey)**2) / (n - 1)

    mse_ey <- bias_ey**2 + var_ey

    par(mfrow=c(2,2),cex=.8)
    plot(x, mean_true ,ylim = c(-3,3), xlab = "X", ylab = "Y", type = "l", col = "red")
    for(i in seq_len(n))
    {
        lines(x, ey[,i])
    }
    lines(x, mean_true, col = "red")

    plot(x, bias_ey ,ylim = c(-0.5,0.5), xlab = "X", ylab = "Bias", type = "l")

    plot(x, var_ey ,ylim = c(0,0.5), xlab = "X", ylab = "Var", type = "l")

    plot(x, mse_ey, ylim = c(0,0.5), xlab = "X", ylab = "MSE", type = "l")

}
```

위의 draw_spline 함수는, 다음 값들을 받아 figure 3.2와 같은 그래프를 그려준다.

y = true_func(x) + e 의 모델을 사용한다.

x : x들의 값. true_func : x값들을 이용하여 y값을 만들어주는 함수. err_sd : e가 따르는 normal distribution의 sd. (mean은 0이다.) 기본값은 0.5 n : graph를 그릴때 추정값을 만들 개수. 기본값은 40. lamb : smoothing spline에 사용할 lambda. 기본값은 $10^{-10}$.

my_func는 단순히 \(sin(0.1 * \pi * X)\)의 값을 리턴하는 함수이다.

위 함수를 이용하면 간편하게 그래프를 만들 수 있다.

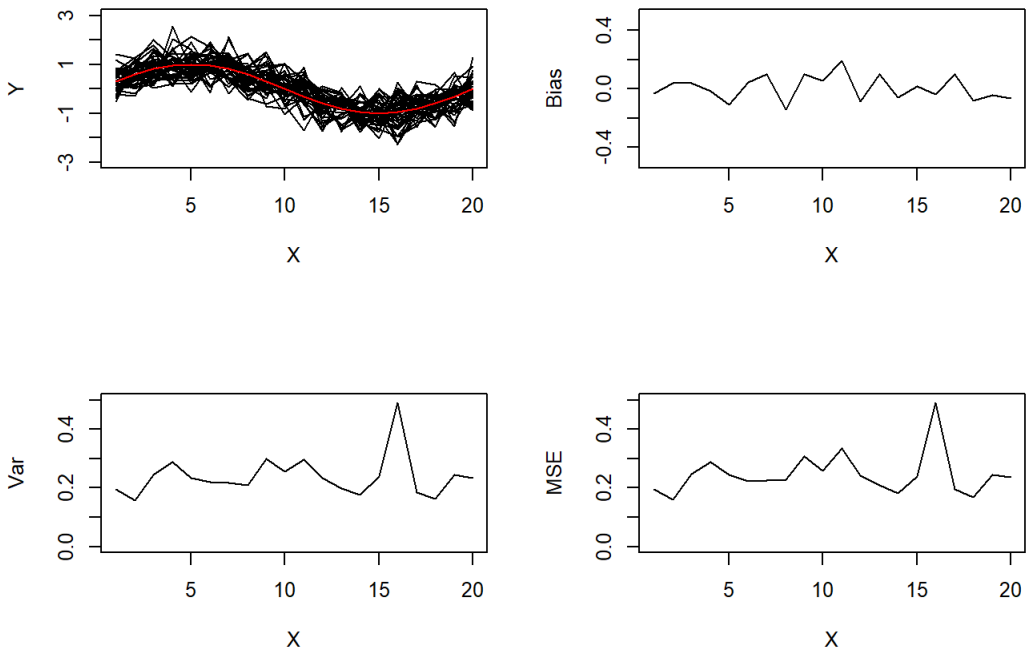
# 1-(b)

smoothing spline의 smoothing parameter는 lambda이므로, 이 값을 바꿔가면서 그래프를 출력해 보겠다.

lambda 값으로는 각각 \(10^{-10}\), \(10^{-3}\), \(10^{-1}\)을 사용하도록 하겠다.

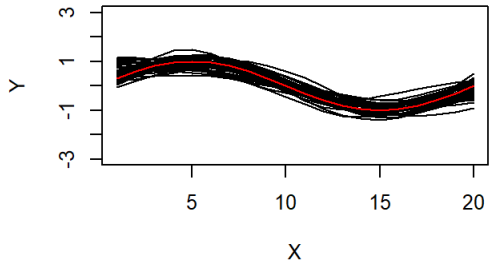
위에서 만든 함수를 이용하여 다음과 같이 그래프를 출력할 수 있다.

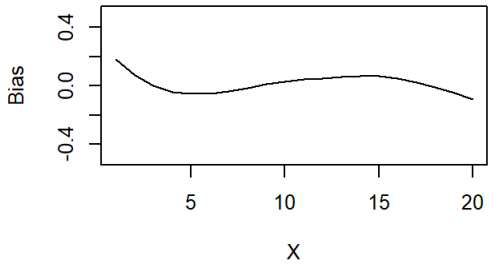
```
x <- seq(1,20) # 1,2,...20
draw_spline(x, my_func, lamb = 1e-10)
```

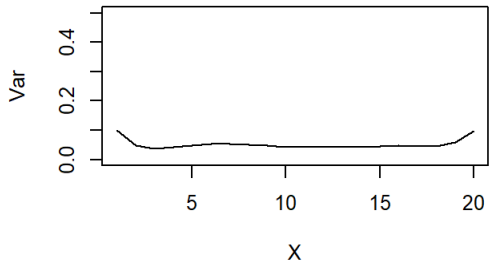


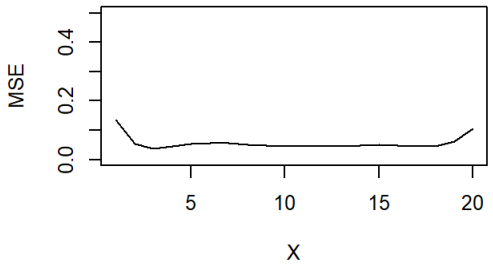

위는 lambda로 \(10^{-10}\)을 사용했을때의 그래프다.

```
draw_spline(x, my_func, lamb = 1e-3)
```



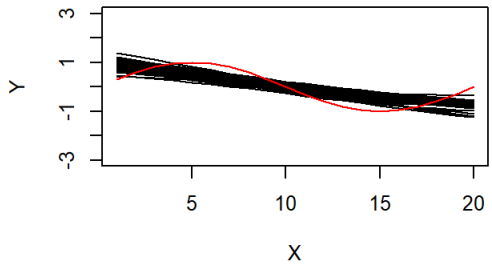



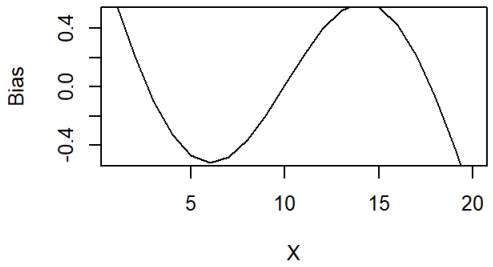



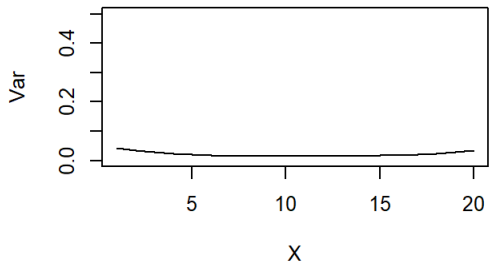



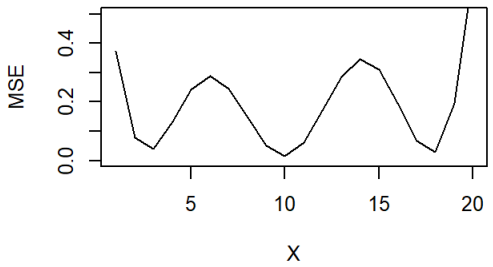

위는 lambda로 \(10^{-3}\)를 사용했을때,

```
draw_spline(x, my_func, lamb = 1e-1)
```




위는 lambda로 \(10^{-1}\)을 사용했을때의 그래프다.

smoothing parameter를 크게 가져갈수록, 즉, smoothing을 더 많이 할수록 전체적인 bias의 절댓값의 폭이 더 커지고, variance는 점점 더 작아짐을 알 수 있다.

이는 smoothing이 많이 될수록, 여러 dataset간에 추정된 값의 차이가 점점 작아지므로 variance가 감소하는것이고,

추정된 값이 smoothing이 많이 될 수록 추정에 사용된 원래 데이터와는 더 멀어지기 때문에 bias의 절댓값이 커짐을 알 수 있다.

또한, MSE의 경우, MSE는 bias의 제곱과 variance의 합이기 때문에, 이 값들의 경향을 따라가는것을 볼 수 있다. 따라서smoothing parameter에 따라 MSE가 무조건적으로 증가한다고도, 감소한다고도 말할수는 없다.

또한, smoothing parameter가 커질수록 양 끝점에서의 (즉, x가 1과 20)일때의 bias가 급격히 증가하고, variance는 다른 점에서의 variance보다 커짐을 확인할 수 있다.

이러한 이유는, smooting spline은 \[$E_{s s}=\sum_{i=1}^{n}\left(Y_{i}-m\left(X_{i}\right)\right)^{2}+\lambda \int_{-\infty}^{\infty}\left(m^{(2)}(x)\right)^{2} d x\] 을 최소화 하는 spline을 선택하게 되는데,

이는 smooting parameter인 lambda가 커질수록, \(m^{(2)}(x)\)를 최소화 하려고 하게 되고, 이것이 최소화되려면 추정된 spline이 선형함수에 가까워 져야 하기 때문에, 데이터는 원 함수인 sine 함수에, 여기에 랜덤한 오차를 더해 데이터가 생성되는데, spline은 선형함수에 가까워 지므로, 양 끝점에서의 값이 원 함수와 차이가 커지게 되고, 따라서 양 끝점의 bias가 크게 증가하게 된다.

또한, lambda가 커질수록 양 끝점의 variance 역시 다른 점에 비해 증가하게 되는데 (lambda가 같은 데이터 내부에서다. lambda가 다를때와 비교한것이 아님), 이 역시 마찬가지로, lambda가 클수록 data를 이용해서 추정한 함수가 선형함수의 형태가 되고, 선형함수의 기울기에 의한 영향이 가장 크게 나타나는 점은 양 끝점이 되므로, 각각의 data set에 의해 추정된 선형함수 형태의 함수의 영향이 극대화 되는 양 끝점에서의 variance가 다른 점에서의 variance보다 크게 되는 것이다.

따라서, MSE역시 이러한 bias와 variance의 영향을 받아, lambda가 커질수록 양 끝에서의 MSE가 상대적으로 다른 점의 MSE보다 증가하는 경향을 보이게 된다.

# Q2

## 2-(a)

(A)에 해당되는 코드는 다음과 같다. (책 197 페이지)

```
function()
{
# (1)
    d1 <- read.table("c:\\datasets\\test31.csv", sep = ",")
    xx <- d1[, 1.]
    yy <- d1[, 2.]
# (2)
    nd <- length(xx)
# (3)
    bw <- 0.18
# (4)
    ex <- seq(from = 0.1, to = 3., by = 0.1)
# (5)
    bwsplus <- bw/0.3708159
    fit.ks <- ksmooth(xx, yy, kernel = "normal", bandwidth = bwsplus, x.points = ex)
# (6)
    ey <- fit.ks$y
# (7)
    par(mfrow = c(1.,1.), mai = c(2., 3., 2., 3.),
        oma = c(2., 2., 2., 2.))
    par(mfrow = c(1., 1.), mai = c(2., 3., 2., 3.),
        oma = c(2., 2., 2., 2.))
    plot(xx, yy, type = "n", xlab = "x", ylab = "y")
    points(xx, yy, pch = 5., cex = 0.8)
    lines(ex, ey)
}
```

(eval = FALSE로, 실행은 되지 않는 코드임)

이를 적절히 고쳐서, 1-(a)에서와 같이 그래프를 출력하도록 바꿔보겠다.

앞의 함수부분에서 estimate 하는 함수 부분만 적절히 고쳐주면 된다.

```
draw_nadaraya <- function(x, true_func, err_sd = 0.5, n = 40, bw = 0.18)
{
    y <- matrix(nrow = length(x), ncol = n)
    ey <- matrix(nrow = length(x), ncol = n)
    bwsplus <- bw/0.3708159

    for(i in seq_len(n)) # save ey to columns
    {
        e <- rnorm(length(x), mean = 0, sd = err_sd)
        y[,i] <- true_func(x) + e

        fit.ks <- ksmooth(x, y[,i], kernel = "normal", bandwidth = bwsplus, x.points = x)

        ey[,i] <- fit.ks$y
    }

    mean_ey <- rowMeans(ey)
    mean_true <- true_func(x)

    bias_ey <- mean_ey - mean_true

    var_ey <- rowSums((ey - mean_ey)**2) / (n - 1)

    mse_ey <- bias_ey**2 + var_ey

    par(mfrow=c(2,2),cex=.8)
    plot(x, mean_true ,ylim = c(-3,3), xlab = "X", ylab = "Y", type = "l", col = "red")
    for(i in seq_len(n))
    {
        lines(x, ey[,i])
    }
    lines(x, mean_true, col = "red")

    plot(x, bias_ey ,ylim = c(-0.5,0.5), xlab = "X", ylab = "Bias", type = "l")

    plot(x, var_ey ,ylim = c(0,0.5), xlab = "X", ylab = "Var", type = "l")

    plot(x, mse_ey, ylim = c(0,0.5), xlab = "X", ylab = "MSE", type = "l")

}
```

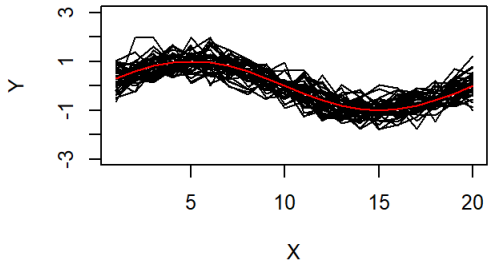
앞의 draw_smooth 함수 부분에서 spline.smooth 부분만 nadaraya-watson estimator를 사용하도록 바꾸어 주었다.

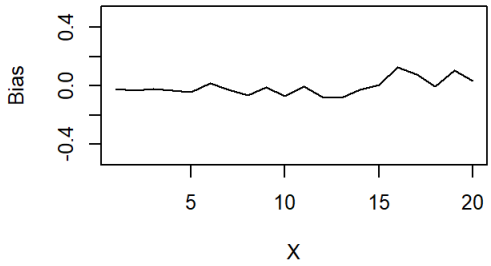
## 2-(b)

1-(b)에서와 같이, 사용하는 true function과 x값들도 같으므로,(x = 1,2…20, true_function = \(sin(0.1 * \pi * X)\)) 이를 그대로 이용해서 그래프를 그려보겠다.

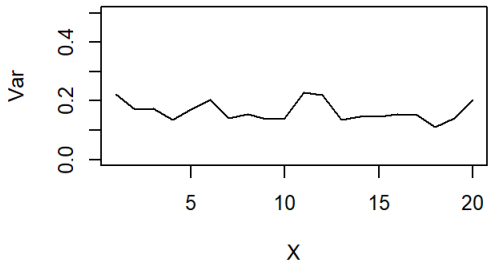
bandwidth 값으로는 0.5, 1, 3을 이용하도록 하겠다.

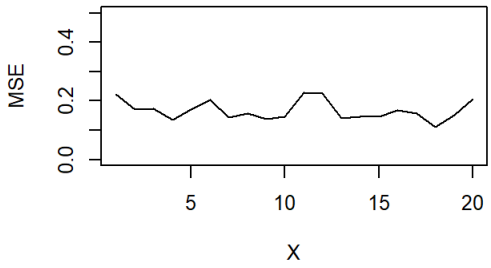
```
draw_nadaraya(x, my_func, bw = 0.5)
```



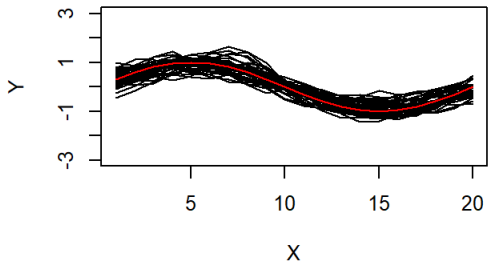

위 그래프는 bandwidth = 0.5

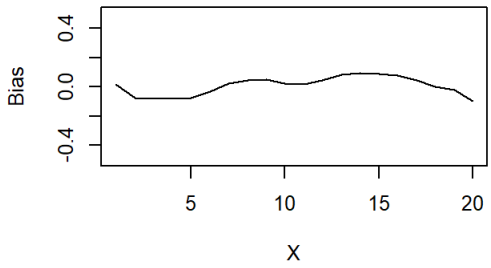


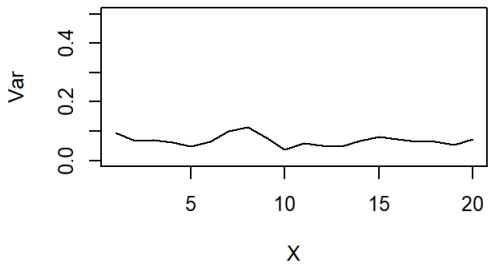

에 해당하는 그래프.

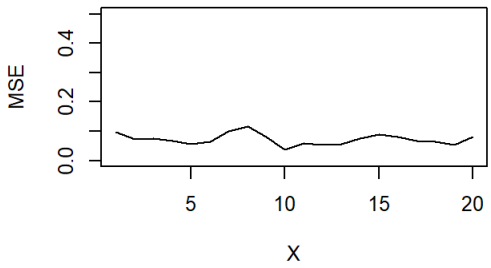
```
draw_nadaraya(x, my_func, bw = 1)
```



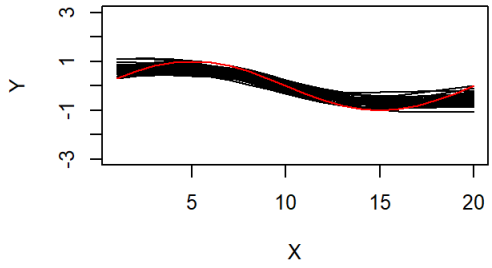

위 그래프는 bandwidth = 1에

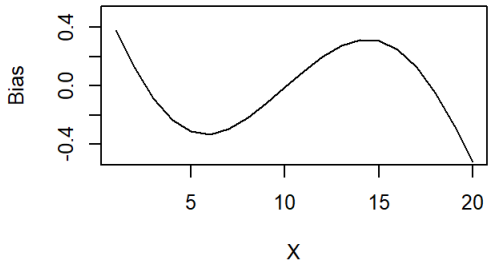


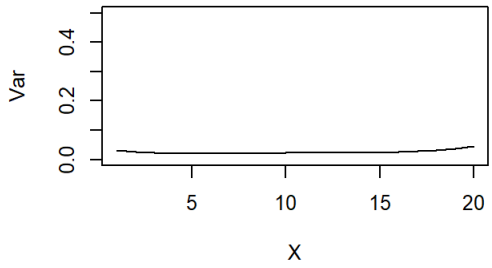

해당하는 그래프.

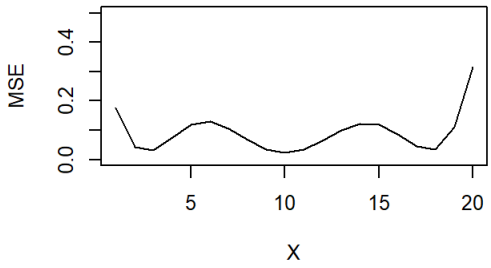
```
draw_nadaraya(x, my_func, bw = 3)
```




위 그래프는 bandwidth = 3에




해당하는 그래프.

그래프들을 모두 살펴보면, smoothing parameter인 bandwidth가 증가할수록 전체적으로 variance는 감소하고, bias의 절댓값은 증가하는 경향이 보임을 알 수 있는데,

이는 smoothing이 많이 될수록, 여러 dataset간에 추정된 값의 차이가 점점 작아지므로 variance가 감소하는것이고,

추정된 값이 smoothing이 많이 될 수록 추정에 사용된 원래 데이터와는 더 멀어지기 때문에 bias의 절댓값이 커짐을 알 수 있다.

또한, MSE의 경우, MSE는 bias의 제곱과 variance의 합이기 때문에, 이 값들의 경향을 따라가는것을 볼 수 있다. 따라서 smoothing parameter에 따라 MSE가 무조건적으로 증가한다고도, 감소한다고도 말할수는 없다.

또한, 앞의 1번과 같이, smoothing parameter가 커질수록 양 끝점에서의 (즉, x가 1과 20)일때의 bias가 급격히 증가하고, variance는 다른 점에서의 variance보다 커짐을 확인할 수 있다.

이러한 이유는, nadaraya-watson estimator는 kernel 함수를 사용하여 다음과 같이 추정값을 구하는데,

$$\hat{m}(x)=\frac{\sum_{i=1}^{n} K\left(\frac{x-x_{i}}{h}\right) Y_{i}}{\sum_{i=1}^{n} K\left(\frac{x-X_{i}}{h}\right)}=\sum_{i=1}^{n} W_{i}(x) Y_{i}$$ 커널 함수는 $\frac{|x - X_i|}{h}$ 이 값이 0에 가까울수록 큰 값을 가지는 특성을 가지게 된다. 또한, 0에서 멀어질수록 normal distribution pdf의 특성때문에 급격하게 값이 감소하게 된다.

따라서, bandwidth인 h가 커질수록, 추정할 x값 근처에서 더 넓은 범위의 데이터의 영향을 받아 추정량을 구하게 된다.

즉, bandwidth가 클수록 smoothing 하는 경향이 더 강해지고, 따라서 bandwidth가 커질수록 추정된 함수는 선형함수에 가까워 지게 된다.

이때 데이터는 원 함수인 sine 함수에, 여기에 랜덤한 오차를 더해 데이터가 생성되는데, 추정된 함수는 선형함수에 가까워 지므로, 양 끝점에서의 값이 원 함수와 차이가 커지게 되고, 따라서 양 끝점의 bias가 크게 증가하게 된다.

또한, bandwidth가 커질수록 양 끝점의 variance 역시 bandwidth 가 같을때, 다른 점에 비해 증가하게 되는데,
이 역시 마찬가지로, bandwidth가 클수록 data를 이용해서 추정한 함수가 선형함수에 가까운 형태가 되고, 선형함수의 기울기에 의한 영향이 가장 크게 나타나는 점은 양 끝점이 되므로, 각각의 data set에 의해 추정된 선형함수 형태의 함수의 영향이 극대화 되는 양 끝점에서의 variance가 다른 점에서의 variance보다 크게 되는 것이다. 즉, 상대적인 크기가 커지게 되는 것이다.

따라서, MSE역시 이러한 bias와 variance의 영향을 받아, bandwidth가 커질수록 양 끝에서의 MSE가 상대적으로 다른 점의 MSE보다 증가하는 경향을 보이게 된다.

# 2-(c)

(B)에 해당되는 코드를 실행되도록 약간 고친 코드는 다음과 같다. (책 199 페이지) (return으로 두값을 리턴하는것은 불가능함)

```
func_B <- function(x1, y1, nd, bandw, ntrial)
{
#   (1)
    kscvgcv <- function(bw, x1 ,y1)
    {
#   (2)
        nd <- length(x1)
        bwsplus <- bw/0.3708159
        fit.ks <- ksmooth(x1, y1, "normal", bandwidth = bwsplus, x.points = x1)
        res <- y1 - fit.ks$y
#   (3)
        dhat1 <- function(x2, bw)
        {
            nd2 <- length(x2)
            diag1 <- diag(nd2)
            bwsplus <- bw / 0.3708159
            dhat <- rep(0, length = nd2)
#   (4)
            for(jj in 1.:nd2) {
                y2 <- diag1[, jj]
                fit.ks <- ksmooth(x2, y2, "normal", bandwidth = bwsplus, x.points = x2)
                dhat[jj] <- fit.ks$y[jj]
            }
            return(dhat)
        }
#   (5)
        dhat <- dhat1(x1, bw)
        trhat <- sum(dhat)
        sse <- sum(res^2.)
#   (6)
        cv <- sum((res/(1. - dhat))^2.)/nd
        gcv <- sse/(nd * (1. - (trhat/nd))^2.)
#   (7)
        return(c(cv, gcv))
    }
#   (8)
    cvgcv <- lapply(as.list(bandw), kscvgcv, x1 = x1, y1 = y1)
    cvgcv <- unlist(cvgcv)
    cv <- cvgcv[seq(from = 1, to = 2*length(bandw), by = 2)]
    gcv <- cvgcv[seq(from = 2, to = 2*length(bandw), by = 2)]
#   (9)
    retval <- list()
    retval$cv <- cv
    retval$gcv <- gcv
    return(retval)
}
```

위 코드는 x와 y값, 그리고 bandwidth list를 받아서, 각 bandwidth에 해당하는 cv와 gcv값을 리턴해주는 함수다. (nd와 ntrial은 사용되지 않는 값임.)

이 함수를 이용해서 가장 좋은 bandwidth를 받아, 그 bandwidth에 해당하는 그래프를 그려보도록 하겠다.

먼저, 가장 낮은 cv와 gcv값을 리턴하는 bandwidth를 구하기 위해, 위 함수를 그대로 이용하여 각 bandwidth당 cv와 gcv값들을 얻어낸다.

임의의 데이터셋 하나를 이용하여 구해보자.

```
bandw = seq(from = 0.1, to = 10, by = 0.01)
x = seq(1,20)
y = my_func(x) + rnorm(20,sd = 0.5)
ret <- func_B(x, y, length(x), bandw, 0)
```

이제, 이렇게 구한 값들중 cv와 gcv가 가장 작은 값들의 index를 얻어오면

```
cv_idx <- which(ret$cv == min(ret$cv, na.rm = TRUE))
gcv_idx <- which(ret$gcv == min(ret$gcv, na.rm = TRUE))

min_bw_cv <- bandw[cv_idx]
min_bw_gcv <- bandw[gcv_idx]

min_bw_cv
```

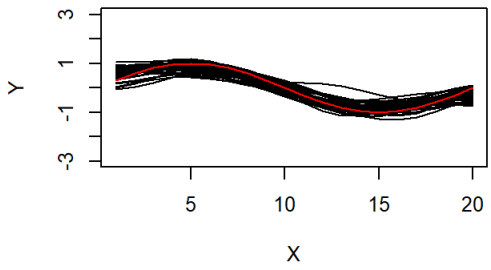
```
## [1] 1.81
```

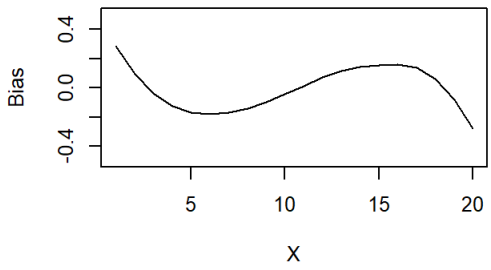
```
min_bw_gcv
```

```
## [1] 1.78
```

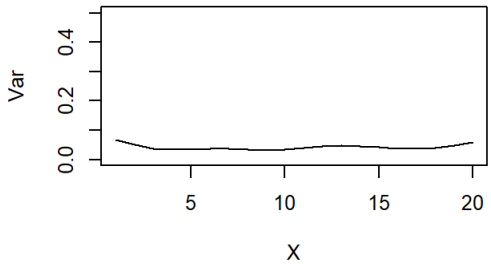
위와 같고, 이 bandwidth를 사용하여 그래프를 그려보자.

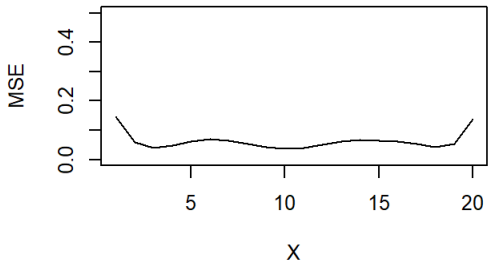
```
draw_nadaraya(x, my_func, bw = min_bw_cv)
```



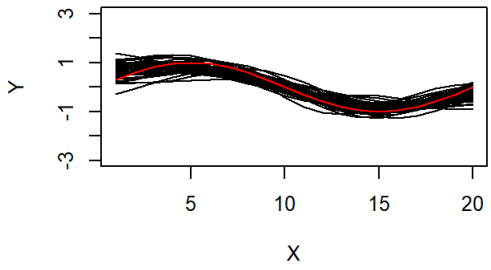

cv가 가장 작은 bandwidth 를

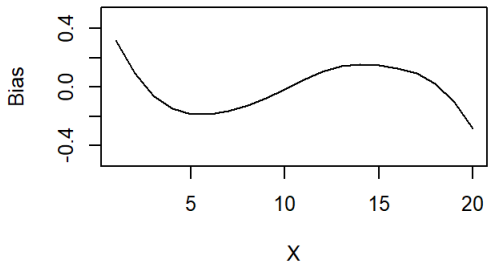


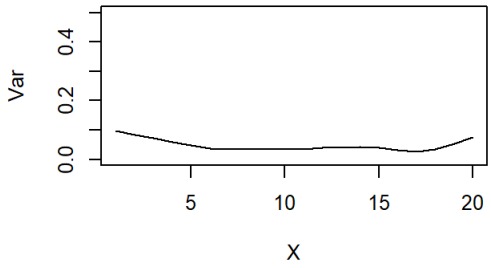

사용했을때

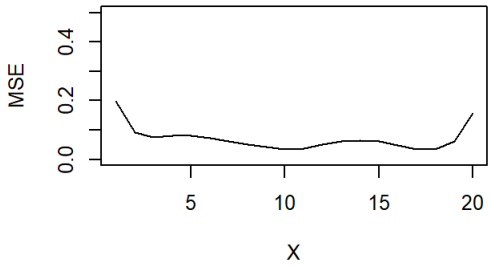
```
draw_nadaraya(x, my_func, bw = min_bw_gcv)
```




gcv가 가장 작은 bandwidth를




사용했을때

각각의 bandwidth를 이용하여 bias, variance, MSE를 구했을때,

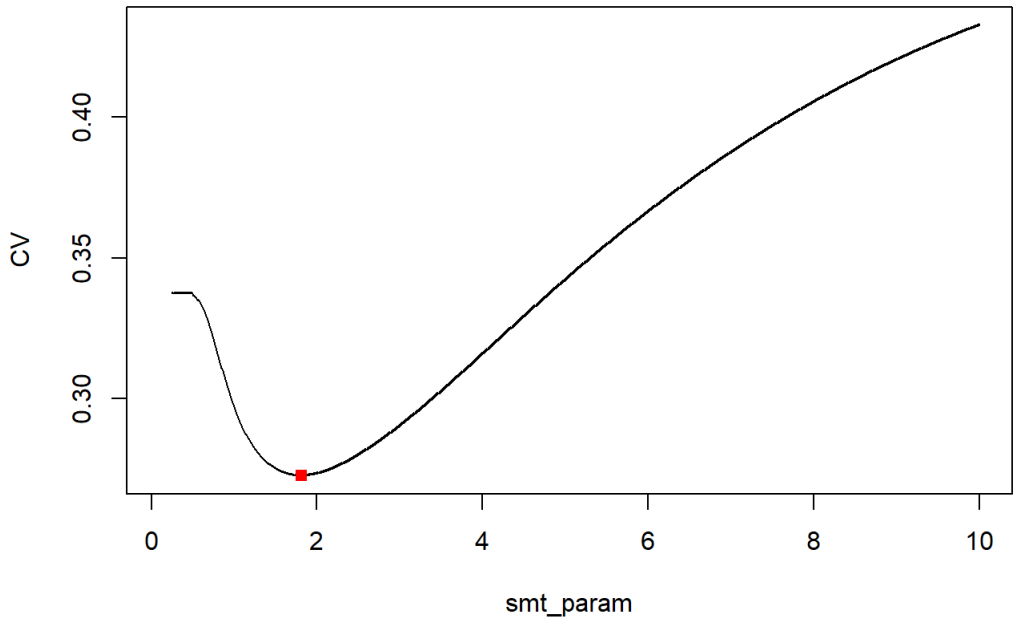
전체적으로 꽤 적은, 괜찮은 MSE의 값이 나옴을 확인할 수 있다.

즉, 충분히 작은 MSE의 값을 뽑아냄을 알 수 있고, 따라서 이를 사용해서 optimal bandwidth를 구해도 될것이다.

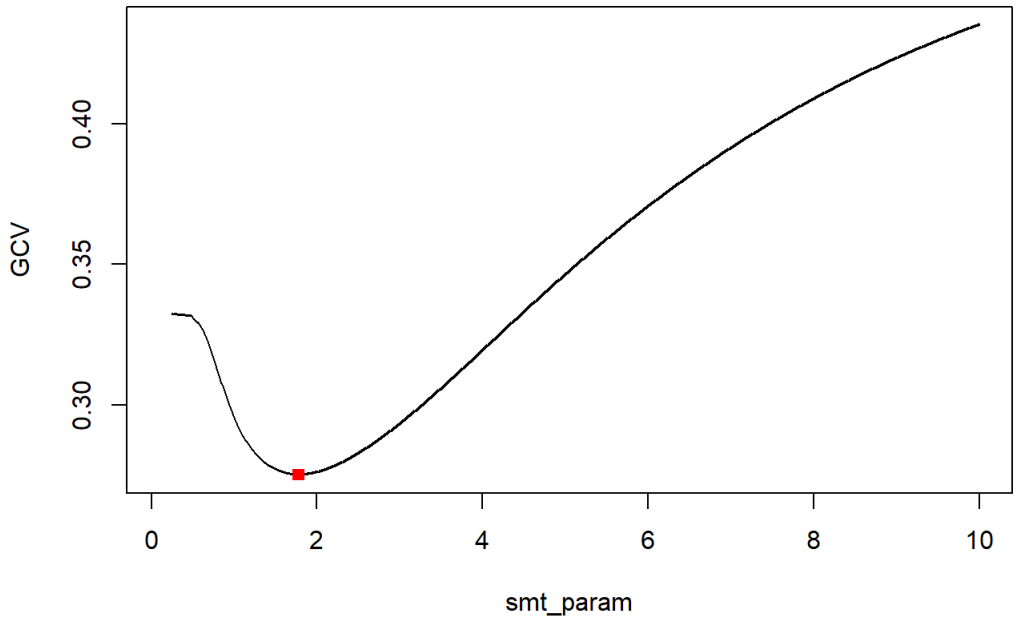
```
plot_cv <- function(smtparam_array, cv_array, description = "")
{
    par(mfrow = c(1, 1))
    plot(smtparam_array, cv_array, type = "n",
     xlab = "smt_param", ylab = description)
    points(smtparam_array, cv_array, pch = 1, cex = 0.1)
    lines(smtparam_array, cv_array, lwd = 1)
    pcvmin <- 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. 이를 이용해서 각 CV와 GCV의 경향을 알아보자.

```
plot_cv(bandw, ret$cv, "CV")
```




```
plot_cv(bandw, ret$gcv, "GCV")
```




위와 같음을 알 수 있다.

위 통계량을 사용해도 괜찮겠지만, 우리는 true \(m(x)\)가 무엇인지 알고 있으므로, MISE'를 best bandwidth를 구하는 척도로 사용할 수 있다.

결국, CV와 GCV는 MISE'의 추정량일 뿐이므로, MISE'을 사용할 수 있다면 더욱 괜찮은 결과를 뽑을 수 있을것이다.

\[MISE'(\hat{m}(x)) = \frac{E[\sum_{i=1}^n(m(X_i) - \hat{m}(X_i))]}{n} \]

위를 각 bandwidth에 대해 구하고, 이 값이 작은 bandwidth를 사용할 것이다.

위 함수를 고쳐서 MISE' 의 정의에 맞게 각 bandwidth에 해당하는 값을 구해보자.

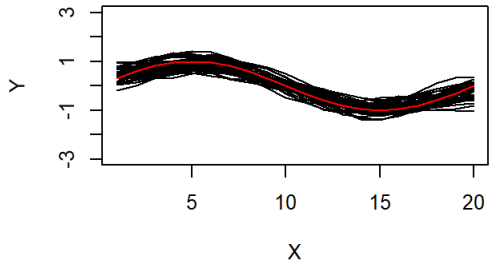
```
get_MISE <- function(x1, y1, nd, bandw)
{
#    (1)
    ksmise <- function(bw, x1 ,y1)
    {
#    (2)
        nd <- length(x1)
        bwsplus <- bw/0.3708159
        fit.ks <- ksmooth(x1, y1, "normal", bandwidth = bwsplus, x.points = x1)
        res <- my_func(x1) - fit.ks$y
#
#    (6)
        mise <- sum(res**2)/nd
#    (7)
        return(mise)
    }
#    (8)
    MISE <- lapply(as.list(bandw), ksmise, x1 = x1, y1 = y1)
    MISE <- unlist(MISE)
#    (9)
    return(MISE)
}
```

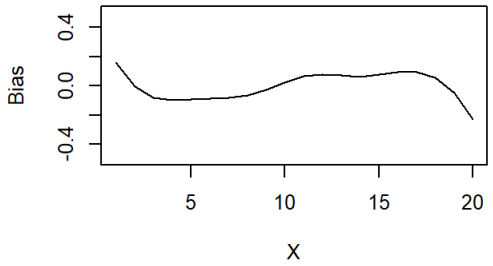
위 함수는 bandwidth에 해당하는 MISE값들을 리턴해준다.

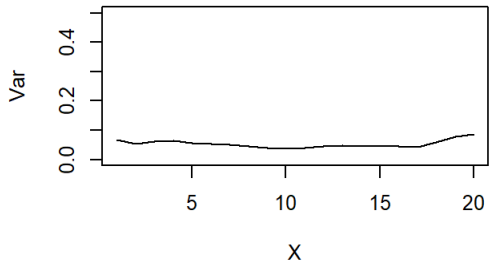
아래와 같이, 앞에서와 마찬가지의 과정으로 최적인 bandwidth를 찾자.

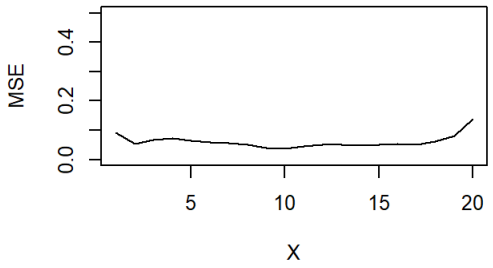
```
bandw = seq(from = 0.1, to = 10, by = 0.01)
mise <- get_MISE(x, y, length(x), bandw)
mise_idx <- which(mise == min(mise, na.rm = TRUE))
min_bw_mise <- bandw[mise_idx]
```

```
draw_nadaraya(x, my_func, bw = min_bw_mise)
```



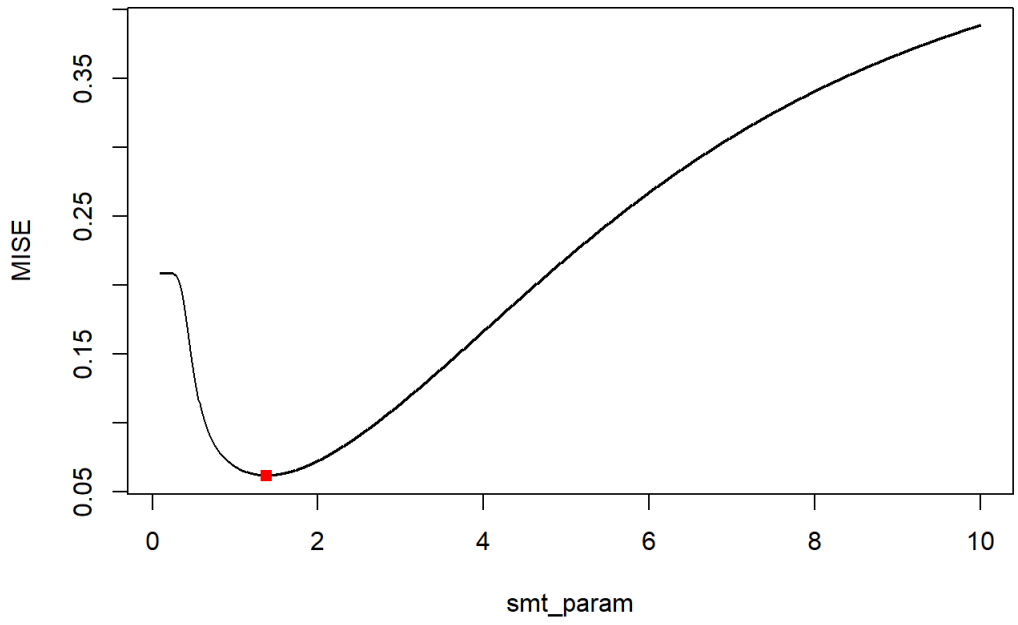




전체적인 MSE의 경향이 매우 작게 유지됨을 알 수 있다.

앞보다도 MSE가 더 작으므로, 최적의 bandwidth가 됨을 알 수 있다.

위에서 MISE의 경향도 다음과 같다.

```
plot_cv(bandw, mise, "MISE")
```




# Q3

\(\widehat{m(x^*)} = q(x^*)^t y = \sum_{i = 1}^n q_j(x^*)Y_j\)꼴로 나타날때, \(q(x^*)\)을 weight diagram vector라 하므로, 이를 plotting 해주는 함수를 만들것이다.

local linear regression에서의 \(q(x^*)^t = e_1^{t} (X^t W X)^{-1} X^t W\) 와 같이 계산되므로 (X와 W는 \(x^*\)에 따라 다르게 생성됨), 이를 이용해서 먼저 \(q(x^*)\)을 구해주는 함수를 만들고 이를 plot하자.

weight function으로는 gaussian kernel을 사용할 것이다.

\(w(\frac{X_i - x^*}{h}) = exp(-\frac{1}{2}(\frac{X_i - x^*}{h})^2)\)

이제, weight diagram vector를 구해주는 함수를 짜면

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

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

}
```

또한, \(\widehat{m(x^*)} = e_1^{t} (X^t W X)^{-1} X^t W y\)이므로,

다시말해서 y로 \(e_1\)~\(e_n\)의 값을 넣으면 \(q(x^*)\)을 구할 수 있게 된다.

따라서, 먼저 local linear regression을 하는 함수를 짜고, 이를 이용하여 weight diagram vector를 리턴하는 함수를 짜자.

```
local_linear <- 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)
}

weight_diag <- function(x_star, xdata, band)
{
    I <- diag(length(xdata))
    wd <- vector(length = length(xdata))
    for(i in 1:length(xdata))
    {
        wd[i] <- local_linear(x_star, xdata, I[,i], band)
    }
    return(wd)
}
```

또한, nadaraya-watson 방법에 의해 weight diagram 벡터를 구하는 함수도 아래와 같이 짤 수 있다.

결국, \(\widehat{m(x^*)} = q(x^*)^t y = \sum_{i = 1}^n q_j(x^*)Y_j\) 일때의 \(q(x^*)\)를 구해주면 되기 때문에, 마찬가지로 \(e_1\).. \(e_n\)을 Y로 사용해서 나다라야-왓슨 방법을 이용해서 fitted된 \(m(x^*)\)값들이 각각 \(q(x^*)_j\)가 되기 때문에, 이를 이용하여 구할 수 있다.

```
weight_diag_nada <- function(x_star, xdata, band)
{
    I <- diag(length(xdata))
    wd <- vector(length = length(xdata))
    bwsplus <- band/0.3708159
    for(i in 1:length(xdata))
    {
        fit.ks <- ksmooth(xdata, I[,i], kernel = "normal", bandwidth = bwsplus, x.points = x_star)
        wd[i] <- fit.ks$y
    }
    return(wd)
}
```

이제, 이 weight_diag 벡터를 각 \(x^* = 1,2, ...20\)까지에 대해 구한다음 ploting 해보자.

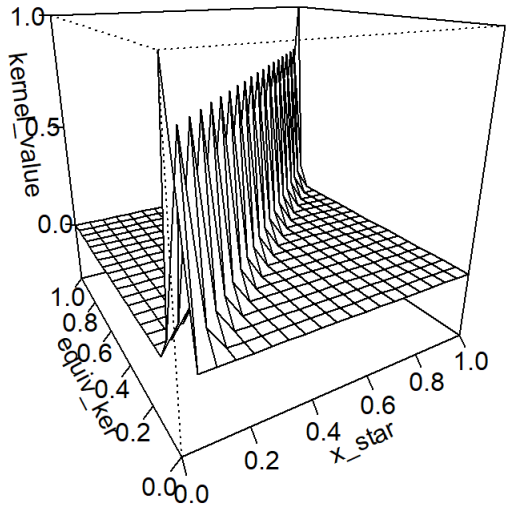
```
plot_weight_diag <- function(band)
{
    xdata <- seq(1,20)

    mat <- lapply(xdata, weight_diag, xdata = xdata, band = band)
    y <- matrix(unlist(mat), nrow = length(xdata), ncol = length(xdata))

    persp(y, zlim = c(-0.3, 1), xlab = "x_star", ylab = "equiv_ker",  zlab = "kernel_value", lab = c(3,3,3)
, theta = -30, phi = 20, ticktype = "detailed")
}

plot_weight_diag_nada <- function(band)
{
    xdata <- seq(1,20)

    mat <- lapply(xdata, weight_diag_nada, xdata = xdata, band = band)
    y <- matrix(unlist(mat), nrow = length(xdata), ncol = length(xdata))

    persp(y, zlim = c(-0.3, 1), xlab = "x_star", ylab = "equiv_ker",  zlab = "kernel_value", lab = c(3,3,3)
, theta = -30, phi = 20, ticktype = "detailed")
}
```

위 함수들을 이용하여, 각 bandwidth에 대한 weight diagram vector를 출력해볼 수 있다.

bandwidth로는 0.5, 1, 3을 사용하도록 하겠다.

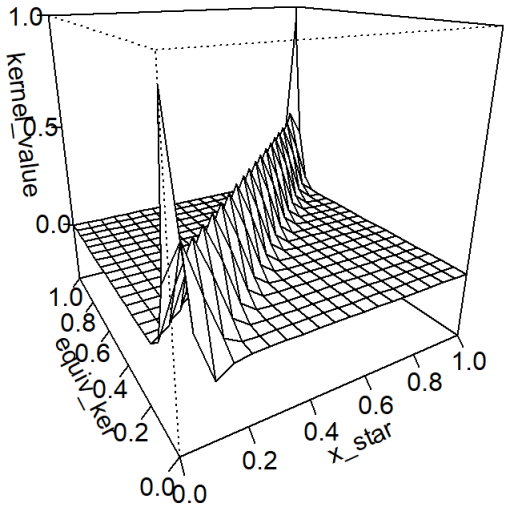
먼저, local linear regression이다.

```
plot_weight_diag(0.5)
```



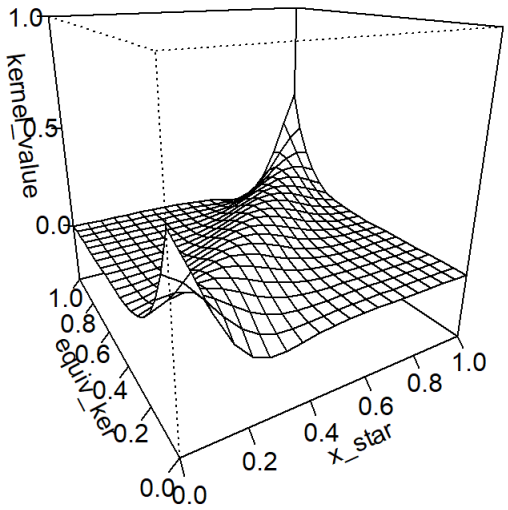

bandwidth = 0.5

```
plot_weight_diag(1)
```




bandwidth = 1

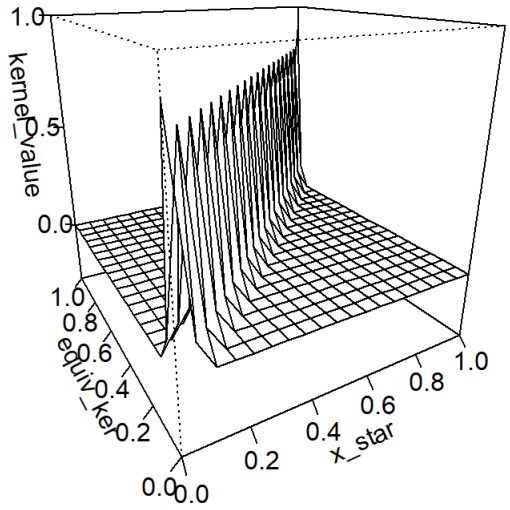
```
plot_weight_diag(3)
```




bandwidth = 3

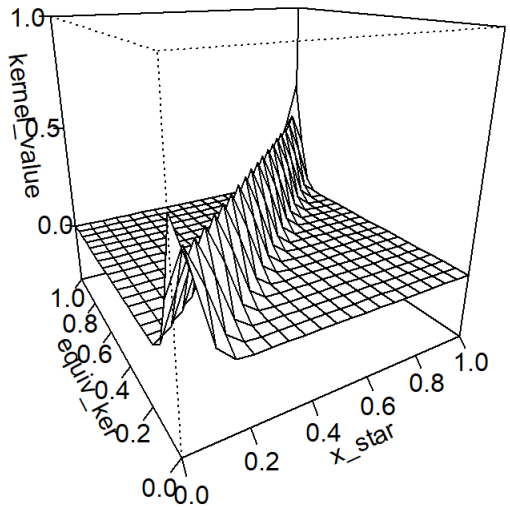
다음은 nadaraya-watson estimate에 의한 값들이다.

```
plot_weight_diag_nada(0.5)
```



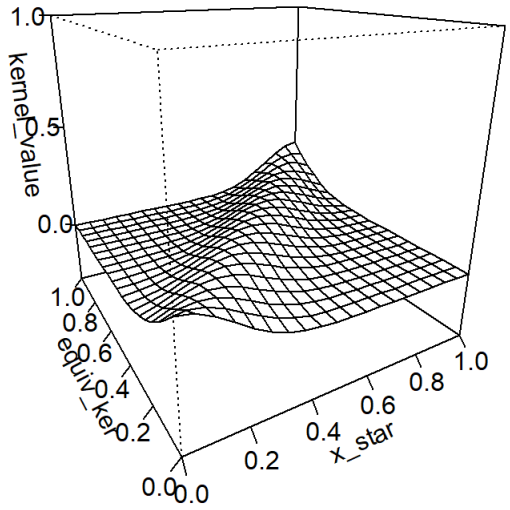

bandwidth = 0.5

```
plot_weight_diag_nada(1)
```




bandwidth = 1

```
plot_weight_diag_nada(3)
```




bandwidth = 3

보면 알 수 있지만, bandwidth가 증가 할수록, weight diagram vector들의 원소들의 크기가 전체적으로 작아지고, 넓은 범위에 분포하게 됨으로, \(x^*\) 근방의 데이터뿐만 아니라, 비교적 먼 거리의 data에도 weight를 주게 됨을 알 수 있다.

두 방법의 차이점으로는, data region의 endpoint에 가까워 질 수록, local linear regression으로 weight diagram vector들을 구했을때와 nadaraya watson 방법으로 weight diagram vector들을 구했을때,

local linear regression 방법의 weight diagram vector들이 nadaraya watson의 경우 보다 data region의 endpoint에 가까워 질 수록 x_star 근방의 데이터에 더 큰 weight를 줌을 알 수 있다.

즉, weight diagram vector 내부의 최댓값의 크기가 같은 weight에서 상대적으로 더 큼을 알 수 있다.

이제, weight diagram vector 내부 성분의 합이 1임을 numerical하게 보이자.

앞에서와 같이, 각각의 \(x^*\)로 \(X_i\)를 사용했을때의 값을 모두 비교해 보겠다.

각 \(x^*\)에서의 weight diagram vector성분의 합을 리턴해주는 함수를 아래와 같이 짜고,

```
weight_diag_sum <- function(band)
{
    xdata <- seq(1,20)

    mat <- lapply(xdata, weight_diag, xdata = xdata, band = band)
    y <- matrix(unlist(mat), nrow = length(xdata), ncol = length(xdata))

    return(colSums(y))
}

weight_diag_sum_nada <- function(band)
{
    xdata <- seq(1,20)

    mat <- lapply(xdata, weight_diag_nada, xdata = xdata, band = band)
    y <- matrix(unlist(mat), nrow = length(xdata), ncol = length(xdata))

    return(colSums(y))
}
```

각각의 값을 계산해보자.

```
weight_diag_sum(0.5)
```

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

```
weight_diag_sum(1)
```

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

```
weight_diag_sum(3)
```

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

```
weight_diag_sum_nada(0.5)
```

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

```
weight_diag_sum_nada(1)
```

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

```
weight_diag_sum_nada(3)
```

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

모든 성분이 1인것을 알 수 있다.

즉, \(x^* = 1, 2... 20\)을 사용했을때의 각 drawing weight vector 성분들의 합이 모두 1임을 확인할 수 있다.

# Q4

## 4-(a)

knot로 \(X_1, X_2... X_n\)을 가지는 finite region에서 정의된 cubic spline 함수는 다음과 같은 형태를 가짐을 안다.

\[s_{finite}(x) = a_0 + a_1 x + \frac{1}{12}\sum_{j=1}^n b_j |x - X_j|^3\] 이러한 spline 함수의 계수는

\[\mathbf{Q}=\left(\begin{array}{ccccc} 1 & 1 & 1 & \cdots & 1 \\ X_{1} & X_{2} & X_{3} & \ldots & X_{n} \end{array}\right)\] , \[\mathbf{R}=\left(\begin{array}{ccccc} 0 & \frac{\left|X_{1}-X_{2}\right|^{3}}{12} & \frac{\left|X_{1}-X_{3}\right|^{3}}{12} & \ldots & \frac{\left|X_{1}-X_{n}\right|^{3}}{12} \\ \frac{\left|X_{2}-X_{1}\right|^{3}}{12} & 0 & \frac{\left|X_{2}-X_{3}\right|^{3}}{12} & \cdots & \frac{\left|X_{2}-X_{n}\right|^{3}}{12} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\left|X_{n}-X_{1}\right|^{3}}{12} & \frac{\left|X_{n}-X_{2}\right|^{3}}{12} & \frac{\left|X_{n}-X_{3}\right|^{3}}{12} & \cdots & 0 \end{array}\right)\] 의 형태일때

natural spline의 경우,

\[\left(\begin{array}{cc} \mathbf{R} & \mathbf{Q}^{t} \\ \mathbf{Q} & \mathbf{0} \end{array}\right)\left(\begin{array}{l} \mathbf{b} \\ \mathbf{a} \end{array}\right)=\left(\begin{array}{c} \mathbf{y} \\ \mathbf{0} \end{array}\right)\]

위 형태의 행렬을 \((b, a)\)에 대해 풀면 계수를 구할 수 있음을 알고있다.

여기서, 위 식의 아래부분,

즉 \[(\begin{array}{cc} \mathbf{Q} & \mathbf{0} \end{array}) (\begin{array}{l} \mathbf{b} \\ \mathbf{a} \end{array}) = (\mathbf{0})\] 부분은 natrual spline의 natural boundary condition에서 나온 부분이므로,

diverse boundary condition에 대해 cubic spline을 구성하기 위해서는 위 식 우변의 2x 1 matrix인 0부분을 \[(\begin{array}{l} k_1 \\ k_2 \end{array})\] 와 같이 임의의 수로 바꾸고, 이 식을 계수 b,a에 대해 풀어서 계수를 구하면 이에 맞는 cubic spline 함수를 구할 수 있게된다.

따라서, 이를 구할 수 있도록 아래와 같이 함수를 짜자.

아래 함수는 cubic spline 함수를 구하고, xvalue에 해당하는 위치에서의 함숫값을 리턴해준다.

이때, 주의할점은 xdata가 sort된 상태로 들어와야한다는 것이다.

따라서 먼저 sorting을 해준다음, 함수를 실행하도록 구현하였다.

```
cubic_spline <- function(xvalue,xdata, ydata, boundary_cond_1 = 0, boundary_cond_2 = 0)
{
    ydata <- ydata[order(xdata)]
    xdata <- xdata[order(xdata)]
    #sort by ascending order. order of ydata, xdata is important!!

    span_xdata_bycol <- replicate(length(xdata),xdata)
    # span xdata by column.
    #if x = c(1,2,3), then
    #matrix is (1 1 1
    #           2 2 2
    #           3 3 3)

    span_xdata_byrow <- t(replicate(length(xdata),xdata))
    # span xdata by row.
    #if x = c(1,2,3), then
    #matrix is (1 2 3
    #           1 2 3
    #           1 2 3)

    R <- abs((span_xdata_bycol - span_xdata_byrow)**3)/12
    Q <- rbind(rep(1, length(xdata)), xdata)

    mat <- rbind(cbind(R, t(Q)), cbind(Q, rep(0,2),rep(0,2)))

    y <- rbind(matrix(ydata, nrow = length(ydata), ncol = 1), boundary_cond_1, boundary_cond_2)

    coef <- solve(mat, y)
    # solve(A,b) -> A^{-1}b.

    #now we have coefficient of spline function.

    #so, we can get value of spline at xvalue.

    val_mat <- t(abs((replicate(length(xvalue), xdata) - t(replicate(length(xdata),xvalue)))**3)/12)

    #xvalue = (z1,z2...zk), then this matrix has 1/12(|z_1-x_1|^3, |z_1-x_2|^3... |z_1-x_n|) at rows.

    retval = cbind(val_mat, rep(1, length(xvalue)), xvalue) %*% coef

    # now we have at retval, xvalue = (z1,z2...zk)'s spline functions value.

    return(as.vector(retval))
}
```

위 함수는 xdata, ydata와 boundary cond1, boundary cond2 를 받아

$$\left(\begin{array}{cc} \mathbf{R} & \mathbf{Q}^{t} \\ \mathbf{Q} & \mathbf{0} \end{array}\right)\left(\begin{array}{l} \mathbf{b} \\ \mathbf{a} \end{array}\right)=\left(\begin{array}{c} \mathbf{y} \\ \mathbf{k} \end{array}\right)$$,

$k = (k1, k2)^t$, k1과 k2는 각각 boundary cond1, boundary cond2에 해당한다.

이를 coefficient에 대해 풀어 coefficient를 구한다음,

xvalue 벡터 내부에 있는 값들에 해당하는 spline 함수의 값들을 벡터로 리턴해준다.

시험삼아 다음과 같이 spline 함수의 값을 구해보면 (natural spline)

```
cubic_spline(c(1, 1.7, 3, 4.2, 5), c(1,1.7,3,4.2,5), c(2,5,4,5,3), 0, 0)
```

```
## [1] 2 5 4 5 3
```

위와 같이 값이 나오는것을 알 수 있다.

natural spline은 data point를 interpolate 하는 함수이므로, ydata와 같은 값이 나오므로 정상적으로 값이 나옴을 알 수 있다.

```
x = seq(1,20)
y = my_func(x) + rnorm(20,sd = 0.5)
ns_my <- cubic_spline(x, x, y, 0, 0)
ns_my
```

```
##  [1]  0.50476297  0.14452281  0.90448304  0.13726244  1.45501318
##  [6]  1.29308012  0.78425700  1.43175175  0.03489973  0.63617551
## [11]  0.04155307  0.50703985 -0.17450181 -0.57032402 -0.70862148
## [16] -0.68406360 -0.37726160 -0.20612318  0.29117032 -0.07019571
```

1번에서 사용했던것과 같은 데이터를 사용해 natural spline을 이용해 추정한 값은 위와 같다.

```
ns_r <- spline(x,y,n=length(x), method = 'natural')
ns_r
```

```
## $x
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##
## $y
##  [1]  0.50476297  0.14452281  0.90448304  0.13726244  1.45501318
##  [6]  1.29308012  0.78425700  1.43175175  0.03489973  0.63617551
## [11]  0.04155307  0.50703985 -0.17450181 -0.57032402 -0.70862148
## [16] -0.68406360 -0.37726160 -0.20612318  0.29117032 -0.07019571
```

```
ns_my - ns_r$y
```

```
##  [1] -3.494982e-13  1.671996e-13 -2.562395e-13  4.690692e-13  2.238210e-13
##  [6]  4.343192e-13  1.942890e-14  5.329071e-15 -5.884182e-15  6.128431e-14
## [11] -2.068901e-13 -1.749711e-13 -1.187939e-13  2.992051e-13 -2.058353e-13
## [16] -2.180478e-13 -5.057066e-14 -1.855655e-12 -1.113942e-12 -3.084200e-13
```

R 내장함수인 spline을 이용해 추정한 값과 거의 같음을 알 수 있다.

# 4-(b)

$\int_{X_{1}}^{X_{n}}\left(\frac{d^{2} m(x)}{d x^{2}}\right)^{2} d x$ 의 값을 numerical하게 구하는 함수를 짜보자.

$s_{finite}(x) = a_0 + a_1 x + \frac{1}{12}\sum_{j=1}^n b_j |x - X_j|^3$ 를 2번 미분하면, 책 155페이지의 (3.167) 식을 참조하면

$\frac{d^{2} s_{f i n i t e}(x)}{d x^{2}}=\frac{1}{2} \sum_{j=1}^{n} b_{j} \cdot\left|x-X_{j}\right|$ 임을 알고있다.

따라서, 위 적분값을 numerical하게 구하는 함수를 짜보자. r의 내장함수인 integrate를 이용할 것이다.

먼저, $s_{finite}$를 두번 미분한값을 제곱한 값을 리턴하는 함수를 짜야한다.

다음과 같이 구현할 수 있다.

```
square_2_deriv_cubic_spline <- function(xvalue, xdata, ydata, boundary_cond_1 = 0, boundary_cond_2 = 0)
{
    ydata <- ydata[order(xdata)]
    xdata <- xdata[order(xdata)]
    #sort by ascending order. order of ydata, xdata is important!!

    span_xdata_bycol <- replicate(length(xdata),xdata)
    # span xdata by column.
    #if x = c(1,2,3), then
    #matrix is (1 1 1
    #           2 2 2
    #           3 3 3)

    span_xdata_byrow <- t(replicate(length(xdata),xdata))
    # span xdata by row.
    #if x = c(1,2,3), then
    #matrix is (1 2 3
    #           1 2 3
    #           1 2 3)

    R <- abs((span_xdata_bycol - span_xdata_byrow)**3)/12
    Q <- rbind(rep(1, length(xdata)), xdata)

    mat <- rbind(cbind(R, t(Q)), cbind(Q, rep(0,2),rep(0,2)))

    y <- rbind(matrix(ydata, nrow = length(ydata), ncol = 1), boundary_cond_1, boundary_cond_2)

    coef <- solve(mat, y)
    # solve(A,b) -> A^{-1}b.

    #now we have coefficient of spline function.

    #so, we can get value of spline at xvalue.
    t(abs((replicate(length(xvalue), xdata) - t(replicate(length(xdata),xvalue)))/2))

    val_mat <- t(abs((replicate(length(xvalue), xdata) - t(replicate(length(xdata),xvalue)))/2))

    #xvalue = (z1,z2...zk), then this matrix has 1/2(|z_1-x_1|, |z_1-x_2|... |z_1-x_n|) at rows.

    retval = val_mat %*% coef[1:length(xdata),]

    # now we have at retval, xvalue = (z1,z2...zk)'s spline functions second derivate value.

    return(as.vector(retval**2))
}
```

이제, integrate를 다음과 같이 이용하면 적분값을 구할 수 있다.

```
get_integral <- function(xdata1, ydata1, boundary_cond_11 = 0, boundary_cond_21 = 0)
{
    retval <- integrate(square_2_deriv_cubic_spline, min(xdata1), max(xdata1), xdata = xdata1, ydata = ydata
1,  boundary_cond_1 = boundary_cond_11,  boundary_cond_2 =  boundary_cond_21)
    return(retval$value)

}
```

이제 boundary condition에 따라 \[\int_{X_{1}}^{X_{n}}\left(\frac{d^{2} m(x)}{d x^{2}}\right)^{2} d x\] 이 언제 최소화되는지를 알기 위한 모든 준비가 끝났다.

boundary condition을 바꿔가면서, xdata = (1, 1.7, 3, 4.2, 5), ydata= (2, 5, 4, 5, 3)인 간단한 경우에 대해

natural spline 일때(boundary condition이 둘다 0일때) 이 적분값이 최소가 됨을 보이자.

```
xdata <- c(1, 1.7, 3, 4.2, 5)
ydata <- c(2, 5, 4, 5, 3)
values <- seq(-2,2,by = 0.1)
integrated_value <- matrix(nrow = length(values), ncol = length(values))

index_to_value <- function(index)
{
    return(-2.1 + index * 0.1)
}

value_to_index <- function(value)
{
    return((value + 2.1)/0.1)
}

for(i in seq_len(length(values)))
{
    for(j in seq_len(length(values)))
    {
         integrated_value[i,j] <- get_integral(xdata, ydata, index_to_value(i), index_to_value(j))
    }
}

min(integrated_value)
```

```
## [1] 80.63653
```

```
integrated_value[value_to_index(0),value_to_index(0)]
```

```
## [1] 80.63653
```

위에서 알 수 있듯이, boundary condition으로 0,0을 주었을때의 integrated value가 가장 작음을 알 수 있다.

따라서, numerical하게 boundary condition이 0,0일때 위 적분값이 최소가 됨을 확인할 수 있었다.

위에서처럼, 1번에서 사용했던 데이터를 이용했을때도 어떻게 되는지 확인해보자.

```
xdata <- seq(1,20)
ydata <- my_func(xdata) + rnorm(20,sd = 0.5)
values <- seq(-2,2,by = 0.1)
integrated_value_d <- matrix(nrow = length(values), ncol = length(values))

for(i in seq_len(length(values)))
{
    for(j in seq_len(length(values)))
    {
         integrated_value_d[i,j] <- get_integral(xdata, ydata, index_to_value(i), index_to_value(j))
    }
}

min(integrated_value_d)
```

```
## [1] 93.3597
```

```
integrated_value_d[value_to_index(0),value_to_index(0)]
```

```
## [1] 93.3597
```

역시 0,0에서 적분값이 최소가 됨을 확인할 수 있다.

# Q5

\[E_{s s}=(\mathbf{y}-\hat{\mathbf{y}})^{t}(\mathbf{y}-\hat{\mathbf{y}})+\lambda \hat{\mathbf{y}}^{t}\left(\mathbf{T} \mathbf{H}^{-1}\right)^{t} \mathbf{R} \mathbf{T} \mathbf{H}^{-1} \hat{\mathbf{y}}\] 위식에서, \(L = (T H^{-1})^t R T H^{-1}\)임을 아므로, 이 L을 구할 수 있으면, \((I + \lambda L)^{-1}\)역시 구할 수 있다.

따라서, smoothing spline의 L을 구할 것이다.

여기서 R과 Q는 문제 4번에서 사용한 행렬이고, T와 H는 다음과 같다.

\[\mathbf{H}=\left[\left(\begin{array}{cc} \mathbf{R} & \mathbf{Q}^{t} \end{array}\right)\left(\begin{array}{cc} \mathbf{R}+\lambda \mathbf{I}_{n} & \mathbf{Q}^{t} \\ \mathbf{Q} & \mathbf{0} \end{array}\right)^{-1}\right]_{(n, n)}\]

\[\mathbf{T}=\left[\left(\begin{array}{cc} \mathbf{R}+\lambda \mathbf{I}_{n} & \mathbf{Q}^{t} \\ \mathbf{Q} & 0 \end{array}\right)^{-1}\right]_{(n, n)}\]

따라서, 이를 이용하여 L과 \((I + \lambda L)^{-1}\)의 값을 구해주는 함수를 만들 것이다.

이 값들은 xdata와 lambda만 있으면 계산이 가능함은 자명하다.

```
smoothing_spline_get_L <- function(xdata, lamb)
{
    xdata <- xdata[order(xdata)]
    #sort by ascending order. order of ydata, xdata is important!!

    n <- length(xdata)

    span_xdata_bycol <- replicate(length(xdata),xdata)
    # span xdata by column.
    #if x = c(1,2,3), then
    #matrix is (1 1 1
    #           2 2 2
    #           3 3 3)

    span_xdata_byrow <- t(replicate(length(xdata),xdata))
    # span xdata by row.
    #if x = c(1,2,3), then
    #matrix is (1 2 3
    #           1 2 3
    #           1 2 3)

    R <- abs((span_xdata_bycol - span_xdata_byrow)**3)/12
    Q <- rbind(rep(1, length(xdata)), xdata)

    lambda_I <- lamb * diag(n)

    mat <- rbind(cbind(R + lambda_I, t(Q)), cbind(Q, rep(0,2),rep(0,2)))
    # mat = (R+lambda_In, Q^t \\ Q 0)

    T_big = solve(mat) # (n+2) x (n+2) matrix. solve(A) returns A^-1
    H_big = cbind(R, t(Q)) %*% T_big #  n x (n+2) matrix.
    H = H_big[1:n,1:n]
    H_inv = solve(H)
    T_mat = T_big[1:n, 1:n]

    TH_inv = T_mat %*% H_inv

    L = t(TH_inv) %*% R %*% TH_inv
    I_lambdaL_inv = solve(diag(n) + lamb * L)

    return(list(L, I_lambdaL_inv))

}
```

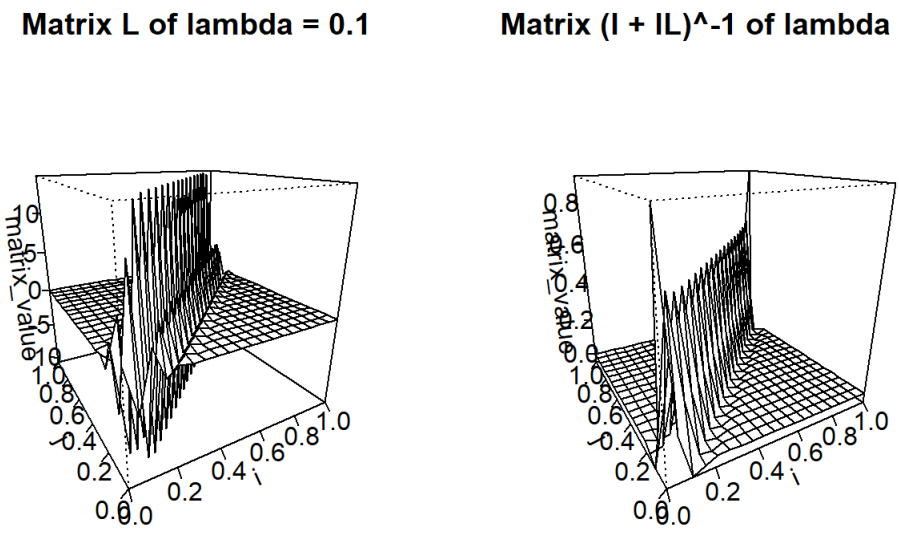
위 함수는 L과 \((I + \lambda L)^{-1}\)을 리스트로 만들어 리턴한다.

이제, x = 1,2…20을 이용해서 각각의 값을 구해보자. lambda로는 0.1, 1, 10을 이용하였다.

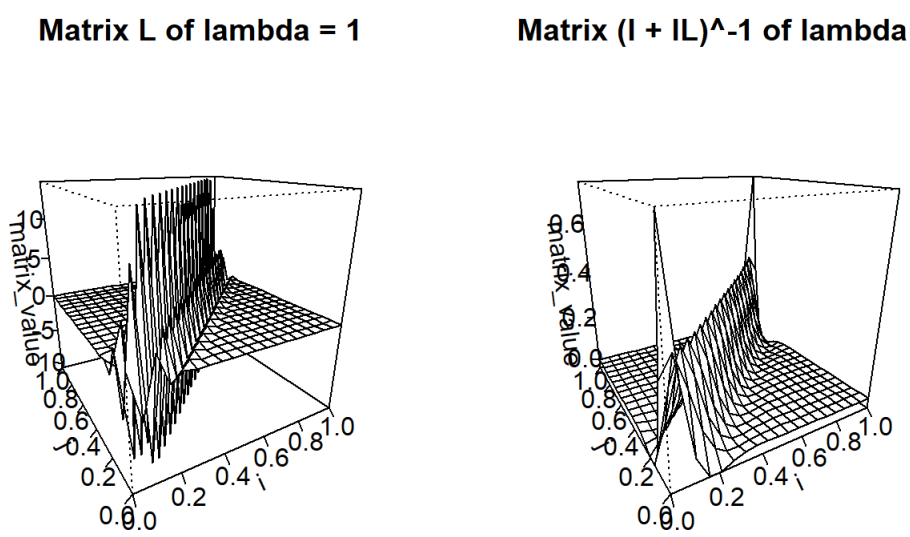
```
X <- seq(1,20)
small_lambda <- smoothing_spline_get_L(X, 0.1)
medium_lambda <- smoothing_spline_get_L(X, 1)
big_lambda <- smoothing_spline_get_L(X, 10)
```

각 lambda에 따른 matrix의 값을 출력해보면 다음과 같다.

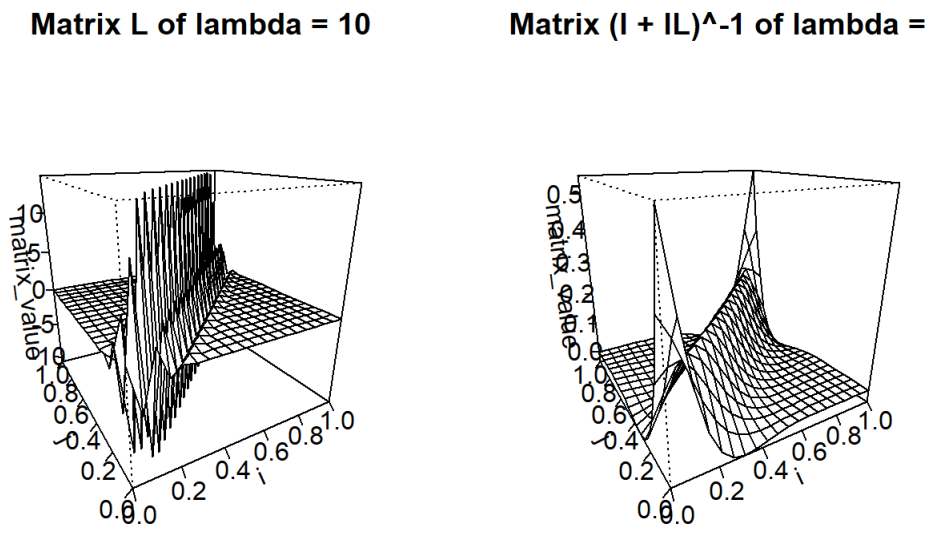
```
library(stringr)

plot_matrix <- function(A, description = "matrix plot")
{
    persp(A, xlab = "i", ylab = "j",  zlab = "matrix_value", lab = c(3,3,3), theta = -30, phi = 20, main =
description, ticktype = "detailed")
}


plot_L <- function(Lmatrices, description = "")
{
    par(mfrow=c(1,2))
    plot_matrix(Lmatrices[[1]], str_c("Matrix L of " ,description))
    plot_matrix(Lmatrices[[2]], str_c("Matrix (I + lL)^-1 of ", description))
}

plot_L(small_lambda,"lambda = 0.1")
```




```
plot_L(medium_lambda,"lambda = 1")
```




```
plot_L(big_lambda, "lambda = 10")
```




각각의 lambda값에 대해 matrix내부의 값들을 확인할 수 있다.

lambda가 커질수록, smoothing power가 늘어나 \((I + \lambda L)^{-1}\)의 diagonal member의 크기가 전체적으로 작아지는것을 확인할 수 있다.

이를, chaper 2에서 했었던 smoothing spline for an equispaced predictor의 경우와 비교해보면,

이때의 L이 S와 같으므로, (책 52페이지, 식 (2.61, 2.62, 2.63) 참조)

\[ \hat{y} = (I + \lambda S)^{-1} y$ \] 에서, 이때의 S와 \((I + \lambda S)^{-1}\)의 값을 구해오는 코드를 짜고, 출력까지 해보자.

chapter 2에서 사용했었던 코드를 참조하면 다음과 같이 짤 수 있다.

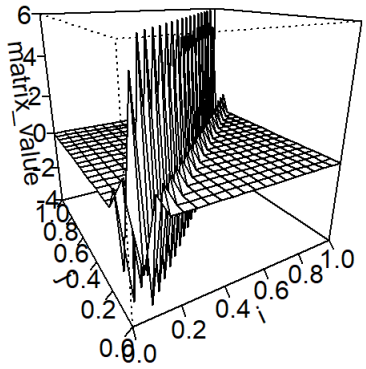
구현의 편의를 위해, 들어오는 데이터의 개수가 5개 이상인 경우에 대해서만 구현하겠다.

```
# get equispaced data
smoothing_spline_get_S <- function(xdata, lamb)
{
    nd <- length(xdata)

    ss <- c(1, -2, 1, rep(0, nd - 3))
    ss <- rbind(ss, c(-2, 5, -4, 1, rep(0, length = nd - 4)))
    for(ii in 1:(nd - 4)){
        ss <- rbind(ss, c(rep(0, ii - 1), 1, -4, 6, -4, 1, rep(0, nd - ii - 4)))
    }
    ss <- rbind(ss, c(rep(0, length = nd - 4), 1, -4, 5, -2))
    ss <- rbind(ss, c(rep(0, length = nd - 3), 1, -2, 1))

    ss_inv <- solve(diag(nd) + lamb * ss)

    return(list(ss, ss_inv))
}
```

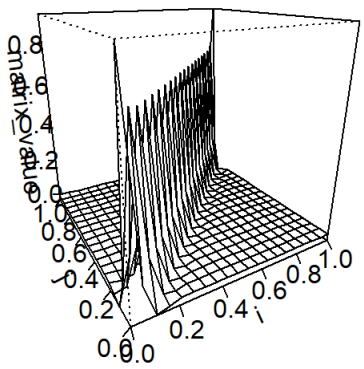
이제, 위와같이 각 lambda에 대해 matrix를 만들고 비교해보자.

```
X <- seq(1,20)
small_lambda_S <- smoothing_spline_get_S(X, 0.1)
medium_lambda_S <- smoothing_spline_get_S(X, 1)
big_lambda_S <- smoothing_spline_get_S(X, 10)

plot_L(small_lambda_S,"S, lambda = 0.1")
```

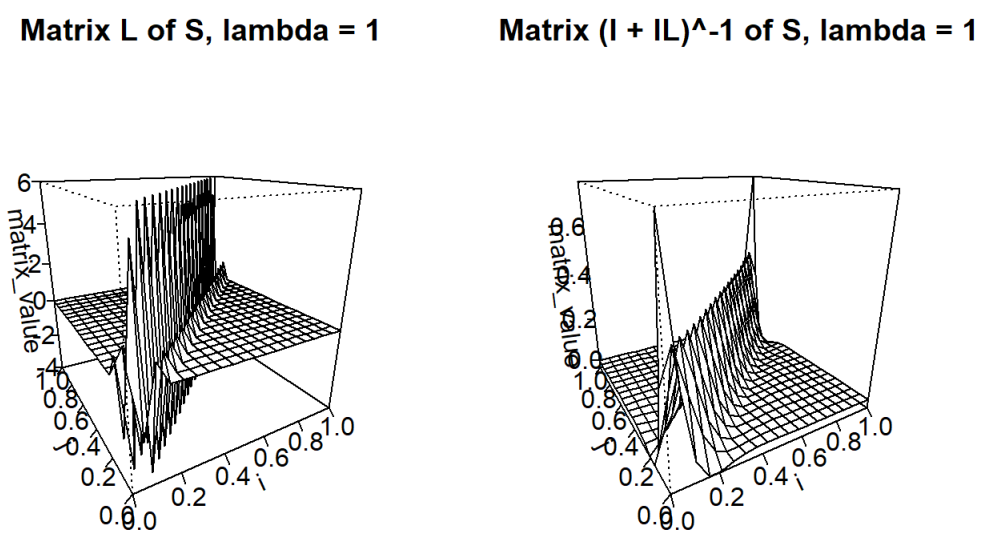
**Matrix L of S, lambda = 0.1**



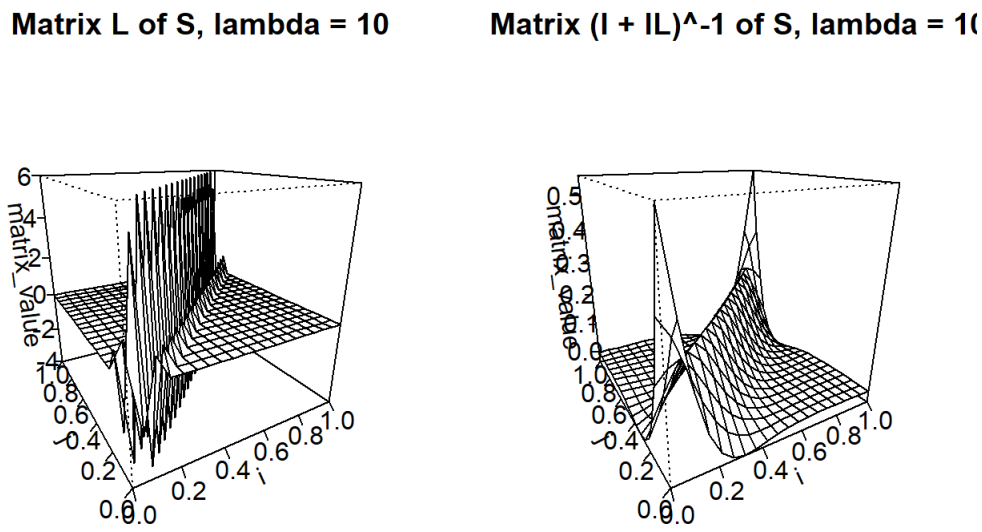

**Matrix (I + lL)^-1 of S, lambda = 0.**




```
plot_L(medium_lambda_S,"S, lambda = 1")
```




```
plot_L(big_lambda_S, "S, lambda = 10")
```




세세한 값의 차이는 나지만, 전체적인 형태는 비슷함을 알 수 있다.

이는 S가 L을 근사한 matrix이기 때문에 그렇다.

# Q6

LOESS는 다음과 같은 식의 값을 최소화 하는 \(x^*\)를 사용하게 된다.

\[E_{L O E S S}\left(x^{*}\right)=\sum_{i=1}^{n}\left(w\left(\frac{X_{i}-x^{*}}{h_{k}\left(x^{*}\right)}\right)\left(a_{0}\left(x^{*}\right)+\sum_{j=1}^{p} a_{i}\left(x^{*}\right)\left(X_{i}-x^{*}\right)^{j}-Y_{i}\right)^{2}\right)\]

여기서, \(h_k(x^*)\)은 \(x^*\)과 가까운 k번째 데이터 까지의 거리가 되는데,

이 때문에 $D = {(X_1, Y_1) … (X_n, Y_n)} $이라 할때, 만약 1~n개의 데이터 중에 k번째 데이터를 뺀 dataset을 \(D^{-k}\)라 하면, \(D^{-k}\)를 통해 추정한 \(\widehat{m^{-k}}(X_i)\)와, \(D^{-k}\)에서 \((X_k, \widehat{m^{-k}}(X_k)\)을 집어넣고 추정한 값을 \(\widehat{m^{k'}}

(X_k)\)이라 하면,

\(\widehat{m^{k'}}(X_k) \neq \widehat{m^{-k}}(X_k)\) 가 될 수 있어,

CV에서 \(Y_k - \widehat{m^{-k}}(X_k) \neq \frac{Y_k - \widehat{m(X_k)}}{1 - H_{kk}}\) 의 관계가 더는 성립하게 되지 않게 되고, 따라서 (I)에서 구했던 CV는 정확한 CV가 아니라, CV의 근삿값이 되게 된다.

\[C V[\hat{m}(x)]=\frac{\sum_{k=1}^{n}\left(Y_{k}-\hat{m}^{-k}\left(X_{k}\right)\right)^{2}}{n}\]

따라서, 원래의 정의인 위와 같이 CV를 구해보자.

이를 실제로 확인해 보기 위해, 원래 정의로 구한 CV와 approximation된 CV의 값을 비교해보자.

아래는 (I)부분에 해당하는 코드를 약간 변형한 것이다. (수업시간에 했던 내용입니다.)

```
##(I) Calculation of CV and GCV for LOESS
locv1 <- function(x1, y1, nd, span1, ntrial)
{
# (1)
  locvgcv <- function(sp, x1, y1)
  {
    nd <- length(x1)
# (2)
    assign("data1", data.frame(xx1 = x1, yy1 = y1))
    fit.lo <- loess(yy1 ~ xx1, data = data1, span = sp,
                    family = "gaussian", degree = 1, surface = "direct")
    res <- residuals(fit.lo)
# (3)
    dhat2 <- function(x1, sp)
    {
      nd2 <- length(x1)
      diag1 <- diag(nd2)
      dhat <- rep(0, length = nd2)
# (4)
      for(jj in 1:nd2) {
        y2 <- diag1[, jj]
        assign("data1", data.frame(xx1 = x1, yy1 = y2))
        fit.lo <- loess(yy1 ~ xx1, data = data1,
                        span = sp, family = "gaussian", degree = 1,
                        surface = "direct")
        ey <- fitted.values(fit.lo)
        dhat[jj] <- ey[jj]
      }
      return(dhat)
    }
# (5)
    dhat <- dhat2(x1, sp)
    trhat <- sum(dhat)
    sse <- sum(res^2)
# (6)
    cv <- sum((res/(1 - dhat))^2)/nd
    gcv <- sse/(nd * (1 - (trhat/nd))^2)
# (7)
    return(list(cv = cv, gcv = gcv))
  }
# (8)
  cvgcv <- lapply(as.list(span1), locvgcv, x1 = x1, y1 = y1)
  cvgcv <- unlist(cvgcv)
  cv <- cvgcv[attr(cvgcv, "names") == "cv"]
  gcv <- cvgcv[attr(cvgcv, "names") == "gcv"]
# (9)
  return(list(cv = cv, gcv = gcv))
}
```

(eval = FALSE라 실행되진 않음)

위 함수를 우리의 목적에 맞게 적절히 고쳐보자.

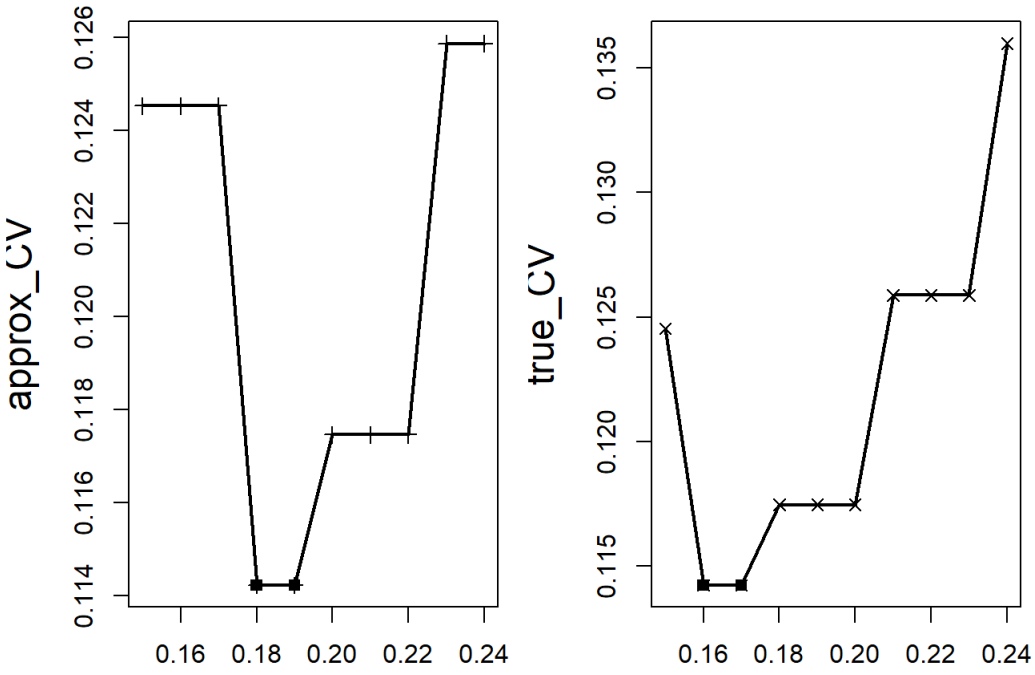
```
loess_approx_cv <- function(sp, x1, y1)
 {
    nd <- length(x1)
# (2)
    assign("data1", data.frame(xx1 = x1, yy1 = y1))
    fit.lo <- loess(yy1 ~ xx1, data = data1, span = sp,
                    family = "gaussian", degree = 1, surface = "direct")
```

```
                    family = "gaussian", degree = 1, surface = "direct")
    res <- residuals(fit.lo)
# (3)
    dhat2 <- function(x1, sp)
    {
      nd2 <- length(x1)
      diag1 <- diag(nd2)
      dhat <- rep(0, length = nd2)
# (4)
      for(jj in 1:nd2) {
        y2 <- diag1[, jj]
        assign("data1", data.frame(xx1 = x1, yy1 = y2))
        fit.lo <- loess(yy1 ~ xx1, data = data1,
                        span = sp, family = "gaussian", degree = 1,
                        surface = "direct")
        ey <- fitted.values(fit.lo)
        dhat[jj] <- ey[jj]
      }
      return(dhat)
    }
# (5)
    dhat <- dhat2(x1, sp)
# (6)
    cv <- sum((res/(1 - dhat))^2)/nd
# (7)
    return(cv)
}

loess_true_cv <- function(sp, x1, y1)
{
    nd <- length(x1)
    res <- vector(length = nd)
    for(i in seq_len(nd))
    {
         df <- data.frame(xx1 = x1[-i], yy1 = y1[-i])
         # delete ith data from dataset
         fit.lo <- loess(yy1 ~ xx1, data = df, span = sp,
                    family = "gaussian", degree = 1, surface = "direct")
         pred <- predict(fit.lo, data.frame(xx1 = x1[i]))
         names(pred) <- NULL
         res[i] <- y1[i] - pred
    }

    cv <- sum((res)^2)/nd

    return(cv)

}

listing_cv <- function(sp, x1, y1)
{
    approx_cv <- loess_approx_cv(sp, x1, y1)
    true_cv <- loess_true_cv(sp, x1, y1)

    return(list(approx_cv = approx_cv, true_cv = true_cv))
}

approx_true_cv <- function(x1, y1, nd, span1)
{
# (8)
  cvlst <- lapply(as.list(span1), listing_cv, x1 = x1, y1 = y1)
  cvlst <- unlist(cvlst)
  approx_cv <- cvlst[attr(cvlst, "names") == "approx_cv"]
  true_cv <- cvlst[attr(cvlst, "names") == "true_cv"]
# (9)
  return(list(approx_cv = approx_cv, true_cv = true_cv))
}
```

이제, 위 함수를 통해 책 (I)의 example case에 대해 실행해보자. (교과서 220~221 page)

GCV에 대한 부분은 필요 없으므로, 아래와 같이 적절히 고쳐서 비교해 보겠다.

```
### Figure Similar to Figure 3.45
# (1)
set.seed(195)
nd <- 40
xx <- seq(from = 1, by = 1, length = nd)^1.8
yy <- sin(0.004 * pi * xx) + rnorm(nd, mean = 0, sd = 0.3)
# (2)
ntrial <- 10
span1 <- seq(from = 0.15, by = 0.01, length = ntrial)
# (3)
output.lo <- approx_true_cv(xx, yy, nd, span1)
approx_cv <- output.lo$approx_cv
true_cv <- output.lo$true_cv
# (4)
par(mfrow = c(1, 2), mar = c(3, 4, 2, 1),
    oma=c(0.5,0.5,0.5,0.5), cex.lab=1.5)
plot(span1, approx_cv, type = "n",
     xlab = "span", ylab = "approx_CV")
points(span1, approx_cv, pch = 3)
lines(span1, approx_cv, lwd = 2)
pcvmin <- seq(along = approx_cv)[approx_cv == min(approx_cv)]
spancv <- span1[pcvmin]
cvmin <- approx_cv[pcvmin]
points(spancv, cvmin, cex = 1, pch = 15)
# (5)
plot(span1, true_cv, type = "n",
     xlab = "span", ylab = "true_CV")
points(span1, true_cv, pch =4)
lines(span1, true_cv, lwd = 2)
pgcvmin <- seq(along = true_cv)[true_cv == min(true_cv)]
spangcv <- span1[pgcvmin]
gcvmin <- true_cv[pgcvmin]
points(spangcv, gcvmin, cex = 1, pch = 15)
```




approx cv가 최소가 되는 span과, true cv가 최소가 되는 span의 값이 다름을 알 수 있다.

또한, 실제 값에서도 차이가 조금씩 남을 알 수 있지만, 근삿값이다 보니 그렇게 큰 차이가 나지는 않음을 알 수 있다.

# Q7

## 7-(a)

\[Y_i = \sin{0.2\pi X_i} + \epsilon_i\]

\[\{X_i\} = \{0.1 ,0.2, ... ,0.1n\}\]

\[\epsilon_{i} \overset{\text{i.i.d.}}{\sim} N(0, 0.1^{2})\] 일때 data를 생성하는 함수는 다음과 같이 간편하게 짤 수 있다.

```
generate_data <- function(n)
{
    x <- seq(from = 0.1, by = 0.1, length = n)

    e <- rnorm(n, mean = 0, sd = 0.1)

    df <- data.frame(x = x, y = sin(0.2 * pi * x) + e)

    return(df)
}
```

데이터 개수인 n을 받아, 해당하는 x와 y를 dataframe으로 만들어서 리턴해주는 함수이다.

이를 통해 100, 500, 2000개의 데이터를 생성하면 다음과 같다.

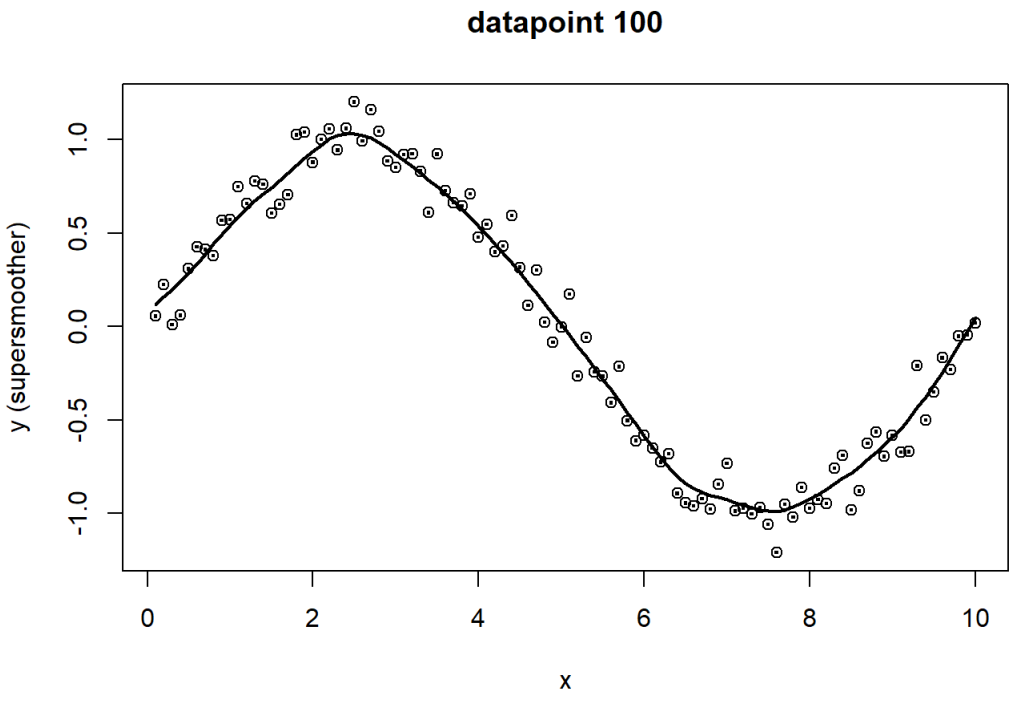
```
data_100 <- generate_data(100)
data_500 <- generate_data(500)
data_2000 <- generate_data(2000)
```

## 7-(b)

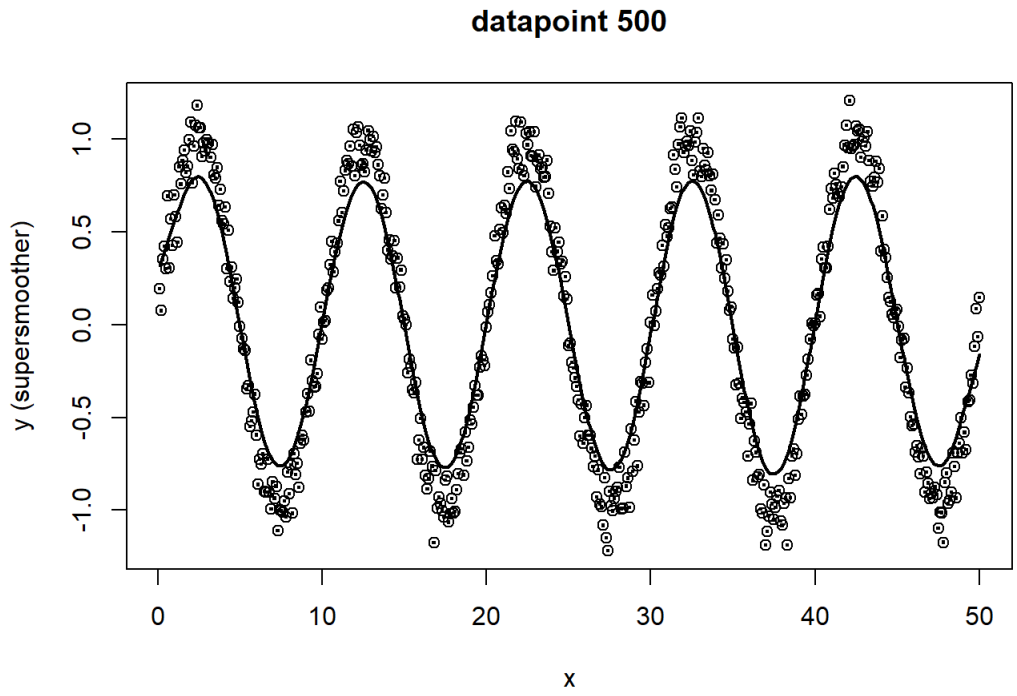
이제, super smoother를 이용해서 각 데이터에 대해 추정을 해보고, 결과를 출력해보자.

다음과 같이 각 data에 대해 plot을 해볼 수 있다.

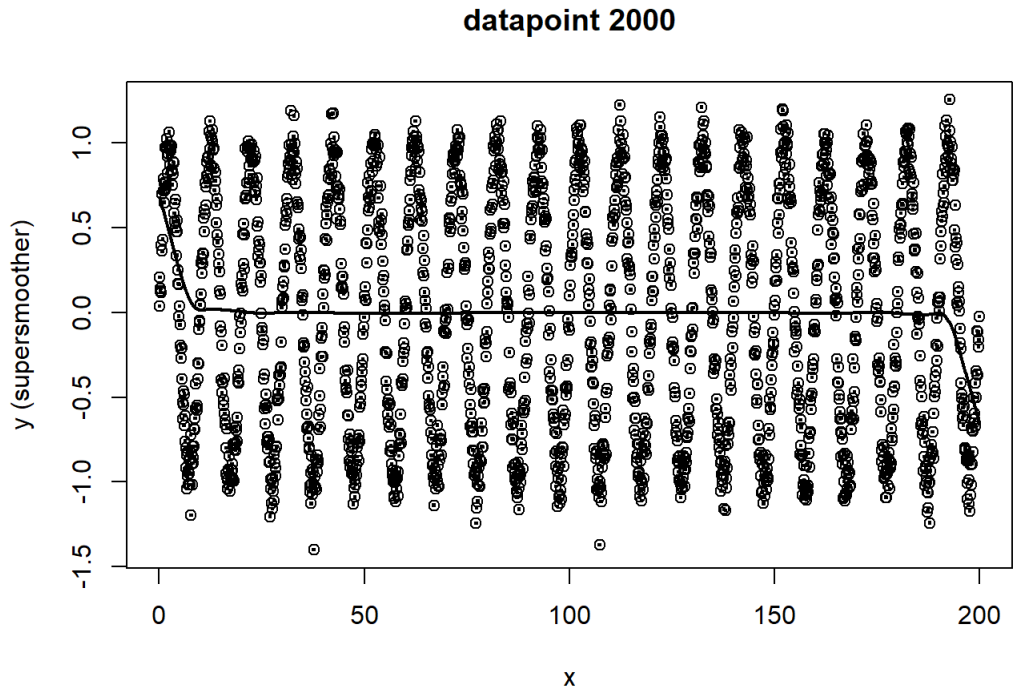
```
plot_supersmooth <- function(df, description = "")
{
    fit.su <- supsmu(df$x, df$y, span = "cv")
    par(mfrow=c(1,1))
    plot(x = df$x, y = df$y, type = "p", xlab = "x", ylab = "y (supersmoother)", main = description)
    points(x = df$x, y = df$y, pch = 15, cex = 0.3)
    lines(x = fit.su$x,y = fit.su$y, lwd = 2)
}

plot_supersmooth(data_100, "datapoint 100")
```




```
plot_supersmooth(data_500, "datapoint 500")
```




```
plot_supersmooth(data_2000, "datapoint 2000")
```




data 개수가 100개일때는 잘 smoothing이 됐다고 말할 수 있고,

data 개수가 500개일때부터 true function값과 약간 동떨어진 추정값이 나오다가,

data 개수가 2000개일때는 truefunction과는 아예 동떨어진 값으로 추정됨을 알 수 있다.

이렇게 값이 나타나는 이유는 다음과 같다.

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,

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

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

위 residual의 크기를 최소화하는 k를 이용하여 $X_i$에서의 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.

여기서, 데이터가 많아질때 이 고정된 k값들이 문제가 되게 되는데,

데이터가 2000개일 경우, span으로 0.05의 값이 들어간 상태로 LOESS를 이용하여 추정된다고 해볼때, 결론적으로 전체 data의 5%인 40개 정도의 데이터를 포함하는 h가 선택되게 된다.

그런데 원 함수가 sine 함수에서, 40개정도의 데이터를 이용하여 smoothing 을 진행하게 되면, 거의 직선에 가깝게 smoothing이 될 것이고, 결론적으로 over-smoothing 문제가 생기게 된다.

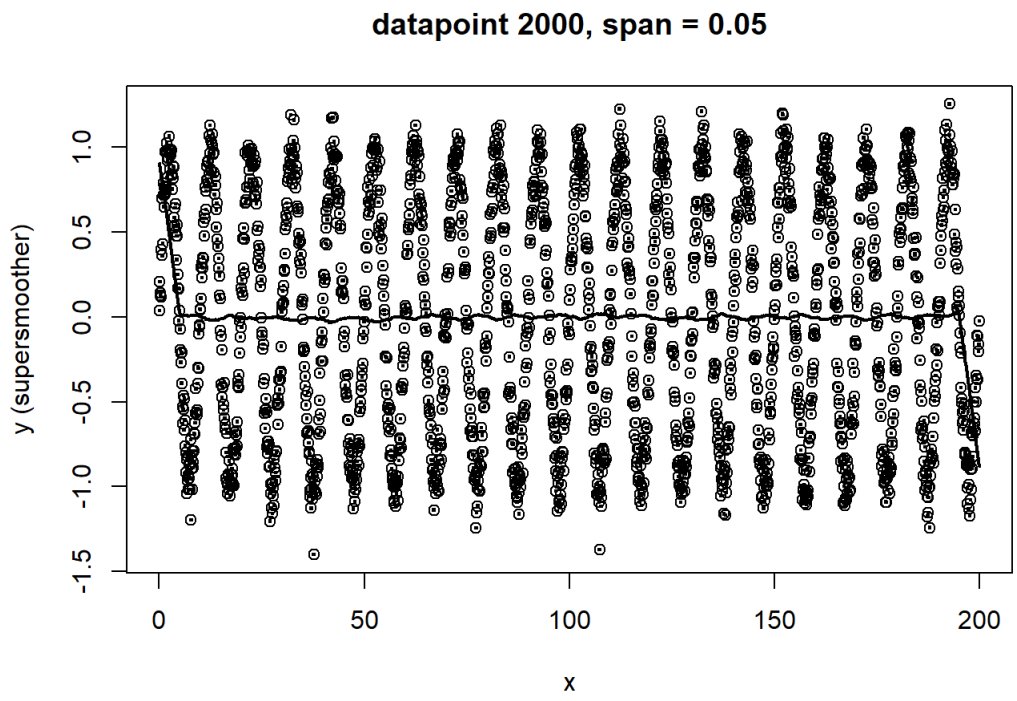
즉, 데이터를 생성하는 원함수가 주기함수인데, 데이터가 많아질수록 super smoother의 span은 고정되어 있으므로 문제가 생기는 것이다.

따라서, 이를 해결하려면 다음과 같이 데이터의 크기가 클때, span을 직접 조정해 주어 이러한 over-smoothing을 막을 수 있다.

위 함수를 약간 변형하여, span을 사용자가 설정할 수 있도록 해보자.

```
plot_supersmooth_span <- function(df, sp = "cv", description = "")
{
    fit.su <- supsmu(df$x, df$y, span = sp)
    par(mfrow=c(1,1))
    plot(x = df$x, y = df$y, type = "p", xlab = "x", ylab = "y (supersmoother)", main = description)
    points(x = df$x, y = df$y, pch = 15, cex = 0.3)
    lines(x = fit.su$x,y = fit.su$y, lwd = 2)
}
```

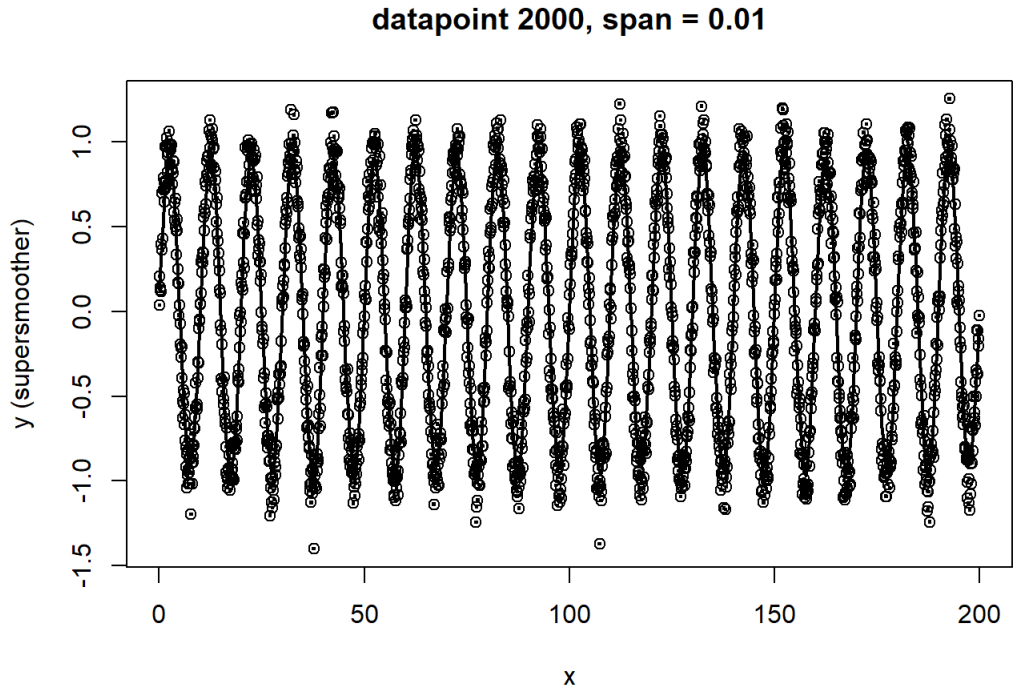
```
plot_supersmooth_span(data_2000,0.05, "datapoint 2000, span = 0.05")
```




0.05의 span만 사용해도 이렇게 over-smoothing된 그래프가 나오므로, 더 span을 줄여줄 것이다.

data 개수가 100일때의 경우와 비슷하게 맞춰주려면, span을 0.2에서 20배는 더 줄여주어야 한다.

```
plot_supersmooth_span(data_2000,0.01, "datapoint 2000, span = 0.01")
```




위와 같이, 0.01의 span을 사용하면, 원 함수와 비슷한 형태의 그래프가 나옴을 알 수 있다.

즉, data가 너무 많아질 경우, span을 데이터의 크기에 따라 직접 조정해 주어야 할 필요가 있음을 알 수 있다.