Math 158, Spring 2018
Jo Hardin
Regression Splines & Smoothing/Kernel Splines – R code

1 Standard Errors on Different Models

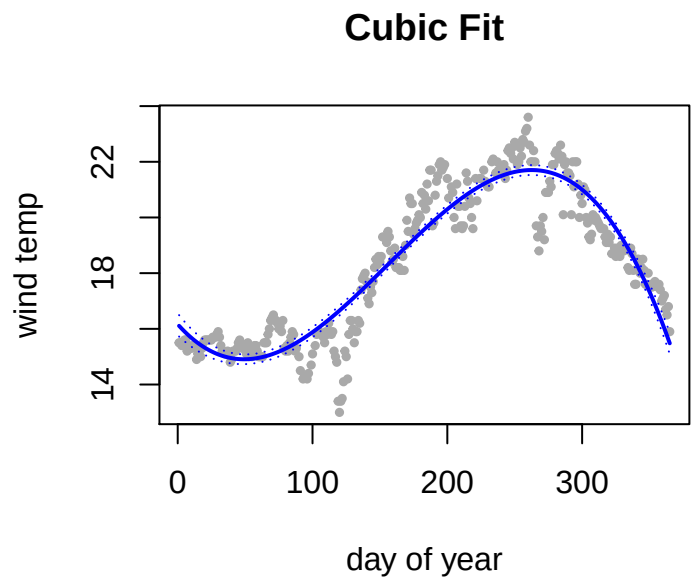
First we scrape some weather data from NOAA. The resulting data we will use is wind temperature at noon across the day of the year (for 2014) measured by a buoy off the coast of Santa Monica.

```
buoy_url <- "http://www.ndbc.noaa.gov/view_text_file.php?filename=46025h2014.txt.gz&dir=data/historical/stdmet/"
buoy_data <- read_table(buoy_url, skip=2, col_names=FALSE)
temp <- read_table(buoy_url, n_max=1, col_names=FALSE)
temp <- unlist(strsplit(unlist(temp), "\\s+"))
names(buoy_data) <- temp

buoy_data2 <- buoy_data %>%
  mutate(WVHT = ifelse(WVHT==99, NA, WVHT)) %>%
  mutate(DPD = ifelse(DPD==99, NA, DPD)) %>%
  mutate(APD = ifelse(APD==99, NA, APD)) %>%
  mutate(MWD = ifelse(MWD==999, NA, MWD)) %>%
  mutate(PRES = ifelse(PRES==9999, NA, PRES)) %>%
  mutate(DEWP = ifelse(DEWP==99, NA, DEWP)) %>%
  select(-VIS, -TIDE) %>% filter(`#YY`==2014)  %>% filter(hh=="12")

buoy_data2$yearday <- yday(mdy(paste(buoy_data2$MM,buoy_data2$DD,
                                     buoy_data2$`#YY`, sep="-")))
```

1.1 Polynomial Regression and Step Functions

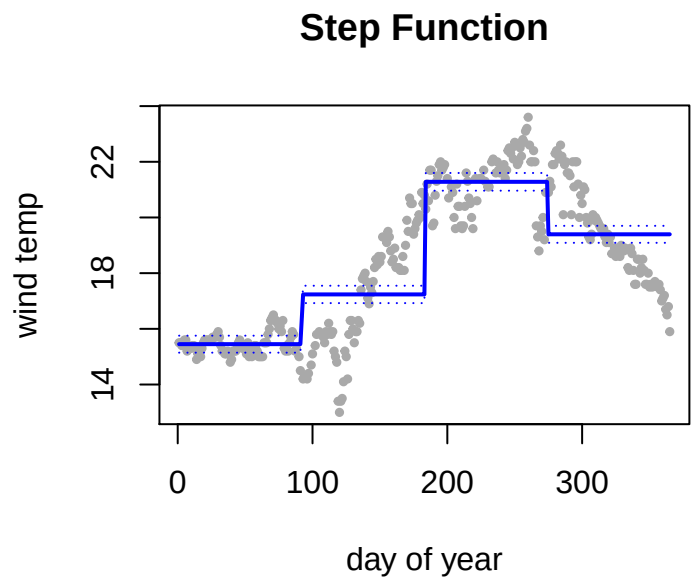
1.1.1 Cubic Model

```
#cubic model
wind.cub <- lm(WTMP ~ poly(yearday,3, raw = TRUE), data=buoy_data2)
summary(wind.cub)$coef

##                                Estimate Std. Error t value  Pr(>|t|)
## (Intercept)                    1.62e+01   1.96e-01    82.3 6.55e-229
## poly(yearday, 3, raw = TRUE)1 -5.44e-02   4.70e-03   -11.6  2.16e-26
## poly(yearday, 3, raw = TRUE)2  6.55e-04   2.99e-05    21.9  3.17e-67
## poly(yearday, 3, raw = TRUE)3 -1.40e-06   5.39e-08   -26.0  2.14e-83

#cubic predictions
wind.cub.pred <- predict(wind.cub, se=TRUE)
wind.cub.se <- cbind(wind.cub.pred$fit +2* wind.cub.pred$se.fit,
                     wind.cub.pred$fit -2*wind.cub.pred$se.fit)

#cubic plot
daylims <- range(buoy_data2$yearday)
plot(buoy_data2$yearday, buoy_data2$WTMP ,xlim=daylims ,cex =.5, pch=19,
     col =" darkgrey ", xlab="day of year", ylab="wind temp")
title("Cubic Fit",outer =F)
lines(buoy_data2$yearday, wind.cub.pred$fit ,lwd =2, col =" blue")
matlines(buoy_data2$yearday, wind.cub.se ,lwd =1, col =" blue",lty =3)
```



1.1.2 Step Functions

```
#cutting the yearday variable
table(cut(buoy_data2$yearday, 4))

##
## (0.636,92]   (92,183]  (183,274]  (274,365]
##         90         86         83         90

wind.step <- lm(WTMP ~ cut(yearday, 4), data=buoy_data2)
summary(wind.step)$coef

##                           Estimate Std. Error t value  Pr(>|t|)
## (Intercept)                  15.45      0.153  101.22 1.29e-258
## cut(yearday, 4)(92,183]       1.79      0.218    8.18  5.32e-15
## cut(yearday, 4)(183,274]      5.83      0.220   26.46  4.86e-85
## cut(yearday, 4)(274,365]      3.95      0.216   18.28  1.12e-52

#step function predictions
wind.step.pred <- predict(wind.step, se=TRUE)
wind.step.se <- cbind(wind.step.pred$fit +2* wind.step.pred$se.fit,
                      wind.step.pred$fit -2*wind.step.pred$se.fit)

#step functions plot
plot(buoy_data2$yearday, buoy_data2$WTMP ,xlim=daylims ,cex =.5, pch=19,
     col =" darkgrey ", xlab="day of year", ylab="wind temp")
title("Step Function",outer =F)
lines(buoy_data2$yearday, wind.step.pred$fit ,lwd =2, col =" blue")
matlines(buoy_data2$yearday, wind.step.se ,lwd =1, col =" blue",lty =3)
```



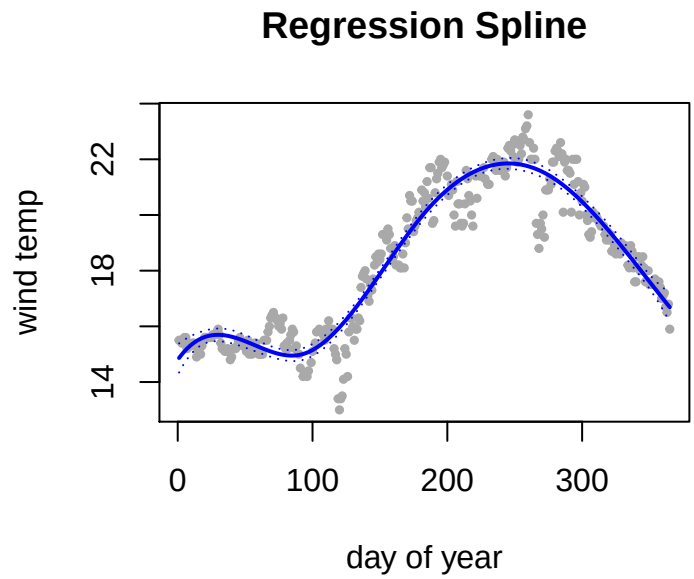
2 Smooth Curves

2.1 Regression Spline

Note that the function `bs` calculate "B-Spline Basis for Polynomial Splines." You can learn more by `?bs` in the `splines` package. The `knots` argument gives "the internal breakpoints that define the spline." The `degree` is the degree of the polynomial.

```
require(splines)
year.knot1 <- bs(buoy_data2$yearday, knots=c(92,183,274), degree=3)

wind.rs <- lm(WTMP ~ year.knot1, data=buoy_data2)
summary(wind.rs)$coef

##              Estimate Std. Error t value  Pr(>|t|)
## (Intercept)    14.86      0.261   56.92 1.70e-176
## year.knot11     1.98      0.488    4.06  6.21e-05
## year.knot12    -2.47      0.321   -7.68  1.69e-13
## year.knot13     6.44      0.391   16.48  2.59e-45
## year.knot14     7.91      0.359   22.03  1.37e-67
## year.knot15     3.64      0.406    8.97  2.05e-17
## year.knot16     1.82      0.363    5.02  8.37e-07

wind.rs.pred <- predict(wind.rs, se=TRUE)
wind.rs.se <- cbind(wind.rs.pred$fit +2* wind.rs.pred$se.fit,
                    wind.rs.pred$fit -2*wind.rs.pred$se.fit)

plot(buoy_data2$yearday, buoy_data2$WTMP ,xlim=daylims ,cex =.5, pch=19,
```

```
     col =" darkgrey ", xlab="day of year", ylab="wind temp")
title("Regression Spline",outer =F)
lines(buoy_data2$yearday, wind.rs.pred$fit ,lwd =2, col =" blue")
matlines(buoy_data2$yearday, wind.rs.se ,lwd =1, col =" blue",lty =3)
```



2.1.1 degrees of freedom

Note that we can fit the model based on placing the knots or based on the number of knots (specified by degrees of freedom which places df-1 internal knots). Consider the following call to our model. Note that there are at least three different ways to think about "degrees of freedom."

- `df` = the number in the argument of the function. Here the number is 6 = # coefficients - 1 = # "explanatory variables"
- df as defined in your text = the number of explanatory variables you are estimating. Note that ISLR generally defines p as the number of explanatory variables, ALSM defines p to be the number of parameters.
- The remaining degrees of freedom (n - # coefficients). Here that number is 349 - 7 = 342. (This last version of "df" is how we are used to thinking about degrees of freedom – how much information do you have to estimate variability?)
- Also note that the `knots` argument overrides the `df` argument.

```
year.knot2 <- bs(buoy_data2$yearday, df=6, degree=3)
wind.rs2 <- lm(WTMP ~ year.knot2, data=buoy_data2)
summary(wind.rs2)
```

```
##
## Call:
## lm(formula = WTMP ~ year.knot2, data = buoy_data2)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0004 -0.4008  0.0002  0.5017  1.8364
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   14.848      0.266   55.76  < 2e-16 ***
## year.knot21    1.999      0.489    4.09  5.4e-05 ***
## year.knot22   -2.445      0.325   -7.53  4.5e-13 ***
## year.knot23    6.402      0.399   16.05  < 2e-16 ***
## year.knot24    8.004      0.362   22.10  < 2e-16 ***
## year.knot25    3.518      0.407    8.64  < 2e-16 ***
## year.knot26    1.874      0.369    5.07  6.5e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.794 on 342 degrees of freedom
## Multiple R-squared:  0.91,Adjusted R-squared:  0.909
## F-statistic:  577 on 6 and 342 DF,  p-value: <2e-16
```

```
wind.rs1 <- lm(WTMP ~ year.knot1, data=buoy_data2) # is very close to above!
summary(wind.rs1)$coef
```

```
##             Estimate Std. Error t value  Pr(>|t|)
## (Intercept)    14.86      0.261   56.92 1.70e-176
## year.knot11     1.98      0.488    4.06  6.21e-05
## year.knot12    -2.47      0.321   -7.68  1.69e-13
## year.knot13     6.44      0.391   16.48  2.59e-45
## year.knot14     7.91      0.359   22.03  1.37e-67
## year.knot15     3.64      0.406    8.97  2.05e-17
## year.knot16     1.82      0.363    5.02  8.37e-07
```

2.1.2 Choosing df with LOOCV

Below is code for using LOOCV to determine the optimal number of knots (as a function of df). Note that the same code could be used for choosing the degree of the polynomial or both the polynomial degree and the number of knots.

```
sse.rs.cv <- c()

for (df.cv in 4:20){
sse.rs.cvi <- 0

    for(i in 1:nrow(buoy_data2)){
    wind.rs.cv <- lm(WTMP ~ bs(yearday, df = df.cv, degree=3),
```

```
                    data=buoy_data2, subset = c(1:nrow(buoy_data2))[-i] )
    wind.rs.cv.pred <- predict(wind.rs.cv,
                               newdata = data.frame(yearday = buoy_data2$yearday[i]), se=FALSE)
    sse.rs.cvi <- sse.rs.cvi + (buoy_data2[i,]$WTMP - wind.rs.cv.pred)^2
  }
  sse.rs.cv <- c(sse.rs.cv, sse.rs.cvi)
}

plot(c(4:20), sse.rs.cv, xlab="df", ylab="LOOCV SSE", type="l", main="Regression Splines")
```

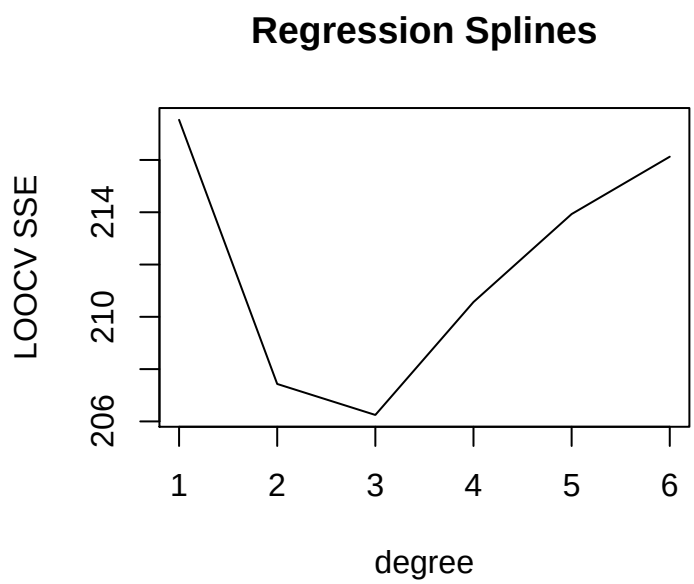

```
c(4:20)[which.min(sse.rs.cv)]

## [1] 20
```

2.1.3 Choosing degree with LOOCV

Equivalent code can be used to find the degree of the polynomial.

```
sse.rs.cv <- c()

for (degree.cv in 1:6){
sse.rs.cvi <- 0

    for(i in 1:nrow(buoy_data2)){
    wind.rs.cv <- lm(WTMP ~ bs(yearday, df = 8, degree=degree.cv),
                    data=buoy_data2, subset = c(1:nrow(buoy_data2))[-i] )
    wind.rs.cv.pred <- predict(wind.rs.cv,
```

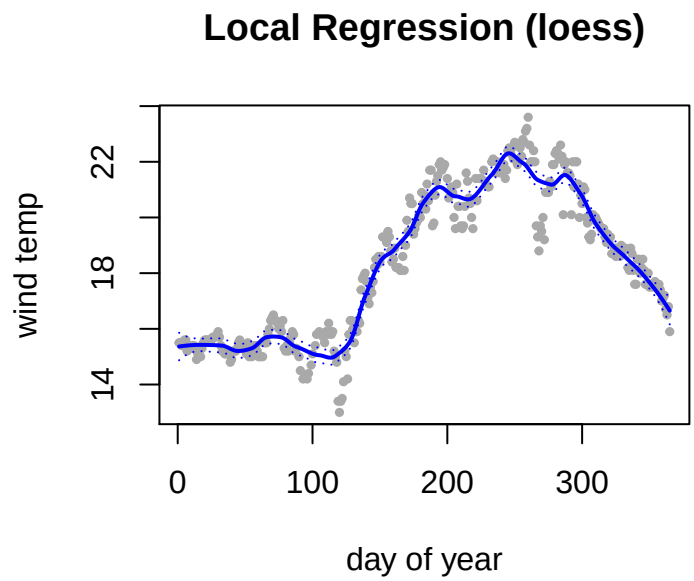
```
                                newdata = list(yearday = buoy_data2$yearday[i]), se=FALSE)
    sse.rs.cvi <- sse.rs.cvi + (buoy_data2[i,]$WTMP - wind.rs.cv.pred)^2
  }
  sse.rs.cv <- c(sse.rs.cv, sse.rs.cvi)
}

plot(c(1:6), sse.rs.cv, xlab="degree", ylab="LOOCV SSE", type="l", main="Regression Splines")
```




```
c(1:6)[which.min(sse.rs.cv)]

## [1] 3
```

2.2 Local Regression (loess)

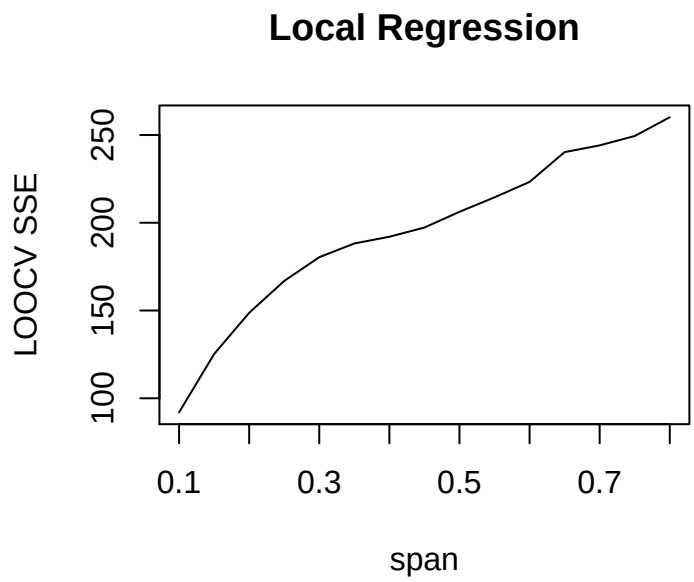
```
wind.lor <- loess(WTMP ~ yearday, span=.2, data=buoy_data2)

wind.lor.pred <- predict(wind.lor, se=TRUE)
wind.lor.se <- cbind(wind.lor.pred$fit +2* wind.lor.pred$se.fit,
                     wind.lor.pred$fit -2*wind.lor.pred$se.fit)


plot(buoy_data2$yearday, buoy_data2$WTMP ,xlim=daylims ,cex =.5, pch=19,
     col =" darkgrey ", xlab="day of year", ylab="wind temp")
title("Local Regression (loess)",outer =F)
lines(buoy_data2$yearday, wind.lor.pred$fit ,lwd =2, col =" blue")
matlines(buoy_data2$yearday, wind.lor.se ,lwd =1, col =" blue",lty =3)
```



2.2.1 Choosing the span with CV

Like with regression splines, cross validation can be used to set the span for loess. Note that loess does not support extrapolation, so we cannot predict for the smalles and largest points in the dataset if they are left out. The code below works easily for us because

```
which.min(buoy_data2$yearday)

## [1] 1

which.max(buoy_data2$yearday)

## [1] 349

sse.loess.cv <- c()

for (span.cv in seq(0.1, 0.8, 0.05)){
sse.loess.cvi <- 0

    for(i in c(1:nrow(buoy_data2))[-c(1,349)]){  # can't predict for min and max obs
    wind.loess.cv <- loess(WTMP ~ yearday, span=span.cv,
                           data=buoy_data2, subset = c(1:nrow(buoy_data2))[-i])
    wind.loess.cv.pred <- predict(wind.loess.cv,
                                  newdata = data.frame(yearday = buoy_data2$yearday[i]), se=FALSE)
    sse.loess.cvi <- sse.loess.cvi + (buoy_data2[i,]$WTMP - wind.loess.cv.pred)^2
  }
  sse.loess.cv <- c(sse.loess.cv, sse.loess.cvi)
}

plot(seq(0.1, 0.8, 0.05), sse.loess.cv, xlab="span", ylab="LOOCV SSE",
     type="l", main="Local Regression")
```




```r
seq(0.1, 0.8, 0.05)[which.min(sse.loess.cv)]

## [1] 0.1
```

3 Imputing using Local Regression (loess)

This TechNote[1] shows examples of loess (local polynomial regression fitting) smoothing for various "span" values. The online R documentation (?loess) says the default span value is 0.75, but doesn't give much guidance, nor visual examples, of how the span value affects smoothing. In addition to simply smoothing a curve, the R loess function can be used to impute missing data points. An example of data imputation with loess is shown.

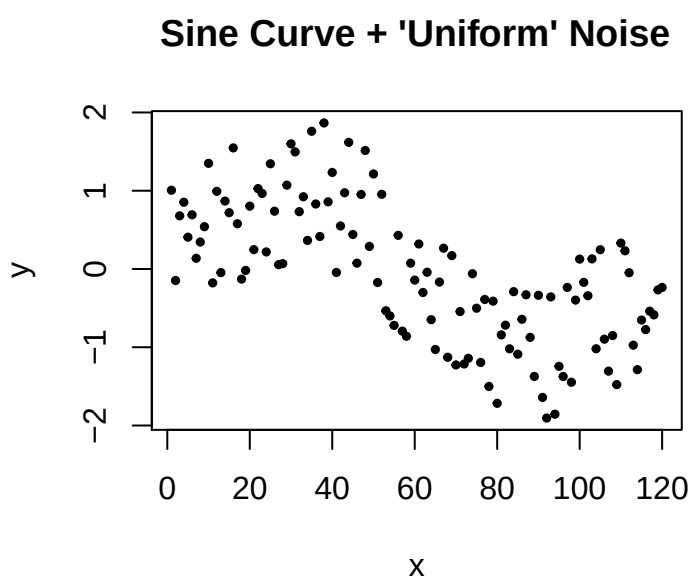
Let's take a sine curve, add some "noise" to it, and then see how the loess "span" parameter affects the look of the smoothed curve.

1. Create a sine curve and add some noise:

```
set.seed(47)
period <- 120
x <- 1:120
y <- sin(2*pi*x/period) + runif(length(x),-1,1)
```

2. Plot the points on this noisy sine curve:

```
plot(x,y, main="Sine Curve + 'Uniform' Noise", pch=19, cex=.5)
```



3. Apply loess smoothing using the default span value of 0.75:

[1] From http://research.stowers-institute.org/efg/R/Statistics/loess.htm

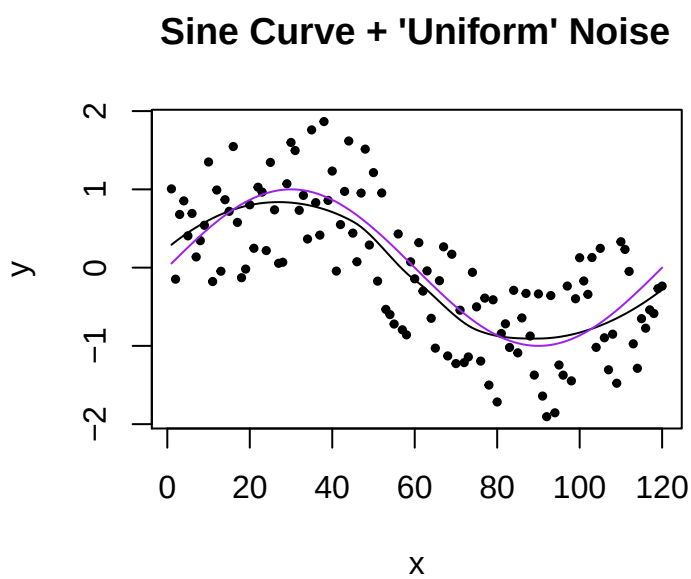
```
y.loess <- loess(y ~ x, span=0.75)
```

4. Compute loess smoothed values for all points along the curve:

```
y.predict <- predict(y.loess, data.frame(x=x))
```

5. Plot the loess smoothed curve along with the points that were already plotted:

```
plot(x,y, main="Sine Curve + 'Uniform' Noise", pch=19, cex=.5)
lines(x,y.predict)
lines(x, sin(2*pi*x/period), col="purple")
```

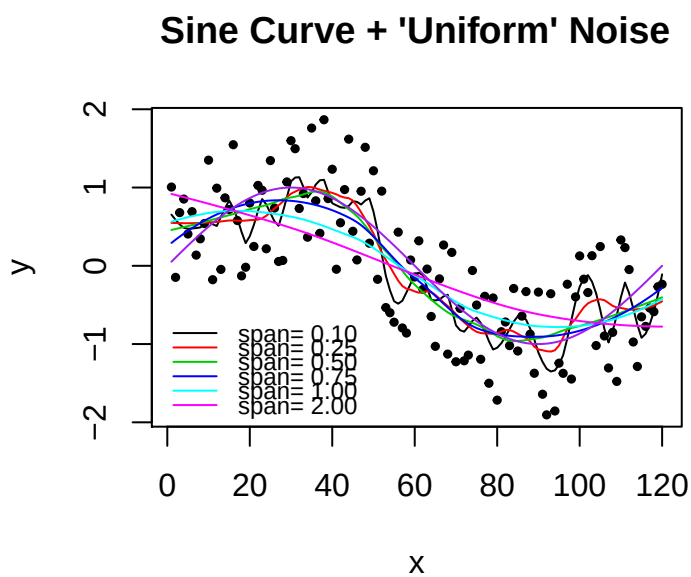


6. Repeat steps 1-5 above for various span values. A script was created to automate this. Run this script by entering the following R statement:[2]

```
set.seed(47)
plot(x,y, main="Sine Curve + 'Uniform' Noise", pch=19, cex=.5)
spanlist <- c(0.10, 0.25, 0.50, 0.75, 1.00, 2.00)
for (i in 1:length(spanlist))
{
  y.loess <- loess(y ~ x, span=spanlist[i], data.frame(x=x, y=y))
  y.predict <- predict(y.loess, data.frame(x=x))

  # Plot the loess smoothed curve
```

[2] http://research.stowers-institute.org/efg/R/Statistics/loess-sin+runif.R

```
  lines(x,y.predict,col=i)

}

legend ("bottomleft",y.intersp=.5,cex=.75,
        c(paste("span=", formatC(spanlist, digits=2, format="f"))),
        lty=SOLID<-1, col=1:length(spanlist), bty="n")

lines(x, sin(2*pi*x/period), col="purple")
```

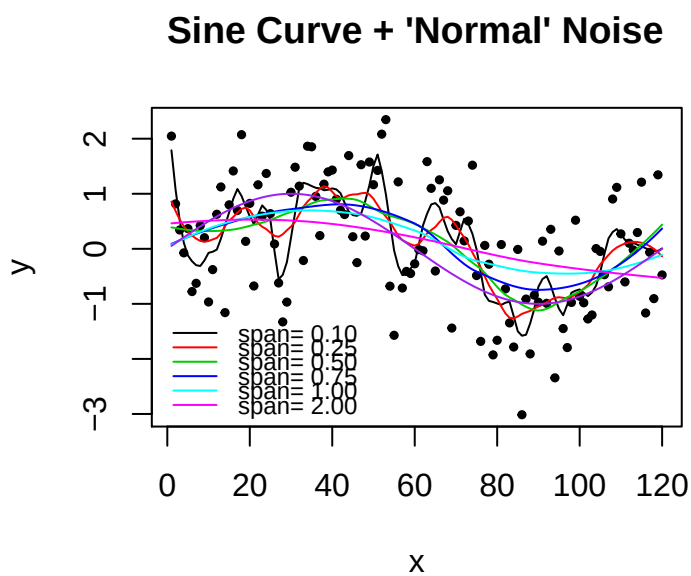


7. Compare "noise" from a uniform distribution from -1 to 1 (above) to Gaussian noise, with mean 0 and standard deviation 1.0 (below):[3]

```
set.seed(47)
period <- 120;
# Create sine curve with noise
x <- 1:120
y <- sin(2*pi*x/period) + rnorm(length(x))

# Plot points on noisy curve
plot(x,y, main="Sine Curve + 'Normal' Noise", pch=19, cex=.5)

spanlist <- c(0.10, 0.25, 0.50, 0.75, 1.00, 2.00)
for (i in 1:length(spanlist))
{
  y.loess <- loess(y ~ x, span=spanlist[i], data.frame(x=x, y=y))
  y.predict <- predict(y.loess, data.frame(x=x))
```

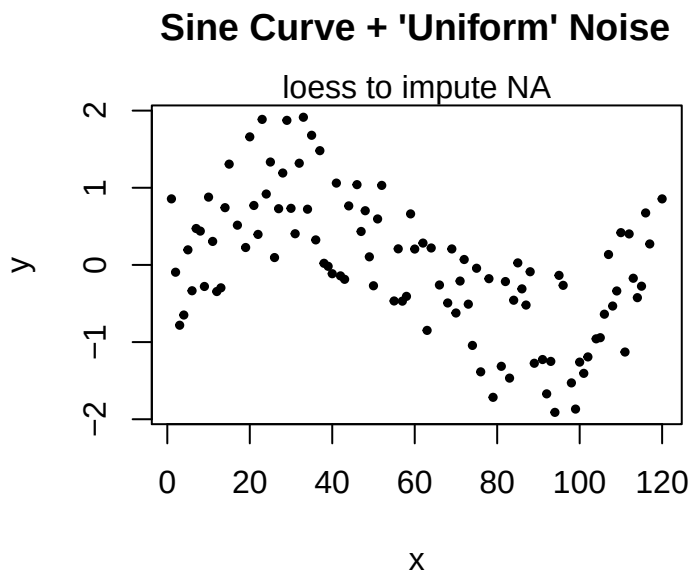
[3] http://research.stowers-institute.org/efg/R/Statistics/loess-sin+rnorm.R

```
  # Plot the loess smoothed curve
  lines(x,y.predict,col=i)
}

legend ("bottomleft",y.intersp=.5,cex=.75,
        c(paste("span=", formatC(spanlist, digits=2, format="f"))),
        lty=1, col=1:length(spanlist), bty="n")

lines(x, sin(2*pi*x/period), col="purple")
```



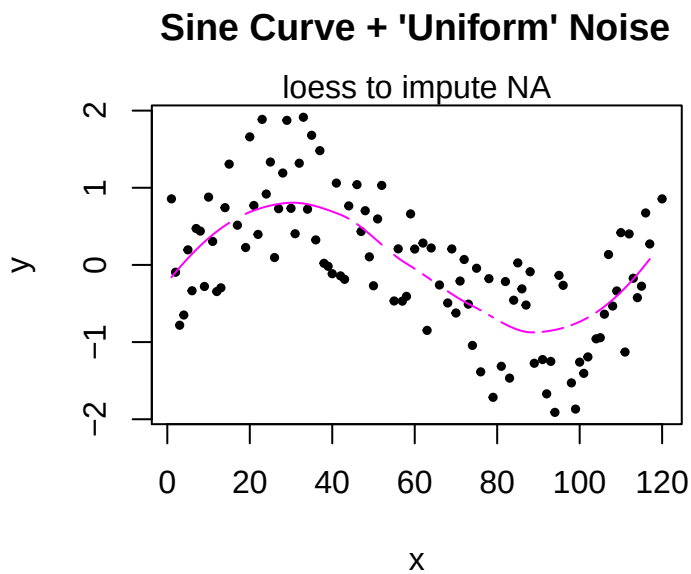
8. Let's use loess to impute data points. Let's start by taking a sine curve with noise, like computed above, but leave out 15 of the 120 data points using R's "sample" function:

```
set.seed(47)
period <- 120
FullList <- 1:120
x <- FullList

 # "randomly" make 15 of the points "missing"
 MissingList <- sample(x,15)
 x[MissingList] <- NA

 # Create sine curve with noise
 y <- sin(2*pi*x/period) + runif(length(x),-1,1)

 # Plot points on noisy curve
 plot(x,y, main="Sine Curve + 'Uniform' Noise", pch=19, cex=.5)
 mtext("loess to impute NA")
```



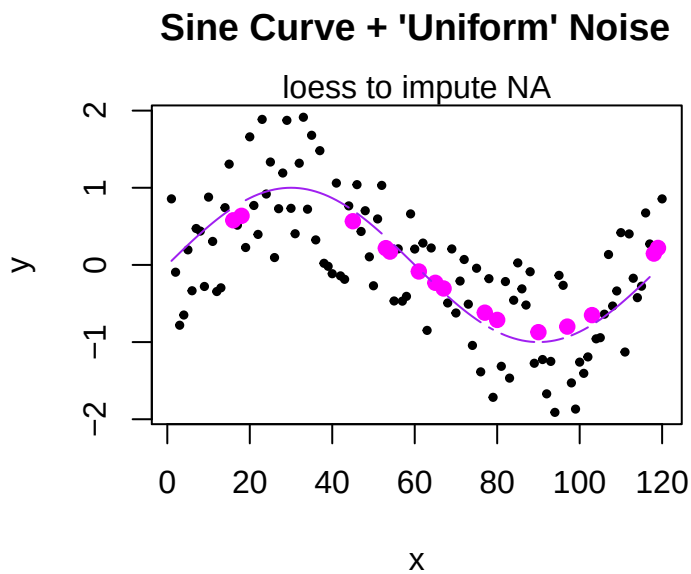
9. As before, use the loess and predict functions to get smoothed values at the defined points:

```
y.loess <- loess(y ~ x, span=0.75, data.frame(x=x, y=y))
y.predict <- predict(y.loess, data.frame(x=FullList))

# Plot the loess smoothed curve showing gaps for missing data
plot(x,y, main="Sine Curve + 'Uniform' Noise", pch=19, cex=.5)
mtext("loess to impute NA")
lines(x,y.predict,col=i)
```



10. Use the loess and predict functions to also impute the values at the missing points:

```
# Show imputed points to fill in gaps
y.Missing <-  predict(y.loess, data.frame(x=MissingList))

 plot(x,y, main="Sine Curve + 'Uniform' Noise", pch=19, cex=.5)
mtext("loess to impute NA")
points(MissingList, y.Missing, pch=FILLED.CIRCLE<-19, col=i)
lines(x, sin(2*pi*x/period), col="purple")
```

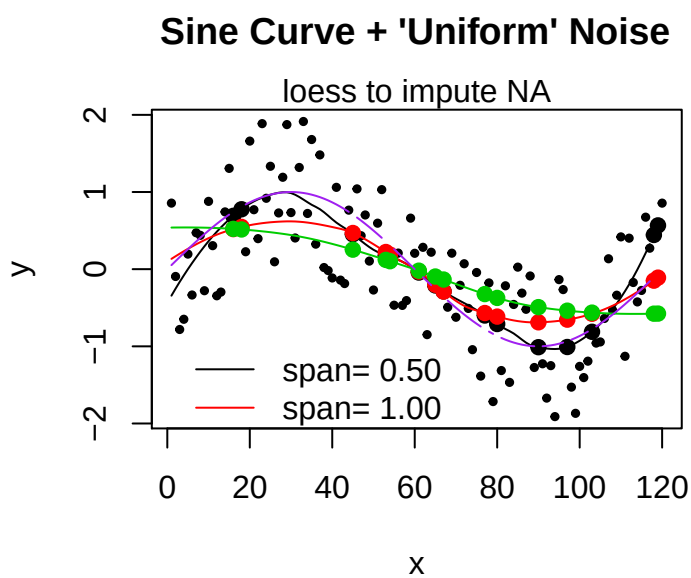


11. Compare the loess smoothed fit and imputed points for various span values:[4]

```
set.seed(47)
period <- 120
FullList <- 1:120
x <- FullList

# "randomly" make 15 of the points "missing"
MissingList <- sample(x,15)
x[MissingList] <- NA

# Create sine curve with noise
y <- sin(2*pi*x/period) + runif(length(x),-1,1)

# Plot points on noisy curve
plot(x,y, main="Sine Curve + 'Uniform' Noise", pch=19, cex=.5)
mtext("loess to impute NA")

spanlist <- c(0.50, 1.00, 2.00)
for (i in 1:length(spanlist))
{
  y.loess <- loess(y ~ x, span=spanlist[i], data.frame(x=x, y=y))
  y.predict <- predict(y.loess, data.frame(x=FullList))

  # Plot the loess smoothed curve showing gaps for missing data
  lines(x,y.predict,col=i)

  # Show imputed points to fill in gaps
```

[4] http://research.stowers-institute.org/efg/R/Statistics/loess-sin+runif-impute.R

```
  y.Missing <-  predict(y.loess, data.frame(x=MissingList))
  points(MissingList, y.Missing, pch=FILLED.CIRCLE<-19, col=i)
}

legend (0,-0.8,
        c(paste("span=", formatC(spanlist, digits=2, format="f"))),
        lty=SOLID<-1, col=1:length(spanlist), bty="n")
lines(x, sin(2*pi*x/period), col="purple")
```



3.1 Discussion/Conclusion

Span values as small as 0.10 do not provide much smoothing and can result in a "jerky" curve. Span values as large as 2.0 provide perhaps too much smoothing, at least in the cases shown above. Overall, the default value of 0.75 worked fairly well in "finding" the sine curve.