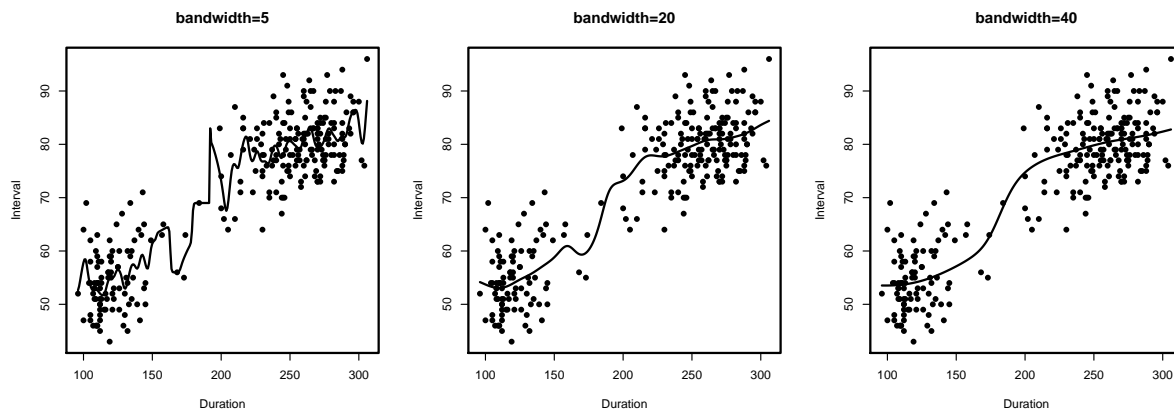


## Stat 8053, Fall 2011: Smoothing, Ch. 11

### Kernel smoothing

Kernel smoothing is essentially *weighted local averaging*. It balances bias and variability using a smoothing parameter that essentially controls how many points get high weight in the averaging.

```
> library(alr3)
> par(pch = 20, lwd = 2)
> oldpar <- par(mfrow = c(1, 3))
> with(oldfaith, {
+   plot(Interval ~ Duration, main = "bandwidth=5")
+   lines(ksmooth(Duration, Interval, "normal", 5))
+   plot(Interval ~ Duration, main = "bandwidth=20")
+   lines(ksmooth(Duration, Interval, "normal", 20))
+   plot(Interval ~ Duration, main = "bandwidth=40")
+   lines(ksmooth(Duration, Interval, "normal", 40))
+ })
> par(oldpar)
```



I first set `pch=20` to use small circles for plotting symbols and `lwd=2` for thicker lines. I set the parameter `mfrow` for three graphs in one row in a separate command. The third call to `par` at the end resets the `mfrow` parameter but not the others, so the default symbol and its size are set globally for all the graphs in this handout. I used `with` to specify that all the functions inside the brackets refer to data in the `oldfaith` data frame. `plot` draws the plots, and `lines` draws the lines, with `ksmooth` determining the coordinates of the lines. "normal" is the name of the kernel function, and the last argument is the smoothing parameter.

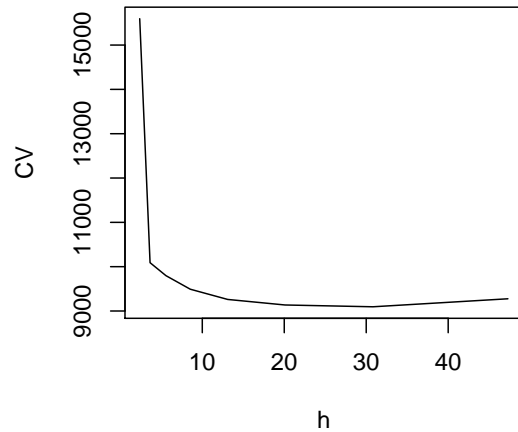
The function `hcv` in the `sm` package uses a brute-force method with cross validation to get a smoothing parameter, and optionally draws a figure.

```
> library(sm)
> system.time(with(oldfaith, hcv(Duration, Interval)))

  user  system elapsed
 0.55   0.07   0.63

> with(oldfaith, hcv(Duration, Interval, display = "lines"))

[1] 26.91588
```

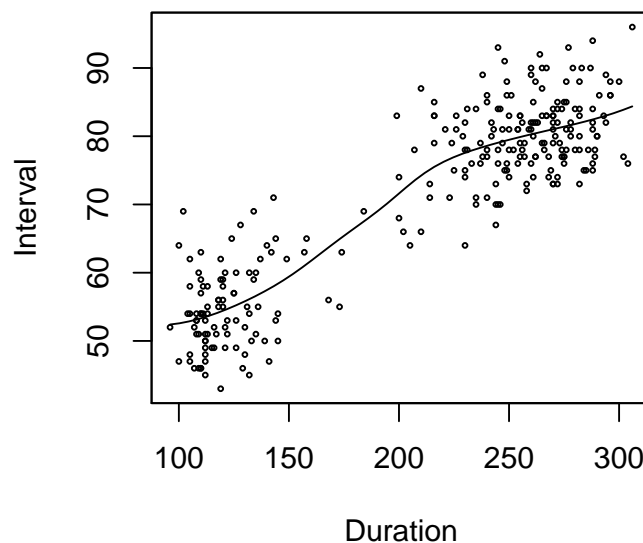


Newer code in *sm* computes cross-validation much more efficiently. While the above plot can't be drawn, nearly the same solution is obtained

```
> system.time(with(oldfaith, print(h.select(Duration,
+ Interval, method = "cv"))))
```

```
[1] 23.90918
user system elapsed
0.02 0.00 0.03
```

```
> with(oldfaith, sm.regression(Duration, Interval, method = "cv"))
```



Unlike most of R, *sm* functions have some limited interactive capability if you install the *rpanel* package and **its dependencies**,

```
> install.packages("rpanel", dependencies = TRUE)
```

For a more interesting graph, which you should try, add the `panel` argument for interactive capability:

```
> library(rpanel)
> with(oldfaith, sm.regression(Duration, Interval, method = "cv",
+   panel = TRUE))
```

This gives a graph with sliders to see the effects of various choices of the smoothing parameter.

## loess

Loess uses *local regression* rather than local averaging. The default in R is to fit using local quadratic polynomials with a nearest neighbor rule, that is “give the closest  $f \times 100\%$  of the data positive weight” rather than a fixed bandwidth rule “give positive weight to all cases within  $h$  of the point of interest”.

```
> (m1 <- loess(Interval ~ Duration, oldfaith))
```

Call:

```
loess(formula = Interval ~ Duration, data = oldfaith)
```

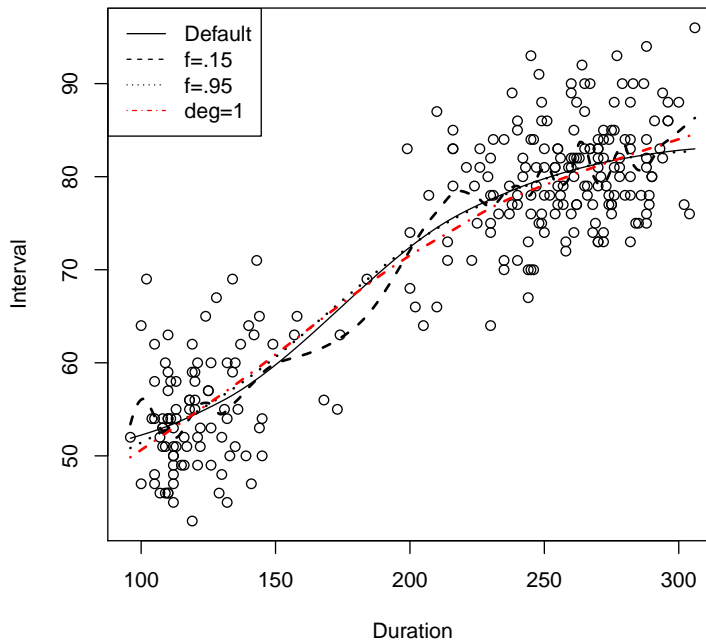
Number of Observations: 270

Equivalent Number of Parameters: 3.97

Residual Standard Error: 5.745

```
> with(oldfaith, {
+   plot(Interval ~ Duration, main = "loess, default is f=.75, deg=1")
+   lines(spline(Duration, fitted(m1)), lty = 1)
+   lines(spline(Duration, fitted(update(m1, span = 0.15))),
+     lty = 2, lwd = 2)
+   lines(spline(Duration, fitted(update(m1, span = 0.95))),
+     lty = 3, lwd = 2)
+   lines(spline(Duration, fitted(update(m1, degree = 1))),
+     lty = 4, lwd = 2, col = "red")
+ })
> legend("topleft", c("Default", "f=.15", "f=.95", "deg=1"),
+   lty = 1:4, col = c("black", "black", "black",
+     "red"))
```

loess, default is f=.75, deg=1



loess has a predict method:

```
> predict(m1, newdata = data.frame(Duration = 50 * (1:8)))
```

```
   1     2     3     4     5     6     7
NA 52.23175 59.77834 72.42780 79.78690 82.83174 NA
  8
NA
```

The NAs are returned because loess doesn't extrapolate.

## Smoothing Splines

Smoothing splines are fundamentally different from the first two methods. Assuming all values of  $x_i$  are unique and ordered, we draw a (curved) line between  $x_i$  to  $x_{i+1}$ , where the curvature is controlled by a balance between small error at the data, which would require  $\hat{f}(x_i) = y_i$ , and smoothness of the function, so the function doesn't fluctuate wildly between observed  $x_i$ .

```
> print(sm2 <- with(oldfaith, smooth.spline(Duration,
+ Interval)))
```

Call:

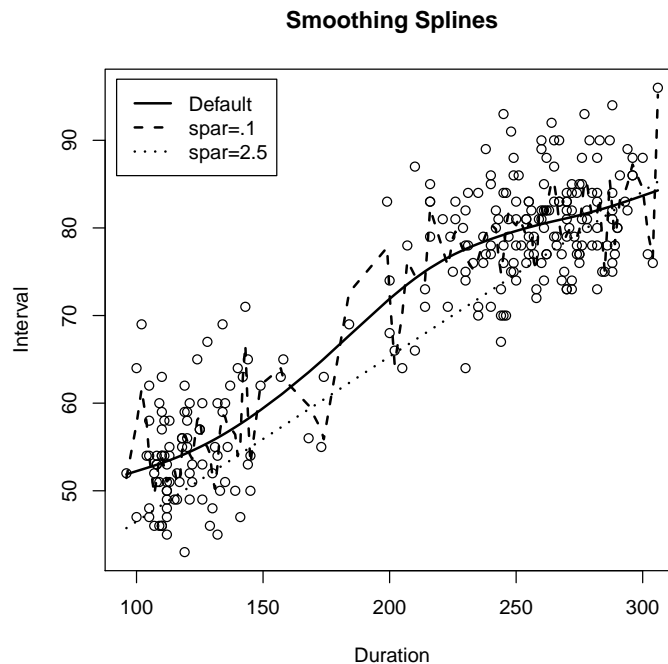
```
smooth.spline(x = Duration, y = Interval)
```

```
Smoothing Parameter spar= 0.9067774 lambda= 0.00955773 (12 iterations)
Equivalent Degrees of Freedom (Df): 5.120469
Penalized Criterion: 4495.873
GCV: 33.49772
```

```

> with(oldfaith, {
+   plot(Interval ~ Duration, main = "Smoothing Splines")
+   lines(sm2, lty = 1, lwd = 2)
+   lines(update(sm2, spar = 0.1), lty = 2, lwd = 2)
+   lines(update(sm2, spar = 2.5), lty = 3, lwd = 2)
+ })
> legend("topleft", c("Default", "spar=.1", "spar=2.5"),
+       lwd = 2, lty = 1:3, inset = 0.02)

```



## Regression splines

Regression splines approximate  $f(x)$  as a linear combination of basis functions with unknown weights. A simple expression of the idea is approximating  $f(x)$  as a sum of powers of  $x$ . For this example, begin by standardizing `Duration` to have mean zero and range from  $-1$  to  $+1$ .

```

> mid <- with(oldfaith, (min(Duration) + max(Duration))/2)
> StdD <- with(oldfaith, (Duration - mid)/max(Duration -
+   mid))
> x <- seq(-1, 1, length = 201)

```

I'm going to draw two plots. The left-plot is just the polynomial basis functions on the interval  $(-1, 1)$ . The right plot of `Interval` against the standardized `Duration`, showing the quartic regression fit. Also shown on this plot are the five components of the quartic fit,  $\hat{\beta}_0 \times 1, \hat{\beta}_1 x, \dots, \hat{\beta}_4 x^4$ ; the quartic fit is just the sum of these curves.

```

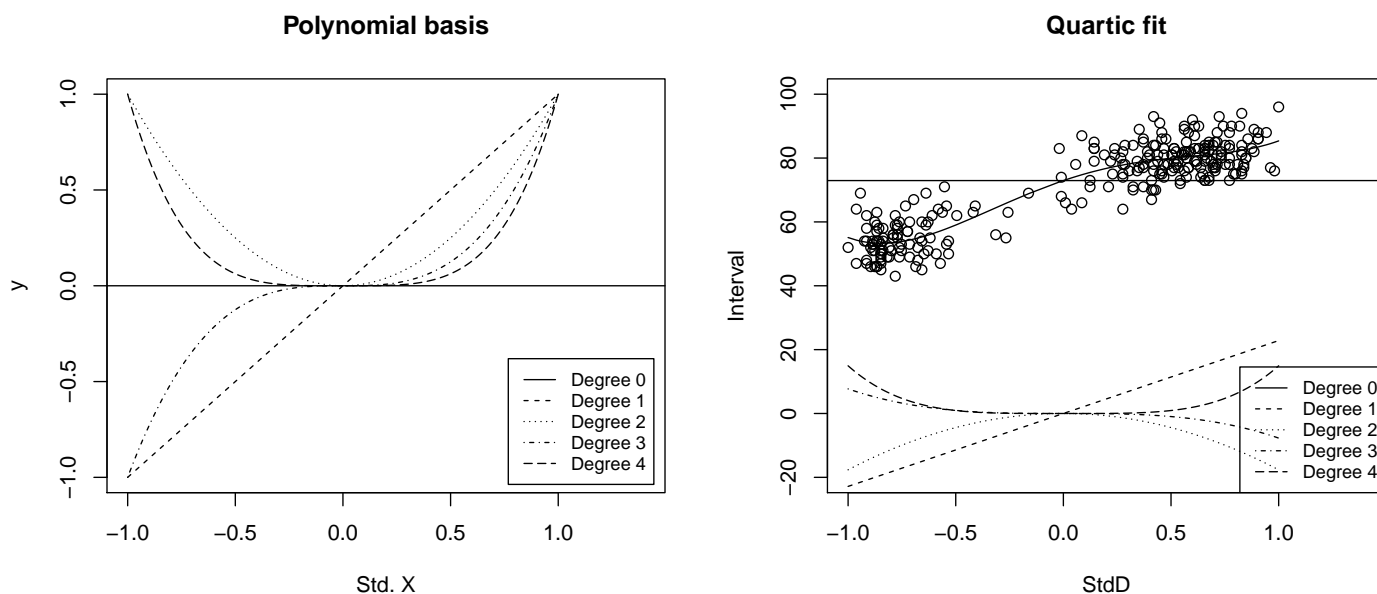
> par(mfrow = c(1, 2))
> plot(x, x, type = "l", lty = 2, xlab = "Std. X", ylab = "y",
+   main = "Polynomial basis", ylim = c(-1, 1), xlim = c(-1,
+   1.4))

```

```

> abline(h = 0, lty = 1)
> lines(x, x^2, lty = 3)
> lines(x, x^3, lty = 4)
> lines(x, x^4, lty = 5)
> legend("bottomright", paste("Degree", 0:4), lty = 1:5,
+       cex = 0.8, inset = 0.02)
> lm1 <- lm(Interval ~ poly(StdD, 4, raw = TRUE), oldfaith)
> plot(Interval ~ StdD, oldfaith, main = "Quartic fit",
+      ylim = c(-20, 100), xlim = c(-1, 1.4))
> lines(sort(StdD), predict(lm1, newdata = data.frame(StdD = sort(StdD))))
> b <- coef(lm1)
> lines(x, b[2] * x, lty = 2)
> abline(h = b[1], lty = 1)
> lines(x, b[3] * x^2, lty = 3)
> lines(x, b[4] * x^3, lty = 4)
> lines(x, b[5] * x^4, lty = 5)
> legend("bottomright", paste("Degree", 0:4), lty = 1:5,
+       cex = 0.8)

```

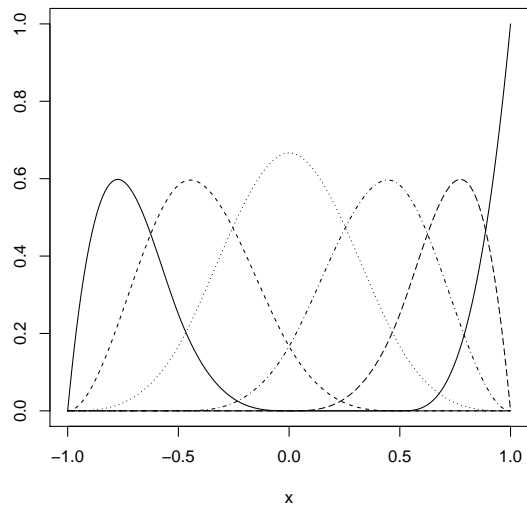


The polynomial basis functions are defined on the whole interval  $(-1, 1)$  and so they may not be very useful for modeling *local* features in a function. Other basis functions can be defined so that they are zero or nearly zero except for a neighborhood of  $(-1, 1)$ . For example, the **B-splines** are nonlinear functions of the range of  $x$ ; the number of knots (or the number of splines) and (3) the placement of the knots. The default behavior of the `bs` function is to select the knots to be equally spaced over the range of  $x$ .

```

> library(splines)
> matplot(x, bs(x, df = 6), type = "l", ylab = "", col = 1)

```



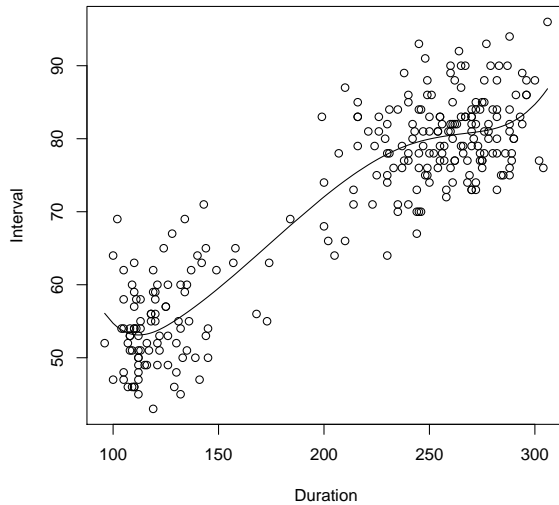
This first argument to `bs` is the value of the predictor, which is used to determine the range. The second default argument is the degrees of freedom. The result of the call to `bs` is a matrix with rows equal to the length of  $x$  and `df` columns. We will only work with `df` to control the values of the splines. One could also work with the number of internal knots, a function of the `df`, and use non-equally spaced knots, which is the default in `bs`.

```
> bspline.predictors <- bs(StdD, 6)
> print(bspline.predictors[1:5, ], digits = 3)

      1      2      3      4      5      6
[1,] 0.00606 0.1736 0.58185 0.238505 0.000 0.00000
[2,] 0.65368 0.0733 0.00205 0.000000 0.000 0.00000
[3,] 0.02806 0.3000 0.54331 0.128617 0.000 0.00000
[4,] 0.47906 0.4396 0.08117 0.000129 0.000 0.00000
[5,] 0.00000 0.0000 0.08846 0.642349 0.268 0.00137

> sm1 <- lm(Interval ~ bs(Duration, 6), oldfaith)
> plot(Interval ~ Duration, oldfaith, main = "Regression splines")
> or <- order(oldfaith$Duration)
> lines(predict(sm1)[or] ~ Duration[or], oldfaith)
```

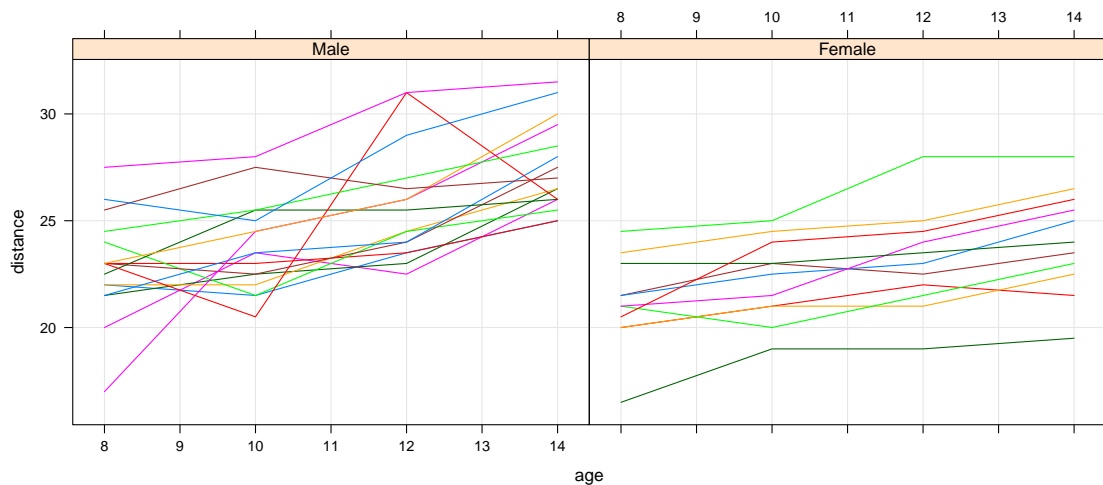
Regression splines



Here is another example, that uses the `Orthodont` data frame from the `nlme` package. The data consist of measures of distance between teeth as a function of age for a number of boys and girls.

```
> data(Orthodont, package = "nlme")
> library(lattice)
> print(xyplot(distance ~ age | Sex, data = Orthodont,
+           group = Subject, type = c("g", "l")))

```



Although the individual growth curves look fairly linear, we will use splines to fit as a function of age. Since the time series are so short, only 7 observations per subject, we can only set `df` to 3, which is the smallest value possible.

```
> library(lme4)
> m1 <- lmer(distance ~ Sex * bs(age, 3) + (1 + bs(age,
+           3) | Subject), data = Orthodont, method = "ML")
> m2 <- lmer(distance ~ bs(age, 3) + (1 + bs(age, 3) |
+           Subject), data = Orthodont, method = "ML")
> anova(m1, m2)

```

```

Data: Orthodont
Models:
m2: distance ~ bs(age, 3) + (1 + bs(age, 3) | Subject)
m1: distance ~ Sex * bs(age, 3) + (1 + bs(age, 3) | Subject)
      Df      AIC      BIC  logLik  Chisq Chi Df Pr(>Chisq)
m2 15 460.20 500.43 -215.10
m1 19 454.51 505.47 -208.25 13.689      4 0.008357

> AIC(m1, m2)

      df      AIC
m1 19 454.5093
m2 15 460.1983

```

We fit separate models for males and females, and draw the population average curves.

```

> ages <- seq(8, 14, length = 100)
> bs3 <- with(Orthodont, bs(age, 3))
> newages <- cbind(1, predict(bs3, ages))
> fitmale <- newages %*% fixef(m1)[c(1, 3:5)]
> fitfemale <- newages[, c(1, 1, 2, 3, 4, 2, 3, 4)] %*%
+   fixef(m1)
> plot(ages, fitmale, type = "l", lwd = 2, col = "red",
+      ylim = c(min(fitmale, fitfemale), max(fitmale,
+      fitfemale)), xlab = "Age", ylab = "Fit distance",
+      main = "Fitted distance")
> lines(ages, fitfemale, type = "l", lwd = 2, col = "blue",
+      lty = 2)
> legend("topleft", inset = 0.01, c("Male", "female"),
+      lty = c(1, 2), lwd = 2, col = c("red", "blue"))

```

Writing  $y$  for the response distance and  $x$  for age,  $f$  for the dummy variable for Sex, and  $b_j(x)$  for the  $j$  b-spline basis function, we have fit

$$E(\widehat{y|x}, f) = \hat{\beta}_0 + \beta_f F + \hat{\beta}_1 b_1(x) + \hat{\beta}_2 b_2(x) + \hat{\beta}_3 b_3(x) + \hat{\beta}_4 b_1(x)f + \hat{\beta}_5 b_2(x)f + \hat{\beta}_6 b_3(x)f$$

Of interest in growth curves is the derivative of the fitted curve, giving the rate of change at each age:

$$\frac{dE(\widehat{y|x}, f)}{dx} = \hat{\beta}_1 b'_1(x) + \hat{\beta}_2 b'_2(x) + \hat{\beta}_3 b'_3(x) + \hat{\beta}_4 b'_1(x)f + \hat{\beta}_5 b'_2(x)f + \hat{\beta}_6 b'_3(x)f$$

where the primes indicate differentiation. If we can compute the derivatives of the b-splines we can get the rate function without much work. It turns out that `bs` contained code that computed the derivatives of the b-splines but did not return them; they were computed for a different purpose. I modified the function to return the derivatives, in a function I call `mybs`.

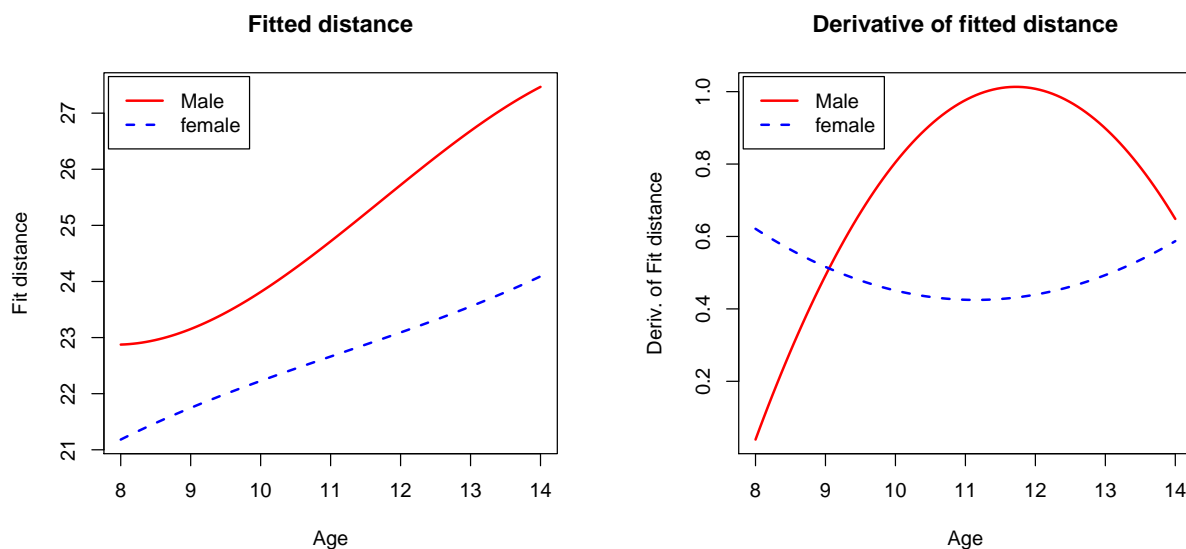
```
> source("http://www.stat.umn.edu/~sandy/courses/8053/Data/mybs.R")
```

To get the first derivative, set up the derivative of the splines for the original data, and then evaluate them at the points that will be used for plotting.

```

> deriv1 <- with(Orthodont, mybs(age, 3, deriv = 1))
> dmale <- predict(deriv1, ages) %*% fixef(m1)[3:5]
> dfemale <- predict(deriv1, ages)[, c(1, 2, 3, 1, 2,
+   3)] %*% fixef(m1)[3:8]
> plot(ages, dmale, type = "l", lwd = 2, col = "red",
+   ylim = c(min(dmale, dfemale), max(dmale, dfemale)),
+   xlab = "Age", ylab = "Deriv. of Fit distance",
+   main = "Derivative of fitted distance")
> lines(ages, dfemale, type = "l", lwd = 2, col = "blue",
+   lty = 2)
> legend("topleft", inset = 0.01, c("Male", "female"),
+   lty = c(1, 2), lwd = 2, col = c("red", "blue"))

```



The graph above at the left could more simply be drawn using

```

> library(effects)
> plot(effect("Sex:bs(age, 3)", m2))

```

but the plot at the right requires building the graph as outlined above.