Splines, Loess, etc

Paul Johnson <pauljohn@ku.edu>

Oct 6, 2011 Updated for newer R, LYX, etc
Orig: Jan 29, 2006

Contents

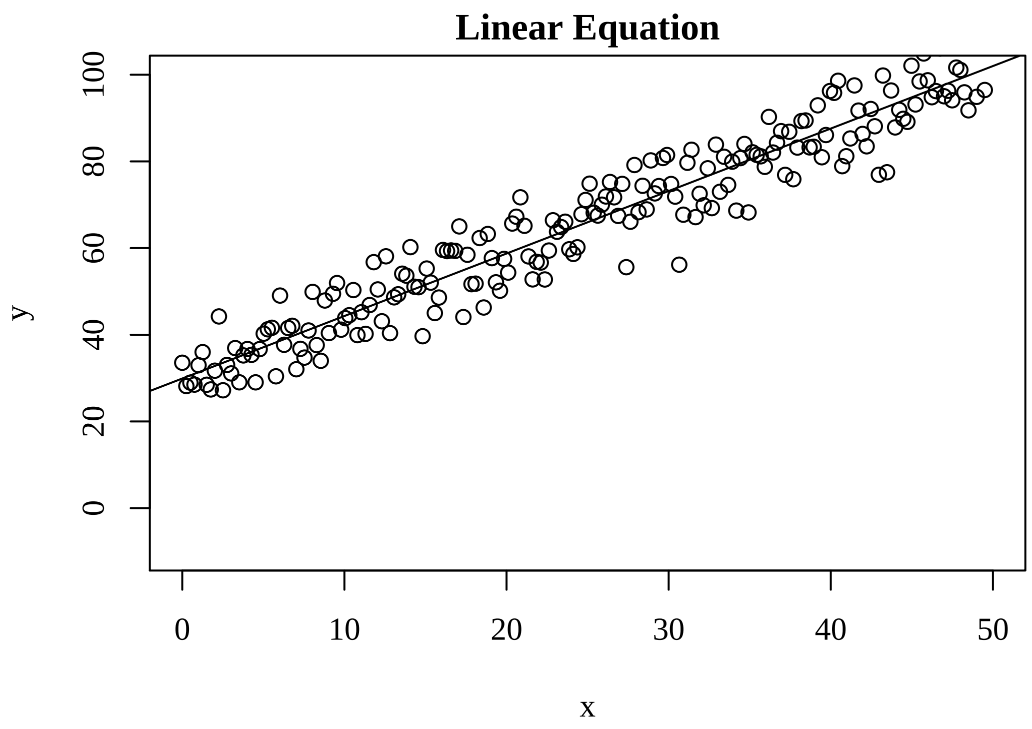
1 Everything good in life is linear

The linear equation is the basis of much social science research. In the following, the "intercept" (or "constant") is 30 and the slope is 1.4.

$$y = 30 + 1.4 \cdot x$$

Note how the slope does not change as x moves from left to right.

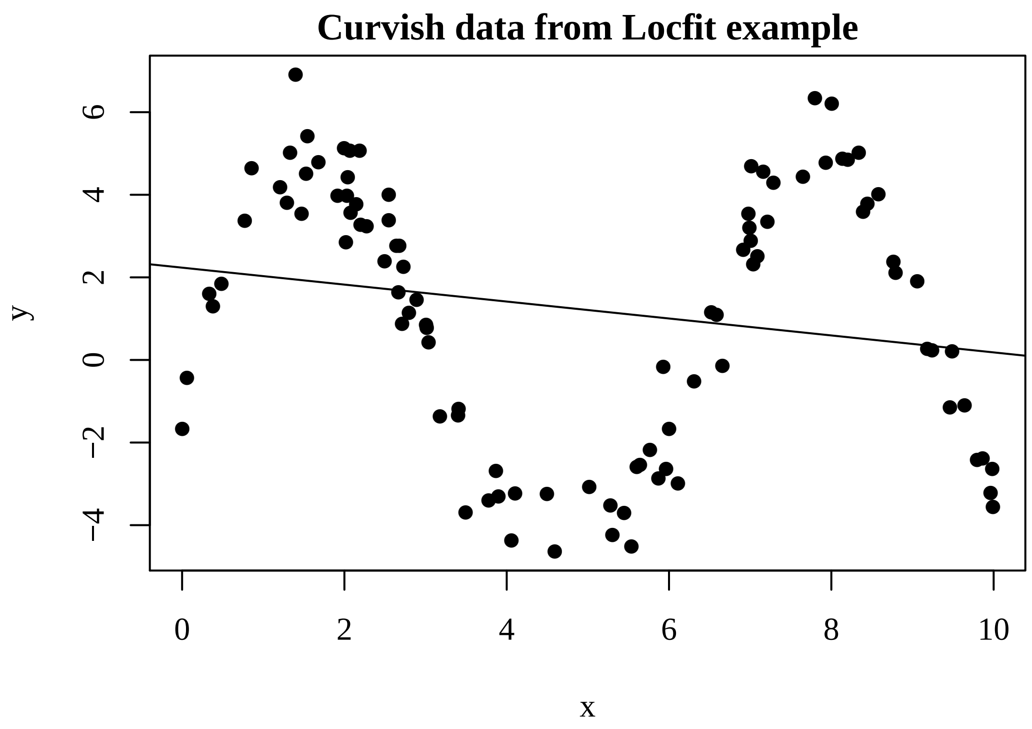
```
x ← seq(from = 0, to = 50, length.out = 200)
y ← 30 + 1.4 * x + 6 * rnorm(200)
plot(x, y, main = "Linear Equation", xlim = c(0, 50), ylim = c(
   −10, 100))
abline(lm(y ∼ x))
```



2 Oops.

What if your data is not a "straight line". Suppose it is created by this process, which I've borrowed from the documentation on the locfit program for R.

```
set.seed(2314)
x <- 10 * runif(100)
y <- 5 * sin(x) + rnorm(100)
plot(x, y, main = "Curvish data from Locfit example", pch = 16)
abline(lm(y ~ x))
```



3 Square that x!

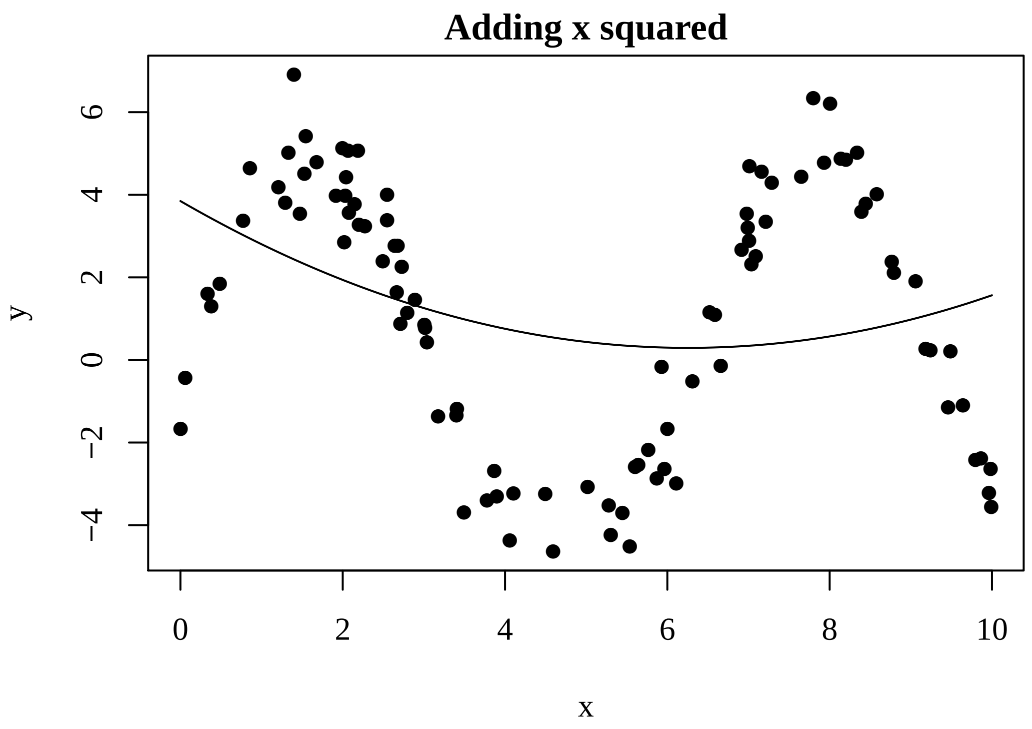
Sometimes people say "there might be nonlinearity" and then they throw in a squared input variable. This is one of the simplest (short-sighted, silly, etc) things to do.

$$y = 3 + 4 \cdot x - 0.09 \cdot x^2$$

If you run a regression with x^2 as an in put variable, and the coefficient is not significantly different from 0, then you are "home free." Right? Well, it is not completely wrong, but almost all wrong. Many political scientists seem to think that it is a sufficient test for nonlinearity.

That is called a "quadratic" equation. The plot illustrating that curve is either a "hill" or a "bowl," depending on whether the coefficient on the squared term is negative (hill) or positive (bowl).

It is completely easy to see the hill or bowl does nothing to help us with the curvy locfit data.



```
summary(mymod1)
```

```
Call:
lm(formula = y ~ x + I(x^2))

Residuals:
    Min      1Q  Median      3Q     Max
-5.5108 -2.6118  0.8003  2.4781  5.8356

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.84765    0.96066   4.005 0.000121 ***
x           -1.13643    0.44710  -2.542 0.012612 *
I(x^2)       0.09083    0.04239   2.142 0.034657 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.047 on 97 degrees of freedom
Multiple R^2: 0.07998, Adjusted R^2: 0.06101
F-statistic: 4.216 on 2 and 97 DF,  p-value: 0.01755
```

The good news is that the significance of the squared term in the regression would give you a warning that there is nonlinearity. Unfortunatly, to most people, it would just be evidence that "the quadratic model is right".

4 Polynomial

If you add in many more terms, say x^3 and x^4, then you have a polynomial. See my handout on approximations. There's a theorem that states that if you keep adding terms, you can approximate any function as closely as you like. To approximate this one, we really do need to add a lot of terms.

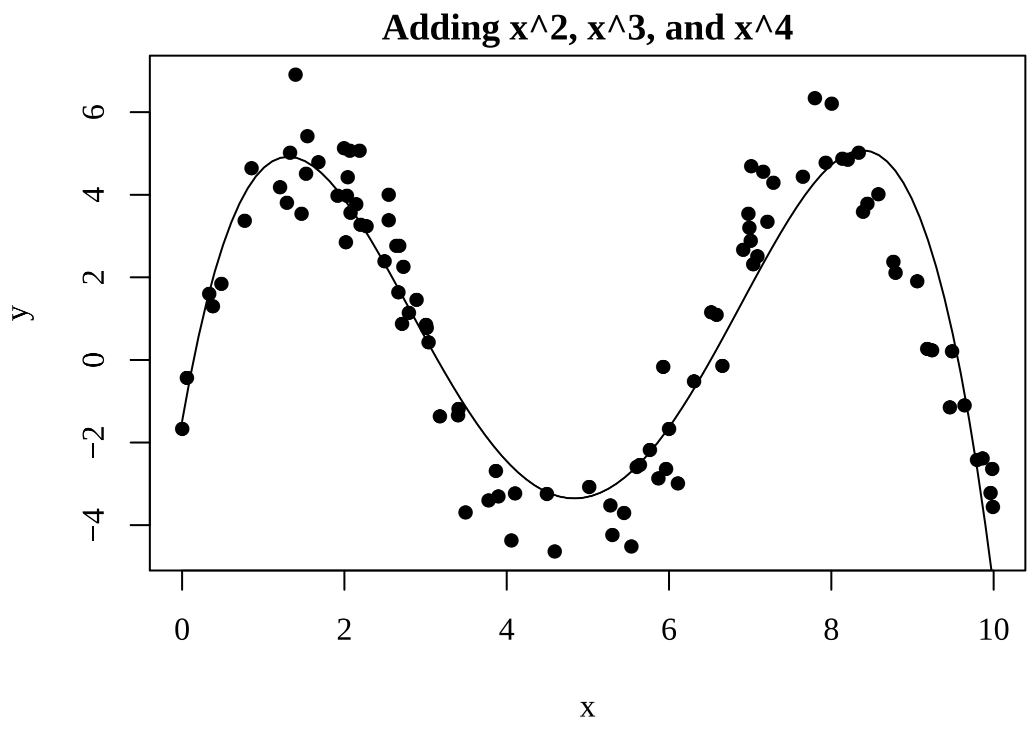
A polynomial has one fewer "hump" than it has terms. Since our graph has 3 "humps", the fitted model would need to include

$$\hat{y}_i = \hat{b}_0 + \hat{b}_1 x_i + \hat{b}_2 x_i^2 + \hat{b}_3 x_i^3 + \hat{b}_4 x_i^4$$

```
mymod1 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4))
summary(mymod1)
```

```
Call:
lm(formula = y ~ x + I(x^2) + I(x^3) + I(x^4))

Residuals:
     Min       1Q   Median       3Q      Max
-2.67548 -0.74047 -0.03925  0.77174  2.88683

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -1.485342   0.633429  -2.345   0.0211 *
x           11.319873   0.793773  14.261   <2e-16 ***
I(x^2)      -6.170693   0.324063 -19.042   <2e-16 ***
I(x^3)       1.033545   0.049191  21.011   <2e-16 ***
I(x^4)      -0.053369   0.002442 -21.855   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.221 on 95 degrees of freedom
Multiple R^2: 0.8554,  Adjusted R^2: 0.8493
F-statistic: 140.5 on 4 and 95 DF,  p-value: < 2.2e-16
```

```
newx <- seq(0, 10, length.out = 100)
predict2 <- predict(mymod1, newdata = data.frame(x = newx))
plot(x, y, main = "Adding x^2, x^3, and x^4", pch = 16)
lines(newx, predict2)
```



If the data has any more random noise in it, then fitting a polynomial like that will be pretty dicey. Work out an example for yourself.

Why isn't this good enough? Sometimes the data is not quite so regular as this example, so one must add many powers. Perhaps the degress of freedom are used up. Usually, the most serious problem is that there is severe multicollinearity between x_i, x_i^2, x_i^3, and so the parameter estimates are very unstable, often not statistically significantly different from 0. There is a way to transform those variables so that they are not correlated with each other at all. That method is known as "orthogonal polynomials." The predicted values from an orthogonal polynomial model are the same as in a model fit with the ordinary values of x_i^p. The standard errors are smaller, however, and one may get some hints about whether the higher order terms are really needed. It is quite unclear to me, however, how to decide because the transformed orthogonal polynomial coding does not translate into something that is understandable in terms of the original problem. I've been working on an example of that.

5 A straight line spline

If you divide the x-axis into sections, and estimate lines, then you can approximate the underlying relationship. If you start with the idea that the underlying truth is $y = b_0 + b_1 x$ up to some "break point" τ_1, and then after that the line changes slope, then a convenient way to do that is to construct a new variable that is equal to 0 if $x_i < \tau_1$ and $x_i - \tau_1$ if $x_i \geq \tau_1$. Represent that by this symbol:

$$(x_i - \tau_1)_+$$

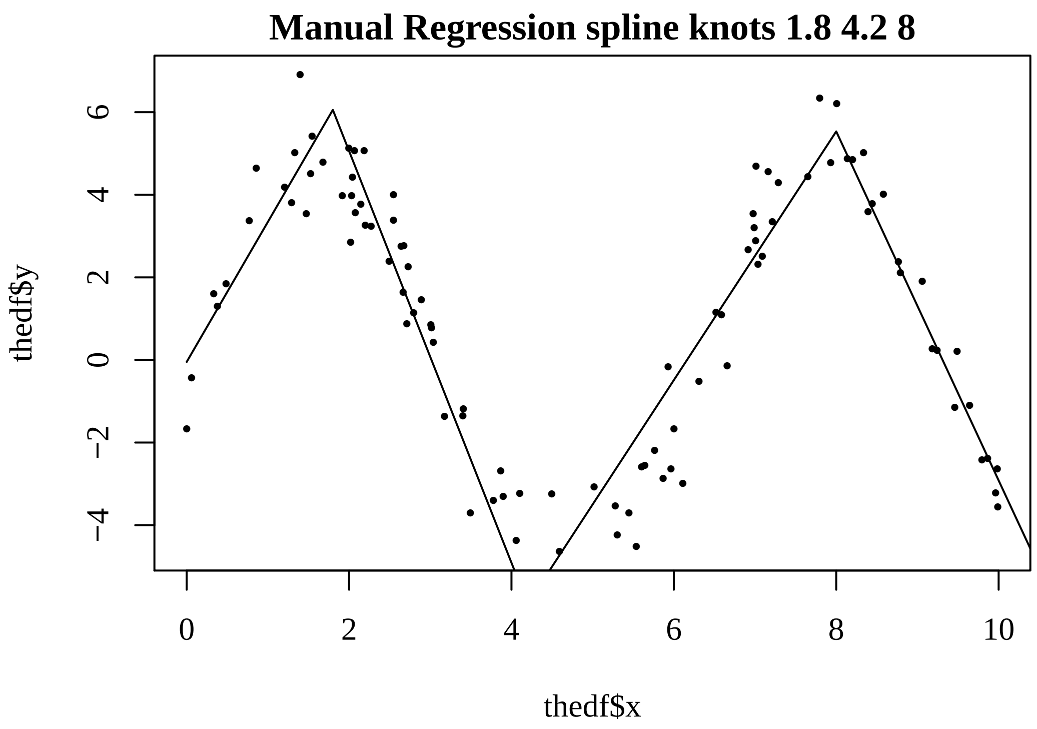
The underlying truth is then represented by

$$y = b_0 + b_1 x_i + b_2 (x_i - \tau_1)_+$$

If there is no change at τ_1, then $b_2 = 0$. The coefficient b_2 represents the additional (or lessened) slope to the right of τ_1. Of course, you can put in as many break points as you want. For the curvy data considered here, we need 3 breakpoints.

You can specify the break points manually and estimate the model with this R code. The fiddling about with newdf is necessary in order to make the plotted lines look nice.

```
knot <- c(1.8, 4.2, 8)
thedf <- data.frame(x = x, y = y)
thedf <- thedf[order(thedf$x), ]
createplusvar <- function(input, k) {
     it <- ifelse(input > k, input - k, 0)
 }
thedf$xp1 <- createplusvar(thedf$x, knot[1])
thedf$xp2 <- createplusvar(thedf$x, knot[2])
thedf$xp3 <- createplusvar(thedf$x, knot[3])
mymod0 <- lm(y ~ x + xp1 + xp2 + xp3, data = thedf)
newx <- seq(round(min(x), 1), round(max(x) + 0.5, 1), by = 0.1)
newdf <- data.frame(x = newx, xp1 = createplusvar(newx, knot[1]),
    xp2 = createplusvar(newx, knot[2]), xp3 = createplusvar(newx,
     knot[3]))
newdf$pred <- predict(mymod0, newdata = newdf)
mytitle <- paste("Manual Regression spline knots", knot[1], knot
   [2], knot[3])
plot(thedf$x, thedf$y, main = mytitle, type = "n")
points(thedf$x, thedf$y, pch = 16, cex = 0.5)
lines(newdf$x, newdf$pred)
rm(thedf, mymod0, newdf)
```



You can see in the figure that I did not guess the splines quite right. I wanted to have a model estimate the break points. I tested 2 packages, “segmented” and “mda”.

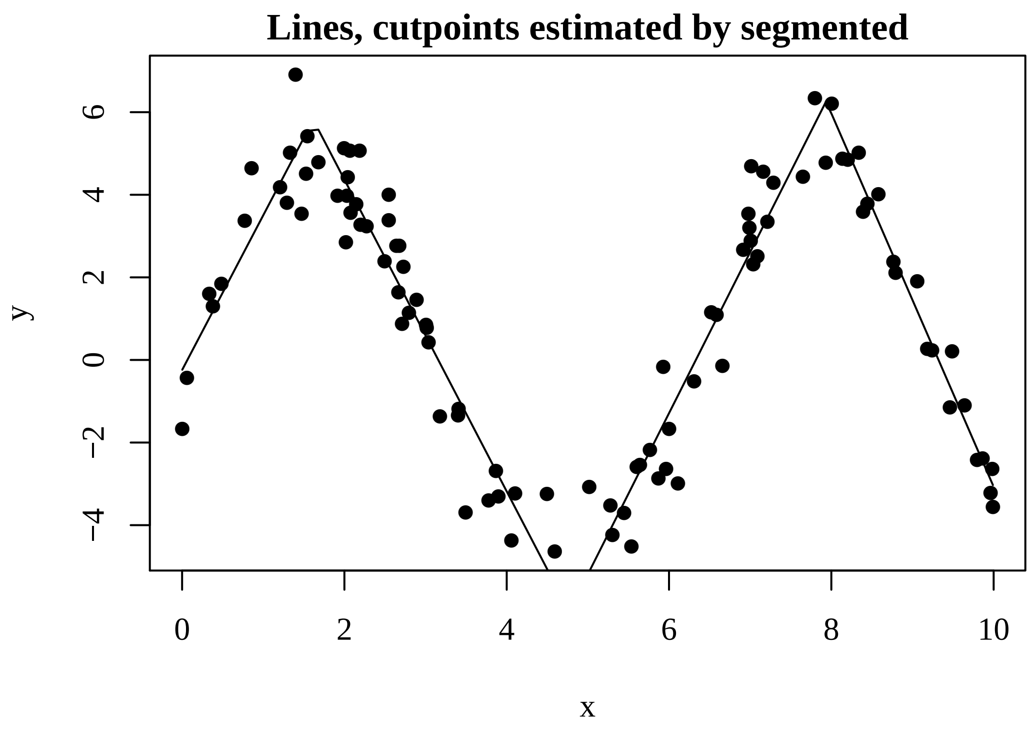
I’ve heard that developing algorithms to calculate the knots is quite a difficult business. This shows that the “segmented” package does work, but only in a fragile way. If you don’t specify the starting estimates for the break points (the option psi) correctly–with the correct number and values of the break points–then the estimator blows up. This particular example is not one that allows it to guess the correct number of break points and give you everything you want. I found it very tough to use the ordinary predict() function with this model, and so to make a plotted line, I had to take some steps that seemed unusual to me.

```
library(segmented)
mymod0 <- lm(y ~ x)
segmod <- segmented(mymod0, seg.Z = ~x, psi = list(x = c(1.8, 4.2
   , 8)), it.max = 200)
summary(segmod)
```

```
  ***Regression Model with Segmented Relationship(s)***

Call:
segmented.lm(obj = mymod0, seg.Z = ~x, psi = list(x = c(1.8,
    4.2, 8)), it.max = 200)

Estimated Break-Point(s):
         Est.   St.Err
psi1.x 1.617 0.06576
psi2.x 4.769 0.07349
```

```
psi3.x 7.936 0.06339

t value for the gap-variable(s) V:  -1.964132e-15 -2.308255e-15 8
   .321184e-15

Meaningful coefficients of the linear terms:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.2450     0.4746  -0.516    0.607
x             3.7483     0.4500   8.330 7.39e-13 ***
U1.x         -7.5239     0.4915 -15.310       NA
U2.x          7.6758     0.2875  26.695       NA
U3.x         -8.4214     0.3705 -22.727       NA
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.9125 on 92 degrees of freedom
Multiple R-Squared: 0.9218,  Adjusted R^2: 0.9158

Convergence attained in 4 iterations with relative change 3
   .710174e-16
```

```
mynewdf <- data.frame(x = x, fitted = segmod$fitted.values)
mynewdf <- mynewdf[order(mynewdf$x), ]
plot(x, y, main = "Lines, cutpoints estimated by segmented", pch
   = 16)
lines(mynewdf$x, mynewdf$fitted)
detach("package:segmented")
```



Except for the difficulty of fitting this, the result is pretty encouraging, at least to the eye. The predicted line seems to "go along" with the data pretty well. Oh, I should mention, we also assumed away the possibility of gaps, or discontinuities. segmented() provides a test for the significance of the assumption that the line segments connect at the break points.

In Venables & Ripley, 4ed p. 235, the MARS (Multiple Adaptive Regression Splines) method is mentioned, which can be found in the "mda" package. This is a linear spline model in which the knots are estimated from the data and the estimation is much more robust:

```
library(mda)
mymars ← mars(x, y, degree = 1)
summary(mymars)
```

```
               Length  Class   Mode
call             4     -none-  call
all.terms        7     -none-  numeric
selected.terms   7     -none-  numeric
penalty          1     -none-  numeric
degree           1     -none-  numeric
nk               1     -none-  numeric
thresh           1     -none-  numeric
gcv              1     -none-  numeric
factor          11     -none-  numeric
cuts            11     -none-  numeric
residuals      100     -none-  numeric
fitted.values  100     -none-  numeric
```

```
lenb              1     -none-  numeric
coefficients      7     -none-  numeric
x               700     -none-  numeric
```

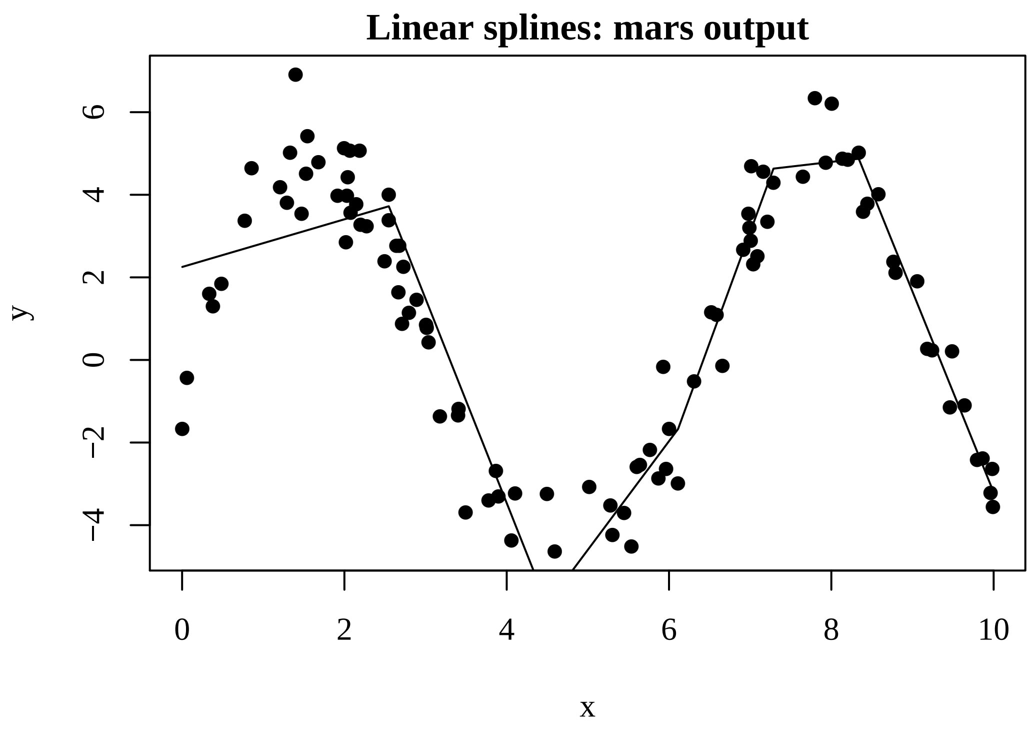
```
mymars$cuts
```

```
          [,1]
 [1,] 0.000000
 [2,] 4.496526
 [3,] 4.496526
 [4,] 8.338125
 [5,] 8.338125
 [6,] 2.547249
 [7,] 2.547249
 [8,] 7.286836
 [9,] 7.286836
[10,] 6.111152
[11,] 6.111152
```

```
mymars$gcv
```

```
[1] 1.777441
```

```
mydf <- data.frame(x, y, fitted = mymars$fitted)
mydf2 <- mydf[order(mydf$x), ]
plot(x, y, main = "Linear splines: mars output", pch = 16)
lines(mydf2$x, mydf2$fitted)
rm(mymars)
```



It could be worse. I tried this with a restricted number of cutpoints.

```
library(mda)
mymars2 ← mars(x, y, degree = 1, nk = 5)
summary(mymars2)
```

```
                     Length  Class   Mode
call                   5     −none−  call
all.terms              4     −none−  numeric
selected.terms         4     −none−  numeric
penalty                1     −none−  numeric
degree                 1     −none−  numeric
nk                     1     −none−  numeric
thresh                 1     −none−  numeric
gcv                    1     −none−  numeric
factor                 5     −none−  numeric
cuts                   5     −none−  numeric
residuals            100     −none−  numeric
fitted.values        100     −none−  numeric
lenb                   1     −none−  numeric
coefficients           4     −none−  numeric
x                    400     −none−  numeric
```

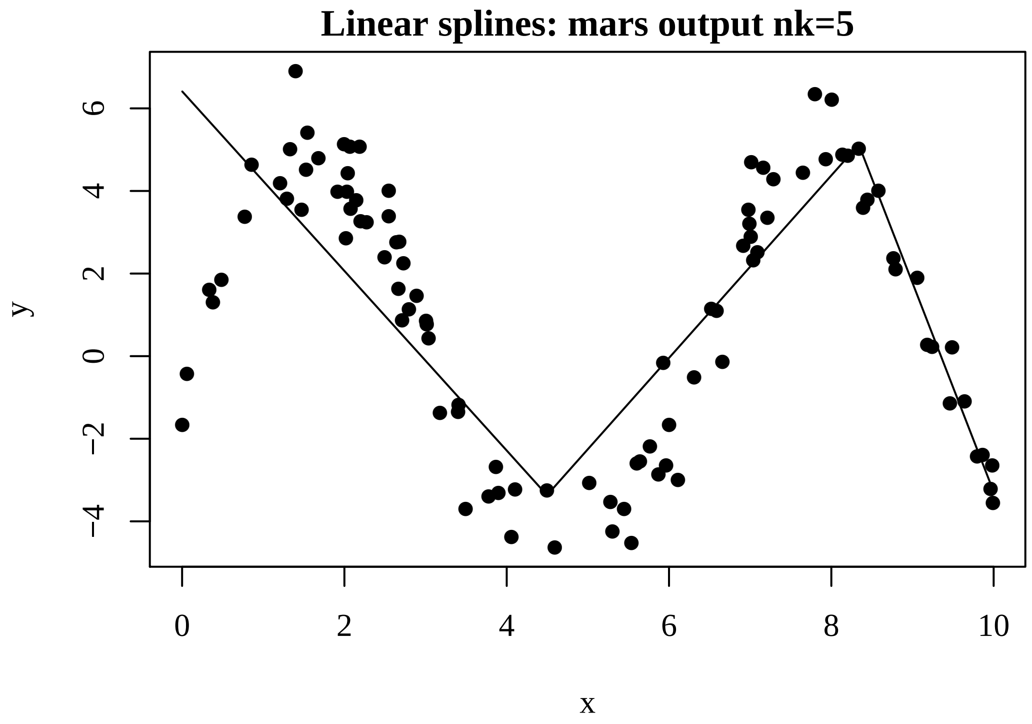
```
mymars2$gcv
```

```
[1] 4.60265
```

```
mymars2$cuts
```

```
         [,1]
[1,] 0.000000
[2,] 4.496526
[3,] 4.496526
[4,] 8.338125
[5,] 8.338125
```

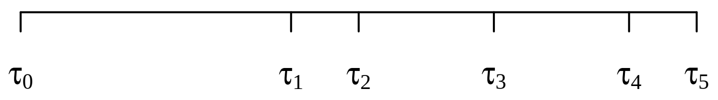
```
mydf ← data.frame(x, y, fitted = mymars2$fitted)
mydf2 ← mydf[order(mydf$x), ]
plot(x, y, main = "Linear splines: mars output nk=5", pch = 16)
lines(mydf2$x, mydf2$fitted)
rm(mymars2, mydf, mydf2)
detach("package:mda")
```



A big philosophical objection against this model is the "kinked" nature of this result. There are sharp turns that we know, for a fact, are "wrong" (in the sense that the true relationship does not have kinks).

6 A Smoother Spline

There are more splines models than you can shake a stick at. There seem to be statisticians who spend their whole lives debating whether we should use "tensor product splines" or "b-splines" or whatever else. Let's focus on the simpler idea of a cubic spline. Consider an input variable x_i and it is defined on an interval with end points (τ_0, τ_{k+1}) that is subdivided into k segments. Here's an example of how the x axis might be segmented with 4 knots in the interior of the line.

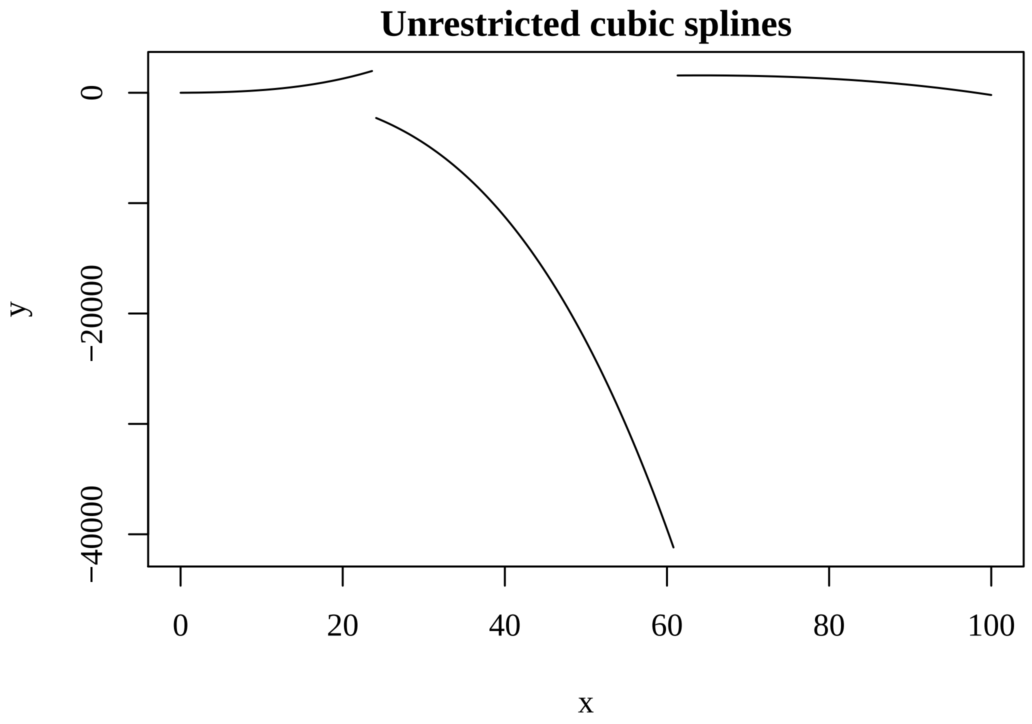


On each interval, we can imagine a cubic model,

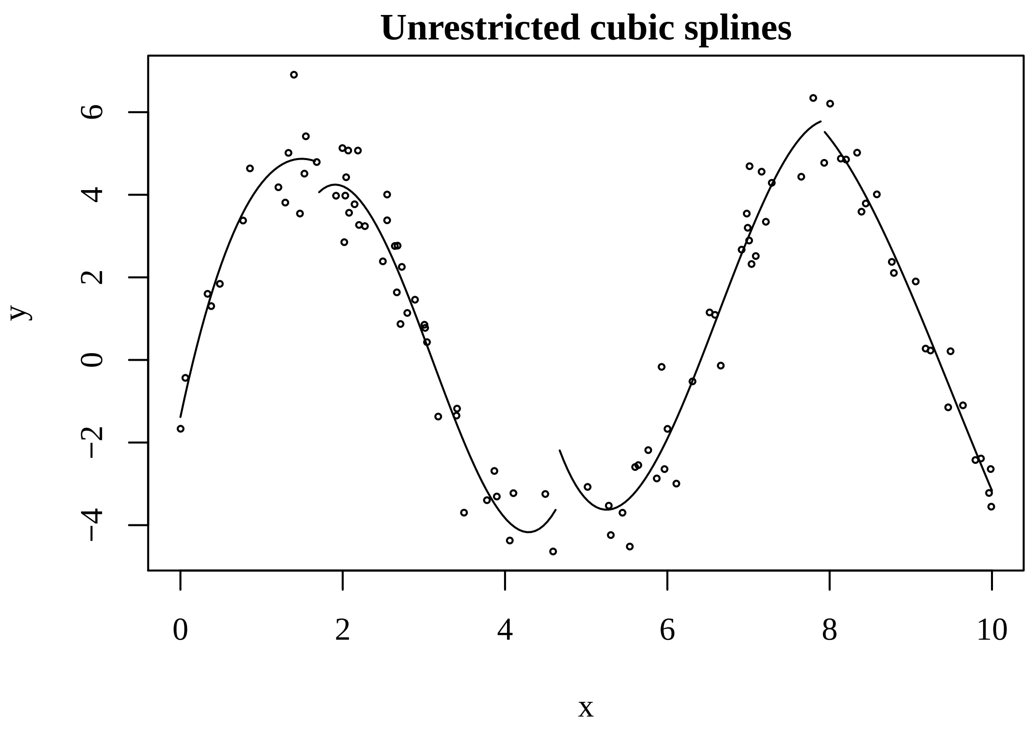
$$\hat{y}_i = \hat{b}_0 + \hat{b}_1 x_i + \hat{b}_2 x_i^2 + \hat{b}_3 x^3$$

A cubic model will be "curvy enough" to fit most relationships. If it does not fit, add more break points (called "knots").

How to "link together" all of these separate cubic models? This is where the adventure begins. If you were completely barefoot, you might just allow them to be completely separate from each other. How much uglier could it get than this? Here's just a phony example I created to get the idea across:



This shows what happens if you fit 4 separate cubic polynomials to the sine wave data that we have been investigating.



(Connected) Cubic Splines: First Fix

If you think the unconnected, discontinuous splines are no good, what do you do to fix it? What do you want?

Each segment's spline must blend smoothly into the next. So assume:

1. No gaps at the knots.

2. Smooth Connections between parts. No pointy kinks at the knots.

Those 2 assumptions have VERY MUCH power on simplifying the problem. In particular, they mean that the slope and second derivatives at the knot points have to match exactly.

Recall that the "pluss function" notation $(x_i - \tau)_+$ refers to a variable that is equal to 0 if $x_i < \tau$ and it is equal to x_i if $x_i > \tau$. Similarly, one can define $(x_i - \tau)_+^2$ or $(x_i - \tau)_+^3$ to refer to variables that are 0 up to τ and are then equal to x_i^2 or x_i^3 after.

I believe these are called "truncated power basis" splines. Note how there is obviously a lot of multicollinearity in data columns that are created in this way. (Obvious, right?)

There is an alternative encoding of the spline variables called the b-spline the multicollinearity is reduced. Harrell's Regression Modeling Strategies states that the difference is not usually substantial (and I trust his judgment).

A first try at a formula for a cubic spline might have us trying to write down a cubic equation for each knot, something silly like

$$\begin{aligned}\hat{y}_i &= \hat{b}_0 + \hat{b}_1 x_i + \hat{b}_2 x^2 + \hat{b}_3 x_i^3 + \\ &\quad \hat{b}_4 + \hat{b}_5(x_i - \tau_1)_+ + \hat{b}_6(x_i - \tau_1)_+^2 + \hat{b}_7(x_i - \tau_1)_+^3 \ (after\, first\, knot) \\ &\quad \hat{b}_8 + \hat{b}_9(x_i - \tau_2)_+ + \hat{b}_{10}(x_i - \tau_2)_+^2 + \hat{b}_{11}(x_i - \tau_2)_+^3 \ (after\, second\, knot)\end{aligned}$$

But you should see right off that we can't really use all of this detail. Because of the restriction that the cubic equations must meet at the knots (no gaps!), it must be that $b_4 = b_8 = 0$. So kick those parameters out. And because the slopes of the cubics must be the same at the knots, we are not allowed to have b_5 and b_6 be nonzero. And there is the condition that the second derivatives must match as well, so $b_6$and b_{10} have to be eliminated.

After throwing away the gaps and changes in slope and curvature at the knots, the cubic spline formula is just

$$\begin{aligned}\hat{y}_i &= \hat{b}_0 + \hat{b}_1 x_i + \hat{b}_2 x^2 + \hat{b}_3 x_i^3 + \\ &+\hat{b}_4 (x_i - \tau_1)_+^3 \; (after\, first\, knot) \qquad (1)\\ &+\hat{b}_5 (x_i - \tau_2)_+^3 \; (after\, second\, knot)\\ &and\, so\, forth \qquad (2)\end{aligned}$$

We suppose that the "basic part" of the relationship–which applies all across the range of x_i– is $\hat{b}_0+\hat{b}_1 x_i+\hat{b}_2 x^2+\hat{b}_3 x_i^3$ and then as $x_i$moves from left to right, the model "picks up" additional terms. After the first interior knot, we have added a quantity $\hat{b}_4(x_i - \tau_1)_+^3$. If $\hat{b}_4$ equals 0, it means the relationship on the second segment is the same as the first. After the second knot, we have BOTH the additional curvature from the first knot and the second knot, so we add on: $\hat{b}_4(x_i - \tau_1)_+^3 + \hat{b}_5(x_i - \tau_2)_+^3$. That means there are 4 parameters estimated for the basic cubic equation and then there is one additional parameter for each of the segments after the first.

You still have "knots" when you use smooth splines. But you don't have kinks.

These can be estimated manually, in a method that exactly matches the linear spline discussed above. There does not seem to be a pre-packaged routine to estimate this primitive version of the model. It is considered a bad idea."The truncated power basis is attractive because the plus-function terms are intuitive and maybe entered as covariates in standard regression packages. However, the number of plus-functions requiring evaluation increase with the number of breakpoints, and these terms often become collinear, just as terms in a standard polynomial regression do." (Lynn A Sleeper and David P. Harrington, "Regression Splines in a Cox Model with Application to Covariate Effects in Liver Disease", Journal of the American Statistical Association, 85(412) December 1990, p.943 (941-949)).

I want to emphasize very much that a change in the coefficient for one segment may have a "ripple effect" across all of the others. If one segment's coefficient gets tuned "way high" then the others have to adjust to compensate. So it is not really as if the cubic model with x_i, x_i^2, and x_i^3 is estimated on the first piece, then the rest are "layered on". Rather, they are all estimated at once, so all of those $\hat{b}'s$ influence each other.

Restricted ("natural") cubic splines

Note that the coherence and structure of the splined relationship flows from the fact that the curvature within any segment depends on its relationship to its neighboring segment. For the segments on the left end and the right end, there is no stabilizing influence of a neighbor.

A restricted cubic spline is one that insists that, on those outer segments, the prediction is based on a linear model, rather than a cubic one. In the unrestricted cubic spline model, there are $k + 4$ parameters to estimate (including the intercept). Harrell observes that the number of parameters (including the intercept) is reduced to k if we adopt a restricted cubic spline.

It is a bit difficult to understand how we lose 4 parameters in the transition from unrestricted to restricted spline. The only no-brainer is this:

Linear Segment 1: $\hat{b}_2 = \hat{b}_3 = 0$

I *think* the two other boundary conditions are

Spline Coefficients sum to 0: $\sum_{j=1}^{k} \hat{b}_{j+3} = 0$

??How To Name this one??: $\sum_{j=1}^{k} \tau_j \hat{b}_{k+3} = 0$

Harrell says Stone and Koo found a way to recode the input variable. Suppose you could somehow magically create these new variables z_{1i}, z_{2i}, and z_{3i} so that you could estimate a model with 5 knots as

$$\hat{y}_i = \hat{b}_o + \hat{b}_1 x_i + \hat{c}_1 z_{1i} + \hat{c}_2 z_{2i} + \hat{c}_3 z_{3i} \tag{3}$$

and after estimating that, translate the estimates of these coefficients back into the form given by a more substantively appealing expression like

$$\hat{y}_i = \hat{b}_0 + \hat{b}_1 x_i + \hat{b}_2 (x_i - \tau_1)^3 + \hat{b}_3 (x_i - \tau_2)^3_+ + \ldots$$

What is the magical encoding? See Harrell p. 20.

$$z_{ji} = (x_i - \tau_j)^3_+ + (x_i - \tau_{k-1})^3_+ \frac{\tau_k - \tau_j}{\tau_k - \tau_{k-1}} + (x_i - \tau_k)^3_+ \frac{\tau_{k-1} - \tau_j}{\tau_k - \tau_{k-1}}$$

What the hell is that? Will you settle for examples?

I load the Design package with the library () command first:

```
library(Design)
```

Design has a function "rcs" that can create the new component variables for us. That loads the splines library as well because Design requires it.

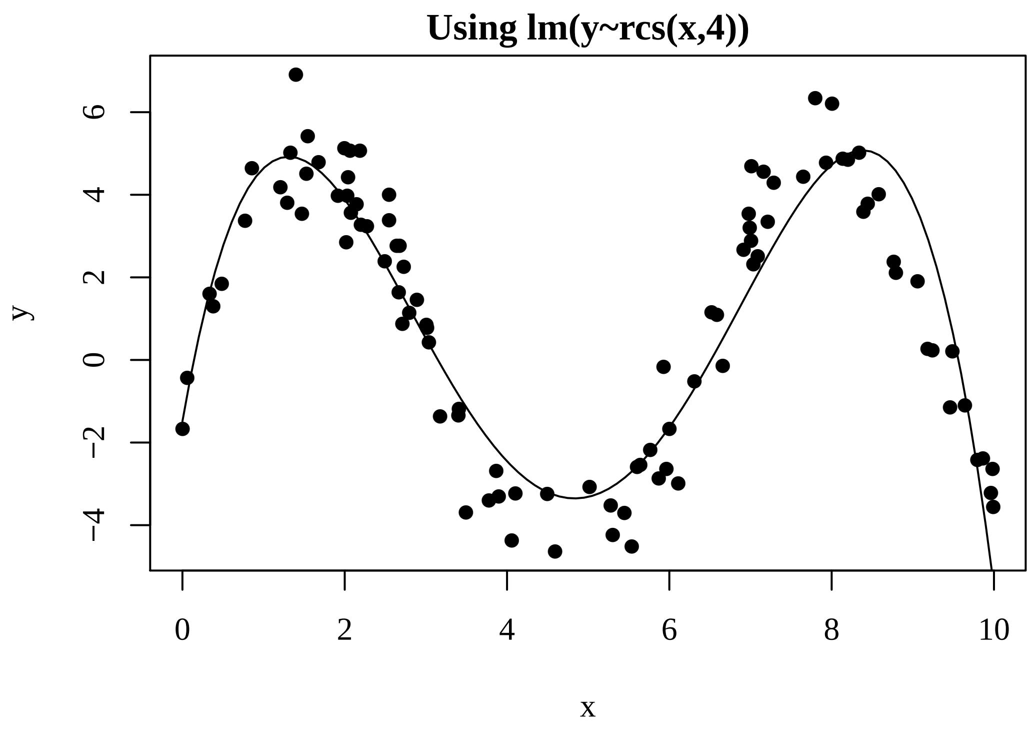
```
asequence ← 1:20
splineMatrix ← rcs(asequence, 4)
splineMatrix
```

```
      asequence   asequence'  asequence''
[1,]          1  0.000000000 0.0000000000
[2,]          2  0.000000000 0.0000000000
[3,]          3  0.000000000 0.0000000000
[4,]          4  0.000000000 0.0000000000
[5,]          5  0.000000000 0.0000000000
```

```
 [6,]          6  0.008264463 0.0000000000
 [7,]          7  0.066115702 0.0000000000
 [8,]          8  0.223140496 0.0003543388
 [9,]          9  0.528925620 0.0203336777
[10,]         10  1.033057851 0.1072551653
[11,]         11  1.785123967 0.3107055785
[12,]         12  2.834710744 0.6802716942
[13,]         13  4.231404959 1.2655402893
[14,]         14  6.015372291 2.1089466708
[15,]         15  8.110359036 3.1645532512
[16,]         16 10.361590909 4.3268181818
[17,]         17 12.638863636 5.5068595041
[18,]         18 14.916136364 6.6869008264
[19,]         19 17.193409091 7.8669421488
[20,]         20 19.470681818 9.0469834711
attr(,"class")
[1] "Design"
attr(,"name")
[1] "asequence"
attr(,"label")
[1] "asequence"
attr(,"assume")
[1] "rcspline"
attr(,"assume.code")
[1] 4
attr(,"parms")
[1]  5.00  7.65 13.35 16.00
attr(,"nonlinear")
[1] FALSE  TRUE  TRUE
attr(,"colnames")
[1] "asequence"   "asequence'"  "asequence''"
```

We have to specify either the number of knots or a vector of positions for knots. If we don't specify the exact positions of the knots, rcs will position them so that the data is divided into equally sized subgroups (quintiles, etc.). We can use $rcs(x)$ in as an input variable in a regression model:

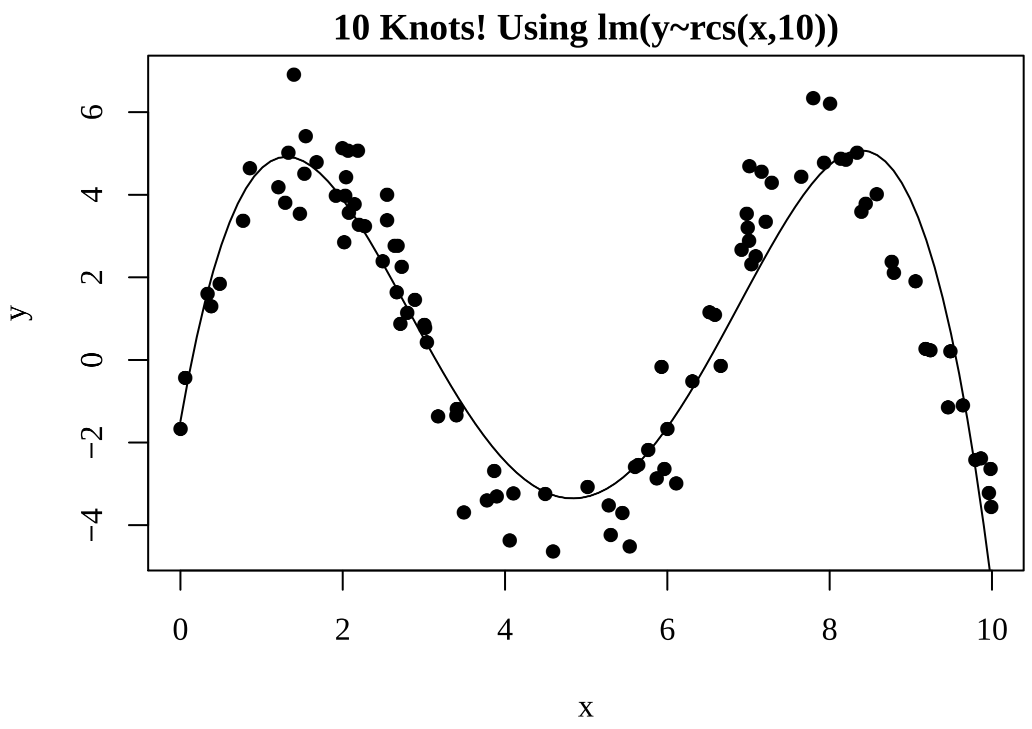
```
mymod4 <- lm(y ~ rcs(x, 4))
newx <- seq(0, 10, length.out = 100)
predict4 <- predict(mymod1, newdata = data.frame(x = newx))
plot(x, y, main = "Using lm(y~rcs(x,4))", type = "n")
points(x, y, pch = 16)
lines(newx, predict4)
```



The summary information on the fitted model, well, only its Mother could love it:

```
Call:
lm(formula = y ~ rcs(x, 4))

Residuals:
    Min      1Q  Median      3Q     Max 
-7.3274 -2.9913  0.7751  2.3304  5.2871 

Coefficients:
               Estimate Std. Error t value Pr(>|t|)    
(Intercept)      5.6662     1.1462   4.944 3.25e-06 ***
rcs(x, 4)x      -2.0683     0.5735  -3.607 0.000494 ***
rcs(x, 4)x'      6.4465     2.3709   2.719 0.007772 ** 
rcs(x, 4)x''   -11.1982     4.6379  -2.415 0.017651 *  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.949 on 96 degrees of freedom
Multiple R^2: 0.1474,   Adjusted R^2: 0.1208
F-statistic: 5.533 on 3 and 96 DF,  p-value: 0.001513
```

What if you make the number of knots much higher, say 10?



And if you thought the other output was ugly, dig this:

```
summary(mymod4)
```

```
Call:
lm(formula = y ~ rcs(x, 10))

Residuals:
     Min       1Q   Median       3Q      Max
-1.85094 -0.52619  0.07638  0.53238  1.94431

Coefficients:
                      Estimate Std. Error t value Pr(>|t|)
(Intercept)            -0.6753     0.4438  -1.522 0.131605
rcs(x, 10)x             4.9678     0.5578   8.905 5.42e-14 ***
rcs(x, 10)x'         -105.6228    19.3026  -5.472 3.97e-07 ***
rcs(x, 10)x''         257.8897   115.8651   2.226 0.028527 *
rcs(x, 10)x'''       -176.5435   212.2693  -0.832 0.407782
rcs(x, 10)x''''       135.7993   158.3145   0.858 0.393292
rcs(x, 10)x'''''     -150.0714    62.0285  -2.419 0.017558 *
rcs(x, 10)x''''''     108.2622    69.4949   1.558 0.122780
rcs(x, 10)x'''''''   -430.5118   138.3122  -3.113 0.002486 **
rcs(x, 10)x''''''''   452.4003   118.7002   3.811 0.000253 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 0.7889 on 90 degrees of freedom
Multiple R^2: 0.9428,  Adjusted R^2: 0.9371
F-statistic: 164.8 on 9 and 90 DF,  p-value: < 2.2e-16
```

```
rm(mymod4, predict4, newx)
```

Choosing the number of knots

The conventional wisdom, following studies by Stone, is that the positioning of the knots is not very influential. If the knots section off the data into evenly sized groups, then the fit is "pretty good." But the number of knots is important. The conventional wisdom also says that the number of knots should usually be between 4 and 7, and my testing confirms that adding more than 7 knots usually does not affect the predictions very much.

There is a danger of "over-fitting," of creating a too-complicated model that would not fit against a sample from the same data-generating process. So one would like to get rid of as many knots as possible while preserving the fit of the predictions.

The Cross Validation (CV) and Generalized Cross Validation (GCV) are frequently discussed criteria. As far as I understand it, these were developed for the "smoothing splines" models (see below; Craven, P. and Wahba, G. (1979). Smoothing noisy data with spline functions. Numer. Math., 31: 377-403) but they are applicable to any of these fitting procedures. As indicated in the MARS example, the gcv is calculated with each fitted model.

The AIC from fitted models can also be used to decide if the number of splines.

Cross Validation

CV stands for Cross Validation. It is a "leave-one-out" analysis of the fit. Remove the $i'th$ observation and re-calculate the predictive curve. Figure out what the prediction is for the input values of that $i'th$ observation. Call that leave-one-out prediction $\check{y}_i$. Cycle through all of the observations and average the squared errors. The Cross Validation estimate of prediction error is

$$CV = \frac{1}{N}\sum_{i=1}^{N}(y_i - \check{y}_i)^2$$

The calculation of CV may be very time-consuming. The model must be re-estimated N times.

Generalized Cross Validation

Think of translating the CV into a generalized linear model framework.

In the R package called "mgcv", one finds a set of methods that will fit models (spline models) to optimize the GCV. In the mgcv documentation, the GCV statistic is defined as

$$GCV = \frac{nD}{(n-df)^2}$$

where deviance is an indicator of the badness of fit (recall the GLM?), n is the sample size and df is the effective degrees of freedom. The effective degrees of freedom reflects the number of degrees of freedom that are used to estimate the model–something like the number of cutpoints or the amount of curvature allowed in the spline.

An alternative criterion in "mgcv" is the Unbiased Risk Estimator

$$UBRE = \frac{D}{n} + \frac{2s \cdot df}{n} - s$$

where s is the "scale parameter" of the generalized linear model (variance of error term in Normal/linear model).

I used the "gam" function in mgcv to experiment with the impact of changing the number of knots in a cubic spline model. The following code illustrates the fitting of one gam model for which the k (knot) parameter is set to 4, and it then fits models for $k = 3, 4, ..., 10$ and chooses the best one by finding the lowest GCV. The mgcv procedure is supposed to automate this choice for us, but for me it just churns up CPU time and never responds. I'm bookmarked that one.

```
library(mgcv)
gam(y ~ s(x, k = 4, bs = "cr"))
```

```
Family: gaussian
Link function: identity

Formula:
y ~ s(x, k = 4, bs = "cr")

Estimated degrees of freedom:
2.6388  total = 3.638848

GCV score: 9.177401
```

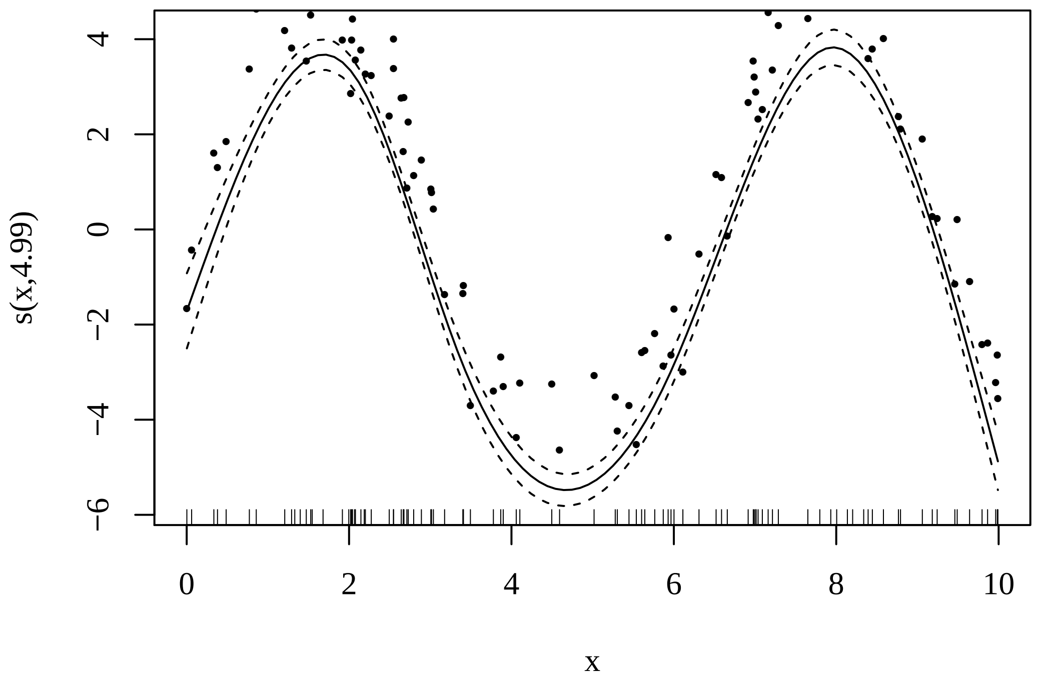
```
mygams <- list()
for (i in 3:10) {
     mygams[[i - 2]] <- gam(y ~ s(x, k = i, bs = "cr"))
 }
gcvresults <- vector()
for (i in 1:7) {
     gcvresults[i] <- mygams[[i]]$gcv.ubre
 }
gcvresults
```

```
[1] 9.1828206 9.1774012 0.9316883 0.7044464 0.7664113 0.7247366 0
   .7093403
```

```
bestone <- mygams[[which.min(gcvresults)]]
summary(bestone)
```

```
Family: gaussian
Link function: identity

Formula:
y ~ s(x, k = i, bs = "cr")

Parametric coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.22900    0.08138    15.1   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Approximate significance of smooth terms:
       edf Ref.df     F p-value
s(x) 4.988      5 276.6  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

R-sq.(adj) =  0.933   Deviance explained = 93.6%
GCV score = 0.70445  Scale est. = 0.66226   n = 100
```

```
plot(bestone)
points(x, y, pch = 16, cex = 0.5)
```



Smoother Matrix and effective degrees of freedom

Recall GCV, where the effective degrees of freedom is used. In the locfit documentation, it is claimed that

$$GCV = \frac{n\sum_{i=1}^{b}(y_i - \hat{y}_i)^2}{(n - tr(S))^2}$$

S represents the smoother matrix, which is the nonparametric equivalent of the "hat matrix" in OLS regression. The symbol "$tr(S)$" stands for the trace of S, which is the sum of the diagonal elements.

First, what is a "Smoother Matrix"?

Recall the "OLS hat matrix" is the matrix H against which you multiply y to convert observations into predictions ($\hat{y} = Hy$). The Smoother matrix is performing exactly that same role,

$$\hat{y} = Sy$$

Imagine an extreme case in which the Smoother Matrix is diagonal,

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

That would be an "interpolation" that gives back each point's actual value as a prediction. It is using N (sum of diagonal) degrees of freedom. On the other hand, if the smoother matrix has small values along the diagonal, then the observations are "gaining strength from each other" and the estimates of each one are not completely separate. Not as much separate information is being used to make each prediction.

The effective degrees of freedom is given by the trace of the Smoother matrix ($df = trace(S)$).

The CV value can be calculated using the Smoother matrix, where S_{ii} is the i't diagonal element.

$$CV = \frac{1}{N}\sum_{i=1}^{N}\left[\frac{y_i - \hat{y}_i}{1 - S_{ii}}\right]^2$$

This does not require the fitting of N separate leave-one-out models, but I'm not sure it helps that much because calculating the Smoother matrix itself might be expensive. Hutchison and de Hoog (1985) found an algorithm to calculate the diagonal elements of S efficiently.

The GCV as a generalization of CV is seen most readily in light of this latter definition of CV. replace the computationally difficult values S_{ii} with the average S_{ii} value, which is $\frac{trace(S)}{N}$. Inserting that into the CV formula gives all N terms a comon denominator, and this it can be re-written

$$CV = \frac{\frac{1}{N}\sum(y_i - \hat{y}_i)^2}{(1 - \frac{trace(S)}{N})}$$

The use of the trace of S as an estimate of the degrees of freedom has motivation in the OLS model. In the OLS model, the hat matrix is H_{ii}, and it is known that $trace(H_{ii}) = p$, the number of fitted parameter estimates. I have a separate handout that describes the hat matrix in depth. It is called "Regression Diagnostics." Thus, one might refer to the unbiased estimated variance of the error term in OLS as

$$\hat{\sigma}^2 = \frac{\sum(y_i - \hat{y}_i)^2}{N-p} = \frac{\sum(y_i - \hat{y}_i)^2}{N - trace(H)}.$$

Extension to multi-dimensional predictors

As the "fields" package (and web documentation) illustrates so well, several predictor variables can be used. The leading method is called "thin plate splines."

7 Local Smoothing Regressions: Lowess (Loess)

Spline approaches try to think of a segmented a relationship in a way that is "wholistic." It fits the relationship of the "whole line" at once while tailoring some pieces for different parts.

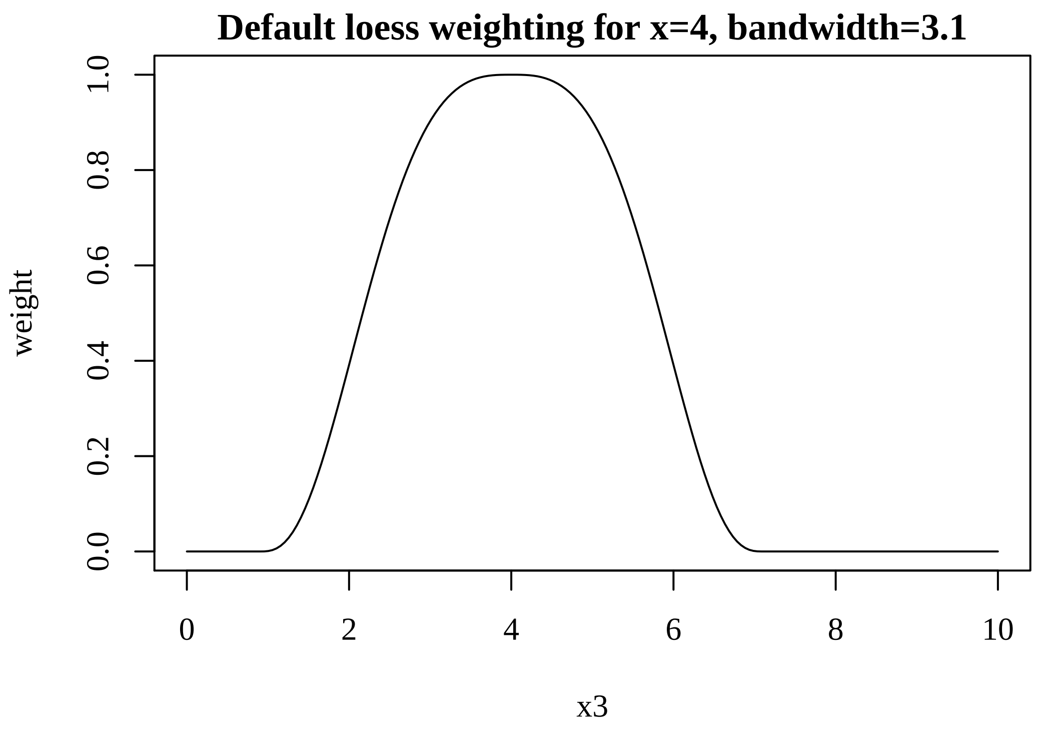
Take a radically different approach by simply calculating a predicted value, one for each x_i. Base the prediction only on observations that are "close to" each particular point. The original proposal was called "lowess" and a revised approach was called (loess).

Consider a particular point, x' (pronounced "x prime"). What is close to x_i? That's where the bandwidth, h, comes into play. Observations that are further than h units away don't matter at all, while the ones inside that bandwith have more weight when they are closer to x'. In the locfit package, as in other loess implementations I'm aware of, the default weight placed on each neighboring observation depends on $\frac{|x_i - x'|}{h}$.

$$W(x_i) = Weight\,on\,(x_i) = \begin{array}{cc} (1 - |\frac{x_i - x'}{h}|^3)^3 & if\,|\frac{x_i - x'}{h}| < 1 \\ 0 & otherwise \end{array}$$

What does that look like, I wonder to myself. Lucky there's R to show it:

```
x3 <- seq(0, 10, length = 200)
xprime <- 4
bandwidth <- 3.1
weight <- (1 - (abs((x3 - xprime)/bandwidth))^3)^3
weight <- ifelse(abs(xprime - x3) > bandwidth, 0, weight)
plot(x3, weight, type = "l", main = "Default loess weighting for
   x=4, bandwidth=3.1")
```



Any kind of formula can be used to calculate predicted values. We might, for simplicity, just use a linear model, $\hat{y}_i = \hat{b}_0 + \hat{b}_1 x_i$, or a quadratic model, $\hat{y}_i = \hat{b}_0 + \hat{b}_1 x_i + \hat{b}_2 x_i^2$. In "ordinary least squares," there are no weights:

$$\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

But the weighted local regression uses the $W(x_i)$ to de-emphasize the far-away values.

$$\sum_{i=1}^{n} W(x_i)[y_i - \hat{y}]^2 = \sum_{i=1}^{n} W(x_i)[y_i - (\hat{b}_0 + \hat{b}_1 x_i + \hat{b}_2 x_i^2]^2$$

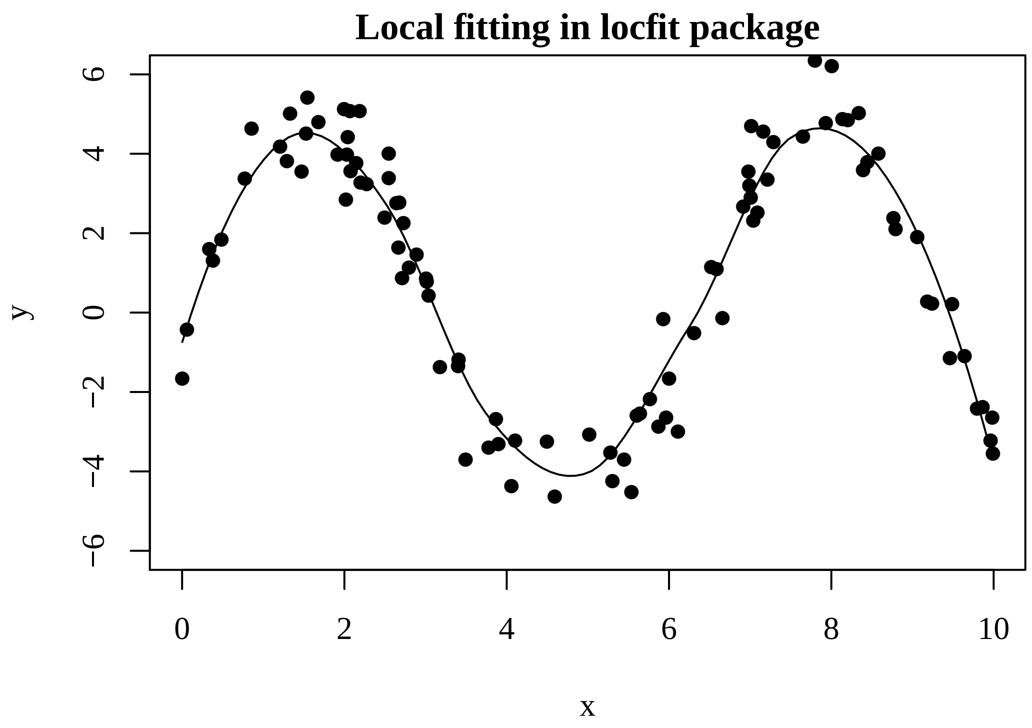
In the MASS package's loess method, and in locfit as well, the bandwidth is not described as a constant, but rather it is an interval wide enough to include a given fraction of the observations in the data. So, instead of thinking of h as a width that is the same for any given point, it may either grow wider or narrower depending on the clustering of observations.

This code will use locfit to calculate the predictions, and it uses the default plot method that is provided with locfit:

```
library(locfit)
```

```
fit1 ← locfit(y ~ lp(x, nn = 0.5))
plot(fit1, main = "Local fitting in locfit package", ylim = c(−6,
    6))
points(x, y, pch = 16)
gcv(fit1)
```

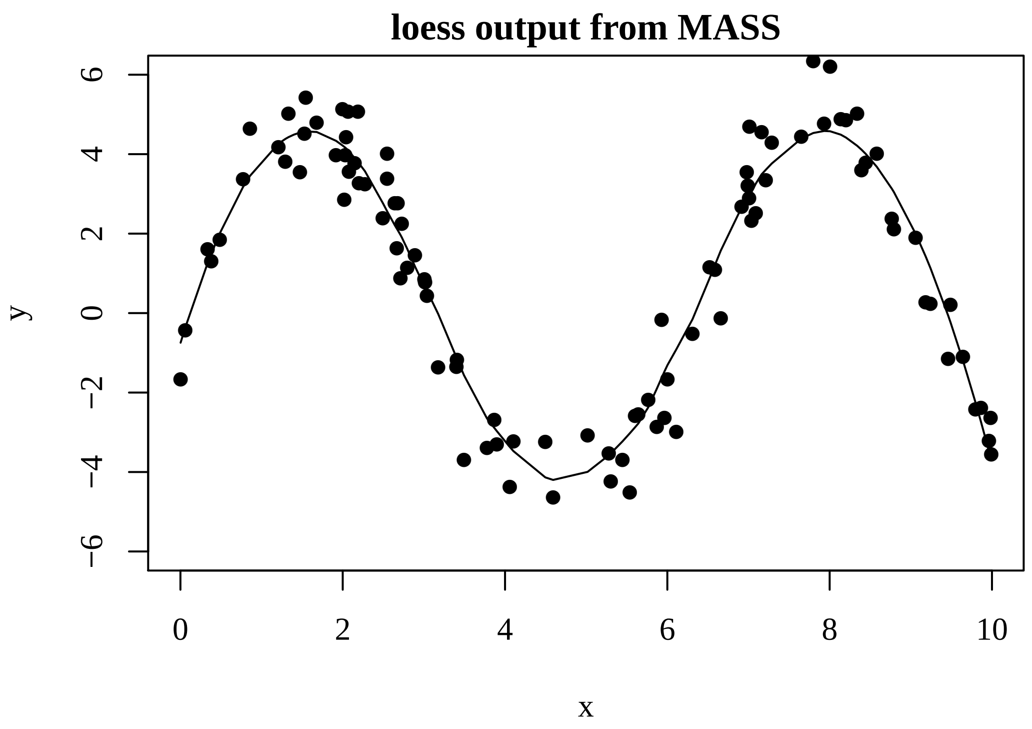
```
        lik        infl        vari         gcv
-33.5388536   6.9679449   6.3546839   0.7750201
```



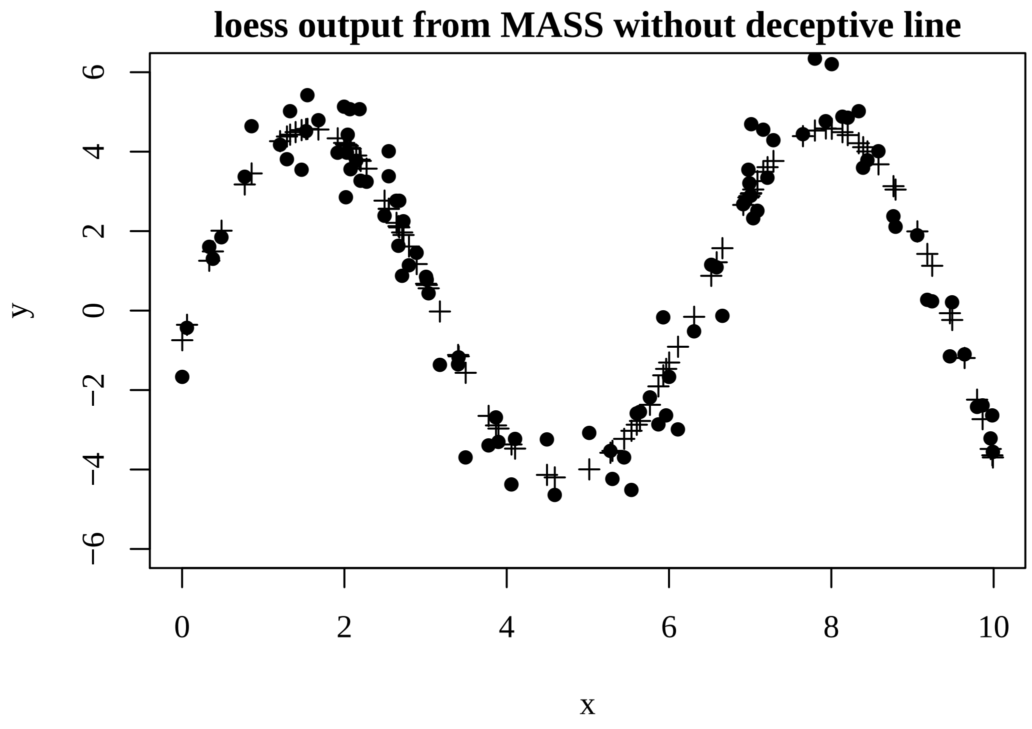
The smooth looking line here is really an optical illusion. The loess approach does not calculate predicted values on intervals. Rather, it calculates predictions for each of the points, and then the graphing tool connects those predicted points and makes them look smooth. I couldn't figure how to reveal that with locfit, but I could get it with MASS package.

The following code uses the MASS package's loess method. Note this is done with a more obvious sequence of calculating the predictions and then plotting them with the default plot method.

```
library(MASS)
fit2 <- loess(y ~ x, span = 0.5)
testx <- x[order(x)]
pred3 <- predict(fit2, data.frame(x = testx))
plot(x, y, pch = 16, main = "loess output from MASS", ylim = c(-6
   , 6))
lines(testx, pred3)
```



```
plot(x, y, type = "n", main = "loess output from MASS without
   deceptive line", ylim = c(−6, 6))
points(x, y, pch = 16)
points(x, fit2$fitted, pch = 3)
```



Extension to several input variables

The loess model, the second-generation one, included the ability to add more predictor variables. The transition from one to several variables is very simple in theory.

Measuring distance. We still weight neighboring points by distance. The only difference is that distance now is more difficult to measure. Recall from the pythagorean theorem ($a^2 + b^2 = c^2$) that the distance between two points $x_1 = (x_{11}, x_{12})$ and $x_2 = (x_{21}, x_{22})$ is $\sqrt{(x_{11} - x_{21})^2 + (x_{21} - x_{22})^2}$. That is a Euclidean method of measuring that works for any number of dimensions, so if there are, say 4 variables being considered, then the distance between case 1, $x_1 = (x_{11}, x_{12}, x_{13}, x_{14})$ and $x_2 = (x_{21}, x_{22}, x_{23}, x_{24})$ is

$$\sqrt{(x_{11} - x_{21})^2 + (x_{21} - x_{22})^2 + (x_{31} - x_{32})^2 + (x_{41} - x_{42})^2}$$

. Refer to that measurement ("norm") as $\|x_1 - x_2\|$. Using that distance measurement, then one simply calculates the weight by replacing $(x_i - x')$ with $\|x_1 - x_2\|$.

Recall that loess calculations are always done with a specific reference point x' for which we need to make a calculation. Suppose the values observed for that specific case are $x' = (x'_1, x'_2, x'_3, x'_4)$. Calculating a predicted value for $x_i$is, with quadratic model, calculated

$$\begin{aligned} \hat{y}_i &= \hat{b}_0 + \hat{b}_1(x_{i1} - x'_1) + \hat{b}_2(x_{i2} - x'_2) + \hat{b}_3(x_{i1} - x'_1)^2 \\ &\quad + \hat{b}_4(x_{i2} - x'_2)^2 + \hat{b}_5(x_{i1} - x'_1)(x_{i2} - x'_2) \end{aligned}$$

8 Smoothing Splines: marriage of Splines and Loess

Loess estimates an equation for every point. Splines seem not to, but rather project their curves across segments of x_i. Smoothing Splines is doing both.

Your goal is still to choose a "best" prediction for each point, a set of "best estimates" for each observation, $\hat{y}_i = f(x_i)$. In the Smoothing Spline approach, we build a natural spline model in which every unique point in x_i is a knot, then we adjust our predictive model to minimize the penalized residual error plus a penalty for "curvy" predictions. Note, since we use natural splines, it is very meaningful to talk about the first and second derivatives at a given point. Here is the objective function

$$SS(f, \lambda) = \sum_{i=1}^{N} (y_i - f(x_i))^2 + \lambda \int_{x_{min}}^{x_{max}} (f''(x))^2 dx$$

The first term is a penalty for a bad fit.

The second term is a "roughness penalty" that you pay for having a model with lots of bumps and bends. (Recall, if f'' is 0, then the relationship is linear).

The smoothing parameter is λ. The second derivative is used because $f(x_i)$ is cubic.

The smoothing spline approach is often justified with mention of a theorem, which states that: The unique minimizer of SS is a natural cubic spline with knots at the unique values of x_i.

How much smoothing?

Recall that, in the restricted cubic spline model, we wondered how many knots to select?

Here we allow ourselves N knots, but then we penalize curvature.

If you set $\lambda = \infty$, you are applying a harsh penalty to curvature, and the result will be that the best model is one in which $f'' = 0$ everywhere. In other words, an Ordinary Least Squares straight line is the best.

If you set $\lambda = 0$, then you are allowing f'' to be huge, in which case the result will be a "connect the dots" interpolation (assuming there's just one observation at each x_i).

Try out the "smooth.spline" function from the MASS bundle's "modreg" library. The default has all.knots=T, but I wrote it in just so I remember

```
library(MASS)
```

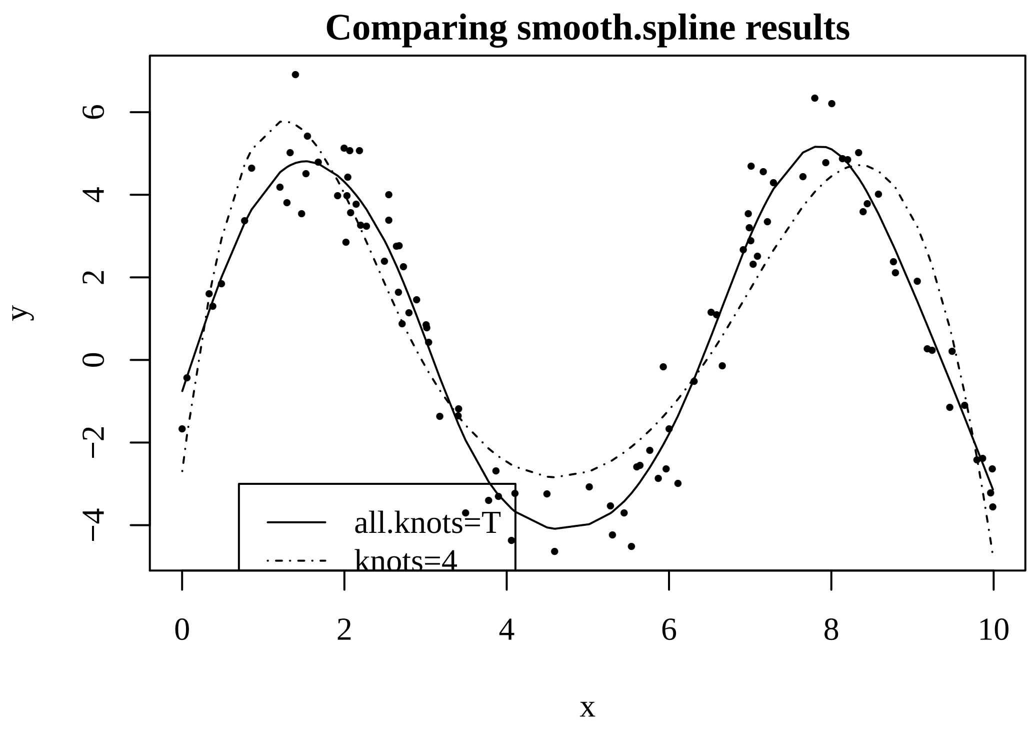
```
ss1 ← smooth.spline(x, y, all.knots = T)
print(ss1)
```

```
Call:
smooth.spline(x = x, y = y, all.knots = T)

Smoothing Parameter  spar= 0.918557  lambda= 0.0001279235 (14
   iterations)
Equivalent Degrees of Freedom (Df): 11.33271
Penalized Criterion: 53.2251
GCV: 27.89309
```

```
plot(x, y, type = "n", main = "Comparing smooth.spline results")
lines(ss1$x, ss1$y, lty = 1)
ss2 ← smooth.spline(x, y, all.knots = F, nknots = 4)
print(ss2)
```

```
Call:
smooth.spline(x = x, y = y, all.knots = F, nknots = 4)

Smoothing Parameter  spar= −0.1997111  lambda= 1.169087e−05 (15
   iterations)
Equivalent Degrees of Freedom (Df): 5.981517
Penalized Criterion: 158.3074
GCV: 23.05232
```

```
lines(ss2$x, ss2$y, lty = 4)
legend(0.7, −3, c("all.knots=T", "knots=4"), lty = c(1, 4))
points(x, y, pch = 16, cex = 0.5)
```



9 AVAS

So far, we have thought of ways to bend the regression line through the data distribution. We have not given ourselves the luxury of transforming the variables. We have seen examples in the past in which logging variables might eliminate clumpiness or make observations appear more normal. However, the process of transforming variables in regression can be quite a bit more elaborate. Consider the article by Tibshirani (Rob Tibshirani (1987), "Estimating optimal transformations for regression". Journal of the American Statistical Association 83, 394.) He outlines a transformation process that makes the scatterplot suitable for a linear regression with homogeneous variance.

The acepack was first brought to my attention by Harrell's Regression Modeling Strategies, which mentions the fact that ordinal predictor variables may be transformed in this way so that their impact can be estimated by a single slope coefficient.

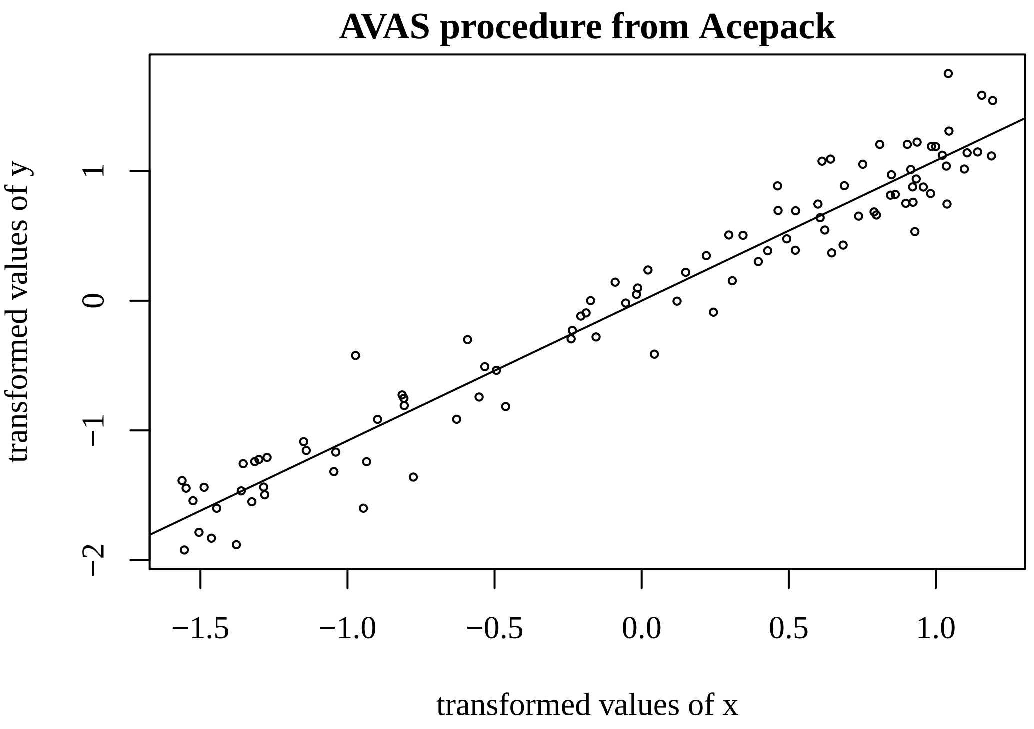
Consider the application of his model to the sine wave that we have used a running example. The transformed x and y are quite pleasantly linear and the linear model fits very nicely.

```
library(acepack)
avasfit ← avas(x, y)
```

```
[1] 100   1
```

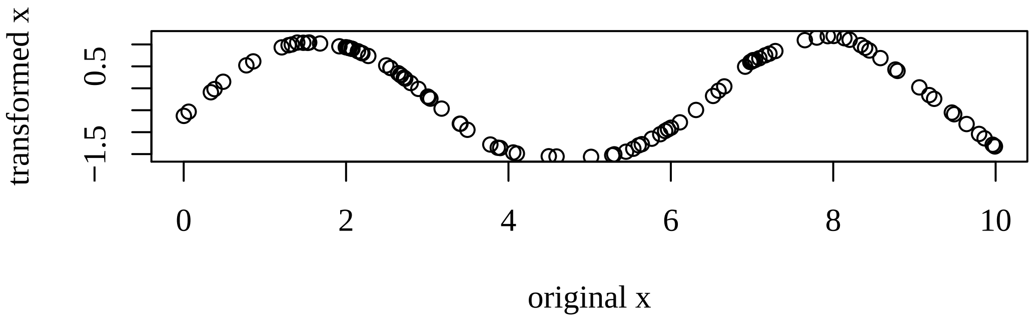
```
plot(avasfit$tx, avasfit$ty, main = "AVAS procedure from Acepack"
   , xlab = "transformed values of x", ylab = "transformed values
```

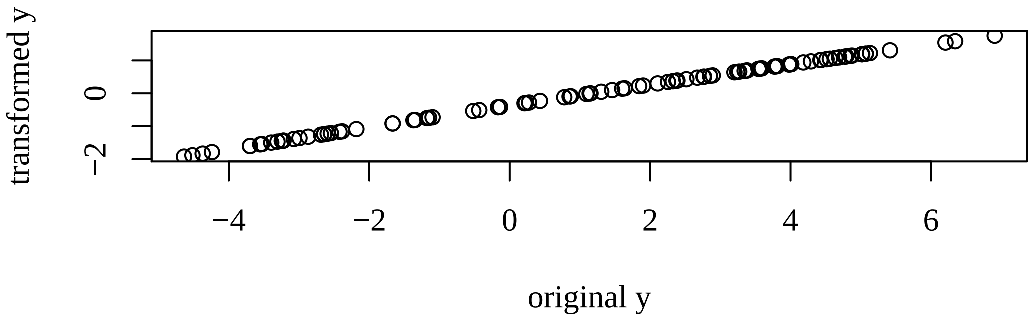
```
    of y",
     type = "n")
points(avasfit$tx, avasfit$ty, cex = 0.5)
abline(lm(avasfit$ty ~ avasfit$tx))
```



What do we have to do to x and y in order to achieve such a beautiful finding?

```
par(mfcol = c(2, 1))
plot(avasfit$x, avasfit$tx, ylab = "transformed x", xlab = "
   original x")
plot(avasfit$y, avasfit$ty, ylab = "transformed y", xlab = "
   original y")
```





Would you rather bend the line, or bend the data, or both? To tell you the truth, I can't quite decide myself. The avas procedure can be applied to many independent variables, and the user can require the program to leave some untransformed or subject some only to monotonic transformations.

I have not seen any research on applying these ideas to logistic regression or other frequently used tools. On the other hand, there has been very eager research on the application of splines and loess in the generalized linear model, as evidenced in packages like gam or mgcv.

But if I had to bet \$100 on a fit of a linear model, I'd be really tempted to take Tibshirani's suggestion.