Notes for ISL Chapter 7: Moving Beyond Linearity

Justin Burruss

2025-12-14

Background

The Python code and notes below are for the ISL study group. This is to try out some of the concepts from Chapter 7 on the `Wage` data set. The document was created in RMarkdown with the Python code running via the `reticulate` library plus a little LaTeX.

Our ground rule: when implementing concepts from the chapter, just use basic Python + Pandas + NumPy. It's OK to use more when visualizing or evaluating results.

Loading the data set

The Pandas library ("Panda" as in "**Pan**el **da**ta") makes it easy to load the CSV and offers some functionality similar to `data.frame` or `data.table` in R.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline

# read CSV
c = pd.read_csv(r"E:\docs\Classes\ISL\Wage.csv")
```

For our `X` and `Y` we will use `age` and `wage`.

```
X = c['age'].astype('float').values
Y = c['wage'].values
```

Natural Cubic Splines

Figure 7.5 in the text shows us a natural cubic spline for `wage` vs. `age` with knots at the 25th, 50th, and 75th percentiles of `age`. Let's see if we can re-create it, then use for a regression. We can first attempt something more concrete and hard-coded for this specific problem, then abstract it later.

We have our `X` and `Y` already loaded, so first step is to select those three knots.

Selecting knots

To re-create what's in the text we need to select three knots at the 25th, 50th, and 75th percentiles of `age`. This is straightforward using `np.percentile`. We're building functions of `X`, so to deal with multiple `Y` values at the same `X`, we take the mean `Y` at each distinct `X`.

```
# For our curve, identify distinct X, taking the mean of Ys at that X
unique_x, mean_y = c.groupby('age') \
    .agg(y=('wage', 'mean')) \
    .pipe(lambda x: (x.index.astype('float').values, x['y'].values))

# Identify three knots at the 25th, 50th, and 75 percentile
knot_idx = np.searchsorted(unique_x, np.percentile(X, [25, 50, 75]))
knot_y = mean_y[knot_idx]
knot_x = unique_x[knot_idx]
```

Cubics

By "cubic" we mean (following variable conventions of the text) a polynomial function of the form

$$C(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$$

where β_3 is not zero. Another way to write these is $f(x) = ax^3 + bx^2 + cx + d$. This is just to say that we have a polynomial with a third power term.

A *cubic spline* is going to give us a cubic between each point of interest. We chose three knots, so with three knots we have five `X` values of interest: $x_0 = min(X)$, 25th percentile knot x_1, 50th percentile knot x_2, 75th percentile knot x_3, and $x_4 = max(X)$.

Five `X` values yields four segments: the segment from x_0 to x_1, the segment from x_1 to x_2, etc. Thus our spline $S(x)$ will have four cubics defined piecewise:

$$S(x) = \begin{cases} C_1(x), & x_0 \leq x \leq x_1 \\ C_2(x), & x_1 \leq x \leq x_2 \\ C_3(x), & x_2 \leq x \leq x_3 \\ C_4(x), & x_3 \leq x \leq x_4 \end{cases}$$

Each cubic has four coefficients, and we have four cubics, so in total we have sixteen coefficients:

$$\begin{aligned} C_1(x) &= \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 \\ C_2(x) &= \beta_4 + \beta_5 x + \beta_6 x^2 + \beta_7 x^3 \\ C_3(x) &= \beta_8 + \beta_9 x + \beta_{10} x^2 + \beta_{11} x^3 \\ C_4(x) &= \beta_{12} + \beta_{13} x + \beta_{14} x^2 + \beta_{15} x^3 \end{aligned}$$

Our task will be to set those sixteen coefficients such that they meet required conditions.

Conditions

What conditions do we put on our cubics for a cubic spline? To start, each cubic needs to fit the two y_i values at the start and end of the segment. For our four cubics that means these eight conditions:

$$C_1(x_0) = y_0 = \beta_0 + \beta_1 x_0 + \beta_2 x_0^2 + \beta_3 x_0^3 \quad \text{condition 1}$$
$$C_1(x_1) = y_1 = \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \beta_3 x_1^3 \quad \text{condition 2}$$
$$C_2(x_1) = y_1 = \beta_4 + \beta_5 x_1 + \beta_6 x_1^2 + \beta_7 x_1^3 \quad \text{condition 3}$$
$$C_2(x_2) = y_2 = \beta_4 + \beta_5 x_2 + \beta_6 x_2^2 + \beta_7 x_2^3 \quad \text{condition 4}$$
$$C_3(x_2) = y_2 = \beta_8 + \beta_9 x_2 + \beta_{10} x_2^2 + \beta_{11} x_2^3 \quad \text{condition 5}$$
$$C_3(x_3) = y_3 = \beta_8 + \beta_9 x_3 + \beta_{10} x_3^2 + \beta_{11} x_3^3 \quad \text{condition 6}$$
$$C_4(x_3) = y_3 = \beta_{12} + \beta_{13} x_3 + \beta_{14} x_3^2 + \beta_{15} x_3^3 \quad \text{condition 7}$$
$$C_4(x_4) = y_4 = \beta_{12} + \beta_{13} x_4 + \beta_{14} x_4^2 + \beta_{15} x_4^3 \quad \text{condition 8}$$

Notice that these eight conditions also capture the requirement that the cubics yield the same value at each overlapping point: $C_1(x_1) = y_1 = C_2(x_1)$, etc. If they did not match we would end up with discontinuities.

As the book notes on page 296, two additional types of constraints are that the cubics yield the same derivative at each overlapping point and the same second derivative at each overlapping point. These will keep the assembled the curve smooth.

The first derivative $C'(x)$ for our cubic $C(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$ is:

$$C'(x) = \beta_1 + 2\beta_2 x + 3\beta_3 x^2 \quad \text{power rule of derivatives}$$

So we add three more conditions for the first derivatives:

$$C_1'(x_1) = C_2'(x_1)$$
$$C_2'(x_2) = C_3'(x_2)$$
$$C_3'(x_3) = C_4'(x_3)$$

Which when we plug in our first derivatives comes out to:

$$\beta_1 + 2\beta_2 x_1 + 3\beta_3 x_1^2 = \beta_5 + 2\beta_6 x_1 + 3\beta_7 x_1^2 \quad \text{condition 9}$$
$$\beta_5 + 2\beta_6 x_2 + 3\beta_7 x_2^2 = \beta_9 + 2\beta_{10} x_2 + 3\beta_{11} x_2^2 \quad \text{condition 10}$$
$$\beta_9 + 2\beta_{10} x_3 + 3\beta_{11} x_3^2 = \beta_{13} + 2\beta_{14} x_3 + 3\beta_{15} x_3^2 \quad \text{condition 11}$$

Then the second derivative is:

$$C''(x) = 2 + (2)(3)\beta_3 x = 2 + 6\beta_3 x \quad \text{power rule of derivatives}$$

We add three more conditions for the second derivatives:

$$C_1''(x_1) = C_2''(x_1)$$
$$C_2''(x_2) = C_3''(x_2)$$
$$C_3''(x_3) = C_4''(x_3)$$

Which when we plug in our second derivatives comes out to:

$$2\beta_2 + 6\beta_3 x_1 = 2\beta_6 + 6\beta_7 x_1 \quad \text{condition 12}$$
$$2\beta_6 + 6\beta_7 x_2 = 2\beta_{10} + 6\beta_{11} x_2 \quad \text{condition 13}$$
$$2\beta_{10} + 6\beta_{11} x_3 = 2\beta_{14} + 6\beta_{15} x_3 \quad \text{condition 14}$$

Finally we have two more conditions to make this a "natural" cubic spline: the second derivative at each endpoint needs to be zero.

$$C_1''(x_0) = 2\beta_2 + 6\beta_3 x_0 = 0 \qquad \text{condition 15}$$
$$C_4''(x_4) = 2\beta_{14} + 6\beta_{15} x_4 = 0 \qquad \text{condition 16}$$

The text frames this in a geometric sense as a requirement that the "function is required to be linear at the boundary".

Knowing all our requirements we move on to finding the sixteen coefficients.

Coefficients

To find the coefficients, let's try simply representing our conditions as given: we can set up a system of sixteen equations with sixteen unknowns and solve. If we continue with the book notation we'll be typing all those $\beta_0, \beta_1, \ldots$ coefficients over and over again, so let's adopt matrix notation and try the $A\mathbf{x} = \mathbf{b}$ approach. We will then apply the hand-coded version and a fit from `scipy` to the same data and compare.

We're already using the variable name x, so instead of $A\mathbf{x} = \mathbf{b}$ let's use $A\mathbf{w} = \mathbf{b}$ ('w' as in weights).

Our `A` will be a square 16x16 matrix, our `b` a vector of length 16.

```
A = np.zeros((16, 16))
b = np.zeros(16)
```

We have just the five points of interest: the edges plus the three knots. We can use `np.concatenate` to stick these together.

```
points_x = np.concatenate(([unique_x[0]], knot_x, [unique_x[-1]]))
points_y = np.concatenate(([mean_y[0]], knot_y, [mean_y[-1]]))
```

As we fill in the elements of A, let's follow the convention that the first four columns are for the coefficients of the first cubic, the second four for the second cubic, and so forth, so we're working left to right. So β_0 is the leftmost column, β_{15} is the rightmost. We'll start with conditions 1 though 8. Let's do this in a for loop as using `slice` should be less error-prone than manually counting columns.

```
# Our matrix A columns are for Beta_0, Beta_1, ..., Beta_15.
# Load each condition.
cond_i = 0

#  C_i(x_{i-1}) = y_{i-1} and C_i(x_i) = y_i = 8 conditions
for i in range(1, 5):
    cols = slice(4*(i-1), (4*(i-1))+4)
    A[cond_i, cols] = \
        [1, points_x[i-1], points_x[i-1]**2, points_x[i-1]**3]
    b[cond_i] = points_y[i-1]
    cond_i += 1
    A[cond_i, cols] = \
        [1, points_x[i], points_x[i]**2, points_x[i]**3]
    b[cond_i] = points_y[i]
    cond_i += 1
```

Next up are conditions nine through 14: the six first derivative and second derivative conditions. These all have a left side equaling a right side, so we will encode their difference in `A` and leave `b` as zero. To illustrate, consider condition 12:

$$
\begin{aligned}
C_1''(x_1) &= C_2''(x_1) && \text{must have same second derivative at } x_1 \\
2\beta_2 + 6\beta_3 x_1 &= 2\beta_6 + 6\beta_7 x_1 && \text{substitute to get condition 12} \\
2\beta_2 + 6\beta_3 - (2\beta_6 + 6\beta_7 x_1) &= 2\beta_6 + 6\beta_7 x_1 - (2\beta_6 + 6\beta_7 x_1) && \text{subtract from both sides} \\
2\beta_2 + 6\beta_3 - 2\beta_6 + 6\beta_7 x_1 &= 0 && \text{condition 12 final form}
\end{aligned}
$$

So we just set the A values to the difference and leave `b` as-is.

```
#  C'_i(x_i)  = C'_{i+1}(x_i) n-1 = 3 conditions
#  C''_i(x_i) = C''_{i+1}(x_i): n-1 = 3 conditions
for i in range(1, 4):
    cols_l = slice(4*(i-1), (4*(i-1))+4)
    cols_r = slice(4*i, 4*i+4)
    A[cond_i, cols_l] = \
        [0,  1,  2*points_x[i],  3*points_x[i]**2]  # C'_i(x_i)
    A[cond_i, cols_r] = \
        [0, -1, -2*points_x[i], -3*points_x[i]**2]  # - C'_{i+1}(x_i)
    cond_i += 1
    A[cond_i, cols_l] = \
        [0,  0,  2,  6*points_x[i]]  # C''_i(x_i)
    A[cond_i, cols_r] = \
        [0,  0, -2, -6*points_x[i]]  # - C''_{i+1}(x_i)
    cond_i += 1
```

Finally we have our last two conditions. Again, no need to set `b` as it is already zero.

```
# C''_1(x_0) = 0 and C''_4(x_4) = 0 [last 2 conditions]
A[14, 0:4]   = [0, 0, 2, 6*points_x[0]]
A[15, 12:16] = [0, 0, 2, 6*points_x[4]]
```

Now the A and $\mathbf{b}$ of $A\mathbf{w} = \mathbf{b}$ is assembled. Solving is a one-liner.

```
w = np.linalg.solve(A, b)
```

We build our four cubics using the coefficients from `w`. NumPy has a class named `Polynomial` for this purpose. Let's use `C` for our cubics to match our notation. To build each cubic we simply pass the four coefficients from `w`. Since these are piecewise defined let's also set the domain and window. We ought to print them as well just to validate that we have four cubics on four intervals in the expected order.

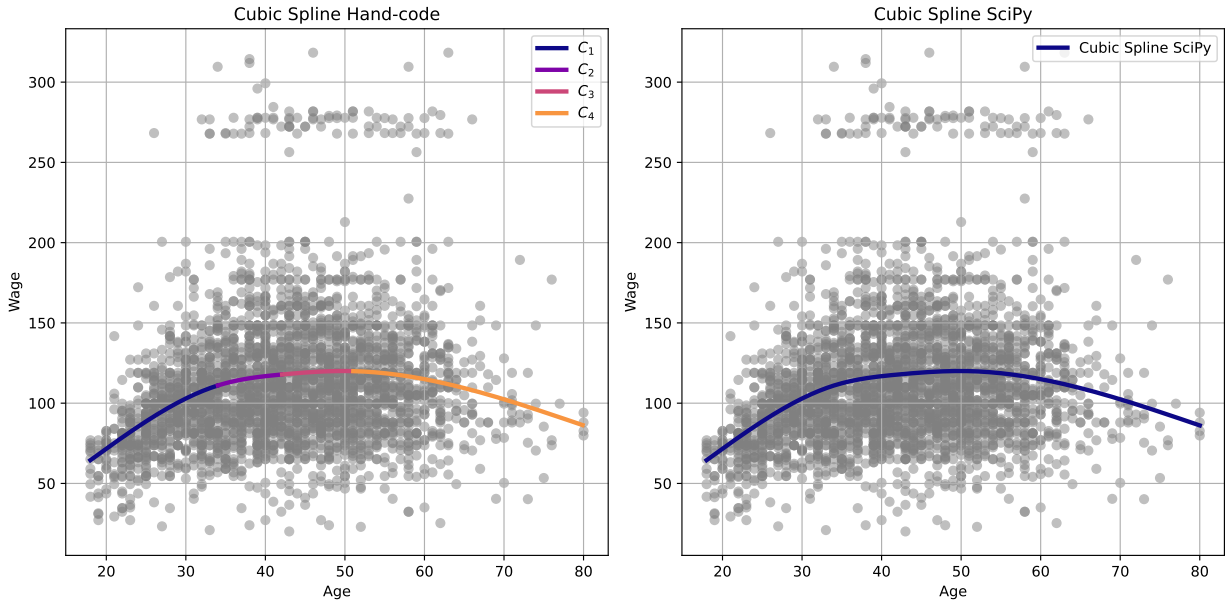
```
# create our four cubics
C = []
for i in range(len(points_x)-1):
    C.append(
        np.polynomial.Polynomial(w[4*i:(4*i)+4],
                                 domain=[points_x[i], points_x[i+1]],
                                 window=[points_x[i], points_x[i+1]]))

# print the cubics
for i, p in enumerate(C):
    a, b = points_x[i], points_x[i+1]
    print(f"C{i+1} [{a}, {b}]: ", p)
```

```
## C1 [18.0, 34.0]:  15.40849454 + 1.01980481 x + 0.14225964 x**2 - 0.00263444 x**3
## C2 [34.0, 42.0]:  -278.15973851 + 26.9228842 x - 0.61959564 x**2 + 0.00483473 x**3
## C3 [42.0, 51.0]:  200.0690368 - 7.23631404 x + 0.19371861 x**2 - 0.00162014 x**3
## C4 [51.0, 80.0]:  -97.42871854 + 10.26355392 x - 0.14941606 x**2 + 0.00062257 x**3
```

We see four polynomials, each has a third power term, and the internal boundaries of each interval line up as required. So far so good.

Let's visualize the results to validate. The line in the book "goes up, levels off, then goes back down again", so we expect that kind of path. We will run a cubic spline using SciPy and do a side-by-side for a tighter comparison. For our spline let's color and label each individual cubic.

```
# 500 points for a smooth curve
plotting_x = np.linspace(min(unique_x), max(unique_x), 500)
plotting_y = np.zeros(len(plotting_x))

# Fit a cubic spline with specified knots
spline = CubicSpline(points_x, points_y, bc_type='natural', extrapolate=True)
y_spline = spline(plotting_x)

# plot side-by-side
colors = plt.cm.plasma(np.linspace(0, 1, len(points_x)))
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.scatter(X, Y, facecolor='gray', alpha=0.5)
for i in range(len(points_x)-1):
    indices = np.where((plotting_x >= points_x[i]) & (plotting_x < points_x[i+1]))
    ax = plt.plot(plotting_x[indices],
                  C[i](plotting_x[indices]),
                  lw=3, color=colors[i], label=f"$C_{i+1}$")
plt.title('Cubic Spline Hand-code')
plt.xlabel('Age')
plt.ylabel('Wage')
plt.legend()
plt.grid()

plt.subplot(1, 2, 2)
plt.scatter(X, Y, facecolor='gray', alpha=0.5)
plt.plot(plotting_x, y_spline, label='Cubic Spline SciPy', color=colors[0], lw=3)
plt.title('Cubic Spline SciPy')
plt.xlabel('Age')
plt.ylabel('Wage')
plt.legend()
plt.grid()

plt.tight_layout()
plt.show()
```



The hand-code matches the SciPy cubic spline and the image from the text.

With that validated, let's compare a linear regression fit against all 3,000 `X` and `Y` with this natural cubic spline fit on just the five `X` values and mean `Y` at those values. Will the training error be much better with the cubic spline?

Comparing with Linear Regression

Let's compare with a linear regression, just checking the training RSS.

```
# linear regression
b = Y
A = np.column_stack((X, np.ones(len(X))))

x, RSS, rank, S = np.linalg.lstsq(A, b, rcond=None)

L = np.polynomial.Polynomial(x[::-1],
                         domain=[points_x[i], points_x[i+1]],
                         window=[points_x[i], points_x[i+1]])

# collect RSS at each segment
RSS_lr = []
RSS_cs = []
for i in range(len(points_x)-1):
    if i == len(points_x)-2:
        indices = (X >= points_x[i]) & (X <= points_x[i+1])
    else:
        indices = (X >= points_x[i]) & (X < points_x[i+1])
    RSS_cs.append((Y[indices] - C[i](X[indices]))**2)
    RSS_lr.append((Y[indices] - L(X[indices]))**2)


# assemble lines to print all at once
```

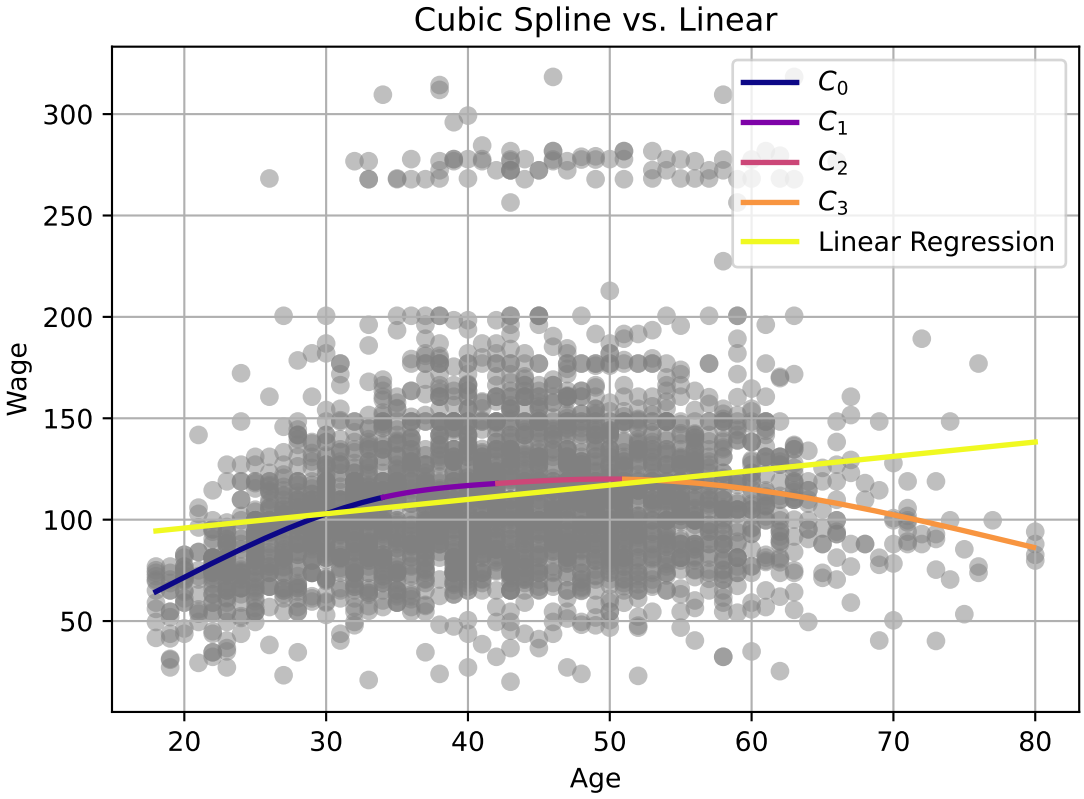
```
lines = ['Training RSS Comparison\n',
         f'Linear Regression    = {sum(np.sum(r) for r in RSS_lr).round(1)}\n',
         f'Natural Cubic Spline = {sum(np.sum(r) for r in RSS_cs).round(1)}\n',
         'By segment:\n',
         f" {'Seg':>3}{'RSS Linear':>14}",
         f"{'RSS Cubic Spline':>20}{'Spline Advantage':>20}\n"]

for i in range(len(RSS_lr)):
    lines.append(f" {i+1:>3}")
    lines.append(f"{np.sum(RSS_lr[i]).round(1):>14}")
    lines.append(f"{np.sum(RSS_cs[i]).round(1):>20}")
    lines.append(f"{(np.sum(RSS_lr[i]) - np.sum(RSS_cs[i])).round(1):>20}\n")

print(''.join(lines))
```

```
## Training RSS Comparison
## Linear Regression    = 5022216.1
## Natural Cubic Spline = 4779112.9
## By segment:
##  Seg    RSS Linear    RSS Cubic Spline    Spline Advantage
##    1      710639.6            599797.2            110842.4
##    2     1301424.8           1258069.8             43354.9
##    3     1521836.8           1497180.5             24656.3
##    4     1488314.9           1424065.3             64249.6
```

It does indeed yield a better fit, especially at that first segment.

Let's plot an overlay the two.

```
# overlay
plt.figure(1)
plt.scatter(X, Y, facecolor='gray', alpha=0.5)
for i in range(len(points_x)-1):
    indices = np.where((plotting_x >= points_x[i]) & (plotting_x < points_x[i+1]))
    ax = plt.plot(plotting_x[indices],
                  C[i](plotting_x[indices]),
                  lw=2, color=colors[i], label=f"$C_{i}$")
plt.plot(plotting_x, L(plotting_x), label='Linear Regression', color=colors[-1], lw=2)
plt.title('Cubic Spline vs. Linear')
plt.xlabel('Age')
plt.ylabel('Wage')
plt.legend()
plt.grid()

plt.show()
```



Everything looks proper.

Now that we know the hand-code logic works, let's revisit that logic to generalize it rather than hard-coding the number of coefficients and so forth.

Generalizing for K knots

For a more general natural cubic spline function we need to support some given number of knots: at least one knot, but no more knots than the number of unique X values minus the two end points. Let's give the number of knots the variable name K ('K' as in knot).

If K is the number of knots, then will will have $K+2$ points of interest $x_0, x_1, \ldots, x_{K+1}$ and $K+1$ cubics $C_1, C_2, \ldots, C_{K+1}$ over $K+1$ segments.

Immediately we can streamline our conditions. What used to be hard-coded as sixteen rules rolls up into just these six, which then work for any valid number of knots:

$C_i(x_{i-1}) = y_{i-1}$	$for\ i = 1...K+1$	$C(x)$ leftmost x condition (1)
$C_i(x_i) = y_i$	$for\ i = 1...K+1$	$C(x)$ rightmost x condition (2)
$C'_i(x_i) = C'_{i+1}(x_i)$	$for\ i = 1...K$	$C'(x)$ internal x condition (3)
$C''_i(x_i) = C''_{i+1}(x_i)$	$for\ i = 1...K$	$C''(x)$ internal x condition (4)
$C''_1(x_0) = 0$		$C''(x)$ leftmost x condition (5)
$C''_{K+1}(x_{K+1}) = 0$		$C''(x)$ rightmost x condition (6)

We have $K+1$ cubics, each with four coefficients, so we get $4(K+1)$ coefficients. Rather than numbering the coefficients $\beta_0, \beta_1, \ldots, \beta_{4(K+1)-1}$ let's identify using $\beta_{0,i}$, $\beta_{1,i}$, $\beta_{2,i}$, $\beta_{3,i}$ $\quad i = 1...K+1$. So for example, the first coefficient of cubic C_1 is $\beta_{0,1}$ and the first coefficient of cubic C_2 is $\beta_{0,2}$.

With K defined as the number of knots, $x_0, x_1, ..., x_{K+1}$ as our $K+2$ points of interest, and our coefficients indexed by cubic number and term we now have one expression to capture any and all our cubics:

$$y_i = C_i(x) = \beta_{0,i} + \beta_{1,i}x + \beta_{2,i}x^2 + \beta_{3,i}x^3 \quad x_{i-1} \le x \le x_i \quad i = 1 \dots K+1$$

This is already much cleaner, and it makes it easier to see patterns. Observe that if x was zero, then the x, x^2, and x^3 terms zero out, leaving $C_i(0) = \beta_{0,i}$, with the upshot that you could simply set intercept term $\beta_{0,i}$ to the desired y_i value for all $i = 1 \dots K+1$. To bring some zeroes into the picture, what if we depart a little from the text and define our cubics differently so that we "reset" at every leftmost point of the cubic, i.e., when $x = x_{i-1}$? We could do that by replacing the x terms with $x - x_{i-1}$, like this:

$$C_i(x) = \beta_{0,i} + \beta_{1,i}(x - x_{i-1}) + \beta_{2,i}(x - x_{i-1})^2 + \beta_{3,i}(x - x_{i-1})^3 \quad x_{i-1} \le x \le x_i \quad i = 1 \dots K+1$$

Thus our cubic spline having K knots with $K+2$ points of interest (leftmost x_0, knots $x_1, ..., x_K$, and rightmost x_{K+1}) is now defined as:

$$S(x) = \begin{cases} C_1(x) = \beta_{0,1} + \beta_{1,1}(x - x_0) + \beta_{2,1}(x - x_0)^2 + \beta_{3,1}(x - x_0)^3, & x_0 \le x \le x_1 \\ C_2(x) = \beta_{0,2} + \beta_{1,2}(x - x_1) + \beta_{2,2}(x - x_1)^2 + \beta_{3,2}(x - x_1)^3, & x_1 \le x \le x_2 \\ \vdots & \\ C_{K+1}(x) = \beta_{0,K+1} + \beta_{1,K+1}(x - x_K) + \beta_{2,K+1}(x - x_K)^2 + \beta_{3,K+1}(x - x_K)^3, & x_K \le x \le x_{K+1} \end{cases}$$

Revisiting Conditions

Let's revisit the conditions: we can do a little work up front with reformulating things so the computer has less work to do later.

Resets at $C_i(x_{i-1})$

Observe that when we call $C_i(x)$ on x_{i-1} (the leftmost point in its segment), the "reset" at every leftmost point of the cubic means we zero out the x, x^2, and x^3 terms, leaving just the intercept term:

$$\begin{aligned} C_i(x_{i-1}) &= \beta_{0,i} + \beta_{1,i}(x_{i-1} - x_{i-1}) + \beta_{2,i}(x_{i-1} - x_{i-1})^2 + \beta_{3,i}(x_{i-1} - x_{i-1})^3 \\ &= \beta_{0,i} + \beta_{1,i}(0) + \beta_{2,i}(0)^2 + \beta_{3,i}(0)^3 \\ &= \beta_{0,i} \end{aligned}$$

For free we get our $C(x)$ leftmost x condition (1) taken care of simply by setting each $\beta_{0,i}$ to the corresponding y_{i-1} leftmost y value.

$$C_i(x_{i-1}) = y_{i-1} = \beta_{0,i} \quad i = 1 \dots K+1$$

In other words, when we go to set up our $A\mathbf{w} = \mathbf{b}$ we now have some of the unknowns just waiting to be filled in.

With C' we also get the zeroing out property:

$$\begin{aligned} C_i'(x_{i-1}) &= \beta_{1,i} + 2\beta_{2,i}(x_{i-1} - x_{i-1}) + 3\beta_{3,i}(x_{i-1} - x_{i-1})^2 \\ &= \beta_{1,i} + 2\beta_{2,i}(0) + 3\beta_{3,i}(0)^2 \\ &= \beta_{1,i} \end{aligned}$$

Likewise with C'':

$$\begin{aligned} C_i''(x_{i-1}) &= 2\beta_{2,i} + 6\beta_{3,i}(x_{i-1} - x_{i-1}) \\ &= 2\beta_{2,i} + 6\beta_{3,i}(0) \\ &= 2\beta_{2,i} \end{aligned}$$

Let $j = i + 1$ and we can see that $C'_{i+1}(x_i) = C'_j(x_{j-1})$ benefits from a similar zeroing out:

$$\begin{aligned} C'_{i+1}(x_i) &= C'_j(x_{j-1}) && x_{i-1} \le x \le x_i \quad i = 1 \dots K \\ &= \beta_{1,i+1} + 2\beta_{2,i+1}(x_{j-1} - x_{j-1}) + 3\beta_{3,i+1}(x_{j-1} - x_{j-1})^2 \\ &= \beta_{1,i+1} + 2\beta_{2,i+1}(0) + 3\beta_{3,i+1}(0)^2 \\ &= \beta_{1,i+1} \end{aligned}$$

$C''_i(x_{i+1})$ gets the same treatment. Let $j = i + 1$, then we see:

$$\begin{aligned} C''_{i+1}(x_i) &= C''_j(x_{j-1}) && x_{i-1} \le x \le x_i \quad i = 1 \dots K \\ &= 2\beta_{2,i+1} + 6\beta_{3,i+1}(x_{j-1} - x_{j-1}) \\ &= 2\beta_{2,i+1} + 6\beta_{3,i+1}(0) \\ &= 2\beta_{2,i+1} \end{aligned}$$

Already we can see that we'll be able to reduce from 16 equations and 16 unknowns to something simpler.

A variable for intervals

We have a lot of $x_i - x_{i-1}$ going on. Let's define two more variables for all the overlapping involved at the internal segment boundaries—the x intervals and while we're at it the y intervals:

$$\begin{aligned} \Delta x_i &= x_i - x_{i-1} && for\ i = 1 \dots K+1 \\ \Delta y_i &= y_i - y_{i-1} && for\ i = 1 \dots K+1 \end{aligned}$$

Now $C_i(x_i)$ and its derivatives are just:

$$\begin{aligned} C_i(x_i) &= \beta_{0,i} + \beta_{1,i}(x_i - x_{i-1}) + \beta_{2,i}(x_i - x_{i-1})^2 + \beta_{3,i}(x_i - x_{i-1})^3 \quad x_{i-1} \le x \le x_i \quad i = 1 \dots K+1 \\ &= \beta_{0,i} + \beta_{1,i}\Delta x_i + \beta_{2,i}\Delta x_i^2 + \beta_{3,i}\Delta x_i^3 && \text{definition of } \Delta x_i \\ C'_i(x_i) &= \beta_{1,i} + 2\beta_{2,i}(x_i - x_{i-1}) + 3\beta_{3,i}(x_i - x_{i-1})^2 \quad x_{i-1} \le x \le x_i \quad i = 1 \dots K+1 \\ &= \beta_{1,i} + 2\beta_{2,i}\Delta x_i + 3\beta_{3,i}\Delta x_i^2 && \text{definition of } \Delta x_i \\ C''_i(x_i) &= 2\beta_{2,i} + 6\beta_{3,i}(x_i - x_{i-1}) \quad x_{i-1} \le x \le x_i \quad i = 1 \dots K+1 \\ &= 2\beta_{2,i} + 6\beta_{3,i}\Delta x_i && \text{definition of } \Delta x_i \end{aligned}$$

With these deltas defined, let's revisit condition (4) to see how we can use the "leftmost point reset" property to solve for $\beta_{3,i}$ in terms of $\beta_{2,i}$, $\beta_{2,i+1}$, and Δx_i.

Solving for $\beta_{3,i}$

With a little algebra we can solve for $\beta_{3,i}$ for $i = 1...K$.

$$\begin{aligned} C''_i(x_i) &= C''_{i+1}(x_i) && for\ i = 1...K && C''(x) \text{ internal } x \text{ condition (4)} \\ 2\beta_{2,i} + 6\beta_{3,i}\Delta x_i &= C''_{i+1}(x_i) \\ 2\beta_{2,i} + 6\beta_{3,i}\Delta x_i &= 2\beta_{2,i+1} && C''_{i+1}(x_i) = 2\beta_{2,i+1} \\ 2\beta_{2,i} - 2\beta_{2,i} + 6\beta_{3,i}\Delta x_i &= 2\beta_{2,i+1} - 2\beta_{2,i} \\ 6\beta_{3,i}\Delta x_i &= 2\beta_{2,i+1} - 2\beta_{2,i} \\ \beta_{3,i} &= \frac{2\beta_{2,i+1} - 2\beta_{2,i}}{6\Delta x_i} && \text{after dividing both sides by } 6\Delta x_i \\ \beta_{3,i} &= \frac{\beta_{2,i+1} - \beta_{2,i}}{3\Delta x_i} && \text{cancellations} \end{aligned}$$

Next, we'll put $\beta_{1,i}$ in terms of $\beta_{2,i}$, $\beta_{2,i+1}$, Δx_i, and Δy_i.

Solving for $\beta_{1,i}$

To solve for $\beta_{1,i}$, we put a few facts together. Recall:

$$\begin{aligned}
C_i(x_{i-1}) &= y_{i-1} && for\ i = 1...K+1 && C(x) \text{ leftmost } x \text{ condition (1)} \\
C_i(x_{i-1}) &= \beta_{0,i} && for\ i = 1...K+1 && \text{leftmost } x \text{ of segment} \\
C_i(x_i) &= y_i && for\ i = 1...K+1 && C(x) \text{ rightmost } x \text{ condition (2)} \\
\Delta y_i &= y_i - y_{i-1} && for\ i = 1...K+1 && \text{definition of } \Delta y_i
\end{aligned}$$

We use these facts to isolate $\beta_{1,i}$ on one side of an equation.

$$\begin{aligned}
\Delta y_i &= y_i - y_{i-1} && \text{definition of } \Delta y_i \\
&= C_i(x_i) - y_{i-1} && C_i(x_i) = y_i \\
&= C_i(x_i) - \beta_{0,i} && C_i(x_{i-1}) = \beta_{0,i} \\
&= \beta_{0,i} + \beta_{1,i}\Delta x_i + \beta_{2,i}\Delta x_i^2 + \beta_{3,i}\Delta x_i^3 - \beta_{0,i} && C_i(x) \text{ evaluated at } x_i \\
&= \beta_{1,i}\Delta x_i + \beta_{2,i}\Delta x_i^2 + \beta_{3,i}\Delta x_i^3 && \beta_{0,i} \text{ cancellation} \\
\Delta y_i - \beta_{2,i}\Delta x_i^2 &= \beta_{1,i}\Delta x_i + \beta_{2,i}\Delta x_i^2 + \beta_{3,i}\Delta x_i^3 - \beta_{2,i}\Delta x_i^2 \\
\Delta y_i - \beta_{2,i}\Delta x_i^2 &= \beta_{1,i}\Delta x_i + \beta_{3,i}\Delta x_i^3 \\
\Delta y_i - \beta_{2,i}\Delta x_i^2 - \beta_{3,i}\Delta x_i^3 &= \beta_{1,i}\Delta x_i + \beta_{3,i}\Delta x_i^3 - \beta_{3,i}\Delta x_i^3 \\
\Delta y_i - \beta_{2,i}\Delta x_i^2 - \beta_{3,i}\Delta x_i^3 &= \beta_{1,i}\Delta x_i \\
(\Delta y_i - \beta_{2,i}\Delta x_i^2 - \beta_{3,i}\Delta x_i^3)\frac{1}{\Delta x_i} &= \beta_{1,i}\Delta x_i \frac{1}{\Delta x_i} \\
(\Delta y_i - \beta_{2,i}\Delta x_i^2 - \beta_{3,i}\Delta x_i^3)\frac{1}{\Delta x_i} &= \beta_{1,i}
\end{aligned}$$

After cancellations and substituting in the solution for $\beta_{3,i}$ we get $\beta_{1,i}$ in terms of Δx_i, $\beta_{2,i}$, and $\beta_{2,i+1}$, and Δy_i over $i = 1 \dots K+1$.

$$\begin{aligned}
\beta_{1,i} &= (\Delta y_i - \beta_{3,i}\Delta x_i^3 - \beta_{2,i}\Delta x_i^2)\frac{1}{\Delta x_i} \\
&= \frac{\Delta y_i}{\Delta x_i} - (\beta_{3,i}\Delta x_i^3 + \beta_{2,i}\Delta x_i^2)\frac{1}{\Delta x_i} \\
&= \frac{\Delta y_i}{\Delta x_i} - \frac{\beta_{3,i}\Delta x_i^3}{\Delta x_i} - (\beta_{2,i}\Delta x_i^2)\frac{1}{\Delta x_i} \\
&= \frac{\Delta y_i}{\Delta x_i} - \frac{\beta_{3,i}\Delta x_i^3}{\Delta x_i} - \frac{\beta_{2,i}\Delta x_i^2}{\Delta x_i} \\
&= \frac{\Delta y_i}{\Delta x_i} - \beta_{3,i}\Delta x_i^2 - \beta_{2,i}\Delta x_i && \Delta x_i \text{ cancellations} \\
&= \frac{\Delta y_i}{\Delta x_i} - (\frac{\beta_{2,i+1} - \beta_{2,i}}{3\Delta x_i})\Delta x_i^2 - \beta_{2,i}\Delta x_i && \text{sub for } \beta_{3,i} \\
&= \frac{\Delta y_i}{\Delta x_i} - (\frac{\beta_{2,i+1} - \beta_{2,i}}{3})\Delta x_i - \beta_{2,i}\Delta x_i && \Delta x_i \text{ cancellations} \\
&= \frac{\Delta y_i}{\Delta x_i} - (\frac{\beta_{2,i+1} - \beta_{2,i}}{3} + \beta_{2,i})\Delta x_i && \text{combine } \Delta x_i \text{ terms} \\
&= \frac{\Delta y_i}{\Delta x_i} - (\frac{\beta_{2,i+1} - \beta_{2,i} + 3\beta_{2,i}}{3})\Delta x_i && \beta_{2,i} = \frac{3\beta_{2,i}}{3} \text{ and combine} \\
&= \frac{\Delta y_i}{\Delta x_i} - (\frac{\beta_{2,i+1} + 2\beta_{2,i}}{3})\Delta x_i \\
&= \frac{\Delta y_i}{\Delta x_i} - \frac{\Delta x_i}{3}(\beta_{2,i+1} + 2\beta_{2,i})
\end{aligned}$$

So we got $\beta_{0,i}$ for free and have found $\beta_{1,i}$ and $\beta_{3,i}$. Last up is $\beta_{2,i}$.

A tridiagonal for $\beta_{2,i}$

Let $j = i + 1$ and we can see how to get $\beta_{1,i+1}$ over $i = 1 \dots K$.

$$\begin{aligned}
\beta_{1,i} &= \frac{\Delta y_i}{\Delta x_i} - \frac{\Delta x_i}{3}(\beta_{2,i+1} + 2\beta_{2,i}) && i = 1 \dots K+1 \\
\beta_{1,j} &= \frac{\Delta y_j}{\Delta x_j} - \frac{\Delta x_j}{3}(\beta_{2,j+1} + 2\beta_{2,j}) && i = 1 \dots K \\
\beta_{1,i+1} &= \frac{\Delta y_{i+1}}{\Delta x_{i+1}} - \frac{\Delta x_{i+1}}{3}(\beta_{2,i+2} + 2\beta_{2,i+1}) && j = i+1
\end{aligned}$$

Our first move is to set up an equation using the first derivative continuity condition, taking care to note that we're just in $i = 1 \dots K$ inner points territory.

$$\begin{aligned}
C_i'(x_i) &= C_{i+1}'(x_i) \quad for\ i = 1 \dots K && \text{condition 3} \\
&= \beta_{1,i+1} && C_{i+1}'(x_i) = \beta_{1,i+1} \\
\beta_{1,i} + 2\beta_{2,i}\Delta x_i + 3\beta_{3,i}\Delta x_i^2 &= \beta_{1,i+1}
\end{aligned}$$

Now come the substitutions using the previously derived expressions for $\beta_{1,i}$, $\beta_{1,i+1}$, and $\beta_{3,i}$.

$$\begin{aligned}
\beta_{1,i+1} &= \beta_{1,i} + 2\beta_{2,i}\Delta x_i + 3\beta_{3,i}\Delta x_i^2 \\
\frac{\Delta y_{i+1}}{\Delta x_{i+1}} - \frac{\Delta x_{i+1}}{3}(\beta_{2,i+2} + 2\beta_{2,i+1}) &= \beta_{1,i} + 2\beta_{2,i}\Delta x_i + 3\beta_{3,i}\Delta x_i^2 \\
&= \frac{\Delta y_i}{\Delta x_i} - \frac{\Delta x_i}{3}(\beta_{2,i+1} + 2\beta_{2,i}) + 2\beta_{2,i}\Delta x_i + 3\beta_{3,i}\Delta x_i^2 \\
&= \frac{\Delta y_i}{\Delta x_i} - \frac{\Delta x_i}{3}(\beta_{2,i+1} + 2\beta_{2,i}) + 2\beta_{2,i}\Delta x_i + 3(\frac{\beta_{2,i+1} - \beta_{2,i}}{3\Delta x_i})\Delta x_i^2 \\
&= \frac{\Delta y_i}{\Delta x_i} - \frac{\Delta x_i}{3}(\beta_{2,i+1} + 2\beta_{2,i}) + 2\beta_{2,i}\Delta x_i + (\frac{\beta_{2,i+1} - \beta_{2,i}}{\Delta x_i})\Delta x_i^2 \\
&= \frac{\Delta y_i}{\Delta x_i} - \frac{\Delta x_i}{3}(\beta_{2,i+1} + 2\beta_{2,i}) + 2\beta_{2,i}\Delta x_i + (\beta_{2,i+1} - \beta_{2,i})\Delta x_i \\
&= \frac{\Delta y_i}{\Delta x_i} - \frac{\Delta x_i}{3}(\beta_{2,i+1} + 2\beta_{2,i}) + (\beta_{2,i+1} - \beta_{2,i} + 2\beta_{2,i})\Delta x_i \\
&= \frac{\Delta y_i}{\Delta x_i} - \frac{\Delta x_i}{3}(\beta_{2,i+1} + 2\beta_{2,i}) + (\beta_{2,i+1} + \beta_{2,i})\Delta x_i \\
\frac{\Delta y_{i+1}}{\Delta x_{i+1}} &= \frac{\Delta y_i}{\Delta x_i} - \frac{\Delta x_i}{3}(\beta_{2,i+1} + 2\beta_{2,i}) + (\beta_{2,i+1} + \beta_{2,i})\Delta x_i + \frac{\Delta x_{i+1}}{3}(\beta_{2,i+2} + 2\beta_{2,i+1}) \\
\frac{\Delta y_{i+1}}{\Delta x_{i+1}} - \frac{\Delta y_i}{\Delta x_i} &= -\frac{\Delta x_i}{3}(\beta_{2,i+1} + 2\beta_{2,i}) + (\beta_{2,i+1} + \beta_{2,i})\Delta x_i + \frac{\Delta x_{i+1}}{3}(\beta_{2,i+2} + 2\beta_{2,i+1}) \\
&= \frac{\Delta x_i}{3}(\beta_{2,i} + 2\beta_{2,i+1}) + \frac{\Delta x_{i+1}}{3}(\beta_{2,i+2} + 2\beta_{2,i+1}) \\
&= \frac{1}{3}(\beta_{2,i}\Delta x_i) + \frac{2}{3}(\beta_{2,i+1}\Delta x_i) + \frac{1}{3}(\beta_{2,i+2}\Delta x_{i+1}) + \frac{2}{3}(\beta_{2,i+1}\Delta x_{i+1}) \\
&= \frac{1}{3}(\beta_{2,i}\Delta x_i) + \frac{2}{3}(\beta_{2,i+1})(\Delta x_i + \Delta x_{i+1}) + \frac{1}{3}(\beta_{2,i+2}\Delta x_{i+1})
\end{aligned}$$

Let's multiply by three so when we go to code this we're not plugging in fractions into the A elements.

$$3(\frac{\Delta y_{i+1}}{\Delta x_{i+1}} - \frac{\Delta y_i}{\Delta x_i}) = \beta_{2,i}\Delta x_i + 2\beta_{2,i+1}(\Delta x_i + \Delta x_{i+1}) + \beta_{2,i+2}\Delta x_{i+1}$$

This is a good place to pause to test what we have so far.

Checking the result

Let's check using the same knots as before. We expect to get the same curve.

As promised, the work we did up front with reformulating things makes it so the computer has less work to do now: we're going to do this with a 3x3 matrix instead of the 16x16 we had before. Our first and last β_2 are zeroes because we want zero second derivative at the boundaries to make this a *natural cubic spline*, so even though we have five points we really just need to solve for the three β_2 values in the interior points (the knots x_1, x_2, x_3).

First we load the left side of the equation above into the $\mathbf{b}$ of our $A\mathbf{w} = \mathbf{b}$. The `diff` function in NumPy is perfect for giving us our deltas, and the built-in slicing functionality in Python lets distinguish the i points (e.g., `delta_x[:-1]`) from the $i+1$ points (e.g., `delta_x[1:]`).

```
delta_x = np.diff(points_x)
delta_y = np.diff(points_y)

# left hand side
b = 3.0 * ((delta_y[1:] / delta_x[1:]) - (delta_y[:-1] / delta_x[:-1]))
```

NumPy already has a `fill_diagonal` function for filling in our A matrix.

```
A = np.zeros((b.size, b.size))

# right hand side, beta_{2,i} * delta x_{i}, lower diagonal
np.fill_diagonal(A[1:], delta_x[1:-1])

# right hand side, 2 * beta_{2,i+1} * (delta x_i + delta x_{i+1}), main diagonal
np.fill_diagonal(A, 2.0 * (delta_x[:-1] + delta_x[1:]))

# right hand side, beta_{2,i+1} * delta x_{i+1}, upper diagonal
np.fill_diagonal(A[:, 1:], delta_x[1:-1])
```

Solving is as usual a 1-liner. For larger matrices there are shortcuts for solving tridiagonals and other sparse matrices, but this is just a 3x3. We do need to remember to put the zeroes on either side: our first and last β_2 are zeroes because we want zero second derivative at the boundaries to make this a *natural cubic spline*.

```
w = np.concatenate(([0], np.linalg.solve(A, b), [0]))
```

Now we can build our four cubics. Let's bring all that work for the coefficients forward and put here, all in one place:

$$\beta_{0,i} = y_{i-1}$$ the "for free" one

$$\beta_{1,i} = \frac{\Delta y_i}{\Delta x_i} - \frac{\Delta x_i}{3}(\beta_{2,i+1} + 2\beta_{2,i})$$

$$\beta_{2,i} = w_{i-1}$$ what we just solved for with $A\mathbf{w} = \mathbf{b}$

$$\beta_{3,i} = \frac{\beta_{2,i+1} - \beta_{2,i}}{3\Delta x_i}$$

The built-in slicing makes it easy to indicate whether we need the i or the $i-1$.

```
# beta_{0,i} = y_{i-1}
beta_0 = points_y[:-1]

# beta_{1,i} = (delta y_i / delta x_i) - (delta x_i)/3 * (beta_{2,i+1} + 2 beta_{2,i})
beta_1 = delta_y / delta_x - delta_x * (w[1:] + 2*w[:-1]) / 3

# beta_2 we solved for
beta_2 = w[:-1]

# beta_3 = (beta_{2,i+1} - beta_{2,i})/(3 delta x_i)
beta_3 = (w[1:] - w[:-1]) / (3 * delta_x)
```

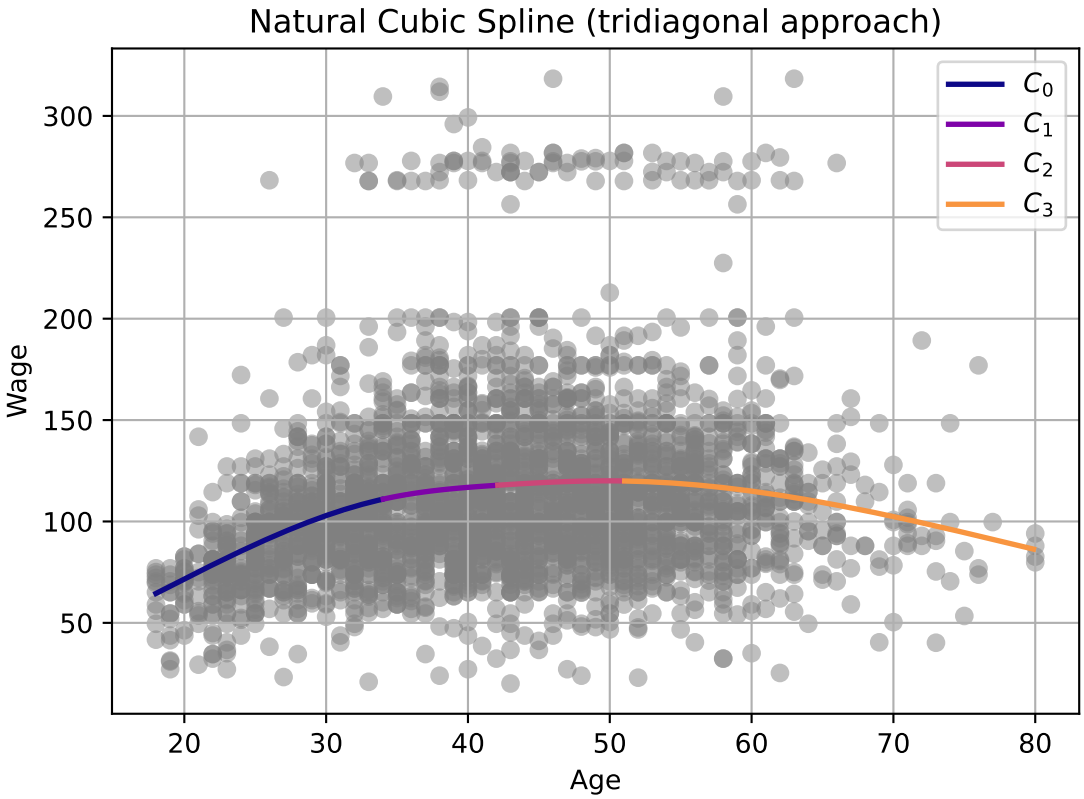
We'll use `Polynomial` again and print. Our coefficients were set up for cubic functions of $\Delta x = x_i - x_{i-1}$, so let's use composition to have the final cubics be functions of `x`.

```
# create our four cubics (use composition since coeffs are for offsets)
C = []
for i in range(len(points_x)-1):
    # local cubic in t = (x - x_i)
    p_local = np.polynomial.Polynomial([
        beta_0[i],
        beta_1[i],
        beta_2[i],
        beta_3[i]])

    # delta_x = x - x_i
    delta = np.polynomial.Polynomial([-points_x[i], 1.0])

    # global polynomial via composition
    p_global = p_local(delta)

    C.append(
        np.polynomial.Polynomial(
            p_global.coef,
            domain=[points_x[i], points_x[i+1]],
            window=[points_x[i], points_x[i+1]]))

# print the cubics
for i, p in enumerate(C):
    a, b = points_x[i], points_x[i+1]
    print(f"C{i+1} [{a}, {b}]: ", p)
```

```
## C1 [18.0, 34.0]:  15.40849454 + 1.01980481 x + 0.14225964 x**2 - 0.00263444 x**3
## C2 [34.0, 42.0]:  -278.15973851 + 26.9228842 x - 0.61959564 x**2 + 0.00483473 x**3
## C3 [42.0, 51.0]:  200.0690368 - 7.23631404 x + 0.19371861 x**2 - 0.00162014 x**3
## C4 [51.0, 80.0]:  -97.42871854 + 10.26355392 x - 0.14941606 x**2 + 0.00062257 x**3
```

Same coefficients, so we're good. We may as well plot.

```
# plot the cubics
plt.figure(1)
plt.scatter(X, Y, facecolor='gray', alpha=0.5)
for i in range(len(points_x)-1):
    indices = np.where((plotting_x >= points_x[i]) & (plotting_x < points_x[i+1]))
    ax = plt.plot(plotting_x[indices],
                  C[i](plotting_x[indices]),
                  lw=2, color=colors[i], label=f"$C_{i}$")
plt.title('Natural Cubic Spline (tridiagonal approach)')
plt.xlabel('Age')
plt.ylabel('Wage')
plt.legend()
plt.grid()
```



Everything looks correct. Now let's tidy it up and put it into a reusable function.

Natural Cubic Spline Function

If we had many data points in `X` we would want to implement this differently, say as a $3 \times n$ banded matrix representation using a banded solver instead of $n \times n$ and a general solver, or to solve using something specifically for tridiagonals like the Thomas algorithm. Notwithstanding these more efficient alternatives, to serve the goal of trying out concepts from Chapter 7 let's just code this up as we've derived it, sparse matrices and all.

```
def nat_cubic_spline(X, Y):
    """Return a list of cubics for a natural cubic spline using a general solver."""
    delta_x = np.diff(X)
    delta_y = np.diff(Y)

    # left hand side
    b = 3.0 * ((delta_y[1:] / delta_x[1:]) - (delta_y[:-1] / delta_x[:-1]))

    # full matrix (OK here, would use banded representation for larger cases)
    A = np.zeros((b.size, b.size))

    # right hand side, beta_{2,i} * delta x_{i}, lower diagonal
    np.fill_diagonal(A[1:], delta_x[1:-1])

    # right hand side, 2 * beta_{2,i+1} * (delta x_i + delta x_{i+1}), main diagonal
    np.fill_diagonal(A, 2.0 * (delta_x[:-1] + delta_x[1:]))

    # right hand side, beta_{2,i+1} * delta x_{i+1}, upper diagonal
    np.fill_diagonal(A[:, 1:], delta_x[1:-1])

    # solve for the interior points (zeroes at the edges for natural cubic spline)
    x = np.concatenate(([0], np.linalg.solve(A, b), [0]))

    # beta_{0,i} = y_{i-1}
    beta_0 = Y[:-1]

    # beta_{1,i} = (delta y_i / delta x_i) - (delta x_i)/3 * (beta_{2,i+1} + 2 beta_{2,i})
    beta_1 = delta_y / delta_x - delta_x * (x[1:] + 2*x[:-1]) / 3

    # beta_2 we solved for
    beta_2 = x[:-1]

    # beta_3 = (beta_{2,i+1} - beta_{2,i})/(3 delta x_i)
    beta_3 = (x[1:] - x[:-1]) / (3 * delta_x)

    # create our four cubics (use composition since coeffs are for offsets)
    C = []
    for i in range(len(X)-1):
        # local cubic in t = (x - x_i)
        p_local = np.polynomial.Polynomial([
            beta_0[i],
            beta_1[i],
            beta_2[i],
            beta_3[i]])

        # delta_x = x - x_i
        delta = np.polynomial.Polynomial([-X[i], 1.0])

        # global polynomial via composition
        p_global = p_local(delta)

        C.append(
            np.polynomial.Polynomial(
```

```
                p_global.coef,
                domain=[X[i], X[i+1]],
                window=[X[i], X[i+1]]))

    return C
```

Now let's try it out and validate we get the same cubics.

```
C = nat_cubic_spline(points_x, points_y)

# print the cubics
for i, p in enumerate(C):
    a, b = points_x[i], points_x[i+1]
    print(f"C{i+1} [{a}, {b}]: ", p)
```

```
## C1 [18.0, 34.0]:  15.40849454 + 1.01980481 x + 0.14225964 x**2 - 0.00263444 x**3
## C2 [34.0, 42.0]:  -278.15973851 + 26.9228842 x - 0.61959564 x**2 + 0.00483473 x**3
## C3 [42.0, 51.0]:  200.0690368 - 7.23631404 x + 0.19371861 x**2 - 0.00162014 x**3
## C4 [51.0, 80.0]:  -97.42871854 + 10.26355392 x - 0.14941606 x**2 + 0.00062257 x**3
```

Everything looks correct.

Cubic Spline Regression

In section 7.4.3 of the book we read that we can model a cubic spline of K knots as as:

$$y_i = \beta_0 + \beta_1 b_1(x_i) + \beta_2 b_2(x_i) + \cdots + \beta_{K+3} b_{K+3}(x_i)$$

This basis representation means just seven coefficients for our 3-knot cubic spline. Let's try it out.

Per the text our fit will need an intercept term, X, X^2, X^3, and one $h(X, \xi)$ per knot where $h()$ is a function of our `X` data x and knots ξ (the lowercase Greek letter Xi) defined as:

$$h(x, \xi) = (x - \xi)_+^3 = \begin{cases} (x - \xi)^3 & if\ x > \xi \\ 0 & otherwise \end{cases} \tag{1}$$

That's easy enough to code up. We'll name it `h()` to match the book.

```
def h(x, knots):
    h = x[:, None] - knots[None, :]
    return np.where(h > 0, h**3, 0)
```

The `x[:, None] - knots[None, :]` part broadcasts to `len(x)` rows by `len(knots)` columns. Now when we go to build our matrix A we can stack directly as follows.

```
A = np.column_stack((np.ones(len(X)),
                     X,
                     X**2,
                     X**3,
                     h(X, knot_x)))
b = Y
```

Fitting is once again a one-liner.

```
w, RSS, rank, S = np.linalg.lstsq(A, b, rcond=None)
```

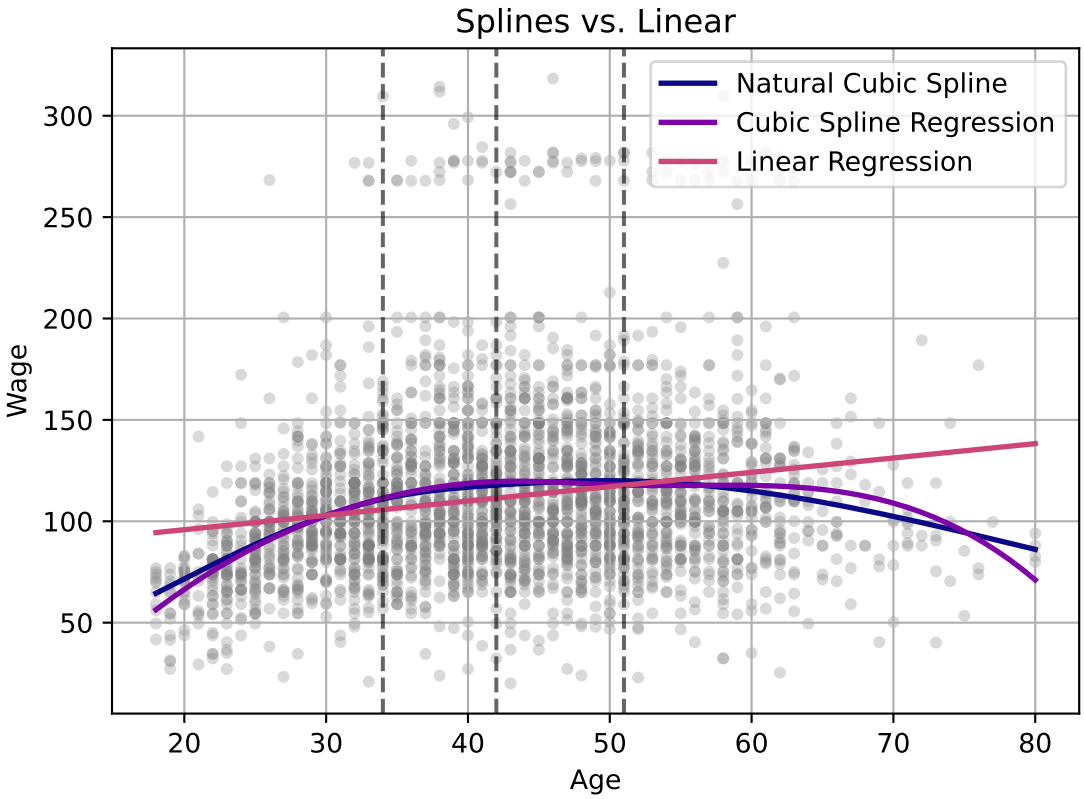
Let's compare.

```
# add Cubic Spline Regression to comparison
Y_hat_cr = A@w

# collect RSS along same segments as Nat. Cubic Spline and Linear Regression
RSS_cr = []
for i in range(len(points_x)-1):
    if i == len(points_x)-2:
        indices = (X >= points_x[i]) & (X <= points_x[i+1])
    else:
        indices = (X >= points_x[i]) & (X < points_x[i+1])
    RSS_cr.append((Y[indices] - Y_hat_cr[indices])**2)

# assemble lines to print all at once
lines = ['Training RSS Comparison\n',
         f'Linear Regression       = {sum(np.sum(r) for r in RSS_lr).round(1)}\n',
         f'Natural Cubic Spline    = {sum(np.sum(r) for r in RSS_cs).round(1)}\n',
         f'Cubic Spline Regression = {sum(np.sum(r) for r in RSS_cr).round(1)}\n',
         'By segment:\n',
         f" {'Seg':>3}{'L. Regression':>17}",
         f"{'Nat. Cubic Spline (K=3)':>28}{'Cubic Spline Regr. (K=3)':>27}\n"]

for i in range(len(RSS_lr)):
    lines.append(f" {i+1:>3}")
    lines.append(f"{np.sum(RSS_lr[i]).round(1):>17}")
    lines.append(f"{np.sum(RSS_cs[i]).round(1):>27}")
    lines.append(f"{np.sum(RSS_cr[i]).round(1):>28}\n")

print(''.join(lines))
```

```
## Training RSS Comparison
## Linear Regression       = 5022216.1
## Natural Cubic Spline    = 4779112.9
## Cubic Spline Regression = 4766247.3
## By segment:
##  Seg    L. Regression     Nat. Cubic Spline (K=3)   Cubic Spline Regr. (K=3)
##    1         710639.6                   599797.2                    597028.6
##    2        1301424.8                  1258069.8                   1256336.3
##    3        1521836.8                  1497180.5                   1492825.0
##    4        1488314.9                  1424065.3                   1420057.4
```

The full cubic spline regression using all 3,000 `X` and `Y` was just slightly better than the natural cubic spline fit on just the five `X` and mean `Y`. Let's overlay the three. With three functions to plot we'll forgo coloring each segment to and instead just indicate the boundaries between segments with dashed vertical lines. Let's also shrink those data points a bit.

```
plotting_y = np.column_stack((np.ones(len(plotting_x)),
                              plotting_x,
                              plotting_x**2,
                              plotting_x**3,
                              h(plotting_x, knot_x))) @ w

plt.figure(1)
plt.scatter(X, Y, facecolor='gray', alpha=0.3, s=12)
plt.plot(plotting_x, spline(plotting_x), label='Natural Cubic Spline', color=colors[0], lw=2)
plt.plot(plotting_x, plotting_y, label='Cubic Spline Regression', color=colors[1], lw=2)
plt.plot(plotting_x, L(plotting_x), label='Linear Regression', color=colors[2], lw=2)
for k in knot_x:
    plt.axvline(k, color="black", linestyle="--", alpha=0.6)
plt.xlabel('Age')
plt.ylabel('Wage')
plt.title('Splines vs. Linear')
plt.grid()
plt.legend()
plt.show()
```



It's interesting just how close the cubic spline regression using all 3,000 `X` and `Y` was to the natural cubic spline fit on just the five `X` and mean `Y`.