WIKIPEDIA

# De Boor's algorithm

In the mathematical subfield of numerical analysis **de Boor's algorithm**[1] is a fast and numerically stable algorithm for evaluating spline curves in B-spline form. It is a generalization of de Casteljau's algorithm for Bézier curves. The algorithm was devised by Carl R. de Boor. Simplified, potentially faster variants of the de Boor algorithm have been created but they suffer from comparatively lower stability.[2][3]

## Contents



## Introduction

A general introduction to B-splines is given in the main article. Here we discuss de Boor's algorithm, an efficient and numerically stable scheme to evaluate a spline curve $\mathbf{S}(x)$ at position $x$. The curve is built from a sum of B-spline functions $B_{i,p}(x)$ multiplied with potentially vector-valued constants $\mathbf{c}_i$, called control points,

$$\mathbf{S}(x) = \sum_i \mathbf{c}_i B_{i,p}(x).$$

B-splines of order $p+1$ are connected piece-wise polynomial functions of degree $p$ defined over a grid of knots $t_0, \ldots, t_i, \ldots, t_m$ (we always use zero-based indices in the following). De Boor's algorithm uses O($p^2$) + O(p) operations to evaluate the spline curve. Note: the main article about B-splines and the classic publications[1] use a different notation: the B-spline is indexed as $B_{i,n}(x)$ with $n = p+1$.

## Local support

B-splines have local support, meaning that the polynomials are positive only in a finite domain and zero elsewhere. The Cox-de Boor recursion formula[4] shows this:

$$B_{i,0}(x) := \begin{cases} 1 & \text{if} \quad t_i \le x < t_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$B_{i,p}(x) := \frac{x - t_i}{t_{i+p} - t_i} B_{i,p-1}(x) + \frac{t_{i+p+1} - x}{t_{i+p+1} - t_{i+1}} B_{i+1,p-1}(x).$$

Let the index $k$ define the knot interval that contains the position, $x \in [t_k, t_{k+1})$. We can see in the recursion formula that only B-splines with $i = k-p, \ldots, k$ are non-zero for this knot interval. Thus, the sum is reduced to:

$$\mathbf{S}(x) = \sum_{i=k-p}^{k} \mathbf{c}_i B_{i,p}(x).$$

It follows from $i \geq 0$ that $k \geq p$. Similarly, we see in the recursion that the highest queried knot location is at index $k+1+p$. This means that any knot interval $[t_k, t_{k+1})$ which is actually used must have at least $p$ additional knots before and after. In a computer program, this is typically achieved by repeating the first and last used knot location $p$ times. For example, for $p = 3$ and real knot locations $(0, 1, 2)$, one would pad the knot vector to $(0, 0, 0, 0, 1, 2, 2, 2, 2)$.

# The algorithm

With these definitions, we can now describe de Boor's algorithm. The algorithm does not compute the B-spline functions $B_{i,p}(x)$ directly. Instead it evaluates $\mathbf{S}(x)$ through an equivalent recursion formula.

Let $\mathbf{d}_{i,r}$ be new control points with $\mathbf{d}_{i,0} := \mathbf{c}_i$ for $i = k-p, \ldots, k$. For $r = 1, \ldots, p$ the following recursion is applied:

$$\mathbf{d}_{i,r} = (1 - \alpha_{i,r})\mathbf{d}_{i-1,r-1} + \alpha_{i,r}\mathbf{d}_{i,r-1}; \quad i = k-p+r, \ldots, k$$

$$\alpha_{i,r} = \frac{x - t_i}{t_{i+1+p-r} - t_i}.$$

Once the iterations are complete, we have $\mathbf{S}(x) = \mathbf{d}_{k,p}$, meaning that $\mathbf{d}_{k,p}$ is the desired result.

De Boor's algorithm is more efficient than an explicit calculation of B-splines $B_{i,p}(x)$ with the Cox-de Boor recursion formula, because it does not compute terms which are guaranteed to be multiplied by zero.

# Optimizations

The algorithm above is not optimized for the implementation in a computer. It requires memory for $(p+1) + p + \cdots + 1 = (p+1)(p+2)/2$ temporary control points $\mathbf{d}_{i,r}$. Each temporary control points is written exactly once and read twice. By reversing the iteration over $i$ (counting down instead of up), we can run the algorithm with memory for only $p+1$ temporary control points, by letting $\mathbf{d}_{i,r}$ reuse the memory for $\mathbf{d}_{i,r-1}$. Similarly, there is only one value of $\alpha$ used in each step, so we can reuse the memory as well.

Furthermore, it is more convenient to use a zero-based index $j = 0, \ldots, p$ for the temporary control points. The relation to the previous index is $i = j + k - p$. Thus we obtain the improved algorithm:

Let $\mathbf{d}_j := \mathbf{c}_{j+k-p}$ for $j = 0, \ldots, p$. Iterate for $r = 1, \ldots, p$:

$$\mathbf{d}_j := (1 - \alpha_j)\mathbf{d}_{j-1} + \alpha_j \mathbf{d}_j; \quad j = p, \ldots, r \quad (j \text{ must be counted down})$$

$$\alpha_j := \frac{x - t_{j+k-p}}{t_{j+1+k-r} - t_{j+k-p}}.$$

After the iterations are complete, the result is $\mathbf{S}(x) = \mathbf{d}_p$.

# Example implementation

The following code in the Python programming language is a naive implementation of the optimized algorithm.

```
def deBoor(k, x, t, c, p):
    """
    Evaluates S(x).

    Args
    ----
    k: index of knot interval that contains x
    x: position
    t: array of knot positions, needs to be padded as described above
    c: array of control points
    p: degree of B-spline
    """
    d = [c[j + k - p] for j in range(0, p+1)]
    for r in range(1, p+1):
        for j in range(p, r-1, -1):
            alpha = (x - t[j+k-p]) / (t[j+1+k-r] - t[j+k-p])
            d[j] = (1.0 - alpha) * d[j-1] + alpha * d[j]
    return d[p]
```

# See also

- De Casteljau's algorithm
- Bézier curve
- NURBS

# External links

- De Boor's Algorithm (http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/de-Boor.html)
- The DeBoor-Cox Calculation (http://www.idav.ucdavis.edu/education/CAGDNotes/Deboor-Cox-Calculation/Deboor-Cox-Calculation.html)

# Computer code

- PPPACK (http://www.netlib.org/pppack/): contains many spline algorithms in Fortran
- GNU Scientific Library (https://www.gnu.org/software/gsl/): C-library, contains a sub-library for splines ported from PPPACK
- SciPy (https://www.scipy.org/): Python-library, contains a sub-library *scipy.interpolate* with spline functions based on FITPACK
- TinySpline (https://github.com/msteinbeck/tinyspline): C-library for splines with a C++ wrapper and bindings for C#, Java, Lua, PHP, Python, and Ruby
- Einspline (http://einspline.sourceforge.net/): C-library for splines in 1, 2, and 3 dimensions with Fortran wrappers

# References

1. C. de Boor [1971], "Subroutine package for calculating with B-splines", Techn.Rep. LA-4728-MS, Los Alamos Sci.Lab, Los Alamos NM; p. 109, 121.
2. Lee, E. T. Y. (December 1982). "A Simplified B-Spline Computation Routine". *Computing*. Springer-Verlag. **29** (4): 365–371. doi:10.1007/BF02246763 (https://doi.org/10.1007/BF02246763).

3. Lee, E. T. Y. (1986). "Comments on some B-spline algorithms". *Computing*. Springer-Verlag. **36** (3): 229–238. doi:10.1007/BF02240069 (https://doi.org/10.1007/BF02240069).
4. C. de Boor, p. 90

**Works cited**


- Carl de Boor (2003). *A Practical Guide to Splines, Revised Edition*. Springer-Verlag. ISBN 0-387-95366-3.


---