# `Interpolations.jl` - mathematical background

Tomas Lycken
tomas.lycken@gmail.com
github.com/tlycken

July 22, 2015

**Abstract**

This document intends to outline the mathematical foundations of the `Interpolations.jl` package, in order to make it easier to understand how the library works, as well as to change or extend its behavior. Since it shows off the derivations that underlie the code, it is also intended to help figuring out if a bug stems from coding or mathematical mistakes.

# Contents

# 1 Introduction

Interpolations.jl uses *B-splines*[1], or *basis splines*, to interpolate a data set with a specified degree. In short, the idea is to use a set of basis functions with limited support, and in each interval between two data points, construct a linear combination of the basis functions which have support in that interval, each yielding a piece in a piecewise defined function with support across the entire data set. B-splines are defined to have the maximum degree of continuity, with minimal support. Specific details on the general mathematics of such interpolations have been extensively presented by Thevenaz, Blu, and Unser [1].

Before going into specific implementation details, we must define a few core concepts.

## 1.1 Interpolation degree

The interpolation degree decides what continuity properties the interpolant (i.e. the object which represents the interpolated data set) has; for example, a linear interpolation is a piecewise linear function, which is continuous but has discontinuous derivatives (or gradients, in higher dimensions).

Because of how B-splines are defined, the interpolation degree also decides the *support* of the *spline functions*. For a linear interpolation, the function value is decided by the value at *two* data points, so its support is two[2].

## 1.2 Grid cells

When looking at a discrete grid, one can consider the data points to define either cell boundaries (with one grid cell spanning the domain between two data points) or cell centers (with one grid cell spanning the domain closest to a single data point). On uniform, 1-spaced grids, this corresponds to having cell boundaries either at integers $1, 2, \ldots$ (where the data points are specified), or at half-integers $\frac{1}{2}, \frac{3}{2}, \ldots$ (exactly centered between data points).

Each interpolation type defines a *grid behavior*, specified by `OnGrid` or `OnCell`. However, interpolations of even (odd) degree will use an odd (even) number of points, and we require them to be symmetric. Thus, even (odd) degree interpolations will use `OnCell` (`OnGrid`) behavior inside the domain, regardless of the `OnCell` or `OnGrid` specification in the interpolation type; `OnCell`/`OnGrid` will be used solely to determine where the edges of the domain lie.

[1] `http://en.wikipedia.org/wiki/B-spline#Definition`

[2] For this reason, linear B-splines are often referred to as *2nd order*, which may be a source of confusion since the interpolating function itself is linear, i.e. of first order. In `Interpolations.jl`, we will try to avoid this confusion by referring to interpolation degree by "linear", "quadratic" etc.

### 1.3 Boundary conditions

For higher interpolation degrees (specifically, from quadratic interpolation and up), the support of the B-splines is too large for the interpolating scheme to be able to figure out the values near the edges of the data sets. In order to close the equation systems at the edges, boundary conditions are used.

### 1.4 Extrapolation behavior

Somewhat orthogonal[3] to the concepts outlined above, is the concept of *extrapolation*, i.e. evaluation of the interpolant outside the domain defined by the data set. For some types of extrapolation, this behavior is defined by a translation of the interpolation coordinate to somewhere inside the domain (e.g. periodic or reflecting boundaries), while for other types it entails a separate calculation entirely (e.g. linear extrapolation).

## 2 Constant B-splines

A constant "polynomial" has the form $p(x) = k$ for some constant $k$, and so requires only one parameter. A constant B-spline, therefore, has support 1. The intuitive way of understanding a constant B-spline is as a *nearest neighbor* interpolation.

Constant B-splines are *interpolating*, using the terminology of Thevenaz, Blu, and Unser [1], so there is no need to set up and solve an equation system for the interpolation coefficients; `Interpolations.jl` simply returns `A[round(Int,x)]` for constant interpolation of the vector `A`[4].

## 3 Linear B-splines

Linear functions require two coefficients - the slope and the elevation; thus, linear B-splines have support 2. Linear B-splines are interpolating just like constant B-splines, so the data points can be used as interpolation coefficients without any pre-filtering. Linear B-splines are implemented in `Interpolations.jl` simply as a piecewise linear interpolation between the two nearest data points.

[3] Although the implementational separation of extrapolation behavior is computationally sound, it does allow for some interesting, yet probably nonsensical, combinations of interpolation degrees, boundary conditions and extrapolation behavior. In cases where the extrapolation behavior is defined as constant or reflecting, it will make sense to specify matching boundary conditions, but other combinations are entirely supported by `Interpolations.jl`. No guarantees are left that the results make sense, though...

[4] Mathematically, there are a few different ways that points that are exactly between two data points are treated, depending on the source; in the case of `Interpolations.jl`, since we are dealing with floating point operations anyway, we use the result of `round(Int, x)` as the nearest neighbor index without worrying about the details.

When `OnCell` behavior is specified, the outermost linear pieces are extended by a half-cell, effectively using linear extrapolation based on the two outermost data points, up to $\frac{1}{2}$ outside the provided data.

# 4 Quadratic B-splines

## 4.1 Cell definition

A quadratic polynomial requires three coefficients, and thus quadratic B-splines are *three-point*, i.e. the interpolating spline function in one grid cell takes three provided data points into account. It is natural to center the grid cells around the middle point, so that the spline function around $x_i$ is is defined on the interval $\delta x \in [-\frac{1}{2}, \frac{1}{2}]$, where $\delta x = x - x_i$ and $i$ ranges from 1 to the number of data points $N$.

## 4.2 General governing equations

Assuming a symmetric spline basis, one finds that

$$y_i(\delta x) = c_{i-1}\left(\tfrac{1}{2} - \delta x\right)^2 + c_i\left(\tfrac{3}{4} - \delta x^2\right) + c_{i+1}\left(\tfrac{1}{2} + \delta x\right)^2 \tag{1}$$

Furthermore, we require that on cell boundaries, the function values and first derivatives are continuous, which yields $y_i(\frac{1}{2}) = y_{i+1}(-\frac{1}{2})$ and $y_i'(\frac{1}{2}) = y_{i+1}'(-\frac{1}{2})$. Inserting these in each-other, we find the following equation for each data point:

$$\frac{1}{8}c_{i-1} + \frac{3}{4}c_i + \frac{1}{8}c_{i+1} = v_i \tag{2}$$

where $v_i$ is the provided data point at $x_i$. Considering the corresponding equation for all provided data points simultanesously, we form a tridiagonal equation system that we need to solve for the coefficients $c_i$. However, since we have two more unknowns than we have data points (namely $c_0$ and $c_{N+1}$), we need to apply boundary conditions to close the system.

## 4.3 Boundary conditions

The boundary conditions are implemented by specifying further conditions on the function value and/or its derivatives at the outer boundary (or, in some cases, close to it). For simplicity, we will here only describe the lower boundary close to $x = 1$, yielding equations for $c_0$ - the same arguments are applied at the upper end of the interval, specifying $c_{N+1}$.

We will have use for the following expressions for the first and second derivatives of the interpolating function:

$$y_i'(\delta x) = \frac{1}{2}\left[c_{i-1}(2\delta x - 1) - c_i \cdot 4\delta x + c_{i+1}(2\delta x + 1)\right]$$
$$y_i''(\delta x) = c_{i-1} - 2c_i + c_{i+1}$$

which were obtained simply by derivation of equation (1) with respect to $\delta x$.

### 4.3.1 Flat boundary conditions

`Flat` boundary conditions are implemented by imposing that $y' = 0$ at the outermost cell boundary.

#### 4.3.1.1 `Quadratic{Flat,OnGrid}`

In this case, the edge is at $x = 1$, corresponding to $(i, \delta x) = (1, 0)$. Inserting this in $y' = 0$ yields

$$\begin{aligned} y'(x_{\text{edge}}) = y_1'(0) &= \frac{1}{2}\left[-c_0 + c_2\right] = 0 \\ \Rightarrow -c_0 + c_2 &= 0 \end{aligned}$$

Thus, the first few rows of the equation system become

$$\begin{bmatrix} -1 & 0 & 1 & & & \\ \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & & & \\ & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & & \\ & & \ddots & \ddots & \ddots & \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} 0 \\ v_1 \\ v_2 \\ \vdots \end{bmatrix}$$

#### 4.3.1.2 `Quadratic{Flat,OnCell}`

$$\begin{aligned} y'(x_{\text{edge}}) = y_1'\left(-\frac{1}{2}\right) &= \frac{1}{2}\left[-2c_0 + 2c_1\right] = 0 \\ \Rightarrow -c_0 + c_1 &= 0 \end{aligned}$$

$$\begin{bmatrix} -1 & 1 & & & & \\ \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & & & \\ & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & & \\ & & \ddots & \ddots & \ddots & \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} 0 \\ v_1 \\ v_2 \\ \vdots \end{bmatrix}$$

### 4.3.2 Linear, or natural, boundary conditions

For linear boundary conditions (in many contexts referred to as "natural"), we want the second-order derivative at the boundary to equal zero, i.e. $y''(x_{\text{edge}}) = 0$, in order for the interpolant to become a linear function at the edge. Since $y_i''(\delta x)$ is independent of $\delta x$, the specification is the same regardless of grid representation.

#### 4.3.2.1 `Quadratic{Line,OnCell}`

#### 4.3.2.2 `Quadratic{Line,OnGrid}`

#### 4.3.2.3 `Quadratic{Natural,OnCell}`

#### 4.3.2.4 `Quadratic{Natural,OnGrid}`

$$y''(x_{\text{edge}}) = \begin{cases} y_1''(0) & \texttt{OnGrid} \\ y_1''(-\frac{1}{2}) & \texttt{OnCell} \end{cases} = c_0 - 2c_1 + c_2 = 0$$

$$\begin{bmatrix} 1 & -2 & 1 & & & \\ \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & & & \\ & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & & \\ & & \ddots & \ddots & \ddots & \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} 0 \\ v_1 \\ v_2 \\ \vdots \end{bmatrix}$$

### 4.3.3 Free boundary conditions

The "free" boundary conditions imposes no restrictions on the interpolating function or its derivatives at the edges; instead, we require a second continuous derivative at the second-to-outermost cell boundary, i.e. $y_1''(\frac{1}{2}) = y_2''(-\frac{1}{2})$ and $y_{N-1}''(\frac{1}{2}) = y_N''(-\frac{1}{2})$.

#### 4.3.3.1 `Quadratic{Free,OnCell}`

#### 4.3.3.2 `Quadratic{Free,OnGrid}`

$$\begin{aligned} y_1''(\frac{1}{2}) = y_2''(-\frac{1}{2}) &\Leftrightarrow c_0 - 2c_1 + c_2 = c_1 - 2c_2 + c_3 \\ &\Rightarrow c_0 - 3c_1 + 3c_2 - c_3 = 0 \end{aligned}$$

$$\begin{bmatrix} 1 & -3 & 3 & -1 & & \\ \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & & & \\ & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & & \\ & & \ddots & \ddots & \ddots & \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} 0 \\ v_1 \\ v_2 \\ \vdots \end{bmatrix}$$

### 4.3.4 Periodic boundary conditions

For periodic boundary conditions, we do not introduce ghost cells 0 and $N+1$ - instead, we wrap the equation coefficients. Thus, we need no extra conditions, except that when evaluating the polynomial at $i = 1$ we need to adjust the indexing to pretend that $i - 1 = N$.

For `OnCell`, periodic boundary conditions seemlessly links the lower and upper domain boundaries. For `OnGrid`, an extra grid cell is inserted "between" the upper and lower boundaries, outside of the ordinary domain, linking the last and first data points together. Thus, in order to interpolate e.g. $\sin(x)$ over one period, you should specify the data *excluding* the wrapping data point, as shown in figure 1.

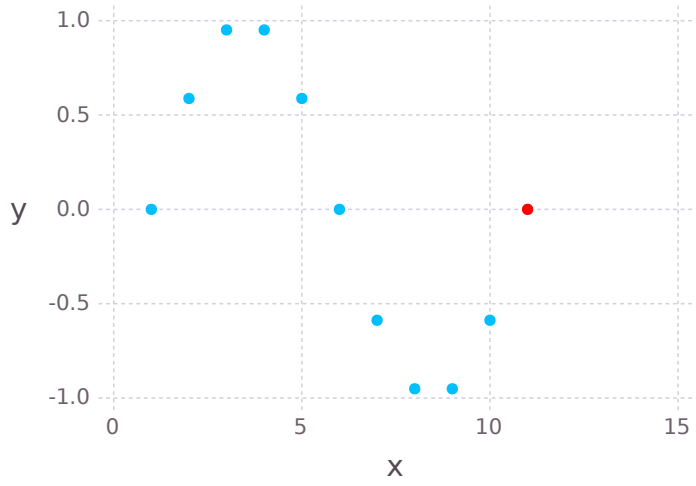


Figure 1: When specifying periodic boundary conditions with `OnGrid`, the data point on the edge should only be included *once*; the red point should *not* be included in the data.

#### 4.3.4.1 `Quadratic{Periodic,OnCell}`

#### 4.3.4.2 `Quadratic{Periodic,OnGrid}`

$$\begin{bmatrix} \frac{3}{4} & \frac{1}{8} & \cdots & & \frac{1}{8} \\ \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & & \\ & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & \\ & & \ddots & \ddots & \ddots \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ c_3 \\ \vdots \end{bmatrix}$$

Also: $y_1(\delta x) = c_N(\frac{1}{2} - \delta x)^2 + c_1(\frac{3}{4} - \delta x^2) + c_2(\frac{1}{2} + \delta x)^2$ (note the index on the first coefficient) with a similar wrapping applied at the upper boundary.

### 4.3.5 In-place boundary conditions

In some cases, speed or memory constraints make it attractive to perform interpolation without making a copy of the original array. This is achieved by calling `interpolate!` rather than `interpolate`.

For quadratic or higher interpolation, this will result in the "destruction" of the original input array, solving the tridiagonal systems in-place. While you can use any of the boundary conditions above, ones that require padding (all but `Periodic`) may result in anomalies at the boundary. Consequently, there are two additional options particularly relevant for in-place interpolation.

#### 4.3.5.1 `Quadratic{InPlace,OnCell}`

Ensures that evaluation at on-grid locations preserves the original input values. For $x$ locations smaller than $3/2$, we clamp the lowest index to 1. This implies that for evaluation of Eq. (2) at $x = 1$, we effectively have $c_0 = c_1$. The tridiagonal system can therefore be written

$$\begin{bmatrix} \frac{7}{8} & \frac{1}{8} & & & \\ \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & & \\ & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & \\ & & \ddots & \ddots & \ddots \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ c_3 \\ \vdots \end{bmatrix}$$

#### 4.3.5.2 `Quadratic{InPlaceQ,OnCell}`

This boundary condition yields the exact (to within machine precision) answer when the underlying function to be interpolated is quadratic along each axis. For example, in 2d

```
x = 1:8
y = (1:9)'
A = (x-4.3).^2 .* (y-5.7).^2
```

would be an example of such a function. Given such functions, this boundary condition is particularly useful for building test suites for algorithms that rely on interpolation: it removes any doubts about whether a disagreement in the third decimal place is due to interpolation or a bug in your code (hint: it's the latter).

These boundary conditions are specified by ensuring that the quadratic typically defined over the domain $[3/2, 5/2]$—specified from input values at $x = 1, 2$, and 3—also matches the input value at $x = 1$ when extended to the range $[1, 5/2]$. From Eq. (1), one obtains

$$\frac{9}{4}c_1 - \frac{1}{4}c_2 + \frac{1}{4}c_3 = v_1.$$

Note that the *evaluation* of interpolation function near the boundary occurs via clamping, identically to `InPlace`. *Consequently, you will get wrong answers over the domain* $[1, 3/2]$. The above formula is used only for setting the boundary conditions to ensure an exact match to the underlying quadratic at all interior ($[2, n-1]$) points.

# References


[1] P. Thevenaz, T. Blu, and M. Unser. "Interpolation revisited [medical images application]". In: *Medical Imaging, IEEE Transactions on* 19.7 (July 2000), pp. 739–758. ISSN: 0278-0062. DOI: 10.1109/42.875199. URL: http://bigwww.epfl.ch/publications/thevenaz0002.pdf.