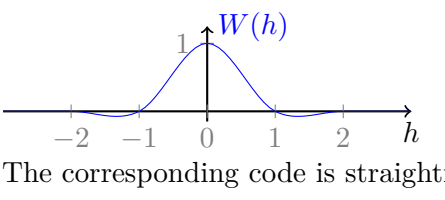
Bicubic interpolation algorithm

Christof Kaufmann

January 19, 2022

1 Inner points

Bicubic interpolation can be separated into x axis and y axis. For each axis a convolutional algorithm can be applied. The xy grid must be rectilinear, i. e. $x_{n+1} - x_n = \delta x$ and $y_{n+1} - y_n = \delta y$ must be the same for all n, but δx and δy may differ, and may even be negative. The cubic kernel function with $a = -\frac{1}{2}$ is according to Wikipedia:



$$W(h) = \begin{cases} \frac{3}{2}|h|^3 - \frac{5}{2}|h|^2 + 1 & \text{if } |h| \le 1 \\ -\frac{1}{2}|h|^3 + \frac{5}{2}|h|^2 - 4|h| + 2 & \text{if } 1 < |h| \le 2 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The corresponding code is straightforward:

```
function w = cubic(h)
  absh = abs(h);
  absh01 = absh <= 1;             % for |h| <= 1
  absh12 = absh <= 2 & ~absh01;   % for 1 < |h| <= 2
  w = (1.5 * absh.^3 - 2.5 * absh.^2 + 1)              .* absh01 + ...
      (-0.5 * absh.^3 + 2.5 * absh.^2 - 4 * absh + 2) .* absh12;
end
```

With this formula it is easy to interpolate a single location using two neighbors around:

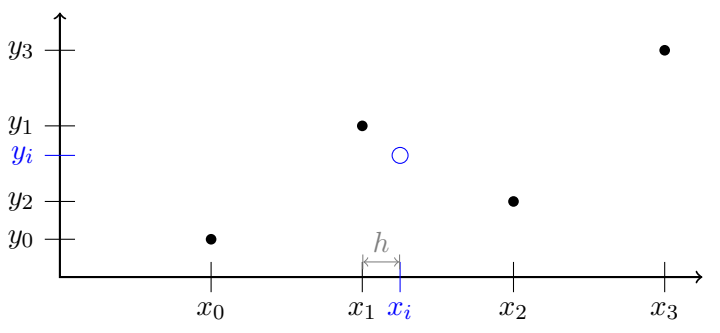
$$y_i = W(-1-h)\,y_0 + W(0-h)\,y_1 + W(1-h)\,y_2 + W(2-h)\,y_3 \tag{2}$$

Let's illustrate this with a simple 1D example:

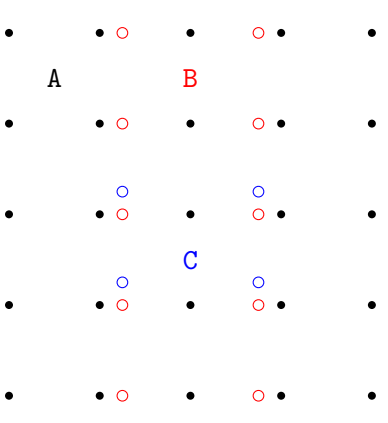
```
x = 1:4;
y = [2, 1, 0.5, 1.5];
xi = 2.25;

% for indexes startig at 1:
idx = floor(xi);
h = xi - idx;
% or more general:
idx = lookup (x, xi);
h = (xi - x(idx)) / abs(x(1) - x(2));

% as single statement:
yi = cubic(-1-h)*y(-1+idx) + cubic(0-h)*y(0+idx) ...
   + cubic(1-h)*y(1+idx) + cubic(2-h)*y(2+idx);
% or as for loop:
yi = 0;
for shift = -1:2
  yi += cubic(shift-h) * y(shift+idx);
endfor
% or vectorized:
shifts = -1:2;
yi = sum(cubic(shifts - h) .* y(shifts + idx));
```



This can be extended to 2D easily:




```
A = magic(5)
xi = [2.25, 3.75];
yi = [2.75; 3.75];

idx = floor(xi);
h = xi - idx;
B = cubic(-1-h) .* A(:, -1+idx) + cubic(0-h) .* A(:, 0+idx) ...
  + cubic(1-h) .* A(:, 1+idx) + cubic(2-h) .* A(:, 2+idx)

idx = floor(yi);
h = yi - idx;
C = cubic(-1-h) .* B(-1+idx, :) + cubic(0-h) .* B(0+idx, :) ...
  + cubic(1-h) .* B(1+idx, :) + cubic(2-h) .* B(2+idx, :);
```

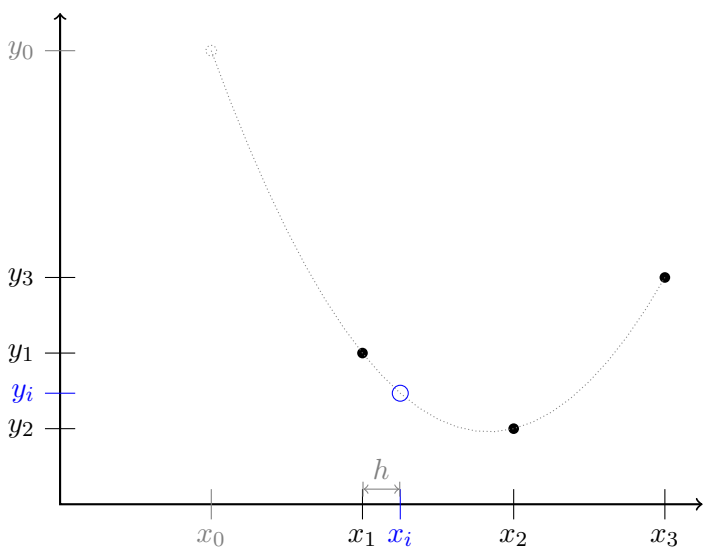
2 Border points

At the border, there are too less neighbors. To apply the algorithm, the matrix can be padded. The simplest padding method would be to add 0s, which is not done in Octave. Still, different approaches are used in different functions.

In `imresize` even `xi` less than 1 (but always greater than 0.5) and thus `idx` equal 0 can occur. Similarly, too large indexes can occur. These points lack two neighbors, but interpolation is still performed. For that, `imresize` uses symmetrical padding.

In **`interp2`** with bicubic interpolation any `xi` less than 1 or larger than the number of columns of the matrix `z` will yield a **`NaN`** or constant extrapolation value `extrap`. So there can only occur situations where one neighbor is missing. In **`interp2`** the missing neighbor is reconstructed using quadratical extrapolation. The cubic interpolation will then degenerate to a quadratic interpolation. The quadratic Lagrange interpolation formula is:

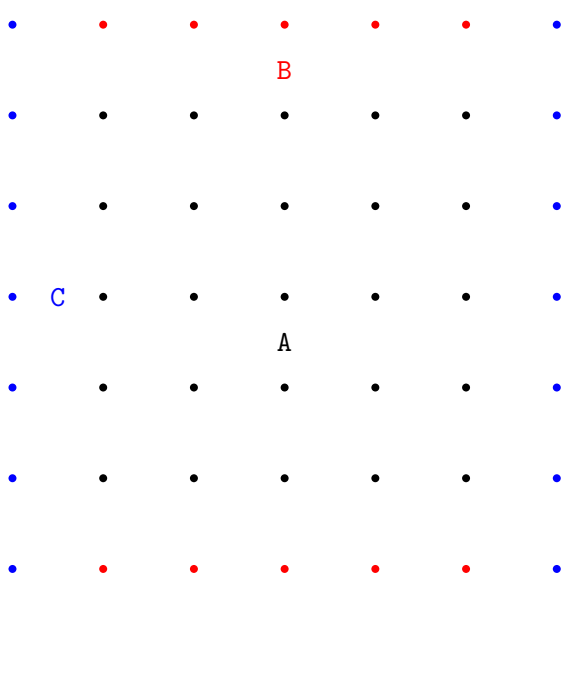
$$q(x) = \frac{(x-x_2)\,(x-x_3)}{(x_1-x_2)\,(x_1-x_3)}\,y_1 + \frac{(x-x_1)\,(x-x_3)}{(x_2-x_1)\,(x_2-x_3)}\,y_2 + \frac{(x-x_1)\,(x-x_2)}{(x_3-x_1)\,(x_3-x_2)}\,y_3 \tag{3}$$



One way to interpolate is to pad with the extrapolated values $y_0 = q(x_0) = 3\,y_1 - 3\,y_2 + y_3$ and then use the cubic convolutional algorithm as usual to interpolate y_i. For a 2D array `A`, a 5×5 matrix in the following example, this could be done in two steps:

```
% Add rows at top and bottom, yields 7x5
B = [3*A(1, :) - 3*A(2, :) + A(3, :);
    A;
    3*A(end-1, :) - 3*A(end-2, :) + A(end-3, :)];

% Add columns left and right, yields 7x7
C = [3*B(:, 1) - 3*B(:, 2) + B(:, 3), ...
    B, ...
    3*B(:, end-1) - 3*B(:, end-2) + B(:, end-3)];
```



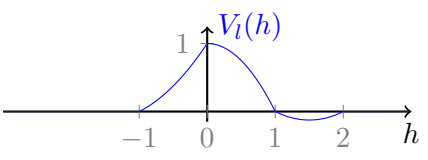
This can be combined into one step to save some memory. Even more memory could be saved by interpolating the borders separately with $N \times 4$ and $4 \times M$ padded matrices at the cost of code complexity.

When the interpolation coordinate match exactly the coordinate of a row or column, it only has to be copied. The kernel function can also handle this, since $h = 0 \Rightarrow W(h) = 1$ for these cases. However, in the special case of copying the last row or column, there are missing two neighbors again. This can be caught and handled separately or solved by padding an additional zero row and column.

The second way is to derive a quadratic convolutional kernel from (3) and apply it directly without padding. This saves memory, but increases code complexity a lot. It holds

$$\begin{aligned} y_i = q(x_i) &= \frac{1}{2}\,(h-1)\,(h-2)\,y_1 - h\,(h-2)\,y_2 + \frac{1}{2}\,h\,(h-1)\,y_3 \\ &= \underbrace{\left(\frac{1}{2}\,h^2 - 3\,h + 2\right)}_{(I)} y_1 + \underbrace{\left(-h^2 + 2\,h\right)}_{(II)} y_2 + \underbrace{\left(\frac{1}{2}\,h^2 - \frac{1}{2}\,h\right)}_{(III)} y_3. \end{aligned}$$

We want to use (I) with $-h$ on $[-1, 0)$, (II) with $1-h$ on $[0, 1)$ and (III) with $2-h$ on $[1, 2)$. Therefore the quadratic kernel function is

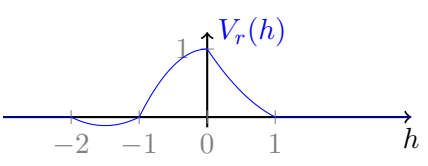


$$V_l(h) = \begin{cases} \frac{1}{2}\,h^2 + \frac{3}{2}\,h + 1 & \text{if } -1 \le h < 0 \\ -h^2 + 1 & \text{if } 0 \le h < 1 \\ \frac{1}{2}\,h^2 - \frac{3}{2}\,h + 1 & \text{if } 1 \le h < 2 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

For the right border it holds $h = x_i - x_2$, since x_i is between x_2 and x_3. Hence, we need a slightly different kernel function.

$$y_i = q(x_i) = \underbrace{\left(\frac{1}{2}\,h^2 + \frac{1}{2}\,h\right)}_{(I)} y_1 + \underbrace{\left(-h^2 + 1\right)}_{(II)} y_2 + \underbrace{\left(\frac{1}{2}\,h^2 - \frac{1}{2}\,h\right)}_{(III)} y_3.$$

We want to use (I) with $-1-h$ on $[-2, -1)$, (II) with $0-h$ on $[-1, 0)$ and (III) with $1-h$ on $[0, 1)$. Therefore the quadratic kernel function for the right border is



$$V_r(h) = \begin{cases} \frac{1}{2}\,h^2 + \frac{3}{2}\,h + 1 & \text{if } -2 \le h < -1 \\ -h^2 + 1 & \text{if } -1 \le h < 0 \\ \frac{1}{2}\,h^2 - \frac{3}{2}\,h + 1 & \text{if } 0 \le h < 1 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$