Paulinternet.nl
Home
GTA SA Savegame Editor
Cubic interpolation
Contact

Cubic interpolation

On this page you can find explanation about (*n*-)cubic interpolation and implementations in Java and C++.
Anything at this page may be copied and modified.
Please contact me if you find an error.

Cubic interpolation

If the values of a function *f*(x) and its derivative are known at x=0 and x=1, then the function can be interpolated on the interval [0,1] using a third degree polynomial. This is called cubic interpolation. The formula of this polynomial can be easily derived.

A third degree polynomial and its derivative:

$$f(x) = ax^3 + bx^2 + cx + d$$
$$f'(x) = 3ax^2 + 2bx + c$$

The values of the polynomial and its derivative at x=0 and x=1:

$$f(0) = d$$
$$f(1) = a + b + c + d$$
$$f'(0) = c$$
$$f'(1) = 3a + 2b + c$$

The four equations above can be rewritten to this:

$$a = 2f(0) - 2f(1) + f'(0) + f'(1)$$
$$b = -3f(0) + 3f(1) - 2f'(0) - f'(1)$$
$$c = f'(0)$$
$$d = f(0)$$

And there we have our cubic interpolation formula.

Interpolation is often used to interpolate between a list of values. In that case we don't know the derivative of the function. We could simply use derivative 0 at every point, but we obtain smoother curves when we use the slope of a line between the previous and the next point as the derivative at a point. In that case the resulting polynomial is called a Catmull-Rom spline. Suppose you have the values p_0, p_1, p_2 and p_3 at respectively x=-1, x=0, x=1, and x=2. Then we can assign the values of *f*(0), *f*(1), *f'*(0) and *f'*(1) using the formulas below to interpolate between p_1 and p_2.

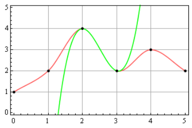
$$f(0) = p_1$$
$$f(1) = p_2$$
$$f'(0) = \frac{p_2 - p_0}{2}$$
$$f'(1) = \frac{p_3 - p_1}{2}$$

Combining the last four formulas and the preceding four, we get:

$$a = -\tfrac{1}{2}p_0 + \tfrac{3}{2}p_1 - \tfrac{3}{2}p_2 + \tfrac{1}{2}p_3$$
$$b = p_0 - \tfrac{5}{2}p_1 + 2p_2 - \tfrac{1}{2}p_3$$
$$c = -\tfrac{1}{2}p_0 + \tfrac{1}{2}p_2$$
$$d = p_1$$

So our cubic interpolation formula becomes:

$$f(p_0,p_1,p_2,p_3,x) = (-\tfrac{1}{2}p_0+\tfrac{3}{2}p_1-\tfrac{3}{2}p_2+\tfrac{1}{2}p_3)x^3+(p_0-\tfrac{5}{2}p_1+2p_2-\tfrac{1}{2}p_3)x^2+(-\tfrac{1}{2}p_0+\tfrac{1}{2}p_2)x+p_1$$



For the green curve:

$$a = -\tfrac{1}{2}\cdot 2 + \tfrac{3}{2}\cdot 4 - \tfrac{3}{2}\cdot 2 + \tfrac{1}{2}\cdot 3 = \tfrac{7}{2}$$
$$b = 2 - \tfrac{5}{2}\cdot 4 + 2\cdot 2 - \tfrac{1}{2}\cdot 3 = -\tfrac{11}{2}$$
$$c = -\tfrac{1}{2}\cdot 2 + \tfrac{1}{2}\cdot 2 = 0$$
$$d = 4$$
$$f(x) = \tfrac{7}{2}(x-2)^3 - \tfrac{11}{2}(x-2)^2 + 4$$

The first and the last interval

We used the two points left of the interval and the two points right of the inverval as inputs for the interpolation function. But what if we want to interpolate between the first two or last two elements of a list? Then we have no p_0 or no p_3. The solution is to imagine an extra point at each end of the list. In other words, we have to make up a value for p_0 and p_3 when interpolating the leftmost and rightmost interval respectively. Two ways to do this are:

- Repeat the first and the last point.
  Left: $p_0 = p_1$
  Right: $p_3 = p_2$
- Let the end point be in the middle of a line between the imaginary point and the point next to the end point.
  Left: $p_0 = 2p_1 - p_2$
  Right: $p_3 = 2p_2 - p_1$

Bicubic interpolation

Bicubic interpolation is cubic interpolation in two dimensions. I'll only consider the case where we want to interpolate a two dimensional grid. We can use the cubic interpolation formula to construct the bicubic interpolation formula.

Suppose we have the 16 points p_{ij}, with i and j going from 0 to 3 and with p_{ij} located at (i-1, j-1). Then we can interpolate the area [0,1] x [0,1] by first interpolating the four columns and then interpolating the results in the horizontal direction. The formula becomes:

$$g(x,y) = f(f(p_{00},p_{01},p_{02},p_{03},y), f(p_{10},p_{11},p_{12},p_{13},y), f(p_{20},p_{21},p_{22},p_{23},y), f(p_{30},p_{31},p_{32},p_{33},y), x)$$

Bicubic interpolation can be used to resize images. However, this is (currently) out of the scope of this article. Please don't ask me questions about that.

Java implementation

```java
public class CubicInterpolator
{
	public static double getValue (double[] p, double x) {
		return p[1] + 0.5 * x*(p[2] - p[0] + x*(2.0*p[0] - 5.0*p[1] + 4.0*p[2] - p[3] + x*(3.0*(p[1] - p[2]) + p[3] - p[0])));
	}
}

public class BicubicInterpolator extends CubicInterpolator
{
	private double[] arr = new double[4];

	public double getValue (double[][] p, double x, double y) {
		arr[0] = getValue(p[0], y);
		arr[1] = getValue(p[1], y);
		arr[2] = getValue(p[2], y);
		arr[3] = getValue(p[3], y);
		return getValue(arr, x);
	}
}

public class TricubicInterpolator extends BicubicInterpolator
{
	private double[] arr = new double[4];

	public double getValue (double[][][] p, double x, double y, double z) {
		arr[0] = getValue(p[0], y, z);
		arr[1] = getValue(p[1], y, z);
		arr[2] = getValue(p[2], y, z);
		arr[3] = getValue(p[3], y, z);
		return getValue(arr, x);
	}
}
```

C++ implementation

```cpp
#include <iostream>
#include <assert.h>

double cubicInterpolate (double p[4], double x) {
	return p[1] + 0.5 * x*(p[2] - p[0] + x*(2.0*p[0] - 5.0*p[1] + 4.0*p[2] - p[3] + x*(3.0*(p[1] - p[2]) + p[3] - p[0])));
}

double bicubicInterpolate (double p[4][4], double x, double y) {
	double arr[4];
	arr[0] = cubicInterpolate(p[0], y);
	arr[1] = cubicInterpolate(p[1], y);
	arr[2] = cubicInterpolate(p[2], y);
	arr[3] = cubicInterpolate(p[3], y);
	return cubicInterpolate(arr, x);
}

double tricubicInterpolate (double p[4][4][4], double x, double y, double z) {
	double arr[4];
	arr[0] = bicubicInterpolate(p[0], y, z);
	arr[1] = bicubicInterpolate(p[1], y, z);
	arr[2] = bicubicInterpolate(p[2], y, z);
	arr[3] = bicubicInterpolate(p[3], y, z);
	return cubicInterpolate(arr, x);
}

double nCubicInterpolate (int n, double* p, double coordinates[]) {
	assert(n > 0);
	if (n == 1) {
		return cubicInterpolate(p, *coordinates);
	}
	else {
		double arr[4];
		int skip = 1 << (n - 1) * 2;
		arr[0] = nCubicInterpolate(n - 1, p, coordinates + 1);
		arr[1] = nCubicInterpolate(n - 1, p + skip, coordinates + 1);
		arr[2] = nCubicInterpolate(n - 1, p + 2*skip, coordinates + 1);
		arr[3] = nCubicInterpolate(n - 1, p + 3*skip, coordinates + 1);
		return cubicInterpolate(arr, *coordinates);
	}
}

int main () {
	// Create array
	double p[4][4] = {{1,3,3,4}, {7,2,3,4}, {1,6,3,6}, {2,5,7,2}};

	// Interpolate
	std::cout << bicubicInterpolate(p, 0.1, 0.2) << '\n';

	// Or use the nCubicInterpolate function
	double co[2] = {0.1, 0.2};
	std::cout << nCubicInterpolate(2, (double*) p, co) << '\n';
}
```

Bicubic interpolation polynomial

This section provides an alternative way to calculate bicubic interpolation. For most purposes this way is probably less practical and efficient than the way it is done above.

We can rewrite the formula for bicubic interpolation as a multivariate polynomial:

$$g(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij}x^i y^j$$

With these values for a_{ij}, the coefficients:

$$a_{00} = p_{11}$$
$$a_{01} = -\tfrac{1}{2}p_{10} + \tfrac{1}{2}p_{12}$$
$$a_{02} = p_{10} - \tfrac{5}{2}p_{11} + 2p_{12} - \tfrac{1}{2}p_{13}$$
$$a_{03} = -\tfrac{1}{2}p_{10} + \tfrac{3}{2}p_{11} - \tfrac{3}{2}p_{12} + \tfrac{1}{2}p_{13}$$
$$a_{10} = -\tfrac{1}{2}p_{01} + \tfrac{1}{2}p_{21}$$
$$a_{11} = \tfrac{1}{4}p_{00} - \tfrac{1}{4}p_{02} - \tfrac{1}{4}p_{20} + \tfrac{1}{4}p_{22}$$
$$a_{12} = -\tfrac{1}{2}p_{00} + \tfrac{5}{4}p_{01} - p_{02} + \tfrac{1}{4}p_{03} + \tfrac{1}{2}p_{20} - \tfrac{5}{4}p_{21} + p_{22} - \tfrac{1}{4}p_{23}$$
$$a_{13} = \tfrac{1}{4}p_{00} - \tfrac{3}{4}p_{01} + \tfrac{3}{4}p_{02} - \tfrac{1}{4}p_{03} - \tfrac{1}{4}p_{20} + \tfrac{3}{4}p_{21} - \tfrac{3}{4}p_{22} + \tfrac{1}{4}p_{23}$$
$$a_{20} = p_{01} - \tfrac{5}{2}p_{11} + 2p_{21} - \tfrac{1}{2}p_{31}$$
$$a_{21} = -\tfrac{1}{2}p_{00} + \tfrac{1}{2}p_{02} + \tfrac{5}{4}p_{10} - \tfrac{5}{4}p_{12} - p_{20} + p_{22} + \tfrac{1}{4}p_{30} - \tfrac{1}{4}p_{32}$$
$$a_{22} = p_{00} - \tfrac{5}{2}p_{01} + 2p_{02} - \tfrac{1}{2}p_{03} - \tfrac{5}{2}p_{10} + \tfrac{25}{4}p_{11} - 5p_{12} + \tfrac{5}{4}p_{13} + 2p_{20} - 5p_{21} + 4p_{22} - p_{23} - \tfrac{1}{2}p_{30} + \tfrac{5}{4}p_{31} - p_{32} + \tfrac{1}{4}p_{33}$$

$a_{23} = -\frac{1}{2}p_{00}+\frac{3}{2}p_{01}-\frac{3}{2}p_{02}+\frac{1}{2}p_{03}+\frac{5}{4}p_{10}-\frac{15}{4}p_{11}+\frac{15}{4}p_{12}-\frac{5}{4}p_{13}-p_{20}+3p_{21}-3p_{22}+p_{23}+\frac{1}{4}p_{30}-\frac{3}{4}p_{31}+\frac{3}{4}p_{32}-\frac{1}{4}p_{33}$

$a_{30} = -\frac{1}{2}p_{01} + \frac{3}{2}p_{11} - \frac{3}{2}p_{21} + \frac{1}{2}p_{31}$

$a_{31} = \frac{1}{4}p_{00} - \frac{1}{4}p_{02} - \frac{3}{4}p_{10} + \frac{3}{4}p_{12} + \frac{3}{4}p_{20} - \frac{3}{4}p_{22} - \frac{1}{4}p_{30} + \frac{1}{4}p_{32}$

$a_{32} = -\frac{1}{2}p_{00}+\frac{5}{4}p_{01}-p_{02}+\frac{1}{4}p_{03}+\frac{3}{2}p_{10}-\frac{15}{4}p_{11}+3p_{12}-\frac{3}{4}p_{13}-\frac{3}{2}p_{20}+\frac{15}{4}p_{21}-3p_{22}+\frac{3}{4}p_{23}+\frac{1}{2}p_{30}-\frac{5}{4}p_{31}+p_{32}-\frac{1}{4}p_{33}$

$a_{33} = \frac{1}{4}p_{00}-\frac{3}{4}p_{01}+\frac{3}{4}p_{02}-\frac{1}{4}p_{03}-\frac{3}{4}p_{10}+\frac{9}{4}p_{11}-\frac{9}{4}p_{12}+\frac{3}{4}p_{13}+\frac{3}{4}p_{20}-\frac{9}{4}p_{21}+\frac{9}{4}p_{22}-\frac{3}{4}p_{23}-\frac{1}{4}p_{30}+\frac{3}{4}p_{31}-\frac{3}{4}p_{32}+\frac{1}{4}p_{33}$

In Java code we can write this as:

```java
public class CachedBicubicInterpolator
{
	private double a00, a01, a02, a03;
	private double a10, a11, a12, a13;
	private double a20, a21, a22, a23;
	private double a30, a31, a32, a33;

	public void updateCoefficients (double[][] p) {
		a00 = p[1][1];
		a01 = -.5*p[1][0] + .5*p[1][2];
		a02 = p[1][0] - 2.5*p[1][1] + 2*p[1][2] - .5*p[1][3];
		a03 = -.5*p[1][0] + 1.5*p[1][1] - 1.5*p[1][2] + .5*p[1][3];
		a10 = -.5*p[0][1] + .5*p[2][1];
		a11 = .25*p[0][0] - .25*p[0][2] - .25*p[2][0] + .25*p[2][2];
		a12 = -.5*p[0][0] + 1.25*p[0][1] - p[0][2] + .25*p[0][3] + .5*p[2][0] - 1.25*p[2][1] + p[2][2] - .25*p[2][3];
		a13 = .25*p[0][0] - .75*p[0][1] + .75*p[0][2] - .25*p[0][3] - .25*p[2][0] + .75*p[2][1] - .75*p[2][2] + .25*p[2][3];
		a20 = p[0][1] - 2.5*p[1][1] + 2*p[2][1] - .5*p[3][1];
		a21 = -.5*p[0][0] + .5*p[0][2] + 1.25*p[1][0] - 1.25*p[1][2] - p[2][0] + p[2][2] + .25*p[3][0] - .25*p[3][2];
		a22 = p[0][0] - 2.5*p[0][1] + 2*p[0][2] - .5*p[0][3] - 2.5*p[1][0] + 6.25*p[1][1] - 5*p[1][2] + 1.25*p[1][3] + 2*p[2][0] - 5*p[2][1] + 4*p[2][2] - p[2][3] - .5*p[3][0] + 1.25*p[3][1] - p[3][2] + .25*p[3][3];
		a23 = -.5*p[0][0] + 1.5*p[0][1] - 1.5*p[0][2] + .5*p[0][3] + 1.25*p[1][0] - 3.75*p[1][1] + 3.75*p[1][2] - 1.25*p[1][3] - p[2][0] + 3*p[2][1] - 3*p[2][2] + p[2][3] + .25*p[3][0] - .75*p[3][1] + .75*p[3][2] - .25*p[3][3];
		a30 = -.5*p[0][1] + 1.5*p[1][1] - 1.5*p[2][1] + .5*p[3][1];
		a31 = .25*p[0][0] - .25*p[0][2] - .75*p[1][0] + .75*p[1][2] + .75*p[2][0] - .75*p[2][2] - .25*p[3][0] + .25*p[3][2];
		a32 = -.5*p[0][0] + 1.25*p[0][1] - p[0][2] + .25*p[0][3] + 1.5*p[1][0] - 3.75*p[1][1] + 3*p[1][2] - .75*p[1][3] - 1.5*p[2][0] + 3.75*p[2][1] - 3*p[2][2] + .75*p[2][3] + .5*p[3][0] - 1.25*p[3][1] + p[3][2] - .25*p[3][3];
		a33 = .25*p[0][0] - .75*p[0][1] + .75*p[0][2] - .25*p[0][3] - .75*p[1][0] + 2.25*p[1][1] - 2.25*p[1][2] + .75*p[1][3] + .75*p[2][0] - 2.25*p[2][1] + 2.25*p[2][2] - .75*p[2][3] - .25*p[3][0] + .75*p[3][1] - .75*p[3][2] + .25*p[3][3];
	}

	public double getValue (double x, double y) {
		double x2 = x * x;
		double x3 = x2 * x;
		double y2 = y * y;
		double y3 = y2 * y;

		return (a00 + a01 * y + a02 * y2 + a03 * y3) +
		       (a10 + a11 * y + a12 * y2 + a13 * y3) * x +
		       (a20 + a21 * y + a22 * y2 + a23 * y3) * x2 +
		       (a30 + a31 * y + a32 * y2 + a33 * y3) * x3;
	}
}
```

This implementation can run faster than the implementation above when `getValue` is called multiple times for one call to `updateCoefficients`.

Similar implementations could be written for other dimensions than two.