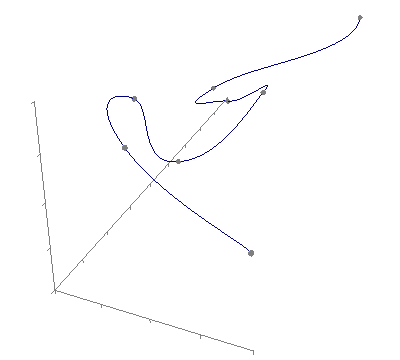
**Interpolation methods**

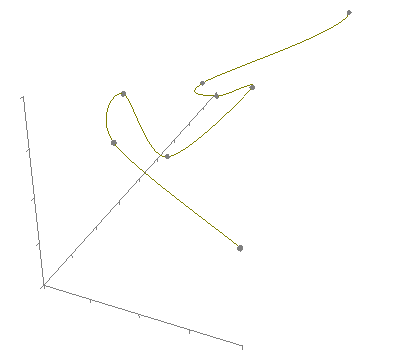
Written by [Paul Bourke](http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/)  
December 1999

Discussed here are a number of interpolation methods, this is by no means an exhaustive list but the methods shown tend to be those in common use in computer graphics. The main attributes is that they are easy to compute and are stable. Interpolation as used here is different to "smoothing", the techniques discussed here have the characteristic that the estimated curve passes through all the given points. The idea is that the points are in some sense correct and lie on an underlying but unknown curve, the problem is to be able to estimate the values of the curve at any position between the known points.

|  |  |  |
| --- | --- | --- |
| Linear interpolation is the simplest method of getting values at positions in between the data points. The points are simply joined by straight line segments. Each segment (bounded by two data points) can be interpolated independently. The parameter mu defines where to estimate the value on the interpolated line, it is 0 at the first point and 1 and the second point. For interpolated values between the two points mu ranges between 0 and 1. Values of mu outside this range result in extrapolation. This convention is followed for all the subsequent methods below. As with subsequent more interesting methods, a snippet of plain C code will server to describe the mathematics.  double LinearInterpolate(  double y1,double y2,  double mu)  {  return(y1\*(1-mu)+y2\*mu);  }  http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/interpolation/linear.gif  **Figure 1: Linear**  Linear interpolation results in discontinuities at each point. Often a smoother interpolating function is desirable, perhaps the simplest is cosine interpolation. A suitable orientated piece of a cosine function serves to provide a smooth transition between adjacent segments.  double CosineInterpolate(  double y1,double y2,  double mu)  {  double mu2;  mu2 = (1-cos(mu\*PI))/2;  return(y1\*(1-mu2)+y2\*mu2);  }  http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/interpolation/cosine.gif  **Figure 2: Cosine**  Cubic interpolation is the simplest method that offers true continuity between the segments. As such it requires more than just the two endpoints of the segment but also the two points on either side of them. So the function requires 4 points in all labeled y0, y1, y2, and y3, in the code below. mu still behaves the same way for interpolating between the segment y1 to y2. This does raise issues for how to interpolate between the first and last segments. In the examples here I just haven't bothered. A common solution is the dream up two extra points at the start and end of the sequence, the new points are created so that they have a slope equal to the slope of the start or end segment.  double CubicInterpolate(  double y0,double y1,  double y2,double y3,  double mu)  {  double a0,a1,a2,a3,mu2;  mu2 = mu\*mu;  a0 = y3 - y2 - y0 + y1;  a1 = y0 - y1 - a0;  a2 = y2 - y0;  a3 = y1;  return(a0\*mu\*mu2+a1\*mu2+a2\*mu+a3);  }  http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/interpolation/cubic.gif  **Figure 3: Cubic**  Paul Breeuwsma proposes the following coefficients for a smoother interpolated curve, which uses the slope between the previous point and the next as the derivative at the current point. This results in what are generally referred to as Catmull-Rom splines.  a0 = -0.5\*y0 + 1.5\*y1 - 1.5\*y2 + 0.5\*y3;  a1 = y0 - 2.5\*y1 + 2\*y2 - 0.5\*y3;  a2 = -0.5\*y0 + 0.5\*y2;  a3 = y1;  Hermite interpolation like cubic requires 4 points so that it can achieve a higher degree of continuity. In addition it has nice tension and biasing controls. Tension can be used to tighten up the curvature at the known points. The bias is used to twist the curve about the known points. The examples shown here have the default tension and bias values of 0, it will be left as an exercise for the reader to explore different tension and bias values.  /\*  Tension: 1 is high, 0 normal, -1 is low  Bias: 0 is even,  positive is towards first segment,  negative towards the other  \*/  double HermiteInterpolate(  double y0,double y1,  double y2,double y3,  double mu,  double tension,  double bias)  {  double m0,m1,mu2,mu3;  double a0,a1,a2,a3;  mu2 = mu \* mu;  mu3 = mu2 \* mu;  m0 = (y1-y0)\*(1+bias)\*(1-tension)/2;  m0 += (y2-y1)\*(1-bias)\*(1-tension)/2;  m1 = (y2-y1)\*(1+bias)\*(1-tension)/2;  m1 += (y3-y2)\*(1-bias)\*(1-tension)/2;  a0 = 2\*mu3 - 3\*mu2 + 1;  a1 = mu3 - 2\*mu2 + mu;  a2 = mu3 - mu2;  a3 = -2\*mu3 + 3\*mu2;  return(a0\*y1+a1\*m0+a2\*m1+a3\*y2);  }  http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/interpolation/hermite.gif  **Figure 4: Hermite**  While you may think the above cases were 2 dimensional, they are just 1 dimensional interpolation (the horizontal axis is linear). In most cases the interpolation can be extended into higher dimensions simply by applying it to each of the x,y,z coordinates independently. This is shown on the right for 3 dimensions for all but the cosine interpolation. By a cute trick the cosine interpolation reverts to linear if applied independently to each coordinate.  For other interpolation methods see the Bezier, Spline, and piecewise Bezier methods http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/.  http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/interpolation/linear3d.gif  **Figure 5: 3D Liner** |  |  |



**Figure 6: 3D Cubic**



**Figure 7: 3D Hermite**