Mathfx

From Unify Community Wiki

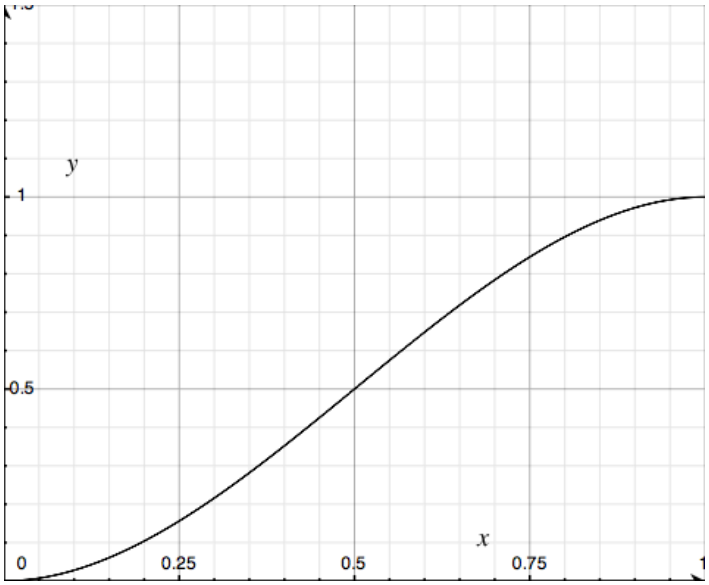
Contents

Description

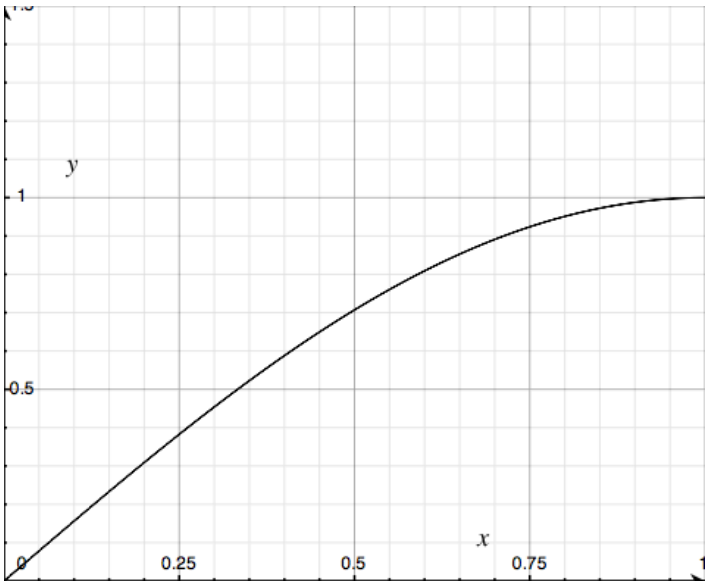
The following snippet provides short functions for floating point numbers. See the usage section for individualized information.

Usage

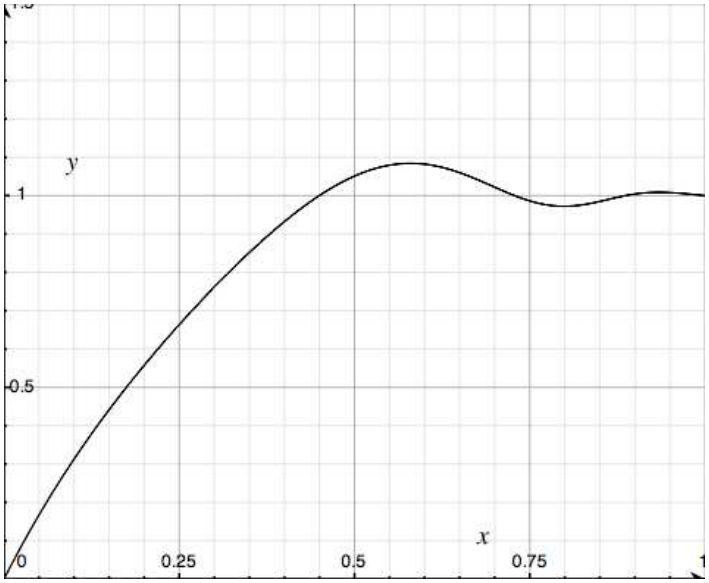
- **Hermite** - This method will interpolate while easing in and out at the limits.



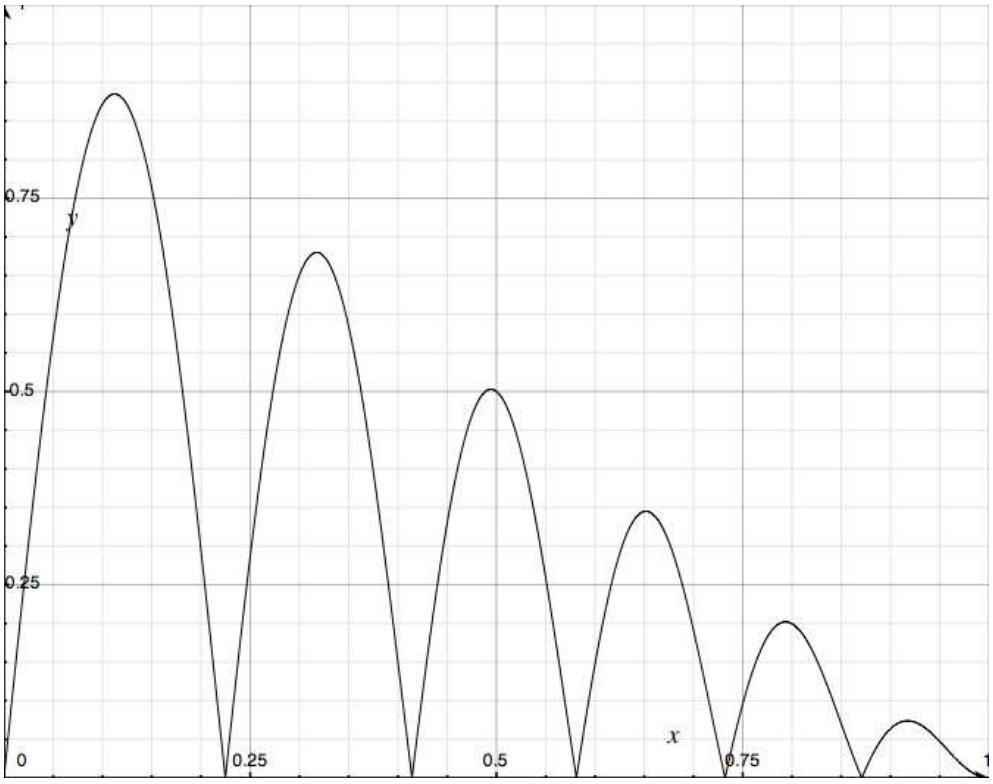
- **Sinerp** - Short for 'sinusoidal interpolation', this method will interpolate while easing around the end, when value is near one.



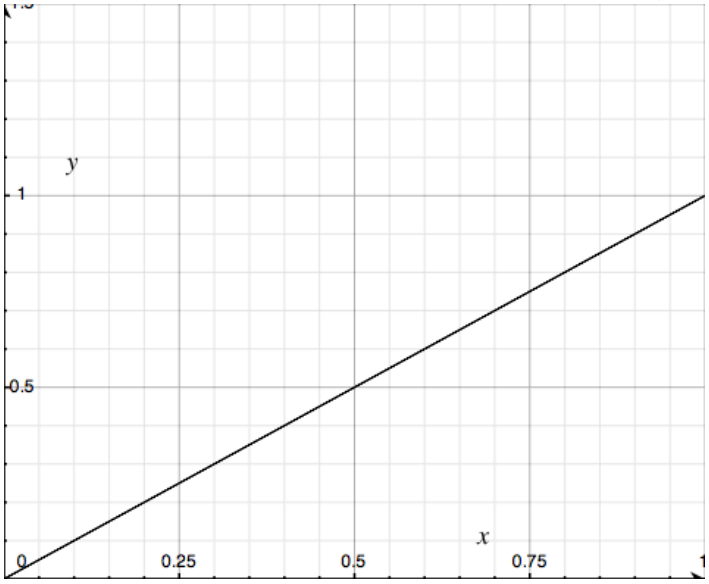
- **Coserp** - Similar to Sinerp, except it eases in, when value is near zero, instead of easing out (and uses cosine instead of sine).
- **Berp** - Short for 'boing-like interpolation', this method will first overshoot, then waver back and forth around the end value before coming to a rest.



- **Bounce** - Returns a value between 0 and 1 that can be used to easily make bouncing GUI items (a la OS X's Dock)



- **SmoothStep** - Works like Lerp, but has ease-in and ease-out of the values.
- **Lerp** - Short for 'linearly interpolate', this method is equivalent to Unity's Mathf.Lerp, included for comparison.



- **NearestPoint** - Will return the nearest point on a line to a point. Useful for making an object follow a track.
- **NearestPointStrict** - Works like NearestPoint except the end of the line is clamped.

History

- Added Approx function for testing float value within an offset range. All thanks to Opless for this!
- Clerp added by Jeff Craighead 10:51, 2 May 2008 (PDT)
- Added JavaScript conversion

C# - Mathfx.cs

```
using UnityEngine;
using System;

public class Mathfx
{
    public static float Hermite(float start, float end,
float value)
    {
        return Mathf.Lerp(start, end, value * value * (3.0f
- 2.0f * value));
    }

    public static float Sinerp(float start, float end,
float value)
    {
        return Mathf.Lerp(start, end, Mathf.Sin(value *
Mathf.PI * 0.5f));
    }

    public static float Coserp(float start, float end,
float value)
    {
        return Mathf.Lerp(start, end, 1.0f -
Mathf.Cos(value * Mathf.PI * 0.5f));
    }

    public static float Berp(float start, float end, float
value)
    {
        value = Mathf.Clamp01(value);
        value = (Mathf.Sin(value * Mathf.PI * (0.2f + 2.5f
```

```
* value * value * value)) * Mathf.Pow(1f - value, 2.2f) +
value) * (1f + (1.2f * (1f - value)));
        return start + (end - start) * value;
    }

    public static float SmoothStep (float x, float min,
float max)
    {
        x = Mathf.Clamp (x, min, max);
        float v1 = (x-min)/(max-min);
        float v2 = (x-min)/(max-min);
        return -2*v1 * v1 *v1 + 3*v2 * v2;
    }

    public static float Lerp(float start, float end, float
value)
    {
        return ((1.0f - value) * start) + (value * end);
    }

    public static Vector3 NearestPoint(Vector3 lineStart,
Vector3 lineEnd, Vector3 point)
    {
        Vector3 lineDirection = Vector3.Normalize(lineEnd-
lineStart);
        float closestPoint = Vector3.Dot((point-
lineStart),lineDirection)/Vector3.Dot(lineDirection,lineDirection);
        return lineStart+(closestPoint*lineDirection);
    }

    public static Vector3 NearestPointStrict(Vector3
lineStart, Vector3 lineEnd, Vector3 point)
    {
        Vector3 fullDirection = lineEnd-lineStart;
        Vector3 lineDirection =
Vector3.Normalize(fullDirection);
        float closestPoint = Vector3.Dot((point-
lineStart),lineDirection)/Vector3.Dot(lineDirection,lineDirection);
        return
lineStart+(Mathf.Clamp(closestPoint,0.0f,Vector3.Magnitude(fullDirection))*lineDirection);
    }
    public static float Bounce(float x) {
        return Mathf.Abs(Mathf.Sin(6.28f*(x+1f)*(x+1f)) *
(1f-x));
    }

    // test for value that is near specified float (due to
floating point inprecision)
    // all thanks to Opless for this!
    public static bool Approx(float val, float about, float
range) {
        return ( ( Mathf.Abs(val - about) < range) );
    }

    // test if a Vector3 is close to another Vector3 (due
to floating point inprecision)
    // compares the square of the distance to the square of
the range as this
    // avoids calculating a square root which is much
slower than squaring the range
    public static bool Approx(Vector3 val, Vector3 about,
float range) {
        return ( (val - about).sqrMagnitude < range*range);
```

```
    }

   /*
     * CLerp - Circular Lerp - is like lerp but handles the
wraparound from 0 to 360.
     * This is useful when interpolating eulerAngles and
the object
     * crosses the 0/360 boundary.  The standard Lerp
function causes the object
     * to rotate in the wrong direction and looks stupid.
Clerp fixes that.
     */
    public static float Clerp(float start , float end,
float value){
        float min = 0.0f;
        float max = 360.0f;
        float half = Mathf.Abs((max - min)/2.0f);//half the
distance between min and max
        float retval = 0.0f;
        float diff = 0.0f;

        if((end - start) < -half){
            diff = ((max - start)+end)*value;
            retval =  start+diff;
            }
            else if((end - start) > half){
                diff = -((max - end)+start)*value;
                retval =  start+diff;
            }
            else retval =  start+(end-start)*value;

            // Debug.Log("Start: "  + start + "   End: " +
end + "  Value: " + value + "  Half: " + half + "  Diff: "
+ diff + "  Retval: " + retval);
            return retval;
        }

}
```

Js - Mathfx.js

I converted from the original C# format above.

```
static function Hermite(start : float, end : float, value :
float) : float
{
    return Mathf.Lerp(start, end, value * value * (3.0 -
2.0 * value));
}

static function Sinerp(start : float, end : float, value :
float) : float
{
    return Mathf.Lerp(start, end, Mathf.Sin(value *
Mathf.PI * 0.5));
}

static function Coserp(start : float, end : float, value :
```

```
float) : float
{
    return Mathf.Lerp(start, end, 1.0 - Mathf.Cos(value *
Mathf.PI * 0.5));
}

static function Berp(start : float, end : float, value :
float) : float
{
    value = Mathf.Clamp01(value);
    value = (Mathf.Sin(value * Mathf.PI * (0.2 + 2.5 *
value * value * value)) * Mathf.Pow(1 - value, 2.2) +
value) * (1 + (1.2 * (1 - value)));
    return start + (end - start) * value;
}

static function SmoothStep (x : float, min : float, max :
float) : float
{
    x = Mathf.Clamp (x, min, max);
    var v1 = (x-min)/(max-min);
    var v2 = (x-min)/(max-min);
    return -2*v1 * v1 *v1 + 3*v2 * v2;
}

static function Lerp(start : float, end : float, value :
float) : float
{
    return ((1.0 - value) * start) + (value * end);
}

static function NearestPoint(lineStart : Vector3, lineEnd :
Vector3, point : Vector3) : Vector3
{
    var lineDirection = Vector3.Normalize(lineEnd-
lineStart);
    var closestPoint = Vector3.Dot((point-
lineStart),lineDirection)/Vector3.Dot(lineDirection,lineDirection);
    return lineStart+(closestPoint*lineDirection);
}

static function NearestPointStrict(lineStart : Vector3,
lineEnd : Vector3, point : Vector3) : Vector3
{
    var fullDirection = lineEnd-lineStart;
    var lineDirection = Vector3.Normalize(fullDirection);
    var closestPoint = Vector3.Dot((point-
lineStart),lineDirection)/Vector3.Dot(lineDirection,lineDirection);
    return
lineStart+(Mathf.Clamp(closestPoint,0.0,Vector3.Magnitude(fullDirection))*lineDirection);
}
static function Bounce(x : float) : float {
    return Mathf.Abs(Mathf.Sin(6.28*(x+1)*(x+1)) * (1-x));
}

// test for value that is near specified float (due to
floating point inprecision)
// all thanks to Opless for this!
static function Approx(val : float, about : float, range :
float) : boolean {
    return ( ( Mathf.Abs(val - about) < range) );
}
```

```
// test if a Vector3 is close to another Vector3 (due to
floating point inprecision)
// compares the square of the distance to the square of the
range as this
// avoids calculating a square root which is much slower
than squaring the range
static function Approx(val : Vector3, about : Vector3,
range : float) : boolean {
   return ( (val - about).sqrMagnitude < range*range);
}

// CLerp - Circular Lerp - is like lerp but handles the
wraparound from 0 to 360.
// This is useful when interpolating eulerAngles and the
object
// crosses the 0/360 boundary.  The standard Lerp function
causes the object
// to rotate in the wrong direction and looks stupid. Clerp
fixes that.
static function Clerp(start : float, end : float, value :
float) : float {
   var min = 0.0;
   var max = 360.0;
   var half = Mathf.Abs((max - min)/2.0);//half the
distance between min and max
   var retval = 0.0;
   var diff = 0.0;

   if((end - start) < -half){
       diff = ((max - start)+end)*value;
       retval =  start+diff;
   }
   else if((end - start) > half){
       diff = -((max - end)+start)*value;
       retval =  start+diff;
   }
   else retval =  start+(end-start)*value;

   // Debug.Log("Start: "  + start + "   End: " + end + "
Value: " + value + "  Half: " + half + "  Diff: " + diff +
"  Retval: " + retval);
   return retval;
}
```

Retrieved from "http://www.unifycommunity.com/wiki/index.php?title=Mathfx"

Categories: Concepts | Math | C Sharp

- This page was last modified 07:39, 23 October 2010.