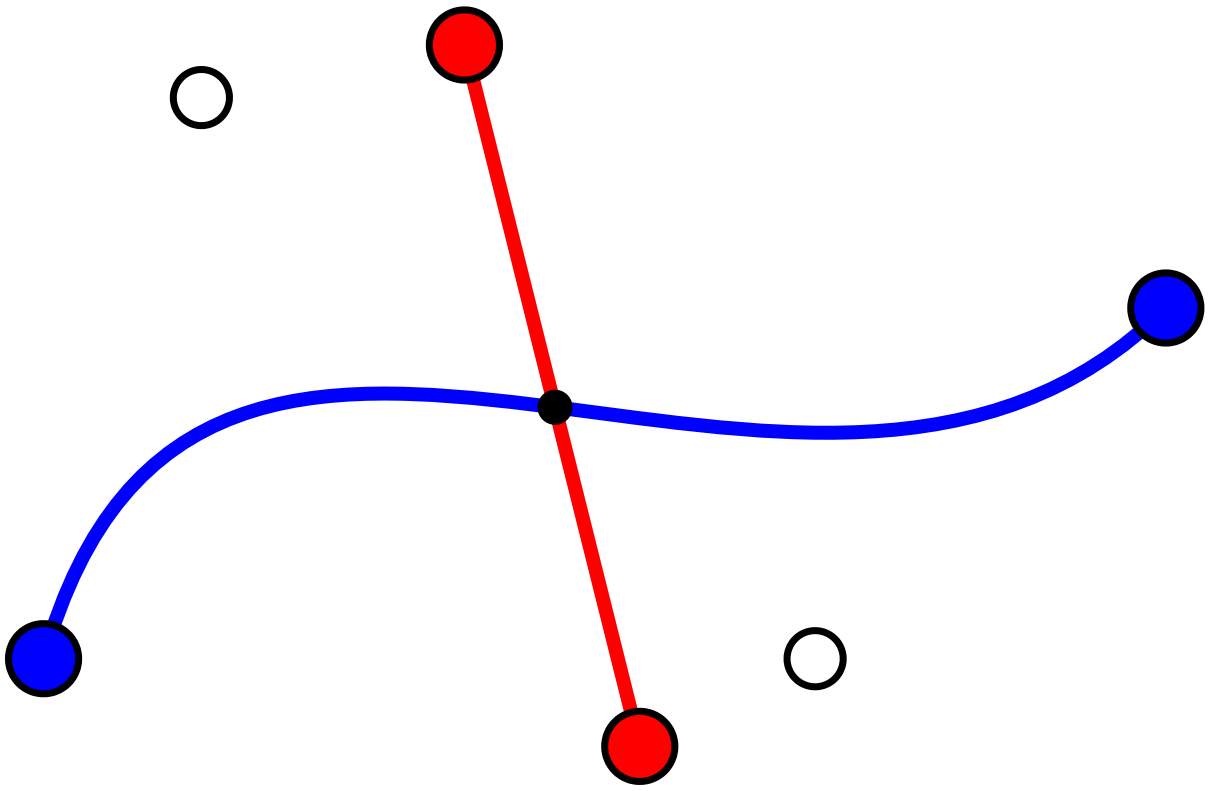
COMPUTING INTERSECTIONS BETWEEN A CUBIC BEZIER CURVE AND A LINE

INTRODUCTION

Last week I was running a Starfish simulation of a molecular transport through a vent. Starfish was designed to support both linear and cubic spline representation of surfaces. But while testing the code by tracing particles, I noticed that the particle hits with curved surfaces were not being computed correctly. Digging deeper I found the issue: I never implemented the algorithm for finding intersection between a cubic and a line! Instead, the code was using the line-line intersection by connecting the two end points of the curve. Oops!

Before continuing, let me just say that I am really starting to enjoy algorithm development using interactive HTML technologies! In the past, I would write the code in Java or C++ and have it output the splines and intersection points to a file which I would subsequently visualize using some plotting program. Or using Matlab, I could do the plotting right from the program. But I would still be confined to testing just a single case. With HTML, we can do something much better: we can interactively manipulate the curves and see the algorithm respond in real time!

DEMO



You can try this out in the demo above. If everything loaded fine, you should see a blue cubic Bezier curve and a red line. You will also see two white circles, these are the two control points $\mathbf{P}_1$ and $\mathbf{P}_2$ defining the cubic. As you change the curves by dragging the large circles, you should see a small black dot track the intersection point. You will see additional points appear if you orient the curve such that there are multiple intersections. This is shown below in Figure 1.

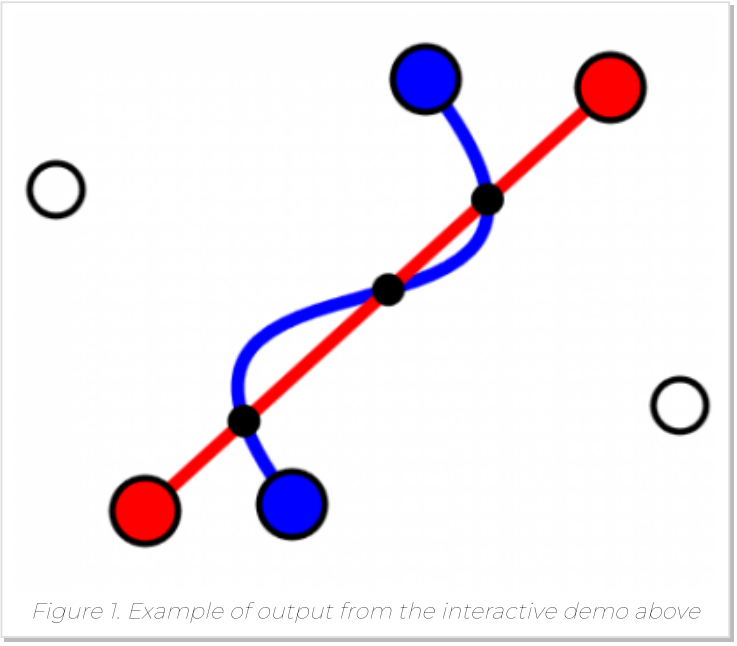


Figure 1. Example of output from the interactive demo above

Math Background

So how does this code work? The visualization is similar to the article on smooth splines through prescribed points. This mathematical algorithm is based on this answer. One way to represent an infinitely-long line is as follows

$$\frac{(x-x_1)}{(x_2-x_1)} = \frac{(y-y_1)}{(y_2-y_1)}$$

which can be rewritten as

$$x(y_2-y_1)+y(x_1-x_2)+x_1(y_1-y_2)+y_1(x_2-x_1)=0$$

or

$$Ax+By+C=0 \qquad (1)$$

We next constrain the (x,y) pairs to those located on the cubic curve. A cubic Bezier curve is given by
$\mathbf{r}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3, \quad t \in [0,1]$
where $\mathbf{r}(t)$ is the position vector. If we substitute these (x,y) components into equation (1), we obtain a cubic equation in t. Finding the intersection points is then a "simple" matter of finding the roots of the cubic equation.

Cubic Roots

One way to find a single root is using Newton's method. Unfortunately, a cubic can have up to 3 roots. This is because, as shown in Figure 1, a line can intersect a cubic spline in up to 3 locations. Since we are using this algorithm for particle tracing, we are interested in the first intersection along the line. There is no guarantee that the Newton's method will converge to this root. As such, we need to find all existing roots and sort them. Finding additional roots with Newton's method is possible but not trivial. Third-order polynomials also have an analytical solution for their roots. But unlike the well known quadratic formula, there are multiple equations for cubic roots. In the end, I ended up using algorithm from Stephen Schmitt's site:

```
/*based on http://mysite.verizon.net/res148h4j/javascript/script_exact_cubic.html#the%20source%20code*/
function cubicRoots(P)
{
    var a=P[0];
    var b=P[1];
    var c=P[2];
    var d=P[3];

    var A=b/a;
```

```
    var B=c/a;
    var C=d/a;

    var Q, R, D, S, T, Im;

    var Q = (3*B - Math.pow(A, 2))/9;
    var R = (9*A*B - 27*C - 2*Math.pow(A, 3))/54;
    var D = Math.pow(Q, 3) + Math.pow(R, 2);    // polynomial discriminant

    var t=Array();

    if (D >= 0)                                 // complex or duplicate roots
    {
        var S = sgn(R + Math.sqrt(D))*Math.pow(Math.abs(R + Math.sqrt(D)),(1/3));
        var T = sgn(R - Math.sqrt(D))*Math.pow(Math.abs(R - Math.sqrt(D)),(1/3));

        t[0] = -A/3 + (S + T);                    // real root
        t[1] = -A/3 - (S + T)/2;                  // real part of complex root
        t[2] = -A/3 - (S + T)/2;                  // real part of complex root
        Im = Math.abs(Math.sqrt(3)*(S - T)/2);    // complex part of root pair

        /*discard complex roots*/
        if (Im!=0)
        {
            t[1]=-1;
            t[2]=-1;
        }

    }
    else                                          // distinct real roots
    {
        var th = Math.acos(R/Math.sqrt(-Math.pow(Q, 3)));

        t[0] = 2*Math.sqrt(-Q)*Math.cos(th/3) - A/3;
        t[1] = 2*Math.sqrt(-Q)*Math.cos((th + 2*Math.PI)/3) - A/3;
        t[2] = 2*Math.sqrt(-Q)*Math.cos((th + 4*Math.PI)/3) - A/3;
        Im = 0.0;
    }

    /*discard out of spec roots*/
    for (var i=0;i<3;i++)
        if (t[i]<0 || t[i]>1.0) t[i]=-1;

    /*sort but place -1 at the end*/
    t=sortSpecial(t);

    console.log(t[0]+" "+t[1]+" "+t[2]);
    return t;
}
```

This algorithm returns an array of parametric intersection locations along the cubic, with -1 indicating an out-of-bounds intersection (before or after the end point or in the imaginary plane). We also need to verify that the intersections are within the limits of the linear segment. This is done by the following code:

```
/*computes intersection between a cubic spline and a line segment*/
function computeIntersections(px,py,lx,ly)
{
    var X=Array();

    var A=ly[1]-ly[0];      //A=y2-y1
    var B=lx[0]-lx[1];      //B=x1-x2
    var C=lx[0]*(ly[0]-ly[1]) +
          ly[0]*(lx[1]-lx[0]);  //C=x1*(y1-y2)+y1*(x2-x1)

    var bx = bezierCoeffs(px[0],px[1],px[2],px[3]);
    var by = bezierCoeffs(py[0],py[1],py[2],py[3]);

    var P = Array();
    P[0] = A*bx[0]+B*by[0];             /*t^3*/
    P[1] = A*bx[1]+B*by[1];             /*t^2*/
    P[2] = A*bx[2]+B*by[2];             /*t*/
    P[3] = A*bx[3]+B*by[3] + C; /*1*/

    var r=cubicRoots(P);

    /*verify the roots are in bounds of the linear segment*/
    for (var i=0;i<3;i++)
    {
        t=r[i];

        X[0]=bx[0]*t*t*t+bx[1]*t*t+bx[2]*t+bx[3];
        X[1]=by[0]*t*t*t+by[1]*t*t+by[2]*t+by[3];

        /*above is intersection point assuming infinitely long line segment,
          make sure we are also in bounds of the line*/
        var s;
        if ((lx[1]-lx[0])!=0)           /*if not vertical line*/
            s=(X[0]-lx[0])/(lx[1]-lx[0]);
        else
            s=(X[1]-ly[0])/(ly[1]-ly[0]);

        /*in bounds?*/
        if (t<0 || t>1.0 || s<0 || s>1.0)
        {
            X[0]=-100;  /*move off screen*/
            X[1]=-100;
        }

        /*move intersection point*/
        I[i].setAttributeNS(null,"cx",X[0]);
        I[i].setAttributeNS(null,"cy",X[1]);
    }
}
```

As you can see, we are always plotting 3 intersection locations, but the out-of-bounds intersections are moved off screen to location (-100,-100). The above code also does not sort the intersections along the line, but this change is ease to implement by storing the `s` parametric positions in array.

Source Code

And that's it. You can download the code by right clicking and selecting "save as" on this link: cubic-line.svg

Related Articles:

- Flow in a Nozzle
- HTML5 + Javascript DSMC Simulation
- Code Optimization: Speed up your code by rearranging data access

Subscribe to the newsletter and follow us on Twitter. Send us an email if you have any questions.

Contact: info@particleincell.com. Find us on Twitter, LinkedIn, and Github.

Site map: projects : publications : jobs : codes : courses : blog
Latest articles: Experimental investigation of QCM-derived sticking coefficients : Quasi Steady-State Testing Approach for High Power Hall Thrusters : 2020 Papers : Particulate Surface Adhesion Sandbox : Setting up an Ubuntu Linux Cluster