

贝塞尔曲线的求导、弧长参数化与分段拟合方法



iceytan

中山大学 计算机技术硕士在读

关注他

5 人赞同了该文章

本文篇幅较长，分为

1. N阶贝塞尔曲线的定义
2. N阶贝塞尔曲线的导数求解
3. 曲线的长度求解，arc-length参数化原理及贝塞尔曲线匀弧长采样逼近
4. 贝塞尔曲线的分段拟合

每个部分，都基于Eigen、OpenCV，给出简单的C++代码实现例子。

1. 基本定义

贝塞尔曲线是参数化曲线（Parametric Curves）的一种，其 n 阶次曲线具有如下的形式：

$$\mathbf{C}(t) = \sum_{i=0}^n B_{i,n}(t) \mathbf{p}_i \quad (1)$$

其中 $t \in [0, 1]$ ，写成矩阵的形式，有：

$$\mathbf{C}(t) = [B_{0,n}(t), B_{1,n}(t), \dots, B_{n,n}(t)] \begin{bmatrix} \mathbf{p}_0^\top \\ \mathbf{p}_1^\top \\ \vdots \\ \mathbf{p}_n^\top \end{bmatrix} \quad (2)$$

(1)与(2)中的 $B_{i,n}(t)$ 称为在 t 参数下的贝塞尔曲线系数，定义是：

$$B_{i,n}(t) = \frac{n!}{i!(n-i)!} (1-t)^{n-i} t^i \quad (3)$$

前半部分实际上是二项式系数，在实现时，可以先使用“杨辉三角”预先计算 $B_{i,n}(t)$ 的前半部分，在进行查询时再乘以对应的 t 相关量。基于Eigen，实现如下：

```

typedef Eigen::Matrix<double, 1, N+1> ParameterType;
typedef Eigen::Matrix<double, N+1, 2> PointsType;

PointsType points_;           // 控制点
ParameterType pascals_triangle_; // 二项式系数

...

void compute_combinator()      // 预计算
{
    pascals_triangle_ = Eigen::Matrix<double, 1, N+1>::Ones();

    for(int i = 2; i <= N; ++i)
        for(int j = i-1; j > 0; --j)
        {
            pascals_triangle_[j] = pascals_triangle_[j] + pascals_triangle_[j-1];
        }
}

Eigen::Vector2d at(const double& t) // 查询点
{
    ParameterType T;
    for(int i=0; i<=N; ++i)
    {
        T[i] = pow(t, i) * pow(1.0-t, N-i);
    }
    return T.cwiseProduct(pascals_triangle_) * points_;
}

```

为了保持高阶时 t 幂次的数值稳定性，可以使用 *De Casteljau's Algorithm* 迭代进行(2)(3)的计算。以二次贝塞尔曲线为例，基于(2)展开有：

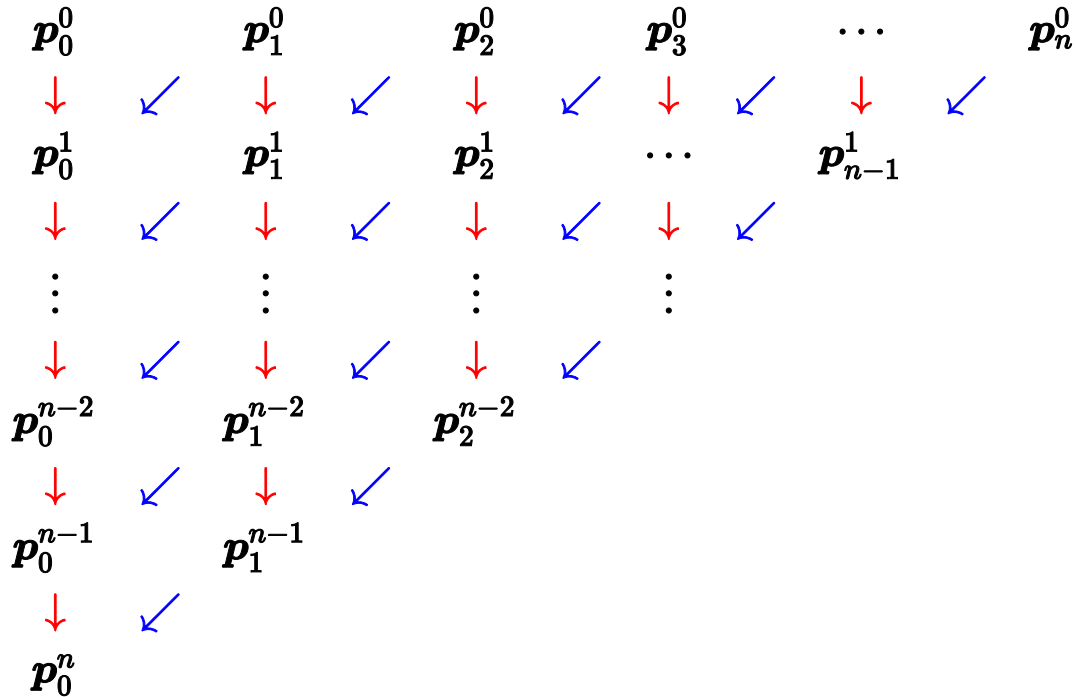
$$\begin{aligned}
 \mathbf{C}_2(t) &= (1-t)^2 \mathbf{p}_0 + 2(1-t)t \mathbf{p}_1 + t^2 \mathbf{p}_2 \\
 &= (1-t) [(1-t) \mathbf{p}_0 + t \mathbf{p}_1] + t [(1-t) \mathbf{p}_1 + t \mathbf{p}_2]
 \end{aligned} \tag{4}$$

可以看到高阶项的计算，可以通过复合两个低阶项来完成，而两个复合的低阶项，实际上是该 $\mathbf{C}_2(t)$ 曲线的控制多边形顶点的线性插值来计算。此外，基于(2)展开三次贝塞尔曲线 (cubic Bezier curve) 有：



$$\begin{aligned}
 \mathbf{C}_3(t) &= (1-t)^3 \mathbf{p}_0 + 3(1-t)^2 t \mathbf{p}_1 + 3(1-t) t^2 \mathbf{p}_2 + t^3 \mathbf{p}_3 \\
 &= (1-t) [(1-t)^2 \mathbf{p}_0 + 2(1-t)t \mathbf{p}_1 + t^2 \mathbf{p}_2] + \\
 &\quad t [(1-t)^2 \mathbf{p}_1 + 2(1-t)t \mathbf{p}_2 + t^2 \mathbf{p}_3].
 \end{aligned} \tag{5}$$

(5) 中后半部分是两个类似 $\mathbf{C}_2(t)$ 的部分以不同权重进行相乘的结果。下图简单描述了起始的控制点 $\mathbf{p}_0^0, \mathbf{p}_1^0, \dots, \mathbf{p}_n^0$ 与最终的 \mathbf{p}_0^n 之间层次：



其中，红色箭头 \downarrow 表示权重 $(1-t)$ ，蓝色箭头 \swarrow 表示权重 t ，以下是简单实现：

```

Eigen::Vector2d at(const double& t)    // 查询点，基于De Casteljau's Algorithm
{
    PointsType temp = points_;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N - i; ++j)
            temp.row(j) = (1.0 - t) * temp.row(j) + t * temp.row(j + 1);
    }
    return temp.row(0);
}

```

2. n-阶导数



2.1 推导

贝塞尔曲线的 k 阶导数可有如下的形式：

$$\begin{aligned} \mathbf{C}^{(k)}(t) &= n(n-1)(n-2)\dots(n-k+1) \sum_{i=0}^{n-k} B_{i,n-k} \mathbf{p}_i^{(k)} \\ \mathbf{p}_i^{(k)} &= \mathbf{p}_{i+1}^{(k-1)} - \mathbf{p}_i^{(k-1)} \end{aligned} \quad (6)$$

为了证明上式，先证明下面的结论 *Theorem 1* 和 *Theorem 2*：

Theorem 1. n 阶贝塞尔曲线的系数项 $B_{i,n}(t)$ 满足以下公式：

$$B'_{i,n}(t) = n(B_{i-1,n-1}(t) - B_{i,n-1}(t)) \quad (7)$$

证明如下：

$$\begin{aligned} \frac{d}{dt} \left(\frac{n!}{(n-i)!i!} (1-t)^{n-i} t^i \right) &= -(n-i) \frac{n!}{(n-i)!i!} (1-t)^{n-i-1} t^i + i \frac{n!}{(n-i)!i!} (1-t)^{n-i} t^{i-1} \\ &= -n \frac{(n-1)!}{(n-1-i)!i!} (1-t)^{n-1-i} t^i + n \frac{(n-1)!}{(n-i)!(i-1)!} (1-t)^{n-i} t^{i-1} \\ &= -n B_{i,n-1}(t) + n B_{i-1,n-1}(t) \end{aligned} \quad (8)$$

Theorem 2. 贝塞尔曲线在 t 处的一阶段导数满足

$$\begin{aligned} \mathbf{C}'(t) &= \sum_{i=0}^{n-1} \mathbf{p}_i^{(1)} B_{i,n-1}(t) \\ \mathbf{p}_i^{(1)} &= n(\mathbf{p}_{i+1} - \mathbf{p}_i) \end{aligned} \quad (9)$$

证明如下：

由于 $B'_{i,n}(t) = n(B_{i-1,n-1}(t) - B_{i,n-1}(t))$ ，并且设 $B_{-1,n-1}(t) = B_{n,n-1}(t) = 0$ ，那么：



$$\begin{aligned}
\mathbf{C}^{(1)}(t) &= \sum_{i=0}^n \mathbf{p}_i B_{i,n}^{(1)}(t) \\
&= \sum_{i=0}^n \mathbf{p}_i n (B_{i-1,n-1}(t) - B_{i,n-1}(t)) \\
&= \sum_{i=1}^n n \mathbf{p}_i B_{i-1,n-1}(t) - \sum_{i=0}^{n-1} n \mathbf{p}_i B_{i,n-1}(t) \\
&= \sum_{i=0}^{n-1} n \mathbf{p}_{i+1} B_{i,n-1}(t) - \sum_{i=0}^{n-1} n \mathbf{p}_i B_{i,n-1}(t) \\
&= \sum_{i=0}^{n-1} n (\mathbf{p}_{i+1} - \mathbf{p}_i) B_{i,n-1}(t)
\end{aligned} \tag{10}$$

为了得到更高阶的导数公式，只需要重复 (9) 式即可得到二阶导数：

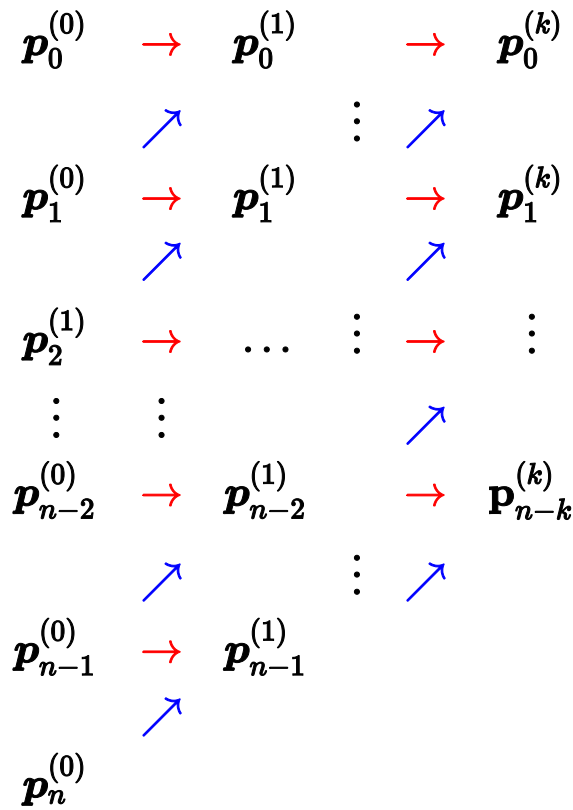
$$\begin{aligned}
\mathbf{C}^{(2)}(t) &= \sum_{i=0}^{n-2} \mathbf{p}_i^{(2)} B_{i,n-1}(t) \\
\mathbf{p}_i^{(2)} &= (n-1) (\mathbf{p}_{i+1}^{(1)} - \mathbf{p}_i^{(1)}) = (n-1)n (\mathbf{p}_{i+2} - 2\mathbf{p}_{i+1} + \mathbf{p}_i)
\end{aligned} \tag{11}$$

不断重复这一过程，最终可以获得 (6) 式。

2.2 实现

我们考虑如何计算 (6) 式中的 $\mathbf{p}_i^{(k)}$ 。观察式 (11)，可发现与 (5), (6) 有高度的相似性，于是我们同样使用 De Casteljau's Algorithm 从底向上迭代计算 $\mathbf{p}_i^{(k)}$ ，下面是一个简单的示意图：





其中蓝色箭头 (\rightarrow) 表示取正, 红色箭头 (\nwarrow) 表示取负。在计算完毕 $p_i^{(k)}$ 后, 使用第一节中的方法计算剩余部分, 话不多说, 上代码:

```

PointType
at(const double& t, const int& derivative_order = 0)
{
    PointsType temp = points_;

    int prefix = 1;
    for(int i = 0; i < derivative_order; ++i) prefix *= N-i;

    // 1. 计算  $p^{(k)}$ 
    for(int k = 0; k < derivative_order; ++k)
    {
        int I_range = N-derivative_order;
        for(int i = 0; i <= I_range; ++i)
        {
            temp.row(i) = temp.row(i+1) - temp.row(i);
        }
    }

    // 2. 计算  $C^{(k)}(t) = \sum_{i=0}^{N-k} B_{i,N-k}(t) p^{(k)}$ 
    int I_range = N - derivative_order;
    for (int i = 0; i < I_range; ++i)
    {

```



```

    int J_range = I_range - i;
    for (int j = 0; j < J_range; ++j)
        temp.row(j) = (1.0 - t) * temp.row(j) + t * temp.row(j + 1);
}
return prefix * temp.row(0);
}

```

基于^[1]中的例子，简单验证一下：

Example 6.2. Consider a cubic Bézier curve defined by control points $(1, 1)$, $(3, 1)$, $(4, 2)$, and $(6, 3)$. The differences of the control points are

$$(3, 1) - (1, 1) = (2, 0), \quad (4, 2) - (3, 1) = (1, 1), \quad (6, 3) - (4, 2) = (2, 1).$$

Multiplication by three yields the control points of the first derivative

$$\mathbf{b}_0^{(1)} = (6, 0), \quad \mathbf{b}_1^{(1)} = (3, 3), \quad \mathbf{b}_2^{(1)} = (6, 3).$$

The derivative can be expressed as the quadratic Bézier curve

$$(6, 0)(1 - t)^2 + (3, 3)2(1 - t)t + (6, 3)t^2.$$

To determine the control points of the second derivative we compute the differences

$$(3, 3) - (6, 0) = (-3, 3), \quad (6, 3) - (3, 3) = (3, 0)$$

and multiply by two to get $\mathbf{b}_0^{(2)} = (-6, 6)$ and $\mathbf{b}_1^{(2)} = (6, 0)$. The second derivative of the cubic Bézier curve can be expressed as the linear curve

$$(-6, 6)(1 - t) + (6, 0)t.$$

To obtain the tangent vector at, for instance, $t = 0.5$, we make a substitution $t = 0.5$ in the first derivative and get

$$\mathbf{C}'(t) = (4.5, 2.25).$$

```

#include <iostream>
#include "bezier.hpp" // 悄咪咪封装了
using namespace std;

void mini_test()
{
    Bezier bezier {{1,1},{3,1},{4,2},{6,3}};
    cout << bezier.at(0.5, 1) << endl; // (4.5, 2.25)
}

int main()
{
    mini_test();
    return 0;
}

```



编译运行：

```
→ ParametricCurves ./cmake-build-debug/ParametricCurves
4.5 2.25
```

似乎可行

3. 弧长的重参数化 (arc-length parametrization)



不等距取点



重参数化后等距取点

在实际的应用中，时常需要“贝塞尔曲线上，弧长1/4位置在哪？”这种需求，如果直接以 $t = 0.4$ 代入曲线方程并取点，是不准确的。这是因为在弧上，每一点处的速度不相同，相同的 Δt 内对应“走过”的弧长也不相同。为了能在贝塞尔曲线上等距采点，考虑面向弧长的重参数化 (arc-length parametrization) 问题。

3.1 求弧长

首先我们先从如何求曲线弧长 (arc-length) 开始。为了求得弧长，设定 $0 = a = t_1 < t_2 < \dots < t_m = b = 1$ ，使用区间内离散点之间的距离之和进行长度的近似：

$$L \approx \sum_{j=1}^{m-1} |\mathbf{C}(t_{j+1}) - \mathbf{C}(t_j)|$$

进一步，根据中值定理，可知：

$$L \approx \sum_{j=1}^{m-1} |\mathbf{C}'(t_j^*)| (t_{j+1} - t_j)$$



当 $t_{j+1} - t_j \rightarrow 0$ 时，我们得到精准的曲线长度：

$$L \approx \int_0^1 |\mathbf{C}'(t)| dt \quad (12)$$

对于贝塞尔曲线来说，这一个定积分是没有解析解的，需要通过数值积分的方式进行求解，这里简单使用Simpson's 3/8 rule进行逼近。这一个算法讲的是如何通过函数点获取积分值：

$$\begin{aligned} \int_a^b f(x) dx &\approx \frac{3h}{8} \left[f(a) + 3f\left(\frac{2a+b}{3}\right) + 3f\left(\frac{a+2b}{3}\right) + f(b) \right] \\ &= \frac{(b-a)}{8} \left[f(a) + 3f\left(\frac{2a+b}{3}\right) + 3f\left(\frac{a+2b}{3}\right) + f(b) \right] \end{aligned} \quad (13)$$

那么结合 (6) (12) (13)，很容易就可以求出任何阶次的贝塞尔曲线的弧长了。参考 zhuanlan.zhihu.com/p/53...，这里给一个自适应的3/8辛普森的简单的实现：

```
struct NumericalIntegration
{
    template <typename F>
    static double
    simpson_3_8(F&& derivative_func, const double& L, const double& R)
    {
        double mid_L = (2*L + R) / 3.0, mid_R = (L + 2*R) / 3.0;
        return (derivative_func(L) +
                3.0 * derivative_func(mid_L) +
                3.0 * derivative_func(mid_R) +
                derivative_func(R)) * (R - L) / 8.0;
    }

    template <typename F>
    static double
    adaptive_simpson_3_8(F&& derivative_func,
        const double& L, const double& R, const double& eps = 0.0001)
    {
        const double mid = (L + R) / 2.0;
        double ST = simpson_3_8(derivative_func, L, R),
            SL = simpson_3_8(derivative_func, L, mid),
            SR = simpson_3_8(derivative_func, mid, R);
        double ans = SL + SR - ST;
        if(fabs(ans) <= 15.0 * eps) return SL + SR + ans / 15.0;
        return adaptive_simpson_3_8(derivative_func, L, mid, eps / 2.0) +
            adaptive_simpson_3_8(derivative_func, mid, R, eps / 2.0);
    }
};
```



```

}
};

```

计算贝塞尔长度时，只需：

```

void computeLength()
{
    auto df = [&](double t) -> double
    {
        return this->at(t,1).norm();
    };
    length_ = NumericalIntegration::adaptive_simpson_3_8(df, 0, 1);
}

```

3.2 arc-length 重参数化

通过定义一个映射 $\Lambda : [a, b] \rightarrow [0, L]$ ，获取原弧线参数 t 的定义域 $[a, b]$ 到弧长区间 $[0, L]$ 上的一个“满射”：

$$\Lambda(t) = \int_a^t |\mathbf{C}'(z)| dz \quad (14)$$

由于这一个函数是严格递增，且连续可导的，那么该映射 Λ 的“反函数”，我们假设为 $\phi(s)$ ，会将 $[0, L] \rightarrow [a, b]$ 。那么此时，在给定 s 位置下，对应的曲线的参数为 $\phi(s)$ ，令 $|\mathbf{C}'(\phi(s))| = |\tilde{\mathbf{C}}'(s)|$ ， $\mathbf{C}'(\phi(s))$ 关于 t 求导有：

$$\begin{aligned} \tilde{\mathbf{C}}'(s) &= \mathbf{C}'(t) \frac{d\phi(s)}{ds} \\ &= \mathbf{C}'(t) \frac{1}{\Lambda'(t)} \\ &= \frac{\mathbf{C}'(t)}{|\mathbf{C}'(t)|} \end{aligned} \quad (15)$$

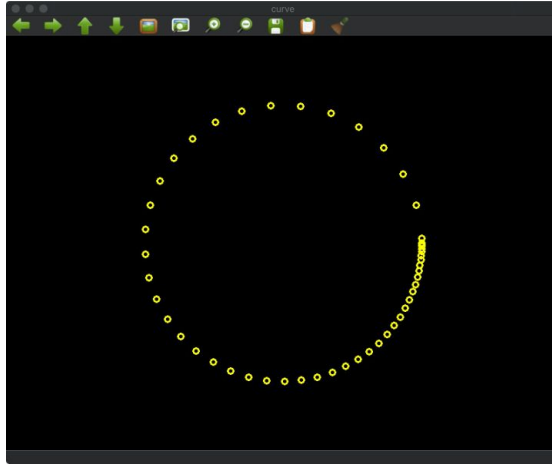
整体取2范数后，得到 $|\tilde{\mathbf{C}}'(s)| = 1$ 。这一个结论和我们先前的说辞是一致的，即：在贝塞尔曲线上，每一点处的速度不相同，相同的 Δt 内对应“走过”的弧长也不相同。当重参数化后的函数导数值恒定为 1，那么基于此即可得到等弧长间距的点。

具体举一个栗子。考虑这样一个曲线：



$$\gamma(t) = (R \cos t^2, R \sin t^2), \quad t \in [0, (2\pi)^{1/2}]$$

直接使用 t 取点时，具有明显的不均匀现象：



接着，我们先求其 $\Lambda(t)$ ，再求其 $\Lambda(t)$ 的反函数：

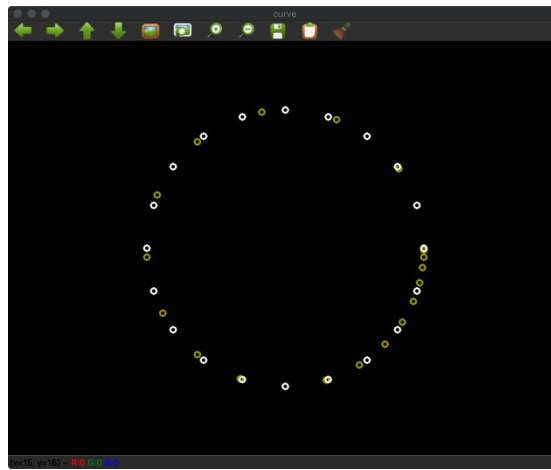
$$\begin{aligned} \Lambda(t) &= \int_0^t |\gamma'_1(z)| dz \\ &= \int_0^t \sqrt{4R^2 z^2 \sin^2 z^2 + 4R^2 z^2 \cos^2 z^2} dz \\ &= \int_0^t 2Rz dz \\ &= Rt^2 \end{aligned} \tag{16}$$

其反函数为 $\phi(s) = \sqrt{s/R}$ ，所以其arc-length 重参数化形式为：

$$\tilde{\gamma}(s) = \gamma(\phi(s)) = \left(R \cos \frac{s}{R}, R \sin \frac{s}{R} \right), \quad s \in [0, 2\pi R]$$

使用这结果，再次计算，有：





白色部分为arc-length参数化后结果

以上涉及的实验代码：

```
string window_name = "curve";
int win_width = 800, win_height = 600, R = 200;

cv::namedWindow(window_name);
cv::Mat img {win_height, win_width, CV_8UC3, cv::Scalar(0)};

auto f = [](double R, double t)->cv::Point2d {return {R*cos(t*t),R*sin(t*t)}};
auto af = [](double R, double s)->cv::Point2d {return {R*cos(s/R),R*sin(s/R)}};

auto draw_func = [&](auto func, double step, double end, const cv::Scalar& color)
{
    double t = 0;
    auto display_bias = cv::Point2d(win_width/2, win_height/2);
    while(t<=end) {
        auto p = func(R, t) + display_bias;
        cv::circle(img, p, 4, color,2);
        t +=step;
    }
};

draw_func(f, sqrt(2*M_PI)/20, sqrt(2*M_PI), {0, 155, 155});
draw_func(af, 2*M_PI*R/20, 2*M_PI*R, {255, 255, 255});

cv::imshow(window_name, img);
cv::waitKey(0);
```

3.3 贝塞尔曲线的 arc-length 参数化逼近



那么，对于贝塞尔曲线来说，可以运用相同的方法求解吗？答案是，**不完全能**，对于高阶次的贝塞尔曲线来说，不具有如同 (16) 一般的闭式解。好了朋友们，那么本文就要在此完毕了吗？先前确实是说了些和求解贝塞尔曲线的arc-length参数化点无关的话，但也是本着科普的心情。那么进入正题：如何通过数值方法求解“从起点出发的 s 弧长位置的贝塞尔参数 t_s ”呢？

在之前的讨论中，我们知道可以基于长度积分公式 (6) 和Simpson's rule (12) 得到贝塞尔曲线的长度，那么若要求得某弧长的参数位置，利用暴力的方法，只需要遍历一遍积分区间 $[0, t_s]$ ，在足够靠近目标弧长值时停下即可。但是，暴力法需要较小的粒度 Δt ，如果贝塞尔曲线很长，需要的查询次数很多，必将消耗大量的时间。接下来的部分，将说明如何使用牛顿法 (Newton's method) 找到 t_s ，为此，将寻找 t_s 的问题建模为：

$$t_s = \arg \min_{t \in [0,1]} (L(t) - s)^2 = \arg \min_{t \in [0,1]} g(t) \quad (17)$$

其中， $L(t)$ 来自式 (12)，表示 t 参数位置对应的曲线位置距离其起点的弧长：

$$L(t) \approx \int_0^t |\mathbf{C}'(t)| dt \quad (18)$$

为了利用牛顿法，求 $g(t)$ 的一阶导数与二阶导数：

$$\begin{aligned} g'(t) &= 2d|\mathbf{C}'(t)| \\ g''(t) &= 2d|\mathbf{C}''(t)| + |\mathbf{C}'(t)|^2 \\ d &= L(t) - s \end{aligned} \quad (19)$$

那么， t_s 可由下面的迭代公式确定：

$$t_{s,n+1} = t_{s,n} - \frac{g'(t)}{g''(t)} \quad (20)$$

在迭代的过程中， $t_{s,0}$ 可以初始化为 $1.0 \times \frac{s}{L}$ ，这一个值在大部分的情况下，已经足够接近真实值了，并且，我们也无须计算 $s = 0$ 以及 $s = L$ 处的点（根据公式 (14) 及其附近相关的说明）。每次更新 t 后，都需要计算一次 $L(t)$ ，这一个计算可以通过之前提及的基于 Simpson' Rule 的长度积分公式完成，但是这样使得每次都需要进行一次计算量不小的数值积分。

```
const auto df = [&](double t) -> double{ return this->at(t, 1).norm(); };
```

...



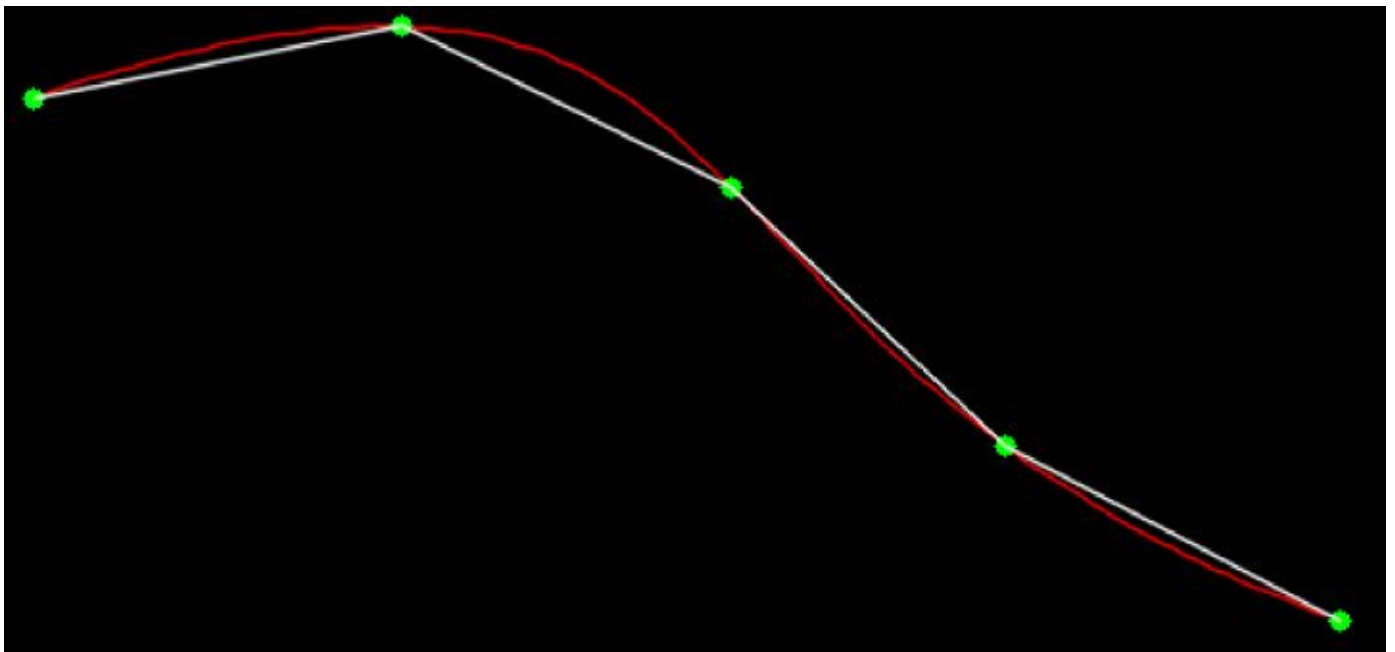
```

for(int iter = 0; iter < max_iter_time; ++iter)
{
    double approx_length = NumericalQuadrature::adaptive_simpson_3_8(df, 0, approx_t);
    double d = approx_length - target_length;
    if (abs(d) < iter_eps) break;

    // Newton's method
    double first_order = this->at(approx_t, 1).norm();
    double second_order = this->at(approx_t, 2).norm();
    double numerator = d * first_order;
    double denominator = d * second_order + first_order * first_order;
    // update
    approx_t = approx_t - numerator / denominator;

    if (abs(approx_t - prev_approx_t) < iter_eps) break;
    else prev_approx_t = approx_t;
}

```



如上图，我们若要均匀采集5个等弧长间距点，则需要进行三轮大（只需计算中间三个）的数值计算，即每一个点都需要用数值积分和牛顿法找到确切点，这样的方法，可以用于单点的精确计算，如果涉及到在曲线上采样等弧长间距的多个点，我们不必每次都使用Simpson's Rule计算每一个采样点的 $L(t)$ ，而是简单地使用采样点与采样点之间的距离（如上图白色直线所示）的累加近似即可，下面给出迭代过程中的简单代码说明：

```

for(int iter = 0; iter < max_iter_time; ++iter)
{
    // 1. 计算上一次迭代确定的 t 参数下，每一个采样点的位置

```



```

for (int j = 1; j < n; j++) dists[j] = (ret[j]-ret[j-1]).norm();

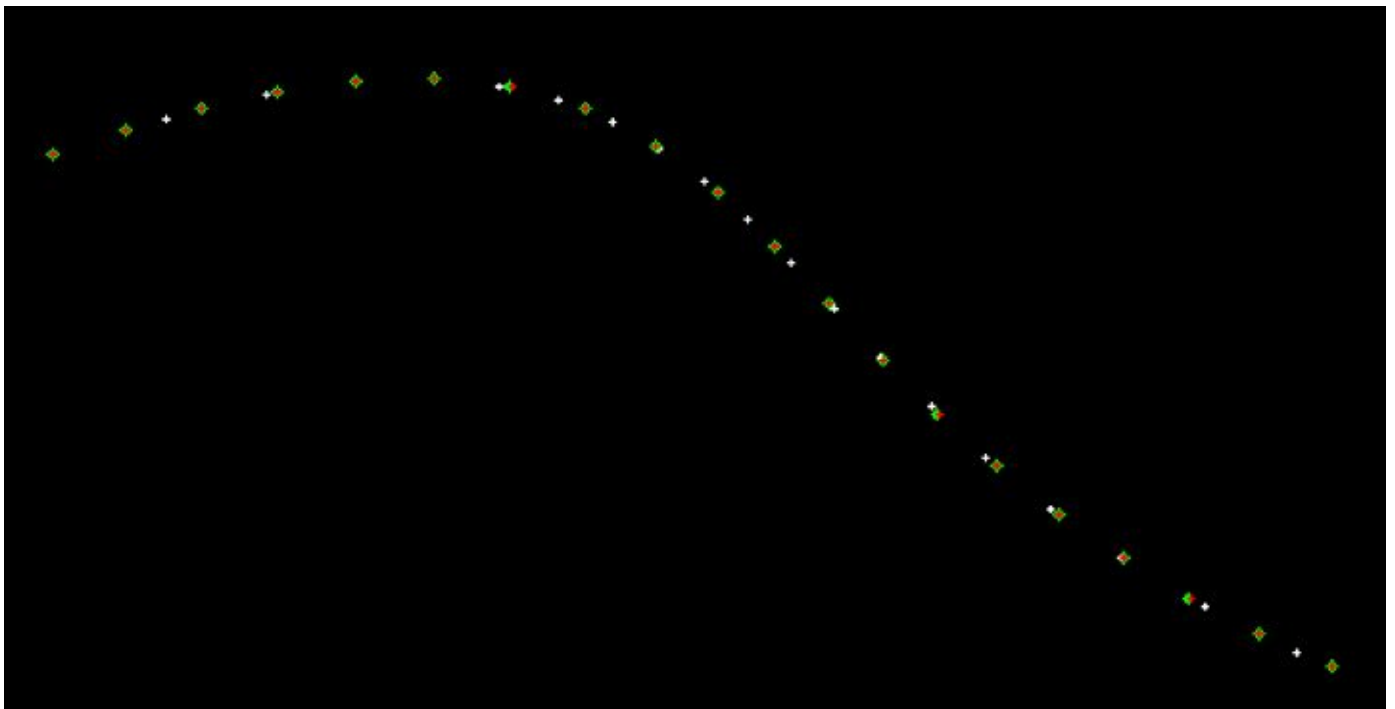
double offset = 0;
for (int j = 1; j < n; j++)
{
    //2. 累计近似弧长并计算误差
    const double err_dist = dists[j] - avg_distance;
    offset += err_dist;

    //3. Newton's method
    double first_order  = this->at(t_array[j], 1).norm();
    double second_order = this->at(t_array[j], 2).norm();
    double numerator    = offset * first_order;
    double denominator  = offset * second_order + first_order * first_order;

    t_array[j] = t_array[j] - numerator / denominator;

    ret[j] = this->at(t_array[j]);
}
}

```



根据以上说明，给出一个简单的例子。上图中，白色点为未参数化时的均 t 间距采样结果，绿色为近似方法的采样结果，而红色为使用Simpson' Rule数值逼近的结果。我们认为Simpson' Rule数值逼近的方法具有最高精度，那么此时可以发现，近似方法大部分接近“真实位置”，而未参数化的点则有较大的“偏差”（本来也肯定不是那个位置）。



4. 多段拟合

多段三次贝塞尔曲线拟合

在这一部分中，简单说明如何进行多段三次贝塞尔曲线的拟合（Piecewise Bezier Curve Fitting）。关于具体如何拟合，请参考《Graphics Gems》中的"Algorithm for Automatically Fitting Digitized Curves"、以及 [github.com/volkerp/fitC...](https://github.com/volkerp/fitCurves/blob/c59eccf26178a42fcb0dfe4e826...) 的实现里的内容。其中大部分涉及到的知识点，如贝塞尔曲线的定义、求导等，均已在以上几个小节说明，其它较为重要的点为：

- 求某一点到贝塞尔曲线的最短距离，对应代码位置为

[https://github.com/volkerp/fitCurves/
blob/c59eccf26178a42fcb0dfe4e826...](https://github.com/volkerp/fitCurves/blob/c59eccf26178a42fcb0dfe4e826...)

github.com



这一部分的算法，与前面的3.3小节比较相似，主要利用“点到曲线的最短距离向量”与“对应曲线位置的切线”垂直这一点构建误差方差，接着使用牛顿法确定参数 t 位置，得到距离残差。

- 构建单条贝塞尔曲线，对应代码位置为

[https://github.com/volkerp/fitCurves/
blob/c59eccf26178a42fcb0dfe4e826...](https://github.com/volkerp/fitCurves/blob/c59eccf26178a42fcb0dfe4e826...)

github.com



这部分比较困难，能看到了这里的朋友应该不难自己看一下原著和代码，此处不多加叙述。

这个文章的涉及到的代码，已经封装成某 `class Beizer` 了，但我绝对不会放出来的。

参考

1. ^ <http://math.aalto.fi/~ahniemi/hss2012/Notes06.pdf>

编辑于 2020-05-15

[贝塞尔曲线](#) [高精度地图](#)

推荐阅读



德州土工膜厂家--官方认证

山东恒阳新材料有限公司



#每日读报60秒#2017.3.2
日读报

大微小观 发表于早

3 条评论

⇌ 切换为时间排序

写下你的评论...



鸭博

檀博乃我辈楷模

1 个月



 赞



yfzhang

16 天前

公式8，最后t那项指数是i-1

 1



iceytan (作者) 回复 yfzhang

16 天前

多谢指正

 赞

