OpenCASCADE Interpolation - Lagrange

eryar@163.com

Abstract. Power basis polynomial is the most simple polynomial function. It also be called power series. OpenCASCADE provides basic computation functions for polynomial functions, such as evaluate the result for a given polynomial, Lagrange interpolation, Hermite interpolation, .etc. The package named PLib, means Polynomial functions Library. The paper focus on the Lagrange interpolation usage of PLib.

Key Words. OpenCASCADE, PLib, Interpolation, Lagrange, 插值

1.Introduction

无穷级数是高等数学的一个重要组成部分，它是表示函数、研究函数性质及进行数值计算的一种工具。由高等数学中的无穷级数的概念可知，函数项级数中简单而常见的一类级数就是各项都是幂函数的函数项级数即所谓的幂级数（Power Series）。因为幂级数的形式简单，易于理解，且可以高效计算曲线上的点及各阶导数，所以在几何造型中经常用幂级数来近似表示曲线曲面。由于将函数展开成幂级数是有条件的，所有并不是所有的曲线曲面都可以用幂级数的多项式来逼近。

对无穷级数概念比较陌生的读者可以把《高等数学》的书找出来翻翻看，重温一下大学的时光。一打开做了笔记有点泛黄的旧书，就会回想起青涩的校园时光。

当时学习《高等数学》的时候感觉很抽象难理解，因为无法将理论与实践联系起来，看不到直观效果，有时也会冒出“学数学有什么用？”这种问题。当你遇到相关的问题再去看国内的教材时，觉得国内的教材写得还是很细致用心的。现在获取信息已经很便利了，如 OpenCASCADE 这个开源的库，其 TKMath 工具箱可以看成是数值计算理论联系实践的一个具体实例。结合 OpenCASCADE 的源码来对相关理论的学习，效率会事半功倍。

本文对幂级数的概念做简单介绍，并结合源程序详细说明 OpenCASCADE 中的 PLib 包中关于幂级数多项式的计算及 Lagrange 插值的用法。

2.Polynomial Evaluation

高等数学的书中把幂级数的重点落在如何将其他的函数展开成幂级数，即用幂级数来逼近函数，而没有介绍如何用数值的方法来对幂级数进行计算。在算法导论一书[2]中找到相关多项式的表示及计算的实现方法，给出了多项式在数据结构上的表示方式及求值算法。对于一个幂级数多项式：

$$C(u)=\sum_{i=0}^{n}a_i u^i$$

多项式的表示有系数表示法和点值表示法。系数表示法（Coefficient Representation）就是将多项式的系数组成一个向量来表示这个多项式。对用系数表示法表示的多项式的求值计算可以采用 Horner 法则，也是是著名的秦九韶算法。此算法的实现代码在《The NURBS Book》[3]一书中给出了，此处略去，只给出 OpenCASCADE 中对幂级数多项式的计算的函数的使用。

```
void testPolynomialEvaluation(void)
{
    // evaluate 1 dimension polynmoial.
    Standard_Real aCoeff[3] = {2.0, 2.0, 3.0};

    Standard_Real aResult = 0.0;

    for (int i = 0; i < 3; ++i)
    {
        PLib::EvalPolynomial(i, 0, 2, 1, aCoeff[0], aResult);

        std::cout << "x=" << i << ", (2.0 + 2.0*x + 3.0*x^2): " << aResult << std::endl;
    }
}
```

从上述代码可以看出，OpenCASCADE 的 PLib 包中对多项式的表示方法是采用的系数表示法。其系数分别为：2.0，2.0，3.0，即表示了幂级数：

$$y=2.0+2.0x+3.0x^2$$

当 x=0, 1,2 时计算结果如下图所示：

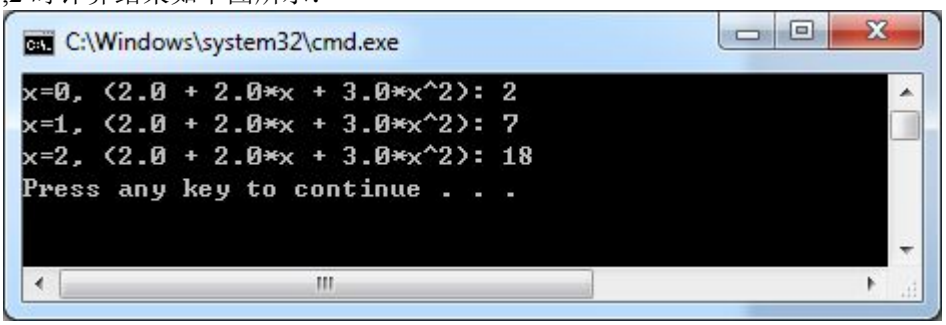


Figure 2.1 Polynomial Evaluation

3.**Polynomial Interpolation**

多项式的插值问题是多项式求值的逆问题。多项式求值问题几何意义就是已知曲线的表达式计算曲线上的点；而插值问题是已经曲线上的一些点来求通过这些点的曲线表达式。多项式插值中最常见最基本的问题是求一次数不超过 n 的代数多项式：

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

使

$$P_n(x_i) = y_i \qquad (i = 0,1,\cdots,n)$$

满足插值条件的多项式称为函数 f(x)在节点 xi 处的 n 次插值多项式。由插值条件可知，插值多项式的系数满足线性方程组：

$$\begin{cases} a_0 + a_1 x_0 + \cdots + a_n {x_0}^n = y_0 \\ a_0 + a_1 x_1 + \cdots + a_n {x_1}^n = y_1 \\ \vdots \\ a_0 + a_1 x_n + \cdots + a_n {x_n}^n = y_n \end{cases}$$

由线性代数可知，其系数行列式是 n+1 阶 Vandermonde 行列式，且：

$$V = \begin{vmatrix} 1 & x_0 & {x_0}^2 & \cdots & {x_0}^n \\ 1 & x_1 & {x_1}^2 & & {x_1}^n \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_n & {x_n}^2 & \cdots & {x_n}^n \end{vmatrix} = \prod_{i=1}^{n} \prod_{j=0}^{i-1} (x_i - x_j)$$

因为插值的点是不同的点，所以行列式 V 不为 0，即线性方程组有唯一解。这也是算法导论一书中这样说“从某种意义上说，多项式的系数表示法与点值表示法是等价的，即用点值形式表示的多项式都对应唯一一个系数形式的多项式”的理论依据。即插值多项式的唯一性。这里不仅指出了插值多项式存在的唯性，而且也提供了一种解法，即通过解线性方程组来确定系数。根据这个思路，给出对上面已知系数求值的多项式进行插值，代码如下所示：

```
void testPolynomialInterpolation(void)
{
    // given three points: (0, 2), (1, 7), (2, 18) to interpolate a polynomial
    math_Matrix A(1, 3, 1, 3, 0.0);
    math_Vector B(1, 3);
    math_Vector X(1, 3);

    A(1, 1) = 1.0; A(1, 2) = 0.0; A(1, 3) = 0.0; B(1) = 2.0;
    A(2, 1) = 1.0; A(2, 2) = 1.0; A(2, 3) = 1.0; B(2) = 7.0;
    A(3, 1) = 1.0; A(3, 2) = 2.0; A(3, 3) = 4.0; B(3) = 18.0;

    // solve functions: Ax = B
    math_Gauss aSolver(A);
    aSolver.Solve(B, X);
```

```
    if (aSolver.IsDone())
    {
        std::cout << X << std::endl;
    }
}
```

已知多项式通过三个点（0, 2），（1, 7），（2, 18），求通过这三个点的多项式表达式。根据插值条件列出线性方程组如下：

$$A=\begin{Bmatrix} a_0+a_1x_0+a_2x_0^{\ 2} \\ a_0+a_1x_1+a_2x_1^{\ 2} \\ a_0+a_1x_2+a_2x_2^{\ 2} \end{Bmatrix}=\begin{Bmatrix} a_0+0+0 \\ a_0+a_1+a_2 \\ a_0+2a_1+4a_2 \end{Bmatrix}=\begin{Bmatrix} 2 \\ 7 \\ 18 \end{Bmatrix}=B$$

将系数 a0,a1,a2 看成线性方程组的待求变量，使用类 math_Gauss 来对线性方程组进行求解，计算结果如下所示：

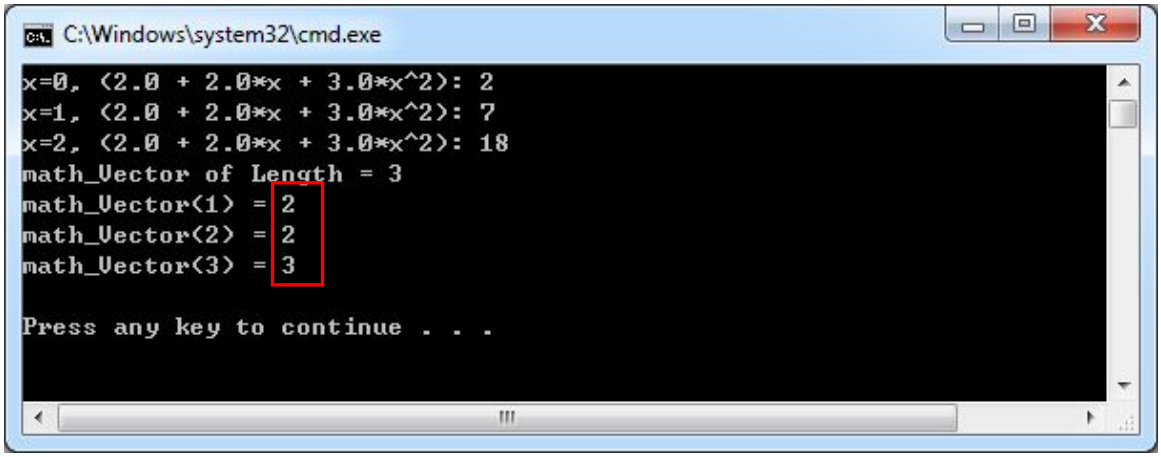
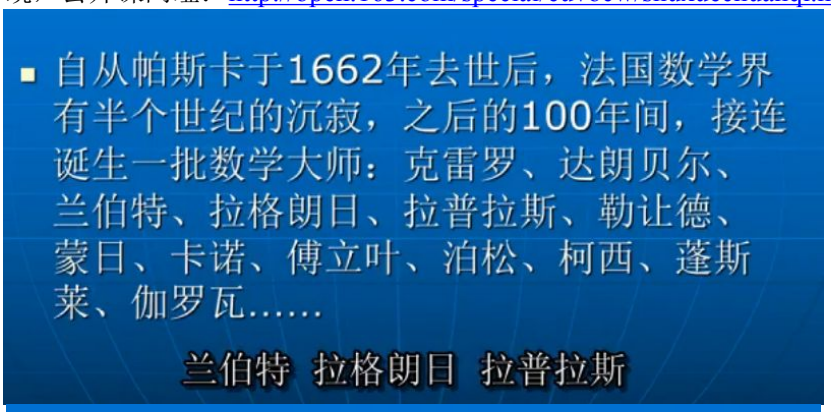


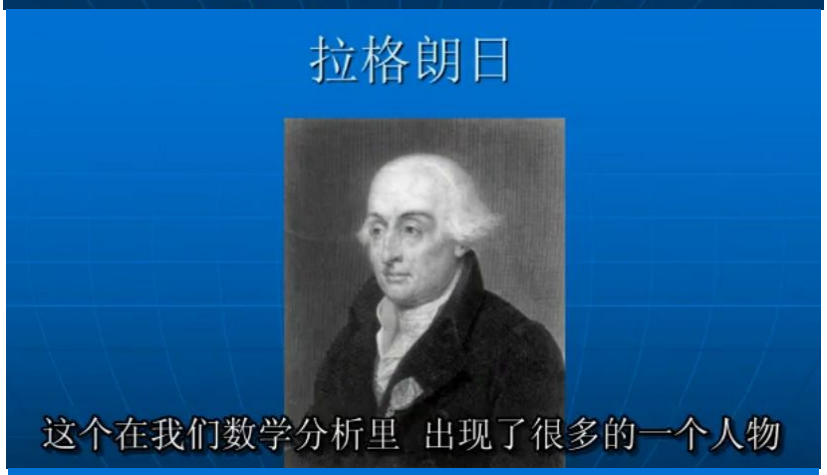
Figure 3.1 Polynomial Interpolation Result

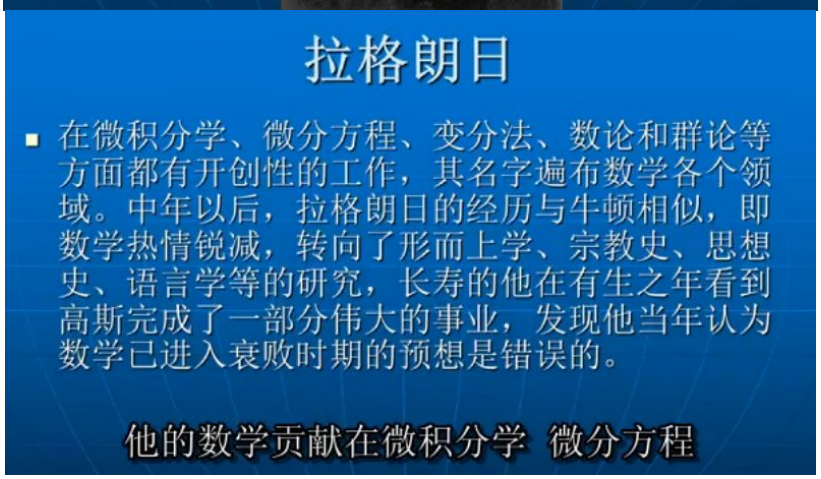
由上图可知，对线性方程组的求解结果与上节点的系数对应。

4.Lagrange Interpolation

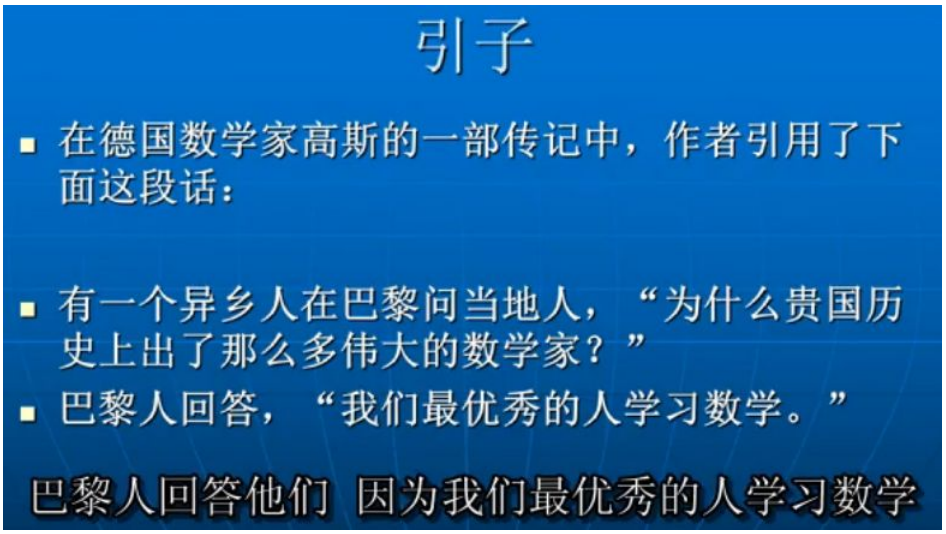
在《计算方法》、《数值逼近》等书中看到 Lagrange 的名字，就想到《高等数学》书中很多与之相关的定理、定义等，如：Lagrange 中值定理、Lagrange 型余项、条件极值的 Lagrange 乘数法等等。在网上搜索了下 Lagrange，原来他也是 OpenCASCADE 的发源地的人：法国人。Joseph-Louis Lagrange，法国著名数学家、物理学家。1736 年 1 月 25 日生于意大利都灵，1813 年 4 月 10 日卒于法国巴黎。他在数学、力学和天文学三个学科领域都有历史性的贡献，其中尤以数学方面的成就最为突出。以下图片来自网易公开课《数学传奇》：从笛卡尔到庞加莱—法国数学的人文传统，公开课网址：http://open.163.com/special/cuvocw/shuxuechuanqi.html







为什么优雅浪漫的法国人的数学大师层出不穷呢？因为他们最优秀的人在学习数学。数学已经成为法国人传统文化中最优秀的一部分了。



数学也是中国传统文化的一部分，中国古代数学成就也很多：《周髀算经》； 《九章算术》（三国时刘徽著）； 祖冲之； 算盘。天文学：天象观察记录， 发明观测仪器：圭表；浑仪；简仪；高表；仰仪，制定历法(农历)。想想后来为什么没有大的发展，原因可能是科举制度造成的，考试的内容偏文。

Lagrange 父亲是法国陆军骑兵里的一名军官，后由于经商破产，家道中落。据 Lagrange 本人回忆，如果幼年时家境富裕，他也就不会作数学研究了。经历了挫折之后没被打倒的人后期成就会更大，像《红楼梦》的作者曹雪芹。严重跑题了，回到 Lagrange 插值问题上来。从 Lagrange 中值定理的证明及 Lagrange 乘数法求极值的方式中可以看出 Lagrange 有个特点，那就是喜欢引入辅助函数来解决问题，可以看出 Lagrange 是非常精明的。对多项式插值也不例外，通过构造了一个 Lagrange 插值基函数来简化多项式的插值，如下公式为 Lagrange 插值基函数：

$$l_i(x)=\frac{(x-x_0)(x-x_1)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_n)}{(x_i-x_0)(x_i-x_1)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_n)}$$

设：

$$\omega_n(x)=(x-x_0)(x-x_1)\cdots(x-x_n)$$

则 Lagrange 插值基函数可以表示为简洁的形式：

$$l_i(x)=\frac{\omega_n(x)}{(x-x_i)\omega'_n(x_i)}$$

则 n 次多项式

$$p(x)=\sum_{i=0}^{n}y_i l_i(x)$$

满足插值条件。OpenCASCADE 的 PLib 包中也提供了 Lagrange 插值的函数来进行多项式插值计算。其用法的代码如下所示：

```
void testLagrangeInterpolation(void)
{
    // given three points: (0,2), (1,7), (2,18) to interpolate a polynomial
    Standard_Real aValues[3] = {2.0, 7.0, 18.0};
    Standard_Real aParameters[3] = {0.0, 1.0, 2.0};
    Standard_Real aResult = 0.0;

    // this do not output the coeff of the interpolate polynomial
    PLib::EvalLagrange(1.5, 0, 2, 1, aValues[0], aParameters[0], aResult);

    std::cout << "Result: " << aResult << std::endl;
}
```

下面对 PLib::EvalLagrange()函数的 7 个参数进行说明：

Parameter	待根据 Lagrange 插值多项式求值的参数
DerivativeRequest	插值多项式导数次数
Degree	插值多项式的次数
Dimension	插值多项式的维次
Values	插值多项式的值
Parameters	插值多项式的参数
Results	参数 Parameter 在 Lagrange 插值多项式中的值

上述代码计算的是这样一个问题，已知 f(0)=2; f(1)=7, f(2)=18，求 f(1.5)的近似值。计算结果如下图所示：

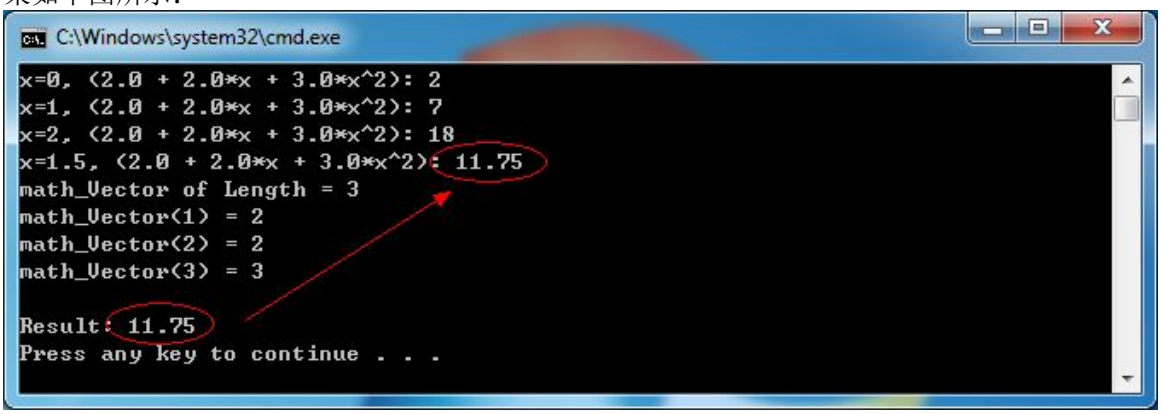


Figure 4.1 Lagrange Interpolation Result

由上图可知，经过 Lagrange 抛物插值得到的结果与直接计算得到的结果吻合。为了更好的说明 OpenCASCADE 中 PLib 包的 Lagrange 插值的用意，下面将《计算方法》[6]中 Lagrange 插值部分中的例题进行计算来结计算结果进行比较。已知 $\sqrt{100}=10,\sqrt{121}=11,\sqrt{144}=12$ 分别用线性插值和抛物插值求 $\sqrt{115}$ 的近似值。相关计算代码如下所示：

```
// sqrt(100)=10, sqrt(121)=11, sqrt(144)=12, evaluate sqrt(115) value.
Standard_Real aSqrtValues[3] = {10.0, 11.0, 12.0};
Standard_Real aSqrtParameters[3] = {100.0, 121.0, 144.0};

// linear interpolation
PLib::EvalLagrange(115.0, 0, 1, 1, aSqrtValues[0], aSqrtParameters[0], aResult);
std::cout << "Linear Interpolate Result: " << aResult << std::endl;
```

```
// Parabolic Interpolation
PLib::EvalLagrange(115.0, 0, 2, 1, aSqrtValues[0], aSqrtParameters[0], aResult);
std::cout << "Parabolic Interpolate Result: " << aResult << std::endl;
```

计算结果如下所示：

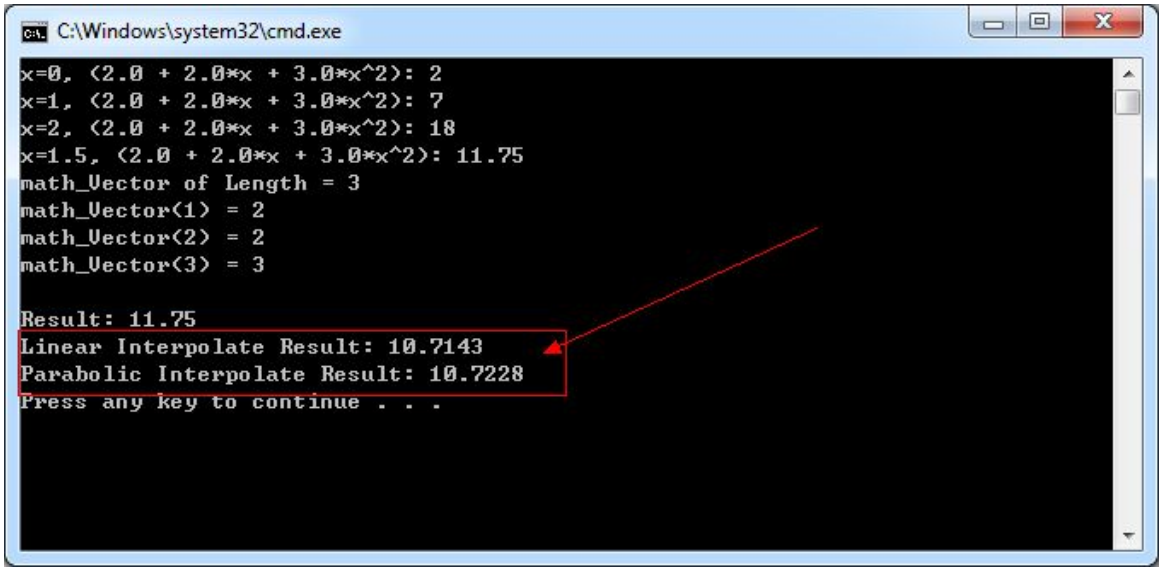


Figure 4.2 Linear Interpolate and Parabolic Interpolate Result

将上图 4.2 的结果与书中的计算结果进行对比发现，下面的结果为《计算方法》书中的结果：

$$\sqrt{115} = L_1(115) \approx 10.714$$
$$\sqrt{115} = L_2(115) \approx 10.723$$

PLib 中 Lagrange 插值结果准确，精度较高。

从上面的结果来看，Lagrange 插值方法比直接解线性方程组的方法要简单，且 Lagrange 插值法比解线性方程组的实现要简单很多，只用一个函数即可。

5.Conclusion

通过将最简单的多项式进行求值，及用求解线性方程组的方法插值和 Lagrange 插值多项式，来学习曲线拟合中要求较高“插值”，因为其要求曲线严格通过插值点。曲线拟合中的逼近就没有这个要求，只是要求曲线与插值点之间的容差尽量小。

通过应用 OpenCASCADE 的 PLib 包中的函数，可以发现对幂级数的多项式的表示法一般会用系数表示法，且都会使用高效的 Horner 法则，也是秦九韶算法。

直接根据定义来插值幂次多项式时，可以使用 Gauss 消元法来求解线性方程组。这种方式计算工作量大，而 Lagrange 插值法结构紧凑，便于编程实现，且代码相对简单。通过对《计算方法》书中例题的计算，来验证 PLib::EvalLagrange()函数的用法及计算结果。

6.References

1. 同济大学数学教研室. 高等数学. 高等教育出版社. 1996
2. 同济大学应用数学系. 线性代数. 高等教育出版社. 2003
3. Thomas H. Cormen. Introduction to Algorithms. The MIT Press. 2001
4. Les Piegl, Wayne Tiller. The NURBS Book. Springer-Verlag. 1995
5. Shing Liu. Polynomial Library in OpenCASCADE. 2013
   http://www.cppblog.com/eryar/archive/2013/05/08/200118.html
6. 易大义,沈云宝,李有法. 计算方法. 浙江大学出版社. 2002
7. 蒋尔雄,赵风光,苏仰锋. 数值逼近. 复旦大学出版社. 2012
8. 王仁宏,李崇君,朱春钢. 计算几何教程. 科学出版社. 2008
9. 蔡天新. 数学传奇. http://open.163.com/special/cuvocw/shuxuechuanqi.html
10. 科学出版社名词室. 新汉英数学词汇. 科学出版社. 2004