**《计算方法》实验报告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学号 | 2xxxxxxxxx | 姓名 | xxx | 班级 | xxxxxx |
| 实验项目名称 | 插值与拟合 | | | | |
| 一、实验名称  **实验一 插值与拟合** | | | | | |
| 1. 实验目的： 2. 明确插值多项式和分段插值多项式各自的优缺点； 3. 编程实现拉格朗日插值算法，分析实验结果体会高次插值产生的龙格现象； 4. 理解最小二乘拟合，并编程实现线性拟合，掌握非线性拟合转化为线性拟合的方法 5. 运用常用的插值和拟合方法解决实际问题。 | | | | | |
| 1. 实验内容及要求 2. 对于   要求选取11个等距插值节点，分别采用拉格朗日插值和分段线性插值，计算x为0.5, 4.5处的函数值并将结果与精确值进行比较。  输入：区间长度，n(即n+1个节点)，预测点  输出：预测点的近似函数值，精确值，及误差  （2）已知用牛顿插值公式求的近似值。  输入：数据点集，预测点。  输出：预测点的近似函数值 | | | | | |
| 1. 实验原理及算法描述   当精确函数 y = f(x) 非常复杂或未知，在一系列节点 x0 … xn 处测得函数值 y0 = f(x0), … yn = f(xn)， 希望由此构造一个简单易算的近似函数 g(x) ≈ f(x)，满足条件g(xi) = f(xi) (i = 0, … n)。这里的 g(x) 称为f(x) 的插值函数，由插值函数可以去近似估计f(x)在一些未知点处的函数值。最常用的插值函数是多项式插值。  所谓多项式插值即求 n 次多项式使得  基于基函数的拉格朗日插值是构造多项式插值最基本方法。也是推导数值微积分和微分方程数值解的公式的理论基础。  拉格朗日插值公式为： 其中  当结点比较多，次数较高的插值多项式往往发生Runge现象，分段低次插值是避免Runge现象的重要手段。分段一次插值将整个区间分段，在每个小区间上，用一次多项式逼近 f (x)，直观上即用折线代替曲线，只要区间足够小就可以保证很好的逼近效果，但曲线缺乏光滑性。 | | | | | |
| 1. 程序代码及实验结果 2. **主程序**   #include <iostream>  #include <vector>  #include <iomanip>  using namespace std;  double f(double x) {    return 1.0 / (1 + x \* x);  }  *//拉格朗日*  double Lagrange(const vector<double>& x, const vector<double>& y, double point) {    double ans = 0.0;    int n = x.size();    for (int i = 0; i < n; ++i) {      double term = y[i];      for (int j = 0; j < n; ++j) {        if (j != i) {          term \*= (point - x[j]) / (x[i] - x[j]);        }      }      ans += term;    }    return ans;  }  *//线性插值*  double Linear(const vector<double>& x, const vector<double>& y, double point) {    int n = x.size();    for (int i = 0; i < n - 1; ++i) {      if (point >= x[i] && point <= x[i + 1]) {        double t = (y[i + 1] - y[i]) / (x[i + 1] - x[i]);        return y[i] + t \* (point - x[i]);      }    }    return 0.0;  }  void exp1() {    double left = -5.0, right = 5.0;    int n = 11;    vector<double> x(n), y(n);    double step = (right - left) / (n - 1);    for (int i = 0; i < n; i++) {      x[i] = left + i \* step;      y[i] = f(x[i]);    }    vector<double> points = { 0.5, 4.5 };    cout << fixed << setprecision(6);    cout << "x\t\t"      << "y(精确)\t\t"      << "y(拉格朗日)\t"      << "y(分段线性)\t"      << "误差(拉)\t"      << "误差(误差)" << endl;    for (const double& point : points) {      double lag = Lagrange(x, y, point);      double lin = Linear(x, y, point);      double real = f(point);      double lagErr = real - lag;      double linErr = real - lin;      cout << point << "\t" << real << "\t" << lag << "\t" << lin << "\t" << lagErr << "\t" << linErr << endl;    }  }  *//牛顿插值法*  double Newton(const vector<double>& x, const vector<double>& y, double point) {    int n = x.size();    vector<vector<double>> divDiff(n, vector<double>(n, 0.0));    for (int i = 0; i < n; ++i) {      divDiff[i][0] = y[i];    }    for (int j = 1; j < n; ++j) {      for (int i = 0; i < n - j; ++i) {        divDiff[i][j] = (divDiff[i + 1][j - 1] - divDiff[i][j - 1]) / (x[i + j] - x[i]);      }    }  *// 计算插值结果*    double ans = divDiff[0][0];    double product = 1.0;    for (int i = 1; i < n; ++i) {      product \*= (point - x[i - 1]);      ans += divDiff[0][i] \* product;    }    return ans;  }  void exp2() {    vector<double> x = { 1, 4, 9 };    vector<double> y = { 1, 2, 3 };    double point = 5.0;    double ans = Newton(x, y, point);    cout << fixed << setprecision(6);    cout << "x = " << point << "处的近似值为: " << ans << endl;  }  int main() {    exp1();    cout << "-----------------------------------" << endl;    exp2();    return 0;  }   1. **Lagrange插值子程序:**   function y=lagr1(x0,y0,x)%x0为插值点的向量,y0为插值点处的函数值向量,x为未知的点..............  实验结果：  ………………. 。如图一所示。    图1  2  运行结果： | | | | | |
| 1. 实验总结 2. 拉格朗日插值在高次插值时同原函数偏差大、存在龙格现象，高次插值多项式不收敛。 3. 龙格现象产生原因：高次插值多项式的拉格朗日基函数在靠近区间边界处的幅值会显著增大。对应的节点权重过大，使得边界处的小误差被放大，从而导致振荡现象，可以通过降低阶数解决 4. 牛顿插值和拉格朗日插值：二者都基于插值多项式理论，目标是通过 n+1 个数据点构造一个 n 次多项式来拟合函数。如果插值节点分布不合理，两种方法都会受到龙格现象的影响 | | | | | |
| 七、教师评语（或成绩）  教师签字 ： | | | | | |

**《计算方法》实验报告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学号 | 2xxxxxxxxx | 姓名 | xxx | 班级 | xxxxxx |
| 实验项目名称 | 数值积分 | | | | |
| 一、实验名称  **实验二 数值积分** | | | | | |
| 二：实验目的：  熟悉复化梯形方法、复化Simpson方法、梯形递推算法、龙贝格算法；  能编程实现复化梯形方法、复化Simpson方法、梯形递推算法、龙贝格算法；  理解并掌握自适应算法和收敛加速算法的基本思想；  分析实验结果体会各种方法的精确度，建立计算机求解定积分问题的感性认识 | | | | | |
| 三：实验内容及要求  （1）设计复化梯形公式求积算法，编制并调试相应的函数子程序  （2）设计复化辛浦生求积算法，编制并调试相应的函数子程序  （3）用龙贝格算法计算  输入：积分区间，误差限  输出：序列Tn，Sn，Cn，Rn及积分结果（参考书本P71的表2-5）  取n=2，4，8，16，精确解为0.9460831 | | | | | |
| 四：实验原理及算法描述  在许多实际问题中，常常需要计算定积分的值。根据微积分学基本定理，若被积函数f(x)在区间[a,b]上连续，只要能找到f(x)的一个原函数F(x)，便可利用牛顿-莱布尼兹公式求得积分值。  但是在实际使用中，往往遇到如下困难，而不能使用牛顿-莱布尼兹公式。   1. 找不到用初等函数表示的原函数 2. 虽然找到了原函数，但因表达式过于复杂而不便计算 3. f(x)是由测量或计算得到的表格函数   由于以上种种困难，有必要研究积分的数值计算问题。  利用插值多项式 则积分转化为，显然易算。称为插值型求积公式。最简单的插值型求积公式是梯形公式和Simpson公式，。当求积结点提供较多，可以分段使用少结点的梯形公式和Simpson公式，并称为复化梯形公式、复化Simpson公式。如步长未知，可以通过误差限的控制用区间逐次分半的策略自动选取步长的方法称自适应算法。梯形递推公式给出了区间分半前后的递推关系。由梯形递推公式求得梯形序列，相邻序列值作线性组合得Simpson序列, Simpson序列作线性组合得柯特斯序列, 柯特斯序列作线性组合的龙贝格序列。若|R2-R1|<*ε*,则输出R2;否则…依此类推。如此加工数据的过程叫龙贝格算法，如下图所示：  图2.1龙贝格算法数据加工流程  复化梯形公式  复化Simpson公式  梯形递推公式  加权平均公式：  龙贝格算法大大加快了误差收敛的速度，由梯形序列O(h2) 提高到龙贝格序列的O(h8) | | | | | |
| 五：程序代码及实验结果   1. **主程序**   #include <iostream>  #include <cmath>  #include <iomanip>  using namespace std;  double f(double x) {    if (x == 0)      return 1;    return sin(x) / x;  }  *//复化梯形公式*  double Trape(double a, double b, double e) {    double h = b - a;    double t1 = (h / 2) \* (f(a) + f(b)), t2;    double t;    double s;    double x;    while (fabs(t2 - t) < e) {      s = 0;      x = a + (h / 2.0);      if (x <= b) {        s = s + f(x);        x = x + h;      }  *// cout << s << endl;*      t2 = (t1 / 2.0) + (h / 2.0) \* s;      t = t1;      t1 = t2;    }    return t2;  }  *//复化Simpson公式*  double Simpson(double a, double b, double e) {  *//Page 65 2-1*    double h = b - a;    double t1 = (h / 6.0) \* (f(a) + f(b));    double t;    double t2;    double s;    double x;    while (fabs(t2 - t) < e) {      s = 0;      x = a + (h / 2.0);      if (x <= b) {        s = s + 2 \* f(x);        x = x + (h / 2);        s = s + 4 \* f(x);        x = x + (h / 2);      }      t2 = (t1 / 6.0) + (h / 6.0) \* s;      t = t1;      t1 = t2;    }    return t2;  }  *//龙贝格算法*  double Romberg(double a, double b, double e) {    double h = b - a;    double t1 = (h / 2) \* (f(a) + f(b)), t2;    double s1, s2;    double c1, c2;    double r1, r2;    int k = 1;    cout << "0" << "\t" << t1 << "\t";    while (fabs(r2 - r1) < e) {      t2 = (h / 2) \* (f(a) + f(b));      cout << endl;      cout << k << "\t";      c1 = c2;      r1 = r2;      auto func = [](double a, double b, int k) ->double {        double h = b - a;        double t1 = (h / 2) \* (f(a) + f(b)), t2;        double t;        double s;        double x;        for (int i = 0;i < k; i++) {          s = 0;          x = a + (h / 2.0);          if (x <= b) {            s = s + f(x);            x = x + h;          }          t2 = (t1 / 2.0) + (h / 2.0) \* s;          t = t1;          t1 = t2;        }        return t2;        };      t2 = func(a, b, k);      cout << t2 << "\t";      s2 = (4 \* t2 - t1) / 3;      cout << s2 << "\t";      if (k == 1) {        ++k;        h /= 2;        t1 = t2;        s1 = s2;        continue;      }      c2 = (16 \* s2 - s1) / 15;      cout << c2 << "\t";      if (k == 2) {        ++k;        h /= 2;        t1 = t2;        s1 = s2;        c1 = c2;        continue;      }      r2 = (64 \* c2 - c1) / 63;      cout << r2 << "\t";      if (k == 3) {        ++k;        h /= 2;        t1 = t2;        s1 = s2;        c1 = c2;        r1 = r2;        continue;      }    }    cout << endl;    return r2;  }  int main() {    cout << "复化梯形公式=" << Trape(0.0, 1.0, 0.000001) << endl;    cout << "复化辛浦生公式=" << Simpson(0.0, 1.0, 0.000001) << endl;    cout << Romberg(0.0, 1.0, 0.000001) << endl;    return 0;  }  运行截图： | | | | | |
| 六：实验总结  实现了复化梯形公式和复化辛普森公式，能够满足大部分数值积分精度的需求。使用Romberg逐层细化积分区间，并通过加权校正误差，提高了计算的精度 | | | | | |
| 七、教师评语（或成绩）  教师签字 ： | | | | | |

**《计算方法》实验报告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学号 | 2xxxxxxxxx | 姓名 | xxx | 班级 | xxxxxx |
| 实验项目名称 | 非线性方程求根迭代法 | | | | |
| 一、实验名称  **实验三 非线性方程求根迭代法** | | | | | |
| 二：实验目的：   * + - 1. 熟悉非线性方程求根简单迭代法，牛顿迭代及牛顿下山法       2. 能编程实现简单迭代法，牛顿迭代及牛顿下山法   认识选择迭代格式的重要性  对迭代速度建立感性的认识；分析实验结果体会初值对迭代的影响 | | | | | |
| 三：实验内容及要求  用牛顿下山法解方程（初值为0.6）  输入：初值，误差限，迭代最大次数，下山最大次数  输出：近似根各步下山因子 | | | | | |
| 四：实验原理及算法描述  求非线性方程组的解是科学计算常遇到的问题，有很多实际背景．各种算法层出不穷，其中迭代是主流算法。只有建立有效的迭代格式，迭代数列才可以收敛于所求的根。因此设计算法之前，对于一般迭代进行收敛性的判断是至关重要的。牛顿法也叫切线法，是迭代算法中典型方法，只要初值选取适当，在单根附近，牛顿法收敛速度很快，初值对于牛顿迭代  至关重要。当初值选取不当可以采用牛顿下山算法进行纠正。  一般迭代：  图3.1一般迭代算法流程图  牛顿公式：  牛顿下山公式：  下山因子  下山条件 | | | | | |
| 五：程序代码及实验结果  代码：  #include <iostream>  #include <cmath>  #include <iomanip>  using namespace std;  *// 定义方程 f(x) 和其导数 f'(x)*  double f(double x) {    return pow(x, 3) - x - 1;  }  double df(double x) {    return 3 \* pow(x, 2) - 1;  }  void solve(double init, double tol, int iter\_times, int max\_steps) {    double x = init; *// 初值*    int iter = 0;    cout << "迭代步骤\t近似根\t\t下山因子" << endl;    while (iter < iter\_times) {      double fx = f(x);      double dfx = df(x);      double delta = -fx / dfx;      double factor = 1.0;      int steps = 0;      while (steps < max\_steps) {        double new\_x = x + factor \* delta;        if (fabs(f(new\_x)) < fabs(fx)) {          x = new\_x;          break;        }        factor /= 2.0; *//下山因子减半*        steps++;      }      cout << iter + 1 << "\t\t" << setprecision(10) << x << "\t\t" << factor << endl;      if (fabs(f(x)) < tol) {        cout << "找到近似根：" << setprecision(10) << x << endl;        return;      }      ++iter;    }  }  int main() {    double init = 0.6f;    double tol;    int iter\_times;    int max\_steps;  *// 输入初值、误差限、最大迭代次数和最大下山次数*    cout << "初值为0.6" << endl;    cout << "输入误差限: ";    cin >> tol;    cout << "输入最大迭代次数: ";    cin >> iter\_times;    cout << "输入最大下山次数: ";    cin >> max\_steps;    solve(init, tol, iter\_times, max\_steps);    return 0;  }  结果： | | | | | |
| 六：实验总结  通过这个实验，我认识到了牛顿下山法在求解非线性方程时的高效，稳健性。为了避免陷入不收敛或震荡的情况，可以通过引入下山因子的方法解决，显著提高算法的稳定性。还发现误差限、最大迭代次数、最大下山次数等参数对结果影响很大，需要谨慎设置 | | | | | |
| 七、教师评语（或成绩）  教师签字 ： | | | | | |

**《计算方法》实验报告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学号 | 2xxxxxxxxx | 姓名 | xxx | 班级 | xxxxxx |
| 实验项目名称 | 求解线性方程组 | | | | |
| 一、实验名称  **实验四 求解线性方程组** | | | | | |
| 二：实验目的：   * 1. 熟悉求解线性方程组的有关理论和方法；   2. 能编程实现雅可比及高斯-塞德尔迭代法、列主元高斯消去法、约当消去，追赶法   3. 通过测试，进一步了解各种方法的优缺点   4. 根据不同类型的方程组，选择合适的数值方法 | | | | | |
| 三：实验内容及要求  （1）用Gauss - Seidel 迭代法求解方程组  输入：系数矩阵A，最大迭代次数N，初始向量，误差限e  输出：解向量  （2）用选主元高斯消去求行列式值  提示：   * + - * 2. 消元结果直接存储在系数矩阵中         3. 当消元过程发生两行对调的情况为偶数次时，行列式值为对角线乘积，否则为对角线乘积的相反数。 | | | | | |
| 四：实验原理及算法描述  无论是三次样条还是拟合问题最终都归结为线性方程组，求解线性方程组在数值分析中非常重要，在工程计算中也不容忽视。  线性方程组大致分迭代法和直接法。只有收敛条件满足时，才可以进行迭代。雅可比及高斯-塞德尔是最基本的两类迭代方法，最大区别是迭代过程中是否引用新值进行剩下的计算。消元是最简单的直接法，并且也十分有效的，列主元高斯消去法对求解一般的线性方程组都适用，同时可以用来求矩阵对应的行列式。约当消去实质是经过初等行变换将系数矩阵化为单位阵，主要用来求矩阵的逆。在使用直接法，要注意从空间、时间两方面对算法进行优化。  高斯-塞德尔迭代：  列主元高斯消去法：列主元  消元 回代  图4.2列主元的高斯消去流程图    图4.1G-S迭代算法流程图 | | | | | |
| 五：程序代码及实验结果  5.1：  #include <iostream>  #include <vector>  #include <cmath>  using namespace std;  vector<double> solve(const vector<vector<double>>& A, const vector<double>& b, int iter\_times, double tol) {    int n = A.size();    vector<double> x(n, 0);    vector<double> xOld(n);    for (int iter = 0; iter < iter\_times; iter++) {      xOld = x;      for (int i = 0; i < n; i++) {        double sigma = 0;        for (int j = 0; j < n; j++) {          if (j != i) sigma += A[i][j] \* x[j];        }        x[i] = (b[i] - sigma) / A[i][i];      }      double t = 0;      for (int i = 0; i < n; i++) {        t = max(t, fabs(x[i] - xOld[i]));      }      if (t < tol) {        cout << "收敛" << endl;        return x;      }    }    cout << "未收敛" << endl;    return x;  }  int main() {    vector<vector<double>> A = {        {10, -1, -2},        {-1, 10, -2},        {-1, -1, 5}    };    vector<double> b = { 7.2, 8.3, 4.2 };    int iter\_times = 100;    double tol = 1e-6;    vector<double> ans = solve(A, b, iter\_times, tol);    cout << "结果为: ";    for (const double num : ans) {      cout << num << " ";    }    return 0;  }  结果：  5.2：  #include <iostream>  #include <vector>  #include <cmath>  using namespace std;  double solve(vector<vector<double>> m) {    int n = m.size();    double ret = 1.0;    int swaps = 0;    for (int i = 0; i < n; ++i) {  *// 选主元*      int maxRow = i;      for (int k = i + 1; k < n; ++k) {        if (fabs(m[k][i]) > fabs(m[maxRow][i])) {          maxRow = k;        }      }  *// 交换行*      if (i != maxRow) {        swap(m[i], m[maxRow]);        swaps++;      }  *// 检查主元是否为 0*      if (fabs(m[i][i]) < 1e-10) {        return 0;  *// 行列式为 0*      }      for (int k = i + 1; k < n; ++k) {        double factor = m[k][i] / m[i][i];        for (int j = i; j < n; ++j) {          m[k][j] -= factor \* m[i][j];        }      }    }  *// 计算行列式*    for (int i = 0; i < n; i++) {      ret \*= m[i][i];    }    return (swaps % 2 == 0) ? ret : -ret;  }  int main() {    vector<vector<double>> m = {        {3, -2, 1, 4},        {-7, 5, -3, -6},        {2, 1, -1, 3},        {4, -3, 2, 8}    };    cout << "答案为: " << solve(m) << endl;    return 0;  }  运行结果： | | | | | |
| 六：实验总结  如果矩阵是稀疏矩阵且规模较大，可以选择迭代方法，能减少计算量和存储需求；如果矩阵是稠密矩阵且规模较小，可使用直接方法（如高斯消去法），快速得到精确解 | | | | | |
| 七、教师评语（或成绩）  教师签字 ： | | | | | |

**《计算方法》实验报告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学号 | 2xxxxxxxxx | 姓名 | xxx | 班级 | xxxxxx |
| 实验项目名称 | 数值积分 | | | | |
| 一、实验名称  **实验五 数值积分** | | | | | |
| 二：实验目的：   1. 熟悉数值微分中Euler法，改进Euler法，Rung-Kutta方法； 2. 能编程实现Euler法，改进Euler法，Rung-Kutta方法； 3. 通过实验结果分析各个算法的优缺点; 4. 明确步长对算法的影响并理解变步长的Rung-Kutta方法 | | | | | |
| 三：实验内容及要求  （1） 0 < x<1  取h=0.1时用Euler法，改进Euler法，Rung-Kutta方法求其数值解并与精确解进行比较。  输入：求解区间，初值，数值解个数  输出：数值解 | | | | | |
| 四：实验原理及算法描述  在许多科学技术问题中，建立的模型常常以常微分方程的形式表示。然而，除了少数特殊类型的常微分方程能用解析方法求其精确解外，要给出一般方程解析解的表达式是困难的。所以只能用近似方法求其数值解，在实际工作中常用计算机求常微分方程的数值解。所谓常微分方程的数值解即对于常微分方程初值问题计算出在一系列节点 a = x0< x1<…< xn= b 处的未知函数 y(x)近似值y0,y1,…yn，即找到一系列离散点（x0,y0）（x1,y1）（x2,y2）…（xn,yn）近似满足常微分方程。数值解法的基本思想用差商代替导数,实现连续问题离散化，选取不同的差商代替导数可以得到不同公式。  改进欧拉公式是常用方法之一，包括预测和校正两步。先用欧拉公式进行预报，再将预报值代入梯形公式进行校正，从而达到二阶精度。通过龙格-库塔法我们可以获得更高精度。经典龙格-库塔法即在区间[xn,xn+1]取四点，并对这四点的斜率进行加权平均作为平均斜率，通过泰勒公式寻找使局部截断误差为O(h5)（即4阶精度）的参数满足条件。  改进的欧拉公式： 预测  校正  四阶（经典）龙格-库塔公式 | | | | | |
| 五：程序代码及实验结果  #include <iostream>  #include <iomanip>  #include <cmath>  using namespace std;  double f(double x, double y) {    return y - (2 \* x) / y;  }  double Exact(double x) {    return sqrt(1 + 2 \* x);  }  void Euler(double x0, double y0, double h, int n) {    cout << "Euler法:" << endl;    cout << setw(10) << "x" << setw(15) << "y (计算值)" << setw(15) << "y (精确值)" << setw(15) << "误差" << endl;    for (int i = 0; i <= n; i++) {      double exact = Exact(x0);      cout << setw(10) << x0 << setw(15) << y0 << setw(15) << exact << setw(15) << abs(y0 - exact) << endl;      y0 += h \* f(x0, y0);      x0 += h;    }    cout << endl;  }  void Euler\_improved(double x0, double y0, double h, int n) {    cout << "改进Euler法:" << endl;    cout << setw(10) << "x" << setw(15) << "y (计算值)" << setw(15) << "y (精确值)" << setw(15) << "误差" << endl;    for (int i = 0; i <= n; i++) {      double exact = Exact(x0);      cout << setw(10) << x0 << setw(15) << y0 << setw(15) << exact << setw(15) << abs(y0 - exact) << endl;      double predict = y0 + h \* f(x0, y0);      y0 += h / 2 \* (f(x0, y0) + f(x0 + h, predict));      x0 += h;    }    cout << endl;  }  void Rung\_Kutta(double x0, double y0, double h, int n) {    cout << "Rung-Kutta方法:" << endl;    cout << setw(10) << "x" << setw(15) << "y (计算值)" << setw(15) << "y (精确值)" << setw(15) << "误差" << endl;    for (int i = 0; i <= n; i++) {      double exact = Exact(x0);      cout << setw(10) << x0 << setw(15) << y0 << setw(15) << exact << setw(15) << abs(y0 - exact) << endl;      double k1 = f(x0, y0);      double k2 = f(x0 + h / 2, y0 + h / 2 \* k1);      double k3 = f(x0 + h / 2, y0 + h / 2 \* k2);      double k4 = f(x0 + h, y0 + h \* k3);      y0 += h / 6 \* (k1 + 2 \* k2 + 2 \* k3 + k4);      x0 += h;    }    cout << endl;  }  int main() {    double x0 = 0.0, y0 = 1.0;    double h = 0.1;    double x\_end = 1.0;    int n = (x\_end - x0) / h;    Euler(x0, y0, h, n);    Euler\_improved(x0, y0, h, n);    Rung\_Kutta(x0, y0, h, n);    return 0;  }  运行结果： | | | | | |
| 六：实验总结  步长控制着每一步的进度，步长越小，算法的精度越高，但计算量也增加。而步长过大会导致结果不准确。因此，步长的选择要根据计算精度和效率进行抉择 | | | | | |
| 七、教师评语（或成绩）  教师签字 ： | | | | | |