样条设计文档

张志心 混合 2106

日期：2024 年 1 月 4 日

1 Spline 基类

```
using NUM = double;
template <class Type = NUM>
class Spline;
```

包含成员如下：

1. `int N,Size`：样条结点数为 Size，坐标从 0 到 N；
2. `std::<vector> X, Y`：输入的样条结点 (X_i, Y_i)；
3. `bool _spline_builded`：表示样条是否建立完成；
4. `std::vector<Poly_n<Type>> splines`：样条，分段多项式形式；
5. `bool _self_checked`：样条输入是否自检完成；
6. `void __self_checked__()`：样条自检函数；
7. `Type operator(const Type &x)`：虚函数，样条求值函数；
8. `Type get_error_i (const Function<Type>& func, vector<Type>& X)const`：对于点列 X，原函数 $func$ 计算无穷范数；
9. `Type get_error_1 (const Function<Type>& func, vector<Type>& X)const`：1 范数；
10. `Type get_error_2 (const Function<Type>& func, vector<Type>& X)const`：2 范数；
11. `Type get_midpoint_error (const Function<Type>& func)const`：对于插值区间中点处的误差向量求无穷范数；
12. `virtual const std::string to_python ()`：虚函数，将样条转化为 python 格式（以分段多项式的形式输出）。

自检函数和构造函数：

```
Spline (const vector<Type> &x, const vector<Type> &y) :
    X(x.size()), Y(y.size()), Size(x.size()), N(x.size() - 1),
_spline_builded(0), splines(x.size() - 1), _self_checked(0) {
  custom_assert(x.size() == y.size(), "Spline: Input X, Y have different length.");
  custom_assert(x.size() >= 2, "Spline: Input shall not be too short.");
      X = x; Y = y;
}
bool _self_checked{};
void __self_check__() {
    custom_assert(is_sorted(X.begin(), X.end()), "Spline: Input points not be sorted by X-
        coordinates.");
    custom_assert(X.end()==unique(X.begin(), X.end()), "Spline: Input points X-coordinates
         duplicate.");
}
```

2 ppForm Spline 类

```
template<class Type, int Order>
class ppForm_Spline : public Spline<Type>
```

包含成员如下：

1. `void buildLinearSpline()`：建立线性样条；
2. `void buildCubicSpline()`：建立三次样条；
3. `void build()`：建立样条接口；

构造方法为直接调用基类构造：

```
ppForm_Spline(const vector<Type> &x, const vector<Type> &y) :
Spline<Type>(x, y) {}
```

求值函数，找到所在的分段多项式，调用多项式求点值：

```
Type operator() (const Type &x) const {
    // find which interval x lies in.
    custom_assert(this->_spline_builded, "ppForm Spline : Spline have not been built yet."
        );
    custom_assert(x >= X[0]-1e-10 && x <= X.back()+1e-10, "ppForm Spline : Input x is out
        of range.");
    auto getValue = [&](int id, const Type& x) -> Type {
        return splines[id](x - X[id]);
    };
if(x >= X[N]) return Y[N];
    return getValue(upper_bound(X.begin(), X.end(), x) - X.begin() - 1, x);
}
```

样条求解部分（目前只实现了 1 次和 3 次）：

```
/******************* begin Linear Spline *******************/
void buildLinearSpline() { // build straightly
    for(int i = 0; i < N; ++i) {
        splines[i] = Poly_n<Type>(vector<Type>({Y[i], (Y[i+1]-Y[i])/(X[i+1]-X[i])}));
    }
}
/******************* end Linear Spline *******************/

/******************* begin Cubic Spline *******************/
void buildCubicSpline(Cubic_Spline_Condition<Type, Nature> cond, Vec<Type> &x,
Vec<Type> &y, Vec<Type> &z, Vec<Type> &b, const vector<vector<Type>>& dd,
const vector<Type> &lambda, const vector<Type> &mu) {
    // 构造三对角矩阵和右端项
    b[0] = 0; b[N] = 0; y[0] = y[N] = 1;
for(int i = 1; i < N; ++i) b[i] = 6*dd[1][i];

for(int i = 1; i < N; ++i) y[i] = 2;
for(int i = 1; i < N; ++i) x[i] = lambda[i]; x[0] = 0;
for(int i = 1; i < N; ++i) z[i-1] = mu[i]; z[N-1] = 0;
}
void buildCubicSpline(Cubic_Spline_Condition<Type, Complete> cond, Vec<Type> &x,
Vec<Type> &y, Vec<Type> &z, Vec<Type> &b, const vector<vector<Type>>& dd,
const vector<Type> &lambda, const vector<Type> &mu) {
```

```
      // 构造三对角矩阵和右端项
      b[0] = 6*(cond.sa - dd[0][0]) / (X[0] - X[1]);
  b[N] = 6*(cond.sb - dd[0][N-1]) / (X[N] - X[N - 1]);
  for(int i = 1; i < N; ++i) b[i] = 6*dd[1][i];

  for(int i = 0; i <= N; ++i) y[i] = 2;
  for(int i = 1; i < N; ++i) x[i] = lambda[i]; x[0] = 1;
  for(int i = 1; i < N; ++i) z[i-1] = mu[i]; z[N-1] = 1;
  }
  void buildCubicSpline(Cubic_Spline_Condition<Type, Second_Derivatives> cond, Vec<Type> &x,
Vec<Type> &y, Vec<Type> &z, Vec<Type> &b, const vector<vector<Type>>& dd,
const vector<Type> &lambda, const vector<Type> &mu) {
      // 构造三对角矩阵和右端项
      b[0] = cond.sa; b[N] = cond.sb; y[0] = y[N] = 1;
  for(int i = 1; i < N; ++i) b[i] = 6*dd[1][i];

  for(int i = 1; i < N; ++i) y[i] = 2;
  for(int i = 1; i < N; ++i) x[i] = lambda[i]; x[0] = 0;
  for(int i = 1; i < N; ++i) z[i-1] = mu[i]; z[N-1] = 0;
  }
  /******************* end Cubic Spline *******************/

  void build() {
      if(this->_spline_builded) { return; }
      if(!this->_self_checked) { this->__self_check__(); this->_self_checked = 1; }
      if(Order == 1) buildLinearSpline();
      else { std::cerr << "ppForm_Spline: build() not implemented.\n"; return; }
      this->_spline_builded = 1;
  }

  template <Cubic_Spline_Type cType>
  void build(Cubic_Spline_Condition<Type, cType> cond) {
      if(this->_spline_builded) { return; }
      if(!this->_self_checked) { this->__self_check__(); this->_self_checked = 1; }
      if(Order == 3) {
    // divided differences calculate
    vector<vector<Type>> dd(2);
    dd[0].resize(N + 1); dd[1].resize(N + 1);
    for(int i = 0; i < N; ++i) dd[0][i] = (Y[i] - Y[i+1]) / (X[i] - X[i+1]);
    for(int i = 1; i < N; ++i) dd[1][i] = (dd[0][i] - dd[0][i-1]) / (X[i+1] - X[i-1]);

    // lambda & mu
    vector<Type> lambda(N), mu(N);
    for(int i = 1; i < N; ++i) lambda[i] = (X[i] - X[i-1]) / (X[i + 1] - X[i - 1]),
                     mu[i] = (X[i+1] - X[i]) / (X[i + 1] - X[i - 1]);


    // construct linear system Ax = b
    Vec<Type> x(N), y(N+1), z(N), b(N+1);
    buildCubicSpline(cond, x, y, z, b, dd, lambda, mu);
    Vec<Type> M = Thomas(y, z, x, b);
    Vec<Type> m(M.Size());

    // calculate m
```

```
    for(int i = 0; i < N; ++i) m[i] = dd[0][i]-(1.0/6)*(M[i+1]+2*M[i])*(X[i+1]-X[i]);
    m[N] = dd[0][N-1]-(1.0/6)*(M[N-1]+2*M[N])*(X[N-1]-X[N]);

    // get ppForm
    splines[0] = Poly_n<Type>(vector<Type>({Y[0], m[0], M[0]/2.0, (M[1]-M[0])/(X[1]-X[0])
        /6.0}));
    for(int i = 1; i < N; ++i) {
            splines[i] = Poly_n<Type>(vector<Type>({Y[i], m[i], M[i]/2.0, (M[i+1]-M[i])/(X[i
                +1]-X[i])/6.0}));
        }
  }
      else { std::cerr << "ppForm_Spline: build() not implemented.\n"; return; }
      this->_spline_builded = 1;
  }
```

3 BForm Spline 类

```
template <class Type, int Order>
class B_Spline : public Spline<Type>
```

包含成员如下：

1. vector<Type> a, t：B 样条系数，B 样条基函数系数和延拓后的所有点（共 $N+Order+Order+1$ 个）。
2. int offset：真实坐标为 $[-offset, N]$；
3. bool _calculate_ppForm：是否已经将样条转化为分段多项式形式保存在 splines 中；
4. bool _B_Spline_Base_build：是否已经建立完成所有 B 样条基函数；
5. Type T(int i)const：返回真实的 $t_i, i = -offset+1, \cdots, N$；
6. class B_Spline_Base：B 样条基函数类，支持加，乘多项式，数乘，求高阶导等运算，本质是维护相关分段多项式；
7. vector<vector<B_Spline_Base>>：B 样条基函数。
8. void build_B_Spline()：递推建立样条基函数；
9. const B_Spline_Base get_B_Spline(int I, int n)const：返回真实下标的 $B^n_{t_I}$；
10. void build()：建立 B 样条函数。

构造函数：

```
B_Spline (const vector<Type> &x, const vector<Type> &y) : Spline<Type>(x, y),
    _calculate_ppForm(0), _B_Spline_Base_build(0) {
  this->__self_check__();
  this->_self_checked = 1;
  Type d = 0; // calculate difference
  for(int i = 0; i < N; ++i) d += X[i+1] - X[i];
  d /= N;
  offset = Order;
  for(int i = offset; i; --i) t.push_back(X[0] - i*d);
  for(int i = 0; i <= N; ++i) t.push_back(X[i]);
  for(int i = 1; i <= offset; ++i) t.push_back(X[N] + i*d);
  build_B_Spline();
}
```

求值函数，找到支撑集包含该点的所有基函数，求值并相加：

```
  Type operator() (const Type &x) const {
  custom_assert(this->_spline_builded, "B Spline : Spline have not been built yet.");
      custom_assert(x >= X[0] - 1e-10 && x <= X.back() + 1e-10, "B Spline : Input x is out
          of range.");
  if(x >= X.back()) return Y.back();
  if(x <= X[0]) return Y[0];
  Type ret = 0;
  int pos = upper_bound(t.begin(), t.end(), x) - t.begin() - 1 - offset;
  for(int i = max(0, pos); i <= pos + Order && i < N + Order; ++i)
    ret += a[i] * get_B_Spline(i-offset+1, Order)(pos, x);
  return ret;
}
```

B 样条基函数的递推计算:

```
  void build_B_Spline() {
  B_Spline_Base_store.resize(N+offset+offset+1);
  for(int i = -offset+1; i <= N+offset; ++i) {
    B_Spline_Base_store[i+offset].resize(Order+1);
    B_Spline_Base_store[i+offset][0] = B_Spline_Base(i);
  }
  for(int j = 1; j <= Order; ++j)
  for(int i = -offset+1; i <= N+offset-j; ++i) {
    B_Spline_Base_store[i+offset][j] =
      B_Spline_Base_store[i+offset][j-1] * (Poly_n<Type>(vector<Type>{-T(i-1), 1}) / (T(i+j
          -1)-T(i-1)))
      + B_Spline_Base_store[i+1+offset][j-1] * (Poly_n<Type>(vector<Type>{T(i+j), -1}) / (T(
          i+j)-T(i)));
  }
  _B_Spline_Base_build = 1;
}
```

B 样条求解过程:

```
  void build(vector<tuple<bool, int, Type>> bc= vector<tuple<bool, int, Type>>(0)) {
      if(this->_spline_builded) { return; }
      if(!this->_self_checked) { this->__self_check__(); this->_self_checked = 1; }
  if(bc.size() < Order - 1) {
    std::cerr << "B Spline : Too few extra conditions.\n";
    return;
  }
  sort(bc.begin(), bc.end());

  Mat<Type> A(N+Order, N+Order); Vec<Type> b(N+Order);

  for(int i = 0; i <= N; ++i) b[i] = Y[i];

      // 点值条件
  for(int j = 0; j <= N; ++j){
    for(int i = j-offset+1; i <= j; ++i) {
      A[j][i+offset-1] += get_B_Spline(i, Order)(j, T(j));
    }
  }

      // 额外边值条件
```

```cpp
  for(int i = 0, j = N+1; i < Order-1; ++i, ++j) {
    if(i < Order-2 && get<0>(bc[i]) == get<0>(bc[i+1]) &&
      get<1>(bc[i]) == get<1>(bc[i+1])) {
        std::cerr << "B Spline : extra condition not satisfiable.\n";
        return;
      }
    else if (get<1>(bc[i]) <= 0 && get<1>(bc[i]) > Order){
      std::cerr << "B Spline : extra condition not satisfiable.\n";
      return;
    }
    else {
      b[j]= get<2>(bc[i]);
      int nd = get<1>(bc[i]);
      if(get<0>(bc[i])) { // right bounder
        for(int k = 0; k < Order; ++k)
          A[j][N-k+offset-1] += (get_B_Spline(N-k, Order).Derivative(nd))(N, T(N));
      } else { // left bounder
        for(int k = 0; k < Order; ++k)
          A[j][k] += (get_B_Spline(k-Order+1, Order).Derivative(nd))(0, T(0));
      }
    }
  }

      // 利用 linear.hpp 中的高斯消元函数
  a = Gauss_elimination(A, b).val;

      this->_spline_builded = 1;
  }
```

转化为 ppForm 形式：

```cpp
  B_Spline_Base ret = a[0] * get_B_Spline(-offset+1, Order);
for(int i = -offset+2; i <= N; ++i) {
  ret = ret + a[i+offset-1] * get_B_Spline(i, Order);
}
auto all_splines = ret.f;
for(int i = 0; i < N; ++i) splines[i] = all_splines[i + Order];
_calculate_ppForm = 1;
```

4 Cardinal B Spline 类

```cpp
template <class Type, int Order>
class Cardinal_B_Spline : public Spline<Type>
```

包含成员如下：

1. `vector<Type> a`：B 样条系数。
2. `Type d, L, R`：样条左右边界，结点间隔；
3. `bool _calculate_ppForm`：是否已经将样条转化为分段多项式形式保存在 `splines` 中；
4. `void buildLinearSpline()`：建立线性样条；
5. `void buildCubicSpline(...)`：建立三次样条；
6. `void build()`：样条建立接口；

7. class Cardinal_B_Spline_Base：B 样条基函数类，支持加，乘多项式，数乘，平移变换，伸缩变换等运算，本质是维护相关分段多项式；
8. vector<Cardinal_B_Spline_Base>：B 样条基函数。
9. const Cardinal_B_Spline_Base get_B_Spline(int i, int n)const：返回真实下标的 $B^n_{t_i}$；

构造函数：

```
Cardinal_B_Spline(const Type &_L, const Type& _R, const int &_n,
const vector<Type>& y) :L(_L),R(_R),d((_R-_L)/(_n-1)),
Spline<Type>(evenspace<Type>(_L, _R, _n), y), _calculate_ppForm(0), a(_n-1+Order){}
// evenspace 用于得到等间隔点列
```

求值函数：

```
Type operator() (const Type &x) const {
custom_assert(this->_spline_builded, "Cardinal B Spline : Spline have not been built yet."
    );
    custom_assert(x >= X[0]-1e-10 && x <= X.back()+1e-10, "Cardinal B Spline : Input x is
        out of range.");
Type ret = 0;
int pos = (int)((x - L) / d);

for(int j = max(-Order+1, pos-Order+1); j <= min(N, pos+1); ++j) {
  ret += a[j+Order-1]*get_B_Spline(j, Order)((x-L)/d);
}
return ret;
}
```

Cardinal B 样条基函数计算，因为具有平移不变性，所以不需要多每个下标都求解。

```
const Cardinal_B_Spline_Base get_B_Spline(int i, int n) const {
static vector<Cardinal_B_Spline_Base> Cardinal_B_Spline_Base_store = {
    Cardinal_B_Spline_Base(0)};
if (n < Cardinal_B_Spline_Base_store.size()) return Cardinal_B_Spline_Base_store[n].shift(
    i);
for(int nn = Cardinal_B_Spline_Base_store.size(); nn <= n; ++nn) {
  Cardinal_B_Spline_Base_store.push_back(
    Cardinal_B_Spline_Base_store[nn-1] * (Poly_n<Type>(vector<Type>({1./nn,1./nn}))) +
    Cardinal_B_Spline_Base_store[nn-1].shift(1)*(Poly_n<Type>(vector<Type>({1,-1./nn}))));
}
return Cardinal_B_Spline_Base_store[n].shift(i);
}
```

样条建立过程：

```
/******************* begin Linear Spline *******************/
void buildLinearSpline() { // build straightly
    for(int i = 0; i <= N; ++i) a[i] = Y[i];
}
/******************* end Linear Spline *******************/

/******************* begin Cubic Spline *******************/
void buildCubicSpline(Cubic_Spline_Condition<Type, Nature> cond) {
Vec<Type> x(N), y(N+1), z(N), b(N+1);
    for(int i = 1; i < N; ++i) x[i] = 1; x[0] = 0;
for(int i = 0; i < N-1; ++i) z[i] = 1; z[N-1] = 0;
```

```
y[0] = y[N] = 6; for(int i = 1; i < N; ++i) y[i] = 4;
for(int i = 0; i <= N; ++i) b[i] = 6*Y[i];
auto _a = Thomas(y, z, x, b).val;
for(int i = 1; i <= N+1; ++i) a[i] = _a[i-1];
a[0] = 2*a[1]-a[2]; a[N+2] = 2*a[N+1]-a[N];
}
void buildCubicSpline(Cubic_Spline_Condition<Type, Complete> cond) {
Vec<Type> x(N), y(N+1), z(N), b(N+1);
    for(int i = 0; i < N; ++i) x[i] = 1;
for(int i = 0; i < N; ++i) z[i] = 1;
y[0] = y[N] = 2; for(int i = 1; i < N; ++i) y[i] = 4;
for(int i = 1; i < N; ++i) b[i] = 6*Y[i];
b[0] = 3*Y[0] + 1/(d)*cond.sa;
b[N] = 3*Y[N] - 1/(d)*cond.sb;
auto _a = Thomas(y, z, x, b).val;
for(int i = 1; i <= N+1; ++i) a[i] = _a[i-1];
a[0] = a[2] - 2/d*cond.sa;
a[N+2] = a[N] + 2/d*cond.sb;
}
void buildCubicSpline(Cubic_Spline_Condition<Type, Second_Derivatives> cond) {
Vec<Type> x(N), y(N+1), z(N), b(N+1);
    for(int i = 1; i < N; ++i) x[i] = 1; x[0] = 0;
for(int i = 0; i < N-1; ++i) z[i] = 1; z[N-1] = 0;
y[0] = y[N] = 6; for(int i = 1; i < N; ++i) y[i] = 4;
for(int i = 0; i <= N; ++i) b[i] = 6*Y[i];
b[0] -= 1/(d*d)*cond.sa;
b[N] -= 1/(d*d)*cond.sb;
auto _a = Thomas(y, z, x, b).val;
for(int i = 1; i <= N+1; ++i) a[i] = _a[i-1];
a[0] = 2*a[1]-a[2]+1/(d*d)*cond.sa; a[N+2] = 2*a[N+1]-a[N]+1/(d*d)*cond.sb;
}
/******************* end Cubic Spline *******************/

void build() {
    if(this->_spline_builded) { return; }
    if(!this->_self_checked) { this->__self_check__(); this->_self_checked = 1; }
    if(Order == 1) buildLinearSpline();
    else { std::cerr << "B Spline: build() not implemented.\n"; return; }
    this->_spline_builded = 1;
}

template <Cubic_Spline_Type cType>
void build(Cubic_Spline_Condition<Type, cType> cond) {
    if(this->_spline_builded) { return; }
    if(!this->_self_checked) { this->__self_check__(); this->_self_checked = 1; }
    if(Order == 3) {
  // construct linear system Ax = b
  buildCubicSpline(cond);
    }
    else { std::cerr << "Cardinal_B_Spline: build() not implemented.\n"; return; }
    this->_spline_builded = 1;
}
```

转化为 ppForm 格式:

```
    Cardinal_B_Spline_Base ret = a[0] * get_B_Spline(-Order+1, Order);
    for(int i = -Order+2; i <= N; ++i) {
        ret = ret + a[i+Order-1] * get_B_Spline(i, Order);
    }
    auto all_splines = ret.changescale(d).shift(L);
    for(int i = 0; i < N; ++i) splines[i] = all_splines.f[i + Order];
    _calculate_ppForm = 1;
```

5 Quadratic Cardinal Spline 类

由于 Quadratic Cardinal Spline 的插值逻辑与别的样条很不一样，因此考虑单独设计。

```
    template <class Type>
    class Quadratic_Cardinal_B_Spline : public Spline<Type>
```

包含成员如下：

1. `int L, R, n`：样条左右端点，左右端点距离；
2. `vector<Type> a`：B 样条系数；
3. `Type B(int i, Type x)`：得到 $B_i^2(x)$；
4. `Poly_n<Type> getB(int seg/*0, 1, 2*/, int i)`：获得 B_i^2 第 seg 段多项式。

构造函数：

```
    Quadratic_Cardinal_B_Spline(const int& x0, const int& _n, const vector<Type>& f, const
        Type& f0, const Type& fn)
: L(x0), R(x0 + _n), n(_n), _calculate_ppForm(0) {
    this->X.resize(n+1);
    for(int i = 0; i <= n; ++i) this->X[i] = L + i;
    Vec<Type> x(n), y(n-1), z(n-1), b(n);
    for (int i = 1; i < n-1; ++i) {
      x[i] = 6; y[i-1] = 1;
      z[i] = 1; b[i] = 8 * f[i];
    }
    x[0] = 5, z[0] = 1, b[0] = 8 * f[0] - 2 * f0;
    x[n-1] = 5, y[n-2] = 1, b[n-1] = 8 * f[n-1] - 2 * fn;
    vector<Type> t = Thomas(x, y, z, b).val;
    a.resize(n+2);
    for (int i = 0; i < n; ++ i) a[i+1] = t[i];
    a[0] = 2 * f0 - a[1];
    a[n+1] = 2 * fn - a[n];
    this->_spline_builded = 1;
}
```

求值函数：

```
    Type operator() (const Type& x) const {
    custom_assert(x >= L-1e-10 && x <= R+1e-10, "Quadratic_Cardinal_B_Spline : Input x out of
        range.");
    int i = floor(x);
    Type res = 0;
    if(i-1 >= L-1 && i-1 <= R) res += a[i-1-L+1] * B(i-1, x);
    if(i   >= L-1 && i   <= R) res += a[i  -L+1] * B(i  , x);
    if(i+1 >= L-1 && i+1 <= R) res += a[i+1-L+1] * B(i+1, x);
    return res;
```

```
}
```

转化为 ppForm 格式：

```
  splines.resize(n);
for(int i = 0; i < n; ++i) for(int j = i; j <= i + 2; ++j) {
  splines[i] += getB(2-j+i, j-1+L) * a[j];
}
_calculate_ppForm = 1;
```

外部接口：

```
  template <class Type>
  Quadratic_Cardinal_B_Spline<Type>
  Quadratic_Cardinal_B_Spline_Interpolation(const Function<Type>& f, const int& l, const int
      & n) {
    vector<Type> y(n);
    for(int i = 0; i < n; ++ i) y[i] = f(l+i+0.5);
    Type f0 = f(l), fn = f(l+n);
    return Quadratic_Cardinal_B_Spline<Type>(l, n, y, f0, fn);
  }
```

6 Curve fitting 类

```
  class Curve_fitting_Order3 { // Here I use B_spline
B_Spline<NUM, 3> x, y; // x 与 y 分别建立样条。
  public:
    Curve_fitting_Order3 (vector<NUM> &_t, vector<NUM> &_x, vector<NUM> &_y) : x(_t, _x), y(
        _t, _y) {}
    void buildNature() {
      x.build({{0, 2, 0}, {1, 2, 0}});
      y.build({{0, 2, 0}, {1, 2, 0}});
    }
    void buildComplete(NUM xa, NUM xb, NUM ya, NUM yb) {
      x.build({{0, 1, xa}, {1, 1, xb}});
      y.build({{0, 1, ya}, {1, 1, yb}});
    }
    void buildSecondDerivatives(NUM xa, NUM xb, NUM ya, NUM yb) {
      x.build({{0, 2, xa}, {1, 2, xb}});
      y.build({{0, 2, ya}, {1, 2, yb}});
    }
    const std::string to_python () {
          return "X = " + x.to_python() + "\nY = " + y.to_python();
      }
  };
```

参考文献

[1] 张庆海. “Notes on Numerical Analysis and Numerical Methods for Differential Equations”. In: (2023).