三维 YinSet 布尔代数程序设计文档

1 数学抽象对象的实现

1.1 class Vec

- **模板:** `template <class T, int Dim>`
  `T` 表示坐标的数据类型, `Dim` 表示空间的维数.
- 表示空间中的向量.
- **成员变量:**
  private:
  (1) `T coord_[Dim];`
  向量的空间坐标.
- **成员函数:**
  public:
  (1) `T& operator[](int i);`
  `const T& operator[](int i) const;`
  空间坐标 `coord_` 的访问函数.
  (2)
  ```
  template <class T2> Vec<T, Dim> operator+(const Vec<T2, Dim>& rhs) const;
  template <class T2> Vec<T, Dim> operator-(const Vec<T2, Dim>& rhs) const;
  template <class T2> Vec<T, Dim> operator*(const Vec<T2, Dim>& rhs) const;
  template <class T2> Vec<T, Dim> operator/(const Vec<T2, Dim>& rhs) const;
  ……
  ```
  向量在空间上的加减乘除.
  (3) `Point<T, Dim - 1> project(int d) const;`
  将点投影到低维空间.
  (4) `int majorDim(int k) const;`
  输出向量变化最大 (`k > 0`) 或者最小 (`k < 0`) 的维度.
- **非成员相关函数:**

(1) `template <class T>`
`T cross(const Vec<T, 2>& va, const Vec<T, 2>& vb);`
`template <class T>`
`Vec<T, 3> cross(const Vec<T, 3>& va, const Vec<T, 3>& vb);`
向量二维和三维的叉乘.

(2) `template <class T, int Dim>`
`T dot(const Vec<T, Dim>& va, const Vec<T, Dim>& vb);`
向量二维和三维的点乘.

1.2 class Point

- **模板:** `template <class T, int Dim>`
`T` 表示坐标的数据类型, `Dim` 表示空间的维数.
- 表示空间中的点.
- **成员变量:**
private:

(1) `T coord_[Dim];`
点的空间坐标.

- **成员函数:**
public:

(1) `T& operator[](int i);`
`const T& operator[](int i) const;`
空间坐标 `coord_` 的访问函数.

(2) `template <class T2> Vec<T, Dim> operator-(const Point<T2, Dim>& rhs) const;`
`template <class T2> Point<T, Dim> operator+(const Vec<T2, Dim>& rhs) const;`
点在空间上的加减.

(3) `Point<T, Dim - 1> project(int d) const;`
将点投影到低维空间.

1.3 class Line

- **模板:** `template <class T, int Dim>`
`T` 表示坐标的数据类型, `Dim` 为空间的维数.

- 表示空间中直线.
- **成员变量:**
  private:
  (1) `Point<T, Dim> fixPoint_;`
  直线上的一点.
  (2) `Vec<T, Dim> direction_;`
  直线的方向.
- **成员函数:**
  public:
  (1) `int majorDim(int k) const;`
  输出直线方向变化最大或最小的维度.
  (2) `bool contain(const Point<T, Dim>& p) const;`
  判断直线与点的包含关系.
  (3) `Line<T, Dim - 1> project(int d) const;`
  输出投影到低纬度的直线.

1.4 class Segment

- **模板:** `template <class T, int Dim>`
  `T` 表示坐标的数据类型. `Dim` 为空间的维数.
- 表示空间中的线段.
- **枚举常量**
  public:
  (1) `enum PointLocationType { Inter = 0, EndPoint = 1, Outer = 2 };`
  表示点和线段的位置关系.
  (2) `enum SegmentIntersectionType { None = 0, InnerSegment = 1, Overlap = 2, EndPoint = 3 };`
  表示线段与直线或线段相交结果类型.
- **成员变量:**
  private:

(1) `Point<T, Dim> vertex_[2];`
线段的两个端点.

- **成员函数:**
public:

(1) `Point<T, Dim>& operator[](int i);`
`const Point<T, Dim>& operator[](int d) const;`
线段端点 `vertex_` 的访问函数.

(2) `int majorDim(int k) const;`
输出线段所在直线的 `majorDim()`.

(3) `Segment<T, Dim - 1> project(int d) const;`
将线段投影到低维空间.

(4) `PointLocationType locatePoint(const Point<T, Dim>& p,`
`int mDim) const;`
输出点和线段之间的位置关系.

(5) `SegmentIntersectionType intersect(const Segment<T, Dim>& seg2,`
`vector<Point<T, Dim>>& result) const;`
线段求交.

(6) `SegmentIntersectionType intersect(Line<T, Dim>& l2,`
`vector<Point<T, Dim>>& result) const;`
线段与直线求交.

1.5 class Plane

- **模板:** `template <class T, int Dim>`
`T` 表示坐标的数据类型, `Dim` 表示空间的维数.

- 表示空间中的二维平面.

- **成员变量:**
private:

(1) `Point<T, Dim> fixPoint_;`
一个平面上的点.

(2) `Vec<T, Dim> normalVec_;`
平面的法向量.

- **成员函数:**
public:

(1) `int majorDim(int k) const;`
输出法向量的 `majorDim()`.

(2) `bool contain(const Point<T, Dim>& p) const;`
判断平面与点的包含关系.

(3) `bool possiblyOverlap(const Plane& plane2) const;`
判断平面是否重合.

(4) `Line<T, 3> intersect(const Plane& plane2) const;`
输出与另一个平面 `plane2` 的交线, 必须已经排除平行的可能.

(5) `Point<T, 3> intersect(const Line<T, 3>& l2) const;`
输出与直线 `l2` 的交点, 必须已经排除平行的可能.

1.6 class Triangle

- **模板:** `template <class T, int Dim>`
`T` 表示坐标的数据类型, `Dim` 表示空间的维数.

- 表示空间中的三角形.

- **枚举常量**
public:

(1) `enum PointLocationType { Inter = 0, OnEdge = 1, OnVertex = 2, Outer = 3 };`
表示点和三角形位置关系.

(2) `enum TriangleIntersectionType { Never = 0, Overlap = 1, PointIntersection = 2, SegmentIntersection = 3 };`
求交结果的类型.

- **成员变量:**
private:

(1) `Point<T, Dim> vertex_[3];`
三角形的顶点.

(2) `Segment<T, Dim> edge_[3];`
三角形的边.

(3) `Plane<T, Dim> plane_;`
三角形所在平面.

- **成员函数:**
  public:

(1) `Point<T, Dim>& vertex(int i);`
`const Point<T, Dim>& vertex(int i) const;`
`Segment<T, Dim>& edge(int i);`
`const Segment<T, Dim>& edge(int i) const;`
三角形顶点, 边 `vertex_`, `edge_` 的访问函数.

(2) `void reverseOrientation();`
`Triangle<T, Dim> getReversedTriangle() const;`
将三角形定向反向.

(3) `auto normalVec() const -> decltype(plane->normalVec_);`
输出三角形所在平面的法向量.

(4) `int edgeID(const Segment<T, Dim>& seg) const;`
输出输入边是三角形的第几条边.

(5) `int majorDim(int k) const;`
输出三角形所在平面的 `majorDim()`.

(6) `Real calculateArea() const;`
输出三角形的面积.

(7) `Triangle<T, Dim - 1> project(int d) const;`
对三角形按某个维度进行投影.

(8) `int barycentric(const Point<T, Dim>& p, Real co[3]) const;`
`Point<T, Dim> barycentric(Real co[3]) const;`
计算给定点和关于三角形的重心坐标的映射.

(9) `PointLocationType locatePoint(const Point<T, Dim>& p) const;`
判断点和三角形位置关系.

(10) `int possiblyIntersect(`
`const Triangle<T, Dim>& tri2) const;`
快速判断两个三角形是否可能相交.

(11) `int intersect(Line<T, Dim>& l,`
`vector<Point<T, Dim>>& result) const;`
计算三角形和直线的求交.

(12) `TriangleIntersectionType intersect(const Triangle<T, Dim>& tri2,`
`vector<Segment<T, Dim>>& result) const;`
计算两个三角形的求交.

1.7 class Rectangle

- **模板:** `template <class T, int Dim>`
`T` 表示坐标的数据类型, `Dim` 表示空间的维数.

- 表示空间中的长方形.

- **枚举常量**
public:

(1) `enum TriangleIntersectionType { Never = 0, Overlap = 1,`
`PointIntersection = 2, SegmentIntersection = 3 };`
与三角形求交结果的类型.

- **成员变量:**
private:

(1) `Point<T, Dim> vertex_[4];`
长方形的 4 个顶点.

(2) `Segment<T, Dim> edge_[4];`
长方形的 4 条边.

(3) `Plane<T, Dim> plane_;`
长方形所在平面.

- **成员函数:**
public:

(1) `bool repOK() const;`
判断 `vertex_` 是一个长方形的四个顶点.

(2) `Point<T, Dim>& vertex(int i);`
`const Point<T, Dim>& vertex(int i) const;`
`Segment<T, Dim>& edge(int i);`
`const Segment<T, Dim>& edge(int i) const;`
`vertex_, edge_` 的访问函数.

(3) `int majorDim(int k) const;`
输出所在平面的 `majorDim()`.

(4) `Rectangle<T, Dim - 1> project(int d) const;`
将长方体投影到低维空间.

(5) `int possiblyIntersect(const Triangle<T, Dim>& tri2) const;`
快速判断与三角形的是否可能相交.

(6) `int intersect(Line<T, Dim>& l,`
`vector<Point<T, Dim>>& result) const;`
计算长方形和直线的相交结果.

(7) `TriangleIntersectionType intersect(const Triangle<T, Dim>& tri,`
`vector<Segment<T, Dim>>& result) const;`
计算长方形与三角形的相交结果.

1.8 class Cuboid

- **模板:** `template <class T>`
`T` 表示坐标的数据类型.
- 表示 $\mathbb{R}^3$ 空间中的表面与坐标平面平行的长方体.
- **成员变量:**
private:

(1) `Point<T, 3> vertex_[8]`
长方体的顶点.

(2) `Rectangle<T, 3> face_[6]`
长方体的表面.

- **成员函数:**
  public:

  (1) `bool repOK() const;`
  判断 `vertex_`, `face_` 表示的是一个表面与坐标平面平行的长方体.

  (2) `Point<T, 3>& vertex(int i);`
  `const Point<T, 3>& vertex(int i) const;`
  `Rectangle<T, 3>& face(int i);`
  `const Rectangle<T, 3>& face(int i) const;`
  顶点和表面 `vertex_`, `face_` 的访问函数.

  (3) `int intersect(const Triangle<T, 3>& tri,`
  `vector<Segment<T, 3>>& result) const;`
  计算长方体和三角形的交线.
  输入: `tri` 求交的三角形.
  输出: 添加交线段到 `result`, 输出 `result.size()`.

  (4) `bool contain(Point<T, 3> p) const;`
  `bool contain(Triangle<T, 3> tri) const;`
  判断长方体与点或三角形 `p`, `tri` 的包含关系.

  (5) `bool partiallyContain(Triangle<T, 3> tri) const;`
  判断长方体是否包含三角形的一部分.

  (6) `void equalOctalDivision(vector<Cuboid<T>>& result) const;`
  沿所有中点将长方体八等分.

1.9 class SurfacePatch

- **模板:** `template <class T>`
  `T` 表示坐标的数据类型.

- 用三角剖分近似的曲面片, 是 $[0,1]^2$ 到 $\mathbb{R}^3$ 空间的一个连续映射.

- **类型别名:**
  public:

  (1) `using TriangleConstIter = vector<Triangle<T, 3>>::const_iterator;`

(2) `using SurfacePatchBoundary = map<Segment<T, 3>, set<TriangleConstIter>, SegmentCompare>;`
曲面片边界. `Segment<T, 3>` 是边界线段, `set<TriangleConstIter>` 是曲面片中以这条线段为边的三角形集合.

- **成员变量:**
private:

(1) `vector<Triangle<T, 3>> vecTriangle_;`
构成曲面片的三角形集.

(2) `SurfacePatchBoundary boundary_;`
曲面片的边界.

- **成员函数:**
public:

(1) `vector<Triangle<T, 3>>& vecTriangle();`
`const vector<Triangle<T, 3>>& vecTriangle() const;`
表示曲面的三角形集合 `vecTriangle_` 的访问函数.

(2) `SurfacePatchBoundary& boundary();`
`const SurfacePatchBoundary& boundary() const;`
曲面片边界 `boundary_` 的访问函数.

(3) `bool repOK() const;`
判断 SurfacePatch 的三角形集合表示的正确性, 如下准则顺序判断

(a) `vecTriangle_` 的三角形两两之间仅相交于 `edge_` 或不相交.

(b) 除 `boundary_` 中的边以外所有边恰好被两个三角形包含且 `edge_` 在两个三角形中定向相反.

输出首个违反的准则, 否则输出-1.

1.10 class GluedSurface

- **模板:** `template <class T>`
`T` 表示坐标的数据类型.

- 表示三维空间中的黏合紧曲面.

- **枚举常量**
  public:

  (1) `enum OrientationType {Negative = -1, Positive = 1 };`
  表示黏合紧曲面的定向.

  (2) `enum PointLocationType {Outer = -1, OnSurface = 0, Inner = 1};`
  表示点和曲面位置关系.

- **成员变量:**
  private:

  (1) `vector<Triangle<T, 3>> vecTriangle_;`
  存储表示黏合紧曲面的所有三角形.

  (2) `OrientationType orientation_;`
  存储黏合紧曲面的类型.

- **成员函数:**
  public:

  (1) `vector<Triangle<T, 3>>& vecTriangle();`
  `const vector<Triangle<T, 3>>& vecTriangle() const`
  表示黏合紧曲面的三角形集合 `vecTriangle_` 的访问函数.

  (2) `OrientationType& orientation();`
  `const OrientationType& orientation() const;`
  黏合紧曲面的定向 `orientation_` 的访问函数.

  (3) `int repOK() const;`
  判断 GluedSurface 的三角形集合表示的正确性, 按如下准则

  (a) 沿非流形点剪开只能得到一个曲面片.

  (b) 曲面片通过 `SurfacePatch::repOK()` 测试.

  (4) `void dispense(vector<Triangle<T, 3>>& aVecTriangle) const;`
  输入: `*this` 黏合紧曲面本身.
  输出: 添加所有 `vecTriangle_` 中三角形到 `aVecTriangle`.

  (5) `OrientationType calculateOrientation() const;`
  计算黏合紧曲面的定向.
  实现: 取表面上任一三角形的内部点, 沿任一方向发射射线, 计算射线与所有三角形

之间的交点, 最远交点与相交三角形法向量法向量点乘小于 0 时内部无界, 大于 0 时内部有界. 若相交于三角形边界或重合时重新选点发射射线.

(6) `PointLocationType locatePoint(Point<T ,3> p) const;`

输入: `*this` 闭合曲面; `p` 需要判断包含关系的点.

输出: 点与曲面的位置关系

实现:

(a) 随机从点 `p` 发射射线 `l`.

(b) 计算 `l` 与表示曲面的所有三角形的交.

(c) 若点在某个三角形上, 输出 0: 在边界上.

(d) 若有交点在三角形边上或重合线段交返回第一步重新发射射线, 若没有进行下一步.

(e) 在所有交点里选择与给定点最近的一个, 将对应的三角形外法向量与射线方向点乘, 大于 0 时在内部, 小于 0 时在外部.

1.11 class YinSetBoundary

- **模板:** `template <int Dim = 3, int Order = 2>`
  `Dim` 是空间的维数. `Order` 是边界表示的精度.
- 构成唯一表示 YinSet 的 GluedSurface 集合.
- **类型别名:**
  public:

  (1) `using IntersectionResult = TriangleIntersector<Real>::IntersectionResult;`

- **嵌套类:**
  private:

  (1) **struct HasseNode**

  – 构建黏合紧曲面的 HasseDiagram 的 Node.

  – **成员变量:**
  public:

  (1) `int depth_;` Node 在图中的深度.

  (2) `int parent_;` Node 的父节点.

  (3) `vector<int> children_;` Node 的子节点.

- **成员变量:**
  private:

  (1) `vector<GluedSurface<Real>> vecGluedSurface_;`
  之间没有恰当交的黏合紧曲面集合.

  (2) `vector<HasseNode> HasseDiagram_;`
  刻画黏合紧曲面关于包含关系构成的偏序集.

  (3) `Real rTiny_;`
  表示 GluedSurface 的所有三角形最长最短边的最大比值.

  (4) `Real minArea_;`
  三角形的最小面积.

- **成员函数:**
  private:

  (1) `void buildHasse();`
  根据黏合紧曲面集构造 Hasse 图.

  public:

  (1) `vector<GluedSurface<Real>>& vecGluedSurface();`
  `const vector<GluedSurface<Real>>& vecGluedSurface() const;`
  返回黏合紧曲面集.

  (2) `int getOrientation(int k) const;`
  返回第 k 个黏合紧曲面的定向.

  (3) `string getHasseFigure() const;`
  输出 Hasse 图.

  (4) `int repOK() const;`
  判断 YinSetBoundary 表示的正确性和鲁棒性, 如下准则顺序判断

  (a) 所有表示边界的三角形满足 `rTiny_`, `minArea_` 条件.

  (b) 每个 `GluedSurface` 的 `repOK()` 测试通过.

  (c) `GluedSurface` 两两之间没有恰当交.

  (d) 直接包含的 `GluedSurface` 的 `orientation_` 相反.

(5) `void intersect(const YinSetBoundary<Dim, Order>& yinSetBoundary, IntersectionResult& intersectionResult0, IntersectionResult& intersectionResult1) const;`
计算两个 YinSet 边界的交线.

(6) `void intersect(const RectDomain<Dim>& grid, IntersectionResult& intersectionResult0, IntersectionResult& intersectionResult1) const;`
输入: `*this` 求交 YinSet 边界; `grid` 求交的控制体网格.
输出: `intersectionResult0, intersectionResult1` 输出分别在 `YinSetBoundary` 和网格上的交线和重合部分.
实现:

(a) 对每个表示 YinSet 的三角形, 计算可能相交的网格平面.

(b) 计算找到的网格平面和三角形的交线段.

(c) 使用网格平面上的二维网格将交线段切为被小长方体网格包含的线段.

(d) 将线段作为结果输出.

(7) `template<class Iter> static void mergeNearbyPoint( vector<Iter>& vecYinSetBoundaryIter, Real lowBound);`
根据给定 lowBound 值调整 `YinSetBoundary` 集合的点坐标, 消除输入数据的误差.

1.12 class YinSet

- **模板:** `template <int Dim = 3, int Order = 2>`
`Dim` 是空间的维数. `Order` 是边界表示的精度.

- 三维空间中的 YinSet.

- **枚举常量**
public:

(1) `enum PointLocationType {Outer = -1, OnBoundary = 0, Inner = 1};`
表示点和 YinSet 位置关系.

- **成员变量:**
private:

(1) `YinSetBoundary boundary_;`
YinSet 边界的黏合紧曲面唯一表示.

(2) `int bettiNumbers_[Dim];`
存储 Betti 数.

- **成员函数:**
public:

(1) `YinSet(const vector<GluedSurface<Real>>& vecGluedSurface);`
给定黏合紧曲面集构造 YinSet.

(2) `YinSet(const string& s, Real lowBound);`
给定 obj 文件路径构造 YinSet.

(3) `const YinSetBoundary& boundary() const;`
YinSet 边界的访问函数.

(4) `int repOK() const;`
验证 YinSet 表示正确性.

(5) `bool isBounded() const;`
返回 YinSet 是否有界.

(6) `int getBettiNumber(int rank) const;`
返回相应的 Betti 数.

(7) `void exportObj(string s, string folder, int precision) const;`
将 YinSet 输出为 obj 文件.

(8) `PointLocationType locatePoint(Point<Real, 3> p) const;`
判断点和 YinSet 的包含关系.

(9) `YinSet<Dim, Order> complement() const;`
YinSet 求补.
实现:

(1) 调用 `CuttingMap.cutYinSetBoundary()` 得到沿边界剪开的曲面片.

(2) 调用 `Triangle::reverseOrientation()` 改变曲面片方向.

(3) 调用 `PastingMap.pasteSurfacePatch()` 粘合得到 YinSet 边界唯一表示.

(4) 计算 `bettiNumbers` 构造 YinSet 输出.

(10) `YinSet<Dim, Order> intersect(const YinSet<Dim, Order>& y2) const;`
YinSet 求交.
实现:

(1) 调用 `CuttingMap.cutYinSetBoundary()` 得到沿边界剪开的曲面片.

(2) 调用 `YinSet::locatePoint()` 标记保留和处理重合的曲面片.

(3) 调用 `PastingMap.pasteSurfacePatch()` 粘合得到 YinSet 边界唯一表示.

(4) 计算 `bettiNumbers` 构造 YinSet 输出.

(11) `YinSet<Dim, Order> union(YinSet<Dim, Order>& y2) const;`
YinSet 求并, 区别于 `intersect()` 保留的曲面片.

(12) `Tensor<YinSet<Dim, Order>, Dim>`
`intersect(RectDomain<Dim> controlVolume) const;`
快速计算网格与 YinSet 在每个控制体内的布尔运算结果.

2 数学抽象操作的实现

2.1 struct Tolerance

- 本程序中所有操作算子的最小空间距离容忍度单例, 默认值为 `1e-12`.

- **成员变量:**
private:

(1) `Real tol;` 最小距离容忍度.

(2) `static Tolerance* pInstance;`

- **成员函数:**
private:

(1) `Tolerance() = default;`
`Tolerance(const Tolerance&) = default;`
`Tolerance& operator=(const Tolerance&) = default;`
限制构造函数.

public:

(1) `static Tolerance* Instance();`
输出单例对象;

(2) `Real getTol() const;`
`void setTol(Real t);`
访问, 设置 `tol`;

2.2 struct PointCompare

- 点在空间中的字典排序.
- **成员函数:**
  public:

(1)
```
template <class T1, class T2, int Dim>
int compare(
const Point<T1, Dim>& lhs, const Point<T2, Dim>& rhs) const;
```
输出 1: lhs $<$ rhs, 0: lhs $==$ rhs, -1: lhs $>$ rhs.

(2)
```
template <class T1, class T2, int Dim>
bool operator()(
const Point<T1, Dim>& lhs, const Point<T2, Dim>& rhs) const;
```
比较算子.

2.3 struct SegmentCompare

- 线段在空间中基于端点的字典排序的排序, 先比较较小端点, 相等再比较另一个端点.
- **成员函数:**
  public:

(1)
```
template <class T1, class T2, int Dim>
int compare(
const Segment<T1, Dim>& lhs, const Segment<T2, Dim>& rhs) const;
```
输出 1: lhs $<$ rhs, 0: lhs $==$ rhs, -1: lhs $>$ rhs.

(2)
```
template <class T1, class T2, int Dim>
bool operator()(
const Segment<T1, Dim>& lhs, const Segment<T2, Dim>& rhs) const;
```
比较算子.

2.4 struct TriangleCompare

- 基于顶点的字典序比较空间中两个三角形.
- **成员函数:**
  public:

  (1) `template <class T1, class T2, int Dim>`
  `int compare(`
  `const Triangle<T1, Dim>& lhs, const Triangle<T2, Dim>& rhs) const;`
  输出 1: lhs < rhs, 0: lhs == rhs, -1: lhs > rhs.

  (2) `template <class T1, class T2, int Dim>`
  `bool operator()(`
  `const Triangle<T1, Dim>& lhs, const Triangle<T2, Dim>& rhs) const;`
  比较算子.

2.5 struct TriangleIntersector

- **模板:** `template <class T>`
  `T` 是点坐标的数据类型.
- 计算三角形集合中三角形的相交结果.
- **类型别名:**
  public:

  (1) `using IntersectionResult =`
  `vector<pair<vector<Segment<T,3>>, vector<pair<int, int>>>>;`
  `vector<Segment<T, 3>>` 存交线段, `vector<pair<int, int>>` 记录重合的三角形.

- **成员函数:**
  public:

  (1) `void work()(`
  `const vector<Triangle<T, 3>>& vecTriangle0,`
  `const vector<Triangle<T, 3>>& vecTriangle1,`
  `IntersectionResult& result0,`
  `IntersectionResult& result1) const = 0;`
  定义三角形求交的接口, 子类中必须重定义的纯虚函数.

2.6 struct DoubleLoopTriangleIntersector

- **模板:** `template <class T>`
  `T` 是点坐标的数据类型.
- 以两两求交的方式对两组三角形计算求交.
- **继承:** `struct DoubleLoopTriangleIntersector : public TriangleIntersector<T>;`
- **类型别名:**
  public:

  (1) `using IntersectionResult = TriangleIntersector<T>::IntersectionResult;`
- **成员函数:**
  public:

  (1)
  ```
  void work(
      const vector<Triangle<T, 3>>& vecTriangle0,
      const vector<Triangle<T, 3>>& vecTriangle1,
      IntersectionResult& result0,
      IntersectionResult& result1) const;
  ```
  对三角形集 `vecTriangle0`, `vecTriangle1` 中三角形两两求交, 输出求交结果.

2.7 struct OctreeTriangleIntersector

- **模板:** `template <class T>`
  `T` 表示坐标的数据类型.
- 使用空间划分法加速布尔运算中的大量三角形求交.
- **继承:**`struct OctreeTriangleIntersector : public TriangleIntersector<T>;`
- **类型别名:**
  public:

  (1) `using IntersectionResult = TriangleIntersector<T>::IntersectionResult;`
- **嵌套类:**
  private:

  (1) **struct OctreeNode**

- **模板:** `template <class T> T` 表示坐标的数据类型.
- 构建空间划分求交中八叉树的节点.
- **成员变量:**
  public:
  (1) `T val_;` 空间划分求交中划分的长方体区域.
  (2) `vector<vector<int>> tris_;` 长方体中相关的三角形.
  (3) `vector<OctreeNode<T>*> child_;` Node 的子节点.

- **成员变量:**
  private:

(1) `int maxLevel_;`
空间划分的最大深度.

- **成员函数:**
  private:

(1)
```
vector<Real> coverDomain(
const vector<Triangle<T, 3>>& vecTriangle0,
const vector<Triangle<T, 3>>& vecTriangle1) const;
```
输出恰当的包含所有三角形的长方体区域.

(2)
```
void dfsConstructTree(OctreeNode<Cuboid<T>>* r,
const vector<Triangle<T, 3>>& vecTriangle0,
const vector<Triangle<T, 3>>& vecTriangle1,
int depth) const;
```
递归构建八叉树. 若没有达到最大深度, 构造子节点, 分配子节点相关的三角形.

(3)
```
OctreeNode<Cuboid<T>>* initOctree(
const vector<Triangle<T, 3>>& vecTriangle0,
const vector<Triangle<T, 3>>& vecTriangle1) const;
```
输入: `vecTriangle0, vecTriangle1` 分别是表示两个 YinSet 的三角形集合; `maxLevel_` 为空间划分树的深度上限.
输出: 调用 `dfsConstructTree()` 递归建立空间八叉树, 输出根节点.

(4) `virtual long calculateNum(OctreeNode<Cuboid<T>>* r) const;`
输出节点 `r` 中需要计算的三角形求交次数.

(5) `int pruneTree(OctreeNode<Cuboid<T>>* r);`
通过 `calculateNum()` 输出求交次数递归剪枝.

(6) `void calculateIntersection(`
`pair<int, int> iA, pair<int, int> iB,`
`const vector<Triangle<T, 3>>& vecTriangle0,`
`const vector<Triangle<T, 3>>& vecTriangle1,`
`IntersectionResult& result0,`
`IntersectionResult& result1);`
三角形求交, 并存储结果到 `result0, result1` 中.

(7) `void callIntersection(OctreeNode<Cuboid<T>>* r,`
`const vector<Triangle<T, 3>>& vecTriangle0,`
`const vector<Triangle<T, 3>>& vecTriangle1,`
`IntersectionResult& result0,`
`IntersectionResult& result1);`
在此节点计算三角形求交, 或子节点中调用 `callIntersection()` 若子节点非空.

public:

(1) `int& maxLevel();`
`int maxLevel() const;`
`maxLevel_` 的访问函数.

(2) `void work(`
`const vector<Triangle<T, 3>>& vecTriangle0,`
`const vector<Triangle<T, 3>>& vecTriangle1,`
`IntersectionResult& result0,`
`IntersectionResult& result1) const;`
输入: `vecTriangle0, vecTriangle1` 是求交的两个三角形集合.
输出: 三角形求交的交线段和重合三角形.
实现:

(1) 调用 `initOctree()` 构造八叉树.

(2) 调用 `pruneTree()` 剪枝.

(3) 调用 `callIntersection()` 计算求交后输出.

2.8 struct Triangulator

- **模板:** `template <class T>`
  `T` 是点坐标的数据类型.
- 将三角形根据交线进行三角剖分.
- **类型别名:**
  public:

  (1) `using IntersectionResult = TriangleIntersector<T>::IntersectionResult;`

  (2) `using PointNeighbors =`
  `map<Point<T, 3>, set<Segment<T, 3>, SegmentCompare>, PointCompare>;`
  点相邻的边.
- **成员函数:**
  private:

  (1) `void separateSegment(`
  `IntersectionResult::iterator aIntersectionResult,`
  `set<Point<T, 3>, PointCompare>& allPoints,`
  `set<Segment<T, 3>, SegmentCompare>& allSegments,`
  `PointNeighbors& pointNeighbors) const;`
  给定一个三角形和它的求交结果, 将所有交线的顶点和三角形顶点存放在 `allPoints` 中, 所有交线和三角形的边在经过 `allPoints` 中的点时断开, 将这些断开后的线存放在 `allSegments` 中. 对于 `allPoints` 中的每个点, 找到 `allSegments` 中以这个点为端点的线, 存储在 `pointNeighbors` 中.

  (2) `void addSegment(`
  `set<Point<T, 3>, PointCompare>& allPoints,`
  `set<Segment<T, 3>, SegmentCompare>& allSegments,`
  `PointNeighbors& pointNeighbors) const;`
  将 `allPoints` 中每两点连成线段, 如果跟 `allSegments` 中的线段重合或有恰当交, 则跳过; 否则添加到 `allSegments`, 并更新 `pointNeighbors`. 此时三角化的三角形的所有边已确定.

  (3) `void generateTriangle(`
  `const Triangle<T, 3>& Tri,`
  `const pair<int, int>& idInput,`

const set<Segment<T, 3>, SegmentCompare>& allSegments,
PointNeighbors& pointNeighbors,
vector<Triangle<T, 3>>& triangulationResult) const;
根据 allSegments 中的线段生成三角形, 这些三角形构成了原三角形关于交线的三角剖分.

(4) void addSegmentToOverlap(
const vector<pair<int, int>>& allOverlapTriangles,
const set<pair<int, int>>& setTriangulatedTriangleID,
const set<Segment<T, 3>, SegmentCompare>& allSegments,
const vector<Triangle<T, 3>>& vecTriangle,
IntersectionResult& intersectionResult) const;
保证重合三角形的三角化相同.

public:

(1) void work(
const vector<Triangle<T, 3>>& vecTriangle0,
const vector<Triangle<T, 3>>& vecTriangle1,
IntersectionResult& intersection0,
IntersectionResult& intersection1,
vector<Triangle<T, 3>>& result0,
vector<Triangle<T, 3>>& result1) const;
输入: 给定三角形集合 vecTriangle0, vecTriangle1; 相应的交线集 intersection0, intersection1.
输出: 三角剖分后的结果 result0, result1.

2.9 struct SurfacePatchFactory

- **模板:** template <class T>
T 表示坐标的数据类型.

- 给用户提供不同的方式生成 SurfacePatch.

- **成员函数:**
public:

(1) `vector<pair<SurfacePatch<T>, int>> createObject(`
`vector<Triangle<T, 3>>& vecTriangle0,`
`vector<Triangle<T, 3>>& vecTriangle1) const;`
生成 YinSet 的边界沿非流形点剪开得到的曲面片.
输入: `vecTriangle0` 和 `vecTriangle1` 包含所有需要粘合的三角形。
输出: 输出曲面片集合和所在 `YinSetBoundary` 的索引.
实现: 粘合所有只被两个三角形包含的边, 若相邻三角形数量大于 2, 保存为包含这些边的三角形为曲面片的 `boundary`.

2.10 struct CuttingMap

- **模板:** `template <class T>`
`T` 表示坐标的数据类型.

- 沿非流形点将 `YinSetBoundary` 分别剪开.

- **成员函数:**
public:

(1) `vector<pair<SurfacePatch<T>, int>>`
`void cutYinSetBoundary(`
`const YinSetBoundary<3, 2>& boundary0,`
`const YinSetBoundary<3, 2>& boundary1) const;`
剪开 `YinSetBoundary`.
实现:

(a) 调用 `TriangleIntersector()` 计算非流形点构成的交线和重合.

(b) 调用 `Triangulator()` 进行三角化, 得到三角形集合.

(c) 调用 `SurfacePatchFactory()` 输出剪开得到的曲面片集合.

(2) `Tensor<vector<pair<SurfacePatch<T>, int>>, 3>`
`cutYinSetBoundaryInControlVolume(`
`const YinSetBoundary<3, 2>& boundary,`
`RectDomain<3> controlVolume) const;`
沿网格与 `YinSetBoundary` 交线和非流形点剪开 `YinSetBoundary` 和网格面, 输出所有控制体包含的曲面片.

2.11 struct PastingMap

- **模板:** `template <class T>`
  `T` 表示坐标的数据类型.

- 沿 `surfacePatch` 边界按好配对粘合.

- **成员函数:**
  private:

  (1) `template<class Iter>`
  `Iter goodPair()(const Triangle<T, 3>& tri, const Segment<T, 3>& edge, const set<Iter>& setNeighboringTriangleIter, int direction) const;`
  输出在给定边和边上相邻的三角形集合中满足好配对的三角形.
  输入: `tri` 给定的三角形; `edge` 三角形的一条边; `setNeighboringTriangleIter` 是所有沿 `edge` 与 `tri` 相邻的三角形的集合; `direction` 好配对旋转的方向.
  输出: 与 `tri` 构成好配对的三角形.
  实现:

  (a) 计算法向量方向.

  (b) 根据法向量夹角计算相邻三角形与三角形 `tri` 之间的夹角.

  (c) 依据 `direction` 输出满足夹角最小或最大的相邻三角形.

  (2) `template<class Iter>`
  `Iter goodPair()(`
  `const SurfacePatch& face.`
  `const Triangle<T, 3>& tri, const Segment<T, 3>& edge,`
  `const set<Iter>& setNeighboringSurfacePatchIter,`
  `int direction) const;`
  曲面片 `face` 沿边 `(tri, edge)` 和集合 `setNeighboringSurfacePatchIter` 中曲面片的好配对粘合.

  (3) `template<class Iter>`
  `void goodPairPaste(`
  `set<Iter>& setSurfacePatchIter, int direction,`
  `vector<set<Iter>>& vecGluedSurfacePatch) const;`
  粘合给定曲面片集合.
  输入: `setSurfacePatchIter` 是需要粘合的曲面片; `direction` 确定粘合方向.

输出: vecGluedSurfacePatch 是好配对粘合的划分结果.
实现: 通过 SurfacePatch 边界上的三角形的好配对粘合.

public:

(1) YinSetBoundary<3, 2> pasteSurfacePatch(
vector<pair<SurfacePatch, int>> vecSurfacePatch);
输入: vecSurfacePatch 包含所有需要粘合的曲面片.
输出: 唯一表示 YinSet 的 YinSetBoundary.
实现:

(a) 正向 SurfacePatch::goodPairPaste() 好配对粘合得到一些曲面片集合构成某个连通分量的边界.

(b) 在每个构成连通分量边界的曲面片集合内, 反向 SurfacePatch::goodPairPaste() 好配对粘合得到黏合紧曲面集合.

(2) Tensor<YinSetBoundary<3, 2>, 3> pasteSurfacePatchInControlVolume(
Tensor<vector<pair<SurfacePatch<T>, int>>, 3>& tensorVecSurfacePatch);
粘合每个控制体中的 SurfacePatch.