User Guide of Cutlib

Library of Cutline Information Generator

Ver. 3.0.0-beta

Functionality Simulation and Information Team
VCAD System Research Program
RIKEN

2-1, Hirosawa, Wako, 351-0198, Japan

`http://vcad-hpsv.riken.jp/`

February 2013

First Edition	version 1.0.0	28 Apr.	2010
	version 2.0.0	10 Nov.	2010
	version 2.0.1	9 May	2011
	version 2.0.2	12 May	2011
	version 2.0.3	21 Apr.	2012
	version 3.0.0-beta	25 Feb.	2013

目次

第 1 章

Cutlib の概要

本ユーザーガイドでは，3 次元直方体領域に存在するポリゴン群に対して，背景にある直交格子 (等間隔，不等間隔，８分木) との交点情報を計算するライブラリについて，その機能と利用方法を説明します．

1.1　概要

本 Cutlib ライブラリは，背景格子とポリゴンの交点情報を計算し，管理する機能を提供します．対象とする背景格子のデータ構造には，直交格子 (等間隔，不等間隔) と Octree の２種類があります．直交格子については，変数配置がセル中心とノード点の両者に対応できます．Octree については，リーフセルに対してのみ計算するインターフェイスと全ノードを対象とするインターフェイスの２種類を提供します．

本ライブラリは，物理現象シミュレーションのためのフレームワーク V–Sphere 内で使用し，ポリゴン管理ライブラリ Polylib と併用することを前提としています．

本ドキュメントの構成は以下のとおりです．この節の残りの部分で，本ライブラリで用いる用語の定義，そして本ライブラリの提供する機能の概略を述べます．続く 2 章で，基本的なデータ構造について説明します．3 章では，交点情報を計算する関数群のインタフェースを説明し，その使用例を示します．4 章では，本ライブラリの性能評価の結果を報告します．5 章ではクラス設計情報について解説します．

1.2　用語の定義

1.2.1　計算基準点，計算基準線分

隣接するセル中心を結ぶ線分，または，隣接するノード間を結ぶ線分を計算基準線分，その両端の点を計算基準点と呼ぶことにします．本ライブラリでは，計算基準点毎に，6 方向 $(\pm x, \pm y, \pm z)$ の計算基準線分を横切るポリゴンの存在を調べます．そしてポリゴンが存在したなら，各方向毎に，最も計算基準点に近い交点を持つポリゴンについての情報 (交点情報) を記録します．したがって一般には，計算基準線分の両端の基準点では，違う交点情報が記録されることになります．各計算基準点毎に，交差したポリゴンが存在しなかったならその情報も含めて，6 組の交点情報が記録されます．

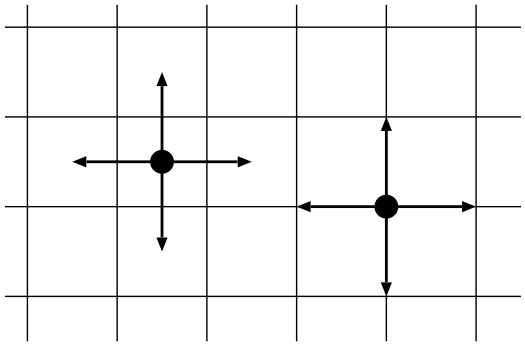

図 1.1　計算基準点 (黒丸) と計算基準線分 (矢印)，左:セル中心間，右:ノード点間

1.2.2　交点情報，交点座標値，境界 ID

交点情報としては，交点座標と境界 ID を記録します．交点座標値は，本ライブラリでは，計算基準点から最短ポリゴン交点までの距離を計算基準線分長 (等間隔格子の場合は格子幅 d) で規格化した値として定義します．特に計算基準線分上にポリゴン交点が存在しなかった場合には，交点座標値を 1.0(無次元の格子幅) と記録します．境界 ID は，複数のポリゴン集合 (ポリゴングループ) を識別する ID で，1〜255 の整数をとります．計算基準線分上にポリゴン交点

が存在しなかった場合には，境界 ID として 0 を記録します．

1.2.3　Octree

本ライブラリは，Octree による木構造セルの交点情報計算にも対応します．対象となるデータ構造は，直交等間隔分割されたセルをルートとして (ルートセル)，各ルートセルに木構造を持たせたものです．各ルートセルを 8 分割して，それを子セルとします．各子セルに対して，さらにそれを 8 分割して子セルの子セルとします．この操作を再帰的に繰り返し，木構造を構築していきます．分割を停止した，すなわち子セルを持たないセルを，リーフセルと呼びます．

1.3　本ライブラリの担当範囲

ポリゴン交点情報を実際に計算するには，以下の手順が必要です．

1. Polylib クラス (並列プログラムの場合は MPIPolylib クラス) のインスタンスを生成
2. 交点計算の対象となるポリゴングループを Polylib のグループ管理ツリーに登録
3. 対象ポリゴンデータの読み込み
4. ポリゴン交点情報の計算

これらの手順のうち，本ライブラリは 4 の部分のみを担当します．本ライブラリが提供するポリゴン交点情報計算関数は，上記 1〜3 までの操作を終了した Polylib クラスのポインタを引数として受け取ります．

Octree データ構造では，SklTree クラスにより既にツリー構造が構築済みであるものとします．

本ライブラリ自身は，MPI によるプロセス並列化はなされていません．ただし，交点情報計算関数の引数として，Polylib クラスではなく，MPIPolylib クラスへのポインタを受け取ることにより，領域分割並列計算に対応できます．その場合には，交点計算の対象として，並列プロセス毎の分割済み空間情報 (グリッド情報，Octree 情報) を交点情報計算関数に渡す必要があります．

なお，直交格子に対する交点情報計算関数は，OpenMP によるスレッド並列化がなされています．

1.4　本ライブラリの提供機能

■交点情報計算関数　直交格子版，Octree 版の各関数を提供します．Octree 版では，計算基準点として，リーフセルのみとするか，全ツリー階層の全セルとするかを選択できます．

■グリッド情報アクセッサクラス　直交格子版の交点情報計算関数には，背景格子の情報と計算基準点の位置 (セル中心またはノード点) をグリッド情報アクセッサクラスを経由して渡します．このクラスをカスタマイズすることにより，等間隔格子だけでなく，不等間隔直交格子にも対応できます．

■交点情報格納用データ構造　メモリ削減のために，交点座標値を 8 ビット圧縮して格納することができます．また，境界 ID の最大値が 31 以下の場合には，それを利用して境界 ID も圧縮格納することができます．直交格子版では，得られた交点情報は，圧縮の有無にかかわらず，Fortran サブルーチンからも読み出し可能な形式で 1 次元配列に格納されます．

■配列ラッパクラス　直交格子版では，交点情報格納用配列を直に扱う必要のない，配列ラッパクラスによるインタフェースも利用可能です．

■Octree セルデータへのアクセッサクラス　Octree 版では，SklCell クラス内のデータ領域へのアクセスを容易にするため，アクセッサ提供クラスを用意しました．

1.5　ヘッダファイルおよび名前空間

本ライブラリの各機能を利用するには，ヘッダファイル Cutlib.h をインクルードする必要があります．また，本ライブラリが提供する関数，クラス，定数は，全て名前空間「cutlib」内で定義されています．本ドキュメントの以降の記述では，スコープ解決演算子「`cutlib::`」を省略しています．実際の利用時には，using 文で cutlib 名前空間の使用を宣言するか，各キーワードの前に明示的にスコープ解決演算子をつける必要があります．

第 2 章

データ構造

本章では，本ライブラリの交点情報計算関数を使用するにあたり理解しておく必要がある，基本的なデータ構造について説明します．それらは，基本データ型，直交格子における配列ラッパクラス，Octree におけるセルデータへのアクセッサクラスからなります．

2.1　交点情報格納用データ構造

基本データ型　計算基準点における 6 方向分の交点情報 (交点座標，境界 ID) を格納する型

配列ラッパクラス　直交格子版の交点情報計算関数で使用する基本データ型配列を内部に持つクラス

セルデータへのアクセッサクラス　Octree 版で使用する，各セルデータ内の交点情報格納領域へのアクセッサを提供するクラス

なお，単独の境界 ID を扱う型として，次のように BidType 型を定義しています．

```
typedef unsigned char BidType;
```

また，交点探査方向には，以下の順に 0〜5 の整数値が割り当てられています．

$0 : -x,\ 1 : +x,\ 2 : -y,\ 3 : +y,\ 4 : -z,\ 5 : +z$

2.1.1　基本データ型

基本データ型は，計算基準点における 6 方向分の交点情報を格納するための型です．これらの型は，Fortran サブルーチンからも読み取り可能なように設計されています．交点座標用，境界 ID 用に，それぞれ 2 種類の型があります．

■交点座標基本データ型

CutPos32 型　交点座標を float(32 ビット) として格納 (図 2.1)

```
typedef float CutPos32[6];
```

CutPos8 型　交点座標を 8 ビット量子化し (値は 0〜255)，3 つずつ，2 つの 32 ビット整数に格納 (図 2.2)

```
typedef int32_t CutPos8[2];
```

■境界 ID 基本データ型

CutBid8 型　0〜255 の境界 ID(8 ビット) を 3 つずつ，2 つの 32 ビット整数に格納 (図 2.3)

```
typedef int32_t CutBid8[2];
```

CutBid5 型　0〜31 の境界 ID(5 ビット) を 6 つまとめて，1 つの 32 ビット整数に格納 (図 2.4)

```
typedef int32_t CutBid5;
```

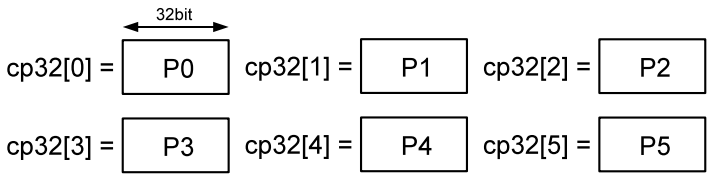

図 2.1　CutPos32 型変数 cp32 への交点座標 P0〜P5(32 ビット浮動小数点数) の格納

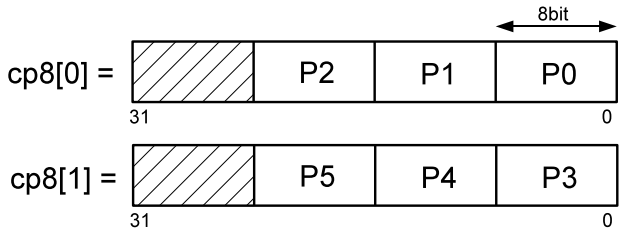

図 2.2　CutPos8 型変数 cp8 への交点座標 P0〜P5(8 ビット符号無し整数) の格納

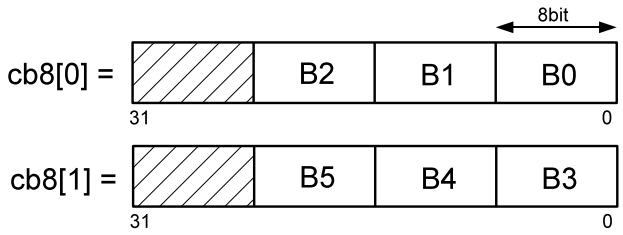

図 2.3　CutBid8 型変数 cb8 への境界 ID B0〜B5(8 ビット符号無し整数) の格納

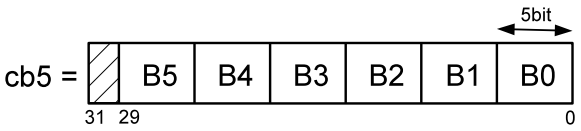

図 2.4　CutBid5 型変数 cb5 への境界 ID B0〜B5(5 ビット符号無し整数) の格納

以降，交点座標基本データ型の総称として **CutPos** 型，境界 ID 基本データ型の総称として **CutBid** 型，という表記を用います．

セルセンタ版およびノード版の交点情報計算関数では，得られた交点情報を，交点座標基本データ型と境界 ID 基本データ型の 2 つの 1 次元配列に格納します．計算基準点の個数を $N_x \times N_y \times N_z$ とすると，計算基準点の位置 (i, j, k) と 1 次元配列のインデックス ijk の間に以下の関係があります．

$$ijk = i + j \times N_x + k \times N_x \times N_y$$

それぞれの型およびその 1 次元配列について，以下のようなアクセス用関数を用意しました[*1]．

方向別アクセス関数

```
float GetCutPos(const CutPos& cp, int d)
float GetCutPos(const CutPos cp[], size_t ijk, int d)
BidType GetCutBid(const CutBid& cb, int d)
BidType GetCutBid(const CutBid cb[], size_t ijk, int d)
```

■引数 (IN)

`CutPos& cp`	CutPos32 型または CutPos8 型
`CutBid& cb`	CutBid8 型または CutBid5 型

[*1] これらの関数はインライン関数として定義されています．

`CutPos cp[]`	CutPos32 型または CutPos8 型の配列
`CutBid cb[]`	CutBid8 型または CutBid5 型の配列
`int d`	交点探査方向 (0〜5)
`size_t ijk`	1 次元配列インデックス

6 方向同時アクセス関数

```
void GetCutPos(const CutPos& cp, float pos[])
void GetCutPos(const CutPos cp[], size_t ijk, float pos[])
void GetCutBid(const CutBid& cb, BidType bid[])
void GetCutBid(const CutBid cb[], size_t ijk, BidType bid[])
```

■引数 (IN)

`CutPos& cp`	CutPos32 型または CutPos8 型
`CutBid& cb`	CutBid8 型または CutBid5 型
`CutPos cp[]`	CutPos32 型または CutPos8 型の配列
`CutBid cb[]`	CutBid8 型または CutBid5 型の配列
`size_t ijk`	1 次元配列インデックス

■引数 (OUT)

`float pos[6]`	交点座標
`BidType bid[6]`	境界 ID

2.1.2　配列ラッパクラス

直交格子版の交点情報計算関数の使用時には，基本データ型配列ではなく，それぞれの配列のラッパクラスを用います．

CutPos32Array　CutPos32 型配列のラッパクラス
CutPos8Array　CutPos8 型配列のラッパクラス
CutBid8Array　CutBid8 型配列のラッパクラス
CutBid5Array　CutBid5 型配列のラッパクラス

ともに，抽象クラス CutPosArray および CutBidArray を実装したものです[*2](図 2.5)．それぞれのクラスは，対応する基本データ型の 1 次元配列をメンバに持っています．また，その配列を Fortran ルーチンに渡せるように，配列へのポインタを得るためのメソッドを備えています．

[*2] 実際には，基本データ型をパラメータに持つテンプレートクラスを通して定義されており，それぞれ，CutPosArrayTemplate<CutPos32>, CutPosArrayTemplate<CutPos8>, CutBidArrayTemplate<CutBid8>, CutBidArrayTemplate<CutBid5> を typedef したものになっています．

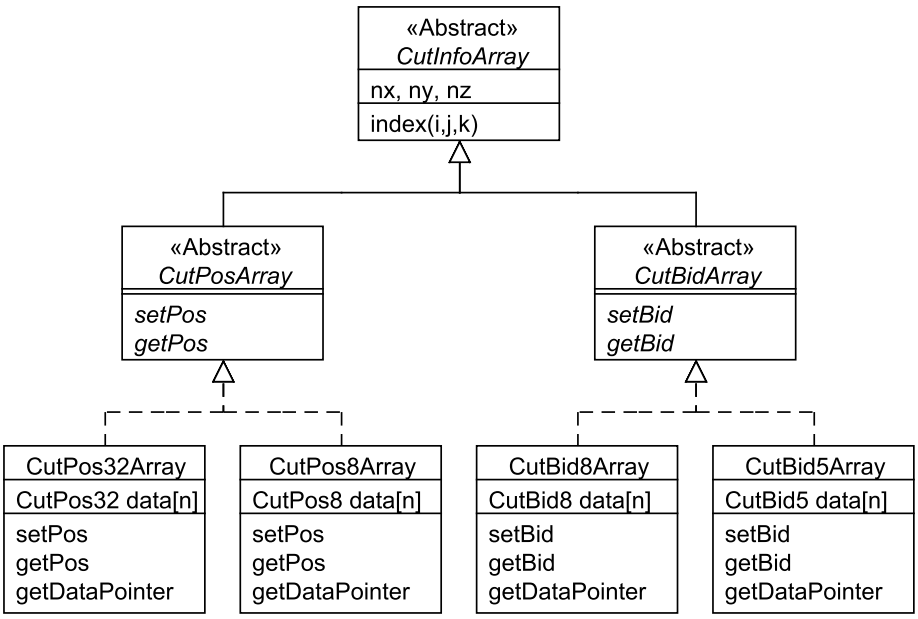

図 2.5　基本データ型配列のラッパクラス

以下の各メソッドの説明では，個々のクラス名の代りに，抽象クラス CutPosArray，CutBidArray を用いています．実際の使用時には，`CutPosArray` → `CutPos32Array` 等，適宜置き換えてください．

コンストラクタ

コンストラクタには，計算基準点の始点と終点を指定するタイプと，計算基準点の総数を指定するタイプの 2 種類あります．

計算基準点の始点と終点を指定:

```
CutPosArray::CutPosArray(int sx, int sy, int sz, int ex, int ey, int ez)
CutBidArray::CutBidArray(int sx, int sy, int sz, int ex, int ey, int ez)
```

■引数 (IN)

`int sx, sy, sz`	計算基準点 始点インデクス
`int ex, ey, ez`	計算基準点 終点インデクス

交点計算範囲には終点インデクスも含まれます．各引数には負のインデクスも指定可能です．以下の説明で登場する各メソッドの 3 次元インデクス引数 `i,j,k` は，この始点と終点の間に収まるなら，負の値も指定できます．

3 次元インデクス `i,j,k` から基本データ型 1 次元配列の添字 `ijk` への変換は以下のように計算されます．

```
ijk = (i-sx) + (j-sy)*(ex-sx+1) + (k-sz)*(ex-sx+1)*(ey-sx+1)
```

計算基準点の総数を指定:

```
CutPosArray::CutPosArray(size_t nx, size_t ny, size_t nz)
CutPosArray::CutPosArray(const size_t ndim[])
CutBidArray::CutBidArray(size_t nx, size_t ny, size_t nz)
CutBidArray::CutBidArray(const size_t ndim[])
```

■引数 (IN)

`size_t nx, ny, nz`	計算基準点数
`size_t ndim[3]`	計算基準点数

これらは，始点・終点指定タイプのコンストラクタにおいて，始点として sx=0, sy=0, sz=0，終点として ex=nx-1, ey=ny-1, ez=nz-1 等と指定した場合と同じです．

データ配列ポインタ取得メソッド

```
CutPos* CutPosArray::getDataPointer()
CutBid* CutBidArray::getDataPointer()
```

返り値は，クラス内部に保持している交点情報基本データ型配列へのポインタです．

■注意　配列ラッパクラスのデストラクタが呼ばれると，その配列領域も開放されてしまうので注意してください．

方向別アクセスメソッド

```
float CutPosArray::getPos(int i, int j, int k, int d)
float CutPosArray::getPos(size_t ijk, int d)
BidType CutBidArray::getBid(int i, int j, int k, int d)
BidType CutBidArray::getBid(size_t ijk, int d)
```

■引数 (IN)

`int i, j, k`	3 次元配列インデックス
`size_t ijk`	1 次元配列インデックス
`int d`	交点探査方向 (0〜5)

6 方向同時アクセスメソッド

```
void CutPosArray::getPos(int i, int j, int k, float pos[])
void CutPosArray::getPos(size_t ijk, float pos[])
void CutBidArray::getBid(int i, int j, int k, BidType bid[])
void CutBidArray::getBid(size_t ijk, BidType bid[])
```

■引数 (IN)

`int i, j, k`	3 次元配列インデックス
`size_t ijk`	1 次元配列インデックス

■引数 (OUT)

`float pos[6]`	交点座標
`BidType bid[6]`	境界 ID

計算基準点開始インデクス取得

```
int CutPosArray::getStartX()
int CutPosArray::getStartY()
int CutPosArray::getStartZ()

int CutBidArray::getStartX()
int CutBidArray::getStartY()
int CutBidArray::getStartZ()
```

配列サイズ関連メソッド

```
size_t CutPosArray::getSizeX()
size_t CutPosArray::getSizeY()
size_t CutPosArray::getSizeZ()

size_t CutBidArray::getSIzeX()
size_t CutBidArray::getSIzeX()
size_t CutBidArray::getSIzeX()
```

getSizeX, getSizeY, getSizeZ, それぞれのメソッドで得られた値の積が，クラス内部に保持している基本データ型配列の長さになります．

2.1.3　セルデータへのアクセッサクラス

Octree 版では，セルに付随するデータは，SklCell クラス内のデータ領域に格納します．交点情報の SklCell クラス内のデータ領域への格納，および，格納された交点情報の取得を容易にするために，アクサッサ提供クラスを用意しました．

CutPos32Octree　CutPos32 型データ用アクセッサ提供クラス
CutPos8Octree　CutPos8 型データ用アクセッサ提供クラス
CutBid8Octree　CutBid8 型データ用アクセッサ提供クラス
CutBid5Octree　CutBid5 型データ用アクセッサ提供クラス

ともに，抽象クラス CutPosOctree および CutBidOctree を実装したものです[*3](図 2.6)．これらのクラスでは，内部に交点情報格納用のデータ領域を持っていません．SklCell クラスのオブジェクトに結合して，そのデータ領域へのアクセスを提供します．

[*3] 実際には，基本データ型をパラメータに持つテンプレートクラスを通して定義されており，それぞれ，CutPosOctreeTemplate<CutPos32,6>, CutPosOctreeTemplate<CutPos8,2>, CutBidOctreeTemplate<CutBid8,2>, CutBidOctreeTemplate<CutBid5,1> を typedef したものになっています．

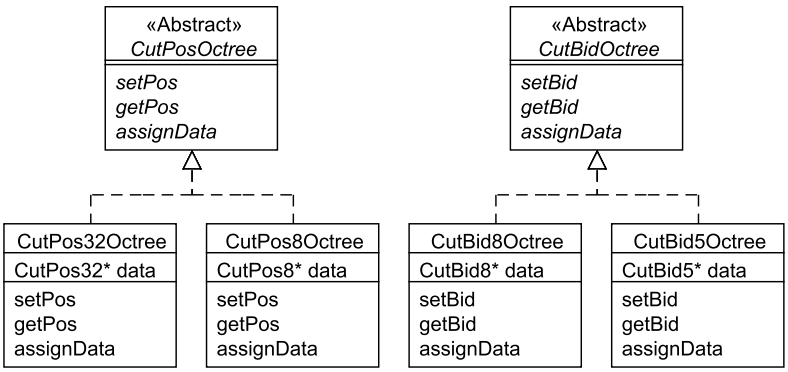

図 2.6　SklCell データへのアクセッサ提供クラス

以下の各メソッドの説明では，個々のクラス名の代りに，抽象クラス CutPosOctree，CutBidOctree を用いています．実際の使用時には，`CutPosOctree` → `CutPos32Octree` 等，適宜置き換えてください．

コンストラクタ

```
CutPosOctree::CutPosOctree(int index)
CutBidOctree::CutBidOctree(int index)
```

■引数 (IN)

`int index`	SklCell データ領域内での交点情報格納開始インデックス

SklCell データ領域に結合

```
void CutPosOctree::assignData(float* data)
void CutPosOctree::assignData(float* data)
```

■引数 (IN)

`float* data`	SklCell データ領域へのポインタ

方向別アクセスメソッド

```
float CutPosOctree::getPos(int d)
BidType CutBidOctree::getBid(int d)
```

■引数 (IN)

`int d`	交点探査方向 (0〜5)

6 方向同時アクセスメソッド

```
void CutPosOctree::getPos(float pos[])
```

```
void CutBidOctree::getBid(BidType bid[])
```

■引数 (OUT)

`float pos[6]`	交点座標
`BidType bid[6]`	境界 ID

必要領域サイズ取得メソッド

```
unsigned CutPosOctree::getSizeInFloat()
unsigned CutBidOctree::getSizeInFloat()
```

■注意　交点情報の格納に必要な領域サイズを float 単位で返します．この値を，Octree 構築時の CreateTree メソッドで指定するセルに格納するデータ数パラメータを決めるのに利用できます．

第 3 章

API 利用方法

本章では，各交点情報計算関数のインタフェースについて述べます．まず、全ての交点情報計算関数に共通するポリゴングループと境界 ID の指定方法と，ポリゴンデータの修復関数について説明します．次に，直交格子版における，グリッド情報と計算基準点情報の指定方法，計算領域の指定方法について説明します．

3.1　共通インタフェース

3.1.1　計算対象ポリゴングループと境界 ID の指定方法

Polylib 設定ファイルを通じて，計算対象ポリゴングループとその境界 ID を指定します．Polylib バージョン 2.1 以降では，Polylib 設定ファイルの書式は TextParser 形式になりました．この設定ファイルに，各ポリゴングループの定義とポリゴングループ間の階層関係を記述します．

本ライブラリでは，ポリゴングループの階層関係において，

階層ツリー構造のリーフノードに属するポリゴングループで，
1〜255 の ID 番号が指定されたポリゴングループ

を交点情報計算の対象とします．この時，Polylib 設定ファイルで指定された ID 番号が，そのまま境界 ID となります．ID 番号は，複数のポリゴングループに重複して与えることができます．なお，Polylib では，設定ファイルにおいて ID 番号の指定が省略されたポリゴングループには，ID 番号として 0 が設定されます．

以下に，次のケースでの Polylib 設定ファイル例を示します．

ポリゴングループ"root/child_A": ファイル child_A.stl，境界 ID 1
ポリゴングループ"root/child_B": ファイル child_B.stl，境界 ID 2

```
Polylib {
    root {
        class_name = "PolygonGroup"
        child_A {
            class_name = "PolygonGroup"
            filepath = "child_A.stl"
            id = "1"
        }
        child_B {
            class_name = "PolygonGroup"
            filepath = "child_B.stl"
            id = "2"
        }
    }
}
```

Polylib 設定ファイルの詳細については，Polylib 利用説明書を参照ください．

3.1.2　ポリゴンデータの最適化

STL ファイル中の各ポリゴンデータは，以下の形式で格納されています．

```
facet normal n_x n_y n_z  # 法線ベクトル
 outer loop
  vertex v0_x v0_y v0_z   # 頂点 0 の座標
  vertex v1_x v1_y v1_z   # 頂点 1 の座標
  vertex v2_x v2_y v2_z   # 頂点 2 の座標
 endloop
```

```
endfacet
```

本ライブラリの交点情報計算関数では，これらのデータを次ぎのように扱います．

- 法線ベクトルの値を信用してそのまま使う
- 法線ベクトルの向きと 3 頂点の並び順に整合性があることを仮定する

ここでの整合性とは，「3 頂点 $\boldsymbol{v}_0, \boldsymbol{v}_1, \boldsymbol{v}_2$ から決まるポリゴンの表裏の向き」と「法線ベクトル $\boldsymbol{n}$ の向き」との間の整合性です．数学的には，内積値

$$(\boldsymbol{v}_1 - \boldsymbol{v}_0) \times (\boldsymbol{v}_2 - \boldsymbol{v}_0) \cdot \boldsymbol{n}$$

の符号が，全ポリゴンデータに対して一致 (常に正，または常に負) していることを要求しています．

ポリゴンデータ修復関数

一般的な STL ファイルには，ポリゴンの法線ベクトル値の精度が悪い物が見られます．また，上記の整合性についても必ずしも期待できません．そこで本ライブラリでは，ポリゴンデータを最適化する (修復する) 以下の補助関数を提供することにしました．

```
void RepairPolygonData(const Polylib* pl,
                       bool doubt_vertex_order = true,
                       bool doubt_normal_quality = ture)
```

pl は，ポリゴンデータ読み込み済みの Polylib クラスオブジェクトへのポインタ．フラグ doubt_vertex_order が真の場合，向きに整合性のないポリゴンに対して，法線ベクトルはそのままにして，頂点 1 と頂点 2 の順番を入れ替えます．フラグ doubt_normal_quality が真の場合には，法線ベクトルを 3 頂点の座標値より再計算して修正します．この時，法線ベクトルの向き (正負) は，修正前と同じになるようにとります．両フラグは省略可能で，その場合のディフォルト値はともに真になります．

関数 RepairPolygonData は，Polylib オブジェクト内部に保持されているポリゴンデータを直接修正します．そのため，Polylib のセーブメソッドをこの後に呼ぶことにより，修正済みのポリゴンデータを STL ファイルの形式で保存することができます．

3.1.3　グリッド情報と計算基準点情報の指定方法

直交格子版の交点情報計算関数には，ユーザがカスタマイズしたグリッド情報アクセッサクラスにより，グリッドデータ情報と計算基準点情報を渡します．グリッド情報アクセッサクラスに必要な仕様は，以下の 2 点のみです．

1. GridAccessor クラスを継承していること
2. 計算基準点座標と各 6 方向の計算基準線分長を返す getSearchRange メソッドを実装すること

getSearchRange メソッドのインタフェースは次のとおりです．

```
void getSearchRange(int i, int j, int k,
                    double center[3], double range[6])
```

このメソッド内で，center[3] に，インデックス (i,j,k) が指定された時の計算基準点座標を返します．range[6] には，各 6 方向に伸びる計算基準線分の長さを $-X, +X, -Y, +Y, -Z, +Z$ の順に格納します．

center[3] に返す値を調整することにより，計算基準点の位置 (セル中心なのか，ノード点なのか) を調整できます．また，range[6] の値に空間依存性を持たせることにより，不等間隔格子にも対応できます．

以下に例として，等間隔格子上のセル中心におけるグリッド情報アクセッサクラスの実装を示します．この例のように，getSearchRange メソッド内部で使用する原点座標やセルピッチなどの情報は，前もってコンストラクタで渡しておきます．

```
#include "GridAccessor/GridAccessor.h"

namespace cutlib {

/// セル中心グリッド情報アクセッサ (等間隔格子).
class Cell : public GridAccessor {

  double o[3];  ///< 原点 (=セル (0,0,0) のマイナス側頂点) の座標値
  double d[3];  ///< セルピッチ

public:

  /// コンストラクタ.
  ///
  ///  @param[in] org   原点 (=セル (0,0,0) のマイナス側頂点) の座標値
  ///  @param[in] pitch セルピッチ
  ///
  Cell(const double org[3], const double pitch[3]) {
    for (int i = 0; i < 3; i++) {
      o[i] = org[i];
      d[i] = pitch[i];
    }
  }

  /// ポリゴン検索領域を指定するメソッド.
  ///
  ///  @param[in]  i,j,k  3 次元インデックス
  ///  @param[out] center 計算基準点座標
  ///  @param[out] range  6 方向毎の計算基準線分の長さ
  ///
  void getSearchRange(int i, int j, int k,
                      double center[3], double range[6]) const {
    center[0] = o[0] + (0.5 + i) * d[0];
    center[1] = o[1] + (0.5 + j) * d[1];
    center[2] = o[2] + (0.5 + k) * d[2];
    range[X_M] = range[X_P] = d[0];
    range[Y_M] = range[Y_P] = d[1];
    range[Z_M] = range[Z_P] = d[2];
  }
};
}
```

本パッケージでは，グリッド情報アクセッサクラスの実装サンプルとて，以下の 3 クラスを提供しています．

Cell クラス　include/GridAccessor/Cell.h
: 等間隔格子，セル中心

Node クラス　include/GridAccessor/NUCell.h
: 等間隔格子，ノード点

NUCell クラス　include/GridAccessor/NUCell.h
: 不等間隔 (NonUniform) 格子，セル中心

また，これらグリッド情報アクセッサクラスの使用例を examples ディレクトリに納めました．

3.1.4　計算領域指定方法

Cutlib バージョン 3 より，グリッド情報アクセッサクラスの導入にともない，計算領域指定方法が変更になりました．

ディフォルト計算領域

交点情報計算関数の呼び出し時に計算対象範囲を指定しなかった場合，以下の 3 次元インデクス (i,j,k) の範囲で交点情報を計算します．配列ラッパクラスのコンストラクタで，始点 (sx,sy,sz) と終点 (ex,ey,ez) を指定している場合には

$$sx \le i \le ex, \quad sy \le j \le ey, \quad sz \le k \le ez,$$

配列サイズ (nx,ny,nz または ndim) を指定している場合は

$$0 \le i \le nx\text{-}1, \quad 0 \le j \le ny\text{-}1, \quad 0 \le k \le nz\text{-}1,$$

または

$$0 \le i \le ndim[0]\text{-}1, \quad 0 \le j \le ndim[1]\text{-}1, \quad 0 \le k \le ndim[2]\text{-}1,$$

となります．

計算対象領域の限定

交点情報計算関数の呼び出し時に以下のパラメータを指定することにより，計算対象領域を限定できます．これにより，計算基準点の一部についてのみ交点情報を計算することが可能になります．

`size ista[3]`	開始計算基準点インデクス
`size_t nlen[3]`	計算基準点数

この場合，3 次元インデクス (i,j,k) の範囲は

$$ista[0] \le i \le ista[0]+nlen[0]\text{-}1,$$
$$ista[1] \le j \le ista[1]+nlen[1]\text{-}1,$$
$$ista[2] \le k \le ista[2]+nlen[2]\text{-}1,$$

となります．この範囲が，配列ラッパクラスのコンストラクタで指定した範囲 (上記ディフォルト計算領域) を越えていた場合には，実行時エラーとなります．

3.1.5　リターンコード

交点情報計算関数は全て，整数値のリターンコードを返します．それらは，Cutlib.h で以下のような enum 定数として定義されています．

`SUCCESS`	`= 0`	成功
`BAD_GROUP_LIST`	`= 1`	「境界 ID, ポリゴングループ名」対応リストが不正
`BAD_POLYLIB`	`= 2`	Polylib オブジェクトが不正 (未初期化等)
`BAD_SKLTREE`	`= 3`	SklTree オブジェクトが不正 (未初期化等)
`SIZE_EXCEED`	`= 4`	`ista[]+nlen[]` が配列サイズを越えている
`OTHER_ERROR`	`= 10`	その他のエラー (現在は未使用)

3.2　交点情報計算関数 インタフェース

■注意　Cutlib バージョン 3 より，直交格子版関数のインタフェースが大幅に変更になりました．また，基本データ型 1 次元配列の直接渡しによるインタフェースは廃止されました．

3.2.1　直交格子版インタフェース

全領域で計算:

```
int CalcCutInfo(const GridAccessor* grid, const Polylib* pl,
                CutPosArray* cutPos, CutBidArray* cutBid)
```

計算領域指定:

```
int CutInfoCell(const int ista[], const size_t nlen[],
                const GridAccessor* grid, const Polylib* pl,
                CutPosArray* cutPos, CutBidArray* cutBid)
```

■引数 (IN)

`int ista[3]`	開始計算基準点インデクス
`size_t nlen[3]`	計算基準点数
`GridAccessor* grid`	グリッド情報アクセッサクラスへのポインタ
`Polylib* pl`	Polylib クラスへのポインタ

■引数 (OUT)

`CutPosArray* cutPos`	CutPos32Array または CutPos8Array へのポインタ
`CutBidArray* cutBid`	CutBid8Array または CutBid5Array へのポインタ

3.2.2　Octree 版インタフェース

リーフセルのみで計算:

```
int CutInfoOctreeLeafCell(SklTree* tree,
                          const Polylib* pl,
                          CutPosOctree* cutPos, CutBidOctree* cutBid)
```

全セルで計算:

```
int CutInfoOctreeAllCell(SklTree* tree,
                         const Polylib* pl,
                         CutPosOctree* cutPos, CutBidOctree* cutBid)
```

計算対象セルを指定:

```
int CutInfoOctree(SklTree* tree,
                  const Polylib* pl,
                  CutPosOctree* cutPos, CutBidOctree* cutBid,
                  bool leafCellOnly = true)
```

■引数 (IN)

`Polylib* pl`	Polylib クラスへのポインタ
`bool leafCellOnly`	true:リーフセルのみ (ディフォルト)/false:全セル

■引数 (IN/OUT)

SklTree* tree	SklTree へのポインタ
`CutPosOctree* cutPos`	CutPos32Octree または CutPos8Octree へのポインタ
`CutBidOctree* cutBid`	CutBid8Octree または CutBid5Octree へのポインタ

3.3　使用例

3.3.1　等間隔直交格子，セル中心間

```
#include "Cutlib.h"
#include "GridAccessor/Cell.h"
using namespace cutlib;

   …

  std::string plConfig;  // Polylib 初期化ファイル名

  size_t ncell[3];    // 全セル数
  double org[3];      // 領域原点座標
  double pitch[3];    // セル間隔

  int ista[3];        // 計算対象領域開始セル位置
  size_t nlen[3];     // 計算対象領域セル数

  /* ここで，各パラメータに値をセット */

  // Polylib 初期化
  Polylib* pl = Polylib::get_instance();
  pl->load(plConfig);

  // ポリゴンデータの修復 (必要なら)
  RepairPolygonData(pl);

  // 配列ラッパクラス (およびその内部の配列データ領域) の生成
  CutPos32Array* cutPos = new CutPos32Array(ncell);
//CutPos8Array*  cutPos = new CutPos8Array(ncell);
  CutBid8Array*  cutBid = new CutBid8Array(ncell);
//CutBid5Array*  cutBid = new CutBid5Array(ncell);

  // グリッド情報アクセッサクラス
  GridAccessor* grid = new Cell(org, pitch);

  // 交点情報計算
  CalcCutInfo(ista, nlen, grid, pl, cutPos, cutBid);

  // 以下は計算した交点情報の読み出し例

  // 6 方向まとめて交点情報を読み出し
  float   pos6[6];
  BidType bid6[6];
  cutPos->getPos(i, j, k, pos6);
  cutBid->getBid(i, j, k, bid6);

  // d(0〜5) 方向のみ読み出し
  float   pos = cutPos->getPos(i, j, k, d);
  BidType bid = cutBid->getBid(i, j, k, d);

  // Fortran 用に配列データをエクスポート
  CutPos32* posData = cutPos->getDataPointer();
//CutPos8*  posData = cutPos->getDataPointer();
  CutBid8*  bidData = cutBid->getDataPointer();
//CutBid5*  bidData = cutBid->getDataPointer();

  // 関数による配列要素からの直接読み出しも可能
  int ijk = i + j*ncell[0] + k*ncell[0]*ncell[1];
  GetCutPos(posData, ijk, pos6);
  bid = GetCutBid(bidData, ijk, d);

   …
```

3.3.2　不等間隔直交格子，セル中心間

```
#include "Cutlib.h"
#include "GridAccessor/NUCell.h"
using namespace cutlib;

   …

  std::string plConfig;  // Polylib 初期化ファイル名

  size_t ncell[3];    // 全セル数

  int ista[3];        // 計算対象領域開始セル位置
  size_t nlen[3];     // 計算対象領域セル数

  /* ここで，各パラメータに値をセット */

  double* x = new double[ncell[0]+1]; // グリッド点 x 座標配列
  double* y = new double[ncell[1]+1]; // グリッド点 y 座標配列
  double* z = new double[ncell[2]+1]; // グリッド点 z 座標配列

  /* ここで，グリッド点座標配列 x,y,z の中身をセット */

  // Polylib 初期化
  Polylib* pl = Polylib::get_instance();
  pl->load(plConfig);

  // ポリゴンデータの修復 (必要なら)
  RepairPolygonData(pl);

  // 配列ラッパクラス (およびその内部の配列データ領域) の生成
  CutPos32Array* cutPos = new CutPos32Array(ncell);
//CutPos8Array*  cutPos = new CutPos8Array(ncell);
  CutBid8Array*  cutBid = new CutBid8Array(ncell);
//CutBid5Array*  cutBid = new CutBid5Array(ncell);

  // グリッド情報アクセッサクラス
  GridAccessor* grid = new NUCell(ncell, x, y, z);

  // 交点情報計算
  CalcCutInfo(ista, nlen, grid, pl, cutPos, cutBid);

  // 以下は計算した交点情報の読み出し例

  // 6 方向まとめて交点情報を読み出し
  float   pos6[6];
  BidType bid6[6];
  cutPos->getPos(i, j, k, pos6);
  cutBid->getBid(i, j, k, bid6);

  // d(0〜5) 方向のみ読み出し
  float   pos = cutPos->getPos(i, j, k, d);
  BidType bid = cutBid->getBid(i, j, k, d);

  // Fortran 用に配列データをエクスポート
  CutPos32* posData = cutPos->getDataPointer();
//CutPos8*  posData = cutPos->getDataPointer();
  CutBid8*  bidData = cutBid->getDataPointer();
//CutBid5*  bidData = cutBid->getDataPointer();

  // 関数による配列要素からの直接読み出しも可能
  int ijk = i + j*ncell[0] + k*ncell[0]*ncell[1];
  GetCutPos(posData, ijk, pos6);
  bid = GetCutBid(bidData, ijk, d);

   …
```

3.3.3　Octree 版: リーフセルのみで計算

```
#include "Cutlib.h"
using namespace cutlib;

   …

  std::string plConfig;  // Polylib 初期化ファイル名

  SklTree* tree;         // Octree クラス

  unsigned dIndex = ...; // 交点情報格納開始位置

  // アクセッサクラスの生成
  CutPos32Octree* cutPos = new CutPos32Octree(dIndex);
//CutPos8Octree*  cutPos = new CutPos8Octree(dIndex);
  CutBid8Octree*  cutBid
                    = new CutBid8Octree(dIndex + cutPos->getSizeInFloat());
//CutBid5Octree*  cutBid
//                  = new CutBid5Octree(dIndex + cutPos->getSizeInFloat());

  // 各セルに確保させるデータ量
  unsigned dLen = cutPos->getSizeInFloat() + cutBid->getSizeInFloat() + … ;

  /* ここで，Octree 構築，各パラメータに値をセット */

  // Polylib 初期化
  Polylib pl = Polylib::get_instance();
  pl->load(plConfig);

  CutInfoOctreeLeafCell(tree, pl, cutPos, cutBid);

  // 以下は計算した交点情報の読み出し例

  // リーフセルを巡回するループ
  for (SklCell* cell = tree->GetLeafCellFirst(); cell != 0;
       cell = tree->GetLeafCellNext(cell)) {

    // アクセッサクラスにセルのデータ領域を結合
    cutPos->assignData(cell->GetData());
    cutBid->assignData(cell->GetData());

    // 6 方向まとめて交点情報を読み出し
    float   pos6[6];
    BidType bid6[6];
    cutPos->getPos(pos6);
    cutBid->getBid(bid6);

    // d(0〜5) 方向みの読み出し
    float   pos = cutPos->getPos(d);
    BidType bid = cutBid->getBid(d);

    …
  }

  …
```

3.3.4　Octree 版: 全セルで計算

```
#include "Cutlib.h"
using namespace cutlib;

// 再帰的にツリーの全セルを巡る関数
void extractDataFromCell(SklCell* cell, CutPosOctree* cp, CutBidOctree* cb);

   …

  std::string plConfig;  // Polylib 初期化ファイル名

  SklTree* tree;         // Octree クラス

  unsigned dIndex = ...; // 交点情報格納開始位置

  // アクセッサクラスの生成
  CutPos32Octree* cutPos = new CutPos32Octree(dIndex);
//CutPos8Octree*  cutPos = new CutPos8Octree(dIndex);
  CutBid8Octree*  cutBid
                    = new CutBid8Octree(dIndex + cutPos->getSizeInFloat());
//CutBid5Octree*  cutBid
//                  = new CutBid5Octree(dIndex + cutPos->getSizeInFloat());

  // 各セルに確保させるデータ量
  unsigned dLen = cutPos->getSizeInFloat() + cutBid->getSizeInFloat() + … ;

  /* ここで，Octree 構築，各パラメータに値をセット */

  // Polylib 初期化
  Polylib pl = Polylib::get_instance();
  pl->load(plConfig);

  CutInfoOctreeAllCell(tree, pl, cutPos, cutBid);

  // 以下は計算した交点情報の読み出し例

  size_t nx, ny, nz;    // ルートセル数
  tree->GetSize(nx, ny, nz);
  for (size_t k = 0; k < nz; k++) {
    for (size_t j = 0; j < ny; j++) {
      for (size_t i = 0; i < nx; i++) {
        SklCell* cell = tree->GetRootCell(i, j, k);
        extractDataFromCell(cell, cutPos, cutBid);
      }
    }
  }

  …

/*
 * 再帰的にツリーの全セルを巡る関数
 */
void extractDataFromCell(SklCell* cell, CutPosOctree* cp, CutBidOctree* cb)
{
  // アクセッサクラスにセルのデータ領域を結合
  cp->assignData(cell->GetData());
  cb->assignData(cell->GetData());

  // 6 方向まとめて交点情報を読み出し
  float   pos6[6];
  BidType bid6[6];
  cutPos->getPos(pos6);
  cutBid->getBid(bid6);

  // d(0〜5) 方向のみ読み出し
  float   pos = cutPos->getPos(d);
```

```
  BidType bid = cutBid->getBid(d);

  // もし子セルがあるなら…
  if (cell->hasChild()) {
    for (TdPos child = 0; child < 8; child++) {
      SklCell* childCell = cell->GetChildCell(child);
      extractDataFromCell(childCell, cp, cb);   // 再帰呼び出し
    }
  }
}

  …
```

第 4 章

性能評価

本章では，Cutlib の動作検証と性能評価について報告します．

4.1　検証環境

以下の環境で検証を行いました．

CPU　Intel Core i7 2.3GHz
メモリ　16GB
OS　Mac OS X 10.8.2
コンパイラ　Intel 13.0.1 および GCC 4.8.0
ライブラリ　Polylib 2.2, TextParser 1.2-d

なお，本ドキュメントで使用している可視化図は，全て ParaView[*1]を用いて作成しています．

4.2　動作検証

examples ディレクトリ下に次の 4 つのサンプルプログラムがあります．

Cell　等間直交隔格子，セル中心
Node　等間直交隔格子，ノード点
NonUniformCell　不等間直交隔格子，セル中心
Octree　Octree(リーフセルのみ，全セル)

各プログラムとも，計算結果を vtk データファイルとして出力します．この vtk データファイルには，交点情報を 3 次元スカラ場ではなく，以下の手順で粒子情報 (＝位置情報＋スカラ属性) として書き出しています．

1. 各セル中心 (ノード版の場合は各ノード点) において，6 方向について，ポリゴン交点が存在していたかをチェックします．
2. ポリゴン交点が存在していたら，実際の交点位置を計算し，その三次元座標を粒子位置，境界 ID 値をその粒子の属性とします．
3. vtk ファイルはプラス用とマイナス用の 2 つを用意して，セル中心のプラス側軸上 (+x,+y,+z) の粒子情報はプラス用ファイルに，マイナス側軸上 (-x,-y,-z) の粒子情報はマイナス用ファイルに出力します．

Cutlib により交点情報が正しく計算されていたなら，プラス出力ファイルとマイナス出力ファイルを同時に可視化表示すると，隣接する計算基準点間にポリゴンが一枚しか存在しない場合には，ポリゴン交点に対応する粒子は 2 つ重なって表示されるはずです．

[*1] http://www.paraview.org

4.2.1　Cell(等間隔直交格子，セル中心)

入力データとして，球の 1/8 表面からなる 2 つの STL ファイルを用意しました (図 4.1).

testA.stl　中心 (0,0,0) 半径 1，ポリゴン数 451，境界 ID 1
testB.stl　中心 (1,1,0) 半径 1，ポリゴン数 451，境界 ID 2

ともに，単位立方体領域を 0.1 間隔で $10 \times 10 \times 10$ に分割して作成した数値データの等値面からなるため，各ポリゴンの大きさのオーダは 0.1 程度になっています.

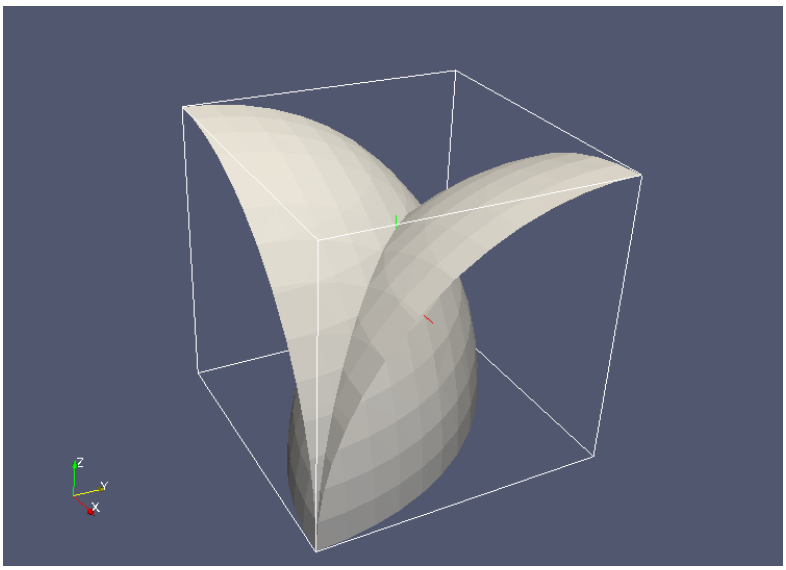

図 4.1　入力データ testA.stl，testB.stl

この入力データに対して，セル数 $10 \times 10 \times 10$(セル間隔 0.1) の計算を行いました．

以下の各図は，プラス出力ファイルからの交点位置に「球」をプロットし，マイナス出力ファイルからの交点位置に「箱」(透明度 0.4) をプロットしています．色は，境界 ID=1 で青，境界 ID=2 で赤，となってます．

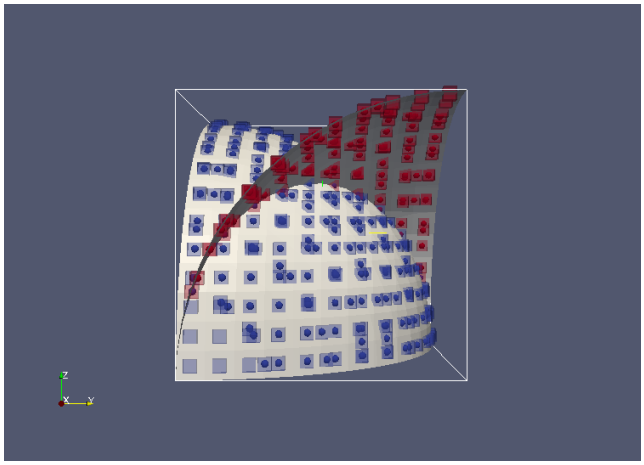

図 4.2　CutPos32, CutBid8 に格納 (+x 方向から)

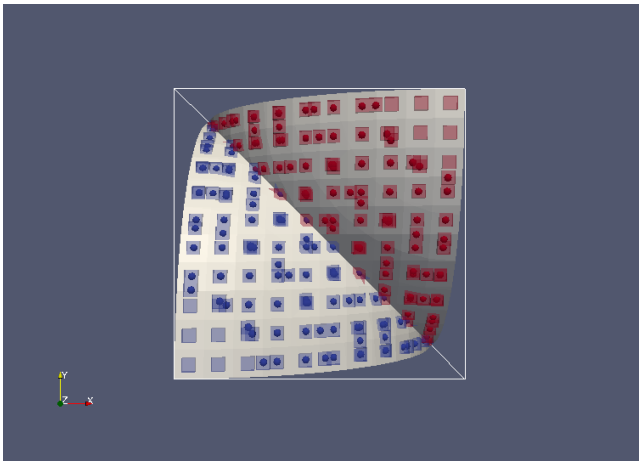

図 4.3　CutPos32, CutBid8 に格納 (+z 方向から)

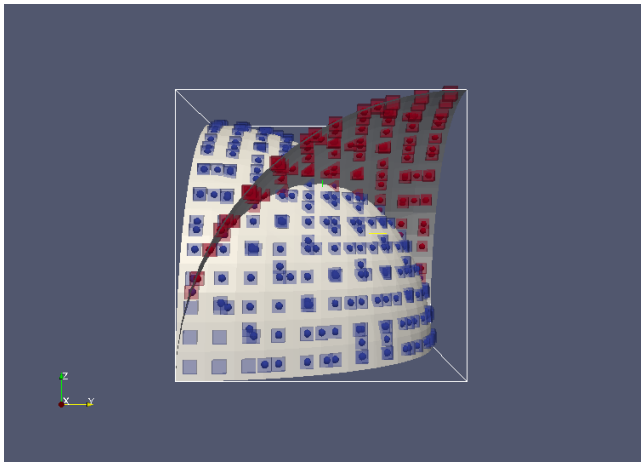

図 4.4　CutPos8, CutBid5 に格納 (+x 方向から)

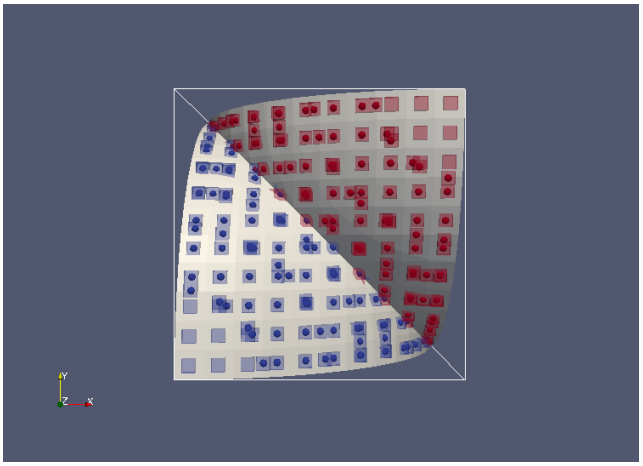

図 4.5　CutPos8, CutBid5 に格納 (+z 方向から)

セル中心の場合には，計算基準点がポリゴン面の外側に存在しない領域がありうるため，そこでは箱または球のみがプロットされています．

4.2.2　Node(等間隔直交格子，ノード点)

セル中心の場合と同じ入力データに対して，ノード数 $11 \times 11 \times 11$ で計算しました．

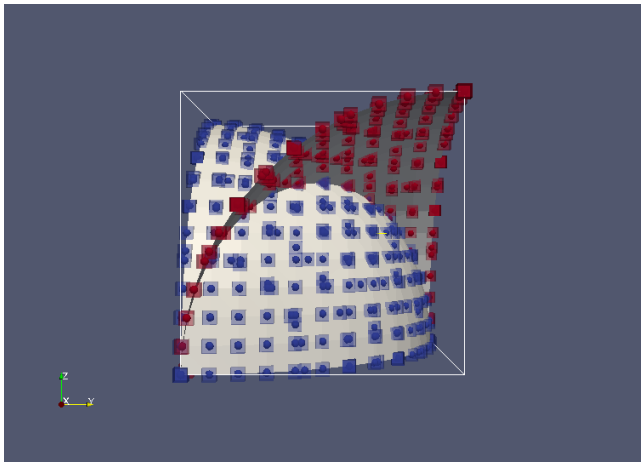

図 4.6　CutPos32, CutBid8 に格納 (+x 方向から)

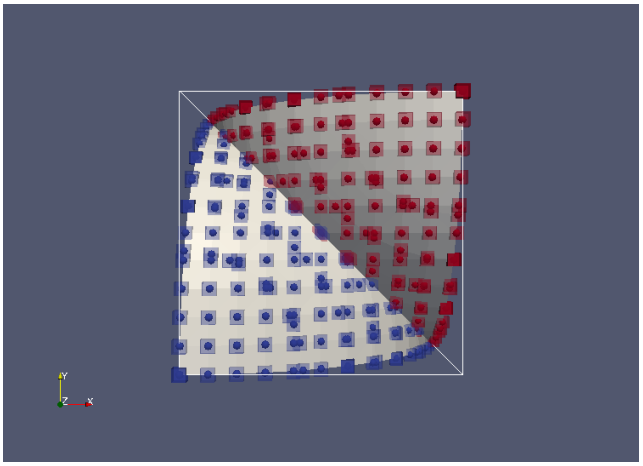

図 4.7　CutPos32, CutBid8 に格納 (+z 方向から)

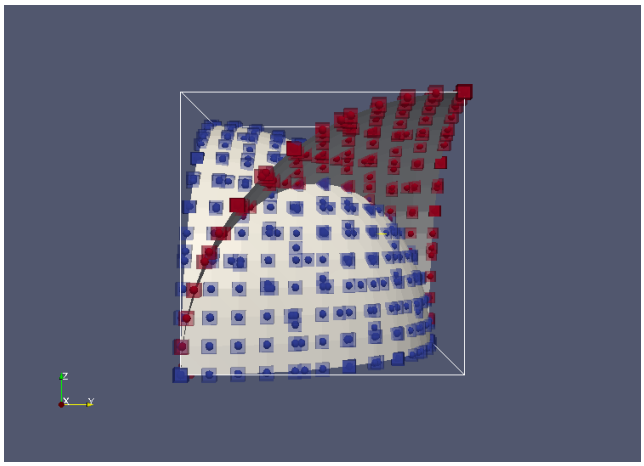

図 4.8　CutPos8, CutBid5 に格納 (+x 方向から)

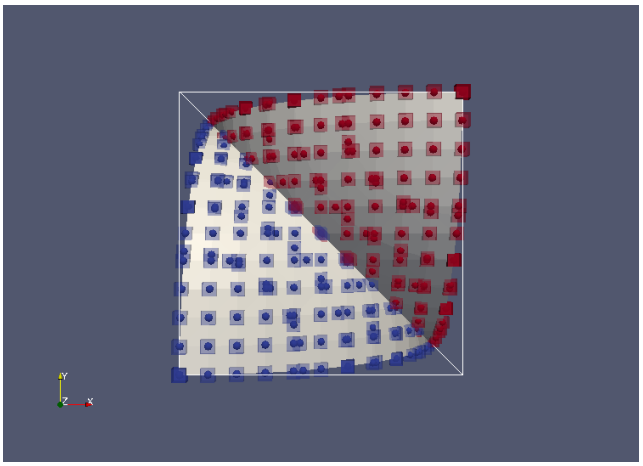

図 4.9　CutPos8, CutBid5 に格納 (+z 方向から)

ノード点の場合では，領域境界付近でも，計算の基準となるノード点がポリゴン面の両側にあるため，全ての箇所で箱と球が重なってプロットされています．

4.2.3　NonUniformCell(不等間隔直交格子，セル中心)

同様な入力データに対して，不等間隔直交格子上でのセル中心計算を行いました．

格子はデータは以下のように作成しています．計算対象領域は単位立方体領域 $[0,1]^3$ であるため，まず，等間隔格子 $x_i^{(0)}$ を次のように決めます．

$$x_0^0 = 0,\ x_1^0 = 1/N,\ x_2^0 = 2/N,\ ...,\ x_{N-1}^0 = (N-1)/N,\ x_N^{(0)} = 1$$

この等間隔格子 x_i^0 に対して，以下の式による変調をかけることにより，不等間隔格子を作成します．

$$x_i = \left(x_i^{(0)}\right)^2, \qquad i = 0, ..., N$$

y 座標，z 座標についても，同様な方法で決めています．

以下に $N = 10$ とした場合の計算結果を示します．

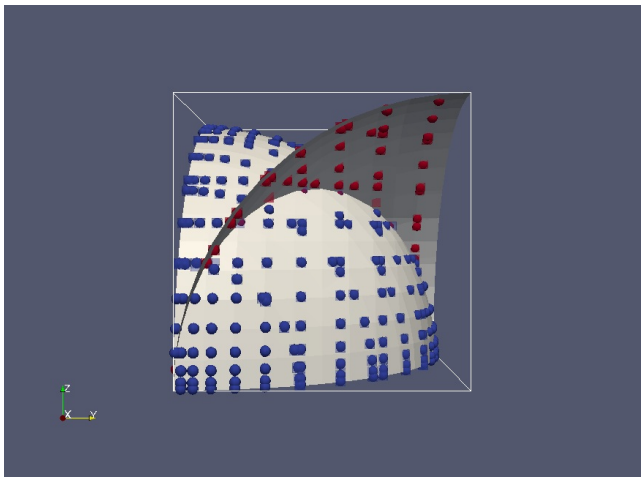

図 4.10　CutPos32, CutBid8 に格納 (+x 方向から)

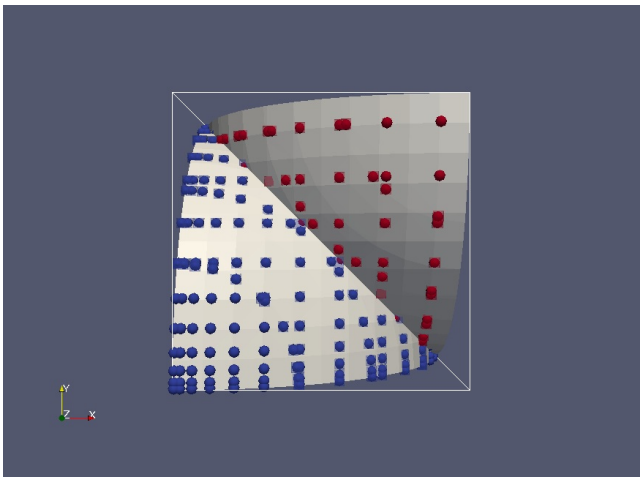

図 4.11　CutPos32, CutBid8 に格納 (+z 方向から)

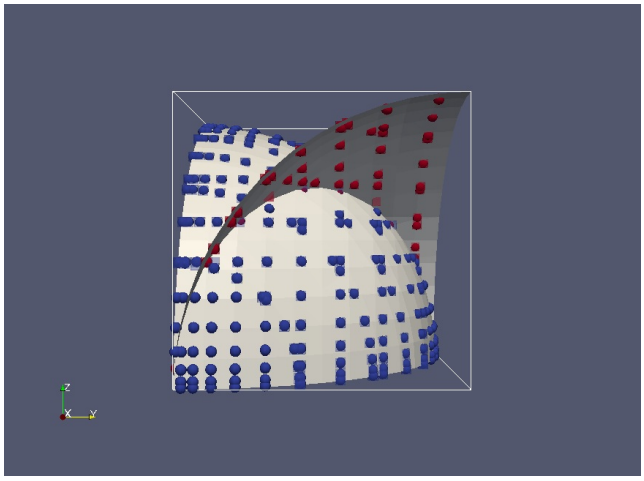

図 4.12　CutPos8, CutBid5 に格納 (+x 方向から)

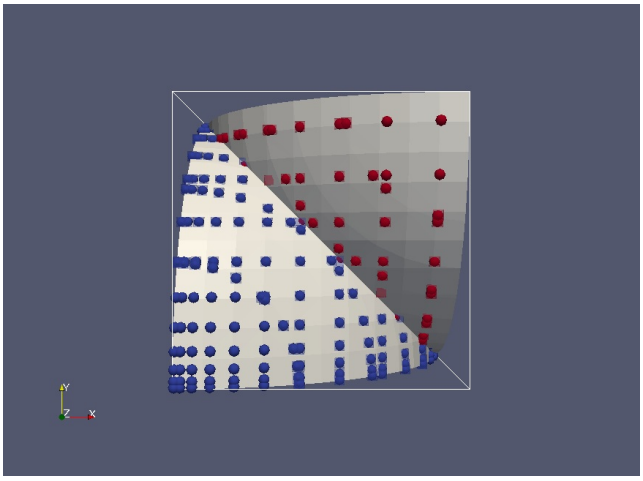

図 4.13　CutPos8, CutBid5 に格納 (+z 方向から)

4.2.4　Octree

同様な入力データに対して，Ocree 版の計算を行いました．ルートセルサイズを 2 × 2 × 2 とし，全ルートセルに対して 2 レベルのツリーを構築しました．したがって，全領域に対してリーフセルは 8 × 8 × 8 個存在し，そのピッチは 0.125 となります．

以下では，交点情報格納に CutPos32 型と CutBid8 型を用いた場合の結果のみを示しますが，CutPos8 型および CutBid5 型を用いた場合も同様な表示が得られています．

全セルについて計算

Octree プログラムでは，全セルについての計算を指定すると，各ツリーレベル毎に分けて交点情報を出力します．以下の各図は，ツリーレベル毎にプロットしたものです．

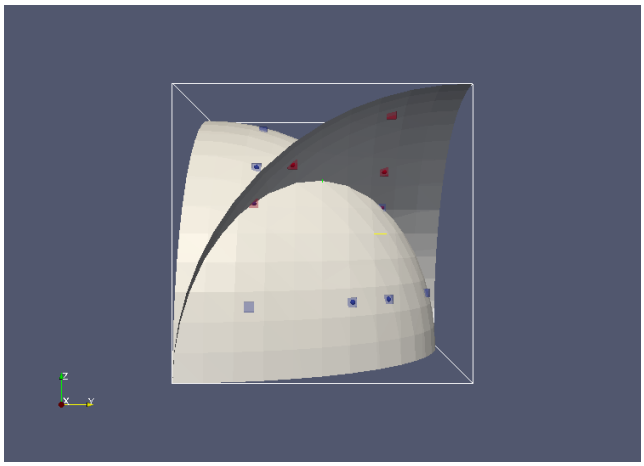

図 4.14　レベル 0 (+x 方向から)

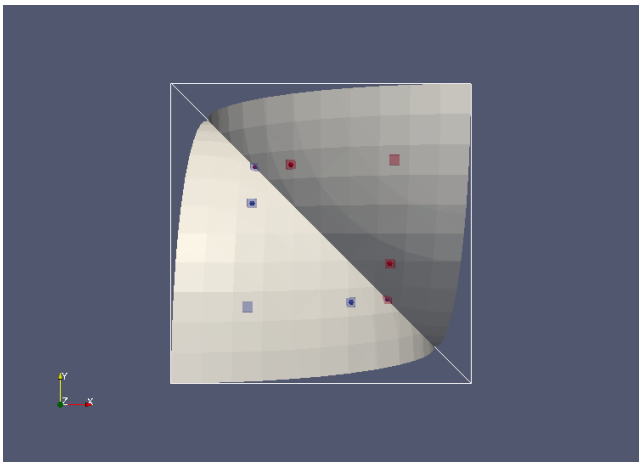

図 4.15　レベル 0 (+z 方向から)

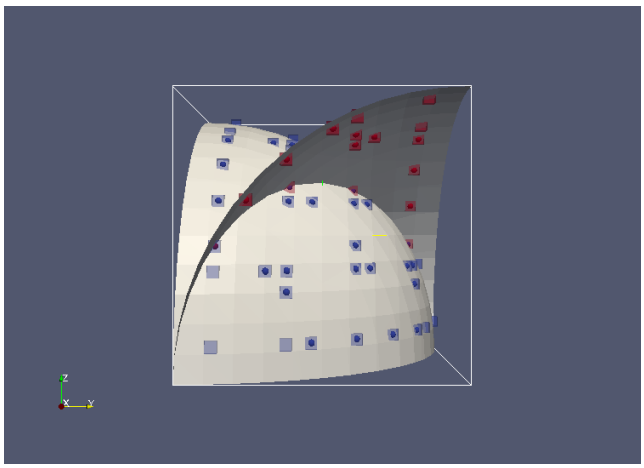

図 4.16　レベル 1 (+x 方向から)

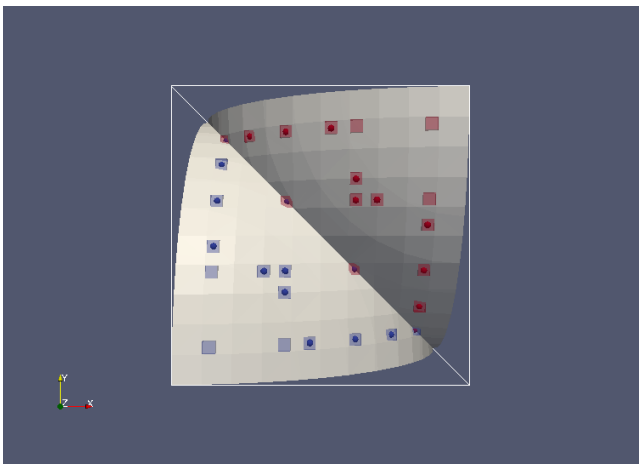

図 4.17　レベル 1 (+z 方向から)

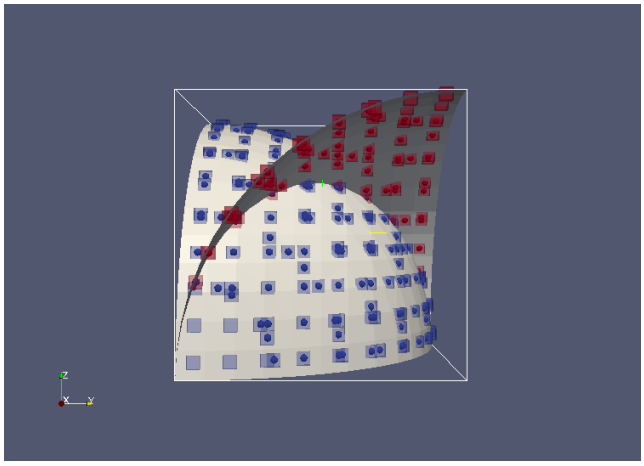

図 4.18　レベル 2 (+x 方向から)

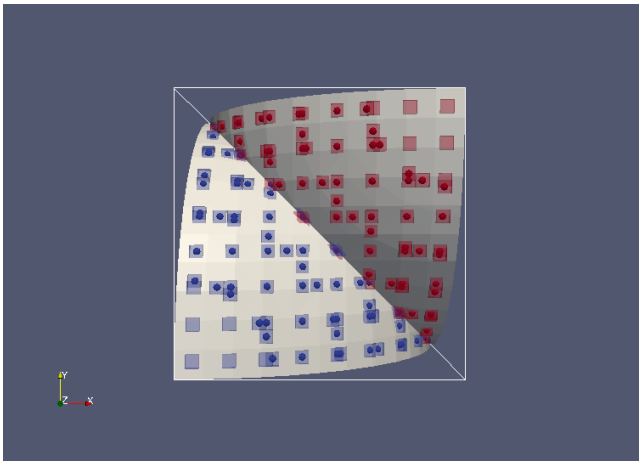

図 4.19　レベル 2 (+z 方向から)

リーフセルのみ計算

同様な条件で，計算対象をリーフセルのみとして計算しました．

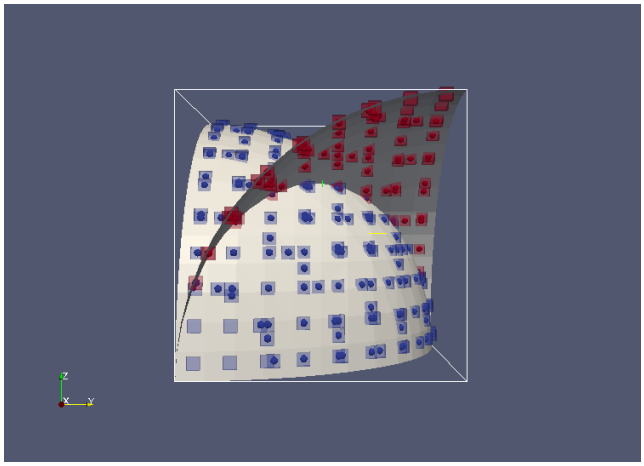

図 4.20　リーフセルのみ (+x 方向から)

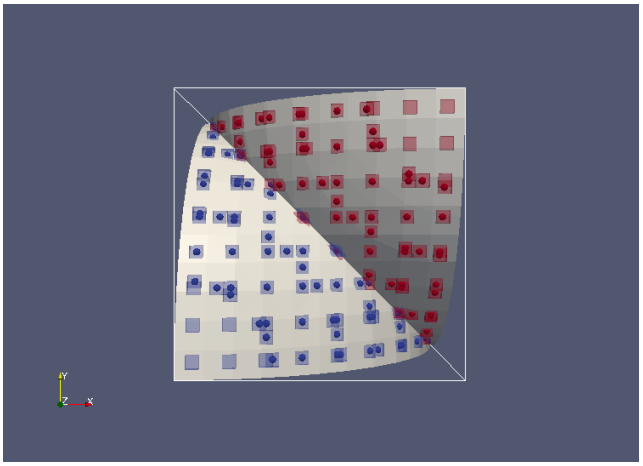

図 4.21　リーフセルのみ (+z 方向から)

4.3　性能検証

以下のような STL ファイル群を用いて，計算時間の測定を行いました (図 4.22).

lageA.stl　半径 0.6，ポリゴン数 16915，境界 ID 1
lageB.stl　半径 0.8，ポリゴン数 30103，境界 ID 2
lageC.stl　半径 1.0，ポリゴン数 47035，境界 ID 3
lageD.stl　半径 1.2，ポリゴン数 36298，境界 ID 4

単位立方体領域内で，ともに (0,0,0) を中心にもつ球面を構成するポリゴン群です．4 ファイルのポリゴン数の合計は 130,351 です．対象領域を $100 \times 100 \times 100$ に分割して作成した数値データの等値面からなるため，各ポリゴンの大きさのオーダは 0.01 程度になっています．

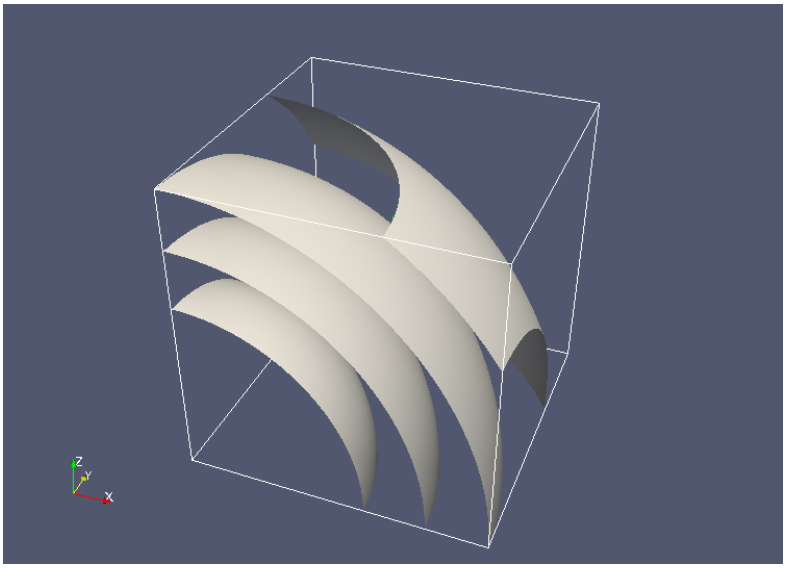

図 4.22　時間測定用 STL ファイル largeA.stl, largeB.stl, largeC.stl, largeD.stl

時間測定計算では，Intel コンパイラ 13.0.1 を用い，Polylib, Octree ライブラリも含めて，全て最適化オプション「-O3 -xHost」により生成された実行プログラムを使用しました．全ての計算ケースで，交点情報は CutPos32 型と CutBid8 型に格納しています．

以下の各表で「全体」と記してある項は，実際には交点情報計算関数内で，入力パラメータのチェックが終了した時点から呼び出し元へのリターン直前までの時間です．「Polylib 検索」と記してある項は，交点情報計算関数内から呼び出した Polylib のポリゴン検索メソッドの合計実行時間 (マルチスレッド実行時はスレッド間での最大値) です．いずれの数値も，単位は秒で，gettimeofday システムコールにより取得した経過時間をもとにしています．

交点情報計算関数の計算時間測定機能は，オプション「-DCUTLIB_TIMING」を付けて Cutlib をコンパイルすると有効になります．

4.3.1　測定結果

等間隔直交格子版，セル中心

セル数 $100 \times 100 \times 100$ について計算

表 4.1　等間隔直交格子，セル中心

スレッド数	1	2	4	8
全体	5.07	3.24	1.96	1.65
Polylib 検索	4.39	2.80	1.72	1.45

いずれのスレッド数での実行でも，Polylib 検索メソッド呼び出しにかかる時間は，全体の 86〜87% となっています．Polylib の検索メソッドの呼び出し回数の合計は，計算基準点数× 4(ポリゴングループ数) の 4,000,000 回です．

計算に使用した CPU(Intel i7) は，4 コア内蔵です．8 スレッド実行では，Hyper–Threading 機能を使うため，並列化効率が大きく低下しています．

以下に 4 スレッド実行時における，スレッドごとの Polylib 検索メソッド実行時間とその呼び出し回数を示します．

表 4.2　4 スレッド実行時のロードバランス

スレッド番号	0	1	2	3
実行時間	1.72	1.70	1.70	1.70
呼び出し回数	999108	1019020	986564	995308

上表の実行時間を見るとロードバランスがある程度保たれていることが分かります．呼び出し回数にばらつきが見られるのは，for ループの分割に「schedule(dynamic)」を指定して，実行時に動的に各スレッドにタスクを分配しているためです．ポリゴンデータは空間内に不均一に分布しているため，for ループに対して単純なブロック分割を行うと，ロードインバランスが発生します．

Octree リーフセル版

ルートセルサイズを $25 \times 25 \times 25$ として，全ルートセルに対して 2 レベルのツリーを構築しました．全領域に対してリーフセルは $100 \times 100 \times 100$ 個存在します．

Octree 版交点情報計算関数はスレッド並列化されていませんので，以降の表は全て 1 スレッドでの実行です．表中の「回数」は，その部分の呼び出し回数です．

表 4.3　Octree リーフセル版

	時間 (秒)	回数
全体	5.15	1
Polylib 検索	4.47	4000000

全体に対する Polylib 検索時間の割合は 86% となっています．Octree リーフセル版では，直交格子版と基本的に同じアルゴリズム (基準点毎に Polylib 検索メソッドの呼び出し) を使用しています．そのため，計算時間にも同様な傾向が見られます．

Octree 全セル版

Octree リーフセル版と同様な条件で計算を行いました．

Octree 全セル版では，計算速度向上のため，ルートセルでのみで Polylib 検索メソッドを使用しています．検索結果は，全ポリゴングループに対する検索結果をまとめたカスタムリストとして管理します．親セルはカスタムリストから各子セルに必要なポリゴン群を抽出したコピーリストを作成し，それを子セルに渡していきます．このアルゴリズムの効果を見るため，リーフセル版と同様に全セルで Polylib 検索メソッドを呼ぶ，テスト用関数 (CutInfoOctreeAllCell0) による計算時間測定も行いました．

表 4.4　Octree 全セル版

	時間 (秒)	回数
全体	0.98	1
Polylib 検索	0.12	62500

表 4.5　Octree 全セル版 (テスト用関数)

	時間 (秒)	回数
全体	6.38	1
Polylib 検索	5.46	4562500

Octree 全セル版では，今回実施した全テストのうち Polylib 検索の回数が最小 ($62500 = 25^3 \times 4$) となっているため，計算時間も最小になっています．全体に対する Polylib 検索時間の割合は 12% です．

また，テスト用関数による結果では，全体に対する Polylib 検索時間の割合は 85% となっており，他の版と同様な傾向が見られます．

4.3.2　考察

直交格子版

新たに施した OpenMP によるスレッド並列化の効果が実証できました．今回の計算は比較的小規模な物であったため，4 スレッド実行時の並列化効率は 64% にとどまりましたが，大規模な系に対する計算では 8 スレッドや 16 スレッドなどでも並列化の効果が期待できると思われます．

Octree 版

Octree 全セル版が，リーフセル版の結果を含む計算をしているにもかかわらず，リーフセル版より約 5 倍高速になっています．Polylib 側のさらなる高速化が望めない場合には，全セル版を元にリーフセル版を実装し直すべきかもしれません．また現バージョンでは未実装の OpenMP によるスレッド並列化も効果が期待できます．

Polylib 検索メソッド呼び出し回数削減チューニング案

現在の実装では，計算基準点毎のポリゴン探査領域が互いに重なりあっています．以下に示す直交格子版への改良案では，ひとつおきに計算基準点を巡ることにより，この無駄を省いています．

■現アルゴリズム

- 計算基準点は全セル中心点 (または全ノード点).
- 各基準点で，一組の Polylib 検索メソッド呼び出し，その結果を元に 6 方向 ($\pm x, \pm y, \pm z$) の交点情報を計算.

■新アルゴリズム

- 各方向ひとつおき (スタッガード) に計算基準点を巡る．巡回する計算基準点の総数は現アルゴリズムの 1/2.
- 各基準点で，一組の Polylib 検索メソッド呼び出し，その結果を元に，6 方向 ($\pm x, \pm y, \pm z$) に加えて，+x 側隣接基準点の-x 方向，-x 側隣接基準点の +x 方向，+y 側隣接基準点の-y 方向，-y 側隣接基準点の +y 方向，+z 側隣接基準点の-z 方向，-z 側隣接基準点の +z 方向，の計 12 組の交点情報を計算.

現アルゴリズムでは，計算基準点とその隣接基準点を結ぶ線分上に交点を持つポリゴンを検索して，基準点に最近接する交点の情報のみを格納しています．一方，新アルゴリズムでは，最近接交点だけでなく，最遠の交点も隣接基準点における逆方向の交点情報として格納します．

Cutlib 3.0.0 へのバージョンアップでは，この改良案を単純に実装するだけだとスレッド間でメモリアクセスの競合が発生してしまうこと，また，最大でも 2 倍の速度向上しか見込めないことなどの理由により，OpenMP によるスレッド並列化を優先しました．

第 5 章

クラス設計情報

本節では，Cutlib の内部設計について説明します．交点情報計算関数内で使用している計算アルゴリズムは，直交格子版，Octree リーフセル版では共通した手法を用いています．一方，Octree 全セル版では，高速化のため別の手法を採用しています．

5.1　直交格子版，Octree リーフセル版

計算基準点および計算基準点を巡る順番が違っている以外は，基本的に同じ計算手法です．各計算基準点では，以下のような計算を行います．

1. 計算基準点の周りにポリゴン探索範囲を定める．
2. ポリゴングループ毎に，Polylib の検索メソッドにより探索範囲内に一部でも含まれるポリゴンのリストを取得．
3. 交点情報の一時的な格納用配列 (長さ 6) を用意．
4. リスト中の各ポリゴンに対して，6 本の計算基準線分上に交点があるか調べる．交点があり，それが既に一時配列に記録されているものよりも基準点に近い場合には，一時配列に上書き記録していく．
5. 全リストの全ポリゴンに対する調査後，一時配列の内容を交点情報格納用配列に書き移す．交点位座標値の規格化はこの時点で行う．交点座標値の量子化格納が指定されている場合も，この時点で量子化を行う．

5.1.1　ポリゴン検索範囲

図 5.1 に，等間隔直交格子上のセル中心を計算基準点にした場合のポリゴン探索範囲を示します．計算基準点をノード点に置いた場合や，Octree リーフセル版も同様に，ポリゴン探索範囲は計算基準点を中心としたセルサイズの 2 倍の辺長を持つ直方体領域として定めます．不等間隔直交格子上のセル中心の場合には，検索対象直方体領域の一片の長さは，左右両隣接セルのセル中心間距離になります．

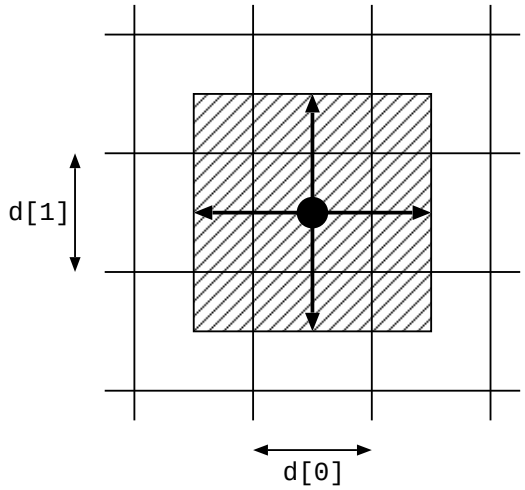

図 5.1　計算基準点 (黒丸)，基準線分 (矢印)，探査範囲 (斜線部分)

5.1.2　交点の存在判定

計算基準点の座標を (x, y, z)，セル間隔を (d_1, d_2, d_3) とすると，$\pm z$ 方向の基準線分上での交点の有無は，次のように 2 ステップで判定できます．

1. ポリゴンを xy 平面に射影して得られる三角形の内部に点 (x, y) が含まれることが必要

2. 実際に交点を計算して，その位置が $z - d_3$〜$z + d_3$ の間にある場合は採用

5.1.3　三角形の内点判定

Polylib 検索メソッドの返すリストの各要素には，ポリゴン頂点座標の他に，法線ベクトルも含まれています．この法線ベクトル情報を用いると，ある点が，ポリゴンを平面に射影した三角形の内部にあるかどうかの判定が，通常の判定法よりも高速に行えます．

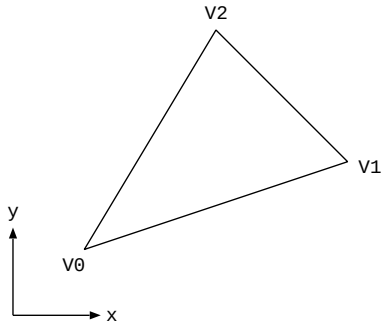

図 5.2　$n_z > 0$，$V_0 \to V_1 \to V_2 \to V_0$ は反時計周り

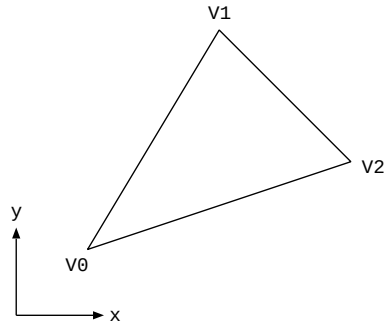

図 5.3　$n_z < 0$，$V_0 \to V_1 \to V_2 \to V_0$ は時計周り

2 次元ベクトルを矢印付きで記すことにすると，xy 平面内で，点 P が三角形 $V_0V_1V_2$ の外部にあるための条件は以下のようになります

$n_z > 0$ の場合:

$$\overrightarrow{V_0V_1} \times \overrightarrow{V_0P} < 0 \text{ または } \overrightarrow{V_1V_2} \times \overrightarrow{V_1P} < 0 \text{ または } \overrightarrow{V_2V_0} \times \overrightarrow{V_2P} < 0$$

$n_z < 0$ の場合:

$$\overrightarrow{V_0V_1} \times \overrightarrow{V_0P} > 0 \text{ または } \overrightarrow{V_1V_2} \times \overrightarrow{V_1P} > 0 \text{ または } \overrightarrow{V_2V_0} \times \overrightarrow{V_2P} > 0$$

この判定法では，1 回目の外積計算で，一部のポリゴンをふるい落すことができます[*1]．

5.1.4　交点座標の計算

三角形ポリゴンの各頂点の 3 次元位置ベクトルを $\boldsymbol{P}_0, \boldsymbol{P}_1, \boldsymbol{P}_2$，その法線ベクトルを $\boldsymbol{n}$ とします．このポリゴンを含む平面内の任意の点を $\boldsymbol{P}$ とすると，次の関係式が成立します．

$$\boldsymbol{n} \cdot (\boldsymbol{P} - \boldsymbol{P}_0) = 0$$

したがって，計算基準点 (x, y, z) に対して，$\pm x$ 方向の基準線分上の交点位置を X，$\pm y$ 方向の基準線分上の交点位置を Y，$\pm z$ 方向の基準線分上の交点位置を Z とすると，それぞれ以下のように求めることができます．

$$\begin{aligned} X &= (\boldsymbol{n} \cdot \boldsymbol{P}_0 - n_y y - n_z z) \,/\, n_x \\ Y &= (\boldsymbol{n} \cdot \boldsymbol{P}_0 - n_z z - n_x x) \,/\, n_y \\ Z &= (\boldsymbol{n} \cdot \boldsymbol{P}_0 - n_x x - n_y y) \,/\, n_z \end{aligned} \tag{5.1}$$

[*1] 法線ベクトル情報を使わない方法では，3 つの外積値の符号が全て等しい場合に内点と判定します．どのような場合でも，最低 2 回の外積計算が必要になります．

5.2　Octree 全セル版

Octree 全セル版では，ルートセルよりツリー構造をたどりながら，全ての階層のセルで交点情報を計算する必要があります．空間的に子セルはその親セルの内部に含まれるため，両者で別個に Polylib 検索メソッドを呼び出すのは非効率と思われます．そこで Octree 全セル版では，以下のようなアルゴリズムを採用しました．

1. Polylib 検索メソッドはルートセルでのみ利用する．ポリゴングループ毎の検索結果リストをひとつのリスト (カスタムリスト) にまとめる．
2. 以下をツリーを下りながら再帰的に繰り返す
   (1) そのセル内で，交点情報を計算，格納
   (2) 子セルが存在する場合は，子セルの探索範囲に含まれるポリゴンをカスタムリストからコピーし，子セル用のカスタムリストを作成
   (3) 子セルに制御を移動

5.2.1　ポリゴンのカスタムリスト

カスタムリストの要素用に，CutTriangle というクラスを定義しました．全ポリゴングループのポリゴンをまとめて処理するため，CutTriangle の属性には，Triangle クラス (へのポインタ) の他に，境界 ID を加えてあります．また，子セルの探査範囲に含まれるかの判定を容易にするために，三角形ポリゴンの Binding Box 情報 (最小座標値，最大座標値) も属性に加えました．

5.2.2　カスタムポリゴンリストのコピー

子セルへ渡すカスタムリストは，子セル毎に作成します．親セルのカスタムリストの各ポリゴンのうち，子セルの探査領域に一部でも含まれているものを子セルのカスタムリストへ (CutTriangle のポインタを) コピーしていきます．

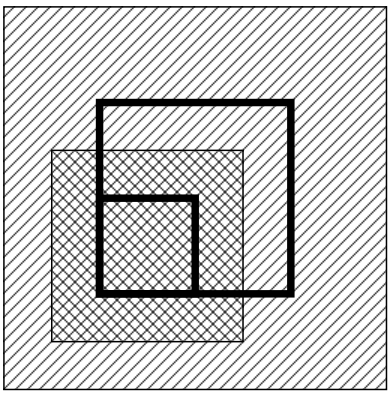

図 5.4　親セルの探査領域 (外側斜線部分)，子セルの探査領域 (内側斜線部分)

5.2.3　交点情報の計算

各セルにおけるポリゴン交点の存在判定，交点情報の計算方法については，Octree リーフセル版と同様な手法を用いています．

第 6 章

アップデート情報

本ユーザガイドのアップデート情報について記します．

6.1　アップデート情報

- Version 3.0.0-beta
  - STL ファイル中のポリゴンデータの取扱方法を変更．コンパイルオプション「-DIGNORE_NORMAL_DIRECTION」「-DIGNORE_STL_NORMAL」の廃止．ポリゴンデータ修復関数 RepairPolygonData の追加
  - グリッド情報アクセッサクラスの導入．直交格子版関数のインタフェースおよび名称の変更
  - 配列ラッパクラスの 3 次元インデックスに負の値も指定可能
  - 直交格子版関数の OpenMP スレッド並列化
  - マニュアルの改修
  - Polylib 2.2，TextParser 1.1 対応
- Version 2.0.3
  - コンパイルオプションに「-DIGNORE_NORMAL_DIRECTION」を指定すると，ポリゴンの表裏が統一されていることを仮定せずに計算
  - コンパイルオプションに「-DIGNORE_STL_NORMAL」を指定すると，STL ファイル中のポリゴン法線データを信用せず，毎回法線を計算
  - マニュアルの改修
  - Makefile を Polylib2.0.3 のコンパイルへ対応
- Version 2.0.2
  - マニュアルの改修
  - Makefile を Polylib2.0.2 のコンパイルへ対応
- Version 2.0.1
  - マニュアルの整備
  - 計算対象となるポリゴングループとその境界 ID を Polylib 初期化ファイルから自動読み取り (Polylib2.0.1)
- Version 2.0.0
  - インターフェース変更 (引数 Boundaries* bList の廃止)
  - Makefile で，Octree 版関数作成の有無を選択可能
  - 交点探査方向順を変更 (-x, +x, -y, +y, -z, +z に)
- Version 1.0.0
  - 初版リリース