Cutlib 3.0 Introduction

2013-09-15

{keno, soichiro.suzuki}@riken.jp

AICS, RIKEN

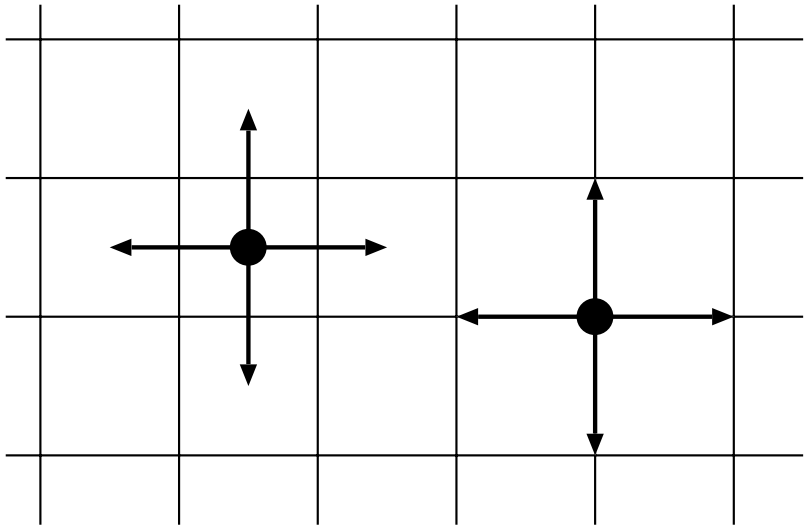
Cutlib ?

- ポリゴンデータを対象に，背景格子との交点を計算する
  - 計算基準点について，６方向の交点を計算．同時に交点ポリゴンのID情報を境界IDとして保持
  - 計算基準点は，セルノードとセルセンターの２つ
- 格子データ
  - 直交等間隔と八分木
- 理研VCAD-PJで開発，AICSでメンテナンス
- ライセンス
  - Ver. 2.0　　VCADライセンス
  - Ver.3.0以降　修正BSDライセンス

Cutlibの特徴

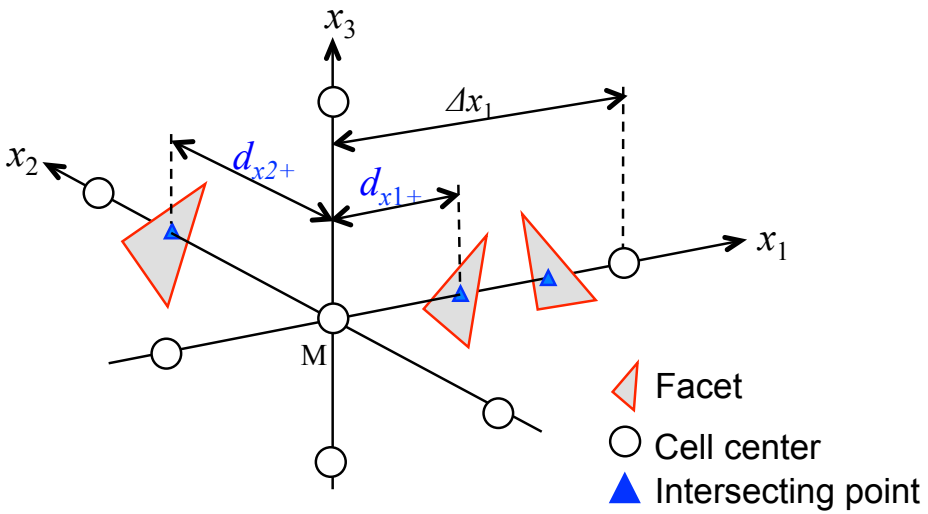
- ポリゴンデータの管理はPolylibの利用を想定
- TextPaser対応
- 格納データ型には，省メモリ版もあり
- Fortranからもアクセス可能
- OpenMPによるスレッド並列済み
- ポリゴンデータの法線情報の再計算
  - 法線情報の整合性をとる

計算基準点と計算基準線分



- 黒丸が計算基準点.
- 隣接する計算基準点間を結ぶ線分を計算基準線分とする（図中の矢印部分）.
- 左はセルセンター，右がセルノードの場合.

交点計算と境界ID



$$0 \leqq d \leqq \Delta x_i$$

- 交差判定．点Mが計算基準点で，6方向の計算基準線分に対して交点計算を行う．
- 一つの計算基準線分上に複数の交点がある場合，最も近い交点を記録する．
- 交点距離は格子幅で無次元化し，[0.0, 1.0]．交点がなければ，1.0
- 境界IDの値域は8bit内，つまり[0, 255]．ただし，ID=0は境界が無いことを示す．
- オプションで交点位置における法線ベクトルを記録可能．

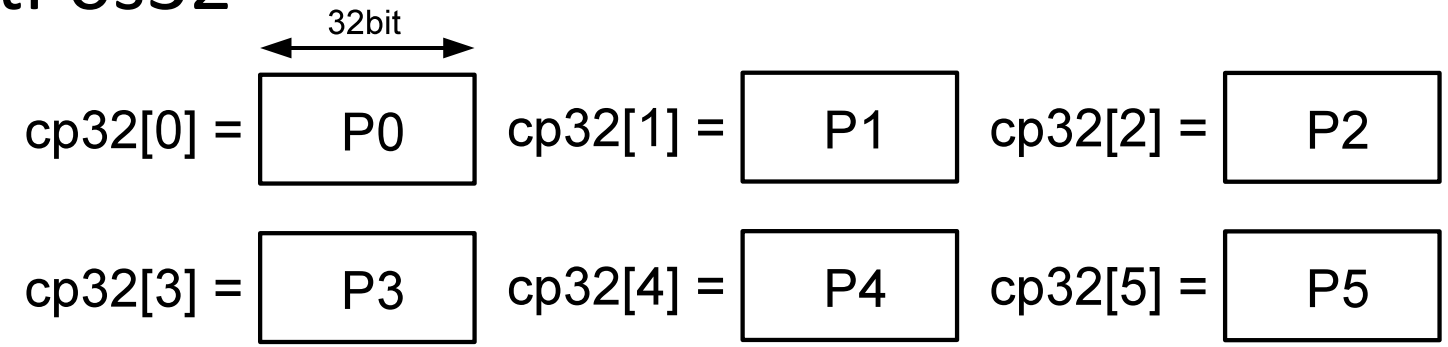
データ型

- 交点座標基本データ型
  - CutPos32型　交点座標をfloatとして格納
  - CutPos8型　　交点座標を8bit量子化

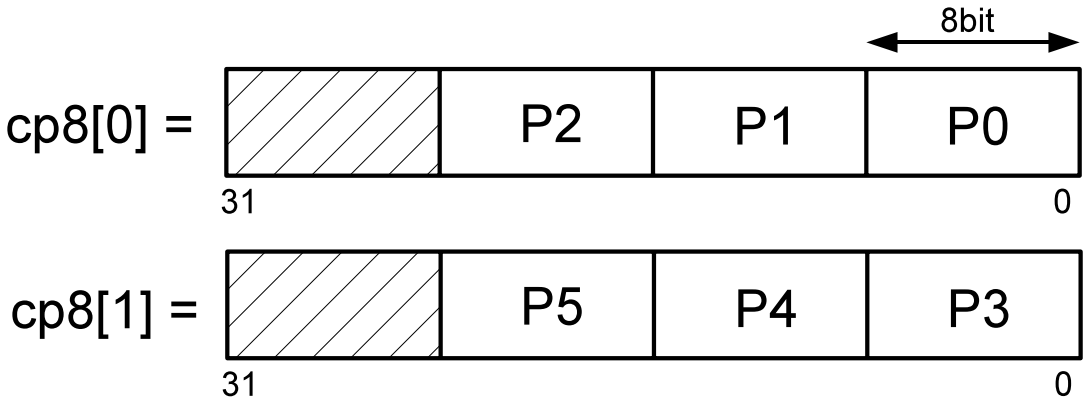
- 境界ID基本データ型
  - CutBid8型　　0～255の境界IDを扱う
  - CutBid5型　　0～31の境界IDを扱う

交点座標基本データ型

- CutPos32

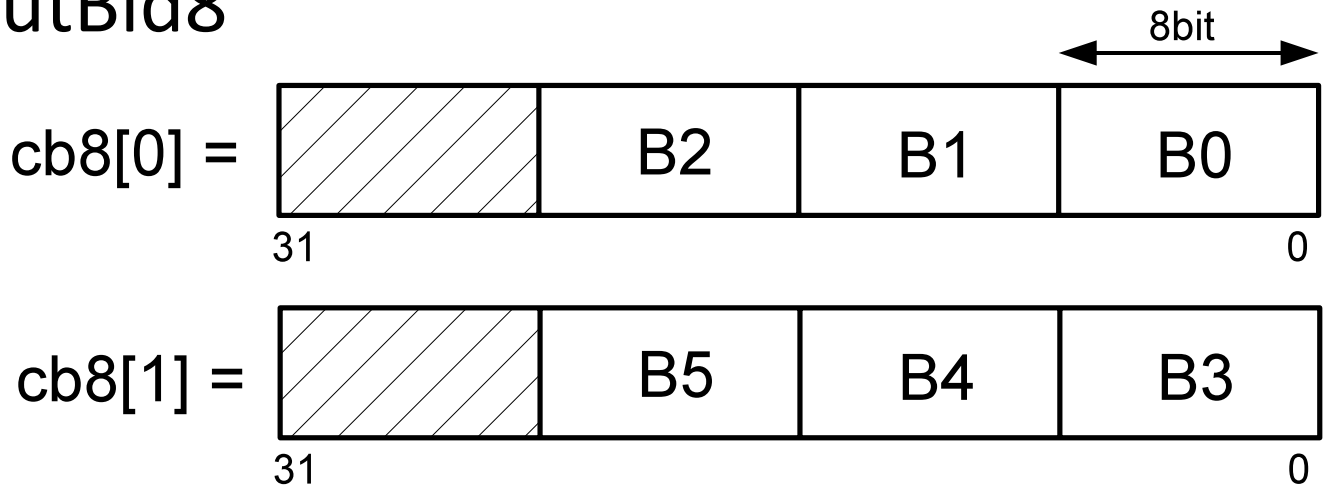


- CutPos8

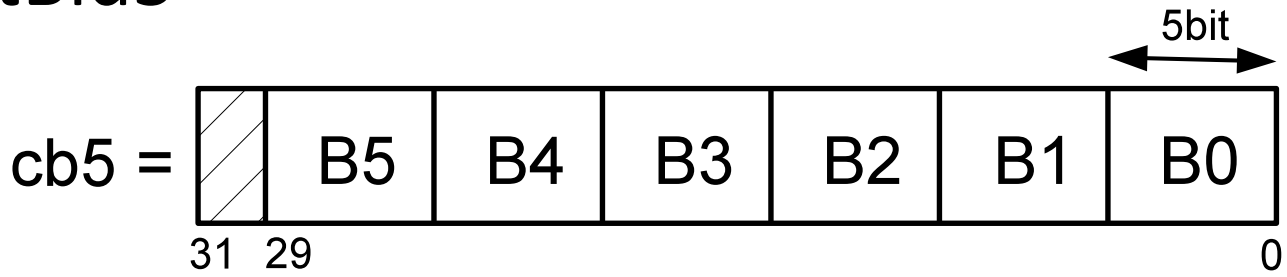


境界ID基本データ型

- CutBid8



- CutBid5



インタフェース

- STLファイル法線ベクトル情報の利用
  - 従来のコードでは、コンパイル時にMakefile内で指定
  - 現コードでは常に利用(必要なら前処理としてポリゴンデータの修復を行う)
- CutBidArray, CutPosArrayクラスのコンストラクタで、3次元領域の下限値が指定可能
- グリッドアクセッサ(GridAccessor)クラスの導入
- 従来コードのインタフェースは、現インタフェースにラッパをかぶせた「互換インタフェース」として残す

STLファイル法線ベクトル情報の利用

- 前提条件
  1. 法線ベクトルの向きと3節点の並び順に整合性があるか
  2. 法線ベクトルの値は信頼できるか

- STLファイルの法線情報を常に利用する
  – そのかわり、前提条件を満たすようにポリゴンデータを修正するユーティリティ関数を提供

ポリゴンデータ修正関数

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

- 各ポリゴンの3節点の並び順の変更と法線ベクトルの再計算を行う
- Polylibの内部データを上書きする
- Polylibのセーブメソッドを呼び出せば、修正後のポリゴンデータをそのままSTLに保存可能

CutBidArray, CutPosArrayコンストラクタ

- 現コードでは、領域サイズ(nx,ny,nz)を指定

  3次元インデックス(I,j,k) → 1次元インデックス(ijk)

  ```
  ijk = i + j * nx + k * nx * ny
  ```

- 新コードでは、領域の下限(sx, sy, sz)と上限(ex, ey, ez)による指定方法を追加

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

  仮想セルがある場合のインデックス変換が不要に

従来コードでは size_t

3次元インデックスの型: int (= int32_t)
1次元インデックスの型: size_t (= uint64_t)

グリッドアクセッサ(GridAccessor)クラス

- セル中心基準orグリッド点基準、等間隔or不等間隔格子、これらを統一的に扱う
- 3次元インデクスを渡すと、交点計算の基準点座標と6方向の計算基準線分長を返す

```
CutlibReturn CalcCutInfo(const Polylib* pl,
                         const GridAccesor* grid,
                         CutPosArray* cutPos,
                         CutBidArray* cutBid);
```

GridAccessor基底クラス

```
class GridAccessor {

public:
  /// コンストラクタ.
  GridAccessor() {}

  /// デストラクタ.
  virtual ~GridAccessor() {}

  /// Polylib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] length  6方向の計算基準線分の長さ
  virtual void getSearchRegion(int i, int j, int k,
                               float center[3],
                               float length[6]) = 0;

};
```

GridAccessorの継承クラスで、このメソッドを実装する

他に、3次元インデックスの範囲をチェックするメソッドのインタフェースを追加予定

例: セル中心(等間隔格子)

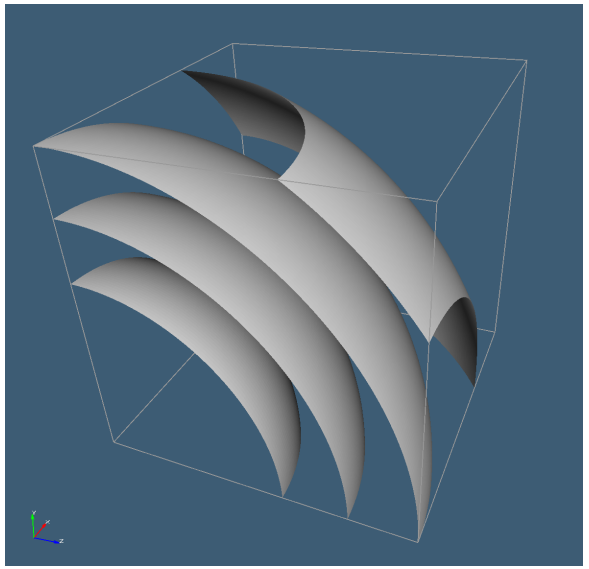
```
class CellCenter : public GridAccessor {

  int n_[3];    // 領域サイズ
  float o_[3];  // オフセット座標
  float p_;     // セルピッチ

public:

  CellCenter(int n[3], float o[3], float p) {
    n_[0] = n[0]; n_[1] = n[1]; n_[2] = n[2];
    o_[0] = o[0]; o_[1] = o[1]; o_[2] = o[2];
    p_ = p;
  }

  void getSearchRegion(int i, int j, int k,
                       float center[3], float length[6]) {
    center[0] = (i + 0.5) * p_ + o_[0];
    center[1] = (j + 0.5) * p_ + o_[1];
    center[2] = (k + 0.5) * p_ + o_[2];
    for (int d = 0; d < 6; d++) length[d] = p_;
  }
};
```

OpenMP並列化

- 交点計算基準点単位での処理をスレッドに割り当てる
  - スレッド数に制限なし(schedule(dynamic), collapse(3)でループ分割)
  - Polylib検索メソッドはスレッドセーフ

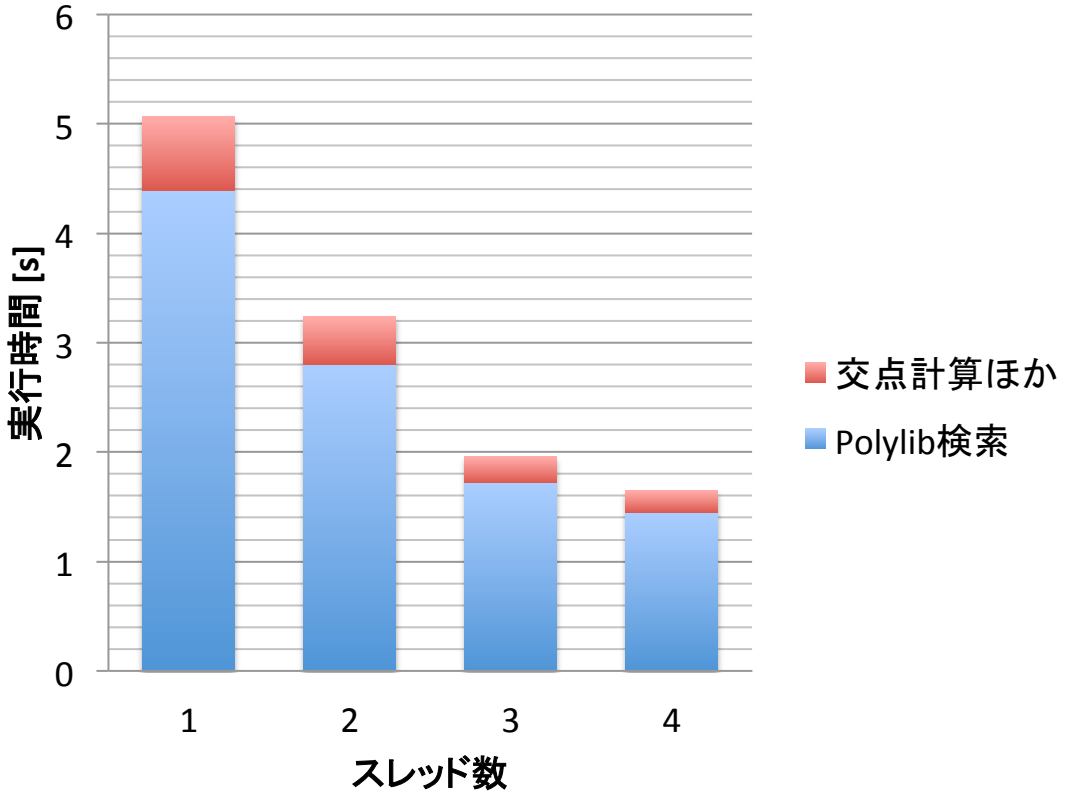
- 評価問題
  - 並列計算でも1プロセスあたり数百万セルが性能の上で合理的
    - 単位立方体領域の100万セル
  - ポリゴンデータ
    - 4グループ，計13万ポリゴン

- テスト環境
  - Intel Core i7@2.3GHz, 4cores，16GB
  - Intel Compiler 13.0.1, -O3 -xHost



OpenMP並列性能（1）

- Polylib検索メソッドは計算基準点数xポリゴングループ数（100^3 x 4）
- Polylib検索メソッドの呼び出しコストは，どのスレッド数の実行でも86%程度
- 比較的小規模な系での並列効率は，4スレッド時に64%程度
- 大規模な系で，8/16スレッドではもう少し改善されるはず



OpenMP並列性能（２）

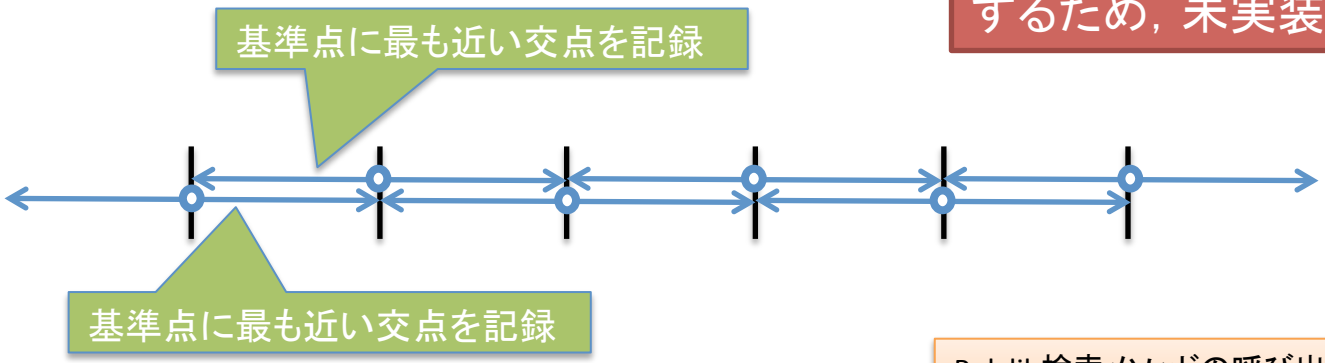
- 4スレッド実行時におけるPolylib検索メソッドの実行時間
- 回数のばらつきは，schedule(dynamic)によるもの
- dynamicスケジューリングにより，ポリゴンデータが空間的に偏在していても，そこそこのロードバランスを確保している

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

Polylib検索メソッド回数の低減

- 従来の実装

たかだか2倍の高速化，
スレッド間で競合が発生
するため，未実装



Polylib検索メソッドの呼び出し回数は，
計算基準点数xポリゴングループ数

- Ver 3.0 の実装

Polylib検索メソッドを、ひとつ置きの基準点から呼び出す >> 回数半減