FDPS仕様書

谷川衝、岩澤全規、細野七月、似鳥啓吾、村主崇行、牧野淳一郎

目 次

1　この文書の概要

この文書は大規模並列粒子シミュレーションの開発を支援する Framework for Developing Particle Simulator (FDPS) の仕様書である。この文書は以下のような構成となっている。節 2 では、FDPS の概要を述べる。FDPS の基本的な考えかたや動作が記述されている。節 3 では、FDPS のファイル構成を述べる。ディレクトリ構成とどのようなファイルがどこのディレクトリにあるかが記述されている。節 4 には、FDPS の API を使用したコードをコンパイルする時にどのようなマクロを用いればよいかが記述されている。節 5 には、FDPS で使用されている名前空間のことが記述されている。節 6 には、FDPS で独自に定義されているデータ型が記述されている。節 7 には、FDPS を使用したコードを記述する際にユーザーが定義する必要があるクラスや関数オブジェクトについて記述されている。節 8 には、FDPS を開始するときと終了するときに呼ぶ必要のある API について記述されている。節 9 には、FDPS にあるモジュールについて記述されている。節 10, 11, 12 には、エラーメッセージ、よく知られているバグ、FDPS の限界について記述されている。節 13 には、ユーザーサポートに関する情報が記述されている。最後に節 14 には、FDPS のライセンスに関する情報が記述されている。

2 FDPS 概要

この節ではFDPSの概要を記述する。FDPSの開発目的、FDPSの基本的な考えかた、FDPSを使用して作成したコードの動作について概説する。

2.1 開発目的

粒子シミュレーションは、重力 N 体シミュレーション、SPH シミュレーション、渦糸法、MPS 法、分子動力学シミュレーションなど理学工学の様々な分野で使用されている。より大きい空間スケール、より高い空間分解能 (または質量分解能)、より長い時間スケールの物理現象を追跡するために、高性能な粒子シミュレーションコードへの要請はますます強くなっている。

高性能な粒子シミュレーションコードを組むためには、シミュレーションコードの大規模並列化を避けることはできない。粒子シミュレーションコードの大規模並列化をする際には、ロードバランスのため動的領域分割、領域分割に合わせた粒子交換、ノード間通信の削減と最適化、キャッシュ利用効率の向上、SIMD ユニット利用効率の向上、アクセラレータへの対応など、数多くの困難な処理を行う必要がある。現在、研究グループは個別にこれらの処理へ対応している。

しかし、上記の処理は粒子シミュレーション共通のものである。FDPS の開発目的は、これらの処理を高速に行うライブラリを提供し、大規模並列化への対応に追われていた研究者の負担を軽くすることである。FDPS を使うことで、研究者がよりクリエイティブな仕事に専念できるようになれば、幸いである。

2.2 基本的な考えかた

ここでは FDPS の基本的な考えかたについて記述する。

2.2.1 大規模並列粒子シミュレーションの手順

まず FDPS において、大規模並列粒子シミュレーションがどのような手順で行われることを想定しているかを記述する。粒子シミュレーションは、以下のような微分方程式を時間発展させるものである。

$$\frac{d\boldsymbol{u}_i}{dt} = \sum_j f(\boldsymbol{u}_i, \boldsymbol{u}_j) + \sum_s g(\boldsymbol{u}_i, \boldsymbol{v}_s) \tag{1}$$

ここで $\boldsymbol{u}_i$ は粒子 i の物理量ベクトルであり、この物理量には質量、位置、速度など粒子が持つあらゆる物理量が含まれる。関数 f は粒子 j から粒子 i への作用を規定する。以後、作用を受ける粒子を i 粒子、作用を与える粒子を j 粒子と呼ぶことにする。$\boldsymbol{v}_s$ は i 粒子から十分遠方にある粒子を 1 つの粒子としてまとめた粒子 (以後、この粒子を超粒子と呼ぶ) の物理量ベクトルである。関数 g は超粒子から i 粒子への作用を規定する。式 (1) の第 2 項は、重力

やクーロン力など無限遠まで到達する長距離力の場合はゼロではない。しかし流体の圧力のような短距離力はゼロである。

大規模並列化された粒子シミュレーションコードは以下の手順で式 (1) を時間発展させる。ここではデータの入出力や初期化は省略している。

1. 以下の 2 段階の手順でどのプロセスがどの粒子の式 (1) を時間発展させるか決める。
   (a) プロセスの間でロードバランスを取れるように、シミュレーションで扱っている空間の領域を分割し、各プロセスの担当領域を決める (領域分割)。
   (b) 各プロセスが、自分の担当する領域に存在する全粒子の物理量ベクトル $\boldsymbol{u}_i$ を持つように、他のプロセスと物理量ベクトル $\boldsymbol{u}_i$ を交換する (粒子交換)。
2. 各プロセスは、自分の担当する全粒子の式 (1) の右辺を計算するのに必要な j 粒子の物理量ベクトル $\boldsymbol{u}_j$ と超粒子の物理量ベクトル $\boldsymbol{v}_s$ を他のプロセスと通信することで集めて、j 粒子のリストと超粒子のリスト (まとめて相互作用リストと呼ぶ) を作る (相互作用リストの作成)。
3. 各プロセスは自分の担当する全粒子に対して、式 (1) の右辺を計算し、$d\boldsymbol{u}_i/dt$ を求める (相互作用の計算)。
4. 各プロセスは、自分の担当する全粒子の物理量ベクトル $\boldsymbol{u}_i$ とその時間導関数 $d\boldsymbol{u}_i/dt$ を使って、全粒子の時間積分を実行し、次の時刻の物理量ベクトル $\boldsymbol{u}_i$ を求める (時間積分)。
5. 手順 1 に戻る。

2.2.2 ユーザーと FDPS の役割分担

FDPS は、プロセス間の通信が発生する処理は FDPS が担当し、プロセス間の通信の発生しない処理はユーザーが担当するという役割分担を基本としている。従って、前節に挙げた、領域分割・粒子交換 (項目 1)・相互作用リストの作成 (項目 2) を FDPS が、相互作用の計算 (項目 3)・時間積分 (項目 4) をユーザーが担当することになる。ユーザーは FDPS の API を呼び出すだけで、大規模並列化に関わる煩雑な処理を避けつつ、高性能な任意の相互作用の粒子シミュレーションコードを手に入れることができる。

2.2.3 ユーザーのやること

ユーザーが FDPS を使って粒子シミュレーションコードを作成するときにやることは以下の項目である。

- 粒子の定義 (節 7)。粒子の持つ物理量 (式 (1) で言えば $\boldsymbol{u}_i$) の指定。例えば質量、位置、速度、加速度、元素組成、粒子サイズ、など。

- 相互作用の定義 (節 7)。粒子間の相互作用 (式 (1) で言えばで関数 f, g) を指定。例えば、重力、クーロン力、圧力、など。
- FDPS の API の呼出 (節 8, 9)

2.2.4 補足

式 (1) の右辺は 2 粒子間相互作用の重ね合わせである。従って、FDPS の API を呼ぶだけでは、3 つ以上の粒子の間の相互作用の計算を行うことはできない。しかし、FDPS はネイバーリストを返す API を用意している。ネイバーリストを用いれば、ユーザーはプロセス間の通信の処理をすることなく、このような相互作用の計算をできる。

節 2.2.1 で示した手順は、全粒子が同じ時間刻みを持っている。そのため、FDPS の API を呼び出すだけでは、独立時間刻みで時間積分を効率的に行うことができない。しかし、上と同じくネイバーリストを返す API があるため、Particle Particle Particle Tree 法を用いて独立時間刻みを実装することは可能であろう。

2.3 コードの動作

ここでは FDPS を使用して作成したコードの動作の概略を記述する。このコードには、4 つのモジュールがあることになる。3 つは FDPS のモジュールで、1 つはユーザー定義のモジュールである。まとめると以下のようになる。

- 領域クラス：全プロセスが担当する領域の情報と、領域分割を行う API を持つ
- 粒子群クラス：全粒子の情報と、プロセスの間での粒子交換を行う API を持つ
- 相互作用ツリークラス：粒子分布から作られたツリー構造と、相互作用リストを作成する API を持つ
- ユーザー定義クラス：ある 1 粒子を定義するクラス、粒子間の相互作用を定義する関数オブジェクトを持つ

これら 4 つのモジュールの間で情報がやり取りされる。これは図 1 で概観できる。図 1 に示された情報のやりとりは、節 2.2.1 に記述された手順 1 から 3 と、これらの手順以前に行われる手順 (手順 0 とする) に対応する。以下はこれらの手順の詳細な記述である。

0. ユーザー定義クラスのうち 1 粒子を定義するクラスが粒子群クラスへ、粒子間の相互作用を定義する関数オブジェクトが相互作用ツリークラスへ渡される
1. 以下の 2 段階でロードバランスを取る
   (a) 領域クラスが持つ領域分割の API が呼ばれる。このとき粒子情報が粒子群クラスから領域クラスへ渡される (赤字と赤矢印)

(b) 粒子群クラスが持つ粒子交換の API が呼ばれる。このとき領域情報が領域クラスから粒子群クラスへ渡される (青字と青矢印)

2. 相互作用ツリークラスが持つ相互作用リストを作成する API が呼ばれる。このとき領域情報が領域クラスから相互作用ツリークラスへ、粒子情報が粒子群クラスから相互作用ツリークラスへ渡される (緑字と緑矢印)

3. 相互作用ツリーククラスが持つ相互作用を定義した関数オブジェクトを呼び出す API が呼ばれる。相互作用計算が実行され、相互作用計算の結果が相互作用ツリークラスから粒子群クラスへ渡される (灰色の字と灰色矢印)

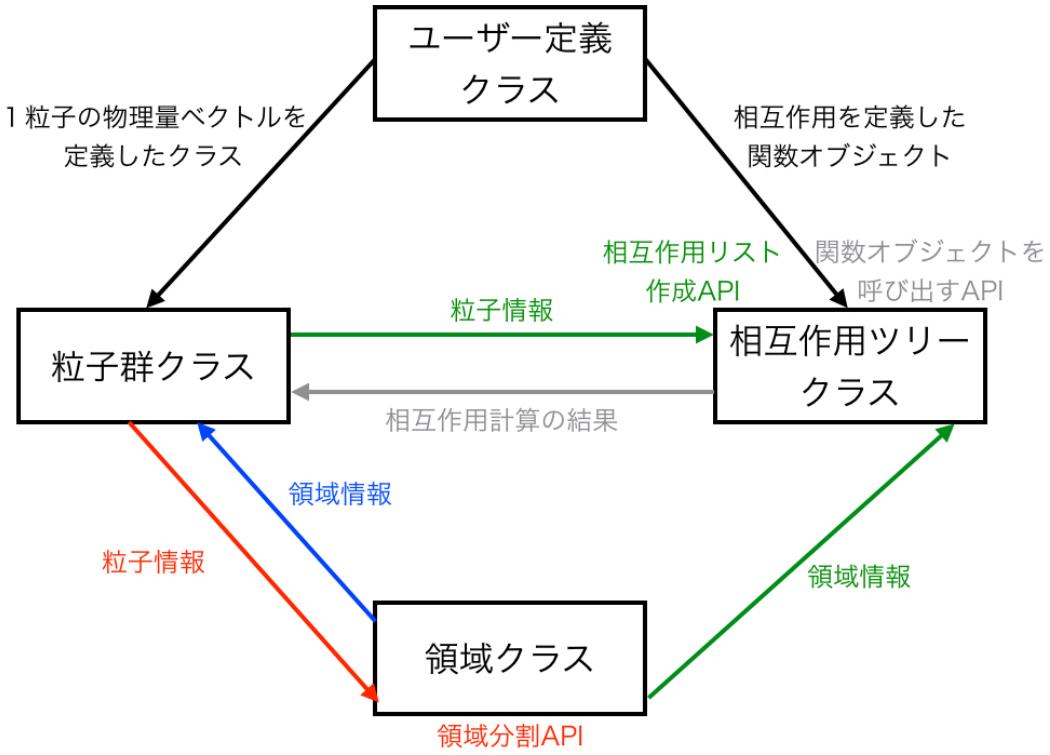

図 1: モジュールインターフェースの模式図

3　ファイル構成

3.1　概要

ここではFDPSのファイル構成について記述する。ドキュメント、ソースファイル、テストコード、サンプルコードの順に記述する。

3.2　ドキュメント

ドキュメント関係のファイルはディレクトリdocの下にある。チュートリアルがdoc_tutorial.pdfであり、仕様書がdoc_specs.pdfである。

3.3　ソースファイル

ソースファイルはディレクトリsrcの下にある。標準機能関係のソースファイルはsrcの直下にある。ディレクトリsrcの直下にあるヘッダファイルparticle_simulator.hppをソースファイルにインクルードすれば、FDPSの標準機能を使用できるようになる。

3.3.1　拡張機能

拡張機能関係のソースファイルはディレクトリsrcの直下のディレクトリにそれぞれ入っている。拡張機能にはParticle Meshがある。

3.3.1.1　Particle Mesh

Particle Meshのソースファイルはディレクトリsrc/particle_meshの下にある。ここでMakefileを編集して、makeを実行すると、ヘッダファイルparticle_mesh_class.hppとライブラリlibpm.aができる。このヘッダファイルをインクルードし、このライブラリをリンクづけすれば、Particle Meshの機能を使用できるようになる。

3.4　テストコード

テストコードはディレクトリtestsの下にある。ディレクトリtestsにカレントディレクトリを移し、make checkを実行するとテストスィートが動作する。

3.5　サンプルコード

サンプルコードはディレクトリsampleの下にある。サンプルコードは２つ用意されており、重力N体シミュレーションとSPHシミュレーションである。

3.5.1 重力 N 体シミュレーション

ディレクトリ sample/nbody の下にソースファイルがある。サンプルコードの実行方法はチュートリアルを参照のこと。

3.5.2 SPH シミュレーション

ディレクトリ sample/sph の下にソースファイルがある。サンプルコードの実行方法はチュートリアルを参照のこと。

4　コンパイル時のマクロによる選択

4.1　概要

FDPS では、座標系や並列処理の有無を選択できる。この選択はコンパイル時のマクロの定義によってなされる。以下、選択の方法について座標系、並列処理の有無の順に記述する。

4.2　座標系

4.2.1　概要

座標系は直角座標系 3 次元と直角座標系 2 次元の選択ができる。以下、それらの選択方法について述べる。

4.2.2　直角座標系 3 次元

デフォルトは直角座標系 3 次元である。なにも行わなくても直角座標系 3 次元となる。

4.2.3　直角座標系 2 次元

コンパイル時に PARTICLE_SIMULATOR_TWO_DIMENSION をマクロ定義すると直交座標系 2 次元となる。

4.3　並列処理

4.3.1　概要

並列処理に関しては、OpenMP の使用／不使用、MPI の使用／不使用を選択できる。以下、選択の仕方について記述する。

4.3.2　OpenMP の使用

デフォルトは OpenMP 不使用である。使用する場合は、PARTICLE_SIMULATOR_THREAD_PARALLEL をマクロ定義すればよい。GCC コンパイラの場合はコンパイラオプションに-fopenmp をつける必要がある。

4.3.3　MPI の使用

デフォルトは MPI 不使用である。使用する場合は、PARTICLE_SIMULATOR_THREAD_PARALLEL をマクロ定義すればよい。

5 名前空間

5.1 概要

本節では、名前空間の構造について述べる。FDPS は ParticleSimulator という名前空間で囲まれている。以下では、ParticleSimulator 直下にある機能と、ParticleSimulator にネストされている名前空間について述べる。

5.2 ParticleSimulator

FDPS の標準機能すべては名前空間 ParticleSimulator の直下にある。

名前空間 ParticleSimulator は以下のように省略されており、この文書におけるあとの記述でもこの省略形を採用する。

```
namespace PS = ParticleSimulator;
```

名前空間 ParticleSimulator の下にはいくつかの名前空間が拡張機能毎にネストされている。拡張機能には ParticleMesh がある。以下では拡張機能の名前空間について記述する。

5.2.1 ParticleMesh

Particle Mesh の機能は名前空間 PartcielMesh に囲まれており、名前空間 ParticleMesh は名前空間 ParticleSimulator の直下にネストされている。また、ParticleMesh は PM と省略されている。これらをまとめると以下のようになっている。

```
ParticleSimulator {
    ParticleMesh {
    }
    namespace PM = ParticleMesh;
}
```

以後、この文書では省略形の PM を用いて記述する。

6 データ型

6.1 概要

FDPS では独自の整数型、実数型、ベクトル型、対称行列型、PS::SEARCH_MODE 型、列挙型が定義されている。整数型、実数型、ベクトル型、対称行列型に関しては必ずしもここに挙げるものを用いる必要はないが、これらを用いることを推奨する。PS::SEARCH_MODE 型、列挙型は必ず用いる必要がある。以下、整数型、実数型、ベクトル型、対称行列型、PS::SEARCH_MODE 型、列挙型の順に記述する。

6.2 整数型

6.2.1 概要

整数型には PS::S32, PS::S64, PS::U32, PS::U64 がある。以下、順にこれらを記述する。

6.2.2 PS::S32

PS::S32 は以下のように定義されている。すなわち 32bit の符号付き整数である。

ソースコード 1: S32

```
1 namespace ParticleSimulator {
2     typedef int S32;
3 }
```

ただし、GCC コンパイラと K コンパイラでのみ 32bit であることが保証されている。

6.2.3 PS::S64

PS::S64 は以下のように定義されている。すなわち 64bit の符号付き整数である。

ソースコード 2: S64

```
1 namespace ParticleSimulator {
2     typedef long long int S64;
3 }
```

ただし、GCC コンパイラと K コンパイラでのみ 64bit であることが保証されている。

6.2.4 PS::U32

PS::U32 は以下のように定義されている。すなわち 32bit の符号なし整数である。

ソースコード 3: U32

```
1 namespace ParticleSimulator {
2     typedef unsigned int U32;
3 }
```

ただし、GCC コンパイラと K コンパイラでのみ 32bit であることが保証されている。

6.2.5 PS::U64

PS::U64 は以下のように定義されている。すなわち 64bit の符号なし整数である。

ソースコード 4: U64

```
1 namespace ParticleSimulator {
2     typedef unsigned long long int U64;
3 }
```

ただし、GCC コンパイラと K コンパイラでのみ 64bit であることが保証されている。

6.3 実数型

6.3.1 概要

実数型には PS::F32, PS::F64 がある。以下、順にこれらを記述する。

6.3.2 PS::F32

PS::F32 は以下のように定義されている。すなわち 32bit の浮動小数点数である。

ソースコード 5: F32

```
1 namespace ParticleSimulator {
2     typedef float F32;
3 }
```

6.3.3 PS::F64

PS::F64 は以下のように定義されている。すなわち 64bit の浮動小数点数である。

ソースコード 6: F64

```
1 namespace ParticleSimulator {
2     typedef double F64;
3 }
```

6.4 ベクトル型

6.4.1 概要

ベクトル型には 2 次元ベクトル型 PS::Vector2 と 3 次元ベクトル型 PS::Vector3 がある。まずこれら 2 つを記述する。最後にこれらベクトル型のラッパーについて記述する。

6.4.2 PS::Vector2

PS::Vector2 は x, y の 2 要素を持つ。これらに対する様々な API や演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 7: Vector2

```
1  namespace ParticleSimulator{
2      template <typename T>
3      class Vector2{
4      public:
5          //メンバ変数2要 素
6          T x, y;
7
8          //コンストラクタ
9          Vector2();
10         Vector2(const T _x, const T _y) : x(_x), y(_y) {}
11         Vector2(const T s) : x(s), y(s) {}
12         Vector2(const Vector2 & src) : x(src.x), y(src.y) {}
13
14         //代入演算子
15         const Vector2 & operator = (const Vector2 & rhs);
16
17         //加減算
18         Vector2 operator + (const Vector2 & rhs) const;
19         const Vector2 & operator += (const Vector2 & rhs);
```

```
20          Vector2 operator - (const Vector2 & rhs) const;
21          const Vector2 & operator -= (const Vector2 & rhs);
22
23          //ベクトルスカラ積
24          Vector2 operator * (const T s) const;
25          const Vector2 & operator *= (const T s);
26          friend Vector2 operator * (const T s,
27                                     const Vector2 & v);
28          Vector2 operator / (const T s) const;
29          const Vector2 & operator /= (const T s);
30
31          //内積
32          T operator * (const Vector2 & rhs) const;
33
34          //外積(返り値はスカラ!!)
35          T operator ^ (const Vector2 & rhs) const;
36
37          //Vector2<U>への型変換
38          template <typename U>
39          operator Vector2<U> () const;
40      };
41  }
42  namespace PS = ParticleSimulator;
```

6.4.2.1 コンストラクタ

```
template<typename T>
PS::Vector2<T>::Vector2()
```

- **引数**

  なし。

- **機能**

  デフォルトコンストラクタ。メンバ x,y は 0 で初期化される。

```
template<typename T>
PS::Vector2<T>::Vector2(const T _x, const T _y)
```

- **引数**

  _x: 入力。const T 型。

  _y: 入力。const T 型。

- **機能**

  メンバ x、y をそれぞれ_x、_y で初期化する。

```
template<typename T>
PS::Vector2<T>::Vector2(const T s);
```

- **引数**

  s: 入力。const T 型。

- **機能**

  メンバ x、y を両方とも s の値で初期化する。

6.4.2.2 コピーコンストラクタ

```
template<typename T>
PS::Vector2<T>::Vector2(const PS::Vector2<T> & src)
```

- **引数**

  src: 入力。const PS::Vector2<T> &型。

- **機能**

  コピーコンストラクタ。src で初期化する。

6.4.2.3 代入演算子

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator =
                       (const PS::Vector2<T> & rhs);
```

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  const PS::Vector2<T> &型。rhs の x,y の値を自身のメンバ x,y に代入し自身の参照を返す。代入演算子。

6.4.2.4 加減算

```
template<typename T>
PS::Vector2<T> PS::Vector2<T>::operator +
               (const PS::Vector2<T> & rhs) const;
```

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  PS::Vector2<T> 型。rhs の x,y の値と自身のメンバ x,y の値の和を取った値を返す。

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator +=
                       (const PS::Vector2<T> & rhs);
```

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  const PS::Vector2<T> &型。rhs の x,y の値を自身のメンバ x,y に足し、自身を返す。

```
template<typename T>
PS::Vector2<T> PS::Vector2<T>::operator -
               (const PS::Vector2<T> & rhs) const;
```

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  PS::Vector2<T> 型。rhs の x,y の値と自身のメンバ x,y の値の差を取った値を返す。

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator -=
                       (const PS::Vector2<T> & rhs);
```

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  const PS::Vector2<T> &型。自身のメンバ x,y から rhs の x,y を引き自身を返す。

6.4.2.5 ベクトルスカラ積

```
template<typename T>
PS::Vector2<T> PS::Vector2<T>::operator * (const T s) const;
```

- **引数**

  s: 入力。const T 型。

- **返り値**

  PS::Vector2<T> 型。自身のメンバ x,y それぞれに s をかけた値を返す。

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator *= (const T s);
```

- **引数**

  rhs: 入力。const T 型。

- **返り値**

  const PS::Vector2<T> &型。自身のメンバ x,y それぞれに s をかけ自身を返す。

```
template<typename T>
PS::Vector2<T> PS::Vector2<T>::operator / (const T s) const;
```

- **引数**

  s: 入力。const T 型。

- **返り値**

  PS::Vector2<T> 型。自身のメンバ x,y それぞれを s で割った値を返す。

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator /= (const T s);
```

- **引数**

  rhs: 入力。const T 型。

- **返り値**

  const PS::Vector2<T> &型。自身のメンバ x,y それぞれを s で割り自身を返す。

6.4.2.6 内積、外積

```
template<typename T>
T PS::Vector2<T>::operator * (const PS::Vector2<T> & rhs) const;
```

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  T 型。自身と rhs の内積を取った値を返す。

```
template<typename T>
T PS::Vector2<T>::operator ^ (const PS::Vector2<T> & rhs) const;
```

- **引数**

  rhs: 入力。const PS::Vector2<T> &型。

- **返り値**

  T 型。自身と rhs の外積を取った値を返す。

6.4.2.7 Vector2<U> への型変換

```
template<typename T>
template <typename U>
PS::Vector2<T>::operator PS::Vector2<U> () const;
```

- **引数**

  なし。

- **返り値**

  const PS::Vector2<U> 型。

- **機能**

  const PS::Vector2<T> 型を const PS::Vector2<U> 型にキャストする。

6.4.3 PS::Vector3

PS::Vecotr3 は x, y, z の 3 要素を持つ。これらに対する様々な API や演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 8: Vector3

```
1  namespace ParticleSimulator{
2      template <typename T>
3      class Vector3{
4      public:
5          //メンバ変数は以下の二つのみ。
6          T x, y, z;
7
8          //コンストラクタ
9          Vector3() : x(T(0)), y(T(0)), z(T(0)) {}
10         Vector3(const T _x, const T _y, const T _z) : x(_x), y(
               _y), z(_z) {}
11         Vector3(const T s) : x(s), y(s), z(s) {}
12         Vector3(const Vector3 & src) : x(src.x), y(src.y), z(
               src.z) {}
13
14         //代入演算子
15         const Vector3 & operator = (const Vector3 & rhs);
16
17         //加減算
18         Vector3 operator + (const Vector3 & rhs) const;
19         const Vector3 & operator += (const Vector3 & rhs);
20         Vector3 operator - (const Vector3 & rhs) const;
21         const Vector3 & operator -= (const Vector3 & rhs);
22
23         //ベクトルスカラ積
24         Vector3 operator * (const T s) const;
25         const Vector3 & operator *= (const T s);
26         friend Vector3 operator * (const T s, const Vector3 & v
               );
27         Vector3 operator / (const T s) const;
28         const Vector3 & operator /= (const T s);
29
30         //内積
31         T operator * (const Vector3 & rhs) const;
32
```

```
33          //外積(返り値はスカラ!!)
34          T operator ^ (const Vector3 & rhs) const;
35
36          //Vector3<U>への型変換
37          template <typename U>
38          operator Vector3<U> () const;
39      };
40  }
```

6.4.3.1 コンストラクタ

```
template<typename T>
PS::Vector3<T>::Vector3()
```

- **引数**

  なし。

- **機能**

  デフォルトコンストラクタ。メンバ x,y は 0 で初期化される。

```
template<typename T>
PS::Vector3<T>::Vector3(const T _x, const T _y)
```

- **引数**

  _x: 入力。const T 型。

  _y: 入力。const T 型。

- **機能**

  メンバ x、y をそれぞれ_x、_y で初期化する。

```
template<typename T>
PS::Vector3<T>::Vector3(const T s);
```

- **引数**

  s: 入力。const T 型。

- **機能**

  メンバ x、y を両方とも s の値で初期化する。

6.4.3.2 コピーコンストラクタ

```
template<typename T>
PS::Vector3<T>::Vector3(const PS::Vector3<T> & src)
```

- **引数**

  src: 入力。const PS::Vector3<T> &型。

- **機能**

  コピーコンストラクタ。src で初期化する。

6.4.3.3 代入演算子

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator =
                       (const PS::Vector3<T> & rhs);
```

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  const PS::Vector3<T> &型。rhs の x,y の値を自身のメンバ x,y に代入し自身の参照を返す。代入演算子。

6.4.3.4 加減算

```
template<typename T>
PS::Vector3<T> PS::Vector3<T>::operator +
               (const PS::Vector3<T> & rhs) const;
```

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  PS::Vector3<T> 型。rhs の x,y の値と自身のメンバ x,y の値の和を取った値を返す。

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator +=
                        (const PS::Vector3<T> & rhs);
```

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  const PS::Vector3<T> &型。rhs の x,y の値を自身のメンバ x,y に足し、自身を返す。

```
template<typename T>
PS::Vector3<T> PS::Vector3<T>::operator -
               (const PS::Vector3<T> & rhs) const;
```

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  PS::Vector3<T> 型。rhs の x,y の値と自身のメンバ x,y の値の差を取った値を返す。

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator -=
                        (const PS::Vector3<T> & rhs);
```

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  const PS::Vector3<T> &型。自身のメンバ x,y から rhs の x,y を引き自身を返す。

6.4.3.5 ベクトルスカラ積

```
template<typename T>
PS::Vector3<T> PS::Vector3<T>::operator * (const T s) const;
```

- **引数**

  s: 入力。const T 型。

- **返り値**

  PS::Vector3<T> 型。自身のメンバ x,y それぞれに s をかけた値を返す。

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator *= (const T s);
```

- **引数**

  rhs: 入力。const T 型。

- **返り値**

  const PS::Vector3<T> &型。自身のメンバ x,y それぞれに s をかけ自身を返す。

```
template<typename T>
PS::Vector3<T> PS::Vector3<T>::operator / (const T s) const;
```

- **引数**

  s: 入力。const T 型。

- **返り値**

  PS::Vector3<T> 型。自身のメンバ x,y それぞれを s で割った値を返す。

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator /= (const T s);
```

- **引数**

  rhs: 入力。const T 型。

- **返り値**

  const PS::Vector3<T> &型。自身のメンバ x,y それぞれを s で割り自身を返す。

6.4.3.6 内積、外積

```
template<typename T>
T PS::Vector3<T>::operator * (const PS::Vector3<T> & rhs) const;
```

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  T 型。自身と rhs の内積を取った値を返す。

```
template<typename T>
T PS::Vector3<T>::operator ^ (const PS::Vector3<T> & rhs) const;
```

- **引数**

  rhs: 入力。const PS::Vector3<T> &型。

- **返り値**

  T 型。自身と rhs の外積を取った値を返す。

6.4.3.7 Vector3<U> への型変換

```
template<typename T>
template <typename U>
PS::Vector3<T>::operator PS::Vector3<U> () const;
```

- **引数**

  なし

- **返り値**

  const PS::Vector3<U> 型。

- **機能**

  const PS::Vector3<T> 型を const PS::Vector3<U> 型にキャストする。

6.4.4 ベクトル型のラッパー

ベクトル型のラッパーの定義を以下に示す。

ソースコード 9: vectorwrapper

```
1  namespace ParticleSimulator{
2      typedef Vector2<F32> F32vec2;
3      typedef Vector3<F32> F32vec3;
4      typedef Vector2<F64> F64vec2;
5      typedef Vector3<F64> F64vec3;
6  #ifdef PARTICLE_SIMULATOR_TOW_DIMENSION
7      typedef F32vec2 F32vec;
8      typedef F64vec2 F64vec;
9  #else
10     typedef F32vec3 F32vec;
11     typedef F64vec3 F64vec;
12 #endif
13 }
```

すなわち PS::F32vec2, PS::F32vec3, PS::F64vec2, PS::F64vec3 はそれぞれ単精度 2 次元ベクトル、倍精度 2 次元ベクトル、単精度 3 次元ベクトル、倍精度 3 次元ベクトルである。FDPS で扱う空間座標系を 2 次元とした場合、PS::F32vec と PS::F64vec はそれぞれ単精度 2 次元ベクトル、倍精度 2 次元ベクトルとなる。一方、FDPS で扱う空間座標系を 3 次元とした場合、PS::F32vec と PS::F64vec はそれぞれ単精度 3 次元ベクトル、倍精度 3 次元ベクトルとなる。

6.5 対称行列型

6.5.1 概要

対称行列型には 2x2 対称行列型 PS::MatrixSym2 と 3x3 対称行列型 PS::MatrixSym3 がある。まずこれら 2 つを記述する。最後にこれら対称行列型のラッパーについて記述する。

6.5.2 PS::MatrixSym2

PS::MatrixSym2 は xx, yy, xy の 3 要素を持つ。これらに対する様々な API や演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 10: MatrixSym2

```
1  namespace ParticleSimulator{
2      template<class T>
3      class MatrixSym2{
```

```
4       public:
5           // メンバ変数 3 要素
6           T xx, yy, xy;
7
8           // コンストラクタ
9           MatrixSym2() : xx(T(0)), yy(T(0)), xy(T(0)) {}
10          MatrixSym2(const T _xx, const T _yy, const T _xy)
11              : xx(_xx), yy(_yy), xy(_xy) {}
12          MatrixSym2(const T s) : xx(s), yy(s), xy(s){}
13          MatrixSym2(const MatrixSym2 & src) : xx(src.xx), yy(src
                .yy), xy(src.xy) {}
14
15          // 代入演算子
16          const MatrixSym2 & operator = (const MatrixSym2 & rhs);
17
18          // 加減算
19          MatrixSym2 operator + (const MatrixSym2 & rhs) const;
20          const MatrixSym2 & operator += (const MatrixSym2 & rhs)
                const;
21          MatrixSym2 operator - (const MatrixSym2 & rhs) const;
22          const MatrixSym2 & operator -= (const MatrixSym2 & rhs)
                const;
23
24          // トレースの計算
25          T getTrace() const;
26
27          // MatrixSym2<U>への型変換
28          template <typename U>
29          operator MatrixSym2<U> () const;
30      }
31  }
32  namespace PS = ParticleSimulator;
```

6.5.2.1 コンストラクタ

```
template<typename T>
PS::MatrixSym2<T>::MatrixSym2();
```

- **引数**

なし。

- **機能**

  デフォルトコンストラクタ。メンバ xx,yy,xy は 0 で初期化される。

```
template<typename T>
PS::MatrixSym2<T>::MatrixSym2
            (const T _xx,
             const T _yy,
             const T _xy);
```

- **引数**

  _xx: 入力。const T 型。

  _yy: 入力。const T 型。

  _xy: 入力。const T 型。

- **機能**

  メンバ xx、yy、xy をそれぞれ_xx、_yy、_xy で初期化する。

```
template<typename T>
PS::MatrixSym2<T>::MatrixSym2(const T s);
```

- **引数**

  s: 入力。const T 型。

- **機能**

  メンバ xx、yy、xy すべてを s の値で初期化する。

6.5.2.2 コピーコンストラクタ

```
template<typename T>
PS::MatrixSym2<T>::MatrixSym2(const PS::MatrixSym2<T> & src)
```

- **引数**

  src: 入力。const PS::MatrixSym2<T> &型。

- **機能**

  コピーコンストラクタ。src で初期化する。

6.5.2.3 代入演算子

```
template<typename T>
const PS::MatrixSym2<T> & PS::MatrixSym2<T>::operator =
                        (const PS::MatrixSym2<T> & rhs);
```

- **引数**

  rhs: 入力。const PS::MatrixSym2<T> &型。

- **返り値**

  const PS::MatrixSym2<T> &型。rhs の xx,yy,xy の値を自身のメンバ xx,yy,xy に代入し自身の参照を返す。代入演算子。

6.5.2.4 加減算

```
template<typename T>
PS::MatrixSym2<T> PS::MatrixSym2<T>::operator +
                (const PS::MatrixSym2<T> & rhs) const;
```

- **引数**

  rhs: 入力。const PS::MatrixSym2<T> &型。

- **返り値**

  PS::MatrixSym2<T> 型。rhs の xx,yy,xy の値と自身のメンバ xx,yy,xy の値の和を取った値を返す。

```
template<typename T>
const PS::MatrixSym2<T> & PS::MatrixSym2<T>::operator +=
                        (const PS::MatrixSym2<T> & rhs);
```

- **引数**

  rhs: 入力。const PS::MatrixSym2<T> &型。

- **返り値**

  const PS::MatrixSym2<T> &型。rhs の xx,yy,xy の値を自身のメンバ xx,yy,xy に足し、自身を返す。

```
template<typename T>
PS::MatrixSym2<T> PS::MatrixSym2<T>::operator -
               (const PS::MatrixSym2<T> & rhs) const;
```

- **引数**

  rhs: 入力。const PS::MatrixSym2<T> &型。

- **返り値**

  PS::MatrixSym2<T> 型。rhs の xx,yy,xy の値と自身のメンバ xx,yy,xy の値の差を取った値を返す。

```
template<typename T>
const PS::MatrixSym2<T> & PS::MatrixSym2<T>::operator -=
                        (const PS::MatrixSym2<T> & rhs);
```

- **引数**

  rhs: 入力。const PS::MatrixSym2<T> &型。

- **返り値**

  const PS::MatrixSym2<T> &型。自身のメンバ xx,yy,xy から rhs の xx,yy,xy を引き自身を返す。

6.5.2.5 トレースの計算

```
template<typename T>
T PS::MatrixSym2<T>::getTrace() const;
```

- **引数**

  なし

- **返り値**

  T 型。

- **機能**

  トレースを計算し、その結果を返す。

6.5.2.6 MatrixSym2<U> への型変換

```
template<typename T>
template<typename U>
PS::MatrixSym2<T>::operator PS::MatrixSym2<U> () const;
```

- **引数**

  なし。

- **返り値**

  const PS::MatrixSym2<U> 型。

- **機能**

  const PS::MatrixSym2<T> 型を const PS::MatrixSym2<U> 型にキャストする

6.5.3 PS::MatrixSym3

PS::MatrixSym3 は xx, yy, zz, xy, xz, yz の 6 要素を持つ。これらに対する様々な API や演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 11: MatrixSym3

```
1  namespace ParticleSimulator{
2      template<class T>
3      class MatrixSym3{
4      public:
5          // メンバ変数 6 要素
6          T xx, yy, zz, xy, xz, yz;
7
8          // コンストラクタ
9          MatrixSym3() : xx(T(0)), yy(T(0)), zz(T(0)),
10                        xy(T(0)), xz(T(0)), yz(T(0)) {}
11         MatrixSym3(const T _xx, const T _yy, const T _zz,
12                    const T _xy, const T _xz, const T _yz )
13                        : xx(_xx), yy(_yy), zz(_zz),
14                        xy(_xy), xz(_xz), yz(_yz) {}
15         MatrixSym3(const T s) : xx(s), yy(s), zz(s),
16                                 xy(s), xz(s), yz(s) {}
17         MatrixSym3(const MatrixSym3 & src) :
18             xx(src.xx), yy(src.yy), zz(src.zz),
```

```
19              xy(src.xy), xz(src.xz), yz(src.yz) {}
20
21          // 代入演算子
22          const MatrixSym3 & operator = (const MatrixSym3 & rhs);
23
24          // 加減算
25          MatrixSym3 operator + (const MatrixSym3 & rhs) const;
26          const MatrixSym3 & operator += (const MatrixSym3 & rhs)
                const;
27          MatrixSym3 operator - (const MatrixSym3 & rhs) const;
28          const MatrixSym3 & operator -= (const MatrixSym3 & rhs)
                const;
29
30          // トレースを取る
31          T getTrace() const;
32
33          // MatrixSym3<U>への型変換
34          template <typename U>
35          operator MatrixSym3<U> () const;
36      }
37  }
38  namespace PS = ParticleSimulator;
```

6.5.3.1 コンストラクタ

```
template<typename T>
PS::MatrixSym3<T>::MatrixSym3();
```

- **引数**

  なし。

- **機能**

  デフォルトコンストラクタ。6要素は0で初期化される。

```
template<typename T>
PS::MatrixSym3<T>::MatrixSym3(const T _xx,
                              const T _yy,
                              const T _zz,
                              const T _xy,
                              const T _xz,
                              const T _yz);
```

- **引数**

  _xx: 入力。const T 型。

  _yy: 入力。const T 型。

  _zz: 入力。const T 型。

  _xy: 入力。const T 型。

  _xz: 入力。const T 型。

  _yz: 入力。const T 型。

- **機能**

  メンバ xx、yy、zz、xy、xz、yz をそれぞれ_xx、_yy、_zz、_xy、_xz、_yz で初期化する。

```
template<typename T>
PS::MatrixSym3<T>::MatrixSym3(const T s);
```

- **引数**

  s: 入力。const T 型。

- **機能**

  6 要素すべてを s の値で初期化する。

6.5.3.2 コピーコンストラクタ

```
template<typename T>
PS::MatrixSym3<T>::MatrixSym3(const PS::MatrixSym3<T> & src)
```

- **引数**

  src: 入力。const PS::MatrixSym3<T> &型。

- **機能**

  コピーコンストラクタ。src で初期化する。

6.5.3.3 代入演算子

```
template<typename T>
const PS::MatrixSym3<T> & PS::MatrixSym3<T>::operator =
                        (const PS::MatrixSym3<T> & rhs);
```

- **引数**

  rhs: 入力。const PS::MatrixSym3<T> &型。

- **返り値**

  const PS::MatrixSym3<T> &型。rhs の 6 要素それぞれの値を自身の 6 要素それぞれに代入し自身の参照を返す。代入演算子。

6.5.3.4 加減算

```
template<typename T>
PS::MatrixSym3<T> PS::MatrixSym3<T>::operator +
               (const PS::MatrixSym3<T> & rhs) const;
```

- **引数**

  rhs: 入力。const PS::MatrixSym3<T> &型。

- **返り値**

  PS::MatrixSym3<T> 型。rhs の 6 要素それぞれの値と自身の 6 要素の値の和を取った値を返す。

```
template<typename T>
const PS::MatrixSym3<T> & PS::MatrixSym3<T>::operator +=
                        (const PS::MatrixSym3<T> & rhs);
```

- **引数**

  rhs: 入力。const PS::MatrixSym3<T> &型。

- **返り値**

  const PS::MatrixSym3<T> &型。rhs の 6 要素それぞれの値を自身の 6 要素それぞれに足し、自身を返す。

```
template<typename T>
PS::MatrixSym3<T> PS::MatrixSym3<T>::operator -
               (const PS::MatrixSym3<T> & rhs) const;
```

- **引数**

  rhs: 入力。const PS::MatrixSym3<T> &型。

- **返り値**

  PS::MatrixSym3<T> 型。rhs の 6 要素それぞれの値と自身の 6 要素それぞれの値の差を取った値を返す。

```
template<typename T>
const PS::MatrixSym3<T> & PS::MatrixSym3<T>::operator -=
                        (const PS::MatrixSym3<T> & rhs);
```

- **引数**

  rhs: 入力。const PS::MatrixSym3<T> &型。

- **返り値**

  const PS::MatrixSym3<T> &型。自身の 6 要素それぞれから rhs の 6 要素それぞれを引き自身を返す。

6.5.3.5 トレースの計算

```
template<typename T>
T PS::MatrixSym3<T>::getTrace() const;
```

- **引数**

  なし

- **返り値**

  T 型。

- **機能**

  トレースを計算し、その結果を返す。

6.5.3.6 MatrixSym3<U> への型変換

```
template<typename T>
template<typename U>
PS::MatrixSym3<T>::operator PS::MatrixSym3<U> () const;
```

- **引数**

  なし。

- **返り値**

  const PS::MatrixSym3<U> 型。

- **機能**

  const PS::MatrixSym3<T> 型を const PS::MatrixSym3<U> 型にキャストする

6.5.4 対称行列型のラッパー

対称行列型のラッパーの定義を以下に示す。

ソースコード 12: matrixsymwrapper

```
1  namespace ParticleSimulator{
2      typedef MatrixSym2<F32> F32mat2;
3      typedef MatrixSym3<F32> F32mat3;
4      typedef MatrixSym2<F64> F64mat2;
5      typedef MatrixSym3<F64> F64mat3;
6  #ifdef PARTICLE_SIMULATOR_TOW_DIMENSION
7      typedef F32mat2 F32mat;
8      typedef F64mat2 F64mat;
9  #else
10     typedef F32mat3 F32mat;
11     typedef F64mat3 F64mat;
12 #endif
13 }
14 namespace PS = ParticleSimulator;
```

すなわち PS::F32mat2, PS::F32mat3, PS::F64mat2, PS::F64mat3 はそれぞれ単精度 2x2 対称行列、倍精度 2x2 対称行列、単精度 3x3 対称行列、倍精度 3x3 対称行列である。FDPS で扱う空間座標系を 2 次元とした場合、PS::F32mat と PS::F64mat はそれぞれ単精度 2x2 対称行列、倍精度 2x2 対称行列となる。一方、FDPS で扱う空間座標系を 3 次元とした場合、PS::F32mat と PS::F64mat はそれぞれ単精度 3x3 対称行列、倍精度 3x3 対称行列となる。

6.6 PS::SEARCH_MODE 型

6.6.1 概要

本節では、PS::SEARCH_MODE 型について記述する。PS::SEARCH_MODE 型は相互作用ツリークラスのテンプレート引数としてのみ使用されるものである。この型によって、相互作用ツリークラスで計算する相互作用のモードを決定する。PS::SEARCH_MODE 型には PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF, PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY がある。以下に、それぞれが対応する相互作用のモードについて記述する。

6.6.2 PS::SEARCH_MODE_LONG

この型を使用するのは、遠くの粒子からの寄与を複数の粒子にまとめた超粒子からの寄与として計算する場合である。開放境界条件における重力やクーロン力に適用できる。

6.6.3 PS::SEARCH_MODE_LONG_CUTOFF

この型を使用するのは、遠くの粒子からの寄与を複数の粒子にまとめた超粒子からの寄与として計算し、かつ有限の距離までの寄与しか計算しない場合である。周期境界条件における重力やクーロン力 (Particle Mesh 法の並用が必要) などに適用できる。

6.6.4 PS::SEARCH_MODE_GATHER

この型を使用するのは、相互作用の到達距離が有限でかつ、その到達距離が i 粒子の大きさで決まる場合である。

6.6.5 PS::SEARCH_MODE_SCATTER

この型を使用するのは、相互作用の到達距離が有限でかつ、その到達距離が j 粒子の大きさで決まる場合である。

6.6.6 PS::SEARCH_MODE_SYMMETRY

この型を使用するのは、相互作用の到達距離が有限でかつ、その到達距離が i, j 粒子両方の大きさで決まる場合である。

6.7 列挙型

6.7.1 概要

本節ではFDPSで定義されている列挙型について記述する。列挙型にはBOUNDARY_CONDITION型が存在する。以下、各列挙型について記述する。

6.7.2 PS::BOUNDARY_CONDITION 型

6.7.2.1 概要

BOUNDARY_CONDITION 型は境界条件を指定するためのデータ型である。これは以下のように定義されている。

ソースコード 13: boundarycondition

```
1  namespace ParticleSimulator{
2      enum BOUNDARY_CONDITION{
3          BOUNDARY_CONDITION_OPEN,
4          BOUNDARY_CONDITION_PERIODIC_X,
5          BOUNDARY_CONDITION_PERIODIC_Y,
6          BOUNDARY_CONDITION_PERIODIC_Z,
7          BOUNDARY_CONDITION_PERIODIC_XY,
8          BOUNDARY_CONDITION_PERIODIC_XZ,
9          BOUNDARY_CONDITION_PERIODIC_YZ,
10         BOUNDARY_CONDITION_PERIODIC_XYZ,
11         BOUNDARY_CONDITION_SHEARING_BOX,
12         BOUNDARY_CONDITION_USER_DEFINED,
13     };
14 }
```

以下にどの変数がどの境界条件に対応するかを記述する。

6.7.2.2 PS::BOUNDARY_CONDITION_OPEN

開放境界となる。

6.7.2.3 PS::BOUNDARY_CONDITION_PERIODIC_X

x 軸方向のみ周期境界、その他の軸方向は開放境界となる。周期の境界の下限は閉境界、上限は開境界となっている。この境界の規定はすべての軸方向にあてはまる。

6.7.2.4 PS::BOUNDARY_CONDITION_PERIODIC_Y

y 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.7.2.5 PS::BOUNDARY_CONDITION_PERIODIC_Z

z 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.7.2.6 PS::BOUNDARY_CONDITION_PERIODIC_XY

x, y 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.7.2.7 PS::BOUNDARY_CONDITION_PERIODIC_XZ

x, z 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.7.2.8 PS::BOUNDARY_CONDITION_PERIODIC_YZ

y, z 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.7.2.9 PS::BOUNDARY_CONDITION_PERIODIC_XYZ

x, y, z 軸方向すべてが周期境界となる。

6.7.2.10 PS::BOUNDARY_CONDITION_SHEARING_BOX

未実装。

6.7.2.11 PS::BOUNDARY_CONDITION_USER_DEFINED

未実装。

7　ユーザー定義クラス・ユーザー定義関数オブジェクト

7.1　概要

本節では、ユーザーが定義するクラスとファンクタについて記述する。ユーザー定義クラスとなるのは、粒子の情報すべてを持つ FullParticle クラス、ある相互作用を計算する際に i 粒子に必要な情報を持つ EssentialParticleI クラス、ある相互作用を計算する際に j 粒子に必要な情報を持つ EssentialParticleJ クラス、ツリーセルのモーメント情報を持つ Moment クラス、ある相互作用 (PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG または PS::SEARCH_MODE_LONG_CUTOFF の場合に限る) を計算する際に超粒子に必要な情報を持つ SuperParticleJ クラス、相互作用の結果の情報を持つ Force クラス、入出力ファイルのヘッダ情報を持つヘッダクラスである。また、ユーザー定義の関数オブジェクトには、j 粒子から i 粒子への作用を計算する関数オブジェクト calcForceEpEp、超粒子から i 粒子への作用を計算する関数オブジェクト calcForceSpEp がある。

この節で記述するのは、これらのクラスや関数オブジェクトを定義する際の規定である。ユーザーはこれらの間でのデータのやりとりや、関数オブジェクト内でのデータの加工についてコードに書く必要がある。これらは上に挙げたクラスのメンバ関数と関数オブジェクト内で行われる。以下、必要なメンバ関数とその規定について記述する。

7.2　FullParticle クラス

7.2.1　概要

FullParticle クラスは粒子情報すべてを持つクラスであり、節 2.3 の手順 0 で、粒子群クラスに渡されるユーザー定義クラスの 1 つである。ユーザーはこのクラスに対して、どのようなメンバ変数、メンバ関数を定義してもかまわない。ただし、FDPS から FullParticle クラスの情報にアクセスする ために、ユーザーはいくつかの決まった名前のメンバ関数を定義する必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

7.2.2　前提

この節の中では、以下のように、FullParticle クラスとして FP というクラスを一例とする。FP という名前は自由に変えることができる。

```
class FP;
```

7.2.3 必要なメンバ関数

7.2.3.1 概要

常に必要なメンバ関数は FP::getPos と FP::copyFromForce である。FP::getPos は FullParticle の位置情報を FDPS に読み込ませるための関数で、FP::copyFromForce は計算された相互作用の結果を FullParticle に書き戻す関数である。これらのメンバ関数の記述例と解説を以下に示す。

7.2.3.2 FP::getPos

```
class FP {
public:
    PS::F64vec pos;
    PS::F64vec getPos() const {
        return this->pos;
    }
};
```

- **前提**

  FP クラスのメンバ変数 pos はある 1 つの粒子の位置情報。この pos のデータ型は PS::F32vec 型または PS::F64vec 型。

- **引数**

  なし

- **返値**

  PS::F32vec 型または PS::F64vec 型。FP クラスのオブジェクトの位置情報を保持したメンバ変数。

- **機能**

  FP クラスのオブジェクトの位置情報を保持したメンバ変数を返す。

- **備考**

  FP クラスのメンバ変数 pos の変数名は変更可能。ただしこの pos のデータ型とメンバ関数 FP::getPos の返値のデータ型が一致していない場合の動作は保証しない。

7.2.3.3 FP::copyFromForce

```
class Force {
public:
    PS::F64vec acc;
    PS::F64    pot;
};
class FP {
public:
    PS::F64vec acceleration;
    PS::F64    potential;
    void copyFromForce(const Force & force) {
        this->acceleration = force.acc;
        this->potential    = force.pot;
    }
};
```

- **前提**

  Force クラスは粒子の相互作用の計算結果を保持するクラス。

- **引数**

  force: 入力。const Force &型。粒子の相互作用の計算結果を保持。

- **返値**

  なし。

- **機能**

  粒子の相互作用の計算結果を FP クラスへ書き戻す。Force クラスのメンバ変数 acc, pot がそれぞれ FP クラスのメンバ変数 acceleration, potential に対応。

- **備考**

  Force クラスというクラス名とそのメンバ変数名は変更可能。FP のメンバ変数名は変更可能。メンバ関数 FP::copyFromForce の引数名は変更可能。

7.2.4 場合によっては必要なメンバ関数

7.2.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合、粒子群クラスのファイル入出力 API を用いる場合、粒子群クラスの API である ParticleSystem::adjustPositionIntoRootDom

を用いる場合、拡張機能の Particle Mesh クラスを用いる場合について必要となるメンバ関数を記述する。

7.2.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合

7.2.4.2.1 FP::getRsearch

```
class FP {
public:
    PS::F64 search_radius;
    PS::F64 getRsearch() const {
        return this->search_radius;
    }
};
```

- **前提**

  FP クラスのメンバ変数 search_radius はある 1 つの粒子の近傍粒子を探す半径の大きさ。この search_radius のデータ型は PS::F32 型または PS::F64 型。

- **引数**

  なし

- **返値**

  PS::F32 型または PS::F64 型。 FP クラスのオブジェクトの近傍粒子を探す半径の大きさを保持したメンバ変数。

- **機能**

  FP クラスのオブジェクトの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

- **備考**

  FP クラスのメンバ変数 search_radius の変数名は変更可能。

7.2.4.3 粒子群クラスのファイル入出力 API を用いる場合

粒子群クラスのファイル入出力 API である ParticleSystem::readParticleAscii, ParticleSystem::writeParticleAscii を使用するときにそれぞれ readAscii, writeAscii というメンバ関数が必要となる。以下、readAscii と writeAscii の規定について記述する。

7.2.4.3.1　FP::readAscii

```
class FP {
public:
    PS::S32 id;
    PS::F64 mass;
    PS::F64vec pos;
    void readAscii(FILE *fp) {
        fscanf(fp, "%d%lf%lf%lf%lf", &this->id, &this->mass,
               &this->pos[0], &this->pos[1], &this->pos[2]);
    }
};
```

- **前提**

  粒子データの入力ファイルの１列目にはFPクラスのメンバ変数idを表すデータが、２列目にはメンバ変数massを表すデータが、３、４、５列めにはメンバ変数posの第１、２、３要素が、それ以降には列がないとする。ファイルの形式はアスキー形式とする。３次元直交座標系を選択したとする。

- **引数**

  fp: FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  粒子データの入力ファイルからFPクラスのid、mass、posの情報を読み取る。

- **備考**

  なし。

7.2.4.3.2　FP::writeAscii

```
class FP {
public:
    PS::S32 id;
    PS::F64 mass;
    PS::F64vec pos;
    void writeAscii(FILE *fp) {
        fscanf(fp, "%d %lf %lf %lf %lf", this->id, this->mass,
               this->pos[0], this->pos[1], this->pos[2]);
    }
};
```

- **前提**

  粒子データの出力ファイルの１列目にはFPクラスのメンバ変数idを表すデータが、２列目にはメンバ変数massを表すデータが、３、４、５列めにはメンバ変数posの第１、２、３要素が、それ以降には列がないとする。ファイルの形式はアスキー形式とする。３次元直交座標系を選択したとする。

- **引数**

  fp: FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  粒子データの出力ファイルへFPクラスのメンバ変数id、mass、posの情報を書き出す。

- **備考**

  なし。

7.2.4.4 ParticleSystem::adjustPositionIntoRootDomain を用いる場合

7.2.4.4.1 FP::setPos

```
class FP {
public:
    PS::F64vec pos;
    void setPos(const PS::F64vec pos_new) {
        this->pos = pos_new;
    }
};
```

- **前提**

  FP クラスのメンバ変数 pos は 1 つの粒子の位置情報。この pos のデータ型は PS::F32vec または PS::F64vec。

- **引数**

  pos_new: 入力。const PS::F32vec または const PS::F64vec 型。FDPS 側で修正した粒子の位置情報。

- **返値**

  なし。

- **機能**

  FDPS が修正した粒子の位置情報を FP クラスのオブジェクトの位置情報に書き込む。

- **備考**

  FP クラスのメンバ変数 pos の変数名は変更可能。メンバ関数 FP::setPos の引数名 pos_new は変更可能。pos と pos_new のデータ型が異なる場合の動作は保証しない。

7.2.4.5 Particle Mesh クラスを用いる場合

Particle Mesh クラスを用いる場合には、メンバ関数 FP::getChargeParticleMesh と FP::copyFromForcePa
を用意する必要がある。以下にそれぞれの規定を記述する。

7.2.4.5.1　FP::getChargeParticleMesh

```
class FP {
public:
    PS::F64 mass;
    PS::F64 getChargeParticleMesh() const {
        return this->mass;
    }
};
```

- **前提**

  FP クラスのメンバ変数 mass は１つの粒子の質量または電荷の情報を持つ変数。データ型は PS::F32 または PS::F64 型。

- **引数**

  なし。

- **返値**

  PS::F32 型または PS::F64 型。１つの粒子の質量または電荷の変数を返す。

- **機能**

  １つの粒子の質量または電荷を表すメンバ変数を返す。

- **備考**

  FP クラスのメンバ変数 mass の変数名は変更可能。

7.2.4.5.2　FP::copyFromForceParticleMesh

```
class FP {
public:
    PS::F64vec accelerationFromPM;
    void copyFromForceParticleMesh(const PS::F32vec & acc_pm) {
        this->accelerationFromPM = acc_pm;
    }
};
```

- **前提**

  FP クラスのメンバ変数 accelerationFromPM_pm は１つの粒子の Particle Mesh による力の情報を保持する変数。この accelerationFromPM_pm のデータ型は PS::F32vec または PS::F64vec。

- **引数**

  acc_pm: const PS::F32vec 型または const PS::F64vec 型。１つの粒子の Particle Mesh による力の計算結果。

- **返値**

  なし。

- **機能**

  １つの粒子の Particle Mesh による力の計算結果をこの粒子のメンバ変数に書き込む。

- **備考**

  FP クラスのメンバ変数 acc_pm の変数名は変更可能。メンバ関数 FP::copyFromForceParticleMesh の引数 acc_pm の引数名は変更可能。

7.3 EssentialParticleI クラス

7.3.1 概要

EssentialParticleI クラスは相互作用の計算に必要な i 粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。EssentialParticleI クラスは FullParticle クラス (節 7.2) のサブセットである。FDPS は、このクラスのデータにアクセスする必要がある。そのため、EssentialParticleI クラスはいくつかのメンバ関数を持つ必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

7.3.2 前提

この節の中では、EssentialParticleI クラスとして EPI というクラスを一例として使う。また、FullParticle クラスの一例として FP というクラスを使う。EPI, FP というクラス名は変更可能である。

EPI と FP の宣言は以下の通りである。

```
class FP;
class EPI;
```

7.3.3 必要なメンバ関数

7.3.3.1 概要

常に必要なメンバ関数は EPI::getPos と EPI::copyfromFP である。EPI::getPos は EPI クラスの位置情報を FDPS に読み込ませるための関数で、EPI::copyFromFP は FP クラスの情報を EPI クラスに書きこむ関数である。これらのメンバ関数の記述例と解説を以下に示す。

7.3.3.2 EPI::getPos

```
class EPI {
public:
    PS::F64vec pos;
    PS::F64vec getPos() const {
        return this->pos;
    }
};
```

- **前提**

  EPI のメンバ変数 pos はある 1 つの粒子の位置情報。この pos のデータ型は PS::F64vec 型。

- **引数**

  なし

- **返値**

  PS::F64vec 型。EPI クラスの位置情報を保持したメンバ変数。

- **機能**

  EPI クラスのオブジェクトの位置情報を保持したメンバ変数を返す。

- **備考**

  EPI クラスのメンバ変数 pos の変数名は変更可能。

7.3.3.3 EPI::copyFromFP

```
class FP {
public:
    PS::S64    identity;
    PS::F64    mass;
    PS::F64vec position;
    PS::F64vec velocity;
    PS::F64vec acceleration;
    PS::F64    potential;
};
class EPI {
public:
    PS::S64    id;
    PS::F64vec pos;
    void copyFromFP(const FP & fp) {
        this->id  = fp.identity;
        this->pos = fp.position;
    }
};
```

- **前提**

  FP クラスのメンバ変数 identity, position と EPI クラスのメンバ変数 id, pos はそれぞれ対応する情報を持つ。

- **引数**

  fp: 入力。const FP &型。FP クラスの情報を持つ。

- **返値**

  なし。

- **機能**

  FP クラスの持つ 1 粒子の情報の一部を EssnetialParticleI クラスに書き込む。

- **備考**

  FP クラスのメンバ変数の変数名、EPI クラスのメンバ変数の変数名は変更可能。メンバ関数 EPI::copyFromFP の引数名は変更可能。EPI クラスの粒子情報は FP クラスの粒子情報のサブセット。対応する情報を持つメンバ変数同士のデータ型が一致している必要はないが、実数型とベクトル型 (または整数型とベクトル型) という違いがある場合に正しく動作する保証はない。

7.3.4 場合によっては必要なメンバ関数

7.3.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_GATHER または PS::SEARCH_MODE_SYMMETRY を用いる場合に必要となるメンバ関数ついて記述する。

7.3.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_GATHER または PS::SEARCH_MODE_SYMMETRY を用いる場合

7.3.4.2.1 EPI::getRsearch

```
class EPI {
public:
    PS::F64 search_radius;
    PS::F64 getRsearch() const {
        return this->search_radius;
    }
};
```

- **前提**

  EPI クラスのメンバ変数 search_radius はある 1 つの粒子の近傍粒子を探す半径の大きさ。この search_radius のデータ型は PS::F32 型または PS::F64 型。

- **引数**

  なし

- **返値**

  PS::F32 型または PS::F64 型。 EPI クラスの近傍粒子を探す半径の大きさを保持したメンバ変数。

- **機能**

  EPI クラスの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

- **備考**

  EPI クラスのメンバ変数 search_radius の変数名は変更可能。

7.4 EssentialParticleJ クラス

7.4.1 概要

EssentialParticleJ クラスは相互作用の計算に必要な j 粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。EssentialParticleJ クラスは FullParticle クラス (節 7.2) のサブセットである。FDPS は、このクラスのデータにアクセスする必要がある。このために、EssentialParticleJ クラスはいくつかのメンバ関数を持つ必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

7.4.2 前提

この節の中では、EssentialParticleJ クラスとして EPJ というクラスを一例として使う。また、FullParticle クラスの一例として FP というクラスを使う。EPJ, FP というクラス名は変更可能である。

EPJ と FP の宣言は以下の通りである。

```
class FP;
class EPJ;
```

7.4.3 必要なメンバ関数

7.4.3.1 概要

常に必要なメンバ関数は EPJ::getPos と EPJ::copyfromFP である。EPJ::getPos は EPJ クラスの位置情報を FDPS に読み込ませるための関数で、EPJ::copyFromFP は FP クラスの情報を EPJ クラスに書きこむ関数である。これらのメンバ関数の記述例と解説を以下に示す。

7.4.3.2 EPJ::getPos

```
class EPJ {
public:
    PS::F64vec pos;
    PS::F64vec getPos() const {
        return this->pos;
    }
};
```

- **前提**

  EPJ のメンバ変数 pos はある 1 つの粒子の位置情報。この pos のデータ型は PS::F64vec 型。

- **引数**

  なし

- **返値**

  PS::F64vec 型。EPJ クラスの位置情報を保持したメンバ変数。

- **機能**

  EPJ クラスの位置情報を保持したメンバ変数を返す。

- **備考**

  EPJ クラスのメンバ変数 pos の変数名は変更可能。

7.4.3.3 EPJ::copyFromFP

```
class FP {
public:
    PS::S64    identity;
    PS::F64    mass;
    PS::F64vec position;
    PS::F64vec velocity;
    PS::F64vec acceleration;
    PS::F64    potential;
};
class EPJ {
public:
    PS::S64    id;
    PS::F64    m;
    PS::F64vec pos;
    void copyFromFP(const FP & fp) {
        this->id  = fp.identity;
        this->m   = fp.mass;
        this->pos = fp.position;
    }
};
```

- **前提**

  FP クラスのメンバ変数 identity, mass, position と EPJ クラスのメンバ変数 id, m, pos はそれぞれ対応する情報を持つ。

- **引数**

  fp: 入力。const FP &型。FP クラスの情報を持つ。

- **返値**

  なし。

- **機能**

  FP クラスの持つ 1 粒子の情報の一部を EPJ クラスに書き込む。

- **備考**

  FP クラスのメンバ変数の変数名、EPJ クラスのメンバ変数の変数名は変更可能。メンバ関数 EPJ::copyFromFP の引数名は変更可能。対応する情報を持つメンバ変数同士のデータ型が一致している必要はないが、実数型とベクトル型 (または整数型とベクトル型) という違いがある場合に正しく動作する保証はない。

7.4.4 場合によっては必要なメンバ関数

7.4.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合に必要なメンバ関数、列挙型の BOUNDARY_CONDITION 型に PS::BOUNDARY_CONDITION_OPEN 以外を選んだ場合に必要となるメンバ関数について記述する。

7.4.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合

7.4.4.2.1 EPJ::getRsearch

```
class EPJ {
public:
    PS::F64 search_radius;
    PS::F64 getRsearch() const {
        return this->search_radius;
    }
};
```

- **前提**

  EPJ クラスのメンバ変数 search_radius はある 1 つの粒子の近傍粒子を探す半径の大きさ。この search_radius のデータ型は PS::F32 型または PS::F64 型。

- **引数**

  なし

- **返値**

  PS::F32 型または PS::F64 型。 EPJ クラスの近傍粒子を探す半径の大きさを保持したメンバ変数。

- **機能**

  EPJ クラスの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

- **備考**

  EPJ クラスのメンバ変数 search_radius の変数名は変更可能。

7.4.4.3 BOUNDARY_CONDITION 型に PS::BOUNDARY_CONDITION_OPEN 以外を用いる場合

7.4.4.3.1 EPJ::setPos

```
class EPJ {
public:
    PS::F64vec pos;
    void setPos(const PS::F64vec pos_new) {
        this->pos = pos_new;
    }
};
```

- **前提**

  EPJ クラスのメンバ変数 pos は 1 つの粒子の位置情報。この pos のデータ型は PS::F32vec または PS::F64vec。EPJ クラスのメンバ変数 pos の元データとなっているのは FP クラスのメンバ変数 position。このデータ型は PS::F32vec または PS::F64vec。

- **引数**

  pos_new: 入力。const PS::F32vec または const PS::F64vec 型。FDPS 側で修正した粒子の位置情報。

- **返値**

  なし。

- **機能**

  FDPS が修正した粒子の位置情報を EPJ クラスの位置情報に書き込む。

- **備考**

  EPJ クラスのメンバ変数 pos の変数名は変更可能。メンバ関数 EPJ::setPos の引数名 pos_new は変更可能。pos と pos_new のデータ型が異なる場合の動作は保証しない。

7.5 Moment クラス

7.5.1 概要

Moment クラスは近い粒子同士でまとまった複数の粒子のモーメント情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。モーメント情報の例としては、複数粒子の単極子や双極子、さらにこれら粒子の持つ最大の大きさなど様々なものが考えられる。このクラスは、EssentialParticleJ クラスから SuperParticleJ クラスを作るための中間変数のような役割を果す。従って、このクラスが持つメンバ関数は、EssentialParticleJ クラスから情報を読み出してモーメントを計算するメンバ関数、少ない数の粒子のモーメントからそれらの粒子を含むより多くの粒子のモーメントを計算するメンバ関数などがある。

このようなモーメント情報にはある程度決っているものが多いので、それらについては FDPS 側で用意した。これら既存のクラスについてまず記述する。その後にユーザーがモーメントクラスを自作する際に必ず必要なメンバ関数、場合によっては必要になるメンバ関数について記述する。

7.5.2 既存のクラス

7.5.2.1 概要

FDPS はいくつかの Moment クラスを用意している。これらは相互作用ツリークラスで特定の PS::SEARCH_MODE 型を選んだ場合に有効である。以下、各 PS::SEARCH_MODE 型において選ぶことのできる Moment 型を記述する。PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCA PS::SEARCH_MODE_SYMMETRY については Moment クラスを意識してコーディングする必要がないので、これらについては記述しない。

7.5.2.2 PS::SEARCH_MODE_LONG

7.5.2.2.1 PS::MomentMonopole

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentMonopole {
    public:
        F32    mass;
        F32vec pos;
    };
}
```

- クラス名 PS::MomentMonopole
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心
- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.2.2 PS::MomentQuadrupole

単極子と四重極子を情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentQuadrupole {
    public:
        F32    mass;
        F32vec pos;
        F32mat quad;
    };
}
```

- クラス名 PS::MomentQuadrupole
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量

  pos: 近傍でまとめた粒子の重心

  quad: 近傍でまとめた粒子の四重極子

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.2.3 PS::MomentMonopoleGeometricCenter

単極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentMonopoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
    };
}
```

- クラス名 PS::MomentMonopoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.2.4 PS::MomentDipoleGeometricCenter

双極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentDipoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
    };
}
```

- クラス名 PS::MomentDipoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心

  dipole: 粒子の質量または電荷の双極子
- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.2.5 PS::MomentQuadrupoleGeometricCenter

四重極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentQuadrupoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
        F32mat quadrupole;
    };
}
```

- クラス名 PS::MomentQuadrupoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

dipole: 粒子の質量または電荷の双極子

quadrupole: 粒子の質量または電荷の四重極子

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.3 PS::SEARCH_MODE_LONG_CUTOFF

7.5.2.3.1 PS::MomentMonopoleCutoff

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentMonopoleCutoff {
    public:
        F32    mass;
        F32vec pos;
    };
}
```

- クラス名 PS::MomentMonopoleCutoff
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

- 使用条件

  EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge, EssentialParticleJ::getPos, EssentialParticleJ::getRSearch を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置、粒子の力の到達距離を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.3 必要なメンバ関数

7.5.3.1 概要

以下では Moment クラスを定義する際に、必要なメンバ関数を記述する。このとき Moment クラスのクラス名を Mom とする。これは変更自由である。

7.5.3.2 コンストラクタ

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    Mom () {
        mass = 0.0;
        pos  = 0.0;
    }
};
```

- **前提**

  Mom クラスのメンバ変数 mass, pos は Mom の質量と位置。

- **引数**

  なし

- **返値**

  なし

- **機能**

  Mom クラスのオブジェクトの初期化をする。

- **備考**

  メンバ変数名の変更可能。メンバ変数を加えることも可能。

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    Mom(const PS::F32 m,
        const PS::F32vec & p) {
        mass = m;
        pos  = p;
    }
};
```

- **前提**

  Mom クラスのメンバ変数 mass, pos は Mom の質量と位置。

- **引数**

  m: 入力。const PS::F32 型。質量

  p: 入力。const PS::F32vec &型。位置。

- **返値**

  なし

- **機能**

  Mom クラスのオブジェクトの初期化をする。

- **備考**

  メンバ変数名の変更可能。メンバ変数を加えることも可能。

7.5.3.3 Mom::init

```
class Mom {
public:
    void init();
};
```

- **前提**

  なし

- **引数**

  なし

- **返値**

  なし

- **機能**

  Mom クラスのオブジェクトの初期化をする。

- **備考**

  なし

7.5.3.4 Mom::getPos

```
class Mom {
public:
    PS::F32vec pos;
    PS::F32vec getPos() const {
        return pos;
    }
};
```

- **前提**

  Mom のメンバ変数 pos は近傍でまとめた粒子の代表位置。この pos のデータ型は PS::F32vec または PS::F64vec 型。

- **引数**

  なし

- **返値**

  PS::F32vec または PS::F64vec 型。Mom クラスのメンバ変数 pos。

- **機能**

  Mom クラスのメンバ変数 pos を返す。

- **備考**

  Mom クラスのメンバ変数 pos の変数名は変更自由。

7.5.3.5 Mom::getCharge

```
class Mom {
public:
    PS::F32 mass;
    PS::F32 getCharge() const {
        return mass;
    }
};
```

- **前提**

  Mom のメンバ変数 mass は近傍でまとめた粒子の全質量または全電荷。この mass のデータ型は PS::F32 または PS::F64 型。

- **引数**

  なし

- **返値**

  PS::F32 または PS::F64 型。Mom クラスのメンバ変数 mass。

- **機能**

  Mom クラスのメンバ変数 mass を返す。

- **備考**

  Mom クラスのメンバ変数 mass の変数名は変更自由。

7.5.3.6 Mom::accumulateAtLeaf

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    template <class Tepj>
    void accumulateAtLeaf(const Tepj & epj) {
        mass += epj.getCharge();
        pos  += epj.getPos();
    }
};
```

- **前提**

  Mom のメンバ変数 mass は近傍でまとめた粒子の全質量または全電荷。この mass のデータ型は PS::F32 または PS::F64 型。Mom のメンバ変数 pos は近傍でまとめた粒子の代表位置。この pos のデータ型は PS::F32vec または PS::F64vec 型。テンプレート引数 Tepj には EssentialParticleJ クラスが入り、クラスはメンバ関数 getCharge と getPos を持つ。

- **引数**

  epj: 入力。const Tepj &型。Tepj のオブジェクト。

- **返値**

  なし。

- **機能**

  EssentialParticleJ クラスのオブジェクトからモーメントを計算する。

- **備考**

  Mom クラスのメンバ変数 mass, pos の変数名は変更自由。引数 epj の引数名は変更自由。その他の変数を加えるのも可能。

7.5.3.7 Mom::accumulate

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    void accumulate(const Mom & mom) {
        mass += mom.mass;
        pos  += mom.mass * mom.pos;
    }
};
```

- **前提**

  Mom のメンバ変数 mass は近傍でまとめた粒子の全質量または全電荷。この mass のデータ型は PS::F32 または PS::F64 型。Mom のメンバ変数 pos は近傍でまとめた粒子の重心または電荷の重心。この pos のデータ型は PS::F32vec または PS::F64vec 型。

- **引数**

  mom: 入力。const Mom &型。Mom クラスのオブジェクト。

- **返値**

  なし。

- **機能**

  Mom クラスのオブジェクトからさらに Mom クラスの情報を計算する。

- **備考**

  Mom クラスのメンバ変数 mass, pos の変数名は変更自由。引数 epj の引数名は変更自由。その他の変数を加えるのも可能。

7.5.3.8 Mom::set

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    void set() {
        pos = pos / mass;
    }
};
```

- **前提**

  Mom のメンバ変数 mass は近傍でまとめた粒子の全質量または全電荷。この mass のデータ型は PS::F32 または PS::F64 型。Mom のメンバ変数 pos は近傍でまとめた粒子の重心または電荷の重心。この pos のデータ型は PS::F32vec または PS::F64vec 型。

- **引数**

  なし

- **返値**

  なし

- **機能**

  上記のメンバ関数 Mom::accumulateAtLeaf, Mom::accumulate ではモーメントの位置情報の規格化ができていない場合ので、ここで規格化する。

- **備考**

  Mom クラスのメンバ変数 mass, pos の変数名は変更自由。引数 epj の引数名は変更自由。

7.5.3.9 Mom::accumulateAtLeaf2

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    PS::F32mat quad;
    template <class Tepj>
    void accumulateAtLeaf2(const Tepj & epj) {
        PS::F64 ctmp    = epj.getCharge();
        PS::F64vec ptmp = epj.getPos() - pos;
        PS::F64 cx = ctmp * ptmp.x;
        PS::F64 cy = ctmp * ptmp.y;
        PS::F64 cz = ctmp * ptmp.z;
        quad.xx += cx * ptmp.x;
        quad.yy += cy * ptmp.y;
        quad.zz += cz * ptmp.z;
        quad.xy += cx * ptmp.y;
        quad.xz += cx * ptmp.z;
        quad.yz += cy * ptmp.z;
    }
};
```

- **前提**

  Mom のメンバ変数 mass は近傍でまとめた粒子の全質量または全電荷。この mass のデータ型は PS::F32 または PS::F64 型。Mom のメンバ変数 pos は近傍でまとめた粒子の代表位置。この pos のデータ型は PS::F32vec または PS::F64vec 型。この pos は Mom::accumulateAtLeaf ですでに求めている。Mom のメンバ変数 quad は近傍でまとめた粒子の四重極子。この quad のデータ型は PS::F32mat または PS::F64mat 型。テンプレート引数 Tepj には EssentialParticleJ クラスが入り、クラスはメンバ関数 getCharge と getPos を持つ。

- **引数**

  epj: 入力。const Tepj &型。Tepj のオブジェクト。

- **返値**

  なし。

- **機能**

  EssentialParticleJ クラスのオブジェクトからモーメントを計算する。

- **備考**

  Mom クラスのメンバ変数 mass, pos, quad の変数名は変更自由。引数 epj の引数名は変更自由。その他の変数を加えるのも可能。

7.5.3.10 Mom::accumulate2

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    PS::F32mat quad;
    void accumulate(const Mom & mom) {
        PS::F64 mtmp    = mom.mass;
        PS::F64vec ptmp = mom.pos - pos;
        PS::F64 cx = mtmp * ptmp.x;
        PS::F64 cy = mtmp * ptmp.y;
        PS::F64 cz = mtmp * ptmp.z;
        quad.xx += cx * ptmp.x + mom.quad.xx;
        quad.yy += cy * ptmp.y + mom.quad.yy;
        quad.zz += cz * ptmp.z + mom.quad.zz;
        quad.xy += cx * ptmp.y + mom.quad.xy;
        quad.xz += cx * ptmp.z + mom.quad.xz;
        quad.yz += cy * ptmp.z + mom.quad.yz;
    }
};
```

- **前提**

  Mom のメンバ変数 mass は近傍でまとめた粒子の全質量または全電荷。この mass のデータ型は PS::F32 または PS::F64 型。Mom のメンバ変数 pos は近傍でまとめた粒子の代表位置。この pos のデータ型は PS::F32vec または PS::F64vec 型。この pos は Mom::accumulate ですでに求めている。Mom のメンバ変数 quad は近傍でまとめた粒子の四重極子。この quad のデータ型は PS::F32mat または PS::F64mat 型。

- **引数**

  mom: 入力。const Mom &型。Mom クラスのオブジェクト。

- **返値**

  なし。

- **機能**

Mom クラスのオブジェクトからさらに Mom クラスの情報を計算する。

- **備考**

  Mom クラスのメンバ変数 mass, pos, quad の変数名は変更自由。引数 epj の引数名は変更自由。その他の変数を加えるのも可能。

7.6 SuperParticleJ クラス

7.6.1 概要

SuperParticleJ クラスは近い粒子同士でまとまった複数の粒子を代表してまとめた超粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。このクラスが必要となるのは PS::SEARCH_MODE に PS::SEARCH_MODE_LONG または PS::SEARCH_MODE_LONG_CUTO を選んだ場合だけである。このクラスのメンバ関数には、超粒子の位置情報を FDPS 側とやりとりするメンバ関数がある。また、超粒子の情報と Moment クラスの情報は対になるものである。従って、このクラスのメンバ関数には、Moment クラスからこのクラスへ情報を変換 (またはその逆変換) するメンバ関数がある。

SuperParticleJ クラスも Moment クラス同様、ある程度決っているものが多いので、それらについては FDPS 側で用意した。以下、既存のクラス、SuperParticleJ クラスを作るときに必要なメンバ関数、場合によっては必要なメンバ関数について記述する。

7.6.2 既存のクラス

FDPS はいくつかの SuperParticleJ クラスを用意している。以下、各 PS::SEARCH_MODE に対し選ぶことのできるクラスについて記述する。まず、PS::SEARCH_MODE_LONG の場合、次に PS::SEARCH_MODE_LONG_CUTOFF の場合について記述する。その他の PS::SEARCH_MODE では超粒子を必要としない。

7.6.2.1 PS::SEARCH_MODE_LONG

7.6.2.1.1 PS::SPJMonopole

単極子までの情報を持つ Moment クラス PS::MomentMonopole と対になる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJMonopole {
    public:
        F64    mass;
        F64vec pos;
    };
}
```

- クラス名 PS::SPJMonopole
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心
- 使用条件

  Moment クラスである PS::MomentMonopole クラスの使用条件に準ずる。

7.6.2.1.2 PS::SPJQuadrupole

単極子と四重極子を情報を持つ Moment クラス PS::MomentQuadrupole と対になる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJQuadrupole {
    public:
        F32    mass;
        F32vec pos;
        F32mat quad;
    };
}
```

- クラス名 PS::SPJQuadrupole
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

  quad: 近傍でまとめた粒子の四重極子
- 使用条件

  Moment クラスである PS::MomentQuadrupole クラスの使用条件に準ずる。

7.6.2.1.3 PS::SPJMonopoleGeometricCenter

単極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心)Moment クラス PS::MomentMonopoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJMonopoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
    };
}
```

- クラス名 PS::SPJMonopoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心
- 使用条件

  PS::MomentMonopoleGeometricCenter の使用条件に準ずる。

7.6.2.1.4 PS::SPJDipoleGeometricCenter

双極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心)Moment クラス PS::MomentDipoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJDipoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
    };
}
```

- クラス名 PS::SPJDipoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心

  dipole: 粒子の質量または電荷の双極子
- 使用条件

  PS::MomentDipoleGeometricCenter の使用条件に準ずる。

7.6.2.1.5 PS::SPJQuadrupoleGeometricCenter

四重極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心)Moment クラス PS::MomentQuadrupoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJQuadrupoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
        F32mat quadrupole;
    };
}
```

- クラス名 PS::SPJQuadrupoleGeometricCenter
- メンバ変数とその情報

  charge: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の幾何中心

  dipole: 粒子の質量または電荷の双極子

  quadrupole: 粒子の質量または電荷の四重極子
- 使用条件

  PS::MomentQuadrupoleGeometricCenter の使用条件に準ずる。

7.6.2.2 PS::SEARCH_MODE_LONG_CUTOFF

7.6.2.2.1 PS::SPJMonopoleCutoff

単極子までを情報として持つクラス Moment クラス PS::MomentMonopoleCutoff と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJMonopoleCutoff {
    public:
        F32    mass;
        F32vec pos;
    };
}
```

- クラス名 PS::SPJMonopoleCutoff
- メンバ変数とその情報

  mass: 近傍でまとめた粒子の全質量、または全電荷

  pos: 近傍でまとめた粒子の重心、または粒子電荷の重心
- 使用条件

  PS::MomentMonopoleCutoff の使用条件に準ずる。

7.6.3 必要なメンバ関数

7.6.3.1 概要

以下では SuperParticleJ クラスを作る際に必要なメンバ関数を記述する。このとき SuperParticleJ クラスのクラス名を SPJ とする。これは変更自由である。

7.6.3.2 SPJ::getPos

```
class SPJ {
public:
    PS::F64vec pos;
    PS::F64vec getPos() const {
        return this->pos;
    }
};
```

- **前提**

  SPJ のメンバ変数 pos はある 1 つの超粒子の位置情報。この pos のデータ型は PS::F32vec または PS::F64vec 型。
- **引数**

  なし
- **返値**

  PS::F32vec 型または PS::F64vec 型。SPJ クラスの位置情報を保持したメンバ変数。
- **機能**

  SPJ クラスの位置情報を保持したメンバ変数を返す。
- **備考**

  SPJ クラスのメンバ変数 pos の変数名は変更可能。

7.6.3.3 SPJ::setPos

```
class SPJ {
public:
    PS::F64vec pos;
    void setPos(const PS::F64vec pos_new) {
        this->pos = pos_new;
    }
};
```

- **前提**

  SPJ クラスのメンバ変数 pos は 1 つの粒子の位置情報。この pos のデータ型は PS::F32vec または PS::F64vec。

- **引数**

  pos_new: 入力。const PS::F32vec または const PS::F64vec 型。FDPS 側で修正した粒子の位置情報。

- **返値**

  なし。

- **機能**

  FDPS が修正した粒子の位置情報を SPJ クラスの位置情報に書き込む。

- **備考**

  SPJ クラスのメンバ変数 pos の変数名は変更可能。メンバ関数 SPJ::setPos の引数名 pos_new は変更可能。pos と pos_new のデータ型が異なる場合の動作は保証しない。

7.6.3.4 SPJ::copyFromMoment

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
}
class SPJ {
public:
    PS::F32    mass;
    PS::F32vec pos;
    void copyFromMoment(const Mom & mom) {
        mass = mom.mass;
        pos  = mom.pos;
    }
};
```

- **前提**

  なし

- **引数**

  mom: 入力。const Mom &型。Mom にはユーザー定義または FDPS 側で用意した Moment クラスが入る。

- **返値**

  なし。

- **機能**

  Mom クラスの情報を SPJ クラスにコピーする。

- **備考**

  Mom クラスのクラス名は変更可能。Mom クラスと SPJ クラスのメンバ変数名は変更可能。メンバ関数 SPJ::copyFromMoment の引数名は変更可能。

7.6.3.5 SPJ::convertToMoment

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    Mom(const PS::F32 m,
        const PS::F32vec & p) {
        mass = m;
        pos  = p;
    }
}
class SPJ {
public:
    PS::F32    mass;
    PS::F32vec pos;
    Mom convertToMoment() const {
        return Mom(mass, pos);
    }
};
```

- **前提**

  なし

- **引数**

  なし

- **返値**

  Mom 型。Mom クラスのコンストラクタ。

- **機能**

  Mom クラスのコンストラクタを返す。

- **備考**

  Mom クラスのクラス名は変更可能。Mom クラスと SPJ クラスのメンバ変数名は変更可能。メンバ関数SPJ::copyFromMoment の引数名は変更可能。メンバ関数SPJ::convertToMoment で使用される Mom クラスのコンストラクタが定義されている必要がある。

7.6.3.6 SPJ::clear

```
class SPJ {
public:
    PS::F32    mass;
    PS::F32vec pos;
    void clear() {
        mass = 0.0;
        pos  = 0.0;
    }
};
```

- **前提**

  なし

- **引数**

  なし

- **返値**

  なし

- **機能**

  SPJ クラスのオブジェクトの情報をクリアする。

- **備考**

  メンバ変数名は変更可能。

7.7 Force クラス

7.7.1 概要

Force クラスは相互作用の結果を保持するクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、この節の前提、常に必要なメンバ関数について記述する。

7.7.2 前提

この節で用いる例として Force クラスのクラス名を Result とする。このクラス名は変更自由である。

7.7.3 必要なメンバ関数

常に必要なメンバ関数は Result::clear である。この関数は相互作用の計算結果を初期化する。以下、Result::clear について記述する。

7.7.3.1 Result::clear

```
class Result {
public:
    PS::F32vec acc;
    PS::F32    pot;
    void clear() {
        acc = 0.0;
        pot = 0.0;
    }
};
```

- **前提**

  Result クラスのメンバ変数は acc と pot。

- **引数**

  なし

- **返値**

  なし。

- **機能**

  Result クラスのメンバ変数を初期化する。

- **備考**

  Result クラスのメンバ変数 acc, pot の変数名は変更可能。他のメンバ変数を加えることも可能。

7.8 ヘッダクラス

7.8.1 概要

ヘッダクラスは入出力ファイルのヘッダの形式を決めるクラスである。ヘッダクラスは FDPS が提供する粒子群クラスのファイル入出力 API を使用し、かつ入出力ファイルにヘッダを含ませたい場合に必要となるクラスである。粒子群クラスのファイル入出力 API とは、ParticleSystem::readParticleAscii, ParticleSystem::writeParticleAscii, である。以下、この節

における前提と、これらの API を使用する際に必要となるメンバ関数とその記述の規定を述べる。この節において、常に必要なメンバ関数というものは存在しない。

7.8.2 前提

この節では、ヘッダクラスのクラス名を Hdr とする。このクラス名は変更可能である。

7.8.3 場合によっては必要なメンバ関数

7.8.3.1 Hdr::readAscii

```
class Hdr {
public:
    PS::S32 nparticle;
    PS::F64 time;
    PS::S32 readAscii(FILE *fp) {
        fscanf(fp, "%d%lf", &this->nparticle, &this->time);
        return this->nparticle;
    }
};
```

- **前提**

  このヘッダは粒子数、時刻の情報を持つ。これらのメンバ変数はそれぞれ nparticle と time である。

- **引数**

  fp: 入力。FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- **返値**

  PS::S32 型。粒子数の情報を返す。ヘッダに粒子数の情報がない場合は-1 を返す。

- **機能**

  粒子データの入力ファイルからヘッダ情報を読みこむ。

- **備考**

  メンバ変数名は入力ファイルに合わせて変更可能。返値に粒子数の情報を指定しない場合、または-1 を指定しない場合の動作は保証しない。

7.8.3.2 Hdr::writeAscii

```
class Hdr {
public:
    PS::S32 nparticle;
    PS::F64 time;
    void writeAscii(FILE *fp) {
        fprintf(fp, "%d %lf", this->nparticle, this->time);
    }
};
```

- **前提**

  このヘッダは粒子数、時刻の情報を持つ。これらのメンバ変数はそれぞれ nparticle と time である。

- **引数**

  fp: 入力。FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- **返値**

  なし。

- **機能**

  粒子データの出力ファイルへヘッダ情報を書き込む。

- **備考**

  メンバ変数名は出力ファイルに合わせて変更可能。

7.9 関数オブジェクト calcForceEpEp

7.9.1 概要

関数オブジェクト calcForceEpEp は粒子同士の相互作用を記述するものであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、これの書き方の規定を記述する。

7.9.2 前提

ここで示すのは重力 N 体シミュレーションの粒子間相互作用の記述の仕方である。関数オブジェクト calcForceEpEp の名前は gravityEpEp とする。これは変更自由である。また、EssentialParitlceI クラスのクラス名を EPI, EssentialParitlceJ クラスのクラス名を EPJ, Force クラスのクラス名を Result とする。

7.9.3 gravityEpEp::operator ()

ソースコード 14: calcForceEpEp

```
1  class Result {
2  public:
3      PS::F32vec acc;
4  };
5  class EPI {
6  public:
7      PS::S32    id;
8      PS::F32vec pos;
9  };
10 class EPJ {
11 public:
12     PS::S32    id;
13     PS::F32    mass;
14     PS::F32vec pos;
15 };
16 struct gravityEpEp {
17     static PS::F32 eps2;
18     void operator () (const EPI *epi,
19                       const PS::S32 ni,
20                       const EPJ *epj,
21                       const PS::S32 nj,
22                       Result *result) {
23
24         for(PS::S32 i = 0; i < ni; i++) {
25             PS::S32    ii = epi[i].id;
26             PS::F32vec xi = epi[i].pos;
27             PS::F32vec ai = 0.0;
28             for(PS::S32 j = 0; j < nj; j++) {
29                 PS::S32    jj = epj[j].id;
30                 PS::F32    mj = epj[j].mass;
31                 PS::F32vec xj = epj[j].pos;
32
33                 PS::F32vec dx   = xi - xj;
34                 PS::F32    r2   = dx * dx + eps2;
35                 PS::F32    rinv = (ii != jj) ? 1. / sqrt(r2)
36                                              : 0.0;
37
```

```
38                  ai += mj * rinv * rinv * rinv * dx;
39              }
40              result.acc = ai;
41          }
42      }
43  };
44  PS::F32 gravityEpEp::eps2 = 9.765625e-4;
```

- **前提**

  クラス Result, EPI, EPJ に必要なメンバ関数は省略した。クラス Result のメンバ変数 acc は i 粒子が j 粒子から受ける重力加速度である。クラス EPI と EPJ のメンバ変数 id と pos はそれぞれの粒子 ID と粒子位置である。クラス EPJ のメンバ変数 mass は j 粒子の質量である。関数オブジェクト gravityEpEp のメンバ変数 eps2 は重力ソフトニングの 2 乗である。ここの外側でスレッド並列になっているため、ここで OpenMP を記述する必要はない。

- **引数**

  epi: 入力。const EPI *型または EPI *型。i 粒子情報を持つ配列。

  ni: 入力。const PS::S32 型または PS::S32 型。i 粒子数。

  epj: 入力。const EPJ *型または EPJ *型。j 粒子情報を持つ配列。

  nj: 入力。const PS::S32 型または PS::S32 型。j 粒子数。

  result: 出力。Result *型。i 粒子の相互作用結果を返す配列。

- **返値**

  なし。

- **機能**

  j 粒子から i 粒子への作用を計算する。

- **備考**

  引数名すべて変更可能。関数オブジェクトの内容などはすべて変更可能。

7.10 関数オブジェクト calcForceSpEp

7.10.1 概要

関数オブジェクト calcForceSpEp は超粒子から粒子への作用を記述するものであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、これの書き方の規定を記述する。

7.10.2 前提

ここで示すのは重力N体シミュレーションにおける超粒子から粒子への作用の記述の仕方である。超粒子は単極子までの情報で作られているものとする。関数オブジェクトcalcForceSpEpの名前はgravitySpEpとする。これは変更自由である。また、EssentialParitlceIクラスのクラス名をEPI, SuperParitlceJクラスのクラス名をSPJ, Forceクラスのクラス名をResultとする。

7.10.3 gravitySpEp::operator ()

ソースコード 15: calcForceSpEp

```
1  class Result {
2  public:
3      PS::F32vec accfromspj;
4  };
5  class EPI {
6  public:
7      PS::S32    id;
8      PS::F32vec pos;
9  };
10 class SPJ {
11 public:
12     PS::F32    mass;
13     PS::F32vec pos;
14 };
15 struct gravitySpEp {
16     static PS::F32 eps2;
17     void operator () (const EPI *epi,
18                       const PS::S32 ni,
19                       const SPJ *spj,
20                       const PS::S32 nj,
21                       Result *result) {
22
23         for(PS::S32 i = 0; i < ni; i++) {
24             PS::F32vec xi = epi[i].pos;
25             PS::F32vec ai = 0.0;
26             for(PS::S32 j = 0; j < nj; j++) {
27                 PS::F32    mj = spj[j].mass;
28                 PS::F32vec xj = spj[j].pos;
29
```

```
30                  PS::F32vec dx   = xi - xj;
31                  PS::F32    r2   = dx * dx + eps2;
32                  PS::F32    rinv = 1. / sqrt(r2);
33
34                  ai += mj * rinv * rinv * rinv * dx;
35              }
36              result.accfromspj = ai;
37          }
38      }
39  };
40  PS::F32 gravitySpEp::eps2 = 9.765625e-4;
```

- **前提**

  クラス Result, EPI, SPJ に必要なメンバ関数は省略した。クラス Result のメンバ変数 accfromspj は i 粒子が超粒子から受ける重力加速度である。クラス EPI と SPJ のメンバ変数 pos はそれぞれの粒子位置である。クラス SPJ のメンバ変数 mass は超粒子の質量である。ファンクタ gravitySpEp のメンバ変数 eps2 は重力ソフトニングの 2 乗である。ここの外側でスレッド並列になっているため、ここで OpenMP を記述する必要はない。

- **引数**

  epi: 入力。const EPI *型または EPI *型。i 粒子情報を持つ配列。

  ni: 入力。const PS::S32 型または PS::S32 型。i 粒子数。

  spj: 入力。const SPJ *型または SPJ *型。超粒子情報を持つ配列。

  nj: 入力。const PS::S32 型または PS::S32 型。超粒子数。

  result: 出力。Result *型。i 粒子の相互作用結果を返す配列。

- **返値**

  なし。

- **機能**

  超粒子から i 粒子への作用を計算する。

- **備考**

  引数名すべて変更可能。関数オブジェクトの内容などはすべて変更可能。

8 プログラムの開始と終了

8.1 概要

プログラムの開始と終了に必要な API などを記述する。

8.2 API

8.2.1 PS::Initialize

プログラムの開始を行うには以下の API を呼び出す必要がある。

```
void PS::Initialize
              (PS::S32 & argc,
               char ** & argv);
```

- **引数**

  argc: 入力。PS::S32 型。コマンドライン引数の総数。

  argv: 入力。char ** &型。コマンドライン引数の文字列を指すポインタのポインタ。

- **返値**

  なし

- **機能**

  FDPS の初期化を行う。FDPS の API のうち最初に呼び出さなければならない。内部では MPI::Init を呼び出すため、引数 argc と argv が変っている可能性がある。

8.2.2 PS::Finalize

プログラムの終了するには以下の API を呼び出す必要がある。

```
void PS::Finalize();
```

- **引数**

  なし

- **返値**

  なし

- **機能**

  FDPS の終了処理を行う。

8.2.3 PS::DisplayInfo

```
void PS::DisplayInfo();
```

- **引数**

  なし

- **返値**

  なし

- **機能**

  FDPS のライセンス情報などを表示する。

9　モジュール

本節では、FDPS のモジュールについて記述する。最初に FDPS の標準機能について、次に FDPS の拡張機能について記述する。

9.1　標準機能

9.1.1　概要

本節では、FDPS の標準機能について記述する。標準機能には４つのモジュールがあり、領域クラス、粒子群クラス、相互作用ツリークラス、通信用データクラスがある。この４つのクラスについて順に記述する。

9.1.2　領域クラス

本節では、領域クラスについて記述する。このクラスは領域情報の保持や領域の分割を行うモジュールである。まずオブジェクトの生成方法を記述し、その後 API を記述する。

9.1.2.1　オブジェクトの生成

領域クラスは以下のように宣言されている。

ソースコード 16: DomainInfo0

```
1 namespace ParticleSimulator {
2     class DomainInfo;
3 }
```

領域クラスのオブジェクトの生成は以下のように行う。ここでは dinfo というオブジェクトを生成している。

```
PS::DomainInfo dinfo;
```

9.1.2.2　API

領域クラスには初期設定関連の API、領域分割関連の API がある。以下、各節に分けて記述する。

9.1.2.2.1　初期設定

初期設定関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 17: DomainInfo1

```
1  namespace ParticleSimulator {
2      class DomainInfo{
3      public:
4          DomainInfo();
5          void initialize(const F32 coef_ema=1.0);
6          void setNumberOfDomainMultiDimension(const S32 nx,
7                                               const S32 ny,
8                                               const S32 nz=1);
9          void setBoundaryCondition(enum BOUNDARY_CONDITION bc);
10         void setPosRootDomain(const F32vec & low,
11                               const F32vec & high);
12     };
13 }
```

9.1.2.2.1.1 コンストラクタ

コンストラクタ

```
void PS::DomainInfo::DomainInfo();
```

- **引数**

  なし

- **返値**

  なし

- **機能**

  領域クラスのオブジェクトを生成する。

9.1.2.2.1.2 PS::DomainInfo::initialize

PS::DomainInfo::initialize

```
void PS::DomainInfo::initialize(const PS::F32 coef_ema=1.0);
```

- **引数**

  coef_ema: 入力。 const PS::F32 型。指数移動平均の平滑化係数。デフォルト 1.0

- **返値**

  なし

- **機能**

  領域クラスのオブジェクトを初期化する。

  指数移動平均の平滑化係数を設定する。この係数の許される値は 0 から 1 である。大きくなるほど、最新の粒子分布の情報が領域分割に反映されやすい。1 の場合、最新の粒子分布の情報のみ反映される。0 の場合、最初の粒子分布の情報のみ反映される。１度は呼ぶ必要があるが、２度呼ぶと例外が送出される。過去の粒子分布の情報を領域分割に反映する必要がある理由については、Ishiyama, Fukushige & Makino (2009, Publications of the Astronomical Society of Japan, 61, 1319) を参照のこと。

9.1.2.2.1.3 PS::DomainInfo::setNumberOfDomainMultiDimension

PS::DomainInfo::setNumberOfDomainMultiDimension

```
void PS::DomainInfo::setNumberOfDomainMultiDimension
                  (const PS::S32 nx,
                   const PS::S32 ny,
                   const PS::S32 nz=1);
```

- **引数**

  nx: 入力。 const PS::S32 型。x 軸方向のルートドメインの分割数。

  ny: 入力。 const PS::S32 型。y 軸方向のルートドメインの分割数。

  nz: 入力。 const PS::S32 型。z 軸方向のルートドメインの分割数。デフォルト 1。

- **返値**

  なし

- **機能**

  計算領域の分割する方法を設定する。nx, ny, nz はそれぞれ x 軸、y 軸、z 軸方向の計算領域の分割数である。呼ばなければ自動的に nx, ny, nz が決まる。呼んだ場合に入力する nx, ny, nz の総積が MPI プロセス数と等しくなければ、例外が送出される。

9.1.2.2.1.4 PS::DomainInfo::setBoundaryCondition

PS::DomainInfo::setBoundaryCondition

```
void PS::DomainInfo::setBoundaryCondition
                  (enum PS::BOUNDARY_CONDITION bc);
```

- **引数**

  bc: 入力。 列挙型。境界条件。

- **返値**

  なし

- **機能**

  境界条件の設定をする。許される入力は、6.7.2 で挙げた列挙型のみ (ただし BOUNDARY_CONDITION_SHEARING_BOX, BOUNDARY_CONDITION_USER_DEFINED は未実装)。呼ばない場合は、開放境界となる。

9.1.2.2.1.5 PS::DomainInfo::setPosRootDomain

PS::DomainInfo::setPosRootDomain

```
void PS::DomainInfo::setPosRootDomain
                (const PS::F32vec & low,
                 const PS::F32vec & high);
```

- **引数**

  low: 入力。 PS::F32vec 型。計算領域の下限 (閉境界)。

  high: 入力。 PS::F32vec 型。計算領域の上限 (解境界)。

- **返値**

  なし

- **機能**

  計算領域の下限と上限を設定する。開放境界条件の場合は呼ぶ必要はない。それ以外の境界条件の場合は、呼ばなくても動作するが、その結果が正しいことは保証できない。

9.1.2.2.2 領域分割

領域分割関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 18: DomainInfo2

```
1 namespace ParticleSimulator {
2     class DomainInfo{
3     public:
4         template<class Tpsys>
5         void collectSampleParticle(Tpsys & psys,
6                                    const bool clear,
```

```
 7                                       const F32 weight);
 8         template<class Tpsys>
 9         void collectSampleParticle(Tpsys & psys,
10                                       const bool clear);
11         template<class Tpsys>
12         void collectSampleParticle(Tpsys & psys);
13
14         void decomposeDomain();
15
16         template<class Tpsys>
17         void decomposeDomainAll(Tpsys & psys,
18                                 const F32 weight);
19         template<class Tpsys>
20         void decomposeDomainAll(Tpsys & psys);
21     };
22 }
```

9.1.2.2.2.1 PS::DomainInfo::collectSampleParticle

PS::DomainInfo::collectSampleParticle

```
template<class Tpsys>
void PS::DomainInfo::collectSampleParticle
                 (Tpsys & psys,
                  const bool clear,
                  const PS::F32 weight);
```

- **引数**

  psys: 入力。 Tpsys &型。領域分割のためのサンプル粒子を提供する粒子群クラス。

  clear: 入力。 bool 型。前にサンプルされた粒子情報をクリアするかどうかを決定するフラグ。true でクリアする。

  weight: 入力。 const PS::F32 型。領域分割のためのサンプル粒子数を決めるためのウェイト。

- **返値**

  なし

- **機能**

粒子群クラスのオブジェクト psys から粒子をサンプルする。clear によってこれより前にサンプルした粒子の情報を消すかどうか決める。weight によってその MPI プロセスからサンプルする粒子の量を調整する (weight が大きいほどサンプル粒子数が多い)。

```
template<class Tpsys>
void PS::DomainInfo::collectSampleParticle
                  (Tpsys & psys,
                   const bool clear);
```

- **引数**

  psys: 入力。 Tpsys &型。領域分割のためのサンプル粒子を提供する粒子群クラス。

  clear: 入力。 bool 型。前にサンプルされた粒子情報をクリアするかどうかを決定するフラグ。true でクリアする。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys から粒子をサンプルする。clear によってこれより前にサンプルした粒子の情報を消すかどうか決める。

```
template<class Tpsys>
void PS::DomainInfo::collectSampleParticle
                  (Tpsys & psys);
```

- **引数**

  psys: 入力。 Tpsys &型。領域分割のためのサンプル粒子を提供する粒子群クラス。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys から粒子をサンプルする。

9.1.2.2.2.2 PS::DomainInfo::decomposeDomain

PS::DomainInfo::decomposeDomain

```
void PS::DomainInfo::decomposeDomain();
```

- **引数**

  なし

- **返値**

  なし

- **機能**

  計算領域の分割を実行する。

9.1.2.2.2.3　PS::DomainInfo::decomposeDomainAll

PS::DomainInfo::decomposeDomainAll

```
template<class Tpsys>
void PS::DomainInfo::decomposeDomainAll
                 (Tpsys & psys,
                  const PS::F32 weight);
```

- **引数**

  psys: 入力。 Tpsys &型。領域分割のためのサンプル粒子を提供する粒子群クラス。

  weight: 入力。 const PS::F32 型。領域分割のためのサンプル粒子数を決めるためのウェイト。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys から粒子をサンプルし、続けてルートドメインの分割を行う。PS::DomainInfo::collectSampleParticle と PS::DomainInfo::decomposeDomain が行うことを一度に行う。weight の意味は PS::DomainInfo::collectSampleParticle と同じ。

```
template<class Tpsys>
void PS::DomainInfo::decomposeDomainAll
                 (Tpsys & psys);
```

- **引数**

  psys: 入力。 Tpsys &型。領域分割のためのサンプル粒子を提供する粒子群クラスのオブジェクト。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys から粒子をサンプルし、続けてルートドメインの分割を行う。PS::DomainInfo::collectSampleParticle と PS::DomainInfo::decomposeDomain が行うことを一度に行う。

9.1.3 粒子群クラス

本節では、粒子群クラスについて記述する。このクラスは粒子情報の保持や MPI プロセス間で粒子情報の交換を行うモジュールである。まずオブジェクトの生成方法を記述し、その後 API を記述する。

9.1.3.1 オブジェクトの生成

粒子群クラスは以下のように宣言されている。

ソースコード 19: ParticleSystem0

```
1 namespace ParticleSimulator {
2     template<class Tptcl>
3     class ParticleSystem;
4 }
```

テンプレート引数 Tptcl はユーザー定義の FullParticle クラスである。

粒子群クラスのオブジェクトの生成は以下のように行う。ここでは system というオブジェクトを生成している。

```
PS::ParticleSystem<FP> system;
```

テンプレート引数 FP はユーザー定義の FullParticle クラスの 1 例である FP クラスである。

9.1.3.2 API

このモジュールには初期設定関連の API、オブジェクト情報取得設定関連の API、ファイル入出力関連の API、粒子交換関連の API がある。以下、各節に分けて記述する。

9.1.3.2.1 初期設定

初期設定関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 20: ParticleSystem1

```
1 namespace ParticleSimulator {
2     template<class Tptcl>
3     class ParticleSystem{
4     public:
5         ParticleSystem();
6         void initialize();
7         void setAverageTargetNumberOfSampleParticlePerProcess
8                         (const S32 & nsampleperprocess);
9     };
10 }
```

9.1.3.2.1.1 コンストラクタ

コンストラクタ

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::ParticleSystem();
```

- **引数**

  なし

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクトを生成する。

9.1.3.2.1.2 PS::ParticleSystem::initialize

PS::ParticleSystem::initialize

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::initialize();
```

- **引数**

  なし

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクトを初期化する。１度は呼ぶ必要があるが、２度呼ぶと例外が送出される。

9.1.3.2.1.3　PS::ParticleSystem::setAverateTargetNumberOfSampleParticlePerProcess

PS::ParticleSystem::setAverateTargetNumberOfSampleParticlePerProcess

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::setAverateTargetNumberOfSampleParticlePerProcess
                (const PS::S32 & nsampleperprocess);
```

- **引数**

  nsampleperprocess: 入力。const PS::S32 &型。１つの MPI プロセスでサンプルする粒子数目標。

- **返値**

  なし

- **機能**

  １つの MPI プロセスでサンプルする粒子数の目標を設定する。呼び出さなくてもよいが、呼び出さないとこの目標数が 30 となる。

9.1.3.2.2　オブジェクト情報の取得設定

オブジェクト情報取得関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 21: ParticleSystem2

```
1  namespace ParticleSimulator {
2      template<class Tptcl>
3      class ParticleSystem{
4      public:
5          Tptcl & operator [] (const S32 id);
6          void setNumberOfParticleLocal(const S32 n);
7          S32 getNumberOfParticleLocal() const;
8          S32 getNumberOfParticleGlobal() const;
9      };
10 }
```

9.1.3.2.2.1　PS::ParticleSystem::operator []

PS::ParticleSystem::operator []

```
template <class Tptcl>
Tptcl & PS::ParticleSystem<Tptcl>::operator []
             (const PS::S32 id);
```

- **引数**

  n: 入力。const PS::S32 型。粒子配列のインデックス。

- **返値**

  Tptcl &型。Tptcl 型のオブジェクト。

- **機能**

  Tptcl 型のオブジェクトを返す。

9.1.3.2.2.2　PS::ParticleSystem::setNumberOfParticleLocal

PS::ParticleSystem::setNumberOfParticleLocal

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::setNumberOfParticleLocal
                 (const PS::S32 n);
```

- **引数**

  n: 入力。const PS::S32 型。粒子数。

- **返値**

  なし

- **機能**

  １つの MPI プロセスの持つ粒子数を設定する。

9.1.3.2.2.3　PS::ParticleSystem::getNumberOfParticleLocal

PS::ParticleSystem::getNumberOfParticleLocal

```
template <class Tptcl>
PS::S32 PS::ParticleSystem<Tptcl>::getNumberOfParticleLocal();
```

- **引数**

  なし

- **返値**

  PS::S32 型。１つの MPI プロセスの持つ粒子数。

- **機能**

  １つの MPI プロセスの持つ粒子数を返す。

9.1.3.2.2.4 PS::ParticleSystem::getNumberOfParticleGlobal

PS::ParticleSystem::getNumberOfParticleGlobal

```
template <class Tptcl>
PS::S32 PS::ParticleSystem<Tptcl>::getNumberOfParticleGlobal();
```

- **引数**

  なし

- **返値**

  PS::S32 型。全 MPI プロセスの持つ粒子数。

- **機能**

  全 MPI プロセスの持つ粒子数を返す。

9.1.3.2.3 ファイル入出力

ファイル入出力関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 22: ParticleSystem3

```
1  namespace ParticleSimulator {
2      template<class Tptcl>
3      class ParticleSystem{
4      public:
5          template <class Theader>
6          void readParticleAscii(const char * const filename,
7                                 const char * const format,
8                                 Theader & header);
9          void readParticleAscii(const char * const filename,
10                                const char * const format);
```

```
11          template <class Theader>
12          void readParticleAscii(const char * const filename,
13                                 Theader & header);
14          void readParticleAscii(const char * const filename);
15          template <class Theader>
16          void writeParticleAscii(const char * const filename,
17                                  const char * const format,
18                                  const Theader & header);
19          void writeParticleAscii(const char * const filename,
20                                  const char * format);
21          template <class Theader>
22          void writeParticleAscii(const char * const filename,
23                                  const Theader & header);
24          void writeParticleAscii(const char * const filename);
25      };
26  }
```

9.1.3.2.3.1 PS::ParticleSystem::readParticleAscii

PS::ParticleSystem::readParticleAscii

```
template <class Tptcl>
template <class Theader>
void PS::ParticleSystem<Tptlc>::readParticleAscii
                (const char * const filename,
                 const char * const format,
                 Theader & header);
```

- **引数**

  filename: 入力。const char *型。入力ファイル名のベースとなる部分。

  format: 入力。const char *型。分散ファイルから粒子データを読み込む際のファイルフォーマット。

  header: 入力。Theader &型。ファイルのヘッダ情報。

- **返値**

  なし

- **機能**

  各プロセスが filename と format で指定された入力ファイルから粒子データを読み出し、データを FullParticle クラスのオブジェクトに格納する。

filenameで、分散しているファイルのベースとなる名前を指定する。formatでファイル名のフォーマットを指定する。フォーマットの指定方法は標準Cライブラリの関数printfの第1引数と同じである。ただし変換指定は必ず3つであり、その指定子は1つめは文字列、残りはどちらも整数である。2つ目の変換指定にはそのジョブの全プロセス数が、3つ目の変換指定にはプロセス番号が入る。例えば、filenameがnbody、formatが%s_%03d_%03d.initならば、全プロセス数64のジョブのプロセス番号12のプロセスは、nbody_064_012.init というファイルを読み込む。

1粒子のデータを読み取る関数はFullParticleクラスのメンバ関数でユーザが定義する。定義方法については節7.2.4.3を参照のこと。

ファイルのヘッダのデータを読み取る関数はTheaderのメンバ関数でユーザが定義する。定義方法については節7.8を参照のこと。

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::readParticleAscii
            (const char * const filename,
             const char * const format);
```

- **引数**

  filename: 入力。const char * const型。入力ファイル名のベースとなる部分。

  format: 入力。const char * const型。分散ファイルから粒子データを読み込む際のファイルフォーマット。

- **返り値**

  なし。

- **機能**

  各プロセスがfilenameとformatで指定された入力ファイルから粒子データを読み出し、データをFullParticleクラスのオブジェクトに格納する。この時、１回ファイルを読み込んで行数を取得した後、もう一度ファイルを読み込みなおし、粒子データを読み込む。

  filenameで、分散しているファイルのベースとなる名前を指定する。formatでファイル名のフォーマットを指定する。formatの指定の仕方は、Theaderが存在する場合の時と同様である。

  1粒子のデータを読み取る関数はFullParticleクラスのメンバ関数でユーザが定義する。このメンバ関数の書式と規約は、分散ファイルから読み出す場合と同様である。

```
template <class Tptcl>
template <class Theader>
void PS::ParticleSystem<Tptcl>::readParticleAscii
            (const char * const filename,
             Theader & header);
```

- **引数**

  filename: 入力。const char * const 型。入力ファイル名。

  header: 入力。Theader &型。ファイルのヘッダ情報。

- **返り値**

  なし。

- **機能**

  ルートプロセスが filename で指定された入力ファイルから粒子データを読み出し、データを FullParticle クラスのオブジェクトに格納した後、各プロセスに分配する。

  1 粒子のデータを読み取る関数は FullParticle のメンバ関数でユーザが定義する。ファイルのヘッダのデータを読み取る関数は Theader のメンバ関数でユーザが定義する。これら 2 つのメンバ関数の書式と規約は、分散ファイルから読み出す場合と同様である。

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::readParticleAscii
            (const char * const filename);
```

- **引数**

  filename: 入力。const char * const 型。入力ファイル名。

- **返り値**

  なし。

- **機能**

  ルートプロセスが filename で指定された入力ファイルから粒子データを読み出し、データを FullParticle クラスのオブジェクトに格納した後、各プロセスに分配する。この時、１回ファイルを読み込んで行数を取得した後、もう一度ファイルを読み込みなおし、粒子データを読み込む。

  1 粒子のデータを読み取る関数は FullParticle クラスのメンバ関数でユーザが定義する。このメンバ関数の書式と規約は、分散ファイルから読み出す場合と同様である。

9.1.3.2.3.2　PS::ParticleSystem::readParticleBinary

PS::ParticleSystem::readParticleBinary

未実装

9.1.3.2.3.3　PS::ParticleSystem::writeParticlAscii

PS::ParticleSystem::writeParticlAscii

```
template <class Tptcl>
template <class Theader>
void PS::ParticleSystem<Tptcl>::writeParticleAscii
            (const char * const filename,
             const char * const format,
             const Theader & header);
```

- **引数**

  filename: 入力。const char * const 型。出力ファイル名のベースとなる部分。

  format: 入力。const char * const 型。分散ファイルに粒子データを書き込む際のファイルフォーマット。

  header: 入力。const Theader &型。ファイルのヘッダ情報。

- **返り値**

  なし。

- **機能**

  各プロセスが filename と format で指定された出力ファイルに FullParticle クラスのオブジェクトのデータと、Theader クラスのオブジェクトのデータを出力する。出力ファイルのフォーマットはメンバ関数 PS::ParticleSystem::readParticleAscii と同様である。

  1 粒子のデータを書き込む関数は FullParticle クラスのメンバ関数でユーザが定義する。定義方法については節 7.2.4.3 を参照のこと。

  ファイルのヘッダのデータを書き込む関数はヘッダクラスのメンバ関数でユーザが定義する。定義方法については節 7.8 を参照のこと。

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::writeParticleAscii
            (const char * const filename,
             const char * const format);
```

- **引数**

  filename: 入力。const char * const 型。出力ファイル名のベースとなる部分。

  format: 入力。const char * const 型。分散ファイルに粒子データを書き込む際のファイルフォーマット。

- **返り値**

  なし。

- **機能**

  各プロセスが filename と format で指定された出力ファイルに FullParticle クラスのオブジェクトのデータを出力する。出力ファイルのフォーマットはメンバ関数 PS::ParticleSystem::readParticleAscii と同様である。

  1 粒子のデータを書き込む関数は FullParticle クラスのメンバ関数でユーザが定義する。このメンバ関数の書式と規約は、分散ファイルに書き込む場合と同様である。

```
template <class Tptcl>
template <class Theader>
void PS::ParticleSystem<Tptcl>::writeParticleAscii
            (const char * const filename,
             const Theader & header);
```

- **引数**

  filename: 入力。const char * const 型。出力ファイル名。

  header: 入力。const Theader &型。ファイルのヘッダ情報。

- **返り値**

  なし。

- **機能**

  各プロセスが filename で指定された出力ファイルに FullParticle クラスのオブジェクトのデータと、ヘッダクラスのオブジェクトのヘッダー情報を出力する。

  1 粒子のデータを書き込む関数は FullParticle クラスのメンバ関数でユーザが定義する。ファイルのヘッダのデータを書き込む関数はヘッダクラスのメンバ関数でユーザが定義する。これら 2 つのメンバ関数の書式と規約は、分散ファイルに書き込む場合と同様である。

```
void PS::ParticleSystem<Tptcl>::writeParticleAscii
            (const char * const filename);
```

- **引数**

  filename: 入力。const char * const 型。出力ファイル名。

- **返り値**

  なし。

- **機能**

  各プロセスが filename で指定された出力ファイルに FullParticle 型の粒子データを出力する。

  1 粒子のデータを書き込む関数は FullParticle のメンバ関数でユーザが定義する。このメンバ関数の書式と規約は、分散ファイルに書き込む場合と同様である。

9.1.3.2.3.4 PS::ParticleSystem::writeParticleBinary

PS::ParticleSystem::writeParticleBinary

未実装

9.1.3.2.4 粒子交換

粒子交換関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 23: ParticleSystem4

```
1 namespace ParticleSimulator {
2     template<class Tptcl>
3     class ParticleSystem{
4     public:
5         template<class Tdinfo>
6         void exchangeParticle(Tdinfo & dinfo);
7     };
8 }
```

9.1.3.2.4.1 PS::ParticleSystem::exchangeParticle

PS::ParticleSystem::exchangeParticle

```
template <class Tptcl>
template <class Tdinfo>
void PS::ParticleSystem<Tptcl>::exchangeParticle
                 (Tdinfo & dinfo);
```

- **引数**

  dinfo: 入力。Tdinfo 型。領域クラスのオブジェクト。

- **返値**

  なし

- **機能**

  粒子が適切なドメインに配置されるように、粒子の交換を行う。どのドメインにも属さない粒子が現れた場合、例外が送出される。

9.1.3.2.5 その他

その他の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 24: ParticleSystem4

```
1 namespace ParticleSimulator {
2     template<class Tptcl>
3     class ParticleSystem{
4     public:
5         template<class Tdinfo>
6         void adjustPositionIntoRootDomain
7                     (const Tdinfo & dinfo);
8     };
9 }
```

9.1.3.2.5.1 PS::ParticleSystem::adjustPositionIntoRootDomain

PS::ParticleSystem::adjustPositionIntoRootDomain

```
template <class Tptcl>
template <class Tdinfo>
void ParticleSystem<Tptcl>::adjustPositionIntoRootDomain
            (const Tdinfo & dinfo);
```

- **引数**

  dinfo: 入力。Tdinfo 型。領域クラスのオブジェクト。

- **返値**

  なし

- **機能**

  周期境界条件の場合に、計算領域からはみ出した粒子を計算領域に適切に戻す。

9.1.4 相互作用ツリークラス

本節では、相互作用ツリークラスについて記述する。このクラスは粒子間相互作用の計算を行うモジュールである。まずオブジェクトの生成方法を記述し、その後 API を記述する。

9.1.4.1 オブジェクトの生成

このクラスは以下のように宣言されている。

ソースコード 25: TreeForForce0

```
1  namespace ParticleSimulator {
2      template<class TSearchMode,
3               class TResult,
4               class TEpi,
5               class TEpj,
6               class TMomLocal,
7               class TMomGlobal,
8               class TSpj>
9      class TreeForForce;
10 }
```

テンプレート引数は順に、PS::SEARCH_MODE 型 (ユーザー選択)、Force クラス (ユーザー定義)、EssentialParticleI クラス (ユーザー定義)、EssentialParticleJ 型 (ユーザー定義)、ローカルツリーの Moment 型 (ユーザー定義)、グローバルツリーの Moment 型 (ユーザー定義)、SuperParticleJ 型 (ユーザー定義) である。

PS::SEARCH_MODE 型に応じてラッパーを用意した。これらのラッパーを使えば入力するテンプレート引数の数が減るので、こちらのラッパーを用いることを推奨する。以下、PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF, PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合のオブジェクトの生成方法を記述する。

9.1.4.1.1 PS::SEARCH_MODE_LONG

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceLong<TResult, TEpi, TEpj, TMom, TSpj>::Normal system;
```

テンプレート引数は順に、Force クラス (ユーザー定義)、EssentialParticleI クラス (ユーザー

定義)、EssentialParticleJ クラス (ユーザー定義)、ローカルツリー及びグローバルツリーの Moment クラス (ユーザー定義)、SuperParticleJ クラス (ユーザー定義) である。

あらかじめ Moment クラスと SuperParticleJ クラスを指定した型も用意した (節 7.5, 7.6 参照)。これらはモーメントの計算方法別に 6 種類ある。以下、粒子の重心を中心とした場合の単極子まで、四重極子までのモーメント計算、粒子の幾何中心を中心とした場合の単極子まで、双極子まで、四重極子までのモーメント計算、のオブジェクトの生成方法をこの順で記述する。ここでは、すべて system というオブジェクトを生成している。

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::Monopole system;
```

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::Quadrupole system;
```

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::MonopoleGeometricCenter system;
```

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::DipoleGeometricCenter system;
```

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::QuadrupoleGeometricCenter system;
```

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラスである。

9.1.4.1.2 PS::SEARCH_MODE_LONG_CUTOFF

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceLong<TResult, TEpi, TEpj, TMom, TSpj>::WithCutoff system;
```

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラス、ローカルツリー及びグローバルツリーの Moment クラス、SuperParticleJ クラスである。

あらかじめ Moment クラスと SuperParticleJ 型を指定したクラスも用意した (節 7.5, 7.6 参照)。モーメント計算の中心を粒子の重心とした場合に、単極子まで計算するものである。ここでは system というオブジェクトを生成している。

```
PS::TreeForForceLong<TResult, TEpi, TEpj>::MonopoleWithCutoff system;
```

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラスである。

9.1.4.1.3 PS::SEARCH_MODE_GATHER

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceShort<TResult, TEpi, TEpj>::Gather system;
```

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラスである。

9.1.4.1.4 PS::SEARCH_MODE_SCATTER

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceShort<TResult, TEpi, TEpj>::Scatter system;
```

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラスである。

9.1.4.1.5 PS::SEARCH_MODE_SYMMETRY

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceShort<TResult, TEpi, TEpj>::Symmetry system;
```

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラスである。

9.1.4.2 API

このモジュールには初期設定関連の API、相互作用計算関連の低レベル API、相互作用計算関連の高レベル API、ネイバーリスト関連の API がある。以下、各節に分けて記述する。

本節の中の API の宣言ではテンプレートクラスのテンプレート引数は省略する。すなわち、本来ならば以下のように記述するべきであるが、

```
template <class TSearchMode,
          class TResult,
          class TEpi,
          class TEpj,
          class TMomLocal,
          class TMomGlobal,
          class TSpj>
void PS::TreeForForce<TSearchMode,
                      TEpi,
                      TEpj,
                      TMomLocal,
                      TMomGlobal,
                      TSpj>::MemberFunction1();

template <class TSearchMode,
          class TResult,
          class TEpi,
          class TEpj,
          class TMomLocal,
          class TMomGlobal,
          class TSpj>
template <class TTT>
void PS::TreeForForce<TSearchMode,
                      TEpi,
                      TEpj,
                      TMomLocal,
                      TMomGlobal,
                      TSpj>::MemberFunction2(TTT arg1);
```

冗長であるので、以下のように省略する。

```
void PS::TreeForForce::MemberFunction1();

template <class TTT>
void PS::TreeForForce::MemberFunction2(TTT arg1);
```

9.1.4.2.1 初期設定

初期設定関連のAPIの宣言は以下のようになっている。このあと各APIについて記述する。

ソースコード 26: TreeForForce1

```
1  namespace ParticleSimulator {
2      template<class TSearchMode,
3               class TResult,
4               class TEpi,
5               class TEpj,
6               class TMomLocal,
7               class TMomGlobal,
8               class TSpj>
9      class TreeForForce{
10     public:
11     void TreeForForce();
12     void initialize(const U64 n_glb_tot,
13                     const F32 theta=0.7,
14                     const U32 n_leaf_limit=8,
15                     const U32 n_group_limit=64);
16     };
17 }
```

9.1.4.2.1.1 コンストラクタ

コンストラクタ

```
void PS::TreeForForce::TreeForForce();
```

- **引数**

  なし

- **返値**

  なし

- **機能**

  相互作用ツリークラスのオブジェクトを生成する。

9.1.4.2.1.2 PS::TreeForForce::initialize

PS::TreeForForce::initialize

```
void PS::TreeForForce::initialize
            (const PS::U64 n_glb_tot,
             const PS::F32 theta=0.7,
             const PS::U32 n_leaf_limit=8,
             const PS::U32 n_group_limit=64);
```

- **引数**

  n_glb_tot: 入力。const PS::U64 型。粒子配列の上限。

  theta: 入力。const PS::F32 型。見こみ角に対する基準。デフォルト 0.7。

  n_leaf_limit。const PS::U32 型。ツリーを切るのをやめる粒子数の上限。デフォルト 8。

  n_group_limit。const PS::U32 型。相互作用リストを共有する粒子数の上限。デフォルト 64。

- **返値**

  なし

- **機能**

  相互作用ツリークラスのオブジェクトを初期化する。

9.1.4.2.2 低レベル関数

相互作用計算関連の低レベル API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 27: TreeForForce1

```
1  namespace ParticleSimulator {
2      template<class TSearchMode,
3               class TResult,
4               class TEpi,
5               class TEpj,
6               class TMomLocal,
7               class TMomGlobal,
8               class TSpj>
9      class TreeForForce{
10     public:
11         template<class Tpsys>
12         void setParticleLocalTree(const Tpsys & psys,
13                                   const bool clear=true);
```

```
14          template<class Tdinfo>
15          void makeLocalTree(const Tdinfo & dinfo);
16          void makeLocalTree(const F32 l,
17                             const F32vec & c = F32vec(0.0));
18          template<class Tdinfo>
19          void makeGlobalTree(const Tdinfo & dinfo);
20          void calcMomentGlobalTree();
21          template<class Tfunc_ep_ep>
22          void calcForce(Tfunc_ep_ep pfunc_ep_ep(TEpi *,
23                                                 const S32,
24                                                 TEpj *,
25                                                 const S32,
26                                                 TResult *),
27                         const bool clear=true);
28          template<class Tfunc_ep_ep, class Tfunc_sp_ep>
29          void calcForce(Tfunc_ep_ep pfunc_ep_ep(TEpi *,
30                                                 const S32,
31                                                 TEpj *,
32                                                 const S32,
33                                                 TResult *),
34                         Tfunc_sp_ep pfunc_sp_ep(TEpi *,
35                                                 const S32,
36                                                 TSpj *,
37                                                 const S32,
38                                                 TResult *),
39                         const bool clear=true);
40          Tforce getForce(const S32 i);
41      };
42  }
```

9.1.4.2.2.1 PS::TreeForForce::setParticleLocalTree

PS::TreeForForce::setParticleLocalTree

```
template<class Tpsys>
void PS::TreeForForce::setParticleLocalTree
            (const Tpsys & psys,
             const bool clear = true);
```

- **引数**

psys: 入力。const Tpsys & 型。ローカルツリーを構成する粒子群クラスのオブジェクト。

clear: 入力。const bool 型。前に読込んだ粒子をクリアするかどうか決定するフラグ。true でクリアする。デフォルト true。

- **返値**

  なし

- **機能**

  相互作用ツリークラスのオブジェクトに粒子群クラスのオブジェクトの粒子を読み込む。clear が true ならば前に読込んだ粒子情報をクリアし、false ならクリアしない。

9.1.4.2.2.2 PS::TreeForForce::makeLocalTree

PS::TreeForForce::makeLocalTree

```
template<class Tdinfo>
void PS::TreeForForce::makeLocalTree
            (const Tdinfo & dinfo);
```

- **引数**

  dinfo: 入力。const Tdinfo &型。領域クラスのオブジェクト。

- **返値**

  なし

- **機能**

  ローカルツリーを作る。領域クラスのオブジェクトから扱うべきルートドメインを読み取り、ツリーのルートセルを決定する。

```
template<class Tdinfo>
void PS::TreeForForce::makeLocalTree
            (const PS::F32 l,
             const PS::F32vec & c = PS::F32vec(0.0));
```

- **引数**

  l: 入力。const PS::F32 型。ツリーのルートセルの大きさ。

  c: 入力。const PS::F32vec &型。ツリーの中心の座標。デフォルトは座標原点。

- **返値**

  なし

- **機能**

  ローカルツリーを作る。ツリーのルートセルを２つの引数で決定する。ツリーのルートセルは全プロセスで共通でなければならない。共通でない場合の動作の正しさは保証しない。

9.1.4.2.2.3　PS::TreeForForce::makeGlobalTree

PS::TreeForForce::makeGlobalTree

```
template<class Tdinfo>
void PS::TreeForForce::makeGlobalTree
            (const Tdinfo & dinfo);
```

- **引数**

  dinfo: 入力。const Tdinfo & 型。領域クラスのオブジェクト。

- **返値**

  なし

- **機能**

  グローバルツリーを作る。

9.1.4.2.2.4　PS::TreeForForce::calcMomentGlobalTree

PS::TreeForForce::calcMomentGlobalTree

```
void PS::TreeForForce::calcMomentGlobalTree();
```

- **引数**

  なし

- **返値**

  なし

- **機能**

  グローバルツリーの各々のセルのモーメントを計算する。

9.1.4.2.2.5　PS::TreeForForce::calcForce

PS::TreeForForce::calcForce

```
template<class Tfunc_ep_ep>
void PS::TreeForForce::calcForce
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              const bool clear=true);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト (節 7.9 参照)。関数の引数は第 1 引数から順に (const) TEpi *型、const PS::S32 型、(const) TEpj *型、const PS::S32 型、TResult *型。

  clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

- **返値**

  なし

- **機能**

  このオブジェクトに読み込まれた粒子すべての粒子間相互作用を計算する。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合に限る。

```
template<class Tfunc_ep_ep, class Tfunc_sp_ep>
void PS::TreeForForce::calcForce
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tfunc_ep_ep pfunc_sp_ep(TEpi *,
                                      const PS::S32,
                                      TSpj *,
                                      const PS::S32,
                                      TResult *),
              const bool clear=true);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、const PS::S32 型、(const) TEpj *型、const PS::S32 型、TResult *型。

  pfunc_sp_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、const PS::S32 型、(const) TSpj *型、const PS::S32 型、TResult *型。

  clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

- **返値**

  なし

- **機能**

  このオブジェクトに読み込まれた粒子すべての粒子間相互作用を計算する。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF の場合に限る。

9.1.4.2.2.6 PS::TreeForForce::getForce

PS::TreeForForce::getForce

```
TResult PS::TreeForForce::getForce(const PS::S32 i);
```

- **引数**

  i: 入力。const PS::S32 型。粒子配列のインデックス。

- **返値**

  TResult 型。PS::TreeForForce::setParticleLocalTree で i 番目に読み込まれた粒子の受ける作用。

- **機能**

  PS::TreeForForce::setParticleLocalTree で i 番目に読み込まれた粒子の受ける作用を返す。

9.1.4.2.2.7 PS::TreeForForce::copyLocalTreeStructure

PS::TreeForForce::copyLocalTreeStructure

今後、追加する。

9.1.4.2.2.8 PS::TreeForForce::repeatLocalCalcForce

PS::TreeForForce::repeatLocalCalcForce

今後、追加する。

9.1.4.2.3 高レベル関数

相互作用計算関連の高レベル API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 28: TreeForForce1

```
namespace ParticleSimulator {
    template<class TSearchMode,
             class TResult,
             class TEpi,
             class TEpj,
             class TMomLocal,
             class TMomGlobal,
             class TSpj>
    class TreeForForce{
    public:
        template<class Tfunc_ep_ep,
                 class Tpsys,
                 class Tdinfo>
        void calcForceAllAndWriteBack
                    (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                             const S32,
                                             TEpj *,
                                             const S32,
                                             TResult *),
                     Tpsys & psys,
                     Tdinfo & dinfo,
                     const bool clear_force = true);
        template<class Tfunc_ep_ep,
                 class Tfunc_sp_ep,
                 class Tpsys,
                 class Tdinfo>
        void calcForceAllAndWriteBack
                    (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                             const S32,
                                             TEpj *,
```

```
                                        const S32
                                        TResult *),
                Tfunc_sp_ep pfunc_sp_ep(TEpi *,
                                        const S32,
                                        TSpj *,
                                        const S32,
                                        TResult *),
                Tpsys & psys,
                TDinfo & dinfo,
                const bool clear_force=true);

        template<class Tfunc_ep_ep,
                 class Tfunc_sp_ep,
                 class Tpsys,
                 class Tdinfo>
        void calcForceAll
                    (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                        const S32,
                                        TEpj *,
                                        const S32,
                                        TResult *),
                     Tfunc_sp_ep pfunc_sp_ep(TEpi *,
                                        const S32,
                                        TSpj *,
                                        const S32,
                                        TResult *),
                     Tpsys & psys,
                     Tdinfo & dinfo,
                     const bool clear_force=true);
        template<class Tfunc_ep_ep,
                 class Tfunc_sp_ep,
                 class Tpsys,
                 class Tdinfo>
        void calcForceAll(Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                                  const S32,
                                                  TEpj *,
                                                  const S32
                                                  TResult *),
                          Tfunc_sp_ep pfunc_sp_ep(TEpi *,
                                                  const S32,
```

```
71                                                  TSpj *,
72                                                  const S32,
73                                                  TResult *),
74                      Tpsys & psys,
75                      Tdinfo & dinfo,
76                      const bool clear_force=true);
77
78      template<class Tfunc_ep_ep,
79               class Tdinfo>
80      void calcForceMakeingTree
81                  (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
82                                           const S32,
83                                           TEpj *,
84                                           const S32,
85                                           TResult *),
86                   Tdinfo & dinfo,
87                   const bool clear_force=true);
88      template<class Tfunc_ep_ep,
89               class Tfunc_sp_ep,
90               class Tdinfo>
91      void calcForceMakingTree
92                  (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
93                                           const S32,
94                                           TEpj *,
95                                           const S32,
96                                           TResult *),
97                   Tfunc_sp_ep pfunc_sp_ep(TEpi *,
98                                           const S32,
99                                           TSpj *,
100                                          const S32,
101                                          TResult *),
102                  Tdinfo & dinfo,
103                  const bool clear_force=true);
104
105     template<class Tfunc_ep_ep,
106              class Tpsys>
107     void calcForceAndWriteBack
108                 (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
109                                          const S32,
110                                          TEpj *,
```

```
111                                                 const S32,
112                                                 TResult *),
113                     Tpsys & psys,
114                     const bool clear=true);
115         template<class Tfunc_ep_ep,
116                  class Tfunc_sp_ep,
117                  class Tpsys>
118         void calcForceAndWriteBack
119                     (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
120                                              const S32,
121                                              TEpj *,
122                                              const S32
123                                              TResult *),
124                      Tfunc_sp_ep pfunc_sp_ep(TEpi *,
125                                              const S32,
126                                              TSpj *,
127                                              const S32,
128                                              TResult *),
129                     Tpsys & psys,
130                     const bool clear=true);
131     };
132 }
133 namespace PS = ParticleSimulator;
```

9.1.4.2.3.1 PS::TreeForForce::calcForceAllAndWriteBack

PS::TreeForForce::calcForceAllAndWriteBack

```
template<class Tfunc_ep_ep,
         class Tpsys,
         class Tdinfo>
void PS::TreeForForce::calcForceAllandWriteBack
            (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                     const PS::S32,
                                     TEpj *,
                                     const PS::S32,
                                     TResult *),
             Tpsys & psys,
             Tdinfo & dinfo
             const bool clear=true);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

  psys: 入力。Tpsys &型。相互作用を計算したい粒子群クラスのオブジェクト。

  dinfo: 入力。Tdinfo &型。領域クラスのオブジェクト。

  clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys の粒子すべての相互作用を計算し、その計算結果を psys に書き戻す。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合に限る。

```
template<class Tfunc_ep_ep,
         class Tfunc_sp_ep,
         class Tpsys,
         class Tdinfo>
void PS::TreeForForce::calcForceAllandWriteBack
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tfunc_sp_ep pfunc_sp_ep(TEpi *,
                                      const PS::S32,
                                      TSpj *,
                                      const PS::S32,
                                      TResult *),
              Tpsys & psys,
              Tdinfo & dinfo
              const bool clear=true);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

pfunc_sp_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TSpj *型、PS::S32 型、TResult *型。

psys: 入力。Tpsys &型。相互作用を計算したい粒子群クラスのオブジェクト。

dinfo: 入力。Tdinfo &型。領域クラスのオブジェクト。

clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys の粒子すべての相互作用を計算し、その計算結果を psys に書き戻す。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF の場合に限る。

9.1.4.2.3.2 PS::TreeForForce::calcForceAll

PS::TreeForForce::calcForceAll

```
template<class Tfunc_ep_ep,
         class Tpsys,
         class Tdinfo>
void PS::TreeForForce::calcForceAll
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tpsys & psys,
              Tdinfo & dinfo
              const bool clear=true);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

  psys: 入力。Tpsys &型。相互作用を計算したい粒子群クラスのオブジェクト。

  dinfo: 入力。Tdinfo &型。領域クラスのオブジェクト。

clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys の粒子すべての相互作用を計算する。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合に限る。PS::TreeForForce::calcForceAllAndWriteBack から計算結果の書き戻しがなくなったもの。

```
template<class Tfunc_ep_ep,
         class Tfunc_sp_ep,
         class Tpsys,
         class Tdinfo>
void PS::TreeForForce::calcForceAll
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tfunc_sp_ep pfunc_sp_ep(TEpi *,
                                      const PS::S32,
                                      TSpj *,
                                      const PS::S32,
                                      TResult *),
              Tpsys & psys,
              Tdinfo & dinfo
              const bool clear=true);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

  pfunc_sp_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TSpj *型、PS::S32 型、TResult *型。

  psys: 入力。Tpsys &型。相互作用を計算したい粒子群クラスのオブジェクト。

dinfo: 入力。Tdinfo &型。領域クラスのオブジェクト。

clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys の粒子すべての相互作用を計算する。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF の場合に限る。PS::TreeForForce::calcForceAllAndWriteBack から計算結果の書き戻しがなくなったもの。

9.1.4.2.3.3 PS::TreeForForce::calcForceMakingTree

PS::TreeForForce::calcForceMakingTree

```
template<class Tfunc_ep_ep,
         class Tdinfo>
void PS::TreeForForce::calcForceMakingTree
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tdinfo & dinfo
              const bool clear=true);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

  dinfo: 入力。Tdinfo &型。領域クラスのオブジェクト。

  clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

- **返値**

  なし

- **機能**

  これより前に相互作用ツリークラスのオブジェクトに読み込まれた粒子群クラスのオブジェクトの粒子すべての相互作用を計算する。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合に限る。PS::TreeForForce::calcForceAllAndWriteBack から粒子群クラスのオブジェクトの読込と計算結果の書き戻しがなくなったもの。

```
template<class Tfunc_ep_ep,
         class Tfunc_sp_ep,
         class Tdinfo>
void PS::TreeForForce::calcForceMakingTree
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tfunc_sp_ep pfunc_sp_ep(TEpi *,
                                      const PS::S32,
                                      TSpj *,
                                      const PS::S32,
                                      TResult *),
              Tdinfo & dinfo
              const bool clear=true);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

  pfunc_sp_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TSpj *型、PS::S32 型、TResult *型。

  dinfo: 入力。Tdinfo &型。領域クラスのオブジェクト。

  clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

- **返値**

  なし

- **機能**

これより前に相互作用ツリークラスのオブジェクトに読み込まれた粒子群クラスのオブジェクトの粒子すべての相互作用を計算する。これを使うのはPS::SEARCH_MODE型がPS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF の場合に限る。PS::TreeForForce::calcForceAllAndWriteBackから粒子群クラスのオブジェクトの読込と計算結果の書き戻しがなくなったもの。

9.1.4.2.3.4 PS::TreeForForce::calcForceAndWriteBack

PS::TreeForForce::calcForceAndWriteBack

```
template<class Tfunc_ep_ep,
         class Tpsys>
void PS::TreeForForce::calcForceAndWriteBack
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tpsys & psys,
              const bool clear=true);
```

- **引数**

  pfunc_ep_ep: 入力。返値がvoid型のEssentialParticleIとEssentialParticleJの間の相互作用計算用の関数オブジェクト。関数の引数は第1引数から順に(const) TEpi *型、PS::S32型、const TEpj *型、PS::S32型、TResult *型。

  psys: 入力。Tpsys &型。相互作用の計算結果を書き戻したい粒子群クラスのオブジェクト。

  clear: 入力。const bool型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。trueならばクリアする。デフォルトtrue。

- **返値**

  なし

- **機能**

  これより前に相互作用ツリークラスのオブジェクトに構築されたグローバルツリーとそのモーメントをもとに、相互作用ツリークラスのオブジェクトに属する粒子すべての相互作用が計算され、さらにその結果が粒子群クラスのオブジェクトpsysに書き戻される。これを使うのはPS::SEARCH_MODE型がPS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRYの場合に限る。PS::TreeForForce::calcForceAllAndWriteBackから粒子群クラスのオブジェクトの読込、

ローカルツリーの構築、グローバルツリーの構築、グローバルツリーのモーメントの計算がなくなったもの。

```
template<class Tfunc_ep_ep,
         class Tfunc_sp_ep,
         class Tpsys>
void PS::TreeForForce::calcForceAllandWriteBack
             (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                                      const PS::S32,
                                      TEpj *,
                                      const PS::S32,
                                      TResult *),
              Tfunc_sp_ep pfunc_sp_ep(TEpi *,
                                      const PS::S32,
                                      TSpj *,
                                      const PS::S32,
                                      TResult *),
              Tpsys & psys,
              const bool clear=true);
```

- **引数**

  pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TEpj *型、PS::S32 型、TResult *型。

  pfunc_sp_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、PS::S32 型、const TSpj *型、PS::S32 型、TResult *型。

  psys: 入力。Tpsys &型。相互作用の計算結果を書き戻したい粒子群クラスのオブジェクト。

  clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

- **返値**

  なし

- **機能**

  これより前に相互作用ツリークラスのオブジェクトに構築されたグローバルツリーとそのモーメントをもとに、相互作用ツリークラスのオブジェクトに属する粒子すべての相互作用が計算され、さらにその結果が粒子群クラスのオブジェクト psys に書

き戻される。これを使うのはPS::SEARCH_MODE型がPS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFFの場合に限る。
PS::TreeForForce::calcForceAllAndWriteBackから粒子群クラスのオブジェクトの読込、ローカルツリーの構築、グローバルツリーの構築、グローバルツリーのモーメントの計算がなくなったもの。

9.1.4.2.4 ネイバーリスト

今後、追加する。

9.1.5 通信用データクラス

本節では、通信用データクラスについて記述する。このクラスはノード間通信のための情報の保持や実際の通信を行うモジュールである。このクラスはシングルトンパターンとして管理されており、オブジェクトの生成は必要としない。ここではこのモジュールのAPIを記述する。

9.1.5.1 API

このモジュールのAPIの宣言は以下のようになっている。このあと各APIについて記述する。

ソースコード 29: Communication

```
1  namespace ParticleSimulator {
2      class Comm{
3      public:
4          static S32 getRank();
5          static S32 getNumberOfProc();
6          static S32 getRankMultiDim(const S32 id);
7          static S32 getNumberOfProcMultiDim(const S32 id);
8          static bool synchronizeConditionalBranchAND
9                      (const bool local);
10         static bool synchronizeConditionalBranchOR
11                     (const bool local);
12         template<class T>
13         static T getMinValue(const T val);
14         template<class Tfloat, class Tint>
15         static void getMinValue(const Tfloat f_in,
16                                 const Tint i_in,
17                                 Tfloat & f_out,
18                                 Tint & i_out);
```

```
19          template<class T>
20          static T getMaxValue(const T val);
21          template<class Tfloat, class Tint>
22          static void getMaxValue(const Tfloat f_in,
23                                  const Tint i_in,
24                                  Tfloat & f_out,
25                                  Tint & i_out );
26          template<class T>
27          static T getSum(const T val);
28      };
29  }
```

9.1.5.1.1 PS::Comm::getRank

```
static PS::S32 PS::Comm::getRank();
```

- **引数**

  なし。

- **返り値**

  PS::S32 型。全プロセス中でのランクを返す。

9.1.5.1.2 PS::Comm::getNumberOfProc

```
static PS::S32 PS::Comm::getNumberOfProc();
```

- **引数**

  なし。

- **返り値**

  PS::S32 型。全プロセス数を返す。

9.1.5.1.3 PS::Comm::getRankMultiDim

```
static PS::S32 PS::Comm::getRankMultiDim(const PS::S32 id);
```

- **引数**

  id: 入力。const PS::S32 型。軸の番号。x 軸:0, y 軸:1, z 軸:2。

- **返り値**

  PS::S32 型。id 番目の軸でのランクを返す。2 次元の場合、id=2 は 1 を返す。

9.1.5.1.4　PS::Comm::getNumberOfProcMultiDim

```
static PS::S32 PS::Comm::getNumberOfProcMultiDim(const PS::S32 id);
```

- **引数**

  id: 入力。const PS::S32 型。軸の番号。x 軸:0, y 軸:1, z 軸:2。

- **返り値**

  PS::S32 型。id 番目の軸のプロセス数を返す。2 次元の場合、id=2 は 1 を返す。

9.1.5.1.5　PS::Comm::synchronizeConditionalBranchAND

```
static bool PS::Comm::synchronizeConditionalBranchAND(const bool local)
```

- **引数**

  local: 入力。const bool 型。

- **返り値**

  bool 型。全プロセスで local の論理積を取り、結果を返す。

9.1.5.1.6　PS::Comm::synchronizeConditionalBranchOR

```
static bool PS::Comm::synchronizeConditionalBranchOR(const bool local);
```

- **引数**

  local: 入力。const bool 型。

- **返り値**

  bool 型。全プロセスで local の論理和を取り、結果を返す。

9.1.5.1.7　PS::Comm::getMinValue

```
template <class T>
static T PS::Comm::getMinValue(const T val);
```

- **引数**

  val: 入力。const T 型。

- **返り値**

  T 型。全プロセスで val の最小値を取り、結果を返す。

```
template <class Tfloat, class Tint>
static void PS::Comm::getMinValue(const Tfloat f_in,
                                  const Tint i_in,
                                  Tfloat & f_out,
                                  Tint & i_out);
```

- **引数**

  f_in: 入力。const Tfloat 型。

  i_in: 入力。const Tint 型。

  f_out: 出力。Tfloat 型。全プロセスで f_in の最小値を取り、結果を返す。

  i_out: 出力。Tint 型。f_out に伴う ID を返す。

- **返り値**

  なし。

9.1.5.1.8　PS::Comm::getMaxValue

```
template <class T>
static T PS::Comm::getMaxValue(const T val);
```

- **引数**

  val: 入力。const T 型。

- **返り値**

  T 型。全プロセスで val の最大値を取り、結果を返す。

```
template <class Tfloat, class Tint>
static void PS::Comm::getMaxValue(const Tfloat f_in,
                                  const Tint i_in,
                                  Tfloat & f_out,
                                  Tint & i_out);
```

- **引数**

  f_in: 入力。const Tfloat 型。

  i_in: 入力。const Tint 型。

  f_out: 出力。Tfloat 型。全プロセスで f_in の最大値を取り、結果を返す。

  i_out: 出力。Tint 型。f_out に伴う ID を返す。

- **返り値**

  なし。

9.1.5.1.9　PS::Comm::getSum

```
template <class T>
static T PS::Comm::getSum(const T val);
```

- **引数**

  val: 入力。const T 型。

- **返り値**

  T 型。全プロセスで val の総和を取り、結果を返す。

9.2　拡張機能

9.2.1　概要

本節では、FDPS の拡張機能について記述する。拡張機能には 1 つのモジュールがあり、Particle Mesh クラスがある。この 1 つのクラスについて記述する。

9.2.2　Particle Mesh クラス

本節では、Particle Mesh クラスについて記述する。このクラスは Particle Mesh 法を用いて粒子の相互作用を計算するモジュールである。オブジェクトの生成方法、API、使用済マクロ、使いかたについて記述する。

9.2.2.1 オブジェクトの生成

Particle Mesh クラスは以下のように宣言されている。

ソースコード 30: ParticleMesh0

```
1 namespace ParticleSimulator {
2     namespace ParticleMesh {
3         class ParticleMesh;
4     }
5 }
```

Particle Mesh クラスのオブジェクトの生成は以下のように行う。ここでは pm というオブジェクトを生成している。

```
PS::PM::ParticleMesh pm;
```

9.2.2.2 API

Particle Mesh クラスには初期設定関連の API、低レベル API、高レベル API がある。以下、各節に分けて記述する。

9.2.2.2.1 初期設定

初期設定関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 31: ParticleMesh1

```
1 namespace ParticleSimulator {
2     namespace ParticleMesh {
3         class ParticleMesh{
4             ParticleMesh();
5         };
6     }
7 }
```

9.2.2.2.1.1 コンストラクタ

コンストラクタ

```
void PS::PM::ParticleMesh::ParticleMesh();
```

- **引数**

  なし

- **返値**

  なし

- **機能**

  Particle Mesh クラスのオブジェクトを生成する。

9.2.2.2.2 低レベル *API*

低レベル API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 32: ParticleMesh1

```
1  namespace ParticleSimulator {
2      namespace ParticleMesh {
3          class ParticleMesh{
4              template<class Tdinfo>
5              void setDomainInfoParticleMesh
6                          (const Tdinfo & dinfo);
7              template<class Tpsys>
8              void setParticleParticleMesh
9                          (const Tpsys & psys,
10                          const bool clear=true);
11             void calcMeshForceOnly();
12             F32vec getForce(F32vec pos);
13         };
14     }
15 }
```

9.2.2.2.2.1 PS::PM::ParticleMesh::setDomainInfoParticleMesh

PS::PM::ParticleMesh::setDomainInfoParticleMesh

```
template<class Tdinfo>
void PS::PM::ParticleMesh::setDomainInfoParticleMesh
            (const Tdinfo & dinfo);
```

- **引数**

  dinfo: 入力。Tdinfo &型。領域クラスのオブジェクト。

- **返値**

  なし

- **機能**

  領域情報を読み込む。

9.2.2.2.2.2　PS::PM::ParticleMesh::setParticleParticleMesh

PS::PM::ParticleMesh::setParticleParticleMesh

```
template<class Tpsys>
void PS::PM::ParticleMesh::setParticleParticleMesh
            (const Tpsys & psys,
             const bool clear=true);
```

- **引数**

  psys: 入力。Tpsys & 型。粒子群クラスのオブジェクト。

  clear: 入力。const bool 型。これまで読込んだ粒子情報をクリアするかどうか決定するフラグ。true ならばクリアする。デフォルトは true。

- **返値**

  なし

- **機能**

  粒子情報を粒子群クラスのオブジェクトから読み込む。

9.2.2.2.2.3　PS::PM::ParticleMesh::calcMeshForceOnly

PS::PM::ParticleMesh::calcMeshForceOnly

```
void PS::PM::ParticleMesh::calcMeshForceOnly();
```

- **引数**

  なし

- **返値**

  なし

- **機能**

  メッシュ上の力を計算する。

9.2.2.2.2.4　PS::PM::ParticleMesh::getForce

PS::PM::ParticleMesh::getForce

```
PS::F32vec PS::PM::ParticleMesh::getForce
            (PS::F32vec pos);
```

- **引数**

  pos: 入力。PS::F32vec 型。メッシュに課された粒子からの力を計算したい位置。

- **返値**

  PS::F32vec 型。メッシュに課された粒子からの力。

- **機能**

  位置 pos でのメッシュに課された粒子からの力を返す。

9.2.2.2.3　高レベル API

高レベル API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 33: ParticleMesh1

```
1  namespace ParticleSimulator {
2      namespace ParticleMesh {
3          class ParticleMesh{
4              template<class Tpsys,
5                       class Tdinfo>
6              void calcForceAllAndWriteBack
7                           (Tpsys & psys,
8                            const Tdinfo & dinfo);
9          };
10     }
11 }
```

9.2.2.2.3.1　PS::PM::ParticleMesh::calcForceAllAndWriteBack

PS::PM::ParticleMesh::calcForceAllAndWriteBack

```
template<class Tpsys,
         class Tdinfo>
void PS::PM::ParticleMesh::calcForceAllAndWriteBack
            (Tpsys & psys,
             const Tdinfo & dinfo);
```

- **引数**

  psys: 入力であり出力。Tpsys & 型。粒子群クラスのオブジェクト。

  dinfo: 入力。const Tdinfo &型。領域クラスのオブジェクト。

- **返値**

  なし

- **機能**

  粒子群クラスのオブジェクト psys に含まれる粒子間のメッシュ力を計算し、その結果を psys に返す。

9.2.2.3 使用済マクロ

このモジュールでは多くのマクロを使っている。これらを別のマクロとして使用した場合にプログラムが正しく動作する保証はない。ここでは使用されているマクロをアルファベティカルに列挙する。

- BINARY_BOUNDARY
- BOUNDARY_COMM_NONBLOCKING
- BOUNDARY_SMOOTHING
- BUFFER_FOR_TREE
- CALCPOT
- CLEAN_BOUNDARY_PARTICLE
- CONSTANT_TIMESTEP
- EXCHANGE_COMM_NONBLOCKING
- FFT3D
- FFTW3_PARALLEL

- FFTW_DOUBLE
- FIX_FFTNODE
- GADGET_IO
- GRAPE_OFF
- KCOMPUTER
- LONG_ID
- MAKE_LIST_PROF
- MERGE_SNAPSHOT
- MULTI_TIMESTEP
- MY_MPI_BARRIER
- N128_2H
- N256_2H
- N256_H
- N32_2H
- N512_2H
- NEW_DECOMPOSITION
- NOACC
- NPART_DIFFERENT_DUMP
- OMP_SCHDULE_DISABLE
- PRINT_TANIKAWA
- REVERSE_ENDIAN_INPUT
- REVERSE_ENDIAN_OUTPUT
- RMM_PM
- SHIFT_INITIAL_BOUNDARY
- STATIC_ARRAY
- TREE2

- TREECONSTRUCTION_PARALLEL
- TREE_PARTICLE_CACHE
- UNIFORM
- UNSTABLE
- USING_MPI_PARTICLE
- VERBOSE_MODE
- VERBOSE_MODE2。

9.2.2.4 Particle Mesh クラスの使いかた

Particle Mesh クラスを使うには以下の４つのことを行う必要がある。

1. Particle Mesh クラスのコンパイル
2. Particle Mesh クラスを使った FDPS コードの記述
3. FDPS コードのコンパイル

以下、詳細に記述する。

9.2.2.4.1 Particle Mesh クラスのコンパイル

以下のように行う。ディレクトリ src の下のディレクトリ particle_mesh の Makefile を適切に編集して make する。編集すべきことは以下の２点である。

- INCLUDE_FFTW に FFTW のヘッダファイルがあるディレクトリを記述する
- param_fdps.h の中の SIZE_OF_MESH (１次元方向のメッシュの数) を設定。推奨値は $N^{1/3}/2$(N は粒子数)。

うまく行けば、同じディレクトリにライブラリ libpm.a とヘッダファイル particle_mesh.hpp ができる。

9.2.2.4.2 FDPS コードを記述

以下のように行う。

- 上でできたヘッダファイルを include する
- PM を計算したい粒子クラスに以下のメンバ関数を加える (この粒子クラスのクラス名を FP とする)

- void FP::copyFromForceParticleMesh(const PS::F32vec & force)。この中で force を好きなメンバ変数にセットする。
- PS::F64 FP::getChargeParticleMesh()。この中で質量を返す。

- このクラスのオブジェクトを生成するときに、PS::PM::ParticleMesh とする

9.2.2.4.3 FDPS コードのコンパイル

上で記述した FDPS コードをコンパイルするには以下のことを行う必要がある。

- ヘッダファイル particle_mesh.hpp のあるディレクトリへのパスを指定する
- ライブラリ libpm.a とリンクする
- FFTW のヘッダファイルがあるディレクトリへのパスを指定する
- FFTW のライブラリとリンクする

10　エラーメッセージ

10.1　概要

FDPS はいくつかのエラーメッセージを用意している。１つはコンパイル時のエラーメッセージであり、もう１つは実行時のエラーメッセージである。以下、この順に記述する。

10.2　コンパイル時のエラーメッセージ

10.3　実行時のエラーメッセージ

標準エラー出力に以下のような書式でメッセージが出力される。

PS_ERROR: *ERROR MESSAGE*
function: *FUNCTION NAME*, line: *LINE NUMBER*, file: *FILE NAME*

- *ERROR MESSAGE*
  エラーメッセージ
- *FUNCTION NAME*
  エラーが起こった関数の名前
- *LINE NUMBER*
  エラーが起こった行番号
- *FILE NAME*
  エラーが起こったファイルの名前

11　よく知られているバグ

12　限界

13　ユーザーサポート

FDPSを使用したコード開発に関する相談は以下のメールアドレスfdps-support@mail.jmlab.jpで受け付けています。以下のような場合は各項目毎の対応をお願いします。

13.1　コンパイルできない場合

ユーザーには以下の情報提供をお願いします。

- コンパイル環境
- コンパイル時に出力されるエラーメッセージ
- ソースコード (可能ならば)

13.2　コードがうまく動かない場合

ユーザーには以下の情報提供をお願いします。

- 実行環境
- 実行時に出力されるエラーメッセージ
- ソースコード (可能ならば)

14　ライセンス

MIT ライセンスに準ずる。標準機能のみ使用する場合は、Iwasawa et al. (2015 in prep), Tanikawa et al. (2016 in prep) の引用を義務とする。拡張機能のうち Particle Mesh クラスを使用する場合は、上記に加え、Ishiyama, Fukushige & Makino (2009, Publications of the Astronomical Society of Japan, 61, 1319), Ishiyama, Nitadori & Makino (2012 SC'12 Proceedings of the International Conference on High Performance Computing, Networking Stroage and Analysis, No. 5) の引用を義務とする。