

FDPS仕様書

FDPS 開発者

目 次

1	概要	9
2	ファイル構成	9
2.1	概要	9
2.2	標準機能関係ファイル	9
2.3	拡張機能関係ファイル	9
2.4	テストコード	9
2.5	サンプルコード	9
3	コンパイル時のマクロによる選択	10
3.1	概要	10
3.2	空間次元数	10
3.2.1	概要	10
3.2.2	直交座標系 2 次元	10
3.2.3	直交座標系 3 次元	10
3.3	並列処理	10
3.3.1	概要	10
3.3.2	OpenMP の使用	10
3.3.3	MPI の使用	10
4	名前空間	11
4.1	概要	11
4.2	ParticleSimulator	11
4.2.1	ParticleMesh	11
5	データ型	12
5.1	概要	12
5.2	整数型	12
5.2.1	概要	12
5.2.2	PS::S32	12
5.2.3	PS::S64	12

5.2.4	PS::U32	13
5.2.5	PS::U64	13
5.3	実数型	13
5.3.1	概要	13
5.3.2	PS::F32	13
5.3.3	PS::F64	14
5.4	ベクトル型	14
5.4.1	概要	14
5.4.2	PS::Vector2	14
5.4.2.1	コンストラクタ	15
5.4.2.2	代入演算子	16
5.4.2.3	加減算	17
5.4.2.4	ベクトルスカラ積	18
5.4.2.5	内積、外積	19
5.4.2.6	Vector2<U>への型変換	19
5.4.3	PS::Vector3	20
5.4.3.1	コンストラクタ	21
5.4.3.2	代入演算子	22
5.4.3.3	加減算	22
5.4.3.4	ベクトルスカラ積	23
5.4.3.5	内積、外積	25
5.4.3.6	Vector3<U>への型変換	25
5.4.4	ベクトル型のラッパー	26
5.5	対称行列型	26
5.5.1	概要	26
5.5.2	PS::MatrixSym2	26
5.5.2.1	コンストラクタ	27
5.5.2.2	代入演算子	29
5.5.2.3	加減算	29
5.5.2.4	トレースの計算	30
5.5.2.5	MatrixSym2<U>への型変換	31
5.5.3	PS::MatrixSym3	31
5.5.3.1	コンストラクタ	32
5.5.3.2	代入演算子	34
5.5.3.3	加減算	34
5.5.3.4	トレースの計算	35
5.5.3.5	MatrixSym3<U>への型変換	36
5.5.4	行列型のラッパー	36
5.6	SEARCH_MODE 型	37
5.6.1	概要	37

5.6.2	SEARCH_MODE_LONG	37
5.6.3	SEARCH_MODE_LONG_CUTOFF	37
5.6.4	SEARCH_MODE_GATHER	37
5.6.5	SEARCH_MODE_SCATTER	37
5.6.6	SEARCH_MODE_SYMMETRY	37
5.7	列挙型	37
5.7.1	概要	37
5.7.2	BOUNDARY_CONDITION 型	38
5.7.2.1	概要	38
5.7.2.2	PS::BOUNDARY_CONDITION_OPEN	38
5.7.2.3	PS::BOUNDARY_CONDITION_PERIODIC_X	38
5.7.2.4	PS::BOUNDARY_CONDITION_PERIODIC_Y	38
5.7.2.5	PS::BOUNDARY_CONDITION_PERIODIC_Z	38
5.7.2.6	PS::BOUNDARY_CONDITION_PERIODIC_XY	39
5.7.2.7	PS::BOUNDARY_CONDITION_PERIODIC_XZ	39
5.7.2.8	PS::BOUNDARY_CONDITION_PERIODIC_YZ	39
5.7.2.9	PS::BOUNDARY_CONDITION_PERIODIC_XYZ	39
5.7.2.10	PS::BOUNDARY_CONDITION_SHEARING_BOX	39
5.7.2.11	PS::BOUNDARY_CONDITION_USER_DEFINED	39
6	ユーザー定義クラス、ファンクタ	40
6.1	概要	40
6.2	FullParticle クラス	40
6.2.1	概要	40
6.2.2	前提	40
6.2.3	必要なメンバ関数	40
6.2.3.1	概要	40
6.2.3.2	getPos	41
6.2.3.3	copyFromForce	42
6.2.4	場合によっては必要なメンバ関数	42
6.2.4.1	概要	42
6.2.4.2	相互作用ツリークラスの SEARCH_MODE 型に SEARCH_MODE_LONG 以外を用いる場合	43
6.2.4.2.1	getRsearch	43
6.2.4.3	粒子群クラスのファイル入出力 API を用いる場合	43
6.2.4.3.1	readAscii, readBinary	44
6.2.4.3.2	writeAscii, writeBinary	45
6.2.4.4	粒子群クラスの adjustPositionIntoRootDomain を用いる場合	46
6.2.4.4.1	setPos	46
6.2.4.5	Particle Mesh クラスを用いる場合	46
6.2.4.5.1	getChargeParticleMesh	47

6.2.4.5.2	copyFromForceParticleMesh	47
6.3	EssentialParticleI クラス	48
6.3.1	概要	48
6.3.2	前提	48
6.3.3	必要なメンバ関数	49
6.3.3.1	概要	49
6.3.3.2	getPos	49
6.3.3.3	copyFromFP	50
6.3.4	場合によっては必要なメンバ関数	51
6.3.4.1	概要	51
6.3.4.2	相互作用ツリークラスのSEARCH_MODE型にSEARCH_MODE_GATHER またはSEARCH_MODE_SYMMETRYを用いる場合	51
6.3.4.2.1	getRsearch	51
6.4	EssentialParticleJ クラス	52
6.4.1	概要	52
6.4.2	前提	52
6.4.3	必要なメンバ関数	52
6.4.3.1	概要	52
6.4.3.2	getPos	52
6.4.3.3	copyFromFP	53
6.4.4	場合によっては必要なメンバ関数	54
6.4.4.1	概要	54
6.4.4.2	相互作用ツリークラスのSEARCH_MODE型にSEARCH_MODE_LONG 以外を用いる場合	55
6.4.4.2.1	getRsearch	55
6.4.4.3	BOUNDARY_CONDITION型にPS::BOUNDARY_CONDITION_OPEN 以外を用いる場合	56
6.4.4.3.1	setPos	56
6.5	SuperParticleJ クラス	57
6.5.1	概要	57
6.5.2	既存のクラス	57
6.5.3	必要なメンバ関数	57
6.5.4	場合によっては必要なメンバ関数	57
6.5.4.1	getPos	57
6.5.4.2	setPos	57
6.5.4.3	copyFromMoment	57
6.5.4.4	convertToMoment	57
6.5.4.5	clear	57
6.6	Moment クラス	57
6.6.1	概要	57

6.6.2	既存のクラス	57
6.6.2.1	概要	57
6.6.2.2	SEARCH_MODE_LONG	58
6.6.2.2.1	MomentMonopole	58
6.6.2.2.2	MomentQuadrupole	58
6.6.2.2.3	MomentMonopoleGeometricCenter	59
6.6.2.2.4	MomentDipoleGeometricCenter	59
6.6.2.2.5	MomentQuadrupoleGeometricCenter	60
6.6.2.3	SEARCH_MODE_LONG_CUTOFF	61
6.6.2.3.1	MomentMonopoleCutoff	61
6.6.3	必要なメンバ関数	61
6.6.3.1	概要	61
6.6.3.2	init	61
6.6.3.3	getPos	62
6.6.3.4	getCharge	63
6.6.3.5	accumulateAtLeaf	63
6.6.3.6	accumulate	64
6.6.3.7	set	65
6.6.3.8	accumulateAtLeaf2	66
6.6.3.9	accumulate2	66
6.7	Force クラス	66
6.7.1	概要	66
6.7.2	前提	66
6.7.3	必要なメンバ関数	66
6.7.3.1	clear	66
6.8	ヘッダクラス	67
6.8.1	概要	67
6.8.2	必要なメンバ関数	67
6.8.2.1	概要	67
6.8.3	場合によっては必要なメンバ関数	67
6.8.3.1	概要	67
6.8.3.2	readParticleAscii	67
6.8.3.3	readParticleBinary	67
6.8.3.4	writeParticleAscii	67
6.8.3.5	writeParticleBinary	67
6.9	calcForceEpEp ファンクタ	67
6.10	calcForceSpEp ファンクタ	67
7	プログラムの開始と終了	68
7.1	概要	68
7.2	API	68

7.2.1	Initialize	68
7.2.2	Finalize	68
8	モジュール	70
8.1	標準機能	70
8.1.1	概要	70
8.1.2	領域クラス	70
8.1.2.1	オブジェクトの生成	70
8.1.2.2	API	70
8.1.2.2.1	初期設定	71
8.1.2.2.1.1	コンストラクタ	71
8.1.2.2.1.2	initialize	71
8.1.2.2.1.3	setNumberOfDomainMultiDimension	72
8.1.2.2.1.4	setBoundaryCondition	73
8.1.2.2.1.5	setPosRootDomain	73
8.1.2.2.2	領域分割	74
8.1.2.2.2.1	collectSampleParticle	74
8.1.2.2.2.2	decomposeDomain	75
8.1.2.2.2.3	decomposeDomainAll	75
8.1.3	粒子群クラス	76
8.1.3.1	オブジェクトの生成	76
8.1.3.2	API	76
8.1.3.2.1	初期設定	76
8.1.3.2.1.1	コンストラクタ	77
8.1.3.2.1.2	initialize	77
8.1.3.2.1.3	setAverateTargetNumberOfSampleParticlePerProcess	78
8.1.3.2.2	オブジェクト情報の取得設定	78
8.1.3.2.2.1	operator []	79
8.1.3.2.2.2	setNumberOfParticleLocal	79
8.1.3.2.2.3	getNumberOfParticleLocal	79
8.1.3.2.2.4	getNumberOfParticleGlobal	80
8.1.3.2.3	ファイル入出力	80
8.1.3.2.3.1	readParticleAscii	81
8.1.3.2.3.2	readParticleBinary	83
8.1.3.2.3.3	writeParticlAscii	83
8.1.3.2.3.4	writeParticleBinary	84
8.1.3.2.4	粒子交換	84
8.1.3.2.4.1	exchangeParticle	85
8.1.3.2.5	その他	85
8.1.3.2.5.1	adjustPositionIntoRootDomain	85
8.1.4	相互作用ツリークラス	85

8.1.4.1	オブジェクトの生成	85
8.1.4.1.1	SEARCH_MODE_LONG	86
8.1.4.1.2	SEARCH_MODE_LONG_CUTOFF	87
8.1.4.1.3	SEARCH_MODE_GATHER	87
8.1.4.1.4	SEARCH_MODE_SCATTER	87
8.1.4.1.5	SEARCH_MODE_SYMMETRY	87
8.1.4.2	API	88
8.1.4.2.1	初期設定	88
8.1.4.2.1.1	コンストラクタ	88
8.1.4.2.1.2	initialize	89
8.1.4.2.2	低レベル関数	89
8.1.4.2.2.1	setParticleLocalTree	90
8.1.4.2.2.2	makeLocalTree	91
8.1.4.2.2.3	makeGlobalTree	92
8.1.4.2.2.4	calcMomentGlobalTree(仮)	92
8.1.4.2.2.5	calcForce	93
8.1.4.2.2.6	getForce	94
8.1.4.2.2.7	copyLocalTreeStructure	94
8.1.4.2.2.8	repeatLocalCalcForce	94
8.1.4.2.3	高レベル関数	94
8.1.4.2.3.1	calcForceAllAndWriteBack	97
8.1.4.2.3.2	calcForceAll	98
8.1.4.2.3.3	calcForceMakingTree	100
8.1.4.2.3.4	calcForceAndWriteBack	102
8.1.4.2.4	ネイバーリスト	103
8.1.5	通信用データクラス	103
8.1.5.1	API	103
8.1.5.1.1	getRank	104
8.1.5.1.2	getNumberOfProc	105
8.1.5.1.3	getRankMultiDim	105
8.1.5.1.4	getNumberOfProcMultiDim	105
8.1.5.1.5	synchronizeConditionalBranchAND	105
8.1.5.1.6	synchronizeConditionalBranchOR	106
8.1.5.1.7	getMinValue	106
8.1.5.1.8	getMaxValue	107
8.1.5.1.9	getSum	107
8.2	拡張機能	108
8.2.1	概要	108
8.2.2	Particle Mesh クラス	108
8.2.2.1	オブジェクトの生成	108

8.2.2.2	API	108
8.2.2.2.1	初期設定	108
8.2.2.2.1.1	コンストラクタ	109
8.2.2.2.2	低レベル API	109
8.2.2.2.2.1	setDomainInfoParticleMesh	110
8.2.2.2.2.2	setParticleParticleMesh	110
8.2.2.2.2.3	calcMeshForceOnly	111
8.2.2.2.2.4	getForce	111
8.2.2.2.3	高レベル API	111
8.2.2.2.3.1	calcForceAllAndWriteBack	112
8.2.2.3	使用済マクロ	112
9	エラーメッセージ	115
9.1	概要	115
10	よくしこむバグ (仮)	116
10.1	概要	116
10.2	ユーザー定義クラス	116
10.2.1	概要	116
10.2.2	FullParticle 型	116
10.2.3	EssentialParticleI 型	116
10.2.4	EssentialParticleJ 型	116
10.2.5	SuperParticleJ 型	116
10.2.6	Moment 型	116
10.2.7	Force 型	116
10.2.8	calcForceEpEp 型	116
10.2.9	calcForceSpEp 型	116
10.2.10	ヘッダ型	116
10.3	プログラム本体	116
10.3.1	概要	116
10.3.2	オブジェクトの生成	116
11	よく知られているバグ	117
12	限界	118
13	ユーザーサポート	119
13.1	概要	119
13.2	ユーザーへのお願い	119
14	ライセンス	120

1 概要

2 ファイル構成

2.1 概要

2.2 標準機能関係ファイル

2.3 拡張機能関係ファイル

2.4 テストコード

2.5 サンプルコード

3 コンパイル時のマクロによる選択

3.1 概要

3.2 空間次元数

3.2.1 概要

3.2.2 直交座標系 2 次元

3.2.3 直交座標系 3 次元

3.3 並列処理

3.3.1 概要

3.3.2 OpenMP の使用

3.3.3 MPI の使用

4 名前空間

4.1 概要

4.2 ParticleSimulator

4.2.1 ParticleMesh

5 データ型

5.1 概要

FDPS では独自の整数型、実数型、ベクトル型、行列型、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、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。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、行列型、SEARCH_MODE 型、列挙型の順に記述する。

5.2 整数型

5.2.1 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。

5.2.2 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 }
4 namespace PS = ParticleSimulator;
```

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。

5.2.3 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 S64;
3 }
4 namespace PS = ParticleSimulator;
```

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。

5.2.4 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 U32;  
3 }  
4 namespace PS = ParticleSimulator;
```

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。

5.2.5 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 U64;  
3 }  
4 namespace PS = ParticleSimulator;
```

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。

5.3 実数型

5.3.1 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。

5.3.2 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 }  
4 namespace PS = ParticleSimulator;
```

5.3.3 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 }
4 namespace PS = ParticleSimulator;
```

5.4 ベクトル型

5.4.1 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。

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

```

5.4.2.1 コンストラクタ

```

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

```

- 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>(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>(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。

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

5.4.2.2 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。

- 返回值

`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。

5.4.2.3 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。

5.4.2.4 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。

- 戻り値

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。

- 戻り値

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。

- 戻り値

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。

- 戻り値

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。

5.4.2.5 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。
- 返回值
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。
- 返回值
T 型。自身と rhs の外積を取った値を返す。

5.4.2.6 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。
- 返回值
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。

5.4.3 PS::Vector3

PS::Vecotr3 は x, y, z 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 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(返り値はスカラー!!)
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 }

```

5.4.3.1 コンストラクタ

```

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

```

- 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>(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>(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。

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

5.4.3.2 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。

5.4.3.3 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。

5.4.3.4 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。

- 返回值

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。

- 返回值

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。

- 返回值

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。

- 返回值

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。

5.4.3.5 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。
- 返回值
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。
- 返回值
T 型。自身と rhs の外積を取った値を返す。

5.4.3.6 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
- 返回值
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。

5.4.4 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 }
14 namespace PS = ParticleSimulator;
```

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。

5.5 対称行列型

5.5.1 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。

5.5.2 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;

```

5.5.2.1 コンストラクタ

```

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

```

- 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>(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>(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。

```
template<typename T>
PS::MatrixSym2<T>(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。

5.5.2.2 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。

5.5.2.3 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。

- 返回值

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。

- 返回值

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。

5.5.2.4 トレースの計算

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

- 引数

なし

- 返回值

T 型。

- 機能

トレースを計算し、その結果を返す。

5.5.2.5 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

5.5.3 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)
27         const;
28     MatrixSym3 operator - (const MatrixSym3 & rhs) const;
29     const MatrixSym3 & operator -= (const MatrixSym3 & rhs)
30         const;
31
32     // トレースを取る
33     T getTrace() const;
34
35     // MatrixSym3<U>への型変換
36     template <typename U>
37     operator MatrixSym3<U> () const;
38 }
39 namespace PS = ParticleSimulator;

```

5.5.3.1 コンストラクタ

```

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

```

- 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>(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>(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。

```
template<typename T>
PS::MatrixSym3<T>(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。

5.5.3.2 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。

5.5.3.3 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。

- 返回值

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。

- 返回值

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。

5.5.3.4 トレースの計算

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

- 引数

なし

- 返回值

T 型。

- 機能

トレースを計算し、その結果を返す。

5.5.3.5 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

5.5.4 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。

5.6 SEARCH_MODE 型

5.6.1 概要

本節では、SEARCH_MODE 型について記述する。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。SEARCH_MODE 型にはSEARCH_MODE_LONG, SEARCH_MODE_LONG_CUTOFF, SEARCH_MODE_GATHER, SEARCH_MODE_SCATTER, 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。

5.6.2 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。

5.6.3 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 法を並用) などに適用できる。

5.6.4 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。

5.6.5 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。

5.6.6 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。

5.7 列挙型

5.7.1 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。

5.7.2 BOUNDARY_CONDITION 型

5.7.2.1 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 }
15 namespace PS = ParticleSimulator;
```

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。

5.7.2.2 PS::BOUNDARY_CONDITION_OPEN

開放境界となる。

5.7.2.3 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。

5.7.2.4 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。

5.7.2.5 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。

5.7.2.6 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。

5.7.2.7 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。

5.7.2.8 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。

5.7.2.9 PS::BOUNDARY_CONDITION_PERIODIC_XYZ

x, y, z 軸方向すべてが周期境界となる。

5.7.2.10 PS::BOUNDARY_CONDITION_SHEARING_BOX

未実装。

5.7.2.11 PS::BOUNDARY_CONDITION_USER_DEFINED

未実装。

6 ユーザー定義クラス、ファンクタ

6.1 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、ある相互作用 (`SEARCH_MODE` 型が `SEARCH_MODE_LONG` または `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、ツリーセルのモーメント情報を持つ `Moment` 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、ユーザー定義ファンクタには、 j 粒子から i 粒子への作用を計算する `calcForceEpEp` ファンクタ、超粒子から i 粒子への作用を計算する `calcForceSpEp` ファンクタがある。

この節で記述するのは、これらのクラスに関する規定である。ユーザーはこれらのクラスの間でのデータのやりとりや、ファンクタ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とファンクタのメンバ関数内で行われる。以下、必要なメンバ関数とその規定について記述する。

6.2 FullParticle クラス

6.2.1 概要

`FullParticle` クラスは粒子情報すべてを持つクラスである。FDPS はこのクラスからいくつかの情報を読み取る。FDPS が情報を読み取るために、この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。

6.2.2 前提

この節の中では、以下のように、名前空間 `ParticleSimulator` を `PS` と省略し、`FullParticle` というクラスを例とする。`FullParticle` という名前は自由に変えることができる。

```
namespace PS = ParticleSimulator;
class FullParticle;
```

6.2.3 必要なメンバ関数

6.2.3.1 概要

常に必要なメンバ関数は `getPos` と `copyFromForce` である。`getPos` は `FullParticle` の位置情報を FDPS に読み込ませるための関数で、`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。

6.2.3.2 getPos

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

- 前提

FullParticle 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。FullParticle クラスの位置情報を保持したメンバ変数。

- 機能

FullParticle クラスの位置情報を保持したメンバ変数を返す。

- 備考

FullParticle 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 FullParticle::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。

6.2.3.3 copyFromForce

```
class Force {
public:
    PS::F64vec acc;
    PS::F64    pot;
};
class FullParticle {
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 FullParticle 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 FullParticle 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。FullParticle のメンバ変数名は変更可能。メンバ関数 FullParticle::copyFromForce の引数名は変更可能。

6.2.4 場合によっては必要なメンバ関数

6.2.4.1 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 SEARCH_MODE 型に 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::adjustPositionIntoRootDomain

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。

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

6.2.4.2.1 *getRsearch*

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

- 前提

FullParticle 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。 FullParticle 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

FullParticle 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

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

6.2.4.3 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 `readParticleAscii`, `readParticleBinary`, `writeParticleAscii`, `writeParticleBinary` を使用するときそれぞれ `readAscii`, `readBinary`, `writeAscii`, `writeBinary` というメンバ関数が必要となる。以下、`readAscii` と `readBinary` の規定は同じであり、`writeAscii` と `writeBinary` の規定も同じである。以下、それぞれの規定について記述する。

6.2.4.3.1 readAscii, readBinary

```
class FullParticle {
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]);
    }
    void readBinary(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FullParticle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、それ以降の列にはデータがないとする。ファイルの形式はアスキー形式(readAsciiの場合)、バイナリー形式(readBinaryの場合)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FullParticleクラスのid、mass、posの情報を読み取る。

- 備考

なし。

6.2.4.3.2 writeAscii, writeBinary

```
class FullParticle {
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]);
    }
    void writeBinary(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FullParticle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、それ以降の列にはデータがないとする。ファイルの形式はアスキー形式(writeAsciiの場合)、バイナリー形式(writeBinaryの場合)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FullParticleクラスのメンバ変数id、mass、posの情報を書き出す。

- 備考

なし。

6.2.4.4 粒子群クラスの `adjustPositionIntoRootDomain` を用いる場合

6.2.4.4.1 *setPos*

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

- 前提

FullParticle 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 FullParticle クラスの位置情報に書き込む。

- 備考

FullParticle クラスのメンバ変数 `pos` の変数名は変更可能。メンバ関数 `FullParticle::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。

6.2.4.5 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 `getChargeParticleMesh` と `copyFromForceParticleMesh` 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。

6.2.4.5.1 *getChargeParticleMesh*

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

- 前提

FullParticle 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の変数を返す。

- 備考

FullParticle クラスのメンバ変数 `mass` の変数名は変更可能。

6.2.4.5.2 *copyFromForceParticleMesh*

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

- 前提

FullParticle 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

FullParticle 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 FullParticle::copyFromForceParticleMesh の引数 acc_pm の引数名は変更可能。

6.3 EssentialParticleI クラス

6.3.1 概要

EssentialParticleI クラスは相互作用の計算に必要な*i*粒子情報を持つクラスである。EssentialParticleI クラスは FullParticle クラス (節 6.2) のサブセットであり、FDPS の内部では、このクラスが FullParticle クラスから情報を読み取る。情報を読み取るために、この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。

6.3.2 前提

この節の中では、名前空間 ParticleSimulator を PS と省略する。このクラスのクラス名を EssentialParticleI とする。また、粒子すべての情報を持つクラスのクラス名を FullParticle とする。この FullParticle は節 6.2 のクラス FullParticle と同一のものである。EssentialParticleI, FullParticle というクラス名は変更可能である。

ParticleSimulator を PS と省略すること、EssentialParticleI と FullParticle の宣言は以下の通りである。

```
namespace PS = ParticleSimulator;
class FullParticle;
class EssentialParticleI;
```


6.3.3 必要なメンバ関数

6.3.3.1 概要

常に必要なメンバ関数は `getPos` と `CopyfromFP` である。`getPos` は `EssentialParticleI` クラスの位置情報を `FDPS` に読み込ませるための関数で、`copyFromFP` は `FullParticle` クラスの情報を `EssentialParticleI` 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。

6.3.3.2 `getPos`

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

- 前提

`EssentialParticleI` 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。`EssentialParticleI` クラスの位置情報を保持したメンバ変数。

- 機能

`EssentialParticleI` クラスの位置情報を保持したメンバ変数を返す。

- 備考

`EssentialParticleI` クラスのメンバ変数 `pos` の変数名は変更可能。

6.3.3.3 copyFromFP

```
class FullParticle {
public:
    PS::S64    identity;
    PS::F64    mass;
    PS::F64vec position;
    PS::F64vec velocity;
    PS::F64vec acceleration;
    PS::F64    potential;
};

class EssentialParticleI {
public:
    PS::S64    id;
    PS::F64vec pos;
    void copyFromFP(const FullParticle & fp) {
        this->id = fp.identity;
        this->pos = fp.position;
    }
};
```

- 前提

FullParticle クラスのメンバ変数 identity, position と EssentialParticleI 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 FullParticle &型。FullParticle クラスの情報を持つ。

- 返値

なし。

- 機能

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

- 備考

FullParticle クラスのメンバ変数の変数名、EssentialParticleI クラスのメンバ変数の変数名は変更可能。メンバ関数 EssentialParticleI::copyFromFP の引数名は変更可能。EssentialParticleI クラスの粒子情報は FullParticle 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。

6.3.4 場合によっては必要なメンバ関数

6.3.4.1 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 `SEARCH_MODE` 型に `SEARCH_MODE_GATHER` または `SEARCH_MODE_SYMMETRY` を用いる場合に必要となるメンバ関数について記述する。

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

6.3.4.2.1 *getRsearch*

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

- 前提

`EssentialParticleI` 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。 `EssentialParticleI` 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

`EssentialParticleI` 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

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

6.4 EssentialParticleJ クラス

6.4.1 概要

EssentialParticleJ クラスは相互作用の計算に必要な j 粒子情報を持つクラスである。FDPS の内部では、このクラスが FullParticle クラスから情報を読み取る。情報を読み取るために、この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。

6.4.2 前提

この節の中では、名前空間 ParticleSimulator を PS と省略する。このクラスのクラス名を EssentialParticleJ とする。また、粒子すべての情報を持つクラスのクラス名を FullParticle とする。この FullParticle は節 6.2 のクラス FullParticle と同一のものである。EssentialParticleJ, FullParticle というクラス名は変更可能である。

ParticleSimulator を PS と省略すること、EssentialParticleJ と FullParticle の宣言は以下の通りである。

```
namespace PS = ParticleSimulator;
class FullParticle;
class EssentialParticleJ;
```

6.4.3 必要なメンバ関数

6.4.3.1 概要

常に必要なメンバ関数は getPos と CopyfromFP である。getPos は EssentialParticleJ クラスの位置情報を FDPS に読み込ませるための関数で、copyFromFP は FullParticle クラスの情報を EssentialParticleJ 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。

6.4.3.2 getPos

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

- 前提

EssentialParticleJ 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。EssentialParticleJ クラスの位置情報を保持したメンバ変数。

- 機能

EssentialParticleJ クラスの位置情報を保持したメンバ変数を返す。

- 備考

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

6.4.3.3 copyFromFP

```
class FullParticle {
public:
    PS::S64    identity;
    PS::F64    mass;
    PS::F64vec position;
    PS::F64vec velocity;
    PS::F64vec acceleration;
    PS::F64    potential;
};

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

- 前提

FullParticle クラスのメンバ変数 identity, mass, position と EssentialParticleJ 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 FullParticle &型。FullParticle クラスの情報を持つ。

- 返値

なし。

- 機能

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

- 備考

FullParticle クラスのメンバ変数の変数名、EssentialParticleJ クラスのメンバ変数の変数名は変更可能。メンバ関数 EssentialParticleJ::copyFromFP の引数名は変更可能。EssentialParticleJ クラスの粒子情報は FullParticle 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。

6.4.4 場合によっては必要なメンバ関数

6.4.4.1 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 SEARCH_MODE 型に 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。

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

6.4.4.2.1 *getRsearch*

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

- 前提

EssentialParticleJ 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。 EssentialParticleJ 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

EssentialParticleJ 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

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

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

6.4.4.3.1 setPos

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

- 前提

EssentialParticleJ 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。EssentialParticleJ クラスのメンバ変数 pos の元データとなっているのは FullParticle 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 EssentialParticleJ クラスの位置情報に書き込む。

- 備考

EssentialParticleJ クラスのメンバ変数 pos の変数名は変更可能。メンバ関数 EssentialParticleJ::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。

6.5 SuperParticleJ クラス

6.5.1 概要

6.5.2 既存のクラス

6.5.3 必要なメンバ関数

6.5.4 場合によっては必要なメンバ関数

6.5.4.1 getPos

6.5.4.2 setPos

longcutoff

6.5.4.3 copyFromMoment

6.5.4.4 convertToMoment

6.5.4.5 clear

6.6 Moment クラス

6.6.1 概要

Moment クラスは近い粒子同士でまとめた複数の粒子のモーメント情報を持つクラスである。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。このクラスが持つメンバ関数はこれらモーメント情報を計算するものや、モーメント情報を計算するためのデータを取得するものである。

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。

6.6.2 既存のクラス

6.6.2.1 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SEARCH_MODE型を選んだ場合に有効である。以下、各SEARCH_MODE型において選ぶことのできるMoment型を記述する。SEARCH_MODE_GATHER, SEARCH_MODE_SCATTER, 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。

6.6.2.2 SEARCH_MODE_LONG

6.6.2.2.1 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。

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

- クラス名 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 6.4) 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 EssentialParticleJ のクラス名は変更自由。

6.6.2.2.2 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。

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

- クラス名 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 6.4) 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 EssentialParticleJ のクラス名は変更自由。

6.6.2.2.3 *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。

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

- クラス名 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 6.4) 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 EssentialParticleJ のクラス名は変更自由。

6.6.2.2.4 *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。

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

- クラス名 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 6.4) 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 EssentialParticleJ のクラス名は変更自由。

6.6.2.2.5 *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。

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

- クラス名 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 6.4) 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 EssentialParticleJ のクラス名は変更自由。

6.6.2.3 SEARCH_MODE_LONG_CUTOFF

6.6.2.3.1 *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。

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

- クラス名 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 6.4) 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 EssentialParticleJ のクラス名は変更自由。

6.6.3 必要なメンバ関数

6.6.3.1 概要

以下では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 Moment とする。これは変更自由である。

6.6.3.2 init

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

- 前提

なし

- 引数
なし
- 返値
なし
- 機能
Moment クラスのオブジェクトの初期化をする。
- 備考
なし

6.6.3.3 getPos

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

- 前提
Moment 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。Moment クラスのメンバ変数 pos。
- 機能
Moment クラスのメンバ変数 pos を返す。
- 備考
Moment クラスのメンバ変数 pos の変数名は変更自由。

6.6.3.4 getCharge

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

- 前提

Moment 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。Moment クラスのメンバ変数 `mass`。

- 機能

Moment クラスのメンバ変数 `mass` を返す。

- 備考

Moment クラスのメンバ変数 `mass` の変数名は変更自由。

6.6.3.5 accumulateAtLeaf

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

- 前提

Moment 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。Moment 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

ツリーのリーフセルのモーメントを計算する。

- 備考

Moment 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。

6.6.3.6 accumulate

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

- 前提

Moment 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。Moment 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 `Moment &`型。Moment クラスのオブジェクト。

- 返値

なし。

- 機能

ツリーのリーフセル以外のセルのモーメントを計算する。

- 備考

Moment 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。

6.6.3.7 set

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

- 前提

Moment 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。Moment 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 `Moment::accumulateAtLeaf`, `Moment::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なので、ここで規格化する。

- 備考

Moment 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。

6.6.3.8 accumulateAtLeaf2

建設中

6.6.3.9 accumulate2

建設中

6.7 Force クラス

6.7.1 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。

6.7.2 前提

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

6.7.3 必要なメンバ関数

常に必要なメンバ関数は 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、clear について記述する。

6.7.3.1 clear

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

- 前提

Force クラスのメンバ変数は acc と pot。

- 引数
なし
- 返値
なし。
- 機能
Force クラスのメンバ変数を初期化する。
- 備考
Force クラスのメンバ変数 `acc`, `pot` の変数名は変更可能。

6.8 ヘッダクラス

6.8.1 概要

6.8.2 必要なメンバ関数

6.8.2.1 概要

6.8.3 場合によっては必要なメンバ関数

6.8.3.1 概要

6.8.3.2 `readParticleAscii`

6.8.3.3 `readParticleBinary`

6.8.3.4 `writeParticleAscii`

6.8.3.5 `writeParticleBinary`

6.9 `calcForceEpEp` ファンクタ

6.10 `calcForceSpEp` ファンクタ

7 プログラムの開始と終了

7.1 概要

プログラムの開始と終了に必要な API を記述する。

7.2 API

以下に API を記述する。このとき

```
namespace PS = ParticleSimulator;
```

となっているものとする。

7.2.1 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 ライブラリの初期化を行う。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。

7.2.2 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 ライブラリの終了処理を行う。

8 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。

8.1 標準機能

8.1.1 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。

8.1.2 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。

8.1.2.1 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 14: DomainInfo0

```
1 namespace ParticleSimulator {  
2     class DomainInfo;  
3 }  
4 namespace PS = ParticleSimulator;
```

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;
```

8.1.2.2 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。

8.1.2.2.1 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 15: 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 }
14 namespace PS = ParticleSimulator;
```

8.1.2.2.1.1 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。

8.1.2.2.1.2 *initialize* *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。1 度は呼ぶ必要があるが、2 度呼ぶと例外が送出される。

8.1.2.2.1.3 *setNumberOfDomainMultiDimension*

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

ルートドメイン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 軸方向のルートドメインの分割数である。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。

8.1.2.2.1.4 *setBoundaryCondition*

setBoundaryCondition

```
void PS::DomainInfo::setBoundaryCondition  
    (enum 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、5.7.2 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。

8.1.2.2.1.5 *setPosRootDomain*

setPosRootDomain

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

- 引数

low: 入力。 PS::F32vec 型。ルートドメインの下限 (閉境界)。

high: 入力。 PS::F32vec 型。ルートドメインの上限 (解境界)。

- 返値

なし

- 機能

ルートドメイン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。

8.1.2.2.2 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 16: DomainInfo2

```
1 namespace ParticleSimulator {
2     class DomainInfo{
3     public:
4         template<class Tpsys>
5         void collectSampleParticle(Tpsys & psys,
6                                     const F32 weight=1.0,
7                                     const bool clear=true);
8         void decomposeDomain();
9         template<class Tpsys>
10        void decomposeDomainAll(Tpsys & psys,
11                                const F32 wgh=1.0);
12     };
13 }
14 namespace PS = ParticleSimulator;
```

8.1.2.2.2.1 *collectSampleParticle*

collectSampleParticle

```
template<class Tpsys>
void PS::DomainInfo::collectSampleParticle
    (Tpsys & psys,
     const PS::F32 weight=1.0,
     const bool clear=true);
```

- 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。デフォルト 1.0。

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。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` から粒子をサンプルする。`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)。`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。

8.1.2.2.2.2 *decomposeDomain*

`decomposeDomain`

```
template<class Tpsys>
void PS::DomainInfo::decomposeDomain();
```

- 引数

なし

- 返値

なし

- 機能

ルートドメインの分割を行う。

8.1.2.2.2.3 *decomposeDomainAll*

`decomposeDomainAll`

```
template<class Tpsys>
void PS::DomainInfo::decomposeDomainAll
    (Tpsys & psys,
     const PS::F32 weight=1.0);
```

- 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。デフォルト 1.0。

- 返値

なし

- 機能

粒子群クラスオブジェクト `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` で行われていることが一度に行われる。`weight` の意味は `PS::DomainInfo::collectSampleParticle` と同じ。

8.1.3 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。

8.1.3.1 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 17: ParticleSystem0

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

テンプレート引数 `Tptcl` はユーザー定義の `FullParticle` 型である。

粒子群クラス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<Tptcl> system;
```

テンプレート引数 `Tptcl` はユーザー定義の `FullParticle` 型である。

8.1.3.2 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。

8.1.3.2.1 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 18: ParticleSystem1

```
1 namespace ParticleSimulator {
```

```

2     template<class Tptcl>
3     class ParticleSystem{
4     public:
5         ParticleSystem();
6         void initialize();
7         void setNumberOfDomainMultiDimension(const S32 nx,
8                                               const S32 ny,
9                                               const S32 nz=1);
10        void setAverageTargetNumberOfSampleParticlePerProcess
11            (const S32 & nsampleperprocess);
12    };
13 }
14 namespace PS = ParticleSimulator;

```

8.1.3.2.1.1 コンストラクタ コンストラクタ

```
void PS::ParticleSystem::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。

8.1.3.2.1.2 *initialize* initialize

```
void PS::ParticleSystem::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。

8.1.3.2.1.3 *setAverateTargetNumberOfSampleParticlePerProcess*

setAverateTargetNumberOfSampleParticlePerProcess

```
void PS::ParticleSystem::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。

8.1.3.2.2 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 19: 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         const S32 getNumberOfParticleLocal();  
8         S32 getNumberOfParticleGlobal();  
9     };  
10 }  
11 namespace PS = ParticleSimulator;
```

8.1.3.2.2.1 *operator []*

operator []

```
Tptcl & PS::ParticleSystem::operator []  
    (const 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 1 つ。

- 機能

Tptcl 型のオブジェクト 1 つ返す。

8.1.3.2.2.2 *setNumberOfParticleLocal*

setNumberOfParticleLocal

```
void PS::ParticleSystem::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。

8.1.3.2.2.3 *getNumberOfParticleLocal*

getNumberOfParticleLocal

```
const PS::S32 PS::ParticleSystem::getNumberOfParticleLocal();
```

- 引数

なし

- 返値

const 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。

8.1.3.2.2.4 *getNumberOfParticleGlobal*

getNumberOfParticleGlobal

```
const PS::S32 PS::ParticleSystem::getNumberOfParticleGlobal();
```

- 引数

なし

- 返値

const 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。

8.1.3.2.3 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 20: 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     template <class Theader>
20     void writeParticleAscii(const char * const filename,
21                             const Theader & header);
22     void writeParticleAscii(const char * const filename,
23                             const char * format);
24     void writeParticleAscii(const char * const filename);
25 };
26 }
27 namespace PS = ParticleSimulator;

```

8.1.3.2.3.1 *readParticleAscii*

readParticleAscii

```

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

```

- 引数

filename: 入力。const char * const 型。

format: 入力。const char * const 型。

header: 入力。Theader &型。

- 返値

なし

- 機能

建設中

```

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

```

- 引数

filename: 入力。const char * const 型。

header: 入力。Theader &型。

- 返値

なし

- 機能

建設中

```
void PS::ParticleSystem::readParticleAscii
    (const char * const filename,
     const char * const format);
```

- 引数

filename: 入力。const char * const 型。

format: 入力。const char * const 型。

- 返値

なし

- 機能

建設中

```
void PS::ParticleSystem::readParticleAscii
    (const char * const filename);
```

- 引数

filename: 入力。const char * const 型。

- 返値

なし

- 機能

建設中

8.1.3.2.3.2 *readParticleBinary*

readParticleBinary

8.1.3.2.3.3 *writeParticleAscii*

writeParticleAscii

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

- 引数

filename: 入力。const char * const 型。

format: 入力。const char * const 型。

header: 入力。const Theader &型。

- 返値

なし

- 機能

建設中

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

- 引数

filename: 入力。const char * const 型。

header: 入力。const Theader &型。

- 返値

なし

- 機能

建設中

```
void PS::ParticleSystem::writeParticleAscii
    (const char * const filename,
     const char * format);
```

- 引数

filename: 入力。const char * const 型。

format: 入力。const char * 型。

- 返値

なし

- 機能

建設中

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

- 引数

filename: 入力。const char * const 型。

- 返値

なし

- 機能

建設中

8.1.3.2.3.4 *writeParticleBinary*

writeParticleBinary

8.1.3.2.4 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: 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 namespace PS = ParticleSimulator;

```

8.1.3.2.4.1 *exchangeParticle*

exchangeParticle

```

template<class Tdinfo>
void PS::ParticleSystem::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。

8.1.3.2.5 その他

8.1.3.2.5.1 *adjustPositionIntoRootDomain*

adjustPositionIntoRootDomain

8.1.4 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。

8.1.4.1 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 22: TreeForForce0

```

1 namespace ParticleSimulator {
2     template<class TSM,
3         class Tforce,

```

```

4         class Tepi,
5         class Tepj,
6         class Tmomloc,
7         class Tmomglb,
8         class Tspj>
9     class TreeForForce;
10 }
11 namespace PS = ParticleSimulator;

```

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

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、SEARCH_MODE 型が SEARCH_MODE_LONG, SEARCH_MODE_LONG_CUTOFF, SEARCH_MODE_GATHER, SEARCH_MODE_SCATTER, SEARCH_MODE_SYMMETRY の場合のオブジェクトの生成方法を記述する。

8.1.4.1.1 SEARCH_MODE_LONG

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceLong<Tforce, Tepi, Tepj, Tmom, Tspj>::Normal system;
```

テンプレート引数は順に、Force 型 (ユーザー定義)、EssentialParticleI 型 (ユーザー定義)、EssentialParticleJ 型 (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 型 (ユーザー定義)、SuperParticleJ 型 (ユーザー定義) である。

あらかじめ Moment 型と SuperParticleJ 型を指定した型も用意した。これらはモーメントの計算方法別に 6 種類ある。モーメント計算の中心を粒子の重心または粒子の幾何中心とした場合に、単極子まで、四重極子まで計算するものである。以下、粒子の重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、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、のオブジェクト方法をこの順で記述する。すべて system というオブジェクトを生成している。

```
PS::TreeForForceLong<Tforce, Tepi, Tepj>::Monopole system;
```

```
PS::TreeForForceLong<Tforce, Tepi, Tepj>::Quadrupole system;
```

```
PS::TreeForForceLong<Tforce, Tepi, Tepj>::MonopoleGeometricCenter system;
```

```
PS::TreeForForceLong<Tforce, Tepi, Tepj>::DipoleGeometricCenter system;
```

```
PS::TreeForForceLong<Tforce, Tepi, Tepj>::QuadrupoleGeometricCenter system;
```

すべての型のテンプレート引数は順に、Force 型、EssentialParticleI 型、EssentialParticleJ 型である。

8.1.4.1.2 SEARCH_MODE_LONG_CUTOFF

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceLong<Tforce, Tepi, Tepj, Tmom, Tspj>::WithCutoff system;
```

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

あらかじめ Moment 型と SuperParticleJ 型を指定した型も用意した。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<Tforce, Tepi, Tepj>::MonopoleWithCutoff system;
```

テンプレート引数は順に、Force 型、EssentialParticleI 型、EssentialParticleJ 型である。

8.1.4.1.3 SEARCH_MODE_GATHER

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceShort<Tforce, Tepi, Tepj>::Gather system;
```

テンプレート引数は順に、Force 型、EssentialParticleI 型、EssentialParticleJ 型である。

8.1.4.1.4 SEARCH_MODE_SCATTER

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceShort<Tforce, Tepi, Tepj>::Scatter system;
```

テンプレート引数は順に、Force 型、EssentialParticleI 型、EssentialParticleJ 型である。

8.1.4.1.5 SEARCH_MODE_SYMMETRY

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceShort<Tforce, Tepi, Tepj>::Symmetry system;
```

テンプレート引数は順に、Force 型、EssentialParticleI 型、EssentialParticleJ 型である。

8.1.4.2 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。

8.1.4.2.1 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 23: TreeForForceI

```
1 namespace ParticleSimulator {
2     template<class TSM,
3             class Tforce,
4             class Tepi,
5             class Tepj,
6             class Tmomloc,
7             class Tmomglb,
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 }
18 namespace PS = ParticleSimulator;
```

8.1.4.2.1.1 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。

8.1.4.2.1.2 *initialize*

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。

8.1.4.2.2 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: TreeForForce1

```
1 namespace ParticleSimulator {
2     template<class TSM,
3         class Tforce,
```

```

4         class Tepi,
5         class Tepj,
6         class Tmomloc,
7         class Tmomglb,
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 & difno);
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,
23                      const bool clear=true);
24        template<class Tfunc_ep_ep, class Tfunc_ep_sp>
25        void calcForce(Tfunc_ep_ep pfunc_ep_ep,
26                      Tfunc_ep_sp pfunc_ep_sp,
27                      const bool clear=true);
28        Tforce getForce(const S32 i);
29    };
30 }
31 namespace PS = ParticleSimulator;

```

8.1.4.2.2.1 *setParticleLocalTree*

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。

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。

8.1.4.2.2.2 *makeLocalTree*

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。

8.1.4.2.2.3 *makeGlobalTree*

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。

8.1.4.2.2.4 *calcMomentGlobalTree*(仮)

calcMomentGlobalTree(仮)

```
template<class Tdinfo>
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。

8.1.4.2.2.5 calcForce

calcForce

```
template<class Tfunc_ep_ep>
void PS::TreeForForce::calcForce
    (Tfunc_ep_ep pfunc_ep_ep,
     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関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *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。粒子間相互作用は短距離力の場合に限る。

```
template<class Tfunc_ep_ep, class Tfunc_ep_sp>
void PS::TreeForForce::calcForce
    (Tfunc_ep_ep pfunc_ep_ep,
     Tfunc_ep_sp pfunc_ep_sp,
     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関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *型。

pfunc_ep_sp: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const SuperParticleJ *型、PS::S32 型、Force *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。粒子間相互作用は長距離力の場合に限る。

8.1.4.2.2.6 *getForce*

getForce

```
Tforce 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

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

- 機能

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

8.1.4.2.2.7 *copyLocalTreeStructure*

copyLocalTreeStructure

今後、追加する。

8.1.4.2.2.8 *repeatLocalCalcForce*

repeatLocalCalcForce

今後、追加する。

8.1.4.2.3 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 25: TreeForForce1

```
1 namespace ParticleSimulator {  
2     template<class TSM,
```

```

3         class Tforce,
4         class Tepi,
5         class Tepj,
6         class Tmomloc,
7         class Tmomglb,
8         class Tspj>
9 class TreeForForce{
10 public:
11     template<class Tfunc_ep_ep,
12             class Tpsys,
13             class Tdinfo>
14     void calcForceAllAndWriteBack(Tfunc_ep_ep pfunc_ep_ep,
15                                 Tpsys & psys,
16                                 Tdinfo & dinfo,
17                                 const bool clear_force =
18                                     true);
19
20     template<class Tfunc_ep_ep,
21             class Tfunc_ep_sp,
22             class Tpsys,
23             class Tdinfo>
24     void calcForceAllAndWriteBack(Tfunc_ep_ep pfunc_ep_ep,
25                                 Tfunc_ep_sp pfunc_ep_sp,
26                                 Tpsys & psys,
27                                 TDinfo & dinfo,
28                                 const bool clear_force=
29                                     true);
30
31     template<class Tfunc_ep_ep,
32             class Tfunc_ep_sp,
33             class Tpsys,
34             class Tdinfo>
35     void calcForceAll(Tfunc_ep_ep pfunc_ep_ep,
36                     Tfunc_ep_sp pfunc_ep_sp,
37                     Tpsys & psys,
38                     Tdinfo & dinfo,
39                     const bool clear_force=true);
40     template<class Tfunc_ep_ep,
41             class Tfunc_ep_sp,
42             class Tpsys,
43             class Tdinfo>

```

```

41     void calcForceAll(Tfunc_ep_ep pfunc_ep_ep,
42                       Tfunc_ep_sp pfunc_ep_sp,
43                       Tpsys & psys,
44                       Tdinfo & dinfo,
45                       const bool clear_force=true);
46
47     template<class Tfunc_ep_ep,
48              class Tdinfo>
49     void calcForceMakeingTree(Tfunc_ep_ep pfunc_ep_ep,
50                               Tdinfo & dinfo,
51                               const bool clear_force=true);
52     template<class Tfunc_ep_ep,
53              class Tfunc_ep_sp,
54              class Tdinfo>
55     void calcForceMakingTree(Tfunc_ep_ep pfunc_ep_ep,
56                              Tfunc_ep_sp pfunc_ep_sp,
57                              Tdinfo & dinfo,
58                              const bool clear_force=true);
59
60     template<class Tfunc_ep_ep,
61              class Tpsys>
62     void calcForceAndWriteBack(Tfunc_ep_ep pfunc_ep_ep,
63                               Tpsys & psys,
64                               const bool clear=true);
65     template<class Tfunc_ep_ep,
66              class Tfunc_ep_sp,
67              class Tpsys>
68     void calcForceAndWriteBack(Tfunc_ep_ep pfunc_ep_ep,
69                              Tfunc_ep_sp pfunc_ep_sp,
70                              Tpsys & psys,
71                              const bool clear=true);
72 };
73 }
74 namespace PS = ParticleSimulator;

```

8.1.4.2.3.1 *calcForceAllAndWriteBack*

calcForceAllAndWriteBack

```
template<class Tfunc_ep_ep,  
        class Tpsys,  
        class Tdinfo>  
void PS::TreeForForce::calcForceAllAndWriteBack  
    (Tfunc_ep_ep pfunc_ep_ep,  
     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関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *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。粒子間相互作用は短距離力の場合に限る。

```
template<class Tfunc_ep_ep,  
        class Tfunc_ep_sp,  
        class Tpsys,  
        class Tdinfo>  
void PS::TreeForForce::calcForceAllAndWriteBack  
    (Tfunc_ep_ep pfunc_ep_ep,  
     Tfunc_ep_sp pfunc_ep_sp,  
     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関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *型。

pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const SuperParticleJ *型、PS::S32 型、Force *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。粒子間相互作用は長距離力の場合に限る。

8.1.4.2.3.2 calcForceAll

calcForceAll

```
template<class Tfunc_ep_ep,
         class Tpsys,
         class Tdinfo>
void PS::TreeForForce::calcForceAll
    (Tfunc_ep_ep pfunc_ep_ep,
     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関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *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。粒子間相互作用は短距離力の場合に限る。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_ep_sp,  
         class Tpsys,  
         class Tdinfo>  
void PS::TreeForForce::calcForceAll  
    (Tfunc_ep_ep pfunc_ep_ep,  
     Tfunc_ep_sp pfunc_ep_sp,  
     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関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *型。

pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const SuperParticleJ *型、PS::S32 型、Force *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。粒子間相互作用は長距離力の場合に限る。`PS::TreeForForce::calcForceAllAndWriteBack` から計算結果の書き戻しがなくなったもの。

8.1.4.2.3.3 *calcForceMakingTree*

`calcForceMakingTree`

```
template<class Tfunc_ep_ep,
         class Tdinfo>
void PS::TreeForForce::calcForceMakingTree
    (Tfunc_ep_ep pfunc_ep_ep,
     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関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に `const EssentialParticleI *`型、`PS::S32` 型、`const EssentialParticleJ *`型、`PS::S32` 型、`Force *`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。粒子間相互作用は短距離力の場合に限る。`PS::TreeForForce::calcForceAllAndWriteBack` から粒子群クラスのオブジェクトの読み込み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_ep_sp,
         class Tdinfo>
void PS::TreeForForce::calcForceMakingTree
    (Tfunc_ep_ep pfunc_ep_ep,
     Tfunc_ep_sp pfunc_ep_sp,
     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関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *型。

pfunc_ep_sp: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const SuperParticleJ *型、PS::S32 型、Force *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。粒子間相互作用は長距離力の場合に限る。PS::TreeForForce::calcForceAllAndWriteBack から粒子群クラスのオブジェクトの読み込みと計算結果の書き戻しがなくなったもの。

8.1.4.2.3.4 calcForceAndWriteBack

calcForceAndWriteBack

```
template<class Tfunc_ep_ep,
        class Tpsys>
void PS::TreeForForce::calcForceAndWriteBack
    (Tfunc_ep_ep pfunc_ep_ep,
     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関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *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。粒子間相互作用は短距離力の場合に限る。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_ep_sp,
        class Tpsys>
void PS::TreeForForce::calcForceAllAndWriteBack
    (Tfunc_ep_ep pfunc_ep_ep,
     Tfunc_ep_sp pfunc_ep_sp,
     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関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const EssentialParticleJ *型、PS::S32 型、Force *型。

pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用関数ポインタ、または関数オブジェクト。関数の引数は第 1 引数から順に const EssentialParticleI *型、PS::S32 型、const SuperParticleJ *型、PS::S32 型、Force *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。粒子間相互作用は長距離力の場合に限る。PS::TreeForForce::calcForceAllAndWriteBack から粒子群クラスのオブジェクトの読み込み、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。

8.1.4.2.4 ネイバーリスト

今後、追加する。

8.1.5 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。

8.1.5.1 API

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

ソースコード 26: 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(const bool
                local);
9         static bool synchronizeConditionalBranchOR(const bool
                local);
10        template<class T>
11        static T getMinValue(const T val);
12        template<class Tfloat, class Tint>
13        static void getMinValue(const Tfloat f_in,
14                                const Tint i_in,
15                                Tfloat & f_out,
16                                Tint & i_out);
17        template<class T>
18        static T getMaxValue(const T val);
19        template<class Tfloat, class Tint>
20        static void getMaxValue(const Tfloat f_in,
21                                const Tint i_in,
22                                Tfloat & f_out,
23                                Tint & i_out );
24        template<class T>
25        static T getSum(const T val);
26    };
27 }
28 namespace PS = ParticleSimulator;
```

8.1.5.1.1 *getRank*

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

- 引数
なし。

- 戻り値

PS::S32 型。全プロセス中でのランクを返す。

8.1.5.1.2 *getNumberOfProc*

- 引数

なし。

- 戻り値

PS::S32 型。全プロセス数を返す。

8.1.5.1.3 *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。

- 戻り値

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。

8.1.5.1.4 *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。

- 戻り値

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。

8.1.5.1.5 *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 AND を取り、結果を返す。

8.1.5.1.6 *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 OR を取り、結果を返す。

8.1.5.1.7 *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。

8.1.5.1.8 *getMaxValue*

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

- 引数
val: 入力。const T 型。
- 返回值
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。
- 返回值
なし。

8.1.5.1.9 *getSum*

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

- 引数
val: 入力。const T 型。
- 返回值
T 型。全プロセスで val の総和を取り、結果を返す。

8.2 拡張機能

8.2.1 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。

8.2.2 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、使用済マクロについて記述する。

8.2.2.1 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 27: ParticleMesh0

```
1 namespace ParticleSimulator {  
2     namespace ParticleMesh {  
3         class ParticleMesh;  
4     }  
5     namespace PM = ParticleMesh;  
6 }  
7 namespace PS = ParticleSimulator;
```

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;
```

8.2.2.2 API

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

8.2.2.2.1 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 28: ParticleMesh1

```
1 namespace ParticleSimulator {
```

```

2     namespace ParticleMesh {
3         class ParticleMesh{
4             ParticleMesh();
5         };
6     }
7     namespace PM = ParticleMesh;
8 }
9 namespace PS = ParticleSimulator;

```

8.2.2.2.1.1 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。

8.2.2.2.2 低レベル API

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

ソースコード 29: ParticleMesh1

```

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

```

```

10         F32vec getForce(F32vec pos);
11     };
12 }
13 namespace PM = ParticleMesh;
14 }
15 namespace PS = ParticleSimulator;

```

8.2.2.2.2.1 *setDomainInfoParticleMesh*

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。

8.2.2.2.2.2 *setParticleParticleMesh*

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。

8.2.2.2.2.3 *calcMeshForceOnly*

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。

8.2.2.2.2.4 *getForce*

getForce

```
PS::F32vec PS::PM::ParticleMesh::getForce  
    (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。

8.2.2.2.3 高レベル API

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

ソースコード 30: ParticleMesh1

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

8.2.2.2.3.1 *calcForceAllAndWriteBack*

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。

8.2.2.3 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

ディカルに列挙する。

- 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
- MULTITIMESTEP
- MY_MPI_BARRIER
- N128_2H
- N256_2H
- N256_H
- N32_2H
- N512_2H

- NEW_DECOMPOSITION
- NOACC
- NPART_DIFFERENT_DUMP
- OMP_SCHEDULE_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 エラーメッセージ

9.1 概要

10 よくしこむバグ(仮)

10.1 概要

10.2 ユーザー定義クラス

10.2.1 概要

10.2.2 FullParticle 型

10.2.3 EssentialParticleI 型

10.2.4 EssentialParticleJ 型

10.2.5 SuperParticleJ 型

10.2.6 Moment 型

10.2.7 Force 型

10.2.8 calcForceEpEp 型

10.2.9 calcForceSpEp 型

10.2.10 ヘッダ型

10.3 プログラム本体

10.3.1 概要

10.3.2 オブジェクトの生成

11 よく知られているバグ

12 限界

13 ユーザーサポート

13.1 概要

13.2 ユーザーへのお願い

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) 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), Ishiyama, Nitadori & Makino (2012) の引用を義務とする。