Sphere

Skeleton for PHysical and Engineering REsearch

Ver 2.0.0

Sphere 概要マニュアル

2010 年 8 月 01 日

目次

０．はじめに

Skeleton for PHysical and Engineering REsearch（以下、SPHERE という）は各種流体解析プログラムを統合し、それらに対して単一のインターフェイスを提供するために C++言語で作成されたフレームワークです。

本手引書では以下について説明します。

・　SPHERE の概念
・　SPHERE のメカニズム
・　SPHERE へのソルバー組込方法

「SPHERE の概念」では SPHERE のしくみとソルバー組込者（以下、開発者という）が SPHERE にソルバーを組み込む際の組込イメージを簡単に解説します。

「SPHERE のメカニズム」では SPHERE を構成している C++クラスについて説明します。また、開発者がソルバーを組み込む際に使用できる共通関数についても説明しています（共通関数の仕様詳細は別紙「共通メソッドとソルバーのコンフィグレーション設定」に収録)。

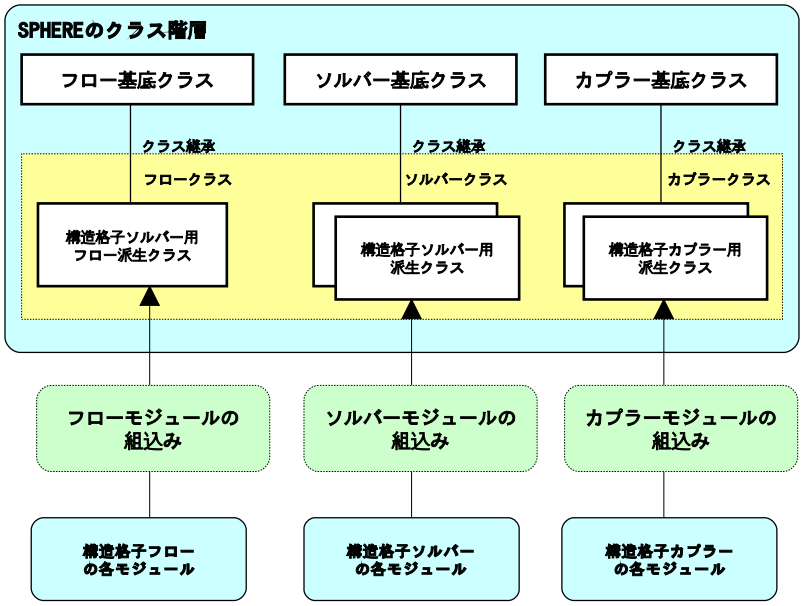
「SPHERE へのソルバー組込方法」では具体的にソルバークラスの作成方法と作成されたソルバークラスへのソルバープログラムの組込方法について解説します。

１．Sphere 概念

SPHERE は C++言語を用いて実装されています。組み込まれるソルバーは C 言語(もしくは C++言語)、Fortran 言語を想定しています。本節では、SPHERE の概念について説明します。

１．１　SPHERE の構成

SPHERE は C++言語を用いてクラス化されたソルバーフレームワークで、ソルバーを組み込むためのソルバークラスをソルバーに提供します。ソルバークラスはソルバー基底クラスの派生クラスとして実装されており、開発者はこのソルバークラスに既存のソルバーモジュールをクラスメソッドとして実装していきます。



１．２　アプリケーションへの入力

SPHERE はコンフィグレーションファイルを入力として起動されます。

使用例：

> sphere　sphere_config.xml

sphere　　：アプリケーション名

sphere_config.xml　：コンフィグレーションファイル

コンフィグレーションファイルとは XML 形式で記述されたファイルでソルバーが計算時に使用する各種パラメータを記述したファイルです。

既存のソルバーを SPHERE に組み込む場合、ソルバー個々に入力ファイルのフォーマットが存在していますが、SPHERE にソルバーを組み込む際には、SPHERE が用意したソルバーコンフィグレーションの記述形式に従って入力ファイルを設計し直し、既存ソルバーの読込みルーチンを SPHERE 用に置き換える作業が必要です。

コンフィグレーションファイルの記述方法及び読み込んだ値の取得方法は別紙「コンフィグレーション文法マニュアル」、「コンフィグレーションクラス」に記載しています。SPHERE では、Sphere(v200)形式のコンフィグレーションファイルと Vsphere 形式のコンフィグレーションファイルの両方を指定できますが、Sphere に実装された新しい機能を使用するためには Sphere(v200)形式である必要があります。

１．３　アプリケーションからの出力

SPHERE を使用して構築されたアプリケーションは SPHERE が用意したファイルフォーマットにて出力することができます。これはインターフェイス共通化の一環ですべてのソルバーが同一形式のファイルを出力することで単一のビューワで可視化を可能とするためです。

SPHERE にてサポートしているファイルフォーマットは以下です。

No	ファイルフォーマット	サポート入出力	説明
1	SPH	入力／出力	計算結果の出力用として実数データをサポートします。
2	SBX	入力／出力	構造定義の入力用として媒質 ID 等の整数データサポートします。
3	SPX	入力／出力	計算結果の出力用として実数データをサポートします。
4	TEXT	出力	計算結果のテキストデータを出力します。

１．４　アプリケーションの制御フロー

SPHERE を使用して構築されたアプリケーションはコンフィグレーションファイルを入力として起動され、計算結果を SPHERE が用意したファイルフォーマットとして出力します。ここではアプリケーションの中の動きについて説明します。

アプリケーション動作の大まかな流れは以下のようになります。

① アプリケーションは起動引数に指定されたコンフィグレーションファイルを読み込みます。

② コンフィグレーションに定義された以下のクラスをインスタンスします。

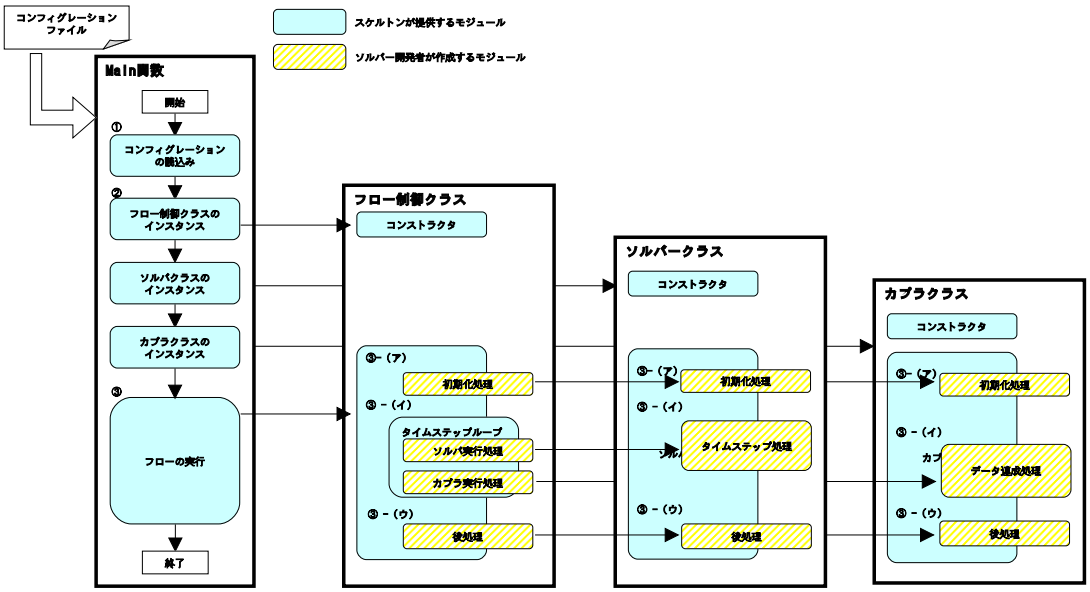
- フロー制御クラス
- ソルバクラス
- カプラクラス

③ インスタンスされたフロー制御クラスによってソルバ、カプラを実行します。
フロー制御クラスからソルバ、カプラの以下の処理を順次実行します。

(ア)　初期化処理

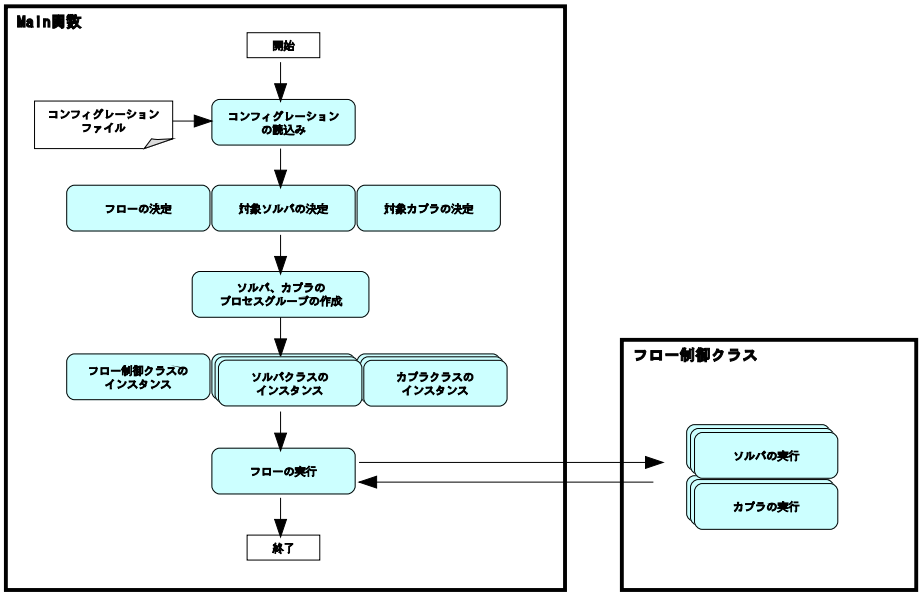
(イ)　タイムステップループ

(ウ)　後処理



１．５　アプリケーションの main 関数の役割

前節ではアプリケーション全体のおおまかな流れについて説明しました。本節ではアプリケーションの main 関数についてもう少し詳しく説明します。



(1) コンフィグレーションの読込み

コンフィグレーションファイルを読み込みます。

コンフィグレーションファイルには対象となるソルバーの識別子とそのソルバーを実行するために必要な各種パラメータが記述されています。

(2) 対象フロー、ソルバー、カプラーの決定

コンフィグレーションファイルに記述されたソルバー識別子から実行するべきソルバーを決定します。

また、ソルバー間の連成の為のカプラー、ソルバー、カプラーの制御を行うフローを決定します。

(3) ソルバー、カプラープロセスグループの作成

コンフィグレーションファイルにソルバーのプロセス数が記述されていますので、ソルバー、カプラーのプロセスグループを作成します。

(4) フロー、ソルバー、カプラークラスのインスタンス

実行するべきフロー、ソルバー、カプラーのクラスをインスタンスします。

(5) フローの実行

ソルバー、カプラーのフロー制御方法が記述されているフロークラスを実行して、制御をフロークラスに移します。

(6) フロー制御　―　対象ソルバー、カプラーの実行

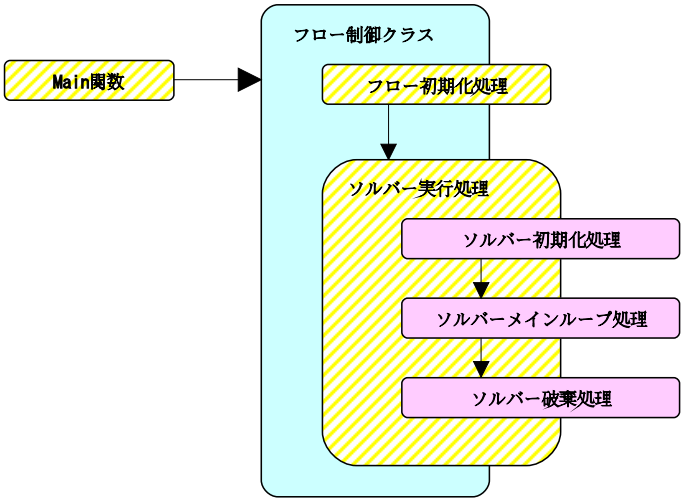
フロークラスに記述されているソルバー、カプラーの実行順に従い、ソルバー、カプラーを実行します。

１．６　フロー制御クラスの役割

インスタンスされた複数ソルバー、カプラー全体の制御を行います。

前述の”アプリケーションの Main 関数”から制御を受け取り、ソルバーの実行順番、カプラーの実行タイミングを制御します。

このフロー制御クラスをソルバー開発者が記述することにより柔軟なソルバーの実行を行うことができます。



(1) Main 関数から制御の移行

Main 関数から制御を受け取ります。

(2) フロー初期化処理

ソルバーを実行する上での動作条件を決定します。

フロー基底クラスでは、以下のパラメータを設定しています。

ソルバーループステップ最大回数

ソルバーステップ基準回数

ソルバーステップ現在回数

開発者は、その他の動作パラメータを必要とする場合は、パラメータの宣言、設定を初期化処理にて行う必要があります。

(3) ソルバー実行処理

複数のソルバーの初期化処理、メインループ処理、破棄処理を記述します。

(4) ソルバー初期化処理

インスタンスされた複数のソルバー、カプラーの初期処理を順次呼び出します。

開発者は、初期化順番の変更、開発者定義の初期化処理の呼び出しを記述することにより、フロー基底クラスに記述された初期化処理を拡張した処理を行うことができます。

(5) ソルバーメインループ処理

インスタンスされた複数のソルバー、カプラーのループステップを実行します。

開発者はソルバーの実行順番、カプラー実行タイミングを記述することにより柔軟なソルバー実行を行うことができます。

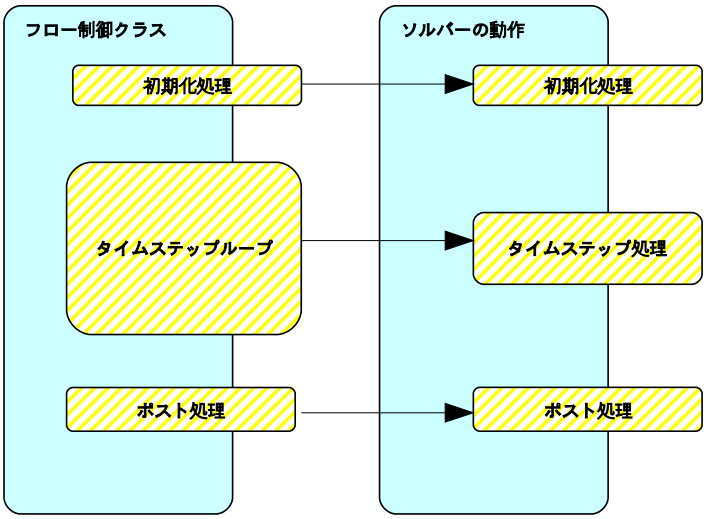
(6) ソルバー破棄処理

インスタンスされた複数のソルバー、カプラーの破棄処理を順次呼び出します。
開発者は、開発者定義の破棄処理の呼び出しを記述することにより、破棄処理を拡張することができます。

１．７　ソルバークラスの役割

ソルバクラスはタイムステップ毎の構造計算処理を記述するクラスです。

コンフィグレーションファイルのソルバー要素により対象となるソルバーがインスタンスされます。フロー制御クラスによりソルバークラスの各処理が呼び出されます。
本節では呼び出されたソルバークラスの各処理について説明します。



(1) 初期化処理

SPHERE では初期化処理を行うメソッドが用意されています。
開発者は用意されたメソッドの中身を記述します。
初期化処理では以下のような処理を開発者が記述する必要があります。

- 配列領域の確保
- コンフィグレーションファイルに記述された値のソルバー内変数への反映
- ファイル出力の初期化
- その他ソルバー固有の初期化処理　等...

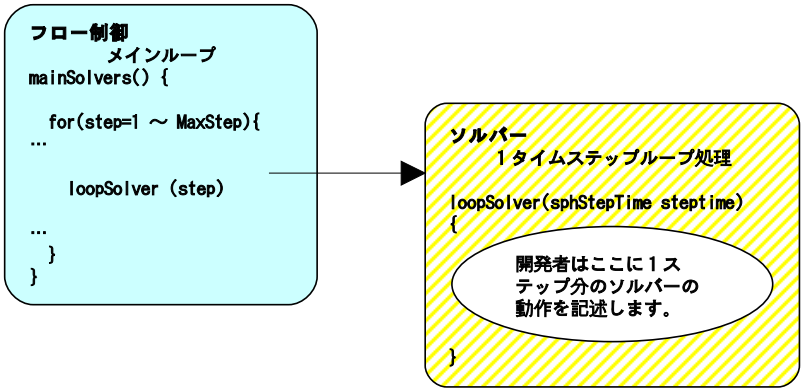
(2) タイムステップ処理

タイムステップ処理もスケルトン基底クラスにメソッドを用意しています。
フロー制御クラスにタイムステップループは繰り返し文（for 文）を用いて記述されていま

す。

ソルバは１ステップ毎にタイムステップ処理がフロー制御クラスから呼び出されます。

開発者は、１ステップ分のソルバーの動作を記述するメソッドにプログラムを記述します。



(3) 後処理

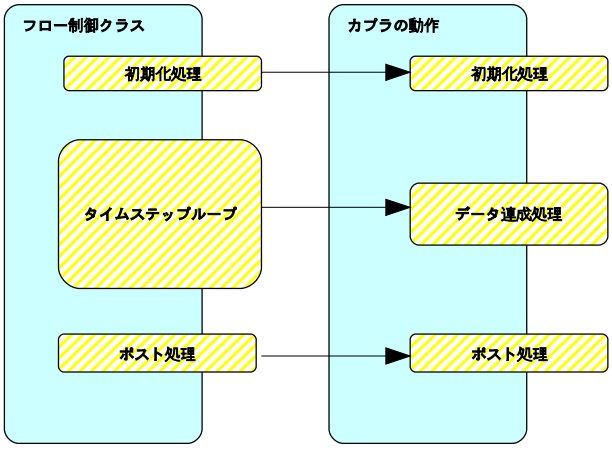
ソルバーにより処理の内容はさまざまだと思いますが、これもスケルトン基底クラスにメソッドが用意されていますので、必要に応じてコードを記述します。

１．８　カプラクラスの役割

カプラクラスは２つのソルバ間でデータ連成を行うクラスです。

コンフィグレーションファイルのカプラ要素により対象となるカプラがインスタンスされます。フロー制御クラスによりカプラクラスの各処理が呼び出されます。

本節では呼び出されたカプラクラスの各処理について説明します。



(1) 初期化処理

SPHERE では初期化処理を行うメソッドが用意されています。

開発者は基底クラスの初期化処理を呼び出すか、データ連成の為の初期化処理を記述する必要があります。

(2) データ連成処理

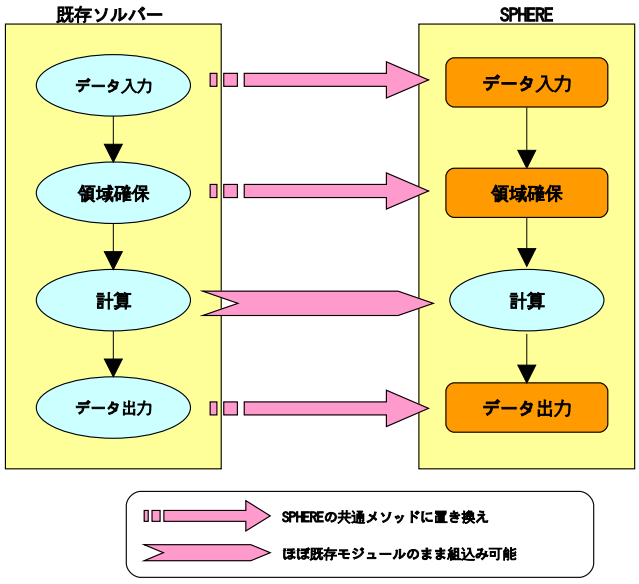
データ連成処理もスケルトン基底クラスにメソッドを用意しています。
基底クラスのデータ連成処理を呼び出すか、データ連成処理を記述する必要があります。

(3) 後処理

カプラにより処理の内容はさまざまだと思いますが、これもスケルトン基底クラスにメソッドが用意されていますので、必要に応じてコードを記述します。

１．９　既存ソルバーを SPHERE へ組込む作業

ソルバー基底クラスでは配列の動的確保、ファイル入出力といった共有メソッド（関数）が実装されていますので、ソルバーを組み込む際にはこれらの共有メソッドを使いつつ各ソルバーのモジュールを修正しながら組み込む必要があります。つまりソルバー本体（計算部分）は既存ソルバーのソースコードをクラスメソッド化してそのまま SPHERE に組込みますが、入出力に関してはインターフェイスの共通化を図る為に SPHERE で用意した共有メソッドに置き換える必要があります。また、物理量データを格納する配列等はダイナミックアロケーションにより領域を確保する形態を推奨するため、SPHERE はダイナミックアロケーション用のメソッドを用意しています。またプログラムの中で使用する変数の類もクラスのメンバ変数として実装する必要があります。



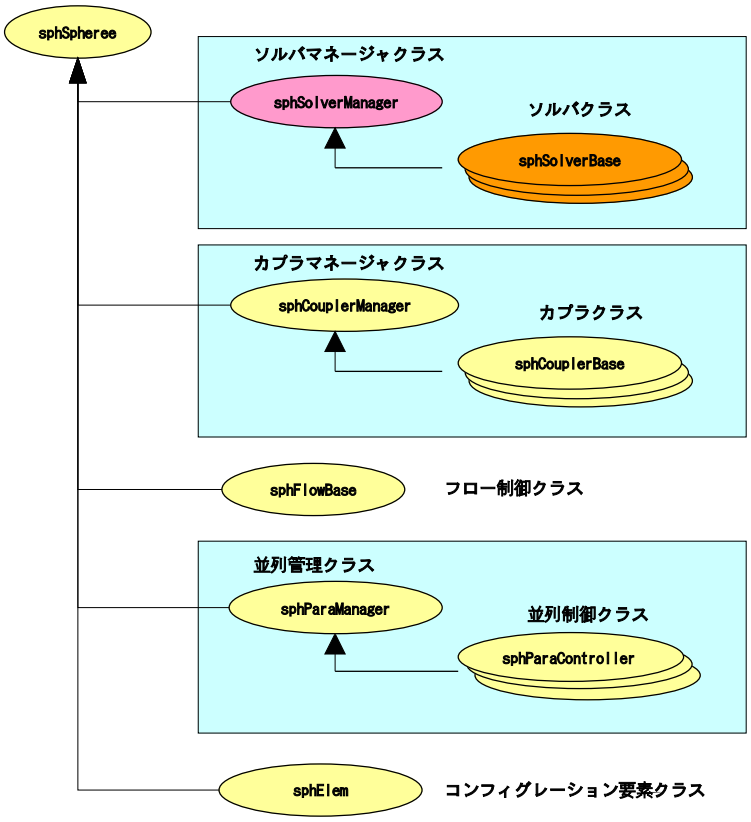
２．SPHERE のメカニズム

前節では SPHERE 及び SPHERE を用いたアプリケーションについて簡単にその構造を述べました。本節では SPHERE のクラスメソッドについて具体的に説明します。

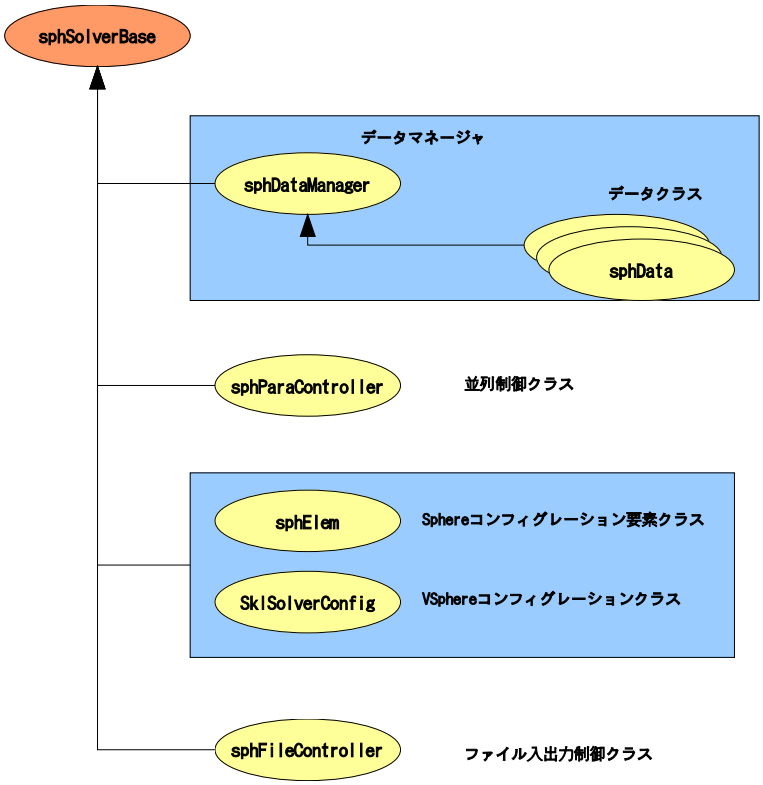
４．１　　SPHERE のクラス構造

SPHERE は以下のクラス構造をもちます。
これらのクラス群は”sph”という接頭辞を持つクラスになっています。

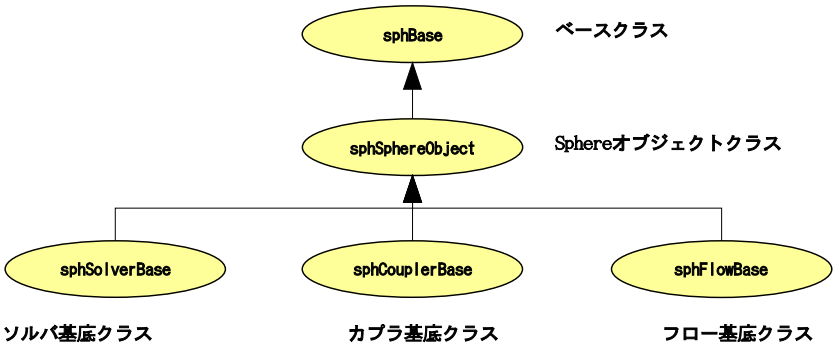
（SPHERE クラス構成）



（ソルバー構成）



（クラス階層）



ただし、開発者はクラス構成に示されたクラスすべての機能について知っている必要はありません。開発者がその機能を認識しなければいけないのは sphSolverBase クラスです。sphSolverBase クラスはその他のクラスが保持するデータとのインターフェイスを開発者に提供しているからです（ユーティリティクラス群は、ソルバークラス内で明示的にインスタンスして使用するため、個別に使用方法を知っておく必要があります）。

以下に各クラスの役割について説明します。

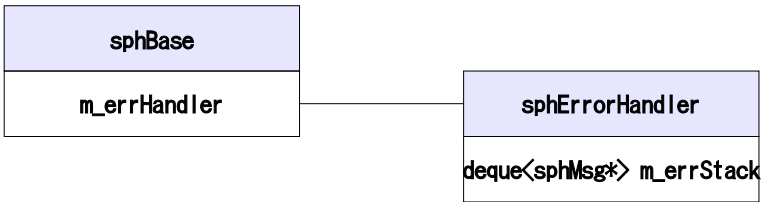
クラス名	役割
sphBase	SPHERE の全クラスのベースクラス エラーハンドリングを行う。
sphSphere	SPHERE のソルバー、カプラー等の管理クラスをメンバーに持ち、インスタンスされた全クラスの管理を行う最上位の管理クラスです。
sphSolverManager	ソルバー管理クラス インスタンスを行った複数のソルバーの管理を行うクラスです。
sphCouplerManager	カプラー管理クラス インスタンスを行った複数のカプラーの管理を行うクラスです。
sphParaManager	並列管理クラス ソルバー別に持つ並列制御クラスの管理を行います。
sphDataManager	データ管理クラス ソルバー毎に持ち、ソルバー内で生成したデータの管理を行います。
sphSphereObject	ソルバー、カプラー、フロークラスの基底クラス SPHERE の実行クラスの基底クラスで、ソルバーを実行する上での基本情報を持ちます。
sphSolverBase	SPHERE のソルバー基底クラス 開発者はこのクラスを派生させて、ソルバークラスを作ります。 SPHERE クラス群の中ではソルバー開発者にとって一番重要なクラスです。
sphCouplerBase	SPHERE のカプラー基底クラス ソルバー内のデータの連成を行うクラスです。 開発者はこのクラスを派生させて、カプラークラスを作り、データ連成の拡張を行うことができます。
sphFlow	フロー制御クラス基底クラス ソルバーの実行制御を行うクラスです。 開発者はこのクラスを派生させて、フロークラスを作り、柔軟なソルバーの実行を行うことができます。
sphData	データオブジェクト要素 ソルバーの計算結果を格納するデータオブジェクトです。
sphElem	コンフィグレーションクラス コンフィグレーションファイルの要素にアクセスを行うクラスです。
sphFileController	ファイルの入出力を行うクラスです。

３．SPHERE のクラス説明

以下、SPHERE のクラスについて説明します。

３．１　ベースクラス（sphBase）

SPHERE で使用しているすべてのクラスの基底クラスです。
エラーハンドリングを行います。



クラス名・メンバー名	説明
sphBase	SPHERE の全クラスのベースクラス エラーハンドリングを行う。
sphBase::m_errHandler	エラーハンドラークラスオブジェクト
sphErrorHandler	エラーハンドラークラス エラーメッセージのスタックを行う。
sphErrorHandler:: m_errStack	エラーメッセージスタック

SPHERE のエラーハンドリングは、エラーメッセージをスタックしエラー発生時、エラー発生箇所までの呼出トレースを行うことができます。

３．１．１　エラー出力マクロ・メソッド

エラー出力時、エラー発生ファイル、メソッド名、行番号を自動で出力する為に以下のマクロ・メソッドを用意しています。

（マクロ）

マクロ名	説明
SPH_ERROR(msg_no) SPH_ERROR(msg_no, msg_fmt, ...)	msg_no のエラー番号とエラー番号に対応したエラーメッセージ、付加エラー情報を出力します。 msg_no: エラー番号 "sphMessageDefine.h"に定義 msg_fmt: 付加エラー情報
SPH_WARNING(msg_no)	msg_no の警告番号と警告番号に対応した警告メッセー

マクロ名	説明
SPH_ WARNING (msg_no, msg_fmt, ...)	ジ、付加情報を出力します。 msg_no: 警告番号 "sphMessageDefine.h"に定義 msg_fmt: 付加情報

（メソッド）

メソッド名	説明
bool exists() const;	エラー・警告メッセージがスタックされているかチェックします。 true:エラー・警告メッセージが存在する。
bool existsError() const;	エラーメッセージがスタックされているかチェックします。 true:エラーメッセージが存在する。
bool existsWarning() const;	警告メッセージがスタックされているかチェックします。 true:警告メッセージが存在する。
void printStackTrace() const;	スタックされているエラー・警告メッセージを標準出力に出力します。
void clear();	スタックされているエラー・警告メッセージをすべて消去します。

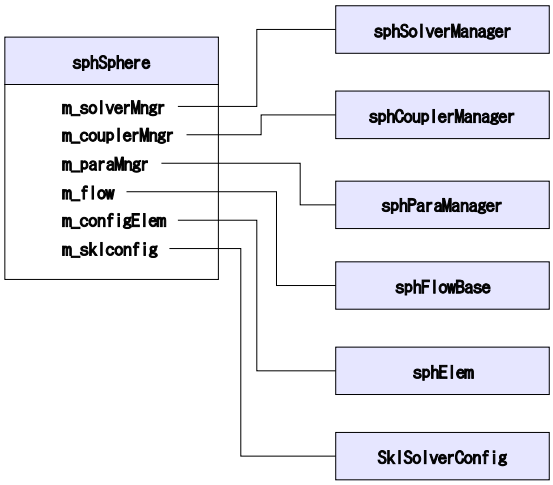
（使用例）

```
if ( (ret = initializeVoxel()) != SPHERE_SUCCESS ) {
   // error
   m_errHandler->SPH_ERROR_INFO(SPH_ERR_RETURN_FAULT, "initializeVoxel fault");
}
if (m_errHandler-> exists()) {
   m_errHandler->printStackTrace();
}
```

３．２　Sphere クラス（sphSphere）

SPHERE のソルバー、カプラー等の管理クラスをメンバーに持ちます。

インスタンスされた全クラスの管理を行う最上位の管理クラスです。

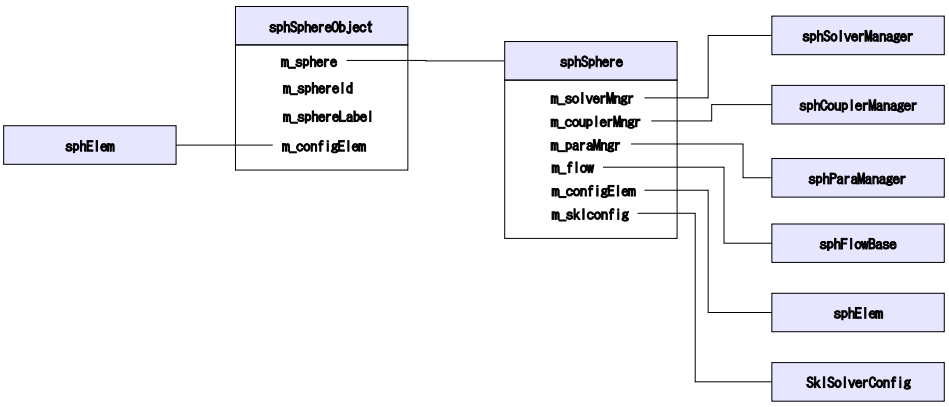


クラス名・メンバー名	説明
sphSphere	SPHERE のソルバー、カプラー等の全クラスをメンバーに持ちインスタンスされた全クラスの管理を行う最上位の管理クラスです。
sphSolverManager sphSphere::m_solverMngr	ソルバー管理クラス インスタンスを行った複数のソルバーの管理を行うクラスです。
sphCouplerManager sphSphere::m_couplerMngr	カプラー管理クラス インスタンスを行った複数のカプラーの管理を行うクラスです。
sphParaManager sphSphere::m_paraMngr	並列管理クラス ソルバー別に持つ並列制御クラスの管理を行います。
sphFlowBase sphSphere::m_flow	フロー制御クラス基底クラス ソルバーの実行制御を行うクラスです。
sphElem sphSphere::m_configElem	起動引数の Sphere(V200)形式のコンフィグレーションのルート要素を示す要素クラス。
SklCfg::SklSolverConfig sphSphere::m_sklconfig	起動引数の VSpher 形式のコンフィグレーションクラス。

Sphere クラスは、SPHERE 起動後に１つだけインスタンスされます。
Sphere オブジェクトクラス（sphSphereObject）には、インスタンスされた Sphere オブジェクトのポインタのみ渡されます。
よって、クラスメンバの"ソルバ管理クラス"、"カプラ管理クラス"、"並列管理クラス", "フロー制御クラス"、"コンフィグレーションクラス"はプロセスに１つだけ存在します。

３．３　Sphere オブジェクトクラス（sphSphereObject）

ソルバー、カプラー、フロークラスの基底クラス SPHERE の実行クラスの基底クラスです。ソルバーを実行する上での基本情報を持ちます。



クラス名・メンバー名	説明
sphSphereObject	ソルバー、カプラー、フロークラスの基底クラス SPHERE の実行クラスの基底クラスで、ソルバーを実行する上での基本情報を持ちます。
sphSphereObject::m_sphere	Sphere 管理クラスオブジェクト
sphSphereObject::m_sphereId	ソルバー、カプラー、フロークラスをインスタンスを行った時の識別 ID コンフィグレーションの識別 ID と同じとなります。
sphSphereObject::m_sphereLabel	ソルバー、カプラー、フロークラスをインスタンスを行った時の識別ラベル コンフィグレーションの識別ラベルと同じとなります。
sphElem sphSphereObject::m_configElem	ソルバー、カプラー、フロークラスの対応するコンフィグレーションの要素を示す要素クラス。

３．３．１　コンフィグレーション取得メソッド

インスタンスを行ったソルバー、カプラー、フロークラスのコンフィグレーションの要素を取得する為に以下のメソッドを用意しています。

メソッド名	説明
const sphElem*	ソルバー、カプラー、フロークラスの対応するコンフィグ

メソッド名	説明
getConfigElem() const;	レーションの要素を示す要素クラスを取得します。
int getSphereId() const;	ソルバー、カプラー、フロークラスをインスタンスを行った時の識別 ID（＝コンフィグレーションの識別 ID）を取得します。
std::string getSphereLabel() const;	ソルバー、カプラー、フロークラスをインスタンスを行った時の識別ラベル（＝コンフィグレーションの識別ラベル）を取得します。

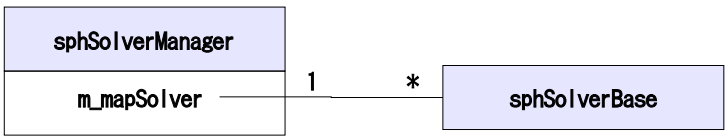
３．３．２　管理情報取得メソッド

Sphere クラス内の管理、制御クラスを取得する為に以下のメソッドを用意しています。

メソッド名	説明
sphSolverManager* getSolverManager();	ソルバ管理クラスを取得します。
sphCouplerManager* getCouplerManager();	カプラ管理クラスを取得します。
sphParaManager* getParaManager();	並列実行管理クラスを取得します。
const sphFlowBase* getFlow() const; sphFlowBase* getFlow();	フロークラスを取得します。

３．４　ソルバ管理クラス（sphSolverManager）

プロセス内でインスタンスされたソルバの管理を行います。



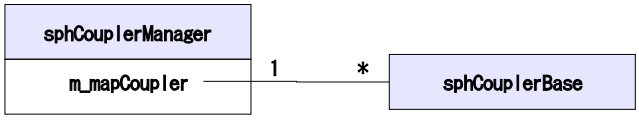
３．４．１　ソルバ取得メソッド

メソッド名	説明
int getSolverCount() const;	インスタンスされているソルバ数を取得します。

メソッド名	説明
sphSolverBase* getSolver(int id = 0) const;	インスタンスされれているソルバを取得します。getSolverCount()と対で使用します。 id:ソルバのリスト ID
sphSolverBase* getSolverById(int solverId) const;	インスタンスされれているソルバをコンフィグレーション識別 ID から検索・取得します。 solverId:識別 ID
sphSolverBase* getSolverByLabel(const char* solverLabel) const;	インスタンスされれているソルバをコンフィグレーション識別ラベルから検索・取得します。 solverId:識別ラベル

３．５　カプラ管理クラス（sphCouplerManager）

プロセス内でインスタンスされたカプラの管理を行います。

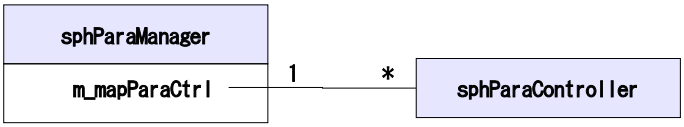


３．５．１　カプラ取得メソッド

メソッド名	説明
int getCouplerCount() const;	インスタンスされているカプラ数を取得します。
sphCouplerBase* getCoupler(int id = 0) const;	インスタンスされれているカプラを取得します。 getCouplerCount()と対で使用します。 id:カプラのリスト ID
sphCouplerBase* getCouplerById(int couplerId) const;	インスタンスされれているカプラをコンフィグレーション識別 ID から検索・取得します。 couplerId:識別 ID
sphCouplerBase* getCouplerByLabel(const char* couplerLabel) const;	インスタンスされれているカプラをコンフィグレーション識別ラベルから検索・取得します。 couplerId:識別ラベル

３．６　並列管理クラス（sphParaManager）

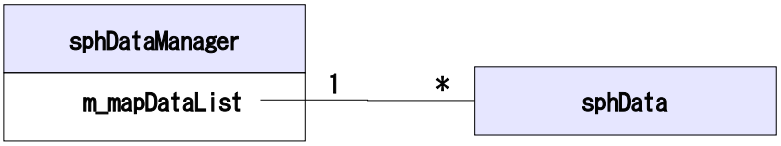
ソルバー別に持つ並列制御クラスの管理を行います。



詳細については「並列形態・並列制御クラスマニュアル」を参照してください。

３．７　データ管理クラス（sphDataManager）

ソルバー毎に持ち、ソルバー内で生成したデータの管理を行います。



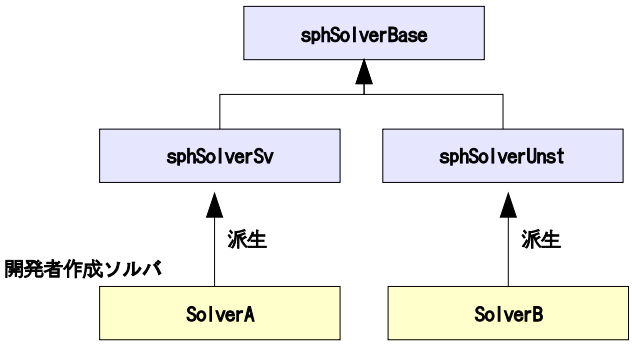
詳細については「データクラスマニュアル」を参照してください。

３．８　ソルバクラス（sphSolverBase）

本節では sphSolverBase クラスについて、その機能と役割について説明します。

３．８．１　ソルバクラスの役割

sphSolverBase クラスは、ソルバークラスの基底クラスになります。SPHERE はその下に構造格子用の sphSolverSv クラスを用意していますので、開発者は sphSolverSv クラスを派生させてソルバークラスを構築します。



上図の”sphSolvA”は sphSolverSv クラスから、”sphSolvB”は sphSolverUnst クラス

から、派生させた各ソルバークラスです。これにより、sphSolverBase, sphSolverSv、sphSolverUnst クラスで定義されているメソッド（関数）をソルバークラス内で使用することが可能になります。

sphSolverBase クラスには、ソルバーのコントロール（プログラムフロー）を規定したソルバー初期化、ソルバーの計算本体、ソルバーポスト処理といったメソッドを用意しています。これらのメソッドは開発者がそのメソッドの中身（プログラム）を自由に書くことができる形になっています。これらのメソッドをスケルトンメソッドと呼びます。また、メモリの確保、コンフィグレーションファイルに記述されたパラメータ値の取得といったメソッドを用意しています。これらのメソッドをスケルトンメソッドに対してライブラリメソッドと呼ぶこととします。

３．８．２　スケルトンメソッド

スケルトンメソッドとは、sphSolverBase クラスで定義されたソルバーのプログラムフローを制御するメソッド群です。sphSolverBase では（純粋）仮想関数として定義されており、sphSolverBase を派生させた各ソルバークラスで実体を持ちます。

ソルバースケルトンでは、これらのメソッドのスケルトンを用意しておりますので、開発者はこのスケルトンに組み込むソルバーに適したコード（プログラムコード）を記述します。以下はスケルトンメソッドの一覧とその説明です。

ソルバーのフローに関するスケルトンメソッド

ソルバーのフローに関するスケルトンメソッド	
①	int initializeSolver()
②	int loopSolver(unsigned int step)
③	int postSolver()

① int initializeSolver()

ソルバーの初期化メソッドです。開発者はこのメソッドに配列のメモリ確保、コンフィグレーションの反映、その他各ソルバー固有の初期化処理を記述します。

② int loopSolver(unsigned int step)

ソルバー本体です。開発者はこのメソッドに１タイムステップ分の動作を記述します。

③　　int postSolver()

ソルバーの後処理メソッドです。計算終了（タイムステップのループ終了）後の、各ソルバー固有の後処理を記述します。

３．８．３　メモリライブラリメソッド

配列の領域確保に関係するライブラリメソッドは以下です。

No	関数定義	サイズ	データ登録	説明
1	sphData * allocateDataObj （ const char *　cfg_label　）	paraCtrl	○	<SphDataObj>要素からデータ生成を行う。 サイズは並列制御クラスから取得し、自ノードサイズを確保する。
2	sphData *allocateDataObj （ const char *　reg_label, SphDCType　dcType, SPL_Datatype　dataType, const size_t　gc = 0, const size_t　data_num = 1, SphCrdDef crddef = CRDDEF_REGULAR ）	paraCtrl	○	データクラス、データ型を指定してデータ生成を行う。 サイズは並列制御クラスから取得し、自ノードサイズを確保する。
3	sphData * allocateDataObj （ SphDCType　dcType, SPL_Datatype　dataType, const size_t　gc = 0, const size_t　data_num = 1, SphCrdDef　crddef = CRDDEF_REGULAR ）	paraCtrl	×	データクラス、データ型を指定してデータ生成を行う。 サイズは並列制御クラスから取得し、自ノードサイズを確保する。
4	sphData　* allocateDataObjExcept (const char *　reg_label, SphDCType　dcType, SPL_Datatype　dataType, const size_t　size[3], const size_t　gc = 0,	引数指定	○	データクラス、データ型、サイズを指定してデータ生成を行う。 ガイドセルの通信は行わない。

No	関数定義	サイズ	データ登録	説明
	const size_t data_num = 1)			
5	sphData * allocateDataObjExcept(SphDCType dcType, SPL_Datatype dataType, const size_t size[3], const size_t gc = 0, const size_t data_num = 1)	引数指定	×	データクラス、データ型、サイズを指定してデータ生成を行う。 ガイドセルの通信は行わない。

サイズ：paraCtrl

並列制御クラスから取得し、自ノードサイズを確保する。

ガイドセルの通信を行うことができる。

引数指定

引数で指定したサイズにより領域を確保する。

ガイドセルの通信を行うことはできない。

データ登録：○ ＝ データ管理クラスに登録を行う。

作成したデータの管理はデータマネージャが行う。

開発者が管理、破棄を行う必要はない。

× ＝ データ管理クラスに登録を行わない。

作成したデータの管理は開発者が行う。

開発者が管理、破棄を行う。

allocateDataObj ライブラリメソッドは引数に従い領域を確保したデータクラスを返します。領域の確保に失敗した場合は、NULL を返します。

データ管理クラスに関係するライブラリメソッドは以下です。

No	関数定義	説明
1	sphData * getDataObj (const char * label)	生成済みのデータオブジェクトを取得する。 データオブジェクト生成時のラベルを指定します。
2	bool registerDataObj (const char * reg_label, sphData* dataObj)	データオブジェクトをデータ管理クラスに登録します。
3	bool deleteDataObj (const char * cfg_label)	データ管理クラスからラベルに指定されたデータオブジェクトを破棄します。

No	関数定義	説明
4	bool deleteDataObj （ sphData* dataObj)	データオブジェクトを破棄します。 データ管理クラスに登録されていれば、データ管理クラスからも破棄します。

deleteDataObj は allocateDataObj で領域を確保した配列に対して、その領域を開放するときに用います。SPHERE はアプリケーション終了時（ソルバークラスのデストラクタが実行される時）にデータ管理クラスに登録されたデータに対しては配列領域を自動的に消去します。

開発者が deleteDataObj ライブラリメソッドをデストラクタに記述して明示的に配列領域を開放する必要はありません。

しかし、データ管理クラスに登録されていないデータ、プログラムの実行中にメモリの節約等で必要の無くなったデータを消去したい場合に deleteDataObj ライブラリメソッドを用いることで開発者が明示的に配列領域を開放することができます。

（補足）

最新版の SPHERE ではここで説明した配列領域確保メソッドとは別に、「データクラス」と「データマネージャ」が実装されています。データクラスとは内部に配列の実体を持ち、その管理とファイル入出力機能を実装したクラス群です。

データクラス導入の主な理由は、並列実行時のノード間のデータ通信にかかるソルバー開発者のコード作成の簡便化にあります。

データクラス、データマネージャの詳細は別紙マニュアルをご参照ください。

３．８．４　ファイル入出力ライブラリメソッド

SPHERE には以下のデータフォーマットに対応したバイナリファイル（計算結果データやモデルデータ）の入出力用ライブラリメソッドが用意されています。

- SPHERE データフォーマット（SPH）
- SBX データフォーマット（SBX）
- SPX データフォーマット（SPX）

ファイル入出力クラスに関係するライブラリメソッドは以下です。

No	関数定義	出力 ステップ・時刻	説明

No	関数定義	出力 ステップ・時刻	説明
1	sphData * allocateDataObj （ const char * cfg_label ）	なし	コンフィグレーションの<SphDataObj>要素配下に<SphFile>要素が記述されている場合、記述ファイルからデータを読み込み、データの生成を行う。
2	sphData * loadFile （ const char* file_label, const size_t size[3], const size_t start_idx[3], size_t gc ）	なし	ファイル識別ラベルから指定サイズ、始点インデックス、ガイドセルにてファイルを読込み、データクラス生成を行う。
3	bool writeFile （ const char * data_cfg_label ）	getCurrentStep getCurrentTime	コンフィグレーションの<SphDataObj>要素配下に<SphFile>要素が記述されている場合、記述ファイルにデータラベルのデータをファイル出力する。
4	bool writeFile （ const char * file_label, const char * data_reg_label ）	getCurrentStep getCurrentTime	ファイル識別ラベルに従ってデータラベルのデータをファイル出力する。
5	bool writeFile （ const char * file_label, sphData * dataObj ）	getCurrentStep getCurrentTime	ファイル識別ラベルに従ってデータをファイル出力する。
6	bool writeFile (const char* data_cfg_label, size_t step, double time, bool gc_flag = false, bool force = false);	引数指定	コンフィグレーションの<SphDataObj>要素配下に<SphFile>要素が記述されている場合、記述ファイルにデータラベルのデータをファイル出力する。
7	bool writeFile(const char* file_label, sphData* dataObj, size_t step, double time, bool gc_flag = false, bool force = false);	引数指定	ファイル識別ラベルに従ってデータをファイル出力する。

出力ステップ・時間

getCurrentStep・getCurrentTime

現在のステップ数、時間を取得して自動出力します。

引数指定

引数で指定されたステップ数、時間を出力します。

ソルバー開発者は SPHERE が用意したデータ入出力用ライブラリメソッドを利用してファイルの読み込みや書き出しを行うことで、ファイル入出力に関するプログラムコード作成の手間を省略することが可能です。(バイナリファイルの入出力ライブラリメソッドに関する詳細はファイル入出力マニュアルを参照ください。)

３．８．５　並列制御クラスの生成ライブラリメソッド

並列制御クラスを生成、初期化を行うライブラリメソッドとして以下があります。

No	関数定義	説明
1	int initializeVoxel （ const size_t　voxelsize[3], int　proc_num = 0, const double　voxel_origin[3] = NULL, const double　voxel_pitch[3] = NULL	並列制御クラスの生成を行う。 分割プロセス数からボクセルサイズを自動分割し、並列制御クラスの生成を行う。
2	int initializeVoxel （ const size_t　voxelsize[3], const unsigned int　division[3], const double　voxel_origin[3] = NULL, const double　voxel_pitch[3] = NULL ）	並列制御クラスの生成を行う。 i,j,k の分割数の情報からボクセルサイズを自動分割し、並列制御クラスの生成を行う。
3	int initializeVoxel （ const sphElem *　nodelist_elem, const double　voxel_origin[3] = NULL, const double　voxel_pitch[3] = NULL ）	並列制御クラスの生成を行う。 ノードリスト要素から並列制御クラスの生成を行う。

詳細については「並列形態・並列制御クラスマニュアル」を参照してください。

４．SPHERE へのソルバーの組込み方法

本節では、既存のソルバープログラムを SPHERE に組み込む方法について説明します。組込み対象となるソルバープログラムは C、C++もしくは Fortran 言語で記述されたプロ

グラムです。

まず、SPHERE に既存のソルバープログラムを組み込む前に、SPHERE への組込みを容易にするために既存ソルバープログラムを修正する必要があります。これは「SPHERE に組み込む前の準備」で説明します。次に「SPHERE のプログラム構成」として、SPHERE のプログラム（ファイル）構成について説明します。この説明の後、実際にソルバープログラムを SPHERE に組み込む方法について説明をします。

４．１　SPHERE に組み込む前の準備

既存のソルバープログラムを SPHERE に組み込む際の前準備として、SPHERE に組込み易い形に既存のソルバープログラムを修正する必要があります（ここに記述する方法でプログラムを修正してから組み込んだ方が組み込み作業が容易になります)。

SPHERE は C++言語を用いてクラス群として実装されています。これに対してソルバープログラムは C、C++もしくは Fortran 言語、あるいはそれら言語の混在環境で構築されているかもしれません。また、SPHERE では物理量データ等を保持する配列の領域は動的に確保することを推奨しています。メモリの動的確保に加えて、配列は基本的に一次元配列となります。これらの制約に対して、SPHERE にコードを組み込みながら対処していくのは非常に困難です。このため、SPHERE では既存ソルバーのプログラムコードを本節で説明する方法で修正することを推奨しています。

４．１．１　どんな修正を必要とするか

既存ソルバーが C もしくは C++言語で記述されており、かつ配列領域が動的に確保されているプログラムであればコードに対して何も修正する必要はありません。修正が必要である可能性があるのはソルバープログラムが Fortran 言語で記述されているプログラムです。このようなプログラムには、以下のような修正を行います。

- Fortran 言語の common を使用しない
- 関数呼び出しにおいて配列データは引数渡しにする
- Fortran 言語で静的に宣言されている配列をなくす
- メインルーチンを C++言語に書き換える

上記のプログラム言語の仕様に係る変更を行うことで、SPHERE へのコード組込みが非常に容易になります。

また、SPHERE はソルバーのアルゴリズムを規定しています。ソルバーは「初期化部」、「タイムステップループ部」、「後処理部」が明確に分かれていないといけません。つまり

ソルバーのプログラムフローについて修正を行う必要があるわけです。

次節では、具体的にプログラムをどのように修正するのかを説明していきます。

４．１．２　Fortran 言語で記述されたコードの修正

前節の「コードに対して行う修正」において Fortran 言語に関するものが３項目ありました。

- Fortran 言語の common を使用しない
- 関数呼び出しにおいて配列データは引数渡しにする
- Fortran 言語で静的に宣言されている配列をなくす

これらの修正がどのような場合に必要となるか、例をあげて説明します。

典型的な例として次ページのようなプログラム（プログラム１）があったとします。

このプログラムは array.h 内で配列長（imax, jmax, kmax）と配列の実体（p）を common を用いて宣言しており、これらの変数をサブルーチン setval, minmax で使用しています。配列 p は静的に領域を確保された配列になっています。

（プログラム１）

array.h

```
      parameter (imax=64, jmax=64, kmax=32)
      common / prs / p(imax, jmax, kmax)
```

func.f

```
      subroutine func()
c
      real dmin, dmax
c
      call setval()
      call minmax(dmin, dmax)
      write(*,*) 'data min : ', dmin
      write(*,*) 'data max : ', dmax
c
      return
      end
```

setval.f

```
      subroutine setval()
c
      include 'array.h'
c
      do k=1,kmax
      do j=1,jmax
      do i=1,imax
```

```
            index = imax*jmax*k+imax*j+i-imax*jmax-imax
            p(i,j,k) = index
        enddo
        enddo
        enddo
c
        return
        end
```

minmax.f

```
        subroutine minmax(dmin, dmax)
c
        include 'array.h'
        real dmin, dmax
c
        dmin = p(1,1,1);
        dmax = p(1,1,1);
c
        do k=1,kmax
        do j=1,jmax
        do i=1,imax
           if( dmin .gt. p(i,j,k) ) dmin = p(i,j,k)
           if( dmax .lt. p(i,j,k) ) dmax = p(i,j,k)
        enddo
        enddo
        enddo
c
        return
        end
```

まずは、「Fortran 言語の common を使用しない」かつ「関数呼び出しにおいて配列データは引数渡しにする」形にプログラムを変更します。

（プログラム２）

func.f

```
        subroutine func()
c
        parameter (imax=64, jmax=64, kmax=32)
        real dmin, dmax
        real p
        dimension p(imax, jmax, kmax)
c
        call setval(imax, jmax, kmax, p)
        call get_minmax(imax, jmax, kmax, p, dmin, dmax)
        write(*,*) 'data min : ', dmin
        write(*,*) 'data max : ', dmax
c
        return
        end
```

setval.f

```
      subroutine setval(imax, jmax, kmax, p)
c
      integer imax, jmax, kmax
      real p
      dimension p(imax, jmax, kmax)
c
      do k=1,kmax
      do j=1,jmax
      do i=1,imax
        index = imax*jmax*k+imax*j+i-imax*jmax-imax
        p(i,j,k) = index
      enddo
      enddo
      enddo
c
      return
      end
```

minmax.f

```
      subroutine get_minmax(imax, jmax, kmax, p, dmin, dmax)
c
      integer imax, jmax, kmax
      real dmin, dmax
      real p
      dimension p(imax, jmax, kmax)
c
      dmin = p(1,1,1);
      dmax = p(1,1,1);
c
      do k=1,kmax
      do j=1,jmax
      do i=1,imax
        if( dmin .gt. p(i,j,k) ) dmin = p(i,j,k)
        if( dmax .lt. p(i,j,k) ) dmax = p(i,j,k)
      enddo
      enddo
      enddo
c
      return
      end
```

プログラム２では array.h において common 宣言されていた静的配列 p と配列長を示す imax, jmax, kmax が無くなり、かわりにサブルーチン func において宣言されています。プログラム２では「Fortran 言語の common を使用しない」かつ「関数呼び出しにおいて配列データは引数渡しにする」という条件をクリアできました。しかし、サブルーチン func はメインルーチンではなくサブルーチンですので、サブルーチン func 内で宣言した配列 p

もサブルーチン func の引数として渡されなければなりません。ここで、メインルーチンを示します。メインルーチンの中で配列 p や配列長 imax, jmax, kmax を宣言し、サブルーチン func の引数としてこれらを渡すことで、完全に「Fortran 言語の common を使用しない」かつ「関数呼び出しにおいて配列データは引数渡しにする」という条件を満たしたことになります。

（プログラム３）

main.f

```
      program main
c
      parameter (imax=64, jmax=64, kmax=32)
      real p
      dimension p(imax, jmax, kmax)
c
      call func(imax, jmax, kmax, p)
c
      stop
      end
```

func.f

```
      subroutine func(imax, jmax, kmax, p)
c
      real p
      dimension p(imax, jmax, kmax)
c
      real dmin, dmax
c
      call setval(imax, jmax, kmax, p)
      call get_minmax(imax, jmax, kmax, p, dmin, dmax)
      write(*,*) 'data min : ', dmin
      write(*,*) 'data max : ', dmax
c
      return
      end
```

次に最後の条件「Fortran 言語で静的に宣言されている配列をなくす」作業に入ります。プログラム３ではメインルーチンに配列 p を静的に定義しています。これをダイナミックアロケーションを行う動的配列にする必要があります。Fortran90 以降の Fortran 言語であれば allocate 命令を用いればよいわけですが、メインルーチンを C++言語に変更するという修正が必要ですので、メインルーチンの C++言語化と一緒に説明します。

４．１．３　メインルーチンをＣ言語に書き換える

前節では「Fortran 言語で静的に宣言されている配列をなくす」作業を行っていませんでした。前節の最後に示したプログラム３のメインルーチンをもう一度見てみます。

main.f

```
      program main
c
      parameter (imax=64, jmax=64, kmax=32)
      real p
      dimension p(imax, jmax, kmax)
c
      call func(imax, jmax, kmax, p)
c
      stop
      end
```

このメインルーチンでは、配列 p が静的に定義されています。ここではこのメインルーチンを C++言語に直し、かつ配列 p をダイナミックアロケーションを行う動的配列にします。

main.C

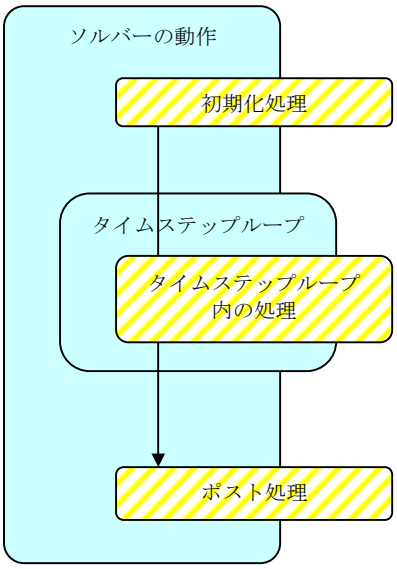
```
extern "C" {
  void func_(int* imax, int* jmax, int* kmax, float* p);
};

int main()
{
  int imax, jmax, kmax;
  float* p;

  imax = 64; jmax = 64; kmax = 32;
  p = new float[imax*jmax*kmax];

  func_(&imax, &jmax, &kmax, p);

  return 0;
}
```

以上で、プログラム言語に係る修正が終了しました。

次節ではソルバーのプログラムフローの修正について説明します。

４．１．４　プログラムフローの修正

SPHERE はソルバーのアルゴリズムを規定しています。ソルバーは「初期化部」、「タイムステップループ部」、「後処理部」が明確に分かれていないといけません。SPHERE の概念の節でもふれましたが、ソルバースケルトンでは「初期化処理」、「タイムステップループ」、「後処理」がメソッドとして独立しています（下図）。



このため、メインルーチンを下のプログラムの様に明確に分割しておく必要があります。

main.C

```
int main()
{
        初期化部（変数の宣言、初期化、領域確保、パラメータ読込み等）
        タイムステップループ部
        ポスト処理部
    return 0;
}
```

以上でソルバーを SPHERE に組み込む前のソルバープログラムの修正は終わりです。

４．２　　ソルバークラスの雛形作成

ソルバーのプログラム修正が終了した後、ソルバーコードを SPHERE に組み込んでいくわけですが、組み込むためのソルバークラスを作成する必要があります。

作成雛形ファイルは以下です。

フォルダ	ファイル名	対応クラス	説明
app	sphCouplerFactory.C	sphCouplerFactory	カプラクラスの生成を行います。
	sphFlowFactory.C	sphFlowFactory	フロー制御クラスの生成を行います。
	sphSolverFactory.C	sphSolverFactory	ソルバクラスの生成を行います。
solverFluid	sphSolverFluid.C	sphSolverFluid	開発者が実装を行うソルバクラスのソースファイルです。
	sphSolverFluid.h		開発者が実装を行うソルバクラスのヘッダファイルです。
	sphSolverFluidDefine.h		開発者が実装を行うソルバクラスの定義ファイルです。
flowFluid	sphFlowFluid.C	sphFlowFluid	開発者が実装を行うフロー制御クラスです。
	sphFlowFluid.h		
couplerFluid	sphCouplerFluid.C	sphCouplerFluid	開発者が実装を行うカプラクラスです。
	sphCouplerFluid.h		

４．３　　ソルバークラスへのソルバーコードの組込み

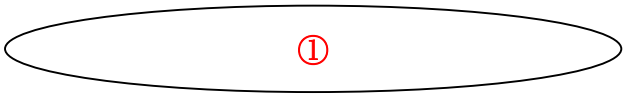
前節で、プロジェクト管理ツールによりソルバークラスの雛形を作成しました。ここではソルバープログラムをこのソルバークラスの雛形に組み込む方法について説明します。

プロジェクト管理ツールにて"Fluid"ソルバーを定義し、sphSolverFluid クラスを作成した場合を例に以下に説明します。

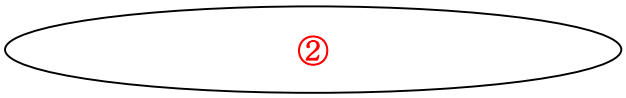
４．３．１　sphSolverFluid クラスのクラス定義ファイルの修正

"sphSolverFluid.h"は sphSolverFluid クラスのクラス定義ファイルとなります。

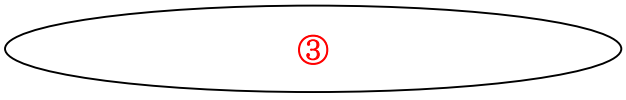
```
#ifndef _SPH_SOLVER_Fluid_CLASS_H_
#define _SPH_SOLVER_Fluid_CLASS_H_

#include "sphSolverSv.h"
#include "sphSolverFluidDefine.h"

/* USER WRITE
    Please write include file here.
*/
```




```
class sphSolverFluid : public SklSolverSv {
public:
  sphSolverFluid(const char* label);
  sphSolverFluid(const char* label, int id);
  virtual ~sphSolverFluid();

  virtual int initializeSolver(sphStepTime* steptime);
  virtual int loopSolver(sphStepTime* steptime);
  virtual int postSolver(sphStepTime* steptime);
  virtual int printSolverUsage(const char* cmd);

public:
/* USER WRITE
    Please write member variable here.
*/
```




```
/* USER WRITE
    Please write class method here.
*/
```




```
};

#endif // _SPH_SOLVER_Fluid_CLASS_H_
```

プログラム中の①、②、③部分に必要なコードを追加します。

①	大域変数やインクルードファイルを記述する
②	メンバ変数を記述する
③	メンバ関数を記述する

①部分には、大域変数やインクルードファイルを記述します。ここで注意しなければいけないことはクラスに属するメンバ変数と違い、この部分に定義する変数は大域変数になるということです。名前空間の衝突をさけるため、唯一と思われる変数名を使用してください。他のソルバークラスのファイル等で同じ名前の変数を使用しているとコンパイル時にエラーになってしまいます。
②部分には sphSolverFluid クラスに属するメンバ変数を定義します。
また、③部分には sphSolverFluid クラスのメンバ関数（メソッド）を宣言します。③部分に追加するメソッドについては「その他のメソッドの追加」を参照してください。

４．３．２　コンストラクタとデストラクタの修正

sphSolverFluid.C ファイル内にはコンストラクタとデストラクタの実体が記述されています。コンストラクタには前節で定義したメンバ変数の初期化コードを記述し、デストラクタにはクラス破壊時の必要な処理を記述します。

```
sphSolverFluid::sphSolverFluid(const char* label) : sphSolverSv(label) {
/* USER WRITE
    Please place initialize member variable here.
*/
                コンストラクタ記述部

}

sphSolverFluid::sphSolverFluid (const char* label, int id) : sphSolverSv(label, id)

/* USER WRITE
    Please place initialize member variable here.
*/
                コンストラクタ記述部

}

sphSolverFluid::~sphSolverFluid() {
/* USER WRITE
    Please place destory member variable here.
*/
                デストラクタ記述部

}
```

４．３．３　ソルバー初期化メソッドの修正

ソルバー初期化メソッドは initializeSolver メソッドです。
このメソッドはコンフィグレーションからの値の取得や配列の領域確保といったコードを記述します。

```
#include "sphSolverFluid.h"

int sphSolverFluid::initializeSolver(sphStepTime* steptime)
{

  // デフォルトのソルバーの初期化
  // ソルバー固有の処理を行う場合は、以下の行の削除、又は処理の追加してください。
  if (sphSolverSv::initializeSolver(steptime) != SPHERE_SUCCESS) return false;

/* USER WRITE
    Please write solver initialize code.
*/
```

初期化記述部

```
  return 1;
}


int sphSolverFluid::initializeVoxel()
{
/**
     コンフィグレーションのSphDomainInfo要素からボクセル情報を
     自動生成しない場合は、以下を削除してボクセル情報の初期化コードを
     記述してください。
*/
    return sphSolverSv:: initializeVoxel();


}
```

sphSolverSv クラスの initializeVoxel()はボクセルサイズをコンフィグレーションの SphDomainInfo 要素から取得して、ノード分割、並列制御クラスを自動生成を行います。
開発者がボクセルサイズを指定してボクセルサイズを初期化する場合は、以下のコードを削除して初期化コードを記述してください。

"return sphSolverSv:: initializeVoxel();"　　（削除）

ボクセルサイズの初期化には、前述の”並列制御クラスの生成ライブラリメソッド”を用意していますので、サイズの取得後これらのメソッドを使用してノード分割、並列制御

クラスの生成を行ってください。

４．３．４　１タイムステップの処理メソッドの修正

ソルバーの１タイムステップの処理を記述するメソッドは loopSolver メソッドです。このメソッドにはソルバーの１タイムステップに行う処理を記述します。

```
#include "sphSolverFluid.h"

int sphSolverFluid::loopSolver(sphStepTime* steptime)
{
/* USER WRITE
    Please write solver main code.
*/

    コード記述部

  return 1;
}
```

loopSolver メソッドの引数 step は現在のステップ数が渡ってきますので、必要に応じてこの変数を loopSolver メソッド内で使用することができます。

４．３．５　ソルバー後処理メソッドの修正

ソルバー後処理メソッドは postSolver メソッドです。

このメソッドにはタイムステップループの終了後のソルバー後処理を記述します。

```
#include "sphSolverFluid.h"

int sphSolverFluid::postSolver(sphStepTime* steptime)
{
/* USER WRITE
    Please write solver post code.
*/

    後処理コード記述部

  return 1;
}
```

４．３．６　Fortran サブルーチン及び関数宣言の追加

ソルバー内で Fortran のサブルーチンまたは関数を用いている場合、これらはクラスメソッドとして定義できません。これらは FortranFuncFluid.h ファイル内で C の名前空間を用いた関数として宣言します。このとき、他ソルバーとの名前の衝突に注意してください。

```
#ifndef _SPH_FORTRAN_FUNC_Fluid_H_
#define _SPH_FORTRAN_FUNC_Fluid__H_
#include "Skl.h"

extern "C" {
/* USER WRITE
    if using fortran subroutine,
    Please write fortran subroutine here.
*/
```

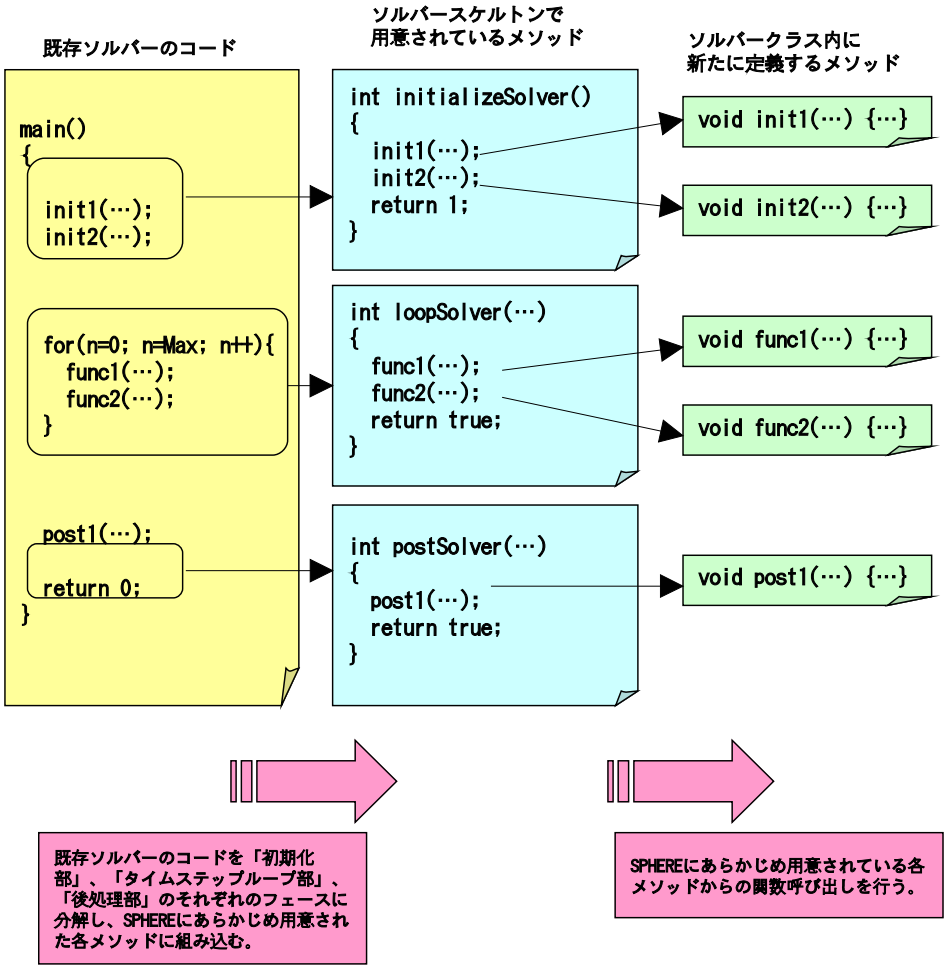
Fortran 関数の宣言記述部

```
}

#endif // _SPH_FORTRAN_FUNC_Fluid_H_
```

４．３．７　その他のメソッドの追加

SPHERE に実装されていない関数を追加する場合には、「sphSolverFluid クラスのクラス定義」節内の③部分に記述します。ここではどのような関数がそれにあたるのか、またそのように関数を追加するのかについて説明します。

ソルバープログラムは下図のような構成になっているとします。



この場合、上図の右側にある関数（init1, init2, func1, func2, post1）をソルバークラスのメソッドとして定義します。これらの関数をメンバ関数（クラスメソッド）として定義することでクラス内で定義したメンバ変数に自由にアクセス可能となり、処理に必要な変数を関数の引数として受け渡す必要がなくなります。ただし、Fortran 関数はクラス内で定義できませんので、前節で説明したように FortranFuncFluid.h に記述する必要があります。

４．３．８　新規ファイルのソルバークラスへの追加

ソルバーが独自にファイルを作成した場合、そのままではプロジェクトがそのファイルを認識することはできません。

５．プロジェクトのコンパイル

ソルバーの組み込みが完了した後は、コンパイルを行い SPHERE の実行モジュール”sphere”を作成します。

前節で説明したとおり、SPHERE はソルバークラスやコンパイル環境を「プロジェクト」で管理しています。プロジェクト管理ツールを用いて作成されたプロジェクトはプロジェクトディレクトリ単位で管理されます。このプロジェクトディレクトリ直下にプロジェクト全体をコンパイルするための Makefile が用意されています。

コンパイルを行う際は、プロジェクトディレクトリ直下の Makefile を用いて make してください。