Formura

チュートリアル

Verseion 1.1.0

2019 年 3 月 29 日

変更履歴

Ver.	更新日時	説明
1.0.0	2019/3/15	初版発行
1.1.0	2019/3/29	第 2 版発行

目次

1 Formura の概要 1

2 環境構築 1
- 2.1 Haskell の環境構築 . . . 1
- 2.2 Formura のインストール . . . 2
- 2.3 可視化に必要なライブラリ等 . . . 2

3 Formura を用いたプログラミングのワークフロー 2

4 Formura の基本機能 3

5 Formura の最小構成 4
- 5.1 Formura 最小プログラム . . . 4
- 5.2 Formura 最小設定 . . . 5
- 5.3 最小ドライバコード . . . 5

6 サンプルコード (1 次元オイラー方程式) 6
- 6.1 次元および軸名の指定 . . . 7
- 6.2 グローバル変数宣言 . . . 7
- 6.3 外部関数宣言 . . . 8
- 6.4 init 関数 . . . 8
- 6.5 step 関数 . . . 10
- 6.6 min, max 関数 . . . 11
- 6.7 数値微分 . . . 11
- 6.8 タプル . . . 12
- 6.9 その他の関数 . . . 12
- 6.10 設定ファイル . . . 12
- 6.11 ドライバコード . . . 13
- 6.12 コンパイル・実行 . . . 15
- 6.13 可視化 . . . 16

7 サンプルコード (3 次元オイラー方程式) 16

7.1 次元および軸名の指定 17
7.2 前進後進演算子と微分演算子 17
7.3 ナブラ演算子と発散 18
7.4 タプルの回転と 3 次元フラックス計算 18
7.5 設定ファイル 20
7.6 ドライバコード 20
7.7 コンパイル・実行・可視化 20

8 サンプルコード（1 次元 MHD 方程式） 20
8.1 テンポラルブロッキング 21

9 サンプルコード (3 次元 MHD 方程式) 22

付録 A Formura の API 一覧 22

付録 B Formura の言語仕様 24
B.1 文字 24
B.2 識別子 24
B.3 プログラム 25
B.4 文 25
B.5 型 25
B.6 式 26
B.7 演算子 28
B.8 左辺式 29

1 Formura の概要

Formura は，ステンシル計算専用のプログラミング言語である．Formura を用いると，例えば流体力学の問題を解くプログラムなどが簡単に記述できる．

Formura ユーザーはまず，Formura ソースコードを作成する．このソースコードを Formura コンパイラで処理することで，ステンシル計算を実行するための C 言語のコードを得る．生成された C 言語コードには計算空間の状態を更新するための関数が含まれている．この状態更新関数を呼び出す main 関数をユーザー側で作成し，Formura が生成した C コードとリンクすることで，ステンシル計算の実行ファイルを得ることができる．

2 環境構築

Formura は Linux 上で動作する．Ubuntu 18.04 LTS 上にインストールする前提で環境構築方法の解説を行う．Formura のビルドと実行および，Formura を使って生成されたコードのコンパイル・実行には以下のソフトウェアが必要である．

- Haskell Tool Stack
- C 言語がコンパイル可能なコンパイラ (GCC, Clang など)
- MPI 環境 (OpenMPI，MPICH など)

このサンプルコードに付属している C 言語コードは，C11 規格を前提にしているためコンパイラが C11 に対応している必要がある．

Ubuntu18.04 で必要なソフトウェアをインストールするためには以下のコマンドを入力する．

```
$ sudo apt install build-essential git mpi-default-dev libtinfo-dev
```

2.1 Haskell の環境構築

Formura のビルドに必要な Haskell Tool Stack をインストールする．

```
$ wget -qO- https://get.haskellstack.org/ | sh
```

2.2 Formura のインストール

適当なディレクトリ上で Formura の github レポジトリをクローンして，Formura をビルドする．

```
$ git clone https://github.com/formura/formura.git
$ cd formura
$ stack install
```

stack のデフォルト動作では Formura の実行ファイルは `~/.local/bin` にインストールされるため，これをターミナルの環境変数 `PATH` に加える．

以上で Formura のインストールは完了する．

2.3 可視化に必要なライブラリ等

サンプルコードに付属している可視化スクリプトは Python3 で記述されている．可視化スクリプトの実行には以下に示す Python ライブラリが必要である．

- Matplotlib
- NumPy
- PyYAML

例として，`apt` を用いてインストールする方法を示す．

```
$ sudo apt install python3-yaml python3-numpy python3-matplotlib
```

この方法以外にも，`pip` を用いる方法や，Anaconda 等のプラットフォームを使用する方法もあるので，各自の環境に合わせてインストールすること．

3 Formura を用いたプログラミングのワークフロー

1. Formura ソースコード・設定ファイルの作成
   ステンシル計算を記述する`*.fmr` ファイルと，格子のグリッド数，MPI ノード数等を設定する`*.yaml` ファイルを作成する．
2. Formura ソースコードのコンパイル

`formura` コマンドを実行して，ステンシル計算を行う C 言語コードを作成する．このソースコードには格子の状態を初期化・更新するための関数が含まれている．

3. FormuraAPI を呼び出す C 言語コードの作成
   Formura が生成した C 言語コードを呼び出すための main 関数および，ファイル入出力関数を記述する．
4. C 言語コードのコンパイル
   `mpicc` を用いて C 言語コード類をコンパイル＆リンクする．
5. 実行・結果の確認
   生成されたバイナリを `run` コマンドで実行する．`run` は `formura` が生成し，内部で `mpirun` を呼び出す．実行結果は gnuplot や Python の Matplotlib などを用いて可視化する．

4 Formura の基本機能

Formura は，ステンシル計算に適した文法を持つ言語である．Formura の最も基本的な文法と機能を以下に示す．

- Formura のプログラムは複数の文からなり，文と文は区切り文字で隔てられている．区切り文字はセミコロン (;) または改行を用いることができる．
- Formura の変数には型を指定することができる．指定できる型は，整数型や浮動小数点数などの基本的な型に加えて，グリッド型やタプル型などがある．
- グリッド型の変数は計算空間中のすべての格子点で値を持つ「場」を表現できる．
- グリッド型の変数にはインデックスのオフセットを指定することができる．オフセットをもたせた変数は，格子点の中間で定義された値を表現することができる．
- Formura のグリッド変数同士の演算では，インデックスを相対的にずらした上で演算を行うことができる．インデックスをシフトしたグリッド変数同士の演算が，ステンシル計算を記述する主な記法になる．
- 複数の型を組み合わせてタプル型を作ることができる．タプル型は C 言語の構造体と似た概念であり，複数の変数を一つにまとめることができる．
- Formura では自作の関数を定義することができる．
- Formura では C 言語の関数を呼び出すことができる．
- Formura のグリッドは周期的境界条件を持つ．

より詳しい解説は付録 B に示す.

5 Formura の最小構成

5.1 Formura 最小プログラム

Formura でコンパイルが通る最小限のコードを用いて，Formura の基本設定について解説する.

Formura の最小コードを，リスト 1 に示す.

リスト 1 Formura の最小コード

```
1  dimension :: 1
2  axes :: x
3
4  begin function U = init()
5    double [0] :: U
6    U[i] = 0.0
7  end function
8
9  begin function U_next = step(U)
10   U_next[i] = U[i]
11 end function
```

このコードで行っていることは以下の 4 つである.

- グリッドの次元の指定
- 軸の名前の宣言
- 状態変数の初期化関数の定義
- 状態更新関数の定義

`dimension ::`は，グリッドの次元数を設定する特殊構文である．`axes ::`は，軸の名前を定義する特殊構文である．`init` は，シミュレーションの状態を記述する状態変数の初期化を行う．`step` は，シミュレーションの状態を記述する状態変数の更新を行う.

上記の特殊宣言および関数は省略することができない.

5.2 Formura 最小設定

Formura で計算する計算空間の格子数および長さは Formura プログラムとは別の設定ファイルを用いて記述する．設定ファイルは YAML 形式で記述し，各軸方向の長さと格子数の設定は必須である．また，この設定ファイルでは MPI のノード数や後述するテンポラルブロッキングの設定も行う．最小限の設定ファイルをリスト 2 に示す．

リスト 2 Formura の最小設定ファイル

```
1 length_per_node: [1.0]
2 grid_per_node: [100]
```

`length_per_node` には各軸方向の長さを記述する．`grid_per_node` には MPI のノード 1 つあたりのグリッド数を記述する．

リスト 2 に書かれた必須の設定項目の他に省略可能な設定項目として，MPI ノード数の設定とテンポラルブロッキングの設定がある．MPI ノード数に関する記述を省略すると，1 ノードだけを用いる計算とみなされる．テンポラルブロッキングに関する記述を省略すると，テンポラルブロッキングは使用されない．それぞれの設定項目の詳細はチュートリアルの中で必要に応じて解説する．

Formura コンパイラのデフォルト動作では，Formura コードと同じベースネームを持ち，拡張子.`yaml` のファイルが設定ファイルとして読み込まれる．

リスト 1 および 2 に示した Formura コードおよび設定ファイルをそれぞれ `hello.fmr`，`hello.yaml` というファイル名で保存したと仮定する．ファイルを保存したディレクトリ上で，

```
$ formura hello.fmr
```

を実行すると，`hello.c` と `hello.h` が生成される．

5.3 最小ドライバコード

Formura が生成した C 言語コードに含まれる API を操作してステンシル計算を実行するドライバコードの最小構成をリスト 3 に示す．

リスト 3 最小ドライバコード

```
1 #include "hello.h"
```

```
2 int main(int argc, char **argv)
3 {
4   Formura_Navi n;
5   Formura_Init(&argc, &argv, &n);
6   Formura_Forward(&n);
7   Formura_Finalize();
8   return 0;
9 }
```

まず，Formura が生成したヘッダファイルをインクルードする（この例では `hello.h`）．構造体 `Formura_Navi` は計算のタイムステップ等を管理する構造体である．関数 `Formura_Init` は Formura の関数の `init` の内容に従って，グリッドの状態を初期化する．関数 `Formura_Forward` は Formura の関数 `step` に従ってグリッドの状態を更新する．1 回の `Formura_Forward` 呼び出しで進むタイムステップ数は通常は 1 ステップである．テンポラルブロッキングが設定されている場合は設定ファイルの `temporal_blocking_interval` の設定値だけタイムステップが進む．関数 `Formura_Finalize` は Formura の計算を安全に終了させる．

6 サンプルコード (1 次元オイラー方程式)

流体力学のオイラー方程式を解く Formura プログラムを例に，Formura の実用的な使用方法を紹介する．1 次元オイラー方程式を HLLC 法を用いて解く Formura コード一式は，`sample/1d_hllc/`ディレクトリにある．`sample/1d_hllc/`ディレクトリの内容は以下の通りである．

- `hllc.fmr`
  Formura ソースコード
- `config.yaml`
  Formura 設定ファイル
- `main.c`
  ドライバおよびファイル IO のソースコード
- `Makefile`
  Formura コンパイル，C コンパイル，計算の実行・計算結果のプロットを行う Makefile

- `formura_data_load.py`
  計算結果のファイルを読み込んで numpy の ndarray 型を出力する関数
- `plot.py`
  計算結果を画面上および PNG ファイルに出力するスクリプト
- `ref_data/`
  Toro[1] の HE-E1RPEXACT から生成したリーマン問題の厳密解

このプログラムで使用されている方程式やソルバーの詳しい説明は「サンプルプログラム解説」に記述されている．

6.1　次元および軸名の指定

`sample/1d_hllc/hllc.fmr` の先頭には，格子の次元と各方向の軸の名前を指定する特殊宣言がある．リスト 4 に特殊宣言の記法を示す．

リスト 4　特殊宣言

```
1 dimension :: 1
2 axes :: x
```

`dimension :: 1` はこのコードが対象とする空間の次元が 1 次元であることをあらわす．`axes :: x` は 1 番目の軸の名前が `x` であることを表す．

ここで指定した軸名は格子の幅を表す変数の名前や，その軸方向の格子数を表す変数の名前に使用される．今回の例では，格子幅を表す変数が `dx`，格子数を表す変数が `total_grid_x` として，暗黙のうちに定義される．

`axes` と `dimension` の宣言は必須である．またここで指定した次元数は，設定ファイル (yaml ファイル) で指定するグリッド数・MPI ノード数の次元と同じでなければならない．

6.2　グローバル変数宣言

Formura で関数スコープの外で宣言した変数はグローバル変数となり，どの関数からも参照することができる．グローバル変数には計算全体で使用するパラメータ等を置く．リスト 5 にグローバル変数宣言の例を示す．

リスト 5　グローバル変数

```
1 double :: max_signal_speed = 4
2 double :: C_cfl = 0.8
3 double :: dt = C_cfl*dx/max_signal_speed
4 double :: gamma = 7.0/5.0
5 double :: gamma1 = gamma - 1.0
6 double :: beta = 1.0  # MINMOD flux limitter
```

1 行目から 6 行目まで，変数の宣言と初期化を行っている．Formura で使用できる型についてのより詳しい解説は，付録 B.5 にある．

6.3　外部関数宣言

Formura では C 言語で定義された外部関数を使用することができる．現時点では，使用できる関数は double 型の値一つを引数にとり，double 型の値を返す関数に限る．サンプルでは，math.h で定義されている平方根関数を外部関数として呼び出している．コードをリスト 6 に示す．

リスト 6　外部関数宣言

```
1 extern function :: sqrt
```

外部関数の使用法は，Formura の通常の関数と同じである．

6.4　init 関数

`init` 関数は各格子における物理量の初期値を与える．`init` 関数は，コード中に必ず定義されていなければならない．

`sample/1d_hllc/hllc.fmr` の `init` 関数をリスト 7 に示す．

リスト 7　init 関数

```
1 begin function (rho, rho*V, e) = init()
2   double [] :: rho, u, v, w, p
3   rho[x]  = center_step_x(x, 1.0, 0.125)
4   u[x] = center_step_x(x, 0.75, 0.0)
5   v[x] = center_step_x(x, 0.0, 0.0)
```

```
6   w[x] = center_step_x(x, 0.0, 0.0)
7   p[x]  = center_step_x(x, 1.0, 0.1)
8   V = (u, v, w)
9   e  = p/gamma1 + 0.5*rho*norm2(V)
10 end function
```

init 関数で使用されている文法事項について詳しく解説する.

6.4.1 関数宣言

Formura の関数は begin function から end function の間に記述する. begin function に続いて,(返り値のタプル) = 関数名 (引数のタプル) の形式で関数の返り値,名前,引数を指定し,次の行から関数本体を記述する.関数の返り値の指定方法はもう一つあり,関数名 (引数のタプル) returns (返り値のタプル) と書くこともできる.

6.4.2 変数宣言

リスト 7 の 2 行目の double [] :: rho, u, v, w, p は変数宣言である. double [] はこれから宣言する変数が倍精度浮動小数点数のグリッドであることを示す. ここでは,密度,速度,圧力の値を持つグリッドとして,rho, u, v, w, p を定義している.

[] の中身を省略せずに書くと,double [0] となる.これは,このグリッドが整数オフセットのインデックス持つことを表す.Formura では有理数オフセットを持つインデックスを指定することが可能である.例えば,格子点と格子点の中間 (境界) で定義されたグリッドの型は double [+1/2] と書くことができる.Formura ではグリッドの型やオフセットは自動的に導出されるので,init 以外で変数宣言を行うことは稀である.

6.4.3 代入

リスト 7 の 3 行目以降では,変数に値を代入している.3 行目では密度を表すグリッド変数 rho に値 center_step_x(x, 1.0, 0.125) を書き込んでいる.ここで,x はグリッドのインデックスを表す.Formura はステンシル計算専用言語なので,すべてのインデックスに対して同じパターンの計算が行われる.したがって,rho[x] = center_step_x(x, 1.0, 0.125) は,"x の取りうるすべての値に対して,center_step_x(x, 1.0, 0.125) の値を評価して代入する" と解釈される.

8 行目では,タプル V が定義されている.V は流速の x,y,z 成分を並べたもので,流速

ベクトルを意味する．ここでは，変数宣言および型定義は省略されている．変数宣言を省略せずに書くなら，(double [], double [], double[]) :: V となる．

9 行目では，グリッド e が定義されている．e はエネルギー密度である．e の代入には，インデックスが省略されている．左辺と右辺でインデックスのオフセットが一致している場合はインデックスの指定を省略することができる．インデックスの指定を省略せずに書くと，e[x] = p[x]/gamma1 + 0.5*rho[x]*norm2(V[x]) となる．

6.5 step 関数

step 関数は Formura の計算を 1 ステップ進めるときに行われる演算を記述する．step は Formura における main 関数の役割を果たす．Formura コード中には必ず step が定義されていなければならない．

sample/1d_hllc/hllc.fmr の step 関数をリスト 8 に示す．

リスト 8 step 関数

```
1 begin function U_next = step(rho, (rhou, rhov, rhow), e)
2   U = (rho, (rhou, rhov, rhow), e)
3   #(U_xL, U_xR) = first_order_LR(U)   # first-order scheme
4   (U_xL, U_xR) = MUSCL_Hancock(U)   # second-order scheme
5   manifest :: F = HLLC_Flux(U_xL, U_xR)
6   U_next = time_evol(U, F)
7 end function
```

step 関数は，現在のタイムステップにおけるグリッドの状態を表す変数を引数にとり，次のタイムステップにおけるグリッドの状態を返す．したがって，step の引数と返り値の型は同じでなければならない．

さらに，step の引数は特別な意味を持つ．step の引数は計算対象の系の状態を保持するために欠かせない状態変数であるとみなされ，必ず配列に保持される．この配列はヘッダファイルに公開されるため，ユーザーが作るドライバコードからも読み出すことができる．ここでは，rho，rhou，rhov，rhow，e が状態変数である．それぞれ，密度，運動量 (x 成分,y 成分,z 成分)，エネルギーである．init の返り値の型は step の引数の型と同じでなければならない．

4 行目では，MUSCL-Hancock 法を使って，状態変数の 2 次精度の補間を行っている．

3 行目，4 行目に見られる#以降の文字はコメントである．このコードでは，3 行目と 4

行目のどちらかをコメントアウトすることで，計算精度を変えることができる．

5 行目は，オイラー方程式の流束ベクトルを HLLC 法を使って計算している．流束ベクトル F には型修飾子の manifest がついている．manifest がついた変数は配列としてメモリ上に保持される．manifest がついていない変数は局所変数として使い捨てにされるため，使用されるたびに再計算される．この再計算を回避したいときに manifest を使用する．

6.6　min, max 関数

計算中何度も使用される min,max 関数の定義について解説する．

Formura では，通常の関数定義に加えてラムダ式を定義することができる．例えば，sample/1d_hllc/hllc.fmr の min 関数，max 関数の定義は 9 のように記述されている．

リスト 9　min 関数 max 関数

```
1 max = fun(a, b) if a > b then a else b
2 min = fun(a, b) if a > b then b else a
3 max3 = fun(a, b, c) max(max(a,b),c)
4 min3 = fun(a, b, c) min(min(a,b),c)
```

min,max,max3,min3 はいずれもラムダ式方式で定義された関数である．ラムダ式は一般に fun(x)（x を使った式）という形式で書かれる．これを特定の名前に束縛することで通常の関数と同じように使用することができる．また，ラムダ式を名前に束縛せずに無名関数として評価することも可能である．

min,max が呼ばれたときに評価される式 if a > b then a else b は $a > b$ が真のとき a を返し，$a > b$ が偽のとき b を返す．一般に条件分岐式は，if（論理式）then（真のときの値）else（偽のときの値）という形式で書かれる．

6.7　数値微分

sample/1d_hllc/hllc.fmr では，数値微分演算子を表す関数 ddx を 10 のように定義している．

リスト 10　微分演算子

```
1 Xf = fun(U) (U[+1/2])
2 Xb = fun(U) (U[-1/2])
```

```
3
4 ddx = fun(F) (Xf(F) - Xb(F))/dx
```

Xf は，グリッドのインデックスを $\frac{1}{2}$ 進める前進演算子，Xb は，グリッドのインデックスを $\frac{1}{2}$ 減らす後退演算子である．ここで，添字のインデックスの文字が省略されている．添字のインデックスの文字を省略した場合，それは左辺のグリッドのインデックスに対して相対的に $\frac{1}{2}$ 進める・減らすという意味になる．

したがって，ddx 関数は，隣り合う格子点の物理量の値の差を取り dx で割った後，格子点の中間点（セル境界面）のオフセットを持つグリッドの値として評価する．

6.8 タプル

Formura ではタプルを使って複数の型の値を組み合わせることができる．例えば，リスト 8 の step 関数は，U = (rho, (rhou, rhov, rhow), e) というタプルを定義している．このタプルの型は (double [], (double [], double [], double []), double []) である．タプル同士の演算では，タプルの外側からパターンマッチを行い，タプル中の同じ場所にいる数値同士に演算が施される．例えば，(a,(b,c)) + (d,(e,f)) は (a+d,(b+e,c+f)) と同じ結果になる．また，タプル型と要素型の演算では，タプルのすべての要素に演算が分配される．(a,b) +c は (a+c, b+c) と同じ結果になる．

6.9 その他の関数

ここで紹介していないユーザー定義関数の意味については，別資料「サンプルコード解説」を参照すること．

6.10 設定ファイル

Formura のグリッドの情報は YAML 形式の設定ファイルに書く．サンプルプログラムの YAML ファイル sample/1d_hllc/config.yaml をリスト 11 に示す．

リスト 11 設定ファイル

```
1 length_per_node: [4.0]
2 grid_per_node: [400]
```

length_per_node は MPI1 ノードあたりの空間幅を実数のリストで指定する．

grid_per_node は MPI1 ノードあたりの格子数を整数のリストで指定する．この例では，計算は 1 ノードのみで行われ，空間幅は 4.0，グリッド数は 400 である．しがたって，格子幅は $\Delta x = \frac{4.0}{400} = 0.01$ となる．

6.11 ドライバコード

ドライバコードは C 言語で記述する．sample/1d_hllc/main.c の main 関数ををリスト 12 に示す．

リスト 12 ドライバコードの main 関数

```
1  int main(int argc, char **argv) {
2    Formura_Navi n;
3    Formura_Init(&argc, &argv, &n);
4    // Initialize Formura stepper
5
6    writeData(&n);
7    while(n.time_step < T_MAX) {
8      Formura_Forward(&n);     // run Formura stepper
9    }
10   writeData(&n);
11
12   Formura_Finalize();    // Finalize Formura
13   return 0;
14 }
```

Formura が生成した 2 つの関数 Formura_Init と Formura_Forward を呼び出すことで，Formura の計算を行う．Formura 側の状態は Formura_Navi 構造体で管理される．Formura_Navi 構造体には，現在のタイムステップ数など Formura の制御に必要な情報が含まれている．

これらは，Formura が出力したヘッダファイルで宣言され，C ファイルで定義されている．Formura_Init はグリッドを初期化して，Formura が計算を始める準備を整える．Formura_Forward は Formura のタイムステップを進める．ドライバコードから使用できる API について，より詳しい解説は付録 A 記載されている．

writeData はユーザーが定義するファイル出力関数である．リスト 13 に定義を示す．

リスト 13 ファイル出力

```
1  static void writeData(const Formura_Navi *n) {
2    FILE *fp = file_open("data/", n->time_step, n->my_rank);
3
4    for (int ix = n->lower_x; ix < n->upper_x; ix++) {
5      const double x = to_pos_x(ix,*n);
6      const double y = 0.0;
7      const double z = 0.0;
8      const double rho = formura_data.rho[ix];
9      const double u = velocity(rho, formura_data.rhou[ix]);
10     const double v = velocity(rho, formura_data.rhov[ix]);
11     const double w = velocity(rho, formura_data.rhow[ix]);
12     const double e_K = kinetic_energy(rho, u, v, w);
13     const double p = pressure(formura_data.e[ix], e_K);
14     const double T = temperature(rho, p);
15     fprintf(fp,
16   "%.17e %.17e %.17e %.17e %.17e %.17e %.17e %.17e %.17e\n
       "
17           , x      // column 0
18           , y      // column 1
19           , z      // column 2
20           , rho    // column 3
21           , p      // column 4
22           , T      // column 5
23           , u      // column 6
24           , v      // column 7
25           , w      // column 8
26           );
27   }
28   fclose(fp);
29 }
```

Formura のグリッドデータは `formura_data.{変数名}`という名前の配列に格納されてい

る．この変数名は Formura コードで指定した名前 (`step` 関数の引数に使用した名前) である．配列の中で有効な値が入っている領域の境界インデックスは `Formura_Navi` 構造体のメンバ変数 `n->lower_x` と `n->upper_x` である．ファイル出力のための for ループ文はこのインデックスの間でまわすことになる．また，特定のインデックスの座標を知りたければ `to_pos_x` 関数を使用する．

6.12 コンパイル・実行

作成した Formura コードをコンパイルする．`hllc.fmr` という名前の Formura コードを，`config.yaml` という設定ファイルとともにコンパイルするためには

```
$ formura --nc config.yaml hllc.fmr
```

を実行する．オプション`--nc` で設定ファイル名を指定する．このオプションが省略された場合，Formura コードと同じベースネームを持つ YAML ファイルが読み込まれる．

上記コマンドを実行すると，`hllc.c` と `hllc.h` が出力される．`hllc.h` には main 関数から呼び出す FormuraAPI の宣言およびグリッド変数の宣言が入っているため，これをドライバコードからインクルードする．

コードのコンパイルは，`mpicc` を使用して，

```
$ mpicc hllc.c main.c -o main -lm
```

とする．ここでは実行ファイルの名前を `main` とする．これは通常の C 言語のコンパイルと変わらないので，必要に応じてコンパイルオプションを追加したり，分割コンパイルすることができる．

プログラムの実行は，`run` コマンドで行う．`run` コマンドは Formura コンパイル時に生成される．`run` コマンドの実行には，実行ファイル名 `main` を引数に与えて，

```
$ mkdir -p data
$ ./run main
```

とする（ファイルの出力先として事前に `data` ディレクトリを作成している．）．

上記の一連の動作はサンプルプログラムに付属の `Makefile` を使って行うこともできる．

```
$ make run
```

とすると，Formura コードのコンパイルから実行までの一連の手順が実行される．

6.13　可視化

サンプルプログラムには，結果の可視化を行う Python スクリプトが付属している．

```
$ python3 plot.py
```

または，

```
$ make plot
```

を実行することで，グラフが描画される．

実行結果は図 1 のようになる．

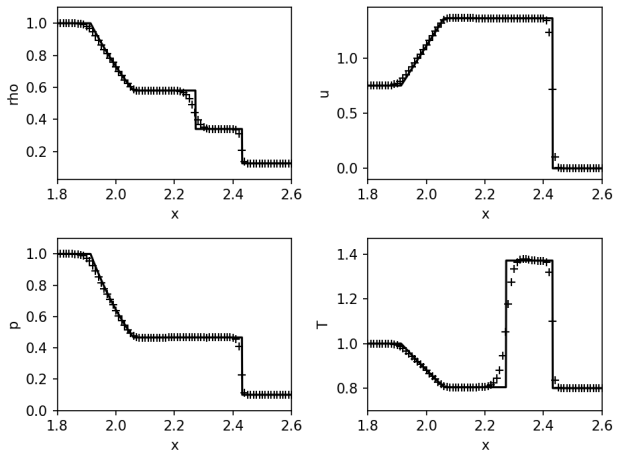

図 1　HLLC 法を用いた 1 次元衝撃波管問題の解．

7　サンプルコード (3 次元オイラー方程式)

3 次元オイラー方程式を HLLC 法を用いて解くサンプルプログラムを用いて，3 次元コードの書き方について解説する．このサンプルプログラムは `sample/3d_hllc/` ディレクトリ以下にある．

7.1 次元および軸名の指定

Formura ソースファイルの sample/3d_hllc/3d_hllc.fmr 特殊宣言文をリスト 14 に示す.

リスト 14 特殊宣言

```
1 dimension :: 3
2 axes :: x,y,z
```

この部分で，計算の次元が 3 次元であり，それぞれの軸の名前が x,y,z であると定義される.

この宣言により，このコードにおけるグリッド型変数はすべて 3 次元グリッドとなる．また，格子幅を表す変数 dx，dy，dz および格子数を表す変数 total_grid_x，total_grid_y，total_grid_z が暗黙のうちに定義される.

7.2 前進後進演算子と微分演算子

3 次元版の前進後進演算子と微分演算子の定義をリスト 15 に示す.

リスト 15 前進後進演算子と微分演算子

```
1  Xf = fun(U) (U[+1/2, +0, +0])
2  Xb = fun(U) (U[-1/2, +0, +0])
3  Yf = fun(U) (U[+0, +1/2, +0])
4  Yb = fun(U) (U[+0, -1/2, +0])
5  Zf = fun(U) (U[+0, +0, +1/2])
6  Zb = fun(U) (U[+0, +0, -1/2])
7
8  ddx = fun(F) (Xf(F) - Xb(F))/dx
9  ddy = fun(F) (Yf(F) - Yb(F))/dy
10 ddz = fun(F) (Zf(F) - Zb(F))/dz
```

前進演算子 Xf,Yf,Zf および後進演算子 Xb,Yb,Zb をラムダ式を使って定義している．3 次元グリッドのオフセットを 3 つの有理数の組で指定している．前進演算子 Xf を使って

`Xf(U)` と書くと，グリッド変数 `U` の添字が x 軸方向に $\frac{1}{2}$ 前進する．他の軸についても同様である．

微分演算子 `ddx,ddy,ddz` はラムダ式を使って定義されており，それぞれの軸について，インデックスを $\frac{1}{2}$ 前進させたものと $\frac{1}{2}$ 後退させたものの差をとって格子幅で割ることで，数値微分を行っている．例えば，グリッド変数 `U` について，`ddx(U)` を評価すると，`(U[+1/2,0,0]-U[-1/2,0,0])/dx` となる．

7.3　ナブラ演算子と発散

微分演算子のタプルを使ってナブラ演算子を定義できる．また，ナブラ演算子を使って，ベクトル場の発散を定義することができる．コード例をリスト 16 に示す．

リスト 16　ナブラ演算子とベクトル場の発散

```
1 sum = fun(A) A(0) + A(1) + A(2)
2
3 nabla = (ddx, ddy, ddz)
4 div = fun(V) sum(fun(i) nabla(i) V(i))
```

`sum` は引数に `(0),(1),(2)` を作用させてから和を取る関数である．`nabla` は先に定義した微分演算子のタプルである．`div` の実装には，`sum` 関数の引数にラムダ式を渡すというトリックを使っている．この式を展開すると，`div = fun(V) ddx(V(0)) + ddy(V(1)) + ddz(V(2))` となる．したがって，`V` にグリッド変数のタプルを渡すことで，ベクトル場の発散を表すことができる．

7.4　タプルの回転と 3 次元フラックス計算

サンプルコードでは，タプルを回転させる演算子 `YZXcycle` と `ZXYcycle` を定義している．これは，3 つの変数を並べたタプルに対して，その要素を右もしくは左に回転させるものである．それぞれの定義をリスト 17 に示す．

リスト 17　3-タプルの回転

```
1 YZXcycle = fun(V) (V(1), V(2), V(0))
2 ZXYcycle = YZXcycle.YZXcycle
```

`YZXcycle` は，3 要素タプルを左に回転させる．したがって，`YZXcycle(X,Y,Z)` は `(Y,Z,X)` と等しい．`ZXYcycle` は右回り回転である．左回り回転を 2 回行うと右回り

回転と等しくなることから，ZXYcycle は YZXcycle の合成で定義されている．ピリオド演算子'.' は関数合成の演算子であり，関数 A，B に対して，(A.B)(x) は A(B(x)) と等しい．

タプルの回転は 3 次元ベクトルに対して軸の向きを入れ替える用途に使うことができる．サンプルコードでは，x 軸方向のフラックスを求める関数 HLLC_Flux と，タプルの回転演算子を組み合わせて，y 軸方向フラックスおよび z 軸方向フラックスを求めている．リスト 18 に x,y,z それぞれの方向のフラックスの計算コードを示す．

リスト 18　各軸方向のフラックス

```
1  begin function (Frho, FrhoV, Fe) = HLLC_Flux_1D((rho_L,
       rhoV_L, e_L), (rho_R, rhoV_R, e_R), cycle)
2    inv_cycle = cycle.cycle
3    cyc_rhoV_L = cycle(rhoV_L)
4    cyc_rhoV_R = cycle(rhoV_R)
5    (Frho, cyc_FrhoV, Fe) = HLLC_Flux((rho_L, cyc_rhoV_L,
       e_L), (rho_R, cyc_rhoV_R, e_R))
6    FrhoV = inv_cycle(cyc_FrhoV)
7  end function
8
9  HLLC_Flux_x = fun(U_L, U_R) HLLC_Flux_1D(U_L, U_R, id)
10 HLLC_Flux_y = fun(U_L, U_R) HLLC_Flux_1D(U_L, U_R,
       YZXcycle)
11 HLLC_Flux_z = fun(U_L, U_R) HLLC_Flux_1D(U_L, U_R,
       ZXYcycle)
```

関数 HLLC_Flux_1D は，セル境界の左右の状態量とタプル回転演算子を受け取り，ベクトルに対して適切な回転を施した上で x 軸方向のフラックスを求める関数 HLLC_Flux を呼び出す．また，その結果に対して逆回転を施して，フラックスの結果を返す．

例として，y 軸方向フラックスを求める関数 HLLC_Flux_y の動作を説明する．この関数は，回転関数として YZXcycle を HLLC_Flux_1D に渡している．HLLC_Flux_1D は，受け取った状態量の速度ベクトルに対して，y 軸成分が先頭に来るように YZXcycle でタプルを回転させる．そして，y 軸成分が先頭にあるベクトルを HLLC_Flux に渡して，y 軸成分が先頭に来ているフラックスを受け取る．そして，運動量フラックスのベクトルに対して，タプルを逆回転させて正しい順番のフラックスベクトルを作る．

7.5 設定ファイル

このサンプルコードの設定ファイルをリスト 19 に示す．

リスト 19 設定ファイル

```
1 length_per_node: [0.1, 0.5, 0.1]
2 grid_per_node: [10, 50, 10]
3 mpi_shape: [1,8,1]
```

3 次元のグリッドを定義するため，1 ノードあたりのサイズ `length_per_node` とグリッド数 `grid_per_node` も 3 つの数字で指定する．左から順に x 軸,y 軸,z 軸のサイズである．

このサンプルでは MPI 並列化を行っている．`mpi_shape` に各軸方向の MPI ノード数を指定する．この場合，y 軸方向に 8 つのノードを作り，それぞれが並列に計算を実行する．`mpi_shpae` は省略可能パラメータである．省略すると各軸方向に 1 ノードと解釈され，並列化せずに計算される．

7.6 ドライバコード

ドライバコードの記法は 1 次元オイラー方程式のサンプルと同様である．

7.7 コンパイル・実行・可視化

コンパイル・実行・可視化は 1 次元オイラー方程式のサンプルと同様である．

8 サンプルコード（1 次元 MHD 方程式）

1 次元 MHD 方程式を HLLD 法を用いて解くサンプルコードを例に，Formura のテンポラルブロッキングの使用法を解説する．このサンプルコードはディレクトリ `sample/mhd_1d_hlld/`にある．

このサンプルコードでは，新たな文法事項は登場しない．MHD 方程式や HLLD 法についての解説は別資料「サンプルコード解説」を参照すること．

8.1　テンポラルブロッキング

テンポラルブロッキングとは，ステンシル計算に対して行うキャッシュ最適化の手法である．ステンシル計算では次ステップを計算するために必要な情報が局所的に限られている．したがって，グリッドのある一部分を取り出してその小領域内で計算可能な範囲で複数ステップ計算を進めることが可能である．小領域の範囲をうまく選ぶと，一連の計算でアクセスするメモリ領域が限定されるため，計算に必要なすべての情報を CPU のキャッシュ内に収めることができる．このようにすると，メモリバンド幅に制限されずに CPU の性能を最大限に活かした計算が可能である．

Formura は設定ファイルにテンポラルブロッキングを行うブロックのサイズとタイムステップ数を指定するだけで，テンポラルブロッキングに対応したコードを自動的に生成する．

テンポラルブロッキングを行う設定ファイルをリスト 20 に示す．

リスト 20　テンポラルブロッキングを行う設定ファイル

```
1 length_per_node: [0.5]
2 grid_per_node: [400]
3 mpi_shape: [8]
4 grid_per_block: [42]
5 temporal_blocking_interval: 5
```

1 から 3 行目まではすでに説明した．4 行目はテンポラルブロッキングを行う領域の幅を指定する．5 行目はテンポラルブロッキングを行うタイムステップ数を指定する．

一つのテンポラルブロッキング幅で使用するメモリ量がキャッシュ量以下になるように設定すると，性能向上の恩恵を受けやすい．

`grid_per_block` と `temporal_blocking_interval` の値はある制約条件を満たす必要がある．`grid_per_block` の値を N_b，`temporal_blocking_interval` の値を N_T，`grid_per_node` の値を N_n，スキームの袖幅を N_s とすると，

$$(N_n + 2N_sN_T) \equiv 0 \mod N_b \tag{1}$$

が成り立つ必要がある．スキームの袖幅とは，ある格子点を 1 ステップ更新するために必要な格子点の集合のうち，最も遠くの点までのグリッド数のことである．時間空間 1 次精度の場合 $N_s = 1$，2 次精度の場合 $N_s = 2$ になる．

9　サンプルコード (3 次元 MHD 方程式)

3 次元 MHD 方程式を解くサンプルプログラムは `sample/mhd_3d_hlld/`ディレクトリにある．このコード使用されている文法はすでに解説済みなので，ここでは解説を省略する．スキームの詳細は「サンプルコード解説」を参照のこと．

付録 A　Formura の API 一覧

Formura コンパイラが生成するヘッダファイルには，ドライバコードから利用するための関数や変数が定義されている．

Formura の計算に使用するグリッド変数は `Formura_Grid_Struct` 構造体のメンバ変数に配列として定義されている．グリッド変数の値を読み出す際は，`Formura_Grid_Struct` 型の外部変数 `formura_data` を使用する．`Formura_Grid_Struct` のメンバ変数名は Formura の `step` 関数の引数に指定した変数名が使用される．

Formura の状態は `Formura_Navi` 構造体で管理される．Formura の API を呼び出す際は必ず `Formura_Navi` 構造体への有効なポインタを渡す．構造体 `Formura_Navi` の主要なメンバを表 1 に示す．これらのメンバ変数を利用して Formura を制御したりデータ加工やファイル出力を行うことができる．

表 1　`Formura_Navi` 構造体の主要フィールド

型	フィールド名	説明
`int`	`time_step`	タイムステップ数
`int`	`lower_{軸名}`	グリッドデータが入っている配列において，有効なデータを得ることができる最小のインデックス
`int`	`upper_{軸名}`	グリッドデータが入っている配列において，有効なデータを得ることができる最大のインデックスの次のインデックス
`double`	`space_interval_{軸名}`	ある軸方向での格子点間隔
`int`	`my_rank`	MPI ランク番号

Formura の主要な API 関数 `Formura_Init`，`Formura_Forward`，`Formura_Finalize` の定義を表 2，3，4 に示す．

表 2　`Formura_Init` 関数の定義

定義	`void Formura_Init(int *,char ***,Formura_Navi *)`
引数 1	コマンドライン引数の個数へのポインタ．
引数 2	コマンドライン引数へのポインタ．
引数 3	`Formura_Navi` 型構造体へのポインタ
説明	Formura のグリッドを初期化する．Formura の状態を管理する `Formura_Navi` 型の構造体を初期化する．

表 3　`Formura_Forward` 関数の定義

定義	`void Formura_Forward(Formura_Navi *)`
引数 1	`Formura_Navi` 型構造体へのポインタ
説明	Formura のグリッドを更新する．`Formura_Navi` 型構造体を更新する．

表 4　`Formura_Finalize` 関数の定義

定義	`void Formura_Finalize(Formura_Navi *)`
引数 1	`Formura_Navi` 型構造体へのポインタ
説明	Formura の計算を安全に終了する．

ファイル出力等に使用する補助的な API 関数 `to_pos_{軸名}`の定義を表 5 に示す．

表 5　`to_pos_軸名`関数の定義

定義	`double to_pos_{軸名}(int, Formura_Navi)`
引数 1	インデックス
引数 2	`Formura_Navi` 構造体の値
返り値	座標
説明	ある軸におけるインデックスの値に対応する座標を返す．

付録 B　Formura の言語仕様

Formura の言語仕様を記載する．言語仕様については formura 文法仕様書 [2] および formura ユーザーマニュアル [3] を参考に，現在のバージョンの Formura 処理系で有効なものを抜粋して記載する．

B.1　文字

Formura で使用できる文字の文法上の区分を表 6 に示す．

表 6　Formura の文字種類一覧

文字種	説明
文区切り文字	改行'\n' またはセミコロン';'
空白文字	ascii 空白文字から改行文字'\n' を除いたもの，およびコメントとみなされる部分（'#' と改行'\n' に挟まれた文字）.
アルファベット	ascii 文字アルファベット'a' から'z'，'A' から'Z'，アンダースコア'_'
数字	'0' から'9' までの数字文字
記号文字	ユニコードの印字可能文字のうちアルファベットと空白文字でないものから'"\(),;[]#を除いたもの.

B.2　識別子

識別子は変数名や関数名，演算子名になる文字列である．識別子を作る方法は二通りある．

- アルファベットから始まり，0 個以上のアルファベットと数字からなる列が続くもの
- 1 個以上の記号文字からなる文字列

B.3　プログラム

Formura のプログラムは文を文区切りで区切ったものである．文区切り文字は改行'\n' またはセミコロン';' である．

B.4　文

Formura の文には代入文，型宣言文，特殊宣言文，関数定義がある．なにもない文（空文）も許される．

B.4.1　代入文

代入文の形式は<左辺パターン>=<式>である．

B.4.2　型宣言文

型宣言文の形式は<型名> :: <変数名>である．変数名はカンマ区切りで複数書くことができる．型宣言時に代入文によって変数を初期化することもできる．

B.4.3　特殊宣言文

特殊宣言文の形式は<特殊宣言名> :: <変数名>である．特殊宣言名には axes と dimension がある．

B.4.4　関数定義文

関数を定義するには，begin function に続けて関数の返り値パターン，関数名を記述し，関数本体の文を並べ，end function で終了する．関数定義文の形式は，begin function <返り値> = <関数名><引数パターン><区切り><文><区切り>...end function である．また，関数宣言文のもう一つのパターンとして，返り値を returns に続けて書くこともできる．この場合の形式は，begin function <関数名><引数パターン> returns <返り値パターン> <区切り><文><区切り>...end function である．

B.5　型

Formura の型は要素型，グリッド型，タプル型がある．

B.5.1　要素型

Formura の基本となる型を要素型と呼ぶ．要素型から派生してグリッド型やタプル型を作ることができる．要素型の一覧を表 7 に示す．

表 7　Formura の要素型一覧

型名	説明
`'float'`	単精度実数
`'double'`	倍精度実数

B.5.2　グリッド型

要素型名の後ろに **[各軸方向のオフセット]** をつけるとグリッド型になる．グリッド型のパターンは，1 次元の場合，**<要素型名> [<軸 1 方向オフセット>]** 2 次元の場合，**<要素型名> [<軸 1 方向オフセット>,<軸 2 方向オフセット>]** である．より高次元の場合も同様である．オフセットは 0 以上 1 未満の有理数で指定し，0 の場合は省略可能である．オフセットは**<正数>/<正数>**という形式で書く．

B.5.3　タプル型

タプル型は，要素型やグリッド型や他のタプル型を組み合わせて作る複合型である．タプル型のパターンは **(<型 1>,<型 2>,...,<型 n>)** である．

B.6　式

Formura の式は，即値，変数名，単項演算子式，二項演算子式，if 式，ラムダ式，カッコ式，格子アクセス式，要素アクセス式，関数呼び出しから構成される．

B.6.1　即値

即値は数字リテラルであり，整数または浮動小数点数である．浮動小数点数の記法は一般的なプログラミング言語における記法と同様である．

B.6.2　変数名

変数名は有効な変数の識別子である．

B.6.3　単項演算子式

単項演算子式の形式は<単項演算子><式>である．

B.6.4　二項演算子式

二項演算子式の形式は<式><二項演算子><式>である．

B.6.5　if 式

if 式の形式は，`if` <条件式> `then` <条件式が `true` のときの評価式> `else` <条件式が `false` のときの評価式> である．

B.6.6　ラムダ式

ラムダ式は関数と同じように呼び出すことができる式である．ラムダ式を変数に代入すると通常の関数と全く同じ使い方ができる．ラムダ式の形式は，`fun`<引数パターン> <式>である．

B.6.7　カッコ式

カッコ式はカッコで囲まれた式である．パターンは (<式>) である．カッコ式は通常のプログラミング言語と同様に，演算子の評価順序を制御したいときに用いる．また，2 項演算子の’,’ を使用して式を並べることでタプル型の値を作ることができる．

B.6.8　格子アクセス式

格子アクセス式はグリッド変数のインデックスを指定してアクセスする方法である．格子アクセス式のパターンは<グリッド変数名>[<カーソル式>,...] である．カーソル式はインデックスを表すパターン変数とオフセット値を組み合わせて他のグリッドアクセス式との間に相対的なインデックス差を作る式である．たとえば，`a[i] = b[i + 1] - b[i - 1]` とかくと，グリッド変数 `a` のインデックス `i` における値 `a[i]` に，グリッド変数 `b` を左に 1 グリッドずらした `b[i+1]` と右に 1 グリッドずらした `b[i-1]` の差を代入するという意味になる．インデックスの識別子は省略することができる．また，添字の差分が 0 の場合は格子アクセス式 `[]` を省くこともできる．格子アクセス式を使ってグリッド同士の演算や代入を行うときは，それぞれの項のグリッドのオフセットが一致している必要がある．例えば，整数オフセットのグリッドと半整数オフセットのグリッドを直接演算することはできない．この場合，必ずどちらかのグリッドのイン

デックスを格子アクセス式 `[i+1/2]` でずらす必要がある．

B.6.9　要素アクセス式

要素アクセス式はタプルの特定の要素にアクセスするための記法である．タプルのインデックスは 0 から始まる．タプル変数 `a` の 2 番目の要素の値を取り出すためには `a(1)` とかく．要素アクセス式のパターンは**<タプル変数名>(<0 以上の整数式>)**

B.6.10　関数呼び出し

関数呼び出しのパターンは**<関数名>(<式>,...)** 引数の式が 1 つしかない場合，() は省略できる．

B.7　演算子

二項演算子の一覧を表 8 に，単項演算子の一覧を表 9 に示す．二項演算子の優先順位は表の上位ほど高い．単項演算子の'+'，'-' の順位は二項演算子の'+'，'-' と同じである．

表 8　Formura の二項演算子一覧

順位	<二項演算子>	結合性	意味
1	' '(併置)	左結合	関数呼び出し，配列アクセス等
2	'.'	右結合	関数合成
3	'**'	右結合	べき乗
4	'*'，'/'	左結合	乗算，除算
5	'+'，'-'	左結合	加算，減算
6	'<'，'<='，'=='，'!='，'>='，'>'	多結合	比較演算子
7	'&&'	左結合	論理積
8	'||'	左結合	論理和
9	','	多結合	カンマ区切りリスト

表 9　Formura の単項演算子一覧

順位	<単項演算子>	意味
5	'+'，'-'	符号

B.8　左辺式

代入文の左辺には変数名，タプルのパターンマッチ，グリッド変数へのアクセスを書くことができる．

B.8.1　変数名

変数名は，有効な変数の識別子である．

B.8.2　タプルのパターンマッチ

タプルのパターンマッチによってタプルの要素を複数の変数に一度に割り当てることができる．例えば，`(1,2,3)` というタプルの各要素を左から順番に a,b,c という変数にそれぞれ代入する場合，`(a,b,c) = (1,2,3)` と書くことができる．代入するタプルが `(1,2,(3,4))` の場合，`(a,b,c) = (1,2,(3,4))` の結果は，`c == (3,4)` になる．

B.8.3　左辺の格子アクセス式

左辺にグリッド変数へのアクセスを書くことができる．ただし，識別子のみ記述できる．つまり `a[i] = ...` は正しいが，`a[i+1] = ...` はエラーとなる．

参考文献

[1] Eleuterio F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Springer-Verlag Berlin Heidelberg, 2009.

[2] 構造格子サブワーキンググループ. formura 文法仕様書. https://github.com/nushio3/formura/blob/master/specification/formura-specification.pdf, 2016.

[3] 構造格子サブワーキンググループ. formura ユーザーマニュアル. https://github.com/nushio3/formura/blob/master/specification/formura-users-manual.pdf, 2016.