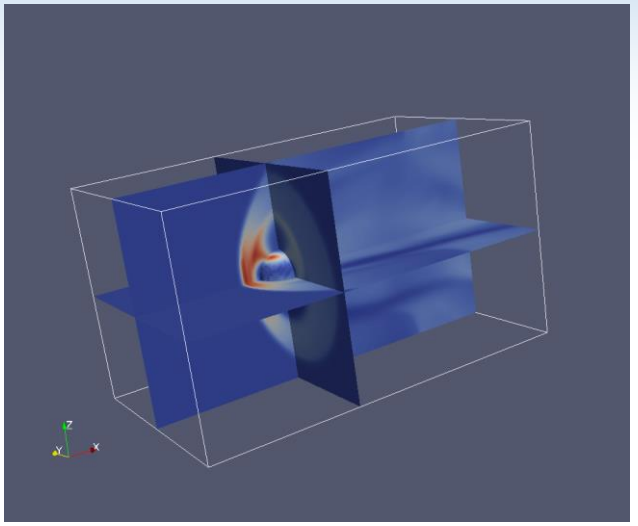
計算機シミュレーションの基礎
～コードのチューニング～

八木 学
（理化学研究所 計算科学研究センター）

KOBE HPC Spring School 2019
2019年3月13日

自己紹介



・専門は宇宙プラズマの**磁気圏領域**
　太陽起源の超音速プラズマ流と惑星磁場の
　相互作用により生じる領域

・磁気流体（MHD）シミュレーションや、テスト粒子計算を用いた水星磁気圏のモデリング

・観測データのデータベース整備、及びデータ解析ソフトウェアの開発など

・普段はFortran, IDL, Pythonを計算・解析に使用しています

サンプルコードに
関する補足

磁気流体方程式

$$\frac{\partial \rho}{\partial t} = -(\mathbf{v}\cdot\boldsymbol{\nabla})\rho - \rho(\boldsymbol{\nabla}\cdot\mathbf{v})$$

$$\frac{\partial \mathbf{v}}{\partial t} = -(\mathbf{v}\cdot\boldsymbol{\nabla})\mathbf{v} - \frac{1}{\rho}\boldsymbol{\nabla}P + \frac{1}{\rho}(\mathbf{J}\times\mathbf{B})$$

$$\frac{\partial P}{\partial t} = -(\mathbf{v}\cdot\boldsymbol{\nabla})P - \gamma P(\boldsymbol{\nabla}\cdot\mathbf{v})$$

$$\frac{\partial \mathbf{B}}{\partial t} = \boldsymbol{\nabla}\times(\mathbf{v}\times\mathbf{B})$$

基本形は一般的な流体方程式とほぼ同じ
違いは磁場に関する項（ローレンツ力）が入る程度）

サンプルコードに関する補足

磁気流体方程式

$$\frac{\partial \rho}{\partial t} = -(\mathbf{v}\cdot\nabla)\rho - \rho(\nabla\cdot\mathbf{v})$$

$$\frac{\partial \mathbf{v}}{\partial t} = -(\mathbf{v}\cdot\nabla)\mathbf{v} - \frac{1}{\rho}\nabla P + \frac{1}{\rho}(\mathbf{J}\times\mathbf{B})$$

Lorentz force

$$\frac{\partial P}{\partial t} = -(\mathbf{v}\cdot\nabla)P - \gamma P(\nabla\cdot\mathbf{v})$$

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla\times(\mathbf{v}\times\mathbf{B})$$

Induction Equation

基本形は一般的な流体方程式とほぼ同じ
違いは磁場に関する項（ローレンツ力）が入る程度）

サンプルコードに関する補足

磁気流体方程式

$$\frac{\partial \rho}{\partial t} = -(\mathbf{v}\cdot\nabla)\rho - \rho(\nabla\cdot\mathbf{v})$$

$$\frac{\partial \mathbf{v}}{\partial t} = -(\mathbf{v}\cdot\nabla)\mathbf{v} - \frac{1}{\rho}\nabla P + \frac{1}{\rho}(\mathbf{J}\times\mathbf{B})$$

~~Lorentz force~~

$$\frac{\partial P}{\partial t} = -(\mathbf{v}\cdot\nabla)P - \gamma P(\nabla\cdot\mathbf{v})$$

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla\times(\mathbf{v}\times\mathbf{B})$$

~~Induction Equation~~

基本形は一般的な流体方程式とほぼ同じ
違いは磁場に関する項（ローレンツ力）が入る程度
→今回は磁場は省略（つまり圧縮性の中性流体）

サンプルコードに関する補足

磁気流体方程式

移流項　　非移流項

$$\frac{\partial \rho}{\partial t} = -(\mathbf{v}\cdot\nabla)\rho - \rho(\nabla\cdot\mathbf{v})$$

$$\frac{\partial \mathbf{v}}{\partial t} = -(\mathbf{v}\cdot\nabla)\mathbf{v} - \frac{1}{\rho}\nabla P + \frac{1}{\rho}(\mathbf{J}\times\mathbf{B})$$

~~Lorentz force~~

$$\frac{\partial P}{\partial t} = -(\mathbf{v}\cdot\nabla)P - \gamma P(\nabla\cdot\mathbf{v})$$

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla\times(\mathbf{v}\times\mathbf{B})$$

~~Induction Equation~~

基本形は一般的な流体方程式とほぼ同じ
違いは磁場に関する項（ローレンツ力）が入る程度
→今回は磁場は省略（つまり圧縮性の中性流体）

サンプルコードに関する補足

差分法

・もっともシンプル（原始的）かつ汎用性のあるスキーム
・故に大抵の時間発展方程式に適用可能

$$\frac{\partial f}{\partial t} = F\left(f, \frac{\partial f}{\partial x}, \frac{\partial^2 f}{\partial x^2}, \dots\right)$$

空間方向：中心差分

二次精度：$\frac{\partial f}{\partial x} = \frac{f_{i+1} - f_{i-1}}{2\Delta x}$

四次精度：$\frac{\partial f}{\partial x} = \frac{-f_{i+2} + 8f_{i+1} - 8f_{i-1} + f_{i-2}}{12\Delta x}$

左右2グリッドを計算に使用

時間積分：

オイラー法：$f^{n+1} = f^n + F(x)\Delta t$

ルンゲクッタ（4次）：

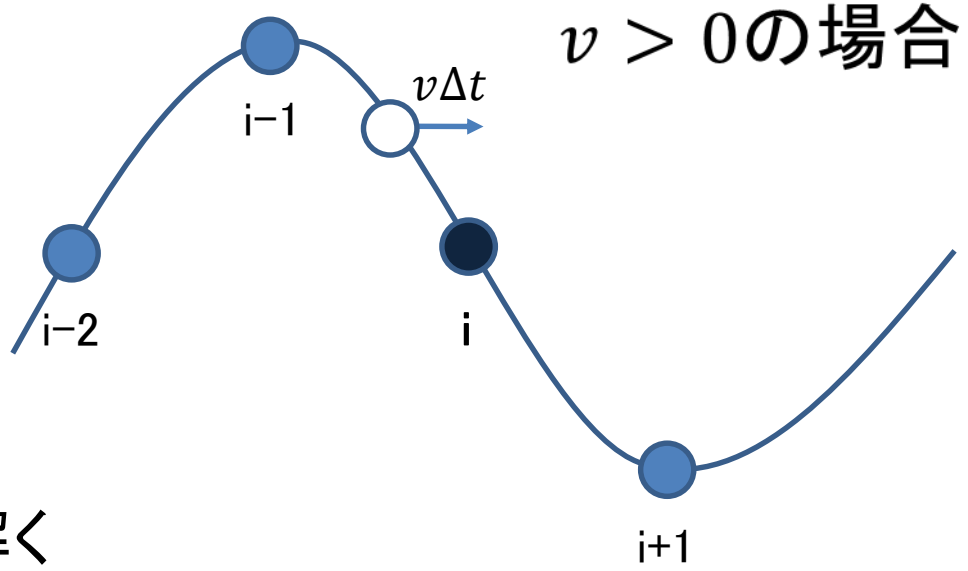
$$k_1 = S(f^n, t_n)$$
$$k_2 = S\left(f^n + \frac{k_1}{2}\Delta t, t_n + \frac{1}{2}\Delta t\right)$$
$$k_3 = S\left(f^n + \frac{k_2}{2}\Delta t, t_n + \frac{1}{2}\Delta t\right)$$
$$k_4 = S(f^n + k_3\Delta t, t_n + \Delta t)$$
$$f^{n+1} = f^n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\Delta t$$

セミラグランジュ法（移流）

サンプルコードに関する補足

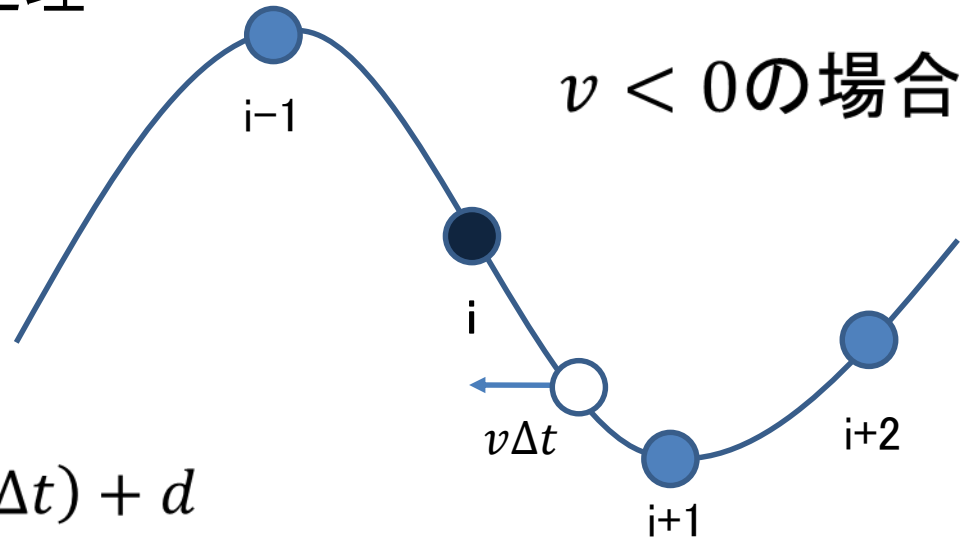
$$\frac{\partial f}{\partial t} = -v\frac{\partial f}{\partial x}$$

$$\therefore f = e^{-i(x-vt)}$$



セミラグランジュ法：

- 方程式から移流項のみを抽出して解く
- それ以外の部分は、普通の差分で処理
- 物理量をグリッド間で補間（風上型）
- 速度の符号から風上を判定



3次精度（3次関数による補間）

$$\mathrm{F(x)} = \mathrm{ax^3 + bx^2 + cx + d}$$

$$f^{n+1} = a(-v\Delta t)^3 + b(-v\Delta t)^2 + c(-v\Delta t) + d$$

チューニングの必要性について

・大規模シミュレーションを行う上で重要
　→数カ月間ジョブを流すようなケースでは研究
　　効率を大きく左右する
　→競争の激しい業界では、素早く計算結果を
　　出す必要がある

とはいえ・・・

・計算を１日早くするためのチューニングに１年かかってしまうようでは『失敗』

・チューニングはコストパフォーマンスを重視

チューニングの方針

・最近のコンパイラの最適化能力は高い
　→コンパイラが最適化しやすいコードを書く

・処理時間の最も重い箇所（ホットスポット）を重点的にチューニングする

・質の良い高速なライブラリは積極的に利用する
　→利用頻度の高い典型問題（フーリエ変換、固有値問題など）は、開発も活発

チューニングの方針

・チューニングを行うとコードの可読性が犠牲になることもあり、大規模なアルゴリズムの改良は難しくなる
　→アルゴリズムはチューニングする前に検討
　→バグのチェックも同様

・まずは可読性の高いコードを書き、アルゴリズムの検討やバグ出しを終えたところで最後にチューニングを行う。

コンパイラによる最適化

・コンパイル時にオプションを指定することで、コンパイラが自動的に最適化してくれる。
  ・インライン展開
  ・SIMD化
  ・自動並列化

・自動最適化は計算順序の入れ替えなども行うため、稀に計算結果が変わってしまうことがある。

・明示的に最適化レベルを0にする（-O0）ことで、結果が変わっていないか確認できる。

コンパイラによる最適化

・コンパイラによっては、強力なコンパイルオプションをまとめたオプションを用意していることもある

Intel compiler : –fast

FUJITSU (京, FX10等) : –Kfast

・デフォルトで最適化オプションが指定されている場合も多い(–O2, –O3等)

・詳しくは各コンパイラのマニュアル参照(スパコンを使う場合、スパコンマニュアルを参照するとよい)

SIMD: Single Instruction Multiple Data

INTEL系：SSE, AVX

・1つの命令を同時に複数のデータに対して行う演算、すなわち一種のベクトル計算である。

・ベクトル長は2~4程度と所謂ベクトル計算機に比べると短いが、それでもSIMD化できるのとできないのでは数倍速度が変わる。

・コンパイルオプションで有効化（例）

gcc : -mavx

Intel : -xHost

FUJITSU : -Ksimd=1

プロファイラの使用

各サブルーチンの経過時間等を計測
Linuxでは gprof コマンドを利用可能

gfortran, Intel fortranの場合

```
gfortran -pg test.f90
# ifort -p test.f90
./a.out
gprof ./a.out gmon.out
```

富士通システム（京、FX等）

```
fipp -C -d <dir> ./a.out                  # 非MPI
fipp -C -d <dir> mpiexec ./a.out  # MPIジョブ
fipppx -A -d <dir>
```

表示例：

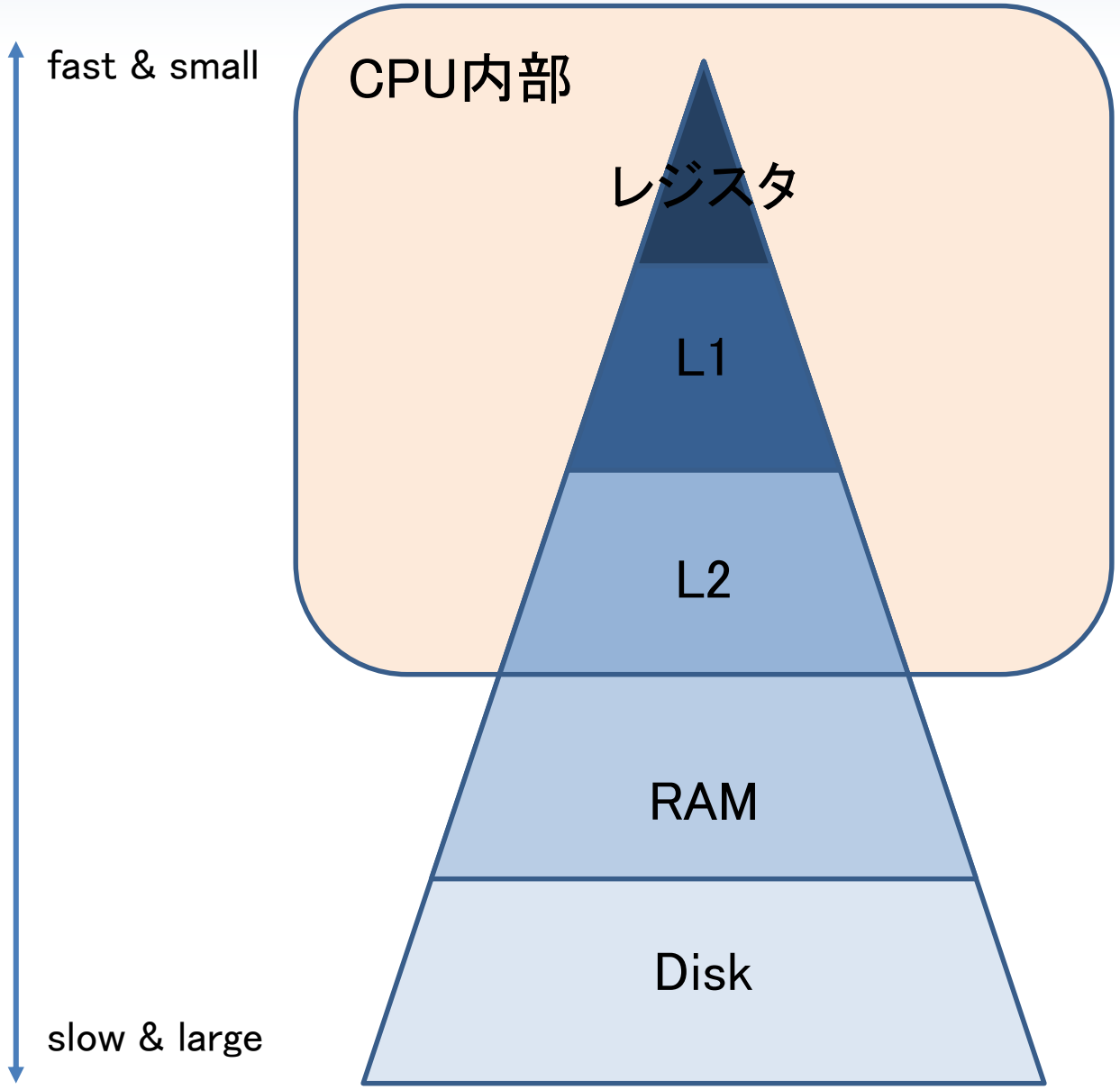
```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
```

各ルーチンの実行時間（割合）

呼び出し回数

ルーチン名

メモリの階層構造



・上に行くほど高速・低容量

・階層により、アクセス速度が桁で変わる（が、容量も桁で落ちる）

アクセス速度

L2キャッシュ　＞ 100GB/sec

RAM　10GB~ /sec

HDD　100MB~ /sec

割り算・べき乗

除算は加算や乗算に比べて計算コストが重いため、乗算の形で表記した方が速い

べき乗表記も可能であれば置き換える

```
do i=1,nx
  f(i) = x(i)/c
enddo
```

```
d = 1.0/c
do i=1,nx
  f(i) = x(i) * d
enddo
```

```
do i=1,nx
  f(i) = x(i)**2
  g(i) = x(i)**0.5
enddo
```

```
do i=1,nx
  f(i) = x(i)*x(i)
  g(i) = sqrt((i))
enddo
```

多項式の計算

多項式は因数分解したほうが計算量が減る。
丸め誤差により結果が変わってしまうこともあるため、注意する必要がある。

```
do i=1,nx
  f(i) = a*x(i)*x(i)*x(i) + b*x(i)*x(i) + c*x(i) + d
enddo
```

```
do i=1,nx
  f(i) = ((a*x(i) + b)*x(i) + c)*x(i) + d
enddo
```

ループ内の依存関係

ループ番号間に依存関係があるとSIMD化の妨げになり、実行速度が大きく損なわれる。工夫次第で依存を解消できるのであれば、書き換えたい。

```
x(1) = dx
do i=2,nx
  x(i) = x(i-1) + dx
enddo
```

```
do i=1,nx
  x(i) = i * dx
enddo
```

実際には、こんな単純に行くことは少ないが・・・

ループ内IF文の除去

- 計算ループ内のIF文は、コンパイラによる自動最適化（SIMD化）の妨げやすい。別の記述で置き換えた方がよい場合もある。
- min, max, sign等で代用できるケースあり。

```
do i=1,nx
  if (x(i) > 0.0) then
    f(i) = c * x(i)
  else
    f(i) = -c * x(i)
  endif
enddo
```

```
do i=1,nx
  f(i) = sign(1.0,x(i))*c*x(i)
enddo
```

ただし関数の方が遅くなることもあり、実際に試すことが重要

ループ内IF文の除去

- 計算ループ内のIF文は、コンパイラによる自動最適化（SIMD化）の妨げやすい。別の記述で置き換えた方がよい場合もある。
- min, max, sign等で代用できるケースあり。

```
do i=1,nx
  if (x(i) > 0.0) then
    f(i) = c * x(i)
  else
    f(i) = d * x(i)
  endif
enddo
```

```
do i=1,nx
  f(i) = 0.5 * x(i) &
       *(c+d+(c-d)*sign(1.0, x(i)))
enddo
```

ただし関数の方が遅くなることもあり、実際に試すことが重要

演習問題1-1

サンプルコード（1d_adv.f90、1d_adv.c）には改善の余地が多く存在している。これを修正・実行し、元のコードと計算時間を比較せよ。ただしコンパイル時には自動最適化を無効にすること（-O0を指定）。

（ヒント）
- 多項式は因数分解可能
- 割り算は、掛け算の形式に変換
- ループ内のif文をsign等を使って置き換える(fortran)

解答例：1d_adv_mod.f90, 1d_adv_mod.c

補足:マシン上の操作

ファイルのコピー

ログイン

ssh -X guestxx@172.25.61.11

演習用のディレクトリの作成

> mkdir 2019spring

> cd 2019spring

演習用のファイルを自分のhomeディレクトリにコピー

> cp /tmp/2019spring/code.tar ./

> tar xvf code.tar

補足：マシン上の操作

″code/f90″のディレクトリ構成

basic： 非並列プログラム
openmp： OpenMP用のプログラム
hybrid： ハイブリッド並列用のプログラム

```
> cd code/f90/basic
> ls
1d_adv.f90    1d_adv_mod.f90    1d_fluid_simple.f90.....
```

補足：マシン上の操作

″code/c″のディレクトリ構成

basic： 非並列プログラム
openmp： OpenMP用のプログラム
hybrid： ハイブリッド並列用のプログラム

```
> cd code/c/basic
> ls
1d_adv.c    1d_adv_mod.c    1d_fluid_simple.c .....
```

補足：マシン上の操作

コンパイル(Fortran)：Intelコンパイラを使用

> module load intel
> ifort **-O0 1d_adv.f90**

コンパイル(C言語)：Intelコンパイラを使用

> module load intel
> **icc -O0 1d_adv.c**

-O0 : 自動最適化をしない
-fast : 自動最適化オプション詰め合わせ

＊詳しくはマニュアル参照

補足：マシン上の操作

ジョブスクリプトの投入

> qsub run.sh

run.sh：スケジューラへの指令（シェルスクリプト）

```
#!/bin/sh
#PBS -q S                                  キュー指定
#PBS -l select=1:ncpus=1                   CPU数
#PBS -N basic                              ジョブ名
#PBS -o serial_out_file                    標準出力の出力先
#PBS -j oe                                 エラー出力を標準出力に
source /etc/profile.d/modules.sh
module load intel                          Intelコンパイラ環境のロード
cd ${PBS_O_WORKDIR}
dplace time ./a.out                        実行
```

時間計測

補足：マシン上の操作

プログラム一覧：

1d_adv	：	**１次元移流方程式**
1d_adv_mod	：	１次元移流方程式（改善案）
1d_fluid_simple	：	１次元流体（オイラー法、空間２次差分）
1d_fluid_rk	：	１次元流体（ルンゲクッタ、空間４次差分）
2d_fluid	：	２次元流体（ルンゲクッタ、空間４次差分）

サンプルプログラムの解説

データ出力先：

1d_adv ⇒ dat_1d_adv

1d_fluid_simple

1d_fluid_rk ⇒ dat_1d_fluid

2d_v101.f90 ⇒ dat_2d_fluid

サンプルプログラムの解説

プログラム中のパラメータ（移流）：1d_adv

nx：空間グリッド数

delt：時間刻み幅（Δt）

velc：速度（v）

プログラム中のパラメータ（流体）：1d_fluid, 2d_fluid

ff(… , ieq), gf(… , ieq)

ieq=1: 密度（ρ）

ieq=2: 圧力（P）

ieq=3: Vx

ieq=4: Vy（2次元のみ）

補足：マシン上の操作

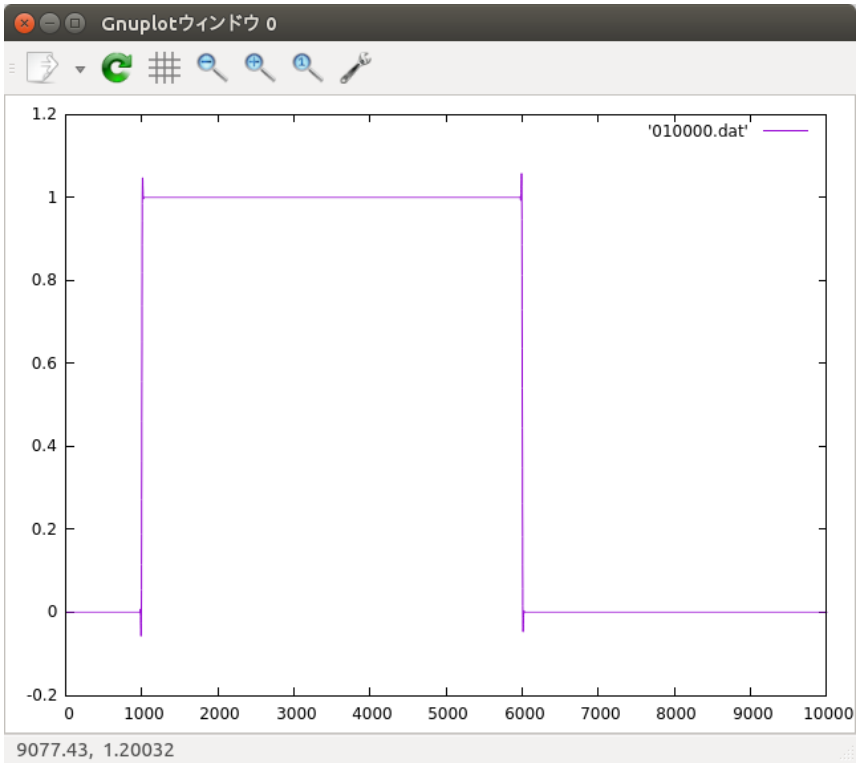
データ出力の確認（Pythonプログラム）

1次元

> cd ~/2019spring/code/f90/basic/dat_1d_adv
> gnuplot

gnuplot> plot "010000.dat" with lines

ファイル名



補足：マシン上の操作

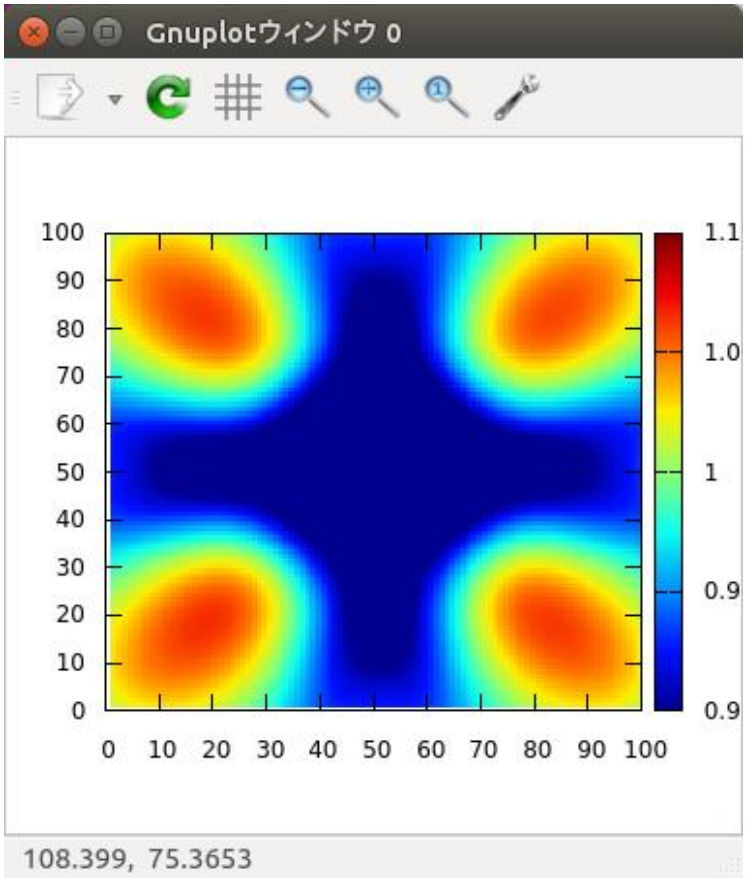
データ出力の確認（Pythonプログラム）

2次元

```
> cd ~/2019spring/code/f90/basic/dat_2d_fluid
> gnuplot

gnuplot> set pm3d map
gnuplot> set ticslevel 0
gnuplot> set palette defined ( 0 '#000090',1
'#000fff',2 '#0090ff',3 '#0fffee',4 '#90ff70',5
'#ffee00',6 '#ff7000',7 '#ee0000',8 '#7f0000')
gnuplot> splot "pre_001000.dat" with pm3d
```

ファイル名



インライン展開

外部関数やサブルーチンはプログラムの可読性向上に役立つが、計算ループ内で繰り返し呼び出す場合は、呼び出しのオーバヘッドが大きい
　→内容をその場に展開する

```
do i=1,nx
  f(i) = a + b * c(i)
  call myfunc(f(i))
enddo
```

```
subroutine myfunc(g)
  g  = g + 1.0
end subroutine
```

```
do i=1,nx
  f(i) = a + b * c(i)
  f(i) = f(i) + 1.0
enddo
```

インライン展開

・最近のコンパイラは、オプションを指定することでインライン展開もやってくれる。

コンパイラによるインライン展開の指定

gfortran -O3

ifort -ipo

frtpx -Kilfunc

・・・ただし、うまく機能しない場合もある

メモリの連続性

real(8) :: a(nx,ny)

配列のメモリ空間上での配置(Fortran)

内側の配列がメモリ空間上で連続する。

double a[nx][ny]

配列のメモリ空間上での配置(C言語)

外側の配列がメモリ空間上で連続する。

```
do i=1,nx
   do j=1,ny
      a(i, j) = i + j
  enddo
enddo
```

F

nx 間隔でメモリアクセスが飛び飛びであり、メモリへの書き込みが非常に遅い

```
do j=1,ny
   do i=1,nx
      a(i, j) = i + j
  enddo
enddo
```

F

アドレスに連続アクセスするように修正

```
for (i=0; i<nx; i++) {
   for (j=0; j<ny; j++) {
      a[i][j] = i + j
   }
}
```

C

C言語の場合は、こちらが連続

演習問題1-2

演習：
２次元流体コード2d_fluid.f90(Fortran), 2d_fluid.c(C言語)において

多重ループ計算においてメモリアクセスが連続していない箇所が多々存在している。これを改善し、速度を計測せよ

Byte/Flopの概念

１回の演算に対して必要なメモリアクセス量を
Byte/Flopで定義する

```
do i = 1, nx
   a(i) = a(i) + b * c(i)
enddo
```

・この例では１回のループにつき、２回の浮動小数点演算に対して３回のデータロードが必要。
　→倍精度とすると 3*8Byte / 2Flop = 12Byte/Flop

・コードはByte/Flopは小さいほうが良い

ループアンローリング

Do/For ループを展開し、ループ内の実行文を増やす
- ループオーバーヘッドの減少（ループ毎に発生するループの終了条件テストを削減）
- レジスタブロッキングを行う

```
do i = 1, nx
   a(i) = a(i) + b * c(i)
enddo
```

```
do i = 1, nx, 4
   a(i) = a(i) + b * c(i)
   a(i+1) = a(i+1) + b * c(i+1)
   a(i+2) = a(i+2) + b * c(i+2)
   a(i+3) = a(i+3) + b * c(i+3)
enddo
```

ループアンローリング

注意点:

- アンロールの段数に対して十分なレジスタが必要
- ループ回数はアンロール段数で割り切れなければならない
- コンパイラの最適化で自動的に行われることもある

```
do i = 1, nx
   a(i) = a(i) + b * c(i)
enddo
```

```
do i = 1, nx, 4
   a(i) = a(i) + b * c(i)
   a(i+1) = a(i+1) + b * c(i+1)
   a(i+2) = a(i+2) + b * c(i+2)
   a(i+3) = a(i+3) + b * c(i+3)
enddo
```

ループ分割・融合

2つのループを1つにまとめることで、オーバーヘッドを減らす（ループ融合）

```
do i = 1, nx
   a(i) = b(i) * c(i)
enddo
do i = 1, nx
   d(i) = e(i) * f(i)
enddo
```

```
do i = 1, nx
   a(i) = b(i) * c(i)
   d(i) = e(i) * f(i)
enddo
```

ループ内の依存関係によりSIMD化されない場合がある
両者を分けることでSIMD化する（ループ分配）

参考

・スカラーチューニングとOpenMPによるコードの高速化 （松本洋介）
http://www.astro.phys.s.chiba-u.ac.jp/hpci/ss2013/presen-ymatsumoto.pdf

・チューニング技法入門 （青山幸也）
http://accc.riken.jp/wp-content/uploads/2015/06/secure_4467_tuning-technique_main.pdf