

プログラミング実習 期末レポート

粒子法による水ロケットの推力計算

佐々木良輔

2020 年 8 月 5 日

1. 背景

水ロケットは圧縮空気で水などの液体を噴射し、その反作用によって飛翔するロケット模型であり、宇宙工学の教材として世界中で幅広く用いられている。

水ロケットの競技会では、しばしば地上に設置したターゲットを狙って水ロケットを発射し、どれだけターゲットに近く着地したかを競うことがある。こうした競技会でより高いスコアを狙う場合、弾道の予測が有効であり、そのためには水ロケットの推力を予測する必要があるが、水ロケットの推力や Δv を与える計算式は一般に知られていない。

したがって、本研究では CFD により圧力や水の量を変化させたときの水ロケットの推力計算を試みる。ただし、水ロケットの推力は水の噴射による反作用のみで生じるものとして、空気の噴射は考慮しない。また水を加圧する圧力は一定とする。

2. ソルバーについて

CFD ソルバーには大きく分けて粒子法と格子法が存在するが、本研究では粒子法を用いる。粒子法とは流体を多数の粒子の集合であるとし、各粒子について計算を行うことで流体の動きをシミュレートする手法である。一方で格子法とは事前に定められた計算領域を格子状の計算点に分割し、固定された計算点において計算を行う手法である。格子法に比べて粒子法は、事前に解析領域を定める必要がない、飛散などの現象を再現できる、メッシュの切り方に依存しないなどの特徴がある。一方で粒子法は各粒子の座標を変数として取り扱うため、メモリの使用量が増加する。

今回は非圧縮性流れを扱うため MPS (Moving Particle Semi-implicit) 法を用いる。[2][1] MPS 法では流体の圧力を粒子数密度を用いて表し、これを一定とするように計算することで非圧縮性を課す。

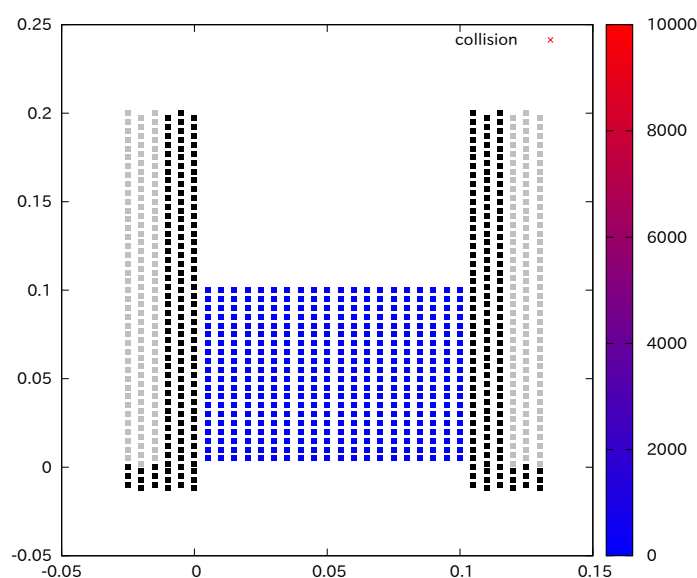


図 1 検証モデル (青点:流体粒子, 灰点:ダミー粒子, 黒点:壁粒子)[1]

水は非圧縮性流体とできるので, Navier-Stokes 方程式は

$$8 \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} = -\frac{1}{\rho} \nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f} \quad (1)$$

ここで \mathbf{u} は速度である. 連続の式は

$$\frac{\partial \rho}{\partial t} + \rho \nabla \cdot \mathbf{u} = 0 \quad (2)$$

$$\nabla \cdot \mathbf{u} = 0 \quad (3)$$

したがって

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{1}{\rho} \nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f} \quad (4)$$

と表される.

MPS 法では粒子間相互作用モデルを用いる. 粒子間相互作用モデルでは重み係数 w を用いて粒子間の相互作用に重みをつける. 重み係数 w を以下のように定義する.[6][5]

$$w(r) = \begin{cases} \frac{r_e}{r} - 1 & 0 \leq r \leq r_e \\ 0 & r_e < r \end{cases} \quad (5)$$

r は粒子間の距離である. これは粒子間距離が r_e 以下の粒子同士が相互作用することを表している. これを用いて粒子数密度 n_i を以下のように定義する.

$$n_i = \sum_{j \neq i} w(|\mathbf{r}_i - \mathbf{r}_j|) \quad (6)$$

n_i が初期条件 n^0 と一致するようにすることで非圧縮性条件を課す. 粒子間相互作用モデルでは各演算子を以下のように離散化する.[4]

$$\langle \nabla f \rangle_i = \frac{d}{n^0} \sum_{j \neq i} \left(\frac{f(\mathbf{x}_j) - f(\mathbf{x}_i)}{|\mathbf{x}_j - \mathbf{x}_i|^2} (\mathbf{x}_j - \mathbf{x}_i) w(|\mathbf{x}_j - \mathbf{x}_i|) \right) \quad (7)$$

$$\langle \nabla \cdot \mathbf{u} \rangle_i = \frac{2d}{n^0} \sum_{j \neq i} \frac{\mathbf{u} \cdot (\mathbf{x}_j - \mathbf{x}_i)}{|\mathbf{x}_j - \mathbf{x}_i|^2} w(|\mathbf{x}_j - \mathbf{x}_i|) \quad (8)$$

$$\langle \nabla^2 f \rangle_i = \frac{2d}{\lambda n^0} \sum_{j \neq i} (f(\mathbf{x}_j) - f(\mathbf{x}_i)) w(|\mathbf{x}_j - \mathbf{x}_i|) \quad (9)$$

$$\text{where } \lambda = \frac{\sum_{j \neq i} |\mathbf{x}_j - \mathbf{x}_i|^2 w(|\mathbf{x}_j - \mathbf{x}_i|)}{\sum_{j \neq i} w(|\mathbf{x}_j - \mathbf{x}_i|)}$$

ここで d は空間の次元であり, 今回は $d = 2$ である.

MPS 法は Semi-implicit とあるとおり、陰解法と陽解法を組み合わせで計算する。陽解法とは、現在のタイムステップの値のみを用いて次のタイムステップの値を決定する手法である。一方で陰解法ではまず現在のタイムステップから次のベクトル場を仮に決定し、それに基づいて計算する。その後、条件を用いて補正するのが陰解法である。MPS 法では Navier-Stokes 方程式の第 2 項 (粘性項) と第 3 項 (外力項) を陽的に解き、第 1 項 (移流項) と連続の式を陰的に解く。

(7) から (9) 式を用いて (3), (4) 式の離散化をする。

$$\frac{\Delta \mathbf{u}}{\Delta t} = \left[-\frac{1}{\rho} \nabla p \right]^{k+1} + [\mu \nabla^2 \mathbf{u}]^k + [\mathbf{f}]^k \quad (10)$$

$$[\nabla \cdot \mathbf{u}]^{k+1} = 0 \quad (11)$$

$[\phi]^k$ は ϕ のタイムステップ k 時点での値を指す。移流項、連続の式は陰的に解くため $k+1$ ステップである。 \mathbf{u} を移流項、粘性項と外力項成分に分け、それぞれ \mathbf{u}^* , \mathbf{u}' とする。

$$\frac{\Delta \mathbf{u}'}{\Delta t} = \frac{\mathbf{u}^{k+1} - \mathbf{u}^*}{\Delta t} = \left[-\frac{1}{\rho} \nabla p \right]^{k+1} \quad (12)$$

$$\frac{\Delta \mathbf{u}^*}{\Delta t} = \frac{\mathbf{u}^* - \mathbf{u}^{k+1}}{\Delta t} = [\mu \nabla^2 \mathbf{u}]^k + [\mathbf{f}]^k \quad (13)$$

ここで \mathbf{u}^* は仮速度であり、移流項での補正を行っていない速度ベクトルである。(13) 式において未知数は \mathbf{u}^* のみなので、計算することができる。 \mathbf{u}^* から速度を計算した後、接近した粒子間に弾性衝突を仮定し、速度と位置を修正する。これを行うことで、後の圧力計算において粒子数密度が以上に高くなるのを避け、時間間隔を大きくすることができる。また (12) 式に辺々 ∇ をかけると

$$\frac{\langle \nabla \cdot \mathbf{u} \rangle^{k+1} - \langle \nabla \cdot \mathbf{u} \rangle^*}{\Delta t} = -\frac{1}{\rho} \langle \nabla^2 p \rangle^{k+1} \quad (14)$$

ここで連続の式から $\langle \nabla \cdot \mathbf{u} \rangle^{k+1} = 0$ なので

$$\langle \nabla^2 p \rangle^{k+1} = \rho \frac{\langle \nabla \cdot \mathbf{u} \rangle^*}{\Delta t} \quad (15)$$

となる。つぎに (2) 式を離散化すると

$$\frac{\rho^* - \rho}{\Delta t} + \rho \langle \nabla \cdot \mathbf{u} \rangle^* = 0 \quad (16)$$

$$\langle \nabla \cdot \mathbf{u} \rangle^* = -\frac{1}{\Delta t} \frac{\rho^* - \rho}{\rho} \quad (17)$$

ここで密度 ρ が粒子数密度 n_i と比例することを用いて

$$\langle \nabla \cdot \mathbf{u} \rangle^* \simeq -\frac{1}{\Delta t} \frac{n^* - n^0}{n^0} \quad (18)$$

これと (15) 式から

$$\langle \nabla^2 p \rangle^{k+1} = -\frac{\rho^0}{\Delta t^2} \frac{n^* - n^0}{n^0} \quad (19)$$

以上と (7) から (9) 式を用いることで、実際に計算機上で連立方程式として圧力を計算することができる。よって (12) 式, (13) 式が解け、タイムステップの計算が終了する.[3]

3. 初期状態・推力計算

3次元では計算量が非常に膨大になるため、ここでは図1のような平面の筒で水ロケットを近似する。

- 壁面はダミー粒子3層と壁粒子3層から成る。
- 筒の直径は0.1 mとする。
- 水を加圧する圧力は1000 Pa, 2000 Pa, 3000 Pa, 4000 Paとする。
- 水粒子は筒の中の $0 \leq y \leq 0.1$, $0 \leq y \leq 0.2$, $0 \leq y \leq 0.3$ に等間隔で分布する。また、水は $0 \leq z \leq 0.1$ の厚さで存在するものとする。
- 粒子の重さは $\frac{\text{水の密度} \times \text{初期状態の水の体積}}{\text{水粒子数}}$ で求める。

また $y \leq -0.01$ となった粒子の運動量を各タイムステップにおける推力とし、その合計を Δv とする。

4. 境界条件

境界条件として、Dirichlet 境界条件と Neumann 境界条件を課す。Dirichlet 境界条件とは基準となる液面での圧力を設定するものである。具体的には、粒子数密度が一定以下の粒子が液面にあると判定し、この粒子に対する圧力を与える。Neumann 境界条件とは壁面付近での圧力勾配を0とするもので、粒子が壁面を透過しないことを課す。具体的には、壁粒子の更に外側にダミー粒子を配置し、この粒子と近傍の粒子の圧力値を等しくすることで近似的に Neumann 境界条件を課す。

5. プログラム評価

CFD では計算点の数が膨大であり、誤差を厳密に評価するのは困難である。したがって、プログラム・計算結果の正しさについて以下の点で検証する。

- 粒子の大きさを変化させたとき発生する Δv が一定である。
- 時間間隔を変化させたとき発生する Δv が一定である。

6. 結果

7. 考察

8. 結論

参考文献

- [1] 粒子法 – el-ement blog. <http://el-ement.com/blog/2017/12/19/particle-method/>. (Accessed on 06/21/2020).
- [2] 粒子法による流れの数値解析. <http://www.nagare.or.jp/download/noauth.html?d=21-3-t02.pdf&dir=31>. (Accessed on 06/20/2020).
- [3] 越塚 誠一室谷 浩平. 粒子法入門 流体シミュレーションの基礎から並列計算と可視化まで . 丸善出版, 2014.
- [4] 東京工業大学デジタル創作同好会. そばやのワク ワク流体シミュレーション~mps 編~ — 東京工業大学デジタル創作同好会 trap. <https://trap.jp/post/345/>, 11 2017. (Accessed on 06/24/2020).
- [5] 物理ベースコンピュータグラフィックス研究室. Mps 法の理論 - pukiwiki for pbeg lab. <http://www.slis.tsukuba.ac.jp/~fujisawa.makoto.fu/cgi-bin/wiki/index.php?MPS%CB%A1%A4%CE%CD%FD%CF%CO>. (Accessed on 06/20/2020).
- [6] 廣瀬三平. 流体シミュレーションにおける様々な手法. https://www.imi.kyushu-u.ac.jp/PDF/Hirose_20160612.pdf, 6 2016. (Accessed on 06/24/2020).

付録 A ソースコード

ソースコード 1 メインプログラム

```
1 program main
2   use consts_variables
3   implicit none
4   integer, parameter :: interval = 10
5   integer :: i, j
6   !call omp_set_num_threads(8)
7
8   dt = 0.0005
9   call InitParticles()
10  call calcConsts()
11  call output(0)
12  do i = 1, 800
13    do j = 1, interval
14      call calcGravity()
15      call calcViscosity()
16      call moveParticleExplicit()
17      call collision()
18      call calcNumberDensity()
19      call setBoundaryCondition()
20      call setSourceTerm()
21      call setMatrix()
22      call GaussEliminateMethod()
23      call removeNegativePressure()
24      call setMinPressure()
25      call calcPressureGradient()
26      call moveParticleImplicit()
27    enddo
28    write (*, *) "Timestep: ", i*interval
29    call output(i)
30  enddo
31
32 end
```

ソースコード 2 ファイル出力用ルーチン

```
1 subroutine output(i)
2   use consts_variables
3   implicit none
```

```

4  integer :: i, k
5  character :: filename*128
6
7  write (filename, '("data2/data", i4.4, ".dat")') i
8  open (11, file=filename, status='replace')
9  do k = 1, NumberOfParticle
10     write (11, '(f6.3,X)', advance='no') Pos(k, 1)
11     write (11, '(f6.3,X)', advance='no') Pos(k, 2)
12     write (11, '(I2,X)', advance='no') ParticleType(k)
13     write (11, '(f15.5,X)', advance='no') Pressure(k)
14     if (CollisionState(k) .eqv. .true.) then
15         write (11, '(I2)', advance='no') 1
16     else
17         write (11, '(I2)', advance='no') 0
18     endif
19     write (11, *)
20 enddo
21 close (11)
22
23 end

```

ソースコード 3 定義値

```

1 module define
2   implicit none
3   integer, parameter :: PARTICLE_DUMMY = -1, PARTICLE_WALL = 1, PARTICLE_FLUID
      = 0
4   integer, parameter :: BOUNDARY_DUMMY = -1, BOUNDARY_SURFACE = 1,
      BOUNDARY_INNER = 0
5
6 end

```

ソースコード 4 定数・変数の宣言

```

1 module consts_variables
2   implicit none
3   !initial value of particle distance
4   real*8, parameter :: PARTICLE_DISTANCE = 0.025
5   real*8, parameter :: FLUID_DENSITY = 1000
6   real*8, parameter :: KINEMATIC_VISCOSITY = 1.0e-6
7   !threshold of whether the particle is surface or inside
8   real*8, parameter :: THRESHOLD_RATIO_BETA = 0.97
9   !圧力計算の緩和係数
10  real*8, parameter :: RELAXATION_COEF_FOR_PRESSURE = 0.2

```



```

11  !圧縮率 (Pa-1)
12  real*8, parameter :: COMPRESSIBILITY = 0.45e-9
13  real*8, parameter :: IMPACT_PARAMETER = 1.0
14  integer, parameter :: MaxNumberOfParticle = 500000
15  integer, parameter :: numDimension = 2
16
17  real*8 :: dt
18  real*8 :: Radius_forNumberDensity, Radius_forGradient, Radius_forLaplacian
19  real*8 :: NO_forNumberDensity, NO_forGradient, NO_forLaplacian
20  real*8 :: Lambda
21  real*8 :: collisionDistance
22
23  real*8 :: Pos(MaxNumberOfParticle, numDimension)
24  real*8 :: Gravity(numDimension)
25  ! 0:Fluid, 1:Wall, 2:Dummy
26  integer :: ParticleType(MaxNumberOfParticle)
27  data Gravity/0d0, -9.8/
28  real*8, allocatable :: Vel(:, :)
29  real*8, allocatable :: Acc(:, :)
30  real*8, allocatable :: Pressure(:)
31  real*8, allocatable :: MinPressure(:)
32  real*8, allocatable :: NumberDensity(:)
33  integer, allocatable :: BoundaryCondition(:)
34  real*8, allocatable :: SourceTerm(:)
35  real*8, allocatable :: CoefficientMatrix(:, :)
36  logical, allocatable :: CollisionState(:)
37
38  integer :: NumberOfParticle
39
40 end

```

ソースコード 5 重み係数計算関数

```

1 real*8 function calcWeight(distance, radius)
2   implicit none
3   real*8, intent(in) :: distance, radius
4   real*8 :: w
5
6   if (distance >= radius) then
7     w = 0
8   else
9     w = radius/distance - 1d0
10  endif

```

```

11  calcWeight = w
12  return
13
14 end function

```

ソースコード 6 粒子間距離計算関数

```

1  real*8 function calcDistance(i, j)
2    use consts_variables
3    implicit none
4    real*8 :: distance2
5    integer :: i, j, k
6
7    distance2 = 0d0
8    do k = 1, numDimension
9      distance2 = distance2 + (Pos(j, k) - Pos(i, k))**2
10   enddo
11   calcDistance = sqrt(distance2)
12   return
13
14 end function

```

ソースコード 7 粒子初期化ルーチン

```

1  subroutine InitParticles()
2    !粒子の初期値設定
3    use define
4    use consts_variables
5    implicit none
6    real*8 :: x, y
7    real*8 :: EPS
8    integer :: iX, iY
9    integer :: nX, nY
10   integer :: i = 1
11   logical :: flagOfParticleGenerarion
12
13   EPS = PARTICLE_DISTANCE*0.01
14   nX = int(1.0/PARTICLE_DISTANCE) + 5
15   nY = int(0.6/PARTICLE_DISTANCE) + 5
16   do iX = -4, nX
17     do iY = -4, nY
18       x = PARTICLE_DISTANCE*dble(iX)
19       y = PARTICLE_DISTANCE*dble(iY)
20       flagOfParticleGenerarion = .false.

```

```

21
22     !dummy particle
23     if (((x > -4*PARTICLE_DISTANCE + EPS) .and. (x <= 1d0 + 4*
24         PARTICLE_DISTANCE + EPS)) &
25         .and. ((y > 0d0 - 4*PARTICLE_DISTANCE + EPS) .and. (y <= 0.6 + EPS
26             ))) then
27         ParticleType(i) = PARTICLE_DUMMY
28         flagOfParticleGenerarion = .true.
29     endif
30
31     !wall particle
32     if (((x > -2*PARTICLE_DISTANCE + EPS) .and. (x <= 1d0 + 2*
33         PARTICLE_DISTANCE + EPS)) &
34         .and. ((y > 0d0 - 2*PARTICLE_DISTANCE + EPS) .and. (y <= 0.6 + EPS
35             ))) then
36         ParticleType(i) = PARTICLE_WALL
37         flagOfParticleGenerarion = .true.
38     endif
39
40     !wall particle
41     if (((x > -4*PARTICLE_DISTANCE + EPS) .and. (x <= 1d0 + 4*
42         PARTICLE_DISTANCE + EPS)) &
43         .and. ((y > 0.6 - 2*PARTICLE_DISTANCE + EPS) .and. (y <= 0.6 + EPS
44             ))) then
45         ParticleType(i) = PARTICLE_WALL
46         flagOfParticleGenerarion = .true.
47     endif
48
49     !empty region
50     if (((x > 0d0 + EPS) .and. (x <= 1d0 + EPS)) .and. (y > 0d0 + EPS))
51         then
52             flagOfParticleGenerarion = .false.
53     endif
54
55     !fluid particle
56     if (((x > 0d0 + EPS) .and. (x <= 0.25 + EPS)) .and. ((y > 0d0 + EPS)
57         .and. (y <= 0.5 + EPS))) then
58         ParticleType(i) = PARTICLE_FLUID
59         flagOfParticleGenerarion = .true.
60     endif
61
62     !generate position and velocity

```

```

55         if (flagOfParticleGenerarion .eqv. .true.) then
56             Pos(i, 1) = x; Pos(i, 2) = y
57             i = i + 1
58         endif
59
60     enddo
61 enddo
62
63 NumberOfParticle = i - 1
64 !allocate memory for particle quantities
65 allocate (Vel(NumberOfParticle, numDimension))
66 allocate (Acc(NumberOfParticle, numDimension))
67 allocate (NumberDensity(NumberOfParticle))
68 allocate (BoundaryCondition(NumberOfParticle))
69 allocate (SourceTerm(NumberOfParticle))
70 allocate (CoefficientMatrix(NumberOfParticle, NumberOfParticle))
71 allocate (Pressure(NumberOfParticle))
72 allocate (MinPressure(NumberOfParticle))
73 allocate (CollisionState(NumberOfParticle))
74 Vel = 0d0
75 Acc = 0d0
76 Pressure = 0d0
77
78 return
79
80 end

```

ソースコード 8 定数計算関連ルーチン

```

1 subroutine calcConsts()
2     !定数の計算
3     use consts_variables
4     implicit none
5
6     Radius_forNumberDensity = 2.1*PARTICLE_DISTANCE
7     Radius_forGradient = 2.1*PARTICLE_DISTANCE
8     Radius_forLaplacian = 3.1*PARTICLE_DISTANCE
9     collisionDistance = 0.5*PARTICLE_DISTANCE
10
11     call calcNZeroLambda()
12
13 end
14

```

```

15 subroutine calcNZeroLambda()
16   use consts_variables
17   implicit none
18   real*8 :: calcWeight
19   real*8 :: xj, yj, xi = 0d0, yi = 0d0
20   real*8 :: distance2, distance
21   integer :: iX, iY
22
23   NO_forNumberDensity = 0d0
24   NO_forGradient = 0d0
25   NO_forLaplacian = 0d0
26   Lambda = 0d0
27
28   do iX = -4, 4
29     do iY = -4, 4
30       if ((iX == 0) .and. (iY == 0)) cycle
31       xj = PARTICLE_DISTANCE*dble(iX)
32       yj = PARTICLE_DISTANCE*dble(iY)
33       distance2 = (xj - xi)**2 + (yj - yi)**2
34       distance = sqrt(distance2)
35
36       NO_forNumberDensity = NO_forNumberDensity + calcWeight(distance,
37         Radius_forNumberDensity)
38       NO_forGradient = NO_forGradient + calcWeight(distance, Radius_forGradient
39         )
40       NO_forLaplacian = NO_forLaplacian + calcWeight(distance,
41         Radius_forLaplacian)
42
43       Lambda = Lambda + distance2*calcWeight(distance, Radius_forLaplacian)
44     enddo
45   enddo
46   Lambda = Lambda/NO_forLaplacian
47 end

```

ソースコード 9 外力項計算ルーチン

```

1 subroutine calcGravity()
2   !外力項（重力）の計算
3   use define
4   use consts_variables
5   implicit none
6   integer :: i, j

```

```

7
8   Acc = 0d0
9   !$omp parallel private(j)
10  !$omp do
11  do i = 1, NumberOfParticle
12  if (ParticleType(i) == PARTICLE_FLUID) then
13      do j = 1, numDimension
14          Acc(i, j) = Gravity(j)
15      enddo
16  endif
17  enddo
18  !$omp end do
19  !$omp end parallel
20
21 end

```

ソースコード 10 粘性項計算ルーチン

```

1  subroutine calcViscosity()
2      !粘性項の計算
3      use define
4      use consts_variables
5      implicit none
6      real*8 :: ViscosityTerm(numDimension)
7      real*8 :: distance, weight
8      real*8 :: calcWeight, calcDistance
9      real*8 :: m
10     integer :: i, j, k
11
12     m = 2d0*numDimension/(NO_forLaplacian*Lambda)*KINEMATIC_VISCOSITY
13
14     !$omp parallel private(ViscosityTerm, distance, weight, j, k)
15     !$omp do
16     do i = 1, NumberOfParticle
17     if (ParticleType(i) == PARTICLE_FLUID) then
18         ViscosityTerm = 0d0
19         do j = 1, NumberOfParticle
20             if (i == j) cycle
21             distance = calcDistance(i, j)
22             if (distance < Radius_forLaplacian) then
23                 weight = calcWeight(distance, Radius_forLaplacian)
24                 do k = 1, numDimension
25                     ViscosityTerm(k) = ViscosityTerm(k) + (Vel(j, k) - Vel(i, k))*

```

```

weight
26     enddo
27     endif
28     enddo
29     do k = 1, numDimension
30         Acc(i, k) = Acc(i, k) + ViscosityTerm(k)*m
31     enddo
32     endif
33     enddo
34     !$omp end do
35     !$omp end parallel
36
37 end

```

ソースコード 11 仮速度・仮位置計算ルーチン

```

1 subroutine moveParticleExplicit()
2     !陽解法による粒子の移動
3     use consts_variables
4     implicit none
5     integer :: i, j
6
7     !$omp parallel private(j)
8     !$omp do
9     do i = 1, NumberOfParticle
10    do j = 1, numDimension
11        Vel(i, j) = Vel(i, j) + Acc(i, j)*dt
12        Pos(i, j) = Pos(i, j) + Vel(i, j)*dt
13    enddo
14    enddo
15    !$omp end do
16    !$omp end parallel
17    Acc = 0d0
18
19 end

```

ソースコード 12 衝突判定ルーチン

```

1 subroutine collision()
2     use define
3     use consts_variables
4     implicit none
5     real*8 :: e = IMPACT_PARAMETER
6     real*8 :: distance

```

```

7  real*8 :: calcDistance
8  real*8 :: impulse
9  real*8 :: VelocityAfterCollision(NumberOfParticle, numDimension)
10 real*8 :: velocity_ix, velocity_iy
11 real*8 :: xij, yij
12 real*8 :: mi, mj
13 integer :: i, j
14
15 CollisionState = .false.
16 VelocityAfterCollision = Vel
17 do i = 1, NumberOfParticle
18   if (ParticleType(i) .ne. PARTICLE_FLUID) cycle
19   mi = FLUID_DENSITY
20   velocity_ix = Vel(i, 1)
21   velocity_iy = Vel(i, 2)
22   do j = 1, NumberOfParticle
23     if (ParticleType(j) == PARTICLE_DUMMY) cycle
24     if (i == j) cycle
25     xij = Pos(j, 1) - Pos(i, 1)
26     yij = Pos(j, 2) - Pos(i, 2)
27     distance = calcDistance(i, j)
28     if (distance < collisionDistance) then
29       impulse = (velocity_ix - Vel(j, 1))*(xij/distance) + &
30               (velocity_iy - Vel(j, 2))*(yij/distance)
31       if (impulse > 0d0) then
32         CollisionState(i) = .true.
33         CollisionState(j) = .true.
34         mj = FLUID_DENSITY
35         impulse = impulse*((1d0 + e)*mi*mj)/(mi + mj)
36         velocity_ix = velocity_ix - (impulse/mi)*(xij/distance)
37         velocity_iy = velocity_iy - (impulse/mi)*(yij/distance)
38       endif
39     endif
40   enddo
41   VelocityAfterCollision(i, 1) = velocity_ix
42   VelocityAfterCollision(i, 2) = velocity_iy
43 enddo
44
45 !$omp parallel
46 !$omp do
47 do i = 1, NumberOfParticle
48   if (ParticleType(i) .ne. PARTICLE_FLUID) cycle

```



```

49     Pos(i, 1) = Pos(i, 1) + (VelocityAfterCollision(i, 1) - Vel(i, 1))*dt
50     Pos(i, 2) = Pos(i, 2) + (VelocityAfterCollision(i, 2) - Vel(i, 2))*dt
51     Vel(i, 1) = VelocityAfterCollision(i, 1)
52     Vel(i, 2) = VelocityAfterCollision(i, 2)
53 enddo
54 !$omp end do
55 !$omp end parallel
56
57 end

```

ソースコード 13 粒子数密度計算ルーチン

```

1 subroutine calcNumberDensity()
2   !粒子数密度の計算
3   use define
4   use consts_variables
5   implicit none
6   real*8 :: distance, weight
7   real*8 :: calcDistance, calcWeight
8   integer :: i, j
9
10  NumberDensity = 0d0
11  do i = 1, NumberOfParticle
12    if (ParticleType(i) == PARTICLE_DUMMY) cycle
13    do j = 1, NumberOfParticle
14      if (ParticleType(j) == PARTICLE_DUMMY) cycle
15      if (i == j) cycle
16      distance = calcDistance(i, j)
17      weight = calcWeight(distance, Radius_forNumberDensity)
18      NumberDensity(i) = NumberDensity(i) + weight
19    enddo
20  enddo
21
22 end

```

ソースコード 14 ディリクレ境界条件判定ルーチン

```

1 subroutine setBoundaryCondition()
2   !境界条件の設定
3   use define
4   use consts_variables
5   implicit none
6   integer :: i
7

```

```

8  !$omp parallel
9  !$omp do
10 do i = 1, NumberOfParticle
11  if (ParticleType(i) == PARTICLE_DUMMY) then
12    BoundaryCondition(i) = BOUNDARY_DUMMY
13  elseif (NumberDensity(i) < (THRESHOLD_RATIO_BETA*NO_forNumberDensity)) then
14    BoundaryCondition(i) = BOUNDARY_SURFACE
15  else
16    BoundaryCondition(i) = BOUNDARY_INNER
17  endif
18  enddo
19  !$omp end do
20  !$omp end parallel
21
22 end

```

ソースコード 15 ポアソン方程式右辺設定ルーチン

```

1  subroutine setSourceTerm()
2  !ポアソン方程式右辺の設定
3  use define
4  use consts_variables
5  implicit none
6  integer :: i
7
8  SourceTerm = 0d0
9  !$omp parallel
10 !$omp do
11 do i = 1, NumberOfParticle
12  if (ParticleType(i) == PARTICLE_DUMMY) cycle
13  if (BoundaryCondition(i) == BOUNDARY_SURFACE) cycle
14  if (BoundaryCondition(i) == BOUNDARY_INNER) then
15    SourceTerm(i) = RELAXATION_COEF_FOR_PRESSURE*(1d0/dt**2)* &
16      (NumberDensity(i) - NO_forNumberDensity)/
17      NO_forNumberDensity
18  endif
19  enddo
20  !$omp end do
21  !$omp end parallel
22 end

```

ソースコード 16 係数行列の計算ルーチン

```

1 subroutine setMatrix()
2   !係数行列の設定
3   use define
4   use consts_variables
5   implicit none
6   real*8 :: distance
7   real*8 :: calcDistance, calcWeight
8   real*8 :: coefIJ, a
9   integer :: i, j
10
11   CoefficientMatrix = 0d0
12   a = 2d0*numDimension/(NO_forLaplacian*Lambda)
13   do i = 1, NumberOfParticle
14     if (BoundaryCondition(i) .ne. BOUNDARY_INNER) cycle
15     do j = 1, NumberOfParticle
16       if (BoundaryCondition(j) == BOUNDARY_DUMMY) cycle
17       if (i == j) cycle
18       distance = calcDistance(i, j)
19       if (distance >= Radius_forLaplacian) cycle
20       coefIJ = a*calcWeight(distance, Radius_forLaplacian)/FLUID_DENSITY
21       CoefficientMatrix(i, j) = -1d0*coefIJ
22       CoefficientMatrix(i, i) = CoefficientMatrix(i, i) + coefIJ
23     enddo
24     CoefficientMatrix(i, i) = CoefficientMatrix(i, i) + COMPRESSIBILITY/(dt
25       **2)
26   enddo
27 end

```

ソースコード 17 ガウスの消去法による圧力計算ルーチン

```

1 subroutine GaussEliminateMethod()
2   use define
3   use consts_variables
4   implicit none
5   real*8 :: Terms, c
6   integer :: i, j, k
7
8   Pressure = 0d0
9   do i = 1, NumberOfParticle - 1
10     if (BoundaryCondition(i) .ne. BOUNDARY_INNER) cycle
11     do j = i + 1, NumberOfParticle
12       if (BoundaryCondition(j) == BOUNDARY_DUMMY) cycle

```

```

13      c = CoefficientMatrix(j, i)/CoefficientMatrix(i, i)
14      !$omp parallel
15      !$omp do
16      do k = i + 1, NumberOfParticle
17          CoefficientMatrix(j, k) = CoefficientMatrix(j, k) - c*
              CoefficientMatrix(i, k)
18      enddo
19      !$omp end do
20      !$omp end parallel
21      SourceTerm(j) = SourceTerm(j) - c*SourceTerm(i)
22  enddo
23 enddo
24
25  i = NumberOfParticle
26  do
27      i = i - 1
28      if (i == 0) exit
29      if (BoundaryCondition(i) .ne. BOUNDARY_INNER) cycle
30      Terms = 0d0
31      !$omp parallel
32      !$omp do reduction(+:Terms)
33      do j = i + 1, NumberOfParticle
34          Terms = Terms + CoefficientMatrix(i, j)*Pressure(j)
35      enddo
36      !$omp end do
37      !$omp end parallel
38      Pressure(i) = (SourceTerm(i) - Terms)/CoefficientMatrix(i, i)
39  enddo
40
41 end

```

ソースコード 18 負圧除去ルーチン

```

1 subroutine removeNegativePressure()
2     !負圧の除去
3     !粒子極密を用いて計算しているため,境界付近で負圧が発生する
4     use consts_variables
5     implicit none
6     integer ::i
7
8     !$omp parallel
9     !$omp do
10    do i = 1, NumberOfParticle

```

```

11     if (Pressure(i) < 0d0) Pressure(i) = 0d0
12     !if (Pressure(i) > 30000d0) Pressure(i) = 30000d0
13 enddo
14 !$omp end do
15 !$omp end parallel
16
17 end

```

ソースコード 19 近傍粒子での最小圧力設定ルーチン

```

1 subroutine setMinPressure()
2   !最小圧力設定ルーチン
3   !引力項による計算の発散を抑制する
4   use define
5   use consts_variables
6   implicit none
7   real*8 :: distance
8   real*8 :: calcDistance
9   integer :: i, j
10
11   do i = 1, NumberOfParticle
12     if (ParticleType(i) == PARTICLE_DUMMY) cycle
13     MinPressure(i) = Pressure(i)
14     !$omp parallel
15     !$omp do
16     do j = 1, NumberOfParticle
17       if (ParticleType(j) == PARTICLE_DUMMY) cycle
18       if (i == j) cycle
19       distance = calcDistance(i, j)
20       if (distance >= Radius_forGradient) cycle
21       if (MinPressure(i) > Pressure(j)) then
22         MinPressure(i) = Pressure(j)
23       endif
24     enddo
25     !$omp end do
26     !$omp end parallel
27   enddo
28
29 end

```

ソースコード 20 圧力勾配計算ルーチン

```

1 subroutine calcPressureGradient()
2   !圧力勾配の計算

```

```

3  use define
4  use consts_variables
5  implicit none
6  real*8 :: weight, distance, distance2
7  real*8 :: calcWeight
8  real*8 :: pIJ
9  real*8 :: deltaIJ(numDimension)
10 real*8 :: gradient(numDimension)
11 integer :: i, j, k
12
13 !$omp parallel private(gradient, distance, distance2, weight, deltaIJ, pIJ,
    j, k)
14 !$omp do
15 do i = 1, NumberOfParticle
16   if (ParticleType(i) .ne. PARTICLE_FLUID) cycle
17   gradient = 0d0
18   do j = 1, NumberOfParticle
19    if (i == j) cycle
20    if (ParticleType(j) == PARTICLE_DUMMY) cycle
21    distance2 = 0d0
22    do k = 1, numDimension
23     deltaIJ(k) = Pos(j, k) - Pos(i, k)
24     distance2 = distance2 + deltaIJ(k)**2
25    enddo
26    distance = sqrt(distance2)
27    if (distance < Radius_forGradient) then
28     weight = calcWeight(distance, Radius_forGradient)
29     pIJ = (Pressure(j) - MinPressure(i))/distance2
30     do k = 1, numDimension
31      gradient(k) = gradient(k) + deltaIJ(k)*pIJ*weight
32     enddo
33    endif
34   enddo
35   do k = 1, numDimension
36    gradient(k) = gradient(k)*numDimension/NO_forGradient
37    Acc(i, k) = -1d0*gradient(k)/FLUID_DENSITY
38   enddo
39 enddo
40 !$omp end do
41 !$omp end parallel
42
43 end

```

```
1 subroutine moveParticleImplicit()
2   !陰解法による粒子の移動
3   use consts_variables
4   implicit none
5   integer :: i, j
6
7   !$omp parallel private(j)
8   !$omp do
9   do i = 1, NumberOfParticle
10  do j = 1, numDimension
11    Vel(i, j) = Vel(i, j) + Acc(i, j)*dt
12    Pos(i, j) = Pos(i, j) + Acc(i, j)*dt**2
13  enddo
14  enddo
15  !$omp end do
16  !$omp end parallel
17  Acc = 0d0
18
19 end
```
