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 10 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運動量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 Computational Fluid Dynamics (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 Moving Particle Semi-implicit (MPS) 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. 水は非圧縮性流体とできるので, Navier-Stokes 方程式は

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

ここで $\boldsymbol{u}$ は速度である. 連続の式は

$$\frac{\partial \rho}{\partial t} + \rho\nabla\cdot\boldsymbol{u} = 0 \tag{2}$$

$$\nabla\cdot\boldsymbol{u} = 0 \tag{3}$$

したがって

$$\frac{\partial \boldsymbol{u}}{\partial t} = -\frac{1}{\rho}\nabla p + \mu\nabla^2\boldsymbol{u} + \boldsymbol{f} \tag{4}$$

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]

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

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.

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

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]

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

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

$$\begin{aligned} \langle \nabla^2 f \rangle_i &= \frac{2d}{\lambda n^0} \sum_{i \neq j} \left(f(\boldsymbol{x}_j) - f(\boldsymbol{x}_i) \right) w(|\boldsymbol{x}_j - \boldsymbol{x}_i|) \\ where\ \lambda &= \frac{\sum_{i \neq j} |\boldsymbol{x}_j - \boldsymbol{x}_i|^2 w(|\boldsymbol{x}_j - \boldsymbol{x}_i|)}{\sum_{i \neq j} w(|\boldsymbol{x}_j - \boldsymbol{x}_i|)} \end{aligned} \tag{9}$$

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て計算する. 陽解法とは, 現在のタイムステップの値を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. 一方では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.

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

$$[\nabla \cdot \boldsymbol{u}]^{k+1} = 0 \tag{11}$$

$[\phi]^k$ 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. $\boldsymbol{u}$ を移流項, 粘性項と外力項成分に分け, それぞれ $\boldsymbol{u}^*$, $\boldsymbol{u}'$ とする.

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

$$\frac{\Delta \boldsymbol{u}^*}{\Delta t} = \frac{\boldsymbol{u}^* - \boldsymbol{u}^{k+1}}{\Delta t} = \left[\mu \nabla^2 \boldsymbol{u} \right]^k + [\boldsymbol{f}]^k \tag{13}$$

ここで $\boldsymbol{u}^*$ 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 $\boldsymbol{u}^*$ のみなので, 計算することができる. 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

$$\frac{\langle \nabla \cdot \boldsymbol{u} \rangle^{k+1} - \langle \nabla \cdot \boldsymbol{u} \rangle^*}{\Delta t} = -\frac{1}{\rho} \langle \nabla^2 p \rangle^{k+1} \tag{14}$$

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

$$\langle \nabla^2 p \rangle^{k+1} = \rho \frac{\langle \nabla \cdot \boldsymbol{u} \rangle^*}{\Delta t} \tag{15}$$

となる. つぎに (2) 式を離散化すると

$$\frac{\rho^* - \rho}{\Delta t} + \rho \langle \nabla \cdot \boldsymbol{u} \rangle^* = 0 \tag{16}$$

$$\langle \nabla \cdot \boldsymbol{u} \rangle^* = -\frac{1}{\Delta t} \frac{\rho^* - \rho}{\rho} \tag{17}$$

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

$$\langle \nabla \cdot \boldsymbol{u} \rangle^* \simeq -\frac{1}{\Delta t} \frac{n^* - n^0}{n^0} \tag{18}$$

これと (15) 式から

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

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]

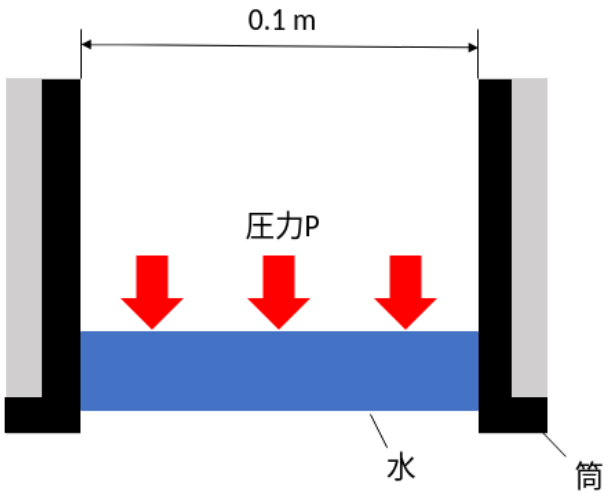

図 1　検証モデル模式図 (青:水粒子, 灰:ダミー粒子, 黒点:壁粒子)

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 から 4000 Pa で 500 Pa 刻みとする.
- 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 $\dfrac{\text{水の密度}\rho \times \text{初期状態の水の量 } V}{\text{水粒子数}}$ 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, その総和を水ロケットが生み出す運動量とする.

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

1 評価手法

CFD では計算点の数が膨大であり, 計算の正誤や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運動量 p が一定である.
- 時間間隔を変化させたとき発生する運動量 p が一定である.

2 評価

表 1 にシミュレーション条件を示す. また表 2 に粒子の大きさ, 時間間隔 (計算条件) を変化させたときの運動量を示す. 表 2 から時間間隔を変化したときの運動量 p の標準偏差は 0.09, 粒子直径を変化させたときの運動量 p の標準偏差は 0.06 となり, 計算条件によらず概ね一定であると言える. 以上からプログラムは正常に動作してると考えられる.

表 1　シミュレーション条件

シミュレート時間	0.12 s
流体密度	1000 $\mathrm{kg\,s^{-1}}$
動粘性係数	1.0×10^{-6} $\mathrm{m^2 s}$
圧力 P	4000 Pa
衝突係数	0.2

表 2　計算条件と運動量

時間間隔 dt / ms	粒子直径 / m	運動量 p / N s
0.20	0.0050	1.66
0.30	0.0050	1.83
0.40	0.0050	1.69
0.20	0.010	1.54
0.20	0.0025	1.63

6.　結果

表 3 に示した条件は固定とする.

表 3　シミュレーション条件 (固定)

時間間隔 dt	0.20 ms
粒子直径	0.0050 m
シミュレート時間	0.24 s
流体密度	1000 $\mathrm{kg\,s^{-1}}$
動粘性係数	1.0×10^{-6} $\mathrm{m^2 s}$
衝突係数	0.2

1　シミュレーション 1

シミュレーション 1 では圧力 P を変化させる. 表 4 にシミュレーション条件とその条件での運動量 p を示す.　また図 2 に各条件での時間-推力グラフを, 図 3 に圧力 P と運動量の関係を示す.　また各シミュレーションの動画を `https://drive.google.com/drive/folders/1AdJtppHVicWeStGBtDoW90rsgOPF6qaP?usp=sharing` に添付する.

表 4　シミュレーション条件と運動量 p

条件	圧力 P / Pa	水の量 V / m^3	運動量 p / Ns
1.1	1000	1.0×10^{-3}	1.36
1.2	1500	-	1.37
1.3	2000	-	1.50
1.4	2500	-	1.62
1.5	3000	-	1.59
1.6	3500	-	1.69
1.7	4000	-	1.82

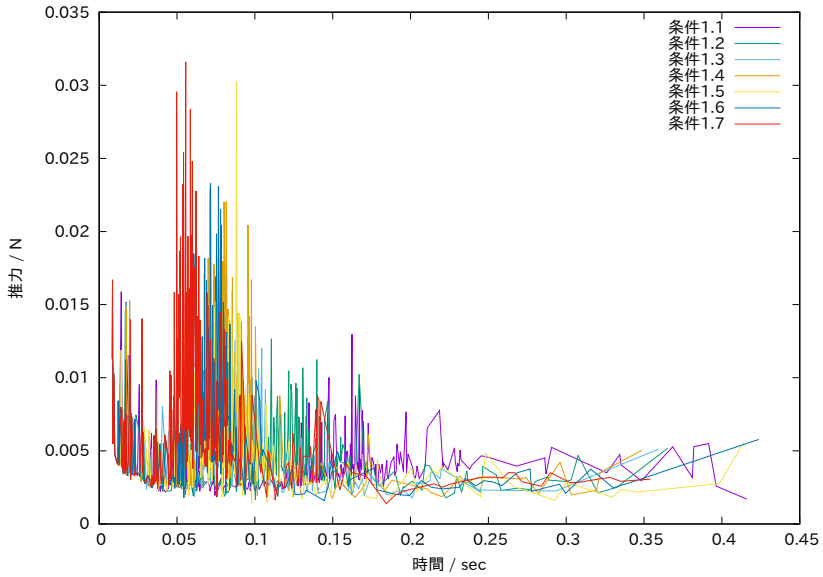

図 2　各条件での推力

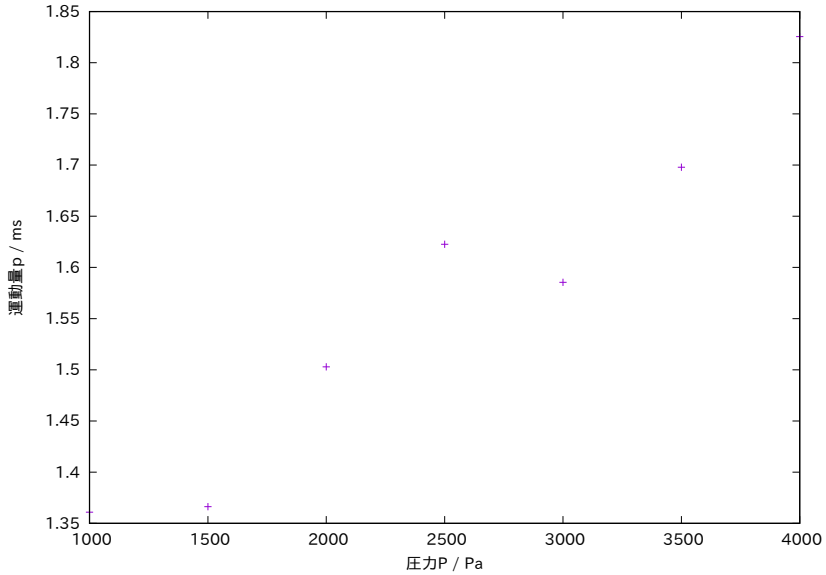

図 3　圧力 P と運動量の関係

2　シミュレーション 2

シミュレーション 1 では水の量 V を変化させる. 表 4 にシミュレーション条件とその条件での運動量 p を示す. また図 4 に各条件での時間-推力グラフを, 図 5 に水の量と運動量の関係を示す. また各シミュレーションの動画を `https://drive.google.com/drive/folders/1AdJtppHVicWeStGBtDoW90rsgOPF6qaP?usp=sharing` に添付する.

表 5　シミュレーション条件と運動量 p

条件	圧力 P / Pa	水の量 V / m^3	運動量 p / Ns
2.1	4000	1.0×10^{-3}	1.83
2.2	-	1.1×10^{-3}	1.85
2.3	-	1.2×10^{-3}	2.10
2.4	-	1.3×10^{-3}	2.28
2.5	-	1.4×10^{-3}	2.56
2.6	-	1.5×10^{-3}	2.59
2.7	-	1.6×10^{-3}	2.77
2.8	-	1.7×10^{-3}	2.90
2.9	-	1.8×10^{-3}	3.10
2.10	-	1.9×10^{-3}	3.32
2.11	-	2.0×10^{-3}	3.50

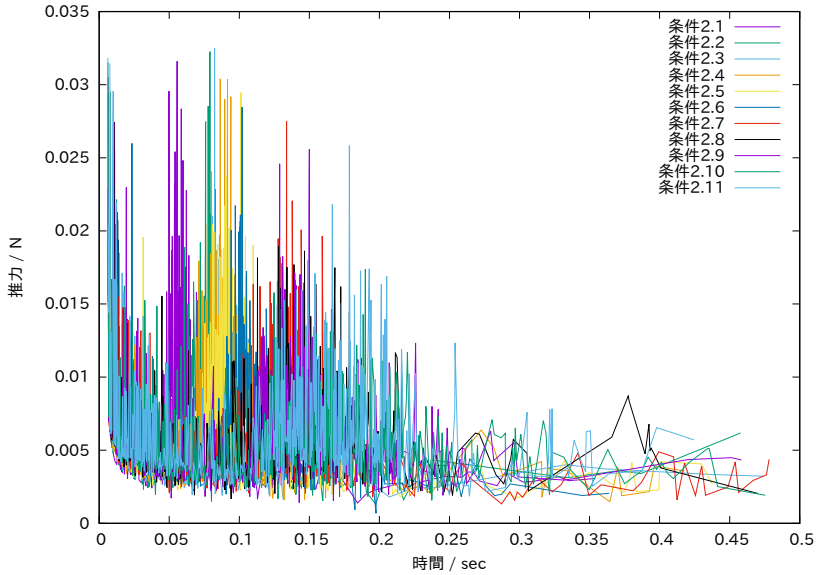

図 4　各条件での推力

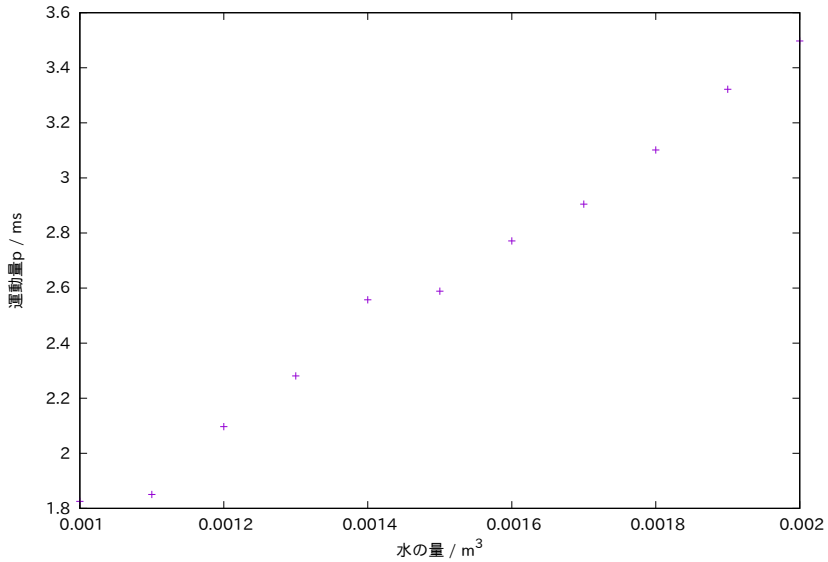

図 5　水の量と運動量の関係

7.　考察

1　シミュレーション 1

圧力 P と運動量 p の相関係数は 0.971 であり, 強い正の相関が見られる. このことから圧力が高ければ高いほどより多くの運動量を獲得できるとわかる. 一方で 2500 Pa と 3000 Pa の間で運動量の減少が見られ, 完全な比例関係でないことも予測される.

図 2 から条件 1.1, 条件 1.3, 条件 1.7 を取り出し, 移動平均を掛けたものを図 6 に示す. 図 6 から

圧力が高いほどピークの推力が高く, ピーク幅が狭いことがわかる．直感的には圧力が高いほど高速で水が噴射され, すぐに水を消費しきるので, 直感に合致する．

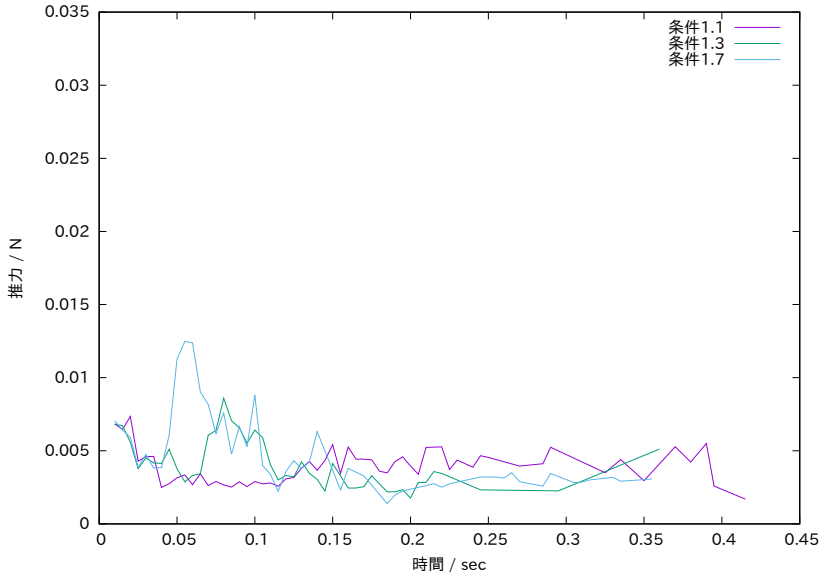

図 6　移動平均を掛けた推力

2　シミュレーション 2

水の量 V と運動量 p の相関係数は 0.995 であり, 強い正の相関が見られる．このことから水の量が多ければ多いほどより多くの運動量を獲得できるとわかる．また水の量 V と運動量 p の関係は最小二乗法を用いて $y = 1701x + 0.06$ の比例関係があると考えられる．

図 4 から条件 2.1, 条件 2.6, 条件 2.11 を取り出し, 移動平均を掛けたものを図 7 に示す．図 7 から水の量が多いほどピークが低くなっていることがわかる．直感的には水の量が多いと同じ圧力でも水の速度が小さくなるので, 直感に合致する．

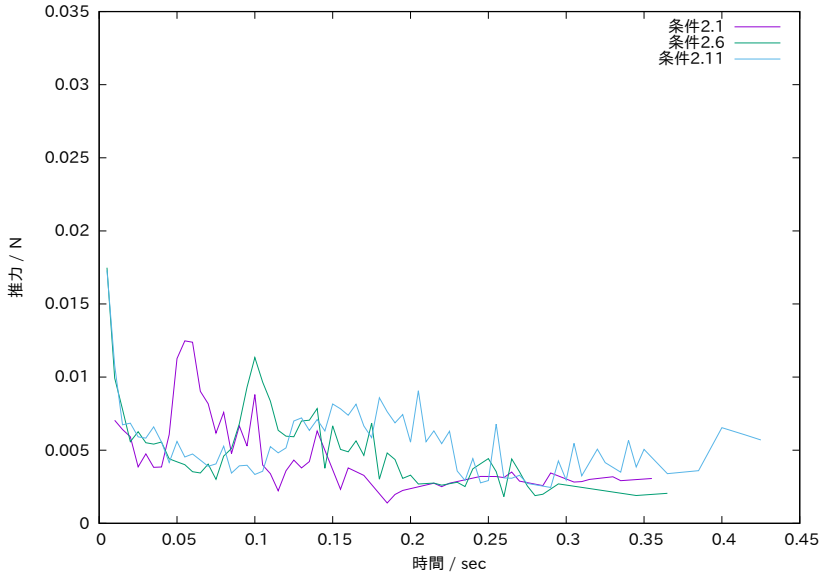

図 7　移動平均を掛けた推力

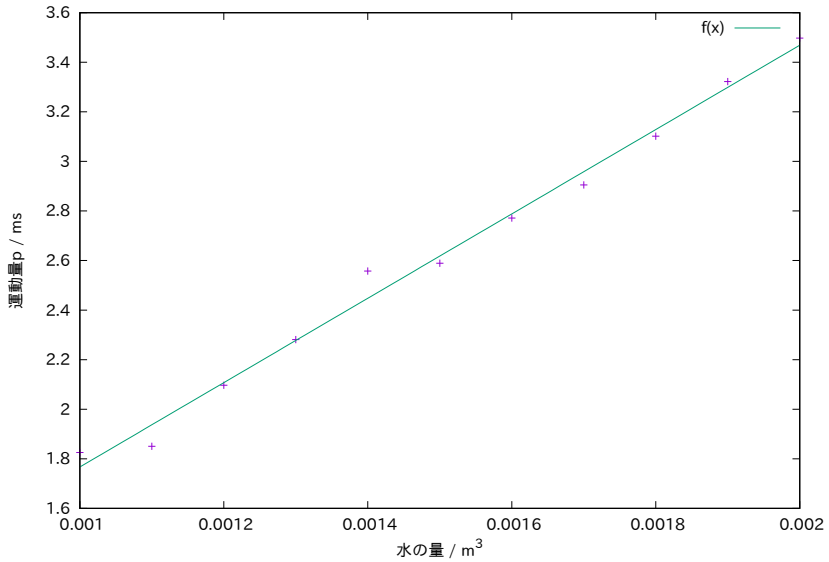

図 8　水の量と運動量の比例関係

8.　結論

水の量, 圧力と推力や運動量の関係を調べることができた. しかし実際の水ロケットでは水の噴射による圧力の低下や空気自体の噴射による推力の発生などが予想されるため, より精度の高いシミュレーションを行う必要がある.

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 pbcg lab. http://www.slis.tsukuba.ac.jp/~fujisawa.makoto.fu/cgi-bin/wiki/index.php?MPS%CB%A1%A4%CE%CD%FD%CF%C0. (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

ソースコードは以下のレポジトリで管理されている.

https://github.com/RyosukeSasaki/particle-fortran

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    character :: filename*128
7    character :: command*128
8    call omp_set_num_threads(8)
9
10   command = 'mkdir -p ' // dir // '; rm ' // dir // '/data*'
11   call system(command)
12
13   filename = dir // "/thrust.dat"
14
15   open (19, file=filename, status='replace')
16   write (*, *) "write data to ", dir, "/. Pressure is ", WaterPressure
17
18   call InitParticles()
19   call calcConsts()
20   call writeInit()
21   call output(0)
22   do i = 0, 240-1
23   do j = 1, interval
24     call calcGravity()
25     call calcViscosity()
26     call moveParticleExplicit()
27     call collision()
28     call calcNumberDensity()
29     call setBoundaryCondition()
30     call setSourceTerm()
31     call setMatrix()
32     call GaussEliminateMethod()
33     call removeNegativePressure()
34     call setMinPressure()
35     call calcPressureGradient()
```

```
36      call moveParticleImplicit()
37      call detach(i*interval+j)
38      call output(i*interval+j)
39    enddo
40    write (*, *) "Timestep: ", (i+1)*interval
41    enddo
42
43    write (19, *) "#deltaV: ", deltaV
44    close (19)
45
46  end
```

ソースコード 2　粒子データ出力用ルーチン

```
1  subroutine output(i)
2    use consts_variables
3    implicit none
4    integer :: i, k
5    character :: filename*128
6
7    write (filename, '("/data", i4.4, ".dat")') i
8    filename = dir // filename
9    open (11, file=filename, status='new')
10   do k = 1, NumberOfParticle
11     write (11, '(f6.3,X)', advance='no') Pos(k, 1)
12     write (11, '(f6.3,X)', advance='no') Pos(k, 2)
13     write (11, '(I2,X)', advance='no') ParticleType(k)
14     write (11, '(f15.5,X)', advance='no') Pressure(k)
15     if (CollisionState(k) .eqv. .true.) then
16       write (11, '(I2,X)', advance='no') 1
17     else
18       write (11, '(I2,X)', advance='no') 0
19     endif
20     write (11, *)
21   enddo
22   close (11)
23
24 end
```

ソースコード 3　初期状態出力用ルーチン

```
1  subroutine writeInit()
2    use consts_variables
3    implicit none
```

```
4   character :: filename*128
5
6   filename = dir // "/init.txt"
7   open (18, file=filename, status='replace')
8   write (18, *) "dt: ", dt
9   write (18, *) "Particle radius: ", PARTICLE_DISTANCE
10  write (18, *) "Fluid density: ", FLUID_DENSITY
11  write (18, *) "Water Size: ", WsizeX, " * ", WsizeY, " * ", WsizeZ
12  write (18, *) "Pressure: ", WaterPressure
13  write (18, *) "Number of Paricle: ", NumberOfParticle
14  write (18, *) "Number of Fluid: ", NumberOfFluid
15  write (18, *) "Mass of Fluid Paricle: ", MassOfParticle
16  close (18)
17
18 end
```

ソースコード 4　推力・運動量計算用ルーチン

```
1  subroutine detach(i)
2    use consts_variables
3    use define
4    implicit none
5    integer :: i,j
6    real*8 :: VelocitySum
7    logical :: n0
8
9    n0 = .true.
10   VelocitySum = 0d0
11   do j = 1, NumberOfParticle
12     if (ParticleType(j) .ne. PARTICLE_FLUID) cycle
13     if (Pos(j, 2) < -0.01d0) then
14       if (detachState(j) .eqv. .false.) then
15         VelocitySum = VelocitySum - Vel(j, 2)
16         detachState(j) = .true.
17         n0 = .false.
18       endif
19     endif
20   enddo
21   deltaV = deltaV + VelocitySum*MassOfParticle
22   if (n0 .eqv. .false.) then
23     write (19, '(f6.4,X)', advance='no') i*dt
24     write (19, '(f15.10,X)', advance='no') VelocitySum*MassOfParticle
25     write (19, *)
```

```
26   endif
27
28 end
```

ソースコード 5　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_INNER = 0
5    integer, parameter :: BOUNDARY_SURFACE_LOW = 1, BOUNDARY_SURFACE_HIGH = 2
6
7  end
```

ソースコード 6　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 = 0.005
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 = 1.0d0
14   integer, parameter :: MaxNumberOfParticle = 500000
15   integer, parameter :: numDimension = 2
16
17   !筒の大きさ
18   real*8, parameter :: sizeX = 0.1d0, sizeY = 0.4d0
19
20   real*8 :: Radius_forNumberDensity, Radius_forGradient, Radius_forLaplacian
21   real*8 :: N0_forNumberDensity, N0_forGradient, N0_forLaplacian
22   real*8 :: Lambda
23   real*8 :: collisionDistance = 0.8*PARTICLE_DISTANCE
24   real*8 :: MassOfParticle
25   real*8 :: deltaV = 0
26
27   integer :: NumberOfParticle
```

```
  integer :: NumberOfFluid

  real*8 :: dt = 0.0002d0
  character :: dir*6 = "data08"
  real*8, parameter :: WaterPressure = 3000
  !水領域の大きさ
  real*8, parameter :: WsizeX = 0.1d0, WsizeY = 0.1d0, WsizeZ = 0.1d0

  real*8 :: Pos(MaxNumberOfParticle, numDimension)
  real*8 :: Gravity(numDimension)
  ! 0:Fluid, 1:Wall, 2:Dummy
  integer :: ParticleType(MaxNumberOfParticle)
  data Gravity/0d0, -9.8d0/
  real*8, allocatable :: Vel(:, :)
  real*8, allocatable :: Acc(:, :)
  real*8, allocatable :: Pressure(:)
  real*8, allocatable :: MinPressure(:)
  real*8, allocatable :: NumberDensity(:)
  integer, allocatable :: BoundaryCondition(:)
  real*8, allocatable :: SourceTerm(:)
  real*8, allocatable :: CoefficientMatrix(:, :)
  logical, allocatable :: CollisionState(:)
  logical, allocatable :: detachState(:)

end
```

ソースコード 7　重み係数計算関数

```
real*8 function calcWeight(distance, radius)
  implicit none
  real*8, intent(in) :: distance, radius
  real*8 :: w

  if (distance >= radius) then
    w = 0
  else
    w = radius/distance - 1d0
  endif
  calcWeight = w
  return

end function
```

ソースコード 8　粒子間距離計算関数

```
real*8 function calcDistance(i, j)
  use consts_variables
  implicit none
  real*8 :: distance2
  integer :: i, j, k

  distance2 = 0d0
  do k = 1, numDimension
    distance2 = distance2 + (Pos(j, k) - Pos(i, k))**2
  enddo
  calcDistance = sqrt(distance2)
  return

end function
```

ソースコード 9　粒子初期化ルーチン

```
subroutine InitParticles()
  !粒子の初期値設定
  use define
  use consts_variables
  implicit none
  real*8 :: x, y
  real*8 :: EPS
  integer :: iX, iY
  integer :: nX, nY
  integer :: i = 1
  logical :: flagOfParticleGenerarion
  logical :: shift = .false.

  EPS = PARTICLE_DISTANCE*0.01
  nX = int(sizeX/PARTICLE_DISTANCE) + 7
  nY = int(sizeY/PARTICLE_DISTANCE) + 7
  do iX = -6, nX
    do iY = -6, nY
      x = PARTICLE_DISTANCE*dble(iX)
      if (shift .eqv. .true.) then
        y = PARTICLE_DISTANCE*dble(iY)
      else
        y = PARTICLE_DISTANCE*(dble(iY)+0.5)
      endif
      flagOfParticleGenerarion = .false.
```

```
26
27        !dummy particle
28        if (((x > -6*PARTICLE_DISTANCE + EPS) .and. (x <= sizeX + 6*
              PARTICLE_DISTANCE + EPS)) &
29            .and. ((y > 0d0 - 0*PARTICLE_DISTANCE + EPS) .and. (y <= sizeY +
                  EPS))) then
30          ParticleType(i) = PARTICLE_DUMMY
31          flagOfParticleGenerarion = .true.
32        endif
33
34        !wall particle
35        if (((x > -3*PARTICLE_DISTANCE + EPS) .and. (x <= sizeX + 3*
              PARTICLE_DISTANCE + EPS)) &
36            .and. ((y > 0d0 - 0*PARTICLE_DISTANCE + EPS) .and. (y <= sizeY +
                  EPS))) then
37          ParticleType(i) = PARTICLE_WALL
38          flagOfParticleGenerarion = .true.
39        endif
40
41        !wall particle
42        if (((x > -6*PARTICLE_DISTANCE + EPS) .and. (x <= sizeX + 6*
              PARTICLE_DISTANCE + EPS)) &
43            .and. ((y > 0 - 3*PARTICLE_DISTANCE + EPS) .and. (y <= 0 + EPS)))
                  then
44          ParticleType(i) = PARTICLE_WALL
45          flagOfParticleGenerarion = .true.
46        endif
47
48        !empty region
49        if (((x > 0d0 + EPS) .and. (x <= sizeX + EPS)) .and. (y > -20d0 + EPS
              )) then
50          flagOfParticleGenerarion = .false.
51        endif
52
53        !generate position and velocity
54        if (flagOfParticleGenerarion .eqv. .true.) then
55          Pos(i, 1) = x; Pos(i, 2) = y
56          i = i + 1
57        endif
58
59      enddo
60      if (shift .eqv. .true.) then
```

```
61        shift = .false.
62      else
63        shift = .true.
64      endif
65    enddo
66
67    NumberOfFluid = 0
68    do iX = -6, nX
69      do iY = -6, nY
70        x = PARTICLE_DISTANCE*dble(iX)
71        y = PARTICLE_DISTANCE*dble(iY)
72        flagOfParticleGenerarion = .false.
73
74        !fluid particle
75        if (((x > 0d0 + EPS) .and. (x <= WsizeX + EPS)) .and. ((y > 0d0 + EPS
              ) .and. (y <= WsizeY + EPS))) then
76          ParticleType(i) = PARTICLE_FLUID
77          flagOfParticleGenerarion = .true.
78          NumberOfFluid = NumberOfFluid + 1
79        endif
80
81        !generate position and velocity
82        if (flagOfParticleGenerarion .eqv. .true.) then
83          Pos(i, 1) = x; Pos(i, 2) = y
84          i = i + 1
85        endif
86
87      enddo
88    enddo
89
90    NumberOfParticle = i - 1
91    !allocate memory for particle quantities
92    allocate (Vel(NumberOfParticle, numDimension))
93    allocate (Acc(NumberOfParticle, numDimension))
94    allocate (NumberDensity(NumberOfParticle))
95    allocate (BoundaryCondition(NumberOfParticle))
96    allocate (SourceTerm(NumberOfParticle))
97    allocate (CoefficientMatrix(NumberOfParticle, NumberOfParticle))
98    allocate (Pressure(NumberOfParticle))
99    allocate (MinPressure(NumberOfParticle))
100   allocate (CollisionState(NumberOfParticle))
101   allocate (detachState(NumberOfParticle))
```

```
102   detachState = .false.
103   Vel = 0d0
104   Acc = 0d0
105   Pressure = 0d0
106   MassOfParticle = FLUID_DENSITY*WsizeX*WsizeY*WsizeZ/NumberOfFluid
107
108   return
109
110 end
```

ソースコード 10　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   N0_forNumberDensity = 0d0
24   N0_forGradient = 0d0
25   N0_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        N0_forNumberDensity = N0_forNumberDensity + calcWeight(distance,
              Radius_forNumberDensity)
37        N0_forGradient = N0_forGradient + calcWeight(distance, Radius_forGradient
              )
38        N0_forLaplacian = N0_forLaplacian + calcWeight(distance,
              Radius_forLaplacian)
39
40        Lambda = Lambda + distance2*calcWeight(distance, Radius_forLaplacian)
41      enddo
42    enddo
43    Lambda = Lambda/N0_forLaplacian
44
45  end
```

ソースコード 11　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 12　粘性項計算ルーチン

```
subroutine calcViscosity()
  !粘性項の計算
  use define
  use consts_variables
  implicit none
  real*8 :: ViscosityTerm(numDimension)
  real*8 :: distance, weight
  real*8 :: calcWeight, calcDistance
  real*8 :: m
  integer :: i, j, k

  m = 2d0*numDimension/(N0_forLaplacian*Lambda)*KINEMATIC_VISCOSITY

  !$omp parallel private(ViscosityTerm, distance, weight, j, k)
  !$omp do
  do i = 1, NumberOfParticle
  if (ParticleType(i) == PARTICLE_FLUID) then
    ViscosityTerm = 0d0
    do j = 1, NumberOfParticle
      if (i == j) cycle
      distance = calcDistance(i, j)
      if (distance < Radius_forLaplacian) then
        weight = calcWeight(distance, Radius_forLaplacian)
        do k = 1, numDimension
          ViscosityTerm(k) = ViscosityTerm(k) + (Vel(j, k) - Vel(i, k))*
              weight
        enddo
      endif
    enddo
    do k = 1, numDimension
      Acc(i, k) = Acc(i, k) + ViscosityTerm(k)*m
    enddo
  endif
  enddo
  !$omp end do
  !$omp end parallel

end
```

ソースコード 13　仮速度・仮位置計算ルーチン

```
subroutine moveParticleExplicit()
  !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 14　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 15　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
```

```
  real*8 :: calcDistance, calcWeight
  integer :: i, j

  NumberDensity = 0d0
  do i = 1, NumberOfParticle
    if (ParticleType(i) == PARTICLE_DUMMY) cycle
    do j = 1, NumberOfParticle
      if (ParticleType(j) == PARTICLE_DUMMY) cycle
      if (i == j) cycle
      distance = calcDistance(i, j)
      weight = calcWeight(distance, Radius_forNumberDensity)
      NumberDensity(i) = NumberDensity(i) + weight
    enddo
  enddo

end
```

ソースコード 16　ディリクレ境界条件判定ルーチン

```
subroutine setBoundaryCondition()
  !境界条件の設定
  use define
  use consts_variables
  implicit none
  integer :: i

  !$omp parallel
  !$omp do
  do i = 1, NumberOfParticle
  if (ParticleType(i) == PARTICLE_DUMMY) then
    BoundaryCondition(i) = BOUNDARY_DUMMY
  elseif (NumberDensity(i) < (THRESHOLD_RATIO_BETA*N0_forNumberDensity)) then
    BoundaryCondition(i) = BOUNDARY_SURFACE
  else
    BoundaryCondition(i) = BOUNDARY_INNER
  endif
  enddo
  !$omp end do
  !$omp end parallel

end
```

ソースコード 17　ポアソン方程式右辺設定ルーチン

```
subroutine setSourceTerm()
  !ポアソン方程式右辺の設定
  use define
  use consts_variables
  implicit none
  integer :: i

  SourceTerm = 0d0
  !$omp parallel
  !$omp do
  do i = 1, NumberOfParticle
    if (ParticleType(i) == PARTICLE_DUMMY) cycle
    if (BoundaryCondition(i) == BOUNDARY_SURFACE) cycle
    if (BoundaryCondition(i) == BOUNDARY_INNER) then
      SourceTerm(i) = RELAXATION_COEF_FOR_PRESSURE*(1d0/dt**2)* &
                      (NumberDensity(i) - N0_forNumberDensity)/
                          N0_forNumberDensity
    endif
  enddo
  !$omp end do
  !$omp end parallel

end
```

ソースコード 18　係数行列の計算ルーチン

```
subroutine setMatrix()
  !係数行列の設定
  use define
  use consts_variables
  implicit none
  real*8 :: distance
  real*8 :: calcDistance, calcWeight
  real*8 :: coefIJ, a
  integer :: i, j

  CoefficientMatrix = 0d0
  a = 2d0*numDimension/(N0_forLaplacian*Lambda)
  do i = 1, NumberOfParticle
    if (BoundaryCondition(i) .ne. BOUNDARY_INNER) cycle
    do j = 1, NumberOfParticle
      if (BoundaryCondition(j) == BOUNDARY_DUMMY) cycle
      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
            **2)
25    enddo
26
27  end
```

ソースコード 19　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 20　負圧除去ルーチン

```
1  subroutine removeNegativePressure()
2    !負圧の除去
3    !粒子枢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 21　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
```

```
  integer :: i, j

  do i = 1, NumberOfParticle
    if (ParticleType(i) == PARTICLE_DUMMY) cycle
    MinPressure(i) = Pressure(i)
    !$omp parallel
    !$omp do
    do j = 1, NumberOfParticle
      if (ParticleType(j) == PARTICLE_DUMMY) cycle
      if (i == j) cycle
      distance = calcDistance(i, j)
      if (distance >= Radius_forGradient) cycle
      if (MinPressure(i) > Pressure(j)) then
        MinPressure(i) = Pressure(j)
      endif
    enddo
    !$omp end do
    !$omp end parallel
  enddo

end
```

ソースコード 22　圧力勾配計算ルーチン

```
subroutine calcPressureGradient()
  !圧力勾配の計算
  use define
  use consts_variables
  implicit none
  real*8 :: weight, distance, distance2
  real*8 :: calcWeight
  real*8 :: pIJ
  real*8 :: deltaIJ(numDimension)
  real*8 :: gradient(numDimension)
  integer :: i, j, k

  !$omp parallel private(gradient, distance, distance2, weight, deltaIJ, pIJ,
      j, k)
  !$omp do
  do i = 1, NumberOfParticle
    if (ParticleType(i) .ne. PARTICLE_FLUID) cycle
    gradient = 0d0
    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/N0_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
```

ソースコード 23　速度・位置計算ルーチン

```
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
```