音響解析における CIP の紹介

はじめに

数値音響解析の分野で、波動的解釈を基盤にした手法に注目が集まっている。幾何的解釈と比べて回折などの現象を物理的に正しく求められるためかもしれない。その分だけポストプロセスでクリエイターが所望する音を生成しやすくなるため、シミュレータに物理的妥当性を求めるのは、それだけでないにしても前提だと思える。波動音響解析の場合、現象の見方に時間領域と周波数領域の 2 種類があるが、時間領域的描像の代表例に FDTD と CIP がある。

FDTD は流体の質量保存則 (連続の式) と運動量保存則を軸に考える。保存則という言葉は連続体力学でよく聞くが、その分野には流束という重要な概念が存在する。大雑把に言えば流束とは、系の境界を通して流入出する物理量のことであり、この流束があることで系の状態は変化する。FDTD は計算格子の数だけ部分系を用意し、系間の流束を数値計算する。そのため、音と言えば波で、波と言えば移流 (伝播) するものだが、FDTD では移流という解釈があまり登場しない。あくまでも興味があるのは隣接する系同士の作用であり、正に流束である。

一方の CIP は音現象の移流に着目している。波動方程式を学んだときに、”ついでに”ダランベールの解も紹介されたと思うが、CIP はダランベールの解の考え方に近い。例えば 1 次元空間において正の向きに進む波を考えよう。時刻ゼロの波形が $\phi(x)$ で表されるとき、時刻 t における波形は $\phi(x-ct)$ で表される (ここで c は移流速度、つまり音速)。このような理論を軸に CIP は状態を時間発展させる。

本資料は音響解析における CIP の紹介を目的としている。既に CIP に関する書籍は存在するが、FVM や FEM と比べると少ないのではないだろうか。また、音響解析に特化した CIP の資料となるとかなり限られる。音響解析において重要と思われない CIP のテクニックに関しては大胆に省き、息切れすることなく CIP を理解できるよう心掛けた。

1 支配方程式

1.1 質量保存則と運動量保存則

音の正体が流体の微小な運動であるならば、音に関する支配方程式は流体の支配方程式から導出できるであろう。この考えは実際に正しく、流体の質量保存則と運動量保存則から音の現象を説明できる。

まず、流体の質量保存則は

$$\partial_t R + \boldsymbol{\nabla}\cdot(R\boldsymbol{v}) = 0$$

で表される。ここで $R = R(\boldsymbol{r},t)$ は系の密度場、$\boldsymbol{v} = \boldsymbol{v}(\boldsymbol{r},t)$ は系の速度場である。音現象は流体の非常に小さな運動であるため、$\boldsymbol{v}$ は微小な値と考えることができる。また、系全体で流速がゼロのときの密度を ρ_0 としたとき、密度の変動 $\rho = R - \rho_0$ も非常に小さいと考えてよい。そこで、2 次以上の微小項を無視したとき、上式は

$$\partial_t \rho + \rho_0 \boldsymbol{\nabla}\cdot\boldsymbol{v} = 0 \tag{1}$$

になる。ただし、ρ_0 は一様かつ定常であると仮定した。式 (1) は一般的な音響問題で利用可能な質量保存則と言える。

次に運動量保存則を考える。流体問題では内力として圧力と粘性応力を考えるが、本資料では粘性応力を無視する。この場合、系内に熱源などが無ければ状態の変化は等エントロピー過程になる。粘性応力を無視したときの運動量保存則は

$$\partial_t(R\boldsymbol{v}) + \boldsymbol{v}\boldsymbol{\nabla}\cdot(R\boldsymbol{v}) = -\boldsymbol{\nabla}P$$

となる。ここで $P = P(\boldsymbol{r},t)$ は圧力場である。上式の運動量保存則に対しても 2 次以上の微小項を無視していく。加えて圧力に関しても流体が静止しているときの圧力 p_0 と変動値 p を用いて $P = p_0 + p$ のように分解する。すると上式は

$$\rho_0 \partial_t \boldsymbol{v} = -\boldsymbol{\nabla} p \tag{2}$$

となる。これが音響問題において利用可能な運動量保存則の式である (特別に線形オイラー方程式と呼ばれている)。

2 つの式 (1)(2) に対して $\rho, \boldsymbol{v}, p$ の 3 つの未知量がある。したがって、状態の時間発展を知るにはもう一つ支配方程式が必要になる。そこで、状態方程式を利用しよう。いま、系は等エントロピー過程に従うため、$PR^{-\gamma}$ は常に一定である。従って、圧力 P を密度の関数 $P(R)$ で表すことができる。これを ρ_0 でテイラー展開し、1 次の項まで考慮したとき、

$$P = p_0 + p = p_0 + \rho \partial_R P|_{\rho_0}$$

なる関係を得る。したがって p に関して

$$p = \rho \partial_R P|_{\rho_0} = c^2 \rho \tag{3}$$

が言える。係数 c^2 は一般的に定数であると仮定してよい (ご想像の通り c は音速に相当する)。式 (3) より、式 (1) は

$$\partial_t p + k\boldsymbol{\nabla}\cdot\boldsymbol{v} = 0 \tag{4}$$

と書き直すことができる。以上より、式 (2)(4) を用いれば系の圧力と速度の時間発展を計算することができる。以下は、支配方程式の再掲である。

音響問題における支配方程式

圧力の微小変動を p、速度場を $\boldsymbol{v}$ とする。以下は一般的な音響問題に用いられている支配方程式である。

$$\partial_t p + k\boldsymbol{\nabla}\cdot\boldsymbol{v} = 0, \quad \rho_0\partial_t\boldsymbol{v} = -\boldsymbol{\nabla}p$$

1.2 移流方程式

前節の質量保存則や運動量保存則から移流方程式を導出する。1 次元空間の場合の質量保存則および運動量保存則は

$$\partial_t p + k\partial_x u = 0, \quad \rho_0 \partial_t u = -\partial_x p$$

となる (ここで u は流速だが、前節と違い 1 次元であるためスカラーとした)。次に天下り的だが、$f = p + Zu$ 及び $g = p - Zu$ なる物理量を考える。ここで $Z = \sqrt{\rho_0 k}$、つまり系の特性インピーダンスである。上の 2 式に対して両辺の和及び差を計算することで、f と g に関する偏微分方程式

$$\partial_t f + c\partial_x f = 0, \quad \partial_t g - c\partial_x g = 0 \tag{5}$$

を得る。このような形をした偏微分方程式を移流方程式と言う。

なぜこの方程式が「移流」と呼ばれているのか、大雑把に議論していこう。まず、$f(x,t) = F(x-ct)$ で表されるような関数を考える。$F_0 = F(0)$ としたとき、F_0 は各時刻において $x = ct$ の位置で確認できる。同様に $F_y = F(y)$ としたとき、F_y は各時刻 t において $x = ct + y$ の位置で確認できる。これは任意の y で成立するので、$F(x-ct) = f(x,t)$ は速度 c で x の正の方向に移流していることが分かる。$F(x-ct)$ を偏微分すると、$\partial_t F + c\partial_x F = 0$ を満たすことが容易に確かめられる。以上より、式 (7) を満たす f は移流性を有するため、式 (7) が移流方程式と言われる訳である。なお、$\partial_t g - c\partial_x g = \partial_t g + (-c)\partial_x g = 0$ であるため、g は負の方向に移流する関数 (波) である。以下は移流方程式の再掲である。

移流方程式

1 次元空間におけるスカラー場 f を考える。以下の方程式を移流方程式と言う (c は移流速度)。

$$\partial_t f + c\partial_x f = 0$$

2 CIP の前準備 (空間補間と移流)

前章の通り音の描像には保存則によるものと移流によるものがあり、その中でも CIP は移流を採用する。$f(x, t+\delta t) = f(x - c\delta t, t)$ というヒントを存分に用いる訳である。そこで、本章は移流方程式の数値計算に関して理解を深めるために、CIP の前に FDM や FVM による解法を議論する。先に述べておくと、FDM は離散点における物理量を利用し、FVM は計算格子内の物理量の積分値を利用する。物理量自体や積分値といったものをモーメントと呼ぶならば、FDM や FVM はシングルモーメント法と言える。一方の CIP は物理量だけでなく微分値も利用し、更に CIP-CSL という発展形は積分値も利用する。従って CIP はマルチモーメント法と言えるだろう。結局のところ、CIP の本質は「どのモーメントをどのように利用するか」にある。したがって、FDM や FVM におけるモーメントの使われ方は非常に参考になる。

また、本章の議論は 1 次元移流方程式に限定する。音響現象の場合、多次元になるとホイヘンスの原理の寄与の重要度が増し、考えるべきことが増える。対策としてフラクショナルステップ法などが施されるが、これは CIP の肝とは言えない。当分の間は 1 次元に限定し、CIP の本質の議論に集中することにしたい。

2.1 空間の補間

正の向きに進む物理量が $f^*(x,t)$ で表されるとする。当然ながら f^* は未知で、シミュレーションで解くべき対象である。そこで、シミュレーション結果を $f(x,t)$ と表記する。一般的に $f(x,t)$ のモデル式として多項式が採用される。ただし、系全体を一つの多項式で表現するのではなく、部分系毎に多項式を用意して f^* と一致するように努める。例えば図 1 は系を $\{[x_{i-1}, x_i] | i = 1, ..., N\}$ の部分系に分割している。また図 1(a) の場合は $f(x,t)$ の表現に 1 次多項式を、図 1(b) の場合は 0 次多項式を用いている。本節では、$[x_{i-1}, x_i]$ における多項式を

$$f^i(x,t) = \sum_m c_m (x - x_i)^m \quad x \in [x_{i-1}, x_i] \tag{6}$$

のように書き表すことにしよう。これは格子内の空間補間と言える。高度なシミュレーション手法では複数の格子をまたぐ多項式を考えたりもするが、本資料では 1 つの格子に納まる多項式を考えることにする (一般的な CIP もそうなっている)。

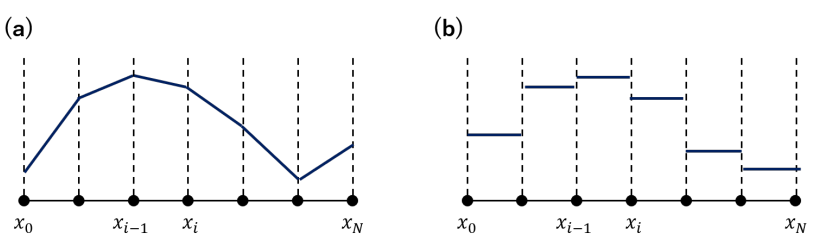

図 1　空間補間のイメージ図。

以上の議論は負の向きに進む物理量 $g(x,t)$ に対しても利用できる。ただし、空間補間の多項式の定義域を $[x_i, x_{i+1}]$ として、

$$g^i(x,t) = \sum_m c_m (x - x_i)^m \quad x \in [x_i, x_{i+1}] \tag{7}$$

のように定義する。なお、f と g のどちらにも当てはまる議論の際は、h の表記をもって両者を表すことにしよう。

2.2 FDM による解法

2.2.1 空間補間

式 (6)(7) のような多項式、つまりパラメトリックな関数で補間する場合、系の状態は係数 (パラメータ)c_m の値で決まる。一方で FDM は離散点 x_i の物理量 h_i をデータとして保持し、系を有限次元の状態ベクトルで表現する。FDM は、係数による表現と有限次元状態ベクトルによる表現が同値であると考える。これは、多項式に対して $f^i(x_i,t) = f_i$ および $f^i(x_{i-1},t) = f_{i-1}$ の制約を課すことに相当する。x の値が所与で 2 つの制約式があるということは、式 (6) から c_m に関する 2 つの連立一次方程式が作れることを意味する。したがって、今回の場合の $h^i(x,t)$ は図 1(a) のような 1 次多項式 $h^i(x,t) = c_1(x - x_i) + c_0$ でなければならない。以上より係数に関する連立一次方程式

$$\begin{aligned} c_0 &= h_i \\ c_1 D + c_0 &= h_{iup} \end{aligned} \tag{8}$$

が得られる。ここで下付き文字 iup は格子における風上側の離散点を意味しており、$f(g)$ の場合は $i-1(i+1)$ である。また、$D = x_{iup} - x_i$ である。この連立一次方程式を解いたときの係数を表 1 に纏めた。

2.2.2 移流処理

係数さえ求まれば後の処理は容易である。FDM の場合、移流に従って各離散点の値を時間発展させる。例えば $f^i(x_i,t) = f_i$ の場合、$f^i(x_i, t+\Delta t) = f^i(x_i - c\Delta t, t)$ より、

$$f_i \leftarrow f^i(x_i - c\Delta t, t) = c_1(-c\Delta t) + c_0$$

となる (ここでクーラン数を 1 以下であることを仮定した)。同様に $g^i(x_i,t) = g_i$ の更新も

$$g_i \leftarrow g^i(x_i + c\Delta t, t) = c_1(c\Delta t) + c_0$$

とすればよい。明らかに上 2 式は

$$h_i \leftarrow h^i(x_i - sc\Delta t, t) = c_1(-sc\Delta t) + c_0 \tag{9}$$

のように纏めることができる。ここで s は波が進行する向きを表しており、正の向きの場合は 1、負の向きの場合は-1 となる。以上の処理を所望の時間まで繰り返すことが、FDM 解法の基本である。

2.2.3 境界条件

最後に境界条件 (B.C.) の処理を紹介しなければならない。図 1 より、f の場合は式 (9) による時間発展を f_0 に施すことができない。$f^0(x,t)$ が定義されていないためである (無理矢理定義するには仮想的な点 x_{-1} が必要になる)。同様に式 (9) は g_N の時間発展を計算できない。これも $g^N(x,t)$ が定義されていないためである (無理矢理定義するには仮想的な点 x_{N+1} が必要になる)。そのため、f_0 と g_N の更新は B.C. に従って行わなければならない。

幸いにも、点 x_0 における g の値 g_0 は求められる。x_0 での反射率を r_0 としたとき、$f_0 = r_0 g_0$ でなければならない。同様に x_N での反射率を r_N としたとき、$g_N = r_N f_N$ となる。このようにして境界での f 及び g を求めればよい。

2.2.4 アルゴリズムと python コード例

以上が FDM の考え方である。本資料を通して言えることだが、移流方程式の求解は空間補間、移流、並びに B.C. の処理の順で進む。以下は FDM におけるアルゴリズムである。

表 1　FDM における係数 (表中 h は列毎に f もしくは g に置き換える)

	f	g
iup	$i-1$	$i+1$
D	$-\Delta x$	Δx
c_0	h_i	
c_1	$(h_{iup} - c_0)/D$	

表 2　シミュレーション条件

計算格子数	計算格子幅 [m]	密度 [kg/m^3]	体積膨張率 [kg/(ms^2)]	境界反射率	クーラン数
101	0.01	1.2	1.4×10^5	0.5	0.5

FDM による移流方程式の求解

1. 初期条件として与えられた圧力 p 及び流速 u より、$f = p + Zu$ と $g = p - Zu$ を計算する。
2. 表 1 に従い各計算格子間の多項式を決定する。
3. 式 (9) を用いて状態を更新する。ただし、多項式が定義されていない故に更新できない点 (f_0 や g_N など) は、B.C. に従い更新する。
4. 2.-3. を繰り返す。

FDM による解法の python コード例を紹介する。シミュレーション条件は表 2 の通りである。流速の初期条件を一様にゼロ、圧力の初期条件を以下の通りにした。

$$p(x) = \begin{cases} 1 & (0.45 \leq x \leq 0.55) \\ 0 & (\text{otherwise}) \end{cases}$$

```
import numpy as np
import math

N = 101
dx = 0.01
rho = 1.2
k = 1.4e+5
c = math.sqrt(k/rho) #sound speed
Z = math.sqrt(rho*k) #impedance
r_0 = 0.5
r_N = 0.5
dt = cfl*dx/c
cfl = 0.5
p = np.zeros(N); p[45:55] = 1.
u = np.zeros(N)
f = p + Z*u
g = p - Z*u

def getParam(h_iup, h_i, D):
        c0 = h_i
        c1 = (h_iup - h_i)/D
        return c0, c1

time_step = 100
for itr in range(time_step):
        cf0, cf1 = getParam(f[:-1], f[1:], -dx) #param for f
        cg0, cg1 = getParam(g[1:], g[:-1], dx) #param for g

        f[1:] = cf1*(-c*dt) + cf0
        g[:-1] = cg1*(c*dt) + cg0

        f[0] = r_0*g[0]
        g[-1] = r_N*f[-1]
```

図 2 に解析結果を示す。図から明らかなように、FDM 結果はシャープな圧力面を維持できていない。時間がたつとともに圧力界面はぼやけてくる (図示せず)。このような現象を数値拡散という。移流問題において数値拡散はときに厄介な存在であり、対策を考えなければならないこともある。

2.3　FVM による解法

2.3.1　空間補間

FVM でも同様に式 (6)(7) による空間補間を行うが、FDM と制約の与え方が異なる。FDM では係数の決定に離散点における物理量 h_i を用いていたが、FVM では格子内の h の積分値を用いる。つまり、多項式の係数は

$$\int_X h^i(x,t)dx = H_i$$

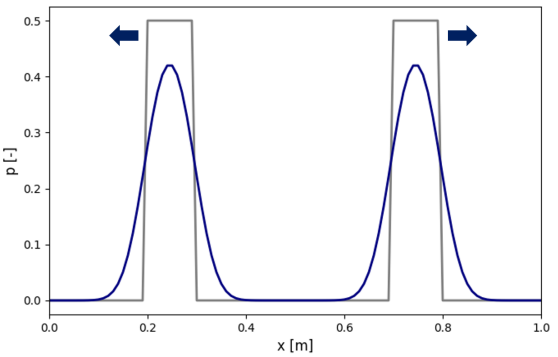

図 2　FDM による解析結果。図中グレーは理論解、青は FDM。どちらも 50 タイムステップ後の結果。図中矢印はそれぞれの波面の進む向きを表している。

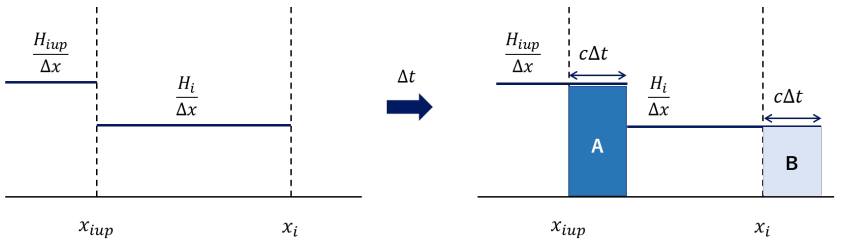

図 3　FVM における積分値の時間変化の概念図。

の制約式より求める。ここで X は格子内の x に関する領域であり、$f(g)$ の場合は $[x_{i-1}, x_i]([x_i, x_{i+1}])$ である。また、H_i は数値計算結果として保持されている積分値である。上式より、各波に対して制約式が 1 つなため、多項式は図 1(b) のような 0 次でなければならない。このとき、上式の積分は

$$\int_X c_0 dx = c_0 \Delta x = H_i$$

と書き換えられるため、係数は $H_i/\Delta x$ だと分かる (表 2)。

2.3.2　移流処理

前項で求められた多項式は当然ながら $h(x, t+\delta t) = h(x - sc\delta t, t)$ に従い移流する。ただし、FDM のときとは違って FVM の場合は積分値を時間発展しなければならない。つまり、式 (9) とは別の形の時間発展処理を施すことになる。

図 3 は多項式が移流したときの概念図である (前項より多項式は 0 次であるため、図 3 は階段状のプロファイルとなっている)。時刻が Δt だけ進んだとき、プロファイルは図 3 中の左図から右図のように変化する。その結果、格子点 x_{iup} および x_i 間の積分値は領域 B の分だけ差分され、領域 A の分だけ増加する。つまり、このセル内の積分値 H_i は

$$H_i \leftarrow H_i + \frac{H_{iup}}{\Delta x} c\Delta t - \frac{H_i}{\Delta x} c\Delta t \tag{10}$$

のように時間変化する。これこそ正に FVM における時間発展処理であり、FDM における式 (9) に相当する。

2.3.3　境界条件

最後に B.C. について議論する。例えば図 3 の H が F(つまり進行波) を指していて、x_{iup} が境界部分であったとする。このとき外部から流入する”A”は定義できない。領域外に物理量は定義されていないためである。ただし境界部分の場合、流入してくる物理量は反射波であることが直ぐに気付く。

図 4 は境界条件反映に関する概念図である。境界 x_0 における流入は、x_0 で後退波より流出した量、つまり図中 A より求まる。ただし、反射率が r であった場合、実際に F に流入されるのは反射率の分だけ少なくなる。以上より、B.C. を加味した結果、F_1 の更新は

$$F_1 \leftarrow F_1 + r\frac{G_1}{\Delta x} c\Delta t - \frac{F_1}{\Delta x} c\Delta t$$

表 3　FVM における係数 (表中 H は列毎に F もしくは G に置き換える)

	f	g
c_0	$H_i/\Delta x$	

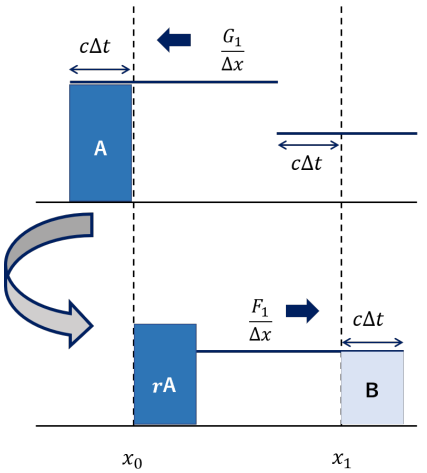

図 4　FVM における境界条件反映の概念図。

となる (同様の手続きで後退波に関する境界条件の反映も求まる)。

2.3.4　アルゴリズムと python コード例

以上が FVM の考え方である。FDM と違って積分値を更新するが、それ以外は概ね似ている。以下は FVM におけるアルゴリズムである。

FVM による移流方程式の求解

1. 初期条件として与えられた圧力 p および速度 u より、$f = p + Zu$ と $g = p - Zu$ のセル積分値 F および G を計算する。
2. 表 2 に従い各計算格子間の多項式を決定する。
3. 式 (10) を用いて状態を更新する。ただし多項式が定義されていないゆえに更新できないセル (F_0など) は B.C. に従い更新する。
4. 2.-3. を繰り返す。

FVM による解法の python コード例は以下の通りである (シミュレーション条件は 2.2.4 項と同様である)。

```
import numpy as np
import math

N = 101
dx = 0.01
rho = 1.2
k = 1.4e+5
c = math.sqrt(k/rho)
Z = math.sqrt(rho*k)
r_0 = 0.5
r_N = 0.5
cfl = 0.5
dt = cfl*dx/c

p = np.zeros(N); p[45:55] = 1.
u = np.zeros(N)
F = np.zeros(N-1); F[45:54] = p[45:54]*dx
G = np.zeros(N-1); G[45:54] = p[45:54]*dx


def getParam(H_i):
        c0 = H_i/dx

        return c0

time_step = 100
for itr in range(time_step):
```

```
cf0 = getParam(F) #param for F
cg0 = getParam(G) #param for G

F_0 = F[0] + r_0*(G[0]/dx)*(c*dt) - (F[1]/dx)*(c*dt)
G_N = G[-1] + r_N*(F[-1]/dx)*(c*dt) - (G[-2]/dx)*(c*dt)

F[1:] = F[1:] + (F[:-1]/dx)*(c*dt) - (F[1:]/dx)*(c*dt)
G[:-1] = G[:-1] + (G[1:]/dx)*(c*dt) - (G[:-1]/dx)*(c*dt)

F[0] = F_0
G[-1] = G_N
```

図 5 に解析結果を示す。多項式の次数が低い分、FDM よりも正確さに欠ける結果になったと思われる。

3 1 次元空間における CIP

前章では FDM や FVM による移流方程式の求解を議論した。FDM は離散点の物理量を基に空間補間を行い、FVM は格子内の積分値を基に空間補間を行った。いずれも 1 種のモーメントしか用いてないので、シングルモーメント法と言える。一方で、本章より議論する CIP は積極的に多種のモーメントを利用する。原案の CIP は物理量と微分値を利用しているが、マルチモーメント法であることを印象付けるように、それ以外のモーメントも利用することで様々な発展形が提案された。

3.1 CIP

原案の CIP は多項式補間に物理量と微分値を利用する。実は、特に移流速度が一定な移流方程式は微分値を上手く扱うことができる。というのも、移流方程式の両辺を x で偏微分してみると、その結果は

$$\partial_t(\partial_x h) + c\partial_x(\partial_x h) = 0 \tag{11}$$

であり、$\partial_x h$ も移流方程式に従うことが分かる。したがって、シミュレーション中で $\partial_x h$ を陽に扱い、前章のように時間発展させることも可能である。

では、各格子の物理量 h_i と微分値 $\partial_x h_i$ がシミュレーションで所与であるとしよう。このとき、多項式 $h^i(x,t)$ には 4 つの制約、$h^i(x_i,t) = h_i, h^i(x_{iup},t) = h_{iup}, \partial_x h^i(x_i,t) = \partial_x h_i, \partial_x h^i(x_{iup},t) = \partial_x h_{iup}$ が与えられる。そのため、$h^i(x,t)$ に 3 次の多項式を採用することができ、その係数は連立一次方程式

$$\begin{aligned}
h^i(x_i,t) &= c_0 = h_i \\
\partial_x h^i(x_i,t) &= c_1 = \partial_x h_i \\
h^i(x_{iup},t) &= c_3 D^3 + c_2 D^2 + c_1 D + c_0 = h_{iup} \\
\partial_x h^i(x_{iup},t) &= 3c_3 D^2 + 2c_2 D + c_1 = \partial_x h_{iup}
\end{aligned}$$

より求まる。この連立一次方程式の解を表 4 に纏めた。

微分値が加わることで、FDM の時と比べ多項式の次数が増加した。これにより解の精度向上が期待できる。もちろん微分値の移流方程式を解く分、計算コストはこれまでより増加するが、それでも CIP を採用する恩恵の方が大きい。CIP における解の更新は FDM と同様で、

$$\begin{aligned}
h_i &\leftarrow h^i(x - sc\Delta t, t) = \textstyle\sum_{m=0}^{3} c_m(-sc\Delta t)^m \\
\partial_x f_i &\leftarrow \partial_x f^i(x - sc\Delta t, t) = \textstyle\sum_{m=1}^{3} m c_m(-sc\Delta t)^{m-1}
\end{aligned} \tag{12}$$

のように施せばよい。

境界条件に関しては 2.2 節から少し変更がある。$f(g)$ に対して $f_0 = r_0 g_0 (g_N = r_N f_N)$ と施す点は同様だが、微分値の場合は反射特性の定義より $\partial_x f_0 = -r_0 \partial_x g_0 (\partial_x g_N = -r_N \partial_x f_N)$ としなければならない。

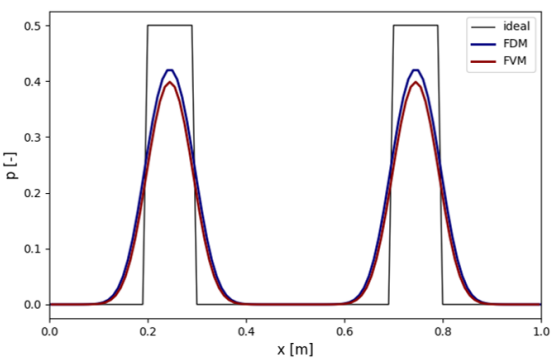

図 5　FVM による解析結果。図中グレーは理論解、青は FDM(図 2)、赤は FVM。どちらも 50 タイムステップ後の結果。

表 4　CIP における係数 (表中 h は列毎に f もしくは g に置き換える)

	f	g
iup	$i-1$	$i+1$
D	$-\Delta x$	Δx
c_0	h_i	
c_1	$\partial_x h_i$	
c_2	$3(h_{iup}-h_i)/D^2-(2\partial_x h_i+\partial_x h_{iup})/D$	
c_3	$(\partial_x h_i+\partial_x h_{iup})/D^2+2(h_i-h_{iup})/D^3$	

本節の議論は複数回の偏微分値 $\partial_x^n f$ でも成立する。したがって原案の CIP 以上に高精度な手法作成は非常に容易と言える。計算リソースの許す限り多数のモーメントの移流方程式を解けばよい訳である。

3.1.1　アルゴリズムと python コード例

以上が CIP の概要である。議論の流れが 2.2 節に似ていると理解頂けたと思う。以下は CIP のアルゴリズムである。

CIP による移流方程式の求解

1. 初期条件として与えられた圧力 p 及び速度 u とそれらの微分値 $\partial_x, \partial_x u$ より、$f=p+Zu$ と $g=p-Zu$、並びに $\partial_x f=\partial_x p+\partial_x u$ と $\partial_x g=\partial_x p-\partial_x u$ を計算する。
2. 表 4 に従い各格子点間の多項式を決定する。
3. 式 (12) に従い状態を更新する。もしくは B.C. を反映させる。
4. 2.-3. を繰り返す。

CIP による解法の python コード例を紹介する。シミュレーション条件は 2.2.4 と同様である。また、微分値は圧力と速度のどちらも一様にゼロとした。

```
import numpy as np
import math

N = 101
dx = 0.01
rho = 1.2
k = 1.4e+5
c = math.sqrt(k/rho)
Z = math.sqrt(rho*k)
r_0 = 0.5
r_N = 0.5
cfl = 0.5
dt = cfl*dx/c

p = np.zeros(N); p[45:55] = 1.; dp = np.zeros(N)
u = np.zeros(N); du = np.zeros(N)
f = p + Z*u
g = p - Z*u
df = dp + Z*du
dg = dp - Z*du

def getParam(h_i, h_iup, dh_i, dh_iup, D):
        c0 = h_i
        c1 = dh_i
        c2 = 3.*(h_iup - h_i)/(D**2) - (2.*dh_i + dh_iup)/D
        c3 = (dh_i + dh_iup)/(D**2) + 2.*(h_i - h_iup)/(D**3)

        return c0, c1, c2, c3

time_step = 100
for itr in range(time_step):
        cf0, cf1, cf2, cf3 = getParam(f[1:], f[:-1], df[1:], df[:-1], -dx)
        cg0, cg1, cg2, cg3 = getParam(g[:-1], g[1:], dg[:-1], dg[1:], dx)
```

```
f[1:] = cf3*((-c*dt)**3) + cf2*((-c*dt)**2) + cf1*(-c*dt) + cf0
g[:-1] = cg3*((c*dt)**3) + cg2*((c*dt)**2) + cg1*(c*dt) + cg0

df[1:] = 3.*cf3*((-c*dt)**2) + 2.*cf2*(-c*dt) + cf1
dg[:-1] = 3.*cg3*((c*dt)**2) + 2.*cg2*(c*dt) + cg1

f[0] = r_0*g[0]
g[-1] = r_N*f[-1]
df[0] = -r_0*dg[0]
dg[-1] = -r_N*df[-1]
```

図 6 に CIP の解析結果を示す。図 5 の結果と比べて、圧力界面のシャープさを維持できている。これは CIP の低い位相誤差に起因する。一方で界面付近にオーバーシュートがあるが、これは 3 次多項式になって顕在化したギブスジャンプである。どちらかと言えば良くない現象だが、時間と共に増大することはない。とにかく、FDM や FVM と比べ非常に良い結果が得られたと言える。

3.2 RCIP

例えば混和性の多相流れなど、界面の濃度勾配が重要な問題では前節のオーバーシュートは好ましくない。そこで、対策として有理関数を用いた CIP が提案された。これは 3 次多項式の代わりに

$$h^i(x,t) = \frac{c_3(x-x_i)^3 + c_2(x-x_i)^2 + c_1(x-x_i) + c_0}{1+\alpha B(x-x_i)} \tag{13}$$

を用いる手法である。ここで α は 0 から 1 を取るハイパーパラメータで、$\alpha = 0$ のときは CIP と同値になる。B は

$$B = \frac{\left|\frac{S-\partial_x h_i}{\partial_x h_{iup}-S}\right| - 1}{D}, \quad S = \frac{h_{iup}-h_i}{D}$$

より求めることができる。また、式 (13) の偏微分は

$$\partial_x h^i(x,t) = \frac{3c_3(x-x_i)^2 + 2c_2(x-x_i) + c_1 - \alpha B h^i(x,t)}{1+\alpha B(x-x_i)} \tag{14}$$

となる。式 (13)(14) 中の各係数は表 5 の通りである。

補間方法が変われど、それ以外はほとんど同じである。以下に python コード例を示す。

```
import numpy as np
import math

N = 101
dx = 0.01
rho = 1.2
k = 1.4e+5
c = math.sqrt(k/rho)
Z = math.sqrt(rho*k)
r_0 = 0.5
r_N = 0.5
```

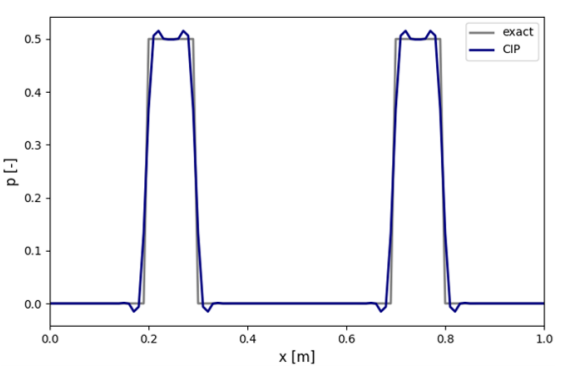

図 6　CIP による解析結果。図中グレーは理論解、青は CIP。どちらも 50 タイムステップ後の結果。

表 5　RCIP における係数 (表中 h は列毎に f もしくは g に置き換える)

	f	g
iup	$i-1$	$i+1$
D	$-\Delta x$	Δx
c_0	h_i	
c_1	$\partial_x h_i + \alpha B h_i$	
c_2	$S\alpha B + (S - \partial_x h_i)/D - c_3 D$	
c_3	$(\partial_x h_i - S + (\partial_x h_{iup} - S)(1 + \alpha B D))/D^2$	

```
cfl = 0.5
dt = cfl*dx/c

p = np.zeros(N); p[45:55] = 1.; dp = np.zeros(N)
u = np.zeros(N); du = np.zeros(N)
f = p + Z*u
g = p - Z*u
df = dp + Z*du
dg = dp - Z*du
alpha = 1.


def getParam(h_i, h_iup, dh_i, dh_iup, D, alpha):
        S = (h_iup-h_i)/D
        B = (np.abs((S - dh_i)/(dh_iup - S + 1e-10)) - 1.)/D + 1e-10
        c0 = h_i
        c1 = dh_i + h_i*alpha*B
        c3 = (dh_i - S + (dh_iup - S)*(1. + alpha*B*D))/(D**2)
        c2 = S*alpha*B + (S - dh_i)/D - c3*D

        return c0, c1, c2, c3, B

time_step = 100
for itr in range(time_step):
        cf0, cf1, cf2, cf3, Bf = getParam(f[1:], f[:-1], df[1:], df[:-1], -dx, alpha)
        cg0, cg1, cg2, cg3, Bg = getParam(g[:-1], g[1:], dg[:-1], dg[1:], dx, alpha)

        f[1:] = (cf3*((-c*dt)**3) + cf2*((-c*dt)**2) + cf1*(-c*dt) + cf0)/(1.+alpha*Bf*(-c*dt))
        g[:-1] = (cg3*((c*dt)**3) + cg2*((c*dt)**2) + cg1*(c*dt) + cg0)/(1.+alpha*Bg*(c*dt))

        df[1:] = (3.*cf3*((-c*dt)**2) + 2.*cf2*(-c*dt) + cf1 - alpha*Bf*f[1:])/ \
                (1.+alpha*Bf*(-c*dt))
        dg[:-1] = (3.*cg3*((c*dt)**2) + 2.*cg2*(c*dt) + cg1 - alpha*Bg*g[:-1])/ \
                (1.+alpha*Bg*(c*dt))

        f[0] = r_0*g[0]
        g[-1] = r_N*f[-1]
        df[0] = -r_0*dg[0]
        dg[-1] = -r_N*df[-1]
```

図 7 に解析結果を示す。図から明らかなように、RCIP の結果ではオーバーシュートが見られない。一方で圧力界面は少しぼやけてしまう。音響解析の場合、これは過度な音圧減衰を招く。

3.3　CIP-CSL2

本資料の冒頭で述べた通り、音響現象には保存則による描像と移流による描像があり、CIP は後者を用いる。CIP に限らず移流性を利用するシミュレーション手法は、計算結果の保存性に難がある。本来保存される運動量や質量が計算とともに減少 (もしくは増加) してしまう訳である。これは物理の第一原理に反するため、当然好ましくない (ただし世に出ている多くのシミュレーション手法が保存則を厳密に満たす訳ではないので、矢面に立たされている CIP は少し不憫だと個人的に感じる)。そんな中 CIP-CSL という手法が提案された。

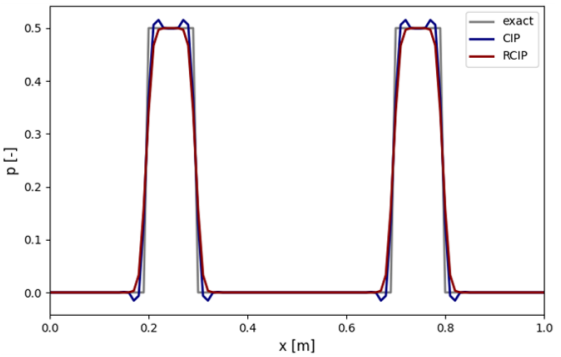

図 7　RCIP による解析結果。図中グレーは理論解、青は CIP、赤は RCIP。どちらも 50 タイムステップ後の結果。

CSL は Conservative Semi Lagrangian の略で、その名の通り保存性を謳っている (Semi Lagrangian はシミュレーションの分類名で、本資料では議論しない)。そして、保存性を向上させるために積分値のモーメントを利用する。

CIP-CSL にはいくつか種類があり、本節で議論するのは CIP-CSL2 である。これは格子点の物理量と格子点間の積分値を利用する。したがって 2 章で紹介した FDM と FVM のハイブリッドと言えるだろう。制約式は 3 つであるため、2 次の多項式で空間補間する (それゆえ名前に 2 という数字が付いている)。まず、2 つの制約式

$$h^i(x_i,t) = h_i = c_0$$
$$h^i(x_{iup},t) = c_2D^2 + c_1D + c_0$$

はこれまでと同様である。これに加えて、

$$\int h^i(x,t)dx = \frac{1}{3}c_2(x-x_i)^3 + \frac{1}{2}c_1(x-x_i)^2 + c_0(x-x_i) + C$$

であるため (C は積分定数)、積分値の制約は

$$H_i = -s\left\{\frac{1}{3}c_2D^3 + \frac{1}{2}c_1D^2 + c_0D\right\}$$

と書ける。以上の連立一次方程式を解いた結果を表 6 に示す。

物理量 h の更新は CIP と同様で、積分値 H の更新は FVM と同様である。本節では H の更新のみ議論する。2.3 節と異なる点は多項式の次数であろう。つまり、図 3 中 B の流出する量が

$$\Delta H_i = -s\left\{\frac{1}{3}c_2(-sc\Delta t)^3 + \frac{1}{2}c_1(-sc\Delta t)^2 + c_0(-sc\Delta t)\right\} \tag{15}$$

のように書き表される (ただしクーラン数は 1 以下と仮定する)。一方で図中 A の流入する量は ΔH_{iup} であるため、時間発展の際に H_i を

$$H_i \leftarrow H_i + \Delta H_{iup} - \Delta H_i \tag{16}$$

に従い更新すればよい。また、B.C. の扱いは図 4 の通りである。

アルゴリズムは FDM や FVM と非常に似ている。以下に python コード例を示す。

```
import numpy as np
import math
```

表 6　CIP-CSL2 における係数 (表中 h は列毎に f もしくは g に置き換える)

	f	g
iup	$i-1$	$i+1$
D	$-\Delta x$	Δx
s	1	-1
c_0	h_i	
c_1	$-6sH_i/D^2 - 2(h_{iup}+2h_i)/D$	
c_2	$6sH_i/D^3 + 3(h_{iup}+h_i)/D^2$	

```
N = 101
dx = 0.01
rho = 1.2
k = 1.4e+5
c = math.sqrt(k/rho)
Z = math.sqrt(rho*k)
r_0 = 0.5
r_N = 0.5
cfl = 0.5
dt = cfl*dx/c

p = np.zeros(N); p[45:55] = 1.
u = np.zeros(N)
f = p + Z*u
g = p - Z*u
F = (f[1:] + f[:-1])*dx/2.
G = (g[1:] + g[:-1])*dx/2.


def getParam(h_i, h_iup, H_i, s, D):
        c0 = h_i
        c1 = -6.*s*H_i/(D**2) - 2.*(h_iup + 2.*h_i)/D
        c2 = 6.*s*H_i/(D**3) + 3.*(h_iup + h_i)/(D**2)

        return c0, c1, c2

def getDeltaH(c0, c1, c2, s):
        epsilon = -s*c*dt
        return -s*(c2*(epsilon**3)/3. + c1*(epsilon**2)/2. + c0*epsilon)

time_step = 200
for itr in range(time_step):
        cf0, cf1, cf2 = getParam(f[1:], f[:-1], F, 1., -dx)
        cg0, cg1, cg2 = getParam(g[:-1], g[1:], G, -1., dx)

        F_0 = F[0] + r_0*getDeltaH(cg0[0], cg1[0], cg2[0], -1.) - \
                getDeltaH(cf0[0], cf1[0], cf2[0], 1.)
        G_N = G[-1] + r_N*getDeltaH(cf0[-1], cf1[-1], cf2[-1], 1.) - \
                getDeltaH(cg0[-1], cg1[-1], cg2[-1], -1.)

        F[1:] += getDeltaH(cf0[:-1], cf1[:-1], cf2[:-1], 1.) - \
                getDeltaH(cf0[1:], cf1[1:], cf2[1:], 1.)
        G[:-1] += getDeltaH(cg0[1:], cg1[1:], cg2[1:], -1.) - \
                getDeltaH(cg0[:-1], cg1[:-1], cg2[:-1], -1.)
        F[0] = F_0; G[-1] = G_N

        f[1:] = cf2*((-c*dt)**2) + cf1*(-c*dt) + cf0
        g[:-1] = cg2*((c*dt)**2) + cg1*(c*dt) + cg0

        f[0] = r_0*g[0]
        g[-1] = r_N*f[-1]
```

3.4 CIP-CSL4

CIP-CSL2 では積分値と格子点の物理量を用いたが、Cip-CSL4 では格子点の微分値も利用する。それゆえ、CIP-CSL4 は CIP と FVM のハイブリッドと言える。微分や物理量、並びに積分の取り扱い方はこれまでに見てきたので、目新しさはそれ程ない。ただし 5 つの制約式が与えられるため、CIP-CSL4 では 4 次の多項式を利用する。

制約式は以下の通りである。まず、CIP と同様に

$$
\begin{aligned}
&h^i(x_i,t) = h_i = c_0 \\
&\partial_x h^i(x_i,t) = \partial_x h_i = \partial_x h_i = c_1 \\
&h^i(x_{iup},t) = h_{iup} = c_4 D^4 + c_3 D^3 + c_2 D^2 + c_1 D + c_0 \\
&\partial_x h^i(x_{iup},t) = \partial_x h_{iup} = 4c_4 D^3 + 3c_3 D^2 + 2c_2 D + c_1
\end{aligned}
$$

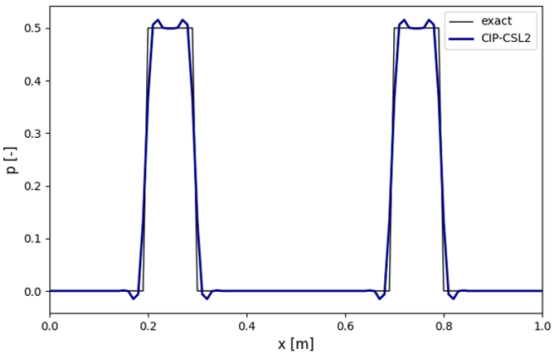

図 8　CIP-CSL2 による解析結果。図中グレーは理論解、青は CIP-CSL2。どちらも 50 タイムステップ後の結果。

表 7　CIP-CSL4 における係数 (表中 h は列毎に f もしくは g に置き換える)

	f	g
iup	$i-1$	$i+1$
D	$-\Delta x$	Δx
s	1	-1
c_0	h_i	
c_1	$\partial_x h_i$	
c_2	$-30sH_i/D^3 - 6(3h_i + 2h_{iup})/D^2 - 3(3\partial_x h_i - \partial_x h_{iup})/(2D)$	
c_3	$-2c_2/D + 4(h_{iup} - h_i)/D^3 - (3\partial_x h_i + \partial_x h_{iup})/D^2$	
c_4	$-c_3/D - c_2/D^2 + (h_{iup} - h_i)/D^4 - \partial_x h_i/D^3$	

が与えられる。また、多項式の積分は

$$\int h^i(x,t)dx = \frac{1}{5}c_4(x-x_i)^5 + \frac{1}{4}c_3(x-x_i)^4 + \frac{1}{3}c_2(x-x_i)^3 + \frac{1}{2}c_1(x-x_i)^2 + c_0(x-x_i) + C$$

であるため (C は積分定数)、積分値に関する制約は

$$H_i = -s\left\{\frac{1}{5}c_4D^5 + \frac{1}{4}c_3D^4 + \frac{1}{3}c_2D^3 + \frac{1}{2}c_1D^2 + c_0D\right\}$$

と書ける。したがって図 3 中 B に相当する流出量 ΔH_i は

$$\Delta H_i = -s\left\{\frac{1}{5}c_4(-sc\Delta t)^5 + \frac{1}{4}c_3(-sc\Delta t)^4 + \frac{1}{3}c_2(-sc\Delta t)^3 + \frac{1}{2}c_1(-sc\Delta t)^2 + c_0(-sc\Delta t)\right\}$$

となる。積分値の更新は式 (16) の通りで、B.C. の扱いもこれまでと同様である。表 7 に多項式の係数を纏めた。

```
import numpy as np
import math

N = 101
dx = 0.01
rho = 1.2
k = 1.4e+5
c = math.sqrt(k/rho)
Z = math.sqrt(rho*k)
r_0 = 0.5
r_N = 0.5
cfl = 0.5
dt = cfl*dx/c

p = np.zeros(N); p[45:55] = 1.; dp = np.zeros(N)
u = np.zeros(N); du = np.zeros(N)
```

```
f = p + Z*u; df = dp + Z*du
g = p - Z*u; dg = dp - Z*du
F = (f[1:] + f[:-1])*dx/2.
G = (g[1:] + g[:-1])*dx/2.


def getParam(h_i, h_iup, dh_i, dh_iup, H_i, s, D):
        c0 = h_i
        c1 = dh_i
        c2 = -30.*s*H_i/(D**3) - 6.*(3.*h_i+2.*h_iup)/(D**2) - 3.*(3.*dh_i - dh_iup)/(2.*D)
        c3 = -2.*c2/D + 4.*(h_iup - h_i)/(D**3) - (3.*dh_i + dh_iup)/(D**2)
        c4 = -c3/D - c2/(D**2) + (h_iup - h_i)/(D**4) - dh_i/(D**3)

        return c0, c1, c2, c3, c4

def getDeltaH(c0, c1, c2, c3, c4, s):
        epsilon = -s*c*dt
        return -s*(c4*(epsilon**5)/5. + c3*(epsilon**4)/4. +c2*(epsilon**3)/3.\
                + c1*(epsilon**2)/2. + c0*epsilon)


time_step = 100
for itr in range(time_step):
        cf0, cf1, cf2, cf3, cf4 = getParam(f[1:], f[:-1], df[1:], df[:-1], F, 1., -dx)
        cg0, cg1, cg2, cg3, cg4 = getParam(g[:-1], g[1:], dg[:-1], dg[1:], G, -1., dx)

        F_0 = F[0] + r_0*getDeltaH(cg0[0], cg1[0], cg2[0], cg3[0], cg4[0], -1.) -\
                getDeltaH(cf0[0], cf1[0], cf2[0], cf3[0], cf4[0], 1.)
        G_N = G[-1] + r_N*getDeltaH(cf0[-1], cf1[-1], cf2[-1], cf3[-1], cf4[-1], 1.) -\
                getDeltaH(cg0[-1], cg1[-1], cg2[-1], cg3[-1], cg4[-1], -1.)

        F[1:] += getDeltaH(cf0[:-1], cf1[:-1], cf2[:-1], cf3[:-1], cf4[:-1], 1.) - \
                getDeltaH(cf0[1:], cf1[1:], cf2[1:], cf3[1:], cf4[1:], 1.)
        G[:-1] += getDeltaH(cg0[1:], cg1[1:], cg2[1:], cg3[1:], cg4[1:], -1.) - \
                getDeltaH(cg0[:-1], cg1[:-1], cg2[:-1], cg3[:-1], cg4[:-1], -1.)
        F[0] = F_0; G[-1] = G_N

        f[1:] = cf4*((-c*dt)**4) + cf3*((-c*dt)**3) + cf2*((-c*dt)**2) + cf1*(-c*dt) + cf0
        g[:-1] = cg4*((c*dt)**4) + cg3*((c*dt)**3) + cg2*((c*dt)**2) + cg1*(c*dt) + cg0

        df[1:] = 4.*cf4*((-c*dt)**3) + 3.*cf3*((-c*dt)**2) + 2.*cf2*(-c*dt) + cf1
        dg[:-1] = 4.*cg4*((c*dt)**3) + 3.*cg3*((c*dt)**2) + 2.*cg2*(c*dt) + cg1

        f[0] = r_0*g[0]
        g[-1] = r_N*f[-1]
        df[0] = -r_0*dg[0]
        dg[-1] = -r_N*df[-1]
```

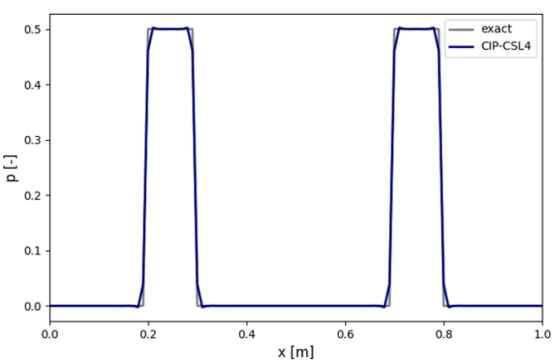

図 9　CIP-CSL4 による解析結果。図中グレーは理論解、青は CIP-CSL4。どちらも 50 タイムステップ後の結果。

4 多次元 CIP の前準備

前章までは 1 次元の移流方程式について議論していた。1 次元空間では進行波と後退波しかないので、いわゆる f と g のみを考えればよかった。一方で多次元の場合、波が進む向きは無数にあるため、これまでの延長線上のような解き方はできない。そこで、フラクショナルステップ法の利用を考える。

例えば支配方程式が

$$\partial_t q = \mathcal{N}_1(q) + \mathcal{N}_2(q) + ... + \mathcal{N}_M(q)$$

のように複数のオペレータで表記できるとき、時間発展を

$$\begin{cases} q^{(1)} = q^t + \mathcal{N}_1(q^t)\Delta t \\ q^{(2)} = q^{(1)} + \mathcal{N}_2(q^{(1)})\Delta t \\ ... \\ q^{t+1} = q^{(M-1)} + \mathcal{N}_M(q^{(M-1)})\Delta t \end{cases}$$

のように分割して行う。このような手法をフラクショナルステップ法 (もしくは方向分離) と言う。

2 次元の質量保存則と運動量保存則は

$$\partial_t \begin{bmatrix} p \\ u \\ v \end{bmatrix} + \begin{bmatrix} 0 & cZ & 0 \\ 1/\rho_0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \partial_x \begin{bmatrix} p \\ u \\ v \end{bmatrix} + \begin{bmatrix} 0 & 0 & cZ \\ 0 & 0 & 0 \\ 1/\rho_0 & 0 & 0 \end{bmatrix} \partial_y \begin{bmatrix} p \\ u \\ v \end{bmatrix} = \mathbf{0}$$

のように纏めることができる (v は y 方向の速度)。これをフラクショナルステップ法に従い

$$\partial_t \begin{bmatrix} p \\ u \\ v \end{bmatrix} + \begin{bmatrix} 0 & cZ & 0 \\ 1/\rho_0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \partial_x \begin{bmatrix} p \\ u \\ v \end{bmatrix} = \mathbf{0} \tag{17}$$

$$\partial_t \begin{bmatrix} p \\ u \\ v \end{bmatrix} + \begin{bmatrix} 0 & 0 & cZ \\ 0 & 0 & 0 \\ 1/\rho_0 & 0 & 0 \end{bmatrix} \partial_x \begin{bmatrix} p \\ u \\ v \end{bmatrix} = \mathbf{0} \tag{18}$$

のように分割する。そして式 (17) に従って状態を更新した後、式 (18) に従って状態を再び更新する。本章では FDM と FVM による解法を議論する。なお本資料を通して、式の導出等の議論は 2 次元までとする。これはアルゴリズム的に 3 次元は 2 次元の延長線上にあるためである。また、計算格子も正方格子に限定し、y 方向の格子刻み幅も Δx と表記する。

4.1 FDM による解法

式 (17) を見ると速度 v は時間変化しないことが分かる。そのため、初めのステップでは p と u のみを更新すればよいことが分かる。また、式 (17) を変換することで

$$\partial_t \begin{bmatrix} f_x \\ g_x \end{bmatrix} + \begin{bmatrix} c & 0 \\ 0 & -c \end{bmatrix} \partial_x \begin{bmatrix} f_x \\ g_x \end{bmatrix} = \mathbf{0} \tag{19}$$

が得られることも明らかであろう (v は無視したため、2 次元の状態ベクトルになっていることに注意)。ここで $f_x(g_x) = p \pm Zu$ である。各成分は正に 1 次元移流方程式と同じ形をしており、これまでに何度も解いてきた。以上より FDM の場合、初めのステップでは p と u を更新するのではなく、代わりに h_x を更新する。これらは 2 次元空間で定義されているが、これまでと同様に、$h_x(x, y, t + \delta t) = h_x(x - sc\delta t, y, t)$ のように更新すればよい。

前述の通り FDM の場合、各格子点での物理量 $h_x(x_i, y_j, t) = h_{ij}$ が与えられる。初めのステップは y 方向の移流がないので、(x_i, y_j) と (x_{iup}, y_j) 間を補間すればよい。2 点の物理量が所与のため、補間式 $h^i(x, t)$ は 2.2 節と同様に 1 次多項式 $c_1^x(x - x_i) + c_0^x$ となる。多項式が得られたあとは、これまで同様に

$$h_{ij} \leftarrow c_1(-sc\Delta t) + c_0$$

に従い更新する。ただし、本来考えていた支配方程式は式 (17)(18) なので、次に進む前に、p と u も更新させる。これらは定義より、$p = (f + g)/2$、$u = (f - g)/(2Z)$ より求まる。

式 (18) の場合は u が変化しない。また、先程と同様に式 (18) は

$$\partial_t \begin{bmatrix} f_y \\ g_y \end{bmatrix} + \begin{bmatrix} c & 0 \\ 0 & -c \end{bmatrix} \partial_y \begin{bmatrix} f_y \\ g_y \end{bmatrix} = \mathbf{0} \tag{20}$$

と書き換えることができる。ここで、$f_y(g_y) = p \pm Zv$ である。そして今度は y の方向に補間を施し、移流方程式に従って $h(x, y, t)$ を更新する。最後に $p = (f_y + g_y)/2$ 及び $v = (f_y - g_y)/(2Z)$ と計算することで、系の状態は晴れて 1 タイムステップ分更新される。

4.1.1 アルゴリズムと python コード例

フラクショナルステップ法を利用するため、2 次元でのアルゴリズムは 1 次元よりも長くなる。しかしながら、基本的には同じことの繰り返しなので、それ程難しくはない。以下は FDM のアルゴリズムである。

FDM による 2 次元移流方程式の求解

1. 所与の圧力と速度より、$f_x = p + Zu$ と $g_x = p - Zu$ を計算する。
2. 空間を補間し、状態を更新 (移流) する。また、必要であれば B.C. を反映させる。
3. 更新された f_x および g_x に従い、p と u も更新する。
4. 所与の圧力と速度より、$f_y = p + Zv$ と $g_y = p - Zv$ を計算する。
5. 空間を補間し、状態を更新 (移流) する。また、必要であれば B.C. を反映させる。
6. 更新された f_y および g_y に従い、p と v も更新する。
7. 1.-6. を繰り返す。

python コード例を以下に記す。シミュレーション条件は表 2 と同様だが、計算格子を x と y の両方向に 101 だけ用意した。また、速度の初期条件は一様にゼロで、圧力の初期条件は以下とする。

$$p(x, y, 0) = \begin{cases} 1 & (0.45 \leq x \leq 0.55, \quad 0.45 \leq y \leq 0.55) \\ 0 & (\text{otherwise}) \end{cases}$$

```
import numpy as np
import math

Nx, Ny = (101, 101)
dx = 0.01
rho = 1.2
k = 1.4e+5
c = math.sqrt(k/rho)
Z = math.sqrt(rho*k)
r = 0.5
cfl = 0.5
dt = cfl*dx/c

p = np.zeros((Nx, Ny)); p[45:55,45:55] = 1.
u = np.zeros((Nx, Ny)); v = np.zeros((Nx, Ny))

def getParam(h_i, h_iup, D):
        c0 = h_i
        c1 = (h_iup-h_i)/D

        return c0, c1

time_step = 100
for itr in range(time_step):
        fx = p + Z*u
        gx = p - Z*u

        cf0, cf1 = getParam(fx[1:,:], fx[:-1,:], -dx)
        cg0, cg1 = getParam(gx[:-1,:], gx[1:,:], dx)

        fx[1:,:] = cf1*(-c*dt) + cf0
        gx[:-1,:] = cg1*(c*dt) + cg0
        fx[0,:] = r*gx[0,:]
        gx[-1,:] = r*fx[-1,:]

        p = (fx + gx)/2.
        u = (fx - gx)/(2.*Z)

        fy = p + Z*v
        gy = p - Z*v

        cf0, cf1 = getParam(fy[:,1:], fy[:,:-1], -dx)
        cg0, cg1 = getParam(gy[:,:-1], gy[:,1:], dx)
```

```
fy[:,1:] = cf1*(−c*dt) + cf0
gy[:,:−1] = cg1*(c*dt) + cg0
fy[:,0] = r*gy[:,0]
gy[:,−1] = r*fy[:,−1]

p = (fy + gy)/2.
v = (fy − gy)/(2.*Z)
```

4.2 FVM による解法

次に FVM による解法を考える。この場合は図 10 に示すように、4 つの格子で構成されたセル内の積分値 H が与えられることになる。2.3 節と類似性を持たせるならば、2 つの格子間の積分、つまり線積分が欲しいところである。そこで、移流方程式をもう一度見直そう。

例えば進行波の移流方程式 $\partial_t f + c\partial_x f = 0$ を考える。このとき、移流方程式の両辺を領域 X で積分すると

$$\partial_t \int_X f dx + c\partial_x \int_X f dx = 0$$

が得られる。つまり移流速度が時空間的に一様であるならば、上記のような積分結果も移流方程式を満たす。図 10 のように、$\int_X h dx$ を h_x と書くことにする。同様な処理で y 方向の線積分から h_y が定義できる。こうなってくると図 10 の $H(i,j)$ は

$$H(i,j) = \int_Y h_x dy = \int_X h_y dx$$

であるため、H と h の関係は 2.3 節のそれらと同様になる。したがって、多次元の FVM では図 10 中の h を軸に移流方程式を解き進めればよい。

FVM による解法では H の値が与えられる。初期条件の p や $(u,v)^{\mathrm{T}}$ から H の初期条件を求めるには、単に $p \pm Zu$ や $p \pm Zv$ を積分すればよい。一般的に、FVM では格子内の物理量を一様だと考えるので、$H(i,j) = \{p(i,j) + Zu(i,j)\}\Delta x^2$ と求められる。更にまた、$H(i,j)$ を制約として使う訳だが、h_x や h_y の空間補間は 0 次多項式になる。従って、図 10 中の補間式は

$$h_x^i(x,t) = \frac{H(i,j)}{\Delta x} = \{p(i,j) + Zu(i,j)\}\,\Delta x$$

のように定まる (当然ながら y 方向も同様である)。

FVM でもフラクショナルステップ法を利用する。初めは x 方向に移流させて H を更新する。H の更新は図 3 及び図 4 と同様である。

4.2.1 アルゴリズムと python コード例

FVM における 2 次元移流方程式の求解

1. 所与の圧力と速度より、$F_x = (p + Zu)\Delta x^2$ と $G_x = (p - Zu)\Delta x^2$ を計算する。
2. 空間補間し、状態を更新する。必要であれば B.C. を反映させる。
3. 更新された F_x と G_x を、$p = (F_x + G_x)/\Delta x^2$ および $u = (F_x - G_x)/\Delta x^2$ より圧力と速度も更新する。
4. 所与の圧力と速度より、$F_y = (p + Zv)\Delta x^2$ と $G_y = (p - Zv)\Delta x^2$ を計算する。
5. 空間補間し、状態を更新する。必要であれば B.C. を反映させる。
6. 更新された F_y と G_y を、$p = (F_y + G_y)/(2\Delta x^2)$ および $v = (F_y - G_y)/(2\Delta x^2)$ より圧力と速度も更新する。
7. 1.-6. を繰り返す。

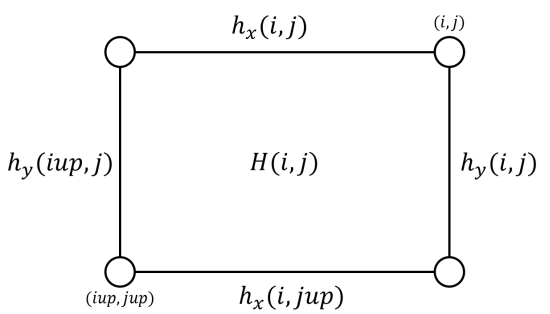

図 10　FVM による解析の概念図。積分値の定義位置を描画。

```
import numpy as np
import math

Nx, Ny = (100, 100)
dx = 0.01
rho = 1.2
k = 1.4e+5
c = math.sqrt(k/rho)
Z = math.sqrt(rho*k)
r = 0.5
cfl = 0.5
dt = cfl*dx/c

p = np.zeros((Nx, Ny)); p[45:54,45:54] = 1.
u = np.zeros((Nx, Ny)); v = np.zeros((Nx, Ny))


def getParam(H):
        c0 = H/dx

        return c0

def getDeltaH(c0):
        return c0*c*dt

time_step = 100
for itr in range(time_step):
        Fx = (p + Z*u)*(dx**2)
        Gx = (p - Z*u)*(dx**2)

        cf0 = getParam(Fx)
        cg0 = getParam(Gx)

        Fx0 = Fx[0,:] + r*getDeltaH(cg0[0,:]) - getDeltaH(cf0[1,:])
        GxN = Gx[-1,:] + r*getDeltaH(cf0[-1,:]) - getDeltaH(cg0[-2,:])

        Fx[1:,:] = Fx[1:,:] + getDeltaH(cf0[:-1,:]) - getDeltaH(cf0[1:,:])
        Gx[:-1,:] = Gx[:-1,:] + getDeltaH(cg0[1:,:]) - getDeltaH(cg0[:-1,:])
        Fx[0,:] = Fx0; Gx[-1,:] = GxN

        p = (Fx + Gx)/2./(dx**2)
        u = (Fx - Gx)/(2.*Z)/(dx**2)

        Fy = (p + Z*v)*(dx**2)
        Gy = (p - Z*v)*(dx**2)

        cf0 = getParam(Fy)
        cg0 = getParam(Gy)

        Fy0 = Fy[:,0] + r*getDeltaH(cg0[:,0]) - getDeltaH(cf0[:,1])
        GyN = Gy[:,-1] + r*getDeltaH(cf0[:,-1]) - getDeltaH(cg0[:,-2])

        Fy[:,1:] = Fy[:,1:] + getDeltaH(cf0[:,:-1]) - getDeltaH(cf0[:,1:])
        Gy[:,:-1] = Gy[:,:-1] + getDeltaH(cg0[:,1:]) - getDeltaH(cg0[:,:-1])
        Fy[:,0] = Fy0; Gy[:,-1] = GyN

        p = (Fy + Gy)/2./(dx**2)
        v = (Fy - Gy)/(2.*Z)/(dx**2)
```

5 多次元空間における CIP

3 章で紹介した通り、CIP は離散点での物理量の微分も利用する。そのために圧力や速度の微分値も陽に扱わなければなかった。つまり微分値も I.C. として与え、$\partial_x h$ を介して時間発展させていく。このような処理を多次元の CIP でも施す。

2 次元の場合でも格子間を 3 次の多項式で補間したいのであれば、物理量 (p, u, v) と各方向の偏微分 ($\partial_x p$、$\partial_x u$、$\partial_x v$、$\partial_y p$、$\partial_y u$、$\partial_y v$) も必要になってくる。そしてそれらの物理量を式 (17)(18) を参考に時間発展させなければならない。

まず、式 (17) に従って x 方向に移流する場合を考える。この間、v や $\partial_x v$、並びに $\partial_y v$ が不変であることは明らかである。そのため、$p, u, \partial_x p, \partial_x u, \partial_y p, \partial_y u$ のみ更新すればよい。p と u はこれまで通り $f_x = p + Zu$ と $g_x = p - Zu$ を介して更新できる。同様に $\partial_x p$ と $\partial_x u$ も $\partial_x f_x$ と $\partial_x g_x$ を用いれば更新できる ($\partial_x h_x$ が移流方程式を満たすことは 3 章で述べた通り)。一方で $\partial_y p$ と $\partial_y u$ の扱いにはかなり困る。確かに

$$\partial_t(\partial_y h_x) \pm c\partial_x(\partial_y h_x) = 0$$

であるから、$\partial_y h_x$ も移流方程式を満たすが、多項式の制約として使えない。これまでのように $h_x^i(x,t) = \sum c_m^x (x_i - x)^m$ と補間したとき、$\partial_y h_x^i$ は常にゼロであるため、連立一次方程式の一つとして組み込めない訳である。そこで、$\partial_y h_x$ は別途多項式を用意しよう。本資料では一般的な手法である M 型 CIP と C 型 CIP を議論する。

5.1 M 型 CIP

各格子点の $\partial_y p$ や $\partial_y u$ は所与であるため、前述の $\partial_y h_x$ も所与である。そのため、$\partial_y h_x$ を本資料で言う所の FDM 的方法は可能と言える。つまり、h_x の補間には 3 次多項式を利用する一方で、$\partial_y h_x$ には 1 次多項式を用いる。このような手法を M 型 CIP と言う。CIP の中で比べたとき、M 型 CIP は計算コストが少なく済む。

5.1.1 アルゴリズムと python コード例

M 型 CIP による 2 次元移流方程式の求解

1. 所与の圧力と速度、並びにそれらの偏微分値より、h_x と $\partial_x h_x$ と $\partial_y h_x$ を計算する。
2. 空間を補間し、状態を更新する。また、必要であれば B.C. を反映させる。
3. 更新された物理量を基に、圧力と速度、並びにそれらの偏微分値も更新する。
4. 所与の圧力と速度、並びにそれらの偏微分値より、h_y と $\partial_y h_y$ と $\partial_x h_y$ を計算する。
5. 空間を補間し、状態を更新する。また、必要であれば B.C. を反映させる。
6. 更新された物理量を基に、圧力と速度、並びにそれらの偏微分値も更新する。
7. 1.-6. を繰り返す。

以下は python コード例である。すぐに気付くが、FDM と 1 次元 CIP のコードに非常に似ている。

```
import numpy as np
import math

Nx, Ny = (101, 101)
dx = 0.01
rho = 1.2
k = 1.4e+5
c = math.sqrt(k/rho)
Z = math.sqrt(rho*k)
r = 0.5
cfl = 0.5
dt = cfl*dx/c

p = np.zeros((Nx, Ny))
p[45:55,45:55] = 1.
dxp = np.zeros((Nx, Ny))
dyp = np.zeros((Nx, Ny))
u = np.zeros((Nx, Ny))
dxu = np.zeros((Nx, Ny))
dyu = np.zeros((Nx, Ny))
v = np.zeros((Nx, Ny))
dxv = np.zeros((Nx, Ny))
dyv = np.zeros((Nx, Ny))

def getParam3(h_i, h_iup, dh_i, dh_iup, D):
        c0 = h_i
        c1 = dh_i
        c2 = 3.*(h_iup - h_i)/(D**2) - (2.*dh_i + dh_iup)/D
        c3 = (dh_i + dh_iup)/(D**2) + 2.*(h_i - h_iup)/(D**3)

        return c0, c1, c2, c3

def getParam1(h_i, h_iup, D):
```

```
        c0 = h_i
        c1 = (h_iup-h_i)/D

        return c0, c1

time_step = 200
for itr in range(time_step):
        fx = p + Z*u
        dxfx = dxp + Z*dxu
        dyfx = dyp + Z*dyu

        gx = p - Z*u
        dxgx = dxp - Z*dxu
        dygx = dyp - Z*dyu

        cf03, cf13, cf23, cf33 = getParam3(fx[1:,:], fx[:-1,:], dxfx[1:,:], dxfx[:-1,:], -dx)
        cg03, cg13, cg23, cg33 = getParam3(gx[:-1,:], gx[1:,:], dxgx[:-1,:], dxgx[1:,:], dx)

        cf01, cf11 = getParam1(dyfx[1:,:], dyfx[:-1,:], -dx)
        cg01, cg11 = getParam1(dygx[:-1,:], dygx[1:,:], dx)

        fx[1:,:] = cf33*((-c*dt)**3) + cf23*((-c*dt)**2) + cf13*(-c*dt) + cf03
        gx[:-1,:] = cg33*((c*dt)**3) + cg23*((c*dt)**2) + cg13*(c*dt) + cg03
        dxfx[1:,:] = 3.*cf33*((-c*dt)**2) + 2.*cf23*(-c*dt) + cf13
        dxgx[:-1,:] = 3.*cg33*((c*dt)**2) + 2.*cg23*(c*dt) + cg13
        dyfx[1:,:] = cf11*(-c*dt) + cf01
        dygx[:-1,:] = cg11*(c*dt) + cg01

        fx[0,:] = r*gx[0,:]
        gx[-1,:] = r*fx[-1,:]
        dxfx[0,:] = -r*dxgx[0,:]
        dxgx[-1,:] = -r*dxfx[-1,:]
        dyfx[0,:] = r*dygx[0,:]
        dygx[0,:] = r*dyfx[-1,:]

        p = (fx + gx)/2.
        dxp = (dxfx + dxgx)/2.
        dyp = (dyfx + dygx)/2.
        u = (fx - gx)/(2.*Z)
        dxu = (dxfx - dxgx)/(2.*Z)
        dyu = (dyfx - dygx)/(2.*Z)

        fy = p + Z*v
        dyfy = dyp + Z*dyv
        dxfy = dxp + Z*dxv

        gy = p - Z*v
        dygy = dyp - Z*dyv
        dxgy = dxp - Z*dxv

        cf03, cf13, cf23, cf33 = getParam3(fy[:,1:], fy[:,:-1], dyfy[:,1:], dyfy[:,:-1], -dx)
        cg03, cg13, cg23, cg33 = getParam3(gy[:,:-1], gy[:,1:], dygy[:,:-1], dygy[:,1:], dx)

        cf01, cf11 = getParam1(dxfy[:,1:], dxfy[:,:-1], -dx)
        cg01, cg11 = getParam1(dxgy[:,:-1], dxgy[:,1:], dx)

        fy[:,1:] = cf33*((-c*dt)**3) + cf23*((-c*dt)**2) + cf13*(-c*dt) + cf03
        gy[:,:-1] = cg33*((c*dt)**3) + cg23*((c*dt)**2) + cg13*(c*dt) + cg03
        dyfy[:,1:] = 3.*cf33*((-c*dt)**2) + 2.*cf23*(-c*dt) + cf13
        dygy[:,:-1] = 3.*cg33*((c*dt)**2) + 2.*cg23*(c*dt) + cg13
        dxfy[:,1:] = cf11*(-c*dt) + cf01
        dxgy[:,:-1] = cg11*(c*dt) + cg01

        fy[:,0] = r*gy[:,0]
```

```
        gy[:,−1] = r*fy[:,−1]
        dyfy[:,0] = −r*dygy[:,0]
        dygy[:,−1] = −r*dyfy[:,−1]
        dxfy[:,0] = r*dxgy[:,0]
        dxgy[:,0] = r*dxfy[:,−1]

        p = (fy + gy)/2.
        dyp = (dyfy + dygy)/2.
        dxp = (dxfy + dxgy)/2.
        v = (fy − gy)/(2.*Z)
        dyv = (dyfy − dygy)/(2.*Z)
        dxv = (dxfy − dxgy)/(2.*Z)
```

5.2 C 型 CIP

M 型 CIP の 1 次多項式部分を CIP らしく変えたのが C 型 CIP である。$\partial_y h_x$ に関する移流方程式

$$\partial_t(\partial_y h_x) \pm c\partial_x(\partial_y h_x) = 0$$

だが、$\partial_y h_x$ の x の偏微分 $\partial_{xy} h_x$ も移流方程式

$$\partial_t(\partial_{xy} h_x) \pm c\partial_x(\partial_{xy} h_x) = 0$$

を満たすことに気付くだろう。そこで、C 型 CIP は $\partial_{xy} h_x$ も陽に解く。そうすることで $\partial_y h_x$ に関する多項式に、4 つの制約を用意することができるため、晴れて 3 次多項式を採用できる訳である。そのためには $\partial_{xy} p$ などの偏微分が所与でなければならない。また、M 型 CIP のときと比べて必要計算メモリ量が増える。しかしながら、C 型にすることで計算結果の正確さが向上することは、図 2 と 6 より容易に想像できる。多項式の係数はこれまで通りであり、アルゴリズム上の難しさは最早見受けられない。以下は python コード例である (M 型 CIP のときからほとんど変えていない)。

```
import numpy as np
import math

Nx, Ny = (101, 101)
dx = 0.01
rho = 1.2
k = 1.4e+5
c = math.sqrt(k/rho)
Z = math.sqrt(rho*k)
r = 0.5
cfl = 0.5
dt = cfl*dx/c

p = np.zeros((Nx, Ny))
p[45:55,45:55] = 1.
dxp = np.zeros((Nx, Ny))
dyp = np.zeros((Nx, Ny))
dxyp = np.zeros((Nx, Ny))
u = np.zeros((Nx, Ny))
dxu = np.zeros((Nx, Ny))
dyu = np.zeros((Nx, Ny))
dxyu = np.zeros((Nx, Ny))
v = np.zeros((Nx, Ny))
dxv = np.zeros((Nx, Ny))
dyv = np.zeros((Nx, Ny))
dxyv = np.zeros((Nx, Ny))

def getParam3(h_i, h_iup, dh_i, dh_iup, D):
        c0 = h_i
        c1 = dh_i
        c2 = 3.*(h_iup − h_i)/(D**2) − (2.*dh_i + dh_iup)/D
        c3 = (dh_i + dh_iup)/(D**2) + 2.*(h_i − h_iup)/(D**3)

        return c0, c1, c2, c3

time_step = 200
```

```
for itr in range(time_step):
        fx = p + Z*u
        dxfx = dxp + Z*dxu
        dyfx = dyp + Z*dyu
        dxyfx = dxyp + Z*dxyu


        gx = p - Z*u
        dxgx = dxp - Z*dxu
        dygx = dyp - Z*dyu
        dxygx = dxyp - Z*dxyu

        cf03, cf13, cf23, cf33 = getParam3(fx[1:,:], fx[:-1,:], dxfx[1:,:], dxfx[:-1,:], -dx)
        cg03, cg13, cg23, cg33 = getParam3(gx[:-1,:], gx[1:,:], dxgx[:-1,:], dxgx[1:,:], dx)

        cf01, cf11, cf21, cf31 = getParam3(dyfx[1:,:], dyfx[:-1,:], dxyfx[1:,:], dxyfx[:-1,:], -dx)
        cg01, cg11, cg21, cg31 = getParam3(dygx[:-1,:], dygx[1:,:], dxygx[:-1,:], dxygx[1:,:], dx)

        fx[1:,:] = cf33*((-c*dt)**3) + cf23*((-c*dt)**2) + cf13*(-c*dt) + cf03
        gx[:-1,:] = cg33*((c*dt)**3) + cg23*((c*dt)**2) + cg13*(c*dt) + cg03
        dxfx[1:,:] = 3.*cf33*((-c*dt)**2) + 2.*cf23*(-c*dt) + cf13
        dxgx[:-1,:] = 3.*cg33*((c*dt)**2) + 2.*cg23*(c*dt) + cg13
        dyfx[1:,:] = cf31*((-c*dt)**3) + cf21*((-c*dt)**2) + cf11*(-c*dt) + cf01
        dygx[:-1,:] = cg31*((c*dt)**3) + cg21*((c*dt)**2) + cg11*(c*dt) + cg01
        dxyfx[1:,:] = 3.*cf31*((-c*dt)**2) + 2.*cf21*(-c*dt) + cf11
        dxygx[:-1,:] = 3.*cg31*((c*dt)**2) + 2.*cg21*(c*dt) + cg11

        fx[0,:] = r*gx[0,:]
        gx[-1,:] = r*fx[-1,:]
        dxfx[0,:] = -r*dxgx[0,:]
        dxgx[-1,:] = -r*dxfx[-1,:]
        dyfx[0,:] = r*dygx[0,:]
        dygx[0,:] = r*dyfx[-1,:]
        dxyfx[0,:] = -r*dxygx[0,:]
        dxygx[0,:] = -r*dxyfx[-1,:]

        p = (fx + gx)/2.
        dxp = (dxfx + dxgx)/2.
        dyp = (dyfx + dygx)/2.
        dxyp = (dxyfx + dxygx)/2.
        u = (fx - gx)/(2.*Z)
        dxu = (dxfx - dxgx)/(2.*Z)
        dyu = (dyfx - dygx)/(2.*Z)
        dxyu = (dxyfx - dxygx)/(2.*Z)

        fy = p + Z*v
        dyfy = dyp + Z*dyv
        dxfy = dxp + Z*dxv
        dxyfy = dxyp + Z*dxyv

        gy = p - Z*v
        dygy = dyp - Z*dyv
        dxgy = dxp - Z*dxv
        dxygy = dxyp - Z*dxyv

        cf03, cf13, cf23, cf33 = getParam3(fy[:,1:], fy[:,:-1], dyfy[:,1:], dyfy[:,:-1], -dx)
        cg03, cg13, cg23, cg33 = getParam3(gy[:,:-1], gy[:,1:], dygy[:,:-1], dygy[:,1:], dx)

        cf01, cf11, cf21, cf31 = getParam3(dxfy[:,1:], dxfy[:,:-1], dxyfy[:,1:], dxyfy[:,:-1], -dx)
        cg01, cg11, cg21, cg31 = getParam3(dxgy[:,:-1], dxgy[:,1:], dxygy[:,:-1], dxygy[:,1:], dx)

        fy[:,1:] = cf33*((-c*dt)**3) + cf23*((-c*dt)**2) + cf13*(-c*dt) + cf03
        gy[:,:-1] = cg33*((c*dt)**3) + cg23*((c*dt)**2) + cg13*(c*dt) + cg03
        dyfy[:,1:] = 3.*cf33*((-c*dt)**2) + 2.*cf23*(-c*dt) + cf13
```

```
dygy[:,:−1] = 3.*cg33*((c*dt)**2) + 2.*cg23*(c*dt) + cg13
dyfx[:,1:] = cf31*((−c*dt)**3) + cf21*((−c*dt)**2) + cf11*(−c*dt) + cf01
dygx[:,:−1] = cg31*((c*dt)**3) + cg21*((c*dt)**2) + cg11*(c*dt) + cg01
dxyfx[:,1:] = 3.*cf31*((−c*dt)**2) + 2.*cf21*(−c*dt) + cf11
dxygx[:,:−1] = 3.*cg31*((c*dt)**2) + 2.*cg21*(c*dt) + cg11

fy[:,0] = r*gy[:,0]
gy[:,−1] = r*fy[:,−1]
dyfy[:,0] = −r*dygy[:,0]
dygy[:,−1] = −r*dyfy[:,−1]
dxfy[:,0] = r*dxgy[:,0]
dxgy[:,0] = r*dxfy[:,−1]
dxyfy[:,0] = −r*dxygy[:,0]
dxygy[:,0] = −r*dxyfy[:,−1]

p = (fy + gy)/2.
dyp = (dyfy + dygy)/2.
dxp = (dxfy + dxgy)/2.
dxyp = (dxyfy + dxygy)/2.
v = (fy − gy)/(2.*Z)
dyv = (dyfy − dygy)/(2.*Z)
dxv = (dxfy − dxgy)/(2.*Z)
dxyv = (dxyfy − dxygy)/(2.*Z)
```

5.3 CIP-CSL2

3 章で述べた通り、CIP-CSL2 は格子内の積分値と離散点の物理量を利用する。ただし、積分値の定義位置は図 10 の $H(i,j)$ であるため、3 章と同様の処理を施すなら、図 10 の線積分が欲しいところである。また、4.2 節で議論した通り、この線積分は移流方程式を満たす。そこで、CIP-CSL2 は格子点の物理量と線積分を陽に更新させる。

本節では線積分値を h で表し、格子点位置の物理量を h の偏微分値として扱う。これは、物理量 p、u、v、$\int_X pdx$、$\int_Y pdy$、$\int_X udx$、$\int_Y udy$、$\int_X vdx$、$\int_Y vdy$、$\int_{XY} pdxy$、$\int_{XY} udxy$、$\int_{XY} vdxy$ を扱うことに相当する (最後の 3 つは更新された H を反映させるため)。フラクショナルステップ法のうち x 方向に移流させるとき、v に関する物理量は変化しない。従ってこの間は p、u、$\int_X pdx$、$\int_Y pdy$、$\int_X udx$、$\int_Y udy$、$\int_{XY} pdxy$、$\int_{XY} udxy$ のみ考えればよい。これらは $\partial_x h_x$ と h_x、h_y、並びに H と関連する (図 10)。$\partial_x h_x$ と h_x の更新は 1 次元 CIP-CSL2 と同様である。また、h_y と H は同様に 1 次元 CIP-CSL2 の手順で更新される。

具体的な移流のさせ方は以下の通りである。まず、$\partial_x h_x$ は格子点の物理量と線積分値 (h_x) が与えられているため、2 次の多項式で補間できる。それを本節では

$$\partial_x h_x^i(x,y,t) = c_2(x-x_i)^2 + c_1(x-x_i) + c_0$$

のように定義する。同様に h_y もセル界面の線積分値と積分値 H が与えられているため、2 次の多項式で補間できる。それを本節では

$$h_y^i(x,y,t) = C_2(x-x_i)^2 + C_1(x-x_i) + C_0$$

のように定義する。ただし両式で x_i の値がことなることに注意してほしい (図 10 の通り、前者は離散点上、後者は格子界面の中心に位置する)。更新したあとは p や u、$\int_X pdx$、$\int_X udx$、$\int_Y pdy$、$\int_Y udy$、$\int_{XY} pdxy$、$\int_{XY} udxy$ に反映させる。つまり、

$$\begin{aligned}
p &\leftarrow (\partial_x f_x + \partial_x g_x)/2 \\
u &\leftarrow (\partial_x f_x - \partial_x g_x)/(2Z) \\
\textstyle\int_X pdx &\leftarrow (f_x + g_x)/2 \\
\textstyle\int_X udx &\leftarrow (f_x - g_x)/(2Z) \\
\textstyle\int_Y pdx &\leftarrow (f_y + g_y)/2 \\
\textstyle\int_Y udx &\leftarrow (f_y - g_y)/(2Z) \\
\textstyle\int_{XY} pdxy &\leftarrow (F+G)/2 \\
\textstyle\int_{XY} udxy &\leftarrow (F-G)/(2Z)
\end{aligned}$$

に従い更新すればよい。同様の処理を y 方向でも行う。

5.3.1 アルゴリズムと python コード例

CIP-CSL2 による 2 次元移流方程式の解法

1. 所与の圧力と速度より、$\partial_x h_x$ と h_x、並びに h_y と H を計算する。
2. 空間補間し、状態を更新する。必要であれば B.C. を反映させる。
3. 更新された物理量を基に、圧力と速度、並びにそれらの積分値も更新する。
4. 所与の圧力と速度より、$\partial_y h_y$ と h_y、並びに h_x と H を計算する。
5. 空間補間し、状態を更新する。必要であれば B.C. を反映させる。
6. 更新された物理量を基に、圧力と速度、並びにそれらの積分値も更新する。
7. 1.-6. を繰り返す。

```
import numpy as np
import math
import sys

Nx, Ny = (101, 101)
dx = 0.01
rho = 1.2
k = 1.4e+5
c = math.sqrt(k/rho)
Z = math.sqrt(rho*k)
r = 1.
cfl = 0.5
dt = cfl*dx/c

x, y = np.indices((Nx, Ny))*dx
p = np.zeros((Nx, Ny)); p[45:55,45:55] = 1.
u = np.zeros((Nx, Ny))
v = np.zeros((Nx, Ny))

Ixp = (p[1:,:] + p[:-1,:])*dx/2.
Ixu = (u[1:,:] + u[:-1,:])*dx/2.
Ixv = (v[1:,:] + v[:-1,:])*dx/2.
Iyp = (p[:,1:] + p[:,:-1])*dx/2.
Iyu = (u[:,1:] + u[:,:-1])*dx/2.
Iyv = (v[:,1:] + v[:,:-1])*dx/2.
Ip = (p[:-1,:-1] + p[1:,:-1] + p[:-1,1:] + p[1:,1:])*(dx**2)/4.
Iu = (u[:-1,:-1] + u[1:,:-1] + u[:-1,1:] + u[1:,1:])*(dx**2)/4.
Iv = (v[:-1,:-1] + v[1:,:-1] + v[:-1,1:] + v[1:,1:])*(dx**2)/4.


def getParam(hi, hiup, H, s, D):
        c0 = hi
        c1 = -6.*s*H/(D**2) -2.*(hiup + 2.*hi)/D
        c2 = 6.*s*H/(D**3) + 3.*(hiup + hi)/(D**2)

        return c0, c1, c2

def getDeltaH(c0, c1, c2, s):
        epsilon = -s*c*dt
        return -sgn*(c2*(epsilon**3)/3. + c1*(epsilon**2)/2. + c0*epsilon)

time_step = 200
for itr in range(time_step):
        dxfx = p + Z*u; dxgx = p - Z*u
        fx = Ixp + Z*Ixu; gx = Ixp - Z*Ixu
        fy = Iyp + Z*Iyu; gy = Iyp - Z*Iyu
        F = Ip + Z*Iu; G = Ip - Z*Iu

        cf0, cf1, cf2 = getParam(dxfx[1:,:], dxfx[:-1,:], fx, 1., -dx)
        cg0, cg1, cg2 = getParam(dxgx[:-1,:], dxgx[1:,:], gx, -1., dx)
        Cf0, Cf1, Cf2 = getParam(fy[1:,:], fy[:-1,:], F, 1., -dx)
```

```
Cg0, Cg1, Cg2 = getParam(gy[:−1,:], gy[1:,:], G, −1., dx)

fx[1:,:] += getDeltaH(cf0[:−1,:], cf1[:−1,:], cf2[:−1,:], 1.) \
− getDeltaH(cf0[1:,:], cf1[1:,:], cf2[1:,:], 1.)
gx[:−1,:] += getDeltaH(cg0[1:,:], cg1[1:,:], cg2[1:,:], −1.) − \
getDeltaH(cg0[:−1,:], cg1[:−1,:], cg2[:−1,:], −1.)
fx[0,:] += r*getDeltaH(cg0[0,:], cg1[0,:], cg2[0,:], −1.) − \
getDeltaH(cf0[0,:], cf1[0,:], cf2[0,:], 1.)
gx[−1,:] += r*getDeltaH(cf0[−1,:], cf1[−1,:], cf2[−1,:], 1.) − \
getDeltaH(cg0[−1,:], cg1[−1,:], cg2[−1,:], −1.)

F[1:,:] += getDeltaH(Cf0[:−1,:], Cf1[:−1,:], Cf2[:−1,:], 1.) − \
getDeltaH(Cf0[1:,:], Cf1[1:,:], Cf2[1:,:], 1.)
G[:−1,:] += getDeltaH(Cg0[1:,:], Cg1[1:,:], Cg2[1:,:], −1.) − \
getDeltaH(Cg0[:−1,:], Cg1[:−1,:], Cg2[:−1,:], −1.)
F[0,:] += r*getDeltaH(Cg0[0,:], Cg1[0,:], Cg2[0,:], −1.) − \
getDeltaH(Cf0[0,:], Cf1[0,:], Cf2[0,:], 1.)
G[−1,:] += r*getDeltaH(Cf0[−1,:], Cf1[−1,:], Cf2[−1,:], 1.) − \
getDeltaH(Cg0[−1,:], Cg1[−1,:], Cg2[−1,:], −1.)

dxfx[1:,:] = cf2*((−c*dt)**2) + cf1*(−c*dt) + cf0
dxgx[:−1,:] = cg2*((c*dt)**2) + cg1*(c*dt) + cg0
fy[1:,:] = Cf2*((−c*dt)**2) + Cf1*(−c*dt) + Cf0
gy[:−1,:] = Cg2*((c*dt)**2) + Cg1*(c*dt) + Cg0

dxfx[0,:] = r*dxgx[0,:]
dxgx[−1,:] = r*dxfx[−1,:]
fy[0,:] = r*gy[0,:]
gy[−1,:] = r*fy[−1,:]

p = (dxfx + dxgx)/2.
u = (dxfx − dxgx)/(2.*Z)
Ixp = (fx + gx)/2.
Ixu = (fx − gx)/(2.*Z)
Iyp = (fy + gy)/2.
Iyu = (fy − gy)/(2.*Z)
Ip = (F + G)/2.
Iu = (F − G)/(2.*Z)


dyfy = p + Z*v; dygy = p − Z*v
fy = Iyp + Z*Iyv; gy = Iyp − Z*Iyv
fx = Ixp + Z*Ixv; gx = Ixp − Z*Ixv
F = Ip + Z*Iv; G = Ip − Z*Iv

cf0, cf1, cf2 = getParam(dyfy[:,1:], dyfy[:,:−1], fy, 1., −dx)
cg0, cg1, cg2 = getParam(dygy[:,:−1], dygy[:,1:], gy, −1., dx)
Cf0, Cf1, Cf2 = getParam(fx[:,1:], fx[:,:−1], F, 1., −dx)
Cg0, Cg1, Cg2 = getParam(gx[:,:−1], gx[:,1:], G, −1., dx)

fy[:,1:] += getDeltaH(cf0[:,:−1], cf1[:,:−1], cf2[:,:−1], 1.) − \
getDeltaH(cf0[:,1:], cf1[:,1:], cf2[:,1:], 1.)
gy[:,:−1] += getDeltaH(cg0[:,1:], cg1[:,1:], cg2[:,1:], −1.) − \
getDeltaH(cg0[:,:−1], cg1[:,:−1], cg2[:,:−1], −1.)
fy[:,0] += r*getDeltaH(cg0[:,0], cg1[:,0], cg2[:,0], −1.) − \
getDeltaH(cf0[:,0], cf1[:,0], cf2[:,0], 1.)
gy[:,−1] += r*getDeltaH(cf0[:,−1], cf1[:,−1], cf2[:,−1], 1.) − \
getDeltaH(cg0[:,−1], cg1[:,−1], cg2[:,−1], −1.)

F[:,1:] += getDeltaH(Cf0[:,:−1], Cf1[:,:−1], Cf2[:,:−1], 1.) − \
getDeltaH(Cf0[:,1:], Cf1[:,1:], Cf2[:,1:], 1.)
G[:,:−1] += getDeltaH(Cg0[:,1:], Cg1[:,1:], Cg2[:,1:], −1.) − \
getDeltaH(Cg0[:,:−1], Cg1[:,:−1], Cg2[:,:−1], −1.)
F[:,0] += r*getDeltaH(Cg0[:,0], Cg1[:,0], Cg2[:,0], −1.) − \
```

```
        getDeltaH(Cf0[:,0], Cf1[:,0], Cf2[:,0], 1.)
        G[:,-1] += r*getDeltaH(Cf0[:,-1], Cf1[:,-1], Cf2[:,-1], 1.) - \
        getDeltaH(Cg0[:,-1], Cg1[:,-1], Cg2[:,-1], -1.)

        dyfy[:,1:] = cf2*((-c*dt)**2) + cf1*(-c*dt) + cf0
        dygy[:,:-1] = cg2*((c*dt)**2) + cg1*(c*dt) + cg0
        fx[:,1:] = Cf2*((-c*dt)**2) + Cf1*(-c*dt) + Cf0
        gx[:,:-1] = Cg2*((c*dt)**2) + Cg1*(c*dt) + Cg0

        dyfy[:,0] = r*dygy[:,0]
        dygy[:,-1] = r*dyfy[:,-1]
        fx[:,0] = r*gx[:,0]
        gx[:,-1] = r*fx[:,-1]

        p = (dyfy + dygy)/2.
        v = (dyfy - dygy)/(2.*Z)
        Iyp = (fy + gy)/2.
        Iyv = (fy - gy)/(2.*Z)
        Ixp = (fx + gx)/2.
        Ixv = (fx - gx)/(2.*Z)
        Ip = (F + G)/2.
        Iv = (F - G)/(2.*Z)
```

5.4 CIP-CSL4

本節では多次元の CIP-CSL4 を議論する。なお、多次元の CIP-CSL4 に関する文献を見つけられなかったため、本節の内容は私個人で考えたものである。それゆえ誤りがあるかもしれないことを留意頂きたい。

3 章で紹介した通り、CIP-CSL4 は物理量とその微分値、並びに積分値を利用する。積分値は格子点間の線積分値とセルの積分値を利用する。こうすることで線積分値に関しても移流方程式を用意できることは前節で紹介した通りである。

一方で微分値の与え方はいくつかあると思われる。まず考えられるのは格子点にのみ微分値を与える方法であろう。この場合、格子点間上の移流は 1 次元 CIP-CSL4 と同様の手続きで進められる。一方で線積分に関する移流では微分値を利用できないので、1 次元 CIP-CSL2 に従って移流させる。本資料ではこの方法を採用する。アルゴリズムは先述の 1 次元 CIP-CSL4 と CIP-CSL2 と同様である。以下の python コード例は M 型による CIP-CSL4 である。

```
import numpy as np
import math
import sys

Nx, Ny = (101, 101)
dx = 0.01
rho = 1.2
k = 1.4e+5
c = math.sqrt(k/rho)
Z = math.sqrt(rho*k)
r = 1.
cfl = 0.5
dt = cfl*dx/c

x, y = np.indices((Nx, Ny))*dx
p = np.zeros((Nx, Ny)); p[45:55,45:55] = 1.
u = np.zeros((Nx, Ny))
v = np.zeros((Nx, Ny))
dxp = np.zeros((Nx, Ny))
dxu = np.zeros((Nx, Ny))
dxv = np.zeros((Nx, Ny))
dyp = np.zeros((Nx, Ny))
dyu = np.zeros((Nx, Ny))
dyv = np.zeros((Nx, Ny))
Ixp = (p[1:,:] + p[:-1,:])*dx/2.
Ixu = (u[1:,:] + u[:-1,:])*dx/2.
Ixv = (v[1:,:] + v[:-1,:])*dx/2.
Iyp = (p[:,1:] + p[:,:-1])*dx/2.
Iyu = (u[:,1:] + u[:,:-1])*dx/2.
Iyv = (v[:,1:] + v[:,:-1])*dx/2.
```

```
Ip = (p[:-1,:-1] + p[1:,:-1] + p[:-1,1:] + p[1:,1:])*(dx**2)/4.
Iu = (u[:-1,:-1] + u[1:,:-1] + u[:-1,1:] + u[1:,1:])*(dx**2)/4.
Iv = (v[:-1,:-1] + v[1:,:-1] + v[:-1,1:] + v[1:,1:])*(dx**2)/4.


def getParamCSL4(hi, hiup, dhi, dhiup, H, s, D):
        c0 = hi
        c1 = dhi
        c2 = -30.*s*H/(D**3) - 6.*(3.*hi+2.*hiup)/(D**2) - 3.*(3.*dhi - dhiup)/(2.*D)
        c3 = -2.*c2/D + 4.*(hiup - hi)/(D**3) - (3.*dhi + dhiup)/(D**2)
        c4 = -c3/D - c2/(D**2) + (hiup - hi)/(D**4) - dhi/(D**3)

        return c0, c1, c2, c3, c4

def getParamFDM(hi, hiup, D):
        c0 = hi
        c1 = (hiup - hi)/D

        return c0, c1

def getParamCSL2(hi, hiup, H, s, D):
        c0 = hi
        c1 = -6.*s*H/(D**2) -2.*(hiup + 2.*hi)/D
        c2 = 6.*s*H/(D**3) + 3.*(hiup + hi)/(D**2)

        return c0, c1, c2

def getDeltaHCSL4(c0, c1, c2, c3, c4, s):
        epsilon = -s*c*dt
        return -s*(c4*(epsilon**5)/5. + c3*(epsilon**4)/4. \
        +c2*(epsilon**3)/3. +c1*(epsilon**2)/2. + c0*epsilon)

def getDeltaHCSL2(c0, c1, c2, s):
        epsilon = -s*c*dt
        return -s*(c2*(epsilon**3)/3. + c1*(epsilon**2)/2. + c0*epsilon)

time_step = 200
for itr in range(time_step):
        fx = p + Z*u; gx = p - Z*u
        dxfx = dxp + Z*dxu; dxgx = dxp - Z*dxu
        Ixfx = Ixp + Z*Ixu; Ixgx = Ixp - Z*Ixu

        dyfx = dyp + Z*dyu; dygx = dyp - Z*dyu

        Iyfx = Iyp + Z*Iyu; Iygx = Iyp - Z*Iyu
        If = Ip + Z*Iu; Ig = Ip - Z*Iu

        cf0, cf1, cf2, cf3, cf4 = \
        getParamCSL4(fx[1:,:], fx[:-1,:], dxfx[1:,:], dxfx[:-1,:], Ixfx, 1., -dx)
        cg0, cg1, cg2, cg3, cg4 = \
        getParamCSL4(gx[:-1,:], gx[1:,:], dxgx[:-1,:], dxgx[1:,:], Ixgx, -1., dx)
        mf0, mf1 = getParamFDM(dyfx[1:,:], dyfx[:-1,:], -dx)
        mg0, mg1 = getParamFDM(dygx[:-1,:], dygx[1:,:], dx)

        Cf0, Cf1, Cf2 = getParamCSL2(Iyfx[1:,:], Iyfx[:-1,:], If, 1., -dx)
        Cg0, Cg1, Cg2 = getParamCSL2(Iygx[:-1,:], Iygx[1:,:], Ig, -1., dx)

        Ixfx[1:,:] += getDeltaHCSL4(cf0[:-1,:], cf1[:-1,:], cf2[:-1,:], cf3[:-1,:], cf4[:-1,:], 1.)\
         - getDeltaHCSL4(cf0[1:,:], cf1[1:,:], cf2[1:,:], cf3[1:,:], cf4[1:,:], 1.)
        Ixgx[:-1,:] += getDeltaHCSL4(cg0[1:,:], cg1[1:,:], cg2[1:,:], cg3[1:,:], cg4[1:,:], -1.)\
         - getDeltaHCSL4(cg0[:-1,:], cg1[:-1,:], cg2[:-1,:], cg3[:-1,:], cg4[:-1,:], -1.)
        Ixfx[0,:] += r*getDeltaHCSL4(cg0[0,:], cg1[0,:], cg2[0,:], cg3[0,:], cg4[0,:], -1.)\
         - getDeltaHCSL4(cf0[0,:], cf1[0,:], cf2[0,:], cf3[0,:], cf4[0,:], 1.)
        Ixgx[-1,:] += r*getDeltaHCSL4(cf0[-1,:], cf1[-1,:], cf2[-1,:], cf3[-1,:], cf4[-1,:], 1.)\
```

```
 - getDeltaHCSL4(cg0[-1,:], cg1[-1,:], cg2[-1,:], cg3[-1,:], cg4[-1,:], -1.)

If[1:,:] += getDeltaHCSL2(Cf0[:-1,:], Cf1[:-1,:], Cf2[:-1,:], 1.)\
 - getDeltaHCSL2(Cf0[1:,:], Cf1[1:,:], Cf2[1:,:], 1.)
Ig[:-1,:] += getDeltaHCSL2(Cg0[1:,:], Cg1[1:,:], Cg2[1:,:], -1.)\
 - getDeltaHCSL2(Cg0[:-1,:], Cg1[:-1,:], Cg2[:-1,:], -1.)
If[0,:] += r*getDeltaHCSL2(Cg0[0,:], Cg1[0,:], Cg2[0,:], -1.)\
 - getDeltaHCSL2(Cf0[0,:], Cf1[0,:], Cf2[0,:], 1.)
Ig[-1,:] += r*getDeltaHCSL2(Cf0[-1,:], Cf1[-1,:], Cf2[-1,:], 1.)\
 - getDeltaHCSL2(Cg0[-1,:], Cg1[-1,:], Cg2[-1,:], -1.)

fx[1:,:] = cf4*((-c*dt)**4) + cf3*((-c*dt)**3) + cf2*((-c*dt)**2) + cf1*(-c*dt) + cf0
gx[:-1,:] = cg4*((c*dt)**4) + cg3*((c*dt)**3) + cg2*((c*dt)**2) + cg1*(c*dt) + cg0
dxfx[1:,:] = 4.*cf4*((-c*dt)**3) + 3.*cf3*((-c*dt)**2) + 2.*cf2*(-c*dt) + cf1
dxgx[:-1,:] = 4.*cg4*((c*dt)**3) + 3.*cg3*((c*dt)**2) + 2.*cg2*(c*dt) + cg1

dyfx[1:,:] = mf1*(-c*dt) + mf0
dygx[:-1,:] = mg1*(c*dt) + mg0

Iyfx[1:,:] = Cf2*((-c*dt)**2) + Cf1*(-c*dt) + Cf0
Iygx[:-1,:] = Cg2*((c*dt)**2) + Cg1*(c*dt) + Cg0

fx[0,:] = r*gx[0,:]; gx[-1,:] = r*fx[-1,:]
dxfx[0,:] = -r*dxgx[0,:]; dxgx[-1,:] = -r*dxfx[-1,:]
dyfx[0,:] = r*dygx[0,:]; dygx[-1,:] = r*dyfx[-1,:]
Iyfx[0,:] = r*Iygx[0,:]; Iygx[-1,:] = r*Iyfx[-1,:]

p = (fx + gx)/2.
u = (fx - gx)/(2.*Z)
dxp = (dxfx + dxgx)/2.
dxu = (dxfx - dxgx)/(2.*Z)
Ixp = (Ixfx + Ixgx)/2.
Ixu = (Ixfx - Ixgx)/(2.*Z)
dyp = (dyfx + dygx)/2.
dyu = (dyfx - dygx)/(2.*Z)
Iyp = (Iyfx + Iygx)/2.
Iyu = (Iyfx - Iygx)/(2.*Z)
Ip = (If + Ig)/2.
Iu = (If - Ig)/(2.*Z)

fy = p + Z*v; gy = p - Z*v
dyfy = dyp + Z*dyv; dygy = dyp - Z*dyv
Iyfy = Iyp + Z*Iyv; Iygy = Iyp - Z*Iyv

dxfy = dxp + Z*dxv; dxgy = dxp - Z*dxv

Ixfy = Ixp + Z*Ixv; Ixgy = Ixp - Z*Ixv
If = Ip + Z*Iv; Ig = Ip - Z*Iv

cf0, cf1, cf2, cf3, cf4 = \
getParamCSL4(fy[:,1:], fy[:,:-1], dyfy[:,1:], dyfy[:,:-1], Iyfy, 1., -dx)
cg0, cg1, cg2, cg3, cg4 = \
getParamCSL4(gy[:,:-1], gy[:,1:], dygy[:,:-1], dygy[:,1:], Iygy, -1., dx)
mf0, mf1 = getParamFDM(dxfy[:,1:], dxfy[:,:-1], -dx)
mg0, mg1 = getParamFDM(dxgy[:,:-1], dxgy[:,1:], dx)

Cf0, Cf1, Cf2 = getParamCSL2(Ixfy[:,1:], Ixfy[:,:-1], If, 1., -dx)
Cg0, Cg1, Cg2 = getParamCSL2(Ixgy[:,:-1], Ixgy[:,1:], Ig, -1., dx)

Iyfy[:,1:] += getDeltaHCSL4(cf0[:,:-1], cf1[:,:-1], cf2[:,:-1], cf3[:,:-1], cf4[:,:-1], 1.)\
 - getDeltaHCSL4(cf0[:,1:], cf1[:,1:], cf2[:,1:], cf3[:,1:], cf4[:,1:], 1.)
Iygy[:,:-1] += getDeltaHCSL4(cg0[:,1:], cg1[:,1:], cg2[:,1:], cg3[:,1:], cg4[:,1:], -1.)\
 - getDeltaHCSL4(cg0[:,:-1], cg1[:,:-1], cg2[:,:-1], cg3[:,:-1], cg4[:,:-1], -1.)
Iyfy[:,0] += r*getDeltaHCSL4(cg0[:,0], cg1[:,0], cg2[:,0], cg3[:,0], cg4[:,0], -1.)\
```

```
 - getDeltaHCSL4(cf0[:,0], cf1[:,0], cf2[:,0], cf3[:,0], cf4[:,0], 1.)
Iygy[:,-1] += r*getDeltaHCSL4(cf0[:,-1], cf1[:,-1], cf2[:,-1], cf3[:,-1], cf4[:,-1], 1.)\
 - getDeltaHCSL4(cg0[:,-1], cg1[:,-1], cg2[:,-1], cg3[:,-1], cg4[:,-1], -1.)

If[:,1:] += getDeltaHCSL2(Cf0[:,:-1], Cf1[:,:-1], Cf2[:,:-1], 1.)\
 - getDeltaHCSL2(Cf0[:,1:], Cf1[:,1:], Cf2[:,1:], 1.)
Ig[:,:-1] += getDeltaHCSL2(Cg0[:,1:], Cg1[:,1:], Cg2[:,1:], -1.)\
 - getDeltaHCSL2(Cg0[:,:-1], Cg1[:,:-1], Cg2[:,:-1], -1.)
If[:,0] += r*getDeltaHCSL2(Cg0[:,0], Cg1[:,0], Cg2[:,0], -1.)\
 - getDeltaHCSL2(Cf0[:,0], Cf1[:,0], Cf2[:,0], 1.)
Ig[:,-1] += r*getDeltaHCSL2(Cf0[:,-1], Cf1[:,-1], Cf2[:,-1], 1.)\
 - getDeltaHCSL2(Cg0[:,-1], Cg1[:,-1], Cg2[:,-1], -1.)

fy[:,1:] = cf4*((-c*dt)**4) + cf3*((-c*dt)**3) + cf2*((-c*dt)**2) + cf1*(-c*dt) + cf0
gy[:,:-1] = cg4*((c*dt)**4) + cg3*((c*dt)**3) + cg2*((c*dt)**2) + cg1*(c*dt) + cg0
dyfy[:,1:] = 4.*cf4*((-c*dt)**3) + 3.*cf3*((-c*dt)**2) + 2.*cf2*(-c*dt) + cf1
dygy[:,:-1] = 4.*cg4*((c*dt)**3) + 3.*cg3*((c*dt)**2) + 2.*cg2*(c*dt) + cg1

dxfy[:,1:] = mf1*(-c*dt) + mf0
dxgy[:,:-1] = mg1*(c*dt) + mg0

Ixfy[:,1:] = Cf2*((-c*dt)**2) + Cf1*(-c*dt) + Cf0
Ixgy[:,:-1] = Cg2*((c*dt)**2) + Cg1*(c*dt) + Cg0

fy[:,0] = r*gy[:,0]; gy[:,-1] = r*fy[:,-1]
dyfy[:,0] = -r*dygy[:,0]; dygy[:,-1] = -r*dyfy[:,-1]
dxfy[:,0] = r*dxgy[:,0]; dxgy[:,-1] = r*dxfy[:,-1]
Ixfy[:,0] = r*Ixgy[:,0]; Ixgy[:,-1] = r*Ixfy[:,-1]

p = (fy + gy)/2.
v = (fy - gy)/(2.*Z)
dyp = (dyfy + dygy)/2.
dyv = (dyfy - dygy)/(2.*Z)
Iyp = (Iyfy + Iygy)/2.
Iyv = (Iyfy - Iygy)/(2.*Z)
dxp = (dxfy + dxgy)/2.
dxv = (dxfy - dxgy)/(2.*Z)
Ixp = (Ixfy + Ixgy)/2.
Ixv = (Ixfy - Ixgy)/(2.*Z)
Ip = (If + Ig)/2.
Iv = (If - Ig)/(2.*Z)
```