工学系のための偏微分方程式：
数値解法のPythonによる実習
Ver. 1.1

２０２４年5月7日（火）

熊本大学理学部　小出眞路

本プリントは「**工学系のための偏微分方程式**」（小出眞路，２００６，森北出版）の付録として公開するものである。２００６年に本の出版に際して公開したC言語による実習書のPython版である。

ここではPythonによるプログラミングの基本的知識を前提とする。Pythonプログラミングの基本的知識とは次の事柄を言う。

- 変数の取り扱い：型，定義，代入，演算 (加減乗除・剰余（余り）)
- 数値の出力
- プログラムの制御（反復，選択）
- 配列
- 関数
- 数値データの図形プロット

Pythonでは、データを可視化する図形処理ライブラリー（matplotlib）も含まれているので、図形プロットではそのライブラリーを用いる。Pythonのプログラミングについては、プログラムしながらでも学ぶことはできるが、できれば他書（例えば、文献 [1]）にひととおり目を通すことをお勧めする。本演習書では、Python3を用いる。Python3のインストール方法や実行方法、プログラムする上での注意点をあとで述べる。

本実習書ではコンピュータの具体的操作方法として主にＵＮＩＸシステム（Mac OS Xを含む）を例に示している。他のＯＳを使う場合はそれぞれコマンドを置き換えて読んでほしい（表1参照)。

OS	Mac/Unix	Windows
Python コンパイル・実行	`python3 file.py`	`python file.py`
Python 実行・書出し	`python3 file.py > data`	`python file.py > data`
ファイルリスト	`ls -rtl`	`dir /od`
ファイル比較	`diff file1.c file2.c`	`fc file1.c file2.c`
ファイルの表示	`more file`	`more file`
ファイルコピー	`cp file1 file2`	`copy file1 file2`
ファイル移動	`mv file1 file2`	`move file1 file2`
ファイル消去	`rm file`	`del file`
ディレクトリ作成	`mkdir directry`	`mkdir directry`
ディレクトリ移動	`cd directry`	`cd directry`
ディレクトリ消去	`rmdir directry`	`rmdir directry`

表 1　Mac/Unix と Windows コマンド対応表

Python の環境設定から実行まで（概観）

Python には様々なインストール方法がある。文献 [1] にならい、Anaconda 社の提供する “Anaconda” を利用することにより手軽に科学技術計算で利用する Python の環境を構築できる。Anaconda は Python の本体だけではなく、主要な科学技術計算用パッケージをまとめてインストールしてくれる。Anaconda の Individual Edition（無償の Anaconda）を以下のサイトからダウンロードできる：

```
https://www.anaconda.com/products/individual
```

本実習書では Python3 を用いるので、インストールする際に『Python 3.*』を選択する（*は数字)。インストールは GUI 画面に従って容易に行うことができるが、詳しいインストール方法は文献 [1] にもある。

Python のコンパイル･実行は、Mac OS X の場合にはシステムコマンドを使う『ターミナル』か『xcode』で行う。Windows OS の場合は、Anaconda によって生成される『Anaconda Prompt』あるいは『Anaconda Prompt (Anaconda3)』を用いる。Windows OS にもとから用意されている『Command Prompt』を使っても Python を起動できるが、自分で環境設定をする必要があり、お勧めしない。文献 [1] の説明で用いられている『Jupyter Notebook』[†1] を使っても良いが、本演習書では Command Line Interface（CLI）を使った場合の説明を行う。

また、本実習書ではプログラムを直感的に見やすくするために、あえて Python 的な書き方をせず C 言語でも流用しやすいようなプログラムを例としてあげた。しかし、計算の実行速度を速くするには、Python 的書き方をしたほうがよいところがあるので、そのような箇所は 2 つの書き方を示した。大規模な計算では Python 的書き方が必要となることもあるかもしれない。

Python プログラミンの注意点　Python のプログラミングで特に注意すべきところは、「文頭の位置」（インデント）が特別な意味を持っているということである。文のインデントにより、その文の階層が指定される。通常 Python では、インデントを 4 文字単位で行う。C 言語であれば、`{ }`で囲んで階層を指定するが、Python ではインデントで文の階層の指定を行うのである。

参考文献

[1] かくあき、『現場で使える！Python 科学技術計算入門』（翔泳社、2020）

[†1] Anaconda で Python をインストールすると、Jupyter Notebook は Python と同時にインストールされている。

目　　次

付 録A

数値解法のPythonによる実習

ここでは「工学系のための偏微分方程式」（小出眞路著，２００６，森北出版）の第６章で示した数値計算法の実習を行う。簡単な例題からはじめて最後には非線形偏微分方程式とくに流体力学方程式を解くまでを扱う。第６章の各節の説明を読み実際にプログラムを組みながら実習を行ってほしい。本を読んでいるだけでは分からないことが理解されるはずである。ここで間違いやすいところについては [注意] で注意を促し，プログラムを組む際の細かいコツは [コツ] に書いた。

A.1　課程１：　拡散方程式の数値解法

課題１では拡散問題を数値的に解いてみよう。まず，ここで扱う数学モデルを示す（第１章例１．９，第２章第６節例題２．３参照)。

数学モデル：

(i) 偏微分方程式　$\dfrac{\partial u}{\partial t}=\dfrac{\partial^2 u}{\partial x^2}$　　$(0<x<1,\ \ 0<t<\infty)$

(ii) 境界条件　　$u(0,t)=u(1,t)=0$　　$(0<t<\infty)$

(iii) 初期条件　　$u(x,0)=\sin\pi x$　　$(\ 0\le x\le 1)$

この数学モデルを数値計算で解くスキームは次のようになる。記号の意味などは第６章のものをそのまま引き継ぐ（ただ、“j” は “i” に変更した)。ここでは虚数単位 i を使うことはないので，メッシュの番号として i を用いることにする。また，メッシュ数を I と記す。

差分スキーム：

(i) 差分方程式　　$u_i^{n+1}=u_i^n+\dfrac{s}{h^2}(u_{i+1}^n-2u_i^n+u_{i-1}^m)$　　$(n=0,1,2,...,\ i=1,2,...,I-1)$

(ii) 境界条件　　$u_0^n=u_I^n=0$　　$(n=0,1,2,...)$

(iii) 初期条件　　$u_i^0=\sin\pi x_i$　　$(i=0,1,2,...,I)$

このスキームに基づいて拡散問題を数値的に解くことになるが，これらの式を見ただけで数値計算のプログラム（「コード」という）を作り上げるのは初心者には難しい。しかし，次から示すような手順を踏めば比較的簡単にコードが作成できる。

手順１　＜変数 x_i の設定＞

変数 x_i に対応する変数配列 `x[i]` を定義・設定し表示する。これは変数の定義，四則演算，反復計算，配列など Python 言語の基本を使えばすぐにプログラムを作ることができる。ここで I は `imax` と置く。例となる Python プログラムを示す。

```
import numpy as np

imax = 20
dx=1.0/imax

x = np.linspace(0, dx*imax,imax+1)

for i in range(imax+1):
```

```
print(x[i])
```

ここで、Python では配列を使うのにも、数値計算ライブラリー numpy（ナンパイ）を使う必要がある。具体的には、このプログラムの最初の行

import numpy as np

により numpy を np という略号で用いることを宣言している。以降でも分かるように、円周率 π の値や三角関数などの数値計算で必要な数学的環境がこれで全部与えられることになる。

このプログラムに diffu.py という名前を付ける。このプログラムをコンパイル・実行すると２１行の数値の並びが出力されるはずである：

```
% python3 diffu.py
```

このデータを目で眺めて確認することもできるが，可視化するようにしておくと後の変化の複雑な数値を扱うときに助かる。Python ではライブラリーが充実していて、その中にデータ図形処理ライブラリー matplotlib（マットプロットリブ）がある。それを用いて、このデータを図形表示する。まず、matplotlib を使うことを宣言するために、diffu.py のファイルの先頭を以下のようにする：

```
import numpy as np
import matplotlib.pyplot as plt
```

次に、diffu.py のファイルの尾部に

```
plt.plot(x, marker='o', color='red')


plt.show()
```

を加える。diffu.py を実行すると、データが図形表示されるはずである。このとき、図形表示と共に、データの値も表示される。このデータの表示がまどろっこしいときは、「#」を文頭につけてデータ出力の部分をコメントアウトするとよい。

```
# for i in range(imax+1):
#     print(x[i])
```

あるいは、文をトリプルクォート（三重引用符）「'''」と「'''」で囲んでも、コメントアウトできる。ただし、トリプルクォートはあくまでも文字列なので、#によるコメントのように実行時に無視されるわけではない。例えば、インデントされているブロックの中でトリプルクォートをコメントとして利用しようとする場合、インデントをあわせないとエラーになる。とういうこともあり、トリプルクォートを用いたコメントアウトはあまり用いられない。

手順２　＜u_i^n の初期条件の設定＞

次に，u_i^n の初期条件を設定する。ここでこの変数を u[i] として定義する。

u[i] の初期値は x[i] の値を設定した後に次の文を入れる。

```
u = np.sin(np.pi * x)
```

ここで，np.pi は円周率 π の値を与え、np.sin は正弦関数を与える。

変数 u[i] の値を出力する。プログラムの plt.plot のところを次のように変更する：

```
plt.plot(x, u, marker='o')
```

手順３　＜１ステップだけ進める＞

これで初期条件の設定が終わった。次に 1 ステップだけ計算を進めて（時間発展させて）みよう。現在のステップの変数 u を変数 uo[i] として、次のステップの変数として u を用いることにする。 1 ステップ進めるには差分方程式 (i) を用いて計算すればよい。変更する箇所は次のようである。

まず、uo[i] を与える。

```
    uo = u.copy()
```

また、imax を定義し dx を定義したあとで時間幅 dt($= s$) も定義する。ついでに κ に相当する変数 kap も計算しておく。

```
dt=0.001
kap=dt/(dx * dx)
```

初期条件を設定した繰り返し文の後に時間発展させる差分式を計算する繰り返し文を入れる。

```
for i in range(1,imax):
    u[i]=uo[i]+kap*(uo[i+1]-2.0*uo[i]+uo[i-1])
```

ここで、はじめにも述べたように、インデントで階層を指定するという Python 独特な文法に注意する。un[i] の端の値が定義されていないのでこれに続いて境界条件に基づき次のようにする。

```
u[0]=0.0
u[imax]=0.0
```

これでプログラムをコンパイル・実行させると少し異なった線が増えているはずである。これが第 1 ステップ目（時刻 $t = s$）の u_i^1 $(i = 0, 1, 2, \cdots, I)$ を表す。

手順４　＜時間発展させる＞

1 ステップ進んだので，次は nmax=7 まで進めてみよう。これは時間発展の文からその数値を出力する文までのところを繰り返すようにすればよい。

```
for n in range(1, nmax + 1 ):
    uo = u.copy()

    for i in range(1,imax):
        u[i] = uo[i] + kap * (uo[i+1] - 2.0 * uo[i] + uo[i-1])

    u[0] = 0
    u[imax] = 0

    plt.plot(x, u, marker='o', lw=2)
```

[コメント]　“u[imax] = 0” は、Python では “u[-1] = 0” と書くことが多いが、ここではプログラムを分かりやすくするために、前者で書いた。

ここで当然，プログラムのはじめ付近で nmax=7 の定義を忘れてはならない。

```
imax = 20
nmax = 7
```

これでプログラムをコンパイル・実行しデータをプロットすると 8 本の線が描かれるはずである。初期の値の三角関数の振幅よりも，小さい振幅の三角関数の線があるとそれが時間が t=dt*nmax=0.007 だけ進んだときの u を表す。少し時刻は違っているが図６．８を参照のこと。

試してみよう 1　　解析解との比較

次に解析解と比較するためにもう少し長く計算してみよう。nmax=70 とすると，時刻が $t = 0.07$ だけ経っていることになるので，解析解によると三角関数の振幅は $e^{-0.07\pi^2} = 1/1.995 \sim 0.5$ になるはずである。ほとんど，そうなっていることを確かめよ。多少違っていることがあれば，それは数値誤差による。しかし，その数値誤差は小さく，十分解析解に近い数値解が得られるはずである。ここで振幅が 0.5 になっているか分かりにくい場合は，図に点 $x = 0$，$y = 0.5$ と点 $x = 1$，$y = 0.5$ を結ぶ線を書かせるようにするとよい。そのために n の繰り返し文が終わった直後に、次の 1 文を入れるとよい：

```
plt.plot([0.5,0.5])
```

試してみよう 2　　第４章例題４．３に対応する計算（シルクハット型関数の初期条件）

第 II 部第４章の例題４．３で示した計算を行ってみよう。このときの解析解は例題４．３で求めた解となる。この問題を解くには単に初期条件を変えるだけでよい。具体的には手順２で設定した初期条件のところを次のように書き換えてやるとよい。

```
u = np.zeros(imax+1)
u[int(0.25/dx):int(0.75/dx+1)]=1
```

プログラムをコンパイル･実行しプロットしてみると解析解とよく一致していることが確認できるであろう（図６．４参照)。

試してみよう 3　　数値不安定性

これで線形拡散方程式を数値的に解くプログラムが出来たことになる。ここで数値不安定性がどんなものかを試してみよう。今までは，$\kappa = cs/h^2 = 1 \times 0.001/0.05^2 = 0.4 < 0.5$ であり，計算は安定であった。ここで，dt=0.002($s = 0.002$) として $\kappa = 0.8$ の場合を計算してみよう。nmax=70 のままでは計算できないので，nmax=3 としたほうが良い。すると最後の u はギザギザした分布となるはずである（図６．５参照)。ギザギザはノイマンの方法で予想される最も危険なモードで説明できる。その振幅は初期の振幅を超え，計算そのものを壊しているのが分かる。数値不安定性は数値安定条件（CFL 条件）が満たされないと十数ステップで計算全体を壊してしまうことが分かるであろう。数値不安定性は数値計算上絶対に避けなくてはならない注意事項のひとつである。dt=0.001，nmax=70 にして計算を元に戻そう。

試してみよう 4　　他の初期条件（魔女の帽子型関数）での計算

次に，初期条件として

$$u(x,0) = \Lambda(x) = \begin{cases} 1 - 4\left|x - \dfrac{1}{2}\right| & \left(\left|x - \dfrac{1}{2}\right| \le \dfrac{1}{4}\right) \\ 0 & \left(\dfrac{1}{4} < \left|x - \dfrac{1}{2}\right| \le \dfrac{1}{2}\right) \end{cases}$$

の場合を計算してみよう。このとき解析解は第 II 部第４章例題４．２のところで求めた解になる（図４．６参照)。数値的にこの問題を解くには単に初期条件を変えてやるだけでよい。具体的には手順２で設定した初期条件のところを次のように書き直してやるとよい。

```
u = np.zeros(imax+1)
for i in range(0,imax+1):
    if( 0.25 < x[i] and x[i] < 0.75 ):
        u[i] = 1.0 - 4.0*abs(x[i]-0.5)
```

プログラムをコンパイル・実行しプロットしてみると解析解とよく一致することが確認できるであろう。

以上４つの実習で分かるように数値計算による解法は解析的解法と違い扱える問題に基本的に制限がなく適用範囲が非常に広い。一方，誤差や数値不安定性など数値計算の設定や得られた解の解釈を慎重に行う必要がある。

手順５　＜出力データの間引き＞

ここで計算のメッシュ数（imax）が大きくなると dt を小さくしなくてはならず，図形出力の線が混み合ってきて見にくくなる。そこで出力するデータをステップを nskip = nmax/10 毎に出力し，出力を１０個程度の時間ステップに間引く。これをするには nskip を初期条件の設定の前あたりで定義・設定し，各ステップ毎にデータを出力するところを次のように if 文で制限すればよい。

```
    nskip = max( nmax / 10, 1)
    if( n % nskip == 0):
```

```
plt.plot(x,u,marker='o',lw=2,label='')
```

これで nmax = 1,000 くらいの長い時間を計算してみよう。ほとんど最終状態 ($t = 10$) では拡散し終わって u がゼロになることが見て取れるはずである。

うまく計算できないときはうまく計算できた手順まで戻ってその手順からもう一度はじめると良い。そのためにもうまく計算できた手順のプログラムは小まめに保存すると良い。

A.2　課程２：　1 階線形移流方程式の数値解法

次に波動方程式の数値計算に入ってみよう。いきなり 2 階線形波動方程式を扱う前に 1 階線形移流方程式 $\dfrac{\partial u}{\partial t} = -\dfrac{\partial u}{\partial x}$ の数値計算の実習を行う。ここでまず扱う数学的モデルを示す。

(i) 偏微分方程式　$\dfrac{\partial u}{\partial t} = -\dfrac{\partial u}{\partial x}$　$(0 < x < 1,\ 0 < t)$

(ii) 境界条件　$u(0,t) = u(1,t) = 0$　$(0 < t)$

(iii) 初期条件　$u(x,0) = \Lambda(x) \equiv \begin{cases} 1 - 4\left|x - \dfrac{1}{2}\right| & \left(\dfrac{1}{4} \le x \le \dfrac{3}{4}\right) \\ 0 & \left(0 \le x < \dfrac{1}{4}, \dfrac{3}{4} < x \le 1\right) \end{cases}$

このモデルの解析解は $u(x,t) = \Lambda(x - t)$ である。（ただし，u がゼロでないところが境界に達するまで。$t \le 0.25$）この数学モデルをラックス・ヴェンドロフ法により差分化すると次のように書ける。記号の意味は前の課程のものを引き継ぐ。

差分モデル

(i) 差分方程式　$u_i^{n+1} = u_i^n - \dfrac{s}{2h}(u_{i+1}^n - u_{i-1}^n) + \dfrac{s^2}{2h^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$　$(i = 1,2,...,I-1,\ n = 0,1,2,...)$

(ii)　境界条件　$u_i^n = u_I^n = 0$　$(n = 0,1,2,....)$

(iii)　初期条件　$u_i^0 = \begin{cases} 1 - 4\left|x_i - \dfrac{1}{2}\right| & \left(\dfrac{1}{4} \le x_i \le \dfrac{3}{4}\right) \\ 0 & \left(0 \le x_i < \dfrac{1}{4}, \dfrac{3}{4} < x_i \le 1\right) \end{cases}$

手順１　＜差分式の書き換え＞

差分モデルの式 (i),(ii),(iii) と前の課程 1 の**試してみよう** 4 の対応する条件は境界条件と初期条件は同じなので，違いは差分方程式 (i) のみである。すなわち，前の課程の最後に作ったプログラムの時間発展させる部分を差分モデルの式 (i) に変えるだけでよい。具体的には時間発展させる文を次のように変更すればよい。kap の定義をここでは s/h=dt/dx に変える。時間発展させる差分式は

```
for i in range(1,imax):
    u[i] = uo[i] - 0.5 * kap  * (uo[i+1] - uo[i-1]) \
         + 0.5 * kap * kap * (uo[i+1] - 2.0 * uo[i] + uo[i-1])
```

とできる。ここで前のプログラムをコピーして作った新しいプログラムの名前は wave1.py としておく。まず，dt = 0.01，nmax=10，imax=50 として計算してみよう。ここで最終時刻は t = dt * nmax =$10 \times 0.01 = 0.1$ である。時刻 $t = 0.1$ での解析解は $u(x,t) = \Lambda(x, t - 0.1)$ なので，右に 0.1 だけずれた点が見られるはずである。確かめてみよう。ほとんど形は保っているが山の頂上付近がすこし滑らかにされているかもしれない。これが数値拡散と呼ばれるものである。

[コツ]　後のためにここでのプラグラムを iryu.py として保存する。

```
cp wave1.py iryu.py
```

試してみよう 1　　数値不安定性

ここで数値不安定性についても体験してみよう。今は，$\kappa = cs/h = 1 \times 0.01/0.02 = 0.5 < 1$ で数値安定になっている。ここで，$s = \texttt{dt} = 0.03$ とすると $\kappa = 1.5$ となり数値安定性の条件が満たされなくなる。実際に計算してみると全く解析解とは関係のないようなギザギザの線が現れる。数値不安定性は計算を跡形もなく破壊してしまう絶対に避けなくてはならない事柄のひとつである。$\texttt{dt} = 0.02$ の場合 ($\kappa = 1$) もやってみよう。ここで初期値が数値拡散なしで幾何学的に右側にずれていくのか分かる。$\texttt{dt} = 0.01$ に戻して次に進む。

試してみよう 2　　矩形型初期条件

初期条件で次のような不連続な関数（シルクハット型関数）を設定してみよう。

$$u(x,0) = \Omega(x) = \begin{cases} 1 & \left(\dfrac{1}{3} \le x \le \dfrac{2}{3}\right) \\ 0 & (\text{その他}) \end{cases}$$

プログラム中の具体的な書き方は課程１の手順４の**試してみよう** 2 の式を用いる。この初期条件で時間発展させると矩形は移動することはするがかなり形が崩れることが分かる。矩形の前方では振動現象が見られ，後方では数値拡散の影響が出ている。この現象はラックス・ヴェンドロフ法で衝撃波や固相と液相などの境界のある場合のように物理量が不連続関数となる数値計算において顕著に現れ，実用上最も厄介な問題である。このような不連続な関数を取り扱う方法はいろいろと開発されている。TVD 法（Total variation diminishing scheme）などの風上差分法が主流であるが[†1]、それとは異なったＣＩＰ法という数値計算法も使われている[†2]。

A.3　課程３：　２階線形波動方程式の数値解法

いよいよ次に２階線形波動方程式 $\dfrac{\partial^2 u}{\partial t^2} = \dfrac{\partial^2 u}{\partial x^2}$ を数値的に解いてみよう。まず，数学モデルを示す（第６章第３節例題６．３参照）。

(i)　偏微分方程式

$$\frac{\partial u}{\partial t} = -\frac{\partial v}{\partial x}, \qquad \frac{\partial v}{\partial t} = -\frac{\partial u}{\partial x} \qquad (0 < x < 1,\ 0 < t < \infty)$$

(ii)　境界条件

$$u(0,t) = u(1,t) = 0, \qquad v(0,t) = v(1,t) = 0 \qquad (0 < t < \infty)$$

(iii)　初期条件

$$u(x,0) = v(x,0) = \Lambda(x) \equiv \begin{cases} 1 - 4\left|x - \dfrac{1}{2}\right| & \left(\dfrac{1}{4} \le x \le \dfrac{3}{4}\right) \\ 0 & \left(0 \le x < \dfrac{1}{4}, \dfrac{3}{4} < x \le 1\right) \end{cases}$$

ここで差分方程式にしやすいように２元連立１階線形波動方程式にする。ベクトル $\mathbf{u} = \begin{pmatrix} u \\ v \end{pmatrix}$，$\mathbf{w} = \begin{pmatrix} v \\ u \end{pmatrix}$ を用いて書き直せば，

(i)　偏微分方程式　　$\dfrac{\partial \mathbf{u}}{\partial t} = -\dfrac{\partial \mathbf{w}}{\partial x} \qquad (0 < x < 1,\ 0 < t < \infty)$

(ii)　境界条件　　$\begin{pmatrix} u(0,t) \\ \dfrac{\partial v}{\partial x}(0,t) \end{pmatrix} = \begin{pmatrix} u(1,t) \\ \dfrac{\partial v}{\partial x}(1,t) \end{pmatrix} = \mathbf{0} \qquad (0 < t < \infty)$

(iii)　初期条件　　$\mathbf{u}(x,0) = \begin{pmatrix} \Lambda(x) \\ \Lambda(x) \end{pmatrix} \qquad (0 \le x \le 1)$

と書ける。ただし，$\mathbf{0} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ である。ラックス・ヴェンドロフ法を用いて差分化した条件に書き直すと

[†1] 藤井孝蔵,「流体力学の数値計算法」（東京大学出版会，1994）
[†2] 矢部孝・内海隆行・尾形陽一,「ＣＩＰ法」（森北出版，2003）

(i) 差分方程式 $\mathbf{u}_i^{n+1} = \mathbf{u}_i^n - \frac{\kappa}{2}(\mathbf{w}_{i+1}^n - \mathbf{w}_{i-1}^n) + \frac{\kappa^2}{2}(\mathbf{u}_{i+1}^n - 2\mathbf{u}_i^n + \mathbf{u}_{i-1}^n) \qquad (n = 0, 1, 2, ..., i = 1, 2, ..., I-1)$

(ii) 境界条件 $u_0^n = u_I^n = 0$
$v_0^n = v_1^n, \quad v_I^n = v_{I-1}^n \qquad (n = 1, 2...)$

(iii) 初期条件 $\mathbf{u}_i^0 = \begin{pmatrix} \Lambda(x_i) \\ \Lambda(x_i) \end{pmatrix} \qquad (i = 0, 1, 2, ..., I)$

ここで差分式 (i) を成分で書くと

$$u_i^{n+1} = u_i^n - \frac{\kappa}{2}(v_{i+1}^n - v_{i-1}^n) + \frac{\kappa^2}{2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

$$v_i^{n+1} = v_i^n - \frac{\kappa}{2}(u_{i+1}^n - u_{i-1}^n) + \frac{\kappa^2}{2}(v_{i+1}^n - 2v_i^n + v_{i-1}^n)$$

である。

手順 1 ＜変数の数 qmax を増やす＞

これまで扱う物理量は 1 つだけであったが，今回は u と v の 2 つになる。今後，非線形方程式のところで流体力学方程式を扱うようになると物理量は 3 つになるので，一般に qmax 個の物理量を扱うようにする。

まずは同じ計算（現在 1 階線形波動方程式）を qmax 個の変数について行うようにプログラムを変更する。ここで，扱う変数は 2 次元配列を用いて，u[0][i], u[2][i],..., u[qmax-1][i] である。具体的な手順は次のようにすればよい。ここで元となるプログラムは前の手順で**試してみよう** 1 をはじめる前のものを使うことが好ましい。新しいプログラムの名前を wave2.py とでもしておこう。ここでは，qmax =2 として u と v はそれぞれ u[0][i]，u[1][i] に対応させる。

(1) 新しく用いる変数を宣言文に加える。

```
imax=50
nmax=10
qmax=2
```

(2) 変数 u を 2 次元配列に変える。（“#” はコメントアウトを意味する）

```
#（配列の作成）
x = np.linspace(0, 1, imax+1)
u = np.zeros((qmax,imax+1))

#（初期値設定）
for i in range(0,imax+1):
    if( 0.25 < x[i] and x[i] < 0.75 ):
        u[0][i] = 1.0 - 4.0*abs(x[i]-0.5)
        u[1][i] = u[0][i]

#（時間発展）
for n in range(1,nmax+1):
    uo = u.copy()

    for i in range(1,imax):
        u[0][i] = uo[0][i] - 0.5 * kap  * (uo[1][i+1] - uo[1][i-1]) \
             + 0.5 * kap * kap * (uo[0][i+1] - 2.0 * uo[0][i] + uo[0][i-1])
        u[1][i] = uo[1][i] - 0.5 * kap  * (uo[0][i+1] - uo[0][i-1]) \
             + 0.5 * kap * kap * (uo[1][i+1] - 2.0 * uo[1][i] + uo[1][i-1])
```

```
#（境界条件）
    u[0][0] = 0.0
    u[0][imax] = 0.0
    u[1][0] = u[1][1]
    u[1][imax] = u[1][imax-1]

#（データの表示）
    nskip = 1
    if( n%nskip == 0):
      plt.plot(x,u[0][:],marker='o',lw=2,label='')
```

初期の出力の u の添字も ‘ [0][i] ’ に変更する必要がある。
これは単に同じ計算を２（=qmax）回しただけなので u[0][i] と u[1][i] の値は以前の u[i] と全く変わらないはずである。このことを確かめよう。確かめるためには u[0][i]，u[1][i] をそれぞれ独立に出力するとよい。u[0][i] の値を出力するには、plt.plot のところを次のように書き換えるとよい：

```
#    if( n%nskip == 0):
#      plt.plot(x,u[0][:],marker='o',lw=2,label='')
     for i in range(imax+1):
         print(x[i],u[0][i])
```

それぞれの出力した先のファイルを data1，data2 として，それらのファイルが一致していることを確認するのがよい。（図で見ただけでは完全に一致しているか保障はない。）

ＵＮＩＸ上ではファイル１とファイル２が一致しているかどうかをみるには diff というコマンドを使う。

```
unix% diff ファイル１　ファイル２
```

として何の出力（応答）もなければ２つのファイルは一致していることになる。例えば以前の（iryu.py の実行結果の）データファイル名を data1 とし，新しく得られたデータファイル名を data2 とすれば

```
unix% diff data1 data2
```

として何の応答もなければ２つの結果は同じと確認できる。Windows ではコマンドプロンプトで diff の代わりに fc コマンドを用いる。

手順２　＜２階線形波動方程式の差分方程式への変更＞

次に実際に２階線形波動方程式の数値解法のプログラムに変更する。ラックス・ヴェンドロフ法の差分方程式を見ると，２階の線形波動方程式と１階の波動方程式の違いは単に右辺第２項の u と v が入れ替わっているだけである。よって，プログラムでも対応するところをの u[0][i] と u[1][i] を入れ替えてやるだけでよい。時間発展のところが次のようになる。

```
    for i in range(1,imax):
        u[0][i] = uo[0][i] - 0.5 * kap  * (uo[1][i+1] - uo[1][i-1]) \
             + 0.5 * kap * kap * (uo[0][i+1] - 2.0 * uo[0][i] + uo[0][i-1])
        u[1][i] = uo[1][i] - 0.5 * kap  * (uo[0][i+1] - uo[0][i-1]) \
             + 0.5 * kap * kap * (uo[1][i+1] - 2.0 * uo[1][i] + uo[1][i-1])
```

ここでの変更は時間発展させる差分式の右辺の第２項で u のはじめの添字 0,1 について４箇所交換すればよいだけである。

このプログラムをコンパイル・実行してデータをみると u[0][i] ，u[1][i] 共に前と同様に右側に移動してゆくのが見られるはずである。数学モデルの解析解が

$$u(x,t)=\Lambda(x-t),\qquad v(x,t)=\Lambda(x-t)$$

であるので，うまくこれが再現できていることになる。ここで，計算経過時刻 t と図形の移動距離から伝播速度を求め，その値が 1 になっていることを確かめよ（図６．９参照)。

試してみよう 1　　初期条件と波の伝播方向

ここで少し初期条件を変えてみよう。プログラムを新しくコピー（`wave2.py`）してからはじめる。

(1)　まず，`u[1][i]` に対応する v の初期条件を

$$v(x,0)=-\Lambda(x)$$

としてみよう。すると `u[0][i]` のデータを見ると形が左側にずれて行くのが観察できるであろう。これは，この初期条件を満たす解析解が

$$u(x,t)=\Lambda(x+t),\quad v(x,t)=-\Lambda(x+t)$$

であることから理解できる。

(2)　次に，`u[1][i]` に対応する v の初期条件を

$$v(x,0)=0$$

としてみよう。すると `u[0][i]` の高さ 0.5 の山形が左右に分かれて伝播してゆくのが見られるはずである。これは，この初期条件を満たす解析解が

$$u(x,t)=\frac{1}{2}[\Lambda(x+t)+\Lambda(x-t)],\quad v(x,t)=\frac{1}{2}[\Lambda(x+t)-\Lambda(x-t)]$$

であることから理解できる。

試してみよう 2　　数値不安定性

初期条件を元に戻して，`dt = 0.03` としてみよう。この場合，$\kappa=1.5$ となるので数値不安定が起こるはずである（図６．１０参照)。数値不安定が起こることを確かめたら，また `dt = 0.01` に戻しておこう。（ついでに `dt=0.02` の場合も見てみよう）

試してみよう 3　　境界での反射

`dt = 0.01` に戻し、`n = 100` として計算を行ってみよう。反射波が現れるが、かなり数値的な拡散で鈍っているのが見られる。

A.4　課程４：　２段階ラックス・ヴェンドロフ法

これまでの課程で線形波動方程式をラックス・ヴェンドロフ法を用いて計算してきた。次に非線形の計算をするために２段階ラックス・ヴェンドロフ法を用いたプログラムに書き換えてゆく。ただし，この課程では今までの計算を２段階ラックス・ヴェンドロフ法を用いて行うだけで，得られる結果は今までと全く同じはずである。（ここでの手順は難しい。前のファイルをセーブして進むと良い。）

ここで２段階ラックス・ヴェンドロフ法にするに次の差分式にすればよい。

$$\mathbf{u}_i^{\overline{n+1}}=\mathbf{u}_i^n-\frac{s}{h}(\mathbf{w}_{i+1}^n-\mathbf{w}_i^n)$$

$$\mathbf{u}_i^{n+1}=\frac{1}{2}\left[\mathbf{u}_i^n+\mathbf{u}_i^{\overline{n+1}}-\frac{s}{h}(\mathbf{w}_i^{\overline{n+1}}-\mathbf{w}_{i-1}^{\overline{n+1}})\right]$$

ここで新しい変数 $\mathbf{w}_i^n$ と $\mathbf{u}_i^{\overline{n+1}}$ が必要であることが分かるであろう。具体的には次のような手順でプログラムを変更していけばよい。ここで，新しいファイル（`laxwen2.py` とでも名付ける）を作って書き直していくようにする。

```
cp wave2.py laxwen2.py
```

手順１＜配列 w の導入＞

w_i^n を用いて差分式を書き直す。すると差分式は 1 本の式で書くことができる。

(1) w_i^n に対応する変数 w[q][i] のための 2 次元配列を作る。

```
w = np.zeros((qmax,imax+1))
```

(2) w_i^n の計算を差分式の計算の直前で行う。

```
for i in range(0,imax+1):
    w[0][i] = u[1][i]
    w[1][i] = u[0][i]
```

あるいは、Python 的な書き方で

```
w[0] = u[1].copy()
w[1] = u[0].copy()
```

としてもよい。後者の書き方の方が Python の実行速度は速くなる。しかし、あまり Python 的な書き方をすると、デバッグ時や後でプログラムを見るときに分かりにくくなる。ここでは、前者の書き方を用いる。

[注意] 必ず w の計算は n の for 文の中で計算すること。もちろん，時間発展の差分式の直前である。

(3) 差分式を w を用いて書き直す。

```
for i in range(1,imax):
    for q in range(0,qmax):
        u[q][i]=uo[q][i]-0.5*kap*(w[q][i+1]-w[q][i-1]) \
            +0.5*kap*kap*(uo[q][i+1] -2.0*uo[q][i] + uo[q][i-1]);
```

この変更にるデータの変化はないはずである。確認せよ（UNIX では diff を用いる)。

手順２ ＜２段階ラックス・ヴェンドロフ法の差分式への変更＞

２段階ラックス・ヴェンドロフ法の差分式に変更する。差分式のところ全体を次のように変更する。

(1) ２段階ラックス・ヴェンドロフ法の中間段階の差分式の変数 $u_i^{\overline{n+1}}$ に対応する変数 uh の２次元配列を作成する。

```
uh = np.zeros((qmax,imax+1))
```

(2) 今までの差分式のところを第 1 段階目の差分式の計算に置き換える。

```
for i in range(0,imax):
    for q in range(0,qmax):
        uh[q][i]=uo[q][i]-kap*(w[q][i+1]-w[q][i])
uh[0][imax] = 0.0
uh[1][imax] = uh[1][imax-1]
```

ここで i=imax での値が計算されていないので，境界条件によりそれを与えている。

(3) 第２段階目の計算を加える。その直前（第 1 段目の計算の直後）に w を計算し直すことも忘れてはならない。

```
for i in range(0,imax+1):
    w[0][i] = uh[1][i]
```

```
        w[1][i] = uh[0][i]

    for i in range(1,imax+1):
        for q in range(0,qmax):
            u[q][i] = 0.5*(uo[q][i] + uh[q][i] - kap  * (w[q][i] - w[q][i-1]))

    u[0][0] = 0.0
    u[1][0] = u[1][1]
```

ここでまた境界条件を整えた。以前の境界条件の設定はここではもう不必要なので，消すと良い。このプログラムの変更によるデータの変化はないはずである。（ただ，丸め誤差程度のずれはあるかもしれない。）確かめよ。

手順３　＜w の計算の関数化＞

ここで w の計算を 1 ステップ毎に 2 回計算する必要があることに気づくであろう。w の計算が簡単な今回のよう場合はこの同じ計算を 2 回繰り返すのも良いが，非線形方程式などを扱う場合 w の計算が複雑になり 2 箇所にほとんど同じことを書くのはかなり面倒になる。そこで，w の計算は「関数」の中でまとめて計算するとよい。例えば次のような関数を作っておけば，それを呼び出すだけで w の計算ができることになる。ここでこの関数に calw とうい名前を付ける。

```
def calw(u, w, imax):
    for i in range(0,imax+1):
        w[0][i] = u[1][i]
        w[1][i] = u[0][i]
```

これは、プログラムの先頭、ライブラリーの宣言の後に置く。計算本体（main）では w を for 文中で計算する代わりに単に calw(uo,w,imax)，calw(uh,w,imax) と書けばよい。ここで、calw は n のループの階層中に置く。ここで，結果が前の計算と同じであることを（UNIX であれば diff を用いて）確認する。

手順４　＜周期境界条件＞

ここでステップ数 nmax を大きくする（nmax=50）と山が境界にぶつかり，計算が崩れてしまう。解析的には完全反射が起こるはずだが数値計算ではそうはならないことが分かる。そこで数値的に扱いやすい境界条件である「周期境界条件」に設定しなおす。周期境界条件とは現象が周期的で端がないという条件である。今回の場合，数学的には $\mathbf{u}(0,t) = \mathbf{u}(1,t)$ と書ける。

(1) uh の境界条件を与えるところを次のように変更する。

```
    for q in range(0,qmax):
        uh[q][imax]=uh[q][0]
```

(2) 次に，u の境界条件を与えるところを次のように書き換える。

```
    for q in range(0,qmax):
        u[q][0]=u[q][imax]
```

こうすると波が右側の境界を横切って出てゆくと，同じような波が左の境界から入ってくるのが分かる。

試してみよう 1　　左向きに伝わる波

初期条件を $v = -u$ として、波が左向きに伝わる計算を行ってみよう。左端から出て行った波が右端から現れるはずである。

試してみよう 2　　両方向に伝わる波

初期条件を $v = 0$ として、波が左右両方向に分かれて伝わる様子を観察してみよう。

以上で非線形方程式（流体力学方程式）をとりあつかう準備が整った。

A.5　課程５：　非線形方程式＜流体力学方程式＞の数値解法

さて，いよいよ非線形方程式の数値解法の例として流体力学方程式の数値計算に取り掛かろう。ここで、新しい名（dydro.py とでもする）のファイルにコピーして書き加えてゆく。

手順１　＜変数 u の数 qmax を３に増やす＞

まず，変数の数 qmax を３に増やす。新しい未定義の配列には他の対応する変数と同じになるようにする。例えば，u[2][i]=u[1][i] や w[2][i]=w[1][i] とすればよい。これは calw の中と main の初期条件のところのみ変更すればよい。この変更は単に変数の個数を増やすだけで計算結果は変わらない。このことを確認する。

手順２　＜ w を流体力学方程式の流束密度にする＞

流束密度の行列 $\mathbf{w}$ を計算する関数 calw での計算を

$$\mathbf{w} \equiv \begin{pmatrix} \rho v \\ \rho v^2 + p \\ (e+p)v \end{pmatrix}$$

に対応するように書き換える。このとき各メッシュにおいて $\mathbf{u}$ の値から p，v を次のように求めてから $\mathbf{w}$ の計算をすると分かりやすい。

$$v = \frac{\rho v}{\rho}, \qquad p = (\gamma - 1)\left(e - \frac{\rho}{2}v^2\right)$$

ここで，ρ，ρv，e は $\mathbf{u} = \begin{pmatrix} \rho \\ \rho v \\ e \end{pmatrix}$ の各成分である。また，γ は比熱比であり，$\gamma = 5/3$ とする。

[注意]　Python では 5/3 と書いても、5.0/3.0 としても同じ計算となる。しかし、C 言語や Fortran では単に 5/3 と書くと、整数の割り算とみなされ、1.0 とされてしまう。それを避けるために、C 言語や Fortran では 5.0/3.0 と書く必要がある。本演習では Python を使っているが、今後他の言語を使うことを考えて 5.0/3.0 と書くことを勧める。

手順３　＜初期条件の設定＞

ここで初期条件を与える。

$$\rho = 1 \qquad (0 \le x \le 1),$$
$$\rho v = 0 \qquad (0 \le x \le 1),$$
$$e = \begin{cases} 1.9 - 4\left|x - \dfrac{1}{2}\right| & \left(\dfrac{1}{4} \le x \le \dfrac{3}{4}\right) \\ 0.9 & \left(0 \le x < \dfrac{1}{4}, \dfrac{3}{4} < x \le 1\right) \end{cases}$$

また，ここでは u[2][i] を出力するようにする。ガスが中央付近 ($x = 0.5$) から両側に向かって広がる様子が見て取れるであろう（図６．１１参照)。(広がる速度を計算せよ。)

試してみよう 1　　爆縮

爆縮の数値シミュレーションを行うために、さらに初期条件を第６章第５節の式 (6.35) に変えてみよう。このときのファイルの名前は、implosion.py としよう。結果については図６．１４と比較せよ。

[コツ]　爆発のシミュレーションで e の初期条件，それも $\dfrac{1}{4} < x < \dfrac{3}{4}$ の場合をちょっと変えるだけでよい。

試してみよう 2　　音波のシミュレーション

音波の伝播のシミュレーションを行うために，初期条件を次のように変えてみよう。ここで比熱比を $\gamma = 5/3$ とする。

$$u = \rho v = A\sin 2\pi x, \quad \rho = 1 + A\sin 2\pi x, \quad e = 0.9(1 + \gamma A \sin 2\pi x)$$

これは音速 $c = \sqrt{\gamma p/\rho} = 1$ の音の伝播を与える初期条件である。はじめは音波の振幅を $A = 0.01$（あるいは，$A = 0.0001$）として線形音波の伝わる様子を確認しよう（図６．１２ (a) 参照）。このとき音波の伝播速度が 1 になっていることを確認せよ。

[注意]　このときファイルを新しくし，そのファイル名を例えば `sound.py` とでもしておく。

うまく線形音波の計算ができているようであれば，音波の振幅 A を 0.01 から徐々に大きくしてゆき 0.1〜0.2 くらいまでの非線形音波の計算を行ってみよう。このとき自発的に衝撃波が形成されるようすが見られる（図６．１２ (b) 参照）。ただ，解析的にはノコギリ歯状の衝撃波が伝わる解になるので，衝撃波のような不連続な関数が現れるはずである。しかし，数値計算の結果では衝撃波の下流に当たるところに振動（ギブス現象）が現れる。これは数値的なもので，不連続な関数の現れる現象をラックス・ヴェンドロフ法で計算したときに現れる。

試してみよう 3　　高密度ガスの衝突

高密度ガスの衝突の数値シミュレーションを行うために、さらに初期条件を第 6 章第 5 節の式 (6.34) に変えてみよう。結果については図６．１３と比較せよ。このときのファイルの名前は、`collision.py` としよう。

これで 1 次元の流体力学方程式を解くコードができたことになる。初期条件を変えてさまざまな計算をしてみることをすすめる。実際は 2 次元・3 次元の数値計算がより用いられる。付録 B にそのための簡単な方針を示した。これらの計算についても挑戦して頂きたい。

付 録B

2次元流体力学方程式の数値解法

ここで2次元流体力学方程式の数値計算のためのプログラム（流体力学コード）の組み方について簡単に述べる。
2次元流体力学方程式は次のように書ける（第 1.4.8 節参照)。

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{\partial \mathbf{w}_1}{\partial x} - \frac{\partial \mathbf{w}_2}{\partial y} \tag{B.1}$$

$$\mathbf{u} \equiv \begin{pmatrix} \rho \\ \rho v_x \\ \rho v_y \\ e \end{pmatrix}, \mathbf{w}_1 \equiv \begin{pmatrix} \rho v_x \\ \rho v_x^2 + p \\ \rho v_x v_y \\ (e+p)v_x \end{pmatrix}, \mathbf{w}_2 \equiv \begin{pmatrix} \rho v_y \\ \rho v_x v_y \\ \rho v_y^2 + p \\ (e+p)v_y \end{pmatrix} \tag{B.2}$$

ここで $e = \dfrac{\rho}{2}v^2 + \dfrac{p}{\gamma - 1}$ は全エネルギー密度である。

次のような手順で1次元流体力学コードから2次元流体力学コードに変えることができる。

1. 1次元流体力学コード（sound.py を用いるとよい）において，$\mathbf{u}$，$\mathbf{w}$ の個数 (qmax) を 4 に増やす。
   このとき，q=3 の変数は例えば q=2 と同じにする。計算結果は以前と全く変わらないはずである。
2. 新しく加わる独立変数 y に対応する $\mathbf{u}$，$\mathbf{w}$ の次元を増やす。
   このとき，独立変数 y については一様として計算する。結果は以前と変わらないことを確認する。

```
u = np.zeros((qmax,imax+1,jmax+1))
w = np.zeros((qmax,imax+1,jmax+1))
```

   [コツ]　このとき，出力は j=0，j=jmax/2，j=jmax の3箇所で出力してみる。どれも同じ結果になることを確認する。
3. 新しく $\mathbf{w}_2$ に対応する配列 w2 を加える。このとき，w を w1 と書きかえる。
   w を w1 と書き換え，w2 を加える。ここで，w2 の値は全てゼロと設定する。
4. 時間発展の差分を2次元の2段階ラックス・ヴェンドロフ法の式にする。
   （第1段階，予測子）

```
uh[q][i][j] = uo[q][i][j] - kap1*(w1[q][i+1][j]-w1[q][i][j]) \
                          - kap2*(w2[q][i][j+1]-w2[q][i][j])
```

   （第2段階，修正子）

```
u[q][i][j] = 0.5*(uo[q][i][j] + uh[q][i][j]
                          - kap1*(w1[q][i][j]-w1[q][i-1][j]) \
                          - kap2*(w2[q][i][j]-w2[q][i][j-1]))
```

   ここで，初期条件を設定する前に kap1=dt * imax，kap2=dt * jmax とする。
5. calw 内で w1，w2 を計算するようにする。
   このとき，配列 w2 を加えるだけで，w1，w2 の計算はまだ変更しなくてもよい。前の計算を再現することを確認する。
6. calw 内で w1，w2 の計算を2次元流体力学方程式の流束密度の式 (B.2) に変更する。
   しかし，このときもまだ前の計算を再現するはずである。確認しながら進む。
7. 境界条件（周期境界条件）を整える。

8. y 方向に非一様な計算がうまく計算できるか試すために，y 方向への音波の計算を行う。
　このとき，初期条件を変えるだけでよい。

[コツ]　ここで，1番目～7番目のプログラムの変更では出力結果は変更前の結果と変わらないはずである。もとのプログラム (`wave2.py`) の出力を 'data0' というファイルにしておいて，常にそれと変更したプログラムの出力結果のファイルが同じであることを確かめながら進むとよい。（ファイルの比較の際は UNIX の場合は `diff`，Windows の場合は `fc` を用いる。）また，最後の8番目の変更でも出力を `j=0,1,2,···,jmax`，`i=0` とすれば結果は変わらないはずである。出力ファイルが同じであることを確認しよう。

試してみよう 1　　円形爆発のシミュレーション

ここで円形爆発の初期条件を与える。

$$\rho = 1 \qquad (0 \le x \le 1),$$
$$\rho v = 0 \qquad (0 \le x \le 1),$$
$$e = \begin{cases} 0.9 - 4\left(r - \dfrac{1}{4}\right) & \left(r < \dfrac{1}{4}\right) \\ 0.9 & \left(r \ge \dfrac{1}{4}\right) \end{cases}$$

ここで、$r = \sqrt{\left(x - \frac{1}{2}\right)^2 + \left(y - \frac{1}{2}\right)^2}$ である。

以上の手順で2次元の流体力学コードができる。このコードの初期条件を変えれば第1章で示したケルビン・ヘルムホルツ不安定性（例1．4）などさまざまな流体計算ができる。重力を入れることができれば、レイリーテイラー不安定性の計算も行うことができる。機会があれば，2次元・3次元数値計算の豊潤な世界を紹介したい。

付 録C

プラズマの数値シミュレーション

これまでは流体力学方程式を考えてきたが，宇宙物理・工学関係でよく用いられる**電磁流体力学**（MHD）を取上げてみよう。基礎方程式は流体力学方程式と同じように，質量密度の保存，運動量の保存，エネルギーの保存と電磁場についてのファラデーの法則を連立させたものである。ここで電気抵抗は無視できるという理想 MHD 条件（$\mathbf{E}+\mathbf{v}\times\mathbf{B}=\mathbf{0}$，$\mathbf{E}$ は電場の強さ，$\mathbf{B}$ は磁束密度）を用いる。方程式は保存型で次のように書ける。

$$\frac{\partial \mathbf{u}}{\partial t}=-\frac{\partial \mathbf{w}}{\partial x}$$

$$\mathbf{u}\equiv\begin{pmatrix}\rho\\ \rho v_x\\ \rho v_y\\ \rho v_z\\ e\\ B_y\\ B_z\end{pmatrix},\qquad \mathbf{w}\equiv\begin{pmatrix}\rho v_x\\ \rho v_x^2+p+B^2/2-B_x^2\\ \rho v_x v_y-B_xB_y\\ \rho v_x v_z-B_xB_z\\ (e+p+B^2/2)v_x-(v_xB_x+v_yB_y+v_zB_z)B_x\\ -B_xv_y+v_xB_y\\ -B_xv_z+v_xB_z\end{pmatrix}$$

ここで，B は磁場の大きさを示し，全エネルギー密度は $e=\dfrac{1}{2}\rho v^2+\dfrac{B^2}{2}+\dfrac{p}{\gamma-1}$ で定義される。ただし，$B^2/2$ が磁気エネルギー密度となるような単位系を取るものとする。例えば，MKSA 単位系の磁束密度 $\mathbf{B}^*$ との関係は μ_0 を真空の透磁率として $\mathbf{B}=\mathbf{B}^*/\sqrt{\mu_0}$ である。系を 1 次元としているので B_x は定数となる。MHD の計算では次のような課題がある。

- プラズマと磁場の相互作用：アルフベン波
  MHD のプログラムを用いると初期条件を次のようにしてアルフベン波の数値計算ができる。ここで，$B_x=1$ とする。

$$\rho=1$$

$$v_x=0$$

$$v_y=A\cos 2\pi x$$

$$v_z=0$$

$$p=0.9$$

$$B_y=-A\cos 2\pi x$$

$$B_z=0$$

  A は振幅である。
- 磁気爆発
  これは磁気圧によるプラズマの爆発を計算する課題である。初期条件を次のようにするとよい。

$$\rho=1$$

$$v_x = 0$$

$$v_y = 0$$

$$v_z = 0$$

$$p = 0.9$$

$$B_y = \max(-\sin 3\pi x, 0)$$

$$B_z = 0$$

ここで Python では関数 max に対応する組み込み関数は `max(配列)` を用いる。

いままで作ってきたプログラムは 1 次元の現象しか取り扱いはできない。流体現象で面白いものは 2 次元（3 次元）の現象に多い。そこでプログラムを 2 次元（3 次元）にまで拡張することが考えられる。

2 次元 MHD の計算では次のような課題が扱える。

- 磁気リコネクション
- パーカー (Parker) 不安定性
- 磁場中のジェットの伝播

参考のために 2 次元 MHD 方程式を示しておく。ここで簡単のため速度，磁場の z 成分はゼロと仮定すると，次のような保存方程式に書ける。（勿論，3 次元の場合もこれを拡張した保存方程式で書けるがここでは省く）

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{\partial \mathbf{w}_1}{\partial x} - \frac{\partial \mathbf{w}_2}{\partial y} \tag{C.1}$$

$$\mathbf{u} \equiv \begin{pmatrix} \rho \\ \rho v_x \\ \rho v_y \\ e \\ B_x \\ B_y \end{pmatrix}, \quad \mathbf{w}_1 \equiv \begin{pmatrix} \rho v_x \\ \rho v_x^2 + p + B^2/2 - B_x^2 \\ \rho v_x v_y - B_x B_y \\ (e + p + B^2/2) v_x - (\mathbf{v}\cdot\mathbf{B}) B_x \\ 0 \\ v_x B_y - v_y B_x \end{pmatrix}, \quad \mathbf{w}_2 \equiv \begin{pmatrix} \rho v_y \\ \rho v_x v_y - B_x B_y \\ \rho v_y^2 + p + B^2/2 - B_y^2 \\ (e + p + B^2/2) v_y - (\mathbf{v}\cdot\mathbf{B}) B_y \\ -v_x B_y + v_y B_x \\ 0 \end{pmatrix}$$

ここで

$$e = \frac{\rho}{2} v^2 + \frac{p}{\gamma - 1} + \frac{B^2}{2}$$

なので

$$p = (\gamma - 1)\left(e - \frac{\rho}{2} v^2 - \frac{B^2}{2} \right)$$

と求められる。$B_x = B_y = 0$ とおくと，$v_z = 0$，$B_z = 0$ の場合の 2 次元の流体方程式になる。