A5. 偏微分方程式 多次元問題

2021, Dec 03 *Daisuke Furihata (降簱 大介)*

Photo by Sunyu on Unsplash

空間次元が多次元の偏微分方程式

これまで空間次元が 1 である問題を扱ってきた．しかしながら，多くの現実の問題は空間次元が 2以上である．そこで，今回は空間次元が 2 以上である偏微分方程式を扱ってみよう．

対象例: 相分離問題モデル Cahn-Hilliard 方程式

たとえば比重の異なる二種類の物質を混ぜて放置すると，(軽いほうが上側に移動することから)全体はその二種類の物質ほぼそれぞれからなる二つの相に分離するという「相分離」現象が起きる．他にも，混ぜた物質同士がそもそも分離しようとする性質をもっている場合も同様に相分離が起きる．

こうした相分離現象のモデル方程式の一つが Cahn-Hilliard 方程式(wikipedia)で，多次元問題

の数値計算の対象としてはなかなか面白いものである．この方程式の仕組みはあまり難しくないので，授業時に白板で解説しよう．

なお，この方程式は時間発展方程式なのだが，普通の手法で数値計算をしようとすると時間方向の刻み幅である Δt を大変小さく取らないと不安定になる(数値解が発散する)性質を強く持つことでも知られている．そのため，相分離現象のシミュレーションを，という意味で計算したいのであればなかなか大変なのだが，今回は計算する空間を小さめに取り，かつ，空間次元も 2次元に抑えることでその雰囲気を味わってみよう．

さてこの Cahn-Hilliard 方程式であるが，対象とする関数 $u(\boldsymbol{x},t)$ に対して次のような偏微分方程式となっている(u の peak 値が ± 1 として問題のない範囲でパラメータを簡潔にしている)．

$$\frac{\partial u}{\partial t} = \Delta \left(-u + u^3 + q\,\Delta u \right) \tag{1}$$

ただし，$q<0$ は相の表面に働く表面張力の強さを表すパラメータで，その絶対値が大きいほど強いことになる．また，境界条件は本来は $\boldsymbol{n}$ を境界での外向き単位法線ベクトルとして以下のようになる．

$$\begin{cases} \nabla u \cdot \boldsymbol{n} &= 0, \\ \nabla(\Delta u) \cdot \boldsymbol{n} &= 0. \end{cases}$$

今回は本格的なシミュレーションが目的ではないので，まあ，この境界条件の代わりに，周期的境界条件を使うことにしよう(領域が矩形領域ならば，これらの境界条件の離散近似は難しくないが)．

いつもの気軽な method of line + Runge-Kutta 法で

これまでのように，偏微分方程式の数値解法の手軽な手法の一つである，「method of line + Runge-Kutta法」で今回もチャレンジしてみよう．ただし，安直な分だけ，Δt をちょいと小さく取る必要があるので，そこは覚悟しておこう．

まず，空間領域 $\Omega = [0,L] \times [0,L]$ に対し、その分割数を $N_x \times N_x$ とし、境界条件として周期的境界条件を離散化しよう．

そして $\Delta x = \Delta y = L/N_x$ として $u(\boldsymbol{x},t)$ の離散化に相当する従属変数を

$$\boldsymbol{u}(t) = \{u_{i,j}(t)\}_{i,j=1}^{N} \tag{2}$$

としよう．なおここでは $u_{i,j}(t) \cong u(x = i\Delta x, y = j\Delta y, t)$ と想定している．

そして周期的境界条件を離散化した境界条件だが，外側にはみ出す点を仮想的に考えたとして

$$
\begin{cases}
u_{0,j} &:= u_{N,j} \text{ for } \forall j, \\
u_{N+1,j} &:= u_{1,j} \text{ 同上}, \\
u_{i,0} &:= u_{i,N} \text{ for } \forall i, \\
u_{i,N+1} &:= u_{i,1} \text{ 同上}
\end{cases} \tag{3}
$$

と解釈するのが良いだろう．

こうしておいて、空間微分(ラプラシアン) Δ を今回は差分近似で離散近似しよう．
u に対するもとのラプラシアンが

$$
\Delta u = \left(\frac{\partial}{\partial x}\right)^2 u + \left(\frac{\partial}{\partial y}\right)^2 u \tag{4}
$$

であることから，二階中心差分を用いて

$$
(\Delta_d\, u)_{i,j} := \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} \tag{5}
$$

とするのが良いだろう．

そして，この離散近似を使って Cahn-Hilliard 方程式全体を離散近似して得られる常微分方程式は

$$
\frac{du_{i,j}(t)}{dt} = \Delta_d\left(-u + u^3 + q\,\Delta_d u\right)_{i,j} \quad \text{for } i = 1, 2, \cdots, N, j = 1, 2, \cdots, N \tag{6}
$$

という $N \times N$ 元連立常微分方程式の形になる．

あとは少しずつプログラムを作っていくだけだ．まずは、問題のパラメータを設定してしまおう．

```
q = -0.001
L = 1.0
Nx = 50
Δx = L/Nx
Nt = 1000000
Δt = 1/Nt
```

次に、ラプラシアンをどうにか構成しよう．

最初に思いつく方法は，係数行列 `D` というものを下記のように定義して，それを用いるも

のだ.

```
using LinearAlgebra
using SparseArrays

v = ones(Nx-1)
D = -2 * I + diagm(1 => v, -1 => v)
D[1,Nx] = D[Nx,1] = 1.0
D = sparse(  D / (Δx^2) )

Laplace1(u) =  D * u + u * D
```

少し解説すると，近似データそのものである行列 u に対して、上で定義した行列 D による $D*u$ をは行列 u の行添字に関する二階差分になる．同様に、$u*D$ は行列 u の列添字に関する二階差分になる.

よって，これらの和である $D*u+u*D$ は u に離散ラプラシアン Δ_d を適用した結果になる，という仕組みだ．これが最初の方法だ.

この方法は単純で間違いが入りにくく，そういう意味で「良い」と言える.

ℹ ただしこれはちょっと遅いのだ．行列と行列との積は計算量が多く，たとえ片方が sparse とはいえその添字処理もあって，どうしても処理に少し時間がかかる.

そこで 2番めの方法だ．これは，行列データを貰って，各要素の計算をベタで行うという，大変にかっこ悪い方法なのだが，その分，ちょっと速い．以下のような感じだ.

```
function Laplace2(u)
  n = size(u)[1]
  v = similar(u)

  # 内部点の計算．素直に二重ループ.
  for i in 2:n-1
    for j in 2:n-1
      @inbounds v[i,j] = u[i+1,j] + u[i-1,j] + u[i,j+1] + u[i,j-1] - 4
    end
  end

  # 境界線上の計算．角は外す.
  for j in 2:n-1
    @inbounds v[1,j] = u[1+1,j] + u[n,j] + u[1,j+1] + u[1,j-1] - 4u[1,
```

```
15        end
16
17        for j in 2:n-1
18          @inbounds v[n,j] = u[1,j] + u[n-1,j] + u[n,j+1] + u[n,j-1] - 4u[n,
19        end
20
21        for i in 2:n-1
22          @inbounds v[i,1] = u[i+1,1] + u[i-1,1] + u[i,1+1] + u[i,n] - 4u[i,
23        end
24
25        for i in 2:n-1
26          @inbounds v[i,n] = u[i+1,n] + u[i-1,n] + u[i,1] + u[i,n-1] - 4u[i,
27        end
28
29        # 4箇所の角
30        @inbounds v[1,1] = u[1+1,1] + u[n,1] + u[1,1+1] + u[1,n] - 4u[1,1]
31        @inbounds v[n,1] = u[1,1] + u[n-1,1] + u[n,1+1] + u[n,n] - 4u[n,1]
32        @inbounds v[1,n] = u[1+1,n] + u[n,n] + u[1,1] + u[1,n-1] - 4u[1,n]
33        @inbounds v[n,n] = u[1,n] + u[n-1,n] + u[n,1] + u[n,n-1] - 4u[n,n]
34
35        return v / (Δx^2)
36      end
```

ああかっこ悪い．間違えそうだし．ただ，今回のデータの規模ですらこちらの方が最初の方法より倍程度は速いのだ．最初の方法が速ければそっちの方を採用したいよなあ…

`mod` 計算を使って添字を循環させればいいじゃん，と思う人もいるだろう．そうするときれいなプログラムが書けるのだが，普通に実装すると 100倍ぐらい遅くなるのだ．それでいいならば，本質的に `mod` を用いて循環添字を実装している `CircularArrays` というパッケージがあるのでそれを使うといいだろう．プログラムは楽になる．か〜な〜り〜遅いけどな．

さて話を戻して，これら2つの計算方法の結果がきちんと一致することと，計算時間が異なることをここでしっかり確かめておこう．そうでないと心配だからね．

そこで計算対象として初期値を作ってしまおう．今回は「最初は二種類の物質をなるべく均等にかき混ぜた状態」を初期状態にするシチュエーションを考えて，$u \cong 0$ な状態を乱数で作り出そう．

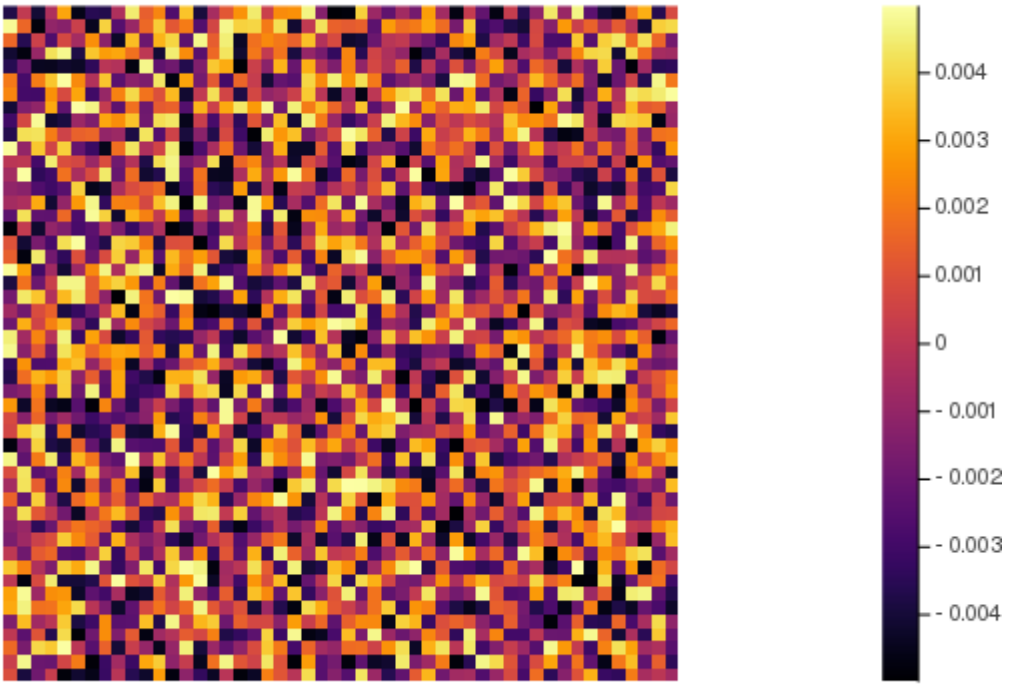
```
1   u0 = 0.01 * (rand(Nx, Nx) .- 0.5)
```

```
50×50 Matrix{Float64}:
0.000907744   -0.0039247    -0.00189216   …   -2.25458e-5   -0.000703019
0.000359209    0.00475919   -0.00200338        0.00139821   -0.00344962
…（値は乱数のため，やるたびに異なる）
```

どんな初期値なのか、目で見ておこう.

```
using Plots
default(aspectratio = 1.0, framestyle = :none)

X = Y = Δx:Δx:L
heatmap(X, Y, u0)
```



さて、これで準備は整ったので、早速ラプラシアン `Δ` の2つの計算方法の値の比較と，計算時間の比較を行おう.

まずは計算結果の比較だ.

```
error = Laplace1(u0) - Laplace2(u0)
maximum( abs.(error) )
```

```
1.4210854715202004e-14 (u0 が乱数なので左の値そのものは 人によって異なる)
```

計算結果はほぼ一致している．どちらの計算方法を使っても計算結果は実質的に変わらないと見て大丈夫だろう．

次に計算時間を見よう．

```
using BenchmarkTools

@benchmark Laplace1(u0)
```

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  11.800 μs …   8.380 ms  ┊ GC (min … max):  0.00% … 99.57%
 Time  (median):     30.600 μs               ┊ GC (median):     0.00%
 Time  (mean ± σ):   32.462 μs ± 205.235 μs  ┊ GC (mean ± σ):  18.79% ±  2.98%
```

```
 11.8 μs       Histogram: log(frequency) by time      54.4 μs <

 Memory estimate: 58.92 KiB, allocs estimate: 6.
```

```
@benchmark Laplace2(u0)
```

```
BenchmarkTools.Trial: 10000 samples with 7 evaluations.
 Range (min … max):   3.929 μs …  1.459 ms  ┊ GC (min … max):  0.00% … 98.92%
 Time  (median):     17.550 μs              ┊ GC (median):     0.00%
 Time  (mean ± σ):   17.419 μs ± 63.903 μs  ┊ GC (mean ± σ):  20.32% ±  5.58%
```

```
 3.93 μs       Histogram: log(frequency) by time      32.1 μs <

 Memory estimate: 39.30 KiB, allocs estimate: 5.
```

先に書いたように，明らかに後者の方が倍近く速い．今回は後者を使うことにしよう．

ということで，微分方程式の右辺の近似計算式は次のようになるだろう．

```
Laplace(u) = Laplace2(u)

f(u) = Laplace( - u + u.^3 + q * Laplace(u) )
```

```
f (generic function with 1 method)
```

さて、これで連立常微分方程式の右辺(の近似)が計算できたことになるので、いつものように Runge-Kutta 法を定義する．

```
function RK(u)
    f1 = f(u)
    f2 = f(u + Δt/2 * f1)
    f3 = f(u + Δt/2 * f2)
    f4 = f(u + Δt * f3)
    return u + Δt * (f1 + 2*f2 + 2*f3 + f4)/6
end
```

```
RK (generic function with 1 method)
```

これで計算手順の準備は整った．早速計算しよう．

```
using ProgressMeter

u = copy(u0)
sq_u = copy(u0)

n_last = Nt
n_skip = div(n_last, 200) # 整数の割り算

@showprogress for i in 1:n_last
    u = RK(u)
    if i % n_skip == 0  # 一定数ステップおきにデータを格納するとして
        sq_u = cat(sq_u, u, dims = (3) )  # 3次元目に追加していく
    end
end
```

```
Progress: 100%|████████████████████████████| Time: 0:13:07
```

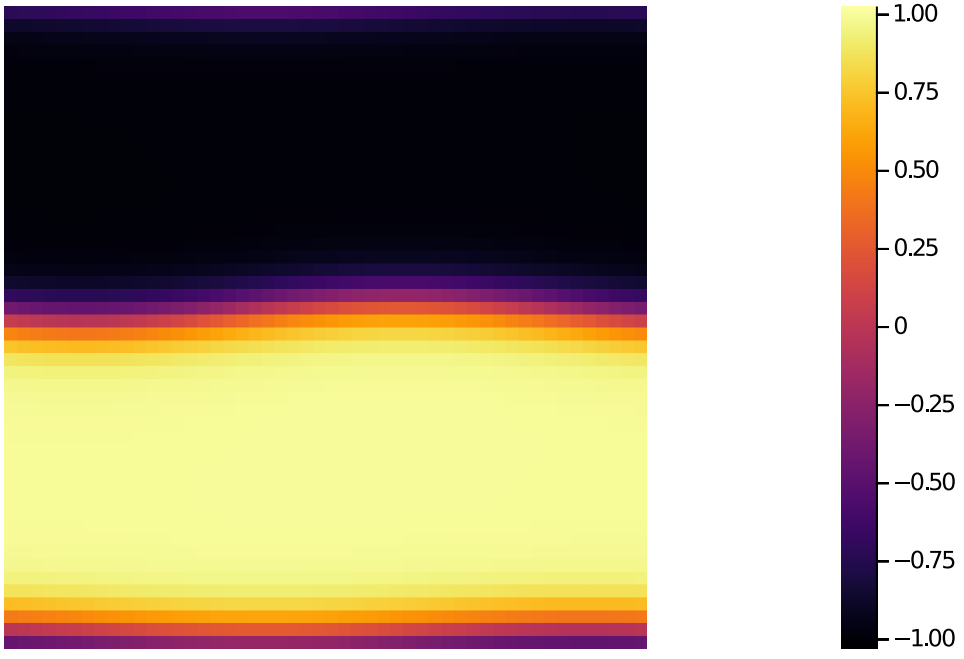
試しに結果の数字をちょろっと見てみる.

```
1    sq_u
```

```
50×50×201 Array{Float64,3}:
[:, :, 1] =
0.000907744  -0.0039247    -0.00189216   …  -2.25458e-5   -0.000703019
0.000359209   0.00475919   -0.00200338       0.00139821   -0.00344962
> …
[:, :, 2] =
-0.000811457  -0.00144386   -0.00211925   …   0.00024822   -0.000244266
-0.000153012  -0.000554191  -0.00112487       0.000227066   9.30991e-5
> …（初期値が乱数のため，結果もやるたびに異なる）
```

あまり変なことにはなっていなさそうなので、最終結果をまずはプロットしてみよう．こういう場合は heatmap 関数を使うのが良さそうだ.

```
1    heatmap(X, Y, sq_u[:,:,end])
```



ふむ，確かに分離しているようだな(初期値が乱数なので，貴殿の結果がこの図と異なってもそれは「当然」だ).

ちなみに，最終結果の u の値のスケールで初期値をあらためてプロットして，「初期値がたしかに均等っぽい」ことも確認しておこう．それには，計算値全体の最大値と最小値をまず次のように調べて，

```
1    vrange = ( minimum(sq_u), maximum(sq_u) )
```

```
(-1.028435658672145, 1.024132326800102)
```

次のように *clim* オプションで値の上下を固定して heatmap を使えば良い.

```
1    heatmap(X, Y, u0, clim = vrange)
```

確かに，時間発展結果に比べると初期値は「ほぼ均等にゼロに近い」ことが見て取れる．よし.

ということで，ふむ、なんとなくうまく計算できているようだ.

すべてのデータを一気に画像化する

あとは heatmapコマンドを好きなだけ繰り返せば途中結果も見られる．しかしまあそれはそれで様子がよくわからないので全データを画像 & 動画にしてしまうことを考えよう．

まず、たくさんの画像ファイルを入れておくディレクトリをコマンドで作らせよう．これは一回だけやれば良い．

```
run(`mkdir figures`)
```

```
Process(`mkdir figures`, ProcessExited(0))
```

注: 上の "mkdir" の実行後、juliabox のファイルビューアー部分を見てみよう．figures という新しいディレクトリができているはずだ．

次に、データを画像ファイルとして保存するコマンドを作ろう．基本的には、**plot** 系コマンドの直後に **savefig** コマンドを使うだけで、そのグラフを画像ファイルとして保存できる．今回は動画ファイルの「元」にしたいので、そうだなあ、"4桁の数字.png" というファイル名になるようにしておく．

```
using Printf # すぐ下の @sprintf を使いたいので.

default( clim = vrange )
# 画像全て，同じ値スケールで色を塗る.

function figure(num)
    s = @sprintf("%8.7f", (num-1) * n_skip * Δt)
    heatmap(X, Y, sq_u[:,:,num], title = "t = $s")
    # 時間をタイトルに表示

    savefig( "figures/" * @sprintf("%04d", (num-1)) * ".png" )
    # 4桁の数字.png というファイル名で保存
end
```

注: "@sprintf" というマクロは、昔から C 系統のプログラム言語で使われている sprintf 命令を模したもので、文字列を「決まったフォーマットで出力する」ものだ．使い方は "sprntf" などで検索してみよう．

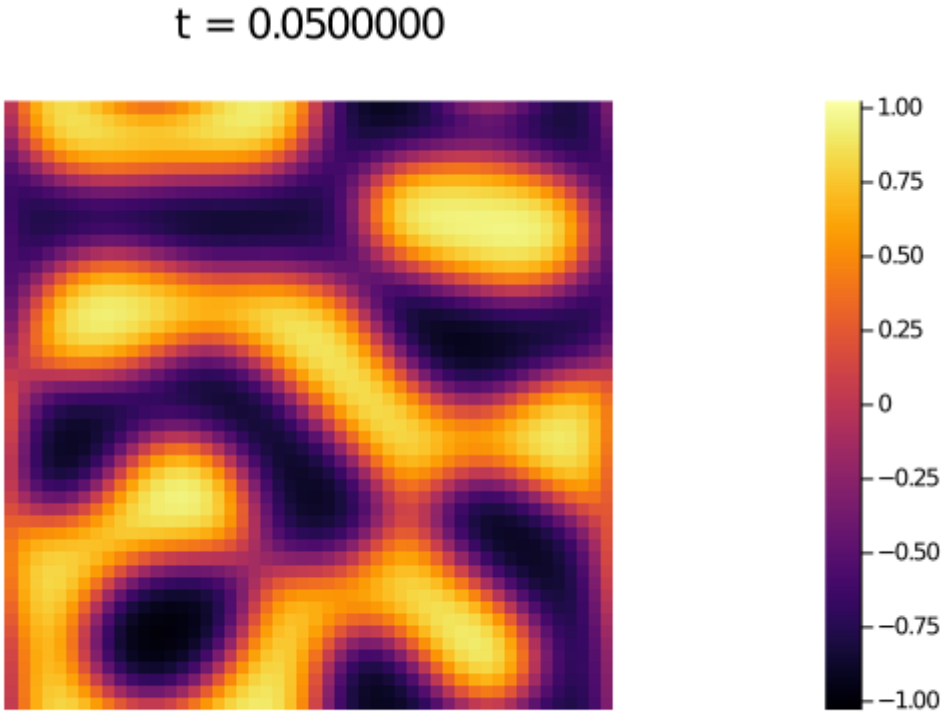
では、全データからたくさんの画像ファイルを作ろう．これは 10秒程度で終わるだろう．

```
@showprogress for n in 1:size(sq_u)[3]
```

```
        figure(n)
    end
```

```
Progress: 100%|████████████████████████████████| Time: 0:00:06
```

そして jupyter のファイルビューアー部分で．figures ディレクトリの下を見てみよう．今回は 201個の画像ファイルが生成されているはずだ．たとえばその 0010.png を見てみると次のような感じになっているだろう．



ffmpeg などの動画作成ツールで動画にしてみよう

さて，数百個もの画像ファイルが有るならばこれをつなげて動画にするのが可視化としては「いつもの手」というやつだ．そして世の中には有名なフリーのツールに FFmpeg というものがあり、これを使うと連番の画像ファイルを無理なくきれいに動画にできる．この手法は便利なので覚えておこう．

自分の PC に FFmpeg がインストールされていない、という人は、これを機にインストールしておこう． Download FFmpeg から windows用，Mac用，Linux用のインストール用ファイルがダウンロードできる．

さて、今回は FFmpeg を julia から直接呼び出して使ってみよう．以下のようにすればよいだろう．動画への変換はほぼ一瞬で終わるはずだ．

(詳しい人向け: terminal からコマンドを打っても良い)

```
1    run(`ffmpeg -y -r 10 -i "figures/%04d.png" -pix_fmt yuv420p -s 1200x800 c
```

```
ffmpeg version 4.4-full_build-www.gyan.dev Copyright (c) 2000-2021 the FFmpeg
  built with gcc 10.2.0 (Rev6, Built by MSYS2 project)
  configuration: --enable-gpl --enable-version3
…
```

これで ch-eq.mp4 という動画ファイルができたはずだ． jupyter のファイル操作画面で、その動画ファイルを view したり、ダウンロードして(jupyter はファイルを1つずつなら指定してから download できるぞ)動かしてみるなどして、確認しよう．

参考のために、そのファイルをここに直接おいておく．

- 動画ファイル (mp4 形式)

あと，もう少し大きな空間領域で計算した結果例も載せておこう(`Δx, Δt` 等は同じまま)．もちろん計算時間はさらに必要で，要した時間はそれぞれ約 30分，15時間というところだ．

- 動画ファイル, 空間領域サイズを 3x3 にしたもの (mp4 形式)
- 動画ファイル, 空間領域サイズを 5x5 にし, t=10 まで計算したもの (mp4 形式)

動画を見ることで，初期のほぼ均一な状況から一瞬にして相分離が発生し，その後，かなりゆっくりと相同士が融合して全体の構造が大きくなっていくことが見て取れるだろう．こうした「挙動の性質」は動画にしないと見えてこないことが多いので，時間発展問題の数値計算結果はなるべく動画にし，考察を加えるようにしておこう．

2021, Dec 03

多次元問題

←
A3. 多段解法, 予測子修正子法

---

10. 境界値問題 via 偏微分方程式
→