講義参考資料：

生命ダイナミクスを捉える：微分方程式と確率微分方程式

寺前　順之介

１．　微分方程式の数値解法

リズムを生み出し、運動し、刻々とその様子を変えて行く。生命現象にとって時間と共に変化する事、つまりダイナミクス、は最も重要で魅力的な性質の一つです。この時々刻々と「変化するもの」を捉える最も基礎的な数学的道具が微分方程式です（詳細は講義で説明します)。微分方程式は生命現象の理解に極めて有用な、まさに「使える！」道具なのです。

方程式と言うからには、それを解いて解を見つけ、その解を見て、知りたかった答えを知る、という流れを想像しますが、残念ながらそれは違います。生命現象の記述に現れる微分方程式は、**まず間違いなく、厳密には解けません**。しかし解けない事こそが生命ダイナミクスの豊かさに繋がります。生命現象に関する微分方程式は、この「解けない事」が出発点です。**「解けない、でも知りたい」。そんな時どうすれば良いか？**講義では、そのための様々な手法と考え方を含め、生命現象と微分方程式を解説しますが、計算機で数値的に微分方程式を解くのも､多くの場合有効です｡講義全体からすれば数値解法はおまけです。しかし、自分で微分方程式を数値解法する技術は習得しておいて全く損のない得だらけの技術です。この参考資料では、特に演習への橋渡しとして、微分方程式の数値解法の最も基礎的な部分を解説します。

皆さんの PC セットアップの都合上、そして会場での演習の則戦力となるために、この参考資料では数値計算の例を「R」で載せていきます。R は本来、統計解析目的の数値計算言語であり、ここでの用い方は少し異端です。しかしプログラミング言語の詳細を知る事は今回の講義の目標では全くなく、むしろ微分方程式数値解法の基礎を知る事がこの資料の目標ですので、ここでは言語としての R らしさは最小限にし、出来る限り微分方程式数値解法の最も基礎的の骨組みを見せるようにします。基礎は簡単で、しかもそれさえ知っていれば、どんなコンピュータ環境でどんなプログラム言語を用いていても微分方程式の数値解法が出来るようになるのです。

１．１　一変数の微分方程式

時間と共に変化する量（例えば、ある化学物質の量、タンパク質の発現量、個体の位置、細胞の膜電位・・・）を「変数」と呼びます。微分方程式は変数が時間と共にどう変化するかのルールを決めます。一変数 x の微分方程式は、例えば

$$\frac{dx}{dt} = ax - x^3 + b \qquad (1)$$

の形です。つまり

x の時間微分 ＝x の何かの式

です。例えば x が位置なら、時間微分 dx / dt は速度なので、微分方程式とは、動く速度が場所毎に決まっている床のような物ですね。**「微分方程式を解く」とは、ある出発点から始めて、速度のルールに従って動いた時に（変化した時に）位置がどのように変わって行くかを求める事です。**方程式の右辺には x 以外の文字、ここでは a と b、も含まれます。これらは時間的に変化しない量（例えば、細胞膜の抵抗値、実験でコントロールする温度、環境中の化学物質の濃度など）で「定数」や「パラメータ」と呼ばれます。x は時間 t の関数なので、正確には、式(1)は

$$\frac{dx}{dt} = ax(t) - x(t)^3 + b \qquad (2)$$

です。左辺の微分は、極限を用いて

$$\lim_{\Delta t \to 0} \frac{x(t+\Delta t) - x(t)}{\Delta t} \qquad (3)$$

ですが，普通の計算機は無限小を扱えないので、これを、十分小さいただの数（時間刻みと呼びます。Δt と書きましょう）に置き換え、近似的に

$$\frac{x(t+\Delta t) - x(t)}{\Delta t} = ax(t) - x(t)^3 + b \qquad (4)$$

とします。分母を払って書き直すと

$$x(t+\Delta t) = \left(ax(t) - x(t)^3 + b\right)\Delta t + x(t) \qquad (5)$$

です。式(5)は、時刻 t での x の値 x(t)から、ほんの少し先、時刻 t+Δt、の x の値 x(t+Δt)を求める式なので、これを繰り返し使い、次々と x を求めれば、x の時間変化が求まります。**これが微分方程式の数値解法です。**これは正確にはオイラー法（Euler method）と呼ばれる手法で、式(5)を求める事を「離散化する」と言います。次々同じ事を繰り返すのは計算機の得意技なので、その様にプログラムを書けば良い。そこでプログラムの概略は、

1. 繰り返し前の準備
2. 式(5)を何度も繰り返して、x を時間発展
3. 繰り返し終わったら結果（x の軌道など）を表示

です。

そのプログラムが図１。ここでは a = 10, b = 5 の時に、x(0) = 10 から出発した x の運命を、時間刻み dt = 0.001 で 500 回、つまり t = 0.01×500 = 5 まで求めています。

```
{
  ## Sample program 1
  ##  Jun-nosuke Teramae, 20120224
  ##
  ## dx / dt = a * x - x ^ 3 + b
  ##  Euler method
  ##
  ## 各行で"#"はコメントを表します。つまり"#"以降はコンピュータに見えません。

  ## 1. 準備
  A <- 10.0  # 式のパラメータをセット
  B <- 5.0   # 記号 「 <- 」 は代入の事

  DT <- 0.001   # 時間刻み
  TT <- 500     # 繰り返し回数
  TOUT <- 5     # 何ステップ毎に出力するか

  x <- 10.0     # xの初期値

  xx <- x       # xの結果のリスト,xx,にまず初期値を入れとく
  tt <- 0.0     # tの結果のリスト,tt,にも最初の時刻を入れておく

  ## 2. 式(5)を繰り返し
  for (i in 1:TT) {                     # i を 1 から N を変えながら、次の括弧{}内を繰り返せ
    dx <- (A * x - x * x * x + B) * DT  # (5)の第一項、「増分」と呼ぶ、を計算
    x <- dx + x                         # (5)を用いて、xを更新

    if (i %% TOUT == 0) {       # i が TOUT で割り切れたら 結果リストに結果を追加
      xx <- rbind(xx, x)        # リストxxに新しいxの値を追加
      tt <- rbind(tt, i * DT); # リストttに新しい時刻を追加、時刻はDTづつ増えるのでi*DT が時刻
    }
  }

  ## 3. 結果の出力
  quartz()                              # 表示用に画面を開く（quartz は mac の場合）
  matplot(tt,xx,type="l",pch=1,lty="solid")    # 横軸をtt、縦軸をxx、として結果を表示

tt
}
```

図１：サンプルプログラム１（色は勝手に付くので気にしない）

具体的には、まず「１．準備」として、

①パラメータの設定。

②数値計算の設定値（時間刻み、繰り返し回数）を設定。（結果を全部出力すると大変なので 5 回に 1 回だけ出力するように、T_OUT という変数もおまけで設定。）

③x と時刻の初期値を設定。

④出力先（求めた x と t をズラッと保存するリスト、xx と tt という名前）を用意し、

最初の値を代入。

の４つを行います。

次がメインの「２．繰り返し」、ここでは for を使って以下を繰り返します。

①現在の x を用いて、式(5)の第一項（「増分」と呼び、dx とした）を求める。

②現在の x に増分 dx を足して、新しい x を求める。

④（T_OUT 毎に）結果のリストに、今求めた新しい値を追加。「rbind(x,y)」は、R の命令「x の一番下に y をくっつけろ」。（「x %% y」は「x を y で割った余り」）

の４つを行います。ここが終わると xx と tt の中は、図２の様な数値が入っています。

最後にグラフとして「３．結果を出力」。

①グラフを出力する場所を指定。

注！この部分だけ、使っている PC により異なります。図３参照！

②結果リストを用いてグラフを描く。「matplot(x,y)」は R の命令で「リスト x を x 軸、y を y 軸としてグラフをかけ」。matplot 中の type や lty はグラフの書き方のオプション。詳細は図４。

xx	tt
10.000000	0.000
7.173439	0.005
6.041963	0.010
5.396491	0.015
4.972185	0.020
4.670516	0.025
4.445066	0.030
4.270675	0.035
4.132355	0.040
4.020544	0.045
3.928818	0.050
.	.
.	.
.	.
3.387625	0.500

図２：xx と tt

```
画面に表示したい時
    macの人：        quartz()
    windowsの人：    windows()
    unixの人：       x11()

PDFファイルに保存（ファイル名を「filename.pdf」とする時）
    pdf("filename.pdf")
    matplot(…)
    dev.off()              #ファイルを閉じる

Postscriptファイルに保存（ファイル名「filename.eps」）
    postscript("filename.eps")
    matplot(…)
    dev.off()              #ファイルを閉じる

bitmapファイルに保存（ファイル名「filename.bmp」）
    bmp("filename.bmp")
    matplot(…)
    dev.off()              #ファイルを閉じる

jpegに保存（ファイル名「filename.jpeg」）
    jpeg("filename.jpg")
    matplot(…)
    dev.off()              #ファイルを閉じる
```

図３：表示デバイス

```
matplotのオプション
type
    "l"    :    線で書く
    "p"    :    点だけで
    "b"    :    点と線で
    "o"    :    点と線の重ね書き

pch
    番号により点を描く記号が変わる
    ○とか△とか
    1      :    点を○で書く

lty
    線のタイプ
    "solid"     ：実線
    "dashed"  ：破線
    "dotted"  ：点線
```

図４：plot オプション

１．２　多変数の微分方程式

多変数でも原理は同様。

「１．準備して」

「２．変数の更新を繰り返し」

「３．出力する」

です。

例えば x,y,z 三変数の微分方程式

$$\begin{aligned}\frac{dx}{dt} &= py - px \\ \frac{dy}{dt} &= rx - xz - y \\ \frac{dz}{dt} &= xy - bz\end{aligned} \qquad (6)$$

は式(4)でやったのと全く同様に離散化出来るので

$$\begin{aligned}x(t+\Delta t) &= \left[py(t) - px(t)\right]\Delta t + x(t) \\ y(t+\Delta t) &= \left[rx(t) - x(t)z(t) - y(t)\right]\Delta t + y(t) \\ z(t+\Delta t) &= \left[x(t)y(t) - bz(t)\right]\Delta t + z(t)\end{aligned} \qquad (7)$$

となり、プログラムは図５。

ここではパラメータを p=10, r=28, b=2.66 とした時の、x=1, y=0 , z=0 からの軌道を t = 0.01×5000 = 50 まで求めています。

図１と比較すれば、単に変数が増えただけだと分かります。**ただし、一点だけ注意すべき点があります。良く似ていますが図５の繰り返し部分を図６の様に書いては行けません！**

図６のように増分と更新を変数毎に順に実行すると、「x <- dx + x」が実行された段階で x は x(t)から x(t+Δt)に変わります。すると y の計算式で、本来 x(t)を使うべき所で x(t+ Δt)を使う事になる。これでは正しい計算にはなりません。必ず、

「全変数の増分を計算」→「全変数の更新」　　(8)

の順で繰り返します。

```
{
  ## Sample program 2
  ##  Jun-nosuke Teramae, 20120224
  ##
  ## dx / dt = p * y - p * x
  ## dy / dt = r * x - x * z  - y
  ## dz / dt = x * y - b * z
  ##  Euler method

  ## 1. 準備
  P <- 10.0   # パラメータをセット
  R <- 28.0
  B <- 2.66

  DT <- 0.01    # 時間刻み
  TT <- 5000    # 繰り返し回数
  TOUT <- 1     # 何ステップ毎に出力するか

  x <- 1.0      # 初期値
  y <- 0.0
  z <- 0.0

  xx <- x       # xの結果のリスト,xx,にまず初期値を代入
  yy <- y       # yの結果のリスト
  zz <- z       # zの結果のリスト
  tt <- 0.0     # tの結果のリスト

  ## 2. 繰り返し
  for (i in 1:TT) {              # i を 1 から N を変えながら、次の括弧{}内を繰り返せ
    dx <- (P * y - P * x) * DT      # 増分を計算
    dy <- (R * x - x * z - y) * DT
    dz <- (x * y - B * z) * DT
    x <- dx + x                     # 変数を更新
    y <- dy + y
    z <- dz + z

    if (i %% TOUT == 0) {  # i が TOUT で割り切れたら 結果リストに追加
      xx <- rbind(xx, x)
      yy <- rbind(yy, y)
      zz <- rbind(zz, z)
      tt <- rbind(tt, i * DT);
    }
  }

  ## 3. 結果の出力
  quartz()                        # 表示画面を開く
  matplot(tt,xx,type="l",pch=1,lty="solid",col="blue")   # 横軸をtt、縦軸をxx
  quartz()                        # 別の表示画面を開く
  matplot(xx,zz,type="l",pch=1,lty="solid",col="red")    # 横軸をxx、縦軸をzz
}
```

図５：多変数のサンプルプログラム２

```
## Sample program 2 の間違いの例
##  Jun-nosuke Teramae, 20120224

## 2. 繰り返し
for (i in 1:TT) {                # i を 1 から N を変えながら、次の括弧{}内を繰り返せ
  dx <- (P * y - P * x) * DT       # 増分を計算
  x <- dx + x                      # 変数を更新
  dy <- (R * x - x * z - y) * DT
  y <- dy + y
  dz <- (x * y - B * z) * DT
  z <- dz + z

  if (i %% TOUT == 0) {  # i が TOUT で割り切れたら 結果リストに追加
    xx <- rbind(xx, x)
    yy <- rbind(yy, y)
    zz <- rbind(zz, z)
    tt <- rbind(tt, i * DT);
  }
}
```

図６：多変数の場合の良くある間違い

１．３　もっと多変数の微分方程式

それぞれ 2 変数の微分方程式で記述される細胞 10 個が、影響を与え合う時を考えると。全部で 2×10 = 20 変数が必要です。この時、図５のように、x, y, z, w, v, …(20 個)…とするのは大変なので、配列を使います。

例えば、n 番目の細胞の変数(v_n, w_n)は自分自身だけでなく、隣の細胞の v, つまり v_{n-1}, にも影響を受ける

$$\frac{dv_n}{dt} = v_n(1-v_n)(v_n-a)+b-w_n+k(v_{n-1}-v_n)$$
$$\frac{dw_n}{dt} = \varepsilon(cv_n-w_n) \qquad (9)$$
$$(n=1,\ldots,10,\quad v_0=v_{10},\ w_0=w_{10})$$

の様な場合（ただし 20 細胞はくるっと円状に並んでいて n = 1 は n= 10 に影響を受けるとしました）は、離散化すると、

$$v_n(t+\Delta t)=\left[v_n(t)(1-v_n(t))(v_n(t)-a)+b-w_n(t)+k(v_{n-1}(t)-v_n(t))\right]\Delta t+v_n(t)$$
$$w_n(t+\Delta t)=\left[\varepsilon(cv_n-w_n)\right]\Delta t+w_n(t) \qquad (10)$$
$$(n=1,\ldots,10,\quad v_0=v_{10},\ w_0=w_{10})$$

となります。，文字が x,y などから v_i などに変わった事以外は、離散化自体は今までと全く同じです。このプログラムは図 7 になります。

```
{
 ## Sample program 3
 ##  Jun-nosuke Teramae, 20120224

 ## 1. 準備
 A <- 0.05          # パラメータをセット
 B <- 0.1
 K <- 0.1
 EPS <- 0.02
 C <- 0.5
 N <- 10            # 細胞数

 DT <- 0.05         # 時間刻み
 TT <- 10000        # 繰り返し回数
 TOUT <- 10         # 何ステップ毎に出力するか

 v <- NULL          # 配列 v を用意
 w <- NULL          # 配列 w を用意
 for (n in 1:N) {   # N 個それぞれに初期値を代入
  v[n] <- runif(1)     # runif(1)は「[0,1)の一様乱数を一つ用意せよ」
  w[n] <- 0.1 * runif(1)
 }

 vv <- v            # 結果のリスト
 ww <- w
 tt <- 0.0

 dv <- NULL         # 配列 dv を用意、NULLは「とりあえず空の配列を用意しろ」
 dw <- NULL         # 配列 dw を用意

 ## 2. 繰り返し
 for (i in 1:TT) {   # i を 1 から N を変えながら、次の括弧{}内を繰り返せ
  ## まず全変数の増分を計算
  dv[1] <- (v[1] * (1 - v[1]) * (v[1] - A) + B - w[1] + K * (v[N] - v[1])) * DT     # 端だけ別扱い
  dw[1] <- (EPS * (C * v[1] - w[1])) * DT
  for (n in 2:N) {     # 残りは繰り返しで一気に書ける
    dv[n] <- (v[n] * (1 - v[n]) * (v[n] - A) + B - w[n] + K * (v[n-1] - v[n])) * DT
    dw[n] <- (EPS * (C * v[n] - w[n])) * DT
  }
  for (n in 1:N) { # 次に全変数を更新
    v[n] <- dv[n] + v[n]
    w[n] <- dw[n] + w[n]
  }
  if (i %% TOUT == 0) {  # i が TOUT で割り切れたら 結果リストに追加
    vv <- rbind(vv, v)
    ww <- rbind(ww, w)
    tt <- rbind(tt, i * DT);
  }
 }

 ## 3. 結果の出力
 quartz()                        # 表示画面を開く
 matplot(tt,vv,type="l",pch=1,lty="solid")   # 横軸をtt、縦軸をvv
 quartz()                        # 別の表示画面を開く
 matplot(vv,ww,type="l",pch=1,lty="solid")   # 横軸をvv、縦軸をww
}
```

図７：配列を用いる多変数のサンプルプログラム３

v[1]などが「配列」です。配列を用いる事で、変数 v_1, v_2,…をプログラム内では v[1], v[2],…で表す事が出来ます。配列の利点は for ループなどを用いて、配列の全ての要素（今は v[1], v[2], …, v[10]など）に同じ事をするプログラムが簡単に書ける事です。

図５と比較すると複雑に見えますが、基本は一緒です。図５と比較して変更された点は、

１．配列を用意する所、v <- NULL や dv <- NULL

２．変数の初期化、増分の計算、変数の更新、で n について for ループを用いる所

の二点だけです。（くるっと円状に並んでいる事を表すために、増分の計算の所で、端（n=1）だけを別個に計算しています。そこも少し複雑に見えるかも知れませんが、それはおまけです。）

こうして配列を使って、ループを回してしまえば、**20 細胞分、あるいは必要なら 100 細胞でも、10000 細胞でも、同様のプログラムを用いて数値計算する事が出来ます。**これは、細胞集団のモデルや複雑なネットワークモデルの数値計算には特に有効です。また、**異なる初期値から出発した沢山の軌道の振る舞いを同時に見たい時**や、次節で扱う確率微分方程式に対して、**少しずつ異なる影響を受けて時間発展する沢山の軌道を同時に見たい時**などにも有効です。

２．　確率微分方程式の数値解法

最後に、普通の微分方程式ではなく、確率微分方程式の数値解法について非常に簡単に述べておきます。詳細は講義で説明するので、ここではほぼ説明無しに、確率微分方程式の数値解法の基礎だけを例のみで示します。

確率微分方程式とは、(1)の様な dx/dt = f(x)の形の式の右辺に、さらに**確率的な項（ノイズ項）が加わった微分方程式**の事です。例えば,

$$\frac{dx}{dt} = ax(t) - x(t)^3 + b + \sigma\xi(t) \qquad (11)$$

の形です（実はこれは数学的に厳密な表記ではありませんが、講義で説明するので今は気にしない）。ξ(t)がノイズを表し、時々刻々全くでたらめな値をとる量です。係数σがノイズの強さを表すパラメータです。式(11)を離散化すると

$$x(t+\Delta t) = \left[ax(t) - x(t)^3 + b \right]\Delta t + \sigma\Delta W + x(t)$$

となります。**ΔW は平均０標準偏差 $\sqrt{\Delta t}$ のガウス分布に従う確率変数です。**ノイズ項以外は、式(5)と同様に離散化され、Δt が掛けられるだけですが、ノイズ項は、この確率変数に離散化されます。そこでプログラムは図８のようになります。

```
{
  ## Sample program 4
  ##  Jun-nosuke Teramae, 20120224
  ## dx / dt = a * x - x ^ 3 + b + sgm * xi
  ##  Euler method

  ## 1. 準備
  A <- 10.0          # パラメータをセット
  B <- 5.0
  SGM <- 10.0

  DT <- 0.001        # 時間刻み
  TT <- 5000         # 繰り返し回数
  TOUT <- 5          # 何ステップ毎に出力するか
  SDT <- sqrt(DT) # ルートDTを計算、sqdt は square root の事、つまりルート

  x <- 10.0          # 初期値

  xx <- x            # xの結果のリスト
  tt <- 0.0          # tの結果のリスト

  ## 2. 繰り返し
  for (i in 1:TT) {      # i を 1 から N を変えながら、次の括弧{}内を繰り返せ
    dx <- (A * x - x * x * x + B) * DT + SGM * SDT * rnorm(1) # 増分を計算、rnorm(1)は「平均0幅1のガウス乱数を一つ用意せよ」
    x <- dx + x          # 更新

    if (i %% TOUT == 0) {      # i が TOUT で割り切れたら 結果リストに結果を追加
      xx <- rbind(xx, x)
      tt <- rbind(tt, i * DT);
    }
  }

  ## 3. 結果の出力
  quartz()                        # 表示用に画面を開く（quartz は mac の場合）
  matplot(tt,xx,type="l",pch=1,lty="solid")   # 横軸をtt、縦軸をxx、として結果を表示
}
```

図８：確率微分方程式の数値解法、サンプルプログラム４

平均０標準偏差$\sqrt{\Delta t}$のガウス変数は、平均０標準偏差１のガウス乱数を$\sqrt{\Delta t}$倍すれば求まるので、増分の計算時には、用意しておいた$\sqrt{\Delta t}$、プログラム中ではSDTと書いています、を掛けて求めます。

連絡：

特にダウンロードしなくても構いませんが、今回例として挙げたサンプルプログラムは、講義当日まで

http://nct.brain.riken.jp/~teramae/sakigake2012/teramae_sample_1.r

http://nct.brain.riken.jp/~teramae/sakigake2012/teramae_sample_2.r

http://nct.brain.riken.jp/~teramae/sakigake2012/teramae_sample_3.r

http://nct.brain.riken.jp/~teramae/sakigake2012/teramae_sample_4.r

としてWeb上に置いておきますので、余裕のある方はどうぞ。