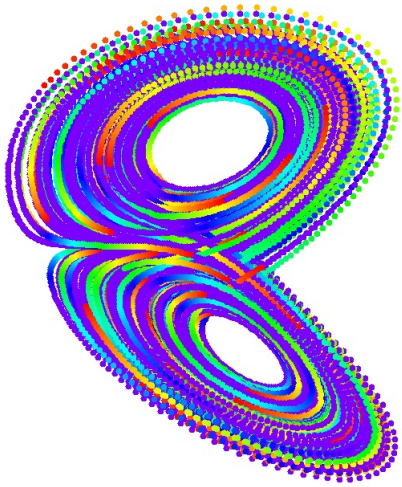

C言語とEGGXによる
コンピュータシミュレーション*

2024年6月30日

*西館数芽　https://sites.google.com/site/nisidatelab/

1　ようこそプログラミングの世界へ

物理シミュレーションを実行します。計算プログラムはC言語で記述します。グラフィックスにはEGGXを利用します。C言語については最初に簡単に説明しますが，あとは実際のプログラミングを通じて必要な時に必要なところをマスターしていきます。

C言語はメモリを直接扱うことができますし，なにより実行速度が速く，軽いアプリケーションを作り上げることができます。コンパイラはスーパーコンピュータにも搭載されていますので，C言語で書いたプログラムで大規模な計算も可能です。しかし一方で文字列処理などはかなり面倒です。そんなときはPerlやRubyというスクリプト言語を使用します。複雑なデータ処理も不得意です。そんなときはPythonという言語に助けを求めます。要は，プログラミングとデバッグに注ぎ込むエネルギーとその結果得られるリターンを考えて，適材適所で言語を使い分ける，といったところでしょうか。

第2章はLinuxの起動とエディタについて解説しています。第3章はC言語の大まかな説明にあてています。コンピュータシミュレーションは第4章からはじまります。腕に覚えのある人はいきなり第4章からはじめても面白いと思います。

前置きはこのくらいにして，さっそく取り掛かりましょう。

謝辞

この教材は理工学部 21 番の端末室で実施しているプログラム言語及び演習のテキストとして書きました。岩手大学情報基盤センターには端末の維持管理において大変お世話になっています。ここに記して感謝します。

本教材のグラフィックスは EGGX を利用しています。X11 グラフィックスライブラリである EGGX を開発し，公開されている山内千里氏に感謝します。EGGX に科学技術シミュレーションのプログラム例を提供されている松田七美男氏に感謝します。付録で紹介している乱数発生アルゴリズムのメルセンヌツイスタを開発し公開されている松本眞氏，西村拓士氏に感謝します。

この原稿は LaTeX/emacs 上の YaTeX（野鳥）の環境で作成されました。YaTeX を開発し公開されている広瀬雄二氏に感謝します。

目 次

2　ターミナル

2.1　起動

プログラミングは Linux 環境下で実行します。

- ディスプレイの電源を入れる
- コンピュータの起動時に **Ubuntu**[1] を選択[2]
- ＩＤを入力
- パスワードを入力

終了時には右上隅の電源マークアイコンから shutdown を選択します。ディスプレイの電源を切ることを忘れずに。

2.2　ターミナルの起動

ターミナル（端末）を起動し，その中で作業をします。ターミナルは，ディスクトップ左のアプリケーション，またはユーティリティから，テレビ画面のアイコンの「端末」を選択し，起動します。初めのワーキングディレクトリの状態は *home* に設定されています。コマンド **pwd** をキーボードから入力し，リターンキー（エンターキー）を打つと現時点のワーキングディレクトリ名（自分のいるディレクトリ名）が取得できます。

[1] ＯＳである Linux のディストリビューションの一つ。
[2] Windows と Ubuntu のデュアルブート設定の場合。

よく使うコマンドを以下に示します。

cd	ホームディレクトリに移動
cd ..	１つ上のディレクトリに移動
cd *dir1*	ディレクトリ *dir1* に移動
ls	ファイルリストの取得
touch *file1*	空のファイル *file1* を作成
rm *file1*	*file1* を削除
pwd	ワーキングディレクトリ名の取得
mkdir *dir2*	ディレクトリを *dir2* を作成
rmdir *dir2*	ディレクトリ *dir2* を削除
cp *file2 file3*	ファイル *file2* をファイル *file3* にコピー
mv *file3 file4*	ファイル *file3* をファイル *file4* に移動

Linux コマンドを解説しているサイトを以下にあげます。

- 「便利コマンド一覧」
- 「初心者のためのよく使う Linux コマンド一覧」

練習　一連の次の作業をしてください。

- コマンドの cd のみ打ち込んでリターンキーを押してホームに移動します。
- ディレクトリ work を作成してください。
- cd work でディレクトリ work に移動しましょう。
- pwd で自分が今いるディレクトリを確認してください。
- 空のファイル karafile を touch コマンドで作成しましょう。
- ls でファイルができたことを確認します。
- rm コマンドでファイル karafile を削除してください。
- コマンドの cd のみ打ち込んでリターンキーを押してホームに移動します。
- コマンド rmdir を使って空のディレクトリ work を削除してください。

注意　**ブラウザ（Firefox）が起動しない場合，**Firefox の初期設定フォルダを削除することで解決することがあります。

```
% rm -fr ～/.mozilla
```

これはホームディレクトリ以下（～/）のピリオドで始まる**.mozilla** ディレクトリを消去するコマンドです。もうすこし詳しく説明すると，**rm** がファイルを削除（リムーブ）する命令で，**-fr** がそのオプションです。オプションの内容ですが **f** で強制的に，**r** でリカーシブ（再帰的に）ということです。つまりディレクトリ**.mozilla** が内包している全ての階層のファイルを強制的に削除することを意味します。ただし自分で Firefox に組み込んだアクセス情報も削除してしまうので，必要に応じてバックアップをとってから実行して下さい。バックアップの例：

```
% cp -r ～/.mozilla ～/.mozilla.org
```

この場合，元の**.mozilla** ディレクトリをまるごとコピーした**.mozilla.org** ディレクトリを生成します。

注意 2　ピリオドで始まるファイルやフォルダは，通常の操作ではユーザーが見ることはできません。これはログイン時の初期設定や，様々なアプリケーションを起動したときの初期設定を保存するファイルやディレクトリがピリオドで始まるファイルやフォルダに保存されていて，通常，ユーザーがいちいち確認する必要が無いためです。上記の例で言えば，ホームで，

```
% ls
```

としても**.mozilla** ディレクトリは表示されません。

```
% ls -a
```

と，全てのファイルを表示するオプション**-a** をつけて **ls** を実行することで初めて**.mozzila** ディレクトリが表示されます。

2.3　エディタ

エディタ **gedit** を使ってプログラムファイルを作成します。本格的作業ができるエディタとしては他に **Visual Studio code** や **emacs**，**vim** などもあります。**emacs** については付録の第 B.1 節を参照してください。

練習　ディレクトリ work を作成し，そこに移動します。

```
% mkdir work ↵
% cd work ↵
```

ターミナルで gedit と入力することでグラフィカルなエディタ **gedit** を立ち上げることができます。[3]以下のコマンドを入力して，Cのプログラムファイル test1.c をコーディングしましょう。C のソースコードファイルは.c（ピリオドに続けて小文字の c）で終わる必要があります。

```
% gedit  test1.c ↵
```

プログラムの中身はソースコード 1 とします。ただし行頭の番号１，２，３は，行数を見やすくするためのもので，プログラムをするときは省いてください。3 行目で printf が字下げされていますが，これはプログラムを見やすくするためです。ここで最初の行の先頭の# （シャープキー，あるいはナンバーキー，ハッシュキー，日本式には「いげた」キー）は，キーボード左上にあります。３行目，printf の中の＼n は改行を指示する命令で，バックスラッシュ（日本語キーボードの場合￥マーク）を入力してから n を入力します。

ソースコード 1: 最初のCのプログラム (test1.c)

```
1 #include <stdio.h>
2 int main(){
3     printf("This␣is␣my␣C␣program!␣\n");
4 }
```

入力し終わったら gedit 右上の「保存」ボタンを押してファイルをセーブします。

練習 もう一つターミナルを立ちあげます。同じように cd コマンドでディレクトリ work に移動します。ソースコードのコンパイルは gcc コマンドを使います。

```
% gcc test1.c ↵
```

できあがった実行ファイルは，デフォルトでは a.out という名前になっています。カレントディレクトリ（自分が今いるディレクトリ）にある a.out を

[3]% はコマンドプロンプト。↵はリターンキーを押す事を示す。

実行するためにはファイルの先頭に./を付けます（ピリオドを入力してからスラッシュを入力）[4]。

```
% ls ↵　ファイルを確認
% ./a.out ↵　ファイル a.out を実行する
```

二つのターミナル間の切り替えは，CTL+TABもしくはALT+TABを押します。

練習 本格的なエディタとして **Visual Studio Code** を使うこともできます。起動は，

```
% code
```

です。最初に日本語化が必要です。 Microsoft 製の C/C++機能拡張も組み込みましょう。**ターミナル**メニューから**新規ターミナル**を選択すると画面の下にターミナルがあらわれます。そこでコンパイルと実行をします。最初のCのプログラム (test1.c) を code で開いてみて，操作性を確かめて下さい。

[4] ./は現在自分がいるディレクトリを指します。

2.4　エリアス

ターミナル環境下での作業を便利にするために**エリアス**を設定します。以下の例ではコマンド **ls** に **ls -F** をエリアスで割り当てています。

% ls ↵　通常のファイルリストコマンド **ls**
Applications　Library　Public　lib
Desktop　Movies　Sites　miniconda3
test1.txt
　このままではディレクトリとファイルの区別がつかない。

% ls -F ↵　オプション**-F** を付けて実行
Applications/　Library/　Public/　lib/
Desktop/　Movies/　Sites/　miniconda3/
test1.txt
　フォルダには名前の後に/マークが入るが,
　ファイルにはこのマークがつかない（フォルダとファイルの区別が容易)。
% alias ls='ls -F' ↵　**エリアスを設定**。
% ls ↵　**ls** と打つと **ls -F** が実行される。
Applications/　Library/　Public/　lib/
Desktop/　Movies/　Sites/　miniconda3/
test1.txt
　ls コマンドだけでフォルダとファイルの区別がつくようになった。

これで便利になったように思われるかもしれませんが，一つ大きな欠点があります。それはログインしたときに毎回エリアスの設定をコマンドベースで行わなければならないと言うことです。ログイン時に自動でこの処理をする方法があります。シェルの初期設定ファイルにエリアスの設定を書き込んでしまいます。以下では，皆さんは **bash** 環境下で作業をしていることを想定しています。

%echo $SHELL ↵　環境変数**$SHELL** の内容を確認。
/bin/bash　**bash** である。
% cp .bashrc .bashrc.org ↵
　初期設定ファイル**.bashrc** の編集に失敗したときのためにコピーを取っておく。
% gedit .bashrc ↵　初期設定ファイル**.bashrc** を編集。

初期設定ファイル**.bashrc** の最後に以下の４行を追加します。[5]

```
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
alias ls='ls -F'
```

コマンド **rm** や,**cp**, **mv** のオプション**-i** は，inquiry（照会，問い合わせ）をしながら実行するという意味です。たとえばファイル test.txt があったとして，**rm -i test.txt** を実行した場合，実際に削除する前に「本当に削除して良いですか？」と聞いてくるようになります。Linux ではコマンドベースで簡単にファイルの削除やコピーができてしまいますので，うっかり大切なファイルを削除してしまったり上書きをしてしまうかもしれません。このようにエリアスを設定しておくことでそのような「事故」を防ぐことができます。

この初期設定ファイルは次回ログイン時から自動で読み取られるようになるので，今後はコマンドでのエリアスの設定は必要無くなります。ただし初期設定ファイルを編集した時点ではまだシェルに設定が反映されていません。初期設定ファイルを編集したときに，その場ですぐに設定をシェルに反映するためには，各自のホームディレクトリで，

```
% source .bashrc ↵
```

を実行します。source でエリアスを認識したことは，

```
% ls ↵
```

を実行して，各フォルダ名の最後に／記号がつくことで判断出来ます。次回ログイン時には .bashr が読み込まれてエリアスが自動設定されるので source の操作は不要です。

[5] ログイン時に実行されるファイルは**.bash_profile** で，そのファイルから呼び出して実行されるのが**.bashrc** です。**.bashrc** には各ユーザー特有のエリアスやパスの設定を記述します。

3　C言語

ここでは当面必要となる C 言語の文法を解説します。ただし細部の解説や厳密な定義，当面使わない機能については一切省きます。C 言語については多くの良書が本屋さんで入手出来ます。自分に合った本を一つ選んで，必要に応じて参照すると理解が深まると思います。

3.1　構造

ソースコード 2: Cのプログラム（test2.c）

```
1 #include <stdio.h>
2 int main(int argc, char *argv[]){
3
4   // プログラム名：test2.c
5   // 機能： "This is my C program!"を表示する。
6
7   printf("This␣is␣my␣C␣program!␣\n");
8     return 0;
9 }
```

解説 #で始まる文はプリプロセッサへの指示です。#include<stdio.h>は，**stdio.h** というヘッダファイルを読み込むことを指示しています。

C のプログラムは関数で出来ています。一般的な関数の定義は，

型 関数名 (第一引数の型　名前，第一引数の型　名前・・・){・・・}

です。プログラムでは int main(int argv, char ∗argc[ ]){・・・} として，一番大元となる関数 main を定義しています。具体的には，関数 main は int 型で整数を返す関数であり，その第一引数 argv は int 型で整数，第二引数 argc は char 型でアスタリスクマーク（∗）がついているので，文字列配列のアドレスを受け取ることを宣言しています。その次の // で始まる行はコメントです。コメントは/* と */で囲まれた領域に書き込むこともできます。

printf は，ダブルクオーテーション (”) で囲まれた部分を画面に印字します。ここで \n は改行命令です。C のプログラムの本体の行は，”;”（セミコロン）で終わることになっています。printf の行もセミコロンで終わっています。

最後に return 0; として，この行までプログラムが正常実行された場合，関数 main の戻り値としてゼロを返すことを示しています。main はゼロを返すことが期待される関数なので int main として最初に宣言されています。

アドレス 変数をメモリに記憶したときの番地。

文字列配列 文字列を区分けして収めておく変数。

バックスラッシュ 改行命令 \n のバックスラッシュ”\” は，キーボード右上の￥マークを半角英数字モードで入力する。

よくあるミス1 プログラム本体は半角英数文字で入力すること。コメントの中身や printf の中のダブルクオーテーションで囲まれた部分以外に 2byte の全角文字が含まれているとコンパイル時にエラーとなります。特に見つけにくいエラーの１つに，全角の空白文字があります。空白文字なので画面上見ているだけではわかりません。カーソルが全角空白文字の上にあるとき，カーソルの幅が倍になるので全角空白文字がそこにあることがわかります。その他によくあるミスは，半角数字の 1 のつもりで英小文字のエルを用いるとか，半角数字のゼロのつもりで英字のオーを打ち込むなど。

よくあるミス2 行の終わりにセミコロンを入力しない，あやまってコロンを入力してしまう。ダブルクオーテーションのかわりにシングルクオーテーションを２個入力する。

3.2 型

3.2.1 void 型

次の func1 関数は，引数に整数型の変数 a をとります。

```
void func1(int a){
        ....
return;}
```

この void 型の関数 func1 は，ただの return で終わっているので，処理が終了した後，値を返さずに処理の主導権を func1 の親関数に引き渡します。関数 func1 が引数をとらない場合，引数の場所にも void を宣言します。

```
void func1(void){
        ....
return;}
```

とします。

3.2.2 型の種類

データ型とデータ長を表に示します[6] 。

データ型	内容	一般的なデータ長
char	文字型	1 Byte （8 bit）で 1 文字を表す
short int	短いデータ長の整数	2 Byte
int	通常の精度の整数	2 Byte 以上のデータ長
long int	より大きな整数	4 Byte 以上のデータ長
float	単精度浮動小数点	
double	倍精度浮動小数点	
long double	データ長の長い倍精度浮動小数点	

表 1: データ型とデータ長

次に型のデータ長を実際に Byte 単位（×8 bit 単位）で表示するプログラムを次に示します。

ソースコード 3: 型のデータ長を表示するプログラム（test3.c)。label

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]){
4
5   printf(">>> data length (byte unit) \n");
6   printf(" char = %lu\n", sizeof(char));
7
8   printf(" short = %lu int = %lu long = %lu\n",
```

[6]char はキャラあるいはチャーと呼ぶ。

```
 9     sizeof(short), sizeof(int),sizeof(long));
10
11   printf("␣float␣=␣%lu␣double␣=␣%lu␣long␣double␣=␣%lu\n",
12     sizeof(float), sizeof(double),sizeof(long double));
13
14   return 0;
15 }
```

字下げ プログラミングをするときは，**字下げ**をするようにしてください。この例では，main の中のプログラムの各行は，2 字分字下げされています。emacs でプログラミングしている場合，行のどこにカーソルがあっても TAB を押すだけで行全体が適切に字下げされます。

行の認識 emacs は，直前の行の終わりに; があるかどうかを見ることで前の行が終わったことを判断しています。TAB を押しても行全体が適切に字下げされない場合，すぐ上の行が; で終わっていない可能性が大です。

ここで printf 内に指定された書式，”%lu” は，符号なし倍精度整数（unsigned long）を 10 進で出力することを示しています。関数 sizeof(...) は，引数内の変数のサイズをバイト単位で返す関数です。また 1 Byte = 8 bit は 0 (OFF) か 1 (ON) の数字が 8 桁つらなったデータの単位です。合計で $2^8 = 256$ 種類の異なる組み合わせがあるため，256 種類の異なる記号を表現することができます。

3.3 変数

変数は，型，変数名１，型，変数名２；として宣言します。変数が定義されたブロック，すなわち **{ からはじまり，} で終わるまで**の中でのみ有効で，局所的に限定（ローカル）されています。これを変数のスコープがローカルであると言います。次の例では int 型の変数 a1 と b1 の２つを定義しています。

```
void func1(void){
        int, a1, int, b1;
        ...
return;}
```

変数 a1, b1 は，関数 func1 の中でのみ有効なローカル変数で，func1 の外では無効です。

以下に示すのは main が始まる前，main の外で定義された変数の例です。プログラム中のどこからでもアクセスできるグローバルな変数となります。これを変数のスコープがグローバルであるといいます。

```
int, a1, int, b1;
void main(void){
        ...
return;}
```

一般にグローバル変数の使用はあまり推奨されません。エラーが発生し，それがグローバル変数にかかわるものであった場合，プログラムのどこで発生したエラーなのかを特定するのが困難になるからです。関数内で閉じたローカルな変数のメリットをあげると，

- 見通しが良くなる
- バグを見つけやすい
- 後からプログラムのメンテナンスが容易になる

などです。ただし我々が目標としているコンピュータシミュレーションでは，ある決まったシステムに対して演算を施すコンパクトな構造のため，グローバル変数を無理にポインタなどを介してローカル変数にするようなことはしません。

3.4 static

static は，宣言されたスコープの範囲で値を保持し続ける変数を定義します。次の例は，関数 sub1 で int 型の static 変数 a を初期値 0 で設定した例です（変数 a が有効なスコープの範囲は関数 sub1 の中のみ）。

ソースコード 4: static を使った例（test4.c）。label

```
1 #include <stdio.h>
2
3 int sub1(void){
```

```
4   static int a=0;
5   int b=0;
6   a += 1;
7   b += 1;
8   printf("in␣sub1␣>>>␣a␣=␣%d␣␣b␣=␣%d\n",a,b);
9   return a;
10 }
11
12 int main(void){
13   int i, a=0, b=0;
14   for(i=0;i<10;i++){
15     b=sub1();
16     printf("a␣=␣%d␣␣b␣=␣%d\n",a,b);
17   }
18   return 0;
19 }
```

解説 int 型の関数 sub1 の中で，`static int a=0;` とローカルで int 型の static 変数 a を定義しています。sub1 が最初に呼び出されると値 0 で初期化されます。sub1 が main から呼び出されるたびに，変数 a は前回呼び出された時の値が回復されます。一方，変数 b は static ではないので，関数 sub1 が呼び出されるたびに初期値 0 がセットされます。

関数 sub1 の最後は `return a;` なので int 型の変数 a の値を返します。そのため関数 sub1 は int 型になっています。一方，main では 3 行目で `b=sub1();` となっているので，main でのローカル変数 b に sub1 から返ってきた値が代入されます。
main でのローカルな変数 a, b と，sub1 でのローカルな変数 a, b は別物です。

出力例

```
$ ./a.out
in sub1 >>> a = 1  b = 1
a = 0  b = 1
in sub1 >>> a = 2  b = 1
a = 0  b = 2
in sub1 >>> a = 3  b = 1
a = 0  b = 3
in sub1 >>> a = 4  b = 1
a = 0  b = 4
in sub1 >>> a = 5  b = 1
a = 0  b = 5
in sub1 >>> a = 6  b = 1
a = 0  b = 6
in sub1 >>> a = 7  b = 1
a = 0  b = 7
in sub1 >>> a = 8  b = 1
a = 0  b = 8
in sub1 >>> a = 9  b = 1
a = 0  b = 9
in sub1 >>> a = 10  b = 1
a = 0  b = 10
```

sub1 のローカルで static な変数 a は，呼び出されるたびに a+=1 を実行して前回の値に 1 プラスした値にセットされます。

3.5 文字列

ソースコード 5: 文字配列を使った例（test5.c）

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char *argv[]){
5    char str1[]="ABCDEFG";
6    char str2[]={'a','b','c','d','e','\0'};
7
8    printf("str1 = %s\n",str1);
9    printf("str2 = %s\n",str2);
10   printf("length of the str1 = %lu\n",strlen(str1));
```

```
11    printf("length␣of␣the␣str2␣=␣%lu\n",strlen(str2));
12  }
```

解説 始めに文字配列 str1 と str2 を定義しています。str1 には，ABCDEFG の文字が連続して入り，最後に文字列終了の記号 \0 が自動的に埋め込まれます。str2 も同様で，a, b, c, d, e, f, g の文字が連続して入ますが，文字列終了の記号 \0 は明示して埋め込んでいます。

関数 strlen() は引数として与えられた文字配列の大きさを返す関数です。この関数は string.h で定義されているため，プログラムの 2 行目で

#include <string .h>

として，システムで用意された定義ファイルを読み込んでいます。

3.6 strcpy

文字列のコピーには strcpy 関数を使います。

ソースコード 6: strcpy の例（test6.c）

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char *argv[]){
5    char str1[]="ABCDEFG";
6    char str2[]={'a','b','c','d','e','\0'};
7
8    printf("str1␣=␣%s\n",str1);
9    printf("str2␣=␣%s\n",str2);
10   printf("length␣of␣the␣str1␣=␣%lu\n",strlen(str1));
11   printf("length␣of␣the␣str2␣=␣%lu\n",strlen(str2));
12
13   strcpy(str1,str2);
14   printf("str1␣=␣%s\n",str1);
15   printf("str2␣=␣%s\n",str2);
16   printf("length␣of␣the␣str1␣=␣%lu\n",strlen(str1));
17   printf("length␣of␣the␣str2␣=␣%lu\n",strlen(str2));
18 }
```

出力例

```
str1 = ABCDEFG
str2 = abcde
length of the str1 = 7
length of the str2 = 5
str1 = abcde
str2 = abcde
length of the str1 = 5
length of the str2 = 5
```

プログラムのなかほどにある，strcpy(str1, str2); で，str2 の中身の文字列を str1 にコピーしています。そのため，出力でも str1 の内容が書き換わっています。

ここで str1 には初期値として 7 文字が，str2 には 5 文字が設定されています。上の例では 5 文字の内容を 7 文字が入る入れ物の str1 にコピーしました。しかしながらこの逆，

strcpy(str2, str1);

のように，少ない文字数の入れ物（str2）に多数の文字（str1 の中身）を入れようとするとトラブルが発生します（バッファーオーバーフロー)。このようなバグはシステムを脆弱にする可能性が高いので，一般には次のような予防的プログラミングが推奨されます。

ソースコード 7: strcpy の例 2（test7.c）

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char *argv[]){
5    char str1[]="ABCDEF";
6    char str2[]={'a','b','c','d','e','\0'};
7
8    printf("str1␣=␣%s\n",str1);
9    printf("str2␣=␣%s\n",str2);
10   printf("length␣of␣the␣str1␣=␣%lu\n",strlen(str1));
11   printf("length␣of␣the␣str2␣=␣%lu\n",strlen(str2));
12
13   if (strlen(str1) < 6) {
14     strcpy(str2,str1);
15   }
16   printf("str1␣=␣%s\n",str1);
```

```
17   printf("str2␣=␣%s\n",str2);
18   printf("length␣of␣the␣str1␣=␣%lu\n",strlen(str1));
19   printf("length␣of␣the␣str2␣=␣%lu\n",strlen(str2));
20 }
```

この場合，strcpy で文字列をコピーする前に，if 文で条件を判断しています。

```
if (strlen(str1) < 6) {
  strcpy(str2,str1);
}
```

str1 が 6 文字より小さいならば (5 文字以下ならば)，str2 に str1 の内容をコピーします。今の場合，str1 は 6 文字ですので，この if 文によりはじかれ，カッコ内の strcpy は実行されません。一方，str1 が ABCDE ならば strcpy が確かに実行される事が確認できます。

練習 ソースコード 7 を書き換え，バッファーオーバーフローの実験をしてみてください。

3.7 式

式は＝の右辺での計算値を左辺に代入します。２つの整数値を読み取り，足し算の結果を表示するプログラムを以下に示します。

ソースコード 8: 足し算の例（test8.c）

```
1  #include <stdio.h>
2
3  int main(void){
4    int a, b;
5    int y;
6
7    printf("input␣a␣and␣b␣>>>␣␣");
8    scanf("%d␣%d",&a,&b);
9    y = a + b;
10   printf("%d␣+␣%d␣=␣%d\n",a,b,y);
```

```
11    return 0;
12 }
```

解説 最初にint型の変数a, b, yを用意します。始めにprintfで，ターミナル上にinput a and b と表示します。入力された値をscanfで読み取ります。ここで%dは整数型で読み取ることを意味します。また&aは，変数aのアドレスを意味しています。プログラムでは，

```
scanf("%d %d",&a,&b);
```

となっています。これは空白で区切られた整数型の値2つを読み取り，整数型変数aとbの記憶アドレスに直接代入することを意味します。つまり，scanfで数値を読み取るためには**変数のアドレスを指定する記号& (アンパサンド) が必要**です。つぎに足し算を実行し，printf関数にて結果を表示します。

出力例

```
input a and b >>>  5 9
5 + 9 = 14
```

次は足し算，引き算，かけ算（＊），割り算（／），余り（%）を求める例です。

ソースコード 9: 演算の例（test9.c）

```
1  #include <stdio.h>
2
3  int main(void){
4    int a, b, y1, y2, y3, y4a, y5;
5    double y4b;
6
7    printf("input␣a␣and␣b␣>>>␣␣");
8    scanf("%d␣%d",&a,&b);
9    y1 = a + b;
10   y2 = a - b;
11   y3 = a * b;
12   y4a = a / b;
```

```
13    y4b = a / (double)b;
14    y5 = a % b;
15    printf("%d␣+␣%d␣=␣%d\n",a,b,y1);
16    printf("%d␣-␣%d␣=␣%d\n",a,b,y2);
17    printf("%d␣*␣%d␣=␣%d\n",a,b,y3);
18    printf("%d␣/␣%d␣=␣%d\n",a,b,y4a);
19    printf("%d␣/␣(double)%d␣=␣%lf\n",a,b,y4b);
20    printf("%d␣/␣(double)%d␣=␣%lf␣\cdots␣%d.\n",
21           a,b,y4a,y5);
22    return 0;
23 }
```

ここで,

```
y4b = a / (double)b;
```

は，分母の変数 b にキャストで double を付けることにより，b 自体を整数から浮動小数点数に格上げしています。そのため右辺の計算全体が浮動小数点数として演算され，その結果が double 型の y4b に格納されます。

出力例

```
input a and b >>>  7 3
7 + 3 = 10
7 - 3 = 4
7 * 3 = 21
7 / 3 = 2
7 / (double)3 = 2.333333
7 divided by 3 is 2 with a remainder of 1.
```

3.8 if 文, for ループ

判定条件が成立した場合に文１の処理を実行させるためには if 文を使用します。

```
if(判定条件)  文１；
```

判定条件が成立した場合に複数の計算を実行させたい場合には，波括弧の { と } で囲まれた，ひとかたまりのブロック文を使用します。

```
if(判定条件){文１；文２；}
```

if 文を扱った例として、素数を判定するコードを以下に示します。

ソースコード 10: 素数の判定（test10.c）

```
1  #include <stdio.h>
2
3  int main(void){
4    int i, inum, id=0;
5
6    printf("input␣natural␣number:␣");
7    scanf("%d", &inum);
8
9    for(i=2;i<inum;i++){
10     if(inum%i==0){
11       id = 1;
12       break;
13     }
14   }
15
16   if(id==0){
17     printf("%d␣is␣a␣prime␣number␣\n",inum);
18   } else {
19     printf("%d␣is␣NOT␣a␣prime␣number␣\n",inum);
20   }
21
22   return 0;
23 }
```

解説 4 行目で最初に int 型を定義するときに、変数 id だけは id=0 として、初期値ゼロを id にセットしています。7 行目の scanf は、プログラムが実行された

ときに端末からの入力を読み取る命令です。ここでは%d で、整数型の数を読み取り、それを int 型変数である inum のメモリ（&inum）に格納しています。

9 行から 14 行までが for ループのブロック（中身）です。

```
for(i=2;i<inum;i++){
  .....
}
```

繰り返し変数 i に初期値 2 を代入し、i<inum が成立している間、i を一つ増やしながら（**i++**）ループを繰り返します。

10 行から 13 行が if 文のブロックです。inum を i で割った余りがゼロかどうか判定（**inum%i==0**）します。ゼロであるならば素数ではないので 11 行目と 12 行目を実行します。11 行目で id に 1 をセットし、12 行目で **break** によって内側の for ループ（ここでは 9 行から 14 行までのブロック）を抜けます。

16 行目から 20 行目では id の値によって出力を変える if ブロックです。id がゼロならば (**id==0**)17 行目が実行され、入力された数値は素数であると表示します。id がそれ以外（**else**）ならば 19 行目が実行され、入力された数値は素数ではないと表示します。

練習 素数では無いと判定したときに、割ることのできる数を少なくても一つ出力するようにプログラムを書き換えなさい。

出力例

```
input natural number: 13333
13333 is NOT a prime number
it can be divided by 67.
```

3.9 ポインタ

ポインタはメモリアドレスにつけるラベルに相当します。ポインタを使うと変数を収めたメモリアドレスを直接やり取りできます。

ソースコード 11: ポインタの例（test-p.c）

```
1  #include <stdio.h>
2
3  int main(void){
4    int *p, i, ii;
5
6    i=100;
7    p=&i;
8
9    for(ii=0;ii<2;ii++){
10     printf("␣␣i␣=␣%d\n", i);
11     printf("␣&i␣=␣%p\n", &i);
12     printf("␣*p␣=␣%d\n", *p);
13     printf("␣␣p␣=␣%p\n", p);
14     printf("␣&p␣=␣%p\n␣\n", &p);
15     i=300;
16   }
17
18   return 0;
19 }
```

解説 最初に整数の宣言で、

```
int *p, i;
```

としています。変数 p の前にアスタリスク「＊」がついています。これは p **は整数型のポインタ変数（***p**）であるとを宣言しています。6 行目で i に 100 を代入し、7 行目で、

```
p=&i;
```

としています。i は通常の int 型（整数）の変数です。変数の前にアンパサンド（&）をつけると、それはメモリ空間のアドレスを表します。この式は**変**

数 i が格納されているメモリアドレスのラベルをポインタ p に貼り付けなさいという命令になります。次に 12 行目で

```
printf(" p = %d\n", *p);
```

としています。ここではポインタ変数にアスタリスクをつけて、その値[7]を参照しています。ここが混乱の元となりやすいのですが、

- 宣言でアスタリスク → ポインタであると宣言している
- 式の中でアスタリスク → ポインタが指している先の値を参照

です。11 行では、

```
 printf("  &i = %p\n", &i);
```

としています。変数の前にアンパサンド（&）をつけると、それはメモリ空間のアドレスを表すので、ここでは変数 i のメモリアドレスを% p 型のメモリアドレスに特化したフォーマットで印字しなさい、という意味です。13 行では、

```
 printf("  p= %p \n", p);
```

としていて、ポインタ p の指し示しているメモリアドレスを印字しています。予め 7 行目で、

```
 p=&i
```

としていたので、変数 i のアドレス（& i）と整数型ポインタの指し示すアドレス（p）はどちらも同じ値となります。14 行目でポインタ変数自体のアドレス（& p）を印字します。15 行目で i に 300 を代入します。ポインタ p は i を指しているため、次のループで印字するときは p の値も i の値と共に変わっています。

[7] ポインタが指しているメモリ空間にある値。

C 言語の基本的解説はここまでにして、その他の機能は実際のプログラムを通して必要に応じて学んでいくことにします。

4　シミュレーション

4.1　乱数

コンピュータシミュレーションでは様々な場面で乱数を使用します。ここでは乱数を発生させるコードを学びます[8]。

ソースコード 12: 乱数を発生させる

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int main(void){
6    int i;
7
8    printf("rand␣max␣=␣%d\n",RAND_MAX);
9    srand((unsigned int)time(NULL));
10
11   for (i = 0; i < 10; i++) {
12     printf("%d\n", rand());
13   }
14
15   return 0;
16 }
```

解説 2 行目の **stdlib.h** は、12 行目の乱数発生関数 **rand()** が定義されているヘッダファイルです。8 行目で発生させる乱数の最大値 **RAND_MAX** を出力させています。**RAND_MAX** の値は予め決められていますが、システムやコンパイラにより異なります。9 行目で **srand()** 関数を呼び出し、**乱数の種**（seed random）を設定しています。

C 言語が提供している乱数の関数は、「乱数の種」からあるアルゴリズムに従って乱数を発生させています[9]。ここでは乱数の種を設定する関数 **srand**

[8] C 言語でのデフォルトの乱数は実は周期性があり，「乱数」としては性質が良くない。従って論文などには使えない。信頼の置けるシミュレーションのためにはメルセンヌ・ツイスタ (MT) などの信頼の置ける乱数の使用が必須である。GSL を用いた MT の導入例は 81 ページを参照のこと。

[9] 乱数の種が同じ場合、毎回、乱数が同じ順で発生してしまいます。

の引数に 1970 年 1 月 1 日 0 時 0 分 0 秒からの秒数を返してくれる関数 **time** を設定し、それを乱数の種としました。

```
srand((unsigned int)time(NULL));
```

これで実行時の度に異なる乱数の種が設定されます。

time 関数の前の **(unsigned int)** はキャストで、その右側の **time(NULL)** の返し値を符合の無い整数型の数に変換します。

関数 **time** は **time_t** 型のポインタ[10]を引数として、引数にも戻り値にも同じ 1970 年からの秒数が設定されます。しかし同じ値は戻り値にも設定されます。そのため必要の無い引数のポインタを無効化するためにヌルポインタ **NULL** を指定しています。

11 行から 13 行で、乱数を 10 個発生させて表示しています。

練習 上のソースコードを書き換えて、1 から 6 までのサイコロの目をランダムに 10 個表示するコードを書きなさい。
(ヒント)　余りを計算する演算子%を使う。

練習 サイコロの一つの目、例えば 1 が出る確率は $\frac{1}{6}$ である。このプログラムでサイコロを 100 回、1000 回振ったとき、1 の目の出る確率はいくらになるか調べるコードを書きなさい。回数を多くすれば多くするほど $\frac{1}{6}$ になるだろうか？

4.1.1 時間

time 関数の引数と戻り値について調べます。

ソースコード 13: 時間の表示

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <time.h>
4
5 int main(void) {
6   time_t now;
7   struct tm *ts;
```

[10]ポインタ：メモリ番地を収めた変数。

```
8
9   while(1){
10    time(&now);
11    ts = localtime(&now);
12
13    printf("␣Year␣␣␣Month␣␣␣Day␣␣(Hour␣:␣Min␣:␣Sec)␣\n");
14    printf("␣%d␣␣␣␣␣␣␣%d␣␣␣␣%d␣␣(␣␣%d␣:␣␣%d␣:␣␣%d)␣\n",
15     ts->tm_year + 1900 , ts->tm_mon + 1 ,
16     ts->tm_mday , ts->tm_hour ,
17     ts->tm_min , ts->tm_sec);
18    sleep(5);
19  }
20  return 0;
21 }
```

解説 6行目で**time_t**型の変数nowを宣言しています。10行目ではnowをポインタとして関数timeに渡し、メモリ番地&nowに時間の値をセットしています。

```
time(&now)
```

7行目では**tm**型の構造体ポインタ***ts**を宣言しています。**ts**は**tm**型の構造を持っているので、メンバ変数の**tm_year**などの「引き出し」を内部に持っています。11行目で**localtime**の戻り値がポインタ**ts**にセットされます。このときにメンバ変数それぞれの引き出しに値が同時にセットされます。15行目から17行目で構造体tsの中のメンバ（引き出し）に矢印記号 $->$（アロー演算子）でアクセスしています。

```
ts->tm_year + 1900 , ts->tm_mon + 1 , ts->tm_mday ,
ts->tm_hour , ts->tm_min , ts->tm_sec
```

9行から19行はwhileループのブロックです。9行と、18行、19行を抜き出して書きます。

```
while(1){
......
sleep(5);
}
```

これは while の次のカッコ内が成立している間、そのブロックを実行しなさいという命令です。今の場合、カッコ内は数字の 1 ですので c 言語では「常に成立」している状態に相当します。最後に **sleep(5)** として、5 秒間、停止するように命令しています。ですので、このプログラムを実行すると 5 秒おきに時間表示が切り替わる無限ループが実現されてしまいます。プログラムを途中で停止させるためには CTL と c のキーを同時に押してください。

以下は strftime 関数を使って得られた時間を任意のフォーマットに従う文字列に変換する例です。

ソースコード 14: 時間の表示、その 2

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <time.h>
4
5  int main(void)
6  {
7      time_t      now;
8      struct tm   *ts;
9      char        buf[80];
10
11     while(1) {
12       now = time(NULL);
13       ts = localtime(&now);
14       strftime(buf, sizeof(buf),
15           "%a␣%Y-%m-%d␣%H:%M:%S␣%Z", ts);
16       printf("%s\n", buf);
17       sleep(5);
18     }
19
```

```
20      return 0;
21  }
```

解説 8行目でキャラ型（文字型）の変数bufを用意しています。ここでは**buf[80]**なので0番から79番までの引き出しに一文字ずつ入れることのできる変数が生成されます。14行目で変数**buf**に、

```
"%a %Y-%m-%d %H:%M:%S %Z"
```

とあるのは、strftime関数に特有の記号を使って出力のフォーマットを指定していて、

```
"曜日 年-月-日 時間: 分: 秒 タイムゾーン"
```

を意味しています。15行目でbufの文字の内容を%s型のフォーマット（string、文字列）として画面に表示させます。

コンピュータ上で得られる乱数は**乱数の種**を元とし、あるアルゴリズムに従って系統的に発生させます。このとき**乱数の種**が同じであれば常に同じ系列の乱数列が発生してしまいます。つまり乱数を使ったコンピュータシミュレーションを何度実行しても全く同じ乱数系列を元とするので、全く同じ結果となってしまいます。通常、乱数を使ったシミュレーションでは、実行時に得られる時間やプロセス毎に固有に割り当てられる番号を**乱数の種**として指定し，毎回異なった系列の乱数を発生させることで乱雑さを実現しています。

4.2 グラフィックス

科学技術計算の結果を可視化すると理解の助けとなり，大変有用である場合が多い。ここではC言語から直接コンピュータグラフィックスを操作する手順を学びます。ただしLinux上のグラフィックス環境であるX11をそのまま用いるのは非常に大変ですので、X11の環境を整えてくれるEGGX(えっぐえっくす)を利用することにします。

ソースコード 15: 円を描く

```
1  #include <eggx.h>
2
3  int main() {
4      int win;
5      win = gopen(640,400);
6      circle(win, 280, 180, 110, 110);
7      drawstr(win,280,180,22,0,"␣ZERO");
8
9      newcolor(win, "Green");
10     circle(win, 350, 350, 110, 110);
11
12     ggetch();
13     gclose(win);
14     return 0;
15 }
```

解説 1行目でヘッダファイルeggx.hを読み込み、EGGX/ProCALL環境を整えます。5行目の

```
win = gopen(640,400);
```

は、左下座標が$(0,0)$、右上が$(640,400)$である矩形のグラフィックスウインドウを作り、それを変数winに接続することを意味しています。6行目の

```
circle(win, 280, 180, 110, 110);
```

で、**win** ウインドウに、座標 (280.180) を中心とした、半径 110（横方向半径 110、縦方向半径 110）の円を描きます。この例では、**drawstr** で文字を書き込み、**newcolor** で色を Green に変えています。コンパイルには、コンパイルコマンド egg を用います。アプリケーションは接尾辞の.c を除いたものが自動で作成されます。プログラムを circle.c とした場合のコンパイルと実行例を示します。

コンパイルと実行

```
egg  circle.c
./circle
```

4.3　ビュフォンの針

4.3.1　アルゴリズム

フランスの自然科学者ビュフォン（Comte de Buffon）は、平行線の上に針を落とし、πの推定値を求める方法を考案しました。ここでは、モンテカルロ法により**ビュフォンの針問題**（Buffon's needle problem）をシミュレートするプログラムを作成し，併せてコンピュータグラフィックスにも挑戦します [1, 2]。

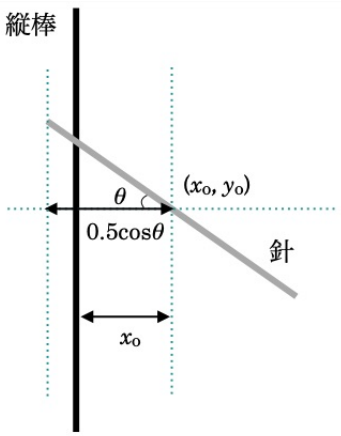

図 1: 縦棒と針の関係

まず初めに幅 1 だけ隔てて互いに平行な縦線を多数引きます。上から長さが 1 の細い針を多数落とします。今、縦棒を $x = 0$ の y 軸と重なる直線、針の中心位置を (x_0, y_0)、水平線からの針の傾きを θ とします（図 1）。針が縦棒と交わるのは、針の中心位置から縦棒までの距離 x_0 が、$0.5\cos(\theta)$ より小さいときです。

独立な確率の変域としては $0 \leq x_0 < \frac{1}{2}$ と、$0 \leq \theta < \frac{\pi}{2}$ で囲まれた矩形領域のみ考えれば良く、その面積は

$$\frac{1}{2} \times \frac{\pi}{2} = \frac{\pi}{4} \tag{1}$$

となります。その中で針が縦線と重なる場合は、θ を 0 から $\frac{\pi}{2}$ まで変化させたときに x_0 が、$0.5\cos(\theta)$ と同じか，より小さいときです。つまり関数の値以下の領域です。2 次元直交グラフで考えると，縦軸に針の中心と縦棒との距離，横軸に θ をとったとき，全変域が矩形領域の $\frac{\pi}{4}$，そのなかで縦棒と交わるのは $0.5\cos(\theta)$ 以下の領域です。その面積は θ に関しての積分で，

$$\int_0^{\frac{\pi}{2}} 0.5\cos(\theta)d\theta = [0.5\sin(\theta)]_0^{\frac{\pi}{2}} = 0.5 \tag{2}$$

と求めることができます。従って針が縦線と重なる確率は $P = \frac{0.5}{\frac{\pi}{4}} = \frac{1}{2} \times \frac{4}{\pi} = \frac{2}{\pi}$ となります。今、多数回試行して針が縦線と交わる率を求め、それを P とすると、逆算して $\frac{2}{P}$ がシミュレーションによって得られた π の値です。

4.3.2　プログラム

サンプルプログラムを下記に示します。

ソースコード 16: ビュフォンの針

```
1  #include <eggx.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #include <time.h>
6
7  #define NUMBER   1000 // number of needles
8  #define X_MAX    10
9  #define Y_MAX    5
10 #define PI       3.1415927  // we use PI!
11
12 int main(void){
13   int    j, dx;
14   int    n1 = 0;          // cross time
15   double x0, y0;          // coordinates
16   double x1, y1, x2, y2;
17   double theta;
18   int    win;
19
20   win = gopen(X_MAX*100+100,Y_MAX*100+50);
21   newlinewidth(win, 3);
22   newcolor(win, "Green");
23   for(dx=1; dx <= X_MAX; dx++){
24     drawline(win, dx*100, 0, dx*100, Y_MAX*100+50);
25   }
26
27   newlinewidth(win, 1);
28   newcolor(win, "Gray");
29   srand((unsigned int)time(NULL));
30
31   for (j = 1; j <= NUMBER; j ++) {
32     x0 = (X_MAX * (double)rand()/RAND_MAX)*100+50;
33     y0 = (Y_MAX * (double)rand()/RAND_MAX)*100+25;
34     theta = 2 * PI * (double)rand()/RAND_MAX;
```

```
35      x1 = x0 + (cos(theta) / 2)*100;
36      y1 = y0 + (sin(theta) / 2)*100;
37      x2 = x0 - (cos(theta) / 2)*100;
38      y2 = y0 - (sin(theta) / 2)*100;
39      drawline(win,  x1, y1, x2, y2);
40      if ((int)(x1/100) != (int)(x2/100)) {
41        n1 = n1 + 1;
42      }
43    }
44    printf("pi␣=␣%10f␣\n", (double)j /n1 * 2);
45
46    ggetch();
47    gclose(win);
48    return 0;
49  }
```

解説 7行目から10行目の#defineはマクロを定義しています。たとえば7行目の

```
#define NUMBER 1000
```

は、プログラム本文中に語句の **NUMBER** が出てくると、機械的にその右辺値の **1000** に置き換えなさい、という命令です。ここでは針の個数1000本を定義しています。π を求めるプログラムなのに，10行目で **PI** を定義しているのは少し変に感じるかもしれません。しかしこれはラジアン単位で針の傾き θ を与え、そこから針の始点と終点の座標を計算するためのものなので必要なのです[11]。

20行目はウインドウを作成する命令です。ここではX_MAXとY_MAXのそれぞれ‘100倍プラスアルファの領域を確保しています。以下、座標系はすべて100倍してウインドウの座標系に変換して描画をします。23行から25行までは10本の縦線をウインドウのx座標で100毎に描く命令です。

[11]乱数の値の範囲が決まっているが，全角度の360°=2π radにおいて一様かつランダムな角度である必要があることから来る（プログラム34行 **theta** の定義）。

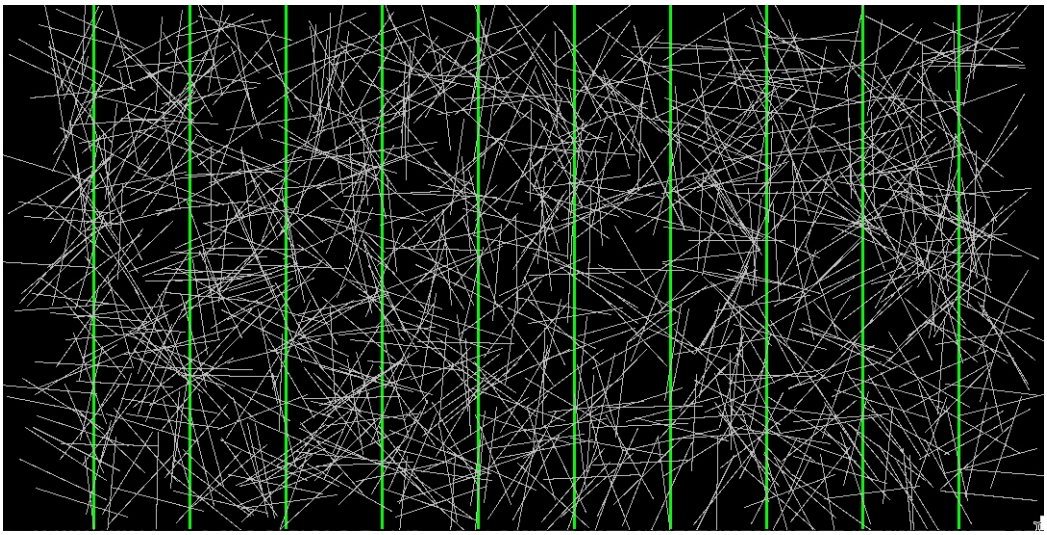

図 2: ビュフォンの針のシミュレーション。緑の縦棒を横切った白い針の数をカウントする。

32 行と 33 行で針の中心座標を乱数で生成します。34 行目で針の傾き θ も乱数で与えます。針の始点 (x_1,y_1) と終点（x_2,y_2）を計算して、39 行目で針を描画します。39 行目では、x_1 と x_2 が縦線をまたがっているかどうかを判別しています。0 から 1000 までの x スケールで 100 毎に縦棒があるので、x_1 と x_2 にを $\frac{1}{100}$ をかけて 0 から 10 までの整数とし、それが異なる値をとる場合に縦棒をまたがっていると判定し、41 行目でカウンタを 1 つ増やします。

練習 プログラムでは 1000 回の試行により π の推定値を求めています。100 回、2000 回、3000 回の試行ではどうなるだろうか。10000 回試行すれば値は π に近づくだろうか。試行回数によりある値（π？）に近づくかどうか数値の表、例えば、

```
100    3.324
200    3.312
300    3.032
..........
```

のようにファイルに出力するためには、ソースコードをどのように変更したら良いだろうか。

（ヒント）7行目で指定した試行回数 NUMBER を 1000000 などの大きな値にしておく。また試行回数と平均値は printf を使って印字するようにする。これによる端末への標準出力は、リダイレクション「>」を使ってファイルに保存できる。例えば buffon の出力をファイル out.dat に保存したい場合、

```
./buffon > out.dat
```

とする。これをグラフ作成ソフトの **gnuplot** で見るためには、

```
% gnuplot ↵
gnuplot > plot “out.dat” using 1:2 w l
```

などとします。

練習 半径 1 の円の $\frac{1}{4}$ の領域がちょうどおさまる、サイズが 1× 1 の四角い領域を考える。x 座標と y 座標を 0 から 1 までの実数の乱数で生成し、その点が円内に含まれるかどうかを判定する。この試行を多数回繰り返すことで π を推定することができる。どのようにプログラムしたら良いだろうか。

メモ この節では EGGX/ProCALL のグラフィックス環境を利用するため、コマンド *egg* をつかってコンパイルしていました。グラフィックス環境が必要無い場合、コードから、

```
#include <eggx.h>
...
gopen(X_MAX*100+100,Y_MAX*100+50);
...
drawline(win, xl*100, 0, xl*100, Y_MAX*100+50);
```

などの EGGX/ProCALL に係わる部分を消去すると実行速度が向上します。ただし通常の C のプログラムとしてコンパイルするため、数学関数を利用するときに注意が必要です。具体的にはソースコードの先頭で、

```
#include <math.h>
```

などとして数学関数が定義されたヘッダファイル **math.h** を読み込み、コンパイルは、

```
gcc  test.c  -lm  -o test
```

などとします。**-lm**（ハイフン・エル・エム）は、数学ライブラリをリンクする命令で、これを指定することによって三角関数などの数学関数を使えるようになります。**-o test** は、最終的にできあがるアプリケーションの名前を指定しています。実行するには、

```
./test
```

とします。この指定が無い場合はデフォルトの名前 **a.out** でアプリケーションが生成されます。

4.4　イジングモデル

4.4.1　アルゴリズム

電子はスピンという物理量を持っています。物質を構成する原子の外核電子が揃った状態が「磁性」を示す源です。隣同士の原子に属する最外殻電子のスピンが揃う傾向にあるとき、物質全体として磁性を示します。一次元の格子上に並んだ原子を考え、i（j）原子に属する外核電子のスピンを S_i（S_j）とします。ここでスピン S は、アップ $(S=1)$ またはダウン（$S=-1$）のいずれかの値を取るものとします。隣同士のスピンが相互作用する強さ（スピン相互作用結合定数）を J と書くとき、この系の全エネルギー（ハミルトニアン）H は、

$$H = -\sum_{<i,j>} JS_iS_j \tag{3}$$

となります。ただし $\sum_{<i,j>}$ は、隣り合う i, j のサイト間でのみ和をとるものとし、簡単のため周期的境界条件[12]を課します。

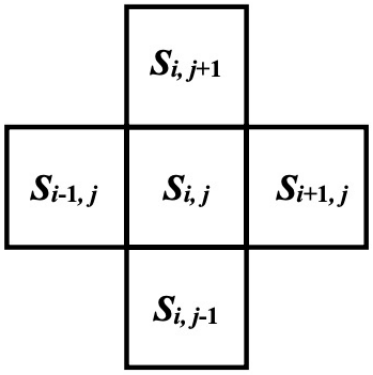

図 3: 最隣接サイトの定義

全エネルギーが低ければ低いほど系は安定なので、$J>0$ であれば、S_i と S_j は同じ値を取った方が H の値が低下し安定します。このとき系は自ら全てのスピンが同一方向に揃った**強磁性状態**を選びます。一方、J が負ならば、S_i と S_j は異なる値、ここでは 1 と -1 の組を取った方が H の値が低下し安定する。このとき系は隣同士のスピンの値が異なる**反強磁性状態**を選びます。このように格子状にスピンを配置し、ハミルトニアンに従ってスピンの状態をシミュレートする系をイジングモデル（Ising model）と言います [3, 4]。スピン相互作用結合定数 J は、その物質が強磁性体になるのか、あるいはそうはならないかを定める定数に相当し、物質によって異なる値を取ります。外部磁場 B がある場合は、式（3）に外部磁場の項を加え、

$$H = -\sum_{<i,j>} JS_iS_j - B\sum_i S_i \tag{4}$$

とします。正の外部磁場が加えられたとき、その方向に正のスピン S_i が向いた方が全エネルギーが低下します（磁化される）。

[12]格子からはみ出た部分は反対側から入ってくる。

ここでは二次元正方格子上に並んだスピンを考えます。最隣接サイト間の相互作用の和は、二次元に拡張されます。系が温度 T にあるときに、ある状態 m が実現する確率 P_m は、**正準分布**

$$P_m = \frac{\exp(-\frac{E_m}{k_B T})}{Z} \tag{5}$$

で与えられる。ここで k_B はボルツマン定数であり、状態和 Z は、

$$Z = \sum_j \exp(-\frac{E_j}{k_B T}) \tag{6}$$

です。今、(i, j) サイトのスピン $S_{i,j}$ に注目し、この系の状態を P_m で表します。スピン $S_{i,j}$ のみを反転させた状態 n が実現する確率は、P_n であり、$P_n > P_m$ であれば、新しい状態 n が採用される可能性が高くなります。これをシミュレートするため両辺を P_m で割り、右辺の定数 1 の代わりに 0 から 1 までの乱数 RND を導入します。

$$\frac{P_n}{P_m} > \mathrm{RND} \tag{7}$$

このときにスピン $S_{i,j}$ を反転させます。ここで確率の比 $\frac{P_n}{P_m}$ は、

$$\exp(-\frac{\Delta E}{k_B T}) = \exp(-\frac{2S_{i,j}(Jg + B)}{k_B T}) \tag{8}$$

です。ただし、

$$g = S_{i+1,j} + S_{i-1,j} + S_{i,j+1} + S_{i,j-1} \tag{9}$$

としました。式（8）を計算し、乱数と比較して次々とサイトの状態を更新していくことで、その温度 T と外部磁場 B、スピン間相互作用の強さ J に依存した状態が得られます。せっかくなので，信頼できる乱数発生器として GSL のメルセンヌツイスタを使用することにします（第 4.11.1 節を参照）。

4.4.2　プログラム

サンプルコードは GitHub に登録されています。Code メニューから Download ZIP を選択するとコードを ZIP 形式でダウンロードできます。あるいは Code メニューから HTTPS アドレスをコピーして，下記の様に git でクローンを作ることもできます。

```
$  git clone https://github.com/KazumeNishidate/Ising.git
```

プログラムを以下に示します。

ソースコード 17: 2次元イジングモデルのコード ising.c

```
1  #include <eggx.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #include <time.h>
6  #include <gsl/gsl_rng.h>
7
8  #define X_MAX 50
9  #define Y_MAX 50
10 #define STEPS 100000
11 #define JKT   0.2   // J/kT
12 #define BKT   0.0   // B/kT
13
14 double ran1(void);
15
16 int main(void){
17   int i, iu, id, j, ju, jd, k, sig;
18   int mat[X_MAX][Y_MAX];
19   int win;
20   double de, pp;
21
22   win = gopen(X_MAX*10,Y_MAX*10);
23   layer(win,0,1);
24
25   for(i=0;i<X_MAX;i++){  // initial state
26     for(j=0;j<Y_MAX;j++){
27       if(ran1()>0.5){
28         mat[i][j]=1;  // up spin
29         newcolor(win, "Yellow");
```

```
30          fillrect(win,i*10,j*10,10,10);
31        }else{
32          mat[i][j]=-1; // down spin
33        }
34      }
35    }
36
37    for(k=0;k<STEPS;k++){
38      i = (int)(X_MAX*ran1());
39      j = (int)(Y_MAX*ran1());
40      iu = i+1; ju = j+1;
41      id = i-1; jd = j-1;
42      if(id<0) id=X_MAX-1; if(iu==X_MAX) iu=0;
43      if(jd<0) jd=Y_MAX-1; if(ju==Y_MAX) ju=0;
44
45      sig = mat[id][j]+mat[iu][j]+mat[i][jd]+mat[i][ju];
46      de  = exp( -2.0*mat[i][j]*(sig*JKT + BKT) );
47      pp  = ran1();
48
49      if(pp<de) mat[i][j] *= -1;  // spin flip
50
51      if(mat[i][j]==1){
52        newcolor(win, "Yellow");
53        fillrect(win,i*10,j*10,10,10);
54      }else {
55        newcolor(win, "Black");
56        fillrect(win,i*10,j*10,10,10);
57      }
58      copylayer(win,1,0);
59    }
60    getchar();
61    gclose(win);
62    return 0;
63  }
64
```

```
65 double ran1(){
66   static const gsl_rng_type *T;
67   static gsl_rng *r;
68   static int cnt=0;
69
70   if(cnt==0){
71     gsl_rng_env_setup ();
72     T = gsl_rng_default;
73     r = gsl_rng_alloc (T);
74     gsl_rng_set(r,time(NULL));
75     cnt =1;
76   }
77   return gsl_rng_uniform(r);
78 }
```

解説　2次元イジングモデルのアルゴリズム。

1. 14行目で main 関数で使用する関数 ran1 の型を宣言します。実際の ran1 関数は 65 行から 78 行で定義します。
2. 38行と39行：サイト (i, j) をランダムに選ぶ。38行から41行で周期的境界条件を課します。
3. 45行目と46行目：もとの状態と、スピン $S_{i,j}$ を反転させた状態のエネルギー差 de を求めます。
4. 47行目：0から1までの乱数 RND を発生させます。
5. 49行目：式（7）を評価する。成立すればスピン $S_{i,j}$ を反転させた状態を採用する。ここで***= -1** は、左辺の値（mat[i][j]）に-1を掛けた値を新しい左辺の値 mat[i][j] とすることを意味しています[13]。

以上を STEPS 回繰り返します（37行目の for 文）。

コンパイルは以下のようにします。

[13]例えば/=は左辺を右辺で割って左辺の値に代入します。+=および-=なども同様の操作。

```
$ egg ising.c -lgsl -lgslcblas
```

-lgsl は，GSL のライブラリをリンクするオプションで，GSL の関数を利用する時は必要となります。**-lgslcblas** は GSL で用意している CBLAS ライブラリをリンクするオプションです[14]。上記のコンパイル法と同等ですが，egg ではなく gcc で直接コンパイルする場合は，

```
$ gcc ising.c -o ising  -leggx -lX11 -lgsl -lgslcblas -lm
```

とします。実行は，

```
$ ./ising
```

とします。図 4 にイジングモデルの途中経過のスナップショットを示します。デフォルトの設定では，$JKT = 0.8$ と大きな値にしています。ここで JKT は，式 8 での exp の中にある $\frac{J}{k_B T}$ ですからスピン間の結合が強い（J が大きい）あるいは温度が低い（T が小さい）状況に相当します。スピン間で同じ方向を向こうとする傾向が高くなり，かつ温度が低いため，計算を進めるうちにアップスピンの領域とダウンスピンの領域ができていきます。

練習 温度が高いときはどうなるだろうか。

練習 外部磁場 B があるときはどうなるだろうか。

4.4.3　高速化

次に、このコードを高速化することを考えます。式 (4) の第一項は、サイト (i, j) に注目すると、$S_{i,j}$ サイトと上下左右の 4 サイトとの J を通した相互作用の「和」を数え上げることに相当する（図 3）。和はスピンの値によって、± 4, ± 2, 0 の計 5 種類の値が考えられる。そのようなスピンの組の例を表 2 に示します[15]。プログ

[14] GSL では, BLAS の部分をより高性能な外部ライブラリに置き換えて利用できるように, CBLAS を別に明示してリンクするようにしている [5]。

[15] この他にも、$S_{i,j+1}$ と $S_{i,j-1}$ のみが -1 の場合など、同じ値を与える様々な組み合わせがあります。

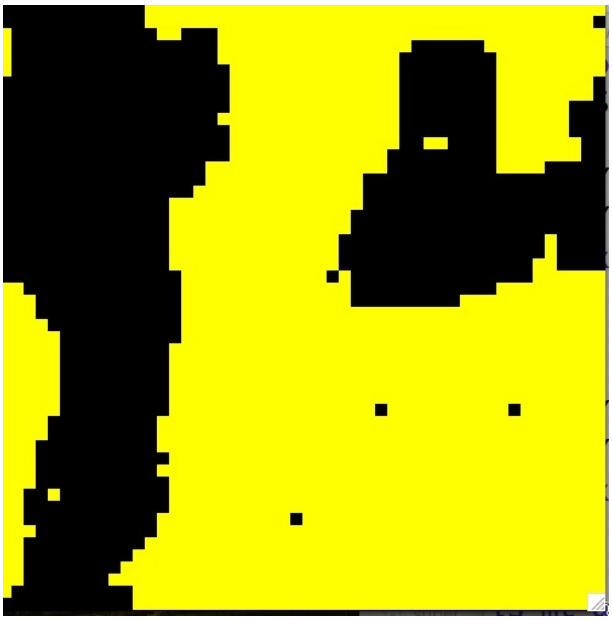

図 4: イジングモデルの途中経過のスナップショット。

ラムでは 43 行目で -mat[i][j]*sig の項がでてきて、それを使って exp の計算をしています。この部分を予め計算して表として用意しておき、for ループの中ではその表を参照するようにすれば、いちいち exp を計算しなくてもすむのでトータルとしての計算量の削減が期待できます。

表 2: $(i,\ j)$ サイトとその最隣接サイト $(i',\ j')$ のスピンの値、及びその和の例。$S_{i,j} = -1$ の場合も含めると計 10 種類の組み合わせがあるが、$-S_{i,j}g$ の値としては $\pm 4,\ \pm 2,\ 0$ に限られます。

$S_{i,j}$	$S_{i,j+1}$	$S_{i,j-1}$	$S_{i+1,j}$	$S_{i-1,j}$	$-S_{i,j}g$
± 1	1	1	1	1	± 4
± 1	1	1	1	-1	± 2
± 1	1	1	-1	-1	0
± 1	1	-1	-1	-1	∓ 2
± 1	-1	-1	-1	-1	∓ 4

練習　高速化したプログラムを作成してみよう。k に関するループに入る前に、1 度だけ以下を計算します。

```
for(i=0;i<5;i++){
  sig = (i-2)*2.0;
  ee[i][0] = exp( -2.0*(-1.0)*(sig*JKT + BKT) );
  ee[i][1] = exp( -2.0*( 1.0)*(sig*JKT + BKT) );
}
```

ここで ee は値を収めておくテーブルで、

```
double ee[5][2];
```

などとして宣言しておきます。de の読み取りは、例えば、

```
  ijspin = (mat[i][j]+1)/2.0;
  de  = ee[sig][ijspin];
```

とすれば良い。ここで $S_{i,j}$ がダウンスピンのときは ijspin=0、アップスピンの時は ijspin=1 です。

アプリケーションの実行時間は **time** コマンドを使うことで得ることができます。

```
% time ./ising ↵
```

コマンドの返し値は、real, user, sys で、それぞれ、

real コマンドを実行するためにかかった時間

user コマンドを実行するためにかかったユーザー CPU 時間

sys コマンドを実行するためにかかったシステム CPU 時間

を意味しています。実行にあたってはプログラムから **getch();** を削除もしくはコメントアウトしてコンパイルし直すように[16]。プログラムのシステムサイズや繰返し回数を変化させて、どれくらい性能が向上するか確かめなさい。

[16] ggetch() の目的は、計算終了後もプログラムがアイドル状態になって画像表示を継続するため。

練習 この系を特徴づける量に**平均磁化率**があります。各サイトのスピンの和を数え上げて、その絶対値をサイトの総数で割る。例えば次の関数 **calc_m** は、平均磁化率を計算して返す関数の例です。

```
double calc_m(int xmax,int ymax,int mm[xmax][ymax]) {
  int i, j;
  double mag=0.0;

  for(i=0;i<xmax;i++){
    for(j=0;j<ymax;j++){
      mag +=mm[i][j];
    }
  }
  return fabs(mag/(xmax*ymax));
}
```

これを **main** 関数の前に置く[17]。**main** 関数の中では、全ての計算が終わってから

```
    av_m=calc_m(X_MAX, Y_MAX, mat);
    printf(" %f  %f \n",JKT,av_m);
```

で呼び出せば良い。ここで av_m、JKT は double の変数とした[18]。たとえば JKT を 0.0 から 1.0 まで 0.05 ずつ変化させて平均磁化率の変化を見ることができる。図 5 に計算結果の例を示します。このときの計算パラメータは、

```
#define X_MAX 10
#define Y_MAX 10
#define STEPS 2000
#define BKT   0.0    /* B/kT */
```

[17] main が始まる前に関数の型宣言をやる必要があるため。
[18] JKT を define の行から削除し、新たに変数宣言する必要がある。

です。グラフを見ると、JKT=0.4 付近で平均磁化率が急激に立ち上がり、1 になる様子がわかります。JKT に依存した 2 次元イジング系のマクロな状態の変化は相転移として知られているものです [4]。このグラフを再現しなさい。系の大きさを変えると相転移はどうなるだろうか。もっとスムーズなグラフの変化を得るためにはどうしたら良いだろうか。

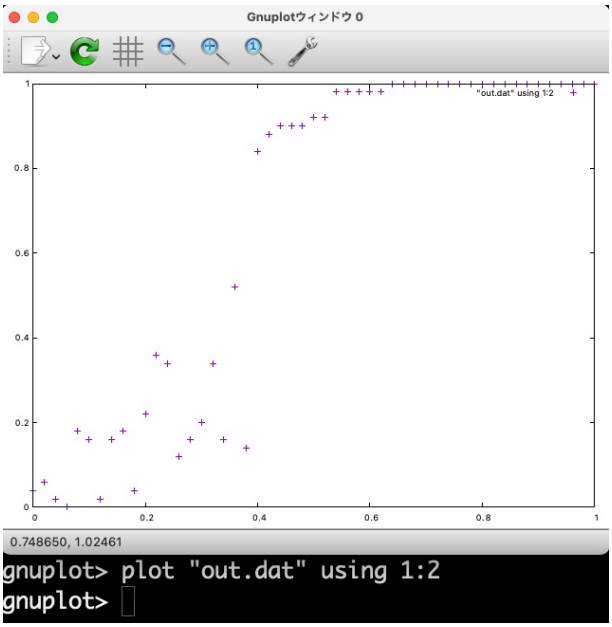

図 5: JKT を変化させたときの平均磁化率

4.4.4 1 次元配列のポインタ

上の例では 2 次元配列のサイズがソースコードの#define で指定されていました。そのためシミュレーションの規模を大きくするために配列のサイズを変更したいときは、#define 文を書き換え、いちいちプログラムのコンパイルをやり直す必要があります。しかしながら実行時に必要なサイズの配列を確保することができれば、そのような無駄な作業は省くことができます。これを実現する仕組みが**動的メモリ確保**です。初めに一次元配列のポインタを考えます。

ソースコード 18: 1 次元配列のポインタ

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char* argv[]){
5    int i, mm, *mat;
6    int cnt=0;
7
8    mm =  atoi(argv[1]);
9    mat =  (int *)calloc(mm, sizeof(int));
10
11   printf("mm␣=␣%d\n", mm);
12   for(i=0;i<mm;i++){
13     cnt++;
14     mat[i] = cnt;
15   }
16
17   for(i=0;i<mm;i++){
18     printf("␣%3d", mat[i]);
19   }
20   printf("\n");
21
22   free(mat);
23   return 0;
24 }
```

解説 4 行目で関数 main に引数を指定しています。この引数はプログラム実行時

に与える値です。argc は引数の個数、argv[] は char 型の配列で、その引数の実体が入ります。たとえば、a.out の実行時に、

```
./a.out 8
mm = 8
   1   2   3   4   5   6   7   8
```

と実行すると、argc は 1、argv[1] には char 型で”8” の**文字**が入る。これを関数 atoi() で整数型に変換し、整数型変数の mm に代入している。5 行目で整数型ポインタ mat を宣言し、8 行目で calloc を使って**動的メモリ割り当て**をしている[19]。割り当てる量は、整数型（sizeof(int)）で mm 個の入れ物。12 行から 15 行までで mat1 に数値を代入し、17 行から 19 行までで中身を印字、22 行目で free() 関数によってメモリを解放する。

4.4.5 2 次元配列のポインタ

2 次元以上の配列へのポインタも同様の手続きで実現できる。[20]。

ソースコード 19: 2 次元配列のポインタ

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char* argv[]){
5    int i, j, mm, nn, *mat;
6    int cnt=0;
7
8    mm =  atoi(argv[1]);
9    nn =  atoi(argv[2]);
10   mat =  (int *)calloc(mm*nn, sizeof(int));
11
12   printf("mm␣=␣%d␣␣nn␣=␣%d\n",mm,nn);
13   for(i=0;i<mm;i++){
```

[19]同様の操作は malloc を使っても実現できる。calloc の利点はメモリ割り当て時に初期値をゼロクリアしてくれること。malloc は単にメモリを割り当てるだけである。

[20]多次元配列へのポインタへ応用の観点から、ここでは 2 次元配列のみを実現するためのポインタへのポインタ（例えば int **mat）は扱いません。

```
14      for(j=0;j<nn;j++){
15        cnt++;
16        mat[i*nn+j] = cnt;
17      }
18    }
19
20    for(i=0;i<mm;i++){
21      for(j=0;j<nn;j++){
22        printf("␣%3d", mat[i*nn+j]);
23      }
24      printf("\n");
25    }
26
27    free(mat);
28    return 0;
29  }
```

実行時には行数と列数の２つの引数を指定します。

```
./a.out 5 8

mm = 5  nn = 8
   1   2   3   4   5   6   7   8
   9  10  11  12  13  14  15  16
  17  18  19  20  21  22  23  24
  25  26  27  28  29  30  31  32
  33  34  35  36  37  38  39  40
```

練習 イジングのプログラム例では２次元格子のサイズがソースコードで決められていて、それを変更するためにはいちいちプログラムを再コンパイルする必要がありました。イジング系を２次元配列のポインタで表現した場合、動的メモリ確保の機能が使えるので実行時に格子サイズを指定して、その範囲でメモリサイズを確保することができます。イジング系のプログラムを２次元配列のポインタを利用するように書き換えてみよ。実行速度に違いはでるか？

4.5　マンデルブロー集合

複素数を用いる例としてマンデルブロー集合を取りあげます。次の漸化式

$$z_{n+1} = z_n^2 + c$$

を考えます。初項を $z_0 = 0$ とし，複素平面上の点 c に対して数列

$$z_1, z_2, z_3, z_4, \cdots, z_k, \cdots$$

を計算します。このとき $\lim_{k\to\infty} z_k$ が収束する複素数値 c の集合を**マンデルブロー集合**と言います。[21]ただし実際には無限大まで数列を調べることは不可能なので，$k = 100$ などの比較的高次の項で打ち切り，z_k の値が収束するかどうかを判定します。また z の絶対値が 2 を超えると数列が発散することが証明されています。そのため絶対値が 2 を超えた段階でマンデルブロー集合の検索対象から除外します。検索対象の c の初期値は複素平面上で実数が+2 から-2，虚数が$+2i$ から$-2i$ とします。これ以外の領域では初めから絶対値が 2 を超え，発散することが明らかなためです。

複素平面上の各点から始まる数列の収束と発散をしらべ，それによって色分けをしてグラフィックス表示します。画面上で原点を含む歪んだ円の中の単一色の領域が収束する点の集合で，マンデルブロー集合に属します。それを取り囲むカラフルで，複雑な構造の縁取りの部分が発散が始まる部分で，閾値の 2 を超えるまでの回数によって色分けされています。それを大きく取り囲む単一色の領域が発散領域です。境界領域での特徴的な構造はいくら拡大しても際限なく繰り返されます。樹木の枝別れや海岸の形状など，自然界には自己相似性と複雑性を併せ持つ「模様」が数多く知られています。これらは発見者のマンデルブローによってフラクタルと命名されました。

ソースコード 20: マンデルブロー集合（mandel.c）

```
1 #include <eggx.h>
2 #include <stdio.h>
3 #include <complex.h>
4 #include <math.h>
5
6 #define X_WIN  500
7 #define Y_WIN  500
8
```

[21]数学者の Benoit B. Mandelbrot が 1982 年に紹介してから有名になりました。

```
 9 #define MYCOLOR  IDL2_EOS_A
10
11 #define RE_MIN  -2.0   // RE => x-axis
12 #define RE_MAX   1.0
13 #define IM_MIN  -1.5   // IM => y-axis
14 #define IM_MAX   1.5
15 #define Z_MAX    2
16 #define IT_MAX   100
17
18 int main(void){
19   int win, cnt, nx, ny;
20   int c_r, c_g, c_b;
21   double complex z, c;
22   double im, re, d_im, d_re;
23
24   win=gopen(X_WIN,Y_WIN);
25   d_im=(double)(IM_MAX-IM_MIN)/(double)Y_WIN;
26   d_re=(double)(RE_MAX-RE_MIN)/(double)X_WIN;
27
28   for(ny=0;ny<Y_WIN-1;ny++){
29     im=IM_MIN+(double)ny*d_im;
30     for(nx=0;nx<X_WIN-1;nx++){
31       re=RE_MIN+(double)nx*d_re;
32       c = re + I*im;
33       z = 0.0+I*0.0; cnt = 0;
34       while(cabs(z)<=Z_MAX && cnt++ < IT_MAX){
35         z = z*z+c;
36       }
37       makecolor(MYCOLOR,(double)IT_MAX, 0.0, (double)cnt,
38     &c_r,&c_g,&c_b);
39       newrgbcolor(win,c_r,c_g,c_b);
40       pset(win,(double)nx,(double)ny);
41     }
42   }
43   ggetch();
```

```
44    gclose(win);
45 }
```

解説 3 行目で複素数関数を定義したヘッダファイル complex.h を読み込みます。6 行目と 7 行目の **X_WIN** と **Y_WIN** で表示ウインドウのサイズ（ここでは 500 ピクセル × 500 ピクセル）を定義します。11 行目と 12 行目で実軸の範囲 (**RE_MIN** から **RE_MAX**) を指定し，13 行目と 14 行目で虚軸の範囲 (**IM_MIN** から **IM_MAX**) を指定します。

15 行目は閾値 **Z_MAX** を定義していて，絶対値が 2 を超えると発散と判定します。16 行目は最大繰り返し数 **IT_MAX** で，この回数だけ関数を適用して発散しなければ収束と判定します。

32 行目と 33 行目では複素数である c と z の初期値をそれぞれ設定しています。複素数変数への代入は (実数値) + **I** ∗ (実数値) とし，虚部の値は **I** を掛ける形式であらわします。

34 行目から 36 行目が **while** ループで，z_k を計算します。直後のカッコ内の条件：

c の絶対値が Z_MAX 以下，かつ[22] cnt の値が IT_MAX より小さい

が成立している間，**z=z*z+c**　を繰り返し実行します。また **cnt++** としているので，この while ループの条件式が評価されるたびに変数 **cnt** の値が 1 増えます（**cnt**=k）。37 行目で **makecolor** 関数を呼び出します。発散と判定されるまでの繰り返しの回数 **cnt** を色の定義表に当てはめ，それに従って決定された赤，緑，青の数値による指定が変数**&c_r, &c_r, &c_b** に格納されます。次の **newrgbcolor** で色を確定し，**pset** でその位置を色のついたペンで塗りつぶします。

使用する色の定義表 **MY_COLOR** は 9 行目で指定しています。定義表は eggx のマニュアルのセクション 2.4.17，**makecolor** の項目で定義されています。ここでは **IDL2_EOS_A** の配色を用いました。

[22] 条件式の”&&” マークは条件式「A かつ B」の「かつ」に相当（A も B も同時に成立）。

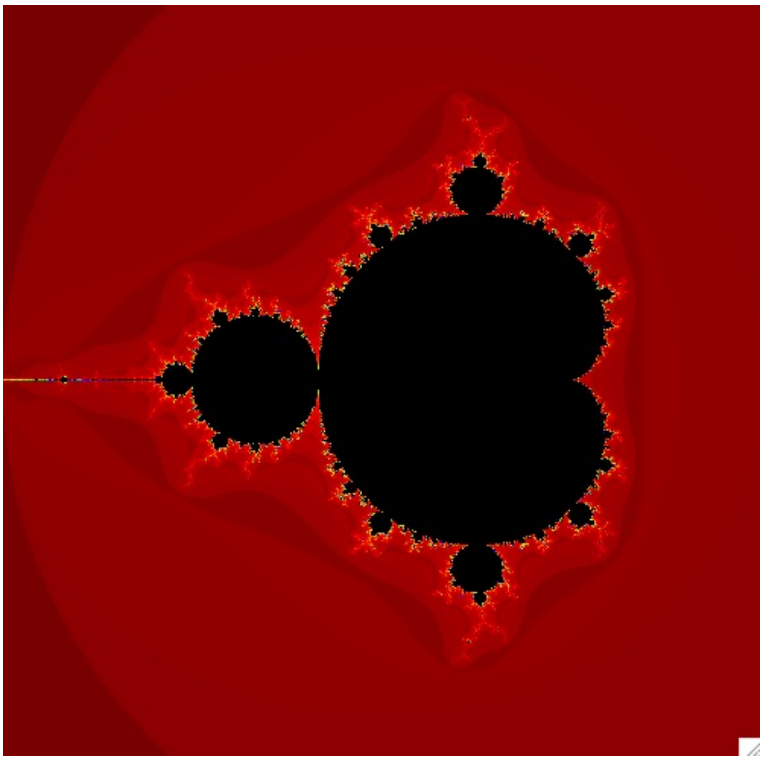

図 6: マンデルブロー集合。

練習 表示する範囲を狭めていき, どんどん画像を拡大していくと様々なパターンが見えてきます。それに加えて最大繰返し回数を変更していくと複雑なパターンが現れてくることもあります。以下に京都大学の宍倉光広氏が公開[23]しているマンデルブローと集合のグラフィックスパラメータをあげます。さらに色の定義表 **MY_COLOR** も変更すると，かなり変わった印象のグラフィックスが得られます。

[23] iDynamics で作製した図のグラフィックスギャラリーから。

表 3: マンデルブロー集合のパラメータセット。ピクセルはいずれも 400×400。

	実軸の範囲	虚軸の範囲	繰り返しの回数
set 1	-1.14 to -0.285	-0.165 to 0.69	200
set 2	-0.783447422527 to -0.730603085175	0.08517861688 to 0.138023019040	2000
set 3	-0.753013475730 to -0.734811555969	0.121201501571 to 0.139403421331	5000
set 4	-0.747254284586 to -0.740984740873	0.131736878013 to 0.138006421726	5000
set 5	-0.745366725245 to -0.744628830600	0.134808552338 to 0.135546446982	5000
set 6	-0.745056944621 to -0.745049584122	0.135208895555 to 0.135216256054	5000
set 7	-0.744646932908 to -0.744443933428	0.134651139877 to 0.134854139357	5000
set 8	-0.744556429926 to -0.744535711735	0.134745522851 to 0.134766241042	10000

練習 マンデルブロー集合の一部を拡大してみよう。新しいパターンを見つけよう。

練習 漸化式でzを三乗にするとどうなるだろうか。

4.6　ジュリア集合

歴史的にはマンデルブロー集合よりも早く，フランスの数学者 Gaston Maurice Julia が 1918 年に発表しました。同じ漸化式

$$z_{n+1} = z_n^2 + c$$

を解きます。ただし今度は c の値を設定して定数とし，z の値を複素平面上に取ります。収束と発散の境界の値をジュリア集合，収束する値を充填ジュリア集合と言います。これもグラフィックスにしたときにカラフルに色と形状が変化する部分がジュリア集合を取り囲む発散のはじまる部分です。ジュリア集合のプログラムは，マンデルブロー集合のプログラムに僅かに変更を加えるだけで実現できます。[24]

表 4: ジュリア集合のパラメータセット。ピクセルはいずれも 500×500。

	c	実軸の範囲	虚軸の範囲	繰返しの回数
set 1	-0.7+0.27015 I	-1.4 to 1.4	-1.4 to 1.4	300
set 2	-0.747593803839+0.083586811697 I	-1.6 to 1.6	-1.6 to 1.6	5000
set 3	-0.764312625000+0.132699750000 I	-1.6 to 1.6	-1.6 to 1.6	500
set 4	0.250165658404-0.000003772500 I	-1.5 to 1.5	-1.5 to 1.5	30000

練習　ジュリア集合のプログラムを作ってみよう。

練習　ジュリア集合の一部を拡大してみよう。どこまで拡大できるだろうか。

練習　c の値を変えてみよう。

[24] マンデルブロー集合の時は c の値で複素平面上を走査しましたが，ジュリア集合の場合は z の値で複素平面上を走査します。

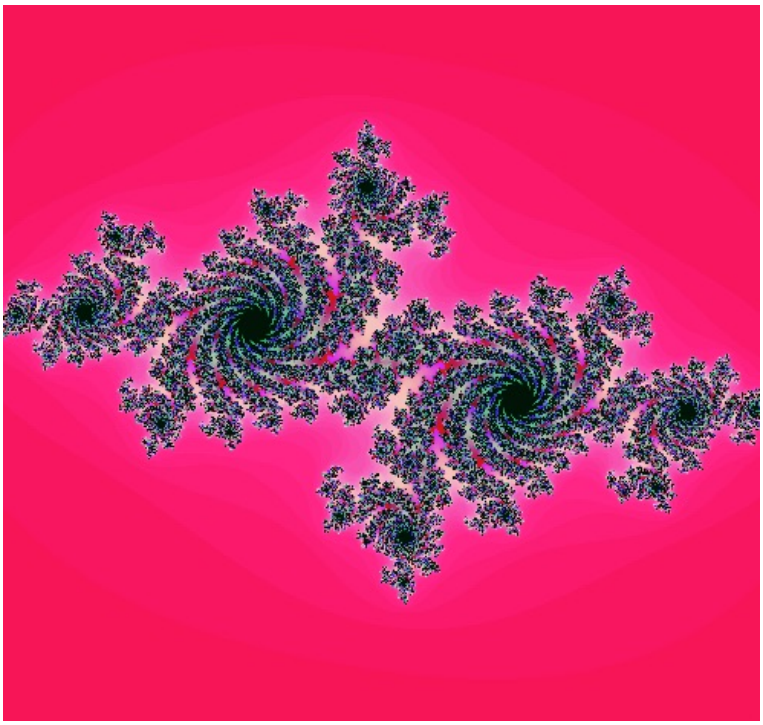

図 7: ジュリア集合。

4.7　常微分方程式

4.7.1　オイラー法

ニュートンの運動方程式は微分方程式であるため、数値解を得るためには**差分法**による数値積分を実行する必要があります。今、速度を v、位置を x としたときのニュートンの運動方程式は、

$$\frac{dv}{dt} = a(t) \tag{10}$$

$$\frac{dx}{dt} = v(t) \tag{11}$$

です。時間ステップを Δt としたとき、$n+1$ ステップ目の速度と位置はテーラー展開を用いて、

$$v_{n+1} = v_n + a_n \Delta t + O((\Delta t)^2) \tag{12}$$

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 + O((\Delta t)^3) \tag{13}$$

となります。この式において右辺の第二項までをとり、Δt に関して 2 次以上の高次の項を無視する近似をオイラー法と言います。オイラー法は多少不安定な解を与える事が知られていて、これを改良したオイラー・クロマー法（修正オイラー法)、オイラー・リチャードソン法なども用いられます。

4.7.2 落下する物体の運動

物体を斜め上方に投げたときの運動を考えます。物体が飛行中にはたらく力は鉛直下向きの重力のみとする。簡単のため重力は $-y$ 方向にはたらいているものとし、$x-y$ の二次元系で運動を議論をします。物体の重さは 1 kg、初期位置は $x=0, y=0$、初期の運動量は $p_x = 1.0$ kg m /s、$p_y = 1.0$ kg m /s、重力加速度を G=9.80665 m/s^2 とします。この物体の運動をオイラー法を使って追跡します。

ソースコード 21: 打ち上げて落下する物体の運動（eular.c）

```
1  #include <eggx.h>
2  #include <stdio.h>
3  #include <math.h>
4  #include <unistd.h>
5
6  #define X_MAX 500  // window size
7  #define Y_MAX 500
8  #define G 9.80665  // (m/s^2)
9
10 int main(void){
11   double mm=1.0;   // mass
12   double dt=0.001; // time step
13
14   double rx=0.0, ry=0.0; // initial position
15   double px=0.4, py=1.0; // initial moment
16   int win;
17
18   win = gopen(X_MAX,Y_MAX);
19   newcolor(win, "Yellow");
20   while(1){
21     rx += (px/mm)*dt;
22     ry += (py/mm)*dt;
23     px += 0.0;
24     py += -mm * G * dt;
25     if(ry < 0.0) break;
26     fillrect(win,rx*2200+20,ry*8000+10,5,5);
27     usleep(10000);
```

```
28    }
29    ggetch();
30    gclose(win);
31  }
```

21 行と 22 行で位置を更新し、23 行と 24 行で運動量を更新しています。27 行の usleep は、マイクロ秒（μsec $= 10^{-6}$sec）単位で計算を一時停止する関数です。今の場合、while ループの中で usleep(10000) と指定しているので、ループ実行時に毎回 $1.0 \times 10^4 \mu$sec 実行が中断されます。

練習 $\sqrt{p_x^2 + p_y^2}$ を一定に保ちながら、p_x と p_y の値を変えて水平方向への最長到達距離を調べなさい[25]　どのような値の時に最長となるか。

練習 空気抵抗は、経験的に速度の 1 乗と速度の 2 乗に比例する事が知られている。速度の 1 乗に比例する空気抵抗の比例係数を k_1、速度の 2 乗に比例する空気抵抗の比例係数を k_2 としたときのプログラムを書きなさい。ここで y 方向の空気抵抗は、物体が上昇中は下向きに、物体が下降中は上向きに働くことに注意すること。

[25] 例えば値を設定後、$p_x* = 1/3.0$, $p_y* = 3.0$ とすれば良い。

4.8　万有引力

4.8.1　ケプラーの法則

ニュートンは、散歩の途中、リンゴが木から落ちる様子を見ていて、ふと次のような疑問を抱いた。**リンゴは地球に向かって落ちるけれど、月は落ちてこないのだろうか？**・・・・・この疑問が万有引力の発見を導いた。これは良く知られた逸話である。しかしながら実際には観測による経験則を詳細に検討することによってニュートンはこの結論にたどり着きました。

ドイツの天文学者ヨハネス・ケプラーは、彼の師であるティコ・ブラーエの観測データを検討し、ケプラーの法則を発表しました（1609〜1618 年）[26]。

第１法則 惑星は太陽を１つの焦点とするだ円上を運動する。

第２法則 惑星と太陽とを結ぶ線分が一定時間に通過する面積は一定である（面積速度一定の法則）。

第３法則 惑星の公転周期 T の 2 乗と軌道だ円の長半径（半長軸の長さ）a の 3 乗の比は，全ての惑星で一定になる。

$$\frac{T^2}{a^3} = k \quad (\text{k は定数}) \tag{14}$$

ニュートンはケプラーの法則を詳細に検討し、そこから数学的に導き出された結論として、万有引力の法則を**発見**しました[27]。その過程で微分積分法を構築しています。ケプラーの法則は惑星と太陽の間に万有引力

$$F = G\frac{m_1 m_2}{r^2} \tag{15}$$

（G: 万有引力定数）を設定することで導くことができます。

4.8.2　惑星の運動

質量 M の太陽の周りを回る質量 m の惑星の運動を考えます。加速度 a は常に太陽を向いていて万有引力によるものです。

$$\begin{aligned} F &= ma \\ &= \frac{mv^2}{r} \\ &= \frac{GMm}{r^2} \end{aligned}$$

[26] リンク先のケプラーの法則についての解説を参照。
[27] 例えば「微積分が導いた宇宙の法則―万有引力の発見は数学の賜物」を参照

単位系として天文単位を採用しましょう[28]。

$$1\mathrm{AU}\,(\text{距離}) = 1.495978707 \times 10^{11}\mathrm{m} \tag{16}$$

時間の単位は 1 年 (3.154×10^7 秒) を 1 AU（時間）とします。このとき万有引力定数×太陽の質量は、

$$GM = 4\pi^2 = 39.4784176044 \tag{17}$$

です。

微分方程式の数値積分法として、比較的安定した解が得られるとして知られている 4 次のルンゲ・クッタ法を解説します [6, 7]。次の 1 階の微分方程式を数値積分によって解くものとします。

$$\frac{dx}{dt} = f(x,t) \tag{18}$$

ここでは x を距離，f を速度とします。4 次のルンゲ・クッタ法では 4 段階でその瞬間の速度とそれによる移動距離を順に求めます。

$$\begin{aligned} k_1 &= f_1(x_n, t_n)\Delta t \\ k_2 &= f_2(x_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2})\Delta t \\ k_3 &= f_3(x_n + \frac{k_2}{2}, t_n + \frac{\Delta t}{2})\Delta t \\ k_4 &= f_4(x_n + k_3, t_n + \Delta t)\Delta t \end{aligned}$$

これらから x_{n+1} ステップでの距離が、

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{19}$$

で与えられます。この関係の解析的導出は文献に示されています [7][29]。それぞれの段階で評価している移動距離 k_i と速度 f_i の関係を以下に示します。

- 初期位置 (x_n, t_n) での速度 f_1 を評価し，Δt 時間後の移動距離 k_1 を求める。
- 上で求めた傾き f_1 をつかって初期位置から半分だけ進み $(\frac{k_1}{2}, \frac{\Delta t}{2})$，そこでの速度 f_2 を評価する。それにより初期位置 (x_n, t_n) から Δt 時間後の移動距離 k_2 を求める。
- 上で求めた速度 f_2 をつかって初期位置から半分だけ進み $(\frac{k_2}{2}, \frac{\Delta t}{2})$，そこでの速度 f_3 を評価する。それにより初期位置 (x_n, t_n) から Δt 時間後の移動距離 k_3 を求める。

[28] 天文単位の定義。
[29] この文献の Fig.2 には図による説明がなされています。

- 上で求めた速度 f_3 をつかって初期位置から最後まで進み $(k_3, \Delta t)$，そこでの速度 f_4 を評価する。それにより初期位置 (x_n, t_n) から Δt 時間後の移動距離 k_4 を求める。

式の導出は大変ですが，なぜこうすると計算が安定するのかを理解することは比較的容易です。式 19 を良く見ると第 2 項の括弧内で，k_2 と k_3 のウエイトが $\frac{2}{6}$ となっていて，一方で第 1 項と第 4 項は $\frac{1}{6}$ と，その半分になっています。すなわち中間点で評価した第二段階と第三段階の速度のウエイトが大きくなっていて，初期位置と最終点で得られた速度のウエイトは低くなっています。ウエイトが大きいと言うことはそれだけ結果に影響を与える度合いが高いということなので，中間点で評価した速度を主に信頼するものとして移動距離を求めていることに相当します。前節のオイラー法では第一段階の速度で Δt 時間後の移動距離を算出していましたが，ルンゲクッタ法ではオイラー法に 3 段階の修正を加えているといえます。ルンゲクッタ法は一階の微分方程式に適用されるので，そのままでは加速度 a を位置の二階微分から求めることはできません。

$$\frac{d^2x}{dt^2} = a(t) \tag{20}$$

$$\tag{21}$$

位置と速度に関する一階の微分方程式に分解して，それぞれ計算します。すなわち速度を v、位置を x としたときの

$$\frac{dv}{dt} = a(t) \tag{22}$$

$$\frac{dx}{dt} = v(t) \tag{23}$$

を解きます。リストに 2 次元太陽系における仮想的な惑星の運動を表示するコードを示します。

ソースコード 22: 惑星の運動（kep.c）

```
1 #include <eggx.h>
2 #include <stdio.h>
3 #include <math.h>
4 #include <unistd.h>
5
6 #define X_MAX 400
7 #define Y_MAX 400
8 #define GM    39.4784176044  // 4*PI*PI
```

```
 9 #define mm   0.02
10 #define dt   0.001
11
12 void rk4(double rx, double ry, double px, double py,
13     double *rxk, double *ryk, double *pxk, double *pyk ){
14
15   double r, rr3;
16   r   = hypot( rx, ry );  // sqrt(rx^2+ry^2)
17   rr3 = 1.0/(r*r*r);
18   *rxk = (px/mm)*dt;          *ryk = (py/mm)*dt;
19   *pxk = -GM*mm*rx*rr3*dt;  *pyk = -GM*mm*ry*rr3*dt;
20 }
21
22 int main(void)
23 {
24   double rx=-1.4, ry=-0.6;
25   double px=0.01, py=0.08;
26   double rxk[4], ryk[4], pxk[4], pyk[4];
27   int win, cnt=0;
28
29   win = gopen(X_MAX,Y_MAX);
30   newcolor(win, "Yellow");
31   fillcirc(win,X_MAX/2,Y_MAX/2,20,20);
32
33   while(1){
34     rk4(rx, ry, px, py,
35         &rxk[0], &ryk[0], &pxk[0], &pyk[0] );
36
37     rk4(rx+0.5*rxk[0], ry+0.5*ryk[0],
38         px+0.5*pxk[0], py+0.5*pyk[0],
39         &rxk[1], &ryk[1], &pxk[1], &pyk[1] );
40
41     rk4(rx+0.5*rxk[1], ry+0.5*ryk[1],
42          px+0.5*pxk[1], py+0.5*pyk[1],
43          &rxk[2], &ryk[2], &pxk[2], &pyk[2] );
```

```
44
45      rk4(rx+rxk[2], ry+ryk[2],
46           px+pxk[2], py+pyk[2],
47           &rxk[3], &ryk[3], &pxk[3], &pyk[3] );
48
49      rx +=(rxk[0]+2*rxk[1]+2*rxk[2]+rxk[3])*(1.0/6);
50      ry +=(ryk[0]+2*ryk[1]+2*ryk[2]+ryk[3])*(1.0/6);
51      px +=(pxk[0]+2*pxk[1]+2*pxk[2]+pxk[3])*(1.0/6);
52      py +=(pyk[0]+2*pyk[1]+2*pyk[2]+pyk[3])*(1.0/6);
53
54      if(cnt%2000==0){
55          newcolor(win, "Green");
56      }else if(cnt%2000==800){
57          newcolor(win, "Blue");
58      }else if(cnt%2000==1600){
59        newcolor(win, "Red");
60      }
61      cnt++;
62      fillrect(win,rx*100+X_MAX/2,ry*100+Y_MAX/2,5,5);
63      usleep(10000);
64    }
65    ggetch();
66    gclose(win);
67
68    return(0);
69  }
```

解説 このコードでは、4次のルンゲ・クッタ法で繰り返し計算するする必要のある部分を関数にしています（12行から20行)。**mainに戻さなければならない関数で計算された値は、ポインタとして引数に定義**しています。一方で、呼び出し時にはアンパサンド（&）を付けてアドレスを渡しています（例えば35行)。ポインタによるmainと関数との間での数値の共有法を整理すると、

- mainから関数を呼ぶときはアドレスを渡す（&を付けて変数を渡す)。
- 関数ではポインタとして引数を受ける（13行目)。

- 関数内で値を参照するときはアスタリスクを付ける（18 行と 19 行の左辺値）。

このようにすると関数と main の間で同じメモリの内容を参照するので結果を共有することが出来ます。

練習 十分に高い山から物体を水平に発射すると，遠くまで物体が飛んでいく。さらに早く発射すると，ついには地球を周回する物体となる。その最小の大きさを第一宇宙速度という。今，上記のプログラムを「地球を周回する物体」を表していると考える。惑星の初期位置および初期の運動量を調節し[30]、この系における第一宇宙速度の軌道，すなわち円軌道を描くようにしなさい。

練習 物体の初速度が第一宇宙速度よりも早くなると軌道はだ円を描くようになる。さらに初速度がはやくなると物体は無限遠に飛んでいく。その最小の速度を第二宇宙速度という。上の練習で定めた初期位置から物体を放出し，第二宇宙速度となる速度を求めよ。

練習 面積速度一定であることグラフィックスで理解することを考える。常に地球と物体間を結ぶ線を連続で描く。その色を一定ステップ毎に変えると扇型の領域がそれぞれの色で埋められる。そのようなグラフィックスを作成せよ。

[30]接線方向に水平に発射する必要がある。

4.9　生存闘争

生態系における被食者と捕食者の生存闘争モデルとして有名な非線形常微分方程式のロトカ・ヴォルテラの方程式 (Lotka-Volterra) を解きます。連立常微分方程式は、

$$\frac{dx}{dt} = ax - cxy \tag{24}$$

$$\frac{dy}{dt} = -by + cxy \tag{25}$$

です。ここで x は被食者の個体数、y は捕食者の個体数、t は時間です。今、x としてウサギを y としてキツネを考える用いたモデルの解説はここにある。はじめにウサギが 1000 羽[31]、キツネが 100 匹いるとします。捕食者であるキツネは被食者のウサギを食べ、繁殖します。そのうち食べ物（ウサギ）が少なくなり、繁殖が滞るようになります。結果として捕食者の数が減ります。そのおかげで今度は被食者のウサギの数が増えます、その結果、キツネが繁殖できます。そのうちウサギの数が少なくなり・・・・・。

この方程式がなぜそのような振る舞いを示すか理解するためには，微分する変数に注目します。

- 式 24 の左辺で微分しているのは変数 x，すなわちウサギの数です。右辺の第一項を見ると，ax ですので x が大きくなれば成るほど自然と増える方向に変化します。x を距離と考えれば，遠くに行けば行くほど速度が増すというわけです。

- これに対するブレーキの項が右辺第二項。y に比例して，つまりキツネの数に比例してウサギの数の増加にブレーキがかかります。

- 式 25 の左辺で微分しているのは変数 y，すなわちキツネの数です。右辺の第一項を見ると，$-by$ ですので y が大きくなれば成るほど自然と減る方向に変化します。y を距離と考えれば，遠くに行けば行くほど減速するというわけです。

- これに対する加速の項が右辺第二項。x に比例して，つまりウサギの数に比例してキツネの数が増加します。

初期条件として、$a = 0.01, b = 0.05, c = 0.0001$ を仮定します。

[31] ウサギの数え方は「羽」。

ソースコード 23: ウサギとキツネ（eruv.c）

```
#include <eggx.h>
#include <stdio.h>
#include <math.h>
#include <unistd.h>
#define DT 0.5
#define MAX_STEP 10000

void rk4(double aa, double bb, double cc,
    double rx, double ry, double *rxk, double *ryk){
  *rxk = ( aa*rx-cc*rx*ry)*DT;
  *ryk = (-bb*ry+cc*rx*ry)*DT;
}

int main(void)
{
  double time;
  double rrx, rry;
  double rxk[4], ryk[4];
  double aa, bb, cc;
  int step;

  /* initial setting */
  rrx = 1000.0; rry = 100.0;
  aa=0.01; bb=0.05; cc=0.0001;

  for(step=0;step<MAX_STEP;step++){

    rk4(aa, bb, cc, rrx, rry, &rxk[0], &ryk[0]);
    rk4(aa, bb, cc, rrx+0.5*rxk[0], rry+0.5*ryk[0],
        &rxk[1], &ryk[1]);
    rk4(aa, bb, cc, rrx+0.5*rxk[1], rry+0.5*ryk[1],
        &rxk[2], &ryk[2]);
    rk4(aa, bb, cc, rrx+rxk[2], rry+ryk[2],
        &rxk[3], &ryk[3]);
```

```
35
36      rrx  += (rxk[0]+2*rxk[1]+2*rxk[2]+rxk[3])*(1.0/6);
37      rry  += (ryk[0]+2*ryk[1]+2*ryk[2]+ryk[3])*(1.0/6);
38      time  = DT*step;
39      printf("%f␣␣%f␣␣%f␣␣\n",time,rrx,rry);
40
41    }
42    return(0);
43  }
```

このコードは、ターミナルで実行します。コンパイルは gcc で行って下さい。ターミナルへの出力をファイルに書き出し、それを gnuplot で表示します。

```
./a.out > test.dat
gnuplot
gnuplot> plot "test.dat" using 1:2 w l, "test.dat" using 1:3 w l
gnuplot> plot "test.dat" using 2:3 w l
```

1 行目で **a.out** の出力を **test.dat** に書き出しています。2 行目で **gnuplot** を実行しています。3 行目で被食者数の時間変化（1 カラム対 2 カラム）のプロットと、捕食者数の時間変化（1 カラム対 3 カラム）のプロットを実行します。gnuplot の終了は **quit** です。4 行目は被食者対捕食者（2 カラム対 3 カラム）のプロット。周期を描いている様子がわかります。

ロトカ・ヴォルテラ方程式は競合する二つの事象の時間変化を表す微分方程式です。したがって被食者・捕食者のみならず，競合する事象が係わる様々な現象についてこの微分方程式を適用することができます。いくつか例をあげます。

- 水産資源の解析。
- ロトカ・ヴォルテラ方程式によるマーケティング競争分析。
- プラズマダイナミクス。

練習 プログラムを eggx 対応に書き換えなさい。eggx の機能を使って実行中にリアルタイムで図 8 や図 9 に相当する画像を描画するようにしなさい。

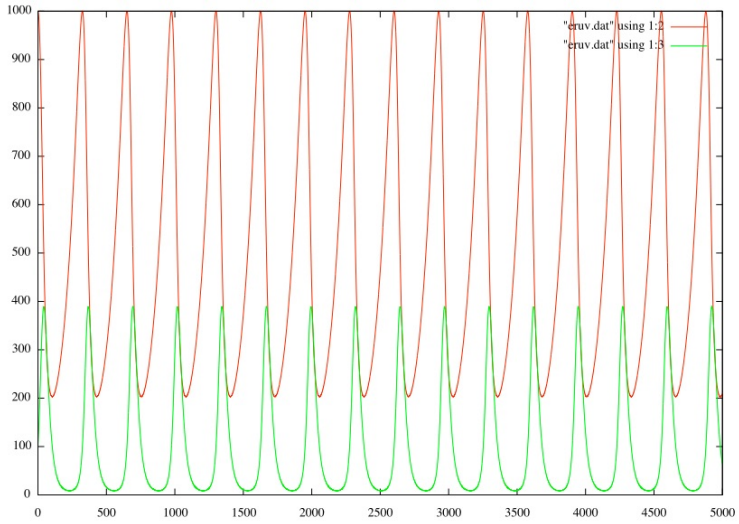

図 8: ウサギの数（赤）と狐の数（緑）の時間変化。ウサギの数が減ってくると狐の数も減る。ウサギの数が増えてくると，時間をおいて狐の数も増えていく。

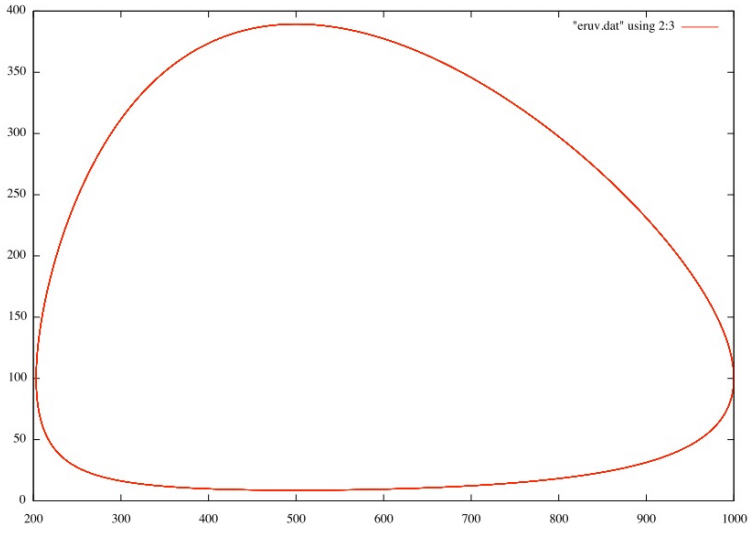

図 9: ウサギの数（x 軸）と狐の数（y 軸）の関係。この位相空間で閉じたサイクルを繰り返す。

4.10　ファン・デル・ポール方程式

自律した振動系を形成する非線形常微分方程式のファン・デル・ポール方程式（Van der Pol Equations）を学びます。系は次式で定義されます。

$$\frac{d^2x}{dt^2} + \mu(x^2 - 1)\frac{dx}{dt} + x = 0 \tag{26}$$

ここで $\mu > 0$ です。第二項と第三項を右辺に移します。

$$\frac{d^2x}{dt^2} = -\mu(x^2 - 1)\frac{dx}{dt} - x \tag{27}$$

右辺の第一項を見ると，係数に $(x^2 - 1)$ とあります。これは $|x| > 1$ ならば必ず正となります。このときの右辺第一項は速度に依存してマイナスの値となりますから，左辺の加速度にブレーキをかける役割を果たします（減衰項)。すなわち $|x| > 1$ において運動の抵抗として働き、振動を小さくする効果をもたらします。一方、$|x| < 1$ ならば必ず正となります。このときの右辺第一項は速度に依存してプラスの値となりますから，左辺の加速度を大きくする役割を果たします。すなわち $|x| < 1$ において負の抵抗として働き、振動を大きくする効果をもたらします。

バネ（自然長 $x = 0$，ばね定数 k）につながれて振動している物体 m に速度に比例した抵抗 λ が働いている場合の微分方程式は以下で表すことができます。

$$m\frac{d^2x}{dt^2} = -\lambda\frac{dx}{dt} - kx \tag{28}$$

ファン・デル・ポール方程式では減衰項の係数 $(x^2 - 1)$ に比例していて非線形となっていたところが線形（係数が定数）であることがわかります。

二階の微分方程式のままではルンゲクッタ法を適用できないので，これを書き換え、一階の連立微分方程式にします。

$$\frac{dx}{dt} = y \tag{29}$$

$$\frac{dy}{dt} = -x + \mu y(1 - x^2) \tag{30}$$

初期条件として、$x(0) = 0.3, \mu = 1.8$ を仮定します。時間刻みは $dt = 0.01$ とします。

ソースコード 24: ファン・デル・ポール（vdp.c）

```
1 #include <stdio.h>
2 #include <math.h>
```

```
3  #include <unistd.h>
4
5  #define dt 0.01
6  #define mu 1.8
7
8  void rk4(double rx, double ry, double *rxk, double *ryk){
9    *rxk =  ry*dt;
10   *ryk = (-rx+mu*ry*(1.0-rx*rx))*dt;
11 }
12
13 int main(void){
14   double time, rrx, rry;
15   double rxk[4], ryk[4];
16   int step, max_step;
17
18   rrx = 0.3; rry = 0.0; // initial set
19   max_step=5000;
20
21   for(step=0;step<max_step;step++){
22
23     rk4(rrx, rry, &rxk[0], &ryk[0]);
24     rk4(rrx+0.5*rxk[0], rry+0.5*ryk[0],
25         &rxk[1], &ryk[1]);
26     rk4(rrx+0.5*rxk[1], rry+0.5*ryk[1],
27         &rxk[2], &ryk[2]);
28     rk4(rrx+rxk[2], rry+ryk[2],
29         &rxk[3], &ryk[3]);
30
31     rrx += (rxk[0]+2*rxk[1]+2*rxk[2]+rxk[3])*(1.0/6);
32     rry += (ryk[0]+2*ryk[1]+2*ryk[2]+ryk[3])*(1.0/6);
33     time = dt*step;
34
35     printf("%f␣␣%f␣␣%f␣␣\n",time,rrx,rry);
36   }
37   return(0);
```

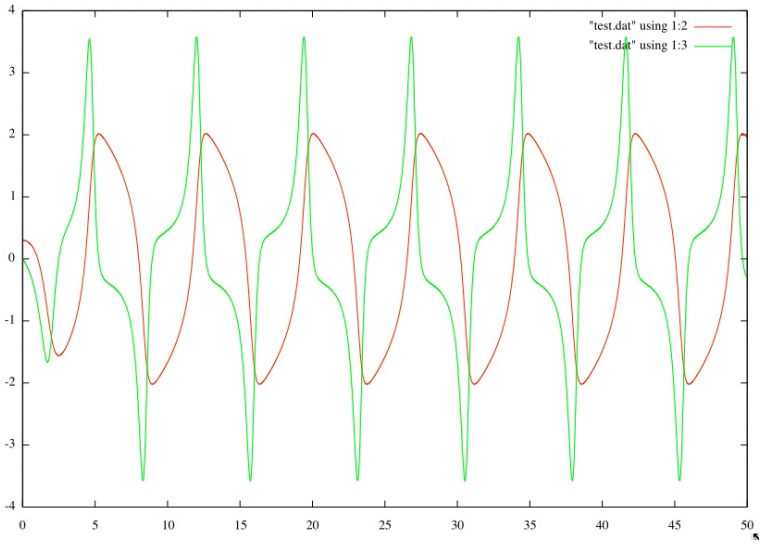

図 10: 時間に対する x の変化（赤）と y の変化（緑）のグラフ。

```
38 }
```

このコードは、ターミナルで実行します。コンパイルは gcc で行って下さい。ターミナルへの出力をファイルに書き出し、それを gnuplot で表示します。

```
$ gcc vdp.c -o vdp
./vdp > test.dat
gnuplot
gnuplot> plot "test.dat" using 1:2 w l, "test.dat" using 1:3 w l
gnuplot> plot "test.dat" using 2:3 w l
```

1 行目で **a.out** の出力を **test.dat** に書き出しています。2 行目で **gnuplot** を実行しています。3 行目で時間に対する x の変化のプロットを、4 行目で x 対 y のプロットを実行します。特に x 対 y のグラフから、系が安定的にサイクルを描く様子がわかります（相空間における「閉軌道サイクル」－リミットサイクル）。

ファン・デル・ポール方程式は応用範囲が非常に広く，神経回路網から生命、経済、ロボット工学などで解析に用いられています。Crucifix は，更新世 (約 258 万年前から 1 万 1700 年前までの時代) における気候の変動現象とその振動について常微分方程式を適用し、リミットサイクルを示しました [8]。

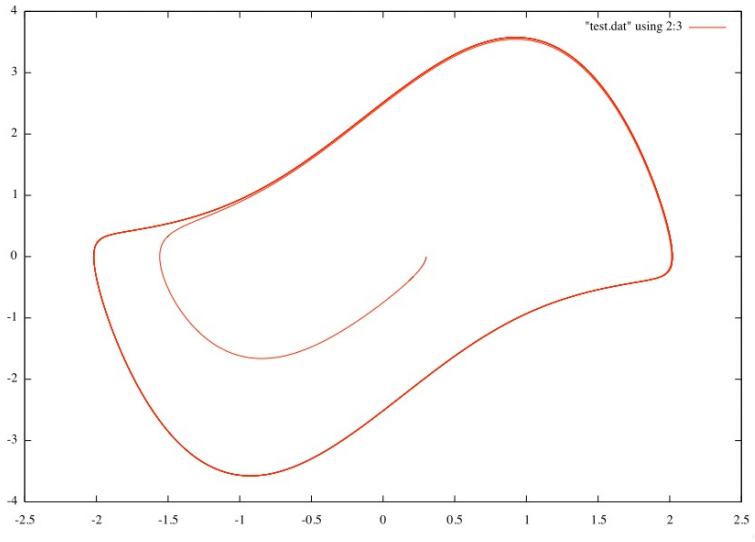

図 11: x-y のグラフ。中心当たりから出発して閉軌道に入り，そこから延々と同じサイクルを繰り返す。

4.11 GNU Scientific Library

GNU Scientific Library (GSL) はオープンソースの数値計算ライブラリですが，非常に良く整備されていて本格的な科学技術シミュレーションに耐えうるほど質の高い関数が揃っています [5]。ここではいくつかの GSL を用いた数値計算例をあげます。

4.11.1 GSL による乱数

コンピュータシミュレーションにとって乱数は重要な役割をはたします。困ったことにデフォルトで使える C の乱数は性質の良いものではありません。ここでは GSL に登録されている乱数発生関数を導入します。

ソースコード 25: GSL を用いて 10 個の乱数を発生させる ran01.c

```
1 #include <stdio.h>
2 #include <gsl/gsl_rng.h>
3 #include <sys/time.h>
4
5 int main (void){
6   const gsl_rng_type *T;
7   gsl_rng *r;
8   int i, n = 10;
```

```
9
10   unsigned int gslseed = time(NULL);
11   gsl_rng_env_setup();
12   T = gsl_rng_default;
13   r = gsl_rng_alloc (T);
14   gsl_rng_set(r,gslseed);
15
16   printf ("Random␣number␣generator:␣%s\n", (*T).name);
17   printf ("Seed:␣%u\n", gslseed);
18   for (i = 0; i < n; i++) {
19     double u = gsl_rng_uniform (r);
20     printf ("%.5f\n", u);
21   }
22   gsl_rng_free (r);
23   return 0;
24 }
```

6 行目は，使用する乱数発生器に関する情報を保持します。例えば 17 行目では発生器の名前を参照して印字しています。10 行目で乱数のシードを決めています。ここでは乱数発生器の名前を指定していませんので，デフォルトの発生器 mt19937 (メルセンヌツイスタ [9]) が設定されます。12, 13 行目で乱数発生器にメモリを割り当て，14 行目で乱数発生の準備が完了します。実際に乱数を発生させているのは 19 行目です。デフォルトの mt19937 で [0.0, 1.0) の範囲で[32]一様分布な乱数を発生させてみます。

コンパイルは，

```
% gcc ran01.c -o ran01  -lgsl -lgslcblas -lm
```

とします。実行結果は，

```
% ./ran01
Random number generator: mt19937
```

[32]0.0 以上で 1.0 より小さい。つまり 0.0 は含むが 1.0 は除外する。

```
Seed: 1709527858
0.34428
0.83927
0.73199
.......
```

となります。

4.11.2 ファン・デル・ポール方程式再訪

GNU Scientific Library (GSL) を用いると高精度に常微分方程式を数値計算で解くことができます。第 4.10 節において，４次のルンゲクッタ法をプログラムすることでファン・デル・ポール方程式を数値計算で解きました。ここでは GSL のマニュアルの Example をとりあげることで，GSL ではどのように常微分方程式を扱っているかについて理解を深めましょう。 サンプルコード 26 は，このリンク先のマニュアルページの Examples にある，最初のコードなのでそこからコピーペーストして実行することができます。

ソースコード 26: GSL でファン・デル・ポール方程式を解くコード。マニュアル記載のコードからパラメータ設定値を変更しています（33，40，41 行）。

```
1  #include <stdio.h>
2  #include <gsl/gsl_errno.h>
3  #include <gsl/gsl_matrix.h>
4  #include <gsl/gsl_odeiv2.h>
5
6  int func(double t, const double y[], double f[],
7      void *params){
8    (void)(t); /* avoid unused parameter warning */
9    double mu = *(double *)params;
10   f[0] = y[1];
11   f[1] = -y[0] - mu*y[1]*(y[0]*y[0] - 1);
12   return GSL_SUCCESS;
13 }
14
15 int jac(double t, const double y[], double *dfdy,
16       double dfdt[], void *params){
17
18   (void)(t); /* avoid unused parameter warning */
19   double mu = *(double *)params;
20   gsl_matrix_view dfdy_mat
21     = gsl_matrix_view_array (dfdy, 2, 2);
22   gsl_matrix * m = &dfdy_mat.matrix;
23   gsl_matrix_set(m, 0, 0, 0.0);
24   gsl_matrix_set(m, 0, 1, 1.0);
```

```
25    gsl_matrix_set(m, 1, 0, -2.0*mu*y[0]*y[1] - 1.0);
26    gsl_matrix_set(m, 1, 1, -mu*(y[0]*y[0] - 1.0));
27    dfdt[0] = 0.0;
28    dfdt[1] = 0.0;
29    return GSL_SUCCESS;
30  }
31
32  int main (void) {
33    double mu = 1.8;
34    gsl_odeiv2_system sys ={func, jac, 2, &mu};
35    gsl_odeiv2_driver *d =
36      gsl_odeiv2_driver_alloc_y_new(&sys,
37            gsl_odeiv2_step_rk8pd,
38                                    1e-6, 1e-6, 0.0);
39    int i;
40    double t = 0.0, t1 = 50.0;
41    double y[2] = { 0.3, 0.0 };
42
43    for (i = 1; i <= 1000; i++) {
44        double ti = i * t1 / 1000.0;
45        int status = gsl_odeiv2_driver_apply(d, &t, ti, y);
46
47        if (status != GSL_SUCCESS) {
48            printf ("error,␣return␣value=%d\n", status);
49            break;
50        }
51
52        printf ("%.5e␣%.5e␣%.5e\n", t, y[0], y[1]);
53    }
54
55    gsl_odeiv2_driver_free (d);
56    return 0;
57  }
```

マニュアルの記述に従い u と v を用いると，ファン・デル・ポール方程式は，

$$u''(t) + \mu u'(t)\{u^2(t) - 1)\} + u(t) = 0 \tag{31}$$

と書くことが出来ます。ここで $v = u'(t)$ として，

$$u' = v \tag{32}$$

$$v' = -u - \mu v(u^2 - 1) \tag{33}$$

とし，以前と同様に一階の微分方程式の組に変形します。また積分の変数変換に必要となるヤコビアン J は，

$$J = \begin{pmatrix} \frac{\partial u'}{\partial u} & \frac{\partial u'}{\partial v} \\ \frac{\partial v'}{\partial u} & \frac{\partial v'}{\partial v} \end{pmatrix} \tag{34}$$

$$= \begin{pmatrix} -1 - 2\mu uv & \mu - \mu u^2 \\ 0 & 1.0 \end{pmatrix} \tag{35}$$

です。ただしプログラム内では次の変数の置き換えをします。

$$u \to y[0] \tag{36}$$

$$v \to y[1] \tag{37}$$

$$u' \to f[0] \tag{38}$$

$$v' \to f[1] \tag{39}$$

これにより，関数 func で定義されている解くべき微分方程式は，

$$f[0] = y[1] \tag{40}$$

$$f[1] = -y[0] - \mu \cdot y[1](y[0]^2 - 1) \tag{41}$$

となります。また関数 jac で定義されているヤコビアンはポインタ m にセットされ，その成分は，

$$\begin{pmatrix} m(1,0) & m(1,1) \\ m(0,0) & m(0,1) \end{pmatrix} = \begin{pmatrix} -1 - 2\mu \cdot y[0] \cdot y[1] & \mu - \mu \cdot y[0]^2 \\ 0 & 1.0 \end{pmatrix} \tag{42}$$

となります。プログラムにおいて常微分方程式はデータ型 gsl_odeiv2_system で定義されます（34 行目）。

```
gsl_odeiv2_system sys={func, jac, 2, &mu};
```

従って関数 func で定義した微分方程式，関数 jac で定義したヤコビアンが sys に収められます。ドライバーオブジェクトは，原始的な取り扱いをしなくても済むように設計されたハイレベルのラッパーで，gsl_odeiv2_driver で生成されます（35 行から 38 行）。また 37 行で， gsl_odeiv2_step_rk8pd と指定しているので，時間発展にルンゲ・クッタ，プリンス・ドーマンド (8,9) 法を採用しています。繰り返しループは 43 行からはじまり，時間 t が 0 から 50 まで計算します。実際に数値積分をしているのは 45 行目の gsl_odeiv2_driver_apply です。ソースコードのファイル名を vdp-gsl.c とすると，コンパイルは，

```
% gcc vdp-gsl.c -o vdp-gsl  -lgsl -lgslcblas -lm
```

します。実行は，

```
% ./vdp-gsl
```

です。

4.11.3 ロトカ・ヴォルテラ方程式再訪

第 4.9 節ではロトカ・ヴォルテラ方程式を手でコーディングした 4 次のルンゲクッタ法で解きました。ここでは前節のようにロトカ・ヴォルテラ方程式を GSL のライブラリで解いてみます。微分方程式を u と v を用いて表すと,

$$u' = au - cuv \tag{43}$$
$$v' = -bu + cuv \tag{44}$$

となります。ここで a, b, c は任意の定数で, 前回同様, $a = 0.01, b = 0.05, c = 0.0001$ とします。従って，解くべき微分方程式は,

$$u' = 0.01u - 0.0001uv \tag{45}$$
$$v' = -0.05u + 0.0001uv \tag{46}$$

となります。プログラムと対応がつくように書き直します。

$$f[0] = 0.01y[0] - 0.0001y[0] \cdot y[1] \tag{47}$$
$$f[1] = -0.05y[1] + 0.0001y[0] \cdot y[1] \tag{48}$$

ヤコビアン J は,

$$J = \begin{pmatrix} \frac{\partial u'}{\partial u} & \frac{\partial u'}{\partial v} \\ \frac{\partial v'}{\partial u} & \frac{\partial v'}{\partial v} \end{pmatrix} \tag{49}$$
$$= \begin{pmatrix} a - cv & -cu \\ -b + cv & cu \end{pmatrix} \tag{50}$$
$$= \begin{pmatrix} 0.01 - 0.0001 \cdot y[1] & -0.0001 \cdot y[0] \\ -0.05 + 0.0001 \cdot y[1] & 0.0001 \cdot y[0] \end{pmatrix} \tag{51}$$
$$= \begin{pmatrix} m(1,0) & m(1,1) \\ m(0,0) & m(0,1) \end{pmatrix} \tag{52}$$

です。あとはファン・デル・ポール方程式のコードを元に，上記の結果を機械的にプログラムに組み込むだけでコードが完成します。以下にサンプルコードを示します。

ソースコード 27: GSL でロトカ・ヴォルテラ方程式を解くコード。

```
1 #include <stdio.h>
2 #include <gsl/gsl_errno.h>
3 #include <gsl/gsl_matrix.h>
```

```
4  #include <gsl/gsl_odeiv2.h>
5
6  int func(double t, const double y[], double f[],
7      void *params){
8    (void)(t); /* avoid unused parameter warning */
9    double mu = *(double *)params;
10   f[0] =  0.01*y[0] - 0.0001*y[0]*y[1];
11   f[1] = -0.05*y[1] + 0.0001*y[0]*y[1];
12   return GSL_SUCCESS;
13 }
14
15 int jac(double t, const double y[], double *dfdy,
16       double dfdt[], void *params){
17
18   (void)(t); /* avoid unused parameter warning */
19   double mu = *(double *)params; // dumy
20   gsl_matrix_view dfdy_mat
21     = gsl_matrix_view_array (dfdy, 2, 2);
22   gsl_matrix * m = &dfdy_mat.matrix;
23   gsl_matrix_set(m, 0, 0, -0.05+0.0001*y[1]);
24   gsl_matrix_set(m, 0, 1,  0.0001*y[0]);
25   gsl_matrix_set(m, 1, 0,  0.01-0.0001*y[1]);
26   gsl_matrix_set(m, 1, 1, -0.0001*y[0]);
27   dfdt[0] = 0.0;
28   dfdt[1] = 0.0;
29   return GSL_SUCCESS;
30 }
31
32 int main (void) {
33   double mu = 0;
34   gsl_odeiv2_system sys ={func, jac, 2, &mu};
35   gsl_odeiv2_driver *d =
36     gsl_odeiv2_driver_alloc_y_new(&sys,
37           gsl_odeiv2_step_rk8pd,
38                                   1e-6, 1e-6, 0.0);
```

```
39   int i;
40   double t = 0.0, t1 =10000.0;
41   double y[2] = {1000.0, 100.0};
42
43   for (i = 1; i <= 10000; i++) {
44       double ti = i * 0.5;
45       int status = gsl_odeiv2_driver_apply(d, &t, ti, y);
46
47       if (status != GSL_SUCCESS) {
48           printf ("error, return value=%d\n", status);
49           break;
50       }
51
52       printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
53   }
54
55   gsl_odeiv2_driver_free (d);
56   return 0;
57 }
```

4.11.4 ジャパニーズアトラクター

1978 年に当時京都大学の上田睨亮氏によって発表されたジャパニーズアトラクターをシミュレートします [10, 11, 12]。元々は非線形インダクタンスをもつ正弦波電圧の印加された直列共振回路に対して提案されたモデル方程式です。変数 u と v を用いると，

$$u''(t) + ku'(t) + u^3(t) = B\cos t \tag{53}$$

と書くことが出来ます。この式を一階の連立微分方程式で表すと，

$$u' = v \tag{54}$$
$$v' = -kv - u^3 + B\cos t \tag{55}$$

となります。プログラムに合わせた表現では，

$$f[0] = y[1] \tag{56}$$
$$f[1] = -k \cdot y[1] - y^3[0] + B\cos t \tag{57}$$

です。ここで k および B は定数で，論文にしたがって $k = 0.1, B = 12.0$ ととります。ヤコビアン J は，

$$J = \begin{pmatrix} \frac{\partial u'}{\partial u} & \frac{\partial u'}{\partial v} \\ \frac{\partial v'}{\partial u} & \frac{\partial v'}{\partial v} \end{pmatrix} \tag{58}$$
$$= \begin{pmatrix} 0 & 1 \\ -3u^2 & -k \end{pmatrix} \tag{59}$$
$$\tag{60}$$

となります。前節のロトカ・ヴォルテラ方程式を解くコードを元に，上記の結果を機械的に組み込むとプログラムができあがります。方程式は決定論的なもので，予測できない振る舞いは入り込む余地が無いように思えます。しかし驚くことにカオスの振る舞いが観測されるのです。上田氏は，1961 年，アナログ計算機でこの現象を発見しました。以下にサンプルコードを示します。

ソースコード 28: GSL でジャパニーズアトラクタを解くコード。パラメータの k と B は 6 行目と 7 行目で与えている。初期値は $u = 3, v = 0$ としています (44 行目)。

```
1 #include <stdio.h>
2 #include <gsl/gsl_errno.h>
3 #include <gsl/gsl_matrix.h>
```

```
4  #include <gsl/gsl_odeiv2.h>
5  #include <math.h>
6  #define KAPPA  0.1
7  #define BBB    12.0
8
9  int func(double t, const double y[], double f[],
10    void *params){
11   (void)(t); /* avoid unused parameter warning */
12   double mu = *(double *)params;
13   f[0] = y[1];
14   f[1] = -KAPPA*y[1] - y[0]*y[0]*y[0]+ BBB* cos(t);
15   return GSL_SUCCESS;
16 }
17
18 int jac(double t, const double y[], double *dfdy,
19      double dfdt[], void *params){
20
21   (void)(t); /* avoid unused parameter warning */
22   double mu = *(double *)params;
23   gsl_matrix_view dfdy_mat
24     = gsl_matrix_view_array (dfdy, 2, 2);
25   gsl_matrix * m = &dfdy_mat.matrix;
26   gsl_matrix_set(m, 0, 0, -3.0*y[0]*y[0]);
27   gsl_matrix_set(m, 0, 1, -KAPPA);
28   gsl_matrix_set(m, 1, 0, 0.0);
29   gsl_matrix_set(m, 1, 1, 1.0);
30   dfdt[0] = 0.0;
31   dfdt[1] = 0.0;
32   return GSL_SUCCESS;
33 }
34
35 int main (void) {
36   double mu = 1.8;
37   gsl_odeiv2_system sys ={func, jac, 2, &mu};
38   gsl_odeiv2_driver *d =
```

```
39      gsl_odeiv2_driver_alloc_y_new(&sys,
40            gsl_odeiv2_step_rk8pd,
41                                    1e-6, 1e-6, 0.0);
42    int i;
43    double t = 0.0, t1 = 2.0*3.14159263;
44    double y[2] = { 3.0, 0.0 };
45
46    for (i = 1; i <= 1000000; i++) {
47        double ti = i * t1 / 100.0;
48        int status = gsl_odeiv2_driver_apply(d, &t, ti, y);
49
50        if (status != GSL_SUCCESS) {
51            printf ("error, return value=%d\n", status);
52            break;
53        }
54        if(i%100==0){
55    printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
56        }
57    }
58
59    gsl_odeiv2_driver_free (d);
60    return 0;
61  }
```

43 行目で ti=2 π としてその間を 100 分割し，微分の時間発展を追っています。55 行目で $x = 2n\pi$ $\{n = 1, 2, 3 \cdots\}$ 毎に (t, u, v) を出力しています。コンパイルと実行は，例えば以下のようにします。ソースコードのファイル名を jpsat.c とすると，コンパイルは，

```
% gcc jpsat.c -o jpsat  -lgsl -lgslcblas -lm
```

とします。実行は，

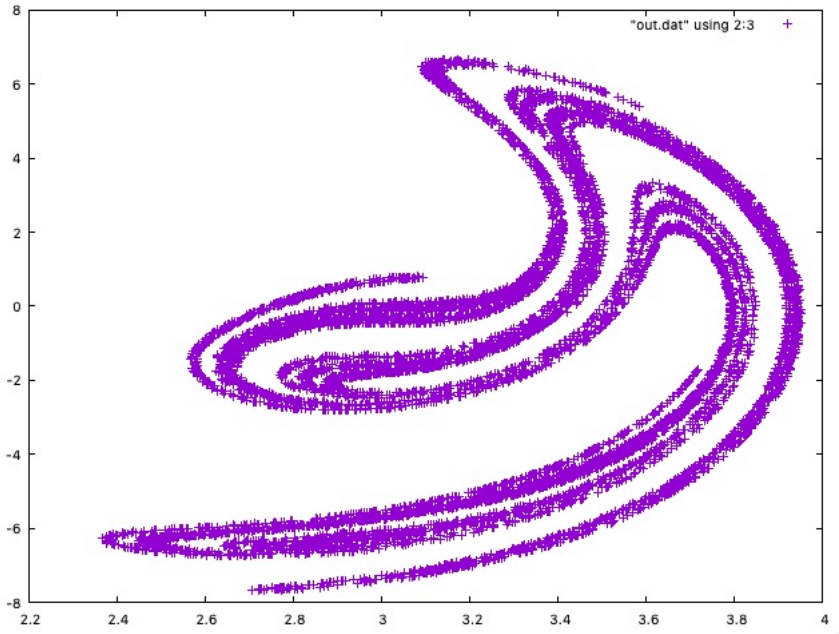

図 12: ジャパニーズアトラクター。

```
% ./jpsat $>$ out.dat
```

として，ファイル out.dat に出力結果を記録します。out.dat に記録されている (u, v) の $t = 2\pi n$ 毎の軌跡を **gnuplot** で見るためには、

$ gnuplot ↵
gnuplot > plot “out.dat” using 2:3 w l

とします。

4.11.5 ローレンツアトラクタ

ローレンツ方程式はエドワード・ローレンツによって大気の熱対流をモデル化するために 1963 年に発表された常微分方程式です [13]。次の 3 つの式で定義されます。

$$\frac{dx}{dt} = \sigma(y - x) \tag{61}$$
$$\frac{dy}{dt} = x(\rho - z) - y \tag{62}$$
$$\frac{dz}{dt} = xy - \beta z \tag{63}$$

u, v, w で表すと,

$$u' = \sigma(v - u) \tag{64}$$
$$v' = u(\rho - w) - v \tag{65}$$
$$w' = uv - \beta w \tag{66}$$

となります。プログラムに合わせた表現では,

$$f[0] = \sigma(y[1] - y[0]) \tag{67}$$
$$f[1] = y[0](\rho - y[2]) - y[1] \tag{68}$$
$$f[2] = y[0]y[1] - \beta y[2] \tag{69}$$

です。ヤコビアン J は,

$$J = \begin{pmatrix} \frac{\partial u'}{\partial u} & \frac{\partial u'}{\partial v} & \frac{\partial u'}{\partial w} \\ \frac{\partial v'}{\partial u} & \frac{\partial v'}{\partial v} & \frac{\partial v'}{\partial w} \\ \frac{\partial w'}{\partial u} & \frac{\partial w'}{\partial v} & \frac{\partial w'}{\partial w} \end{pmatrix} \tag{70}$$
$$= \begin{pmatrix} -\sigma & \sigma & 0 \\ \rho - y[2] & -1 & -y[0] \\ y[1] & y[0] & -y[2] \end{pmatrix} \tag{71}$$

です。あとはこれらをプログラムに機械的に組み込むと完成です。以下にサンプルコードを示します。

ソースコード 29: GSL でローレンツアトラクタを解くコード。パラメータの σ, β, ρ は 5 行から 7 行で与えている。初期値は $u = 10, v = 10, w = 10$ としています (52 行目)。

```
1 #include <stdio.h>
```

```
2  #include <gsl/gsl_errno.h>
3  #include <gsl/gsl_matrix.h>
4  #include <gsl/gsl_odeiv2.h>
5  #define SIG  10
6  #define BET  (8.0/3.0)
7  #define RHO  28
8
9  int func(double t, const double y[], double f[],
10    void *params){
11   (void)(t); /* avoid unused parameter warning */
12   double mu = *(double *)params;
13
14   f[0] = SIG*(y[1] - y[0]);
15   f[1] = y[0]*(RHO - y[2]) - y[1];
16   f[2] = y[0]*y[1] - BET*y[2];
17   return GSL_SUCCESS;
18 }
19
20 int jac(double t, const double y[], double *dfdy,
21      double dfdt[], void *params){
22
23   (void)(t); /* avoid unused parameter warning */
24   double mu = *(double *)params; // dumy
25   gsl_matrix_view dfdy_mat
26     = gsl_matrix_view_array (dfdy, 3, 3);
27   gsl_matrix * m = &dfdy_mat.matrix;
28   gsl_matrix_set(m, 0, 0,     y[1]);
29   gsl_matrix_set(m, 0, 1,     y[0]);
30   gsl_matrix_set(m, 0, 2,    -y[2]);
31   gsl_matrix_set(m, 1, 0, RHO-y[2]);
32   gsl_matrix_set(m, 1, 1,       -1);
33   gsl_matrix_set(m, 1, 2,    -y[0]);
34   gsl_matrix_set(m, 2, 0,     -SIG);
35   gsl_matrix_set(m, 2, 1,      SIG);
36   gsl_matrix_set(m, 2, 2,      0.0);
```

```
37   dfdt[0] = 0.0;
38   dfdt[1] = 0.0;
39   dfdt[2] = 0.0;
40   return GSL_SUCCESS;
41 }
42
43 int main (void) {
44   double mu = 0;
45   gsl_odeiv2_system sys ={func, jac, 3, &mu};
46   gsl_odeiv2_driver *d =
47     gsl_odeiv2_driver_alloc_y_new(&sys,
48           gsl_odeiv2_step_rk8pd,
49                                   1e-6, 1e-6, 0.0);
50   int i;
51   double t = 0.0, t1 =10000.0;
52   double y[3] = {10.0, 10.0, 10.0};
53
54   for (i = 1; i <= 10000; i++) {
55       double ti = i /100.0 ;
56       int status = gsl_odeiv2_driver_apply(d, &t, ti, y);
57
58       if (status != GSL_SUCCESS) {
59           printf ("error, return value=%d\n", status);
60           break;
61       }
62       printf ("%.5e %.5e %.5e\n", y[0], y[1], y[2]);
63   }
64
65   gsl_odeiv2_driver_free (d);
66   return 0;
67 }
```

(u, v, w) の軌跡を **gnuplot** で見るためには、

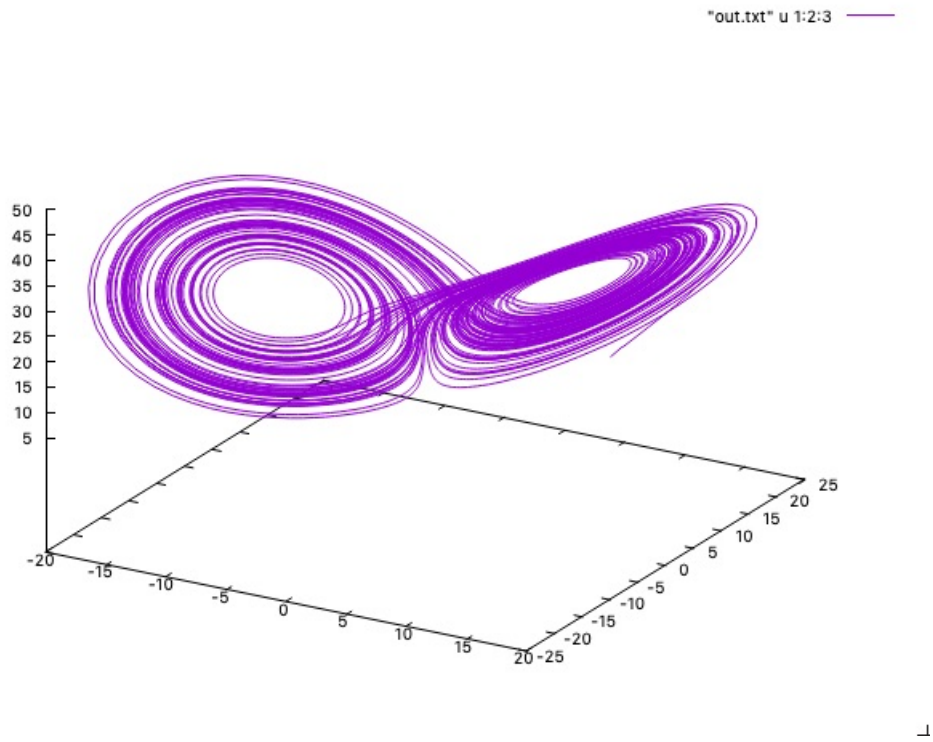

図 13: ローレンツアトラクター。gnuplot 内ではマウスでつかんで動かすことで角度を変えて見ることが出来ます。

```
$ gnuplot ↵
gnuplot > splot "out.txt" u 1:2:3 w l
```

とします（図 13）。

ローレンツアトラクタは初期条件に非常に敏感です。少しでも異なると後の結果が大きく異なってきます。

試しに w の初期条件の値を 0.00001 だけ増やした場合を計算します。

```
double y[3] = {10.0, 10.0, 10.00001};
```

この出力を out2.txt とし，もとのコードの出力を out1.txt とします。

```
gnuplot > splot "out1.txt" u 1:2:3 w l, "out2.txt" u 1:2:3 w l
```

とすると先ほどのような 3D グラフが得られます（図 14）。ここで元の出力が赤の

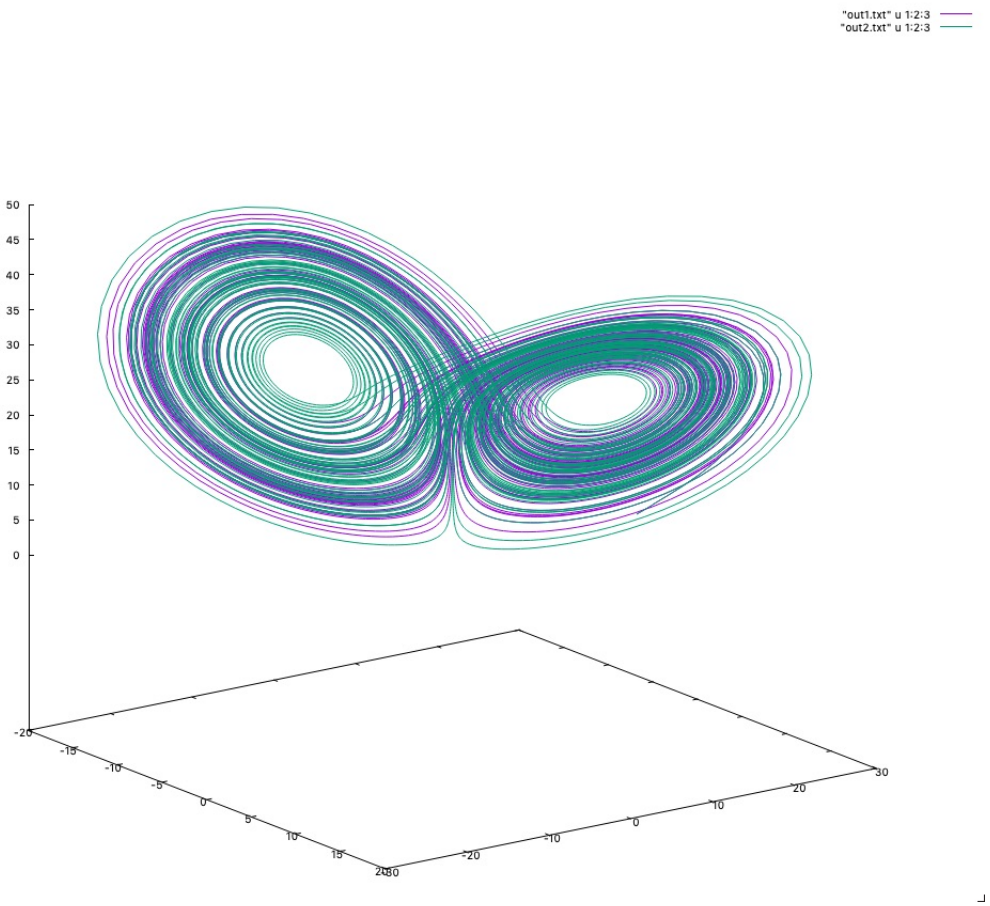

図 14: 元々の条件のプロット（赤）と，w の初期条件を 0.00001 だけ増やしたときのプロット（緑）。

線で，ほんの少し w の初期条件を変えた場合の出力が緑で描かれています。

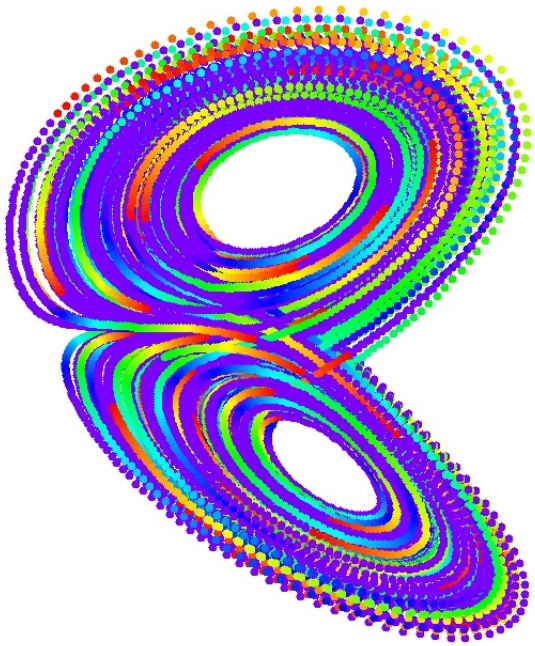

図 15: eggx でグラフィックスを表示した場合。

eggx でグラフィックスを出すようにすることもできます。サンプルコードは GitHub に登録されています。

図 15 では計算ステップ毎に表示するカラーが変化するようにしています。プログラムでは (x, y, z) の座標を回転させディスプレイの座標 (X, Y) に投影しています（回転行列は付録の第 A.2 節にて定義)。ちょうど 3 次元の物体を斜めから見たことに相当します。

ダウンロードしたフォルダには,

```
egg.c  function.c  main.c  Makefile  prototypes.h  rw.h
```

が含まれています。関数の定義は function.c で行っています。メイン関数は main.c で定義しています。Makefile はコンパイルのルールを記述したファイルです。ターミナルでこのフォルダの中に移動し，そこで

```
% make
```

とキーボードから打つとコンパイルが実行されます。

```
% make clean
```

で，フォルダ内を掃除します。

練習 フォルダの中にあるファイルが，それぞれ何をしているか調べてみてください。

練習 例えば一度 make した後，egg.c のみを少し書き換えて再度 make をしてみてください。

練習 ローレンツアトラクタの eggx 版を参考に，前節のジャパニーズアトラクタを eggx 対応にしてみよう。

4.11.6 Thomas のアトラクタ

Thomas が提案したアトラクタの方程式です [14, 15]。

$$\frac{\partial x}{\partial t} = \sin y - bx \tag{72}$$
$$\frac{\partial y}{\partial t} = \sin z - by \tag{73}$$
$$\frac{\partial z}{\partial t} = \sin x - bz \tag{74}$$
$$\tag{75}$$

ここで定数は $b = 0.208186$ とします。u, v, w で表すと,

$$u' = \sin v - bu \tag{76}$$
$$v' = \sin w - bv \tag{77}$$
$$w' = \sin u - bw \tag{78}$$

となります。プログラムに合わせた表現では,

$$f[0] = \sin y[1] - b \cdot y[0] \tag{79}$$
$$f[1] = \sin y[2] - b \cdot y[1] \tag{80}$$
$$f[2] = \sin y[0] - b \cdot y[2] \tag{81}$$

です。ヤコビアン J は,

$$J = \begin{pmatrix} -b & \cos y[1] & 0 \\ 0 & -b & \cos y[2] \\ \cos y[0] & 0 & -b \end{pmatrix} \tag{82}$$

です。あとはこれらをプログラムに機械的に組み込むとプログラムは完成です。ただし図の範囲やステップ数などの表示に関するパラメータは別途最適化する必要があります。

サンプルコードは GitHub に登録されています。

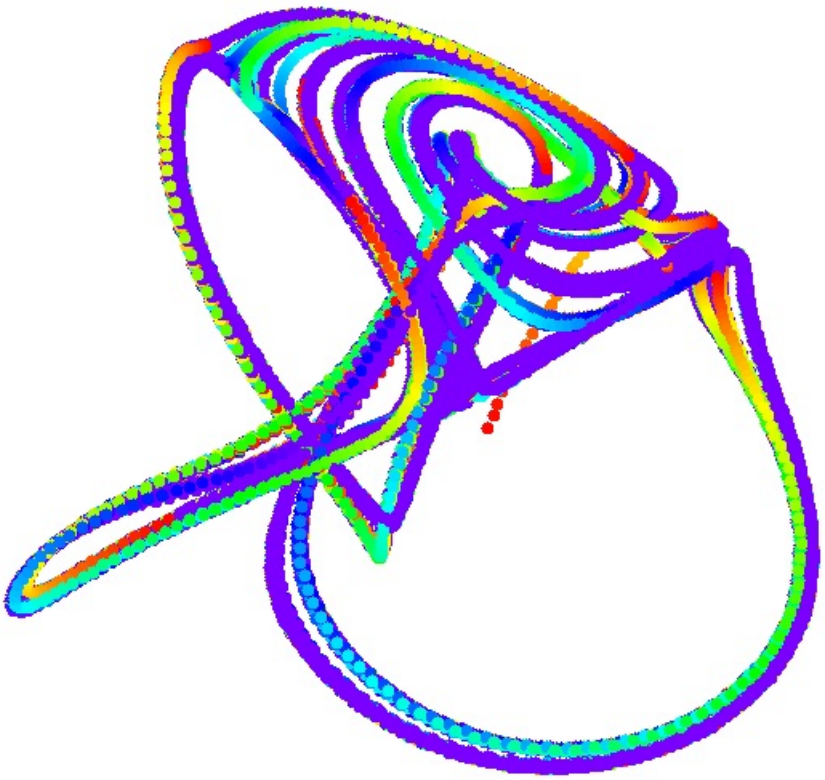

図 16: Thomas のアトラクタ。

4.11.7 RLC 直列回路

RLC 直列回路の過渡現象を見てみます。以下，MIT の OpenCourseWare のテキストを参考に式を書き下します。まず電圧については以下の式が成立します。

$$V_R + V_L + V_C = E \tag{83}$$

回路を流れる電流 i は，

$$i = C\frac{dV_C}{dt} \tag{84}$$

です。R と L にかかる電圧は，

$$V_R = iR = RC\frac{dV_C}{dt} \tag{85}$$

$$V_L = L\frac{di}{dt} = LC\frac{d^2V_C}{dt^2} \tag{86}$$

式 85 と式 86 を式 83 に代入して，

$$\frac{d^2V_C}{dt^2} + \frac{R}{L}\frac{dV_C}{dt} + \frac{1}{LC}V_C = \frac{1}{LC}E \tag{87}$$

もちろん，解析解も得られていて[33]，

$$V_C = E + A_1e^{S_1t} + A_2e^{S_2t} \tag{88}$$

で与えられます。ここで，

$$S_1 = -\frac{R}{2L} + \sqrt{\left(\frac{R}{2L}\right)^2 - \frac{1}{LC}} \tag{89}$$

$$S_2 = -\frac{R}{2L} - \sqrt{\left(\frac{R}{2L}\right)^2 - \frac{1}{LC}} \tag{90}$$

$$\tag{91}$$

です。ここでルートの中の値によって場合分けをします。

1. $R = 2\sqrt{\frac{L}{C}}$ の場合（ルートの中がゼロ）。
   Critically damped system
2. $R > 2\sqrt{\frac{L}{C}}$ の場合（ルートの中が正）。
   Over damped system
3. $R < 2\sqrt{\frac{L}{C}}$ の場合（ルートの中が負）。
   Under damped system

[33] 導出過程は標準的な電気回路の教科書に掲載されています。

一方，微分方程式を数値計算で解くためには式 84 と式 87 を用いて二つの一階微分方程式を立てる必要があります。

$$\begin{cases} \frac{1}{C}\frac{di}{dt} + \frac{R}{L}\frac{i}{C} + \frac{1}{LC}V_C = \frac{1}{LC}E \\ \frac{dV_C}{dt} = \frac{i}{C} \end{cases}$$

1 つ目の式の両辺に C をかけて，$i \to x$，$V_C \to y$ の置き換えをします。

$$\begin{cases} \frac{dx}{dt} + \frac{R}{L}x + \frac{1}{L}y = \frac{1}{L}E \\ \frac{dy}{dt} = \frac{x}{C} \end{cases}$$

よって，

$$\begin{cases} \frac{dx}{dt} = (E - Rx - y))/L \\ \frac{dy}{dt} = \frac{x}{C} \end{cases}$$

が解くべき方程式です。変数 u, v を用いると，

$$\begin{cases} u' = (E - Ru - v)/L \\ v' = u/C \end{cases}$$

プログラムに合わせた表現では，

$$f[0] = (E - Ry[0] - y[1])/L \tag{92}$$
$$f[1] = y[0]/C \tag{93}$$

です。ヤコビアン J は，

$$J = \begin{pmatrix} -\frac{R}{L} & -\frac{1}{L} \\ \frac{1}{C} & 0 \end{pmatrix} \tag{94}$$

です。あとはこれらをプログラムに機械的に組み込むとプログラムは完成です。

ソースコード 30: GSL で RLC 直列回路を解くコード。

```
1 #include <stdio.h>
2 #include <gsl/gsl_errno.h>
3 #include <gsl/gsl_matrix.h>
4 #include <gsl/gsl_odeiv2.h>
```

```
5  #include <math.h>
6  #define RR 500.0
7  #define LL (47.0/1000)
8  #define CC (47.0/1000000000)
9  #define EE 0.0
10
11 int func(double t, const double y[], double f[],
12     void *params){
13   (void)(t); /* avoid unused parameter warning */
14   double mu = *(double *)params;
15   f[0] = (EE - RR*y[0] -y[1])/LL;
16   f[1] = y[0]/CC;
17   return GSL_SUCCESS;
18 }
19
20 int jac(double t, const double y[], double *dfdy,
21       double dfdt[], void *params){
22
23   (void)(t); /* avoid unused parameter warning */
24   double mu = *(double *)params;
25   gsl_matrix_view dfdy_mat
26     = gsl_matrix_view_array (dfdy, 2, 2);
27   gsl_matrix * m = &dfdy_mat.matrix;
28   gsl_matrix_set(m, 0, 0,  1.0/CC);
29   gsl_matrix_set(m, 0, 1,     0.0);
30   gsl_matrix_set(m, 1, 0,  -RR/LL);
31   gsl_matrix_set(m, 1, 1, -1.0/LL);
32   dfdt[0] = 0.0;
33   dfdt[1] = 0.0;
34   return GSL_SUCCESS;
35 }
36
37 int main (void) {
38   double mu = 0.0;
39   gsl_odeiv2_system sys ={func, jac, 2, &mu};
```

```
40   gsl_odeiv2_driver *d =
41     gsl_odeiv2_driver_alloc_y_new(&sys,
42           gsl_odeiv2_step_rk8pd,
43                                   1e-6, 1e-6, 0.0);
44   int i;
45   double t = 0.0, t1 = 1.0;
46   double y[2] = { 0.0, 10.0 };
47
48   for (i = 1; i <= 10000; i++) {
49       double ti = i * t1 / 10000000.0;
50       int status = gsl_odeiv2_driver_apply(d, &t, ti, y);
51
52       if (status != GSL_SUCCESS) {
53           printf ("error, return value=%d\n", status);
54           break;
55       }
56       if(i%100==0){
57   printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
58       }
59   }
60
61   gsl_odeiv2_driver_free (d);
62   return 0;
63 }
```

サンプルコードは GitHub に登録されています。

初期値は 6 行目から 9 行目で与えていて，ここでは MIT のテキストに従って，$L = 47\text{mH}$，$C = 47\text{nF}$ としています。初期電圧は 10V で，46 行目で与えています。特に $R = 500\Omega$ の場合，Under damped system に相当し，電圧が振動する様子が見られます（MIT のテキストで Figure 2 (a) に相当する）。

出力ファイルを out.txt としたとき，次の gnuplot のコマンドで表示することができます。

gnuplot > plot ”out.txt” u 1:3 w l

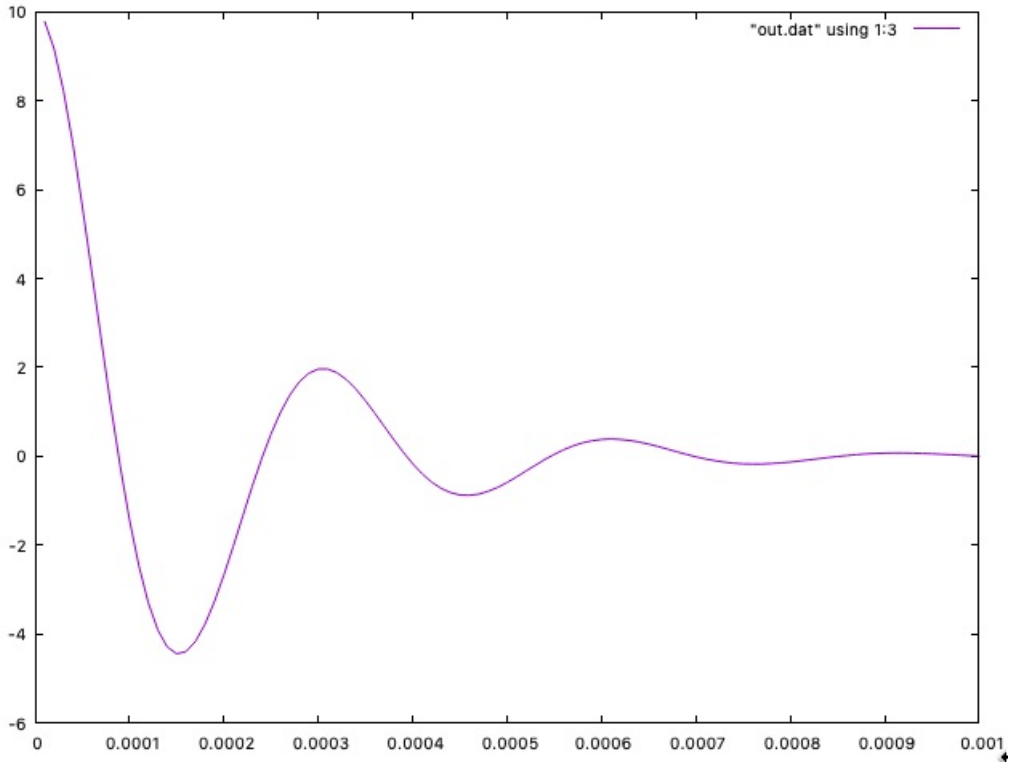

図 17: $R = 500\Omega$ における電圧特性。横軸は時間（s)。

練習 MIT の OpenCourseWare のテキストの 7 ページの Figure 4 のグラフ，および 8 ページの Figure 5 のグラフを画くようにプログラムを書き換えてみよう。

4.11.8 RLC 並列回路

RLC 並列回路の過渡現象を見てみます。以下，MIT の OpenCourseWare の p.9 以降を参照します。テキストの式 (1.33) と式 (1.30) は，

$$\begin{cases} \frac{d^2iL}{dt^2} + \frac{1}{RC}\frac{diL}{dt} + \frac{1}{LC}iL = \frac{1}{LC}I_S \\ V = L\frac{diL}{dt} \end{cases}$$

一番目の方程式に二番目の式の V を代入して一階の微分方程式にして，

$$\begin{cases} \frac{1}{L}\frac{dV}{dt} + \frac{1}{RC}\frac{1}{L}V + \frac{1}{LC}iL = \frac{1}{LC}I_S \\ V = L\frac{diL}{dt} \end{cases}$$

一番目の式の両辺に L をかけ，$V \to x, i \to y$ の置き換えをすると，解くべき方程式は，

$$\begin{cases} \frac{dx}{dt} + \frac{1}{RC}x + \frac{L}{C}y = \frac{1}{C}I_S \\ \frac{dy}{dt} = \frac{1}{L^2}x \end{cases}$$

です。変数 u, v を用いると，

$$\begin{cases} u' = (I_S - \frac{1}{R}u - Lv)/C \\ v' = \frac{1}{L^2}u \end{cases}$$

プログラムに合わせた表現では，

$$f[0] = (E - Ry[0] - y[1])/L \tag{95}$$
$$f[1] = y[0]/C \tag{96}$$

です。ヤコビアン J は，

$$J = \begin{pmatrix} -\frac{R}{L} & -\frac{1}{L} \\ \frac{1}{C} & 0 \end{pmatrix} \tag{97}$$

となります。あとはこれらをプログラムに機械的に組み込むとプログラムは完成です。

サンプルコードは GitHub に登録されています。

ソースコード 31: GSL で RLC 並列回路を解くコード。

```
1  #include <stdio.h>
2  #include <gsl/gsl_errno.h>
3  #include <gsl/gsl_matrix.h>
4  #include <gsl/gsl_odeiv2.h>
5  #include <math.h>
6  #define PIE 3.14159265
7  #define RR 20000.0
8  #define LL (47.0/1000)
9  #define CC (47.0/1000000000)
10 #define IS 5.0
11
12 int func(double t, const double y[], double f[],
13     void *params){
14   (void)(t); /* avoid unused parameter warning */
15   double mu = *(double *)params;
16   f[0] = (IS - (1.0/RR)*y[0] -LL*y[1])/CC;
17   f[1] = y[0]/(LL*LL);
18   return GSL_SUCCESS;
19 }
20
21 int jac(double t, const double y[], double *dfdy,
22       double dfdt[], void *params){
23
24   (void)(t); /* avoid unused parameter warning */
25   double mu = *(double *)params;
26   gsl_matrix_view dfdy_mat
27     = gsl_matrix_view_array (dfdy, 2, 2);
28   gsl_matrix * m = &dfdy_mat.matrix;
29   gsl_matrix_set(m, 0, 0,  1.0/(LL*LL));
30   gsl_matrix_set(m, 0, 1,          0.0);
31   gsl_matrix_set(m, 1, 0, -1.0/(CC*RR));
32   gsl_matrix_set(m, 1, 1,       -LL/CC);
33   dfdt[0] = 0.0;
34   dfdt[1] = 0.0;
```

```
35   return GSL_SUCCESS;
36 }
37
38 int main (void) {
39   double mu = 0.0;
40   gsl_odeiv2_system sys ={func, jac, 2, &mu};
41   gsl_odeiv2_driver *d =
42     gsl_odeiv2_driver_alloc_y_new(&sys,
43           gsl_odeiv2_step_rk8pd,
44                                   1e-6, 1e-6, 0.0);
45   int i;
46   double t = 0.0, t1 = 1.0;
47   double y[2] = { 0.0, 10.0 };
48   double omegazero = 1.0/sqrt(LL*CC);
49
50   for (i = 1; i <= 50000; i++) {
51       double ti = i * t1 / 10000000.0;
52       int status = gsl_odeiv2_driver_apply(d, &t, ti, y);
53
54       if (status != GSL_SUCCESS) {
55           printf ("error,␣return␣value=%d\n", status);
56           break;
57       }
58       if(i%100==0){
59   printf ("%.5e␣%.5e␣%.5e\n", omegazero*t/PIE,
60     y[0], y[1]*LL);
61       }
62   }
63
64   gsl_odeiv2_driver_free (d);
65   return 0;
66 }
```

ここではテキストに合わせて，$C = 47\text{nF}$，$L = 47\text{mH}$，$I_S = 5\text{A}$，$R = 20\text{k}\Omega$ としています（7 行から 11 行)。出力ファイルを out.txt としたとき，gnuplot のコマンドで，

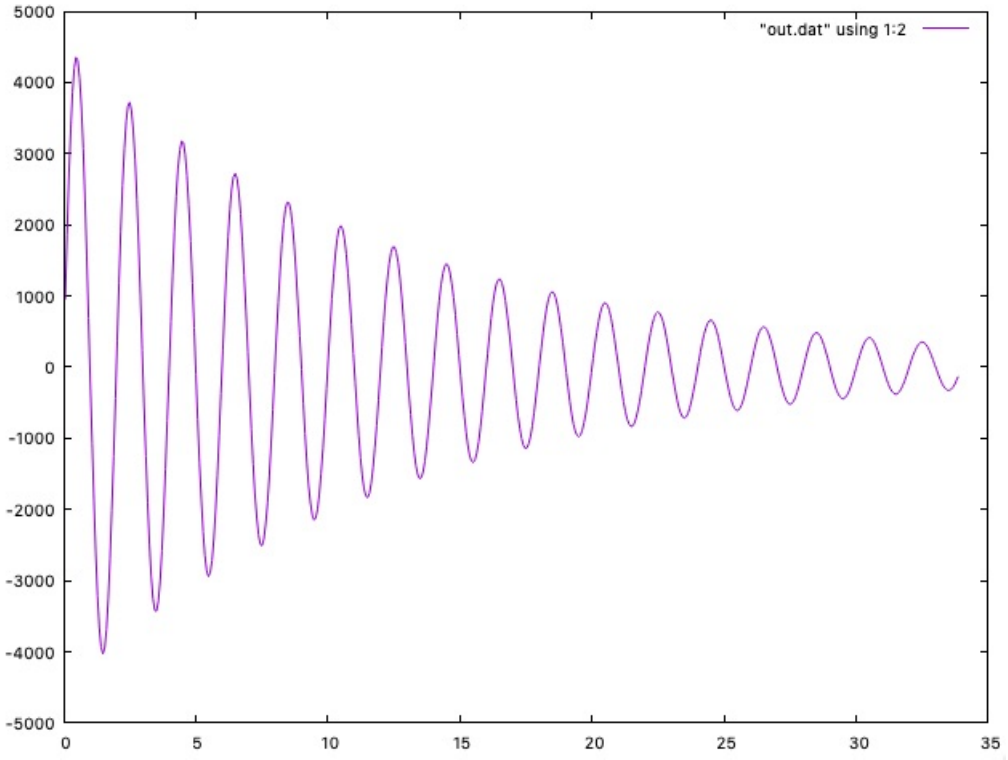

図 18: $R = 20k\Omega$ における電圧特性。横軸は時間で，$\omega_0 t/\pi$ の次元。

gnuplot > plot ”out.txt” u 1:2 w l

とすると横軸が $\omega_0 t/\pi$，縦軸が電圧 V のグラフをプロットすることができます。また，

gnuplot > plot ”out.txt” u 1:3 w l

とすると横軸が $\omega_0 t/\pi$，縦軸が電流 iL のグラフをプロットすることができます。これらは前述のテキストの p.15 のグラフに相当します。

練習 MIT の OpenCourseWare のテキストの Figure 11, Figure 12 を再現してみよう。

4.12　ゲームオブライフ

4.12.1　ルール

碁盤上の人工生命体，コンウェイのゲームオブライフ（Conway’s Game of Life）をシミュレートします。舞台は 2 次元の正方格子です。各セルは，生命体が 1 匹いる、あるいは何もいない状態にあります。注目するセルを取り囲む 8 個のセルの状態によって，次のタイムスップでのそのセルの状態がきまります。このように，あるルールによってセルの状態を変えていくシステムをセルオートマトンと言います [16]。ここでセルオートマトンの定義をしておきます。

- 各セルはなんらかの状態を持っている。
- 各セルの次のステップでの状態は，現在の近隣サイトの状態にあるルールを適用して決められる。
- 全セルに並列にルールを適用し，一度に全体の状態を更新する。

各セルが独立に状態を更新できるので並列計算に特に適したアルゴリズムと言えます。

ゲームオブライフの更新ルールです。

- 空のセルの周囲に 3 匹の生命体がいれば，そのセルに新しい生命体が誕生する。
- 生命体の周囲に，2 匹あるいは 3 匹の生命体がいるとき，そのセルの生命体はそのままとする。
- 生命体の周囲に 1 匹の生命体がいるとき，あるいはまったく生命体がいないとき，その生命体は消えて無くなる。
- 4 匹以上の生命体に囲まれた生命体は消えて無くなる。
- その他の場合はそのままとする。

プログラミングのやりかたはいろいろありますが，せっかくなので計算の役割毎に関数を分け，ファイルも分割してコードをつくってみます。

サンプルコードは GitHub に登録されています。図 19 にゲームオブライフのスナップショットを示します。ここで生命体の色は egg.c で決めています。

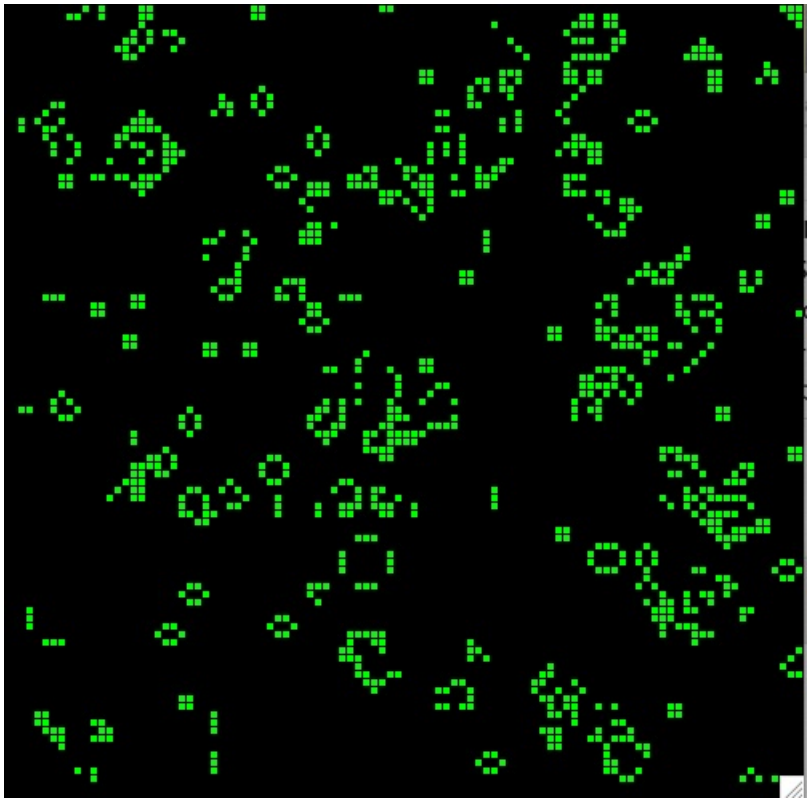

図 19: ゲームオブライフのスナップショット。

```
newcolor(win, "Green");
```

このリンク先にある X11 の「色名称」であればどれでも受け付けてくれます。
ディレクトリ program の中にあるソースコードは,

Makefile, dynamics.c, init.c, prototypes.h, rw.h, control.c, egg.c, main.c, ran.c

の合計 9 つのファイルで構成されます。この中で C のプログラムソースは**.c** で終わるファイル, 関数宣言等を記述しているヘッダファイルが**.h** で終わるファイル, プログラムをコンパイルするルールを記述しているのが **Makefile** です。
以下にメイン関数を示します。

ソースコード 32: ゲームオブライフのメイン関数（main.c）

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <time.h>
5  #include <unistd.h>
6  #include "rw.h"
7  #include "prototypes.h"
8
9  /* LIFE CA uder cyclic boundary condition */
10
11 int main(void){
12
13   get_control_param();
14   init_mem();
15
16   set_init_conf();
17   //  set_blinker();
18
19   mk_copy(sys.mat0, sys.mat1);
20
21   for(sys.time_step=0;sys.time_step<ctl.max_time_step;
22       sys.time_step++){
```

```
23      egg_disp();
24      life();
25      usleep(100000);
26      mk_copy(sys.mat1, sys.mat0);
27    }
28    return 0;
29  }
```

6 行目と 7 行目で自作のヘッダファイル rw.h と prototypes.h を読み込んでいます。rw.h には構造体などが定義さてています。protorypes.h には自作の関数の型を定義しています。例えばヘッダファイル rw.h の最初の方で次のようにして**構造体 calc_control** を定義しています。

```
typedef struct {
  int max_time_step;    /* max time step */
  int mat_size;         /* system size   */
  int shift;            /* column shift = mat_size + 4       */
  double concentration; /* initial concentration probability */
} calc_control;
```

これはタンスの設計図のようなもので，それぞれの引き出し（**メンバ**）にどのようなデータを収めるのかを決めています。今の場合，このタンスの規格の名前は calc_control です。その上でヘッダファイル rw.h の最後の方で，calc_control ctl; と宣言します。これで 4 つの引き出しをもつ calc_control の規格に則ったタンスの実態 **ctl** ができあがります。

- 構造体は沢山の引き出しをもったタンスの設計図。
- いろいろな引き出し（メンバ）を造ることができる。
- プログラム中で**構造体の名前 実体の名前;** のように宣言することで構造体の実体が生成される。

例えば,

```
calc_control ctl1;
calc_control ctl2;
calc_control ctl3;
```

と宣言すると，上記の構造を持った 3 つの構造体の実体，**ctl1**, **ctl2**, **ctl3** が生成されます。

プログラムではctl という構造体の実体を作りました。その中のメンバmax_time_step を参照するときは，ピリオドを使います。まとめると，構造体の実体のメンバにアクセスするときは下記のようにします。

- 「構造体の実体 . メンバ名」とピリオドを使う。

プログラムではピリオドを用いてメンバを参照しています。ただしポインタに結びつけられた構造体の場合，第 4.1.1 節で見たようにポインタが構造体である時のメンバにアクセスするときはアロー演算子 $->$ を使います。

シミュレーションでの初期計算パラメータは control.c ファイルで設定しています。具体的には 11 行目で最大計算タイムステップ，12 行目でシステムサイズ，13 行目で初期配置における生命体の存在濃度を設定しています。

ソースコード 33: パラメータ設定関数（control.c）

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <time.h>
5  #include "rw.h"
6  #include "prototypes.h"
7
8  void get_control_param(void){
9    long time_dumy;
10
11   ctl.max_time_step = 20000;
12   ctl.mat_size = 100;
13   ctl.concentration = 0.2;
```

```
14
15    ctl.shift = ctl.mat_size+4;
16
17    /* set a "seed" for random number generator */
18    sys.random_seed = (long *)calloc(1, sizeof(long));
19    *sys.random_seed=-time(&time_dumy);
20  }
```

rw.h ではもう一つ構造体を作っていて，こちらは system_property という名前の設計図です。最後の方で同じく system_property sys; として宣言し，構造体の実態 **sys** を作ります。構造体 sys の二番目のメンバには int 型の変数 time_step が収められていて，それを参照するときは，sys.time_step とします。ここで sys.time_step は現在のタイムステップで，main 関数内で for ループを使った繰り返しのために用います。

つぎに構造体の system_property では，

```
int *mat0;
int *mat1;
int *mat2;  /* not used */
```

として**整数型のポインタ** mat0, mat1, mat2 を宣言しています。これらは 2 次元正方格子での各セルの値を保持するマトリックスとして使います。init.c にある init_mem() 関数で，**calloc を使って動的にポインタの記憶領域を割り当てています。**

```
void init_mem(void){
  int *mat_mem0, *mat_mem1, *mat_mem2;
  int n;

  n = ctl.mat_size+4;

  mat_mem0 = (int *)calloc(n*n, sizeof(int));
```

```
  mat_mem1 = (int *)calloc(n*n, sizeof(int));
  mat_mem2 = (int *)calloc(n*n, sizeof(int));
  sys.mat0 = mat_mem0;
  sys.mat1 = mat_mem1;
  sys.mat2 = mat_mem2;
}
```

割り当てている領域は **n*n** です。ここで **n = ctl.mat_size+4;** としているのは，あとで周期的境界条件の計算に使うために正方格子の外枠を 2 列の予備のセルで覆う必要があり，反対側と併せて 4 列補ってやります。[34]

セルの更新ルールは dynamics.c にある関数 life() で定義します。

```
    nn_sum = n+e+s+w+ne+se+sw+nw;
    if(x==0 && nn_sum==3) y = 1;        /* rule set 1 */
    else if(x==1 && nn_sum==2) y = 1;   /* rule set 2 */
    else if(x==1 && nn_sum==3) y = 1;   /* rule set 2 */
    else if(x==1 && nn_sum==0) y = 0;   /* rule set 3 */
    else if(x==1 && nn_sum==1) y = 0;   /* rule set 3 */
    else if(x==1 && nn_sum>=4) y = 0;   /* rule set 3 */
    else y=x;
```

ここで n, e, s, w はそれぞれ北，東，南，西の再隣接セルの値（0 か 1）で，ne, se, sw, nw はそれぞれ北東，南東，南西，北西の再隣接セルの値です。これらの再隣接セルの値を足し合わせて **nn_sum** に代入します。その上で今注目しているセル **x** と **nn_sum** の値から，次のステップでの値 **y** を決めています。

このままでは計算が速く進みすぎるので，main.c の 25 行目で，**usleep(100000)** を指定し，for ループ実行時に毎回 $1.0 \times 10^5 \mu$sec 中断するようにしている。この値は実行する PC の能力に合わせて変更すると良い。

練習 make をしてみよう。ターミナルで program のディレクトリに移動し，**make** とコマンドを打つとファイルのコンパイルがなされ，最終的にアプリケーショ

[34] 右端と左端で 4 列，上端と下端で 4 列です。

ン **life** ができあがります。実行には現在のディレクトリ名をつけて ./life とします。

```
$ make
$ ./life
```

また **make clean** とコマンドを打つとコンパイルされていたファイル **.o** の全てが消去され，次回の **make** では全てのソースファイルが一から再コンパイルされます。

練習 control.c では，最大タイムステップ，マトリックスサイズ，最初の段階で life によって占有されたサイトである確率（concentration，濃度）が定義されている。これらを書き換えてみよう。書き換え終わったらそのファイルを保存します。**make** とコマンドを打つと，変更のあったファイルのみ再コンパイルしてくれます。

4.12.2 マウス入力

次にマウスでなぞって生命体の初期設定ファイルを作ります。以下のプログラムはディレクトリ initconf の中にある gm1.c です。

ソースコード 34: マウスで初期設定ファイルを作るコード gm1.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <eggx.h>
4  FILE *fpout;
5  #define SCALE 20
6  #define NX 100
7  #define NY 100
8
9  int main(void){
10   int *mat;
11   int i, j, cnt=0;
12   int win,b ;
13   int ix, iy;
14   float x=0,y=0 ;
15
```

```
16    mat = (int *)calloc(NX*NY, sizeof(int));
17    win = gopen(NX*SCALE,NY*SCALE) ;
18    fpout = fopen("./position.txt","w");
19
20    newcolor(win, "Black");
21    gclr(win);
22
23    newcolor(win, "Yellow");
24    while( 1 ){
25      int type ;
26      if ( ggetxpress(&type,&b,&x,&y) == win ) {
27        if( type == ButtonPress ){
28
29    ix = x/SCALE; x = (double)ix*SCALE;
30    iy = y/SCALE; y = (double)iy*SCALE;
31
32    fillrect(win,x,y,SCALE,SCALE);
33    printf("button=%d␣x=%g␣y=%g\n",b,x,y) ;
34    mat[ix*NX+iy] = 1;
35    cnt++;
36    if(cnt>10) break;
37        } else if( type == KeyPress ){
38    if( b=='q' ) break ;
39    printf("key␣code␣=␣%d\n",b) ;
40        }
41      }
42    }
43
44    for(i=0;i<NX;i++){
45      for(j=0;j<NY;j++){
46        printf("%2d", mat[i*NX+j]);
47        fprintf(fpout,"%2d", mat[i*NX+j]);
48      }
49      printf("\n");
50      fprintf(fpout,"\n");
```

```
51    }
52    fclose(fpout);
53    gclose(win);
54    free(mat);
55    return 0;
56  }
```

18 行目でマウスでマークしたセルを書き込むファイル **position.txt** を開いています。26 行目，27 行目でマウスボタンが押されたときの座標を読み取ります。34 行目でマウスが押されたセルを記録しています。最後に 47 行目で全体のセルのデータを書き込んでいます。コンパイルには **egg** コマンドを使用します。

```
%  egg gm1.c
```

実行時に画面をマウスを押しながらなぞると，その箇所の色が変わります。マウスによる入力を終了するためには「q」キーを押します。マウスでマーカーをつけた箇所が 1，マーカーをつけない箇所が 0 の 100×100 の行列がファイル **position.txt** に書き出されます。

ゲームオブライフの実行時に初期設定としてこの **position.txt** を読み出すようにすれば良いわけです。

初期設定の読み出しには main.c で呼び出している set_blinker を使うことにします。直前の初期設定関数 set_init_conf() をコメントアウトして，set_blinker を引数無しに呼び出すように設定します。

```
   // set_init_conf();
   set_blinker();
```

init.c における set_blinker() は，

```
void set_blinker(){
  int i, j;
```

```
  int *field=sys.mat0;

  FILE *fp = fopen("../initconf/position.txt","r");
  for(i=2; i<ctl.mat_size+2; i++){
    for(j=2; j<ctl.mat_size+2; j++){
      fscanf(fp, " %2d", &field[(ctl.mat_size+4)*i+j]);
      //      printf(" %d",field[(ctl.mat_size+4)*i+j]);
    }
    //    printf("\n");
  }
  //  printf("\n");
}
```

としています。この関数では，ファイルへのポインタを定義し，現在のディレクトリ initconf にあるファイル position.txt を読み出しモードで開いています。ファイルの読み出しには fscanf を使っています。

練習 マウスで life を設計して実行してみよう。

4.12.3 データ入力

Game of Life には沢山の生命体がいます。これまでに見つかった生命体は Lexicon (辞典) にまとめられています。 Lexicon にも色々あって，記述されている生命体のフォーマットも異なります。ここでは，Stephen Silver 氏の Life Lexicon を参考にすることにします。リンク先をみると生命体は. と O のテキストで記述されていることがわかります。そのデータを持ってきて自分の life で実行させることを考えます。

- ターミナルでディレクトリ initconf2 に移動します。
- Lexicon から生命体を 1 匹選びます。
- ピリオドと O（オー）のテキストでできた「生命体」の部分のみをまるまるマウスで選択しコピーします。普通はマウスの左ボタンを押しながら選択するとコピーできます。

- ターミナル上で次のコマンドを実行します。

```
% cat > data.txt
```

ここでリターンキーを押すとターミナルが入力待ちになります。つまりターミナルからそのまま入力することで data.txt というファイルを造ることが出来ます。

- マウスでペーストします。マウスの真ん中ボタンを押すとペーストされます。
- これでファイル入力がおわったのでコントロールキーと D を押します。
- 次はこのディレクトリにあるパールスクリプト lex2pos.pl を今作ったファイル data.txt を引数にして実行します。

```
% ./lex2pos.pl data.txt
```

- これで position.txt ができました。
- あとは life の init.c で,

```
FILE *fp = fopen("../initconf/position.txt","r");
```

となっているところを

```
FILE *fp = fopen("../initconf2/position.txt","r");
```

とすれば実行時にディレクトリ initconf2 のファイルを読み取ってくれます。

ソースコード 35: パールスクリプト lex2pos.pl

```
1 #!/usr/bin/perl
2 #
3 # evaluate the nearest neighbors
4 #
5 my $cnt=0;
```

```
my $xscale = 100;
my $yscale = 100;
$file=$ARGV[0];
open(FL1, $file) || die "I␣can't␣open␣QE␣input␣file.␣\n";
open(POS, ">␣position.txt") || die "Error:$!";
while(<FL1>) {

   $_=~s/=/ /;
   @lines = split(/\^|\s+/, '␣'.$_);
   $num=length($lines[1]);
   my @lifes=split(//,$lines[1]);
   for($i=0;$i<$num;$i++){
       if($lifes[$i] eq ".") {print "␣0";}
       if($lifes[$i] eq "O") {print "␣1";}
   }
   for($i=$num;$i<$xscale;$i++){
       print "␣0";
   }
   print "\n";
   $cnt++;
}

for($j=$cnt;$j<$yscale;$j++){
    for($i=0;$i<$xscale;$i++){
  print "␣0";
    }
    print "\n";
}
```

パールスクリプトの lex2pos.pl がやっていることは，データファイルを読み取ってピリオドをゼロに，アルファベット大文字のオーを 1 に変換し，それを 100 × 100 の行列に埋め込んで出力するということです。たとえば元のデータファイルが 30 × 35 であっても行列の左下隅に置いて他はゼロで埋め尽くしています。

練習 Lexicon から生命体を選んで自分の life で実行してみよう。

4.13　エキサイト CA

規則正しい心臓の鼓動，脳におけるシナプスの反応，これらの活動を支えるリズムはどこから来ているのでしょうか。これは何も生命現象に限ったことではありません。例えばベロウソフ・ジャボチンスキー反応と呼ばれる化学反応では，シャーレ内の薬品が互いに反応拡散を繰り返して，延々とパターンを描きます（ビデオはここで見ることができる）。そのようなリズムを刻むことができるセルオートマトンモデルに取り組みます。

Greenberg-Hastings モデルは以下のように定義されます [17, 18, 19]。

- もしも $\xi_n(x) = i > 0$ ならば，$\xi_{n+1}(x) = i + 1$。ただし κ の modulo とする。
- もしも $\xi_n(x) = 0$ で，少なくても θ 個の近隣セルが 1 ならば，$\xi_{n+1}(x) = 1$ となる。それ以外は $\xi_{n+1}(x) = 0$。

非常にシンプルでこれが複雑な振動パターンを生むとは，ちょっと信じられないかもしれません。まず，$\xi_n(x)$[35]は，サイト x における n タイムステップでの状態です。それが整数で 0 より大きければ 1 つ増やすということです。κ の modulo ですから，κ を法にすると言うことで，これはつまり κ で割ったあまりです[36]。ここで 0 は休眠状態に，1 はエキサイト状態に相当します。はじめ値が 1 でエキサイトしたセルがあったとして，タイムステップ毎に値を 1 つ増やしていき，κ になった段階で値がゼロとなり，休眠します。ですので κ の役割はセルの時計で，エキサイトしてから何ステップ後に回復して休眠するかを決めています。つまりエキサイト状態からのリカバリーの時間を決めています。二つ目のルールが言っていることは，値が 0 の休眠中のセルは，その近隣にエキサイト状態になったばかりの（値が 1 の）セルが θ 個以上ならば，自分もエキサイト状態になる（値が 1 になる）ということです。これはセル同士の接触による興奮の伝播を表しています。

次に問題になるのは「近隣のセル」の定義です。Fisch らの論文ではダイヤモンド型（D）か箱形（B）で分け，かつ中心サイトからの距離で区分けしています [20]。例えば $\rho = 1D$ での近接サイトは，中心からの距離が 1 以内でダイヤモンド型なので N, S, E, W（北，南，東，西のアルファベット）の 4 個。同様に $\rho = 1B$ での近接サイトは，中心からの距離が 1 以内で箱型なので上記に NE, NW, SE, SW の 4 個を加えた計 8 個となります。Durrett らの論文では，もっと一般化して中心のサイトからの半径 ρ 以内に含まれるサイトを近隣サイトと呼んでいます [19]。論文の Figure 1 では $\rho = \sqrt{20}$ の半径内にあるセルを近接サイトと呼んでいます。ここで $\rho = \sqrt{(\delta x)^2 + (\delta y)^2}$ であり，δ は中心セルからの距離です。図 20 に $\rho = \sqrt{20}$

[35] ξ はギリシャ文字のグザイ。
[36] κ はギリシャ文字のカッパ。

		nw1	nw2	n4	ne1	ne2		
	nw3	nw4	nw5	n3	ne3	ne4	ne5	
nw6	nw7	nw8	nw9	nn	ne6	ne7	ne8	ne9
nw10	nw11	nw12	nw	n	ne	ne10	ne11	ne12
w4	w3	ww	w	X	e	ee	e3	e4
sw1	sw2	sw3	sw	s	se	se1	se2	se3
sw4	sw5	sw6	sw7	ss	se4	se5	se6	se7
	sw8	sw9	sw10	s3	se8	se9	se10	
		sw11	sw12	s4	se11	se12		

図 20: $\rho = \sqrt{20}$ の近接サイト。

の近接サイトを示します。たとえば w3 のセルは $\sqrt{4^2} = \sqrt{16} < \sqrt{20}$ なので範囲内ですが，その左となりになるとこの制限を超えてしまいます。また nw3 のセルは，$\sqrt{3^2 + 3^2} = \sqrt{18} < \sqrt{20}$ なので範囲内ですが，このセルの上も左となりも範囲外となります。つまり図の近隣サイトの中に興奮したばかりのサイトが θ 個あれば，0 で休眠中の中心のセル x がエキサイトセル（1）になることができます。

サンプルコードは GitHub に登録されています。表示があまりにも急速に変化してしまい画面がちらつく場合は，main において usleep をつかって強制的に計算を遅らせるなど環境に合わせて適切に調整して下さい。

ここで θ と κ は control.c の，get_control_param 関数で設定します。

```
ctl.mat_size = 400;

ctl.theta = 9;
ctl.kappa = 6;
```

デフォルトの設定は 400×400 の正方格子で，$\theta = 9$，$\kappa = 6$ で，これは前述の Durrett らの論文の Figure 1 の左下の図に相当します（図 21）[37]。

練習 Durrett らの論文の Figure 1 に相当するシミュレーションを実施せよ。

[37] 論文では 240×240 の正方格子ですが，この図は 400×400 の正方格子の場合を表しています。

図 21: デフォルトの設定におけるエキサイト CA のスナップショット。

練習 $\rho = 2D$ および $\rho = 2B$ の場合におけるエキサイト CA を実施せよ。パラメータ依存性に変化はあるか。

4.14　伝染病 CA

ランダムウォーク・セルオートマトン（RWCA）で伝染病のモデルをつくりましょう[38]。

- 各 walker はランダムに歩き回る。
- １名以上の感染者を接触した walker は接触感染する。このときの接触の定義は，上下左右に斜め方向も含めた合計 8 個の隣接サイトとする。
- 感染者はそこに留まる。
- 感染した場合に健康値を $-n$ とする。例えば -10。
- 感染者は，系のタイムステップをアップデートする度に健康が 1 回復していく。
- 健康値が -1 となったときに完全に回復したと見なす。
- 健康が回復した感染者は通常の walker となる。

サンプルコードは GitHub に登録されています。

プログラムでは，初めに全マス目の 20%に walker を配置します。そのためソースコードの control.c で，

```
ctl.concentration = 0.2;
```

としています。また初期状態において walker のうちの 80%が感染している設定にします。

```
ctl.inf_init_rate = 0.8;
```

感染者を「赤」で表示し，感染したときの状態を -5 とします。

[38] RWCA については付録の第 A.1 節を見よ。

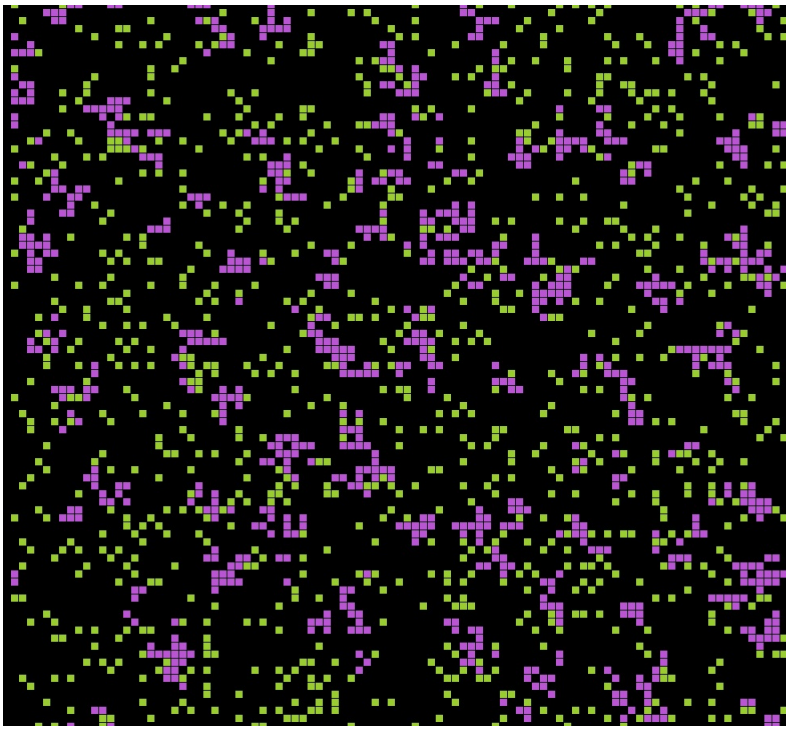

図 22: 伝染病 CA のスナップショット。緑が健康な walker，紫が感染者。

```
    ctl.inf_illness   = -5;
```

この感染状態は系のタイムステップと共に 1 つ増えていきます。-1 になった段階で，次のステップで感染状態を脱します。また感染者は回復するまでは移動しないこととする。make と実行は次のようにします。

```
$ make
$ ./infect
```

練習 プログラムが正確に計算しているかどうか確かめてみよう。files ディレクトリに出力されるファイル out をチェックする。gnuplot を起動し，

```
gnuplot> plot "out" using 1:2 w l, "out" using 1:3 w l
```

とすれば，タイムステップ対感染者，タイムステップ対非感染者のグラフが

得られる。直接ファイル out を確認するには，例えばコマンド head（ファイルの先頭を表示）で，

```
% ls
out
% head out
0  1587  407 1994
1  1528  466 1994
2  1514  480 1994
3  1514  480 1994
4  1528  466 1994
5  1532  462 1994
6  1597  397 1994
7  1605  389 1994
8  1605  389 1994
9  1620  374 1994
..........
..........
..........
```

などとすれば良い。4 カラム目の数値が感染者と非感染者を会わせた数で，各タイムステップで一定（今の場合は総数 1994）となり正しく計算されていることがわかります。

練習 初期の walker の濃度，初期の感染率，及び感染したときの状態の重さ (マイナスの値の大きさ)，および系自体の大きさによっては次の 3 パターンに落ち着くと考えられる。(1) 全て回復し感染者はゼロ。(2) 感染者と非感染者が共存し，ほぼ一定値に落ち着く。(3) 感染者と非感染者の数がタイムステップで周期的に振動する。(3) の状態にするにはどのようなパラメータを設定すれば良いだろうか[39]。周期的に変化している様子を gnuplot で確認してみよう。

[39] パラメータは control.c で設定しています。

4.15 DLA

Witten らが提案した拡散律速凝集（diffusion-limited aggregation: DLA）モデルを扱います [21]。2 次元の舞台の中心に「種」を置きます。遠く離れた所から粒子を放出します。粒子はランダムにこの舞台の上を歩き回ります（ランダムウォーク）。種の隣にたどり着いたとき，その粒子は種の一部となってそのサイトに留まります。再び遠く離れた，ランダムに選択された所から粒子を放出します。同じ過程をくりかえします。中心に据えた種が生長していくにつれ，沢山の枝が生えてきます。自己相似性の形態をもった，フラクタルなパターンが生成されます。DLA は適用範囲が広く，結晶成長のほか，絶縁破壊などにも同様な構造を見ることができます [22, 23, 24]。

ランダムウォークと言うことですから，我々の RWCA を直接応用できそうです。「種」は重度感染者で，隣に来て感染すれば回復しない，すなわちそこに留まることにすれば良い。Witten らのモデルは 1 度に 1 つの粒子がランダムウォークで種に到着するモデルでした。我々の RWCA は多粒子が同時にランダムウォークするのでシミュレーションの効率は良さそうです。

- 中心に種を置く。種の状態は -1 とする。
- ある濃度 p になるよう系全体に多数の粒子を置く。
- 各粒子はランダムウォークで系を移動する。
- 種と接触した粒子はそこに留まり，種の一部と化す。ここで接触の定義は，上下左右に斜め方向も含めた合計 8 個の隣接サイトに種があるときとする。
- 種の一部と化したときのタイムステップを t とすると，その種の値を $-t$ としておく。

プログラムは GitHub からダウンロードできます。

練習 図のバックグラウンドカラーと，表示される色のグラディエーションを変えてみよう。

練習 松下らの解説の p.477 の図 3 は亜鉛金属森である [22]。このように方向性のあるパターンの DLA を実現するためにはどうしたら良いだどうか。

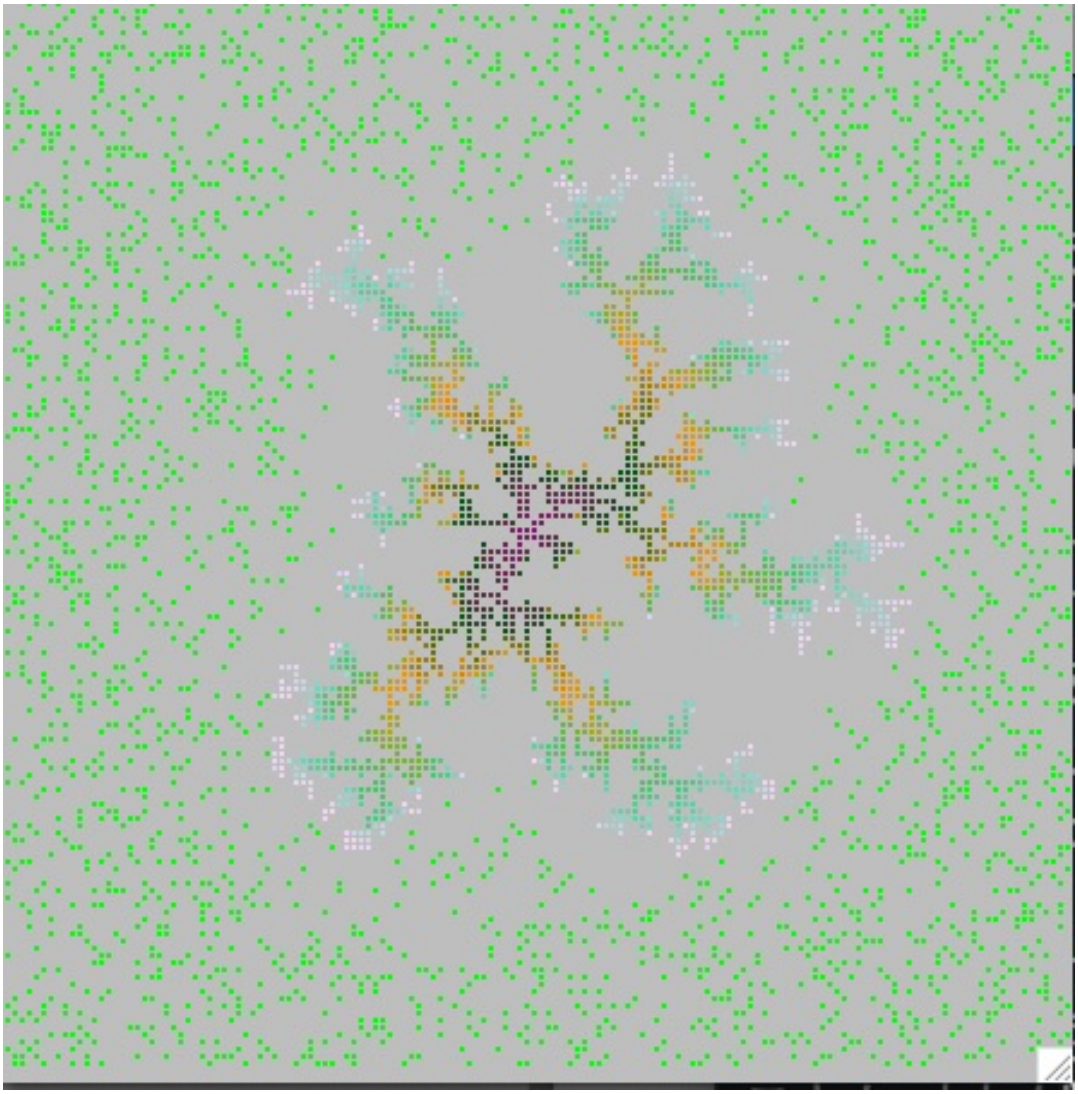

図 23: RWCA のダイナミクスによる DLA シミュレーションのスナップショット。緑がランダムウォーカー。中心から周囲に広がるパセリの枝葉のようなものが固定化されたウォーカーで，成長過程にあるクラスターを形成している。

4.16　プレデター CA

捕食者（predator: プレデター）と被食者（prey: プレイ）がいて，そのバランスの上に自然界は成り立っています。この弱肉強食の世界を RWCA でシミュレートします。

- 2 次元のマス目上にプレデターとプレイを配置する。
- RWCA のダイナミクスに従って移動させる。
- 移動の度にそれぞれのエネルギー値が 1 減る。
- プレデターに隣接するプレイは，ある確率でつかまえられて食べられ，その存在が消滅する。この場合の隣接とは上下左右の 4 つの隣接サイトとする。
- プレデターは他のプレデターと出会うとある確率でその子を産む。この場合の出会うとは上下左右の 4 つの隣接サイトにいる時とする。
- プレイは他のプレイと出会うとある確率でその子を産む。

はたして，この世界で一体何が起こるでしょうか。「持続可能な社会」は可能でしょうか，というのがこのシミュレーションのテーマです。サンプルプログラムです。プログラムは GitHub に登録されています。。やや系が複雑ですので，設定するパラメータが多くなります。ソースファイル control.c から抜粋します。

```
ctl.concentration = 0.05;  /* initial concentration probability */
ctl.p_of_predator = 0.10;  /* percentage of the predators */

ctl.predator_energy = 10;  /* initial energy of predator */
ctl.prey_energy = 20;      /* initial energy of prey */

ctl.predator_reborn = 0.03; /* reborn probability of predator */
ctl.prey_reborn = 0.05;     /* reborn probability of prey */
ctl.increased_vitl = 4;     /* vital is increased by eating prey */
```

ctl.concentration は初期における walker の存在密度を指定しています。**ctl.p_of_predator** は，walker のうち捕食者である確率を指定しています。上の例では初期の段階

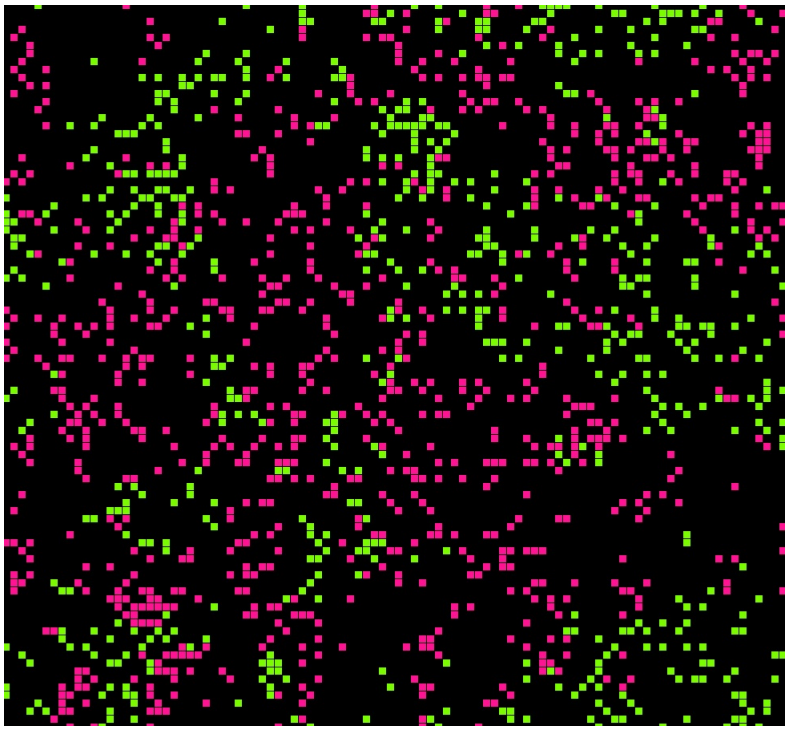

図 24: プレデター CA のスナップショット。このタイムステップでは赤い捕食者の数が優勢。

で全マス目の 5 %に walker がいて，そのうちの 10 % がプレデターとなります。**ctl.predator_energy** はプレデターの持っているエネルギー値で，1 タイムステップ毎に減っていきます。**ctl.prey_energy** はプレイの持っているエネルギー値で，1 ステップ移動する度に減っていきます。エネルギー値がゼロになるとプレデターもプレイも餓死します。
main.c では，

```
count_walkers();
breeding();
energy_consumption();
hunt();
```

としていて，walker を数え上げる関数 count_walkers()，繁殖をさせる関数 breeding()，エネルギーを消費させる関数 energy_consumption()，プレデターが狩りをする関数の hunt() 順に呼び出します。これらは dynamics.c に定義されています。

練習 プログラムが正確に計算しているかどうか確かめてみよう。files ディレクトリに出力されるファイル out をチェックする。gnuplot を起動し，

```
gnuplot> plot "out" using 1:2 w l, "out" using 1:3 w l
```

とすれば，タイムステップ対プレデター，タイムステップ対プレイのグラフが得られる。

練習 ロトカ・ヴォルテラの方程式（第 4.9 節）のような周期的振動が得られるパラメータセットを探そう。そのためには初期条件を固定する必要がある。よって毎回決まった初期配置にするために control.c で *sys.random_seed に定数を設定する。その上でプレデターとプレイの数が振動するようなパラメータセットを見つけると良い。

練習 新しいルールを作ろう。例えば「たまに」プレイがプレデターを殺す。

練習 新しいルールを作ろう。プレイは足が遅いとする。これは例えば，プレイの場合，50 %の確率で新しい方向を向かないとすれば良い。

練習 オリジナルのルールを作ろう。

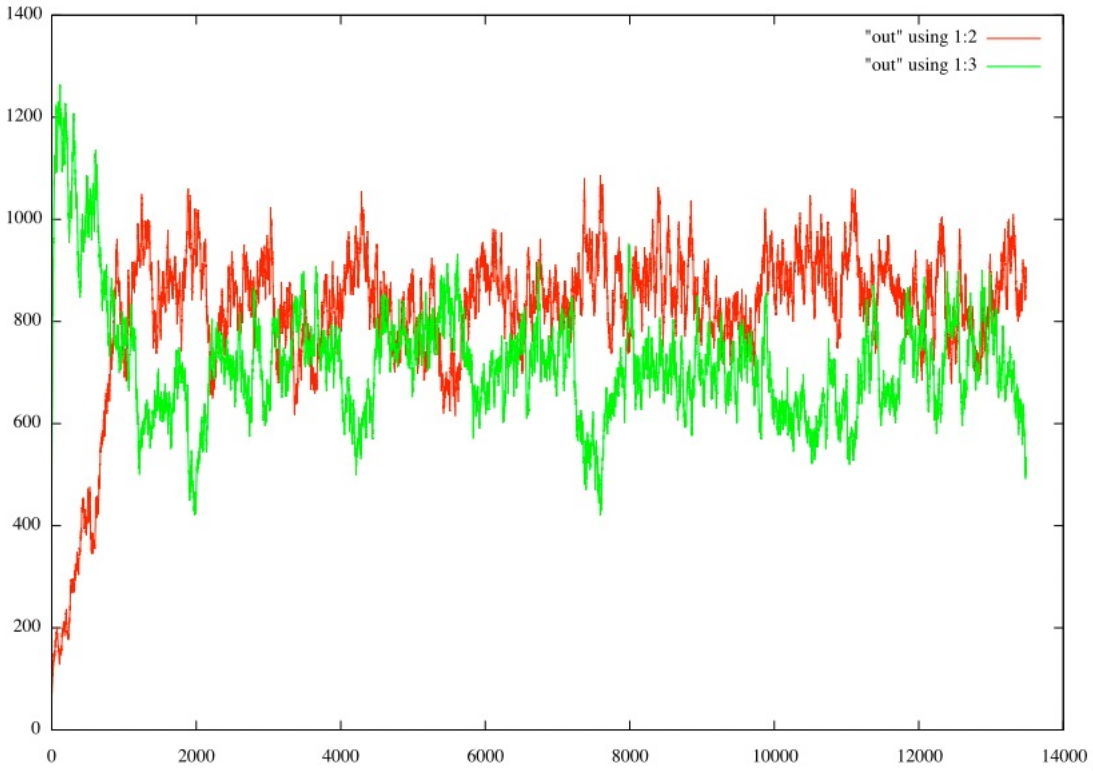

図 25: プレデター CA での捕食者（赤）と被食者（緑）の頭数の時間変化。

4.17　交通流 CA

4.17.1　１車線モデル

渋滞は嫌なものです。普段なら 10 分もあればつく距離が 1 時間もかかることもあります。そのような状況を CA でシミュレートしてみましょう。ここでは Nagel と Schreckenberg によって導入された 1 車線 CA に取り組みます [25]。論文の 2222 ページにアップデートのルールが定義されています。

1. **加速**：もしも車の速度 v が $v_{\max}$ よりも低くて，かつ前方の車までの距離が $v+1$ よりも大きかったならば，速度は 1 加速する［$v \to v+1$］。
2. **減速（他の車のために）**：もしもサイト i にいる車の前方のサイト $i+j$ (ただし $j \leq v$) に他の車がいるならば，サイト i の車のスピードは $j-1$ に減速する［$v \to j-1$］。
3. **ランダム化**：確率 p で，（速度が 1 より大きな）車は 1 減速する［$v \to v-1$］。
4. **車の移動**：それぞれの車は v サイト前進する。

1 番目の加速ルールが言っていることは，「前方が空いていたら，衝突しない程度まで加速していいよ」と言うことです。2 番目の減速ルールが言っていることは「前方に車が迫ってきたら，衝突しないように減速しましょう」と言うことです。3 番目は「ランダムにアクセルをゆるめる人がいる」です。それぞれの車が衝突しない範囲で加速と減速をして速度が定まり，4 番目のステップで車を移動させます。このアルゴリズムは RWCA の 1 次元版のコードを元にすると比較的容易に実現できます。プログラムは GitHub に登録されています。 プログラムを実行すると 1 車線の道路が画面に表示されます。道路自体は黒で，速度によって色分けされています。ここで白は速度 0 です。プログラムでは main.c で，

```
    usleep(300000);
```

としていて，強制的にゆっくりとした計算にしています。これは表示される egg の画面で，確かに車が上記の 4 ステップに従って移動していることを確かめるためです。

プログラムの内部構造的には，１次元の配列を作り，そこに車があれば 1 以上の値にしています。またその値から 1 を引いたものを速度としています。つまり

図 26: 交通流 CA の表示。車は左からやってきて右に抜ける。車の速度で色分けをしている。車が存在しないところは黒。

1 ならば速度ゼロの車がそこにあることになります。control.c でパラメータを設定しています。

```
ctl.max_time_step = 10000; // 計算ステップ数
ctl.mat_size = 50;         // 道路の長さ
ctl.concentration = 0.1;   // 道路における車の密度
ctl.max_velocity = 3;      // 車の最大速度
```

また，main.c では以下のように 4 段階で系をアップデートしています。

```
accelerate();  // step 1
decelerate();  // step 2
rand_decelerate(); //step 3
move_tf_1d();  // step 4
```

この系ではある車の密度において交通渋滞が観測されます。図 27 に示したのは道路の様子の時系列変化です。時間が進むにつれ，車の進行方向と逆に白い尾根が流れていくように見えます。白いところは速度ゼロの車です。つまり渋滞が引き起こされて，それが後方に伝播していく様子が見えているのです。

練習 図 27 は，main.c で，

```
egg_traffic();
```

を有効にすると表示される画面のスナップショットです。密度や最大速度などの設定パラメータを色々変えて渋滞の後方伝播の図を作りましょう[40]。

[40]計算を速くするために main.c での usleep をコメントにすること。

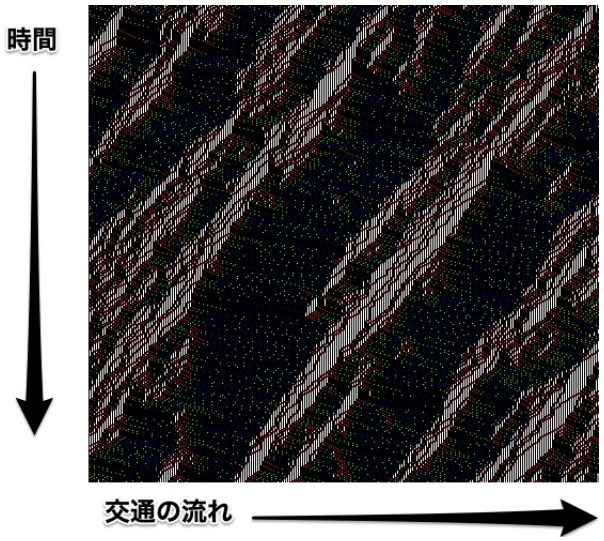

図 27: 交通流ＣＡの時間変化。縦軸は上から下方向に時間経過を表す。横軸は交通の流れで左からやってきて右に抜ける。白は速度 0 の車。

練習 車の平均の速度を計算する関数を作り，その時間変化を外部ファイルに書き出すようにしてみましょう。

練習 Nagel らの論文の Fig.4 は，横軸に車の密度をとり，縦軸に１ステップ当たり，車１台あたりの平均速度をプロットしたものである。同じようなグラフがつくるためには，プログラムをどう書き変えたら良いだろうか。

4.17.2　２車線交通流ＣＡモデル

Nagel と Schreckenberg の１車線交通流モデルを基本として，日本の道路事情に合わせた，2 車線の高速道路ＣＡを組み立てましょう。普通は左が一般走行車線で右が追い越し車線です。追い越し車線を走り続けてはいけません。左車線を走っていて前方に車が迫ってきたとき，かつ右車線に余裕があるならば，右車線に移り追い越しをかける。追い越しが終わった車は，左車線に余裕があるときには速やかに左車線に戻ります。これらの事項を元に，2 車線の高速道路ＣＡシミュレーションのモデルを組み立てましょう。

1. **左車線の車の車線変更**：もしも左車線のサイト i，速度 v の車の前方の $j(j \leq i+v)$ までの区間に他の車が存在するとき，右車線の i から $i+v$ まで車が存在しないならば，確率 p_l で右車線にうつる。

2. **右車線の車の車線変更**：もしも右車線のサイト i，速度 v の車の前方の $j(j \leq i+v)$ までの区間に他の車が存在するとき，左車線の i から $i+v$ まで車が存在しないならば，確率 p_r で左車線にうつる。

3. **左車線加速**：もしも左車線の車の速度 v が $v^{\max}$ よりも低くて，かつ前方の $r+v+1$ まで他の車が存在しなかったら，速度を 1 加速する $[v \to v+1]$。

4. **右車線加速**：もしも右車線の車の速度 v が $v^{\max}$ よりも低くて，かつ前方の $r+v+1$ まで他の車が存在しなかったら，速度を 1 加速する $[v \to v+1]$。

5. **左車線減速**：もしも左車線の速度 v の車の前方のサイト $j(j \leq i+v)$ に他の車がいるならば，サイト i の車は，速度を $j-1$ に減速する。

6. **右車線減速**：もしも右車線の速度 v の車の前方のサイト $j(j \leq i+v)$ に他の車がいるならば，サイト i の車は，速度を $j-1$ に減速する。

7. **ランダム化**：確率 ps で，（速度が 1 より大きな）車は 1 減速する $[v \to v-1]$。

8. **車の移動**：それぞれの車は v サイト前進する。

以上を 4 ステップで実行します。

$$1 \to 2 \to (3,4,5,6,7) \to 8 \tag{98}$$

ここで括弧内は同時に処理する項目を表しています。プロトタイプのプログラムは GitHub からダウンロード出来ます。control.c で設定されるデフォルトの初期設定について述べます。

図 28:　2 車線の交通流ＣＡのスナップショット。上段が左車線，下段が右車線（追い越し車線)。上段には白の速度 0 の車が多い。

- 50 サイト，２車線が設定される (ctl.mat_size = 50)。
- 初期の車の存在確率は 0.2 (ctl.concentration = 0.2)。
- 最高速度は 5 セルの移動（ctl.max_velocity = 5)。

program のディレクトリで make をするとアプリケーション rw ができます。これを実行するには，端末の画面で，

```
$ ./rw
```

とします。

　計算後，files のディレクトリに，タイムステップでの各車線の車の台数を記録したファイル，files/cars ができあがります。タイムステップ毎の右車線にいる車の台数，左車線にいる車の台数，全体の台数をプロットするには，files のディレクトリで gnuplot を立ちあげ，

```
gnuplot¿ plot [0:2000][]"cars" using 1:2 w l,"cars" using 1:3 w l,"cars" using 1:4 w l
```

とします。

　ソースコードの main.c を見ると，最初の方に，

```
  lane_change(sys.mat0,sys.mat0b,0.2);    // step 1
  lane_change(sys.mat0b,sys.mat0,0.9);    // step 2
```

としている行があります。始めの lane_change 関数は，確率 0.2 (最後の引数で指定) で左車線から右車線に移動することを指示しています。同様に次の lane_change 関数は確率 0.9 で右車線から左車線に移動することを指示しています。したがって自然と左車線を通行する車の台数が多くなります。

練習　右車線を走る車の平均速度と，左車線を走る車の平均速度を毎時間ステップ出力し，グラフにしよう。

練習　右車線に故障して動けいない車（障害物）がある場合をシミュレートしよう。

練習　３車線交通流ＣＡモデルを作ろう。

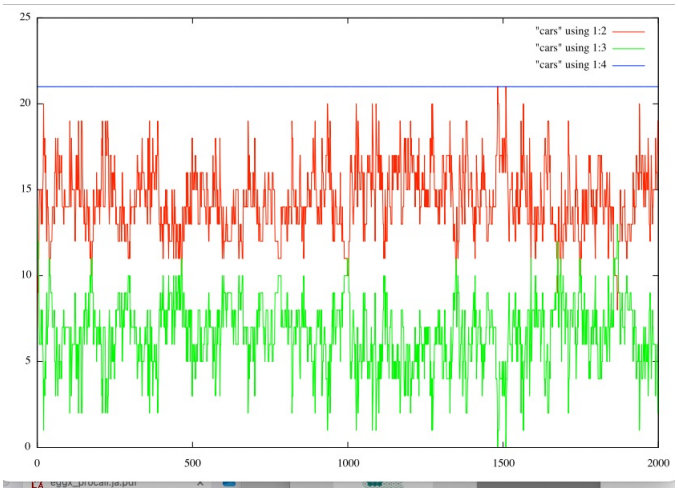

図 29:　2 車線の交通流ＣＡにおける車の台数の時系列変化。赤が左車線，緑が右車線（追い越し車線），青が全体の車の台数。

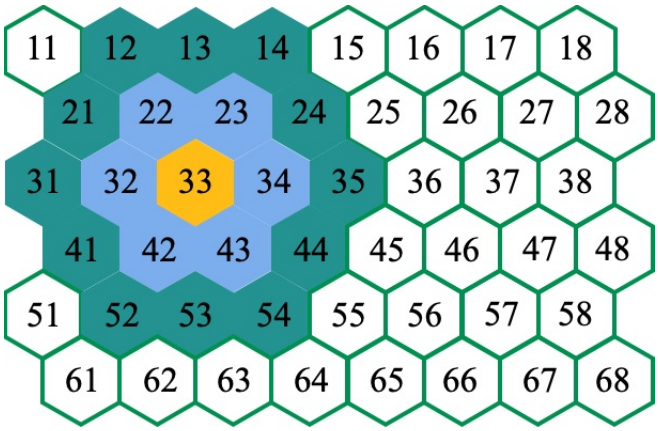

図 30:　33 番のセルを中心とした六方格子。

4.18　六方格子の世界

4.18.1　格子の組み立て

この節では六方格子を組み立てます（図 30）。

黄色で塗りつぶした 33 番のセルに注目します。第一隣接は青で塗りつぶした 6 個のセルです。第二隣接は緑で塗りつぶした 12 個のセルです。今度は正方格子で同じ番号を同じ色に塗りつぶします（図 31）。今度は六方格子で 46 番のセルに注目，同じように隣接サイトを塗りつぶします（図 32）。これに対応する正方格子を示します（図 33）。 図 31 と図 33 を見比べると，再隣接サイトの分布の形状が左右逆転しているのがわかります。これは六方格子の奇数行と偶数行が互いにずれていて，正方格子にマッピングしたときにそのずれが矯正されるためです。従っ

図 31: 図 30 に対応して同じ番号のセルを同じ色で塗りつぶした正方格子。

図 32:　46 番のセルを中心とした六方格子。

図 33: 図 32 に対応して同じ番号のセルを同じ色で塗りつぶした正方格子。

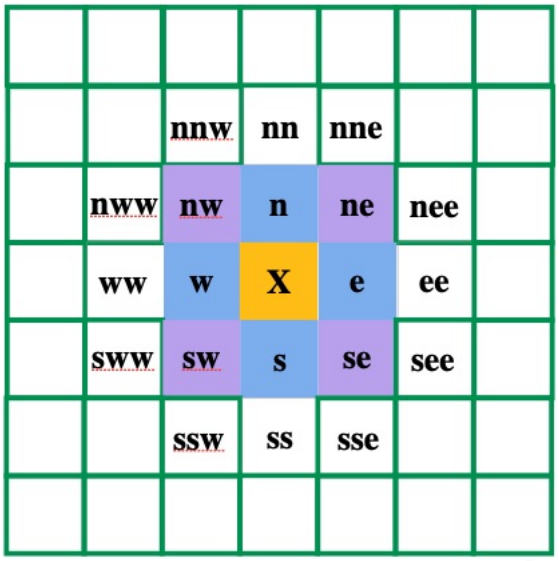

図 34: 2 次元正方格子にマッピングされた六方格子のセル。

てプログラムで扱う 2 次元配列で，同様の隣接サイトの定義をすれば六方格子の CA がシミュレーションできます。図 34 に六方格子の隣接サイトをマッピングしました。このとき中心のセル X が奇数行か偶数行かで隣接サイトの定義を変えると良いわけです。表 5 に隣接サイトの定義を示します。

表 5: 隣接サイトの定義

X のある行	定義	セル
奇数行	第一隣接	n, e, s, w, nw, sw
	第二隣接	nnw,nn,nne,ne,ee,se,sse,ss,ssw,sww,ss,nww
偶数行	第一隣接	n, e, s, w, ne, se
	第二隣接	nnw,nn,nne,nee,ee,see,sse,ss,ssw,sw,ww,nw

4.18.2　雪の結晶

六方格子上の雪の結晶のセルオートマトンモデルを構築します [26]。CA の更新ルールを以下に示します。

- 初期状態として中心のセルに氷の種（1）をおき，あとのセルは雪が無い（0）ものとする。
- 空のセルの，ただ 1 つの第一隣接サイトに氷があるとき，空のセル（0）は氷（1）となる。

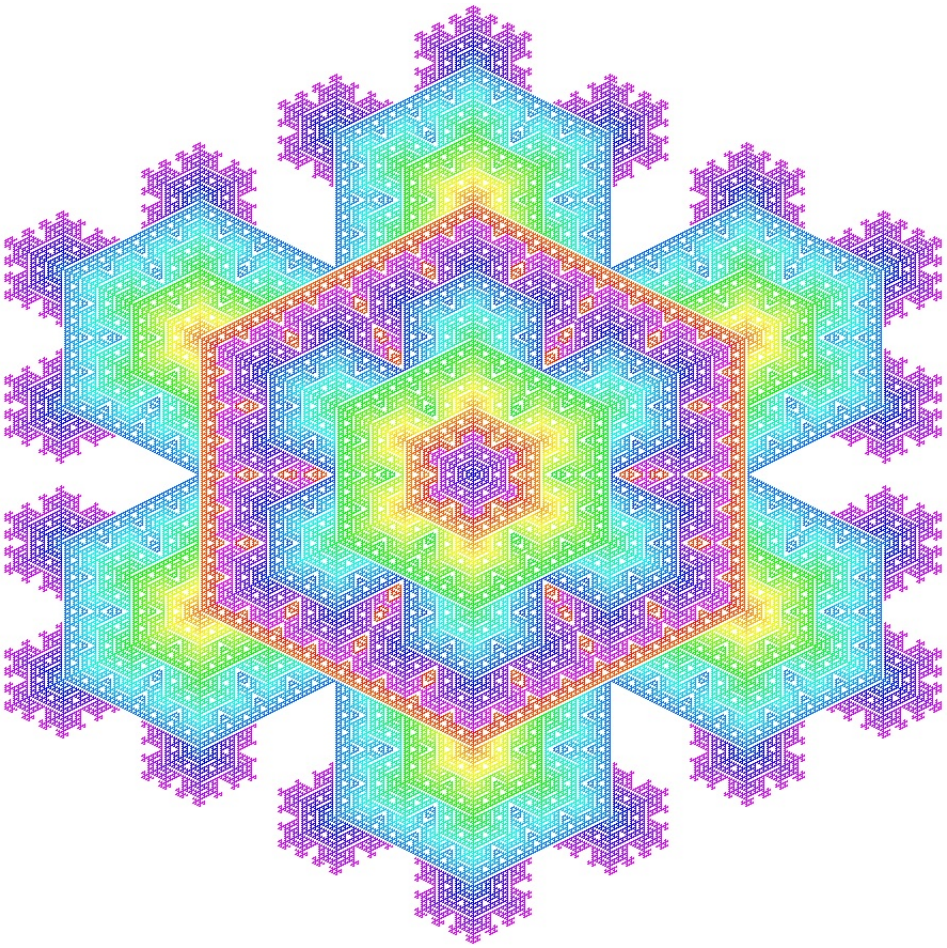

図 35:　六方格子上の雪の結晶。

- 氷のセル（１）は氷のまま。

プログラムは２次元の RWCA を元にして，前節で解説した六方格子から正方格子へのマッピングを施すと比較的簡単にコーディングできます。

GitHub からサンプルコードをダウンロード出来ます。図 35 に 450×450 の格子に雪の結晶ＣＡルールを 110 回施したときの図を示します。実行にかかった時間はノートＰＣで数秒でした。

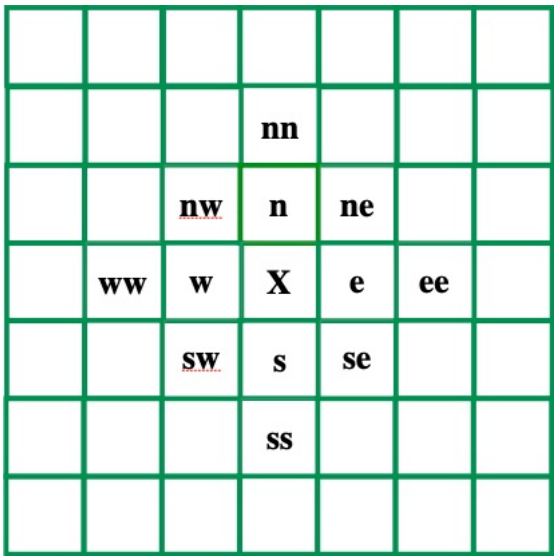

図 36: 2 次元正方格子。

A　付録 I

A.1　RWCA

ここではランダムウォーク・セルオートマトン，**RWCA**，について解説します [27, 28, 29]。簡単のため 2 次元の正方格子を考えます。各セルは「デタラメに歩き回ろうとする人 (ランダムウォーカー)」がいるか，あるいは誰もいない状態にあります。セルにいるウォーカーは，他のウォーカーとの衝突は避け，上下左右のいずれか一つの空いているセルに移動できます。これを系全体で並列にアップデートするセルオートマトンです。図 36 を元に RWCA を定義します。

rule set1： 今，x のセルにウォーカーがいるとします。彼はセル n の方向を向いていて，次のタイムステップでセル n に移動しようとしています。n のセルが空のとき，

- ne に，n を向いているウォーカーがいるときは移動できない。
- nw に，n を向いているウォーカーがいるときは移動できない。
- nn に，n を向いているウォーカーがいるときは移動できない。
- その他の場合は n に移動できる。

同様に，x のセルにセル e の方向を向いていて，e のセルが空のとき (rule set2)，セル s の方向を向いていて，s のセルが空のとき (rule set3)，セル w の方向を向い

ていて，w のセルが空のとき (rule set4) を定義します。その他の場合は x のウォーカーは移動できず，新たに向く方向を選び直す（rule set5）（表 6）。

表 6: ウォーカー x に対する状態遷移表。数字はそれぞれ，N(1), W(2), S(3), E(4) を向くウォーカーを表す。次のタイムステップでの値は y。rnd は 1 から 4 までの新しいランダムな整数。

	x	n	e	s	w	ne	se	sw	nw	nn	ee	ss	ww	y
rule set 1	if 1	0				4								rnd
	elif 1	0							2					rnd
	elif 1	0								3				rnd
	1	0												0
rule set 2	if 2		0			3								rnd
	elif 2		0				1							rnd
	elif 2		0								4			rnd
	2		0											0
rule set 3	if 3			0			4							rnd
	elif 3			0				2						rnd
	elif 3			0								1		rnd
	3			0										0
rule set 4	if 4				0			1						rnd
	elif 4				0				3					rnd
	elif 4				0								2	rnd
	4				0									0
rule set 5	1													rnd
	2													rnd
	3													rnd
	4													rnd

次に空のセルに対する更新ルールを決めます（表 7）。

rule set 6： 今，x のセルが空とします。少なくても，

- セル n とセル e に x を向くウォーカーがいる場合，x のセルは空のまま。
- セル n とセル s に x を向くウォーカーがいる場合，x のセルは空のまま。
- セル n とセル w に x を向くウォーカーがいる場合，x のセルは空のまま。
- セル e とセル s に x を向くウォーカーがいる場合，x のセルは空のまま。

- セル e とセル w に x を向くウォーカーがいる場合，x のセルは空のまま。
- セル s とセル w に x を向くウォーカーがいる場合，x のセルは空のまま。

rule set 7： x のセルが空で，上記の rule set 6 が成立せず，

- セル n に x を向くウォーカーがいる場合，ウォーカーは x のセルにやってきて，新たに面する方向を選ぶ。
- セル e に x を向くウォーカーがいる場合，ウォーカーは x のセルにやってきて，新たに面する方向を選ぶ。
- セル s に x を向くウォーカーがいる場合，ウォーカーは x のセルにやってきて，新たに面する方向を選ぶ。
- セル w に x を向くウォーカーがいる場合，ウォーカーは x のセルにやってきて，新たに面する方向を選ぶ。

rule set8： x のセルが空で，上記のいずれも成立しない場合，空のままとなる。

表 7: 空のセルに対する状態遷移表

	x	n	e	s	w	ne	se	sw	nw	nn	ee	ss	ww	y
rule set 6	if 0	3	4											0
	elif 0	3		1										0
	elif 0	3			2									0
	elif 0		4	1										0
	elif 0		4		2									0
	elif 0			1	2									0
rule set 7	elif 0	3												rnd
	elif 0		4											rnd
	elif 0			1										rnd
	elif 0				2									rnd
rule set 8	elif 0													0

3 次元の RWCA も同様にして定義できます。これらの RWCA プログラムは GitHub からダウンロード出来ます。

動的振る舞いの解析は，例えばウォーカーの位置に関するタイムステップに依存した平均二乗変位（mean squared displacement: msd）

$$\delta(t) = \frac{1}{N}\sum_i |\mathbf{r}_i(t) - \mathbf{r}_i(0)| \tag{99}$$

を求めることで可能です[41]。ここで和は全てのウォーカー N にわたってとります。真のランダムウォークならば，長時間においてこの値は $2dDt$ に近づいていくことが知られています。ここで d は系の次元，D は拡散係数と呼ばれる物理量です。これは program/main.c で，

```
        calc_msd();
```

が有効になっていると自動で計算します。書き出すファイルは files/msd です。例えば 100×100 の 2 次元格子で人口密度 $\rho = 0.2$，10000 ステップまでの RWCA の系の計算は手元のノート PC でおよそ 2 分少々でした。この系は二次元ですので，$d = 2$ として，

$$\delta(t) = 4Dt \tag{100}$$

です。計算値から上式にフィッティングをかけて拡散係数 D を求め，RWCA による計算値とフィッティング関数を同時にプロットしてみます。gunuplot を立ちあげます。

```
gnuplot> plot "msd" using 1:2 w l
gnuplot> f(x)=4*a*x
gnuplot> fit f(x) "msd" u 1:2 via a

  Final set of parameters            Asymptotic Standard Error
  =======================            ==========================
  a               = 0.150298         +/- 1.691e-05    (0.01125%)
```

$D = 0.150298$ であることがわかりました。

フィッティング関数と計算値のプロットは次のようにします（図 37）。

```
gnuplot> plot f(x), "msd" using 1:2 w l
```

信頼性のある計算をするための注意事項を次にあげます。

- 付録の第 4.11.1 節で解説するメルセンヌツイスタなどの乱数発生器を使用する。
- 平均を十分に取れるように何度も試行を重ねる。プログラムでは例えば, main でループを廻し，10000 ステップの計算を 10 回やって「平均の平均二乗変

[41] 例えば能勢修一氏の「分子動力学法入門」, p.33, 式 (6.16) を見よ

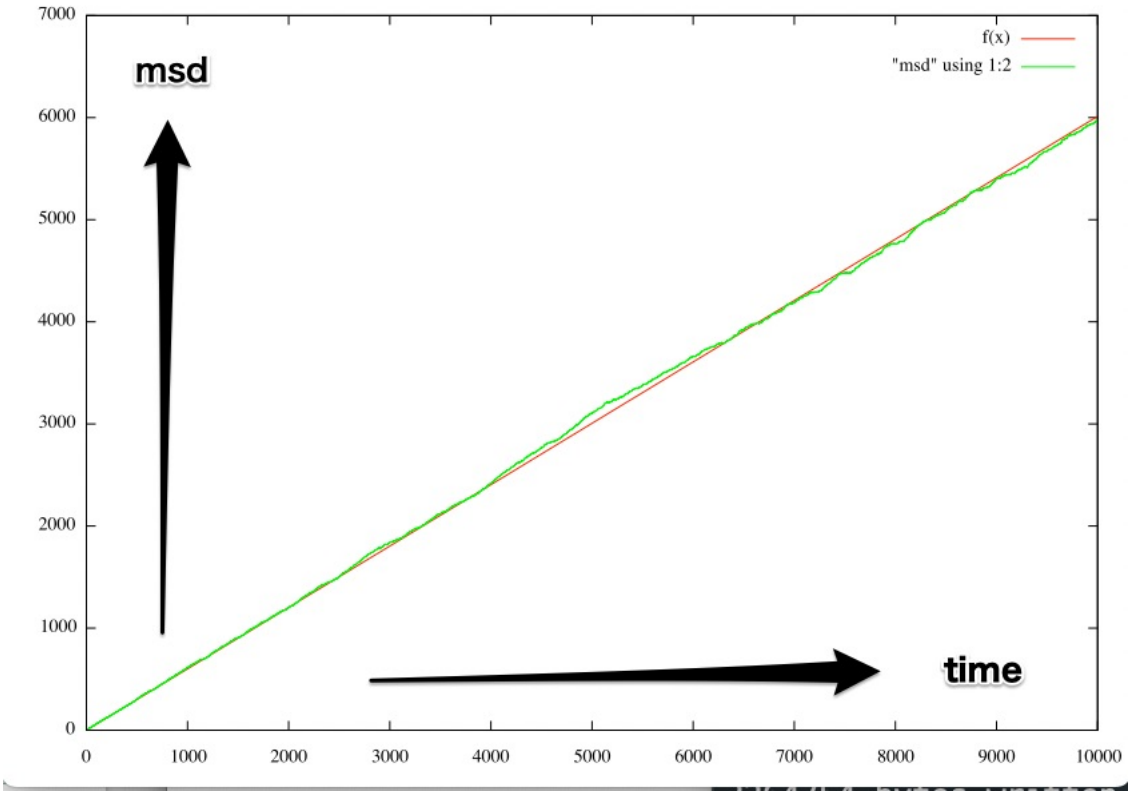

図 37: 平均二乗変位のフィッティング関数 $4Dt$（赤）と計算値（緑）。

位」を計算する関数も用意しています。このようにすると，より信頼のある値が得られます[42]。

A.2　座標変換

3 次元の物体は正面から見るよりも斜めから見ると，より立体であることを把握できる場合が多い。ここでは任意の角度から物体を見るための座標変換式を導入する。3 次元にある物体の座標が (x, y, z) で与えられるとき，この物体を y 軸まわりに ψ 度，x 軸まわりに θ 度，z 軸まりに ϕ 度回転させて得られる新たな座標 (X, Y, Z) は以下で定義される。

$$\begin{aligned}\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} &= R_z(\phi)R_x(\theta)R_y(\psi)\begin{pmatrix} x \\ y \\ z \end{pmatrix} \\ &= (R)_{zxy}\begin{pmatrix} x \\ y \\ z \end{pmatrix}\end{aligned}$$

と書く。このとき回転行列 $(R)_{zxy}$ は，

$$\begin{aligned}(R)_{zxy} &= \begin{pmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & \sin\psi \\ 0 & -\sin\psi & \cos\psi \end{pmatrix} \\ &= \begin{pmatrix} \cos\phi\cos\psi + \sin\phi\sin\theta\sin\psi & \sin\phi\cos\theta & -\cos\phi\sin\psi + \sin\phi\sin\theta\cos\psi \\ -\sin\phi\cos\psi + \cos\phi\sin\theta\sin\psi & \cos\phi\cos\theta & \sin\phi\sin\psi + \cos\phi\sin\theta\cos\psi \\ \cos\theta\sin\psi & -\sin\theta & \cos\theta\cos\psi \end{pmatrix}\end{aligned}$$

実際にはこの転置行列[43]をとり，

$$(R)^T_{zxy} = \begin{pmatrix} \cos\phi\cos\psi + \sin\phi\sin\theta\sin\psi & -\sin\phi\cos\psi + \cos\phi\sin\theta\sin\psi & \cos\theta\sin\psi \\ \sin\phi\cos\theta & \cos\phi\cos\theta & -\sin\theta \\ -\cos\phi\sin\psi + \sin\phi\sin\theta\cos\psi & \sin\phi\sin\psi + \cos\phi\sin\theta\cos\psi & \cos\theta\cos\psi \end{pmatrix}$$

を物体の座標に作用させると，視点から見て意図した方向に物体が回転する。その後，座標の Z 成分を破棄し，(X, Y) のみで EGGX のウインドウに表示すると，物体を斜めから見た画像が得られる。

[42] エルゴードの定理による。
[43] 直交行列なので逆行列 = 転置行列。

B　付録 II

B.1　Emacs

エディタ **emacs** を使ってプログラムファイルを作成します。ターミナルで emacs と入力することでグラフィカルな emacs を立ち上げることができます。[44]

```
% emacs ↵
```

ターミナル内にとどまるキャラクタベースの emacs の立ち上げにはオプション-nw をつけます。

```
% emacs  -nw ↵
```

emacs の終了はキーボード左にある CTL（コントロールキー）と x を押し，続けて CTL と c を押します。

```
CTL+x, CTL +c
```

例えばファイル test1.c を emacs で編集し，作成してからセーブし，終了するためには下記のようにします。

```
% emacs   test1.c↵   ：ファイル編集作業を開始
CTL+x, CTL +s        ファイルをセーブ
CTL+x, CTL +c        emacs を終了
```

他に emacs でよく使うキーは ESC（エスケープキー）で，キーボードの左上にあります。emacs の操作体系では，ESC を**メタキー**と呼びます。ESC は**長く押すとエラーになる**ので，軽く一度だけ押し下げるようにする必要があります。emacs でエラーになった時は，とりあえず CTL +g を押してください（強制的に Quit 信号を送る）。

```
CTL+g   emacs 内で押す
```

練習 emacs には自分で出来るチュートリアルコースが搭載されています。emacs を立ち上げて，メタキーを軽く 1 度押し，X キーを押してください。この段階で，端末の下の方に

[44] % はコマンドプロンプト。↵はリターンキーを押す事を示す。

M-x

と表示されるはずです。これはメタ-X が入力されたことを示します。つづけて，help まで入力して[スペース]，w を入力して[スペース]，t を入力して[スペース]，これで help-with-tutorial と補完してくれます。

M-x help-with-tutorial ↵

help-with-tutorial を見ながら実際に試して見ることでエディタ emacs の使い方をマスターすることができます（４５分ほどのコース)。特にコントロールキー[CTL]を使ったカーソルの動かし方は，コマンドを入力するときにも使えるので，是非マスターしてください。

B.2 GDB

デバッガとはソースコードに紛れ込んだバグを見つけ出すツールです。デバッガ「GDB」を使って実際にバグ潰しをしてみます。ネブラスカ大学リンカーン校（University of Nebraska–Lincoln ）の Dr. Chris Bourke さんによる解説に従って GDB に挑戦しましょう。Chris Bourke さんが用意してくれた Github にアクセスしてバグのあるコード **primesProgram-buggy.c** をダウンロードします。

https://onl.sc/mRt17aA

こちらからもダウンロードできます。

コードをダウンロードしたら，デバッギングオプション**-g** をつけてコンパイルします。

```
% gcc -lm primesProgram-buggy.c -g
```

emacs をテキストモードで立ちあげます。

```
emacs -nw
```

次に emacs の中から gdb を立ちあげます。

```
ESC+x gdb ↵
```

emacs の画面の一番下の行に下のような文言が表示されます。

```
Run gdb (like this): gdb -i=mi a.out ↵
```

これは emacs 内で，実行形式ファイル **a.out** に対して gdb をかけることを確認しています。そこでまたリターンキーを押します。

これでプログラムが実行されるのですが，しばらくしても反応がありません。どうやら無限ループに入っている様子です。コントロールキーと c を二度押して強制的に実行を停止させます。

```
CTL+c, CTL +c
```

B.3 job の概念

Linux を初めとする Unix 系の OS では job の概念が重要です。ここではわざとターミナルモードでの emacs でプログラムを編集し（emacs -nw），CTL+z で

emacs の job にサスペンド（Suspend，一時停止）をかけて，フォアグラウンドでプログラムをコンパイル，実行してみます[45]。

```
% emacs -nw test1.c ↵　ファイル test1.c を編集
編集が終わったら，CTL+z を押して job に Suspend 命令を送る。
［1］+ Stopped                 emacs -nw test1-1.c
job 番号 %1 の emacs が一時停止したことがわかる。
% gcc test1.c ↵　ここでコンパイル
% ls ↵　ファイルを確認
% ./a.out ↵　現在自分がいるディレクトリにあるファイル a.out を実行する
% jobs ↵　job の確認
［1］+ Stopped                 emacs -nw test1.c
job 番号%1 の emacs が一時停止している。
% fg ↵　job をフォアグラウンド（表）に持ってきて停止を解除する。
```

複数の job を立ち上げてそれらを一時停止にすることもできます。

```
% emacs -nw test1.c
編集が終わったら，CTL+z を押して job に Suspend 命令を送る。
［1］+ Stopped emacs -nw test1.c
% emacs -nw      2 番目の job の emacs を立ち上げる
CTL+z を押して job に Suspend 命令を送る。
［2］+ Stopped emacs -nw
% emacs -nw      3 番目の job の emacs を立ち上げる
CTL+z を押して job に Suspend 命令を送る。
［3］+ Stopped emacs -nw
% jobs      コマンド jobs で，今現在の job のリストを出力する
［1］  Stopped emacs -nw test1.c
［2］- Stopped emacs -nw
［3］+ Stopped emacs -nw
% kill %3      3 番目の job に kill 命令を出して終了させる
% fg %1      1 番目の job をフォアグラウンド（fg）に持ってくる。
```

練習　上の例を実際入力して試してみてください。

[45] もちろん emacs 上でのコンパイルもできますが，比較的小さなプログラムの開発にはここで説明する方法で十分です。

注意 emacs の job を 2 つ以上立ち上げると，今現在編集作業を実行している emacs が，どの job なのか解らなくなります。emacs の job は常に 1 つのみとなるよう，**コマンド jobs で、今 suspend している job を確認**するようにしてください。

アンパサンドでの実行 グラフィカルモードの emacs をアンパサンド（&）を付けてバックグラウンドで起動して作業することもできます。

B.4　EGGX のインストール

EGGX のホームページから持ってくることができます。PDF マニュアルもそこにあるのでオンラインで参照することができます。

ソースコードは GIT にも登録されているので，それを持ってきて make することもできます。

```
%  git clone https://github.com/cyamauch/eggx
%  cd eggx
%  cd src
%  ls
00README.1st            Makefile.netbsd        eggx.h
00README.1st.ja.EUC     Makefile.old           eggx_base.c
00README.1st.ja.Mac     Makefile.sgi           eggx_base.h
00README.1st.ja.SJIS    Makefile.sun           eggx_color.c
00README.1st.ja.UTF-8   Makefile.sun64         eggx_color.h
ChangeLog.ja            README                 eggx_tg.h
EXAMPLES                README.ja              eggxlib.h
.................................
```

ディレクトリ src で make をする。例えば Linux 用ならば

```
%  make -f Makefile.linux
```

とする。マニュアルは EGGX のページで参照することができます。ディレクトリ doc には，そのソースがあるので，TeX でコンパイルすることもできます。

B.5　プログラムリスト

以下に GitHub へのリンクをリストします。

- Ising
- Lorenz-Attractor
- Thomas-Attractor
- RLC-series
- RLC-parallel
- GameOfLife
- Excite-CA
- Infect-CA
- DLA-CA
- Predator-CA
- Traffic-CA
- Traffic-2lanes-CA
- Snowflake-CA
- RWCA

リンク先のページ右側にある **Code** メニューから **Download ZIP** を選択すると ZIP アーカイブ形式でプログラムパッケージをダウンロードすることができます。あるいは git を使って HTTPS 経由でクローンを作ることもできます。GameOfLife を例に取ると，**Code** を開いたときに **HTTPS** のアドレスが表示されるので，それをコピーします。ターミナルで git clone に続けてアドレスをペーストし実行します。

```
$ git clone https://github.com/KazumeNishidate/GameOfLife.git
```

参考文献

[1] 間瀬茂, 神保雅一, 鎌倉稔成, 金藤浩司. 工学のためのデータサイエンス入門. 数理工学社, 2005.

[2] 石川宏. Cによるシミュレーション・プログラミング(SOFTBANK BOOKS). ソフトバンククリエイティブ, 1994. https://www.amazon.co.jp/dp/4890525491/ref=cm_sw_em_r_mt_dp_7ZSGOH1M1JQ71C6A9160.

[3] 矢部孝、川田重夫、福田昌宏. シミュレーション物理入門. 朝倉書店, 1989. https://www.amazon.co.jp/dp/4254130449/ref=cm_sw_em_r_mt_dp_8RSO8FKY4RZ6NV3ZMSM7.

[4] 木原太郎. Cによる統計物理. 東京大学出版会, 1993. https://www.amazon.co.jp/dp/4130626000/ref=cm_sw_em_r_mt_dp_JOD15WB4HKODFGTT3374.

[5] 北本 朝展. オープンソースによる数値計算：GNU Scientific Library. 人工知能学会誌, 21(4):470–477, 7 2006. https://www.jstage.jst.go.jp/article/jjsai/21/4/21_470/_pdf.

[6] 渡辺尚貴. 計算物理のための c/c++言語入門. WEB ページ, 2000. http://cms.phys.s.u-tokyo.ac.jp/~naoki/CIPINTRO/.

[7] Munnujahan Ara Azmol Huda, Mohammad Wahiduzzaman. Graphical interpretation of the slopes used in the derivation of classical fourth order runge-kutta (rk4) formula. IJRSI, IX:27–32, 2022. https://www.rsisinternational.org/journals/ijrsi/digital-library/volume-9-issue-4/27-32.pdf.

[8] M. Crucifix. Oscillators and relaxation phenomena in pleistocene climate theory. Phil. Trans. R. Soc. A, 370:1140–1165, 2012. https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2011.0315.

[9] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. ACM Transactions on Modeling and Computer Simulation, 8:3–30, 1998. https://doi.org/10.1145/272991.272995.

[10] 上田睆亮. 非線形性に基づく確率統計現象 – Daffing 方程式で表わされる系の場合 –. 電気学会論文誌 A, 98:167–173, 1978. https://www.jstage.jst.go.jp/article/ieejfms1972/98/3/98_3_167/_pdf/-char/ja.

[11] 上田睨亮. カオス現象の解説と一提言. 日本原子力学会誌, 52:150–154, 2010. https://www.jstage.jst.go.jp/article/jaesjb/52/3/52_150/_pdf/-char/ja.

[12] 神部勉. オイラー方程式の新しい解表現, および最初のカオス理論のポアンカレ (1890) の再認識と上田アトラクタの発見 (1961). 数理解析研究所講究録, 1776:45–58, 2012. https://www.kurims.kyoto-u.ac.jp/~kyodo/kokyuroku/contents/pdf/1776-04.pdf.

[13] E. N. Lorenz. Deterministic nonperiodic flow. J. Atmos. Sci, 20:130–141, 1964. https://doi.org/10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2.

[14] R. Thomas. Deterministic chaos seen in terms of feedback circuits: Analysis, synthesis, 'labyrinth chaos'. Int. J. Bifurcation and Chaos, 9:1889–1905, 1999. https://www.worldscientific.com/doi/abs/10.1142/S0218127499001383.

[15] M. Kaufman and R. Thomas. Emergence of complex behaviour from simple circuit structures. C. R.- Biol., 326:205–214, 2003. https://www.sciencedirect.com/science/article/pii/S1631069103000635.

[16] Bastien Chopard. Cellular Automata and Lattice Boltzmann Modeling of Physical Systems, pages 287–331. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[17] J. Greenberg, B. Hassard, and S. Hastings. Pattern formation and periodic structures in systems modeled by reaction-diffucsion equations. Bull. Amer. Math. Soc., 84:1296–1327, 1978. https://www.semanticscholar.org/paper/Pattern-formation-and-periodic-structures-in-by-Greenberg-Hassard/c57b7e0a826d436e41857a3926e122aec9cc427d.

[18] J. Greenberg and S. Hastings. Spatial patterns for discrete models of diffusion in excitable media. SIAM J. Appl. Math, 34:515–523, 1978.

[19] Richard Durrett and David Griffeath. Asymptotic behavior of excitable cellular automata. Experimental Mathematics, 2:183–208, 1993. https://www.semanticscholar.org/paper/Asymptotic-Behavior-of-Excitable-Cellular-Automata-Durrett-Griffeath/692d875fd5938ace6364b5eb6417d23ab5af1881.

[20] R. Fisch, J. Gravner, and D. Griffeath. Threshold-range scaling of excitable cellular automata. Statistics and Computing, 1:23–39, 1991.

[21] Jr. T.A. Witten and L. M. Sander. Diffusion-limited aggregation, a kinetic critical phenomenon. Phys. Rev. Lett., 47:1400–1403, 1981. https://journals.aps.org/prl/pdf/10.1103/PhysRevLett.47.1400.

[22] ”松下貢，早川美穂，近藤宏，本田勝也，豊木博泰，本庄春雄，太田正之輔”. DLA とそれに関連した現象 – An invitation to funny physics –. 物性研究, 48:473–506, 1987. https://repository.kulib.kyoto-u.ac.jp/dspace/bitstream/2433/92810/1/KJ00004774802.pdf.

[23] M. Matsushita. Physics of formation of fractal patterns. Koubunshi (Japanese, 48(2):70, 1999. https://www.jstage.jst.go.jp/article/kobunshi1952/48/2/48_2_70/_pdf.

[24] 太田正之輔. DLA 結晶成長とフラクタル次元. 物性研究, 93:33–69, 2009. https://repository.kulib.kyoto-u.ac.jp/dspace/bitstream/2433/169149/1/KJ00005756695.pdf.

[25] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. J. Phys. I France, 2:2221–2229, 1992. https://hal.science/jpa-00246697/document.

[26] N. H. Packard. Lattice models for solidification and aggregation. In S. Wolfram, editor, Theory and Application of Cellular Automata, pages 305–310. World Scientific, 1986.

[27] K. Nishidate, M. Baba, and R.J. Gaylord. Cellular automaton model for random walkers. Phys. Rev. Lett., 77:1675–1678, 1996. https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.77.1675.

[28] K. Nishidate, M. Baba, and R.J. Gaylord. Cellular automaton model for biased diffusive traffic flow. J. Phys. Soc. Jpn., 65:3415–3418, 1996. https://journals.jps.jp/doi/abs/10.1143/JPSJ.65.3415.

[29] K. Nishidate, M. Baba, H. Chiba, T. Ito, K. Kodama, and K. Nishikawa. Cellular automaton model for biased diffusive traffic flow. J. Phys. Soc. Jpn., 69:1352–1355, 2000. https://journals.jps.jp/doi/abs/10.1143/JPSJ.69.1352.

著者紹介

西館数芽，博士（理学）
青森県むつ市出身
1965 年 3 月生まれ

岩手大学理工学部
電気電子通信コース
nisidate_at_iwate-u.ac.jp
kazume.nishidate_at_gmail.com
(_at_は@に読みかえてください)