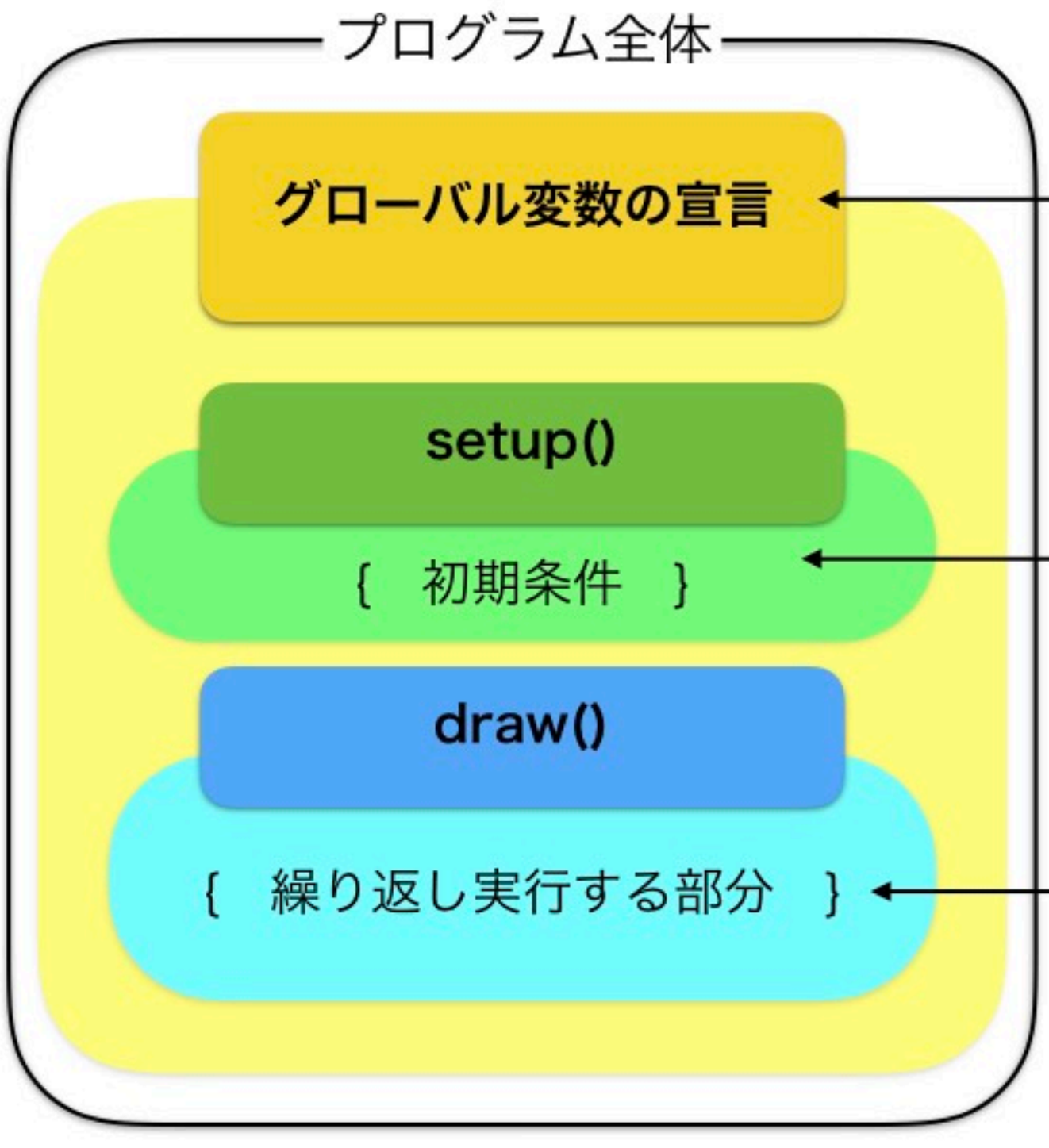
現象数理III
ランダムウォークを使った現象のモデリングとシミュレーション

第4回・第５回　2020年12月23日

担当：岩本真裕子

（復習）プログラム全体像



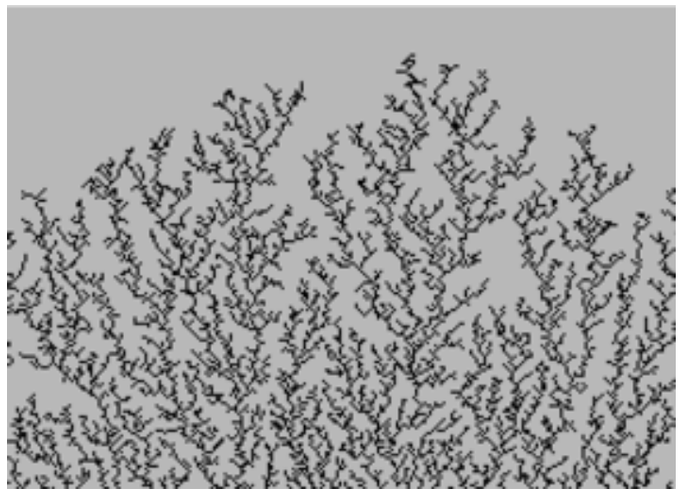
setup()の中でもdraw()の中でも使う変数は一番上にかく。

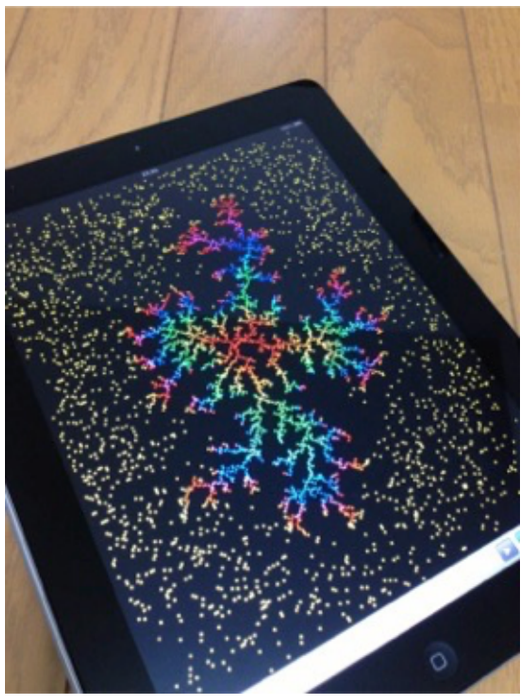
１回だけ実行される

繰り返し実行される
１秒間に６０回
（回数は変更可能）

DLA

デタラメが作り出すパターン（DLA）





- DLAのシミュレーションを実行・観察できる iPhone, iPad アプリ TheDLA は、遊びながら，でたらめな運動がどのようにして自然界に見られるような樹枝状の形を作り上げるのかを学ぶことができます。遊ぶには, iPhone , iPad や iPad touch等の iOS 機器が必要です。The DLAの製作者ページはこちら。

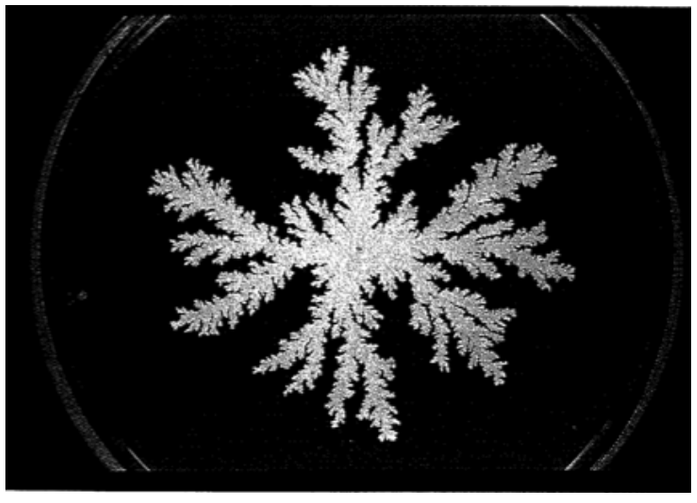
拡散律速凝集 DLA
Diffusion Limited Aggregation

DLAは、日本語では、**拡散律速凝集**とよばれる。律速とは、その現象の時間スケールを決定づけるもっとも遅い事象を指す。拡散律速凝集は、私たちの身の回りにもよく見られる現象である。

例１：**電気分解**　$Zn^{2+} + 2e^- \rightarrow Zn$

硫酸亜鉛（$ZnSO_4$）水溶液に、亜鉛よりもイオン化傾向が小さい金属（例えばシャーペンの芯でOK）を刺して電気を流すと、水溶液中の亜鉛イオンは、電子e^-を受け取って**亜鉛**となる。

このとき、水溶液中をランダムウォークしている亜鉛イオンが、シャーペンの芯にランダムにくっつきながら成長していくため、下のような樹状パターンができる。



例２：**枯草菌**のコロニーパターン（写真はFujioka & Matsushita, 1989）

真性粘菌である枯草菌は、飢餓状態の時に凝集しパターンを作る。

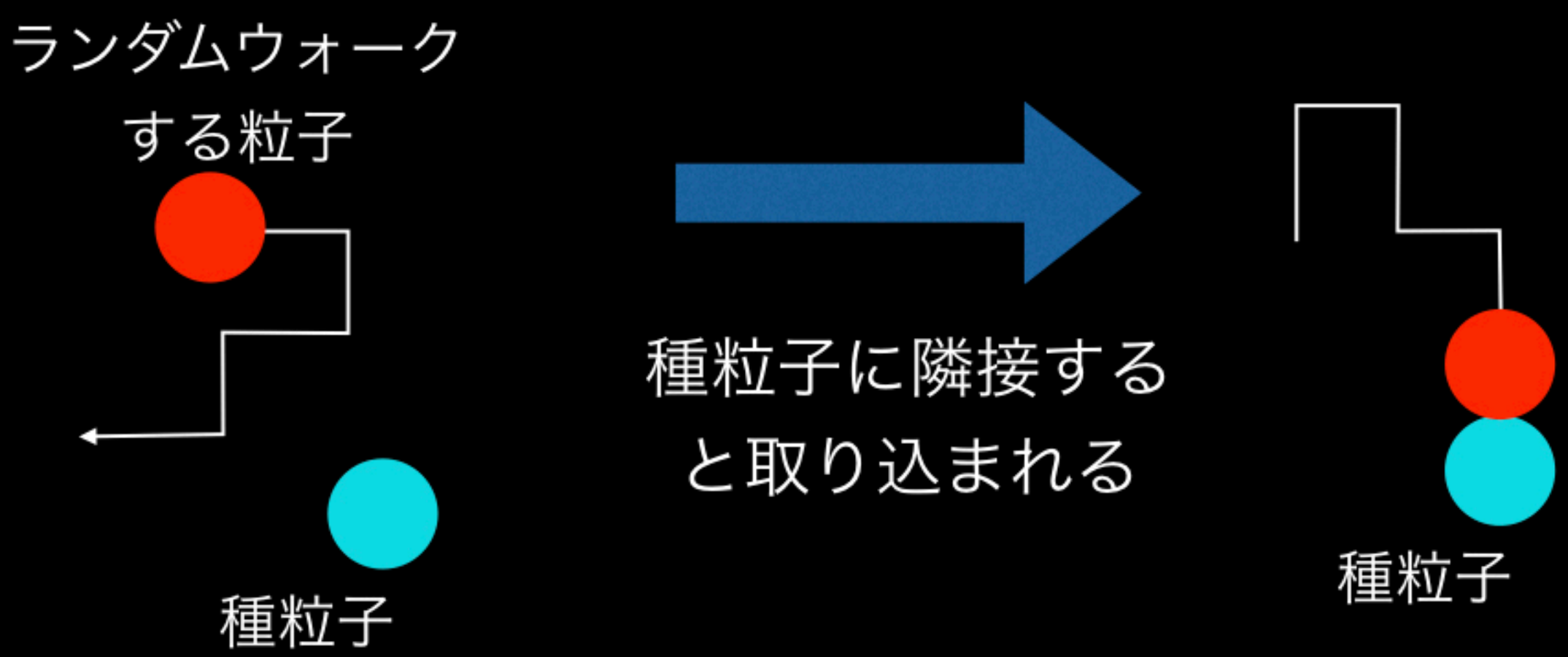
DLAのアルゴリズム

では、どのようにしてDLAパターンは作られているのか。それはとても簡単で、**ランダムウォークしている粒子が種粒子にくっついていく**、というだけである。

種粒子とよばれる粒子が平面上にあって固定されているとする（水色）。そこから少し離れたところにランダムウォークする粒子を（たくさん）置く（赤色）。この粒子を**自由粒子**とよぶ。

このランダムウォークする自由粒子は平面上を動き回り、**たまたま種粒子の隣にくる**場合があるでしょう。種粒子の隣に自由粒子がきたら**自由粒子は種粒子に付着**するとし、それ自身が種粒子と同等の粒子となるとする。

ランダムウォーク
＋簡単なルール



参考文献：フラクタルの物理(I)(II), 松下貢, 裳華房

本日の課題1

- DLAパターンのシミュレーションを作り、「dla_学生番号」として保存せよ。
- プログラム作成の際には、次ページ以降のヒントを参考にすると良い。

基本的な設定について

- 粒子を　NUM+1　個用意する
- 粒子の状態を表す変数flagを用意する
- 初期設定：０番目の粒子を種粒子(flag = 1)、それ以外を自由粒子(flag = 0)とする
- 初期設定：自由粒子の位置はランダム（ただし整数値、粒子の直径ごと、重なってもOK）とする。（ここは少し工夫が必要）
- 自由粒子がランダムウォークする１歩は粒子の直径分とする
- 自由粒子と種粒子との距離を測り、直径以下なら自由粒子を種粒子に変える（flag: 0 → 1）

ヒントその1：
フラッグを立てる

DLAシミュレーションの粒子は、**自由粒子**と**種粒子**がいる。粒子がどちらの状態であるかを知る必要がある。そこで、**状態を表す変数flag(整数型int)を用意**する。

例えば、

- i番目の粒子が**自由粒子**ならflag[i] = **0**とする。（旗をあげない）
- i番目の**種粒子**ならflag[i] = **1**とする。（旗をあげる）

旗があがっていない粒子は動けるが、旗をあげた粒子は動けない。

ヒントその2
整数の乱数の生成

乱数は**random(10,20)**などを使うと作ることができるが、標準では実数値である。つまり、**random(10,20)**とすると、**10以上20未満の実数値**が生成される。

しかし、整数値のみの乱数が使いたいことがある。その場合は、以下の方法のいずれかを使うと良い。

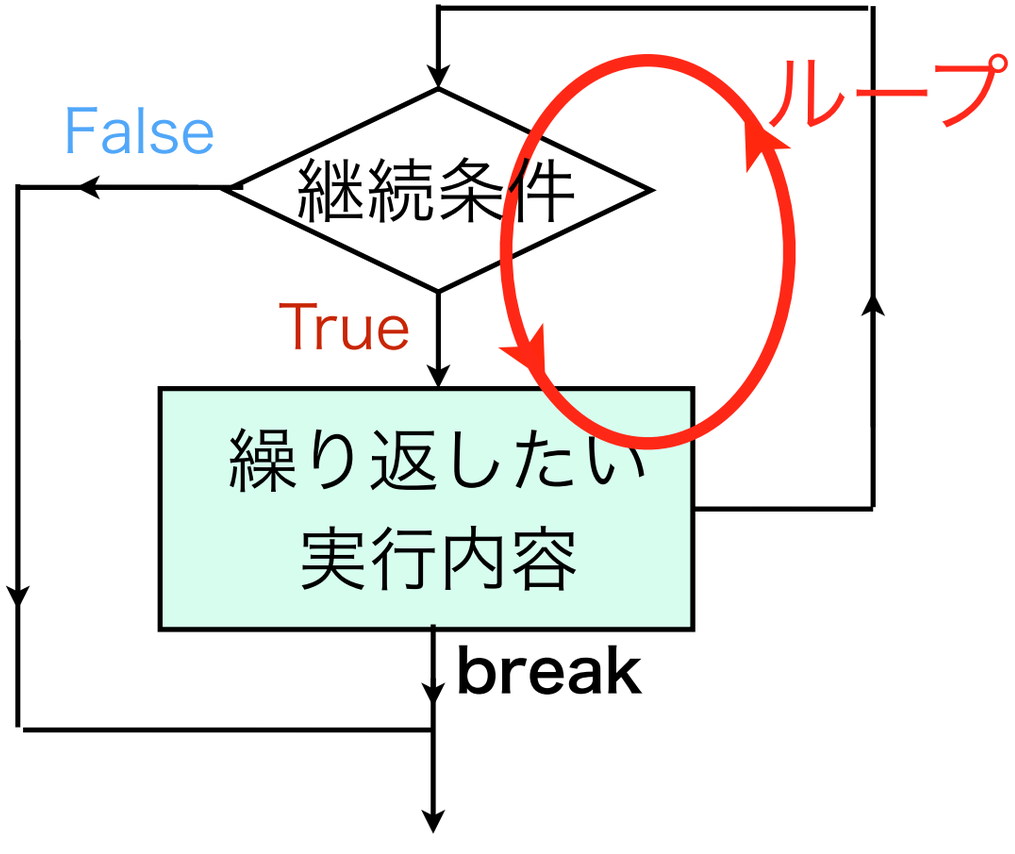
小数点以下を**切り捨て**る　　**floor(random(10,20))**

小数点以下を**四捨五入**する　**round(random(10,20))**

小数点以下を**切り上げ**る　　**ceil(random(10,20))**

ヒントその３　break

for文の中で**break**文を使用すると、それ以降の処理を行わず、**for**文の繰り返し処理が終了



ヒントその４：アルゴリズム

①変数の宣言

・粒子の数NUM （intで用意する。数を代入しておく。）

・粒子の位置xとy （float で をNUM+1個分配列で用意する。）

・粒子の状態flag （intで 配列でNUM+1個分用意する。）

・粒子の半径r （floatで用意する。）

②setup関数に初期値を入れる

```
void setup()
{
    サイズ、背景、フレームレート、半径などの設定
    ０番目の粒子のflagと位置の設定（種粒子）
    １番目からNUM番目までの粒子のflagと位置の設定（種粒子）
}
```

次ページへ続く

③draw関数に時間発展を入れる

```
void draw()
{
   描画(自由粒子（flag = 0）は赤、種粒子（flag=1）は青)


   for(int i=1; i <= NUM; i++)
  {
     ＜もしi番目の粒子が自由粒子だったら＞

       for(int j = 0; j <= NUM; j++)
       {
           ＜もしj番目が種粒子だったら＞
                i番目の粒子とj番目の粒子の距離を計算
                ＜もしその距離が直径以下だったら＞
                        i番目のflagを1にする
                        break;
          }
    }

   for(int i=1; i <= NUM; i++)
   {
   ＜もし自由粒子だったら＞ランダムウォーク
   }
}
```

蟻の行動

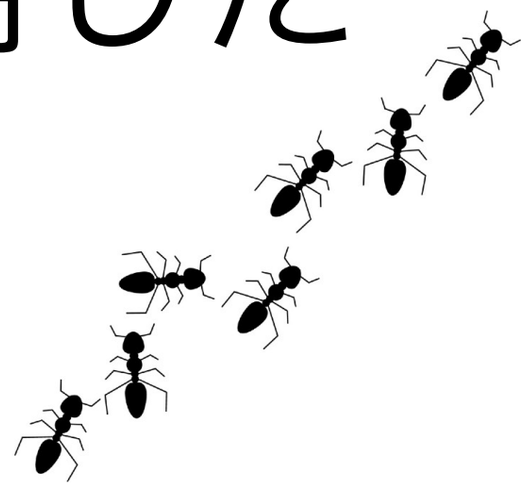
ランダムウォークを利用したアリの行動

- ランダムウォークを利用して蟻の行動をシミュレーションしよう。
- 餌を探しに出た蟻はどのようにして巣に帰るのか？

次のような観察結果がある。

- 蟻は**巣から出て、餌を探す時にはランダムウォーク**する。
- **巣に帰るときには**何故か**巣の方向に向かってほぼ一直線**に戻る。

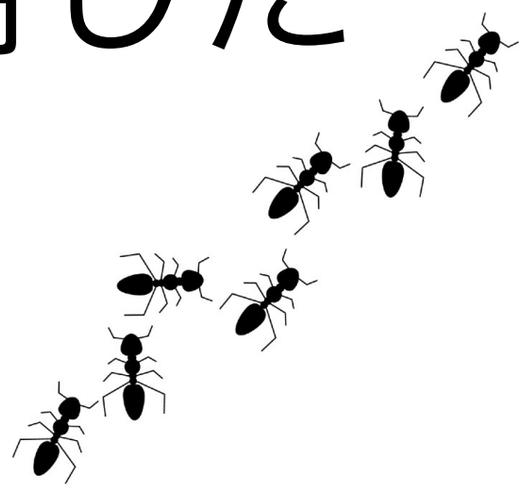
ランダムウォークを利用したアリの行動



さらに観察すると、次のような事実がわかる。

- 蟻は餌を求めるためにランダムウォークで動き回るが、動いたところにある種の化学物質(フェロモン)をつけて、足跡を残す。
- （餌を見つけるか、あるいはあきらめて）巣に帰るときには、**付けた化学物質（足跡）の多い方へ向かって進む。**（このとき、餌を見つけた蟻はそのことを知らせるために、別の異なる化学物質を足跡としてつける。）
- （注意）蟻と言えば蟻の行列を思い浮かべる。上記の話は、行列を作る前の話である。

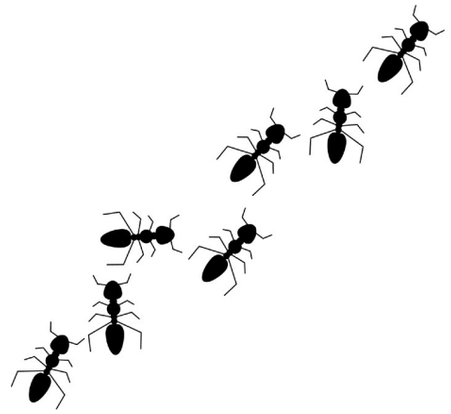
ランダムウォークを利用したアリの行動



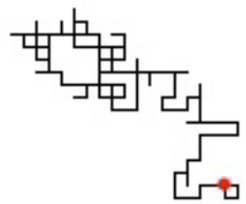
アリの行動をまとめると

- **蟻は餌を探す時にはランダムウォークを行う。**
- **そのとき足跡フェロモンを分泌する。**
- **餌を見つけると、フェロモンをたどって巣に戻る。**
- そのときには道しるべフェロモンという別の物質を分泌し、巣に帰った後にその道しるべフェロモンをたどることで餌の場所に行列をつくりつつ、さらに道しるべフェロモンを強化すると言われている。

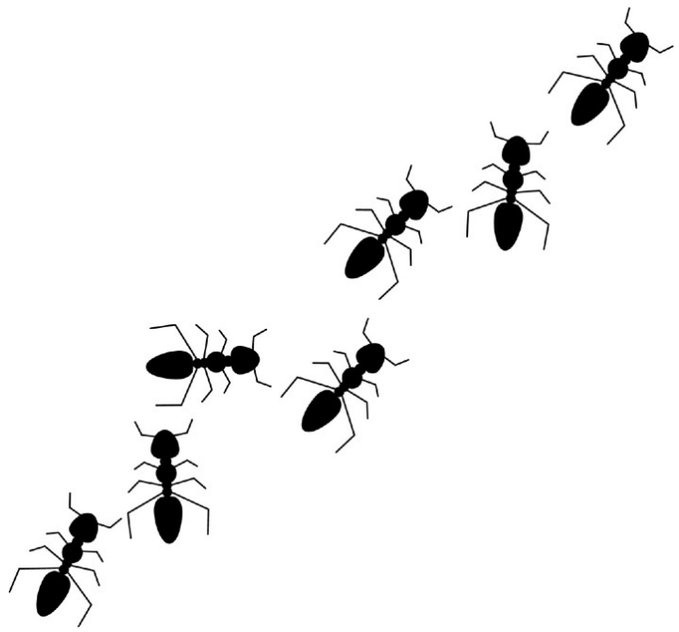
アリの軌跡



練習1　２次元配列を使って、１匹の蟻がランダムウォークした軌跡を描画するシミュレーションを作成せよ。



ヒント



- ヒント１）　２次元配列の宣言

int x[][] = new int[NUM][step];

- ヒント２）　時間ステップ数

frameCount

- ヒント３）　用意した配列より大きい数が入るとエラーになってしまうので、例えば1000ステップで止めると決めて、繰り返しループを止める。

if(frameCount >= 999) noLoop();

２次元配列について

- 例えば、

float a[2][3];

と宣言すると、

a[0][0], a[1][0],

a[0][1], a[1][1],

a[0][2], a[1][2]

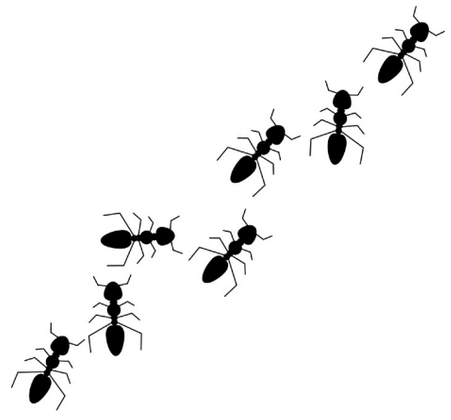
の計６つ(2×3) の整数型の変数が用意される。

- ６つの変数に２重for文で値を代入することができる。

```
for(int i = 0; i < 2; i++)
{
    for(int j = 0; j < 3; j++)
    {
        a[i][j] = i*j;
    }
}
```

これによって、
a[0][0] = 0.0
a[1][0] = 0.0
a[0][1] = 0.0
a[1][1] = 1.0
a[0][2] = 0.0
a[1][2] = 2.0
と値が代入される。

足跡を数える

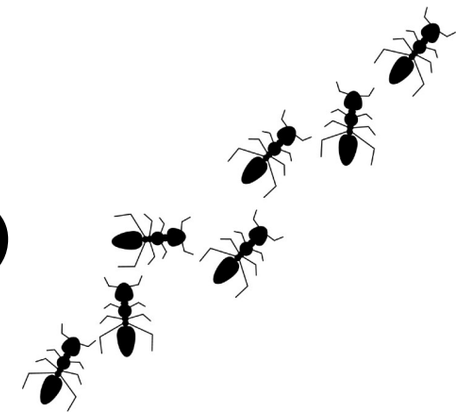


- 蟻は**移動するたびに足跡フェロモンを残す**ということが観察結果からわかっているので、足跡を数える仕組みをランダムウォークシミュレーションのプログラムに組み込もう。
- 考えている格子点全てについて、**足跡を保持している変数を用意してやればよい。**この場合、２次元配列変数を使うと簡単である。具体的には、

int count[][] = new int[x方向の格子の数][y方向の格子の数];

として宣言した配列変数 count に足跡の個数を記録する。

足跡を数える



練習2　１匹の蟻の足跡フェロモンの量を表示するシミュレーションを作成せよ。蟻の衝突は考えない。

- ヒント１）　初期値として、count[i][j]の全てに0を代入せよ。
- ヒント２）　蟻がその場所に来たら

count[蟻のx格子][蟻のy格子] += 1;

- ヒント３）　塗りつぶしにcount[i][j]を使って、フェロモンの量を色で表現せよ。

```
for(int i = 0; i < 100; i++)
 {
  for(int j = 0; j < 100; j++)
  {
   fill(255-min(255,2*count[i][j]));
   rect(i*5, j*5, 5,5);
  }
```

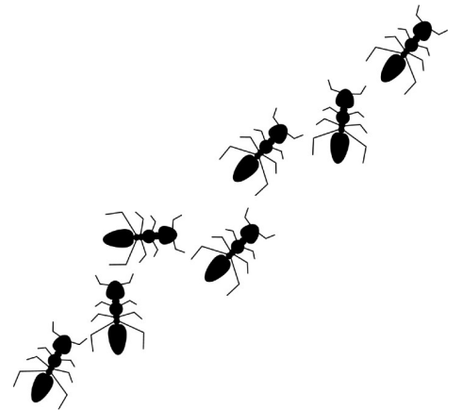
最大値最小値を求めるときは、maxやminを使用することが可能である。

最小値を求める　　**min(a,b)**

最大値を求める　　**max(a,b)**

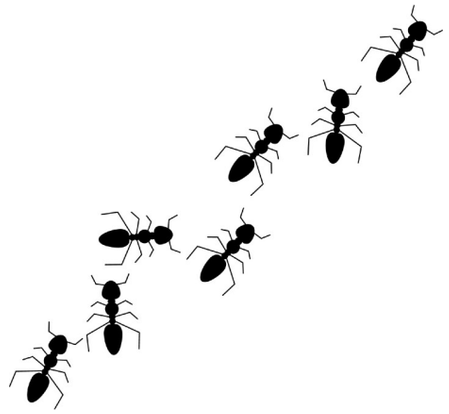
a,bに入るのは、int型かfloat型（同じである必要がある）で、最大３つまでの数を比較できる。

アリが巣から
ランダムウォーク



練習３　M匹の蟻が一斉に巣から出てくる状況を考える。このとき、各格子の足跡の合計を画面に表示するプログラムを作成せよ。蟻の衝突は考えない。

巣に帰るアリ

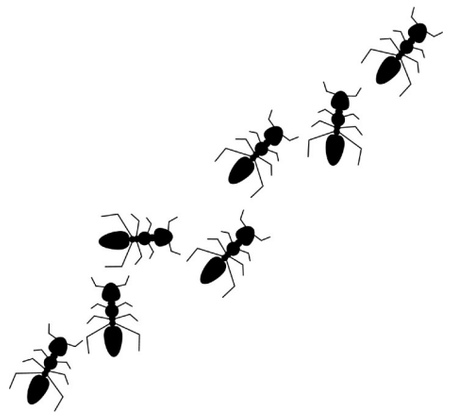


- **足跡の分布**は巣から離れるにつれて減少しているように見え、この傾向は蟻の数Mが大きくなるにつれ強くなった。

蟻の帰巣行動について、蟻は次の様なルールに従い、巣に帰るとする。

- **蟻はある回数動くと、餌を見つけられたか否かによらず、（疲れたので）巣に帰る。**
- **巣に帰るときには化学物質量（足跡数）の多い方に向かって移動する。**

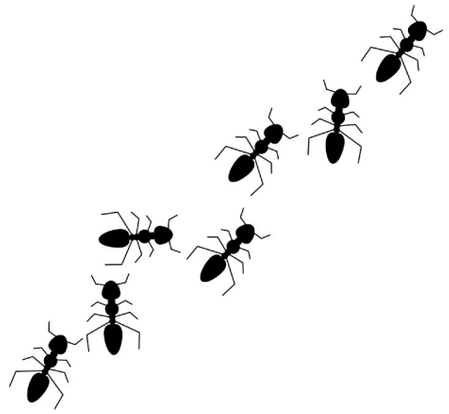
巣に帰るアリ



- 蟻の帰巣行動をシミュレーションするための手順を考えよう。
- **最終的に到達した地点から帰巣行動をはじめる**ため、M匹の蟻それぞれがランダムウォークの結果到達した**最終到達点の座標を記録**しておく必要がある。
- ヒント１）最終到達点の座標を記録しておく配列変数を用意する。
- ヒント２）delayで指定した時間、計算を停めることができる。括弧の中には数字を入れる。単位はミリ秒なので、次のようにかくと100ステップ目で３秒間計算がとまり、その後また始まる。

```
if(frameCount == 100) delay(3000);
```

本日の課題２



- **巣からランダムウォークして出て行った蟻が5秒間止まったのち、巣に帰るシミュレーション**を作成せよ。蟻は４方向をみてフェロモンの量が多い格子へ進むとする。

本日の課題（発展）

- 課題２で、ランダムウォークしたM匹の蟻のうち、ある時間内に**何匹が帰巣に成功するか**調べよ。

本日の課題（発展）

- 巣からランダムウォークして出て行った蟻が5秒間止まったのち、**すべての蟻が巣に戻る**ようにシミュレーションせよ。

以下の点に工夫が必要である。

1. ４方向のうち、**２方向以上のフェロモンの量が同じ**であった場合どうするか。
2. **４方向に比べて自分のいるところの方がフェロモンの量が多い**場合はどうするか。