演習２：集合の知性を設計する

(05) 05/20　(06) 05/27

A｜Unity環境の整備・簡単なルールの設計

(07) 06/10　(08) 06/17

B｜ボイドルール１・２・３の実装

(09) 06/24　(10) 07/01

C１｜集合知の解析

(12) 07/08　(13) 07/15

C２｜マイルール（ルール４）の実装・視点の操作

(14-15) 07/22

C２｜発表（One-Minute Movie）

演習２-B

ボイドルールの設計（ルール２・ルール３）

衝突回避（ルール２）　整列行動（ルール３）

BoidManager.cs

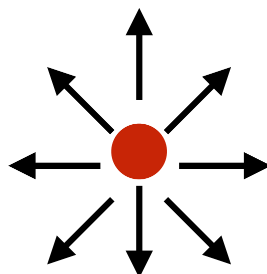
```
//ルールの係数
public float c1 = 0.1f;
public float c2 = 5.0f;
public float c3 = 0.01f;
```

Update()

```
//ルール2の適用
if (rule2) {
    ApplyRule2 ();
}
//ルール3の適用
if (rule3) {
    ApplyRule3 ();
}
```

rule2 が true のときに, ApplyRule2を適用

rule3 が true のときに, ApplyRule2を適用



指定された点から

離れるようにするためのベクトル計算

位置による反発作用（斥力）

<速度 v'> を<点 A> から離れる方向に修正する

（今回は, 修正ベクトルの絶対値を1に正規化します）

A 斥力点

(ax, ay)

速度ベクトル

$v' = (vx, vy)$

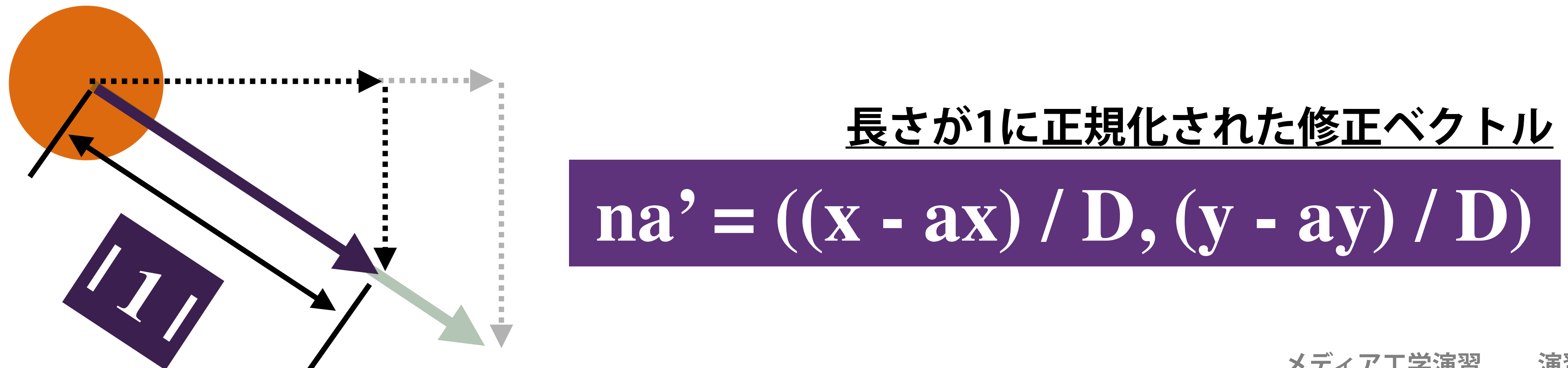
(x, y)

位置

修正ベクトル

$a' = (x - ax, y - ay)$

D 距離



A

(ax, ay)

速度ベクトル

v’ = (vx, vy)

修正ベクトル

a’ = (x - ax, y - ay)

(x, y)

位置

D

距離

修正後の速度ベクトル

v2’ = (vx + (x - ax) / D, vy + (y - ay) / D)

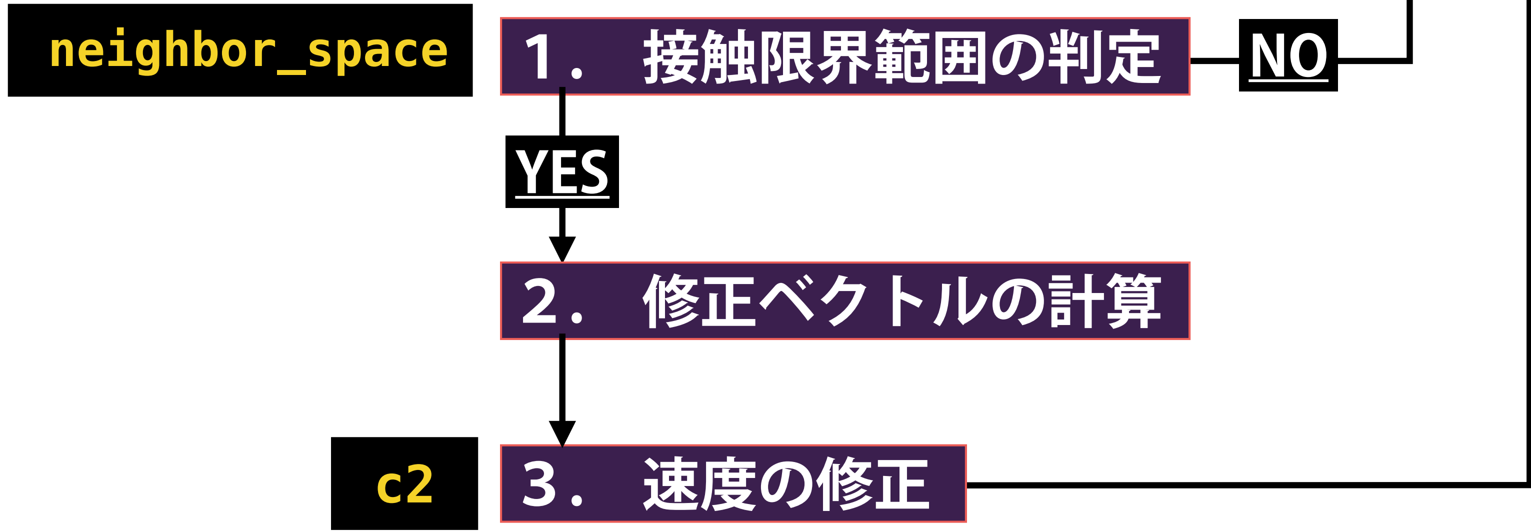
係数（c2）を付した三次元表現では，…

v2’ = (vx, vy, vz) + c2 * (x-ax, y-ay, z-az) / D

ルール２（反発）

BoidManager
<int> bsum
ボイドの総数
<float> neighbor_space
各ボイドの接触限界距離
<float> c2
斥力の強さ

ルール２（分離）の実装

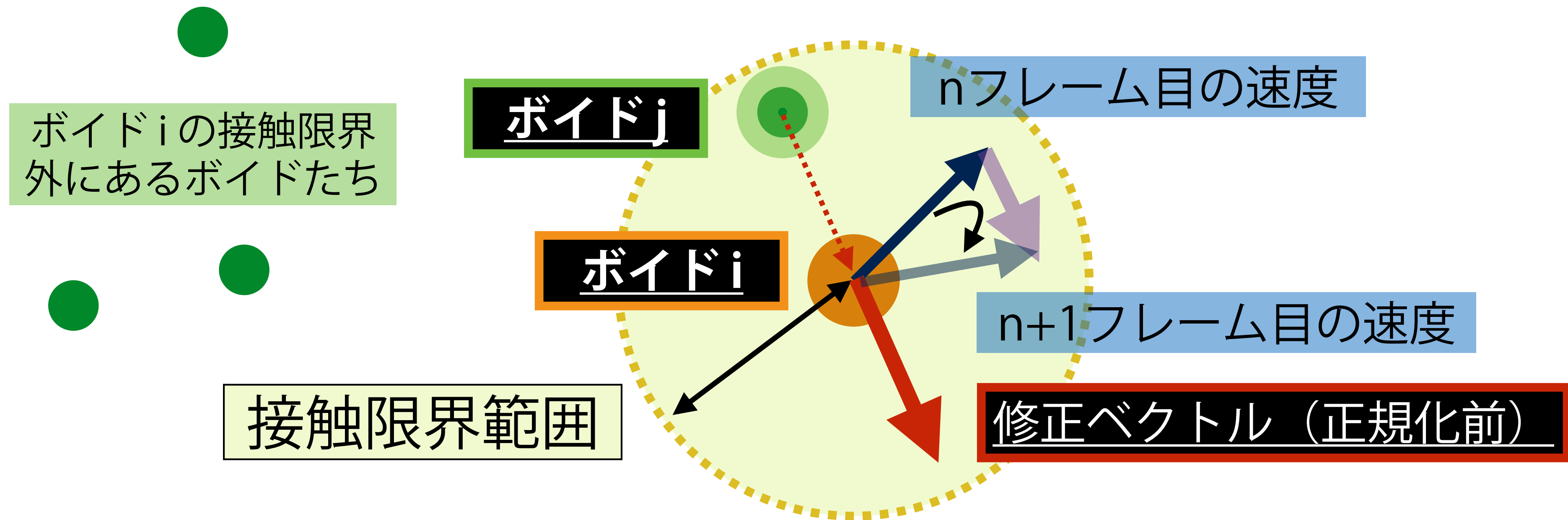


上記のフローのように，（平均を求めるような手法ではなく）
逐次的に速度を修正する手法が最も効率的な分離を可能とするようです．

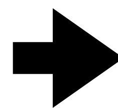
ルール２（反発）

ルール２（分離）の実装

１．ボイドｉに関する接触限界範囲の判定



ボイドｊ が ボイドｉ の 接触限界範囲 にあるか？

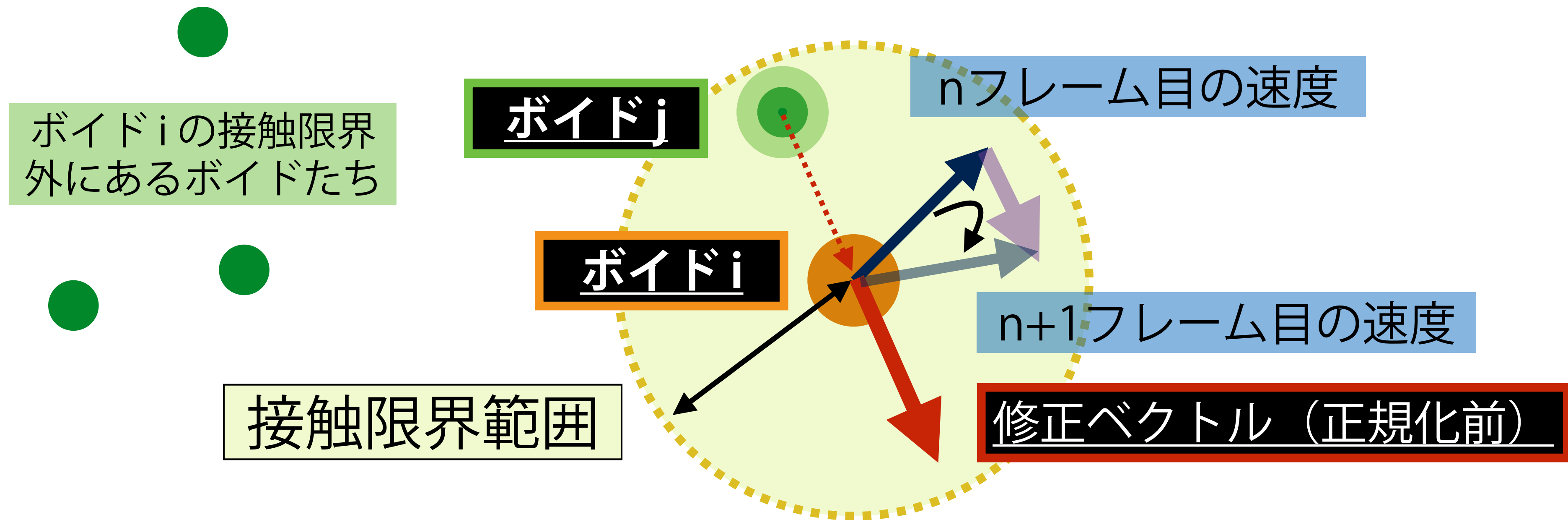


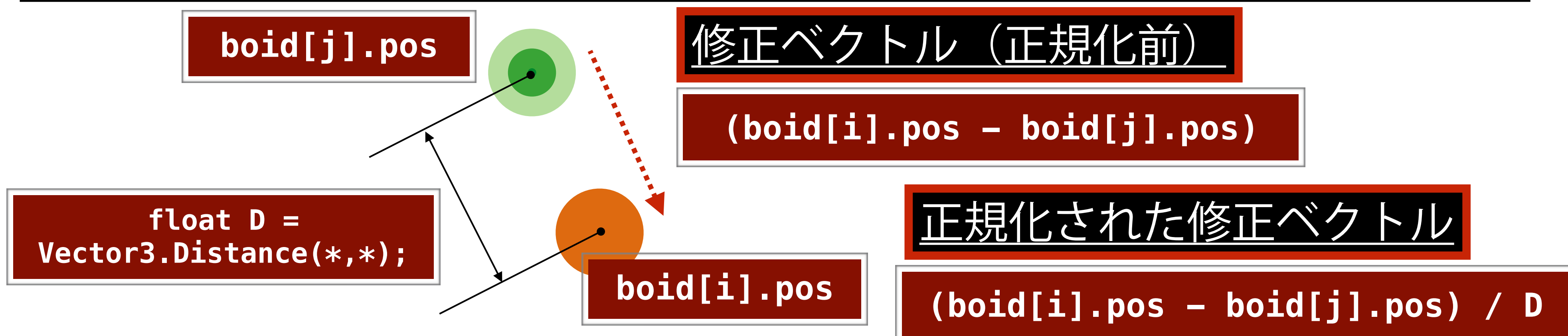
YES　２．修正ベクトルの計算

NO　ボイドｊ+1 の １．接触限界範囲の判定 へ

ルール２（分離）の実装

2. 修正ベクトルの計算





反発ルール（ルール２）関数内部の記述例

全てのボイド（i=0…pop-1）について，
1) 接触限界範囲内にあるボイドを見つけ，
2) 引数である係数c2を用いて反発する方向に速度を修正します．

BoidManager.cs

```
void ApplyRule2()
{
    for (int i = 0; i < bsum; i++) {

        Vector3 ipos = boid [i].pos;

        for (int j = 0; j < bsum; j++) {

            Vector3 jpos = boid [j].pos;
            float dis = Vector3.Distance (ipos, jpos);

            if (  1． 接触限界範囲の判定

                Vector3 cv = new Vector3 ();

                2． 修正ベクトルの計算

                boid [i].setVelocity (boid [i].vel + cv);
            }
        }
    }
}
```

i 番目のボイドの位置をiposとする．

j番目のボイドの位置をjposとする．

ボイド i とボイド j の距離を disとする．

neighbor_spaceを使って条件を書きます．

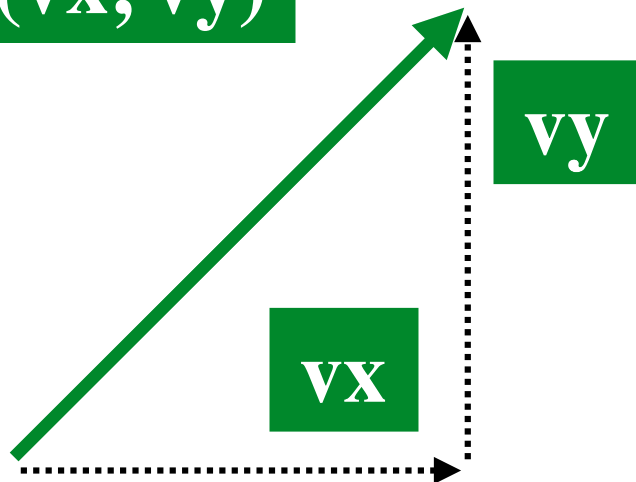
係数 c2 を使います．

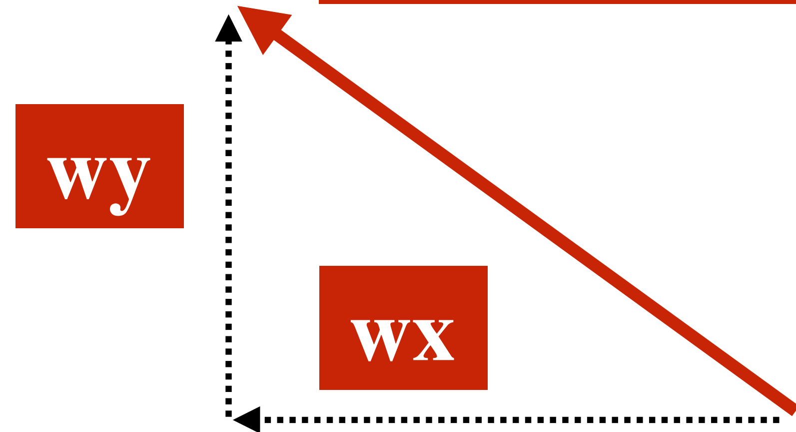
ボイド i の速度を更新．

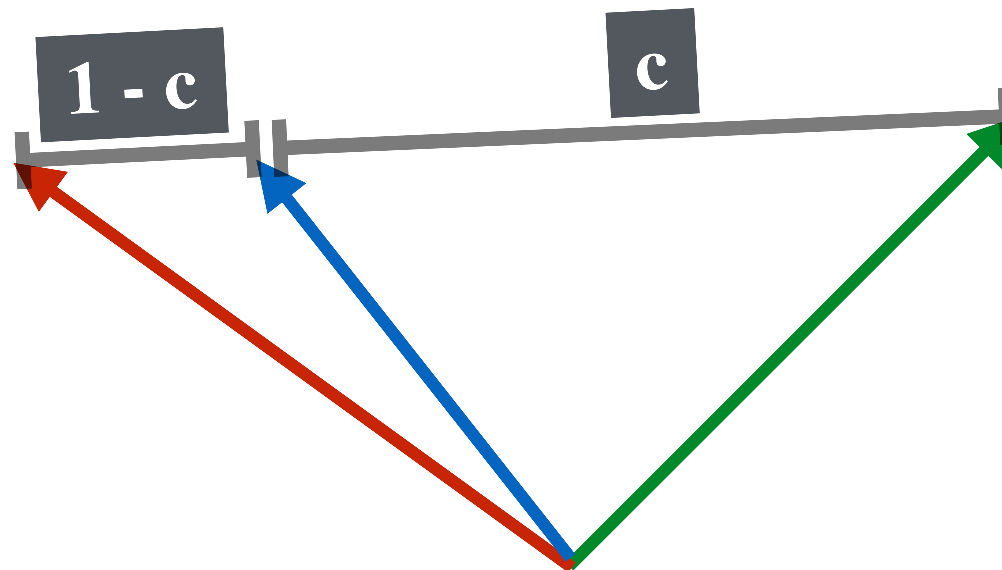
速度を，別の速度（マスターベクトル）に徐々に向けるためのベクトル計算

速度による引き込み
(内分ベクトルによる方法)

内分ベクトルの計算







v’とw’を c:1-c に内分するベクトル

(c*wx + (1 - c)*vx, c*wy + (1 - c)*vy)

<速度 v'> を<速度w'> の方向に修正する
（内分ベクトルによる方法）

マスターベクトル

w' = (wx, wy)

速度ベクトル

v' = (vx, vy)

1 - c3

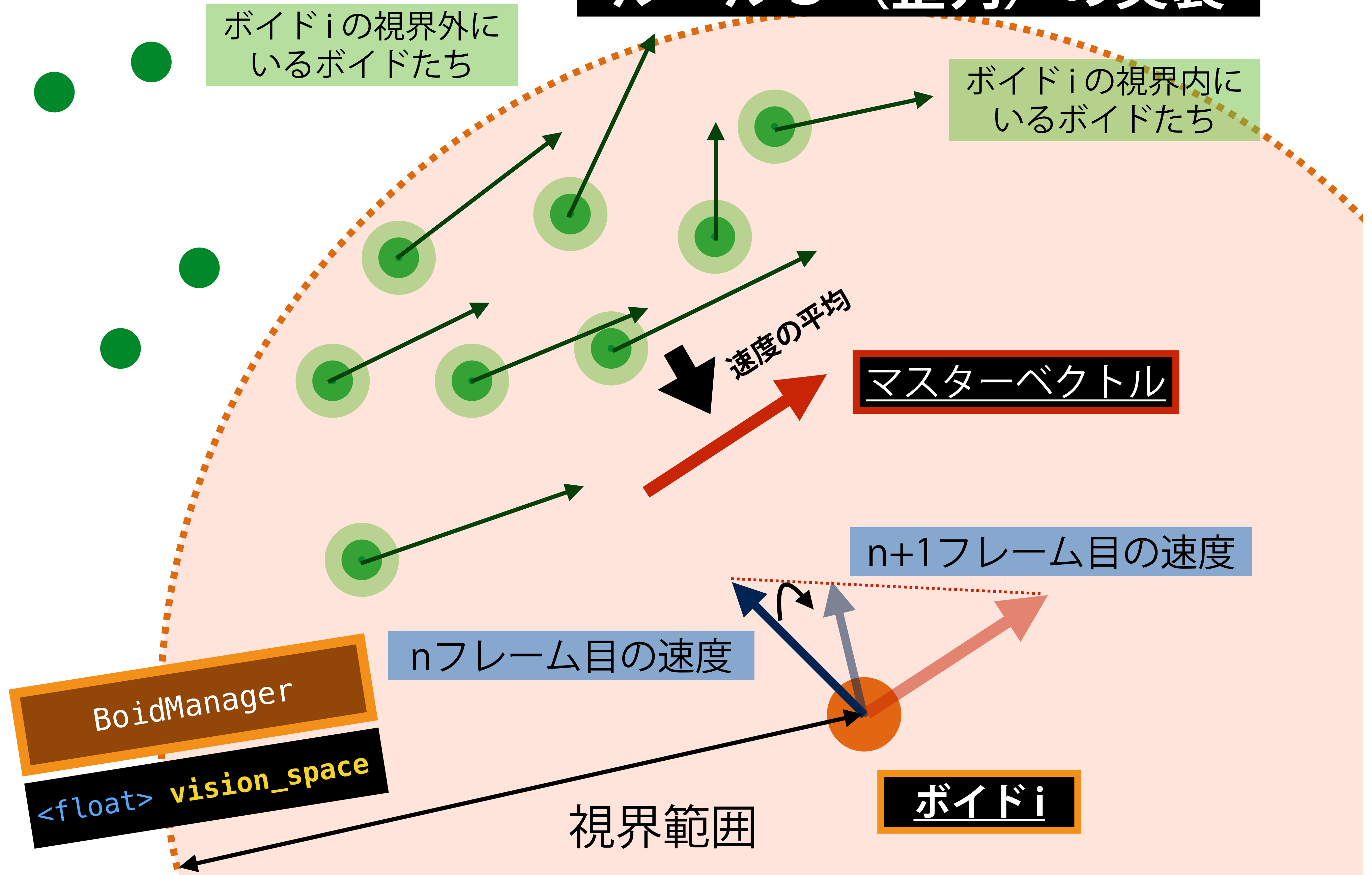
c3

更新された速度ベクトル

v' = (c3*wx+(1-c3)*vx,
c3*wy+(1-c3)*vy)

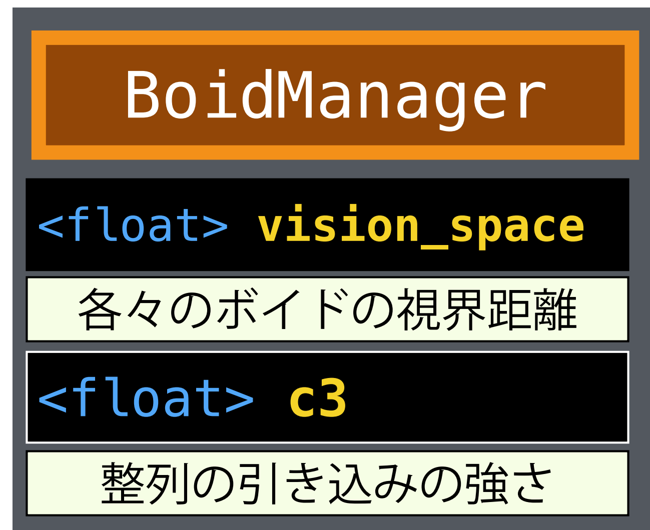
c3 が 1 に近い程, すぐマスターベクトルに引きこまれる.

ルール３（整列）



ルール３（整列）の実装

マスターベクトルの計算



ボイドｉの視界内にいるボイドたち

マスターベクトル

c3

1 - c3

ボイドｉ

視界内の全てのボイドのベクトルを集める.

全てのボイドの速度を総和し, 平均化したものをマスターベクトルとする.

整列ルール（ルール３）関数内部の記述例

BoidManager.cs

```
void ApplyRule3()
{
    for (int i = 0; i < bsum; i++) {
        Vector3 ipos = boid [i].pos;
        Vector3 ivel = boid [i].vel;

        float count = 0;
        Vector3 velSum = new Vector3 (0f, 0f, 0f);

        for (int j = 0; j < bsum; j++) {
            Vector3 jpos = boid [j].pos;
            Vector3 jvel = boid [j].vel;

            if (j!=i && Vector3.Distance (ipos, jpos) < vision_space) {

            }
        }

        if (count > 0) {

        }
    }
}
```

i 番目のボイドの位置・速度をipos, ivelとする.

count: 視界内にいるボイドの数

velSum: 視界内ボイドの速度総和

j番目のボイドの位置をjposとする.

count, velSumに対する処理を書きます.

係数 c3 を使います.