

演習 2 : 集合の知性を設計する

(05) 05/19 (06) 05/26

A | Unity環境の整備・簡単なルール設計

(07) 06/02 (08) 06/09

B | ボイドルール 1・2・3 の実装

(09) 06/16

C | 課題 1 : 集合知の解析

(10) 06/23

D 1 | SIR (感染モデル)

(11) 06/30 (12) 07/07 (13) 07/14

D 2 | 課題 2 : マイルール・感染ルール・視点操作

(14-15) 07/21

D 3 | 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)

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

```
//ルールの適用
public bool rule1 = false;
public bool rule2 = false;
public bool rule3 = false;
```

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

ApplyRules()

```
//ボイドルールマネージャ
BoidRuleManager rule;
```

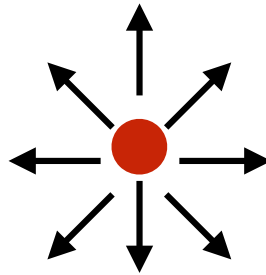
//ボイドルールマネージャによるルールの適用

```
rule.SetBoid(this);
rule.ApplyRule();
```

Update()

BoidManager.cs

BoidRuleManager.cs

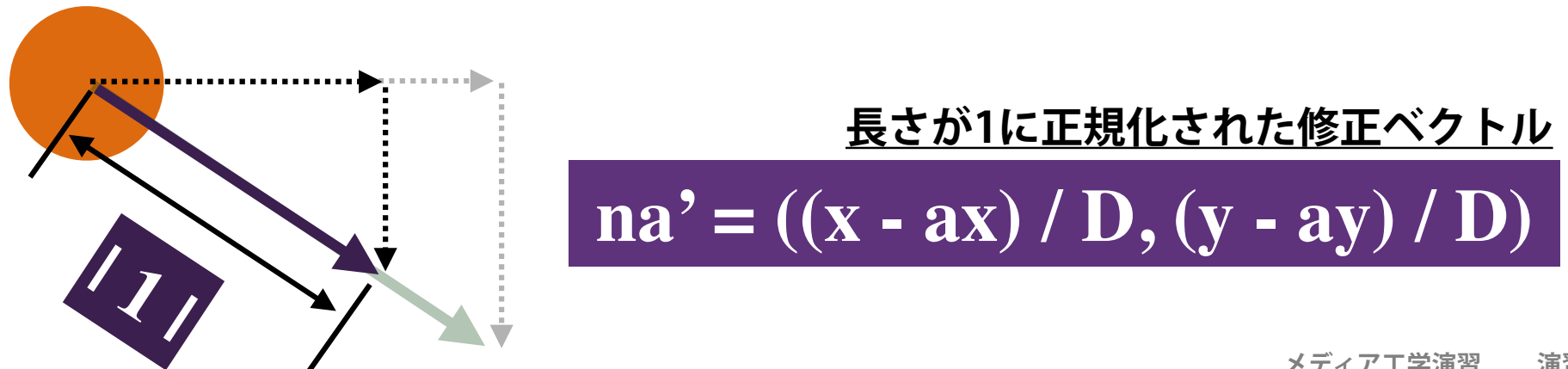
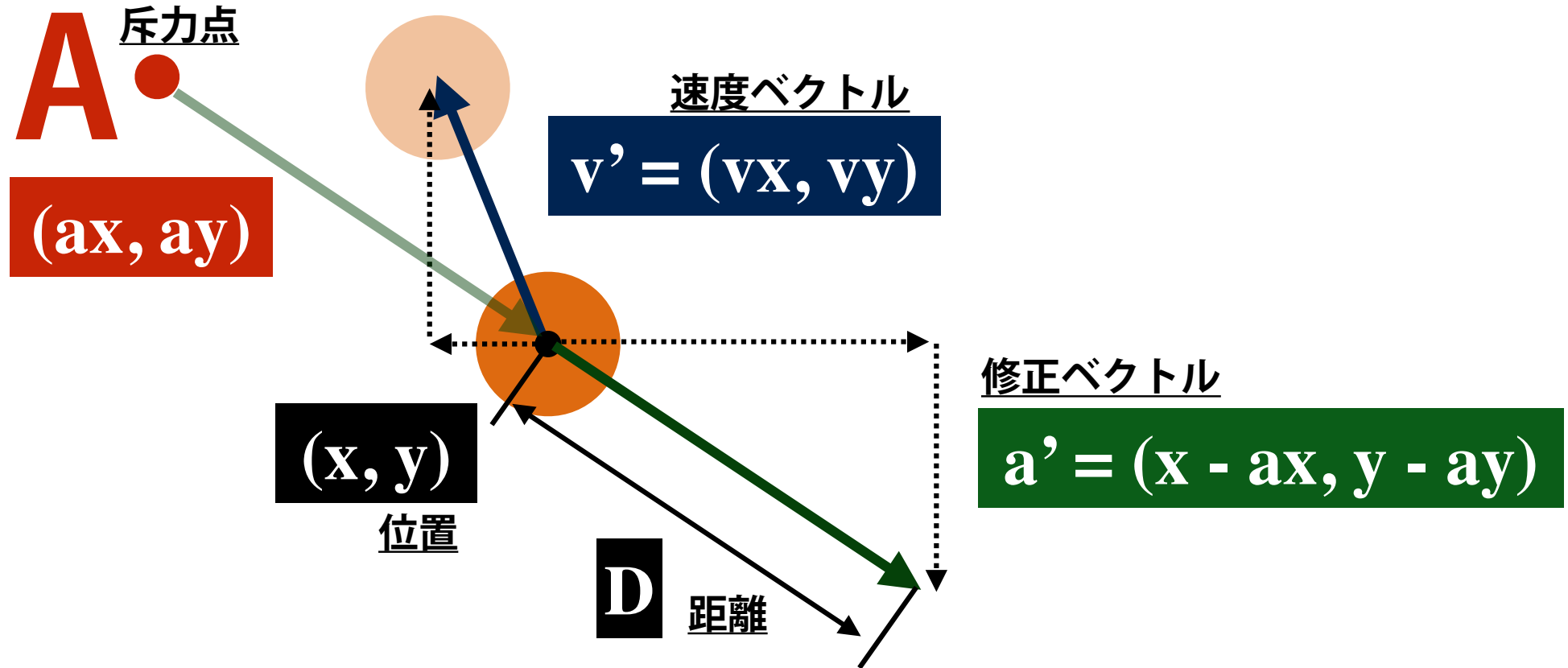


指定された点から
離れるようにするためのベクトル計算

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
 (ax, ay)

速度ベクトル

$$\mathbf{v}' = (v_x, v_y)$$

(x, y)

位置

修正ベクトル

$$\mathbf{a}' = (x - ax, y - ay)$$

D

距離

修正後の速度ベクトル

$$\mathbf{v}2' = (v_x + (x - ax) / D, v_y + (y - ay) / D)$$

係数 (c2) を付した三次元表現では, ...

$$\mathbf{v}2' = (v_x, v_y, v_z) + c2 * (x-ax, y-ay, z-az) / D$$

ルール 2 (反発)

BoidRuleManager

<int> **pop**

ボイドの個体数

<float> **c2**

斥力の強さ

SingleBoid

<Vector3> **pos**

各ボイドの三次元位置

<Vector3> **vel**

各ボイドの速度

<float> **neighbor_space**

各ボイドの接触限界距離

void **setVelocity(Vector3 v)**

速度更新メソッド

ルール 2 (分離) の実装

for(int I=0;i<pop;i++)

ボイド i

for(int j=0;j<pop;j++)

ボイド j

neighbor_space

1. 接触限界範囲の判定

NO

YES

2. 修正ベクトルの計算

c2

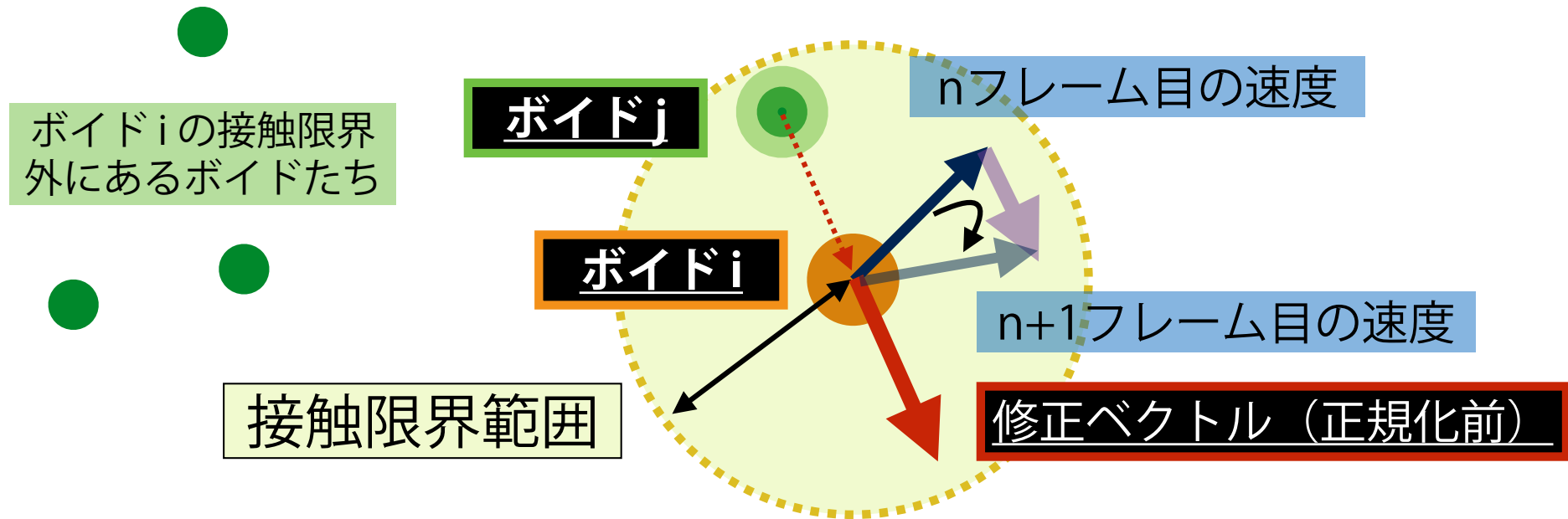
3. 速度の修正

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します。

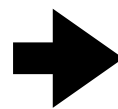
ルール 2 (反発)

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

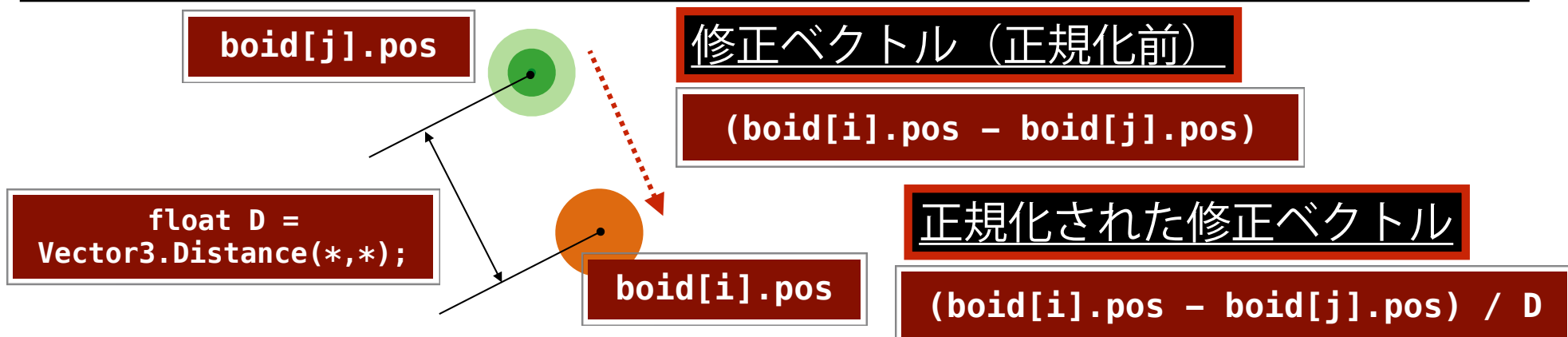
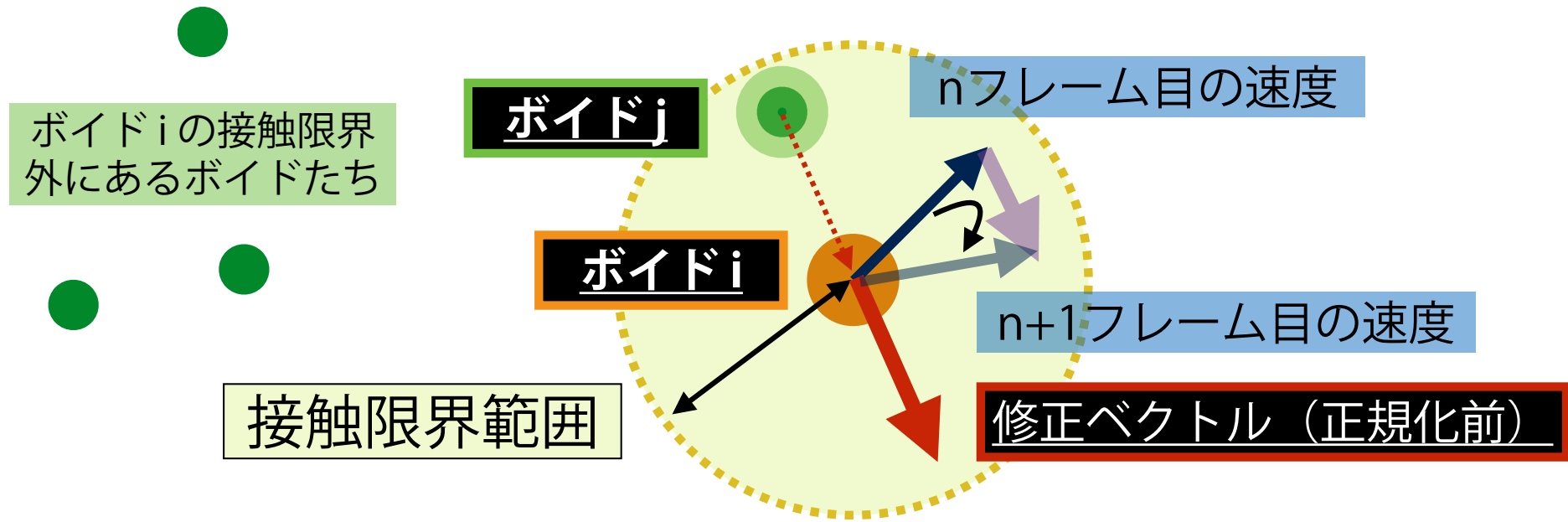
1. 接触限界範囲の判定



ルール 2 (反発)

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 ($i=0\dots\text{pop}-1$) 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。

BoidRuleManager.cs

```
void ApplyRule2()
```

```
{
```

```
    for (int i = 0; i < pop ; i++) {
```

```
        Vector3 ipos = boid [i].pos; i 番目のボイドの位置をiposとする.
```

```
        for (int j = 0; j < pop ; j++) { j 番目のボイドの位置をjposとする.
```

```
            Vector3 jpos = boid [j].pos; ボイド i とボイド j の距離を dis とする.  
            float dis = Vector3.Distance (ipos, jpos);
```

```
            if ( 1. 接触限界範囲の判定 neighbor_space を使って条件を書きます.
```

```
                Vector3 cv = new Vector3 ();
```

```
                2. 修正ベクトルの計算 係数 c2 を使います.
```

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

```
                ボイド i の速度を更新.
```

```
            }
```

```
        }
```

```
    }
```

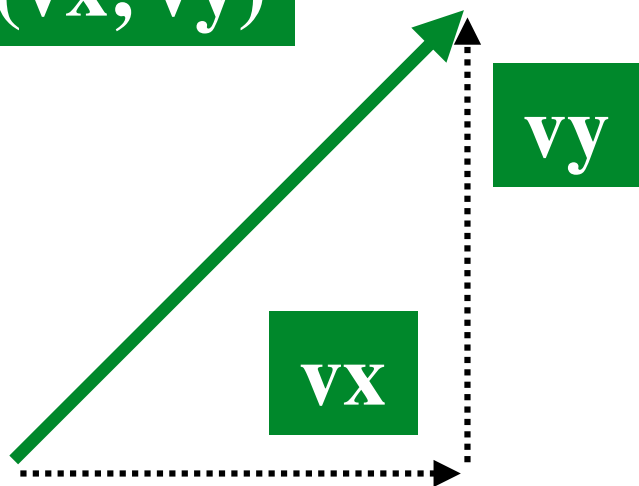
```
}
```

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

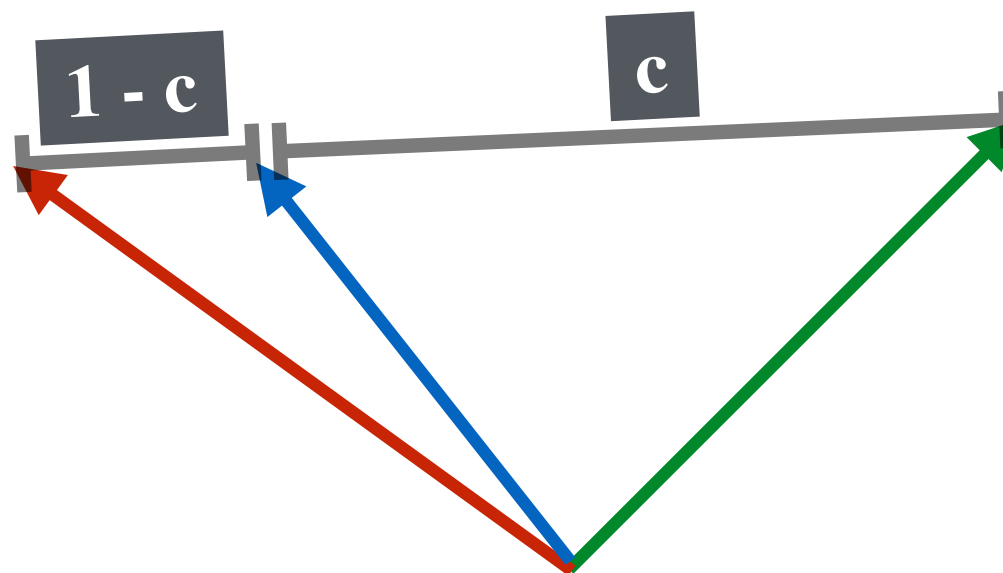
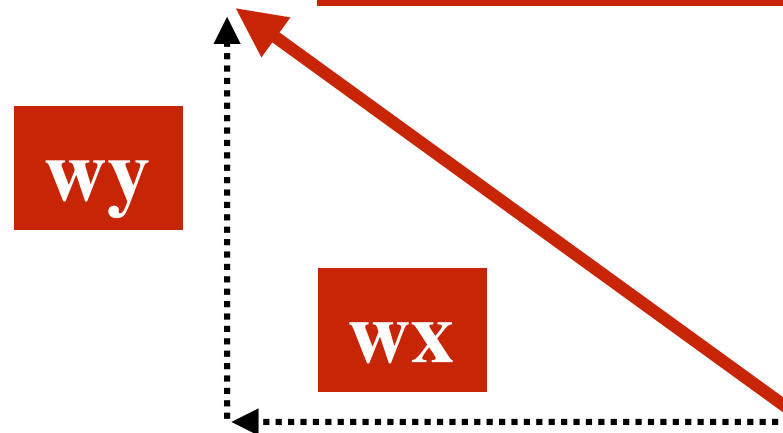
速度による引き込み
(内分ベクトルによる方法)

内分ベクトルの計算

$$v' = (v_x, v_y)$$



$$w' = (w_x, w_y)$$



v' と w' を $c:1-c$ に内分するベクトル

$$(c * w_x + (1 - c) * v_x, c * w_y + (1 - c) * v_y)$$

<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

$$w' = (w_x, w_y)$$

速度ベクトル

$$v' = (v_x, v_y)$$

更新された速度ベクトル

$$v' = (c_3 * w_x + (1 - c_3) * v_x, \\ c_3 * w_y + (1 - c_3) * v_y)$$

c_3 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

ボイド*i*の視界外に
いるボイドたち

ボイド*i*の視界内
にいるボイドたち

速度の平均

マスターベクトル

n+1フレーム目の速度

nフレーム目の速度

視界範囲

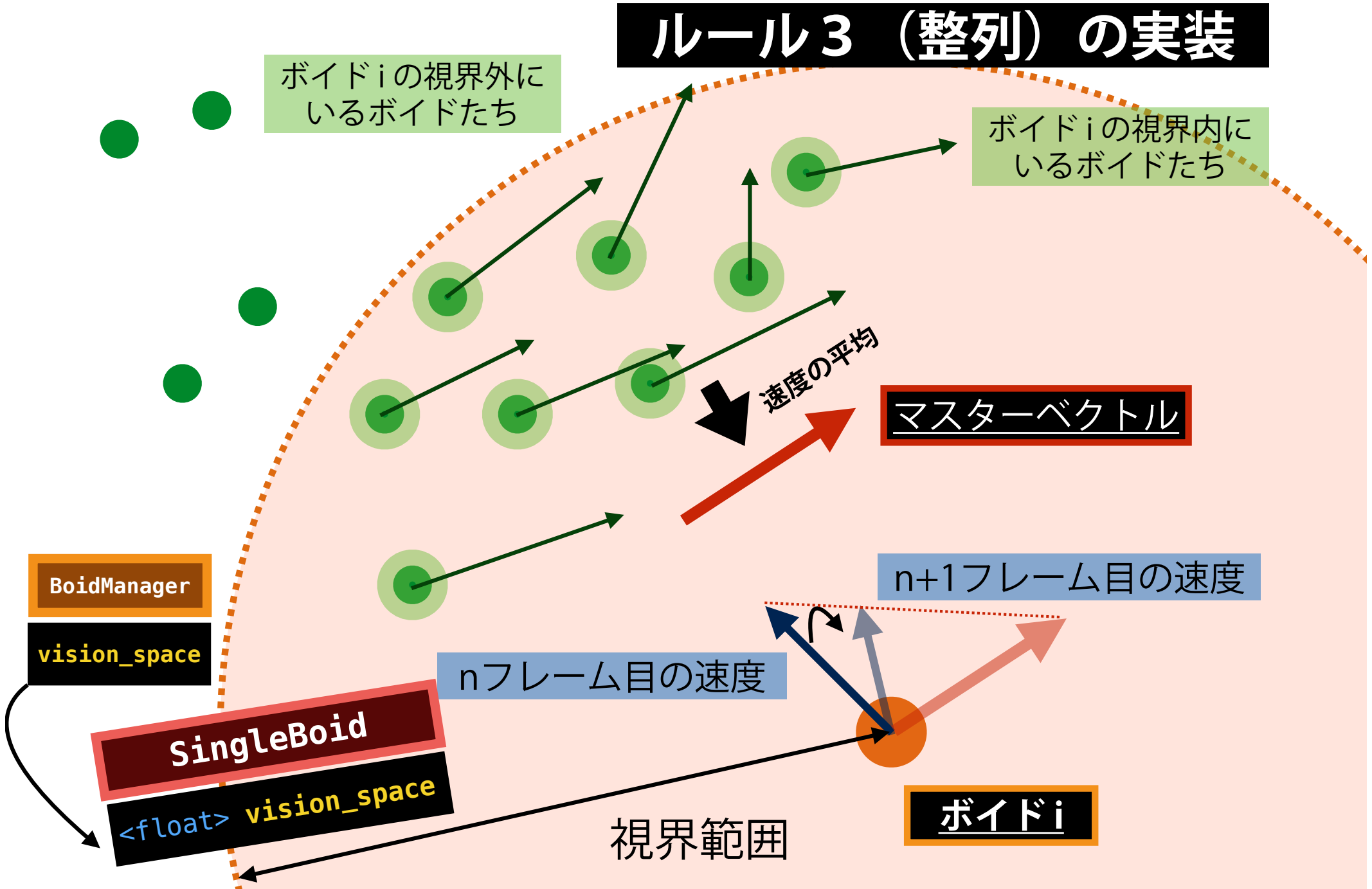
ボイド*i*

BoidManager

vision_space

SingleBoid

<float> vision_space



ルール3 (整列) の実装

マスターベクトルの計算

SingleBoid

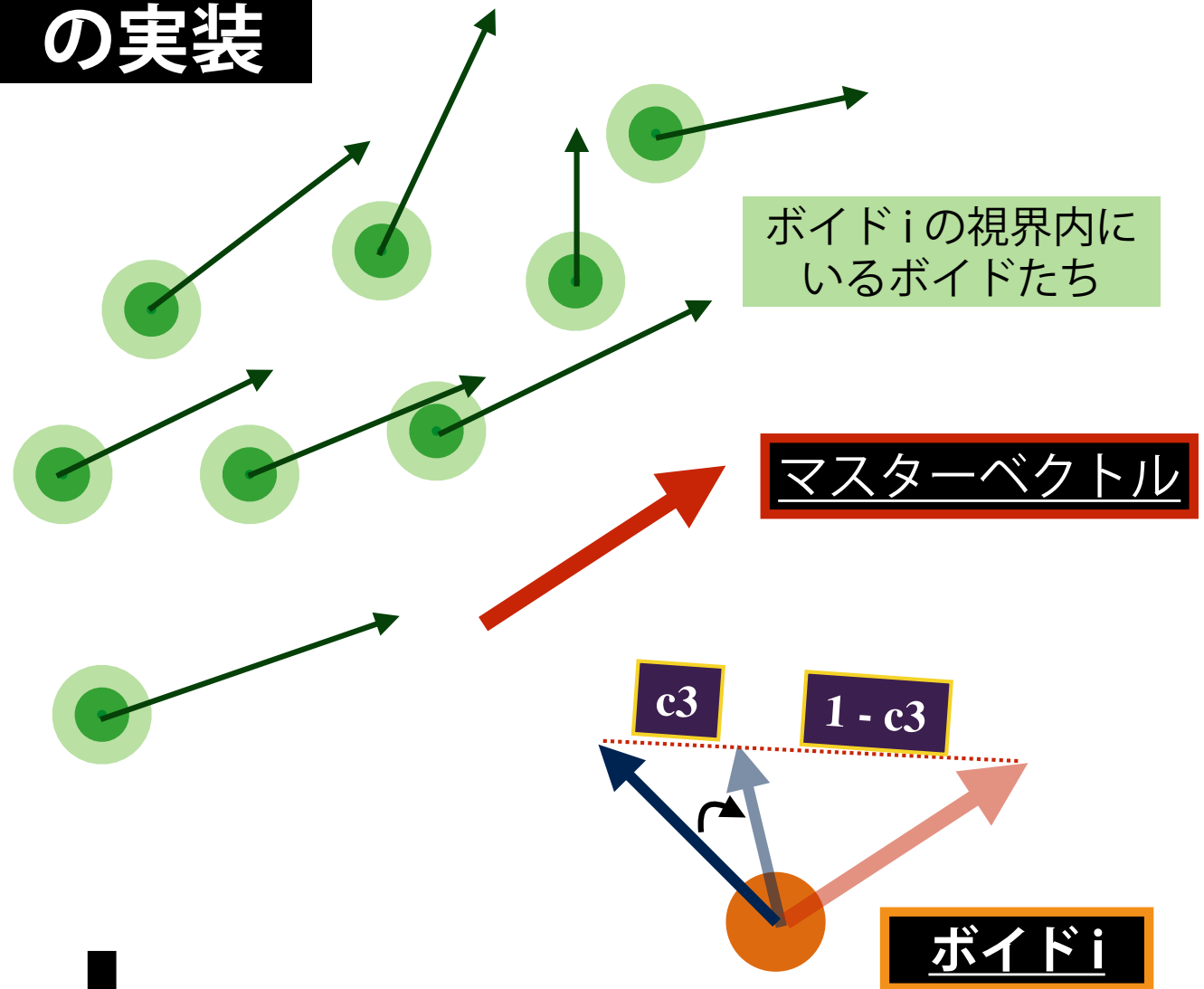
<float> vision_space

各ボイドの視界距離

BoidRuleManager

<float> c3

整列の引き込みの強さ



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

```
void ApplyRule3()
```

```
{
```

```
    for (int i = 0; i < pop; i++) {
```

```
        Vector3 ipos = boid [i].pos;
```

```
        Vector3 ivel = boid [i].vel;
```

```
        float ivision_space = boid[i].vision_space;
```

```
        float count = 0;
```

```
        Vector3 velSum = new Vector3 (0f, 0f, 0f);
```

```
        for (int j = 0; j < pop; j++) {
```

```
            Vector3 jpos = boid [j].pos;
```

```
            Vector3 jvel = boid [j].vel;
```

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

```
                [Redacted]
```

```
            }
```

```
        }
```

```
        if (count > 0) {
```

```
[Redacted]
```

```
        }
```

```
    }
```

i 番目のボイドの位置・速度・視界距離を ipos, ivel, ivs 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。

BoidRuleManager.cs