Applying Physical Forces in Drone Path Planning algorithm

2019. 02. 14

Donghyun Koh(andy1902@gmail.com), Junggyu Bae (baejgk@gmail.com)

Athman Bouguettaya (athman.bouguettaya@sydney.edu.au)

Jonghwa Park (suakii@gmail.com)

Applying Physical Forces in Drone Path Planning algorithm

Donghyun Koh (andy1902@naver.com),

Junggyu Bae (baejgk@gmail.com)

Athman Bouguettaya
(athman.bouguettaya@sydney.edu.au)

Jonghwa Park (suakii@gmail.com)

Abstract

What is the most important factor when the drones fly from the sky? It may be a factor related to safety and speed. By default, the drones must not collide with each other when moving. Also, each drone should reach its destination as fast as possible without collision. In this study, we tried to construct an algorithm to reach the destination most effectively by defining the attraction and the repulsive force between drones. In this study, the most effective constants were found by calculating the total reaching time according to the constant values of the equation in the attraction force and the repulsive force acting between the drones. Using this constant, we applied it to the flight of packed drones, and it was possible to create an algorithm that made the packed drones reach its distance without colliding with each other.

Applying Physical Forces in Drone Path Planning algorithm

I. Introduction

1.1 The purpose of the research

These days, things are delivered home by truck or motorcycle. However, this delivery method is not an effective delivery method because it depended on the road situation. Recently, an effective method of transporting goods, not limited to road conditions, has been proposed. It is a form that carries objects through drones. The first thing to consider when transporting drones is safety. If the drones collide in the sky, it can lead to serious accidents such as damage to property and damage to people under the sky. So many people are trying to develop algorithms to make the drones reach their destination without colliding. In this study, we made an algorithm that does not collide with other drones when the drone reaches the target point.

1.2 Theoretical background

1.2.1 Coulomb's law

Coulomb's law, or Coulomb's inverse-square law, is a law of physics for quantifying Coulomb's force, or electrostatic force. Electrostatic force is the amount of force with which stationary, electrically charged particles either repel, or attract each other. This force and the law for quantifying it, represent one of the most basic forms of force used in the physical

sciences, and were an essential basis to the study and development of the theory and field of classical electromagnetism. The law was first published in 1785 by French physicist Charles-Augustin de Coulomb. [1]

In its scalar form, the law is:

$$F = k_e \frac{q_1 q_2}{r^2} \quad (k = 9 \times 10^9 Nm^2 C^{-2})$$

q1 and q2 are the signed magnitudes of the charges, and the scalar r is the distance between the charges. The force of the interaction between the charges is attractive if the charges have opposite signs. The Coulomb's law between two elctrons are as follows. [1]

$$F = k \frac{e^2}{r^2} \quad (e = 1.602 \times 10^{-19})$$

1.2.1 Electric current

An electric current is a flow of electric charge. In electric circuits this charge is often carried by moving electrons in a wire. It can also be carried by ions in an electrolyte, or by both ions and electrons such as in an ionised gas (plasma). [2]

1.3 Precedent Research

Prior to the present study, we developed an algorithm to reach each destination without colliding with many drones in two dimensions. Its name is Dynamic Partial Preempting Algorithm.

1.3.1 Dynamic Partial preempting Algorithm

This algorithm is an algorithm in which the drones preempt their paths

in randomly assigned order and move the next drones ahead of the preempted paths. First, we developed an optimal algorithm for a single drone and developed it using it. The following are variables and functions used in this algorithm.

- **vector<Point> finalpath[]**

It is the final 'set' to get. finalpath [i] is the path set of the drones i.

- **Point P_next[]**

The partial preemption is made only to the next point on the path for each drone, and the point preempted by the drone i is called P_next [i].

- **vector Segs_Preempt**

It is a 'set' of partial preempted paths for each drone as a line segment.

- **Point find_next(Ps,Pg)**

This function is a function that returns the next point of Ps in the optimal path where one drone goes from Ps to Pg.

Algorithm 1 Find_Path

```
mindist←L (L : Safety distance)
while All drone hasn't finished do
  for i ≤ n do
    if Drone_i is Arrived then
      continue
    end if
    P_next[i]←find_next(S[i],G[i])
    mindist←min(mindist,dist(S[i],P_next[i]))
    Tmp←Point which distance between S[i] and Tmp is mindist and which is inner division
    point of segment (S[i],P_next[i])
    Add segment (S[i],Tmp) to Segs_Preempt
    Add Tmp to Preempt
  end for
  for i ≤ n do
    Q←Point which distance between S[i] and Q is mindist and which is inner division point
    of segment (S[i],P_next[i])
    Add Q to finalpath[i]
    S[i]←Q
    clear Segs_Preempt, Preempt
  end for
end while
```

Fig 1. Dynamic Partial preempting Algorithm's Pseudocode

We implemented the above algorithm, obtained the path, and visualized it.

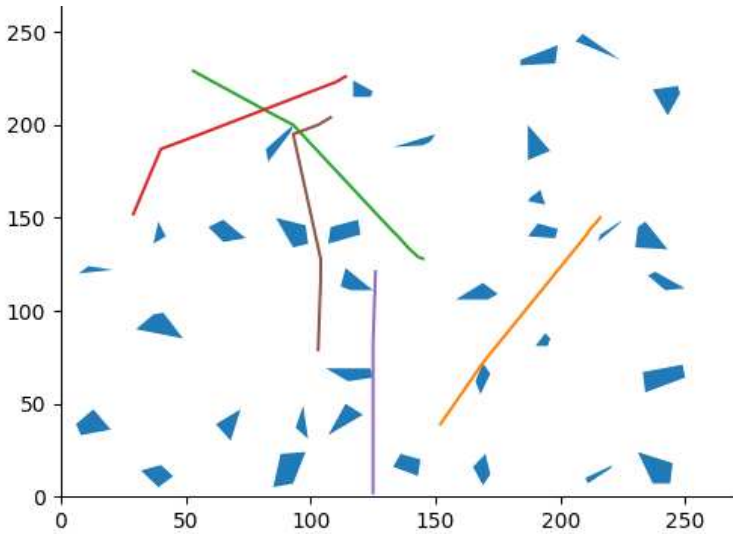


Fig 2. Path of 5 drones

1.3.2 Algorithm's Performance

To verify the efficiency of the above algorithm, we simulated this algorithm with Gazebo. As a result of simulating 15 drones, the time taken for all drones to arrive increased by 7.05% on average compared to before the algorithm was applied. On the other hand, the minimum distance between the drones increased by about 19.2% on average. That is to say, the safety is higher.

1.3.3 Problem of the algorithm

After applying the Dynamic Partial Preempting Algorithm, the collision of drones was reduced. However, this algorithm need to be improved because the collision of drones was still occurring.

II. Research process and method

2.1 Outline of Research

2.1.1 Idea of Research

The idea of the drone's attraction and repulsive force came from the equation $F=k\frac{e^2}{r^2}$. If the drones are repulsive, the drones will be able to reach the target without colliding. Therefore, in this study, we used an attractive and repulsive force as $F=\frac{G}{r^n}$. This force has made it possible to calculate the time it takes for the drones to reach their destination.

2.1.2 Goals of Research

The objectives of this study are as follows.

1. Ensure that the force between the drones defined as $F=\frac{G}{r^n}$ can reach the destination without collision.

2. Graph the total time of arrival from the force $F=\frac{G}{r^n}$ to the constant and find out the most effective constant.

3. Implement the drones' pack flight using the power obtained from 2

2.2 Implementation of the movement of the 2D drone through 'Processing'

First, we tried to understand the validity of the algorithm in the language Processing. By selecting a language called Processing, we were able to visually check the drones' movements, fix problems, and modify the algorithm to better match the actual situation.

First, we defined the drones' repulsive force as follows.

$$F=\begin{cases}\frac{G}{r^n} & (0 \le r < 3s) \\ 0 & (3s \le r)\end{cases}_{(s\,:\,safety\,distance)}$$

The reason for defining the force as zero for distances greater than three times the safety distance is that the drone need not consider the force at a distance and to avoid a sudden increase in the amount of calculation when the amount of drones increases.

Second, the drones were subjected to some noise in the repulsive force. This was to prevent drones from vibrating. Consider the

Fig 3. When a drone is vibrates despite the repulsion

situation shown in Fig 3. The situation shown in Figure 1 is when there are other drones in the same direction as the destination. In this situation, when the drones approach each other, they move toward the destination without being repulsive. Then, when the distance between them is less than three times the safety distance, the repulsive force acts and moves in the opposite direction of the destination. If the distance is more than three times the safety distance again, the drones move to the destination direction. As this phenomenon continued, the drones oscillated. Therefore, the program was designed so that the drone turns around each other by giving a small noise value in an arbitrary direction perpendicular to the direction of the repulsive force of the drone.

Considering this situation, the power of the drones can be summarized as follows.

$$\vec{a_i} = \frac{\overrightarrow{destination} - \overrightarrow{position_i}}{\left|\overrightarrow{destination} - \overrightarrow{position_i}\right|} size\,of\,potential\,difference + \sum_{j=1, j\neq i}^{N} \left(\frac{G}{\left|position_i - position_j\right|^n}\hat{r} + \overrightarrow{noise}\right)$$

$$\triangle\vec{r_i} = \frac{\vec{a_i}}{\left|\vec{a_i}\right|} * \max size\,of\,velosity$$

N above is the total number of drones. $\hat{r}$ and $\overrightarrow{noise}$ are vectors perpendicular to each other.

If we apply the force like the above process, we can confirm that it reaches without collision from the two-dimensional space to each destination.

2.3 Implementation of 3D drone motion using 'C'

In 2.2, if the 2D drone movement was implemented in Processing, which

was focused on visualization, the purpose of C ++ implementation was to collect data quickly. The implementation method and the expression used are the same as Processing.

2.4 Implementation of Drones' Pack Flight using 'Processing'

First of all, we will explain the validity of why the idea of acting repulsive in the drones' flight is necessary. The collision between clusters can be prevented through the repulsive force of the drones. However, there may be a question as to why attractive and repulsive forces must act in the drones' packs. Theoretically, if all the drones move in the same direction and at the same speed in the initial state of the drones, there is no collision between the drones in the cluster at all. However, in the actual sky, the element of wind is largely influenced, and there is a possibility that a collision or dislocation of a drones within the community occurs. Therefore, this section explains the process of implementing the cluster flight of drones considering these factors.

We have implemented the cluster flight of the drone through the constant values obtained from Section 2.3. In order to realize the cluster flight of drones, a certain number of drones were grouped into one class and defined as one class. In the defined class, the element that distinguishes each drone is the center of gravity and the radius.

There are two main reasons for defining the center of gravity. First, to define the vector to the destination. At each moment, the center of gravity of the pack was defined, and the vector between the destination and the center of gravity was obtained. This is done for each drone so that the drone can reach its destination. The second reason is to maintain the pack. To maintain the pack, the force between the drones

in the cluster and the center of gravity was defined as follows.

$$F=\begin{cases}0 & (0 \le r < 5s) \\ \dfrac{G}{r^n} & (5s \le r)\end{cases} \quad (s : safety distance)$$

In addition, all drones in the pack have a repulsive force between each other.

The reason for newly defining the radius in the cluster class is to prevent collision between the clusters. First, the radius is the distance from the drone, which is the furthest away from the center of gravity of the drones. By defining the radius, the pack can be seen as a new large drone with a radius of radius, and collision between packs is prevented by defining repulsive force between the packs so that the distance between the packs acting like a drone does not become less than the safety distance.

III. Results and Discussion

3.1 Relation between T,n (results of 2.3)

When the time it takes for the drones to arrive at their destination is T, suppose $G=1$ in $F=\frac{G}{r^n}$ to know relationship between n and T. And we have simulated many cases. As a result, the T-n graph has two major types.

- Type which has reduced section

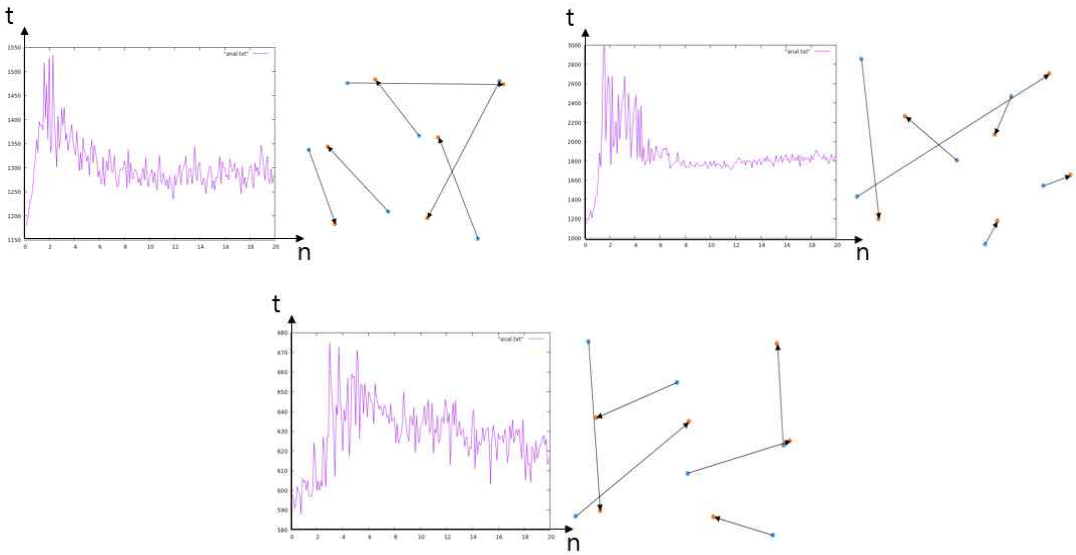


Fig 4. The graph with the decreasing section and the movement path of each drone

- Type without reduction section

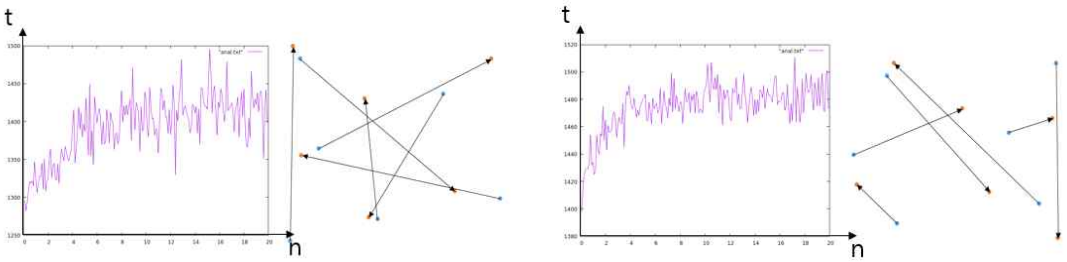
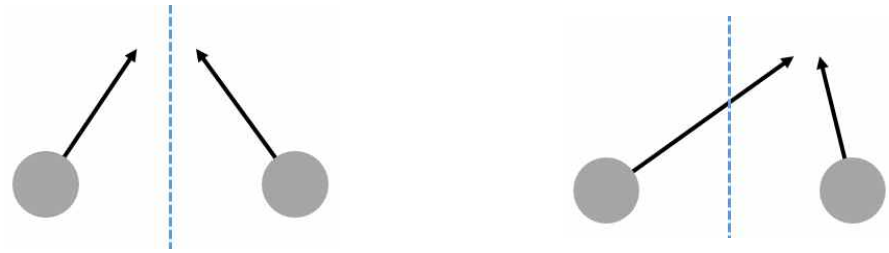


Fig 5. The graph with no reduction interval and the movement path of each drone

The reason why the two types of graphs come out is that the distribution of the origin and destination of the drones is different. I set up these assumptions and analyzed how they differ. As a result, we have found that the paths of several drones are symmetric in the graph with no decreasing interval.



(a) Symmetric example (b) Asymmetric example

Fig 6. Symmetry and Asymmetry

To confirm this hypothesis, we simulated the situation where two drones collided symmetrically. As a result, the following graph was drawn regardless of the angle formed by the path of the two drones.

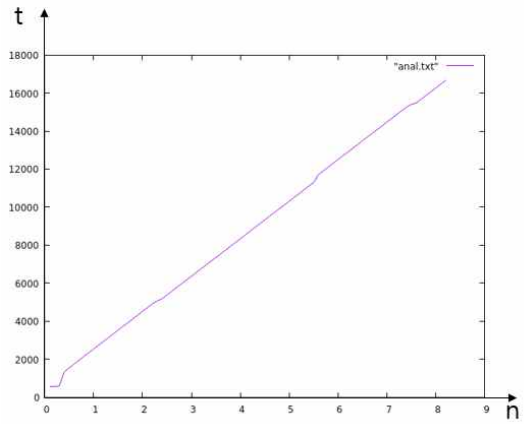
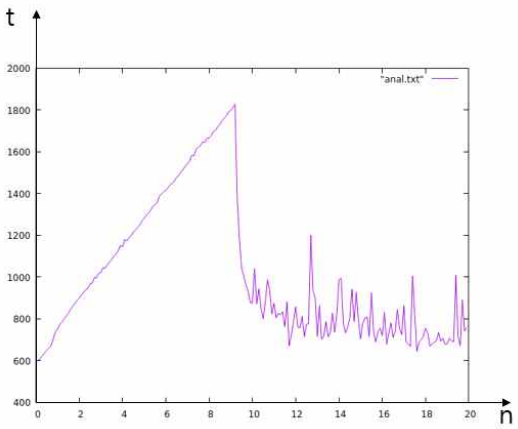
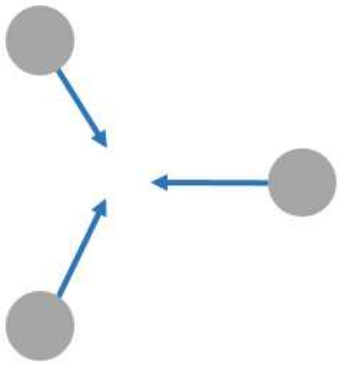


Fig 7. T-n graph of a two drones with symmetrical path

Similarly, the graph for three and four drones is as follows.

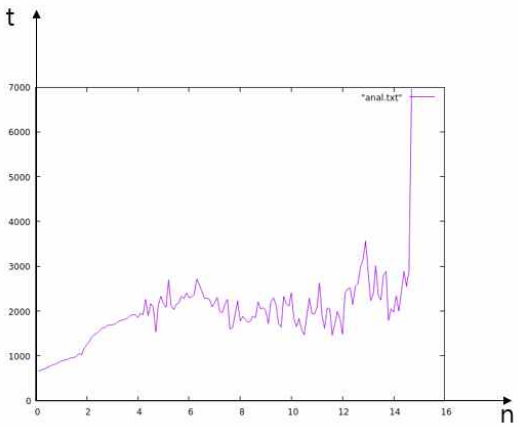


(a) T-n graph

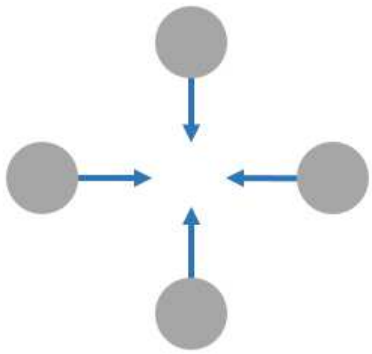


(b) Three symmetrical drones

Fig 8. T-n graph of three drones with symmetrical path



(a) T-n graph



(b) Four symmetrical drones

Fig 9. T-n graph of four drones with symmetrical path

When the drones collide symmetrically with each other, the smaller n is, the less time is required. That is, the correlation between n and T is high. On the other hand, when the path of the drones is not symmetrical, the distance between the drones does not have much effect, so that G and T are more relevant.

3.2 Suggestions

The size and the direction of the noise vector are random. This gives a huge influence in T, which is the time when all the drones arrived at their destination. Therefore, the alteration of the noise vector would be another future task.

Acknowledgments

This study was conducted as part of the ORP of the Gyeonggi Science High School in 2019.

We would like to thank Professor Athman Bouguettaya of The University of Sydney - School of Computer Science for this work.

IV. References

[1] Coulombs' law : Wikipedia : https://en.wikipedia.org/wiki/Coulomb%27s_law

[2] Electric current : https://en.wikipedia.org/wiki/Electric_current

V. Appendix

The appendix contains the code we wrote and used during the study.

1. Implementation of the movement of the 2D drone through 'Processing' (2.2)

```
float velsi = 2;
float dronenum = 20;
float safedist = 50;
float errange = 5;
float N = 2;
float G = 100;
PImage dronepic;

ArrayList<Drone> drone;
ArrayList<PVector> start,end;

void setup()
{
  size(800,800);
  background(255);
  dronepic = loadImage("dp.PNG");
  dronepic.resize(20,20);

  start = new ArrayList<PVector>();
  end = new ArrayList<PVector>();
  drone = new ArrayList<Drone>();

  for(int i=0;i<dronenum;i++)
  {
    PVector tmp1;
    PVector tmp2;

    while(true)
    {

      tmp1 = new PVector(random(0,width),random(0,height));

      if(checkst(i,tmp1) == true)
      {
        start.add(tmp1);
        //println(tmp1);
        break;
      }

    }

    while(true)
    {
      tmp2 = new PVector(random(0,width),random(0,height));

      if(checken(i,tmp2) == true)
```

```
        {
          end.add(tmp2);
          //println(tmp2);
          break;
        }

      }

      drone.add(new Drone(tmp1,tmp2,velsi,10));
    }

    //drone.add(new Drone(new PVector(0,0),new PVector(width,height),velsi,10));
    //drone.add(new Drone(new PVector(width,height),new PVector(0,0),velsi,10));
    //drone.add(new Drone(new PVector(width,0),new PVector(0,height),velsi,10));
    //drone.add(new Drone(new PVector(0,height),new PVector(width,0),velsi,10));
}

void draw()
{
  background(255);

  for(int i=0;i<drone.size();i++)
  {
    fill(0);
    //ellipse(start[i].x,start[i].y,100,10);

    fill(255,0,0);
    ellipse(end.get(i).x,end.get(i).y,10,10);
  }

  for(int i=0;i<drone.size();i++) drone.get(i).display();

  for(int i=0;i<drone.size();i++)
  {
    PVector tmp;
    tmp = new PVector(0,0);

    for(int j=0;j<drone.size();j++)
    {
      if(i==j) continue;

      Drone tmp1=drone.get(i);
      Drone tmp2=drone.get(j);
      float di = dist(tmp1,tmp2);

      if(di < safedist) println("Doomed!");

      if(di < safedist + 2*velsi && di > safedist)
      {
        PVector force = PVector.sub(tmp1.pos,tmp2.pos);
        force.normalize();
        force.mult(G*pow(velsi,2*N)/(pow(di-safedist,N)));

        tmp.add(PVector.add(force,(PVector.random2D()).mult(100)));
      }
```

```
      }

      drone.get(i).addForce(tmp);
    }

    for(int i=0;i<drone.size();i++) drone.get(i).Process();

    for(int i=drone.size()-1;i>=0;i--)
    {
      Drone tmp = drone.get(i);
      if(PVector.dist(tmp.pos,end.get(i)) < errange)
      {
        drone.remove(i);
        start.remove(i);
        end.remove(i);
      }
    }

}

boolean checkst(int x,PVector y)
{
  for(int i=0;i<x;i++)
  {
    if(PVector.dist(y,start.get(i)) < safedist) return false;
  }
  return true;
}

boolean checken(int x,PVector y)
{
  for(int i=0;i<x;i++)
  {
    if(PVector.dist(y,end.get(i)) < safedist) return false;
  }
  return true;
}


class Drone
{
  float velsize,rad;
  PVector pos,vel,dest;
  int idx;

  Drone(PVector tmppos, PVector tmpdest, float tmpvelsize, float tmprad)
  {
    pos = tmppos;
    dest = tmpdest;
    velsize = tmpvelsize;
    rad = tmprad;

    vel = PVector.sub(dest,pos);
    vel.normalize();
```

```
    vel.mult(velsize);
  }

  void Process()
  {
    pos.add(vel);

    vel = PVector.sub(dest,pos);
    vel.normalize();
    vel.mult(velsize);
  }

  void addForce(PVector acc)
  {
    vel.add(acc);
    vel.normalize();
    vel.mult(velsize);
  }

  void display()
  {
    image(dronepic,pos.x,pos.y);
  }

}

float dist(Drone k, Drone l)
{
  return (PVector.sub(k.pos,l.pos)).mag();
}
```

2. Implementation of 3D drone motion using 'C' (2.3)

Due to the similarity of 1, the code is omitted.

3. Implementation of Drones' Pack Flight using 'Processing' (2.4)

```
float velsi = 1;
float dronenum = 5;
float safedist = 50;
float errange = 50;
float n = 0.4;
float G = 1000;
PImage dronepic;
int total = 3;

ArrayList<Pack> P;

void setup()
{
  size(800,800);
  background(255);
  dronepic = loadImage("dp.PNG");
```

```
  dronepic.resize(20,20);

  P = new ArrayList<Pack>();
  P.add(new Pack(new PVector(700,0),new PVector(0,800)));
  P.add(new Pack(new PVector(0,400),new PVector(800,400)));
  P.add(new Pack(new PVector(400,700),new PVector(400,0)));
}

void draw()
{
  background(255);

  for(int i=0;i<total;i++) P.get(i).cal_com_rad();
  for(int i=0;i<total;i++) P.get(i).acc();

  for(int i=0;i<total;i++){
    for(int j=0;j<i;j++){
      if(i!=j) push(i,j);
    }
  }

  wind();

  for(int i=0;i<total;i++) P.get(i).move();
  for(int i=0;i<total;i++) P.get(i).arrive();
  for(int i=0;i<total;i++) P.get(i).show();

  if(total==0) {
    background(255);
    noLoop();
  }
}

class Drone
{
  float velsize;
  PVector pos,vel;
  int idx;

  Drone(PVector tmppos, float tmpvelsize)
  {
    pos = tmppos;
    velsize = tmpvelsize;
    vel = new PVector(0,0);
  }

  void display()
  {
    image(dronepic,pos.x,pos.y);
  }

  void move()
  {
    vel.normalize();
    vel.mult(velsize);
```

```
    pos.add(vel);
    vel = new PVector(0,0);
  }

  void addForce(PVector acc)
  {
    vel.add(acc);
  }

}

float dist(Drone k, Drone l)
{
  return (PVector.sub(k.pos,l.pos)).mag();
}

class Pack
{
  PVector com, destination;
  float rad;
  ArrayList<Drone> drones;

  Pack(PVector temp_start, PVector temp_dest)
  {
    drones = new ArrayList<Drone>();
    for(int i=0;i<dronenum;i++)
    {
      PVector p;
      while(true)
      {
        p = new PVector(temp_start.x+random(0,100),temp_start.y+random(0,100));
        if(check(i,p) == true)
        {
          break;
        }
      }
      drones.add(new Drone(p,velsi));
    }

    destination = temp_dest;
  }

  void cal_com_rad()
  {
    com = new PVector(0,0);
    for(int i=0;i<drones.size();i++)
    {
      com.add(drones.get(i).pos);
    }
    com = new PVector(com.x/dronenum, com.y/dronenum);

    rad = 0;
    for(int i=0;i<drones.size();i++){
      if(rad<PVector.sub(com,drones.get(i).pos).mag()) rad = PVector.sub(com,drones.
get(i).pos).mag();
```

```
  }
}

void acc()
{
  for(int i=0;i<drones.size();i++)
  {
    PVector tmp = new PVector(0,0);

    // go to destination
    tmp.add(PVector.mult(PVector.sub(destination,com).normalize(),10));

    for(int j=0;j<drones.size();j++)
    {
      if(i==j) continue;

      //push
      Drone dronea= drones.get(i);
      Drone droneb= drones.get(j);
      float di = (PVector.sub(dronea.pos,droneb.pos)).mag();

      if(di < safedist) println("Crashed at pack"+1+" between "+i+" and "+j);

      if(di < safedist + 3*velsi && di > safedist)
      {
        PVector force = PVector.sub(dronea.pos,droneb.pos);
        force.normalize();
        force.mult(G*pow(velsi,2*n)/(pow(di-safedist,n)));
        tmp.add(PVector.add(force,(PVector.random2D()).mult(100)));
      }
    }

    float di = (PVector.sub(drones.get(i).pos,com)).mag();
    if(di > safedist + 5*velsi){
      PVector force = PVector.sub(com,drones.get(i).pos);
      force.normalize();
      force.mult(G*pow(velsi,2*n)/pow(di-safedist,n));
      tmp.add(force);
    }

    drones.get(i).addForce(tmp);
  }
}

void show()
{
  fill(255);
  //ellipse(com.x,com.y,2*rad,2*rad);
  //ellipse(com.x,com.y,5,5);
  //line(com.x,com.y,destination.x,destination.y);
  for(int i=0;i<drones.size();i++)
  {
    drones.get(i).display();
  }
}
```

```
  void arrive()
  {
    if((PVector.sub(com,destination)).mag() < errange){
      for(int i=drones.size()-1;i>=0;i--) drones.remove(i);
      println("arrived");
      total--;
    }
  }

  void move()
  {
    for(int i=0;i<drones.size();i++) drones.get(i).move();
  }

  void add_wind(PVector W)
  {
    for(int i=0;i<drones.size();i++){
      if(random(0,2)<1) drones.get(i).addForce(W);
    }
  }

  void Push_all(PVector P)
  {
    for(int i=0;i<drones.size();i++){drones.get(i).addForce(P);}
  }

  boolean check(int x,PVector y)
  {
    for(int i=0;i<x;i++)
    {
      if(PVector.dist(y,drones.get(i).pos) < safedist) return false;
    }
    return true;
  }
}

void push(int a, int b)
{
  PVector push = new PVector(0,0);
  float r1=P.get(a).rad, r2=P.get(b).rad;

  float dist = PVector.sub(P.get(a).com,P.get(b).com).mag();
  if(dist < r1+r2+safedist) println("Crashed between pack");
  if(dist < r1 + r2 + safedist + 3*velsi && dist > r1 + r2 + safedist){
    push = PVector.sub(P.get(a).com,P.get(b).com);
    push.normalize();
    push.mult(G*pow(velsi,2*n)/pow(dist-r1-r2-safedist,n));
    push.add((PVector.random2D()).mult(100));
  }

  P.get(a).Push_all(push);
  P.get(b).Push_all(PVector.mult(push,-1));
}
```

```
void wind() {

  fill(255);
  triangle(20,752,20,768,5,760);
  triangle(40,752,40,768,55,760);
  triangle(22,750,38,750,30,735);
  triangle(22,770,38,770,30,785);

  int k=0;
  PVector w = new PVector(0,0);
  if(keyPressed){
    if (key == CODED){
      fill(0);
      if (keyCode == UP)    {triangle(22,750,38,750,30,735); w = new PVector(0,-1);}
      if (keyCode == DOWN)  {triangle(22,770,38,770,30,785); w = new PVector(0,1);}
      if (keyCode == RIGHT) {triangle(40,752,40,768,55,760); w = new PVector(1,0);}
      if (keyCode == LEFT)  {triangle(20,752,20,768,5,760);  w = new PVector(-1,0);}
    }
  }
  w.mult(50);

  for(int i=0;i<total;i++) P.get(i).add_wind(w);

}
```