A spherical model For Assignment 1 "Gliding in Space"

Student Name: Keyang Qian

Student ID: u7261302

Course Id: COMP2310

Assessment: Assignment 1 "Gliding in Space"

Date of Completion: 17th September, 2021

DECLARATION: I hereby declare that the attached assignment is my own work. I understand that if I am suspected of plagiarism or another form of cheating, my work will be referred to the Examination Officer, which may result in me being expelled from the program.

Contents

1. Introduction

The report will introduce all major design decisions for assignment 1 "Gliding in Space", including the failed attempts. The repost will begin with attempts and design decisions for stage 2, and then attempts and design decisions for stage 3 and stage 4, and provide testing codes and results at last. The model can be described as a spherical model. The overall design goal is to keep as many vehicles alive as possible for as long as possible, while there may be subgoal for each attempt, e.g., the most important subgoal for stage 2 is to let vehicles be distributed evenly in the sphere around the energy globe.

2. Stage 2 Design

2.1 The Problem

There's a swarm of vehicles that can automatically follow simple default behaviors to keep them in motion together and avoid collisions. Vehicles have a local charge to keep them alive. They constantly consume energy to keep their onboard systems running and consume substantially more energy when accelerating or decelerating. They replenish their charge to full by passing energy globes in close proximity, and those run out of charge will disappear.

The overall design goal is to keep as many vehicles alive as possible for as long as possible. Stage 2 does not allow for a central coordinator, and all planning and scheduling now needs to be done on the individual vehicles only using local communication. [1]

The least distance that vehicle can detect the globe and get charged can be seen in

Swarm/swarm_configuration.ads, named as Energy_Globe_Detection with a value of 0.07 .

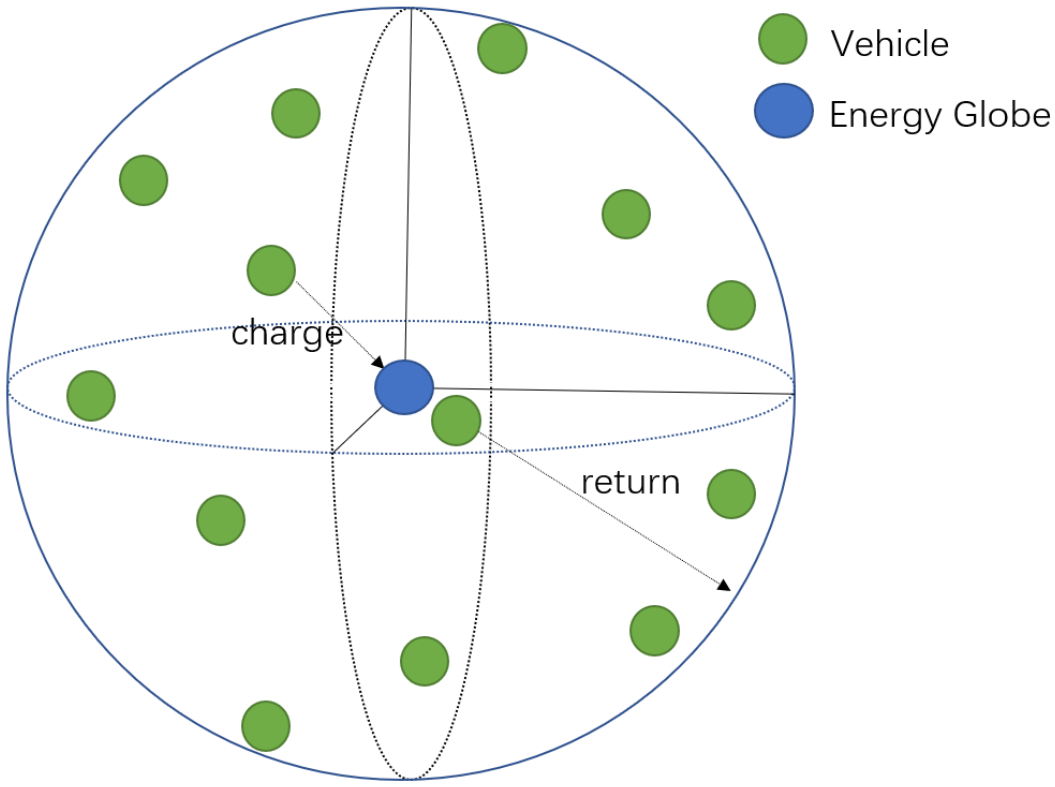
The longest distance within which distance vehicles can communicate with each other can be seen in Swarm/swarm_configuration.ads, named as Comms_Range with a value of 0.2 .

2.2 First Attempt

2.2.1 Basic Idea

There's only one globe at this stage, so the let vehicles evenly distributed in the space around the globe is a natural conclusion from theoretical analysis.

The basic idea of the first attempt is to let the vehicles form a sphere with the globe as center of sphere through message passing. Vehicles with low charge can automatically dash to the globe and return to its place after getting charged.



It can be called a spherical model. The report will use the spherical model to solve the whole assignment.

2.2.2 Basic Program Structure

vehicle_task_type.adb includes six parts:

1. Declaration and initialization;
2. Loop: see if find a energy globe;
3. Send Pre_Message;
4. Receive Message;
5. Update information with received message;

6. Set destination and throttle; End loop.

Pre_Message is the newest message information maintained by the vehicle. After declaration and initialization, task begins and runs in a loop until the vehicle disappears.

It can also be described by pseudocode:

```
Declaration and initialization;
select
    Flight_Termination.Stop;
then abort
    Outer_task_loop : loop
        Wait_For_Next_Physics_Update;
        if find_globe then
            update globe information;
        send Pre_Message;
        receive Message;
        update information by using Message;
        update Radius and set destination and throttle;
    end loop Outer_task_loop;
```

This is the basic program structure the report use for all stages.

2.2.3 Message Structure

Basic message requirement includes:

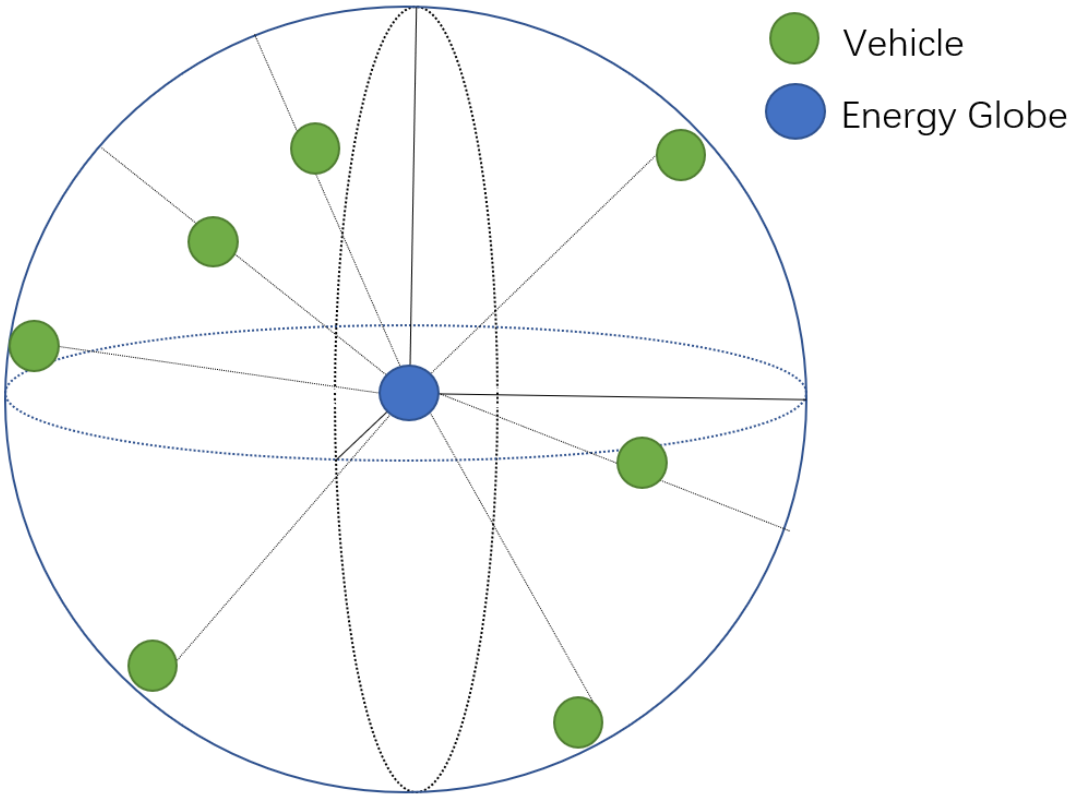
1. Vehicle_Num(Vehicle_No): Number of Vehicles;
2. Position, Velocity: informations of the finding globe
3. Time_Of_Finding: timestamp of finding the energy globe

Vehicle_Num (named Vehicle_No in stage 2/3 code) should be initialized as Vehicle_No for each task.

2.2.4 Form the Sphere

To form the vehicle sphere around energy globe, the first thing is to let all vehicles connect with others. It can be done by let all vehicles go to the origin point (others => 0) at first. And since globes move around the origin point at this stage, it's easy for vehicles to find the globe.

The real problem is how to let vehicles evenly distributed in the space around the globe. To simplify the problem, let all vehicles hold their own moving path that start from spherical surface to the globe. Vehicles can only move on their moving path.



So we can set the destination of vehicles by:

Destination := Position + (Globe.Position - Position) * ((abs(Globe.Position - Position) - Radius) / abs(Globe.Position - Position));

Which means that the vehicle just go to the globe position and stop when the distance between the globe and the vehicle is Radius.

How to make sure all paths have no conflict with others and are evenly distributed? For this attempt, just assume that vehicles are born to be randomly placed around the globe so there's little conflict on their path to the globe, especially when there's only 64 vehicles. And if conflict really occurs, remember that collision avoidance reflexes are always active and prevent vehicles from crashing into each other. Vehicles may slow down, waiting until the vehicle in its way moves away, and then really have a conflict-free move path. Since the

globe is moving all the time, the sphere surface, moving paths and vehicles are moving with the globe, it can be guessed that conflicts between moving paths will be solved automatically through collision avoidance reflexes and vehicles moving all the time.

2.2.5 Find Globe Position

A vehicle can only detect the globe when the globe is within the distance of Energy_Globe_Detection (0.07), but sphere Radius is longer than that because a sphere with such a short radius can not contain many vehicles, and vehicles' communicating distance (named Comms_Range with a value of 0.2) is longer than that.

Thus, vehicles often have to guess where the globe is now. For every loop of vehicle task, update globe position regarding time passing:

Globe.Position := Pre_Message.Position + Pre_Message.Velocity * Real(To_Duration(Clock - Pre_Message.Time_Of_Finding));

Pre_Message.Position is the newest known position of the globe, and Pre_Message.Velocity is the newest known velocity of it. Pre_Message.Time_Of_Finding is the last time known when a vehicle find the globe. Pre_Message is updated by all messages the vehicle task have received.

2.2.6 Find The Number of Vehicles

The number of vehicles is maintained in Num_Of_Vehicles. Num_Of_Vehicles initially store Vehicle_No, which is the unique id of vehicles. When vehicles receive a

Message.Vehicle_No that is bigger than Num_Of_Vehicles, let it replace Num_Of_Vehicles. Thus, a vehicle consider the biggest Vehicle_No it meets as Num_Of_Vehicles for this attempt.

Thus, vehicles can find the real number of vehicles only when the vehicle with the biggest id pass them a message. Since vehicles all run to (0, 0, 0) at first and the message sending range is much bigger than the globe detection range, there's no big problem especially when there's only 64 vehicles.

After find the number of vehicles, an automatic adjusted sphere radius can be calculated.

2.2.7 Automatic Adjusted Sphere Radius

Radius need to be automatically adjusted regarding Num_Of_Vehicles. Since vehicles are mainly evenly distributed in the sphere surface, and sphere surface area can be calculated by:

$$S = 4\pi R^2$$

It can be figured that radius is propotional to Sqrt(Num_Of_Vehicles).

Thus, radius can first be calculated by:

Radius := Radius_Rate * Energy_Globe_Detection * Sqrt(Real(Num_Of_Vehicles) / Default_Vehicle_Num);

Default_Vehicle_Num is a constant variable of value 64.0, which means that when Num_Of_Vehicles = 64, Radius := Radius_Rate * Energy_Globe_Detection.

Radius_Rate is a constant hyper parameter, the ratio of radius to Energy_Globe_Detection when it's Default_Vehicle_Num. It is usually in range of 1 .. 3.

Also, Radius need to be automatically adjusted regarding the current charge of vehicle:

Radius := Radius * Real((Current_Charge / Charge_Init));

Charge_Init is initialized at the beginning of the task. It can be assumed that vehicles are full-charged initially.

But do Radius need to be adjusted when a vehicle disappears? For this model, there is no need. Because if a vehicle can get charged successfully on its moving path before another vehicle disappearing, it can still do it after that. And as for vehicle communication, since vehicles' communicating distance (named Comms_Range with a value of 0.2) is much longer than Energy_Globe_Detection, if vehicles are evenly distributed in the sphere surface, unless most of vehicles have disappeared, survived vehicles would not lose communication.

2.2.8 Pseudocode

The pseudocode for this attempt is provided as followed.

```
select
    Flight_Termination.Stop;
then abort
    Outer_task_loop : loop
        Wait_For_Next_Physics_Update;
        if find_globe then
            update globe and pre and send pre;
```

```
        else
            send pre;
        receive message;
        if bigger_num_from_message then
            update num;
        if newer position then update globe and pre;
        update globe.position;
        update radius;
        calculate and set destination around the globe;
    end loop Outer_task_loop;
```

2.2.9 Problem

This attempt is failed. All vehicles gradually move to the globe's position when losing charge, which makes the space around the globe too crowded. Also, vehicles are not really forming a sphere evenly.

2.3 Second Attempt

2.3.1 Evenly Distribute Vehicles On the Sphere

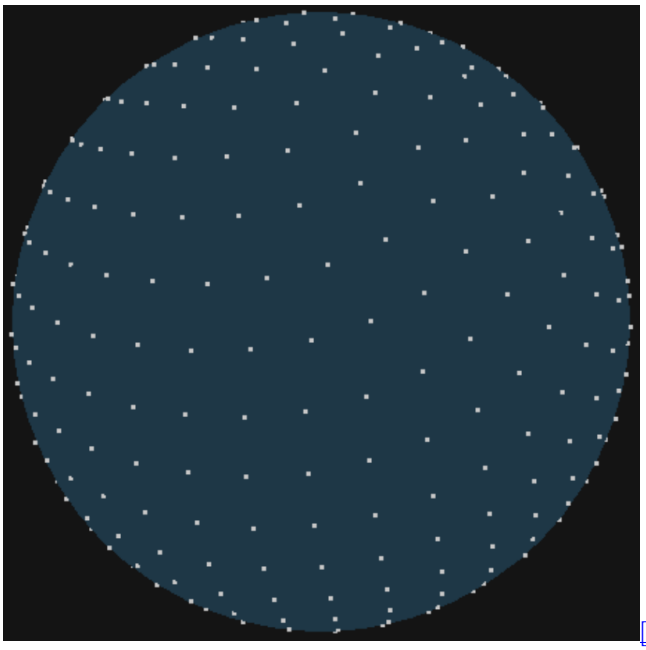
The Fibonacci sphere algorithm is an easy and fast way to evenly distribute n points on a sphere.[2]

When Radius := 1, then the coordinate of point i (x_n, y_n, z_n) is given by:

$$z_n = (2n - 1)/N - 1$$
$$x_n = \sqrt{1 - z_n^2} \cdot \cos(2\pi n\phi)$$
$$y_n = \sqrt{1 - z_n^2} \cdot \sin(2\pi n\phi)$$

When $\phi = (\sqrt{5} - 1)/2 \approx 0.618$, which is known as golden ratio.

Sample is like this:

 [3]

The Ada code is:

```
Z_Vector := (2.0 * Real(Vehicle_No) - 1.0) / Real(Num_Of_Vehicles) - 1.0;

X_Vector := Sqrt(1.0 - (Z_Vector ** 2)) * Cos(2.0 * PI * Real(Vehicle_No) * Phi);

Y_Vector := Sqrt(1.0 - (Z_Vector ** 2)) * Sin(2.0 * PI * Real(Vehicle_No) * Phi);
```

However, Radius is not equal to 1, and Globe.Position is not the origin point. Thus, it follows with this:

```
Destination(x) := Globe.Position(x) + Radius * X_Vector;

Destination(y) := Globe.Position(y) + Radius * Y_Vector;
```

Destination(z) := Globe.Position(z) + Radius * Z_Vector;

The code above successfully evenly distributes vehicles on the sphere.

2.3.2 Problem

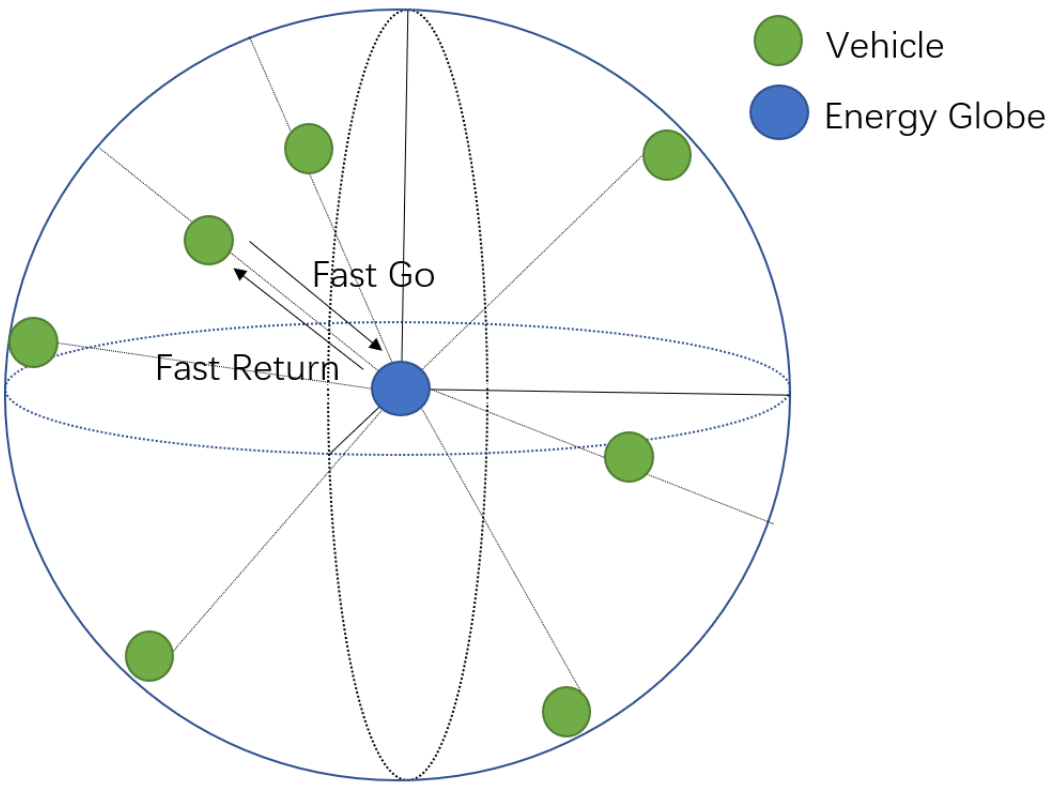
This attempt is failed. The main problem remains, all vehicles gradually move to the globe's position when losing charge, which makes the space around the globe too crowded.

2.4 Third Attempt

2.4.1 Fast Go Fast Return

The problem is that the closer to the globe, the smaller space there is. The radius of sphere that vehicles can get charged is equal to Energy_Globe_Detection, which is too small to contain all the vehicles.

To relieve the crowd around energy globe, vehicles can not gradually and slowly move to the globe and get charged like before. Instead, they need to follow *fast go fast return* rule.



Radius should be calculated like this to relieve the crowd while preventing vehicles rush to get charged losing connection with others:

Radius := Radius_Rate * Energy_Globe_Detection * Sqrt(Real(Num_Of_Vehicles) / Default_Vehicle_Num);

Radius := Radius * Sqrt(Real(Current_Charge / Charge_Init));

As for charging determination, define Estimated_Charge as how much energy left after reaching the globe if the vehicle dash to charge now.

When Estimated_Charge < Alarmed_Charge, the vehicle should go to charge as fast as it could by:

Destination := Globe.Position;

Set_Destination(Destination);

Set_Throttle(Throttle * 2.0);

Throttle * 2.0 is 1.0, the maximum throttle for vehicles.

And as for Estimated_Charge, apparently its calculation can be simplified by:

Estimated_Charge := Current_Charge - Current_Discharge_Per_Sec * Estimated_Time;

Estimated_Time is the required time to reach the globe, which can be calculated by:

$$x = \frac{v_0 + v_t}{2}t = v_0 t + \frac{1}{2}at^2$$

That is:

Estimated_Time := (Sqrt((Real(abs(Velocity))**2) + 2.0 * abs(Acceleration) * Radius) - abs(Velocity)) / abs(Acceleration);

The calculations above are not really precise, because Current_Discharge_Per_Sec and Acceleration are changing all the time, hence precise estimation is not realistic.

2.4.2 Pseudocode

Pseudocode is updated as followed:

```
select
    Flight_Termination.Stop;
then abort
    Outer_task_loop : loop
        Wait_For_Next_Physics_Update;
        if find then update globe and pre and send pre;
        else send pre;
        receive;
```

```
        if bigger then update num;
        if newer position then update globe and pre;
        update globe.position;
        update radius;
        update estimate_time;
        if charge alarmed then go to globe;
        else calculate and set destination around the globe;
    end loop Outer_task_loop;
```

2.4.3 Problem

It is a successful attempt. In small-scale cases, vehicles are evenly distributed on the sphere. But when the number of vehicles comes to 512, tests show that some vehicles can't communicate with vehicle with the biggest id, then they don't know the exact number of vehicles. Therefore, vehicles are not perfectly evenly distributed on the sphere.

2.5 Fourth Attempt

2.5.1 Find The Number of Vehicles Correctly

To solve the problem, just update Pre_Message.Vehicle_Num (Vehicle_No) every time a larger Vehicle_Num is found, so that vehicles pass message with the biggest Id they've seen.

Thus, Num_Of_Vehicles is replaced by Pre_Message.Vehicle_Num.

It seems to be a simple code bug fixing, but the result is unexpected.

2.5.2 Pseudocode

```
select
    Flight_Termination.Stop;
then abort
    Outer_task_loop : loop
        Wait_For_Next_Physics_Update;
        if find then update globe and pre and send pre;
        else send pre;
        receive;
        if bigger then update num and pre;
        if newer position then update globe and pre;
        update globe.position;
        update radius;
        update estimate_time;
        if charge alarmed then go to globe;
        else calculate and set destination around the globe;
    end loop Outer_task_loop;
```

2.5.3 Problem

It is a successful attempt. However, the first 90 seconds' crowding becomes more extreme when the number of vehicles is 512. That's probably because when all vehicles get the real number of vehicles and then the correct Radius, the average distance they need to move in the first 90 seconds is longer than when some of them get a smaller number of vehicles and Radius.

3. Stage 3 Design

3.1 Problem

Stage 3 requires further coordination between vehicles as multiple energy globes are to be considered. Globes can appear and vanish at random, yet there will be a minimum of 2 globes around at all times.[1]

3.2 Globe Validation

By observation, it can be found that energy globes diminish very often. If the past time vehicles find the globe (Pre_Message.Time_Of_Finding) till now is longer than Invalid_Time, then vehicles think that the current globe is invalid.

The initial globe, that initialized as the origin point, is also invalid specially. Thus, set Valid_Globe to recorgnize the initial globe. Valid_Globe := False initially, but it's set true

when a new globe is found or passed by Message received.

3.3 Update The Closer Globe

If a closer and valid globe is passed to the vehicles by Message, then let the vehicle go around this globe.

If the globe stores in Pre_Message is already invalid, then update it whenever get a valid globe through Message.

If a vehicle directly find a globe, then it's a closer and valid globe anyway, so update Pre_Message immediately.

If a vehicle find multiple globes at a time, update for the closest globe. Since the possibility to find more than two globes is really low, it is ignored.

3.4 Other Optimizations

It is frequent to see that vehicles follow a globe moving to a far distance and find the globe diminished. In this case, it's unlikely for those vehicles to find a globe nearby quickly. It can also be observed by calling out axis lines that globes usually appear around the origin point, and model code also proves that. Therefore, when vehicles find their globe invalid now, go back to the origin point to search for another globe.s

Another optimization is that since the number of vehicles around one globe normally is only a third to a half of Vehicle_Num, reduce Radius_Rate from 2.4 in stage 2 to 1.5

accordingly. This value is not being adjust for many times, because at this stage, the time for testing a reliable result is extremely long.

The best solution is to automatically adjust Radius regarding the number of vehicles around the globe, but it's really difficult to implement, because globes move and diminish in a fast speed, and there's often two globes getting close with each other. However, since the communication distance for vehicles is long enough, vehicles won't lost connection for not evenly distributed enough, thus the demand to automatically adjust Radius is small.

3.5 Pseudocode

```
select
    Flight_Termination.Stop;
then abort
    Outer_task_loop : loop
        Wait_For_Next_Physics_Update;
        if find
            update globe, pre for the closest globe and send pre;
        else
            send pre;
        receive message;
        if id_bigger
            update num and pre;
```

```
        if globe_closer and message_is_valid
            update globe and pre;
        elsif message_is_valid and pre_is_not_valid
            update globe and pre;
        update globe.position;
        update radius;
        update estimate_time
        if charge alarmed
            go to globe;
        else
            calculate and set destination around the globe;

    end loop Outer_task_loop;
```

4. Stage 4 Design

4.1 Problem

The swarm shrinks itself to a specific size. This stage will require a fully distributed method to share information and agree on action.[1]

4.2 First Attempt

4.2.1 Basic Idea

The easiest solution is definitely let some vehicles destroy themselves according to its id. Another way is to randomly select some ids of vehicles to destroy themselves. However, these are not real distributed method.

The most exciting and *distributed* idea is to ensure that only those vehicles that choose to destroy themselves know that they are in the destruction plan. It can also fully protect their privacy.

4.2.2 Consensus Decision Making

Assume that there's no malicious nodes or failed nodes in the vehicle network. Make the decision by consensus, which means that the destruction plan begins only when every vehicle agree with it, and all volunteers in the destruction plan destroy themselves together at the same time, that is, the plan execution needs to satisfy consistency.

This can be implemented by setting a Tolerant_Time. Only when the volunteer number in the destruction plan is enough and the last updating time of the plan till now is longer than Tolerant_Time will the volunteers start to destroy themselves.

4.2.3 Message Structure

For this attempt, Message includes two more variables than in stage 3:

Number_Of_Volunteers that stores the number of volunteers in the destruction plan.

When_Decide, a timestamp that stores the time when the last decision made for the destruction plan.

4.2.4 Volunteers Destroy Themselves

Volunteers will destroy themselves by going to (others => INFINITY). INFINITY is a constant big number. This way, volunteers would not get charged anymore and disappear very soon in theory.

4.2.5 Update The Plan

All vehicles try to be the volunteer initially. If the destruction plan from Message is not the same with the vehicle's, take the one with bigger number of volunteers. If the volunteer number is the same, take the one with earlier timestamp of last updating.

Store another two information: if the vehicle want to destroy itself and the time of making this decision.

If vehicle doesn't want to destroy itself but will be out of charge soon, add itself to the destruction plan. Else if vehicle doesn't want to destroy itself but the volunteer number is lower than the required number, also add itself to the destruction plan.

If vehicle want to destroy itself but find the time of making this decision is later than When_Decide of the newest destruction plan, it has to cancel the determination to destroy itself.

4.2.6 Pseudocode

```
select
    Flight_Termination.Stop;
then abort
    Outer_task_loop : loop
        Wait_For_Next_Physics_Update;
        if find
            update globe, pre for the closest globe and send pre;
        else
            send pre;
        receive message;
        if receive_destruction_plan is longer or (the_same_size and order)
            update pre;
        if Will_Destruct
            if When_Decide > pre.When_Decide
                undo decide_district;
        else
            if out_of_charge
                update Now_Destruct;decide to charge; update pre;
            elsif pre.Number_Of_Volunteers < required
```

```
                    decide to charge; update pre;
            if pre.Number_Of_Volunteers >= required and Will_Destruct and Clock -
pre.When_Decide > Tolerant_Time
                update Now_Destruct;
            if id_bigger
                update num and pre;
            if globe_closer and is_valid
                update globe and pre;
            elsif newer position and is_valid and pre_is_not_valid
                update globe and pre;
            update globe.position;
            update radius;
            update estimate_time
            if charge alarmed
                go to globe;
            else
                calculate and set destination around the globe;
            if Now_Destruct
                go to infinity to get destroyed;

        end loop Outer_task_loop;
```

4.2.7 Problem

This is a failed attempt. There's either no effective way to judge if the destruction plan is the newest, or no way to correctly judge if the vehicle is in the destruction plan. Which side is true depends on the way to update and judge from the destruction plan information. Thus, a volunteer list in the Message that stores the destruction list of vehicle ids is really required. However, this means that privacy of volunteers can not be protected, because each node in the vehicle network can read the volunteer list.

4.3 Second Attempt

4.3.1 Message Structure

New Message structure adds Vehicles, which is an array of vehicle ids of the volunteers.

4.3.2 Update The Plan

Now vehicles can see if itself is in the destruction plan correctly.

If vehicle want to destroy itself but find itself not in the newest volunteer list, cancel the determination to destroy itself.

If vehicle want to destroy itself, add its id to the end of the volunteer list.

4.3.3 Other Optimizations

There's no more need to let out-of-charge vehicles to try to participate the destruction plan because if the plan is full, they'll want to join it; else, the volunteer list array may be out-of-bound.

There's no demand to store the extra local When_Decide, because there's now a new way for vehicles to see if itself is in the destruction plan.

Considering the average number of the vehicles is shortened again, reduce the number of Radius_Rate from 1.5 in stage 3 to 1.3. This value is not being adjust for many times, because at this stage, the time for testing a reliable result is extremely long, and it's not really significant because the main aim in stage 4 is to control the vehicle number in the first two minutes.

4.3.4 Pesudocode

```
select
    Flight_Termination.Stop;
then abort
    Outer_task_loop : loop
        Wait_For_Next_Physics_Update;
        if find
            update globe, pre for the closest globe and send pre;
        else
```

```
        send pre;
    receive message;
    if receive_destruction_plan is longer or (the_same_size and order)
        update pre;
    if Will_Destruct
        if not in the destruction list
            undo decide_district;
    else
        if pre.Number_Of_Volunteers < required
            decide to charge; update pre;
    if pre.Number_Of_Volunteers >= required and Will_Destruct and Clock -
pre.When_Decide > Tolerant_Time
        update Now_Destruct;
    if id_bigger
        update num and pre;
    if globe_closer and is_valid
        update globe and pre;
    elsif newer position and is_valid and pre_is_not_valid
        update globe and pre;
    update globe.position;
    if destruction started
        Num_Of_Vehicles := Target_No_of_Elements and rollback the value
```

```
eventually;
        update radius;
        update estimate_time
        if Now_Destruct
            go to infinity to get destroyed;
        else
            if charge alarmed
                go to globe;
            else
                calculate and set destination around the globe;

    end loop Outer_task_loop;
```

4.3.5 Problem

This is a failed attempt. The main problem is that let vehicles go to (others => INFINITY) to destroy themselves seems to cause some bugs that there's little energy globe appears around the origin point after the destruction plan begins. Vehicles can not find energy globe around the origin point.

4.4 Third Attempt

4.4.1 Volunteers Destroy Themselves

A new way for volunteers to destroy themselves is to set no destinations, so that they're taken over by the default swarming behavior.

Another way to let volunteers destroy themselves is to stop them getting charged, but the implementation is a little more complex.

4.4.2 Pseudocode

```
select
    Flight_Termination.Stop;
then abort
    Outer_task_loop : loop
        Wait_For_Next_Physics_Update;
        if find
            update globe, pre for the closest globe and send pre;
        else
            send pre;
        receive message;
        if receive_destruction_plan is longer or (the_same_size and order)
            update pre;
```

if Will_Destruct

if not in the destruction list

undo decide_district;

else

if pre.Number_Of_Volunteers < required

decide to charge; update pre;

if pre.Number_Of_Volunteers >= required and Will_Destruct and Clock - pre.When_Decide > Tolerant_Time

update Now_Destruct;

if id_bigger

update num and pre;

if globe_closer and is_valid

update globe and pre;

elsif newer position and is_valid and pre_is_not_valid

update globe and pre;

update globe.position;

if destruction started

Num_Of_Vehicles := Target_No_of_Elements and rollback the value eventually;

update radius;

update estimate_time

if Now_Destruct

```
            let the default swarming behaviour take over to get destroyed;
        else
            if charge alarmed
                go to globe;
            else
                calculate and set destination around the globe;

    end loop Outer_task_loop;
```

4.4.3 Other Optimizations

The code quality is improved to facilitate maintenance. For this attempt, initialization is done together with the definition part to the greatest extent. Other initialization is done together after the unique id of the vehicle is given.

All codes written for stage 4 are separated together from other codes, which would greatly enable further improvement. Initialization of stage 3 variables are also separated.

All variables are defined with detailed annotations above. All functional codes have their detailed annotations as well to achieve maintainability eventually.

4.4.4 Conclusion

This is a successful attempt that satisfied the idea of consensus decision making.

5. Tests

The conclusions for each attempts above are given by the following tests.

5.1 Stage 2

5.1.1 First Attempt

This is a failed attempt. It can be observed that nearly half of the vehicles could not survive for 2 minutes when the total number of vehicles is 64.

There's almost no vehicle disappearing in the first minute, but then most of the vehicles need to get charged, and the space around the energy globe is too crowded to fulfill their wills.

It can also be observed that vehicles all go to the origin point at first, then they are still being controlled till the end, and they are automatically evenly distributed in the sphere approximately, moving closer while losing charge, which meets the assumptions. This is the successful and expected part for the attempt.

Testing codes for vehicle_message_type.ads (which is used for stage 2 & 3) is:

-- Suggestions for packages which might be useful:

with Ada.Real_Time; *use Ada.Real_Time;*

-- with Swarm_Size; *use Swarm_Size;*

```
with Vectors_3D;                    use Vectors_3D;

package Vehicle_Message_Type is

    -- Replace this record definition by what your vehicles need to communicate.

    type Inter_Vehicle_Messages is

        record
            -- Vehicle Number
            Vehicle_No : Positive;
            -- informations of finding single globe
            Position : Vector_3D;
            Velocity : Vector_3D;
            -- time of finding the energy globe
            Time_Of_Finding : Time;
        end record;

end Vehicle_Message_Type;
```

Testing codes for vehicle_task_type.adb is:

```
-- Suggestions for packages which might be useful:

with Ada.Real_Time;                 use Ada.Real_Time;
--  with Ada.Text_IO;                   use Ada.Text_IO;
with Exceptions;                    use Exceptions;
with Real_Type;                     use Real_Type;
--  with Generic_Sliding_Statistics;
--  with Rotations;                     use Rotations;
with Vectors_3D;                    use Vectors_3D;
with Vehicle_Interface;             use Vehicle_Interface;
with Vehicle_Message_Type;          use Vehicle_Message_Type;
--  with Swarm_Structures;              use Swarm_Structures;
with Swarm_Structures_Base;         use Swarm_Structures_Base;
with Swarm_Configuration;           use Swarm_Configuration;
with Ada.Numerics.Long_Elementary_Functions;
use Ada.Numerics.Long_Elementary_Functions;

package body Vehicle_Task_Type is

   task body Vehicle_Task is

      Vehicle_No : Positive;--pragma Unreferenced (Vehicle_No);
```

```
   -- You will want to take the pragma out, once you use the "Vehicle_No"
   -- initial destination
   Destination : Vector_3D := (others => 0.0);
   -- information of finding globes
   Globe : Energy_Globe;
   Message : Inter_Vehicle_Messages;
   -- Pre_Message actually stores the newest information of the globe
   Pre_Message : Inter_Vehicle_Messages;
   Num_Of_Vehicles: Positive;
   Radius : Real;
   Charge_Init : Vehicle_Charges;
   Throttle : constant Throttle_T := 0.5;

begin

   -- You need to react to this call and provide your task_id.
   -- You can e.g. employ the assigned vehicle number (Vehicle_No)
   -- in communications with other vehicles.
   accept Identify (Set_Vehicle_No : Positive; Local_Task_Id : out Task_Id) do
      Vehicle_No    := Set_Vehicle_No;
      Local_Task_Id := Current_Task;
   end Identify;
```

```
-- Replace the rest of this task with your own code.
-- Maybe synchronizing on an external event clock like "Wait_For_Next_Physics_Update",
-- yet you can synchronize on e.g. the real-time clock as well.

-- Without control this vehicle will go for its natural swarming instinct.
--    test
--    if(Vehicle_No = 1) then
--    Set_Destination(Destination);
--    Set_Throttle(2.0);
--    end if;

--    Initialize the variables, so that go to (0, 0, 0) at first, and remain the same if not
receiving a new globe place.
Num_Of_Vehicles := Vehicle_No;
Globe.Position := (others => 0.0);
Globe.Velocity := (others => 0.0);
Pre_Message.Vehicle_No := Vehicle_No;
Pre_Message.Position := Globe.Position;
Pre_Message.Velocity := Globe.Velocity;
Pre_Message.Time_Of_Finding := Clock;
Set_Destination(Destination);
```

```
Set_Throttle(Throttle);
Charge_Init := Current_Charge;

select

   Flight_Termination.Stop;

then abort

   Outer_task_loop : loop

      Wait_For_Next_Physics_Update;

         -- Your vehicle should respond to the world here: sense, listen, talk, act?
      if (Energy_Globes_Around'Length /= 0) then
         Globe := Energy_Globes_Around(1);
         Pre_Message.Position := Globe.Position;
         Pre_Message.Velocity := Globe.Velocity;
         Pre_Message.Time_Of_Finding := Clock;
         Send(Message => Pre_Message);
         --    Put_Line("yes" & Message.Position(x)'Image);
      else
```

```
            Send(Message => Pre_Message);
        end if;

        Receive(Message => Message);
        if (Message.Vehicle_No > Num_Of_Vehicles) then
            Num_Of_Vehicles := Message.Vehicle_No;
        end if;
        if (Message.Time_Of_Finding > Pre_Message.Time_Of_Finding) then
            Globe.Position := Message.Position;
            Globe.Velocity := Message.Velocity;
            Pre_Message.Position := Globe.Position;
            Pre_Message.Velocity := Globe.Velocity;
            Pre_Message.Time_Of_Finding := Message.Time_Of_Finding;
        end if;
        -- update globe position regarding time passing
        Globe.Position := Pre_Message.Position + Pre_Message.Velocity *
Real(To_Duration(Clock - Pre_Message.Time_Of_Finding));
        -- update radius regarding vehicle number updating
        Radius := 5.0 * Energy_Globe_Detection * Sqrt(Real(Num_Of_Vehicles) / 64.0);
        -- update radius regarding charge changing
        Radius := Radius * Real((Current_Charge / Charge_Init));
        -- update destination regarding the change of radius and globe
```

```
                Destination := Position +
                   (Globe.Position - Position) * ((abs(Globe.Position - Position) - Radius) /
abs(Globe.Position - Position));
                --  test
                --  Put_Line("yes" & Energy_Globe_Detection'Image);
                Set_Destination(Destination);
                Set_Throttle(Throttle);
             end loop Outer_task_loop;

          end select;

       exception
          when E : others => Show_Exception (E);

       end Vehicle_Task;

end Vehicle_Task_Type;
```

5.1.2 Second Attempt

This is a failed attempt, the crowding issue remained as what the first attempt's like, still too crowded for space close to the globe. The surviving time of 64 vehicles is more than 60

seconds but it's less than 90 seconds for 63 vehicles to survive. And in 3 minutes, the number of surviving vehicles is less than a half when the total number of vehicles is 64.

It can also be observed that vehicles do form a perfect evenly-distributed sphere around the energy globe, and move with the globe, which is the successful part of the attempt.

Testing codes for vehicle_task_type.adb is:

```
-- Suggestions for packages which might be useful:

with Ada.Real_Time;                    use Ada.Real_Time;
--  with Ada.Text_IO;                  use Ada.Text_IO;
with Exceptions;                       use Exceptions;
with Real_Type;                        use Real_Type;
--  with Generic_Sliding_Statistics;
--  with Rotations;                    use Rotations;
with Vectors_3D;                       use Vectors_3D;
with Vehicle_Interface;                use Vehicle_Interface;
with Vehicle_Message_Type;             use Vehicle_Message_Type;
--  with Swarm_Structures;             use Swarm_Structures;
with Swarm_Structures_Base;            use Swarm_Structures_Base;
with Swarm_Configuration;              use Swarm_Configuration;
with Ada.Numerics.Long_Elementary_Functions;
use Ada.Numerics.Long_Elementary_Functions;
```

```
package body Vehicle_Task_Type is

   task body Vehicle_Task is

      Vehicle_No : Positive;--pragma Unreferenced (Vehicle_No);
      -- You will want to take the pragma out, once you use the "Vehicle_No"
      -- initial destination
      Destination : Vector_3D := (others => 0.0);
      -- information of finding globes
      --    Globes_Found : Natural := 1;
      Globe : Energy_Globe;
      Message : Inter_Vehicle_Messages;
      -- Pre_Message actually stores the newest information of the globe
      Pre_Message : Inter_Vehicle_Messages;
      Num_Of_Vehicles: Positive;
      Radius : Real;
      X_Coordinate, Y_Coordinate, Z_Coordinate : Real;
      Charge_Init : Vehicle_Charges;
      Estimated_Time : Real;
      Throttle : constant Throttle_T := 0.5;
      PI : constant Real := 3.1415926536;
```

```
   Phi : constant Real := (Sqrt(5.0) - 1.0) / 2.0;


begin

   -- You need to react to this call and provide your task_id.
   -- You can e.g. employ the assigned vehicle number (Vehicle_No)
   -- in communications with other vehicles.
   accept Identify (Set_Vehicle_No : Positive; Local_Task_Id : out Task_Id) do
      Vehicle_No       := Set_Vehicle_No;
      Local_Task_Id    := Current_Task;
   end Identify;

   -- Replace the rest of this task with your own code.
   -- Maybe synchronizing on an external event clock like "Wait_For_Next_Physics_Update",
   -- yet you can synchronize on e.g. the real-time clock as well.

   -- Without control this vehicle will go for its natural swarming instinct.
   --    test
   --    if(Vehicle_No = 1) then
   --    Set_Destination(Destination);
   --    Set_Throttle(2.0);
   --    end if;
```

```
-- Initialize the variables, so that go to (0, 0, 0) at first, and remain the same if not
receiving a new globe place.
        Num_Of_Vehicles := Vehicle_No;
        Globe.Position := (others => 0.0);
        Globe.Velocity := (others => 0.0);
        Pre_Message.Vehicle_No := Vehicle_No;
        Pre_Message.Position := Globe.Position;
        Pre_Message.Velocity := Globe.Velocity;
        Pre_Message.Time_Of_Finding := Clock;
        Set_Destination(Destination);
        Set_Throttle(Throttle);
        Charge_Init := Current_Charge;

        select

            Flight_Termination.Stop;

        then abort

            Outer_task_loop : loop
```

```
Wait_For_Next_Physics_Update;


    -- Your vehicle should respond to the world here: sense, listen, talk, act?
if (Energy_Globes_Around'Length /= 0)
then
    Globe := Energy_Globes_Around(1);
    Pre_Message.Position := Globe.Position;
    Pre_Message.Velocity := Globe.Velocity;
    Pre_Message.Time_Of_Finding := Clock;
    Send(Message => Pre_Message);
    --   Put_Line("yes" & Message.Position(x)'Image);
else
    Send(Message => Pre_Message);
end if;


Receive(Message => Message);
if (Message.Vehicle_No > Num_Of_Vehicles)
then
    Num_Of_Vehicles := Message.Vehicle_No;
end if;
if (Message.Time_Of_Finding > Pre_Message.Time_Of_Finding)
then
```

```
            Globe.Position := Message.Position;
            Globe.Velocity := Message.Velocity;
            Pre_Message.Position := Globe.Position;
            Pre_Message.Velocity := Globe.Velocity;
            Pre_Message.Time_Of_Finding := Message.Time_Of_Finding;
         end if;
         -- update globe position regarding time passing
         Globe.Position := Pre_Message.Position + Pre_Message.Velocity *
Real(To_Duration(Clock - Pre_Message.Time_Of_Finding));
         -- update radius regarding vehicle number updating
         Radius := 2.4 * Energy_Globe_Detection * Sqrt(Real(Num_Of_Vehicles) / 64.0);
         -- update radius regarding charge changing
         Radius := Radius * Sqrt(Real(Current_Charge / Charge_Init));
         Estimated_Time := (Sqrt((Real(abs(Velocity))**2) + 2.0 * abs(Acceleration) *
Radius) - abs(Velocity)) / abs(Acceleration);
         if (Real(Current_Charge) - Estimated_Time * Current_Discharge_Per_Sec <
Real(0.1 * Charge_Init))
         then
            Destination := Globe.Position;
            Set_Destination(Destination);
            Set_Throttle(Throttle * 2.0);
         else
```

```
                    Z_Coordinate := (2.0 * Real(Vehicle_No) - 1.0) / Real(Num_Of_Vehicles) - 1.0;
                    X_Coordinate := Sqrt(1.0 - (Z_Coordinate**2)) * Cos(2.0 * PI *
Real(Vehicle_No) * Phi);
                    Y_Coordinate := Sqrt(1.0 - (Z_Coordinate**2)) * Sin(2.0 * PI *
Real(Vehicle_No) * Phi);
                    Destination(x) := Globe.Position(x) + Radius * X_Coordinate;
                    Destination(y) := Globe.Position(y) + Radius * Y_Coordinate;
                    Destination(z) := Globe.Position(z) + Radius * Z_Coordinate;
                    Set_Destination(Destination);
                Set_Throttle(Throttle);
                end if;
            end loop Outer_task_loop;

        end select;

    exception
        when E : others => Show_Exception (E);

    end Vehicle_Task;

end Vehicle_Task_Type;
```

5.1.3 Third Attempt

This is the first successful attempt for the assignment. Many observations for the model performance are done here.

Now all 64 vehicles can survive forever. Then tests for 128, 256, 512 initial vehicles to survive for 10 minutes are also done.

In addition, special test for 31 initial vehicles is done to see how the model will perform when the initial number of vehicles is a very small, singular and prime number. Those are special cases for the assignment, and robustness of the spherical model need to be tested.

Testing codes for vehicle_task_type.adb is:

```
-- Suggestions for packages which might be useful:

with Ada.Real_Time;                    use Ada.Real_Time;
--  with Ada.Text_IO;                      use Ada.Text_IO;
with Exceptions;                       use Exceptions;
with Real_Type;                        use Real_Type;
--  with Generic_Sliding_Statistics;
--  with Rotations;                        use Rotations;
with Vectors_3D;                       use Vectors_3D;
with Vehicle_Interface;                use Vehicle_Interface;
with Vehicle_Message_Type;             use Vehicle_Message_Type;
--  with Swarm_Structures;                 use Swarm_Structures;
```

```
with Swarm_Structures_Base;          use Swarm_Structures_Base;
with Swarm_Configuration;            use Swarm_Configuration;
with Ada.Numerics.Long_Elementary_Functions;
use Ada.Numerics.Long_Elementary_Functions;

package body Vehicle_Task_Type is

   task body Vehicle_Task is

      Vehicle_No : Positive;--pragma Unreferenced (Vehicle_No);
      -- You will want to take the pragma out, once you use the "Vehicle_No"
      -- initial destination
      Destination : Vector_3D := (others => 0.0);
      -- information of finding globes
      --    Globes_Found : Natural := 1;
      Globe : Energy_Globe;
      Message : Inter_Vehicle_Messages;
      -- Pre_Message actually stores the newest information of the globe
      Pre_Message : Inter_Vehicle_Messages;
      Num_Of_Vehicles: Positive;
      Radius : Real;
      X_Coordinate, Y_Coordinate, Z_Coordinate : Real;
```

```
   Charge_Init : Vehicle_Charges;
   Estimated_Time : Real;
   Throttle : constant Throttle_T := 0.5;
   PI : constant Real := 3.1415926536;
   Phi : constant Real := (Sqrt(5.0) - 1.0) / 2.0;

begin

   -- You need to react to this call and provide your task_id.
   -- You can e.g. employ the assigned vehicle number (Vehicle_No)
   -- in communications with other vehicles.
   accept Identify (Set_Vehicle_No : Positive; Local_Task_Id : out Task_Id) do
      Vehicle_No    := Set_Vehicle_No;
      Local_Task_Id := Current_Task;
   end Identify;

   -- Replace the rest of this task with your own code.
   -- Maybe synchronizing on an external event clock like "Wait_For_Next_Physics_Update",
   -- yet you can synchronize on e.g. the real-time clock as well.

   -- Without control this vehicle will go for its natural swarming instinct.
   --  test
```

```
-- if(Vehicle_No = 1) then
-- Set_Destination(Destination);
-- Set_Throttle(2.0);
-- end if;

-- Initialize the variables, so that go to (0, 0, 0) at first, and remain the same if not receiving a new globe place.
Num_Of_Vehicles := Vehicle_No;
Globe.Position := (others => 0.0);
Globe.Velocity := (others => 0.0);
Pre_Message.Vehicle_No := Vehicle_No;
Pre_Message.Position := Globe.Position;
Pre_Message.Velocity := Globe.Velocity;
Pre_Message.Time_Of_Finding := Clock;
Set_Destination(Destination);
Set_Throttle(Throttle);
Charge_Init := Current_Charge;

select

   Flight_Termination.Stop;
```

```
then abort

   Outer_task_loop : loop

      Wait_For_Next_Physics_Update;

         -- Your vehicle should respond to the world here: sense, listen, talk, act?
      if (Energy_Globes_Around'Length /= 0)
      then
         Globe := Energy_Globes_Around(1);
         Pre_Message.Position := Globe.Position;
         Pre_Message.Velocity := Globe.Velocity;
         Pre_Message.Time_Of_Finding := Clock;
         Send(Message => Pre_Message);
         --    Put_Line("yes" & Message.Position(x)'Image);
      else
         Send(Message => Pre_Message);
      end if;

      Receive(Message => Message);
      if (Message.Vehicle_No > Num_Of_Vehicles)
      then
```

```
            Num_Of_Vehicles := Message.Vehicle_No;
        end if;
        if (Message.Time_Of_Finding > Pre_Message.Time_Of_Finding)
        then
            Globe.Position := Message.Position;
            Globe.Velocity := Message.Velocity;
            Pre_Message.Position := Globe.Position;
            Pre_Message.Velocity := Globe.Velocity;
            Pre_Message.Time_Of_Finding := Message.Time_Of_Finding;
        end if;
        -- update globe position regarding time passing
        Globe.Position := Pre_Message.Position + Pre_Message.Velocity *
Real(To_Duration(Clock - Pre_Message.Time_Of_Finding));
        -- update radius regarding vehicle number updating
        Radius := 2.4 * Energy_Globe_Detection * Sqrt(Real(Num_Of_Vehicles) / 64.0);
        -- update radius regarding charge changing
        Radius := Radius * Sqrt(Real(Current_Charge / Charge_Init));
        Estimated_Time := (Sqrt((Real(abs(Velocity))**2) + 2.0 * abs(Acceleration) *
Radius) - abs(Velocity)) / abs(Acceleration);
        if (Real(Current_Charge) - Estimated_Time * Current_Discharge_Per_Sec <
Real(0.1 * Charge_Init))
        then
```

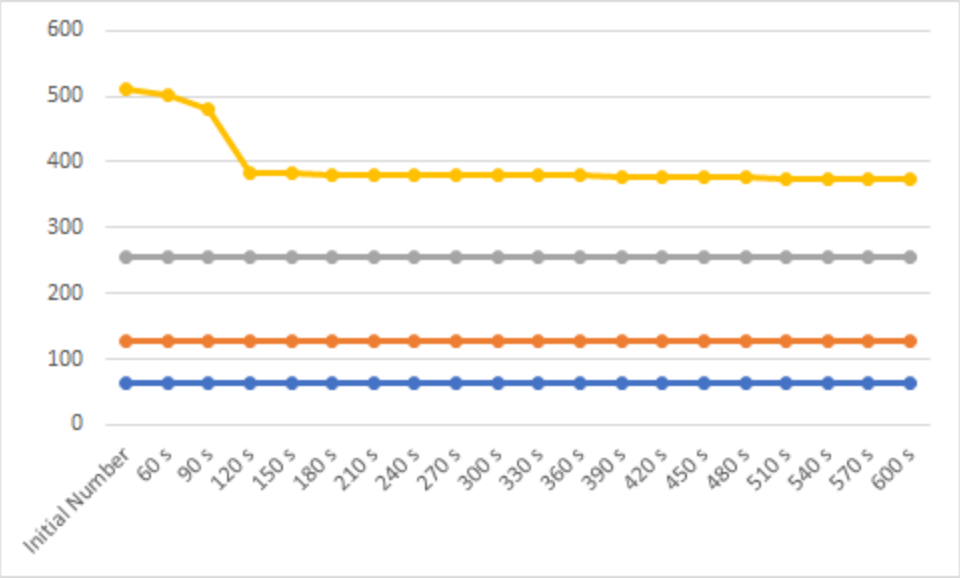
```
                Destination := Globe.Position;
                Set_Destination(Destination);
                Set_Throttle(Throttle * 2.0);
            else
                Z_Coordinate := (2.0 * Real(Vehicle_No) - 1.0) / Real(Num_Of_Vehicles) - 1.0;
                X_Coordinate := Sqrt(1.0 - (Z_Coordinate**2)) * Cos(2.0 * PI *
Real(Vehicle_No) * Phi);
                Y_Coordinate := Sqrt(1.0 - (Z_Coordinate**2)) * Sin(2.0 * PI *
Real(Vehicle_No) * Phi);
                Destination(x) := Globe.Position(x) + Radius * X_Coordinate;
                Destination(y) := Globe.Position(y) + Radius * Y_Coordinate;
                Destination(z) := Globe.Position(z) + Radius * Z_Coordinate;
                Set_Destination(Destination);
            Set_Throttle(Throttle);
            end if;
        end loop Outer_task_loop;

    end select;

exception
    when E : others => Show_Exception (E);
```

end Vehicle_Task;

end Vehicle_Task_Type;

Average testing results for testing twice:

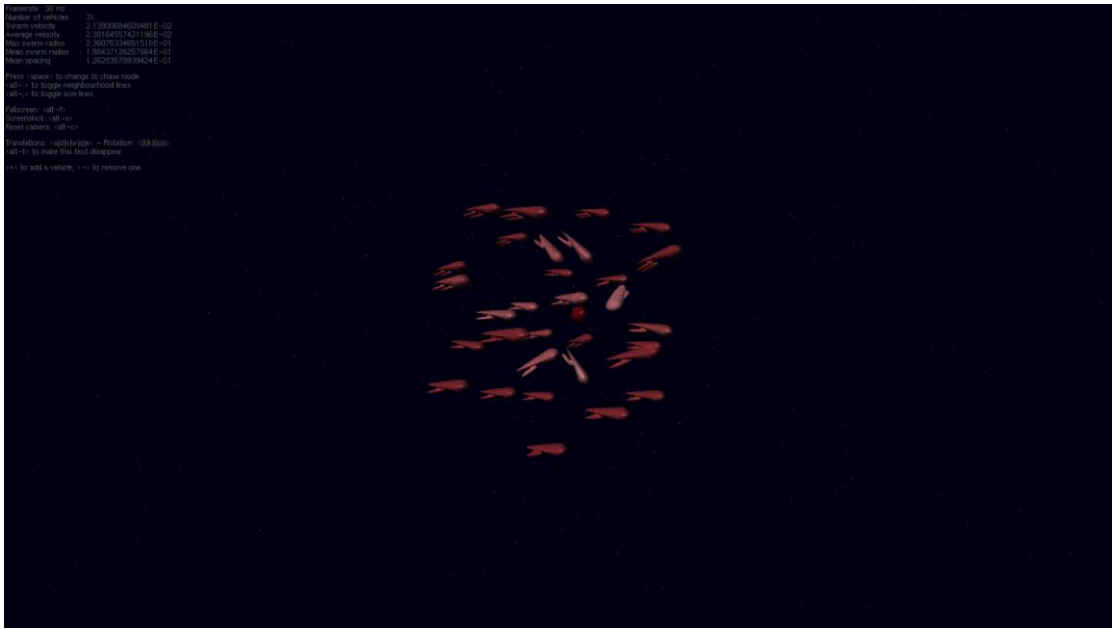
Initial Number	Average Frame Rate	After 60 s	90 s	120 s	10 min	Survival Rate For 10 min
31	30 Hz	31	31	31	31	100 %
64	30 Hz	64	64	64	64	100 %
128	30 Hz	128	128	128	128	100 %
256	16 - 17 Hz	256	256	256	256	100 %
512	9 - 10 Hz	501.5	475.5	374	365.5	71.39 %

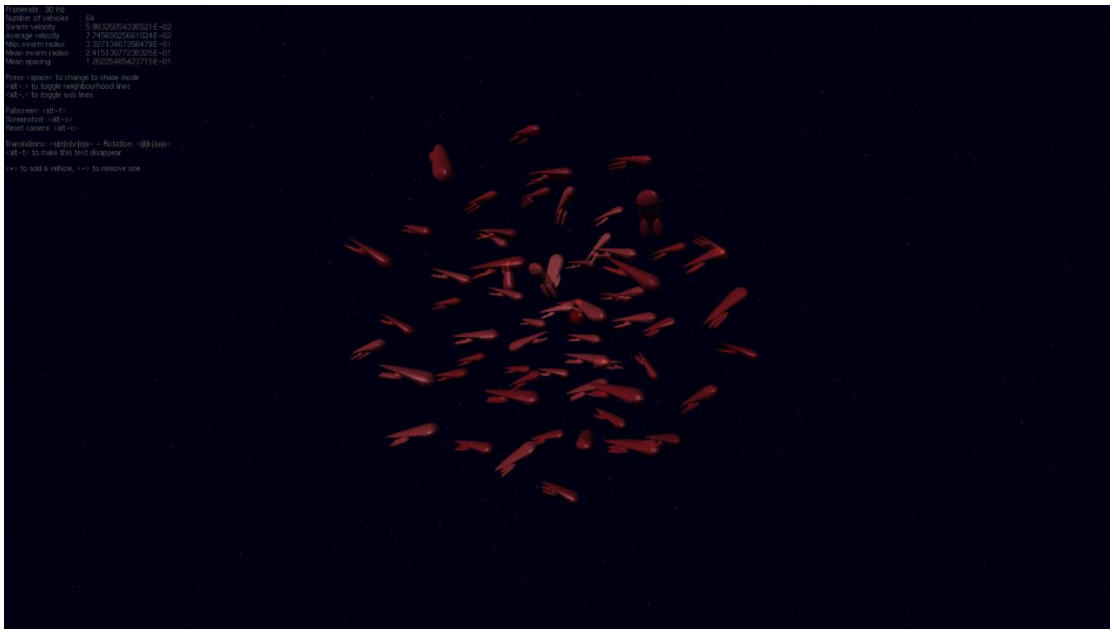


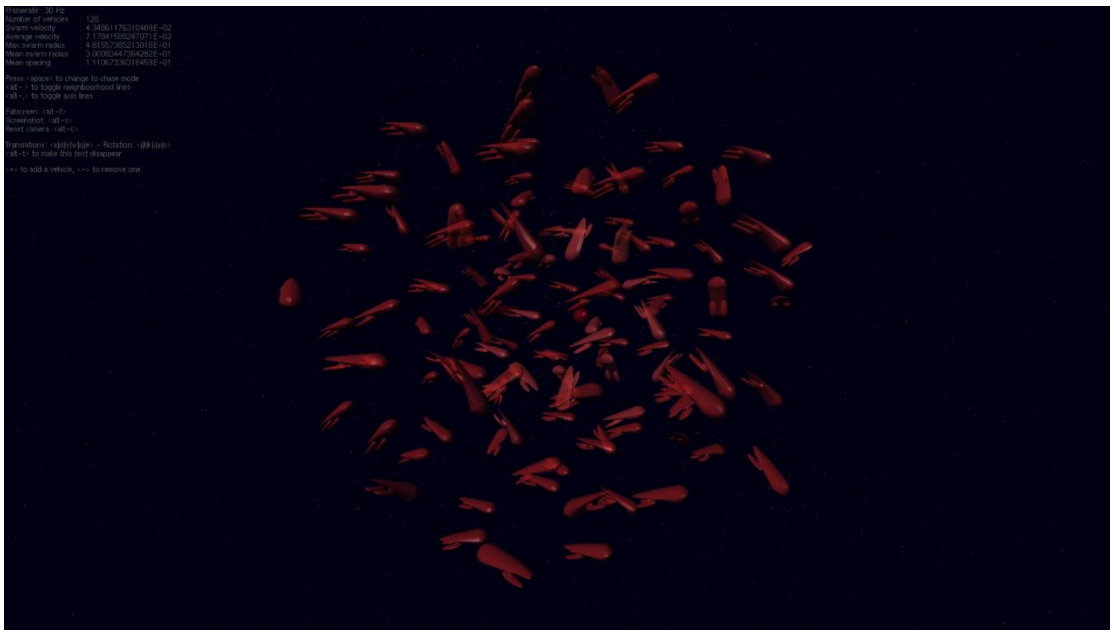
The only problem is that when the vehicle number comes to 512, about a fourth of them can't get their first charge during the busiest first-charging-time. And after 2 minutes, there're still about 1 vehicle each minute that run out of their energy, but the number of vehicles can be maintained for bigger than 360 for longer than 10 minutes. As discussed in the discussion part, that's because some vehicles can't communicate with vehicle with the biggest id, then they don't know the exact number of vehicles. Therefore, vehicles are not

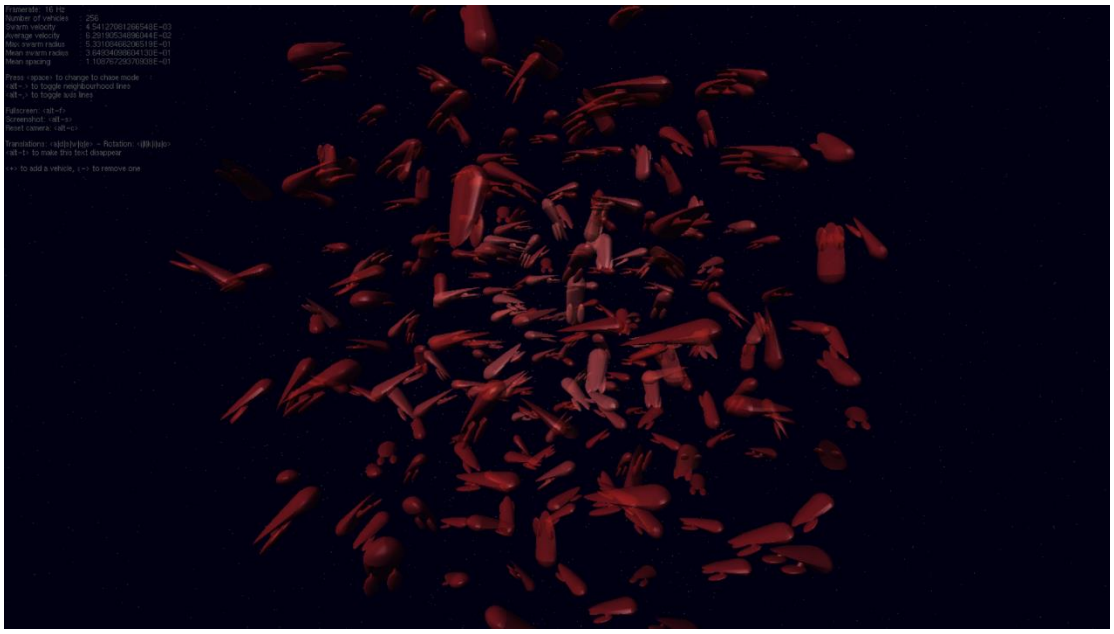
perfectly evenly distributed on the sphere.

It can also be observed that then the vehicle number is 31, 64, 128 or 256, vehicles quickly form a perfect sphere of vehicles and move with the globe:









5.1.4 Fourth Attempt

This is a successful attempt.

Testing codes for vehicle_task_type.adb is:

-- Suggestions for packages which might be useful:

```
with Ada.Real_Time;                    use Ada.Real_Time;
--  with Ada.Text_IO;                      use Ada.Text_IO;
with Exceptions;                       use Exceptions;
with Real_Type;                        use Real_Type;
--  with Generic_Sliding_Statistics;
--  with Rotations;                        use Rotations;
with Vectors_3D;                       use Vectors_3D;
with Vehicle_Interface;                use Vehicle_Interface;
with Vehicle_Message_Type;             use Vehicle_Message_Type;
--  with Swarm_Structures;                 use Swarm_Structures;
with Swarm_Structures_Base;            use Swarm_Structures_Base;
with Swarm_Configuration;              use Swarm_Configuration;
with Ada.Numerics.Long_Elementary_Functions;
use Ada.Numerics.Long_Elementary_Functions;

package body Vehicle_Task_Type is

   task body Vehicle_Task is

      Vehicle_No : Positive;--pragma Unreferenced (Vehicle_No);
      -- You will want to take the pragma out, once you use the "Vehicle_No"
```

```
   -- initial destination
   Destination : Vector_3D := (others => 0.0);
   -- information of finding globes
   --    Globes_Found : Natural := 1;
   Globe : Energy_Globe;
   Message : Inter_Vehicle_Messages;
   -- Pre_Message actually stores the newest information of the globe
   Pre_Message : Inter_Vehicle_Messages;
   Num_Of_Vehicles: Positive;
   Radius : Real;
   X_Coordinate, Y_Coordinate, Z_Coordinate : Real;
   Charge_Init : Vehicle_Charges;
   Estimated_Time : Real;
   Throttle : constant Throttle_T := 0.5;
   PI : constant Real := 3.1415926536;
   Phi : constant Real := (Sqrt(5.0) - 1.0) / 2.0;

begin

   -- You need to react to this call and provide your task_id.
   -- You can e.g. employ the assigned vehicle number (Vehicle_No)
```

```
-- in communications with other vehicles.
accept Identify (Set_Vehicle_No : Positive; Local_Task_Id : out Task_Id) do
    Vehicle_No      := Set_Vehicle_No;
    Local_Task_Id   := Current_Task;
end Identify;

-- Replace the rest of this task with your own code.
-- Maybe synchronizing on an external event clock like "Wait_For_Next_Physics_Update",
-- yet you can synchronize on e.g. the real-time clock as well.

-- Without control this vehicle will go for its natural swarming instinct.
--  test
--  if(Vehicle_No = 1) then
--  Set_Destination(Destination);
--  Set_Throttle(2.0);
--  end if;

--  Initialize the variables, so that go to (0, 0, 0) at first, and remain the same if not
receiving a new globe place.
Num_Of_Vehicles := Vehicle_No;
Globe.Position := (others => 0.0);
Globe.Velocity := (others => 0.0);
```

```
Pre_Message.Vehicle_No := Vehicle_No;
Pre_Message.Position := Globe.Position;
Pre_Message.Velocity := Globe.Velocity;
Pre_Message.Time_Of_Finding := Clock;
Set_Destination(Destination);
Set_Throttle(Throttle);
Charge_Init := Current_Charge;

select

    Flight_Termination.Stop;

then abort

    Outer_task_loop : loop

        Wait_For_Next_Physics_Update;

            -- Your vehicle should respond to the world here: sense, listen, talk, act?
        if (Energy_Globes_Around'Length /= 0)
        then
            Globe := Energy_Globes_Around(1);
```

```
      Pre_Message.Position := Globe.Position;
      Pre_Message.Velocity := Globe.Velocity;
      Pre_Message.Time_Of_Finding := Clock;
      Send(Message => Pre_Message);
      --    Put_Line("yes" & Message.Position(x)'Image);
   else
      Send(Message => Pre_Message);
   end if;

   Receive(Message => Message);
   if (Message.Vehicle_No > Num_Of_Vehicles)
   then
      Num_Of_Vehicles := Message.Vehicle_No;
      Pre_Message.Vehicle_No := Message.Vehicle_No;
   end if;
   if (Message.Time_Of_Finding > Pre_Message.Time_Of_Finding)
   then
      Globe.Position := Message.Position;
      Globe.Velocity := Message.Velocity;
      Pre_Message.Position := Globe.Position;
      Pre_Message.Velocity := Globe.Velocity;
      Pre_Message.Time_Of_Finding := Message.Time_Of_Finding;
```

```
            end if;
            -- update globe position regarding time passing
            Globe.Position := Pre_Message.Position + Pre_Message.Velocity *
Real(To_Duration(Clock - Pre_Message.Time_Of_Finding));
            -- update radius regarding vehicle number updating
            Radius := 2.4 * Energy_Globe_Detection * Sqrt(Real(Num_Of_Vehicles) / 64.0);
            -- update radius regarding charge changing
            Radius := Radius * Sqrt(Real(Current_Charge / Charge_Init));
            Estimated_Time := (Sqrt((Real(abs(Velocity))**2) + 2.0 * abs(Acceleration) *
Radius) - abs(Velocity)) / abs(Acceleration);
            if (Real(Current_Charge) - Estimated_Time * Current_Discharge_Per_Sec <
Real(0.1 * Charge_Init))
            then
                Destination := Globe.Position;
                Set_Destination(Destination);
                Set_Throttle(Throttle * 2.0);
            else
                Z_Coordinate := (2.0 * Real(Vehicle_No) - 1.0) / Real(Num_Of_Vehicles) - 1.0;
                X_Coordinate := Sqrt(1.0 - (Z_Coordinate**2)) * Cos(2.0 * PI *
Real(Vehicle_No) * Phi);
                Y_Coordinate := Sqrt(1.0 - (Z_Coordinate**2)) * Sin(2.0 * PI *
Real(Vehicle_No) * Phi);
```

```
                Destination(x) := Globe.Position(x) + Radius * X_Coordinate;

                Destination(y) := Globe.Position(y) + Radius * Y_Coordinate;

                Destination(z) := Globe.Position(z) + Radius * Z_Coordinate;

                Set_Destination(Destination);

            Set_Throttle(Throttle);

            end if;

        end loop Outer_task_loop;


    end select;


exception

    when E : others => Show_Exception (E);


end Vehicle_Task;


end Vehicle_Task_Type;
```

Average testing results for testing twice is:

Initial Number	Average Frame Rate	After 60 s	90 s	120 s	10 min	Survival Rate for 10 min
64	30 Hz	64	64	64	64	100 %
128	30 Hz	128	128	128	128	100 %
256	16 - 17 Hz	256	256	256	256	100 %
512	10 - 11 Hz	462	339.5	334	334	65.23 %

As for 512 vehicles, the survival rate for 120-seconds is lower than the last attempt, which

is probably because when all vehicles get the real number of vehicles, the average distance they need to move in the first 90 seconds is longer than when some of them get a smaller number of vehicles, then the first 90 seconds' crowding becomes more extreme.

But all vehicles would get charged after the first 120 seconds, because they get their unique place which is evenly distributed on the sphere.

5.2 Stage 3

Since this stage, results of the test is more random, and adjusting parameters is much more time-consuming.

The main aim of the assignment is to try out solutions. Thus, little time is paid for adjusting parameters. There's only one attempt for stage 3.

Testing codes for vehicle_task_type.adb is:

```
-- Suggestions for packages which might be useful:

with Ada.Real_Time;                    use Ada.Real_Time;
--  with Ada.Text_IO;                  use Ada.Text_IO;
with Exceptions;                       use Exceptions;
with Real_Type;                        use Real_Type;
--  with Generic_Sliding_Statistics;
--  with Rotations;                    use Rotations;
with Vectors_3D;                       use Vectors_3D;
```

```
with Vehicle_Interface;                 use Vehicle_Interface;
with Vehicle_Message_Type;              use Vehicle_Message_Type;
--    with Swarm_Structures;                    use Swarm_Structures;
with Swarm_Structures_Base;             use Swarm_Structures_Base;
with Swarm_Configuration;               use Swarm_Configuration;
with Ada.Numerics.Long_Elementary_Functions;
use Ada.Numerics.Long_Elementary_Functions;

package body Vehicle_Task_Type is

   task body Vehicle_Task is

      -- stage 2
      Vehicle_No : Positive;--pragma Unreferenced (Vehicle_No);
      -- You will want to take the pragma out, once you use the "Vehicle_No"
      -- initial destination
      Destination : Vector_3D := (others => 0.0);
      -- information of finding globes
      --    Globes_Found : Natural := 1;
      Globe : Energy_Globe;
      Message : Inter_Vehicle_Messages;
      -- Pre_Message actually stores the newest information of the globe
```

```
      Pre_Message : Inter_Vehicle_Messages;
      Num_Of_Vehicles: Positive;
      Radius : Real;
      X_Coordinate, Y_Coordinate, Z_Coordinate : Real;
      Charge_Init : Vehicle_Charges;
      Estimated_Time : Real;
      Throttle : constant Throttle_T := 0.5;
      PI : constant Real := 3.1415926536;
      Phi : constant Real := (Sqrt(5.0) - 1.0) / 2.0;

      -- stage 3
      -- Newest invalid globes that have found
      Valid_Globe : Boolean := False;
      Invalid_Time : constant Real := 2.0;
      Time_Init : constant Time := Clock;

   begin

      -- You need to react to this call and provide your task_id.
      -- You can e.g. employ the assigned vehicle number (Vehicle_No)
      -- in communications with other vehicles.
      accept Identify (Set_Vehicle_No : Positive; Local_Task_Id : out Task_Id) do
```

```
      Vehicle_No          := Set_Vehicle_No;
      Local_Task_Id    := Current_Task;
   end Identify;

   -- Replace the rest of this task with your own code.
   -- Maybe synchronizing on an external event clock like "Wait_For_Next_Physics_Update",
   -- yet you can synchronize on e.g. the real-time clock as well.

   -- Without control this vehicle will go for its natural swarming instinct.
   --    test
   --    if(Vehicle_No = 1) then
   --    Set_Destination(Destination);
   --    Set_Throttle(2.0);
   --    end if;

   --    Initialize the variables, so that go to (0, 0, 0) at first, and remain the same if not
receiving a new globe place.
   Num_Of_Vehicles := Vehicle_No;
   Globe.Position := (others => 0.0);
   Globe.Velocity := (others => 0.0);
   Pre_Message.Vehicle_No := Vehicle_No;
   Pre_Message.Position := Globe.Position;
```

```
Pre_Message.Velocity := Globe.Velocity;
Pre_Message.Time_Of_Finding := Clock;
Set_Destination(Destination);
Set_Throttle(Throttle);
Charge_Init := Current_Charge;

select

    Flight_Termination.Stop;

then abort

    Outer_task_loop : loop

        Wait_For_Next_Physics_Update;

            -- Your vehicle should respond to the world here: sense, listen, talk, act?
        if (Energy_Globes_Around'Length /= 0)
        then
            Globe := Energy_Globes_Around(1);
            Pre_Message.Position := Globe.Position;
            Pre_Message.Velocity := Globe.Velocity;
```

```
            Pre_Message.Time_Of_Finding := Clock;
            -- find 2 globes at a time is low possibility
            if (Energy_Globes_Around'Length > 1)
            then
                if(abs(Position - Energy_Globes_Around(2).Position) < abs(Position -
Pre_Message.Position))
                then
                    Globe := Energy_Globes_Around(2);
                    Pre_Message.Position := Globe.Position;
                    Pre_Message.Velocity := Globe.Velocity;
                end if;
            end if;
            Send(Message => Pre_Message);

        else
            Send(Message => Pre_Message);
        end if;

        Receive(Message => Message);
        if (Message.Vehicle_No > Num_Of_Vehicles)
        then
            Num_Of_Vehicles := Message.Vehicle_No;
```

```
            Pre_Message.Vehicle_No := Message.Vehicle_No;
        end if;
        if (Real(To_Duration(Clock - Message.Time_Of_Finding)) < Invalid_Time)
        then
            if (abs(Position - Message.Position) < abs(Position - Pre_Message.Position))
            then
                Globe.Position := Message.Position;
                Globe.Velocity := Message.Velocity;
                Pre_Message.Position := Globe.Position;
                Pre_Message.Velocity := Globe.Velocity;
                Pre_Message.Time_Of_Finding := Message.Time_Of_Finding;
                Valid_Globe := True;
            elsif (Message.Time_Of_Finding > Pre_Message.Time_Of_Finding
                    and (Real(To_Duration(Clock - Pre_Message.Time_Of_Finding)) >=
Invalid_Time
                            or Valid_Globe = False))
            then
                Globe.Position := Message.Position;
                Globe.Velocity := Message.Velocity;
                Pre_Message.Position := Globe.Position;
                Pre_Message.Velocity := Globe.Velocity;
                Pre_Message.Time_Of_Finding := Message.Time_Of_Finding;
```

```
                Valid_Globe := True;
            end if;
        end if;
        -- update globe position regarding time passing
        Globe.Position := Pre_Message.Position + Pre_Message.Velocity *
Real(To_Duration(Clock - Pre_Message.Time_Of_Finding));
        if (Real(To_Duration(Clock - Pre_Message.Time_Of_Finding)) >= Invalid_Time)
        then
            Globe.Position := (others => 0.0);
            Pre_Message.Position := Globe.Position;
            Pre_Message.Velocity := (others => 0.0);
            Pre_Message.Time_Of_Finding := Time_Init;
        end if;
         --   update radius regarding vehicle number updating
        Radius := 1.5 * Energy_Globe_Detection * Sqrt(Real(Num_Of_Vehicles) / 64.0);
        -- update radius regarding charge changing
        Radius := Radius * Sqrt(Real(Current_Charge / Charge_Init));
        Estimated_Time := (Sqrt((Real(abs(Velocity)) ** 2) + 2.0 * abs(Acceleration) *
Radius) - abs(Velocity)) / abs(Acceleration) + Invalid_Time;
        if (Real(Current_Charge) - Estimated_Time * Current_Discharge_Per_Sec <
Real(0.1 * Charge_Init))
        then
```

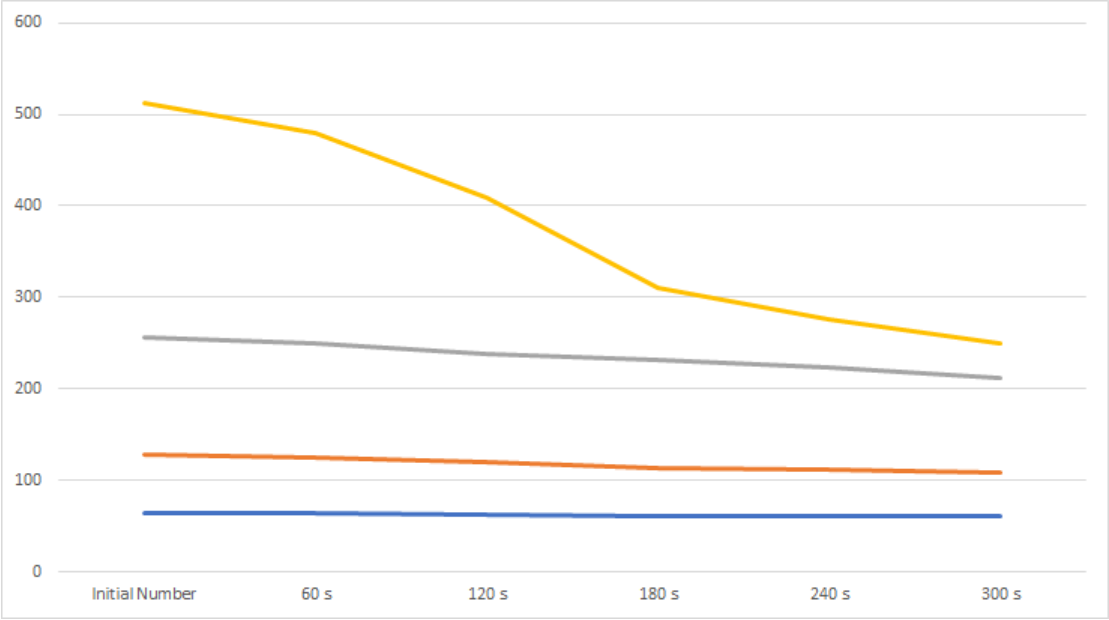
```
                Destination := Globe.Position;
                Set_Destination(Destination);
                Set_Throttle(Throttle * 2.0);
            else
                Z_Coordinate := (2.0 * Real(Vehicle_No) - 1.0) / Real(Num_Of_Vehicles) - 1.0;
                X_Coordinate := Sqrt(1.0 - (Z_Coordinate ** 2)) * Cos(2.0 * PI *
Real(Vehicle_No) * Phi);
                Y_Coordinate := Sqrt(1.0 - (Z_Coordinate ** 2)) * Sin(2.0 * PI *
Real(Vehicle_No) * Phi);
                Destination(x) := Globe.Position(x) + Radius * X_Coordinate;
                Destination(y) := Globe.Position(y) + Radius * Y_Coordinate;
                Destination(z) := Globe.Position(z) + Radius * Z_Coordinate;
                Set_Destination(Destination);
                Set_Throttle(Throttle);
            end if;
        end loop Outer_task_loop;

    end select;

exception
    when E : others => Show_Exception (E);
```

end Vehicle_Task;

end Vehicle_Task_Type;

Average testing results for testing twice is:

Initial Number	Average Frame Rate	After 60 s	120 s	180 s	240 s	300 s	Survival Rate For 5 min
64	30 Hz	64	63	61.5	61	60.5	94.53 %
128	30 Hz	124.5	119.5	114	111.5	109	85.16 %
256	16 - 17 Hz	249.5	237.5	231.5	223	212.5	83.01 %
512	12 - 13 Hz	479.5	409	311	270.5	243.5	47.56 %



By comparing the results to that of stage 2, it can be figured that survival rate is lower, and the number of surviving vehicles can't be stable even after the first 2 minutes, because energy globes are randomly disappearing and the length of time for vehicles to find another globe and get charged can be very long if it's out of luck.

It can always be seen that some vehicles form a sphere and follow an energy globe to a

far position, and the energy globe suddenly disappeared, then the vehicles go the origin point and most of them can find another globe, which is within the expected program logic and assumptions.

Because of the lower surviving number of vehicles in average, the average running frame is bigger than that for stage 2 program, and is increasing as time goes by and vehicles gradually vanishing. This happened in stage 4 tests as well.

5.3 Stage 4

5.3.1 First Attempt

This is a failed attempt. Vehicles can not automatically limit their scale and destroy some of themselves.

Testing codes for vehicle_message_type.ads (which is used only for this attempt) is:

```
with Ada.Real_Time;          use Ada.Real_Time;
with Swarm_Size;             use Swarm_Size;
with Vectors_3D;             use Vectors_3D;

package Vehicle_Message_Type is

   -- Replace this record definition by what your vehicles need to communicate.

   type Inter_Vehicle_Messages is
```

```
        record
            -- Vehicle Number
            Vehicle_No : Positive;
            -- informations of finding single globe
            Position : Vector_3D;
            Velocity : Vector_3D;
            -- time of finding the energy globe
            Time_Of_Finding : Time;

            -- stage 4
            -- the number of vehicles already scheduled for destruction
            Number_Of_Volunteers : Natural;
            -- the time for last deside the destruction plan
            When_Decide : Time;

        end record;

end Vehicle_Message_Type;
```

Testing codes for vehicle_task_type.adb is:

```
-- Suggestions for packages which might be useful:

with Ada.Real_Time;                 use Ada.Real_Time;
--  with Ada.Text_IO;                   use Ada.Text_IO;
with Exceptions;                    use Exceptions;
with Real_Type;                     use Real_Type;
--  with Generic_Sliding_Statistics;
--  with Rotations;                     use Rotations;
with Vectors_3D;                    use Vectors_3D;
with Vehicle_Interface;             use Vehicle_Interface;
with Vehicle_Message_Type;          use Vehicle_Message_Type;
--  with Swarm_Structures;              use Swarm_Structures;
with Swarm_Structures_Base;         use Swarm_Structures_Base;
with Swarm_Configuration;           use Swarm_Configuration;
with Swarm_Size;                    use Swarm_Size;
with Ada.Numerics.Long_Elementary_Functions;
use Ada.Numerics.Long_Elementary_Functions;

package body Vehicle_Task_Type is

   task body Vehicle_Task is
```

```
-- stage 2

Vehicle_No : Positive;--pragma Unreferenced (Vehicle_No);

-- You will want to take the pragma out, once you use the "Vehicle_No"

-- initial destination

Destination : Vector_3D := (others => 0.0);

-- information of finding globes

-- Globes_Found : Natural := 1;

Globe : Energy_Globe;

Message : Inter_Vehicle_Messages;

-- Pre_Message actually stores the newest information of the globe

Pre_Message : Inter_Vehicle_Messages;

Num_Of_Vehicles: Positive;

Radius : Real;

X_Coordinate, Y_Coordinate, Z_Coordinate : Real;

Charge_Init : Vehicle_Charges;

Estimated_Time : Real;

Throttle : constant Throttle_T := 0.5;

PI : constant Real := 3.1415926536;

Phi : constant Real := (Sqrt(5.0) - 1.0) / 2.0;


-- stage 3

-- Newest invalid globes that have found
```

```
      Valid_Globe : Boolean := False;
      Invalid_Time : constant Real := 2.0;
      Time_Init : constant Time := Clock;

      -- stage 4
      Now_Destruct : Boolean := False;
      Will_Destruct : Boolean;
      When_Decide : Time;
      Tolerant_Time : constant Duration := 6.0;
      INFINITY : constant Real := 99999999.0;

   begin

      -- You need to react to this call and provide your task_id.
      -- You can e.g. employ the assigned vehicle number (Vehicle_No)
      -- in communications with other vehicles.
      accept Identify (Set_Vehicle_No : Positive; Local_Task_Id : out Task_Id) do
         Vehicle_No      := Set_Vehicle_No;
         Local_Task_Id   := Current_Task;
      end Identify;

      -- Replace the rest of this task with your own code.
```

```
-- Maybe synchronizing on an external event clock like "Wait_For_Next_Physics_Update",
-- yet you can synchronize on e.g. the real-time clock as well.

-- Without control this vehicle will go for its natural swarming instinct.
--    test
--    if(Vehicle_No = 1) then
--    Set_Destination(Destination);
--    Set_Throttle(2.0);
--    end if;

--    Initialize the variables, so that go to (0, 0, 0) at first, and remain the same if not receiving a new globe place.
Num_Of_Vehicles := Vehicle_No;
Globe.Position := (others => 0.0);
Globe.Velocity := (others => 0.0);
Pre_Message.Vehicle_No := Vehicle_No;
Pre_Message.Position := Globe.Position;
Pre_Message.Velocity := Globe.Velocity;
Pre_Message.Time_Of_Finding := Clock;

Charge_Init := Current_Charge;
```

```
Set_Destination(Destination);
Set_Throttle(Throttle);

-- stage 4 initialization, try to be in the destruction plan
Will_Destruct := False;
Pre_Message.Number_Of_Volunteers := 0;



select

   Flight_Termination.Stop;

then abort

   Outer_task_loop : loop

      Wait_For_Next_Physics_Update;

         -- Your vehicle should respond to the world here: sense, listen, talk, act?
      if (Energy_Globes_Around'Length /= 0)
      then
```

```
                Globe := Energy_Globes_Around(1);
                Pre_Message.Position := Globe.Position;
                Pre_Message.Velocity := Globe.Velocity;
                Pre_Message.Time_Of_Finding := Clock;
                -- find 2 globes at a time is low possibility
                if (Energy_Globes_Around'Length > 1)
                then
                    if(abs(Position - Energy_Globes_Around(2).Position) < abs(Position -
Pre_Message.Position))
                    then
                        Globe := Energy_Globes_Around(2);
                        Pre_Message.Position := Globe.Position;
                        Pre_Message.Velocity := Globe.Velocity;
                    end if;
                end if;
                Send(Message => Pre_Message);

            else
                Send(Message => Pre_Message);
            end if;

            Receive(Message => Message);
```

```
-- stage 4
if (Message.Number_Of_Volunteers > Pre_Message.Number_Of_Volunteers)
then
    Pre_Message.Number_Of_Volunteers := Message.Number_Of_Volunteers;
    Pre_Message.When_Decide := Message.When_Decide;
elsif (Message.Number_Of_Volunteers = Pre_Message.Number_Of_Volunteers and
Message.When_Decide < Pre_Message.When_Decide)
then
    Pre_Message.Number_Of_Volunteers := Message.Number_Of_Volunteers;
    Pre_Message.When_Decide := Message.When_Decide;
end if;
if (Will_Destruct)
then
    if (When_Decide < Pre_Message.When_Decide)
    then
        Will_Destruct := False;
    end if;
else
    if (Current_Charge < 0.01 * Charge_Init)
    then
        Now_Destruct := True;
```

```
When_Decide := Clock;

Will_Destruct := True;

Pre_Message.Number_Of_Volunteers := Pre_Message.Number_Of_Volunteers + 1;

Pre_Message.When_Decide := When_Decide;

elsif (Pre_Message.Number_Of_Volunteers < Num_Of_Vehicles - Target_No_of_Elements)

then

When_Decide := Clock;

Will_Destruct := True;

Pre_Message.Number_Of_Volunteers := Pre_Message.Number_Of_Volunteers + 1;

Pre_Message.When_Decide := When_Decide;

end if;

end if;

if (Pre_Message.Number_Of_Volunteers >= Num_Of_Vehicles - Target_No_of_Elements

and Will_Destruct and To_Duration(Clock - Pre_Message.When_Decide) > Tolerant_Time)

then

Now_Destruct := True;

end if;
```

```
if (Message.Vehicle_No > Num_Of_Vehicles)
then
    Num_Of_Vehicles := Message.Vehicle_No;
    Pre_Message.Vehicle_No := Message.Vehicle_No;
end if;
if (Real(To_Duration(Clock - Message.Time_Of_Finding)) < Invalid_Time)
then
    if (abs(Position - Message.Position) < abs(Position - Pre_Message.Position))
    then
        Globe.Position := Message.Position;
        Globe.Velocity := Message.Velocity;
        Pre_Message.Position := Globe.Position;
        Pre_Message.Velocity := Globe.Velocity;
        Pre_Message.Time_Of_Finding := Message.Time_Of_Finding;
        Valid_Globe := True;
    elsif (Message.Time_Of_Finding > Pre_Message.Time_Of_Finding
            and (Real(To_Duration(Clock - Pre_Message.Time_Of_Finding)) >=
Invalid_Time
                or Valid_Globe = False))
    then
        Globe.Position := Message.Position;
```

```
                Globe.Velocity := Message.Velocity;
                Pre_Message.Position := Globe.Position;
                Pre_Message.Velocity := Globe.Velocity;
                Pre_Message.Time_Of_Finding := Message.Time_Of_Finding;
                Valid_Globe := True;
            end if;
        end if;
        -- update globe position regarding time passing
        Globe.Position := Pre_Message.Position + Pre_Message.Velocity *
Real(To_Duration(Clock - Pre_Message.Time_Of_Finding));
        if (Real(To_Duration(Clock - Pre_Message.Time_Of_Finding)) >= Invalid_Time)
        then
            Globe.Position := (others => 0.0);
            Pre_Message.Position := Globe.Position;
            Pre_Message.Velocity := (others => 0.0);
            Pre_Message.Time_Of_Finding := Time_Init;
        end if;

        -- stage 4
        if (Now_Destruct)
        then
            Destination := (others => INFINITY);
```

```
                Set_Destination(Destination);
                Set_Throttle(Throttle);
            else

                -- update radius regarding vehicle number updating
                Radius := 1.5 * Energy_Globe_Detection * Sqrt(Real(Num_Of_Vehicles) /
64.0);
                -- update radius regarding charge changing
                Radius := Radius * Sqrt(Real(Current_Charge / Charge_Init));
                Estimated_Time := (Sqrt((Real(abs(Velocity)) ** 2) + 2.0 * abs(Acceleration) *
Radius) - abs(Velocity)) / abs(Acceleration) + Invalid_Time;
                if (Real(Current_Charge) - Estimated_Time * Current_Discharge_Per_Sec <
Real(0.1 * Charge_Init))
                then
                    Destination := Globe.Position;
                    Set_Destination(Destination);
                    Set_Throttle(Throttle * 2.0);
                else
                    Z_Coordinate := (2.0 * Real(Vehicle_No) - 1.0) / Real(Num_Of_Vehicles)
- 1.0;
                    X_Coordinate := Sqrt(1.0 - (Z_Coordinate ** 2)) * Cos(2.0 * PI *
Real(Vehicle_No) * Phi);
```

```
                        Y_Coordinate := Sqrt(1.0 - (Z_Coordinate ** 2)) * Sin(2.0 * PI *
Real(Vehicle_No) * Phi);
                        Destination(x) := Globe.Position(x) + Radius * X_Coordinate;
                        Destination(y) := Globe.Position(y) + Radius * Y_Coordinate;
                        Destination(z) := Globe.Position(z) + Radius * Z_Coordinate;
                        Set_Destination(Destination);
                        Set_Throttle(Throttle);
                    end if;

                end if;

            end loop Outer_task_loop;

        end select;

    exception
        when E : others => Show_Exception (E);

    end Vehicle_Task;

end Vehicle_Task_Type;
```

5.3.2 Second Attempt

This is a failed attempt.

Vehicles can reach an agreement after some time, then the volunteers in the destruction plan do go infinitely far away to destroy themselves, which is the successful and expected part of the test.

However, then the rest of vehicles that are not in the destruction plan can't find an energy globe around the origin point anymore. It can be observed that they turned around and waited for another energy globe until they were out of energy. This is probably because when some vehicles go to really far away, there's a bug occur and energy globes would not appear around the origin point then.

Testing codes for vehicle_message_type.ads (which is used only for this attempt) is:

```
-- Suggestions for packages which might be useful:

with Ada.Real_Time;              use Ada.Real_Time;
with Swarm_Size;                 use Swarm_Size;
with Vectors_3D;                 use Vectors_3D;

package Vehicle_Message_Type is

   -- Replace this record definition by what your vehicles need to communicate.
   type Vehicle_Type is array(Positive range<>) of Positive;
```

```
type Inter_Vehicle_Messages is

   record
      -- Number of Vehicles
      Vehicle_No : Positive;
      -- informations of finding single globe
      Position : Vector_3D;
      Velocity : Vector_3D;
      -- time of finding the energy globe
      Time_Of_Finding : Time;

      -- stage 4
      -- the destruction plan list, stores id of vehicles
      Vehicles : Vehicle_Type(1 .. Target_No_of_Elements);
      -- the number of vehicles already scheduled for destruction
      Number_Of_Volunteers : Natural;
      -- the time for last deside the destruction plan
      When_Decide : Time;

   end record;

end Vehicle_Message_Type;
```

Testing codes for vehicle_task_type.adb is:

```
-- Suggestions for packages which might be useful:

with Ada.Real_Time;                    use Ada.Real_Time;
--  with Ada.Text_IO;                  use Ada.Text_IO;
with Exceptions;                       use Exceptions;
with Real_Type;                        use Real_Type;
--  with Generic_Sliding_Statistics;
--  with Rotations;                    use Rotations;
with Vectors_3D;                       use Vectors_3D;
with Vehicle_Interface;                use Vehicle_Interface;
with Vehicle_Message_Type;             use Vehicle_Message_Type;
--  with Swarm_Structures;             use Swarm_Structures;
with Swarm_Structures_Base;            use Swarm_Structures_Base;
with Swarm_Configuration;              use Swarm_Configuration;
with Swarm_Size;                       use Swarm_Size;
with Ada.Numerics.Long_Elementary_Functions;
use Ada.Numerics.Long_Elementary_Functions;

package body Vehicle_Task_Type is
```

```
task body Vehicle_Task is


   -- stage 2

   Vehicle_No : Positive;--pragma Unreferenced (Vehicle_No);

   -- You will want to take the pragma out, once you use the "Vehicle_No"

   -- initial destination

   Destination : Vector_3D := (others => 0.0);

   -- information of finding globes

   -- Globes_Found : Natural := 1;

   Globe : Energy_Globe;

   Message : Inter_Vehicle_Messages;

   -- Pre_Message actually stores the newest information of the globe

   Pre_Message : Inter_Vehicle_Messages;

   Num_Of_Vehicles: Positive;

   Radius : Real;

   X_Coordinate, Y_Coordinate, Z_Coordinate : Real;

   Charge_Init : Vehicle_Charges;

   Estimated_Time : Real;

   Throttle : constant Throttle_T := 0.5;

   PI : constant Real := 3.1415926536;

   Phi : constant Real := (Sqrt(5.0) - 1.0) / 2.0;
```

```
    -- stage 3
    -- Newest invalid globes that have found
    Valid_Globe : Boolean;
    Invalid_Time : constant Real := 2.0;
    Time_Init : constant Time := Clock;

    -- stage 4
    Now_Destruct : Boolean;
    Will_Destruct : Boolean;
    Tolerant_Time : constant Duration := 5.0;
    INFINITY : constant Real := 99999999.0;

begin

    -- You need to react to this call and provide your task_id.
    -- You can e.g. employ the assigned vehicle number (Vehicle_No)
    -- in communications with other vehicles.
    accept Identify (Set_Vehicle_No : Positive; Local_Task_Id : out Task_Id) do
        Vehicle_No      := Set_Vehicle_No;
        Local_Task_Id   := Current_Task;
    end Identify;
```

-- Replace the rest of this task with your own code.

-- Maybe synchronizing on an external event clock like "Wait_For_Next_Physics_Update",

-- yet you can synchronize on e.g. the real-time clock as well.

-- Without control this vehicle will go for its natural swarming instinct.

-- Initialize the variables, so that go to (0, 0, 0) at first, and remain the same if not receiving a new globe place.

-- Num_Of_Vehicles is a shorter local name of Pre_Message.Vehicle_No

```
Num_Of_Vehicles := Vehicle_No;
Globe.Position := (others => 0.0);
Globe.Velocity := (others => 0.0);
Pre_Message.Vehicle_No := Vehicle_No;
Pre_Message.Position := Globe.Position;
Pre_Message.Velocity := Globe.Velocity;
Pre_Message.Time_Of_Finding := Clock;

Charge_Init := Current_Charge;

Set_Destination(Destination);
Set_Throttle(Throttle);
```

```
-- stage 3 initialization

Valid_Globe := False;

-- stage 4 initialization, try to be in the destruction plan initially

Now_Destruct := False;

Will_Destruct := True;

Pre_Message.Number_Of_Volunteers := 1;

Pre_Message.When_Decide := Clock;

Pre_Message.Vehicles(1) := Vehicle_No;




select

    Flight_Termination.Stop;


then abort

    Outer_task_loop : loop


        Wait_For_Next_Physics_Update;


            -- Your vehicle should respond to the world here: sense, listen, talk, act?
```

```
if (Energy_Globes_Around'Length /= 0)
then
    Globe := Energy_Globes_Around(1);
    Pre_Message.Position := Globe.Position;
    Pre_Message.Velocity := Globe.Velocity;
    Pre_Message.Time_Of_Finding := Clock;
    -- find 2 globes at a time is low possibility
    if (Energy_Globes_Around'Length > 1)
    then
        if(abs(Position - Energy_Globes_Around(2).Position) < abs(Position -
Pre_Message.Position))
        then
            Globe := Energy_Globes_Around(2);
            Pre_Message.Position := Globe.Position;
            Pre_Message.Velocity := Globe.Velocity;
        end if;
    end if;
    Send(Message => Pre_Message);

else
    Send(Message => Pre_Message);
end if;
```

```
Receive(Message => Message);


-- stage 4
if (Message.Number_Of_Volunteers > Pre_Message.Number_Of_Volunteers)
then
    Pre_Message.Number_Of_Volunteers := Message.Number_Of_Volunteers;
    Pre_Message.When_Decide := Message.When_Decide;
    Pre_Message.Vehicles := Message.Vehicles;
elsif (Message.Number_Of_Volunteers = Pre_Message.Number_Of_Volunteers
        and Message.When_Decide < Pre_Message.When_Decide)
then
    Pre_Message.Number_Of_Volunteers := Message.Number_Of_Volunteers;
    Pre_Message.When_Decide := Message.When_Decide;
    Pre_Message.Vehicles := Message.Vehicles;
end if;
if (Will_Destruct)
then
    Will_Destruct := False;
    for i in 1..Pre_Message.Number_Of_Volunteers loop
        if (Pre_Message.Vehicles(i) = Vehicle_No) then
            Will_Destruct := True;
```

```
                end if;
            end loop;
        else
            if (Pre_Message.Number_Of_Volunteers < Num_Of_Vehicles -
Target_No_of_Elements)
            then
                Pre_Message.When_Decide := Clock;
                Will_Destruct := True;
                Pre_Message.Number_Of_Volunteers :=
Pre_Message.Number_Of_Volunteers + 1;
                Pre_Message.Vehicles(Pre_Message.Number_Of_Volunteers) :=
Vehicle_No;
            end if;
        end if;

        if (Pre_Message.Number_Of_Volunteers >= Num_Of_Vehicles -
Target_No_of_Elements
            and Will_Destruct and To_Duration(Clock - Pre_Message.When_Decide) >
Tolerant_Time)
        then
            Now_Destruct := True;
        end if;
```

```
if (Message.Vehicle_No > Pre_Message.Vehicle_No)
then
    Num_Of_Vehicles := Message.Vehicle_No;
    Pre_Message.Vehicle_No := Message.Vehicle_No;
end if;
if (Real(To_Duration(Clock - Message.Time_Of_Finding)) < Invalid_Time)
then
    if (abs(Position - Message.Position) < abs(Position - Pre_Message.Position))
    then
        Globe.Position := Message.Position;
        Globe.Velocity := Message.Velocity;
        Pre_Message.Position := Globe.Position;
        Pre_Message.Velocity := Globe.Velocity;
        Pre_Message.Time_Of_Finding := Message.Time_Of_Finding;
        Valid_Globe := True;
    elsif (Message.Time_Of_Finding > Pre_Message.Time_Of_Finding
            and (Real(To_Duration(Clock - Pre_Message.Time_Of_Finding)) >=
Invalid_Time
                    or Valid_Globe = False))
    then
```

```
                Globe.Position := Message.Position;
                Globe.Velocity := Message.Velocity;
                Pre_Message.Position := Globe.Position;
                Pre_Message.Velocity := Globe.Velocity;
                Pre_Message.Time_Of_Finding := Message.Time_Of_Finding;
                Valid_Globe := True;
            end if;
        end if;
        -- update globe position regarding time passing
        Globe.Position := Pre_Message.Position + Pre_Message.Velocity *
Real(To_Duration(Clock - Pre_Message.Time_Of_Finding));
        if (Real(To_Duration(Clock - Pre_Message.Time_Of_Finding)) >= Invalid_Time)
        then
            Globe.Position := (others => 0.0);
            Pre_Message.Position := Globe.Position;
            Pre_Message.Velocity := (others => 0.0);
            Pre_Message.Time_Of_Finding := Time_Init;
        end if;

        -- stage 4
        if (Now_Destruct)
        then
```

```
                Destination := (others => INFINITY);
                Set_Destination(Destination);
                Set_Throttle(Throttle);
            else
                -- update radius regarding vehicle number updating
                Radius := 1.3 * Energy_Globe_Detection * Sqrt(Real(Num_Of_Vehicles) /
64.0);
                -- update radius regarding charge changing
                Radius := Radius * Sqrt(Real(Current_Charge / Charge_Init));
                Estimated_Time := (Sqrt((Real(abs(Velocity)) ** 2) + 2.0 * abs(Acceleration) *
Radius) - abs(Velocity)) / abs(Acceleration) + Invalid_Time;
                if (Real(Current_Charge) - Estimated_Time * Current_Discharge_Per_Sec <
Real(0.1 * Charge_Init))
                then
                    Destination := Globe.Position;
                    Set_Destination(Destination);
                    Set_Throttle(Throttle * 2.0);
                else
                    Z_Coordinate := (2.0 * Real(Vehicle_No) - 1.0) / Real(Num_Of_Vehicles)
- 1.0;
                    X_Coordinate := Sqrt(1.0 - (Z_Coordinate ** 2)) * Cos(2.0 * PI *
Real(Vehicle_No) * Phi);
```

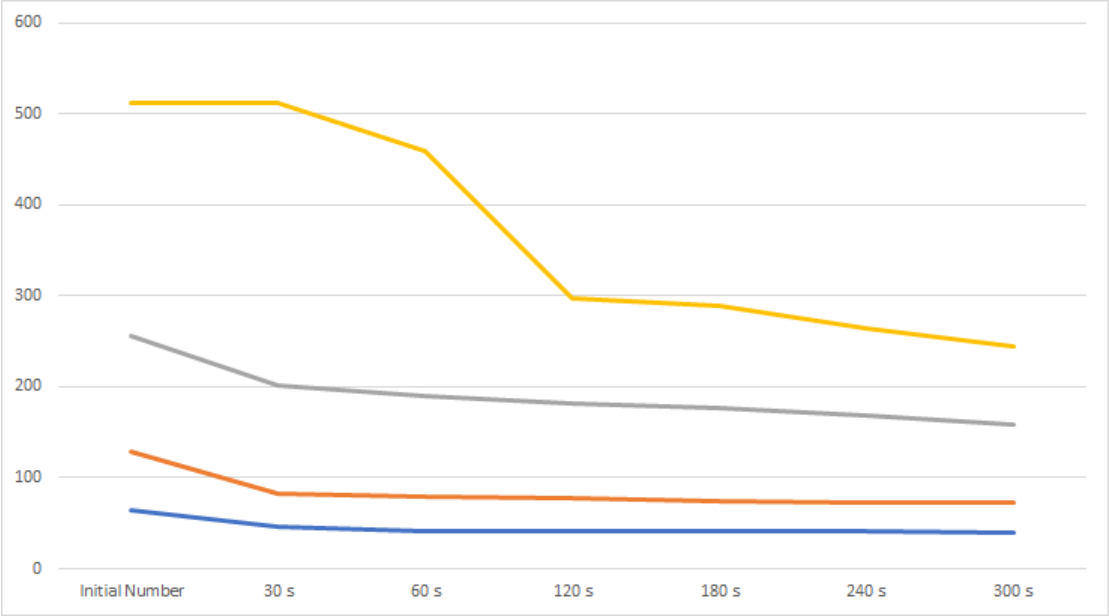
```
                        Y_Coordinate := Sqrt(1.0 - (Z_Coordinate ** 2)) * Sin(2.0 * PI *
Real(Vehicle_No) * Phi);
                        Destination(x) := Globe.Position(x) + Radius * X_Coordinate;
                        Destination(y) := Globe.Position(y) + Radius * Y_Coordinate;
                        Destination(z) := Globe.Position(z) + Radius * Z_Coordinate;
                        Set_Destination(Destination);
                        Set_Throttle(Throttle);
                    end if;
                end if;

            end loop Outer_task_loop;

        end select;

    exception
        when E : others => Show_Exception (E);

    end Vehicle_Task;

end Vehicle_Task_Type;
```

5.3.3 Third Attempt

This is a successful attempt.

Average testing results for testing twice is:

Initial Number	Target Number	Average Frame Rate	After 30 s	60 s	Achieving Rate For 1 min	120 s	Achieving Rate For 2 min	240 s	300 s	Achieving Rate For 5 min
64	42	30 Hz	46.5	42	100 %	41	97.62 %	40.5	39.5	94.05 %
128	80	30 Hz	82	79	98.75 %	77	96.25 %	73.5	72.5	90.63 %
256	190	23 - 24 Hz	202	190	100 %	182.5	96.05 %	167.5	159	83.68 %
512	380	13 - 14 Hz	511.5	459	120.79 %	298.5	78.55 %	265.5	247	65 %



Because of lower number of vehicles, the average frame rate is even bigger that that in stage 3, especially when the initial number of vehicles is 256.

Usually, volunteers will go for its natural swarming instinct (turning white) and get themselves destroyed in the first 35 seconds when initial number of vehicles is 64 or 128. The bigger initial number of vehicles is, the longer time it takes to agree on the destruction plan and to execute the plan, and there is some randomness for the length of that consuming time.

When the number of initial vehicles is 256, volunteers for the destruction plan can live for about 63 seconds at most. As for 512 initial vehicles, it takes over 30 seconds for vehicles to reach a consensus on the destruction plan and volunteers for the destruction plan can live till about 70 seconds at most, when the busiest first-charging-time have already come and many other vehicles can't get charged and vanished at that time, too. If the plan can be agreed sooner, the first-charging-time would be less busy and more vehicles can survive.

Thus, as for 512 initial vehicles' case, both achieving rate for 60 seconds and for 120 seconds are not precisely suitable to judge whether the destruction plan is carried out correctly or wrongly.

Another important thing observed is that volunteers in the destruction plan executed the plan, that is let themselves being taken over by the default swarming behavior and turn white in the screen, almost synchronously. When initial number of vehicles is 64, volunteers all turn white in 1 second when the destruction plan begins. As for 256 initial vehicles, volunteers all turn white in around 2 seconds when the destruction plan begins, which is also satisfying the 5-second grace-period assumption, which convincing states that the consensus decision making of the destruction plan satisfies consistency, as expected.

It can also be observed that volunteers that go for its natural swarming instinct can really hardly get charged that it is never being directly observed, which satisfies the assumption of the program.

Testing codes for vehicle_message_type.ads (which is used only for this attempt) is:

with Ada.Real_Time; *use Ada.Real_Time;*

```
with Swarm_Size;                    use Swarm_Size;
with Vectors_3D;                    use Vectors_3D;

package Vehicle_Message_Type is

   type Vehicle_Type is array (Positive range <>) of Positive;

   type Inter_Vehicle_Messages is

      record
         -- Number of Vehicles
         Vehicle_Num : Positive;
         -- informations of the finding globe
         Position : Vector_3D;
         Velocity : Vector_3D;
         -- timestamp of finding the energy globe
         Time_Of_Finding : Time;

         -- stage 4 definitions
         -- the destruction plan list, stores id of vehicles
         Vehicles : Vehicle_Type (1 .. Target_No_of_Elements);
         -- the number of vehicles already scheduled for destruction
```

```
        Number_Of_Volunteers : Natural;
        -- the time when the last decision made for the destruction plan
        When_Decide : Time;

    end record;

end Vehicle_Message_Type;
```

Testing codes for vehicle_task_type.adb is:

```
with Ada.Real_Time;                          use Ada.Real_Time;
with Exceptions;                             use Exceptions;
with Real_Type;                              use Real_Type;
with Vectors_3D;                             use Vectors_3D;
with Vehicle_Interface;                      use Vehicle_Interface;
with Vehicle_Message_Type;                   use Vehicle_Message_Type;
with Swarm_Structures_Base;                  use Swarm_Structures_Base;
with Swarm_Configuration;                    use Swarm_Configuration;
with Swarm_Size;                             use Swarm_Size;
with Ada.Numerics.Long_Elementary_Functions; use
Ada.Numerics.Long_Elementary_Functions;
```

```
package body Vehicle_Task_Type is

   task body Vehicle_Task is

      -- unique Id of the vehicle
      Vehicle_No : Positive;
      -- initial destination, vehicles go to origin point at first
      Destination : Vector_3D := (others => 0.0);
      -- information of finding globes
      -- Globes_Found : Natural := 1;
      -- initialize the variables, so that go to (0, 0, 0) at first, and remain the same if not receiving
a new globe place
      Globe : Energy_Globe :=
        (Position => (others => 0.0),
         Velocity => (others => 0.0));
      -- Message store the message just received.
      Message : Inter_Vehicle_Messages;
      -- Pre_Message actually stores the newest information of the globe
      -- will be initialized once Vehicle_No is given.
      Pre_Message : Inter_Vehicle_Messages;
      -- Radius stores the distance between globe and vehicle's destination -- the radius of sphere
      Radius : Real;
```

-- X, Y, Z store the coordinate of vehicle's position if globe is origin of coordinates and radius is 1

X_Coordinate, Y_Coordinate, Z_Coordinate : Real;

-- Estimated_Time is the estimated time to catch the globe

Estimated_Time : Real;

-- Charge_Init stores the initial charge of the vehicle, which is considered to be full-charged

-- so it is actually a constant variable, but should be assigned after the task begins

Charge_Init : Vehicle_Charges;

-- Throttle is the throttle of vehicles normally.

Throttle : constant Throttle_T := 0.5;

-- Default_Vehicle_Num is used to adjust the radius of the model

-- in respect to Radius_Rate Pre_Message.Vehicle_Num to Default_Vehicle_Num

Default_Vehicle_Num : constant Real := 64.0;

-- Radius_Rate is the ratio of radius to Energy_Globe_Detection when it's Default_Vehicle_Num

Radius_Rate : constant Real := 1.3;

-- PI is namely pi, ratio of the circumference of a circle to its diameter

PI : constant Real := 3.1415926536;

-- Phi is the golden ratio, known as 0.618

Phi : constant Real := (Sqrt (5.0) - 1.0) / 2.0;

-- stage 3 definitions & initialization

-- Valid_Globe is used to check if Globe is really a globe,

-- because initially it's (others => 0) to let vehicles go to the origin point

-- it's false only when globe is the initial (others => 0), which means globe is not valid

Valid_Globe : Boolean := False;

-- if the last time we found the globe till now is beyond Invalid_Time, then we think it's vanished

Invalid_Time : constant Real := 2.0;

-- Time_Init is when the task begins

Time_Init : constant Time := Clock;

-- stage 4 definitions & initialization

-- vehicles tries to be in the destruction plan initially

-- Now_Destruct shows if the vehicle need to destruct itself now

Now_Destruct : Boolean := False;

-- Will_Destruct shows if the vehicle want to be in the destruction plan

Will_Destruct : Boolean := True;

-- if the last we fulfill the destruction plan time till now is beyond Tolerant_Time, then the destruction plan should start

Tolerant_Time : constant Duration := 5.0;

-- other variables that would be used:

-- Energy_Globe_Detection: the distance that vehicle can detect the globe and get charged,

in Swarm/swarm_configuration.ads

-- Target_No_of_Elements: the target number of vehicles for stage 4, in Vehicle_Interface/swarm_size.ads

```
begin
   -- accept the unique id of the vehicle
   accept Identify (Set_Vehicle_No : Positive; Local_Task_Id : out Task_Id) do
      Vehicle_No      := Set_Vehicle_No;
      Local_Task_Id   := Current_Task;
   end Identify;
   -- store the initial charge of the vehicle which is thought to be full-charged
   Charge_Init := Current_Charge;
   -- initialize Pre_Message
   Pre_Message.Vehicle_Num := Vehicle_No;
   Pre_Message.Position := Globe.Position;
   Pre_Message.Velocity := Globe.Velocity;
   Pre_Message.Time_Of_Finding := Clock;

   -- stage 4 initialization
   Pre_Message.Number_Of_Volunteers := 1;
   Pre_Message.When_Decide := Clock;
   Pre_Message.Vehicles (1) := Vehicle_No;
```

```
-- ensure all vehicles go to origin point at first
Set_Destination (Destination);
Set_Throttle (Throttle);

-- repeat the following loop till the end of the task
select
    Flight_Termination.Stop;

then abort
    Outer_task_loop : loop
        Wait_For_Next_Physics_Update;

        -- if find the globe, update the information, then send the message

        if Energy_Globes_Around'Length /= 0
        then
            -- by default, only find 1 globe, because find 2 is of very low possibility
            Globe := Energy_Globes_Around (1);
```

```
                Pre_Message.Position := Globe.Position;
                Pre_Message.Velocity := Globe.Velocity;
                Pre_Message.Time_Of_Finding := Clock;
                -- find 3 globes at a time is almost impossible, so only consider finding 1 and 2
at a time
                if Energy_Globes_Around'Length > 1
                then
                    if abs (Position - Energy_Globes_Around (2).Position) < abs (Position -
Pre_Message.Position)
                    then
                        Globe := Energy_Globes_Around (2);
                        Pre_Message.Position := Globe.Position;
                        Pre_Message.Velocity := Globe.Velocity;
                    end if;
                end if;
                Send (Message => Pre_Message);

            else
                Send (Message => Pre_Message);
            end if;

            -- receive the message
```

```
Receive (Message => Message);

-- stage 4 part starts here

-- update the destruction plan if it's longer, or as long but earlier made
if Message.Number_Of_Volunteers > Pre_Message.Number_Of_Volunteers
then
    Pre_Message.Number_Of_Volunteers := Message.Number_Of_Volunteers;
    Pre_Message.When_Decide := Message.When_Decide;
    Pre_Message.Vehicles := Message.Vehicles;
elsif Message.Number_Of_Volunteers = Pre_Message.Number_Of_Volunteers
        and then Message.When_Decide < Pre_Message.When_Decide
then
    Pre_Message.Number_Of_Volunteers := Message.Number_Of_Volunteers;
    Pre_Message.When_Decide := Message.When_Decide;
    Pre_Message.Vehicles := Message.Vehicles;
end if;

-- if Will_Destruct, check if the vehicle is in the destruction plan after the plan
update
-- elsif plan is not full, try to be in the plan
if Will_Destruct
```

```
then
    Will_Destruct := False;
    for i in 1 .. Pre_Message.Number_Of_Volunteers loop
        if Pre_Message.Vehicles (i) = Vehicle_No then
            Will_Destruct := True;
        end if;
    end loop;
elsif Pre_Message.Number_Of_Volunteers < Pre_Message.Vehicle_Num -
Target_No_of_Elements
then
    Pre_Message.When_Decide := Clock;
    Will_Destruct := True;
    Pre_Message.Number_Of_Volunteers := Pre_Message.Number_Of_Volunteers
+ 1;
    Pre_Message.Vehicles (Pre_Message.Number_Of_Volunteers) :=  Vehicle_No;
end if;
-- stage 4 part ends here

-- update Pre_Message.Vehicle_Num
if Message.Vehicle_Num > Pre_Message.Vehicle_Num
then
    Pre_Message.Vehicle_Num := Message.Vehicle_Num;
```

```
            end if;

            -- update globe if message.globe is valid and closer, or valid and pre_is_not_valid
            if Real (To_Duration (Clock - Message.Time_Of_Finding)) < Invalid_Time
            then
                if abs (Position - Message.Position) < abs (Position - Pre_Message.Position)
                then
                    Globe.Position := Message.Position;
                    Globe.Velocity := Message.Velocity;
                    Pre_Message.Position := Globe.Position;
                    Pre_Message.Velocity := Globe.Velocity;
                    Pre_Message.Time_Of_Finding := Message.Time_Of_Finding;
                    -- since globe is no longer the origin point, update Valid_Globe
                    Valid_Globe := True;
                elsif Real (To_Duration (Clock - Pre_Message.Time_Of_Finding)) >=
Invalid_Time
                   or else Valid_Globe = False
                then
                    Globe.Position := Message.Position;
                    Globe.Velocity := Message.Velocity;
                    Pre_Message.Position := Globe.Position;
                    Pre_Message.Velocity := Globe.Velocity;
```

```
                Pre_Message.Time_Of_Finding := Message.Time_Of_Finding;
                Valid_Globe := True;
            end if;
        end if;

        -- update globe position regarding time-passing
        Globe.Position := Pre_Message.Position + Pre_Message.Velocity * Real
(To_Duration (Clock - Pre_Message.Time_Of_Finding));
        if Real (To_Duration (Clock - Pre_Message.Time_Of_Finding)) >= Invalid_Time
        then
            Globe.Position := (others => 0.0);
            Pre_Message.Position := Globe.Position;
            Pre_Message.Velocity := (others => 0.0);
            Pre_Message.Time_Of_Finding := Time_Init;
        end if;

        -- part for set destination

        -- stage 4 part starts here
        -- see if the vehicle is in the destruction plan
        for i in 1 .. Pre_Message.Number_Of_Volunteers loop
            if Pre_Message.Vehicles (i) = Vehicle_No then
```

```
                Now_Destruct := True;
            end if;
        end loop;
        -- if the vehicle is now destructing itself, let the default swarming behaviour take
over
        if Pre_Message.Number_Of_Volunteers >= Pre_Message.Vehicle_Num -
Target_No_of_Elements
            and then Now_Destruct and then To_Duration (Clock -
Pre_Message.When_Decide) > Tolerant_Time
        then
            null;
        else
            -- the vehicle is not going to destruction itself now
            Now_Destruct := False;
        -- stage 4 part ends here


            -- update radius regarding Energy_Globe_Detection and number of vehicles
            -- Comms_Range(within which distance vehicles can communicate with each
other, in Swarm/swarm_configuration.ads) is 0.2
            -- while Energy_Globe_Detection is 0.07
            Radius := Radius_Rate * Energy_Globe_Detection * Sqrt (Real
(Pre_Message.Vehicle_Num) / Default_Vehicle_Num);
```

```
-- update radius regarding charge changing
Radius := Radius * Sqrt (Real (Current_Charge / Charge_Init));
-- cauculate the Estimated_Time regaring to Velocity and Acceleration
-- stage 4: and add the Invalid_Time
Estimated_Time := (Sqrt ((Real (abs (Velocity)) ** 2) + 2.0 * abs (Acceleration)
* Radius)
                                - abs (Velocity)) / abs (Acceleration) + Invalid_Time;

-- if vehicle needs to be charged, rush to the globe as fast as it can, or else find its
position around the globe
if Real (Current_Charge) - Estimated_Time * Current_Discharge_Per_Sec <
Real (0.1 * Charge_Init)
then
   Destination := Globe.Position;
   Set_Destination (Destination);
   Set_Throttle (Throttle * 2.0);
else
   -- the aim is to evenly distribute the vehicles around the globe
   -- see the algorithm at:
   --      https://stackoverflow.com/questions/9600801/evenly-distributing-n-
points-on-a-sphere/26127012#26127012
   Z_Coordinate := (2.0 * Real (Vehicle_No) - 1.0) / Real
```

```
(Pre_Message.Vehicle_Num) - 1.0;
                        X_Coordinate := Sqrt (1.0 - (Z_Coordinate ** 2)) * Cos (2.0 * PI * Real
(Vehicle_No) * Phi);
                        Y_Coordinate := Sqrt (1.0 - (Z_Coordinate ** 2)) * Sin (2.0 * PI * Real
(Vehicle_No) * Phi);
                        Destination (x) := Globe.Position (x) + Radius * X_Coordinate;
                        Destination (y) := Globe.Position (y) + Radius * Y_Coordinate;
                        Destination (z) := Globe.Position (z) + Radius * Z_Coordinate;
                        Set_Destination (Destination);
                        Set_Throttle (Throttle);
                    end if;
                end if;

            end loop Outer_task_loop;

        end select;

    exception
        when E : others => Show_Exception (E);

    end Vehicle_Task;
```

end Vehicle_Task_Type;

6. References

[1] The assignment web page,

<https://cs.anu.edu.au/courses/comp2310/assessment/assignment1/>

[2] A stackoverflow page that discuss about algorithms to evenly distributes points on a sphere.

<https://stackoverflow.com/questions/9600801/evenly-distributing-n-points-on-a-sphere/26127012#26127012>

[3] An demo page of the Fibonacci sphere algorithm to evenly distributes points on a sphere.

<https://openprocessing.org/sketch/392066>

7. Acknowledgement

Thanks to all the classmates, tutors and lecturers who helped me via piazza. Thanks to those who discussed the assignment with me, including Xinyu Tian, Zidong Liu and a senior, who wrote a blog for it <https://blog.davidz.cn/anu-comp2310-assignment1/>.