A Kind of Aircraft Collision Avoidance Controller Based on Hybrid and Embedded Systems

Ziqi Zhao

Abstract

The most important thing in the aviation industry is safety. But when the aviation industry was just beginning, aircraft were very difficult to be controlled by humans. Because of the large number of buttons, complicated panels, and operation steps on the plane, the pilot operation caused many aircraft accidents. So today, I designed an aircraft collision avoidance system based on the embedded system. The logic of this obstacle avoidance system is relatively simple, so the computing speed is fast. But the disadvantage is that the avoidance path is not the shortest one in all possible paths.

1 Introduction

For the aircraft collision avoidance system, we can divide it into three modules: the motion module, i.e., the overall path planning and movement, the collision judgment module, and the collision avoidance module. The main difficult thing is how to do dynamic planning the path of the aircraft. We can choose to calculate all possible paths and then output or calculate each movement based on the current situation. I decided to calculate each step based on the current situation to improve calculation efficiency.

2 Function Introduction

There are three key points of the whole logic. The overall path planning, the judgment of the collision, and the action if the collision occurred. For my aircraft collision avoidance system, to maximize the computing efficiency and reduce the risk of collision, I unify the initial path planning as follows. First, move horizontally, when the plane's current horizontal coordinate and target point's horizontal coordinate are the same, then move vertically. In short, move the plane horizontally first, and then move vertically. This algorithm gives this system better performance. For the avoidance part, to reduce the amount of computation, I let aircraft A make a uniform avoidance process, i.e., turn to the right only, and aircraft B maintains the same path.

2.1 Get the Next Step Direction

The logic of the aircraft movement is complex. So I handle the aircraft movement and the aircraft direction separately to make sure the plane will not go backward. In the function `next_step_direction`, I calculate which direction the aircraft will go for the next step. We can calculate it based on the coordinates of the current position and the coordinates of the next position. Then compare it to get the corresponding direction.

```
next_step_direction: x1, y1, pnx, pny → next_step_direction_indicator
List[int, int] plane_cur_position := [x1,y1], plane_next_position := [pnx, pny]
if (x1 = pnx)
    if (y1 > pny)
        then {next_step_direction_indicator := down}
    if (y1 < pny)
        then {next_step_direction_indicator := up}
if (y1 = pny)
    if (x1 > pnx)
        then {next_step_direction_indicator := left}
    if (x1 < pnx)
        then {next_step_direction_indicator := right}
```

Figure 1: Compiled version of the *next_step_direction*

2.2 Move One Step Forward

Because I handle the aircraft movement and the aircraft direction separately, so the movement of the aircraft will always be 'forward' based on the current direction. I designed two functions for the movement of the aircraft. They are `flight_step_1` and `flight_step_2`. In the function `flight_step_1`, it just performs the basic move operation, i.e., moving horizontally then vertically. In the function `flight_step_2`, I need to consider whether the plane will go backward after this movement. Suppose that plane A turns once to the right to avoid plane B. Then, if the next step of plane A will make a backward path, we need to let plane A turn right again to avoid going backward. To achieve this, I have a list that records the historical coordinates of aircraft A. And the while loop is used to change the next direction of the plane until it can avoid a collision also not go backward. In this function, there is also a mandatory parameter. If this parameter is not empty, it sets the aircraft A to this coordinate point directly.

```
flight_step_1: x1, y1, x2, y2 → x3, y3
List[int, int] plane_cur_position := [x1,y1], plane_destination := [x2,y2]
if (plane_cur_position = plane_destination)
        then {x3 := x1, y3 := y1}
else if (x1 < x2)
        then {x1 := x1 +1}
else if (x1 > x2)
        then {x1 := x1 - 1}
else if (y1 < y2)
        then {y1 := y1 + 1}
else if (y1 > y2)
        then {y1 := y1 - 1}
```

Figure 2: Compiled version of the *flight_step_1*

2.3 Collision Judgement

In the collision judgment module, we must consider whether those two aircraft are within their communication ranges. So I designed the function `communicate` and the function `collision`. We can evaluate it by calculating the difference between two planes' current coordinates. After that, we can do the collision judgment process. We find three conditions that will cause the aircraft to collide: When the next position of aircraft A and the next position of aircraft B is the same. When the next position of aircraft A and the current position of aircraft B is the same. When the current position of aircraft A and the next position of aircraft B is the same.

```
collision:  communicateable, pacx, pacy, pbcx, pbcy, panx, pany, pbnx, pbny → collision_detect
bool communicateable := False, collision_detect = False
List[int, int] plane_a_cur_position := [pacx, pacy], plane_b_cur_position := [pbcx, pbcy],
plane_a_next_position := [panx, pany], plane_b_next_position := [pbnx, pbny]
if (communicateable = True)
    if (plane_a_next_position = plane_b_next_position)
        then {collision_detect := True}
    if (plane_a_next_position = plane_b_cur_position)
        then {collision_detect := True}
    if (plane_a_cur_position = plane_b_next_position)
        then {collision_detect := True}
```

Figure 3: Compiled version of the *collision*

```
communicate: pax, pay, pbx ,pby → communicateable
List[int, int] plane_a_cur_position := [pax, pay], plane_b_cur_position := [pbx, pby],
bool communicateable := False
if (abs(pax - pbx) <= 4 | abs(pay - pby) <= 4)
        then {communicateable := True}
else
        then {communicateable := False}
```

Figure 4: Compiled version of the *communicate*

2.4 Turn Right

The function `turn_right` operates for the right turn of the aircraft. It will get the plane's next step direction and then change it to the corresponding direction.

```
turn_right: next_step_direction_indicator_copy →  next_step_direction_indicator
string next_step_direction_indicator_copy :=none,  next_step_direction_indicator := none
next_step_direction_indicator :=  next_step_direction_indicator_copy
if (next_step_direction_indicator_copy := down)
        then {next_step_direction_indicator := left}
if (next_step_direction_indicator_copy := up)
        then {next_step_direction_indicator := right}
if (next_step_direction_indicator_copy := left)
        then {next_step_direction_indicator := up}
if (next_step_direction_indicator_copy := right)
        then {next_step_direction_indicator := down}
```

Figure 5: Compiled version of the *turn_right*

2.5 Turn Right and Go Forward

My `turn_goforward` function combined both function `turn_right` and function `flight_step_1`. It will make a right turn and take a step forward. This function aims to test whether a right turn will result in a backward movement.

```
turn_goforward: next_step_direction_indicator, x1, y1 → x3, y3
List[int, int] plane_cur_position := [x1, y1], plane_copy_cur_position := [x3, y3]
string next_step_direction_indicator :=none,
x3 := x1
y3 := y1

if (next_step_direction_indicator := down)
        then {y3 := y3 - 1}
if (next_step_direction_indicator := up)
        then {y3 := y3 + 1}
if (next_step_direction_indicator := left)
        then {x3 := x3 - 1}
if (next_step_direction_indicator := right)
        then {x3 := x3 + 1}
```

Figure 6: Compiled version of the *turn_goforward*

flight_step_2: x1, y1, x2, y2, x3, y3, x4, y4, bx, by, bnx, bny, history_path, communicateable → x3, y3

List[int, int] plane_cur_position := [x1,y1], plane_destination := [x2,y2], plane_copy_cur_position := [x3, y3], plane_b_cur_position := [bx, by], plane_b_next_position := [bnx, bny], compel_param = [x4, y4],
List[[int, int], [int, int]] history_path := $[[x_1, y_1],[x_2, y_2],\ldots,[x_n, y_n]]$
string next_step_direction_indicator := none

```
if (plane_cur_position ! ⊥)
        then {x3 := x4, y3 := y4}
else if (plane_cur_position = plane_destination)
        then {x3 := x1, y3 := y1}
else if (x3 < x2)
        then {x3 := x3 +1}
else if (x3 > x2)
        then {x3 := x3 - 1}
else if (y3 < y2)
        then {y3 := y3 + 1}
else if (y3 > y2)
        then {y3 := y3- 1}

While(((len(history_path) >= 2) & plane_cur_position = history_path[-2])) |
(communicateable = True & plane_cur_position = plane_b_next_position) |
(communicateable = True & ((plane_cur_position = plane_b_cur_position) &
(plane_cur_position = plane_b_next_position))))
        if (x1 = x3)
                if (y1 > y3)
                        then {next_step_direction_indicator := down}
                if (y1 < y3)
                        then {next_step_direction_indicator := up}
        if (y1 = y3)
                if (x1 > x3)
                        then {next_step_direction_indicator := left}
                if (x1 < x3)
                        then {next_step_direction_indicator := right}
```

```
if (next_step_direction_indicator = down)
        then {next_step_direction_indicator := left}
if (next_step_direction_indicator = up)
        then {next_step_direction_indicator :=  right}
if (next_step_direction_indicator = left)
        then {next_step_direction_indicator := up}
if (next_step_direction_indicator = right)
        then {next_step_direction_indicator := down}



if (next_step_direction_indicator = down)
        then {y3 := y3 - 1}
if (next_step_direction_indicator = up)
        then {y3 := y3 + 1}
if (next_step_direction_indicator = left)
        then {x3 := x3 - 1}
if (next_step_direction_indicator = right)
        then {x3 := x3 + 1}
```

Figure 7: Compiled version of the *flight_step_2*

3 Overall Logic

If both aircraft A and B do not arrive at the destination, we need to judge whether they are in the communication area. Then aircraft A and aircraft B move one step and then calculate the coordinates of the next movement after the movement. Then, collision detection is performed. If a collision occurs, aircraft A turns to the right. After that, calculate the next step coordinates after aircraft A turns to the right. If the coordinates of the next step are the same as the coordinates of the previous step, then turn right again and go forward one step. Because at this point, the aircraft has no possible go backward, and no collision will occur. So set the compel parameter to the calculated coordinates. Pass the force parameter into the `flight_step_2` function, which will return this coordinate directly as the new calculated coordinate of aircraft A.

4 Conclusions and Further Work

This algorithm cannot guarantee the calculated path is the shortest, but it will not be much longer than the shortest path. However, the computing speed and efficiency of this system are very high. The basic optimization way is to allow aircraft A to turn both right and left. So that the aircraft can avoid rounding path, no rounding path can make the path shorter. A kind of advanced optimization is to use BFS to do a path search, which searches all possible paths for the aircraft to avoid the collision, and then finds the shortest path.

5 Tests

Here are some tests as a monitor to prove my system fulfills the safety requirement.

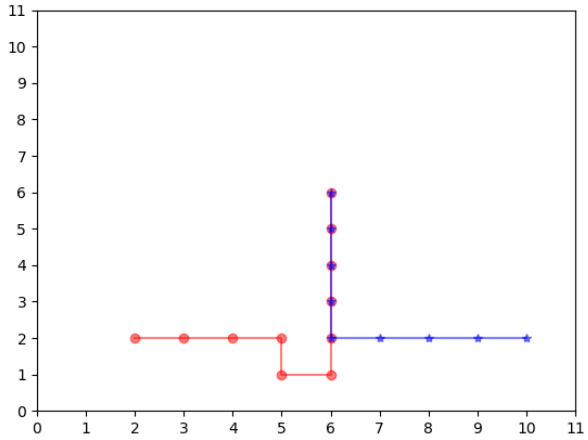

Figure 8: Aircraft A initial coordinate: [2,2], destination coordinate[6,6]. Aircraft B initial coordinate: [10,10], destination coordinate[6,6].

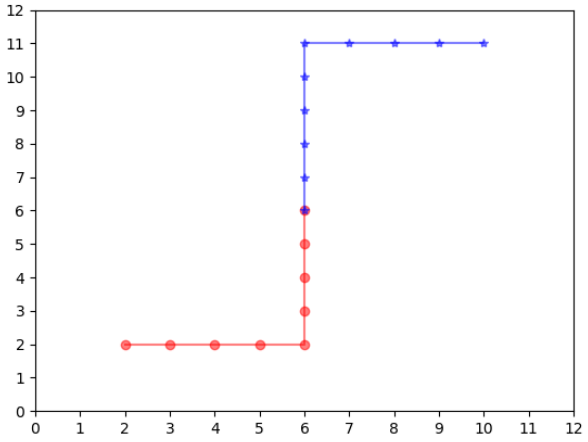

Figure 9: Aircraft A initial coordinate: [2,2], destination coordinate[6,6]. Aircraft B initial coordinate: [11,11], destination coordinate[6,6].

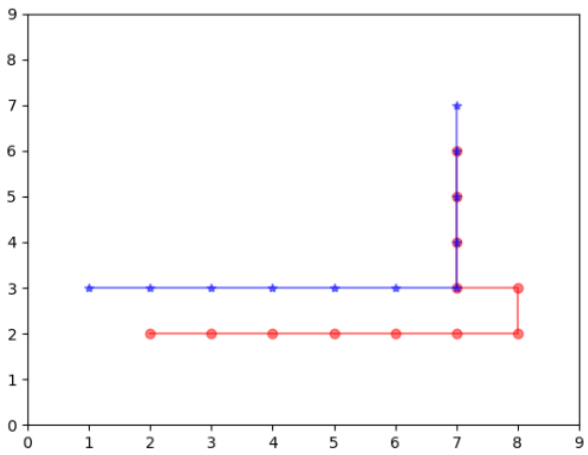

Figure 10: Aircraft A initial coordinate: [2,2], destination coordinate[7,6]. Aircraft B initial coordinate: [1,3], destination coordinate[7,7].

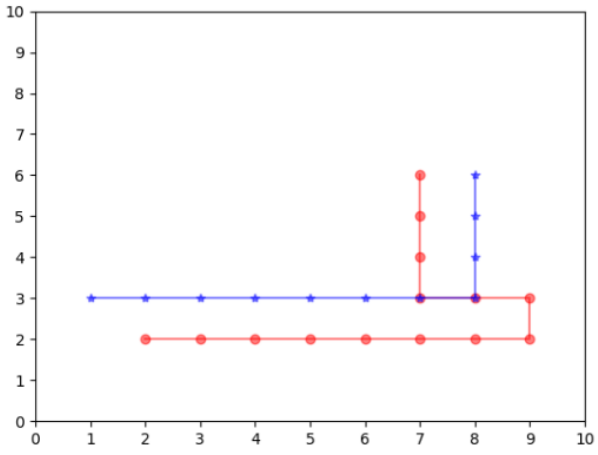

Figure 11: Aircraft A initial coordinate: [2,2], destination coordinate[7,6]. Aircraft B initial coordinate: [1,3], destination coordinate[8,6].

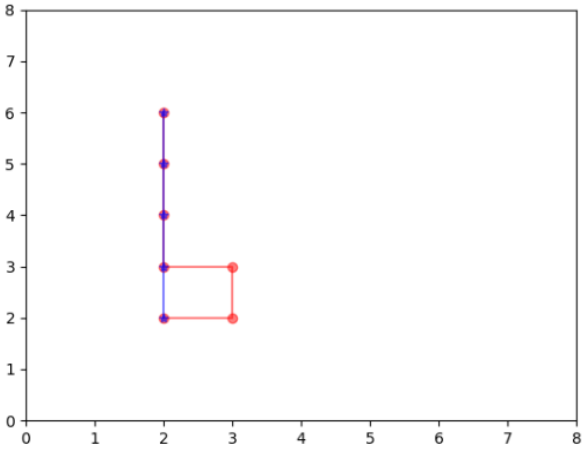

Figure 12: Aircraft A initial coordinate: [2,2], destination coordinate[2,6]. Aircraft B initial coordinate: [2,2], destination coordinate[2,6].

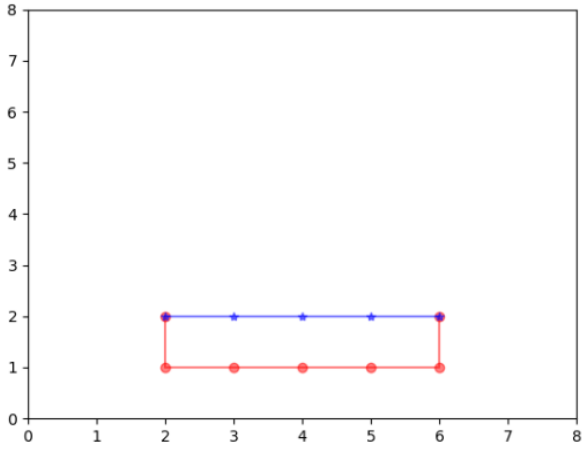

Figure 13: Aircraft A initial coordinate: [2,2], destination coordinate[6,2]. Aircraft B initial coordinate: [2,2], destination coordinate[6,2].