CS-Note* 28

Tommy Ingulfsen
August 16, 2002

Pushing Tin Project - Algorithms

1 Introduction

In this document we try to explain how the Pushing Tin program actually works. The core algorithms are elaborated in detail, and weaknesses and assumptions are pointed out.

Many procedures have a strong mathematical flavour, such as Proximity.Min_Distance, which finds the shortest distance between two aircraft within a certain time interval. These procedures are not easy to comment in code, and therefore a lot of space is devoted to them below.

The intention is that this document should give a reader that is familiar with SPARK and Ada a deep insight in the system, and therefore we get quite detailed in the descriptions. It will probably be easier to follow the explanations here if a copy of the code is readily available.

Section 2 is a listing of all the files and packages in the system. Section 3 concerns the actual operation of the program, explaining the algorithms in use. The Generator, the SPARK subsystem, the Display, the Scheduler and the OpenGL graphics display are explained in detail. Section 4 is a discussion of various questions that arose from the design and implementation of the application. Section 5 lays out a few of the possibilities for extending the program and section 6 explains briefly how the system can be tweaked easily.

2 File and package listing

There are 25 files of source code in the system. These files belong to the SPARK Short-Term Conflict Alert subsystem:

File Name	Type	Description
globals.ads	SPARK	global constants
line_utils.ads	SPARK	def of a line
square_package.ads	SPARK	spec of square root implementation
square_package.adb	SPARK	square root implementation
points.ads	SPARK	data structure for a point
radar_view.ads	SPARK	spec of SPARK boundary package
radar_view.adb	SPARK	SPARK boundary implementation
situation.ads	SPARK	spec of data struct to hold previous points and derive lines
situation.adb	SPARK	implementation of Situation package
results.ads	SPARK	spec of data struct to hold STCA results
results.adb	SPARK	implementation of Results package
proximity.ads	SPARK	spec of STCA proximity check
proximity.adb	SPARK	implementation of proximity check
spark_subsys.ads	SPARK	spec of STCA “main” subprogram
spark_subsys.adb	SPARK	implementation of STCA “main” subprogram

The non safety-critical code was written in ordinary Ada95. With reference to the design drafts, this applies to the Generator, the Scheduler and the Display in addition to a couple of the data structures used.

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

File Name	Type	Description
stca.adb	Ada95/OpenGL	Main OpenGL program
push_tin.ads	Ada95/OpenGL	spec of the Scheduler and its graphics display
push_tin.adb	Ada95/OpenGL	Scheduler and graphics implementation
generate.ads	Ada95	spec of the Generator
generate.adb	Ada95	implementation of the Generator
aircrafts.ads	Ada95	spec of data types to control aircraft
airport.ads	Ada95	spec of data structure to control entire radar
airport.adb	Ada95	implementation of Airport package
display.ads	Ada95	spec of the Display
display.adb	Ada95	implementation of the display

3 Algorithms

Following is a description of all the major algorithms in the application. Some of the algorithms are so simple and straightforward that they are not described here. In addition to a low-level view of the various procedures, some aspects of Ada and SPARK code are discussed where appropriate.

3.1 The Generator

The Generator (files "generate.ads" and "generate.adb") relies heavily on Ada's random number generators. The Airport package, defined in "airport.ads" and "airport.adb" is also part of the Generator.

3.1.1 The Radar System

As laid out in the design drafts, the flight paths are organised in an "X" with four entry or exit points known as VORs. In order to control the aircraft, each VOR is given a set of co-ordinates, and together all the VORs define a co-ordinate system for the entire program:

VOR number	X range	Y range	Z
1	0 – 4000	500 – 5500	0
2	20000 – 24000	500 – 5500	0
3	20000 – 24000	500 – 5500	66667
4	0 – 4000	500 – 5500	0

These values have been chosen to make the paths dangerous, by making them quite narrow, and to ensure a "nice" speed through the system. The co-ordinates are explained in figure 2.

Aircraft may enter the system at any VOR, and will leave at the diagonally opposite VOR. So, for example, a plane starting its journey through our simulated radar at VOR 2 will end it at VOR 4. See figure 1.

The range in Z, which specifies the "depth", is 66667 meters. All aircraft travel at the same, constant speed. There is no "speed" parameter in the system, but on update all aircraft are moved a distance that roughly corresponds to 800 km/h (about 500 mph). At that speed, an aircraft would need about 5 minutes to complete a trip through the simulated radar system.

3.1.2 Airport holds all the aircraft

While programming one realised that a new package was needed to hold all the aircraft within the radar's view in some structure. Hence the Airport package was created.

Since all the aircraft travel at constant speed and are only introduced one at a time, a queue could have been successfully implemented to handle planes. However, if the system was to be updated some time in the future with different speeds for aircraft, a queue would no longer be able to handle the requirements for aircraft leaving and entering the radar's view due to the inherent properties of queues.

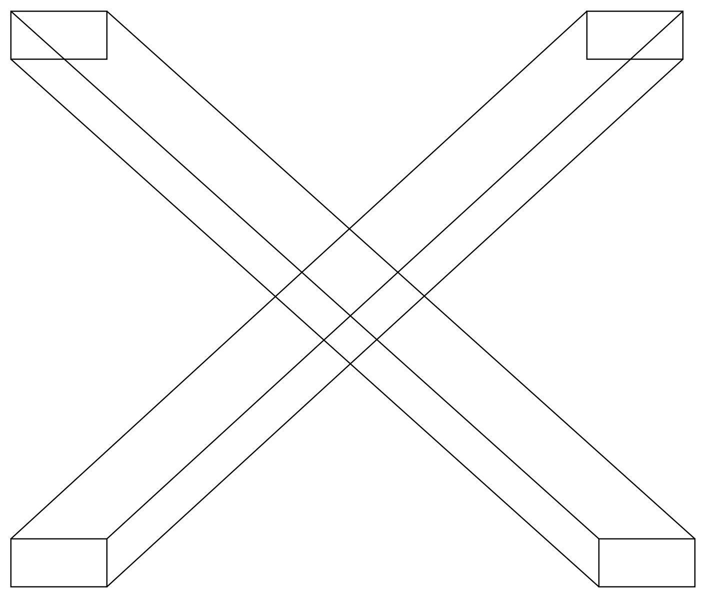


Figure 1: Airways

A simple array is used with a method to add and a method to remove aircraft. Adding simply involves appending the new plane to the end of the array until there are 100 aircraft in the system (the maximum number of aircraft is limited in "globals.ads"). When an aircraft is removed its index in the array is noted, and all aircraft above that index are shifted down in order to preserve the ordering. Moving aircraft forward is accomplished using vector mathematics such as:

```
New_X_Position := Old_X_Position + Speed * X_Direction;
```

In this part of the application, removal of aircraft that have passed the limits of the radar (Z coordinates greater than 66667 or less than 0) is also carried out. Information about this deletion is stored in an array that is made accessible to the Display.

3.1.3 Operation

The "main" Generator subprogram is known as "Main_Generate" and this is the only procedure called explicitly by the Scheduler to advance the system. `Main_Generate` uses several subprograms to carry out its tasks, which include introducing new aircraft, moving aircraft forward, changing altitude of aircraft. This is the pseudo code for `Main_Generate`:

1. With probability 0.142857 introduce 1 new aircraft using `New_Aircraft`.
2. For all aircraft, change altitude with probability 0.10 using `Change_Altitude` (note that this means that the y-direction is changed, but the altitude itself does not change instantaneously).
3. Move all aircraft one step forward using `Airport.Update_System`.

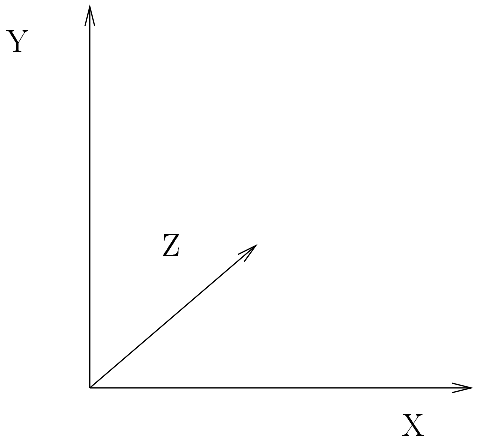


Figure 2: Co-ordinate System

The probability constants are defined in "generate.ads".

The most complicated procedure in this part of the program is `New_Aircraft`. When a new plane is generated, it must be assigned a starting VOR, a direction, a starting point within the VOR and a callsign.

The assignments are done using the 2 Ada random number generators, `Ada.Numerics.Float_Random` and `Ada.Numerics.Discrete_Random`. The latter is used for integers. Starting VOR and starting points are chosen randomly. By choosing a random point at the opposite VOR, the aircraft is assigned a direction in the form of a vector:

```
X_Direction := End_Position.X - Start_Position.X
```

And likewise for the other dimensions.

3.1.4 Possible extensions and dead ends

To make the entire program more advanced and realistic, aircraft could have their headings changed in the X-dimension as well so that they would no longer move linearly.

While this would be easy to implement in the Generator, considerably more difficulty would be involved when the Situation package (see below) was updated correspondingly. The mathematics becomes complicated when the lines are no longer straight.

NOTE: While programming, there was considerable difficulty in making Ada's random number libraries work properly. There were two reasons for this.

1. **Only a single integer range may be used within any one scope.** In the `New_Aircraft` procedure it is desirable to work with 5 ranges, one for each VOR and 1 for choosing the start VOR. However, since a separate package must be declared to use `Ada.Numerics.Discrete_Random` this is not possible. One has to resort to using floating point random numbers and convert them into integers appropriately.

2. **Timing problems.** At first, the Generator was tested on its own, executing once at a time. But when a loop with a time delay was introduced (see "push_tin.adb"), all sorts of odd problems arose. In the end it was discovered that these difficulties were related to the initialisation of the random number generators. For each library there is a generator which must be initialised, in order

for the resulting number sequence to have good random properties. To ensure correct behaviour, the integer random number generator in `New_Aircraft` is initialised using a random number from `Main_Generate`. There are two initialisation procedures for this purpose, and the one that is used for `Ada.Numerics.Discrete_Random` in `New_Aircraft` is known as the *time independent Reset procedure*.

3.2 The SPARK subsystem

The safety-critical part of the Pushing Tin program was written in SPARK. Briefly, its requirements were to:

- Translate the radar system "snapshot" into a more implementation-independent form (this refers to the implementation of the `Generator`).
- Without using all the information from the Generator, predict how the aircraft would move in relation to each other in the future.
- Detect dangerous flight paths.
- Give a response to the user to indicate potential hazards.

3.2.1 The SPARK boundary

Integrating the SPARK subsystem with the rest of the program was achieved by using two boundary packages, `Radar_View` and `Results`.

`Radar_View` is a simple package that lets the Scheduler copy the current state of the radar from `Airport` into a rather simpler form for use by the Short-Term Collision Alert subsystem (STCA). This is performed after each update by the Generator using the procedure `Radar_View.New_Update`, which simply copies all the aircraft positions from `Airport`. The STCA uses `Radar_View.Get_New_Point` and `Radar_View.Get_Size` to read this information.

`Results` is rather more complicated. In order to hold all the information needed for the Display, it uses an array of a record. The single global variable used to hold this information is called `Hazardous` and its type is `Spark_Results`. `Hazardous` has two parts to it, an array of an array that holds the result itself and an integer representing its size.

For the two-dimensional array, some conventions are used.

- The first dimension holds the "first" aircraft. That is, when the Display prints something like "Danger of collision between X Y Z and ..." then the first set of co-ordinates refers to the first dimension in the array.
- In the worst case, every aircraft could potentially be on collision course with every other. Hence the size of the second dimension equals that of the first dimension and is set to the upper bound on aircraft (set to 100 in package Globals).
- To hold how many aircraft the one in the first dimension can collide with, a size is associated with each first dimension.
- In order to refer to the "first" aircraft, a copy is stored in the first element of the second dimension.
- There are two categories of hazards, `Immediate` and `Future`. `Immediate` means that the distance between two aircraft is already too small and `Future` indicates that the distance will get too small in the future if the aircraft continue on the same heading.

To understand this, consider figure 3, where a situation in which 2 conflict sets is shown. According to the `Hazardous` array, there is a danger of collision between BA1441 and KL7321 and between SA3390 and LF8891. The first element in the first row holds the "first" aircraft, in this case BA1441 and SA3390.

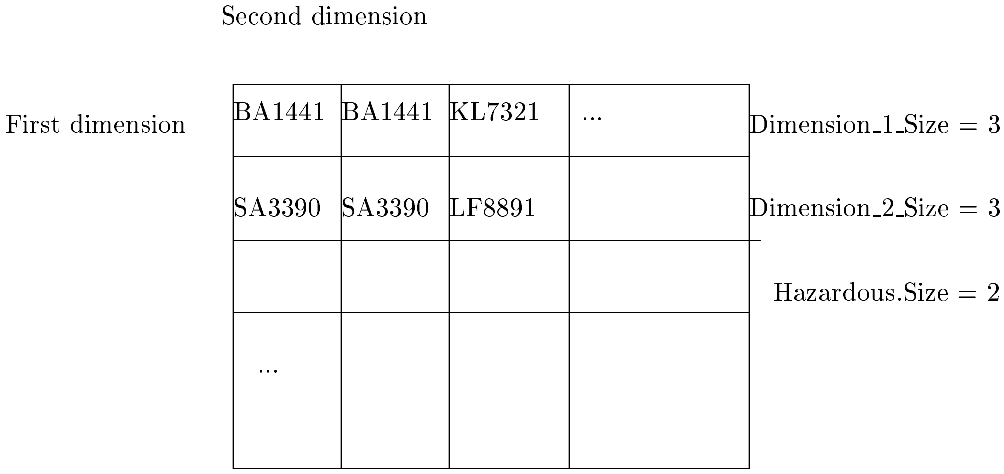


Figure 3: The Hazardous array in package Results

Then, to be able to refer to the "first" aircraft, it is stored as the second element in the corresponding row as well.

Most of the complicated processing is within `Results.New_Update`. This procedure is called by the proximity checker each time a (potential) hazard is detected. The Display calls `Results.Read` in order to get the proximity check results.

3.2.2 The "main" SPARK subprogram

Since the STCA consists of many different procedures and functions, it would be too complicated to have the Scheduler call them all to obtain the results for output. So, the Scheduler only calls `Radar_View.New_Update` and then hands over control to `Spark_Subsys.Spark_Main`.

`Spark_Subsys.Spark_Main` is the "main" procedure of the STCA. It is a brief subprogram that calls `Situation.Update_Situation` to obtain a fresh radar image with predicted headings and then performs the proximity check by calling `Proximity.Min_Distance`. The two last procedures are explained below.

3.2.3 Deriving the future radar situation

The Generator holds all the information on aircraft, including the headings and their callsigns. It would be very easy to use this in a proximity check. The ambitions for the Pushing Tin program were a bit higher than that however, and so the package `Situation` was created to solve this. `Situation` is probably the most complex part of the entire application.

The core is `Situation.Update_Situation`, a procedure that is called by the main SPARK program (package `Spark_Subsys`). Here, a global variable called `Situation.Situation_Now` is updated to hold the current positions of all aircraft together with a corresponding line that represent their derived heading. Figure 4 is a picture of how an aircraft is represented with a current position and a heading in vector form.

The key to `Situation` is that the previous radar snapshot is stored. By comparing the previous with the current situation (as defined in `Radar_View`) the directions of aircraft can be obtained. See figure 5.

The calculations in this part of the program are based upon the assumption that aircraft move linearly. The pseudo algorithm for `Situation.Update_Situation` is

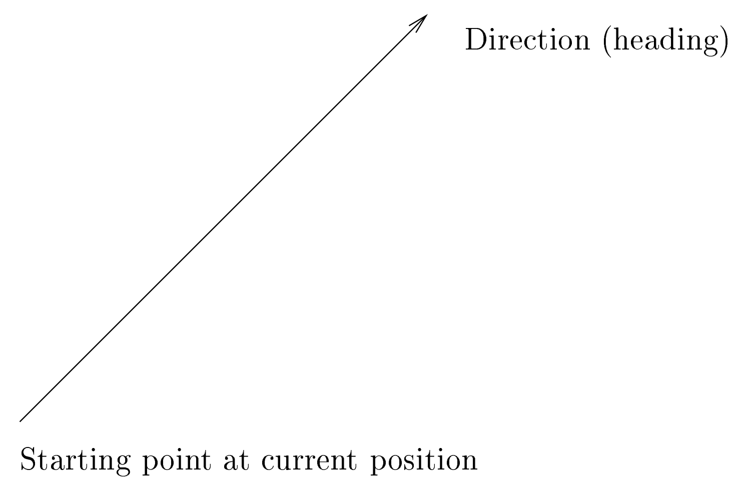
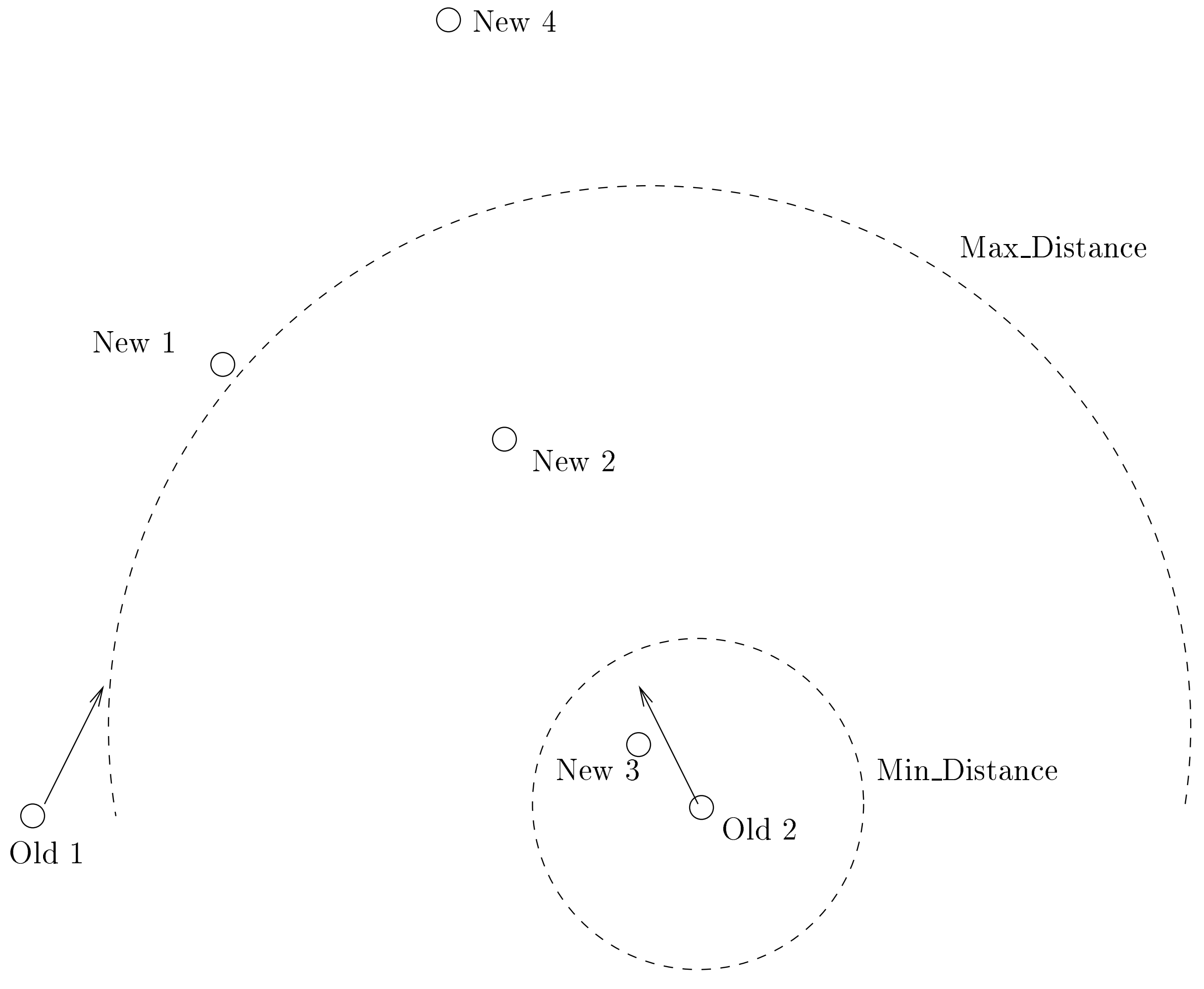


Figure 4: Co-ordinate System



When mapping Old 2 to a new position, New 3, New 1 and New 4 will be disregarded. No positions within Min_Distance or out width Max_Distance are taken into account.

Figure 5: Deriving the next position

1. Remove any aircraft that have moved out of radar range (this is based upon their previous positions and headings).
2. Append any new aircraft on to the end of the array within `Situation.Situation_Now`.
3. Set all elements of the `Radar_View.New_Points` array and the `Situation.Situation_Now` array to not reserved and not updated respectively.
4. If there are any aircraft in the system then

```
for all aircraft in  Situation
  for all aircraft in Radar_View

    if current pt in Radar_View is not reserved and
      previous pt in Situation is not yet updated then

      if the distance betw. the 2 pts is <= Max_Distance and
                                         >= Min_Distance then

        if previous pt in Situation has an assigned direction then
           if the current pt in Radar_View is on that direction then

             previous pt in Situation := current pt in Radar_View
             current pt in Radar_View := reserved
             previous pt in Situation := updated

           end if

           else if no direction yet
             previous pt in Situation := current pt in Radar_View
             give previous pt in Situation the corresponding direction
             current pt in Radar_View := reserved
             previous pt in Situation := updated

        end if
      end if
    end if
  end loop
end loop
```

5. Remove all aircraft that have not been updated.

`Radar_View.New_Array` holds the new aircraft positions, while `Situation.Situation_Now` holds the old positions. A new position can only be assigned to 1 aircraft (unless there has been a mid-air collision!) and an old position should only be updated once. So positions in `Radar_View.New_Array` are initialised to `not reserved` and positions in `Situation.Situation_Now` are initialised to `not yet updated`.

3.2.4 The proximity check

The core of the STCA system is the check for whether two aircraft are breaching, or will breach, a safe separation criteria. This proximity check is located in the package `Proximity`, whose "main" procedure is called `Proximity.Min_Distance`. The check is carried out in a highly mathematical manner, and the

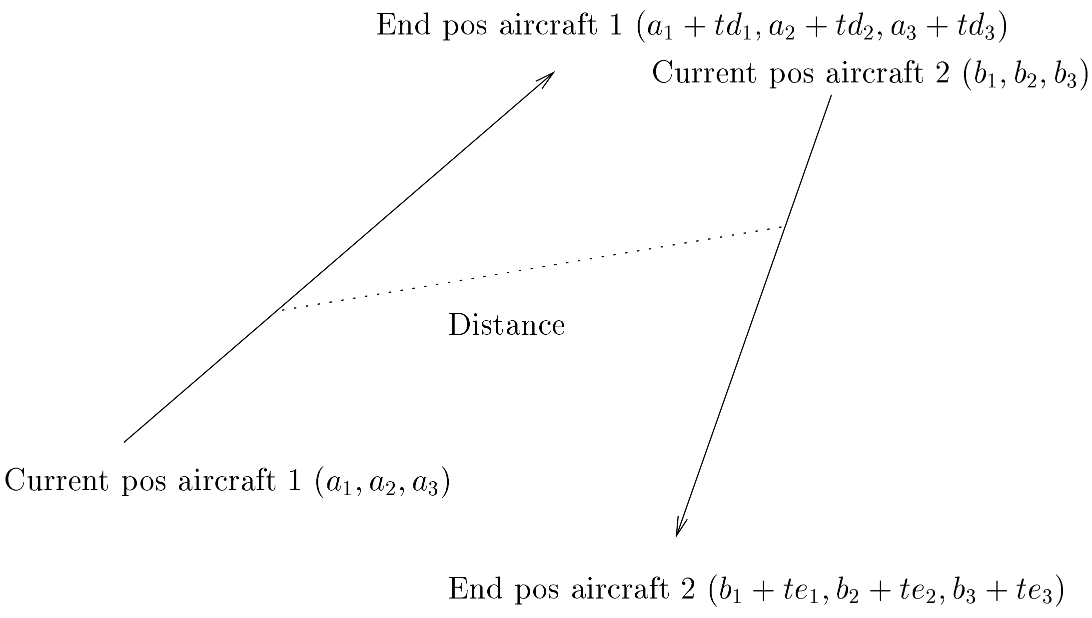


Figure 6: Proximity calculation

code for the subprograms within `Proximity` will be very hard to understand for someone who is not familiar with the algorithm.

Vector algebra is used to calculate the minimum distance between two aircraft in the future. Figure 6 gives a picture of how the computations are accomplished. The notation for the vector algebra is this:

1. At any one time we consider 2 aircraft, whose vectors are given by $\vec{X}$and $\vec{Y}$
2. $\vec{X} = (a_1 + td_1, a_2 + td_2, a_3 + td_3$ and $\vec{Y} = (b_1 + te_1, b_2 + te_2, b_3 + te_3)$ where
   a_1, a_2, a_3 is the starting point of $\vec{X}$ and
   b_1, b_2, b_3 is the starting point of $\vec{Y}$.
   d_1, d_2, d_3 is the direction of $\vec{X}$ and
   e_1, e_2, e_3 is the direction of $\vec{Y}$.
   t is the velocity vector.
3. The subscripts denote dimensions x, y and z respectively, so for example a_1 is the x-component of the first aircraft.

Two aircraft are moving linearly in almost opposite directions. The tasks that `Proximity.Min_Distance` has to carry out are

1. Check if the aircraft are breaching the safe separation criteria already.
2. Check if the aircraft will breach the safe separation criteria in the future if they carry on at the same heading.
3. Update `Result` if a hazard has been detected.

`Proximity.Min_Distance` is a general procedure to find the minimum distance between two line segments, and therefore takes two lines as arguments only. Pythagoras is used extensively to determine the distance between two points, as shown in figure 7.

An aircraft is represented in vector form in three dimensions by e.g. X_1, X_2, X_3. Each dimension is made up of the starting point a_n and the direction in that dimension multiplied by the time td_1. By

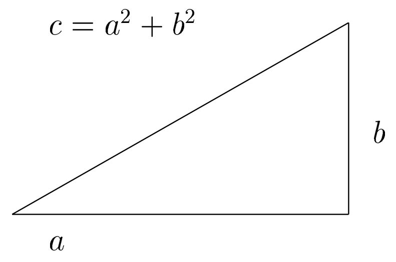


Figure 7: Distance between two points

introducing the time parameter t, and extending the theorem of Pythagoras to three dimensions, we obtain the equation that describes the distance between two points (aircraft in this case) as they move along a straight line:

$$\text{Distance} = \sqrt{(X_1 - Y_1) + (X_2 - Y_2) + (X_3 - Y_3)} \tag{1}$$

This is verified by any standard mathematics textbook for undergraduate engineering students.

The minimum distance is found from (1) when the time parameter t is at its smallest. In other words, we must differentiate (1). The expression derived by differentiation is

$$t = -\frac{a_1d_1 - a_1e_1 - b_1d_1 + b_1e_1 + a_2d_2 - a_2e_2 - b_2d_2 + b_2e_2 + a_3d_3 - a_3e_3 - b_3d_3 + b_3e_3}{d_1^2 - 2d_1e_1 + e_1^2 + d_2^2 - 2d_2e_2 + e_2^2 + d_3^2 - 2d_3e_3 + e_3^2} \tag{2}$$

So by evaluating (2) and substituting the result into (1) we will get the minimum distance within the next time interval.

The two constants in `Proximity.Min_Distance`, `Min_Separation_Now` and `Min_Separation_Future` define what the separation criteria are. These two constants are set to 1610 and 3220 respectively. If an aircraft is travelling at 800km/h (500mph) then it will need 14.5 seconds to travel 3220 meters. The reason for providing different values for the current and future separation was to discriminate better between the two types of hazards.

With the above explanation in mind, we arrive at the following pseudo algorithm for `Proximity.Min_Distance`:

1. `Result := False`.
2. If the two lines passed in are identical, `Result := False` (should two aircraft start at exactly the same point in space, it is probably too late to issue a warning anyway!).
3. If the current distance is too small, `Result := True`.
4. If the directions are parallel, the `Result` will have been set by point 2 above (in this case only the current distance between two points determines whether or not the situation is hazardous).
5. If none of the preceding computations have set `Result = True` then find the time in the future that gives the shortest distance between the aircraft, by evaluating (2).
6. If the time from above is in the correct interval then substitute the time found at point 5 into (1).
7. If the result of point 6 is less than `Min_Separation_Future` then `Result := True`.

The procedure `Spark_Subsys.Spark_Main` detects what the value of `Result` is and updates the boundary package `Result` accordingly.

3.3 The Display

The `Display` package is very simple and consists of a single procedure `New_Display` that outputs information to the screen. This information includes

- New aircraft that have been added to the system by the Generator.
- Aircraft that have been removed from the radar system by `Airport.Update_System`.
- Aircraft in danger of colliding with each other.

3.4 The Scheduler

The Scheduler is the main Ada program of the application, and is to be found in `push_tin.adb`. It does not carry out any processing of its own, but merely calls other procedures. Observe that it is in this part of the system that the execution "speed" is set.

Briefly the `Pushingtin` procedure performs these operations:

1. Initialise the radar data structure (defined in Airport).
2. do
   (a) Call the Generator to update the system.
   (b) Wait 5 seconds.
   (c) Update the SPARK boundary using `Radar_View.New_Update`.
   (d) Call `Spark_Subsys.Spark_Main` to perform the proximity check.
   (e) Call the Display to show the results of the current loop iteration.
   (f) Call the OpenGL libraries to update the graphical display
3. 80 times

And that is really all the `Pushingtin` procedure does.

3.5 The graphical display

The graphical display code is written in OpenGL, and can be found in the files "stca.adb" and "push_tin.adb". Its operation is simple:

1. Initialise the OpenGL libraries, set up the window and lighting.
2. Set the colour to green and draw the "X"-shaped air routes.
3. Draw all the aircraft in the `Airport.Rs`-array as spheres.
4. Draw all the aircraft in the `Results.Hazardous`-array as red spheres (if the alert is `Immediate`) or as purple spheres (if the alert is `Future`).
5. Reset the colour to green again.

It really is the procedure `Stca` that is the "main" subprogram. It declares `Pushingtin` as the `glutIdleFunc` and `Push_Tin.Update_Graphics` as the `glutDisplayFunc`.

4 Discussion

During the design phase the Informed method developed by Praxis Critical Systems was used. The package structure that was arrived at (see the Final Design Document) is a result of applying Informed to our requirements and specifications.

There is one property of the design that we were uncertain about. The packages `Situation` and `Proximity` seem to reside out width the SPARK boundary, hence being (potentially) visible to the rest of the application. By including a `with Situation` in for example`airport.adb` one would gain access to `Situation`. At first sight, this appears to be a significant weakness of the design, since presumably all safety-critical features should be hidden from the ordinary Ada packages. However, we found that the code would become unnecessarily complex if we let `Situation` and `Proximity` be child packages of the SPARK "core" package `Spark_Subsys`, and so we ended up with separate packages instead.

It is an open question if this is bad practise or not. In all the documentation we have seen on SPARK design, there have never been any details about how to interface a SPARK program to a program written in another programming language. We perceive that in most cases, it will not be feasible to write the entire system in SPARK, but rather use it for the most safety-critical parts. It would therefore be beneficial if an answer could be found in the future.

4.1 Strengths and weaknesses

The application was designed and coded within 8 weeks, and so is obviously quite simple. The most important simplifications include

- All aircraft travel at the same speed.
- Speed is constant.
- Movement of aircraft is linear.
- The proximity checker (see `Proximity.Min_Distance`) does not include a check for aircraft at different flight levels, but simply measures the linear distance between aircraft. This is one of the problems that real-world systems have to address. When many planes are gathered in a holding stack, care has to be taken to ensure that too many alerts are avoided. This is because there is a difference between `vertical` and `horizontal` separation in the Air Traffic Controller's world, the latter being larger than the former. But there is no concept of vertical or horizontal separation in the Pushing Tin system.

Obvious strengths are:

- The modular design should make changes to the system localised, so extensions will be relatively easy to implement.
- Since the proximity check is isolated within a single package, a new algorithm could easily be accommodated simply by writing a new package, provided the new algorithm is also based upon vector mathematics (most are).
- Despite its simplicity, the program is a realistic example of a Short-Term Collision Alert system.

5 Future work

5.1 Graphical Interface

At the moment, the interface to the program consists of a command line display and a graphical display.

The Display module writes output to the command line such as messages concerning aircraft about to collide. The graphics window is a visualisation of the aircraft. Each aircraft is shown as a sphere

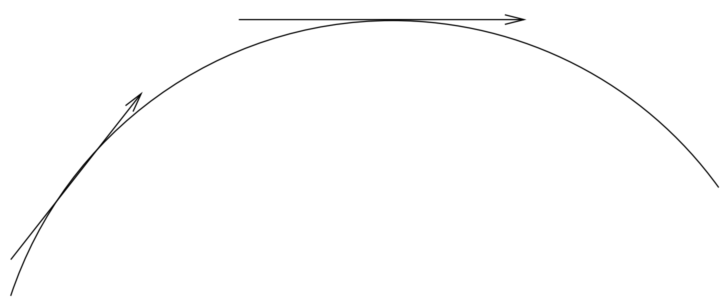


Figure 8: Using straight lines to represent non-linear movement

within the big “X” that limits flight paths. The colour of the sphere changes depending on whether or not the planes might collide in the future. Green is safe, purple means that the aircraft might collide in the future (`Future warning`) and red is an `Immediate` warning, for aircraft that are presently too close to each other.

A possible extension would be a Graphical User Interface (GUI). This could be a graphical interface displaying how aircraft move around in space, perhaps coupled with a simple control interface with buttons. For example, it might be nice to have a button to “freeze” the simulation and then resume it or the ability to change the delays in the main loop. Ultimately, the application has the potential of a game, where the user could act as an Air Traffic Controller by issuing orders to aircraft.

5.2 Alternative algorithms

It may seem restrictive to base the system upon the mathematics of straight lines, but as figure 8 shows, a straight line can be used for determining the headings of aircraft flying along curves as well.

An alternative algorithm that was considered in the early design stages is based upon intersections of three dimensional boxes. Here, each aircraft is assigned a kind of box that represents the airspace into which no other aircraft may intrude within the next time interval. By computing the intersections (if any) between these boxes we will obtain a highly accurate picture of potential hazards.

We can realistically expect an increase in accuracy because the boxes can be constructed so that they take into account more than just simple linear distances. For example, one could leave just a little space behind each aircraft, in the expectation that it is unlikely that any dangerous situations may arise from aircraft moving behind others (unless their directions are parallel of course). Also, the height of the boxes could be changed depending upon the aircraft’s flight level to reflect different vertical separation criteria at different altitudes. This is illustrated in figure 9.

Observe that although this is a different way of performing the test for dangerous flight headings, we will not have to abandon the vector representation of aircraft. It will be enough to implement a new package in place of `Proximity` that have the same procedure `Min_Distance` but perhaps with a slightly different interface.

Unfortunately, implementing an algorithm based upon this theory was found to be too hard and time consuming for this project. A further alternative algorithm, based upon vectors, can be found at http://geometryalgorithms.com/Archive/algorithm_0106/algorithm_0106.htm

Another extension that was considered early in the project was to have different speeds for aircraft. The main consequence of this, with respect to the STCA system, is that the mathematics become somewhat more complex. To cater for several airspeeds, another variable, s, is introduced to represent the time for one of the aircraft. Thus, 1 remains

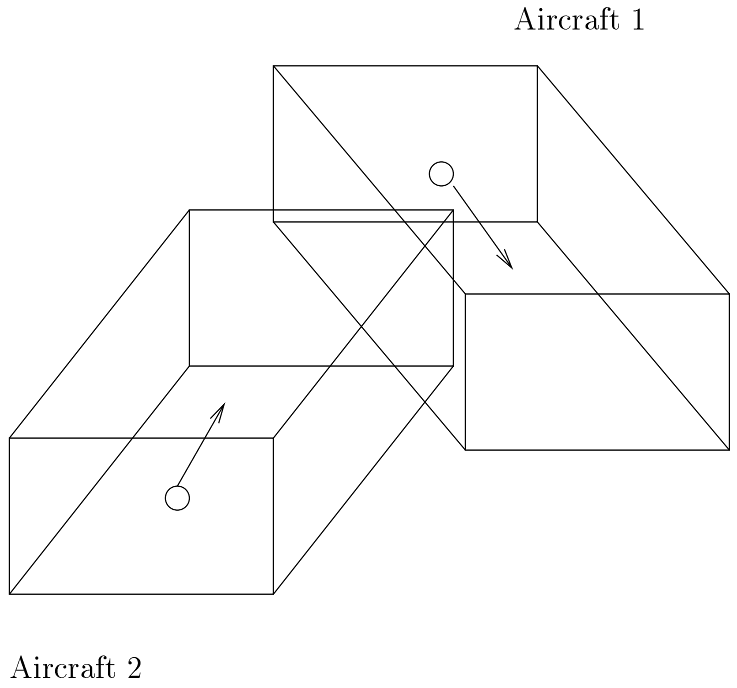


Figure 9: Alternative proximity check algorithm

$$\text{Distance} = \sqrt{(X_1 - Y_1) + (X_2 - Y_2) + (X_3 - Y_3)} \quad (3)$$

but now

$X_1 = a_1 + td_1$ and $Y_1 = a_1 + sd_1$ and so on. This signifies that the planes described by X and Y travel at different speeds.

One further point to conclude this document concerns integer overflow in Ada. It was discovered that the evaluation of 1 in `Proximity.Min_Distance` sometimes gave strange results. This was due to integer overflow, where the numbers got so big that a variable `Distance` got negative. The patch for this is to check for a negative `Distance` and ignore it if it occurs, because in this case the distance between aircraft is so big that it does not matter anyway.

6 Adjusting the system

There are 2 simple ways of tweaking the application's performance to suit different preferences.

Firstly, it is possible to change the delay between each program iteration. This can be done in `Push_Tin.Pushingtin` by changing the number of seconds that `delay` is multiplied by. The `delay` can thus be set to any value, but observe that the program will behave unexpectedly if the `delay` is set to 0. Also, removing the counter `Push_Tin.Pushingtin` in will cause the program to run indefinetely. It should be obvious how to do this when looking at the code.

Secondly, the number and type of warnings can be adjusted by changing the values of variables `Min_Separation_Future` and `Min_Separation_Now`. The larger these values, the greater the number of warnings will be. Setting them equal to each other will cause most hazards to be flagged as belonging to the `Immediate` category.

7 Acknowledgements

I would like to thank

- Andrew Ireland, Heriot-Watt University - the supervisor of the Pushing Tin project and leader of the NuSpade research project.
- Bill James Ellis, Heriot-Watt University - for his endless patience in helping me sort out problems in the code.
- Andrew Wallace, Heriot-Watt University - for putting us on the right track for finding a good proximity check algorithm.
- Paul Turner, Heriot-Watt University - for explaining the mathematics of vectors in 3-D space

since without them I would still have my head stuck into the sand...

8 Conclusion

With the Pushing Tin STCA program we have created a safety-critical simulator concerning air traffic control. The program is intended to be used for proofs and demonstrations as well as being an open source of working SPARK code, of which very little exists at the moment.

By no means is the application perfect, and if anyone would like to modify the source code they are welcome to do so. We would be glad to hear from anyone attempting such modifications.