GDP Draft Report

GDP Group

March 30, 2022

Contents

1 Introduction

The primary goal of this project is to construct a fully autonomous golf caddy. There are caddies with varying ranges of autonomy available on the market alread, from just having a motor so that no pushing from the golfer is required, to being able to automatically following the golfer. There is currently no product available that requires no operation from the golfer after the power up phase. The currently available models that can follow the golfer would follow them into a bunker, onto the green, or onto water if not prevented by a remote control operated by the golfer.

Currently it is against golf association rules for any tools which indicate which club should be used to be employed by golfers. Our sponsor James Siabati indicates that this may change in the near future, so including this ability in our autonomous caddy is a good decision.

2 Project Aim and Objectives

3 Design Criteria

4 Mechanical Design

5 Software

In this section all software required for the operating cycle of the AGC is discussed. The AGC main computer is a Raspberry Pi 4 running a 64-bit Debian port, the same is true for the Raspberry Pi used in the tracking pod. The GPS navigation system requires high precision decimal storage to operate properly. Data type `float32` can store 23 significant bits, compared to data type `float64` which can store 52 significant bits[1]. The calulcations shown below in Fig. (1) show the physical significance of which datatype is chosen.

Only 180° need to be accounted for because sign bit is separate, so to represent 180° the bits required is:

$$\log_2 180 \approx 7.49185 bits$$

This leaves the $n - 7.49185$ significant bits left for sub degree representation, the physical precision in meters we can then derive from each data type of a longitude value at the equator can be calculated as shown in Eqs. () respectively.

For `float32` $n = 23$:

$$\frac{\frac{R_e \cdot \pi}{180}}{2^{(52-\log_2 180)}} \approx 2.386m$$

For `float64` $n = 52$:

$$\frac{\frac{R_e \cdot \pi}{180}}{2^{(52-\log_2 180)}} \approx 4.44nm$$

The radius of Earth. R_e, value used was $6371000m$.

Figure 1: Calculations showing physics precision of single vs double precision floats

The 180° is normalised to a power of two so we are able to use the non integer value of bits in the calculation for physical precision, were we to not perform this normalisation then $7.4918 \rightarrow 8$ full bits are required to represent the degrees and the final precision is slightly less for each datatype. To ensure this calculated precision is representative of our system all GPS coordinates are normalised to a power of two. We can see from the calculations that a significantly higher precision is achieved using double precision floating point data type to store GPS values. The precision yielded by the double precision floating point is unnecessary, however the precision of the single precision floating point is too low as our GPS module is capable of providing more accurate GPS measurements as discussed in section 6. Therefore double precision floating point datatypes will be used to store GPS data, which requires us to use a 64-bit compatible operating system which is why the 64-bit Debian port was chosen. The higher precision achieved comes at the cost of slightly higher compte time for calculations, however as most of the algorithms onboard operate only on small amounts of data, this effect is unnoticable when operating the AGC.

5.1 Control Software

This section discusses the software used to make the AGC follow the golfer. All of the control software apart from the elctronics specific code, such as motor controller communication, was written first using the OpenGL simulator that was built throughout the year. Building and using a simulator allowed us to write and test all the control software / algorithms that would be required to make the AGC follow the golfer as intended while avoiding hazard regions.

All of the control software tested in the simulator was written in C as this was easiest to use with OpenGL, however we decided to change to PyQt5 for designing the onboard GUI. As the ACG software was now to be written in Python the control algorithms were re-written in Python, however upon testing it was found that their performance was significantly slower than the C implementations. This is most likely a large number of array passing and manipulations are required, which is significantly slower in Python where function arguments are passed as object reference instead of pointers in C. Therefore the original C control algorithms were compiled as a dynamic link library (DLL) and the Python "C Types" library was used to load the DLL and create Python wrappers for the control algorithms. This significantly improved the performance as all of the resource intensive calculations were now performed with a compiled C program.

The main control loop that determines the AGC's behaviour is shown below in Fig. (2).

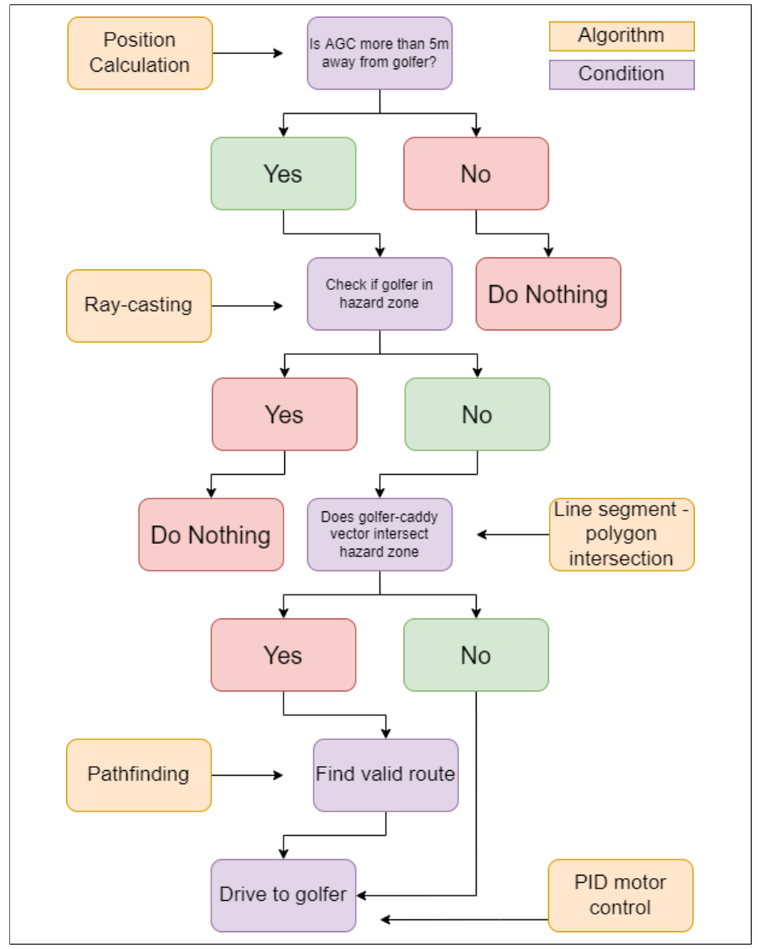

Figure 2: Main AGC control loop

5.1.1 Map Processing

As discussed in the introduction to section 5, all GPS coordinates are normalised to a power of two. We also "pad" the hazard zone coordinates, before normalisation, by a safety factor of two meters, preventing inacurate GPS readings or hazard zone coordinates resulting in the AGC entering a hazard zone. We "pad" the hazard zones by calculating the coordinates of centroid of the polygon, then extending the vector from the centroid to each perimeter point by two meters. The centroid of a polygon can be calculated from the coordinates of it's perimeter points as shown in Eqs. (1 and 2).

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)$$

$$C_x = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i) \quad (1)$$

$$C_y = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i) \quad (2)$$

5.1.2 Ray Casting

The ray casting algorithm is used to determine whether a point falls within a polygon. The algorithm can only accept polygons with straight edges; our polygons are represented by a series of GPS points along their perimeter, and so our polygons are defined by a series of small straight edges. The concept of this algorithm is that given a point you wish to test, if you cast a ray from the point in one direction to infinity, the number of intersections the ray has with the polygon in question indicates whether the point falls in the polygon or not. If an even number of intersections are found, then the point lies outside the polygon, and if an odd number of intersections are found the point lies within the polygon. A visual of this concept can be seen below in Fig. (3).

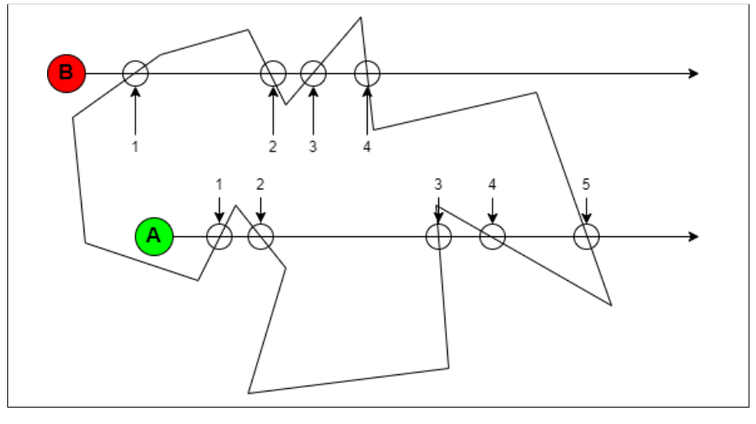

Figure 3: Raycasting Example

The figure shows two points which are to be tested to determine if they lie within the polygon. Casting the ray to the right, towards $x \to \inf$ in our implementation, we can count the intersections each ray has with the polygon. Point A has five intersections and so lies outside the polygon, point B has four intersections and lies within the polygon.

There are additional considerations in our implementation to accout for the ray intersecting a vertex or for the ray being collinear with a polygon edge. If a vertex is intersected then two intersections are counted, one for each edge forming the vertex, and the result is not affected. If the ray is collinear with a polygon edge then only a single intersection is counted, but the ray will undoubtedly also intersect at least one of the vertices on the collinear edge which will be appropriately accounted for. The algorithm has been tested and does not fail in either of these cases.

5.1.3 Segment Polygon Intersection

Before the AGC starts moving towards any target destination, it first checks that it's intended translation vector joining it's location to the target location does not pass through any hazard zones. It does this by checking that it's intended translation vector doesn't intersect any edges of polygons. The diagram shown below in Fig. (4) depicts two finite length line segments intersecting.

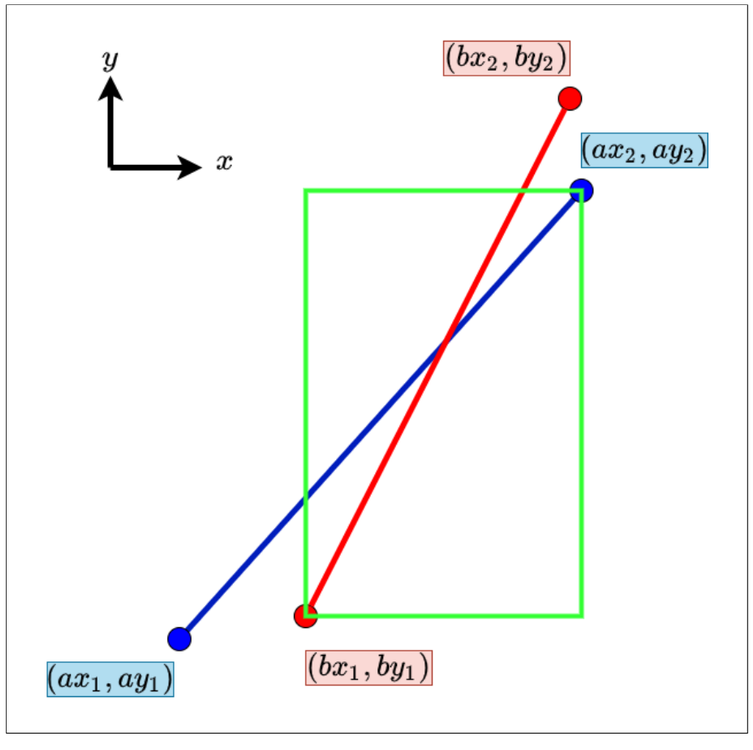

Figure 4: Line Segment Intersection Diagram

The edges of polygons are defined only by the positions of two vertices, as is the AGC intended translation vector. To calculate where the intersection point of these two line segments are a mathematical equation for a line must be constructed for each segment as shown in Eqs. (3 and 4) respectively, then these equations can be solved simultaneously to determine the location of the intersection as shown in Fig. (5).

After the intersection point is calculated we ensure that the intersection point falls within the smallest rectangle bounded by one point from the first and second coordinates of either segment. We do this because the equation for a line is infinite, and any lines with non equal gradient will intersect, but we are only interested if the intersection occurs between points the AGC intends to travel. The bounding rectangle is demonstrated by the green rectangle in Fig. (4).

$$ay_1 - y = \frac{ay_2 - ay_1}{ax_2 - ax_1} \times (x - ax_1) \tag{3}$$

$$by_1 - y = \frac{by_2 - by_1}{bx_2 - bx_1} \times (x - bx_1) \tag{4}$$

In our implementation, before this system is solved, we check if the lines have equal gradient, and if they are collinear or not. If they have equal graident, defined by:

$$\frac{ay_2 - ay_1}{ax_2 - ax_1} = \frac{by_2 - by_1}{bx_2 - bx_1}$$

Then we check if they are collinear, defined by:

$$(x - ax_1) = (x - bx_1)$$

If they are collinear then they intersect, if the have equal gradient but are not collinear then they do not intersect. If one or both of the lines are vertical, $(y_1 = y_2)$ then the system is not solved as below but trivially using the x position of the line.

For simplification:

$$m_a = \frac{ay_2 - ay_1}{ax_2 - ax_1} \qquad m_b = \frac{by_2 - by_1}{bx_2 - bx_1}$$

Now we solve the system defined by Eqs. (3 and 4) to determine the point of intersection resulting in the x coordinate given by Eq. (5) and y coordinate given by Eq. (6).

$$x = \frac{ax_1 m_a - ay_1 - bx_1 m_b + by_1}{m_a - m_b} \tag{5}$$

$$y = \frac{ax_1 m_a m_b - ay_1 m_b - bx_1 m_a m_b + by_1 m_a}{m_a - m_b} \tag{6}$$

Figure 5: Calculation of Line Segment Intersection Point

5.1.4 Pathfinding

The pathfinding algorithm is only used if the control loop shown in Fig. (2) determines that the AGC should move to the golfer but the a direct route is not possible due to the presence of a hazard zone between them. Initially the A* pathfinding algorithm was researched as it is a well known algorithm with which we could choose some heuristic function to minimise, in our case that heuristic function would likely be the shortest distance to reach the golfer. A* is generally used for pathfinding on a discrete graph, we do not have a discrete graph available and so would have to construct one for each golf course or a local graph surrounding the hazard we wish to avoid. This adds unnecessary complexity in terms of implementation and computation to our system which we wish to avoid. Additionally, A* has time complexity of $O(b^d)$ if d is the shortest path length and b is the branching factor, and the algorithm has high memory requirements as all generated nodes are kept in memory [2]. It was decided we would not use A* algorithm. The next consideration was to simply "pad" the perimeter of the hazard polygon and follow it all the way around until the AGC reaches the golfer. This concept is shown in the diagram below in Fig. (6) where the red line denotes the path that would be taken by the AGC to avoid a hazard zone. Clearly this is an inefficient path the AGC will unnecessarily follow the contour of the polygon. Additionally, this method increases the risk that the AGC enters the hazard zone if our GPS system is not performing well, or if our GPS mapping of hazard zones is erronous.

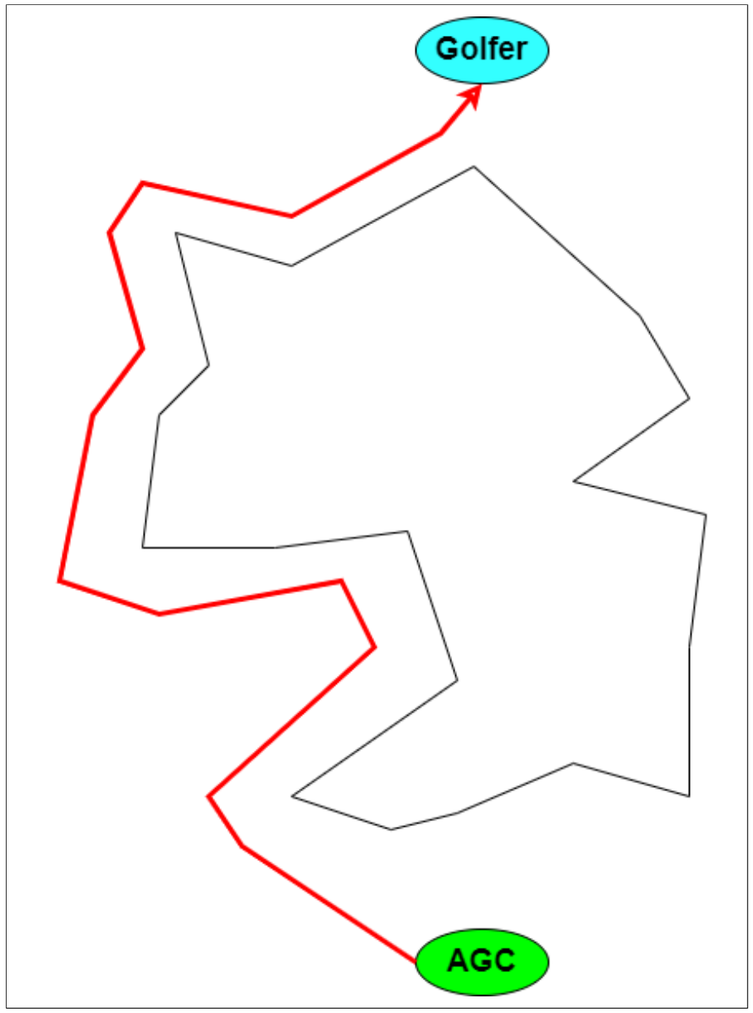

Figure 6: Path derived by "padding" polygon

A custom pathfinding algorithm was designed and implemented. The path it finds is not the optimal solution, but it computes quickly and the path derived is more efficient than using the "padding" method. Fig. (7) shows the main steps performed by the algorithm to determine where the AGC should go.

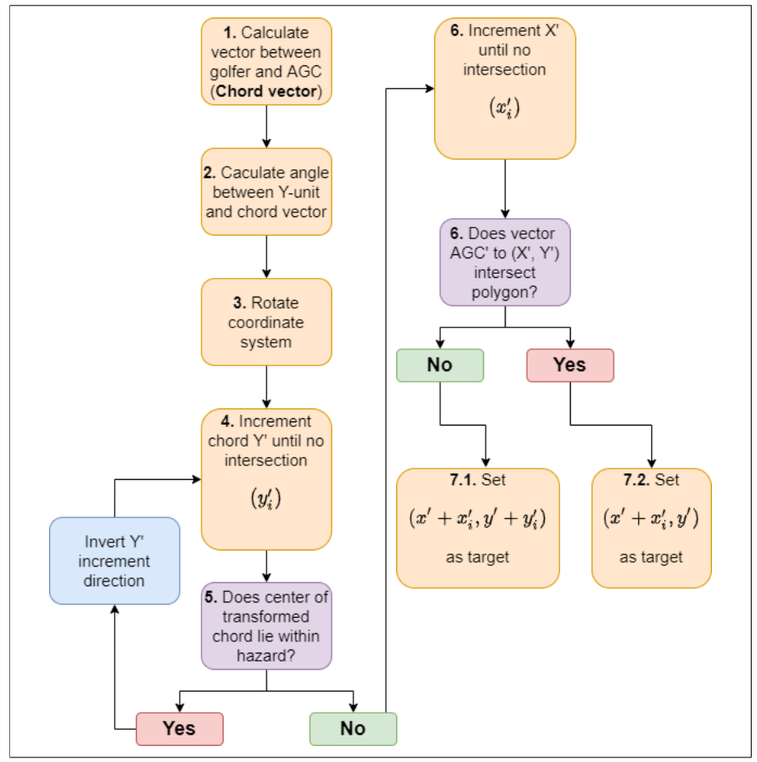

Figure 7: Pathfinding algorithm main steps

First the chord vector, $\overrightarrow{\mathbf{C}}$, is calculated, along with the angle, θ, between $\overrightarrow{\mathbf{C}}$ and the Y-unit vector, $\hat{\mathbf{y}}$, using Eq. (). The relevant polygon, golfer position, and AGC position are rotated about the z-axis with the center of rotation as the origin by θ using Eq. () where $\overrightarrow{p}$ is a three dimensional point and $\overrightarrow{p'}$ is the rotated point. The polygon is rotated by applying Eq. () to each vertex. The purpose of the rotation was to aid the design process of the algorithm and make the code more readable, removing the necesscity for vector manipulations throuhout the algorithm instead using only simple arithmetic operations. This results in $\overrightarrow{\mathbf{C}'}$ having constant y. The value of y' is increment in one meter steps until no intersection between the translated chord vector, $\overrightarrow{\mathbf{C}'_t}$, and the rotated polygon, $\mathbf{P}'$. If any point of $\overrightarrow{\mathbf{C}'_t}$ lies within the polygon the increment direction is reversed and incremented until there is no intersection and no points of $\overrightarrow{\mathbf{C}'_t}$ lie within the polygon. This increment process is repeated in the x' direction for the vector joining the rotated AGC position, (x'_{AGC}, y'_{AGC}), to $(x'_{AGC}, y'_{AGC} + y'_i)$. Once the incrementation

process if finished such that no intersections are found, the vector joining (x'_{AGC}, y'_{AGC}) to $(x'_{AGC} + x'_i, y'_{AGC} + y'_i)$ is checked for intersections. If none are found then $(x'_{AGC} + x'_i, y'_{AGC} + y'_i)$ is rotated back to the original coordinate system and set as the position target for the AGC. If an intersection is found then $(x'_{AGC} + x'_i, y'_{AGC})$ is rotated back to the original coordinate system and set as the AGC position target. This process repeats until the golfer is reached. Incrementing in the y' direction is equivelant to incrementing perpendicular to $\overrightarrow{\mathbf{C}}$ in the original coordinate system, and incrementing it by x' is equivelant to incrementing in the direction of $\overrightarrow{\mathbf{C}}$.

5.1.5 Pathfinding Diagram

Fig. (8) shows a visual of how the pathfinding algorithm will determine intermediary positions for the AGC to move it towards the golfer. For the generation of this figure the pathfinding algorithm was only called again after the target position is reached, resulting in the very wide path that can be seen in the final path diagram. On the AGC the pathfinding algorithm is called every time new GPS data is available, approximately every second, resulting in a smoother and closer path being found.

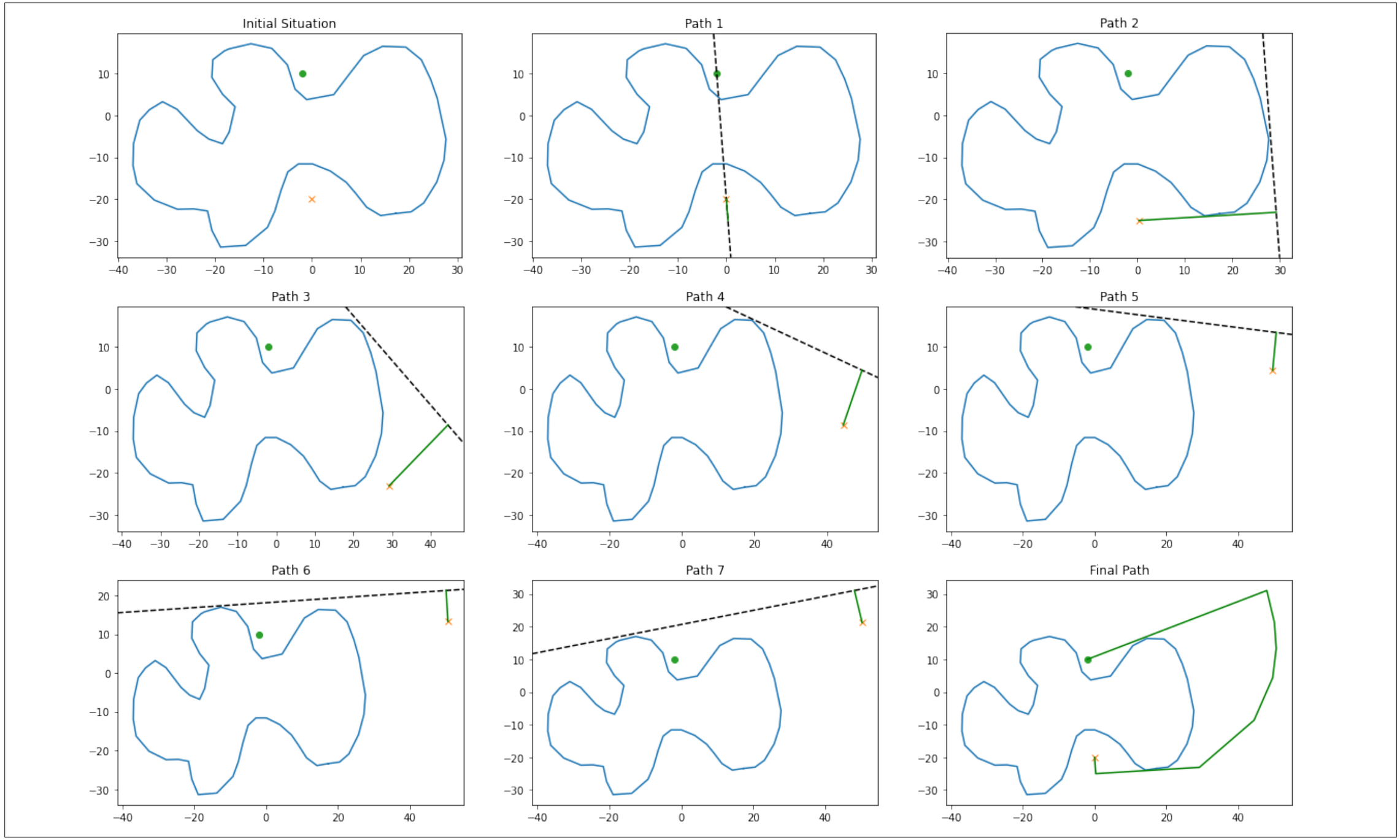

Figure 8: Visualisation of path determined by pathfinding algorithm

5.2 Communications

The EMG30 motors we are using are controlled by the MD25 motor controller board via and I^2C interface. The communication interface for this was written in C and is capable of handling multiple I^2C devices. Currently the only device we have on the communications network is the MD25. The GPS in the golfer's pod communicates to the main Raspberry Pi via Bluetooth, which is written in Python using the PyBluez library. The GPSs communicate with the Raspberry Pis via serial commuication implemented using the Pyserial library.

5.3 GUI

5.3.1 Course Map

The course map was implemented on the GUI using the map system that was used on the simulator. We attempted to embed a map web app called folium which looks better than the map developed for the simulator, however, it doesn't allow us to retreive coordinates for a point clicked on the map in real latitude / longitude coordinates or in screen coordinates. This shortfall makes it impossible to give the golfer the ability to click a point on the map and see how far away it is from the AGC, using the simulator map allows us to add that feature.

6 Electronics

6.1 Motors

The motors we decided to use for our prototype version are the EMG30 motors because we already have them and this will save cost. However, they are underpowered as they will take too long to accelerate the AGC to full speed, and won't work at on anything more than a slight inclin; for a final version the EMG49 motors should be used. We chose these motors by considering the top speed we wish the AGC to able to go, and how fast we want it to accelerate to that speed on a 10° incline. The calculations shown below in Fig. (9) show how we calculated the necessary toruqe and rpm we require from our motors.

The top speed we chose for the AGC was 5.0 kilometers per hour, equivalent to average walking speed. The wheel radius, W_r of the AGC wheels is 0.115 meters. We can calculate RPM using Eq. (7):

$$rpm = \frac{7.5 * \frac{1000}{3600}}{2\pi W_r} \cdot 60 \approx 115rpm \tag{7}$$

The AGC should be able to accelerate to this speed, V, within $t = 5$ seconds on a 10° incline. For predicted final mass, $M = 15kg$, the accelerating force, F, required can be calculated with Eq. (8):

$$F = \frac{V}{t} + Mg_0 \sin(10°) \approx 29N \tag{8}$$

Finally the torque required from each motor can bel calulated using Eq. (9):

$$T = W_r * F \approx 1.7Nm \tag{9}$$

Figure 9: Motor requirement calculations

According to their datasheet the EMG49 motors can provided a loaded rpm of 122 and a torque of 1.56, close to our requirements set out above. As the torque is slightly lower than calulated, the AGC will accelerate slightly to top speed on a 10° incline in slightly over five seconds.

References

[1] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, 1991.

[2] S. Russell and P. Norvig, *Artificial intelligence: A modern approach*. Pearson, 3 ed., 2009.