The Autonomous Coordinate System

Michael Peng

October 2017

1 Introduction

The robot's ability to locate itself during the Autonomous Period in FTC is crucial to point scoring. The following are the specified point-scoring actions during it in *Relic Recovery*.

- Jewel-knocking
- Storing glyph in correct or incorrect cryptobox
- Parking in safe zone

All of these activities require precision in the robot's position and orientation. For this reason, the robot should have strong self-awareness regarding its positional relationship to key game elements.

1.1 This Solution

The primary goal of *ACS* is to provide environmental awareness to the robot in a manner as independent as possible. In this solution, the only external factors on which it relies are the individual motor encoders in the drivetrain. Using these encoders, the robot can determine its relative position to the starting point.

2 Description

The *ACS*, in operation, references an existing two-dimensional game map that illustrates a quadrant of the game arena, to which the robot's operations are limited. The implementation of this obstacle map involves coplanar point-to-point polygons that describe either a game obstacle or the map boundary. Additional named points, along with direction facing, describe robot positions in which game tasks can be performed. In the case of *Relic Recovery*, for example, positions for the following tasks can be included:

- Reading the cipher

- Knocking off a jewel
- Putting a glyph in each of three cryptobox columns

3 Implementation

Although code for the robot controller is in Java, data structures in this document are in Python-like pseudocode for improved clarity. *ACS* uses the following Java libraries:

- **JTS Topology Suite** is an open source Java library for manipulating and creating vector geometry. JTS is necessary for its ability to help *ACS* simulate the game environment geometrically and perform manipulations on individual elements.

3.1 Data Structures

3.1.1 Positions

```
struct Position:
  location: jts.Coordinate
  orientation: angle in radians
```

3.1.2 Map

```
struct GameMap:
  obstacles: dict<str, jts.Polygon>
  boundary: jts.Polygon
  positions: dict<str, Position>
```

3.2 Key Functions

Pathfinding for the robot is done using the *A** algorithm for obstacle avoidance. Nodes are arranged in a grid with a distance of 1 inch in between; There is a path from each node to every node next to it, including diagonal ones. During path generation, if a node touches an obstacle or the map boundary, it cannot be used in the path.

A specialization of A* for *ACS* is as follows.

Algorithm 1 Find path from a to b

procedure A*(a, b)
  $f \leftarrow \{[a, 0]\}$ ▷ Priority queue: Frontier
  $p \leftarrow \{a : \textbf{nil}\}$ ▷ Key-value map: Path
  $c \leftarrow \{a : 0\}$ ▷ Key-value map: Cost to node
  while $\#f > 0$ **do**
    $s \leftarrow$ pop item from f ▷ Coordinate: Current node
    if $s = b$ **then**
      return BACKTRACK(s, a, p) ▷ From s to a in p
    end if
    for $n \leftarrow$ ADJACENTPOINTS(s) **do** ▷ Coordinate: Next node
      $o \leftarrow c[s] +$ JTS.DISTANCE(s, n) ▷ Real: Distance to next node
      if $(n \notin c) \vee (o < c[n])$ **then** ▷ New cheapest cost to n
        $c[n] \leftarrow o$
        put n into f with priority $o +$ JTS.DISTANCE(b, n)
        $p[n] \leftarrow s$
      end if
    end for
  end while
end procedure

If the algorithm above were to be used with the result directly fed to the drivetrain manager, the robot would stop itself everytime a new node is reached. To alleviate this ineffiency, the following algorithm converts the steps into a multi-point path that describes when the driving direction changes.

Algorithm 2 Convert nodes P to points p

Require: P to be an array of coordinates
procedure CONVERTPATH(P)
  $l \leftarrow (0, 0)$ ▷ Last vector
  $o \leftarrow \emptyset$ ▷ Output path array (REPEATABLE) (ORDERED)
  for i **in** $[1, \#P)$ **do** ▷ Index
    $v \leftarrow \overrightarrow{(P[i-1])(P[i])}$ ▷ Current vector
    if $v \neq l$ **then**
      $o \leftarrow o \cup \{P[i-1]\}$
      $l \leftarrow v$
    end if
  end for
  return $o \cup \{P[\#P - 1]\}$
end procedure

Consequently, the results from the above algorithm are converted to calls to the Drivetrain Manager in the following algorithm.

Algorithm 3 Carry out navigation of points p

Require: p to be an array of coordinates

```
procedure NAVIGATE(p)
    for i in [1, #p] do                                  ▷ Index
        Drivetrain.move(\overrightarrow{(p[i-1])(p[i])})
    end for
end procedure
```