Sphero's Hero – agent description

The agent consists of two files:

- arena.py – supporting classes providing information about arena
- hero.py – game and sphero controlling class

The file "arena.py" provides three primary classes:

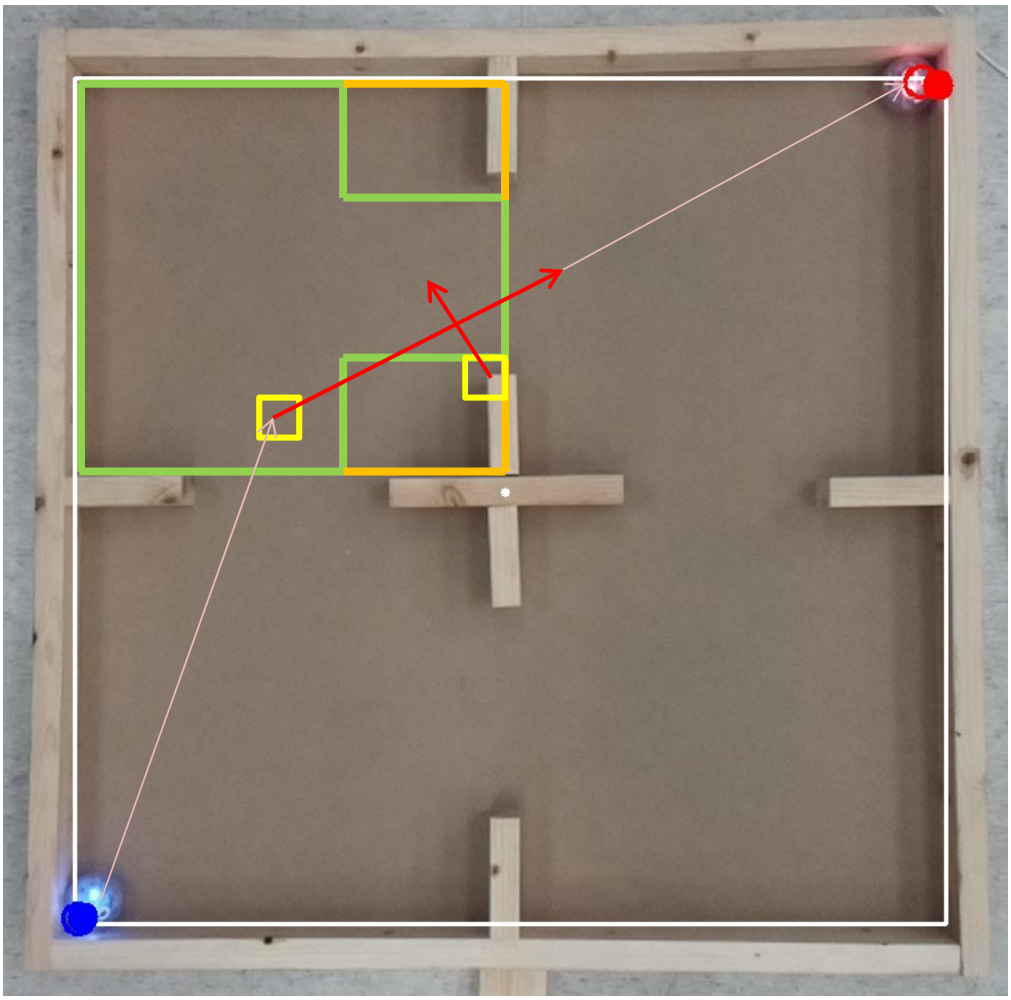
- APoint – defines a point (x,y) and operations that can be performed on a point including normal math operations and simple comparisons. All points are processed using this class to simply implementation.
- Abounds – defines a rectangle (x0,y0,x1,y1 and width,height) and operations that can be performed including shifting and warping (change size) as well as retrieve various attributes. Most rectangle operations within the arena use this class.
- Arena – contains many attributes about the arena including size, base locations, and the move table. It also implements the move table generation and simple operations related to the arena.

The move table is the key to how the agent determines where to go. All the information is generated at game start (when the agent is launched) once the base locations are received. This is a grid of cells laid over the arena that for each point determines a good target location for a move to avoid the obstacles. This is computed by a fixed set of targets depending on the location of the cell within the quadrant. A sample image showing two cells is on the next page (only moves for the first quadrant is shown; also note the cell alignment is not as precise as shown due to sloppiness is arena configurations). This image is shown from perspective of blue; for playing red the directions are all reversed.

Two sample cells are shown in the picture outlined in yellow with the desired move from that cell shown in red. All cells in the green box would have the same target while all cells in the orange box would have the alternate target. Note that the alternate target avoids impact with barrier moving back into the quadrant in such a manner as to enter the green box. This allows the next move to enter the following quadrant. Note also that the cell in the green box is approximately where any moves entering from the lower quadrant would result. Thus under ideal conditions (pink lines) one move would be from blue base up to enter the first quadrant, then one move to enter second quadrant, and finally one move to the red base.

All moves from the first quadrant enter the second horizontally. All moves within the second quadrant (red base) are directly to the red base. All moves from the last quadrant enter the second vertically. To support opponent avoidance, two sets of moves are generated from the second quadrant (blue base), one vertically and one horizontally. The location of the opponent is used when selecting which of the two home-quadrant moves to select. When the agent reaches the opponent base and is holding the flag, the move table is reflected with all angles rotated 180degrees. This allows use of one table for both directions.

For testing, the arena module provides a visual test operation. Just execute 'python arena.py' (with the camera active) and you can move the cell of interest around (use arrow keys) to visually see the target of each cell. It will show planned moves as well as potential opponent locations and which move would be used for avoidance. (Look at the source code to determine colors; you can also generate a static view [code edit] for an image showing all cells with all moves but it is not very useful.)



The file "hero.py" provides a single class agent that provides all the actual movement control. Use of either the camera location or the sphero odometer is supported (simply converts odometer position to pixels). Due to drift in the odometer positioning, it will occasionally reset the position based on the camera even when using the odometer for movement.

The original camera-based tracker lagged the real location so the agent attempts to ensure it has a good camera location by waiting on three position messages all of the same value (plus/minus a small delta). [This was experimented with both time-based and count-based latch.] After the camera-based tracker was updated and found to be fast enough, the use of the odometer was less useful although the support logic remains.

On startup this creates a single Arena class instance, obtains the base locations and passes those to the arena class for move table creation. It then completes ROS initialization subscribing and publishing the appropriate topics. It also computes a sphero-move to pixel scalar to use if basing actions on the odometer. This was done as it was found the original scalar used by the arena code was not as accurate. Finally it resets the sphero's location and heading with the origin of the player's base so that all computations using that information can have a known origin. (This action is ignored in the simulation environment.) Finally it goes into wait-on-start state (will report it is waiting once each 10 seconds).

Whenever the various ROS messages are received, the agent just retains that data for use by the movement logic. Four primary input messages are processed:

- Opponent location – from the camera tracker, just stored for use in opponent avoidance
- Pixel location – from camera tracker, the value is retained and if within 10 pixels of last location it is marked as "stable" for possible use in resetting the odometer location
- Odometer location – from sphero odometer, value retained for possible use by movement
- Sphero velocity – from sphero, used to detect if sphero is not moving; if value is within zero plus/minus 20 then it is considered "stuck" and a flag set for the movement logic to consider

The agent runs at a 12Hz clock rate which was found to be the fastest that provided valid behavior. Experimentation found that changing this made little difference so this was the value selected. The agent sends a move request on each clock tick (with one exception) based on the current location so it is continually requesting a move to the target cell location (from the move table). If the state of holding the flag changes, the move table target will immediately reflect this and the agent will reverse itself.

For each clock tick (12Hz) the agent performs two actions:

1) If waiting for a stable tracker or not moving (stuck), it needs to re-evaluate its position and possibly attempt a random move.
    a. If the camera tracker was marked as stable, reset the odometer to this location.
    b. If stuck, generate a random angular move
2) Attempt a movement based on the current location
    a. If waiting on stable tracker or just performed an angular move, wait for 5 ticks (this was needed to avoid movement hampering an attempt to recover from the stuck position)

b. Otherwise get the target location from the move table based on current location, having the flag, and the opponent location and attempt to move there. If the move table reports that the reported position is outside of the arena, wait for a position reset.

When generating the twist message to send to the sphero, the velocity is scaled depending on distance to the target location. Since it is expected to be a straight line (from the move table), the further the point the higher the velocity can be. Experimentation showed this has to be a narrow range (currently 20-40) as too large a value causes the sphero to overshoot and too small just moves too slowly.

That is effectively the complete movement behavior; just repeat those actions on each tick. Note that this means an attempt to move to the same target point is performed repeatedly each time possibly based on an updated current location while the target location won't necessarily change within the same quadrant. During play you may occasionally notice the sphero jerking which is due to the cell transitioning between the green (other quadrant target) and orange (same quadrant target).

It is also possible for the sphero to "dance" when in its home quadrant (or opponent quadrant if holding the flag) and the opponent is detected as crossing the left/right line. For opponent avoidance, a line is drawn between the two quadrants and the opponent location simply compared to being left or right of this line. Note that when near a base, very little movement is needed to cross this line so just a wobble in the opponent's movement near the base can trigger a change in avoidance direction. (This "dance" is especially noticeable when the Sphero's Hero agent is competing against itself.)

An attempt was made to allow resetting of the sphero's orientation during game play but was never fully completed. (You can find part of this logic in the function 'compute_line' in hero.py). The approach was to retain the last few location points for the sphero, perform a line fitting, and if close enough to straight, then use that angle as the new heading. Unfortunately it was found that as the game progressed, the sphero made fewer straight line moves at a time when it would become more necessary. (Also lack of correct angle computation and apparent little need for this effort prevented inclusion.) It seems the approach would require stopping normal movement to instead move in a straight line just to get a heading; but any little interruption (barriers or opponent) would void that result losing valuable time.

If the agent is run with the test option (3rd command line parameter), this will enable some debug statements reporting what it is doing as well as enabling a visual image being published. Displaying the image on topic '/{color}_sphero/hero_test' provides a visual presentation of the current perceived sphero location, the current move cell, and the planned target point. This visual is especially useful for odometer operation as it shows the difference between the real location (camera location) and where the location is perceived based on sphero odometer conversion to pixels. (The straight line approximation can also be seen here if a few lines of code are restored.)

The Sphero's Hero team hopes you enjoy the challenge and are able to learn something from our agent.