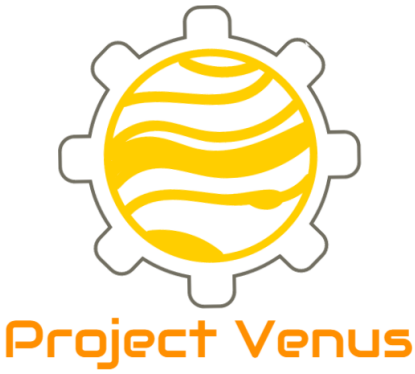


Technical Specification

SDP Group 15-H

Emilia Bogdanova
Patrick Green
Julijonas Kikutis
David McArthur
Aseem Narang
Ankit Sonkar

The University of Edinburgh

Contents

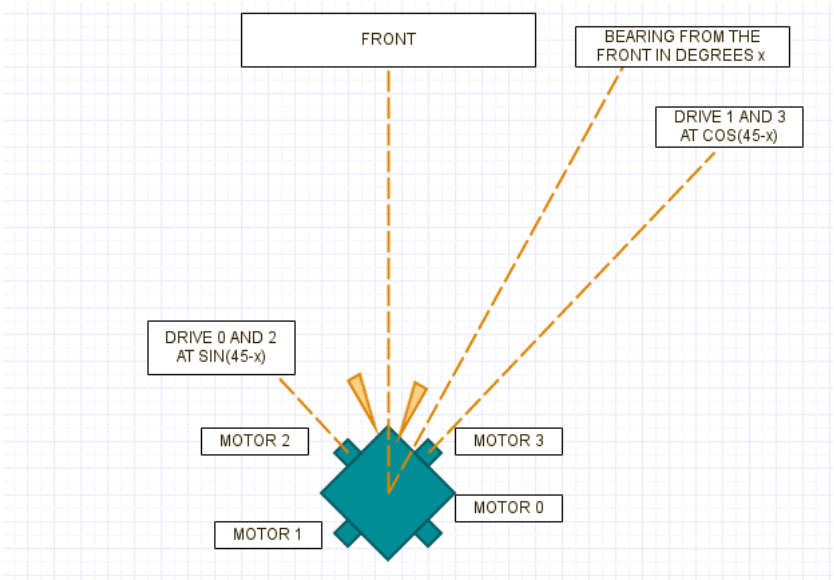


Figure 1: Visualisation of the calculating the motor power components

1 System architecture

2 Hardware components

2.1 Motors

2.2 Chassis structure

2.3 Holonomics

Holonomic motion was an obvious choice for the final robot. It enabled fast motion in any direction meaning the robot could respond very to a change in play. It also was perfectly suited to a strategy using potential fields as an optimal direction would be output and implemented instantly using the firmware. The motion was constructed by calculating the angle between the desired direction and the robot orientation. This was used to compute the driving components that are sent to the Arduino as seen in Figure 1. The matrix multiplication below was used to extract the driving powers and then each power was scaled up by a factor of 100 divided by the absolute maximum of the four motor powers.

$$\begin{pmatrix} \text{MOTOR 0} \\ \text{MOTOR 1} \\ \text{MOTOR 2} \\ \text{MOTOR 3} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \\ -1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(45-x) \\ \sin(45-x) \end{pmatrix}$$

2.4 Weight distribution

2.5 Grabber

The grabber consists of two symmetric parts placed one slightly above the other as seen in Figure 2. To keep the robot within dimensions the grabber arms overlap enabling a large span but still within regulations when closed. A symmetric gear system is used so that both arms are opened and closed at the same time. Unlike the original design shown in Figure 3, the arm axles were placed closer together so that the grabbed ball would always be in the same position increasing the reliability of the sensor and the kick.

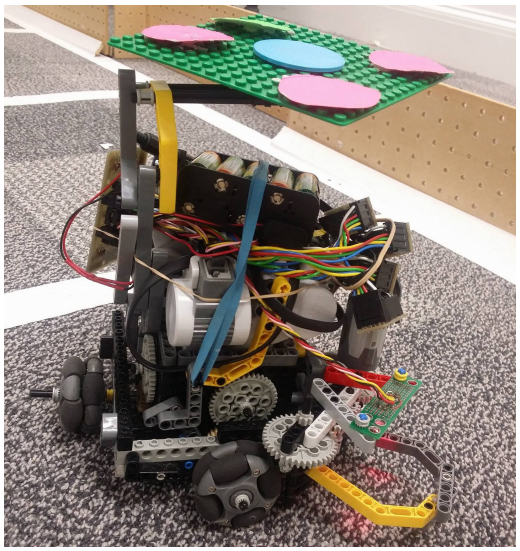


Figure 2: The current robot

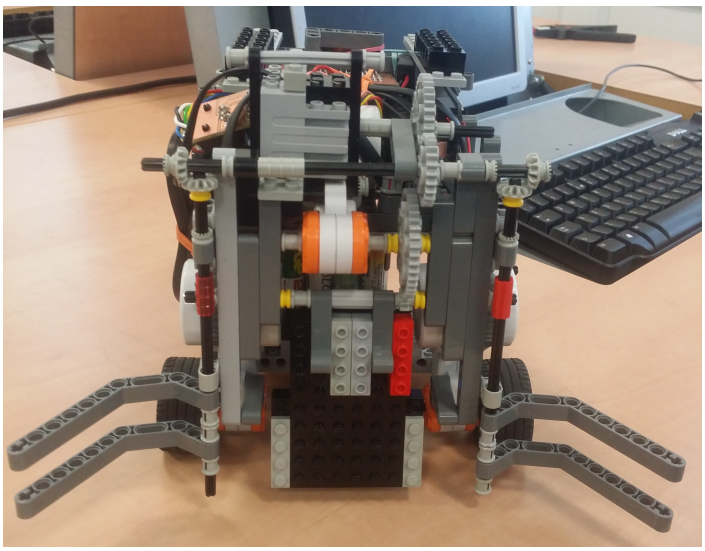


Figure 3: The previous robot with opened grabber

2.6 Kicker

The newest design does not have a kicker. Instead, the kick is made by turning the robot and opening the grabber at the same time. The main incentive of this design was to reduce the asymmetric weight distribution which would have been catastrophic to optimising the holonomic movement. The robot rotates in the intended kick direction when it is aligning itself with the goal. In doing this the centrifugal force holds the ball towards the end of the grabber and the ball is in contact with the claw that will apply the kicking force. This increases the accuracy of the kick. As the error margin of the kick is asymmetric and biased towards undershooting, the kick is initiated in different directions in different positions on the pitch. The y dimension is split into four segments: 0, $\frac{1}{4}$, $\frac{1}{2}$, and $\frac{3}{4}$. In Figures 4 and 5 the robots demonstrate each segment and its correct and incorrect kick directions. This kicking mechanism is not only more powerful than the previous design but it is also able to confuse the defence strategies of opponents tournament[1].

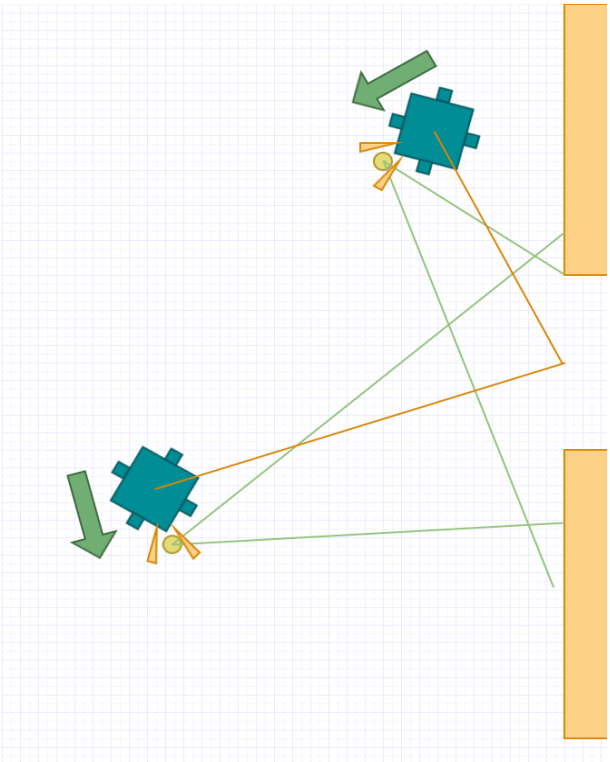


Figure 4: Correct kick rotation

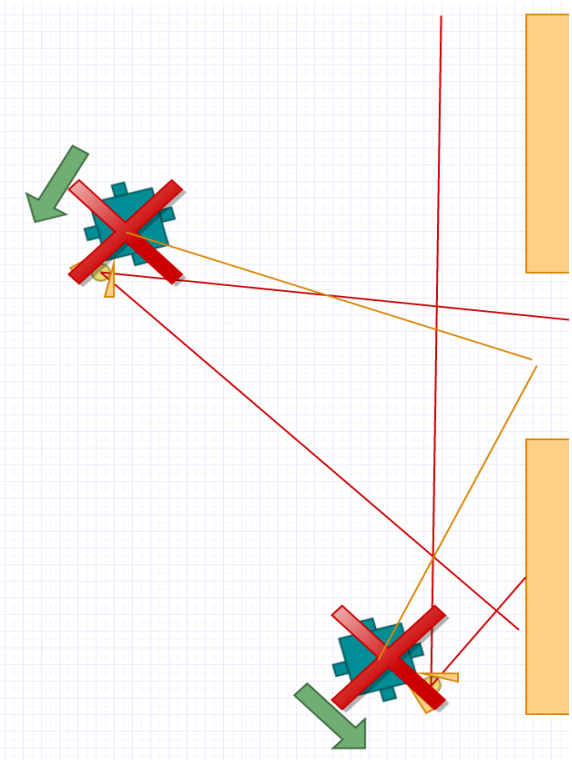


Figure 5: Incorrect kick rotation

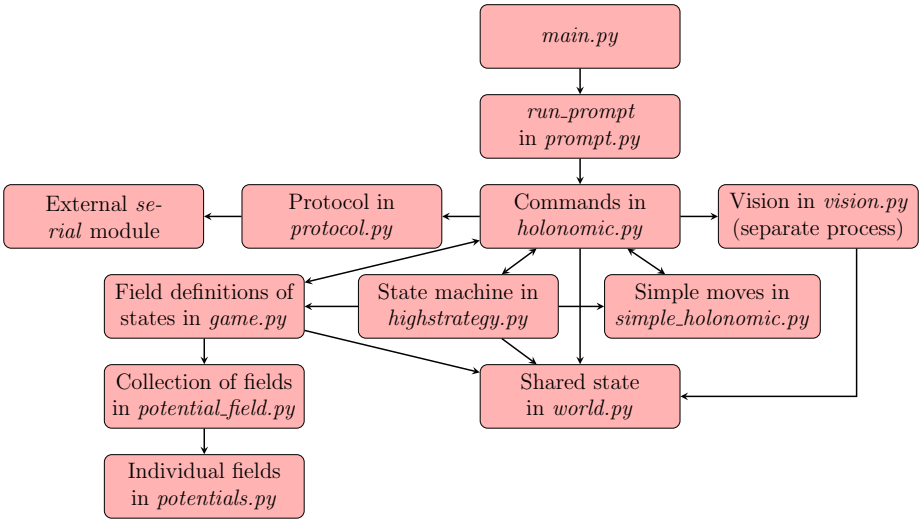


Figure 6: Dependency graph of Python modules in the system

3 Documentation of the code

The code is subdivided into several Python modules as seen in Figure 6. The module *main* is used to launch the *prompt* module. The commands in the prompt are provided by the *commands* module. It creates instances of protocol, vision, and state machine objects from the respective modules. The vision object is run on a separate thread to process frames asynchronously with constantly updated shared world state kept in a *World* object. At the same time the prompt is provided for the user. The command to run the strategy is

[1]Footage of one such example of a confusing spin kick is available at `https://youtu.be/VJLo2x2gdGk`.

called `hs`. This command constantly queries the state machine in the *highstrategy* module. The state machine checks the world state to decide which state it is currently in and then performs the associated action. It is done either by handing over the execution to *game* module to construct a potential field and perform the best action based on it or performing a predefined move from the *simple_holonomic* module.

All code except the color calibration user interface in the *vision* module is an original work. Libraries used are *pyserial*, *numpy*, *scipy*, and *OpenCV*.

3.1 Communications

The communication interface between the Arduino and PC is low level as the PC decides and specifies the individual motor numbers and rotary encoder value or time duration for which they will be powered. Then the Arduino sends acknowledgement to the PC about the arrival of the command, turns the motors on, and sets the specified timeouts to stop them. The messages are human-readable, newline-terminated and tokens inside them are separated by spaces. The specified motor power can be negative, in which case it means backwards direction. Each message which changes the state of the robot has a sequence number and checksum that are checked in the Arduino. The messages used in the protocol are listed in Table 1. The communication messages are constructed in the *Protocol* class using methods named after the corresponding message in `control/protocol.py` and these are used in the motion commands in `control/holonomic.py`.

Rotate the motors for n ms	`M seqNo checksum n motorNo power...`
Rotate the motors for n rotary values	`R seqNo checksum n motorNo power...`
Rotate the motors indefinitely	`V seqNo checksum motorNo power...`
Stop all motors	`S seqNo checksum`
Return `D` if all motors are stopped	`I`
Handshake, reset the sequence number	`H`
Query light sensor, return `D` or `N`	`A threshold`
Transfer a byte to I2C bus	`T byteInDecimalASCII`

Table 1: Messages available in the protocol

3.2 Arduino

The Arduino code uses the *SerialCommand* library to buffer and tokenize the commands received over the serial link. Messages changing the world state are sent with a sequence number and checksum which is the sum of all the parameters following it. If the sequence number matches the one from the last command, acknowledgement is sent but no further work is performed for the duplicate message. This handles the situation arising when an acknowledgement from the Arduino does not reach the PC and and the PC generates a duplicate message. If the checksum does not match, the Arduino code ignores the message and does not send an acknowledgement, thus forcing the PC to send the same message again.

If any of the motor move commands are sent, the motors are started immediately using the *SDPArduino* library. Then the Arduino schedules when to stop the motors and there are

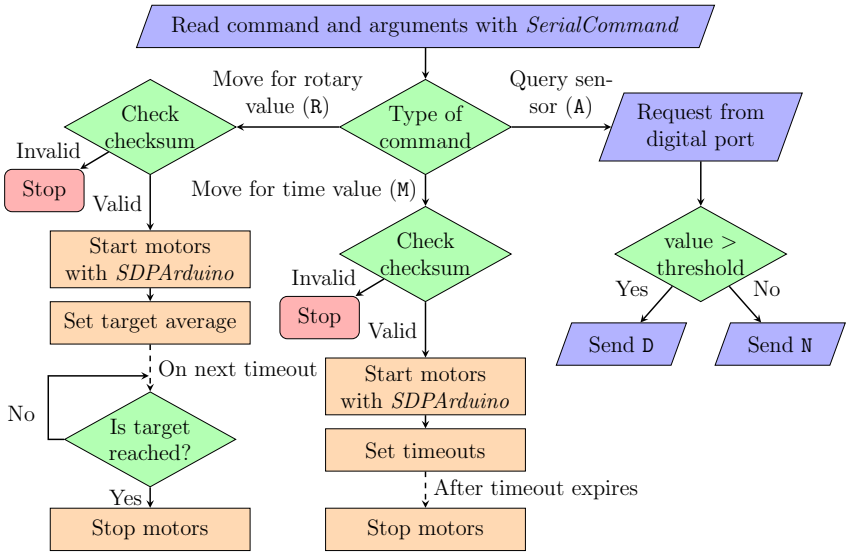


Figure 7: Flowchart of message processing in the Arduino

two methods to perform that: either a time value or rotary encoder value. The flowchart for these two methods, along with querying the light sensor, are detailed in Figure 7. In the case of the time value, a timeout is set to stop each single motor using *setTimeout* from the *SimpleTimer* library. In the case of rotary encoder value, *setInterval* is used which calls a function every 5 ms that queries the rotary encoder board and stops the motors when the average of all four motor rotary encoder values reaches the target value. These approaches using timers allows the robot to receive commands asynchronously, that is, a command is not blocking during its execution and the PC software could, for example, send another command simultaneously to engage the kicker while the robot is in motion. The Arduino message handling is located in `arduino/arduino.ino` file.

A buffer of upcoming motor jobs has also been implemented in the Arduino code to ensure continuous motion but it was deemed unnecessary as vision could issue new commands quickly enough without the robot coming to halt. The current implementation also never stops a motor when a new command arrives for the same motor, instead it just executes the new command, ensuring continuous motion.

3.3 Vision

In order to detect the robots and ball the following steps are executed:

1. The dictionary of the camera capture settings (brightness, contrast, hue and saturation) are read from `room0/1.txt`, pitch dimensions from `pitch0/1.txt`, and color thresholds from `color0/1.txt`.
2. The aforementioned settings are used, a frame is read, and the Gaussian blur is applied to create more solid features.
3. A unique application to speed up the feed is the variable `world.undistort` which lets the strategy to conditionally remove the barrel distortion when it is unnecessary.
4. All color thresholds are added to the same mask and K-means is used to group them together to form various spots in the image.

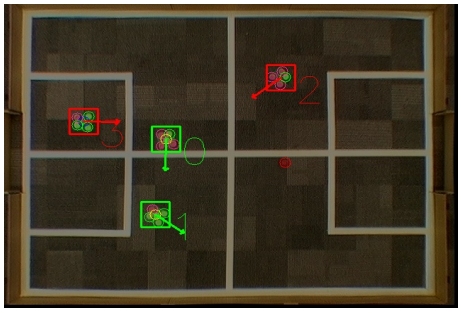


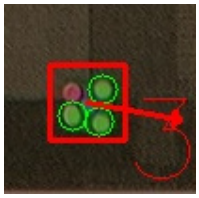
Figure 8: The four robots marked on the vision feed.

5. Then the color is found from the center pixel of the cluster. The spot is either kept or removed depending on the minimum area for that color which varies as some colors are more difficult to see.
6. The individual methods required for finding the robots and ball are then implemented as outlined below.
7. When the vision system is closed using the escape key, the most recent calibration values and the current camera settings defined by the slider bars are saved and can be used on the next execution.

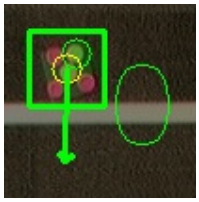
3.3.1 Finding the robots

Due to the varying light intensities over the pitch a highly tolerant method of identifying robots is used. The robot identities are pre-defined and dependent on the choice of the center and corner spot as seen in Figure 8. Numbers are assigned sequentially to our robot, teammate, first enemy, and second enemy.

There are two independent ways the robot can be identified:



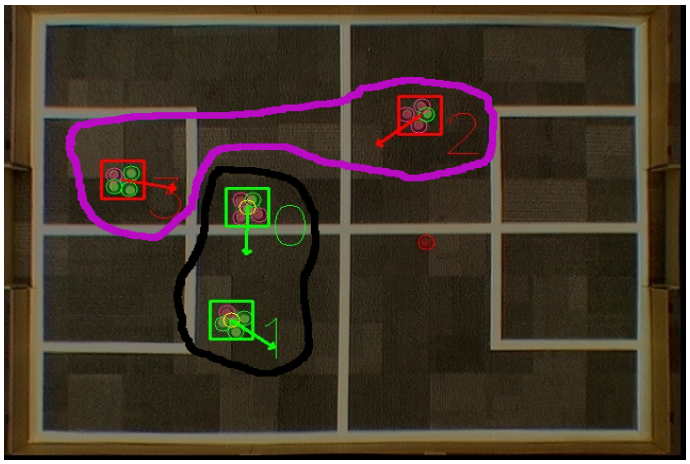
The three spot method: Finds the largest of the three distances between spots, then draws a vector from the midpoint of this edge to the spot not included in the edge. The orientation is then computable by rotating this vector by a fixed amount. Then, the robot center is calculated using the average of the spot centres.

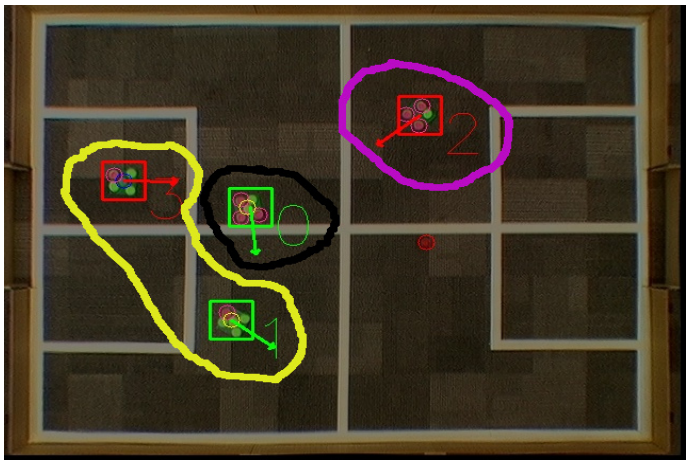


The two spot method: As each spot is distinguishable, a vector can be constructed between the center spot and the corner spot and rotated a fixed amount to find the orientation. Then, the robot center is calculated using the center spot.

As there will always be two robots with the same plate configuration required for the methods above, the algorithm needs some of the robots to be identified using more information. As seen above, the robots from the same teams will have the same two spot plate configuration. Also, for the three spot method the robots 0, 2 and 1, 3 will have the same plate configurations. To solve this, the spots are initially grouped together into potential robots in the order of descending areas. The groups are run through several filters for each type of robot accepting the groups if there exists a specific numbers of coloured spots. Each spot

that is found is circled with its respective color on the vision feed. The sets of spots on the filers bellow correspond to finding the robot zero.





FILTER 1 { ● ● ● ● }
To pass into here it must have a correct team spot and no other visible team color. It must also have three of its three spot color.

FILTER 2 { ● ● ● ● , ● ● }
To pass into here it must have a correct team spot. It also must have more than one of its three spot color or less than two of its corner spot color.

FILTER 3 { ● ● ● ● , ● ● , ● ● ● }
To pass into here no team spot is required, it must only have more than one of its three spot color or less than two of its corner spot color.

As each identity is found, its corresponding filters are blocked and its position is noted so that a robot does not get detected in the same place as the one that is already detected. Once any group of spots gets through a filter of a given identity, the identity of that robot is then updated using either the three spot or two spot method depending on what information is available. If a robot is not found, it keeps its original position. To handle cases where robots are lifted off the pitch, the system creates a ghost robot marked in blue on the vision feed. This tells the strategy to ignore the contribution to the game of that robot but the robot continues existing on the pitch in the vision thread.

3.3.2 Finding the ball

The algorithm looks for 10 red spots and checks through them in the order of descending area. To prevent a pink being misclassified as the ball, each red spot is checked whether it exists close to the centres of the robots. As the robots can shield the ball from view, a method is used which determines whether a specific robot is in range of the ball. If so, when the ball is shielded, an imaginary ball is placed along the orientation vector of that robot until the ball is found. This method is in place for all robots except for Venus because the sensor is being used instead to determine whether Venus has the ball.

3.4 Strategy

3.4.1 Potential fields

The strategy relies on the physical properties of a potential field. This makes it possible to compound together separate tasks that the robot has to undertake with ease, such as avoiding others and grabbing a ball, at the same time. Each interaction is implemented using the same behaviour as a charged electron would have when interacting with various physical objects of different charges. Using this it is possible to swerve in front of the kicks and navigate mazes of objects fast and efficiently.

3.4.2 State machine

As seen in Table 2 bla bla

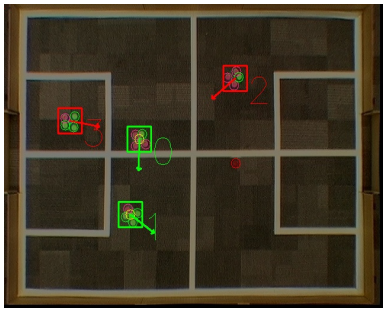
State	**Trigger**
FREE_BALL_YOURS	Ball not in range for any Robot
ATTACK_GOAL	venus has ball and Goal shot safe
ATTACK_PASS	venus has ball and Goal shot not safe but pass safe
RECEIVE_PASS	friend has ball and pass safe
ENEMY1_BALL_TAKE_GOAL	Enemy has possession and venus closest to guarding goal
ENEMY_BALL_TAKE_PASS	Enemy has possession and venus closest to blocking pass
FREE_BALL_NONE_GOALSIDE	
FREE_BALL_1_GOALSIDE	
FREE_BALL_2_GOALSIDE	
FREE_BALL_BOTH_GOALSIDE	

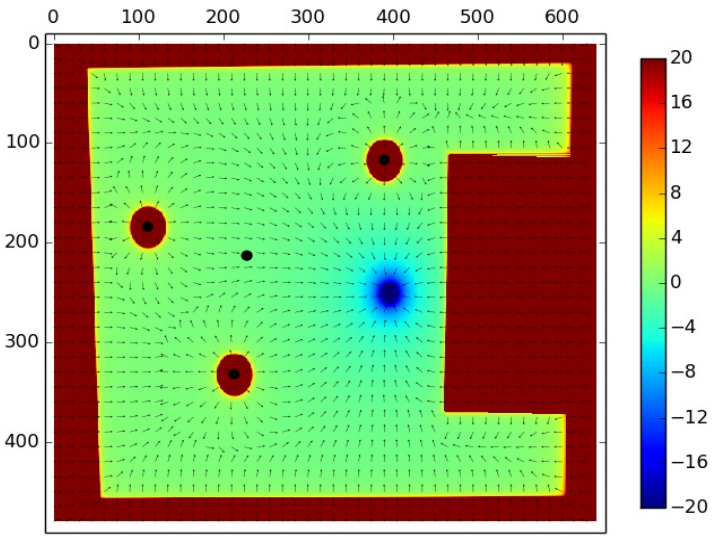
Table 2: States available in the strategy system

3.4.3 State fields

The following graphs were plotted using the command `map STATE_NAME` which is ideal for visualisation and quick testing of newly defined fields. The graphs below contain quiver plots of the force acting on Venus at a given point in space. This is drawn on top of a heat map of the value of the potential field at a given point.

FREE_BALL_YOURS

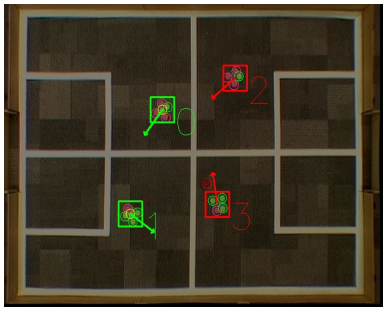


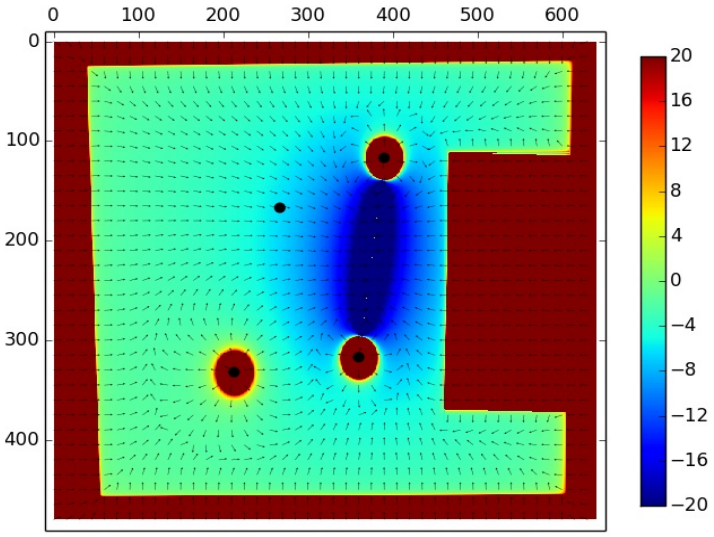


Forces due to obstacles: The walls exert a force governed by a $1/r^3$ law on the pitch side of the wall where r is the perpendicular distance to the wall. On the opposite side of the wall an attractive potential $-1/r^3$ is used instead to pull any robots into the pitch. An identical field is used in the penalty box, however, to avoid getting stuck in a local minima at the sides, the field inside the box pushes the robot to the front only and does not attract it back over the side it came from. As the box is finite, any position that is not perpendicular will exhibit a similar force, only r will be the distance to the closest corner. The robots use a $1/r^2$ law where r is the radial distance from the edge of the robot represented by the solid circle in red.

Grabbing: The ball uses an attractive radial field of $-1/r^2$. The amplitude of the field is larger than that of the robot to enable fast navigation. Once a potential of -4 is reached, the grab is initiated using quantised distance and angle motions enabled by the motor encoders. The -4 value has been chosen as it implies the ball is clear of any obstacles as otherwise the potential value would be higher.

ENEMY_BALL_TAKE_PASS



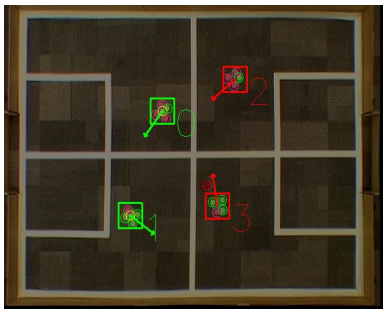


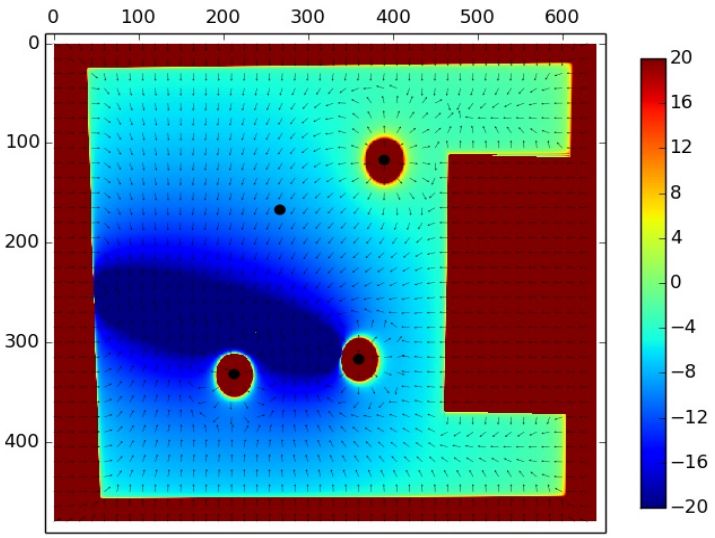
Blocking pass: A finite axial field is used to enable smooth motion during the block of a pass from any point in the pitch. The points in question are first rotated so that the field is flush with the x-axis and the force is calculated using the following equation and rotated back. d is the perpendicular distance, a is the parallel distance to the end with the smaller x value and b is similar dimension in the exact opposite direction.

$$\left(\frac{1}{\sqrt{b^2+d^2}} - \frac{1}{\sqrt{a^2+d^2}}, \frac{b}{d\sqrt{b^2+d^2}} + \frac{a}{d\sqrt{a^2+d^2}}\right)$$

Interceptions: When the current potential of Venus reaches -12, the block is satisfactory and the robot faces the ball with its grabber open. When the ball is shot as it is moving, the FREE_BALL_YOURS state is activated causing an attraction to the moving ball.

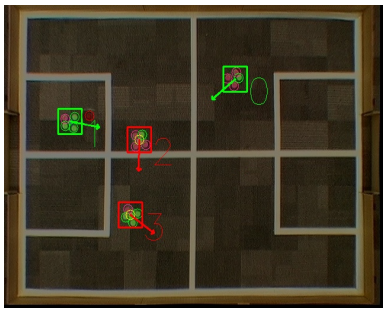
ENEMY2_BALL_TAKE_GOAL (ENEMY1_BALL_TAKE_GOAL)

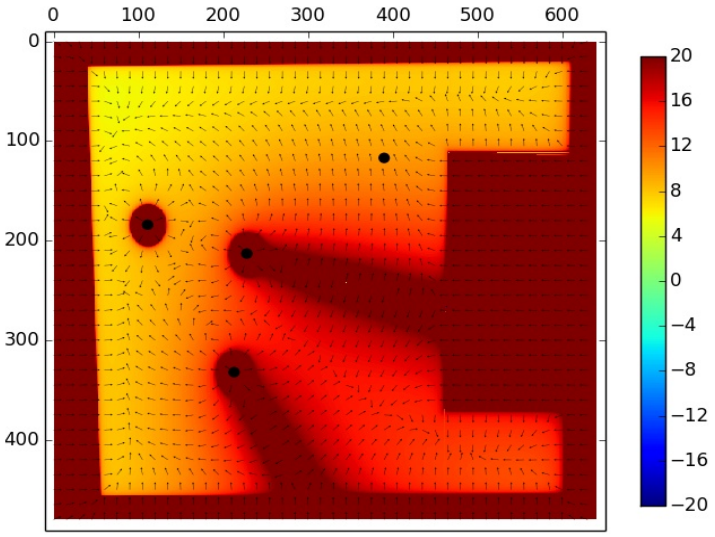




Blocking goal: The same field is used from the state above but for a block between the goal and an enemy robot. The current satisfactory potential for this block is -14 in order to be sure to block the whole goal.

FREE_BALL_NONE_GOALSIDE (FREE_BALL_1_GOALSIDE, FREE_BALL_2_GOALSIDE)





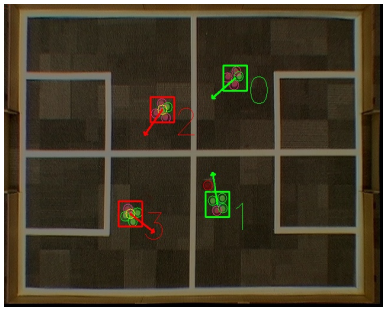
Optimising position: Given your team mate is either fetching or has the ball you want to find the best position to receive a pass. Two fields are added in order to implement this:

1. The shadowed array blocking the pass.
2. The shadowed array blocking the goal.

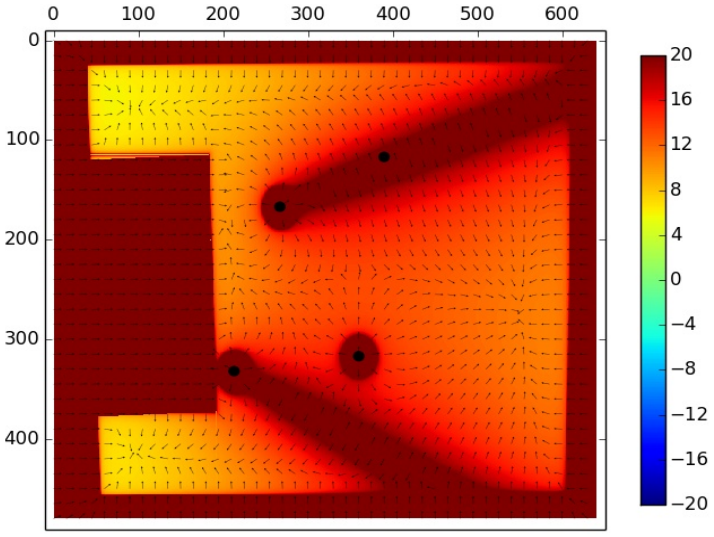
This state represents a double blocked pass. An identical type of field is used in the defence states, however, one end is placed at the start of the shadow and the other three pitch lengths away. This is so that Venus is repelled perpendicular to the shadow and also parallel around the enemy robots if needs be.

Once a satisfactory potential of 6 is reached, Venus will turn and wait to except the pass. As the other robots move, Venus will keep trying to minimise his potential until a value of 6 is obtained.

FREE_BALL_BOTH_GOALSIDE (FREE_BALL_1_GOALSIDE, FREE_BALL_2_GOALSIDE)



Local minima: Whilst optimising position Venus can get stuck as enemy players close up to defend the same shot. To deal with this, a timer is used so that if Venus has been sitting in a unsatisfactory potential for too long, we remove the least import field. In this case, the furthest player blocking the goal.



Handling opponents that divide up defensive positions: As outlined in the state above, this graph represents a double blocked goal and the final two states that are not shown are a mixture of this state and the state above.

4 Sensors

4.1 Rotary encoder board

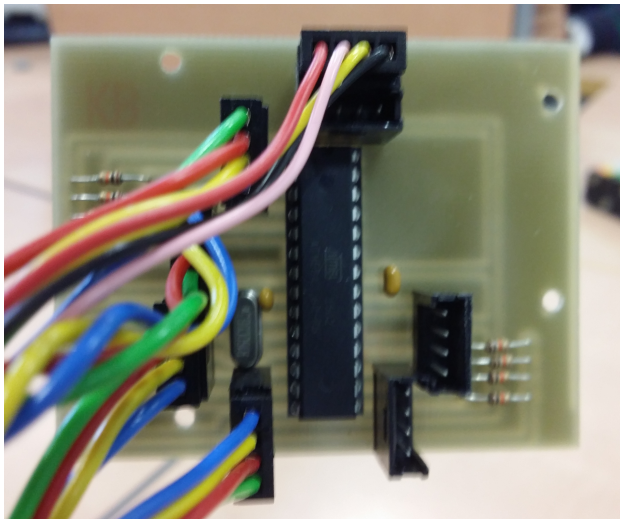


Figure 9: The rotary encoder board

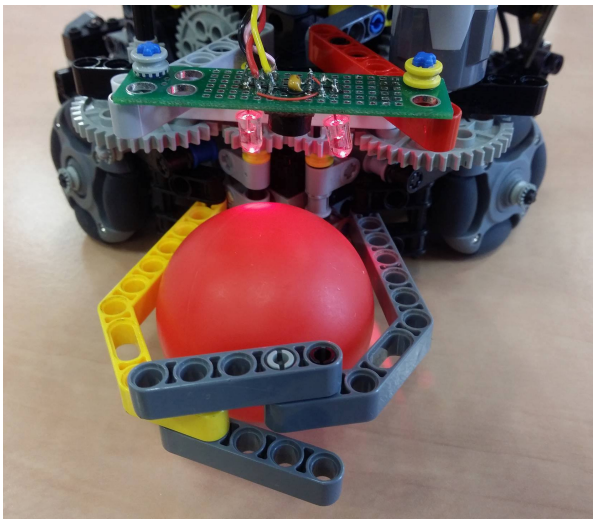


Figure 10: The light sensor with the ball

Each NXT motor is connected to the rotary encoder board which is depicted in Figure 9. The connections in the anticlockwise order from the top are: the I2C bus to the Arduino, back right motor, back left motor, front left motor, and front right motor. Using this board

the information about the amount of rotations the motor has performed since the last query is available for the Arduino code as a separate integer for every motor. Every 5 ms the board is queried whether the target value has been reached. After the average of the rotary values of all four motors becomes greater or equal to the target value, the motors are stopped.

4.2 Light sensor

The light sensor is located above the grabber as seen in Figure 10. It is used to check whether the robot has successfully acquired the ball after grabbing. The sensor provides an integer value that increases the greater the distance to the nearest object in its direct line of sight is. Then the Arduino compares the integer to a predefined threshold corresponding to the red ball. Sometimes a white line inside the pitch can be mistaken for the ball due to their similar sensor values. An IR sensor was also tested and its returned values were deemed less reliable than those of the light sensor.