Mindstorm Troopers Group Report

Group 6

April 24, 2014

The System Design Project (SDP) is a group-oriented practical, with the task of using Lego Mindstorms to play two-a-side football similar to table football (foosball). Our team, the Mindstorm Troopers, approached this task by dividing the work among four different teams Engineering, Strategy, Robot Programming, and Vision.

1 Organization

At the first meeting, a team leader was appointed by the group. After a week, a leader for each individual section was appointed by the team leader, based on initial involvement in the project. At first, the team met once a week officially to discuss progress and tasks. The group leader was responsible for assigning tasks to each team. In addition to this, as well as organizing meetings, the group leader worked with all four teams, and was directly involved in all aspects of the project. After allocation of tasks to teams, the individual team leaders were responsible for distributing the tasks within their teams. Our weekly meeting quickly expanded to multiple meetings as often as possible, so that team members could work on their assignments together, and progress more quickly.

Initially, our organizational system worked well, however with less guidance from the course organizer after Milestone 3, we quickly discovered that we needed our own methods of keeping track of progress. We implemented specific, short deadlines, which not only allowed for tasks to be finished as quickly as possible, but allowed for feedback from the rest of the team in case any more changes were needed.

At various times during the semester, task reallocation was necessary due to problems we encountered such as illness, delays, or imbalanced workloads. Tasks were reallocated as needed to ensure that work progressed as smoothly as possible. Naturally, this also involved changes in leadership of the four teams.

While our organizational structure worked well, there are things that we would have done differently if we did SDP a second time. In particular, we would prioritize work at the beginning differently. We evenly split up our group into four teams at the initial meeting, and for the most part, people stayed in those teams. However, we should have initially put more of a focus on the engineering, since until the very end of the course we didn't have a stable robot to test on. Had more people worked on the robot as the primary task, we would have had less delays in later stages, and wouldn't have had to rebuild the robot so many times.

2 Early Robot Designs

The very first robot was built according to the instructions from the Lego kit and two light sensors were mounted on it. After the first milestone the robots had to be reconstructed, the light sensors

removed and a kicker mounted. The kicker used rubber bands, the motor would push the leg back, stretching the bands, so that it would push the leg forward and make it kick the ball, strength of the kicker could be varied simply by changing bands. The defender was built to be wide in size, however the kicker was never mounted on it, since a new design was developed due to size constraints.

3 Omni-wheel structure

Omni-wheels were acquired to have robot mobile and easily turning. Three prototypes were developed and built: a standard 4-wheel and two 3-wheel robots. Triangular 3-wheel prototype was chosen based on the discovery of Omni-pilot class in java, supporting this formation. As the readily available sockets on the NXT were in use, an extra socket had to be activated. An old type motor was mounted, since additional extensions did not support the newer ones. The size, weight and weakness of the motor were problematic. Even though the robot was meant to have an advantage in mobility and speed, because of its size and high center of mass the robot failed to move around fast without falling. It was evident that the current design had too many disadvantages and practically no advantages therefore it was decided to change to an alternative one.

4 Later Robot Designs

The kicker was continuously changed to the teams specifications, including gearing, attachment of the kicker to the robot, the connection to the robot, and parts of the kicker itself. The holonomic movement was first designed with a triangular base, which we found to be too unstable, not robust enough, and with too high of a centre of mass. Then we moved on to a linear holonomic design, using three motors for movement, just as the triangular base, but all in a straight line, with two going forwards and one sideways enabling full holonomic movement while increasing the compactness, stability and robustness.

The chassis itself only served to hold the motors together and keep the NXT brick in place. However, it is vital to hold all three motors dedicated to movement completely in place, especially with holonomic movement, thus the three motors were secured at three different points of the motor to ensure no movement of the motors themselves.

We soon realized, we had problems with the standard ball bearing included with our kit – due to the ball bearing not being part of LEGO it could not be secured in place properly. Thus we eventually bought smaller ball bearings and connectors which allowed us to save space and increase stability as the chassis now lies on a rectangular base instead of a triangular one.

5 Robot code

The robot code was always kept up to date to the design. All of the code was written in Java using Lejos. Initially, the code was written to control a nonholonomic wheeled robot and had the move, turn, kick functions. At this point, the attacker and defender robots had different designs, therefore the classes contained constants for each robot, but afterwards, the design was unified for both robots and a single class with same constants was used for each robot. As the robot was upgraded to use holonomic movement, the code was also changed to adapt to this. The movement was changed to include an angle at which to move. The wheels were programmed on the 3 ports of the NXT brick, whereas the kicker was connected on the I2C board. Later on, catching command was added as a catcher was built on the robots. The principle of the code for executing commands

remained the same through all the designs: the commands were put into a list (which acted as a queue) and a new command was taken out after the old one finished. The reason for choosing a list was that we needed an abort command to flush all commands out of the queue. A queue would need to be fully read and then put back in before every command, whereas a list can be checked without removing the items. The turning and movement were done by going a certain amount of rotations by the motor, which was calibrated for robots individually (for the holonomic attacker and defender the constants were the same, because the wheelbase was the same, meaning the constants would be the same too).

6 Strategy (Early Stages)

We started working on strategy after Milestone 1, and from the beginning our goal was to produce reusable, easily extendable and highly modularised code. We decided that Strategy would become the linchpin that connected the various parts of the project so the first task we undertook was to develop concise methods for knowledge representation and passing information around throughout the project. For that purpose we worked closely with the vision team to create the world class, an object to comprehensively store our knowledge of the world as seen by vision and perform some preliminary pre-processing of that information to make it available at a convenient format for the rest of the strategy, hence functioning as an interface between vision the the rest of the project. To aid the cohesion of the project we developed two primary data structures (objects), namely Point and Vector, which together with some tertiary mathematical functions form our geometry package was developed mainly between Milestones 2 and 3 by the strategy team. The geometry and world packages together became the necessary foundation on which the rest of the strategy code was subsequently built upon. The packages perform two important functions: they ensure the cohesion of the project by providing a well thought out way of storing all the relevant data that strategy uses in objects specifically designed for the needs of the project. Secondly, they allow a layer of abstraction by hiding away all the mathematical functions and conversions necessary to process to information from vision and returning boolean predicates for the strategy to query.

The next stage was planning and deciding on the actual actions performed by our robots. This task presented some very interesting differences from previous years implementations. We observed that by not allowing contact between the robots and dividing the pitch in sections, we could treat the behaviour of the robot in each section as an almost independent closed world problem with the ball position being the only significant external influence. Thus we modeled the planning phase of our strategy system after a finite states machine. Our planner transitions between stages based on the position of the ball on the pitch, and in each state it develops a plan on how to perform a basic task such as intercepting or kicking the ball. We found that the FSA model had a number of desired properties: It is modular and easily extendable as states are mostly independent from one another and can easily be removed or added a needed; indeed the number of states varied throughout the development of this project. It was also beneficial from an organizational perspective as different people could be allocated to work on different states concurrently and it also allowed to add or remove the desired complexity in our strategy.

7 Strategy (Later stages)

Initially a full set of commands on performing a task (plan) was developed within each state, stored in a queue and transmitted over to the robot. This implementation however presented us with significant issues as plans were formed much faster than the robot could execute them and our communications were overflowed with unnecessary commands.

To solve this problem, we divided the strategy into two parts: goal and move. In the goal part, every second we update the current goal of the robot by analysing the data fed in from the vision and record this in a data structure Goal. Goal contains the point we want the robot to go, the behavior we would like the robot to take (e.g. catch and kick), together with two descriptions of the goal (isNull and isAbort). The isNull tells if the Goal is null – if it is, it will not be processed later. Meanwhile, the isAbort indicates if this Goal will abort the previous Goal, and instead replace it with the new Goal. After generating a Goal in this structure, we compare this new Goal to the last Goal being processed, this process is called filtering. In the filtering, if the new Goal we fed in is similar to the last selected Goal(e.g. point too close, doing the same operation), the new goal will be set to Null and will not be processed. Otherwise, the new goal will be set to be an abort goal which will abort the previous goal and process the current goal immediately. When a new goal is selected, it will be passed to the Move part. In the Move part, firstly the goal will be checked to see if it can be achieved (e.g.if the goal tells the robot to go somewhere outside the pitch, it will not be processed.). After checking, new commands will be added to the command queue and be sent to the robot later.

indentThis new structure nicely fixed the command overflow problem which appeared before before while still keeping a relatively quick reaction time.

8 Bluetooth and I2C

Our Bluetooth implementation is based on Group 4 from the previous year with the necessary changes to allow communication with two robots at the same time and to deal with our slightly different command structure. It is otherwise a fairly standard implementation using the standard leJOS API and the BlueCove library.

After the 2nd milestone it was agreed that we wished to move to a holonomic wheel design to allow for greater flexibility in movement. Requiring three individually powered wheels, we decided that we would use all three available NXT Motor ports to power the wheels. This decision was based on us initially planning to use the holonomic libraries (OnmiPilot) provided by leJOS that were not able to use anything other than the inbuilt NXT ports. This left us requiring at least one additional motor port for the kicker. As the NXT brick is I2C enabled, it was a viable option to create additional motor ports utilising an I2C multiplexer. The multiplexer is an electronic circuit board with 4 ports, each port controlled by 2 registers (one for speed, one for direction). These registers accept an unsigned byte value, this posing an, at first non-obvious problem - Java bytes are signed by default. As the register accepts values between 1 and 255, the optimal value for maximum speed as a signed byte is -1. -1 (1111 1111) as a signed byte is equivalent to 255 as an unsigned byte. The other key consideration with the multiplexer board is the amount of external power that should be connected by means of batteries. Insufficient power will at worst make the motor fail to operate, at best significantly reduce the available power and speed of the motor. We experimented with several different types of motors and battery types and power. Initially, utilising a 9v PPC battery we connected a Lego 43362 (Technic Mini-Motor) to the multiplexer. Testing showed that although this combination offered high speed (340 rpm) it did not offer the required power (No-load current 9mA) to exploit this speed meaning our kick was very weak. The standard NXT Servo motor was available for use but it was decided that this was too large to mount in an appropriate place on the robot, as well as not being maximal in either power (60mA) or speed (170rpm) irrespective of the amount of battery power. The Power Functions XL (PF XL) motor had the most power (90 mA) of any available to the team as well as being of reasonable speed (220rpm), two 4 x AA battery packs wired in series (combined 12v) allowed this motor to operate at its maximum potential.

9 Vision

We did not need to use vision up until milestone 3, however we did start researching last years projects and figuring out how it works, straight after milestone 1. We decided that apart from the code used to grab the frames from the camera we can write it ourselves from scratch with a fairly simple implementation. The initial approach was to loop through the pixels in each frame and take an average of the points for each colour as being the location of an object (ball, blue or yellow robot). The edges of the pitch were calculated in a similar way, by considering the white pixels on the frame to generate a rectangular pitch divided into 4 sections. Detection of the ball seemed like a good starting point, but when we got to the blue and yellow pixels the detection process was not accurate enough, therefore we introduced a function that enabled us to change the camera settings from within the code. We moved on to detecting the black dot on the plates in order to get the direction of the robots. This was done by detecting dark pixels within a close range of each robot location respectively. For testing the accuracy we drew the objects with different colours on the frame displayed in a GUI and monitored the behaviour. Moreover, this was our vision system for milestone 3: simplistic and straightforward.

After the third milestone we decided to implement a GUI, to help easily adjust to changing the pitch and variations of the lighting in the room. We picked the raw design from last years group 4 project and implemented what we considered useful. We noticed that most of last years groups used an algorithm for clustering, but we kept our naive and minimalist approach as it gave good results.

Our new GUI enabled us to manually adjust the camera settings (Brightness, Contrast, Saturation, Hue and Chroma Gain), and the thresholds for the coloured pixels in the RGB and HSV system, which can all be saved in a file and automatically loaded next time the vision system is running. The other thing that the GUI is very useful for is lets us select which pitch we are playing on, our team colour and the shooting direction. The GUI proved to be very helpful in debugging all elements of our project. Our new features (which are implemented in the final design as well), prompted us to improve the object detection. This time around we draw the pitch boundaries, by dragging a rectangle with the mouse on the screen, rather than detecting them, because we needed to consider the times when the robots or the ball get next to the white edges. Afterwards we go about detecting the green plates by using the HSV values, one in each quarter of the pitch. This way we are looking at yellow, blue and black dot pixels within the green plate. For precision, in several consecutive frames a plate will be detected in an immediate proximity of the plate detected in the previous frames. In order to make sure that the pixels our vision detects are actually on the green plate, we made it possible to set the each plates rough location manually, by clicking on its physical location in the image on the GUI. We also made it possible to view the detected pixels for each object, in order to help us out with testing.

Our vision system generates data that is passed on as elements of a World object. We use points and vectors for: the location on the ball, the edges of the pitch, location of the robots (both ours and the opponents), the direction of the robots, and the location of the goal with respect to where the leftmost and rightmost edges are.

10 Repository

`https://github.com/kubakaszyk/sdp.git`