Team Nine Paper

Introduction

MASLab contest over IAP presents the participants a difficult problem to solve—building an autonomous robot with vision to score points with balls. Even though the essence of the contest can be summarized into a one-liner, the complexity of the problem is on the other extreme. Much of the difficulty of the problem comes from the time and resource constraint. You only have four weeks to solve the problem at hand. Attending all the lectures and mock contests, participants realistically have only three weeks. Time is a big issue for this contest. This report will present solution that the team came up with after putting in roughly one thousand hours during IAP. Even though the result was not as expected, the Legend of Drunken Panda will continue.

Overall Strategy

The overall strategy of our robot dealt with essentially three different problems; how to allow our robot to navigate the unknown playing field and identify objects successfully, how to capture the red balls, and how to score in the goal area. The problem dealing with the navigation of the unknown playing field and identifying objects was by far the most complex and dealt with the integration of software, sensors, and structural design. The capture and scoring problems were ones which were mostly mechanical situations. At the beginning of the design process our team sat down and discussed a game plan to solve each of these problems.

It was inherent from day one that we would be using our camera to deal with identifying objects. The obvious color differences between balls, walls and goals quickly gave us a method of identifying one object from another. However, there was still the issue of depth perception and navigation. We decided that a combination of IR sensors could be used to give our robot a strong sense of how far away it was from a wall. On top of this we decided that we would have our robot scan the field with it's IR sensors to make a map. To navigate about the field we decided that we would use a combination of a gyro and quadraphase encoders to enable us to determine the changes in direction and distances traveled by the robot. To bring all of this information together in an efficient manner it was decided that our software structure should make use of a multi-threaded behavioral model control system.

To solve the issue of capturing the red balls it was decided that a mechanism that could pick up the balls continuously. The hope was that this would cut down on the time spent capturing the ball. To accomplish this it was decided that a belt system would be used to raise the balls up a series of ramps and into a storage level. This belt would continuously run so that the robot simply had to run over the balls.

To maximize our score, the balls were to be put through the field goal. To do this a ball shooter was designed. However, if the ball shooter mechanism proved to be incapable of

providing the forces desired, then a chute would be used to allow the balls to roll through the mouse holes.

Mechanical Design

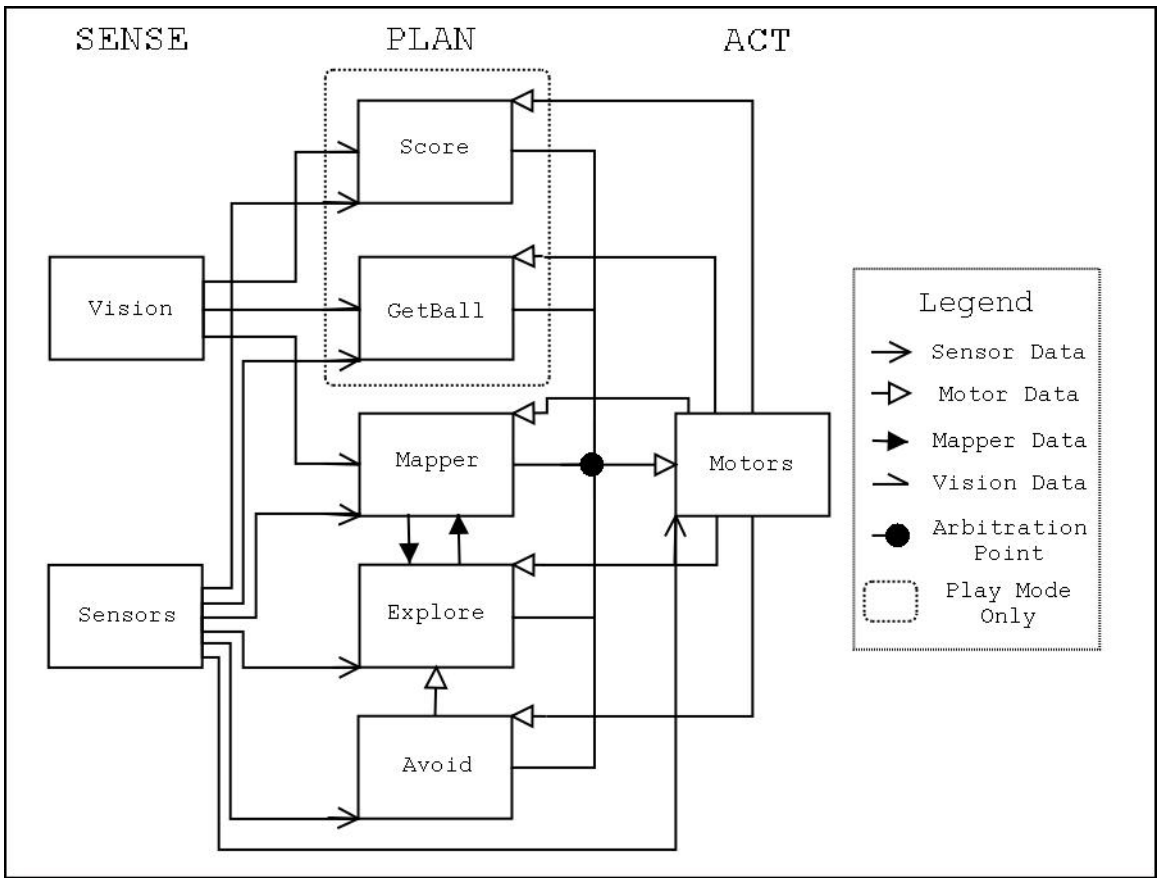
For our robot, we planned on integrating various mechanical modules to create an interesting robot. The idea behind creating mechanical modules is just like the idea of creating modules in our programming. Each could be independently modified, and the performance on a particular module could be tested before all the rest were un and running.

Our first most interesting idea was to try and create a ball shooter like in a tennis ball launcher or like the spinners found in a Matchbox car set. To create this module, we first realized that we would need a high speed motor in order to spin the wheels fast enough to accelerate a ball between them. The difficulty in creating this module is that it was hard to prototype. With our other modules, we could make a quick inefficient models of what we wanted by using the drill press and modifying some pieces of hardboard, but with the shooter, inefficiencies would quickly stall the high speed motor we needed to shoot a ball. Once we acquired a motor and some wheels to use for the shooter, we carefully designed all the gear spacing and the structure that we would need to act as bearings for the shafts. The whole design went through two iterations before we had shooter wheels that had almost frictionless turning and a properly coupled motor. Once the module was completed we gauged it at being able to shoot at ball a little over 10 ft/s. Sadly, since we finished this module only four days before contest, we didn't really have the time to mount it properly to our robot and use it in the contest.

Our second most critical module was a roller assembly that used a belt to pull balls up to the top of our robot. This module, unlike the shooter, was easy to prototype. Once we had our basic design, we made rollers on a lathe to dramatically better our rolling, and then we made a housing out of sheet metal by drilling some holes our using a mill. The roller assembly encountered the classic problems that come with all belt assemblies: drift and a lack of belt tension. Due to a lack of materials, we couldn't make the belt out of a nice rubbery material, so instead we used some backward duct tape stuck to itself. The choice of material was not ideal because the tape had some viscoelastic effects, but as a tradeoff, the stickiness proved useful in gabbing balls and carrying them up the belt. On competition day the belt itself ran into a completely new set of difficulties and ended up sticking to itself in an unexpected way.

Software Architecture

Our original plan for the software was to implement a multithreaded behavioral model control system, where sensor data would be fed into a set of behavior modules, which would then output information and instructions to each other and to actuator modules. As the code evolved, however, the structure of the modules came to resemble the classical sense-plan-act model, as illustrated below.



Each module was a self-contained Java thread responsible for a single behavior. To communicate with other modules, we designed a thread-safe Edge class for storing arbitrary Data Packet objects; each module had a list of incoming and outgoing edges, from which they could read or post information. For example, the Sensors module had no incoming edges, but had outgoing edges to nearly all of the other modules. Every 100ms, the Sensors module would poll the robots sensors and write a new SensorDataPacket to each of its outgoing edges. Other modules could then check if their incoming sensor edge had changed, and could read the information and process it, likely resulting in the posting of some new information.

This software model had the advantage of being very easy to test and to extend. It was easy to test because each module could be individually tested using JUnit software by simply supplying false inputs to the module and then reading its outputs. To extend the software, you just needed to wire the new module into the system as if adding a component into an electrical circuit. An example of the flexibility of this system was our implementation of our Exploration and Play rounds. In the Exploration round, we simply removed the GetBall and Score modules (and their corresponding edges) from the system; this way, our robot would still Avoid, Explore, and Map, but it would ignore balls and goals. Then, during the Play round, we added these modules back in.

The only real difficulty with this software model was the heavy threading it required. Our team had limited experience with Java threading, causing us to write code which was

likely not completely thread-safe. As a result, we sometimes found we could "fix problems" by polling the sensors less often (to reduce the likelihood of race conditions), and our log file often had statements out of order, making debugging more difficult.

The other notable feature of our software was our exploration and mapping code. To explore the map efficiently, we would have our robot execute "exploration circles," where the robot would spin 360 degrees while recording data from the front-facing IR sensors and camera. This data could then be filtered to produce a local occupancy grid around the robot. Using a heuristic, we could then determine which direction the robot should travel in next, favoring directions which seemed to be more open and which were farther away from previously visited points. We also stored this local occupancy grid into a global grid, allowing our robot to build a map of the environment. This map was combined with feature information from the camera, enabling the robot to request the shortest path (determined using A*) to the nearest goal or ball.

Vision Software

Simplicity and speed is the design motto for the vision portion of the Panda. The main class files are: PandaVisionModule and various Feature classes (Red ball, barcode, and goal). PVM is responsible for capturing images from the camera, locating the

features, and estimating distance and bearing. All these information will be posted to other interested modules such as Score and GetBall in the form of VisionDataPacket.

First step in improving speed is using the underlying data buffer of the images. Calling image.getRaster().getDataBuffer().getData() returns an integer array that is compact and fast to access. Traversing the array is really fast as compared to using the getRGB method of BuffereImage. The final version of the PandaVisionModule can process an image of 160x120 pixels in 30 milliseconds. With this kind of speed, the software can either run image analysis frequently or save computation time for other parts such as threading.

Feature identification relies on color identification. Various thresholds are determined to distinguish red, green, black, and so forth. Then, it is just a matter of traversing through the array to determine the region of interest. PandaVisionModule uses the expanding rectangle approach as resented in vision lecture to identify interested color patches. That enables the Panda to see multiple balls, barcodes, etc. Estimation of bearing and distance is solved by fitting exponential graphs to data collected from the camera. Blue line filtering was also implemented using the same tricks in feature identification.

The final version of the PandaVisionModule excelled in redball identification and blue line filtering. However, goal and barcode identification weren't solved due to time constraint. Though they could be properly identified in the analysis all the time, extracting useful information from them wasn't perfect. Those information were right or left goal post and reading from an incomplete barcode.

Conclusion

Time worked against the team throughout MASLab. It proved impossible to do design iterations as taught by the institute. The team did all it could to finish the robot, but much of it was still untested on contest day. All in all it was a good character-building experience. The hours were long and there was little support from the staff besides replacing defected parts. If anyone is looking for a good hacking activity for IAP, MASLab is it.

Suggestions

For the future participants—you can forget about those good design iteration principles you learned from engineering classes because you don't have time. This contest isn't about building a perfect robot by the end of IAP. It is about presenting something that works. It might not be perfect, but at least it is functional. That goes for both hardware and software. Hardware needs to be done a week before the contest so the software can be tested to work out the kinks. You should build only two robots, the PegBot and the final one. And of course, if you are a genius and hardcore hacker who doesn't sleep, please ignore all the above suggestions.

For the staff—please be more organized. Putting stuff on WIKI doesn't mean you can run the course remotely. It takes personal contacts to announce key information. Also, move the guest lectures into the first week since most people need the time at the end to finish the robot. The mock contests are waste of time since you only get to run your robot once in three hours. What kind of productivity is that for testing? Also, just make the lectures online since they are pretty self-explanatory. That's my two cents.
