Preserving Line Of Sight While Panning Robotic Problem

Cyrus Thompson and Matthew Piorko
Department of Computer Science, Northeastern University
thompson.cy@husky.neu.edu, piorko.m@husky.neu.edu

Abstract

Visual observations in a real world environment often have the primary objective of perceiving a singular thing, such as an object or a place. Yet many perception based equipments such as security cameras have the ability to move. Thus to use this ability without violating the equipments primary objective, namely preserving line of sight, panning is required. This paper highlights that a perfect solution to this problem does exist assuming there is perfect knowledge of the environment and that there are fairly effective fast solutions to this problem without any knowledge of the environment. Accounting for the environment and other requirements will determine which solution should be used.

Problem Statement

As the use of robotics in products increases, the abilities of said products are also increased. Yet while a security camera may gain the ability to have the field of movement equivalent to a robotic arm, its purpose is still observation, and usually of a specific point of interest such as a stairwell or a door. In order to to allow the camera to retain its old functions along with its ability to use its new field of movement, a solution to preserving line of sight while panning is needed.

While it is true that preserving line of sight while panning as a robotics problem has many applications, such as a retaining a laser based connection to a specific point while moving, the pet example that shall be used in this paper will be the previously mentioned security camera with a local field of movement. The reason for this is that we authors believe that this will be the most widespread use of this problem especially in the immediate future.

The environment and the point being panned across also makes a difference. If the environment and the point are static, then one can precomputate every state to determine if it has line of sight is possible. If the environment is the only thing static then the knowledge of the obstacles is possible and thus a traditional inverse kinematics and pathing with special configurations that will be later in detail is possible. Yet if the environment and the point are dynamic any traditional algorithm would be ineffective and an online algorithm would be required.

Originally when approaching this problem we looked into any potential state of the art solutions. Surprisingly, there seems to be a lack of work in this area. For many specific applications there were specific solutions, however there was only one general solution found and this was to simply pre-calculate all paths that the camera could follow. While this takes a lot of upfront computation, this results in paths that can be easily run. The number one difficulty with this strategy is that in a dynamic environment these paths will sometimes break line of sight, and since information about every path must be stored, is also incredibly memory intensive. Yet the algorithms that we experimented with in the static environments could be used to generate the paths for these pre calculation strategy in addition to dynamic environments. Upon concluding the experimental portion of this project it would seem any general solution, including ones discussed in this paper, is simply too slow, assuming one is working in a real world dynamic environment and requires a low chance of error. This is simply because obstacles in a dynamic environment can move, and thus any precomputed path might not be valid at the time of execution. Additionally, recalculating the path at every step is too slow to keep up with a constantly changing system.

Approach

Since this paper will be covering multiple solutions there is more than one algorithm that requires some background information to be explained. To begin at the simplest, the term bug algorithm will be used to describe the approach that was most effective in the completely dynamic environment and simply means an algorithm that only has local knowledge and reacts simplistically. In all others cases the initial requirement for the algorithm is pathing. In these cases we will be using primarily the variations of two algorithms, sPRM and sRRT, in addition to the relatively simple bug algorithm.

Any solution of this problem requires that, at any point within a path between two points in configuration space, the camera is always pointing at a certain point. The two possible approaches are planning before taking any action, or simply attempting to move in a direction. The first solution we looked at was planning and required custom edits to the existing sPRM and sRRT.

The first algorithm we attempted is sRRT. When we began experimentation we attempted to keep everything within the configuration space so we could use the existing sRRT, only changing the method of vertex selection. We were able to do this by modeling the line of sight as a prismatic joint. Thus, for any pathing algorithm, all that needs to happen is the end effector (placed at the end of the prismatic joint) must stay on the target in order for the line of sight to remain. Therefore, any pathing algorithm with that restriction can use this model assuming the environment is static and will output a proper general solution for this preserving line of sight while panning robotic problem. The specific sRRT implementation that we created changes the way the vertices are screened. Every time a vertex for the randomly growing tree is proposed, the robot uses forward kinematics, specifically using sampled vertices that represent configurations to check whether the end effector on the prismatic joint has not changed location and is still on the point. In our implementation sufficiently close to the point seems to work better then exactly on the point. A secondary thing that this implementation requires is the ability for the prismatic joint to move faster than all other joints can move it combined. This is because the location of the end effector does not change but the configuration of all other joints can as long as they keep the prismatic joint pointing at the target. This means the algorithm must allow for the end effector to move in any direction as long as the angle hits the target point. This effectively means the prismatic joint must accommodate by extending and retracting to the appropriate length. While this method works there are a few problems with this that will be covered in the experimentation section.

The second attempt in pathing was sPRM. One simply samples the environment for valid vertexes and creates a vertex for the current configuration and the final configuration, then where there is a possible unimpeded connection between two close vertices an edge is created between them. This is repeated for a number of iterations to create a full graph. Once this graph is created, assuming there were enough sampled points, there will be a path from the start configuration to the goal configuration.

Again, we compute samples from the state space and plug that in to an existing sPRM solver. The algorithm behind this is that every iteration draws a random sample, as usual. The nearest neighbors in state space are then checked for collisions, and if possible, a connection is drawn. This generates the r-disc graph that we are familiar with. Similar to sRRT, the only practical difference that our implementation has is how to model the line of sight. Again, a prismatic joint is used, and the prismatic joint is required to be ‘pointing’ at the target in order to be a viable sample. Here, the length of the joint is also sampled, so that the end effector can appear at the correct position.

The final pathing algorithm was a variation of RRT designed to execute faster than the previous two attempts. It did not model the camera and instead assumed the camera was able to view in a 360 degree range. The very common spherical security camera design has this feature, and it allowed for quicker planning, so we it is effectively a requirement for users of this algorithm to have a 360 degree range. Since many modern security cameras have this feature, this algorithm will have uses in these cameras. The algorithm itself did not use a prismatic joint, and instead checked whether the end effector had a straight path to the goal. This is equivalent to checking that it has a possible line of sight.

The final approach we tried was a reactionary bug algorithm. We decided on a perception based approach because of its speed and effectiveness in dynamic environments where obstacles are not constant in either count size or location. Specifically, we used the pre-trained neural net, alexnet, and fine tuned it into a binary classifier. The two classifications we trained it to identify were target and not target. Put simply, this algorithm would be given a rgb image and would tell if the target was in the image or not. If the robot, at that position, had line of sight and could move to that position that did not, it would not move to the location. While this may be simplistic it was quite effective. Conversely, if it did not have line of sight, it would move to try and get line of sight.

Experiments And Comparisons

For all implementations and experiments matlab was the language and environment used. The Peter Corke's Robotics Toolbox was the primary robotic package used.

The Prismatic sRRT was the first implementation. An arbitrary valid initial configuration was selected and a goal configuration was selected and two obstacles were placed.

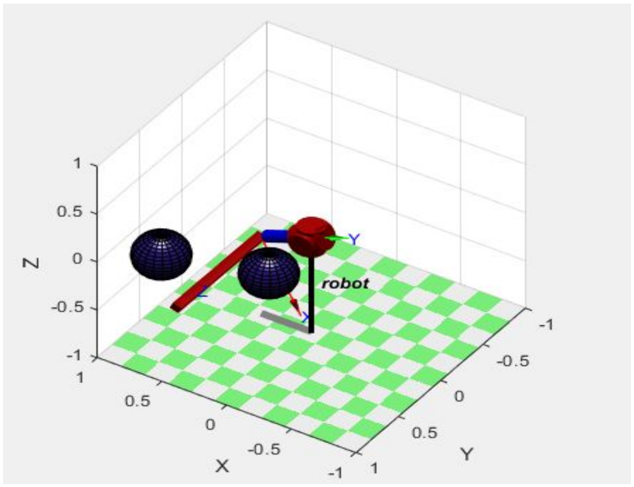


Figure 1. The final environment for the Prismatic sRRT

While in theory using arbitrary points for the start and end states is valid, in experimentation many of the inputs simply ran forever. Yet even in the successful cases, the run time of this was multiple minutes on a state of the art desktop. I believe this is because the number of sampled vertices is required to be very large, and the inverse kinematic and edge checks simply take too to calculate long per sample.

The second attempt in pathing was using sPRM. Similar to sRRT, the initial state is seeded as random with some obstacles and the target in the environment. However, the runtime was again not manageable. Although a solution could be found if sufficiently few vertices were sampled, a realistic run would require many more samples to counteract the large amount of obstacles in the world. By allowing the computer to run for a lengthy amount of time, a reasonable solution was found. The cause of the excessive computation is again edge checking, as with sRRT. Interestingly, calculating whether samples were valid was not too computationally expensive, although many samples were rejected due to not 'pointing' at the target.

The final pathing algorithm was another type of RRT designed to be much quicker than the previous two attempts and it was. This algorithm consistently found a solution in under 10 seconds. This is simply because it had a much wider range of samples it could accept because it no longer had to worry about the end effector being on target it only checked that the end effector had a possible line of sight to the target.

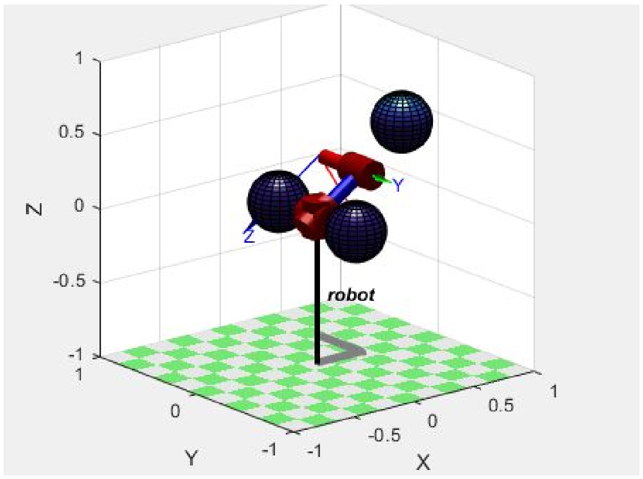
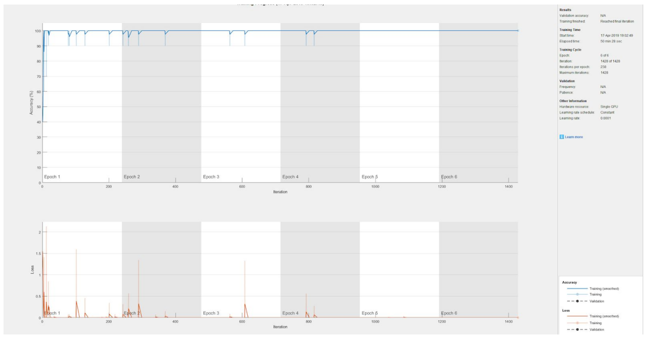


Figure 2 The final environment for the non prismatic RRT

The Reactionary Perception based bug Algorithm was very successful. We began with alexnet and then did tanspher learning (also known as fine tuning) to create a binary classifier which decided if an image was a target or not. Specifically, we decided that, in light of the fact that security cameras were our pet example, we used pictures of a hallway as a target and pictures of wall and corners and other non hallway things as non target. We generated a labeled image database ourselves from virtual three dimensional space which totaled over 2,000 images.



Results
Validation accuracy: N/A
Training finished: Reached final iteration

Training Time
Start time: 17-Apr-2019 19:02:49
Elapsed time: 50 min 28 sec

Training Cycle
Epoch: 6 of 6
Iteration: 1428 of 1428
Iterations per epoch: 238
Maximum iterations: 1428

Validation
Frequency: N/A
Patience: N/A

Other Information
Hardware resource: Single GPU
Learning rate schedule: Constant
Learning rate: 0.0001

Learn more

Accuracy
Training (smoothed)
Training
Validation

Loss
Training (smoothed)
Training
Validation

Figure 3. The training of our binary image classifier which as shown 50 minutes took to train.

After the image classifier was trained, we created a shell program that was easily able to output quickly whether a given image is a target or not. The result from both the system testing as seen in figure 4 and our testing showed an almost perfect accuracy at almost instantaneous speeds when trained. Combined with a class that allowed for screenshots to be taken, this allowed us to take simulated input from what was currently on screen and then decide which way to move based on whether it could see a target (in this case, a hallway) or not.

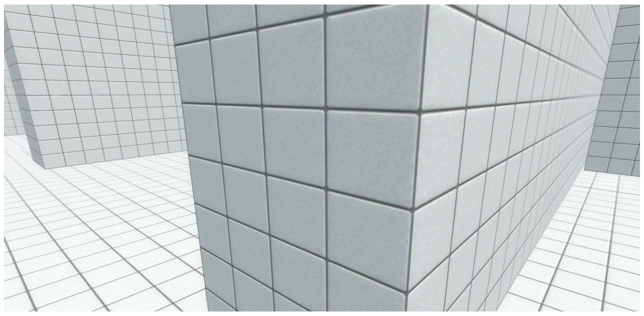


Figure 4. Example of an obstacle image used in training

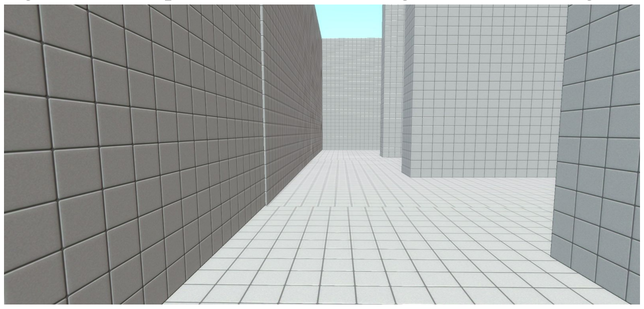


Figure 5. Example of a hallway image used in training

In comparison between the perception based algorithm and the prismatic planning algorithms we must first address that they are not exactly the same problem. The pathing algorithms plan before the robot moves, while the perception based algorithm is reactionary and controls the robot at run time. Yet in the end they should both result in the robot moving without losing line of sight on some target. In theory in a static environment they are both a hundred percent successful at doing this. Yet the pathing algorithms will be able to get from one location to another much quicker at runtime since it knows where it is going, while the reactionary algorithm has no such path. On the other hand, in a dynamic environment where obstacles are not consistent, a planned path would often result in failure if followed. This could be remedied if a new path could be created at every timestep, but sadly the slow run time of the path algorithms does not permit this. Even for the quick RRT, the runtime takes a few seconds, and in that time the path that it was creating might become invalid. The perception based algorithm on the other hand is quick and would work no matter how inconsistent the environment was. Thus we would say that in a dynamic environment, the perception based reactionary algorithm is a better choice hands down. The difficulty for both of them is that they both take a large amount of resources before the first use. The perception based algorithm required to be trained on the environment and possible obstacles, which on one of our machines took an hour to collect the images and an hour to train the images. Every time the sRRT and sPRM algorithms are used it simply takes a long time to execute, and it is not even consistently long due to its randomness in selecting samples. However, quick RRT is not too expensive compared to other methods, but this algorithm has the difficulty of assuming that the camera can rotate on its own, which is not always true.

Results From Additional Experiments

When it comes to practical implementations, both the pathing algorithms and the perception reactionary algorithm have potential. For the pathing algorithms our experiments showed that actually trying to create a path during robot operation is simply too slow, although it will eventually work. Thus in implementation, what likely should be done is the modern day standard of simply running the algorithm many times beforehand, then the algorithm will return a path and can simply cache the paths. The fact that the pathing algorithms have an element of randomness in sampling means that a slightly different path may be returned each time, and if they are all saved, then reaching almost any valid point will be possible through one of these paths. When it comes to the perception based algorithm, the high accuracy is good but it is not very efficient, especially in the static case, due to the random movements which will eventually get it to any valid location. Thus using the reactionary algorithm as an input to a search algorithm would be more efficient. For example, using A* to build a graph and go in a general direction towards a goal is possible, and while we did not implement A* to prove this, if we were to treat A* as a black box one is able to give it all the input that it requires to run. This can be done by dynamically creating a graph based on the results of the algorithms perception, namely, creating an edge if you can keep line of sight while moving to a point; and don't create an edge if you cannot.

Conclusion

Our overall conclusion is that the native algorithms (sRRT and sPRM) were not effective in this domain when trying to model the general case. Although they could find solutions in our test environments, these environments had few obstacles and still took an excessive amount of time to calculate. It would be difficult to convert these algorithms into online versions due to this constraint. In our pet example, the camera would not have sufficient time to maneuver around people moving in front of the door due to this computational overhead. The quick RRT, while much quicker than the other pathing algorithms, makes an assumption about the camera's field of possible view and is thus not a general solution. The bug algorithm was shown to be effective in dynamic environments but less than ideal for more static environments. Thus, in the end, the environment and equipment determines the algorithm that should be used in the Preserving Line Of Sight While Panning Robotic Problem.