

---

# Autonomous Mapping and Navigation Through Utilization of Edge-Based Optical Flow and Time-to-Collision

Madhu Krishnan, Mike Wu, Young H. Kang, and Sarah Lee

---

## Abstract

This paper proposes a cost-effective approach to map and navigate an area with only the means of a single, low-resolution camera on a “smart robot,” avoiding the cost and unreliability of radar/sonar systems. Implementation is divided into three main parts: object detection, autonomous movement, and mapping by spiraling inwards and using A\* Pathfinding algorithm. Object detection is obtained by editing Horn–Schunck’s optical flow algorithm to track pixel brightness factors to subsequent frames, producing outward vectors. These vectors are then focused on the objects using Sobel edge detection. Autonomous movement is achieved by finding the focus of expansion from those vectors and calculating time to collisions, which are then used to maneuver. Algorithms are programmed in MATLAB and JAVA, and implemented with LEGO Mindstorm NXT 2.0 robot for real-time testing with a low-resolution video camera. Through numerous trials and diversity of the situations, validity of results is ensured to autonomously navigate and map a room using solely optical inputs.

---

## Keywords

Autonomous Mapping and Navigation • Smart Robot • Horn–Schunck’s optical flow algorithm • Sobel edge detection • A\* Pathfinding algorithm

---

## Introduction

Unmanned robotics optimizes human time and effort tremendously and effectively has become the epitome of efficient robotics systems. One of the numerous problems autonomous robotics focus on solving is mapping and navigation. People have been trying to utilize the accuracy of

robotics to complete such tasks mainly with radar transmission. Efforts with this type of detection have led to successful results advanced as unmanned vehicles that can drive without collision (Guizzo 1). However, many of these methods are unreliable or expensive—unappealing to the general public as well as less developed areas in the world. Specifically, radar-based applications rely solely on emitting waves rendering them susceptible to interference. Radar also cannot take advantage of other multiple data inputs such as color and texture.

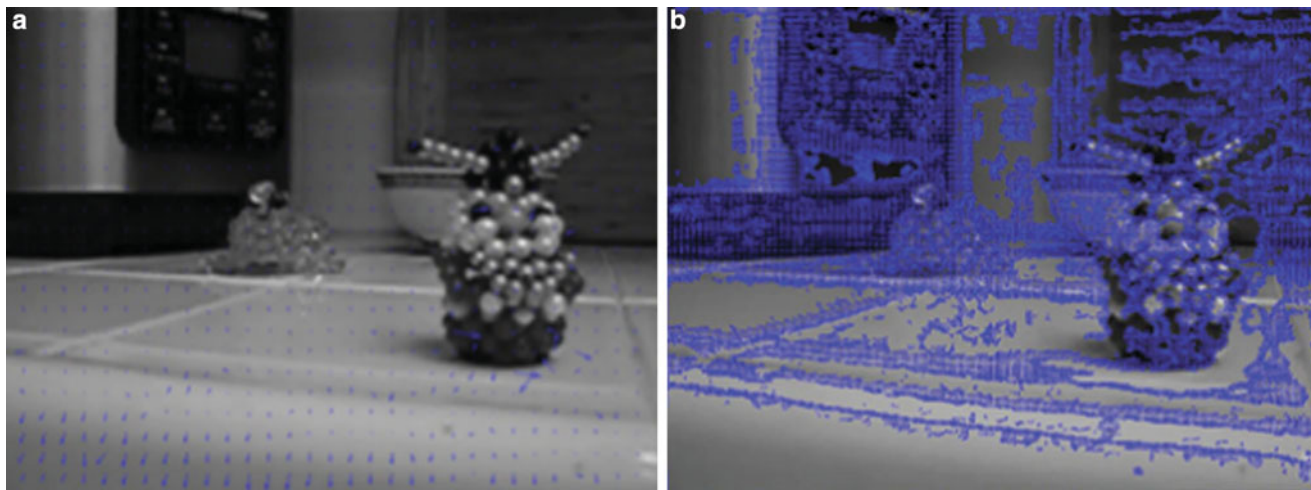
Other solutions for the mapping problem such as using the Sharp IR Range finder or Roomba also prove ineffective. The Rangefinder cannot be used by itself with the objective of mapping a room especially due to its thin beam width. The Roomba’s method for touching objects to maneuver and store data is even slower and more inefficient for the mapping problem. As the field of automated robotics endeavors to create advanced solutions to more complicated

---

M. Krishnan  
University of California, San Diego, CA 92093, USA  
e-mail: [mvkrishnan@ucsd.edu](mailto:mvkrishnan@ucsd.edu)

M. Wu  
Yale University, New Haven, CT 06520, USA  
e-mail: [mike.wu@yale.edu](mailto:mike.wu@yale.edu)

Y.H. Kang (✉) • S. Lee  
Torrey Pines High School, San Diego, CA 92093, USA  
e-mail: [hoonkang95@gmail.com](mailto:hoonkang95@gmail.com); [hye.in.sarah.lee@gmail.com](mailto:hye.in.sarah.lee@gmail.com)



**Fig. 1** The edited Horn–Schunck optical flow

issues, equipment that can obtain more information from the external environment becomes more desired.

Thus, this paper proposes an alternative approach to the mapping problem, one that is cost-effective and available to those in less developed countries if needed. Instead of shouldering the heavy cost of radar/sonar systems and equipment, it uses a robot attached to a single, low-resolution camera to obtain more accurate data from the environment and autonomously navigate and map the terrain. Processing vector images, it carries out calculations for object detection through Horn–Schunck optical flow algorithm and responds to those detected objects through time-to-collision induced reactions. The paper endeavors to create a “smart robot” that will respond to any given situation and decide by itself accordingly, creating a much simpler solution to the problem at hand. Through algorithms and video processing (ideally in real-time), the robot travels given the initial direction of the object solely from optical input while avoiding all objects until the arrived goal is reached and a complete map is obtained. The implementation can be split into two main steps: *navigation and mapping*. We first explain the navigation algorithms for object detection and autonomous movement in section “Navigation Algorithms” and then, a map-building method in section “MAP-building Method”. Experimental results are given in section “Experimental Results”. Finally, concluding remarks are given in section “Conclusion”.

## Navigation Algorithms

### Object Detection

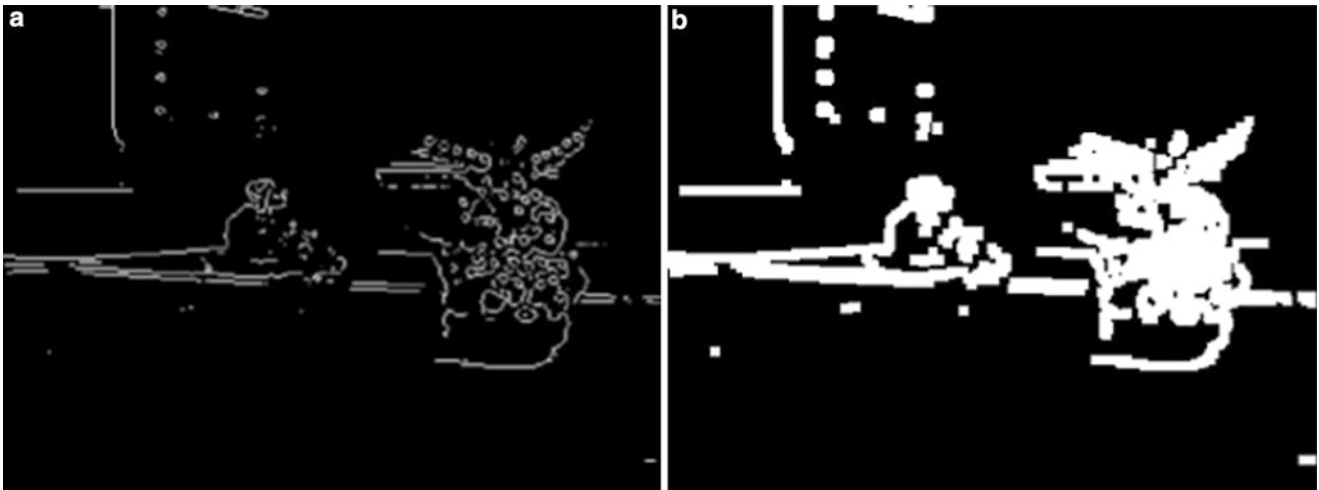
A cheap video camera is used to provide optical input, keeping the end product convenient and more importantly, cost-effective. Then taking black and white converted image

frames from the camera, Horn–Schunck optical flow is modified and applied to subsequent frames. The following optical flow equation is used:

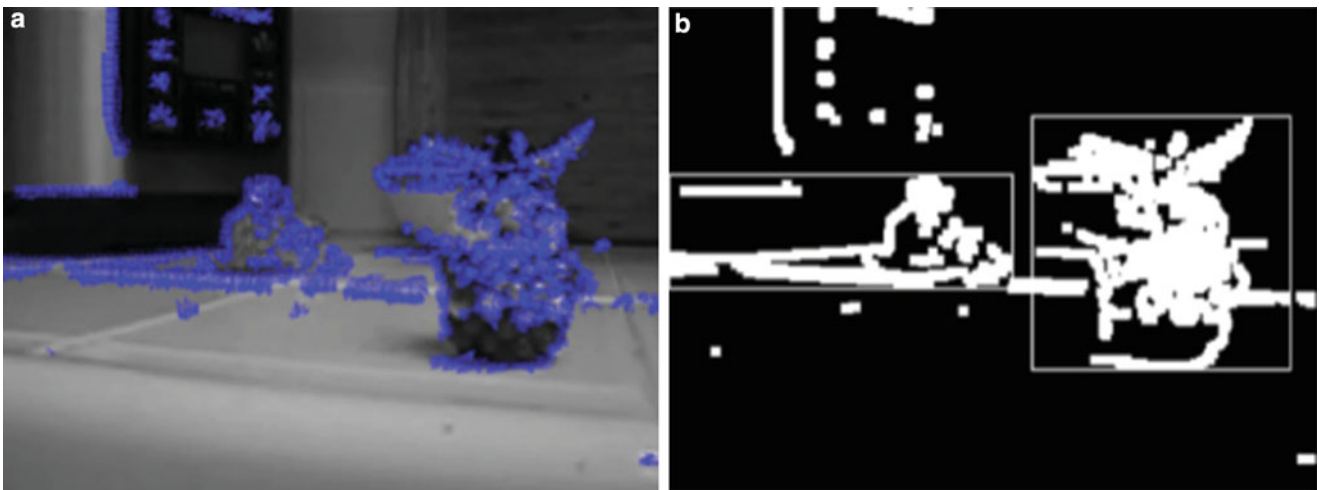
$$E = \left[ (I_x u + I_y v + I_t)^2 + \alpha (|\nabla u|^2 + |\nabla v|^2) \right] dx dy \quad (1)$$

where  $\alpha$  is the constant that controls the smoothness of the pixel movement,  $I_z$  is image derivative with respect to  $z$ ,  $u$  and  $v$  stand for the flow vectors. The modified version traces each pixel’s specific luminance factor onto the next image frame at time  $(T + 1)$  based on image intensity derivatives. The 2D optical flow vectors,  $u$  and  $v$ , are then used to calculate pixel motion and generate a gradient of motion vectors between subsequent frames—the vector length representing the distance traveled by the pixel. Figure 1 illustrates the edited Horn–Schunck optical flow. As the object is approached and “grows” bigger, the product of optical flow is a picture with groups of pixels that represent the outward movement of vectors. The high density of flow vectors compensates for missing vectors in homogenous objects since they are made up for by their surrounding pixels. Though Horn–Schunck optical flow does optimize accuracy, however, it becomes difficult to distinguish anything in the field due to the dense optical field in which all the vectors surround both actual objects and the background. Thus edge detection is needed to separate the noise from the objects by distinctly tracing the edges of the objects and ignoring the unimportant details.

Given the plot of optical flow vectors, edge detection altered with the Sobel operator is then applied to the image frame, which outlines objects and simplifies the images. Sobel focuses specifically on the center of the image frame effectively focusing more on immediate objects present in the image frame rather than the background. This function caters to the needs of the robot to quickly and accurately



**Fig. 2** Sobel edge detection and enhancement for the visibility



**Fig. 3** Edge detection algorithms: Illustration (a) The vector-tracing algorithm (b) The box-blobbing algorithm

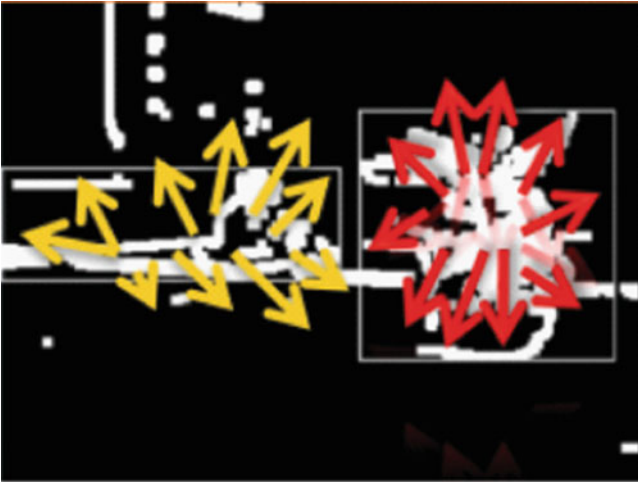
identify objects to avoid. The actual edges are then widened by 10 pixels, functioning as a “bolding” action (see Fig. 2). The purpose of these procedures is not only to make the objects more noticeable with wider edges, but also to lay down the foundations for the next progression of noise minimization: vector clusters.

After bolding object edges, a vector-edge tracing algorithm overlaid the optical flow vectors onto the edge-detected frames. As a result, only the vectors lying inside the 10 pixel width outlines were shown (see Fig. 3a). The fusion creates distinct blobs of vectors most likely to be associated with objects as the Sobel and tracing algorithm ignores the background. Because the clumps of flow are certain pixels away from each other, it becomes possible to define distinct blobs in comparison to one conjoined cluster. A blob-boxing algorithm is then used to separate the vector clusters with boxes based on an 80 % overlap threshold (see

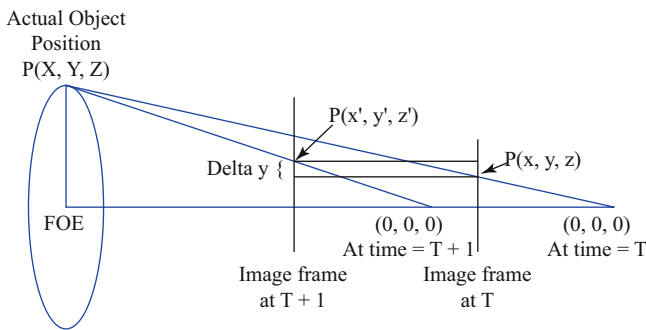
Fig. 3b). With this algorithm, the number and relative column locations of the objects in the image frame can then be extracted. Thus, this code effectively minimizes the processing of distracting details while the general image of the object stays intact. Overall this method increases productivity of optical flow since the motion vectors now focus only on the objects in an organized fashion.

### Autonomous Movement

The blob-boxing algorithm essentially provides information to calculate Focus of Expansion (FOE), a crucial input needed to ultimately find time-to-collision. FOE is the source of vector expansion as the video camera moves closer to an object (Fig. 3). As mentioned previously during optical flow, as the object grows, the optical flow motion vectors



**Fig. 4** Box frames to calculate the multiple FOE's



**Fig. 5** Time-to-collision's use of comparing equilateral triangles

expand outwards. FOE is the origin of those vectors. However, because of the possibility of multiple objects in one image frame, the boxes created in the blob-boxing algorithm become separate images frames from which FOE can be calculated individually. So with multiple objects, the vectors specific to each individual object in its own respective “box frame” are averaged to calculate numerous focuses of expansion (Fig. 4). These values are crucial as variable  $y$  and  $dy$  in time-to-collision calculations as  $dy$  is the change in distance from FOE per frame change and  $y$  is the vertical distance from FOE.

The time-to-collision (TTC) code is the core algorithm that allows the robot to move autonomously using only optical inputs. It calculates the distance in frames until collision with a stationary/moving object at a specific time without knowing the robot's speed; it does not calculate real measurements but enough for relative comparison. Referring to Fig. 5, the equation for time-to-collision is a comparison of equilateral triangles:  $y/z = Y/Z$ . Though  $z$  actually depends on camera specifications, it is assumed as 1. The origin lies on the  $z$ -axis, and  $x$  and  $y$  are based on pixel “ $p$ ”.  $P(X, Y, Z)$  is the coordinate of Focus of Expansion (FOE), the

origin of the vectors, but it remains unknown since it is the coordinates of the actual image.

Since  $z = 1$ , the equation turns into  $y = Y/Z$ . Taking a derivative with respect to time yields the following equation:

$$\frac{dy}{dt} = \frac{1}{Z} \cdot \frac{dY}{dt} - \frac{Y}{Z^2} \cdot \frac{dZ}{dt} \quad (2)$$

Assuming that  $P(X, Y, Z)$  is stationary,  $dY/dt$  in (2) can equal 0, allowing for  $Y$  to be substituted by  $yZ$ . Then, the result is as follows:

$$\frac{dy}{dt} = -\frac{y}{Z} \cdot \frac{dZ}{dt} \quad (3)$$

The last step involves dividing both sides in (3) by  $y$  and taking the reciprocal, which finally leads to the following equation for time-to-collision:

$$y \cdot \frac{1}{\frac{dy}{dt}} = -Z \cdot \frac{1}{\frac{dZ}{dt}} \quad (4)$$

Since  $dy$  (change in distance from FOE per frame change) is known,  $dy/dt$  in (4) can be found by comparing the pixel vertical movement of two frames, resulting in a ratio that should equal the negative of the actual movement on the  $Z$ -axis. So finally accumulating all the information from the previous codes, the focus of expansion (FOE) can find the time-to-collision—more importantly, without any distance or speed information. Rather, for the whole process, only two points are needed, making this algorithm remarkably efficient especially when installed in a reaction system. All these calculations are verified and simulated with MATLAB.

Cumulating all the above source codes, specific hardware for robot real-time testing was created. The robot was made from LEGO Mindstorm NXT 2.0 robot with a low-resolution video camera. Because the robot was programmed with Lejos in Java, a portal was needed to pass MATLAB time-to-collision calculations to Java. Thus, the time-to-collision information was transformed into a bar graph; essentially the image frame was split into 5 bars (vertical regions), each containing object(s). Then the height of the bars represents the time-to-collision (lowest) of objects in that bar space (Fig. 11). Each column would have time-to-collision calculated which then influences the heading angle to react to the certain time-to-collision.

First, a threshold was made: time-to-collision values greater than 300 frames were taken away because there was no need to sacrifice TTC running time on non-immediate situations. Next different types of reaction situations were analyzed. For example, there could be 5 bars meaning is an object with a valid time-to-collision in each of the bars. However, there also could be different

combinations of 4, 3, 2, and 1 bar(s) that could be passed to the Java compiler. Thus three types were generalized.

The first type, given 5 or any combination of 4 or 3 bars, moved the robot from that object at relatively large header angle fluctuations. The header angle changes  $10^\circ$  every time for 5 bars,  $8^\circ$  for 4 bars and  $6^\circ$ – $7^\circ$  for 3 bars. The direction of change was set to clockwise unless otherwise specified. This given statement is overridden by the actual place of the bars. For example, if there were 3 bars located at the left 3 bars of the 5, then it is more logical to move to the right. Such adjustments were made. The second type, given 1 or 2 bars that are close to each other, moved the robot away from those bars (again clockwise or counterclockwise based on location of the bar) at  $5^\circ$  of header angle change per analysis. The third situation was when the bars are split. If 4 bars were split into 2 and 2 it was not enough room for the robot to go in between, so it defaulted to the first situation. However, if there existed 2 bars split into 1 and 1, then the robot was allowed to try and maneuver around it first.

In each trial, the bounding box and time-to-collision algorithms were checked if they were operating in real life. The robot was first set to stand still as time-to-collision calculations were made, allowing for the bounding box to separate objects from the background while the TTC number remained around 400. This served as a control. From there, the algorithm was testing in three different ways. In the first situation, the robot was allowed to roam randomly with stationary boxes/objects. The second situation had the robot turn away from objects in order to follow a path towards a set destination. The third way ensured that the program was not case-specific or “pre-programmed” and consisted of placing boxes in front of the robot and observing its reactions.

---

## MAP-Building Method

At this point, only a random and vague reaction system is developed. So after establishing autonomous movement from the time-to-collision algorithm, a systematic method of movement is needed to achieve the ultimate goal of mapping a blueprint. The mapping algorithms first creates a pseudo-infinite grid, a  $999 \times 999$  matrix of zeros since the room size would be unknown (each unit length representing the robot’s length). Then it positions the robot virtually in the center of the grid at (500,500), so that it can proceed to any direction of the grid regardless of which corner the robot began from physically in the room. From there, the robot is programmed to turn  $90^\circ$  counterclockwise, moving a set value  $x$  to be used for time-to-collision. If the time-to-collision becomes less than  $\alpha 100$  frames, then an object are assumed to be on the next point, triggering the robot to move back by the set value  $x$  to the center of the original point.

This process run in a loop until the robot finally saw an empty spot allowing it to proceed to the next grid square. However, note that this procedure to move is only observed in the first part of mapping: detecting walls (as described next). To keep track of the robot’s position virtually in a grid-representing matrix, numbers are used to represent the status of each coordinate: 0 is an unknown spot, 1 is a visited spot, 2 is an object, 3 is part of a wall, and 5 is an inaccessible coordinate.

With this, first the robot is programmed to go around the room once using the moving procedure described above. It will stay on the periphery to detect the positions of the walls. The number 4 is used virtually to mark the initial position of the round, to notify the robot when it made a complete round in the actual room. After all the walls are detected, a `resizeGrid` function then fits the grid into a more appropriate matrix by finding the smallest and greatest row and column values in which 3’s are present. This cut down inefficiencies since it is unnecessary for the robot to consider the entire pseudo-infinite matrix for every situation as most of the indices will be 0.

After the grid is resized and the initial 4 reached, the robot is then prompted to find a virtual ideal path around the room using the knowledge it has of the room so far: the walls. The purpose of the ideal path is only to guide the robot in a much more efficient manner as it would then be able to reference the ideal path as an original path while compensating for objects in the real environment. This is done in a separate virtual grid to not affect the grid the robot would actually map; the real grid is kept the same as before the ideal path process begins. The ideal path is made in a spiraling fashion as there will be almost no overlaps of 1’s (visited spots) in an ideal setting. To do this, the virtual robot is programmed to continuously “hug the wall” using a virtual wall as reference. So, much like the code that finds walls, the ideal path algorithm also marks the beginning position of the round as a 4. It then marks the visited spots as 1’s as the virtual robot makes its first round in the grid. Once the initial 4 is reached again, the ideal path algorithm then changes all the 1’s and 4’s recorded to 3’s, thus creating a virtual wall within the separate virtual grid. With the virtual wall established, the virtual robot can then repeat the process using it as reference again to make another round until the whole room is covered. These points are stored in order in a list to be used when the physical robot begins to make rounds.

To avoid the virtual robot trapping itself in the starting corner when it comes back to the initial 4 position, a calculation is set to move the virtual robot to the nearest zero. It calculates the Euclidean distances to the closest zeros (unvisited spots), moves the virtual robot to the smallest distance, and continues to find ideal path. Though moved, the virtual robot’s direction is kept same since it defaults to the original one to continue the process. The ideal path is

finished when the virtual robot is fully trapped in the middle and the closest zero is at a distance significantly far away, indicating that the zero is inaccessible and outside of the room. It also shows that the virtual robot has been on every single possible position in the separate virtual grid.

Once the ideal path is found, the physical robot could now make its rounds following the original ideal path. In this portion, the robot does not move as described when it finds the walls originally. Instead, it adheres to the ideal path, only to deviate when an object is detected through time-to-collision. Only positions that have value 0 are visited to make the mapping efficient as possible. As obstacles are detected along the way, the robot uses A\* Pathfinding algorithm to give the shortest path between two points avoiding objects. However, because not all objects are known, the algorithm is used continuously as the robot moves and continues to encounter objects much like the functions of a GPS.

## Experimental Results

The current experiments itself are the building and design of an autonomous robot, so the results naturally are measured through the accuracy of the robot's codes and instructions. Thus, the analysis of the algorithms is observed by the robot's ability to avoid objects; if it is successful in this endeavor, then this related that the time-to-collision calculations are working and in cascade, the object detection codes as well. TTC values are also printed for a more precise understanding of what happened in the trials.

The nature of the results were recorded and observed in 9 trials, 3 trials for each of the 3 situations described in the Methods section. First, the control was established to make sure that the time-to-collision was accurate and consistent. While the robot was held by the wheels (so no movement) the time-to-collision kept at a consistent range from 380 to 490. The values were mainly about 440, 443, 450, 451, 446 etc. This proved that the little fluctuations were due to light and that the relative time-to-collision seems to be working properly. When the robot was let go after the control results, the time-to-collision dropped considerably to 70 as there was a washing machine in front of it. Testing the following trials was done in a garage.

In the first type of situation, the robot was left to roam the room for itself with a random pathway. The time-to-collision values printed were all over the place since there were multiple objects in the image frame that the robot had to detect and turn away from. Refer to the measurement results in Fig. 6 in which the elapsed time is the time taken to calculate/process the time-to-collision value, the TTC value is the number of frames before the robot will hit the object, the column number refers to the location of that object, and the heading angle change relates to its specific

```
Elapsed time is 0.972144 seconds.
TTC: 70.1227
Column Number: 2
Heading Angle Change: 10

Elapsed time is 0.374341 seconds.
TTC: 17.1003
Column Number: 2
Heading Angle Change: 40

Elapsed time is 0.878175 seconds.
TTC: 10.5562
Column Number: 2
Heading Angle Change: 40

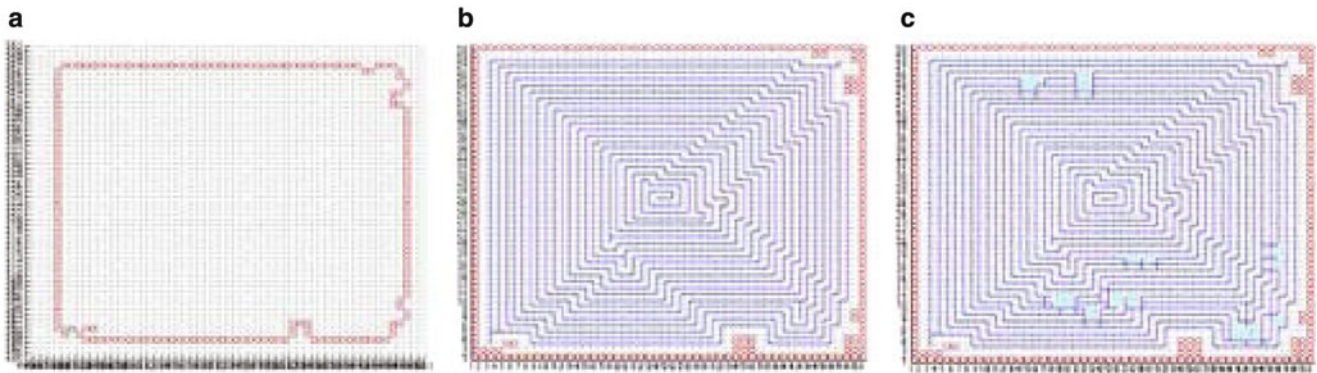
Elapsed time is 0.469931 seconds.
TTC: 102.0256
Column Number: 2

Elapsed time is 0.387387 seconds.
TTC: 32.0021
Column Number: 2
Heading Angle Change: 30
```

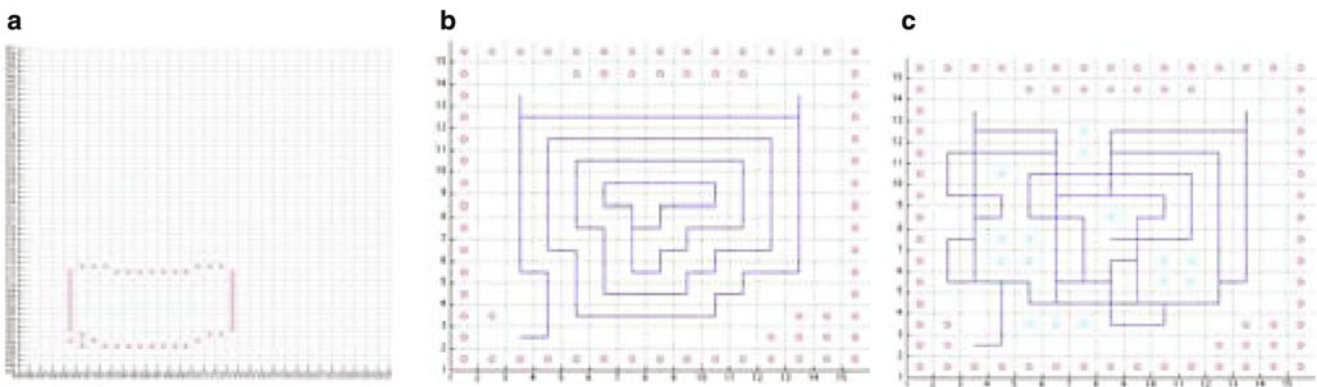
**Fig. 6** Illustrative measurement

TTC value. Since the elapsed time is below 1 s, it indicates that the TTC calculations are pretty much in real time. They jumped from low to high values unexpectedly, as the robot would turn into empty spaces, these fluctuations occurring in a random cycle. Consequently, the time-to-collision values reflected the random nature of these trials. However, though "random", the trials proved the algorithms to be quite robust in its ability to avoid objects. Surprisingly, there were no crashes until the very end of trial 3. In the first two trials, the robot successfully started, stopped, and turned with respective header angles. In the trial 3 crash, however, the robot turned and hit an object that was not in its image frame, but rather in the swing of its header angle. Though it indicated partial failure in these trials, in the context of mapping, these random movements would not exist but rather be replaced with systematic movement across the grid. Thus, this crash is not deemed significant given the overall context.

In the second type of situation, the TTC values were in a more consistent pattern since the robot was bounded by boxes to direct the robot towards a specific destination on a path. The robot successfully passed all trials without hitting any of the boxes and was able to "bounce off" of all the obstacles to finally make it to the end of the garage. The process, however, was painfully slow and inefficient, because at that stage the robot had only a random/vague reaction system (not systematic like in mapping). But it is important to note that these trials were mainly to test the TTC accuracy and if application was even possible.



**Fig. 7** Virtual progression of the mapping procedures in a relatively large grid



**Fig. 8** Virtual progression of the mapping procedures at a more realistic size of  $15 \times 15$  (units are in the robot's length)

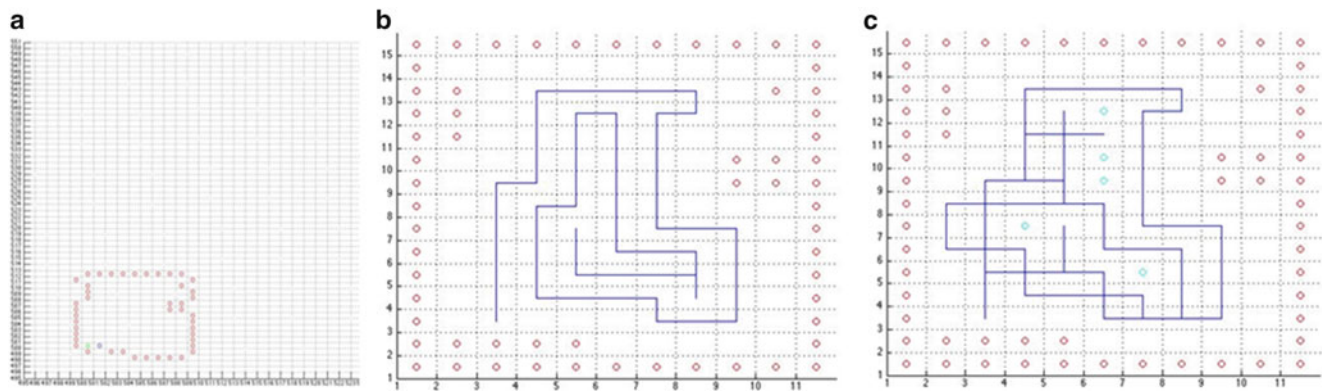
The third situation was probably the most exciting since it involved sporadically placing objects in front of the robot and observing its reactions. In all three trials under this condition, the robot successfully avoided all obstacles even if objects were immediately placed in front of it again after turning. This proved that the program was not preprogrammed, but rather could respond to various situations. The significance of this flexibility is immense since the whole point was to build a code and robot that could “think” for itself as a smart robot. Noting that in all situations, time-to-collision was accurate enough for the robot to avoid objects, it proves the TTC and all the object detection algorithms to be robust and able to be used in a broad spectrum of applications.

The mapping portion was also found to be successful in finding a systematic way to map a room with most efficiency. A virtual space was used to test our algorithms with many trials. Results were printed out on a grid using different colored circles to mark walls or objects and lines to show ideal and robot paths (see Fig. 7). For Figs. 7, 8, and 9, in part (a), the wall finding algorithm is finished and in part (b), the grid is resized to the size of that room. The ideal path

is drawn as shown and then in part (c), the virtual robot successfully detected objects and avoided them, ending in the middle of the room.

First, the wall-mapping algorithm that finds walls and boundaries of the room ran smoothly with no kinks after debugging. The moving method used to initially find walls worked well in the virtual setting (though in real life, that is still to be tested for certainty). The advantage to virtual testing was that many trials could be run in relatively short time. With this, complete accuracy and full execution of the wall finding code was confirmed with 20–25 successful trials of this individual code (Fig. 8a). The cutting grid algorithm worked flawlessly as well for its numerous trials. Regardless of size, big ( $50 \times 50$ ) or small ( $11 \times 15$ ), or random shape the room was made into, the algorithm was able to successfully cut the excess units in the grid and fill in gaps on the outside with extra number threes to indicate those units are part of a wall and not inaccessible zeros (Fig. 8b). The product of the resizeGrid algorithm can be seen with the ideal path pictures in Figs. 7b, 8b, and 9b.

The ideal path algorithm also proved to be robust to all different types of grids: square, rectangle, big, small, some



**Fig. 9** Virtual progression of the mapping procedures with a rectangular room of  $11 \times 10$  deemed more realistic to real life situations

with irregular walls. Though, at first, the code broke when the robot was trapped at the end of each spiral around the room, this was later fixed as stated in Methods by calculating the closest zero and moving to it. After this alteration, the entire code ran smoothly for all time. However, though these algorithms ran flawlessly, once the virtual robot began to actually map, complications rose up. First, sometimes the A\* Pathfinder gave longer paths, reducing the optimal efficiency of the mapping process expected. It was also found that for some odd reason, the entire code just broke and produced unreadable grids with awry paths. However, with careful analysis, it was found that it was not our codes specifically that caused this, but the addition of the A\* Pathfinding code. Our analysis stopped here as we did not understand the A\* code, preventing fixes, but for many cases the virtual mapping did in fact work (Fig. 9c).

Overall, the virtual trials for mapping confirmed that the codes were robust and working well enough to be implemented in real life in combination with object detection and TTC codes.

## Conclusion

The current work has shown that it is indeed possible to autonomously navigate and map a room using solely optical inputs. The various elements for implementation, including object detection, time-to-collision, and the mapping codes, have been proved to be robust in their flexibility and accuracy passing numerous trials and various situations. This opens the door to the idea that our codes and methods can be used for a broad number of other applications beyond mapping such as autonomous vehicles or exploration robots while staying cost-effective and completely reliable.

The validity of the conclusions come from the sheer number of trials ran and the diversity of the situations that were tested, proving the continuity and thus, validity of the results. The continuous success of the TTC trials reflect that all codes leading up to object detection worked and the time-to-collision calculations were accurate in avoiding crashes. Likewise, the near total success of the many mapping trials also show validity of the results produced.

To improve, future work towards the possibility of developing a unique path finding algorithm that optimizes for this work's blueprint robot. Furthermore, another type of robot can be designed to systematically map and maneuver using time-to-collision. However, because the time-to-collision measurements are only estimates, it is predicted that the actual robot movements will lack precision and not be consistent since they are reliant on the TTC values. This would lead to faulty communication between the positioning in the grid and the real room.

Differentiation between walls and objects is also another future experiment that must be looked into. In a standard indoor setting, objects will most likely be placed on the walls and to increase the usefulness and accuracy of mapping a blueprint, a robot would need to detect the differences. This can be achieved with the use of RGB gradients and comparing sharp changes between color values combined with the use of Sobel edge detection to perhaps detect texture as well. Thresholds would be found during experimentation to fine tune the color detection especially when both the wall and object are similar colors. In the far future, object identification can be added on to object recognition. This could be achieved through perhaps a database of classifications made experimentally that narrow down objects to their identification. This would be the ultimate goal of the blueprint problem.



---

## References

1. Guido Zunino, *Simultaneous Localization and Mapping for Navigation in Realistic Environments*, Licentiate Thesis, Royal Institute of Technology Numerical Analysis and Computer Science, 2002
2. Guido Zunino, *Simultaneous Localization and Mapping for Navigation in Realistic Environments*, Licentiate Thesis, Royal Institute of Technology Numerical Analysis and Computer Science, 2002
3. Erico Guizzo, "How Google's Self-Driving Car Works," IEEE Spectrum
4. Pawan Kumar, Len Bottaci, Quasim Mehdi, Norman Gough, and Stephane Natkin, "EFFICIENT PATH FINDING FOR 2D GAMES," Proceedings of CGAIDE 2004, 2004
5. Horn, Berthold K.P., and Brian G. Schunck. "Determining Optical Flow." Artificial Intelligence, MIT: 185-203. Web. 20 Jan. 2011
6. Amaury Negre, Christophe Braillon, James L. Crowley, and Christian Laugier, "Real time Time To Collision from variation of Intrinsic Scale," INRIA, Grenoble, France