Implementing a Real-Time Hough Transform on a Mobile Robot

John Morrison, Eric Chown, Bill Silver

Bowdoin College

Abstract. Robotic vision is a challenging problem due both to the uncertain nature of real-world environments and the computational constraints of mobile platforms. Many standard computer vision algorithms have high computational requirements and are often seen as unsuitable for use in an embedded system such as the Aldebaran Nao. Many current approaches use fragile algorithms and reduced resolutions to achieve a low processing latency. We implement the Hough Transform, a standard line detection algorithm, for real-time use in the RoboCup Standard Platform League. Using assembly language instructions to expose the processor's full potential, this project implements a real-time Hough transform for line detection on 320x240 pixel images.

1 Introduction

The primary source of perceptual information for the robots in the RoboCup Standard Platform League (SPL) [6] is their camera, forcing real-time vision processing to become a topic of intense research and development. The SPL environment and platform offer a particular set of constraints which challenge any potential vision system, such as a relatively slow (by desktop standards) processor, a narrow field of view, unpredictable images, partially obscured objects, and limited processing time per frame. The output of the vision system is the foundation for higher level cognition and skilled soccer playing.

The current state of the art in RoboCup vision systems, mostly based on color segmentation and color run-length encoding, is ad-hoc and tied tightly to the RoboCup environment. The methods described by the top competitors in the league [3, 9, 8] for identifying object shapes in the image are largely collections of hand coded heuristics for what defines a "goal post" or a "line" as a human would describe it. Domain information about the specific objects in question is, of course, required for reliable detection, but the goal of this project is to show that general pattern recognition techniques are a viable option for SPL vision systems.

Computer vision research has produced many pattern recognition algorithms which are more reliable, more robust, and more general than these hand designed region building heuristics. Unforunately, there has not been sufficient research yet to optimize these algorithms for the SPL. This is why teams in have been hesitant to use them in competition.

To put the performance requirements of a real-time vision system in context, consider that with a 640x480 YUV 422 image, there are 614,400 values to be processed. So, allocating 20 ms for vision processing there are only 32 nanoseconds of processing time available per value. With a 500 MHz processor, like the Nao's Geode, that means there are only 16 machine cycles available per value. For comparison, a single integer divide can take 24 cycles, and one floating point arctangent can take up to 354 cycles [1]. It is with these constraints in mind that this project was undertaken.

The first step of the new vision processing is edge detection. Edge detection provides the points of interest which will be used in the Hough transform for line detection. After the edge detection, the edge points are processed with the Hough transform. The Hough transform works through a voting mechanism. It tallies the "votes" of each edge point for a particular set of possible lines in the image. The voting mechanism makes the algorithm robust, yet straightforward.

This project aims to establish the Hough transform as a reliable and applicable vision algorithm for the embedded RoboCup Nao platform. The Hough transform has been used infrequently [8, 128] in work by other teams, but when it was, it was often in a limited or prohibitively slow manner. The Hough transform is an algorithm which suits the highly uncertain RoboCup environment well, and, when properly implemented, is also acceptable for full image processing on an embedded platform.

We seek a general algorithm for pragmatic reasons. General vision algorithms are robust in various circumstances and can be re-purposed and reused in new situations. Many current SPL vision systems tightly integrate the steps of object detection. When the rules change, or the environment is altered, the ad-hoc system can require extensive revising to be adapted. A good general algorithm will be easily adaptable. The current SPL vision solutions maintain high frame rates by skipping lots of pixels, sampling the images down to paltry 160x120 or below. This loss of resolution degrades accuracy and can cause mis-detections because of insufficient visual evidence.

2 Assembly Language

In order to achieve the desired real-time efficiency of this project, we wrote the majority of the code for this project in x86 assembly language. Assembly language can sometimes be the only way to truly squeeze every bit of efficiency from a CPU. It lets the programmer use the ideal instruction mix for the task, a job a compiler can sometimes struggle to do.

In addition, modern architectures have vector processing instructions which have no easily accessible and embeddable C/C++ equivalent. The vector processing units, known as Single Instruction Multiple Data (SIMD) units, are capable of operating simultaneously on multiple pieces of data, often with special instructions capable of performing complex operations with a single instruction. The Geode LX processor can operate the x86 MMX instruction set which uses 64-bit registers to operate on multiple 1, 2, or 4 byte values with a single instruction. We make extensive use of vector instructions throughout this project.

3 Edge Detection

Edge detection, as applied here, is the process of finding points of significant and rapid change in the intensity of one or more channels of an image. The physical boundaries between objects on the field cause brightness discontinuities which are captured in an image and are largely independent of lighting, viewpoint, and camera settings. This makes edge points a more reliable measure for object detection than the brightness values themselves.

To strike a balance between computational requirements and accuracy, we chose to use the Sobel operator [5] in the edge detection process. It requires only a single pass over the image, but performs very well as an edge detector. The Sobel Operator is applied to the image's Y channel. Edges in the Y channel provide good boundaries of field lines across lighting conditions.

3.1 Sobel Operator

To detect the edges in images, we first calculate the gradient at every pixel in an image by convolving the image's Y channel with the Sobel operator [5, 147]. The Sobel operator is a pair of kernels as depicted in Table 1, which approximate the (x, y) components of gradient. The magnitudes in x and y are used to compute a magnitude and direction of the gradient vector at each pixel.

To run the Sobel edge detector in real-time with the computational constraints of the Nao, the operation needs to be optimized significantly. The Sobel operator is an ideal candidate for SIMD instructions. G_x and G_y are calculated independently at each pixel. By hand coding in x86 assembly language and using the MMX instructions, we were able to calculate four pixels' gradients in parallel, resulting in a significant speed increase.

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

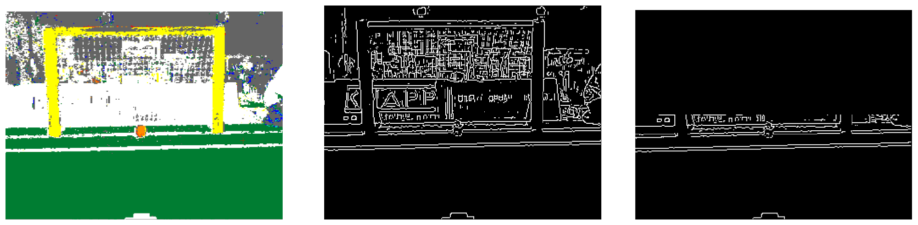
(a) G_x

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

(b) G_y

Table 1: x,y Sobel Kernels

As processor speeds increase, memory accesses and memory bandwidth become a limiting factor in high performance computations. The 3DNow! instruction set provides the prefetch instruction to help lower the memory bottleneck. By bringing soon to be accessed memory into the fastest cache, the programmer can remove some waiting for I/O. By prefetching two rows below the current pixel during the Sobel operator execution, we reduced cache miss penalties. When a new row is accessed, it is already in the L1 cache because the 64 KB L1 cache is large enough for more than four rows of pixels.



(a) The color segmented image (b) Edge detection on the entire image (c) Edge detection only performed below the field horizon

Fig. 1: Reducing edges with the field horizon

3.2 Edge Peak Detection

Once the gradient magnitude and direction is calculated at each pixel, the gradient values need to be compared to each other to find the actual edge points in the image. The edges are found by locating the peaks in the gradient map, the points of greatest change, which are the desired output of the edge detector. The goal of the peak detection phase is to also ensure that only one edge point is found for each real edge point in the image.

Using the gradient direction, an asymmetric peak test is then applied in the gradient direction to all remaining candidate points. The peak test determines if the candidate edge point's gradient magnitude is greater than that of the pixel in the opposite direction of the candidate's gradient, and greater than or equal to the magnitude of the pixel in its gradient direction.

Once a pixel passes the asymmetric peak test, it is labeled an edge peak and its location in the image, along with its gradient direction, are appended to the list of previously found edge peaks. The final list of edge peaks is then sent to the Hough transform to be processed further.

A significant limiting factor in peak detection computation time is the angle calculation. Floating-point calculations are slow and provide more accuracy than needed for our purposes. To remove this bottleneck, we chose to replace the floating point angles with eight-bit binary angles. An eight-bit binary number represents the range 0-255, so a radian angle t can be computed as an eight-bit binary angle with the formula $t_{int} = t_{radian} * \frac{128}{\pi}$. These binary angles are accurate to 1.41 degrees.

The machine level floating point arctangent function is also too slow, taking up to 354 machine cycles [1]. By precomputing an arctangent lookup table, we removed the costly arctangent function call. The table was indexed by the result of $\frac{G_y}{G_x}$. According to the Geode LX Databook, an integer divide of 2 byte-long values can take up to 16 machine cycles, while an integer multiply of similar length values only takes 3 cycles [1]. To take advantage of this, we precomputed another lookup table of $\frac{1}{G_x}$, indexed on G_x.

The power of using assembly language comes in the ability of the programmer to exploit every corner of a machine's instruction set. The x86 instruction set is large and has evolved over time to accommodate many different needs and applications. The advantage of the x86 instruction set's breadth is that it has many optimization opportunities, but they can be hard for a compiler to use. The author of the code has the best idea what the code is intended to do, and the circumstances it will encounter, so he can choose the best instruction mix to accomplish that task. That power has great rewards in the edge detection phase of this project.

4 Hough Transform

The Hough transform was patented in 1962 as a "method and means for recognizing complex patterns in photographs" [4], specifically straight lines, from a set of points. The transform has also since been generalized to find any parameterizable general pattern, [2] but we will only use it here for line finding.

The Hough transform has many properties which make it suitable for line finding in RoboCup. The field lines in RoboCup are uniform, straight lines but are often occluded. The Hough transform deals well with occlusion through the voting mechanism [5]. A line which is partially covered by a ball, or a robot, will still be recognized as crossing through the body of the robot. This is necessary for RoboCup vision as other robots are often obstructing lines.

Without specific attention paid to its implementation, the Hough transform can be memory and processor intensive [8, 5] which has limited its adoption in the league in the past. Specific attention was paid during this project to memory access patterns and computation ordering during the Hough voting procedure.

4.1 Voting

The first step in the Hough transform is marking the Hough accumulator space for every peak found during the edge detection. Looping through the list of edge peaks, we increment the $\{r, \theta\}$ accumulator bins which correspond to each possible line which could pass through that edge peak.

A basic implementation of the transform would increment the bin for every line that could possibly pass through that line. For a single edge peak, over 256 bins would need to be incremented, at least one per angle. Filling so many bins would quickly overwhelm a real-time system. An image with 2,000 edge peaks would have to fill many more than 512,000 (and experimentally, likely more than 1,000,000) bins.

However, the gradient angle provides sufficient information to limit that search to within a small error range. The gradient direction approximation computed in previous steps is approximately equal, within an error margin, to the angle of the line, so we can limit the marking in the Hough space to angles which are within a predetermined error margin of the gradient angle. The error margin needed is small, ± 5 was found sufficient for this project. Thus, instead of incrementing bins for 256 angles, only bins corresponding to 10 different angles, representing a span of approximately 14 degrees, are incremented.

While limiting the span of Hough bins to mark is a significant improvement in running time, it is not significant enough, and there is still more time to be recovered. Thus far all the algorithms and implementations have operated sequentially in a single pipeline using only basic integer instructions. The Geode actually has two hardware pipelines, an integer unit (IU) and a floating point unit (FPU). The FPU handles MMX instructions, as well, reusing its 64-bit registers for SIMD commands. With its dual pipelines, the Geode is capable of executing an integer instruction and an MMX instruction every clock tick.

With this in mind, we redesigned the Hough voting procedure to intersperse MMX and integer instructions. By injecting a mix of instructions into the processor, we are able to interleave computations and significantly decrease the necessary run time of the routine.

We set up two pipelines which run simultaneously and communicate using a pair of ping-pong buffers:

1. **MMX/FP Unit:** The MMX registers compute the locations in memory to be incremented.
2. **Integer Unit:** The IU increments the Hough bins at the locations computed by the MMX pipeline.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

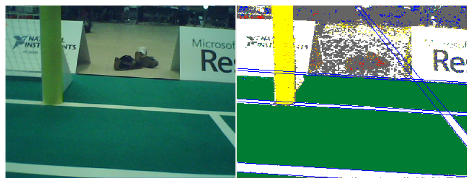
Table 2: 2x2 Boxcar Kernel

After the voting is complete, we smooth the Hough Space to reduce the noise. The smoothing is completed with a 2x2 "boxcar" kernel, as seen in Table 2. The kernel is applied to each bin in the accumulator by averaging each bin with the bins to the right, to the right and below, and below it. This helps to smooth noise in the Hough space due to errant edge peaks. The boxcar is simple, fast, in place, and can be implemented effeciently using MMX instructions.

After the Hough space has been filled, the bins with the highest values must be found. First the Hough space is scanned for bins with a non-zero value. During the smoothing step, we subtracted a fixed noise threshold and clamped the value at zero. Once a bin with a positive value is found, the eight bins surrounding it are compared with the current bin. If the current bin has a value greater than its eight surrounding bins, it is marked as a peak and added to the end of a list of peaks. Its r and t indices, along with its "score," the value of its accumulator bin, are recorded.

5 Results

The Hough transform based line finder presented here is a step forward from the region and heuristic based approaches that are the current state of the art in the SPL. From edge detection through line pairing, this field line detection system is straight forward and effective. The entire system has only three parameters: the edge detection noise threshold, the Hough angle spread parameter, and the Hough noise threshold. Compared with the ad-hoc color based systems, which contain dozens of different parameters, the new system is more robust and does not require significant calibration.


(a) Hough line detection with a good color table

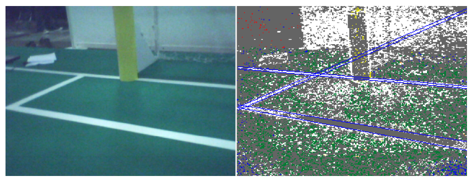

(b) Hough line detection with a poor color table

Fig. 2: New line detection does not depend on color calibration

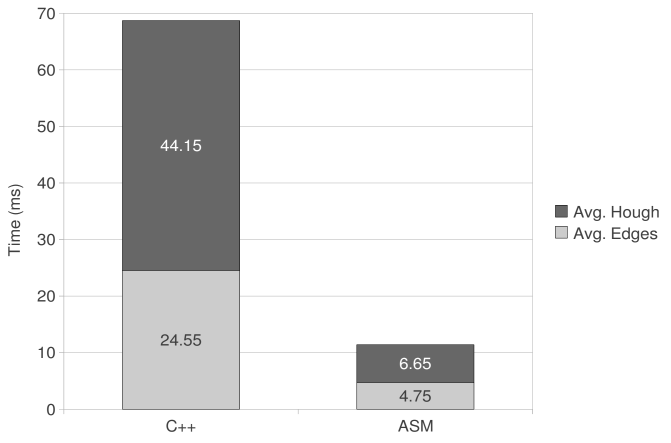


Fig. 3: Average Full Image Line Detection Run Time

The new field line detection system using assembly language coded routines is fast. Compared with a C++ implementation of the same algorithm, it will run significantly faster while producing similar results. Differences between the two implementations come from floating point rounding and a small difference in gradient angle calculations between the implementations. When compared across a test set of 15 images with varying composition, the assembly version averages a factor of six improvement over the C++ version. Figure 3 compares the average run times over entire images (without the field horizon limiting mentioned in Section 3.1), breaking each version down into the edge detection and Hough transform steps. The C++ version averages a run time of 68.7 milliseconds per image, whereas the assembly version averages 11.4 milliseconds.

This project's goal is to build a Hough transform suitable for use in a robotic system running at 30 frames per second. At 30 frames per second, all the processing for each frame must occur in under 33 milliseconds, including not only vision, but also motion and behavior processing. A line finding system running at more than 15 milliseconds was decided to be unacceptable, as it left too little time for remaining processing. As such, the 11.4 millisecond average run time is acceptable for use in a real-time RoboCup SPL vision system.

5.1 Edge Detection Performance

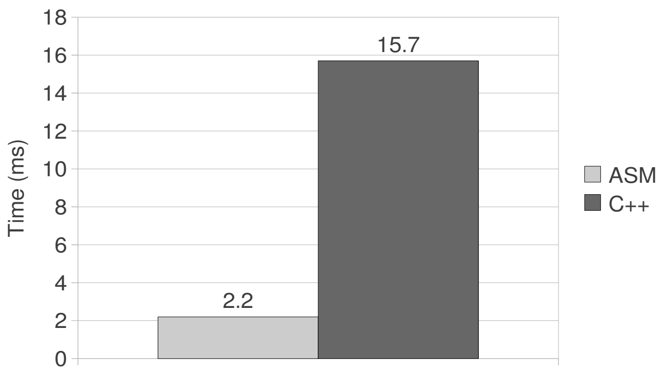


Fig. 4: Average Sobel Operator Run Times

The edge detection step benefits the most from the transition to assembly language using the MMX instructions. A speedup factor of seven is achieved in the switch. As the Sobel operator is applied to every pixel in the image, the C++ and assembly raw timing values can be compared.

The combination of SIMD instructions and memory prefetching to improve cache hits yields a significant improvement. Memory access time is effectively cut to zero, leaving only computational time as a constraint. The SIMD instructions help to reduce even that by operating on multiple values simultaneously.

5.2 Hough Transform Performance

The Hough voting routine, as measured in these experiments, is shown to be an acceptable real-time vision algorithm for RoboCup. The C++ version is indeed

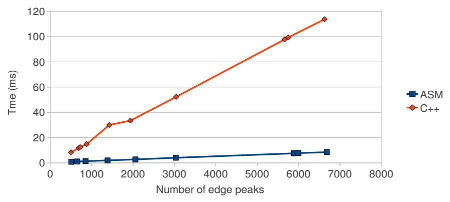


Fig. 5: Average Hough Voting Run Time

too slow for real-time use, as the busier images cause it to take over 100 milliseconds to complete. This is over three times the desired frame rate, so it is clearly not acceptable. The assembly version, however, takes under nine milliseconds, or one tenth of the time to complete on a similar number of edge points. This is a considerable speed up, given that there is no loss of accuracy between the two versions.

The Hough space smoothing routine is a simple operation involving four sums and a subtraction. The assembly version developed here runs over twice as fast as the C++ version. The output of GCC for this code, even using `-mmmx`, which allows the compiler to use MMX instructions, does not use MMX instructions and is thus limited to operating on one bin at a time. Efficient compiler generation of SIMD code is difficult [7] and programmers cannot rely on a compiler to generate any code in particular.

6 Conclusion and Future Work

In the future, the generality of edge detection and shape detection could be extended to the RoboCup goal posts and the ball. These objects are currently detected with color blobbing and present similar opportunities for improvements. A circular Hough transform would be possible with the ball, removing the dependency on color calibration to see the most important object in RoboCup vision. The orange ball presents a strong contrast to the green carpet in the V channel and presents a good opportunity to move slowly away from color calibrated information. The SPL plans to switch away from the orange ball to other styles of ball where the predetermined color is not important. A more general approach to ball detection would be helpful here.

The argument can be made that with faster processors there will be no need to implement solutions like this, but that argument falls apart when other changes are considered. For instance, doubling the dimensions of an image quadruples the pixel count. Higher resolution cameras can quickly overwhelm faster processors. A faster frame rate, higher accuracy demands, or even just more complex objects demand more processing efficiency and power than a faster processor can provide alone. Speed and necessity will always be an issue, so in order to move away from engineered solutions, efficient implementations of general purpose algorithms are necessary.

The transition to general image processing algorithms will improve vision processing robustness in changing environments, and is a necessary transition for the advancement of the sport of robot soccer. By replacing the color segmented and run-length encoding based system with a system using the Hough transform to identify field lines, we have taken a first step towards gaining independence from calibration and manual tuning. This will lead to increased productivity for the roboticist, as they do not have to spend so long calibrating the robot to new situations, and increased performance on the robot, as it can deal with fluctuations within its environment.

We have shown that general vision algorithms can still perform in this real-time environment and are a suitable replacement for the engineered solutions the SPL has typically used. Commonly held beliefs that the Hough transform is too slow for RoboCup are here shown to be baseless. A well designed system can be both robust and fast.

Also, computer scientists often shun hand written assembly as adding needless complexity and non-portability when optimizing compilers are available. As the results here show, an experienced programmer designing a complex system can still beat a compiler by making full use of the processor's architecture and instruction set in ways a compiler may not be able. The compiler's non-use of SIMD instructions is a great example. It is hard in a high level language such as C or C++ to instruct a compiler that it can make use of vector instructions for a certain complex task. By writing it oneself, a programmer can achieve optimal performance from the application.

References

1. Advanced Micro Devices, Inc., One AMD Place, Sunnyvale, CA 94088. *AMD Geode^{TM}LX Processors Data Book*.
2. D. H Ballard. Generalizing the hough transform to detect arbitrary shapes*. *Pattern Recognition*, 13(2):111–122, 1981.
3. Tucker Hermans, Johannes Strom, George Slavov, Jack Morrison, Andrew Lawrence, Elise Krob, and Eric Chown. Northern Bites 2009 Team Report, 2009.
4. Paul Hough. Method and Means for Recognizing Complex Patterns, 1962.
5. Ramesh Jain, Rangachar Kasturi, and Brian G. Schunck. *Machine Vision*. McGraw-Hill Science/Engineering/Math, 1995.
6. RoboCup Standard Platform League. `http://www.tzi.de/spl`.
7. Rainer Leupers. Code selection for media processors with SIMD instructions. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '00, page 4–8, New York, NY, USA, 2000. ACM. ACM ID: 343679.
8. Adrian Ratter, David Claridge, Bernhard Hengst, Brad Hall, Brock White, Benjamin Vance, Hung Nguyen, Jayen Ashar, Stuart Robinson, and Yanjin Zhu. rUNSWift Team Report 2010, 2010.
9. Thomas Röfer, Judith Müller, and Tim Laue. B-Human Team Report and Code Release 2010, October 2010.