Optical Flow for Obstacle Detection in Mobile Robots

by

Kurt Alan Steinkraus

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

© Massachusetts Institute of Technology 2001. All rights reserved.
Optical Flow for Obstacle Detection in Mobile Robots

by

Kurt Alan Steinkraus

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2001, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

In this thesis, I investigate optical flow and how it can be used to help a mobile robot avoid obstacles. I develop an algorithm to find locations where the optical flow has a high probability of being correct when computed. In order to do the optical flow computation in real time, I give several methods to speed up the basic optical flow computations without sacrificing accuracy. The robot can then find obstacles as it moves around by using this reliably correct optical flow to create a model of the floor and of how it is moving relative to the robot. By identifying regions of the image which do not move as part of the floor, the robot identifies the obstacles around it.

Thesis Supervisor: Leslie P. Kaelbling
Title: Professor of Computer Science and Engineering
Acknowledgments

I would like to thank the following people, who have made this thesis possible:

- Professor Leslie Kaelbling, my advisor, for giving me direction, for challenging me during our discussions, and for her very helpful feedback.

- The Defense Advanced Research Projects Agency, for partially funding my research under DARPA contract #DABT63-99-1-0012.

- Professor John Laird at the University of Michigan, for introducing me to AI and answering my many questions.

- My parents, for their love and support.

- My wife, Karen, who has seen less of me in our first three weeks of marriage than she would have liked as I worked on finishing this thesis but who has shown me great love and patience nonetheless.
Contents

1 Introduction ................................................. 9

2 Robot setup .................................................. 11
   2.1 Rationale for vision only ............................... 12
   2.2 Environment assumptions ............................... 13
   2.3 Camera model and correction of distortion .......... 13

3 Optical flow ................................................ 17
   3.1 Optical flow methods .................................... 18
      3.1.1 Patch matching .................................... 18
      3.1.2 Other optical flow methods ....................... 20
   3.2 Discretization issues ................................... 22
   3.3 Sub-pixel matching ...................................... 25
   3.4 Reliable correctness .................................... 29
   3.5 Speedups for real-time execution ..................... 31
      3.5.1 First successful match ............................. 32
      3.5.2 Multi-level matching ............................... 33
      3.5.3 Using the focus of expansion ..................... 34
      3.5.4 Faster correctness confidence ..................... 35
      3.5.5 Optical flow between confidently correct points 38

4 Depth maps from images .................................. 39
   4.1 Focus of expansion and optical flow method .......... 39
4.1.1 Focus of expansion computation from movement 41
4.1.2 Focus of expansion computation using optical flow 42
4.1.3 Focus of expansion computation from sparse optical flow 43
4.1.4 Other ways to find the focus of expansion 46
4.1.5 Accounting for rotation 47
4.2 Least-squares minimization method 48

5 Finding the floor and obstacles 51
5.1 Meaning of “find the floor” 52
5.2 Filling to the nearest edge 53
5.3 Fitting a plane to 3D points 54
5.4 Fitting a floor flow to optical flow 56
5.5 Real-world performance and limitations 58

6 Conclusion 61
List of Figures

2-1  Projective perspective model ........................................ 14
2-2  Radial distortion undone ............................................. 16

3-1  Moving point projected to its optical flow ....................... 18
3-2  Optical flow of patch at table edge ............................... 20
3-3  Discretization issue .................................................. 22
3-4  Within-half matching ................................................. 23
3-5  Diagonal shift overlap ................................................. 25
3-6  Fitting a parabola within half ...................................... 28
3-7  Reliable correctness of optical flow ................................ 30
3-8  Optical flow away from the focus of expansion ................. 34
3-9  Hypothetical pixel intensity values ................................ 36
3-10 Regions passing the interval test .................................. 37

4-1  Depth to point using focus of expansion method ............... 40

5-1  Coordinate system .................................................... 56
5-2  Floor flow consistency test ......................................... 59
Chapter 1

Introduction

Mobile robots would be useful in a lot of industrial, commercial, and military applications. Currently, though, an obstacle to the widespread deployment of mobile robots is, well, the existence of obstacles. Mobile robots that move in a dynamic real-world environment will not, in general, know ahead of time how to get from point A to point B, or perhaps even where point A and point B are. A robust mobile robot implementation needs to be able to dynamically detect and avoid obstacles in its environment.

Dynamic obstacle detection has several advantages over the alternatives. One alternative is to operate the robot in a space without any obstacles, an example of which would be a suspended robotic warehouse crane, free to move in any direction it chooses inside the warehouse. This solution is not generally feasible, since most domains where a mobile robot would be useful are spaces that people inhabit. The layouts of these areas are either unalterable or are constrained by the requirements of people rather than robots.

A second alternative to dynamic obstacle detection is the need to know ahead of time where the obstacles are. This is certainly possible in a lot of situations, but it is often not a satisfactory solution because the environment that the robot will encounter may not be known ahead of time in enough detail to specify the locations of obstacles. An example might be sending an autonomous robot to Mars or behind enemy lines on a battlefield.
Another hurdle in specifying obstacles ahead of time is that they have to be described to the robot in enough detail that it will be able to avoid them “driving blind.” Thus, it would not be possible just to tell a robot, “there are three buildings on the left.” Rather, the exact position of each part of each building that the robot may bump into needs to be specified. The process of pre-building this map will generally be prohibitively expensive, even if automated, because of the level of detail and precision required.

A third problem with specifying obstacles ahead of time lies not in the obstacles themselves but in the robot’s movement. A robot driving blind with a pre-conceived world map will keep track of its position by noting how far it moves and turns. Real-world sensors and actuators, unlike ideal ones, have noise and inaccuracies, and so the robot will invariably lose track of its exact position. These types of errors compound until, after a while, the robot’s sense of where it is no longer has any correspondence to reality. This makes the internal map useless for avoiding obstacles over a long period of time unless the robot can somehow localize itself in the world, and localization requires obstacle detection.

So, in summary, dynamic obstacle detection is desirable because the robot does not have to be programmed with precise and complete obstacle locations ahead of time, because the robot can take into account changes in the environment, and because the robot can compensate for inaccurate measurement of self-movement. The only drawback to detecting obstacles at run time is that it is hard to do correctly given current technology.

The goal of my research is to enable a robot to locate obstacles in the world at run-time. This work can be used as-is for some applications, and a parallel goal has been to provide the first processing stage in a complete system that builds maps dynamically.
Chapter 2

Robot setup

The robot that was used to test the techniques that I developed is a B21r mobile robot made by iRobot (http://www.irobot.com). The robot has a complete range of sensors, including sonar, infrared, a laser range-finder, bump sensors, and a video camera. The camera is mounted on a pan-tilt unit and outputs 320 by 240 resolution at 24-bit color. The software framework for interfacing with the robot’s sensors takes about one tenth of a second per iteration, creating an upper bound of ten processing iterations per second.

In addition to the sensors listed above, the robot also keeps track of its current position; that is, it notes how far it has moved in what direction, what angle it has turned to, and the orientation of the camera. These self-motion sensors tend to give slightly but consistently erroneous measurements. For instance, when driven straight forward, the robot veers slightly to the right, and the pan-tilt unit is angled down 5° more than it reports.

These ego-sensor errors are systematic and therefore fixable, in the sense that they can be compensated for in software. I decided not to fix them at the sensor level, however, because I want to make sure that whatever techniques are developed can deal with noisy data. Such noisy data would occur in any mass-produced consumer robot that is not manufactured as precisely as the B21r robot is. The noisy sensor data helped ensure that these techniques are robust enough to work on robots with less accurate sensors.
2.1 Rationale for vision only

The B21r has several different types of sensors, but the research presented here only uses the video camera and ego-motion sensors. While the laser range finder and other sensors provide additional information that is potentially useful in determining the location of obstacles, using them would add additional complexity and would make it harder to interpret just how well the vision part of the algorithm being investigated is doing. Robot vision is used because it seems to provide the best balance in several factors such as sensor noise, amount of information, and cost.

The most popular type of sensors used in obstacle detection are sonar sensors (e.g. Thrun [14]). This isn’t because they are highly accurate; in fact, they are notoriously prone to false readings and failure. Rather, it is because the readings are very easy to interpret, being estimates of the straight line distance to the nearest object in a certain direction. This makes sonar sensors very attractive when a robot has only a small amount of processing power available. Sonar sensors are also relatively cheap and easy to hook up.

A more expensive but much higher quality alternative to sonar sensors is a laser range finder. A laser range finder measures the distance to surrounding objects very accurately and with more directional granularity than is possible with sonar sensors (e.g., one measurement every one degree accurate to a few millimeters or less).

A third type of sensor used in mobile robot obstacle detection is a camera, sometimes fixed, sometimes mounted on a unit that allows it to tilt and swivel (as on the B21r). A camera is not only much cheaper than a laser range finder, but it also captures much more data than either of the above two sensors. As such, it should be possible to make better robots using cameras than distance sensors.

With the increased amount of data in images vs. distance measurements comes the requirement of more processing power and more sophisticated ways of interpreting the data. A way to think of the trade-off is that a camera provides too much information as compared to too little information provided by distance sensors. Too much information is a better problem to have, though, provided methods exist for
reliably extracting whatever manageable data points are needed.

2.2 Environment assumptions

To develop algorithms allowing a mobile robot to robustly detect and avoid obstacles, it is necessary to enforce some assumptions about the world in which the robot operates. These assumptions trade generality for feasibility, allowing the task to be completed within the scope of current processing limits and techniques.

One assumption, commonly called the brightness constancy assumption, is that an object will have constant brightness as it is viewed from different viewpoints. This is equivalent to saying that objects have diffuse rather than specular reflective properties. While most objects in the world have at least a small specular reflective component, this assumption of diffuse reflectivity turns out to be a good approximation to reality as long as the different viewpoints are fairly close together. That is, when the robot moves only a small distance between capturing two images, the image intensity of the observed objects changes very little if at all.

Another assumption is that objects in the world do not move. This is quite restrictive, but it has been necessary in order to concentrate on the algorithms for obstacle detection. This assumption can be relaxed later when the obstacle detection algorithms are robust. Images captured by the robot could be segmented into parts that are moving and those parts that are not, where the segmentation is informed by memory of previous movement, object recognition, texture information, and so on. Trying to add those features in at the present time, though, would just add unnecessary sources of noise and error, obscuring the performance of the core obstacle detection algorithm.

2.3 Camera model and correction of distortion

Almost all computer vision algorithms assume that images are captured using a projective perspective model (see figure 2-1). The coordinate system is relative to the
robot, where the robot’s camera is at the origin and looking along the \( y \)-axis. An image plane is imagined to sit one unit in front of the robot, centered about and perpendicular to the \( y \)-axis. A point \( p = (x, y, z) \) in the world is projected onto the image plane to a point \( p' = (u, v) \) using the following equations:

\[
\begin{align*}
    u &= x/y \\
    v &= z/y.
\end{align*}
\]

Figure 2-1: The projective perspective model. A point \( p \) is projected onto the imaginary image plane by finding the plane’s intersection with a ray from the robot to \( p \).

The camera I use has a Computar H3616FI lens, with a focal length of 3.6mm and a field of view of 92.6 degrees horizontally and 71.7 degrees vertically. Because the field of view is so large, the camera lens introduces very noticeable distortion in the image (the “fisheye effect”).

Since the projective perspective model is so much easier to perform useful calculations on, the actual image coordinates are transformed into planar projective coordinates by using a radial distortion model. Suppose that the center of the cam-
era image is at \((c_x, c_y)\). The radial distortion model relates a point \((x_0, y_0)\) in the distorted image point and its corresponding point \((x, y)\) in the corrected planar image as follows:

\[
x = x_0 + (x_0 - c_x)(k_1 r_0^2 + k_2 r_0^4 + \ldots)
\]
\[
y = y_0 + (y_0 - c_y)(k_1 r_0^2 + k_2 r_0^4 + \ldots);
\]

where \(r_0 = \sqrt{(x_0 - c_x)^2 + (y_0 - c_y)^2}\), and \(k_i\) are constants found through camera calibration. The intuition behind this model is that points are increasingly distorted the more distant they are from the center of the image.

Most of the benefit of the radial distortion correction comes through the first few terms in \(r_0\). In fact, using just the \(r_0^2\) term flattens the image well enough for the purposes of obstacle detection. An added bonus of using no higher order terms than \(r_0^2\) is that the inverse of the distortion equations is solvable. The inverse is easiest to express in polar coordinates, with the origin being at the center of the image. If \((r \cos \theta, r \sin \theta)\) is a point in the projective image plane, then the corresponding point in the radially distorted image, \((r_0 \cos \theta_0, r_0 \sin \theta_0)\), has the same angle \(\theta_0 = \theta\) and has radius

\[
r_0 = \sqrt[3]{\frac{r}{2k_1} + \sqrt{\frac{r^2}{4k_1^2} + \frac{1}{27k_1^3}}} + \sqrt[3]{\frac{r}{2k_1} - \sqrt{\frac{r^2}{4k_1^2} + \frac{1}{27k_1^3}}}.
\]

Undoing the radial distortion gives coordinates that lie in a plane in front of the robot. The remaining step is to find out how far away that plane is, so that the plane can then be scaled to be one unit in front of the robot. It is easy to find the distance to the plane because the horizontal and vertical fields of view are known. The distance to the image plane is

\[
\text{distance to image plane} = \frac{\text{image half - width}}{\tan(\text{half horizontal field of view})}.
\]

It is important that the image half-width used here be the flattened image’s half width, not that of the warped image.
Figure 2-2: The radial distortion created by the camera lens (left) is undone to create an image in which straight lines really do appear straight (right).
Chapter 3

Optical flow

Optical flow is one common, useful piece of information that can be extracted from images taken by a mobile robot [6]. Optical flow describes information present in sequential images about how the different parts of the image are moving. As the robot moves forward, objects that are far away and toward the middle of the image will tend to flow more slowly than nearby objects. Thus, one use for the optical flow is to recover information about the distance between the robot and the obstacles seen in the image.

If the robot moves with pure translational movement, there is a point in the image called the focus of expansion (or focus of contraction), so called because all optical flow is colinear with that point and either away from it or towards it. The focus of expansion is where the robot’s movement vector intersects with the imaginary projective image plane.

An important thing to note is that, most of the time, the optical flow varies smoothly across the image. As will be shown later, the magnitude of optical flow is a continuous function of how far away the object is and of the distance (in the image) of the object from the focus of expansion. Thus, as long as the depth to each point in the scene varies continuously, so will the optical flow at that point.
As the robot moves forward, point \( p \) moves to point \( q \) in the coordinate system relative to the robot, and the optical flow at \( p' \) is the vector from \( p' \) to \( q' \). Since the robot is moving straight forward, the focus of expansion is at the center of the image plane.

### 3.1 Optical flow methods

Suppose a pair of images \( I_1 \) and \( I_2 \) are given, between which the robot has moved incrementally. (This work will extend to a sequence of images by considering them in sequential pairs.) There are many different ways to find the optical flow given a pair of images, each way having different advantages and disadvantages as far as speed and correctness are concerned [4, 2, 11]. The way most optical flow algorithms proceed is to try to match corresponding points in the first and second images.

#### 3.1.1 Patch matching

The simplest method of deciding whether a point \( x_1 \) in \( I_1 \) corresponds to a point \( x_2 \) in \( I_2 \) is by comparing a small square region (a "patch") of image intensity around each of them. Each pixel in the patch \( p_1 \) around \( x_1 \) in \( I_1 \) is compared to the corresponding pixel in the patch \( p_2 \) around \( x_2 \) in \( I_2 \), and the difference between them found. These individual pixel differences are combined into a patch difference, usually by totalling...
the sum of their squares.

One issue with patch matching is that the viewpoint has changed between images, and thus no place looks exactly like it did before. Despite this, corresponding patches can really be expected to look alike enough to match, as long as the robot only moves a small distance. This small movement guarantees that the viewpoint doesn’t change very much, and while globally the picture may look somewhat different, locally patches will mostly stay the same.

A second issue with patch matching is having a patch contain a discontinuity in the optical flow. Having the optical flow be mostly continuous across the image is good for patch matching, because it means that the small neighborhood around $x_1$ will tend to optically flow in the same direction as $x_1$ itself. However, when the robot sees a nearby object that visually overlaps a more distant object, there is a discontinuity in the scene depth at that boundary, and therefore in the optical flow as well. Part of the farther object will be occluded or revealed as the robot moves, and this will likely cause simple patch matching to fail at that location.

The problem of patches containing a discontinuity in the optical flow can be partly avoided by making the patch size small, so that fewer patches contain discontinuities. This is a good idea anyway because, as explained above, objects look different from different angles, but they’re similar enough to match well if the robot movement is small and if only a small neighborhood is considered. There is, however, a limit to how small patches can be made and still contain enough information to do the matching well. Having the patch size be too small leads to many false positives.

Making patches small cannot eliminate this optical flow discontinuity problem because the places that give the most reliable optical flow are those places with interesting texture, and often object boundaries fit this description. These optical flow boundaries sometimes cause patch matching to fail altogether, which cannot be avoided. Sometimes at a discontinuity, though, the patch contains an object whose optical flow is dominant, and this optical flow becomes the optical flow of the patch (see figure 3-2).

For the purposes of finding obstacles, this should not pose a serious problem. If
any optical flow is found where a near object overlaps one or more far objects and if the correct optical flow is discontinuous, then the flow found by patch matching should be continuous with whichever object is closest to the robot. This is because while the back object may be occluded or revealed, the front object’s movement will always be seen by the robot, so any found optical flow should match this front movement.

Since one of the primary purposes of finding obstacles in the world is to avoid them, it is a good thing that the closer of several objects will be detected at a discontinuity. Preferring false positives to false negatives when finding obstacles allows the robot to be sure of avoiding crashes. Hypothetically, the optical flow found at a discontinuity could make an object appear to have a slightly larger extent than it actually has, and this could cause the robot to dismiss as blocked small gaps that it can actually squeeze through. In general, though, such cases are few and far between.

### 3.1.2 Other optical flow methods

A slightly more complicated variation on patch matching involves running a filter on the images, then finding image point correspondence not by comparing intensities but by comparing the corresponding outputs of the filter. Such variations might apply wavelet or Gabor filters of differing orientations and scales, or perhaps a Laplacian filter.
Another local optical flow method uses image intensity derivatives, rather than straight image intensities as in patch matching. The rate of change in image intensity for each point is calculated, as well as the horizontal and vertical image intensity gradients. Under the assumption that image intensity of each object remains constant over time, this method uses a gradient constraint equation,

$$\nabla I(x, t) \cdot v + \frac{\partial I(x, t)}{\partial t} = 0,$$

where $I(x, t)$ is the image intensity of pixel $x$ at time $t$ and $v$ is the image velocity at the point $x$, i.e., $v$ is the optical flow to solve for.

A different, global approach to finding optical flow is to assume that it can be parameterized nicely over some portion (or all) of the image and then to find the best fitting parameters. This works quite well when the composition of the world is simple, permitting a parameterization with a low number of parameters. While the correct way to construct this parameterization can theoretically be deduced from the image (e.g., the work of Ju et al. [10]), such algorithms do not yet operate in real time, meaning that the parameterization really must be known beforehand. In the cases where this parameterization approach applies, it certainly gives a more accurate optical flow than other methods, because it integrates information over a large area of the image and is therefore less susceptible to local errors.

I chose patch matching to calculate optical flow because it is the simplest and most general method. In order to get good temporal image derivatives, the differential techniques generally require several frames of similar motion in a row, and this simply cannot be guaranteed on a mobile robot that is likely to be processing images at under ten frames per second but at the same time weaving around to avoid obstacles. The parameterization method is undesirable because, in general mobile robotic situations, the make-up of the world cannot be known ahead of time. This makes it impossible to decide ahead of time what a good model of the optical flow will be. Methods like that of Ju et al. [10] that create the parameterization on the fly are far too slow to be useful on a real-time mobile robot.
3.2 Discretization issues

The patch-matching method of optical flow has problems with discretization. The world is a continuous place, yet images are composed of a discrete number of pixels. The usual camera model assumes that each pixel’s intensity value is an integration of all the intensity values corresponding to its particular solid angle. This becomes a problem when looking at scenes with sharp discontinuities in the image intensity. If normal patch matching were to be used, then these two patches in figure 3-3 would clearly fail to match, even though they are looking at the same object.

![Image of two grid patterns with black splotches and grey shading](image)

Figure 3-3: (a) A single black splotch surrounded by white, when entirely contained within a pixel, causes that one pixel to be black. (b) The same black splotch, when straddling the boundaries of 4 different pixels, causes each of those pixels to have a light grey color.

A method proposed by Birchfield and Tomasi [5] for dealing with this problem is to allow patches to match each other “within half.” This means that one patch can match another patch at an integral pixel location even if the exact point where the match occurs is up to one half pixel away horizontally and/or vertically. In this scheme, for instance, if a patch around (100, 100) in $I_1$ matches with one around (96, 102) in $I_2$, the actual correspondence might be $(100, 100) \leftrightarrow (96.2, 101.5)$ or
To more easily understand how the method works, consider one-dimensional patches. Figure 3-4 shows three adjacent pixels in a patch, along with their corresponding intensity values before and after movement.

![Figure 3-4: In image I₁, the three pixels a, b, and c are viewing a single black square which covers pixel b. In image I₂, the camera has been shifted slightly to the left, so that the scene appears to have moved to the right. The black square now covers both pixels b and c, and causes their reported intensity values to be different shades of grey.](image)

The intuition behind matching within half is to linearly interpolate pixel intensity values. This allows the intensity values at non-integral pixel coordinates to be filled in. When figuring out the pixel difference at two points with integral coordinates, either of the points may be shifted by up to half a pixel horizontally (and/or vertically in 3D) to find the best possible match. In figure 3-4, the intensity values of pixel b are not the same in images I₁ and I₂. The intensity of b in I₂ does, however, match an interpolated intensity value that is within half a pixel of b in I₁. This gives the pixel difference between I₁ and I₂ at pixel b to be 0. Thus, the pixel difference is considered to be 0 if some permittable shift allows the intensity values (interpolated or otherwise) to match exactly. If no perfectly matching offset for either pixel can be
found, then the pixel difference is reported as the smallest obtainable difference over all permittable shifts.

This method at first may seem intractable because every offset up to a half is allowed; it seems to require an infinite number of shifts and tests. Because linear interpolation is used, however, each pixel can be associated with a range of intensity values that it will match exactly. This perfect-match range will always be dictated by just three values (more in the two-dimensional case). These three values are the intensity at the pixel itself, and the interpolated intensities halfway towards each of its neighbors. The maximum and minimum of these three values gives the perfect-match range for this pixel. The pixel difference between this pixel and any other is 0 if the perfect-match range contains the other pixel’s intensity value; otherwise, it is the distance from the other pixel’s intensity value to the closest endpoint of the range.

The extension to two-dimensional patches is relatively straight-forward, with the only question really being whether to allow or disallow interpolating towards diagonally adjacent pixels and using them to calculate a pixel’s perfect-match range. Since there is nothing to prevent objects from moving diagonally in the image, the correct choice is to allow diagonal adjacencies. Once again, the perfect-match range is dictated by the intensity values at a pixel itself and when shifted halfway towards each of its neighbors. When shifting a pixel diagonally, it is important to note that it is incorrect to simply interpolate the intensity values of this pixel and the adjacent one. This is because when the shifted pixel will actually overlap two more adjacent pixels (see figure 3-5). The diagonally interpolated intensity values are one quarter of each of the overlapped pixels, i.e. $\frac{1}{4}(A + B + C + D)$, rather than the simple diagonal interpolation, which would be $\frac{1}{2}(B + C)$.

The pixel-by-pixel differences using the within half method are combined the same way that normal pixel differences are, using a sum of squares. These pixel differences are not combined in any way that shares information about which direction and how far the individual pixels may have been shifted when calculating the pixel differences. (This data is not even explicitly calculated.) Thus, adjacent pixels do not have to
Figure 3-5: Pixel C's perfect-match range is being determined. When it is shifted up and to the right, it overlaps pixels A and D as well as pixel B.

have the same shifts. Some pixels in a patch may shift left when calculating the pixel difference, while others may shift right, up, down, or any other direction. This lack of correlation and the inherent optimism in the method ("take the lowest pixel difference available!") combine to cause some false positive patch matches.

This lack of intra-patch correlation could actually be avoided by artificially increasing the image resolution by some large factor, interpolating the extra pixels, then doing exact patch matching. This would require much more memory and processing time, however, and Birchfield and Tomasi found that doing this explicit interpolation did not give significantly better results.

3.3 Sub-pixel matching

The method of patch matching within half is sufficient if all that is needed is that a good match is found when one exists. There are some cases, though, in which it is desirable not only to know approximately where the match occurs but to achieve sub-pixel matching resolution. For instance, this would give much more precise optical flow, and it would especially help in the cases where the optical flow is so short that the discretization error is a significant fraction of the optical flow's length.

A way of doing sub-pixel matching is by locally interpolating patch difference values and taking the point at which this interpolation reaches its minimum. Following the work of Nishihara [13], a paraboloid can be fit to the patch differences calculated at integral coordinates. The points for fitting the paraboloid are chosen to be the
patch differences at the best matching point and at its four orthogonal neighbors. The apex of the fitted parabola gives the likely location of the best sub-pixel match.

The method is developed by Nishihara using pixel differences not on the original image intensities but on a filtered version of the image. After blurring the image to reduce the effects of image noise, a Laplacian transform is applied, and finally the sign of each pixel is taken; the result is known as a SLOG (signed Laplacian of Gaussian) image. This produces a black and white image where an edge between two differently colored regions in the image gives rise to stable black and white regions along it in the SLOG image. At the time, this preprocessing was necessary in order to be able to run the computations in real time. It throws away most of the information from the intensity image, though, and is extremely susceptible to noise in areas with no texture. Operating on image intensities in real time is now possible and provides results that are clearly superior to those using SLOG images.

The equations for this sub-pixel matching strategy are given below. It is important to note that, if we use a point and its four orthogonally adjacent points to define a paraboloid, then the two directions are independent of each other in that the apex’s x- and y-coordinates can be found separately.

Since each dimension can be handled separately, consider a one-dimensional case. Suppose that three points, \((x-1, \alpha), (x, \beta), \text{ and } (x+1, \gamma)\), are given. The parabola \(y = ax^2 + bx + c\) through these three points satisfies

\[
\begin{align*}
\alpha &= a(x-1)^2 + b(x-1) + c \\
\beta &= ax^2 + bx + c \\
\gamma &= a(x+1)^2 + b(x+1) + c.
\end{align*}
\]

These equations give \(a, b, \text{ and } c\) to be

\[
\begin{align*}
a &= \frac{\alpha - 2\beta + \gamma}{2} \\
b &= \frac{\gamma - \alpha}{2} - (\alpha - 2\beta + \gamma)x
\end{align*}
\]
\[ c = \beta - \left( \frac{\gamma - \alpha}{2} \right)x + \left( \frac{\alpha - 2\beta + \gamma}{2} \right)x^2. \]

A parabola's apex is found at \(-\frac{b}{2a}\), so the \(x\)-coordinate of the apex will be

\[ \frac{-b}{2a} = x - \frac{\gamma - \alpha}{2\alpha - 4\beta + 2\gamma}. \]

This is done in the horizontal and vertical image dimensions separately, and the results are combined to find the exact sub-pixel point where the patches match best.

This sub-pixel matching method will only work (i.e. quadratically interpolate the way it is supposed to) if the patch differences are exact and not within half. In other words, the points to which the paraboloid is fit must have precisely those patch differences at precisely those places. The within-half patch matching, however, only claims that the patch differences it returns are valid for some unspecified location within half a pixel of the matching point. Without knowing exactly where those differences occur, the fitting paraboloid cannot be determined; in particular, only the value of the best match is known, and not its location. Clearly, then, the parabola-fitting cannot blindly base its calculations on the patch differences returned by the within-half method.

One possible solution would be to find the corresponding simple patch differences and use those to fit the parabola. Using this solution, though, the parabola's apex is no longer guaranteed to be within one half pixel of the best within-half match. This is because the location at which the lowest within-half patch difference occurs may not be the location at which the lowest simple patch difference occurs.

Since the properties of within-half patch matching make it so desirable, a different way of paraboloid-fitting must be found. The solution lies in noticing what is implied by the assumption that the patch difference curve looks locally like a paraboloid. Since paraboloids are strictly increasing away from the apex, we conclude that the surrounding within-half patch differences actually occur at a location that is nearest to the apex yet still within the allowable shift of half a pixel. This desired shift is the maximum shift of half a pixel towards the apex. This gives us the precise location of
all of the four surrounding within-half patch differences. The only within-half patch difference whose location remains unknown is the one in the center. While there is not an exact sub-pixel location known for the center patch difference, just having the difference itself is enough to determine a paraboloid. This is because there is just one parabola that fits two points and a height (the patch difference) for the apex.

Here are the equations for finding the fitting parabola with such information, once again done in one dimension for the sake of clarity. Suppose that points \((x - 1, \alpha), (x, \beta),\) and \((x + 1, \gamma)\) are given, where \(\alpha, \beta,\) and \(\gamma\) are the within-half patch differences occurring at \(x - 1, x,\) and \(x + 1.\) The patch differences \(\alpha\) and \(\gamma\) are deduced to actually occur one half pixel in toward the apex, so they are really 1 pixel apart. In order to simplify the equations, the origin is moved to lie at the apex of the parabola. The apex has a \(y\)-coordinate of \(\beta,\) so let \(u = \alpha - \beta\) and \(v = \gamma - \beta\) be the modified patch differences for the left and right points. The \(x\)-coordinate of the apex is unknown, but the apex is between the two outer points, so let \(w\) be the distance from the left point to the origin (see figure 3-6).

Figure 3-6: The origin is moved to lie at the apex of the parabola being fit.

The parabola, by virtue of having its apex at the origin, has form \(y = ax^2.\) We
thus have two unknowns, \( a \) and \( w \), and two points on the parabola:

\[
\begin{align*}
    u &= a(-w)^2 \\
    v &= a(1 - w)^2.
\end{align*}
\]

Solving for \( w \) gives

\[
    w = \frac{v \pm \sqrt{uv}}{v - u}.
\]

These two solutions come from having the apex lie on the axis outside of or between the two given points. Both are equally valid solutions to the equations, but the desired solution is the latter, so

\[
    w = \frac{v - \sqrt{uv}}{v - u}.
\]

Extrapolating this idea to two-dimensional images is done, as with simple paraboloid fitting, by separately calculating the answer horizontally and vertically and then combining those answers into the single point answer. Even using within-half patch differences, then, it is still possible to achieve sub-pixel resolution by quadratic fitting.

### 3.4 Reliable correctness

So far, all that has been discussed about optical flow is how to do the patch matching well. The biggest problem when computing optical flow on real images is that, at a lot of places, there is no way to find the optical flow no matter how good the patch matching is. Consider, for example, approaching a blank white cube. A point in the middle of one of the cube’s faces will look like it could be flowing in any direction; it is only at edges that patches look different from their neighbors. Even edges suffer from the “aperture problem,” where it is impossible to tell where along the edge to match a patch because all of the edge looks the same locally. Only at corners and at places that have corner-ish texture (rather than, say, stripes) can patches be confidently matched.

The natural question to ask is whether the places likely to have reliably correct
Figure 3-7: Consider two successive frames captured by the robot as it moves towards a box. At A, there is no texture, so the correct match is unknown. At B, the texture suffers from the aperture problem. Only at C can the correct match be determined.

optical flow can be picked out of the image. The simplest way to estimate correctness reliability is to keep track of the patch differences when computing the optical flow. If one particular location has many optical flow candidates all with patch differences very close to the best one, then that location is likely not to have a reliably correct optical flow. Unique optimal optical flow candidates, on the other hand, signal a reliably correct optical flow. This method works alright, but its big drawback is the amount of computation required. Places with unreliable optical flow still have to have that flow computed before being recognized as such. It is desirable instead to identify unreliable places in some computationally less intensive way.

A commonly used method of determining locations likely to have correct optical flow is to run an edge filter on the image, finding all the horizontal and vertical edges. Locations are given a likelihood of having correct optical flow based on how “edgy” they are. It is also possible to search for edges at many different image scales and combine these results into a single reliability measure. The intuition is that blank, homogenous areas of the image are not likely to give correct optical flow, whereas those areas with texture or sharp contrasts are more likely to. This method works fairly well, but trouble arises with places like B in figure 3-7. It is clearly possible for an edge to have both horizontal and vertical edginess components and yet still suffer from the aperture problem. Edginess alone, therefore, is not a good enough test, because it only tests for the aperture problem in two directions—horizontally.
and vertically. What is needed is a way to test in all directions.

The key observation is that when $p_1$ matches multiple patches $p_2$ close to the correct one, it also matches multiple patches close to itself in $I_1$. In figure 3-7, for example, patch $B$ would match a patch centered one pixel up and two to the right of itself. This correctly indicates that, in addition to the true matching patch, there will be a false matching patch centered one pixel up and two to the right of the true match.

A good test to determine the reliably correct optical flow locations in an image, therefore, is to measure how well a patch matches those near itself. If $p_1$ does not look similar to any other patch nearby itself in $I_1$, then the optical flow calculations should not falsely match any patches in $I_2$ nearby the true matching patch $p_2$.

If those patches that match others close to themselves are to be marked as having unreliable optical flow, it is important to figure out whether a patch matches one close to itself. It is undesirable simply to use the sum of squares of the pixel differences, for instance. This is because patches in low-contrast areas will naturally have lower patch differences than those in high contrast areas. If this were not taken into account, the sharp edges in the image would be labelled as unreliable and areas with no texture labelled as reliably correct—almost the exact opposite of what the labels should be.

The best way to do patch matching is therefore to have each test take into account the contrast of the patch being matched. The lower the contrast is, the better a patch must match one at a small offset from itself in order to be labelled as reliably correct. Patch contrast can be taken to be some simply computed value such as the difference between the maximum and minimum pixel intensity values in the patch.

### 3.5 Speedups for real-time execution

A lot of the algorithms discussed so far are fine theoretically but require far too much processing power to be feasible in real time. For instance, a single patch matching takes $O(p^2a^2)$ time to find a match or to conclude none exists, where the patch size is $p$ by $p$ and the search area size is $a$ by $a$. Finding reliable places for optical flow
has similar complexity, and these computations are too time-consuming to do for a significant fraction of the points in an image. Finding optical flow using patch matching at each point in a 320 by 240 image with a patch size of 7 and a search area of 11 requires roughly $320 \times 240 \times 7^2 \times 11^2 = 455,347,200$ pixel-to-pixel comparisons.

Greater execution speed can be achieved by reducing the number of points at which optical flow is calculated, and also by reducing the time it takes to calculate optical flow. These two goals can be approached independently of each other, but it is important to be aware of how the former will affect the computations which use the optical flow. Some uses for the optical flow may require it to be densely calculated, ruling out reducing the number of points at which it is calculated.

### 3.5.1 First successful match

One immediately apparent optimization is to halt the patch matching early if a good enough match is found, where “good enough” means that the patch difference is below some appropriate threshold. The search for a matching patch is done starting at the center of the initial guess and moving gradually further away. The search continues until either the entire search area is covered or a good enough match is found. It is important to search at least one pixel beyond whatever good enough match is chosen; if this were not done, then a good enough match might be chosen when in fact the best match occurs one step beyond it. Checking one extra step ensures that the chosen match is at least at a local minimum in the patch difference function.

This optimization relies heavily on the initial guesses being close to correct. The run-time decreases quadratically as the guesses get closer to being correct, because of the reduced area necessary to search for patch matches. At a minimum, though, the initial guesses at optical flow need to be such that the closest good-enough match is the best match.

Having the optical flow guess be one of no movement is a reasonable default, but better guesses can be constructed with the information available during optical flow computation. For instance, robots have inertia and will generally not change their trajectories too much from frame to frame. It is therefore reasonable to assume that
the current optical flow being computed will be similar to the last one computed, and so the previous optical flow can be used to guess at the current one. It is possible not only to use the previous optical flow but also to use whatever part of this optical flow that has been calculated so far. Recall that optical flow usually varies continuously over the image. Thus, if a point has one or more neighbors whose optical flow has already been calculated, then the average of those neighbors’ flows gives a reasonable guess about this point’s optical flow.

A pitfall when implementing this feature is that the optical flow being found might not be reliably correct enough. If it is frequently wrong, it will give bad guesses, which may start subsequent matches on a wild goose chase when it suggests where to search. So, if the optical flow is not correct in general, then only those places which pass a reliable correctness test like the one already described should be allowed to seed future searches.

### 3.5.2 Multi-level matching

A common method employed to facilitate quicker patch matching is that of using multi-level, hierarchical matching [3]. For each image, several copies are made, each one at a resolution successively reduced by a factor of $f$ (usually 2). The patch to be matched has similar copies made of it. Starting at the lowest resolution, the appropriate best patch match is found, then that information is used to inform the next higher resolution’s search. It should be apparent that, at each higher resolution, only the $O(f^2)$ places corresponding to the previous resolution’s match need to be considered for possible matches.

This method is very good at dealing with large search areas, which are needed when the robot movement between images is large. It is not absolutely necessary, though, that the robot be able to correctly identify the optical flow of fast-moving objects that it sees. The robot can just decide not to travel in that direction so that it does not collide with whatever is moving. Multi-level matching also has the drawback that it will not co-exist with the optimization of choosing the first successful match. It cannot be easily integrated with any search that progresses outward from a center.
point. This allows for no benefit that could be gained by having a correct guess at the optical flow.

3.5.3 Using the focus of expansion

All optical flow is away from the focus of expansion if the robot is translating forward (and towards the focus of contraction if the robot is translating backwards). So, in the special case that (a) the robot is simply translating forwards and (b) the objects in the image are stationary, it only makes sense to check patch matches which give optical flow away from the focus of expansion. This special case is actually general enough for the task of avoiding stationary obstacles, as long as the robot analyzes frames between which it is translating and not rotating.

Figure 3-8: Optical flow towards and away from the focus of expansion is given by the dashed line. The grey area is the small angle of error.

In practice, it is necessary to check not only points $x_2$ collinear with $x_1$ and the focus of expansion, but also points one pixel on either side and which fall within a small angle of error (see figure 3-8). This is due to discretization issues, and due to uncertainty about the location used as the focus of expansion. Even so, the number of locations that must be checked for patch matches are greatly reduced, especially when using larger search areas. For instance, using an 11 by 11 search area and assuming flow away from the focus of expansion, the number of patch comparisons might be reduced from 121 to 18 (marked with a white 'x' in figure 3-8).
This method of using the focus of expansion to narrow the search for matching patches works very well experimentally. It complements the strategy of stopping at the first successful match, because it tells that strategy which direction to progressively search in. It has the added bonus of automatically pruning those possible optical flows which must be incorrect because they don’t flow away from the focus of expansion. Thus it gives a boost to correctness as well as to speed. The only downside to this method is that it assumes that the focus of expansion is already known. Some methods of calculating the focus of expansion use the optical flow as input.

### 3.5.4 Faster correctness confidence

The method described above for picking out places in the image likely to have correct optical flow works well but is computationally expensive. Comparing each patch against locations near it in the same image is very similar to the patch matching that happens when finding optical flow. The number of pixel comparisons it would take to scan a 320 by 240 image using this method, searching for self-matches up to 2 squares away with a patch size of 7 by 7, would be $320 \times 240 \times 7^2 \times 5^2 = 94,080,000$. This is too large a number of comparisons to do each frame given current computational power.

Recall, though, that the point of comparing a patch to those nearby itself was to see if the patch suffered from the aperture problem. It would really be sufficient to see whether a patch matches itself when shifted a fraction of a pixel in any possible direction. One problem with this formulation is that there are an infinite number of possible directions in which to try shifting a fraction of a pixel. Also, even if some small number of these directions were chosen to be tested, it is not possible to compute patch differences at non-integral pixel coordinates, at least not with the method of calculating patch differences described above. However, consider the pixel intensity values in figure 3-9.

The center pixel has the same intensity value as the left pixel, so this center pixel can be thought of as having an aperture problem to the left. Also, assuming that pixel intensities interpolate reasonably, the center pixel would match the interpolated value
slightly up and to the right, because $75 \leq 100 \leq 103$. These observations suggest a good way of telling if a patch suffers from the aperture problem in any one particular direction: check to see if each pixel in the patch will match an interpolated value in that direction.

Of course, there are infinitely many directions to be accounted for, and it is unclear what type of interpolation to do between neighboring pixels. Consider the center pixel in figure 3-9 being checked for the aperture problem at an angle halfway between the neighboring pixels 75 and 103. It could be labelled a match because $75 \leq 100 \leq 103$, or it could be disqualified because the linearly interpolated value at that point would be $\frac{1}{2}(75 + 103) = 89$.

Worrying about such things as precisely interpolating adjacent pixels makes the resulting algorithm too slow and complicated. A simpler, faster method which works well involves dividing the $360^\circ$ of possible directions into eight regions, one for each space between neighboring pixels. The center pixel’s intensity value is compared to each of the eight pairs of adjacent neighbors, using the same interval test as above. If its intensity value lies between the neighbors’ intensity values, the pixel is assumed to suffer from the aperture problem in that direction (see figure 3-10). This gives eight binary values, one for each of the eight regions, which can be packed into a byte. Finding which directions a whole patch matches itself when slightly shifted, therefore, consists of ANDing together the corresponding bytes for its constituent pixels.
Figure 3-10: The eight regions are shown. The regions which pass the interval test (i.e. which show the pixel to have the aperture problem in that direction) are colored grey.

This test does not quite work properly on images taken from a real camera because of image noise. While an intensity of 100 will match the interval between neighboring intensities of 95 and 105, noise may cause these actual intensities to appear to be, say, 96 for the center pixel and 98 and 103 for the neighbors. These observed values are certainly possible given typical camera noise, but they give an incorrect result when the interval test is applied to them.

Compensation for image noise could be done at the ANDing step, in that a couple of errant pixels could be ignored. This involves looking at each direction's boolean value individually and therefore disallows ANDing as a very fast way to combine the information from all constituent pixels. A better solution is to artificially extend the neighbors' interval by a small amount. In the example above, the interval [98, 103] might be extended by 5 on either side to be [93, 108], which then gives the correct result. More false positives will occur at individual pixels, but testing reveals that, in virtually all cases, a false positive in one pixel will be handled by having a true negative in some other pixel in the same patch.

Using this speedup, optical flow can be calculated only at those locations where it is likely to be correct, greatly reducing the number of points at which optical flow needs to be calculated. This reduces the execution time, both in optical flow calcu-
lations and in subsequent calculations which use all calculated optical flow needles. The execution time is reduced by skipping precisely those calculations which produce the least correct results, which makes the resulting optical flow of higher quality. The benefit of extra quality makes this speedup better than something like only calculating the optical flow every $N$ pixels horizontally and vertically. In most images, the number of points with reliable optical flow is a small enough fraction of the total that computations can be run on them in real-time. If the number of points is still too large, then combining these two methods should make the computation manageable.

### 3.5.5 Optical flow between confidently correct points

There is an additional optimization when calculating optical flow only for confidently correct points. It comes from the observation that if part of an object is marked as having reliable optical flow in one image, it will probably be marked as having reliable optical flow in the next image too. In other words, it is almost always the same parts of the same objects that give reliable optical flow. This suggests that the only locations in $I_2$ at which to try patch matching are those that are confidently correct in $I_2$.

The idea of picking out feature points in both images and then only matching feature points to feature points is by no means limited to confidently correct points. Edges, wavelet filters, etc. all provide values to direct the patch matching. The point is, though, that if the reliably correct points are being calculated for each image, and if optical flow is only being calculated at the confidently correct points, then this optimization can be implemented with almost no extra processing required.
Chapter 4

Depth maps from images

Objects nearby the robot move faster in the captured images than objects far away do, and from these differences in image speed, the distances to objects can be calculated. Without data other than a sequence of images, it is impossible to calculate absolute distances to objects. All that can be deduced is how distant each object is relative to others. An simple way to see that only relative distances can be calculated is to consider scaling the world larger by a factor of two, with the robot at the scaling origin. If the robot moves twice as far in the scaled world as it does in the original, then the images captured in both will be exactly the same. Other information is required to figure out the scale of the world and to allow calculation of absolute distances to objects. Such other information might be the actual distance that the robot moved, the size of objects that it sees, or the height of its camera from the floor.

4.1 Focus of expansion and optical flow method

Suppose that a robot captures an image sequence while moving some unknown distance straight forward with no rotation. From each pair of successive frames, the (relative) distance to each object in the image can be calculated using the focus of expansion and the optical flow.

To understand the formulas clearly, consider reducing the problem to two dimen-
sions, looking at a side view of the robot. (The extension to three dimensions is trivial.) The projective image plane is in front of the robot and perpendicular to its direction of motion. Suppose the robot moves incrementally forward, and suppose we observe a particular point before and after movement. The point doesn’t actually move—the robot does—but the point’s coordinates relative to the robot change. Let the robot-relative locations of this point be labelled $P$ before and $Q$ after movement. Let the projections of $P$ and $Q$ on the image plane be $P'$ and $Q'$ respectively. Let the distance from the focus of expansion to $P'$ be $d_1$, and the distance between $P'$ and $Q'$ be $d_2$.

Figure 4-1: This is a side view as the robot moves forward a distance of $b$.

The relative distance to $Q$ that will be calculated is $\frac{a}{b}$. This gives relative distance measurements in multiples of the distance that the robot has moved between frames. Notice that, strictly speaking, this does not give the Euclidean distance between the camera and the point $Q$. Instead, it gives a projected distance; i.e., the distance from the camera to the projection of $Q$ on the robot’s direction-of-motion vector. This projected distance is also known as the frames-to-contact (FTC), because it is the number of frames until the robot would collide with the point $Q$, assuming it moves
the same distance between frames.

Using similar triangles,

\[
\frac{(a + b)}{c} = \frac{e}{d_1} \quad \text{and} \quad \frac{a}{c} = \frac{e}{d_1 + d_2}.
\]

Solving the second equation for \( c \) and substituting into the first gives

\[
c = \frac{a(d_1 + d_2)}{e} \quad \Rightarrow \quad \frac{(a + b)e}{a(d_1 + d_2)} = \frac{e}{d_1}.
\]

Solving for \( \frac{a}{b} \) gives

\[
\frac{a}{b} = \frac{d_1}{d_2}
\]

If we want to know what the distance to the point was before movement, i.e., the
distance to \( P \) rather than to \( Q \), the formula becomes

\[
\frac{a + b}{b} = \frac{d_1 + d_2}{d_2}.
\]

This way of calculating relative distances to objects requires that the location
of the focus of expansion be known. Recall that knowing the location of the focus
of expansion is also useful for other things, such as speeding up the optical flow
calculation process. There are several ways to calculate the focus of expansion, each
with its own advantages and disadvantages.

### 4.1.1 Focus of expansion computation from movement

Given a robot that keeps track of its self-movement precisely, the easiest method of
finding the focus of expansion is to calculate it directly, without any reference at all
to the captured images. The focus of expansion is calculated as the intersection of
the robot’s translation vector with the image plane.

Unfortunately, this method generally does not work well in mobile robots. First,
it relies on the robot’s reported movement exactly matching the robot’s actual move-
ment. Any measurement noise or bumps in the robot’s path get directly translated
into errors in the placement of the focus of expansion. Second, it requires that the robot have hardware which can precisely keep track of its movement. In inexpensive, mass-produced robots, any internal sense of odometry and timing cannot be as accurate as would be needed by this method; it would only be useful as a rough estimate. Finally, it is desirable (though not absolutely necessary) to be able to operate in passive navigation mode, i.e. to discern from images alone what the geometry of the world is. Humans can do this, and in some situations it would be a useful thing to allow a robot to do. For instance, it would allow training a robot to learn the layout of a site for which lots of video has already been taken.

It is theoretically possible to calculate the robot’s movement from images and then to use the calculated movement to figure out where the focus of expansion is. This technique, though, adds an extra step going from image data to the focus of expansion location. The results from using the image data to directly calculate the focus of expansion should be at least as precise, if not more so. Section 4.2 below and work by Irani et al. [8] give examples of how to deduce robot motion from captured images.

4.1.2 Focus of expansion computation using optical flow

A method of calculating the focus of expansion which does not require explicit knowledge of the robot’s movement involves finding the vanishing point of the optical flow. Observe that optical flow is always away from the focus of expansion, and that it tends to be smaller when near the focus of expansion than when far from it. This is apparent from the distance formula $d_1 = \frac{d_2}{d_1}$, where $d_1$ is the distance from the focus of expansion to $x_1$.

Finding this location at which the optical flow vectors have zero length is made difficult by errors in the optical flow. Rather than looking for a single zero vector, it is better to sum the lengths of the optical flow vectors in a small patch and see which patch has the smallest sum. Also, rather than computing these sums for patches over the entire image, doing a gradient descent greatly reduces the number of sums to be calculated and should suffice to find the minimum. If the optical flow is close enough
to being correct, there will be no local minima to get stuck at, so the gradient descent should really find the focus of expansion.

The downside to this method is that it needs the optical flow to be computed densely; as discussed earlier, there are large regions of the image for which the optical flow cannot be reliably correctly computed. The optical flow needs to have a high degree of correctness to ensure that there are no local minima other than the global minimum. Parameterization methods of optical flow computation densely compute the optical flow well, but they are either too slow for real-time use or make restrictive assumptions about the scene composition. If the optical flow is only calculated at reliable points, it will generally not be dense enough to do the desired gradient descent. There will not even necessarily be reliable optical flow at the focus of expansion; this would make it impossible to verify its location even given a correct guess.

4.1.3 Focus of expansion computation from sparse optical flow

The best method I have found to compute the focus of expansion uses optical flow that may be sparse but that has a high likelihood of being correct. The intuition behind this method is to find the least-squares fit to all the lines through the optical flow vectors. Since optical flow points directly away from the focus of expansion, it makes sense to try to find a point that is as close to being collinear with all the flow vectors as possible.

Recall that the distance from a point \((x, y)\) to a line through the points \((x_0, y_0)\) and \((x_1, y_1)\) is

\[
\frac{|(y_0 - y_1)x + (x_1 - x_0)y + (x_0y_1 - x_1y_0)|}{\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}}.
\]

Indexing the points by \(i\), the least squares minimization is

\[
\min_{(x_{FoE}, y_{FoE})} \sum_{i=1}^{n} \frac{|(y_{0i} - y_{1i})x_{FoE} + (x_{1i} - x_{0i})y_{FoE} + (x_{0i}y_{1i} - x_{1i}y_{0i})|}{\sqrt{(x_{1i} - x_{0i})^2 + (y_{1i} - y_{0i})^2}}^2.
\]

Taking the absolute value in the numerator is unnecessary because the expression is
squared. The resulting minimization, converted to matrix form, becomes

$$\min_{(x_{\text{FoE}}, y_{\text{FoE}})} \left\| \begin{bmatrix} \vdots \\ \frac{y_0 - y_i}{\sqrt{(x_i - x_0)^2 + (y_i - y_0)^2}} \\ \vdots \\ \frac{x_0 - x_i}{\sqrt{(x_i - x_0)^2 + (y_i - y_0)^2}} \end{bmatrix} \begin{bmatrix} x_{\text{FoE}} \\ y_{\text{FoE}} \end{bmatrix} - \begin{bmatrix} \vdots \\ \frac{x_i y_0 - x_0 y_i}{\sqrt{(x_i - x_0)^2 + (y_i - y_0)^2}} \\ \vdots \end{bmatrix} \right\|^2.$$ 

This method of finding the focus of expansion works well and requires little processing power (taking the optical flow as already computed). It is also not as demanding as the previous method, because it does not require that the optical flow be dense. In fact, it will work with just two points' worth of optical flow data, albeit poorly. The big downside of using this method is that the optical flow needs to be calculated first. Ordering the optical flow computation before finding the focus of expansion means that the focus of expansion cannot be used to speed up the optical flow calculations as described previously.

It seems to be desirable but impossible to use the focus of expansion and the optical flow to calculate each other, to derive the benefits of knowing one when computing the other. It is possible, though, to partially derive the benefits of using each to calculate the other by using an iterative process. Even a rough guess at the location of the focus of expansion can still accelerate the optical flow calculations considerably. So, given an initial approximate guess at the focus of expansion, perhaps due to calculations from robot movement, the optical flow can be calculated relatively quickly on a small number of points in the image. It is important that the few optical flow vectors calculated in this step be scattered across the entire image; allowing all parts of the image to contribute means that the conclusions drawn from this data will be likely to accurately represent the entire image. It is also important that these few optical flow vectors have a high probability of being correct; this can be arranged by using the correctness confidence measure from above. This sparse optical flow can be used to calculate a highly accurate estimate of the focus of expansion, which can then in turn be used to calculate the entire optical flow.

There are two points to note when finding the focus of expansion at the intersection
of optical flow lines. One is that it only makes use of the direction of each point’s optical flow, rather than also using its distance. The reason is that the length of a point’s optical flow vector is dependent not only on its distance from the focus of expansion but also on how close the object is to the robot. While optical flow vectors are generally shorter near the focus of expansion, they do not uniformly and strictly approach zero unless the robot is approaching a large, relatively flat surface. In many cases, including the important case of approaching a thin nearby obstacle, this generalization is invalid. It is theoretically possible to use stored information about the surrounding obstacles to guess how far away points are and use the guesses to better inform the focus of expansion calculations. This would be far more complicated than the direct method, however, and would provide little benefit.

The other point of note is that each individual optical flow vector is given equal weighting in determining the focus of expansion. If some correctness confidence measure is available, then it can be used to weight the process in favor of optical flow vectors more likely to be correct. Suppose the optical flow vector \( i \) is given weight \( w^i \). Then the least squares minimization to perform is

\[
\min_{X_{\text{FoE}}, Y_{\text{FoE}}} \left( \begin{array}{c}
\vdots \\
\frac{w^i(y_1^i - y_0^i)}{\sqrt{(x_1^i - x_0^i)^2 + (y_1^i - y_0^i)^2}} \\
\vdots \\
\end{array} \right) \left( \begin{array}{c}
\vdots \\
\frac{w^i(x_1^i - x_0^i)}{\sqrt{(x_1^i - x_0^i)^2 + (y_1^i - y_0^i)^2}} \\
\vdots \\
\end{array} \right) \left( \begin{array}{c}
X_{\text{FoE}} \\
Y_{\text{FoE}} \\
\vdots \\
\end{array} \right) - \left( \begin{array}{c}
\vdots \\
\frac{w^i(y_1^i - x_1^i - y_0^i)}{\sqrt{(x_1^i - x_0^i)^2 + (y_1^i - y_0^i)^2}} \\
\vdots \\
\end{array} \right).
\]

Of particular concern when paying attention only to the direction of each optical flow vector is the noise added by discretization. For instance, an optical flow vector from (100, 100) to (102.4, 101.6) will get discretized as being from (100, 100) to (102, 102), giving an angle that is different by over 11°. Note that the discretization error can only change each coordinate by up to 0.5, so it will affect short vectors much more than long ones. A reasonable way of compensating for discretization error, then, is to weight long vectors more than short vectors in determining the focus of expansion. “Long vectors” might be those which have length greater than some specified value, so that the individual discretization errors are upper-bounded.
4.1.4 Other ways to find the focus of expansion

A variation on the above method is given by Jain [9]. The focus of expansion is expressed as the best collinear fit with the optical flow, but this method uses the triangle inequality instead of distance to a line. If points $A$, $B$, and $C$ are given and $d(x, y)$ is the Euclidean distance between $x$ and $y$, then $d(A, B) + d(B, C) \geq d(A, C)$, with equality holding exactly when $A$, $B$, and $C$ are collinear in that order. If $A$ is some point (possibly the focus of expansion) with optical flow from $B$ to $C$, then $d(A, C) - d(A, B) \leq d(B, C)$, with equality holding exactly when $A$ is the focus of expansion.

Suppose the calculated optical flow is from points $\{f_i\}$ to corresponding points $\{f'_i\}$. For each optical flow vector and for any point $A$,

$$d(A, f'_i) - d(A, f_i) \leq d(f'_i, f_i).$$

Summing these equations over $i$ gives

$$\sum_i [d(A, f'_i) - d(A, f_i)] \leq \sum_i d(f'_i, f_i)$$

$$\sum_i d(A, f'_i) - \sum_i d(A, f_i) \leq \sum_i d(f'_i, f_i)$$

Plugging the focus of expansion in for $A$ gives equality, or should give equality if all the optical flow vectors are exactly aligned with it. To compensate for small errors, the focus of expansion is chosen to be the $A$ that maximizes the left hand side. Note that the left hand side sums use $\{f_i\}$ and $\{f'_i\}$ separately, meaning that finding the correspondence between the $f_i$ and $f'_i$ is unnecessary. This is potentially a big win because, rather than having to find the optical flow, it is only necessary to find the same feature points in both images. Depending on what those feature points are, this could be a much easier and much quicker process than finding the optical flow.

The downside to this method is that it relies on finding exactly the same feature points in both images. This may or may not be a valid assumption depending on how the feature points are chosen. One or two stray points will not affect the results.
considerably, but if the number of feature points in each image is significantly different, there is no easy way to tell which are erroneous. It may not even be the case that any are erroneous—there may simply be more feature points—but there is no way of recovering and finding the focus of expansion accurately without a lot of extra computation which would negate the benefits of this method.

A different approach altogether is given by Negahdaripour and Horn [12] and has to do with image intensity gradients. The spatial derivatives of image intensity are calculated, then the best motion fit is found which preserves the brightness constancy assumption. This approach has the same issue as all of the methods discussed so far, in that it assumes translational motion. Rotational motion adds much more complexity, computational cost, and probability of false results. Another problem with this method is that it is necessary to calculate the image intensity derivatives which, as mentioned earlier, are difficult to calculate correctly on a meandering robot running at 10 frames per second or less.

4.1.5 Accounting for rotation

Purely translational movement is assumed by the depth map calculations which use the focus of expansion. Unfortunately, a robot will not necessarily move in a perfectly straight line even when it is programmed to. It may rotate due to motor error, or it may drive over bumps and other anomalies on the floor.

Notice that, unlike translational movement, robot rotation causes each part of the image to have optical flow solely based on where it is in the image plane, and not based on how far away objects are. If the robot knows how it has rotated, therefore, it can subtract out the rotation without needing to do any distance-to-object calculations. It can then assume translational movement, proceeding with depth map calculations which use the focus of expansion. As discussed above, though, factors external to the robot make consistently measuring its ego-motion accurately a futile task. It also may not be possible to know the robot’s movement, such as when doing passive navigation.

One possible way to figure out the robot’s rotation is to plug the optical flow
into an algorithm that takes image point correspondence and calculates the robot’s movement. Such an algorithm is presented below in section 4.2. Of course, such algorithms give distances to the image points directly, without having to do the extra step of finding the focus of expansion.

Another possible way of dealing with rotation between images is to subtract it out by holding the background still. Objects which are far away from the robot can essentially be thought of as stationary with respect to it, because the distance they move relative to the robot is insignificant relative to how far away they are. So, if the robot can transform the images so that the “stationary” background actually appears stationary, then it will have subtracted out the rotation. Irani et al. [8] present a method for doing this, as well as a procedure to use the found image warping to deduce the robot’s motion.

### 4.2 Least-squares minimization method

A method of finding the depth to objects in the image without needing the focus of expansion is given by Weng and Huang [15]. It uses least-squares minimization to find the robot motion that best fits the optical flow data; using these calculations, it estimates the relative distance to the points for which there is optical flow.

Suppose some point has positions relative to the robot of \( x = (x, y, z)^t \) before and \( x' = (x', y', z')^t \) after movement. The projections of these points on the image plane are \( X = \frac{x}{y} \) and \( X' = \frac{x'}{y'} \). Let \( R \) be the rotation matrix and \( T \) the translation vector corresponding to the points’ movement relative to the robot, so that

\[
x' = Rx + T, \\
\text{or } y'X' = yRX + T \text{ in image space.} \tag{4.1}
\]

Let \( T_s \) be a unit vector aligned with \( T \). Dividing both sides of (4.1) by \( ||T|| \) and
pre-crossing by $T_s$ gives

$$T_s \times \frac{y'}{||T||} X' = T_s \times \left( R \frac{y}{||T||} X + \frac{T}{||T||} \right)$$

$$= T_s \times R \frac{y}{||T||} X$$

$$= \frac{y}{||T||} T_s \times RX$$

$$= \frac{y}{||T||} [T_s] \times RX, \quad (4.2)$$

where the $[\cdot] \times$ operator converts a 3-vector $v$ into a 3 by 3 matrix $[v] \times$ such that $[v] \times v' = v \times v'$. Pre-multiplying both sides of (4.2) by $X'^t$

$$X'^t \cdot \left( T_s \times \frac{y'}{||T||} X' \right) = X'^t \cdot \left( \frac{y}{||T||} [T_s] \times RX \right)$$

$$0 = X'^t \frac{y}{||T||} [T_s] \times RX$$

$$= X'^t [T_s] \times RX,$$

where the left-hand side is zero because $T_s \times X'$ is perpendicular to $X'$ and so the dot product vanishes.

This method goes on to find the best matrix $E = [T_s] \times R$ which is the simultaneous least squares minimization of the equations $X'^t EX_i$, where the optical flow is given by the points $\{X_i\}$ to $\{X'_i\}$. It then uses least squares minimization to estimate $T_s$ by noticing that it should be orthogonal to all three columns of $E$. With $T_s$ in hand, it is straightforward to find $R$ and then $T$. Strictly speaking, $T$ cannot be computed because of the problem that distances can only be computed relatively rather than absolutely; what actually gets calculated is $\hat{T} = \frac{T}{||T||}$. This gives the motion of the points and therefore the motion of the robot.

Having the points’ motion allows the relative depths $\tilde{y}_i$ and $\tilde{y}'_i$ to be calculated, where

$$(\tilde{y}_i', \tilde{y}_i) = Y_i = \left( \frac{y_i'}{||T||}, \frac{y_i}{||T||} \right)^t.$$
The relative depths are found by using least squares to minimize

\[
\| \begin{bmatrix} X'_i & -RX_i \end{bmatrix} Y_i - T \|.
\]

This method of finding relative distances to scene objects is better than those which use the focus of expansion because it does not require the computation of an intermediate point, and because it automatically handles any rotation on the part of the robot. Although it is relatively unintuitive, it results in a method that can be computed efficiently online.
Chapter 5

Finding the floor and obstacles

Given optical flow and depth-to-object data, there are many different techniques to detect obstacles. Some of these techniques do not actually find the obstacles; instead, they find the floor; i.e., those locations where there are no obstacles. It may be the case that certain obstacles, such as tables or chairs, leave the floor visible beneath them, so that if the robot assumes that it can drive anywhere it sees floor, it will be mistaken. As the robot approaches such an obstacle, though, the floor beneath the obstacle will gradually disappear, being completely hidden once the robot is right up against the obstacle. If the robot is driving slowly enough, this will be sufficient to allow it to avoid bumping into the obstacle. For the remaining cases, an extra parallel step of processing can be added to notice objects not extending to the floor, calculate how far away they are (based on their optical flow), and mark their locations as occupied.

It may be unclear at first why detecting and labelling the floor is difficult, because theoretically the robot has distances to every object in the image, and so it can just test each point to see if it lies in the current model of the floor plane. One problem is that the orientation of the floor plane with respect to the robot may not always be known. There will be image noise and algorithmic error, which require that the data be checked to see if noise or a freak coincidence (e.g. aliasing in the image) is causing an individual point to be labelled incorrectly. Also, the calculated optical flow cannot be dense and reliable using patch matching. Finding the floor is hard
because it may only be at a few points that the floor's optical flow is reliable, and the goal is to discern the whole visible floor from those few points. Compare this to a computer game where the player travels around in a world drawn in wireframe. The player can tell where the floor is, but it is certainly not due to the optical flow of any textured floor region. It should therefore be possible to use the optical flow at just a few points to figure out the placement of the visible objects.

5.1 Meaning of “find the floor”

What exactly does it mean to find the floor? It means that, for any point in the image, the robot should be able to make an informed guess as to whether that spot is part of the floor. The robot may not give a yes/no answer, but rather a probability. These probabilities can be either be thresholded to decide the floor/obstacle question, or they can be combined over time to allow the robot to be more certain whether a location is occupied by an object.

A difficulty arises because of the low percentage of image points having reliably correct optical flow; this means that only a small fraction of image points can be given a reliable probability of being on the floor. The rest of the image cannot just be ignored, however; for example, that would cause the robot to crash into the middle of a textureless obstacle whose the optical flow is unreliable except at its edges.

Assuming that the optical flow will be of no use at most locations, it is necessary to find something else to fill in the places with unreliable optical flow. Observe that these unreliable points cannot have their optical flow calculated correctly, and yet, for some of them, there is no optical flow consistent with the intensity images that is also consistent with the motion of the floor (as estimated from the reliably correct points). For instance, as a robot advances towards a table, the leading edge may suffer from the aperture problem, giving it unreliable optical flow. There is no optical flow, however, which would be calculated from the images and which marks that leading edge as being on the floor.

An idea that takes advantage of this observation is to check the consistency of the
estimated floor motion with the images. Such an algorithm would use the estimated motion of the floor to calculate an optical flow field. This vector field would tell what each point’s optical flow would be if it were part of the floor. Since the robot has an idea of how certain points on the floor are optically flowing, it can extrapolate from those points to figure out how each image point would flow were it on the floor. This constructed optical flow can then be tested at each point to see if it actually fits the images. This test involves computing the patch difference between the two ends of the constructed optical flow. A positive test result (i.e., the constructed optical flow is consistent with the images) does not necessarily mean anything; a large textureless area will always positively match any optical flow hypothesized for the floor, regardless of the flow’s direction and magnitude. A negative test result, however, indicates that the image point in question is definitely not on the floor. This test can be applied to the whole image to find those locations whose appearance is consistent with being a part of the floor.

Notice that this “floor flow consistency test” does not rely directly on any optical flow; it relies instead upon estimated floor movement (which may in turn be based on optical flow). This means that as long as the floor flow estimate is good, a point’s optical flow may be incorrectly calculated, and yet that point will still be given a correct probability of being on the floor. This is an additional advantage gained by performing the floor consistency test in image space as opposed to 3D space.

5.2 Filling to the nearest edge

A simple, fast method of finding the floor is given by Horswill [7] and uses image intensities, not optical flow. Edge detection is run on the image, and in each column, the edge closest to the bottom of the image is found. Pixels further down in the image correspond to points nearer the robot, so the edge closest to the bottom of the image in each column is the edge closest to the robot in that column. Those nearest edges are assumed to be the nearest objects in those particular directions, meaning that all the space in between them and the robot must be floor space. This requires
the robot to keep a buffer of visible floor space around it at all times.

This method is undesirable for a mobile robot moving around in the real world, mostly because the nearest edge detected will not necessarily correspond to an obstacle. Consider, for instance, a checkerboard floor, an edge between carpet and tile, or a scuff mark on the floor. All of these would be labelled obstacles and avoided by robots using this method, though a person would have no trouble identifying these features and walking right over them.

5.3 Fitting a plane to 3D points

A better method uses the relative 3D locations which are computed from the optical flow. The simplest way of using this data to find points on the floor is to find a floor plane which fits the available points. Points are labelled more likely to be on the floor the nearer they are to the floor plane.

This not only raises the question of exactly how near the floor plane a point needs to be in order to be thought of as on the floor (and driven over by the robot), but how to find the floor plane in the first place. There are complicated methods to find a subset of points which form a plane; an easy approach which works well enough in simple situations is to iteratively fit a plane using weighted least squares. The idea is to weight points based on how close they are to the previous floor-plane guess. Given new weights, a new floor-plane guess is fitted to the 3D points using weighted least-squares.

The initial floor-plane guess can be derived in many ways. If the robot has a rough idea of its own geometry and orientation, it may know approximately where the floor is. If not, perhaps because it is operating in passive navigation mode, the previous frame's floor plane will be a good guess. If neither of these is a possibility, then texture information or the Horswill algorithm given above can provide a starting point. Regardless of how the process starts, it keeps iterating back and forth between weights and floor plane guesses until both stabilize.

It is important that the initial guess is relatively close to being correct, so that
the floor points get initially weighted highly enough to pull the floor estimate towards them. Even so, it may be that the plane found by this iterative algorithm is not the floor. One usual reason for the floor not being found correctly is that there are hardly any floor points visible with reliably correct optical flow. If these points are too few in number, then there is no way that the correct floor will be identified, no matter how good the initial guess and plane-fitting are. If the robot has enough information to recognize such a situation, it needs to discard the erroneous results and try some other method.

A different approach to finding the floor would be to apply a Hough transform to the 3D points. This allows the dominant planes to be recognized, and selecting the best floor plane then involves applying whatever selection criteria are available. For instance, the ways described above to generate an initial guess at a floor plane will produce points thought to be part of the floor, so that the dominant plane containing the most of these likely floor points could be chosen. Notice that this method avoids committing early to any particular plane, giving it the option of discarding a very well-populated plane in favor of one containing fewer points simply because the latter meshes better with other outside data. The drawback of this method is that it requires an unreasonable amount of memory and computational power to store the Hough transform. This makes it unavailable as a real-time solution for current mobile robots.

When doing the floor flow consistency test described in section 5.1 to determine those points definitely not on the floor, it is necessary not only to know where the floor is, but also the robot’s movement between frames. This movement can be easily found by calculating the locations of points before and after the movement between images. A good guess at the robot’s movement is the average of the 3D motion of these points. Actually, the points are stationary and only “moving” relative to the robot, so the robot’s estimated movement is the reverse of the average point movement.

This method of finding the floor and robot movement by fitting a plane to 3D points is relatively straightforward, but it has its drawbacks. Perhaps the biggest of these is the requirement of transforming each optical flow data point into a 3D point
in order to use it to find the floor. The floor flow consistency test would additionally need to calculate, for each point tested, the optical flow that that point would have if it were on the floor. If the floor plane and movement of the robot are known in 3D space, then they need to be converted to image space to get points’ optical flows. This means that there will likely be one image space to 3D space (or vice versa) conversion for each pixel in the image.

5.4 Fitting a floor flow to optical flow

To avoid costly computations that convert between image space and 3D space, it is desirable to be able to find the floor and robot movement while staying in image space. This involves parameterizing the image space movement of points in a plane.

Recall that the equations to project points onto the image plane are $u = x/y$ and $v = z/y$, where $(x, y, z)$ is a point in 3D space and $(u, v)$ is the corresponding coordinate on the image plane. Suppose that $(x, y, z)$ is has an instantaneous rotation speed of $\Omega_x$, $\Omega_y$, and $\Omega_z$ around the $x$-, $y$-, and $z$-axes respectively and translation velocity of $(T_x, T_y, T_z)$. According to Adiv [1], the instantaneous velocity $(\alpha, \beta)$ of the

---

Figure 5-1: The camera-relative coordinate system, showing direction of rotation.
image coordinate \((u, v)\) is

\[
\alpha = -\Omega_x u v + \Omega_y (1 + u^2) - \Omega_z v + \frac{T_x - T_z u}{z} \quad (5.1)
\]

\[
\beta = \Omega_x (1 + v^2) + \Omega_y u v + \Omega_z u + \frac{T_y - T_z v}{z}. \quad (5.2)
\]

As discussed earlier, a plane has the form \(k_x x + k_y y + k_z z = 1\) as long as it does not pass through the origin. Dividing both sizes by \(z\) and substituting into (5.1) and (5.2) gives

\[
\alpha = a_1 + a_2 u + a_3 v + a_7 u^2 + a_8 u v
\]

\[
\beta = a_4 + a_5 u + a_6 v + a_7 u v + a_8 v^2,
\]

where

\[
\begin{align*}
a_1 &= \Omega_y + k_z T_x \\
a_2 &= k_x T_x - k_z T_z \\
a_3 &= -\Omega_z + k_y T_x \\
a_4 &= -\Omega_x + k_z T_y \\
a_5 &= \Omega_z + k_x T_y \\
a_6 &= k_y T_y - k_z T_z \\
a_7 &= \Omega_y - k_x T_z \\
a_8 &= -\Omega_x - k_y T_z.
\end{align*}
\]

Thus, the instantaneous optical flow for each point on any plane (including the floor plane) can be specified by the eight parameters \(a_1\) through \(a_8\). The optical flow calculated above is not actually instantaneous, but given a small enough step between frames it approximates instantaneous optical flow well enough for the purposes of parameterization. All that is required is that the robot only move a small distance between frames.
This method does not separately compute the location of the floor and the robot movement, although it is possible to work backwards from the parameterization to deduce both up to a scalar. If the only goal is to run the floor flow consistency algorithm, though, these two items do not need to be separated. In fact, computing them as a combined item will generally give less noisy results than as separate items. This is because the two separate computations are not guaranteed to react to noisy 3D locations in the same way; when performing only one computation, there is no such possibility.

5.5 Real-world performance and limitations

The floor flow consistency test works well in identifying areas of the image unlikely to correspond to floor points, as can be seen in figure 5-2. It correctly identifies the box as not being part of the floor. It also correctly identifies the sheets of paper as being part of the floor, something that is difficult for methods which use features like image intensity edges to find the floor.

The example in figure 5-2 also highlights the weaknesses of the method. The part of the box that is very low to the floor has an optical flow that is very close to that of the floor it occludes. This means that the patch difference computed for the endpoints of the floor’s optical flow is close to the patch difference computed for the endpoints of the box’s actual optical flow, and so that part of the box is labelled as likely to be part of the floor. The left edge of the box is aligned with the focus of expansion and is therefore not recognized as being above the floor because of the aperture problem. Also, the textureless regions of the box are also labelled as likely to be part of the floor.

These problems can be partially handled by taking into account information other than the optical flow, such as texture or object recognition. For instance, being able to segment the scene by texture or by color would allow entire patches to be labelled as being part of the floor or not based on reliable labels at their corners or edges. These reliable probability labels can be distinguished from unreliable ones by considering
Figure 5-2: The top two images are what the robot saw before and after movement. The bottom-left image shows the points that have been labelled as having reliably correct optical flow. The bottom-right image shows the probability of each pixel being on the floor, calculated using the floor flow consistency test. Lighter areas have been labelled as less likely to be part of the floor than dark areas.

whether locations have reliably correct optical flow.
Chapter 6

Conclusion

Optical flow has been shown to be useful data for mobile robots to use when detecting and avoiding obstacles. Calculating optical flow only requires the robot to be equipped with a camera, which is a relatively cheap sensor considering the large amount of data it gives that allows the objects in the world can be discerned in great detail.

One typical problem with optical flow is that it is normally very unclear how reliable the optical flow is at different points in the image. While it may be alright to avoid this problem by plastering the walls and floor of a lab with highly textured material, this is hardly a feasible for the rest of the world. A robot suitable for mass consumption needs to be able to ignore the large featureless areas it sees while extracting all the information it can from the few good data points. The fast algorithm I developed to locate points with reliably correct optical flow gives a good way to figure out where the few good data points are, and it is fast enough to execute in real-time on every image. It is extremely useful as long as subsequent processing can fill in the gaps between these good data points.

Another typical problem that occurs with the patch matching method of calculating optical flow is that of discretization errors. The parts of the image that have the sharpest edges and texture are going to suffer most from discretization issues, which certainly needs to be addressed since these sharp edges and textures are exactly the places where the optical flow is most reliable. Allowing patches to match within half is a part of the solution but sometimes does not give the optical flow in enough
detail to be useful. I extended the paraboloid fitting of Nishihara [13] to situations when doing patch matching within half, allowing both the benefit of within-half patch matching (being sure to find matches when they exist) and the benefit of quadratic fitting (sub-pixel optical flow resolution).

The optical flow gives 3D points for the parts of the image being tracked, and these points can be used to find obstacles. Finding the obstacles in the world is essentially equivalent to finding the floor around the robot, and it turns out that finding the floor is in some ways easier. The floor can be described in terms of its orientation and position relative to the robot, so that if the robot knows its own movement, it knows the floor’s optical flow. I developed an easier method of finding the floor which works by parameterizing the floor’s movement (relative to the robot) in image space. Working in image space avoids costly transformations to and from 3D space.

Given a model for the floor’s movement relative to the robot, I developed a floor flow consistency test. Each point in the image has an optical flow vector calculated for it that would be its optical flow if it corresponded to a floor point. The image is checked for consistency with this constructed optical flow using patch differences; points that have too large of a patch difference are considered inconsistent. These inconsistent points cannot be part of the floor and therefore correspond to obstacles.

A robot that uses these optical-flow obstacle-finding methods will thus find the corners and edges of obstacles, which are not consistent with floor flow. It will generally be unable to recognize the middle of obstacles, though, solely based on the optical flow data and interpretations thereof. This is because algorithms that use optical flow encounter difficulties when the optical flow is locally ambiguous. Such situations occur in large non-textured areas and also in edges which suffer from the aperture problem. Optical flow then seems to be a useful tool in identifying parts of obstacles, but it needs to be complemented by other sorts of information extracted from the images in order to allow a robot to robustly detect and avoid obstacles. As described in section 5.5, segmenting the image by texture is one possibility that would help fill in the reliable data gotten from optical flow processing. Another would be to classify seen objects into categories that allow them to be given likely 3D shapes
and dimensions.

Considering the calculation of the optical flow itself, there are various possibilities for improving its accuracy and precision. Processing more than two frames at once, for instance, will help deal with image noise and discretization errors, although it would require careful attention to the image alignments. Methods of calculating the optical flow other than the simple patch matching described here are able to avoid some of the issues associated with it. Matching based on the output of wavelet filters, for instance, may avoid discretization issues and the aperture problem, and optical flow parameterization methods like that of Ju et al. [10] will become very attractive if they attain enough accuracy and speed for real-time processing.
Bibliography


