Pixel-level Data Fusion Techniques Applied to the Detection of Gust Fronts

by

Samuel M. Kwon

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Science and Engineering

and

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

© Massachusetts Institute of Technology 1994. All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
May 2, 1994

Certified by .....
Richard L. Delanoy
Lincoln Laboratory Staff
Thesis Supervisor

Certified by .....
W. Eric L. Grimson
Associate Professor
Thesis Supervisor

WITHDRAWN
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
FROM
MIT LIBRARIES
JUL 18 1994
LIBRARIES

Accepted by .....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Pixel-level Data Fusion Techniques Applied to the Detection of Gust Fronts

by

Samuel M. Kwon

Submitted to the Department of Electrical Engineering and Computer Science
on May 2, 1994, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Science and Engineering
and
Master of Science in Electrical Engineering and Computer Science

Abstract

Automatic detection and tracking of gust fronts from radar data is a difficult task for three reasons: there are multiple features associated with gust fronts in radar data, none of these features are unique to gust fronts, and the discriminatory power of each feature varies for several reasons. As a result, a number of physical properties must be considered simultaneously, and the information gained from each must be properly combined in order to achieve good overall detection performance. In this thesis, existing tools of data fusion were improved and applied to the fusion of radar data in the context of gust front detection. A mechanism for computing pixel-level weights was developed and tested with an eye towards eventually fusing evidence from two radar sources, the Next Generation Weather Radar (NEXRAD) and the Terminal Doppler Weather Radar (TDWR). The technique of using pixel-level weights was tested and automatically scored using TDWR data that had been labelled by human analysts. Both improved detection and a reduction of false alarms were seen, thereby demonstrating the value of using pixel-level weights for combining evidence of varying reliability from several detectors in the context of gust front detection.

Thesis Supervisor: Richard L. Delanoy
Title: Lincoln Laboratory Staff

Thesis Supervisor: W. Eric L. Grimson
Title: Associate Professor

Contents

1 Introduction **9**

1.1 Motivation 9

1.2 Contribution of Thesis 9

1.3 Thesis Overview 10

2 Gust Fronts **11**

2.1 Physical Description 11

2.2 Motivation for Detection 11

2.2.1 Dangers 11

2.2.2 Planning 13

2.3 Radar Signature 14

2.3.1 Reflectivity 14

2.3.2 Velocity Convergence 14

2.3.3 Motion 15

2.4 Challenges to Detection Algorithm 16

2.4.1 Occlusion 16

2.4.2 Orientation 17

2.4.3 Sensitivity 17

2.4.4 Imitation 19

3 Gust Front Detection **20**

3.1 Design philosophy behind MIGFA 20

3.2 Overview of Machine Intelligent Gust Front Algorithm 21

3.3 Feature Detection, Functional Template Correlation, and Interest Images 24

4 Data Fusion **27**

4.1 Proposed method 27

4.2 Orientation 29

4.2.1 Orientation Considerations for Gust Fronts 31

4.2.2 Computing Weights 33

4.2.3 Augment with confirming and disconfirming weights 35

4.2.4 Implementation 36

4.3 Range 38

4.3.1 Range Considerations for Gust Fronts 38

4.3.2 Computing Weights 39

4.3.3 Implementation 41

4.4 Occlusion 42

4.4.1 Occlusion Considerations for Gust Fronts 42

4.4.2 Computing Weights 43

4.4.3 Implementation 43

4.5 Putting it all together 45

4.5.1 Combine Interest Images 48

4.5.2 Implementation Issues 49

5 Results, Conclusions, and Recommendations **51**

5.1 Results 51

5.2 Evaluation 53

5.3 Conclusion 54

A Code Listing **55**

A.1 Set Up Templates to be used by Weighting Functions 55

A.1.1 Lisp Code 55

A.1.2 C Code 56

A.2 Orientation Weight Computation 58

A.2.1 Lisp Code 58
A.2.2 C Code 58
A.3 Range Weight Computation 60
A.3.1 Lisp Code 60
A.3.2 C Code 60
A.4 Occlusion Weight Computation 61
A.4.1 Lisp Code 61
A.5 Combining Weights 61
A.5.1 Lisp Code 61
A.5.2 C Code 62
A.6 Top Level Function for requesting the computation of pixel-level weights for one interest image 63
A.6.1 Lisp Code 63
A.7 Calling for computation of pixel-level weights in MIGFA 63
A.7.1 Lisp Code fragments 63
A.8 Computing an Average Weighted Interest Image 64
A.8.1 Lisp Code 64
A.8.2 C Code 65
A.9 Lines to enable Lisp to access C functions 66

List of Figures

2-1 Gust front created by a downdraft of air from a thunderstorm (adapted from [1]) 12
2-2 Gust front passing over runway 13
2-3 Thin line reflectivity and velocity convergence signature 14
2-4 Convergence signature associated with a gust front 15
2-5 History of gust front's movement 16
2-6 Occlusion caused by clouds in reflectivity image 17
2-7 Detection affected by gust front orientation in velocity image 18
2-8 Range sensitivity effects on gust front detection 18

3-1 Overview of the Machine Intelligent Gust Front Algorithm 22
3-2 Processed scan summary 23
3-3 Functional Template for thin-line feature detection 24
3-4 Interest Images 26

4-1 Old Interest Image Weighting Scheme: Each interest image is given a confirming and disconfirming weight for the entire image 28
4-2 New Interest Image Weighting Scheme: Each interest image has a corresponding pixel-weights image containing both confirming and disconfirming weights 28
4-3 Doppler Effect 29
4-4 Weather Reflectors 30
4-5 Oblique Motion 31
4-6 Front Orientation 32

4-7 Interest Confidence Orientation 32
4-8 Orientation Image 33
4-9 Compute Orientation Weights 34
4-10 Confirming and Disconfirming Weights 37
4-11 Sample Interest, Velocity, Orientation, and Orientation Weights . . . 38
4-12 Range sensitivity effects on gust front detection 39
4-13 Compute Range Weights 40
4-14 Sample Reflectivity, Interest, and Range Weights 41
4-15 Occlusion caused by clouds 42
4-16 Compute Occlusion Weights 44
4-17 Compute a combined pixel-level weights image 46
4-18 Graph of the squared average for two values 47
4-19 Compute combined-interest-image 48

5-1 MIGFA without pixel-level weights 52
5-2 MIGFA with pixel-level weights 52

List of Tables

5.1 MIGFA Performance without Pixel-level Weights 53
5.2 MIGFA Performance with Pixel-level Weights 53

Chapter 1

Introduction

1.1 Motivation

Two major concerns of all transportation systems are safety and efficiency. An endeavor is being made by the Federal Aviation Administration to improve both the safety and efficiency of air travel by providing air traffic controllers with tools to better detect and predict short term weather conditions around airports.

Machine vision techniques applied to weather radar data allow computers to play a major role in this through the automatic detection and tracking of important weather features. As with many real world systems, no one technique or algorithm is independently sufficient for the job, so many are employed. Combining information from each of these, data fusion, is a task that poses some interesting problems.

1.2 Contribution of Thesis

This thesis will describe and discuss a method of data fusion that employs information at the pixel level in the context of gust front detection. This pixel-level data fusion technique is incorporated into the Machine Intelligent Gust Front Algorithm (MIGFA). Prior algorithms for weather detection have used general image processing techniques devoid of object- and context- dependent knowledge, and applied thresholds at early stages of processing. MIGFA represents a new approach using several

intelligent (knowledge-based) feature detectors whose outputs are assimilated before thresholding.

This thesis demonstrates ways to better use human knowledge in the fusion of feature detector outputs. It will describe various factors that were considered and the ways these factors were incorporated into pixel-level data fusion for MIGFA working on data from the Terminal Doppler Weather Radar (TDWR).

1.3 Thesis Overview

This thesis consists of five chapters. Chapter two provides a physical description of gust fronts and discusses motivations for detecting and tracking them. Chapter three describes a method of gust front detection used by MIGFA. Chapter four describes a method of knowledge- and context-based pixel-level data fusion used to augment MIGFA. Chapter five contains a summary and discussion of the results, and recommends directions for future research.

Chapter 2

Gust Fronts

2.1 Physical Description

When a strong downdraft of air from a thunderstorm reaches the ground, it will tend to spread out horizontally as shown in Figure 2-1. The boundary where the spreading outflow of cool air meets a calm area or an opposing wind is know as a gust front. This boundary can typically grow to be several kilometers long and can propagate very long distances away from the generating storm.

2.2 Motivation for Detection

There are two main motivations for detecting and tracking gust fronts: gust fronts can be dangerous, and gust front prediction can help air traffic controllers plan. A reliable system that can detect and track gust fronts is of great value because it can improve both the efficiency and safety of air travel.

2.2.1 Dangers

An aircraft taking off or landing is very vulnerable to sudden changes in wind speed and direction. The turbulence associated with a gust front is such that it can pose a serious hazard to aircraft in these situations. Also, an unanticipated gust front can create delays as traffic is rerouted. These delays can increase the risk of human error

Figure 2-1: Gust front created by a downdraft of air from a thunderstorm (adapted from [1])

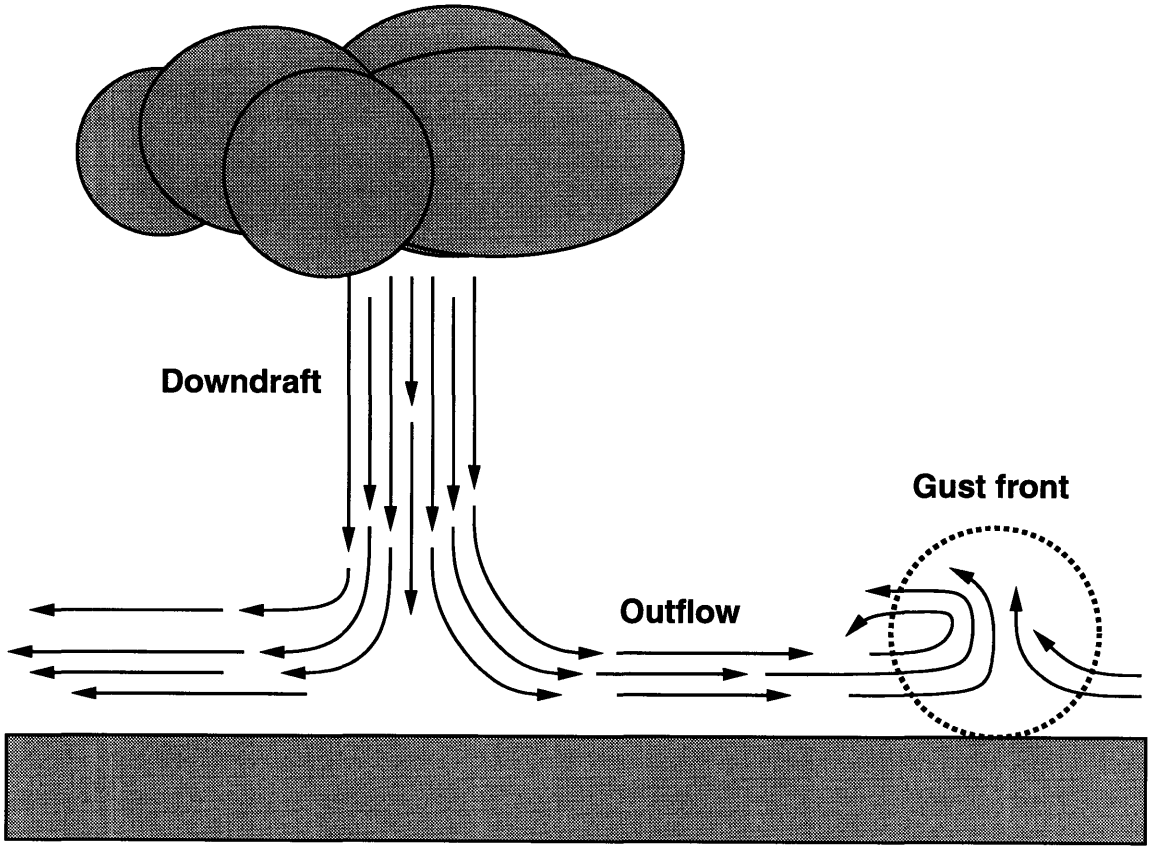
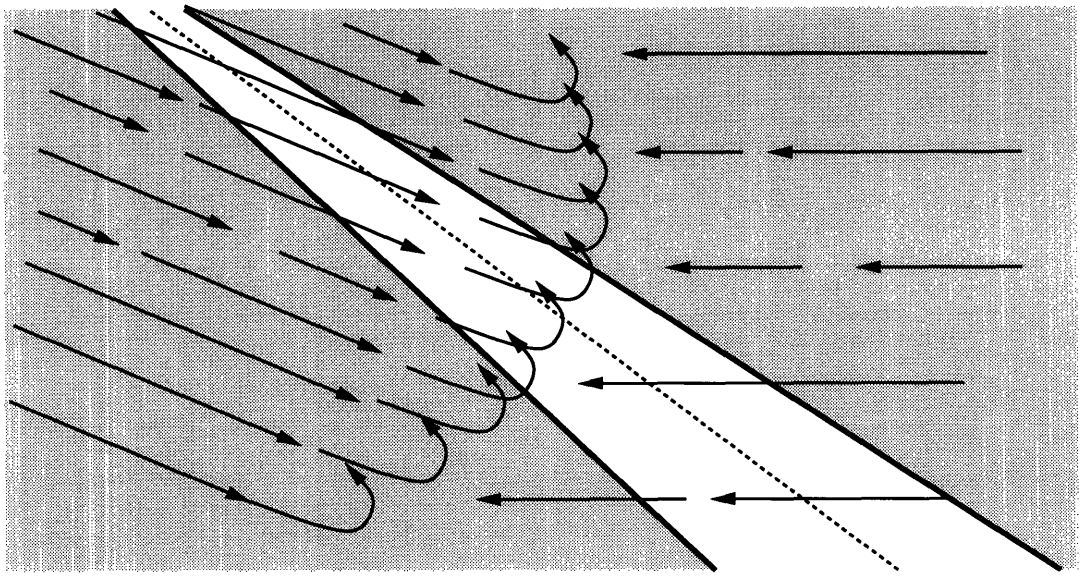


Figure 2-2: Gust front passing over runway



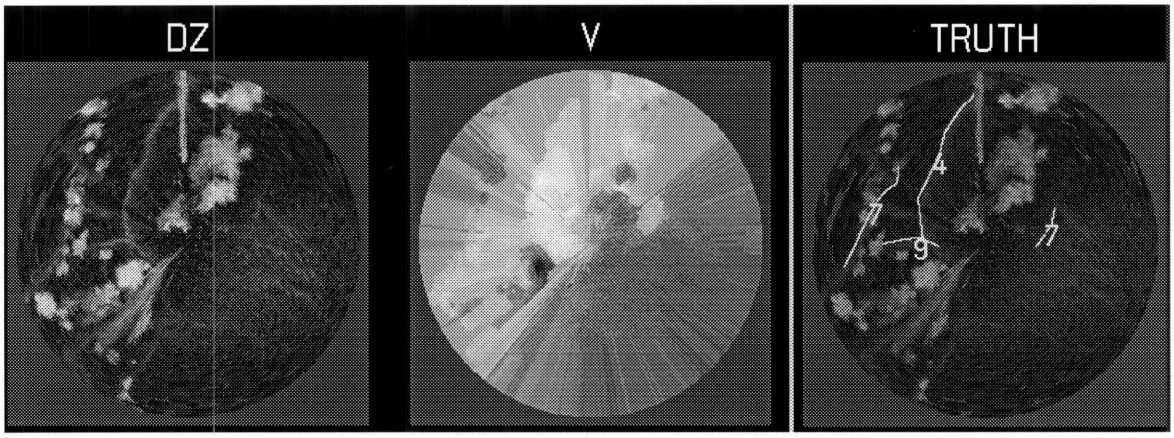
as air traffic controllers try to compensate by reducing the distance between aircraft taking off and landing.

2.2.2 Planning

Gust fronts present a boundary between regions with different wind velocity characteristics, and winds following a passing gust front tend to persist for long periods of time. Since runway planning must take into account wind direction and speed, the tracking of gust fronts can enable air traffic controllers to make short term wind condition predictions over specific runways, which in turn can help them streamline airport traffic. Figure 2-2 shows the passing of a gust front over a runway. Before the passage of the gust front, planes would have landed from left to right. After the passage of the gust front, planes will have to land from right to left.[1]

[1]Planes are typically scheduled to land into a headwind.

Figure 2-3: Thin line reflectivity and velocity convergence signature



2.3 Radar Signature

Three physical characteristics enable us to track gust fronts on radar, a thin line of increased reflectivity, a line of velocity convergence, and gust front motion.

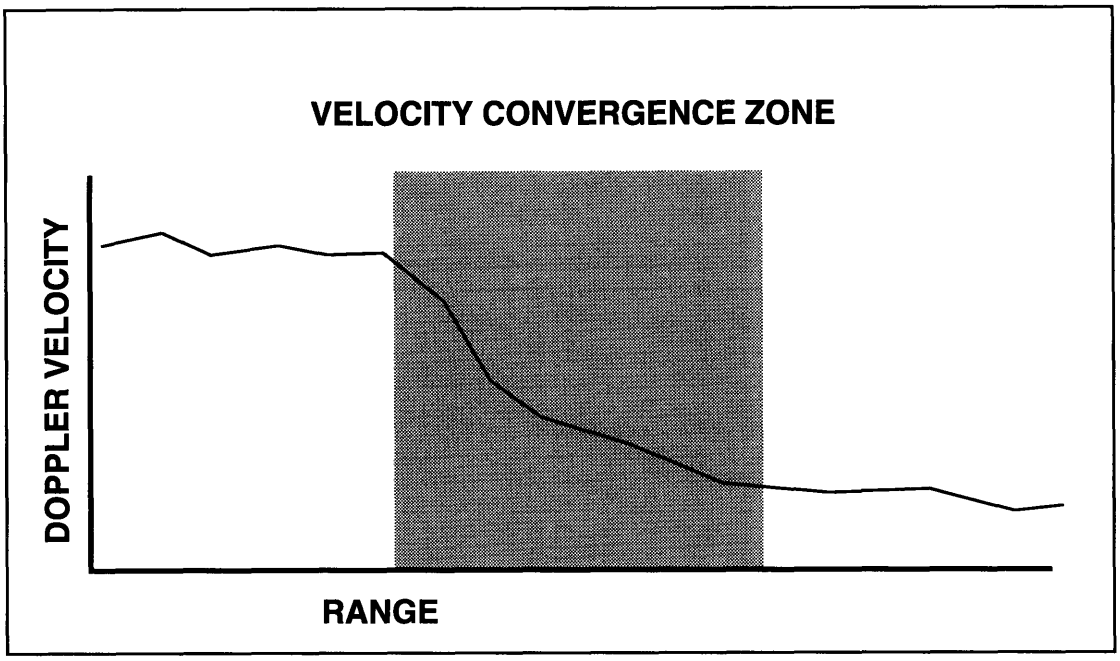
2.3.1 Reflectivity

A thin line of reflectivity is caused by concentrations of scatterers (possibly dust particles, insects, or water droplets) carried in the gust front itself. These thin lines vary in width but typically do not exceed 3 kilometers. Radar reflectivity is measured in dBZ, and gust front reflectivities typically exhibit values between 10 and 20 dBZ[1]. A gust front's thin line reflectivity signature can be seen in Figure 2-3 in the image labeled DZ. (The TRUTH image shows gust fronts that have been labeled by a human analyst.)

2.3.2 Velocity Convergence

A gust front marks the boundary where the cool outflow of air from a thunderstorm converges with ambient air or opposing winds (Figure 2-2). On Doppler radar velocity images, a sudden drop in velocity values is usually associated with the location of a

Figure 2-4: Convergence signature associated with a gust front

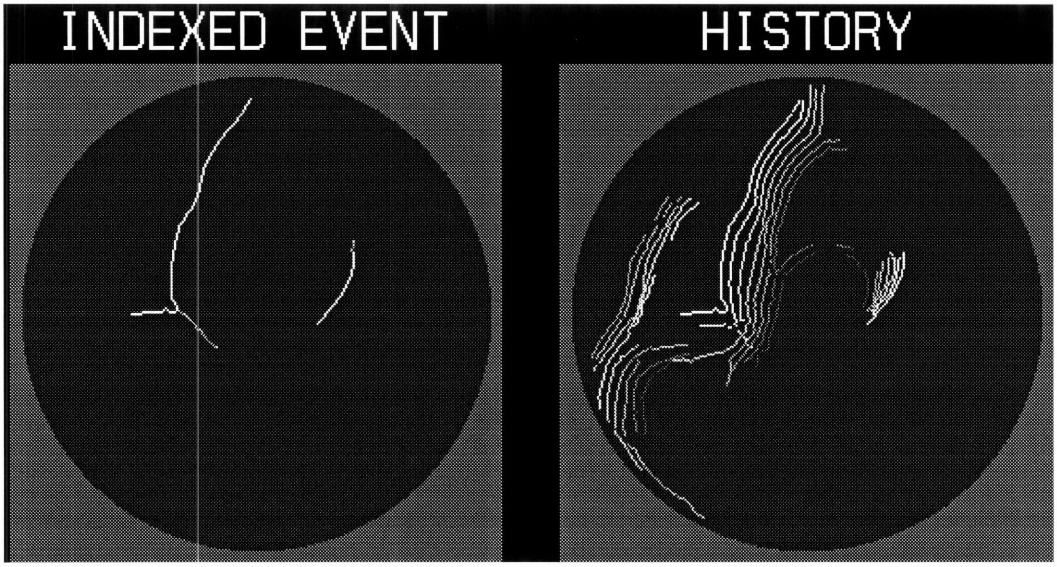


gust front. Figure 2-4 shows the convergence that would typically be seen along a single radial of the radar image. The convergence of an entire front can seen on a Doppler radar image like the one shown in Figure 2-3; the gust front labeled 4 in the TRUTH image displays clear velocity convergence in the V image. Algorithms before MIGFA relied almost exclusively on this characteristic[4].

2.3.3 Motion

Gust fronts move at a steady speed in a direction generally perpendicular to the orientation of the thin line and convergence boundary. A comparison of consecutive scans should reveal this motion. If a thin line and convergence boundary do not appear to move at all, the signature on radar probably does not belong to a gust front or at least to one that is of concern to air traffic controllers (unless it is experiencing a change in behavior caused by an event like a collision with another gust front). Figure 2-5 shows the history of a typical gust front's movement.

Figure 2-5: History of gust front's movement



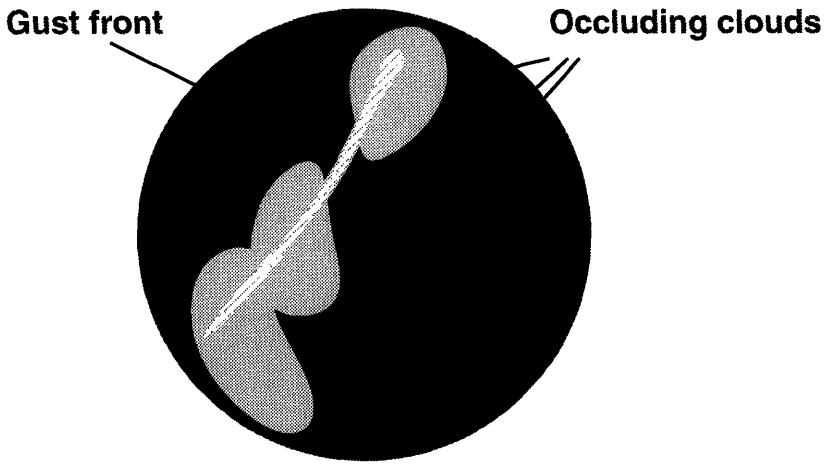
2.4 Challenges to Detection Algorithm

When considered individually, none of the three characteristics mentioned guarantees the existence of a gust front, because none are unique to gust fronts – nor do fronts necessarily show all of these characteristics. There are many factors that can cause us to receive weak, ambiguous, or even contradictory signals.

2.4.1 Occlusion

Cloud formations and heavy precipitation can prevent us from seeing a reflectivity thin line. The reflectivity of clouds and heavy rain is often comparable to or higher than that of gust fronts, and these can cause the partial or even complete occlusion of passing gust fronts (Figure 2-6). Static physical features like mountains can create residual ground clutter to hide gust fronts too. In Figure 2-3, the gust front labeled 7 on the left side is partially occluded by clouds.

Figure 2-6: Occlusion caused by clouds in reflectivity image



2.4.2 Orientation

Doppler radar only measures the component of the wind in a direction parallel to the radar beam. Gust fronts oriented perpendicular to the radar beam can provide strong Doppler signatures, while gust fronts oriented parallel to the radar beam may not be seen at all in velocity images. Figure 2-7 depicts both of these situations. In Figure 2-3, the gust front labeled 9 is oriented parallel to the radar beam and does not show up clearly in the velocity image.

2.4.3 Sensitivity

A radar's sensitivity varies over range. Very close to the radar, we may have trouble with things like ground clutter. The sensitivity of a radar is at a maximum at a small distance away from the radar, and then decreases over distance. Because of this variation in sensitivity, a gust front with a certain reflectivity signature may be visible because it is close to the radar, while a gust front with the same reflectivity may be invisible because it is at a distance away from the radar where the minimum detectable signal is greater than the gust front's signal (Figure 2-8).

Figure 2-7: Detection affected by gust front orientation in velocity image

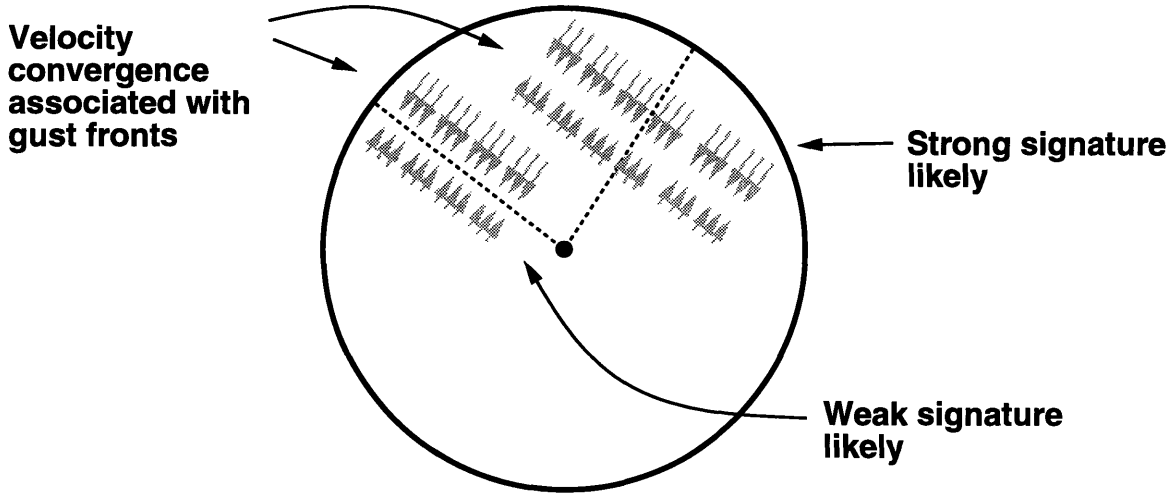
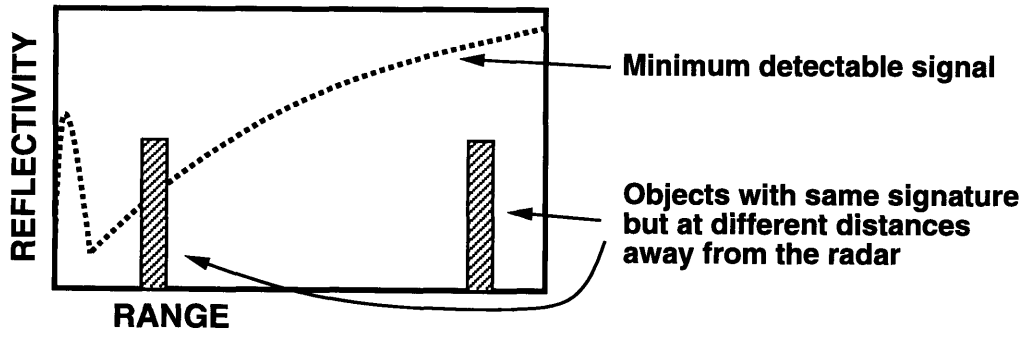


Figure 2-8: Range sensitivity effects on gust front detection



2.4.4 Imitation

Other things can look like gust fronts on radar. For example, vertical shears, often present in severe thunderstorms, can bias low-altitude velocity estimates, producing apparent convergence signatures[1]. Flocks of birds, clouds of dust produced by construction sites, elongated low-intensity precipitation echoes, and ground clutter, can produce thin line signatures–some of these can also exhibit gust front-like motion[1].

Chapter 3

Gust Front Detection

Ideally, we want a system that will detect all gust fronts in all conditions with no false alarms. The ideal is extremely difficult to achieve, even for human analysts, because of the physical characteristics of gust fronts and radars. Humans do as well as they do because they make use of knowledge about the sensor, knowledge about the behavior of gust fronts, and knowledge of weather conditions and how they affect the likelihood of gust fronts appearing on radar. Humans can assimilate weak, ambiguous, or even contradictory evidence.

The following are a few more tractable goals.

1. Detect clear obvious cases with very high reliability.
2. Generate very few false alarms.

Improved performance of the Machine Intelligent Gust Front Algorithm (MIGFA) over prior algorithms is due in large part to its ability to use human knowledge in detecting gust fronts.

3.1 Design philosophy behind MIGFA

The conventional wisdom in computer vision/object recognition research has been to use general image processing operations, ideally devoid of object- and context-dependent knowledge, at the initial states of processing[1]. Such operations might include edge detection, segmentation, cleaning, and motion analysis.

Image characteristics are then extracted from these general operations and represented symbolically. Machine intelligence is then applied on the symbolic representations at the higher levels of processing[1].

In contrast, sensor-, object- and context-dependent knowledge is applied in the earliest (image-processing) levels of MIGFA processing. Knowledge of the problem is used in three ways. First, knowledge is used to select from a library, those feature detectors that are selectively indicative of the object being sought. Second, knowledge is also incorporated within feature detectors through the design of matched filters that are customized to the physical properties of the sensor, the environment, and the object being sought. Third, knowledge about the varying reliability of the selected feature detectors is used to guide data fusion[1].

3.2 Overview of Machine Intelligent Gust Front Algorithm

The system diagram in Figure 3-1 provides an overview of MIGFA. In preparation for processing, input images DZ (reflectivity image) and V (Doppler velocity image) from the current radar scan are converted from polar to Cartesian representation and scaled to a useful resolution.

These images are passed to several independent feature detectors that attempt to localize features that are indicative of gust fronts. The outputs of these feature detectors, most based on some application of Functional Template Correlation[3], are expressed as interest images that specify evidence for where and with what confidence a gust front may be present. These interest images are fused to form a combined interest image, providing an overall map of evidence for where gust fronts are believed to exist[1].

From the combined interest image, fronts are extracted and integrated with prior history. The updated history is then used to predict where the gust fronts might be several minutes in the future. These predictions are used in processing subsequent images through a feature detector called ANTICIPATION, which selectively sensitizes

Figure 3-1: Overview of the Machine Intelligent Gust Front Algorithm

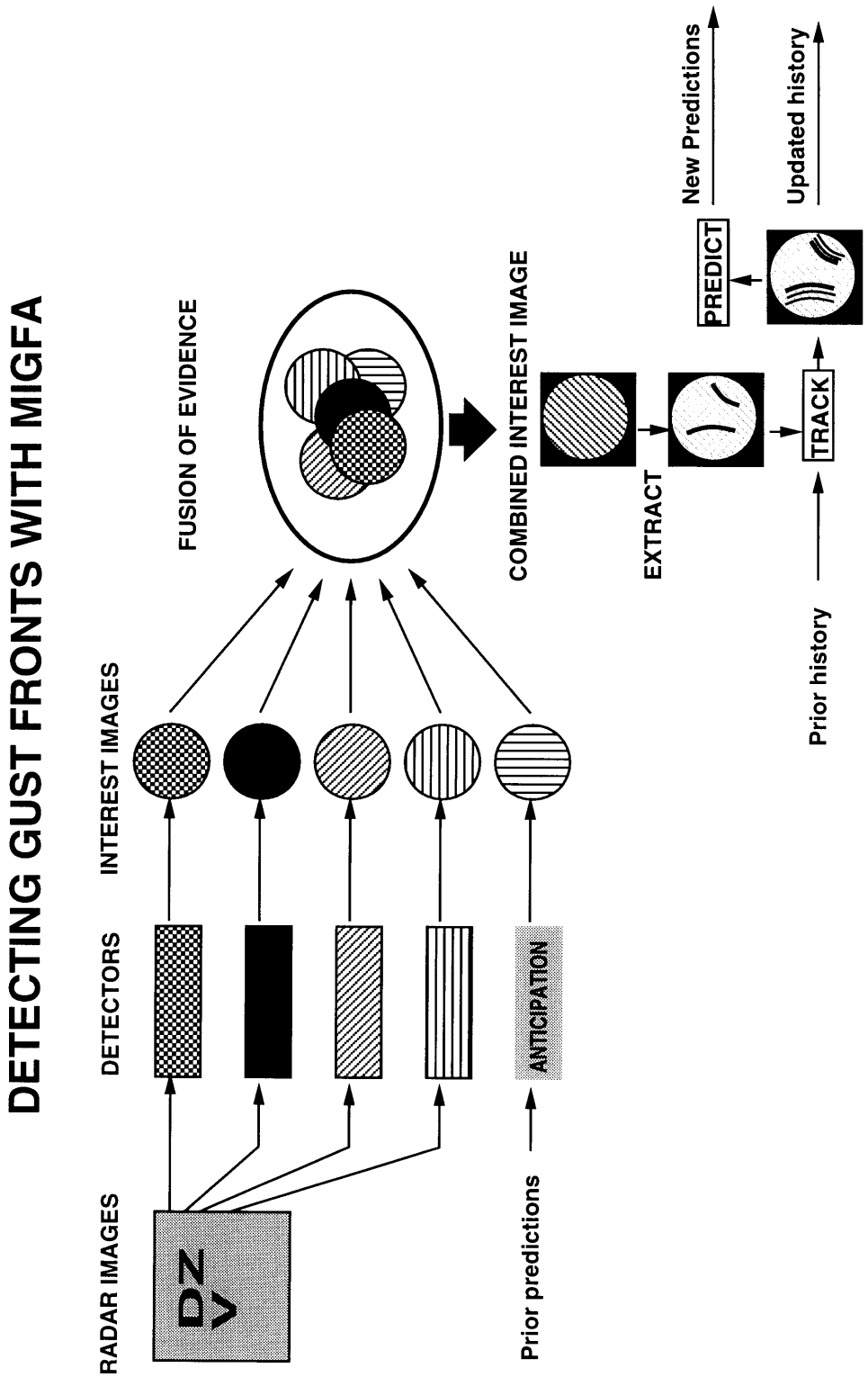
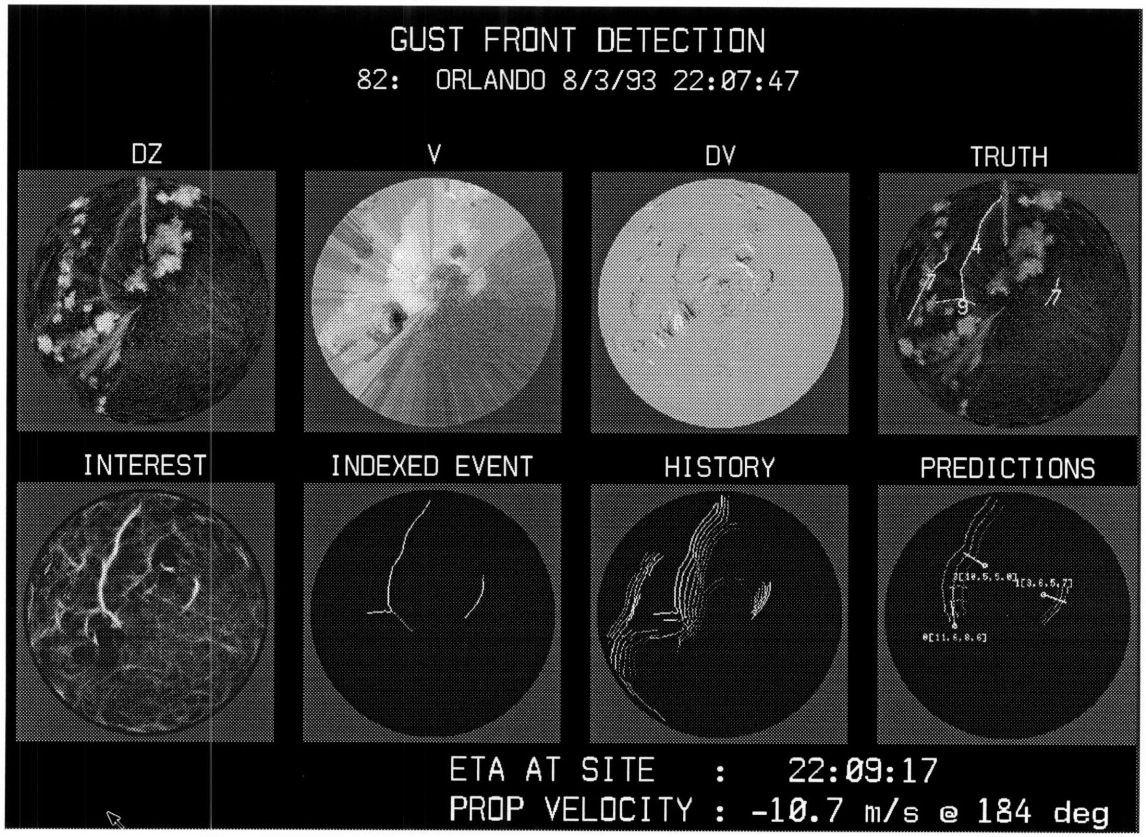


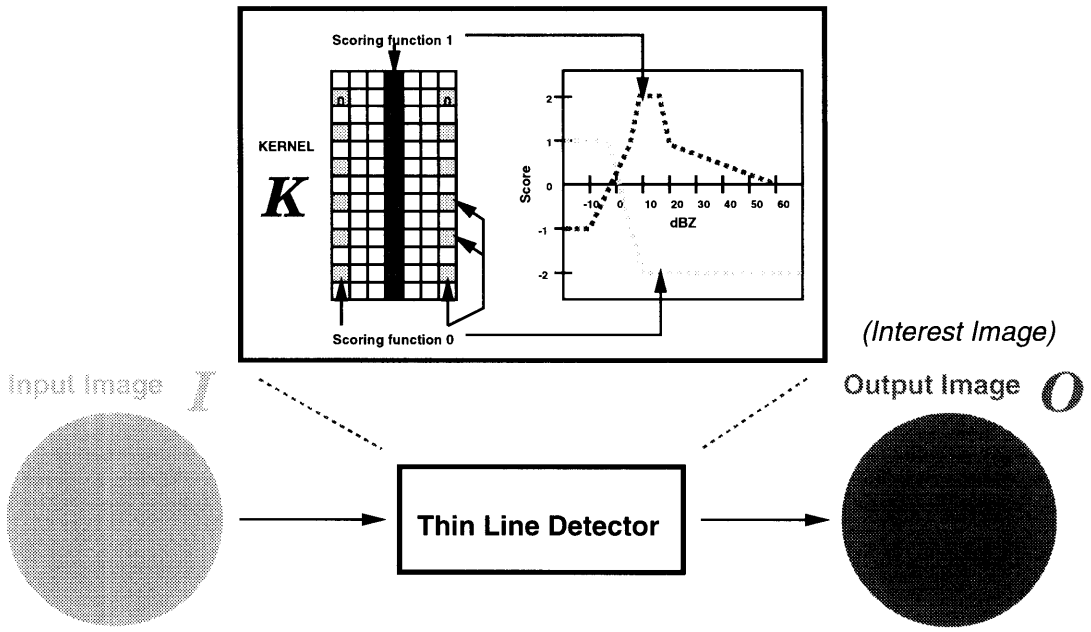
Figure 3-2: Processed scan summary



the system to detecting gust fronts at specific locations.

Figure 3-2 is a summary of the processing steps for a single example. The images correspond to the steps outlined in Figure 3-1. INTEREST shows the combined interest image, INDEXED EVENT displays the extracted gust front, HISTORY displays the updated history, and PREDICTIONS displays MIGFA's predictions.

Figure 3-3: Functional Template for thin-line feature detection



3.3 Feature Detection, Functional Template Correlation, and Interest Images

Most of the feature detectors used in MIGFA are based on some application of Functional Template Correlation (FTC). (William Freeman discusses some similar ideas in his thesis, "Steerable filters and local analysis of image structure"[5].) FTC is described in detail by Richard Delanoy in [3], but can be briefly explained through the study of an example.

Given some input image I, the output image O is generated through the use of a given functional template. Figure 3-3 shows the functional template for thin line detection. A functional template consists of a kernel K and some number of scoring functions. Each pixel of the kernel corresponds to one of the scoring functions, or is considered nil and does not participate in the computation of the score.

The kernel K is placed over a region of I (with the center of the region being

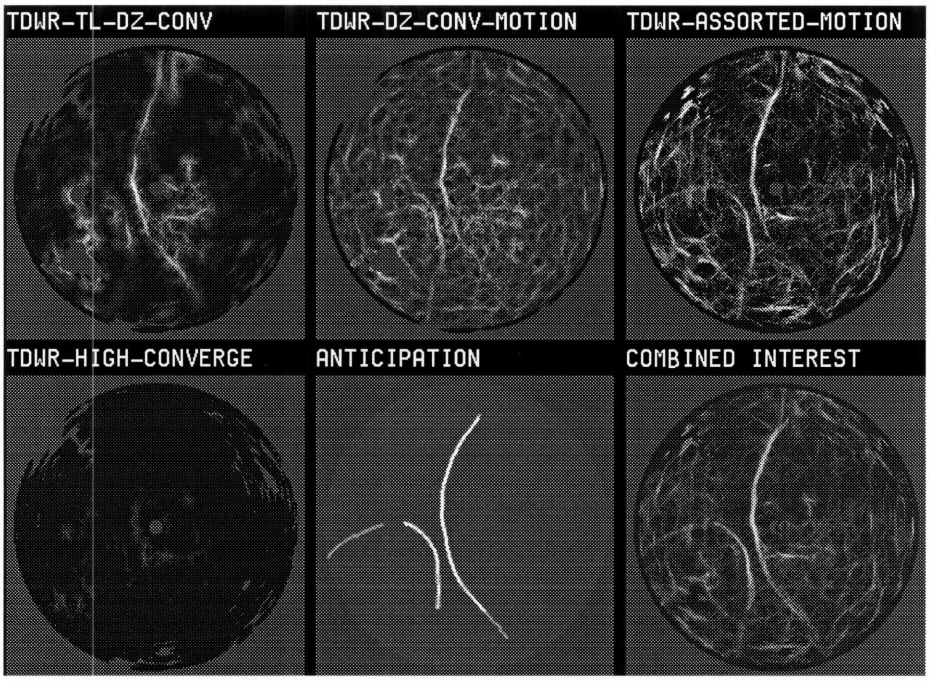
I(x,y)) to compute a score for the pixel *O(x,y)* of the output image *O*. If the shape of the object being matched can vary in orientation in *I*, then *K* is placed over I at several orientations. To compute the score, each pixel of *K* uses the value of the pixel directly beneath it in *I*, as an argument to the scoring function associated with that particular pixel of *K*. The average of all the values generated by the pixels of *K* is the score for *O(x,y)*. If many orientations are being probed, then the score assigned to *O(x,y)* is the maximum across all orientations. *K* is moved across *I* to generate scores for all the pixels of *O*.

The functional template in Figure 3-3 reflects what we know about the physical characteristics of gust fronts. Gust fronts typically exhibit a thin line of reflectivity between 10 and 20dBZ, and hence the functional template will contribute high scores for a string of pixels in *I* with values between 10 and 20dBZ through scoring function 1. Surrounding the thin line, should be areas of low reflectivity, so scoring function 0 will contribute high scores for pixels with reflectivity values less than 0dBZ. The unspecified locations of *K* provide error bands to reflect the fact that gust fronts can vary somewhat in width.

The output image generated by functional template correlation can be used as an interest image. In an interest image, each pixel has a score which reflects the detector's belief in the existence of a particular object over that pixel. This dimensionless score provides a common domain in which to combine outputs from several independently functioning detectors[2].

Figure 3-4 shows some of the interest images generated by MIGFA operating on TDWR. Interest image TDWR-TL-DZ-CONV is generated by a feature detector that couples a DZ thinline functional template with a convergence functional template scanning the velocity image. TDWR-DZ-CONV-MOTION is generated by a feature detector that couples a convergence functional template with a motion detector. TDWR-ASSORTED-MOTION is the output of several motion detectors. TDWR-HIGH-CONVERGE is the interest image generated by a detector looking for very high regions of convergence in the velocity image.

Figure 3-4: Interest Images



Chapter 4

Data Fusion

MIGFA's approach to gust front detection seems to be better than previous approaches. The introduction of more knowledge to the detection stage of processing seems to have improved overall performance[1]. No one feature detector is meant to be a perfect, or even necessarily a good, discriminator of gust fronts and background. But, when used together, several weakly discriminating feature detectors can achieve robust performance depending on how the outputs are combined. Another improvement in performance is expected with the introduction of more knowledge to the method of combination.

Currently, each detector's interest image is assigned one confirming and one disconfirming weight to reflect a level of confidence in the entire interest image (Figure 4-1). But, it is not the case that all the pixels of a given interest image are equally reliable, because the reliability of different regions of evidence on the same interest image can vary for several reasons as described in Chapter 2.

4.1 Proposed method

A better approach would be to provide a finer grade assignment of weights to smaller regions in the interest image. In the limit, we could assign each pixel a weight reflecting our level of confidence in the evidence provided by that particular pixel (Figure 4-2). This is in fact the approach taken, because it provides the finest possible

Figure 4-1: Old Interest Image Weighting Scheme: Each interest image is given a confirming and disconfirming weight for the entire image

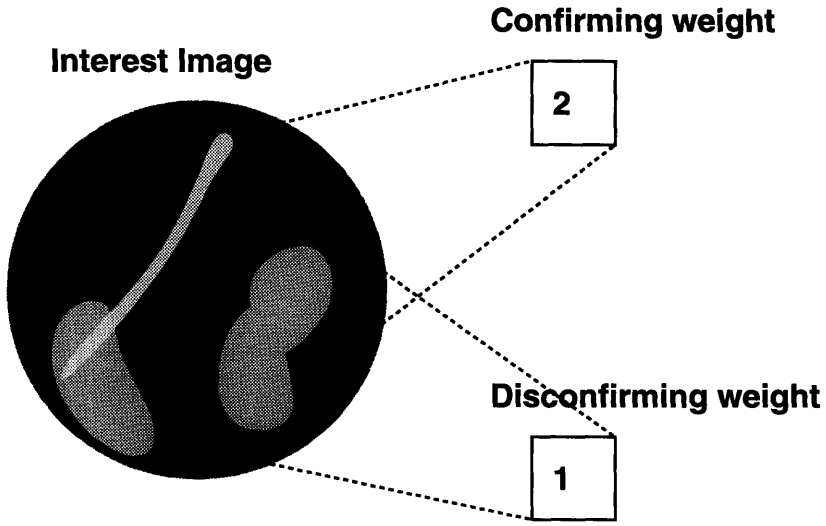


Figure 4-2: New Interest Image Weighting Scheme: Each interest image has a corresponding pixel-weights image containing both confirming and disconfirming weights

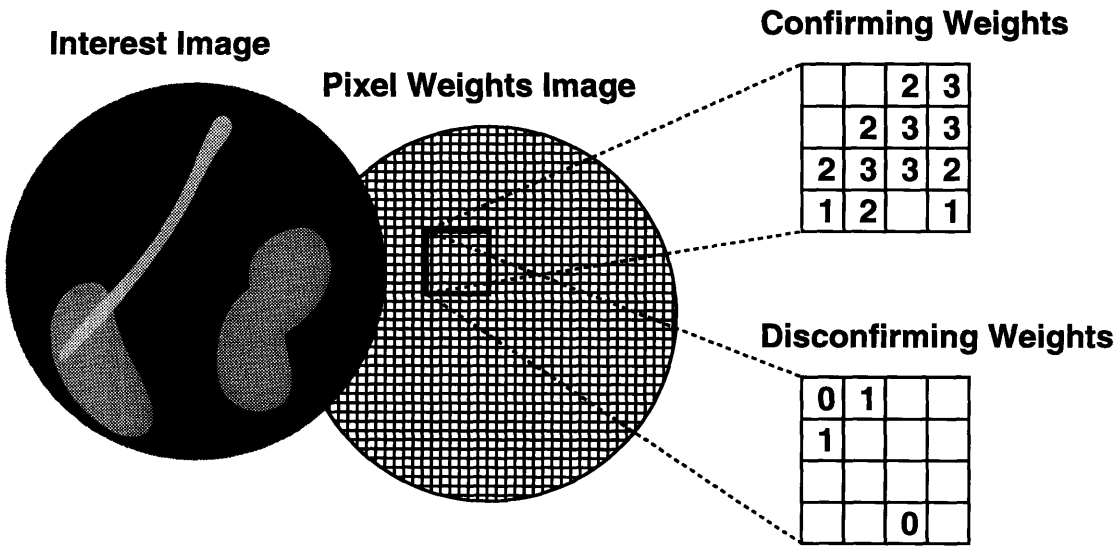
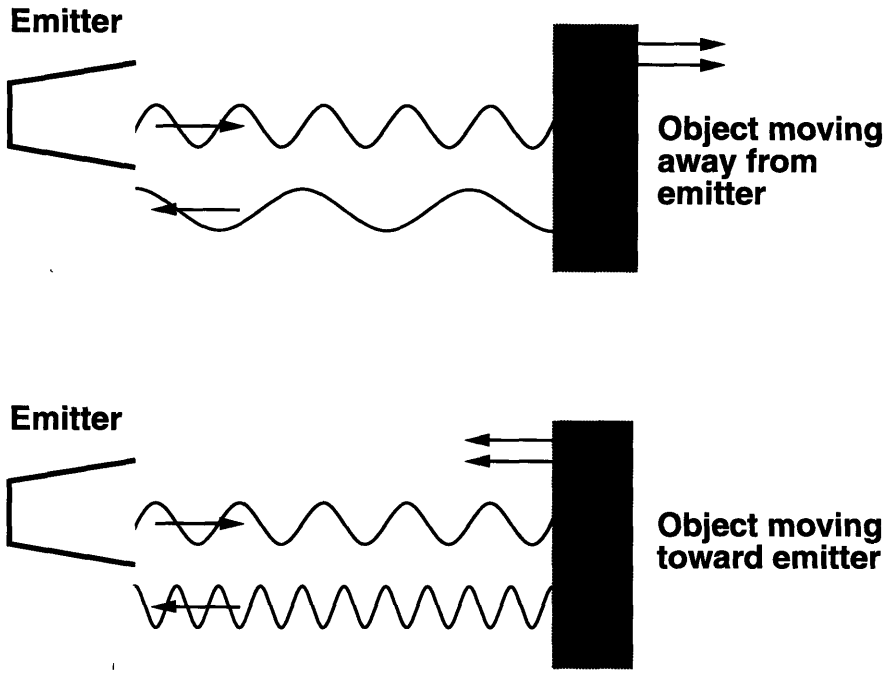


Figure 4-3: Doppler Effect



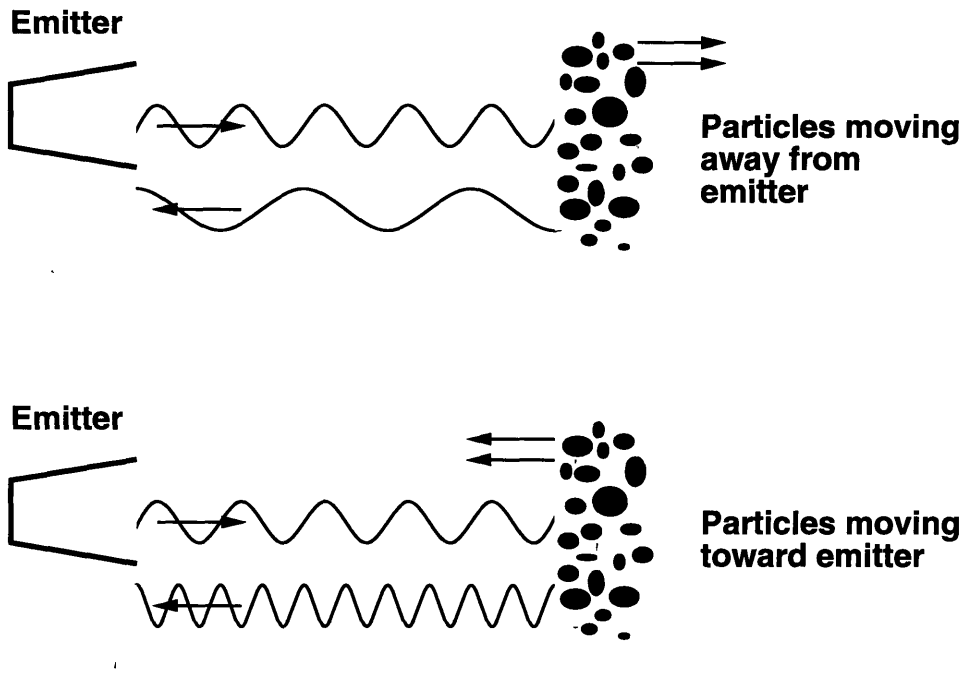
grain map of reliability, and eliminates the need for mechanisms for dividing up large interest images into smaller weightable regions.

What follows in this chapter is a discussion of some factors influencing an interest image region's reliability, an algorithm for incorporating this knowledge into our processing through the generation of pixel-level weights, and a description of how this pixel-level weight computation is actually realized (in a Lisp- and C-based computer vision development environment called SKETCH). This chapter will conclude with a section describing how the newly computed pixel-level weights are used to fuse the various interest images into one final combined interest image.

4.2 Orientation

The Terminal Doppler Weather Radar (TDWR) uses the well known Doppler effect. Simply explained, when a radar beam of a given frequency f is aimed at an object moving away from the emitter, the reflected frequency will be less than f (Figure 4-3).

Figure 4-4: Weather Reflectors

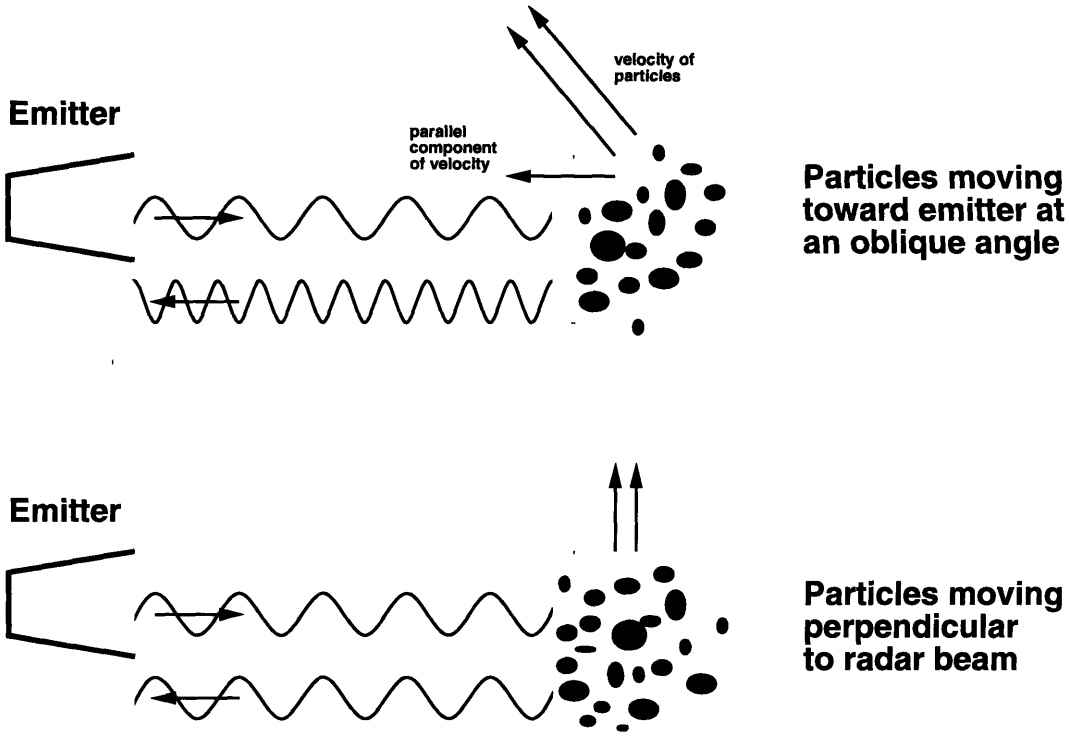


On the other hand, when a radar beam of a given frequency f is aimed at an object moving toward the emitter, the reflected frequency will be greater than f (Figure 4-3).

In weather applications, water droplets, dust particles, and sometimes insects, provide us with our reflecting objects. Water droplets from a cloud moving slowly away from a radar will return a signal that is equal to or slightly less than f (Figure 4-4). Particles caught in a rapidly approaching gust front will return a signal that is usually greater than f (Figure 4-4).

A problem arises when we consider particles moving at some oblique angle (Figure 4-5). Then, the signal received near the emitter will give us only a measure of the component of the velocity parallel to the radar beam. In the worst case, particles moving perpendicular to the radar beam will not be detectable at all through the observance of any change in reflected beam's frequency (Figure 4-5).

Figure 4-5: Oblique Motion



4.2.1 Orientation Considerations for Gust Fronts

Gust fronts move in a direction perpendicular to the thin line's orientation. Therefore, a gust front oriented perpendicular to a radar beam can be clearly seen on a Doppler radar scan (and consequently will be clearly highlighted in one of MIGFA's interest images) (Figure 4-6). On the other hand, a gust front oriented parallel to a radar beam will be very difficult to see on a Doppler radar scan (also Figure 4-6).

As a result, if a line of pixels oriented perpendicular to a radar beam is highlighted in one of the interest images, we can be very confident that it is from a gust front. On the other hand, if a line of pixels oriented parallel to a radar beam is highlighted in one of the interest images generated by a detector working with Doppler velocity values, we may be skeptical of whether or not that line was created by a gust front because we know that gust fronts oriented in this manner are very hard to see in Doppler images. Figure 4-7 depicts these two situations.

Figure 4-6: Front Orientation

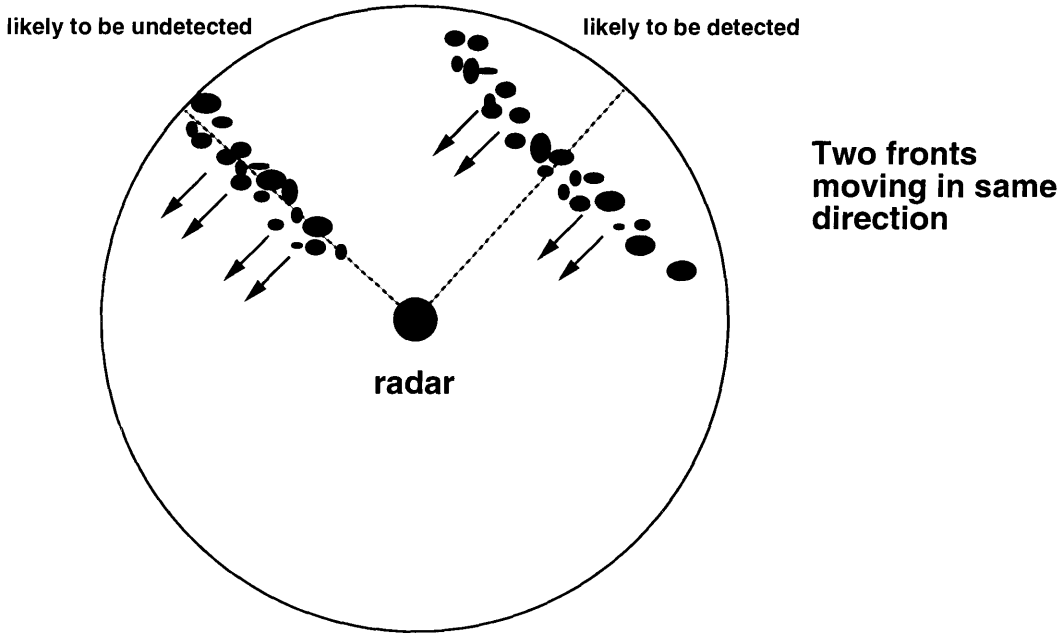


Figure 4-7: Interest Confidence Orientation

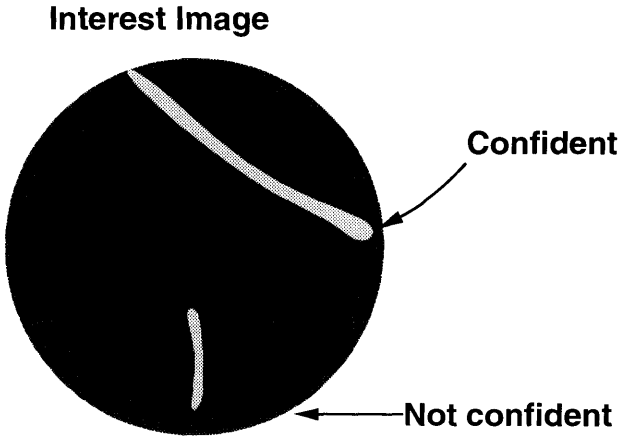
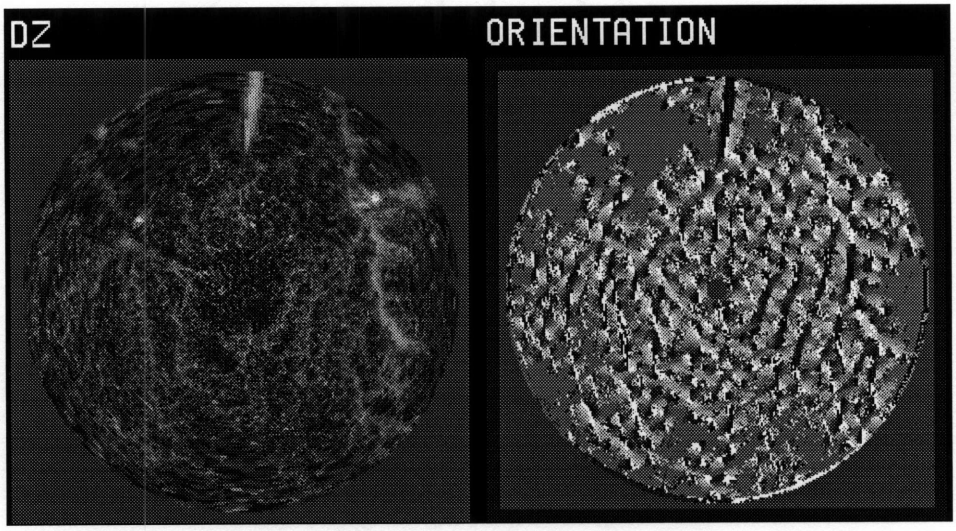


Figure 4-8: Orientation Image



4.2.2 Computing Weights

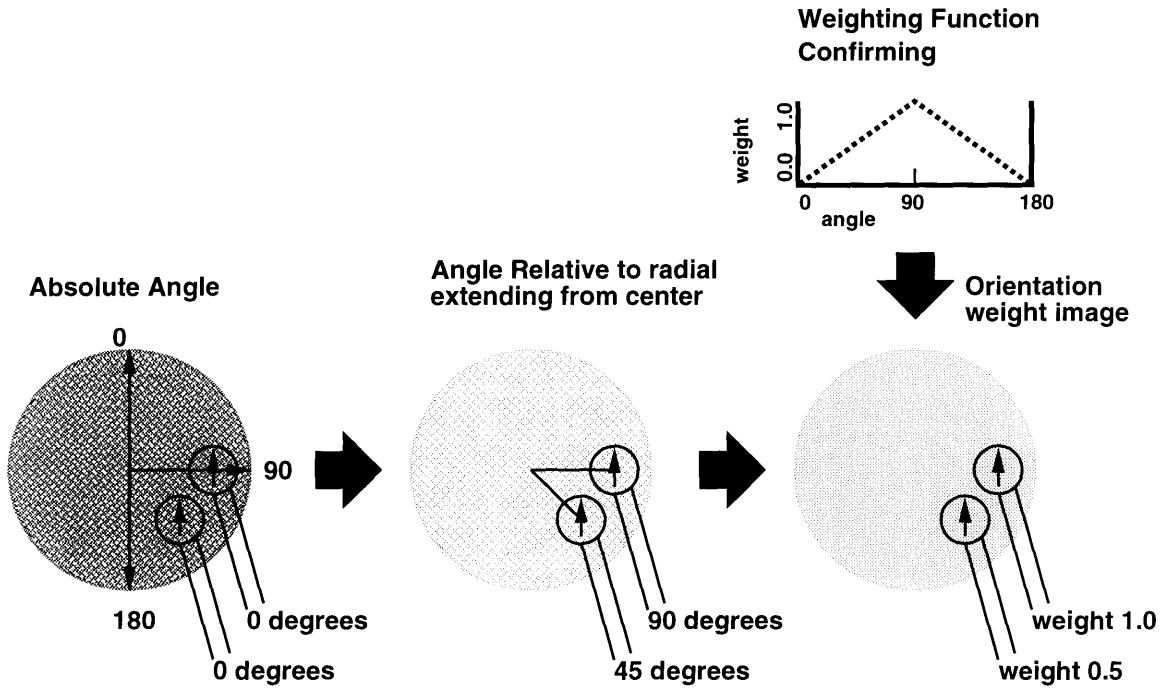
MIGFA feature detectors using Functional Template Correlation will record an orientation value for each pixel, which is an angle measure in degrees, of the orientation at which the functional template provided the best fit (Figure 4-8). This orientation value is used to compute an **orientation-weight** for each pixel of the interest image.

The diagram in Figure 4-9 outlines the algorithm used to compute orientation-weights from the orientation map provided by the feature detector. The orientation map uses absolute angle values where a vertical arrow pointing upward would designate zero degrees and a horizontal arrow pointing rightward would designate 90 degrees. This absolute orientation map is converted to a map of orientation values relative to the radials extending from the center.

In Figure 4-9, two vertically oriented pixels (orientation zero degrees) are given orientation values of 45 and 90 degrees in the relative orientation map.

A weighting function is then used to determine a weight for each pixel based on its relative orientation. A relative orientation of 90 degrees would result in a high weight, while relative orientations of zero or 180 degrees would result in a low weight. The assignment of weights for the other angles between 0 and 180 degrees, can be

Figure 4-9: Compute Orientation Weights



adjusted to conform to any realizable function that is determined empirically. A linear relationship is shown in the figure.

4.2.3 Augment with confirming and disconfirming weights

The issues described above are relevant when trying to deal with pixels offering confirming evidence. But, a separate weighting scheme is required to deal with pixels offering disconfirming evidence.[1]

A flatter weight function for disconfirming evidence

When an area of clear air is analyzed by a feature detector, the interest image will return low, disconfirming values everywhere, and the orientation with the best fit becomes a rather meaningless statistic. A clear area's orientation value could just as easily be zero or 90 degrees. For this reason, we would like disconfirming evidence (as indicated by low interest image values) to not depend too heavily on the orientation angle. So our disconfirming weight function should be much flatter than our confirming weight function.

Lower weights for disconfirming evidence in general

Because gust fronts with the right orientation can often pass unnoticed by Doppler velocity-based detectors, we should not rely on them too heavily for disconfirming evidence. Disconfirming pixels from velocity-based interest-images are thus in general given a lower weight than confirming pixels.

Slightly greater weight for perpendicularly oriented disconfirming evidence

A moderate to low interest-value (but a value still considered disconfirming) oriented perpendicular to a radial could be an accurate reading of some existing low wind shear. This is more probable than an accurate reading of some low wind shear oriented

[1] On a scale between 0.0 and 1.0, final interest values greater than 0.5 are considered confirming, while the the values below are considered disconfirming.

parallel to a radial. Hence, we would still like to give stronger weight (though it be disconfirming) to pixels oriented perpendicular to a radial.

Figure 4-10 shows an augmented orientation weighting scheme which references the interest image to choose either the confirming or disconfirming weighting function. The disconfirming weighting function is much flatter than the confirming function, has lower values, but still indicates more confidence in the pixels oriented 90 degrees relative to a radial from the center.

Both confirming and disconfirming weights are stored on the same orientation weight image. There is no need for separate confirming and disconfirming weight images because each pixel of the interest-image can be only one or the other, and would use the weight stored in the corresponding location of the weighting image.

Figure 4-11 shows a Doppler velocity scan and its corresponding interest, orientation, and weight images.

4.2.4 Implementation

Appendix A lists the code used to implement the orientation weight computation. Portions requiring high speed computation have been written in C, while the rest of the functions are implemented in Lisp.

Weighting functions are implemented as tables created by the function `build-function-table`. Representing the functions in a table transforms the computation of weight to a simple table lookup, thereby reducing run-time computation requirements. This also enables the user to easily customize the weighting function in a very direct and intuitive manner, without having to resort to a cumbersome formulaic representation.

An **angle-index** is built to facilitate the conversion from absolute orientation to relative orientation. The angle-index is built as an image with each pixel containing a number indicating the absolute angle between a radial going through that pixel and a radial pointing straight up. This early pre-computation allows the later conversion from absolute to relative, to be achieved through a table lookup and a subtraction. Again we enjoy an increase in run-time performance over an explicit formulaic com-

Figure 4-10: Confirming and Disconfirming Weights

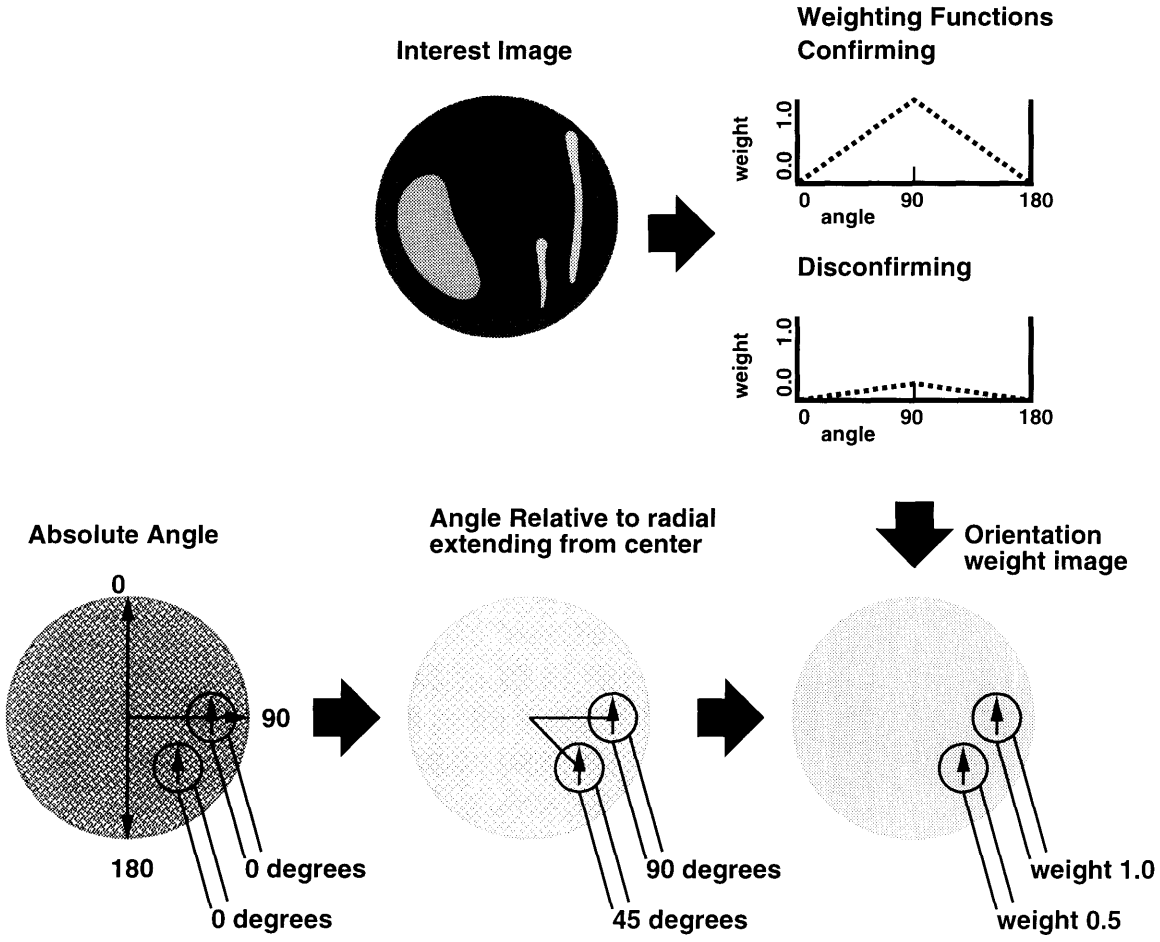
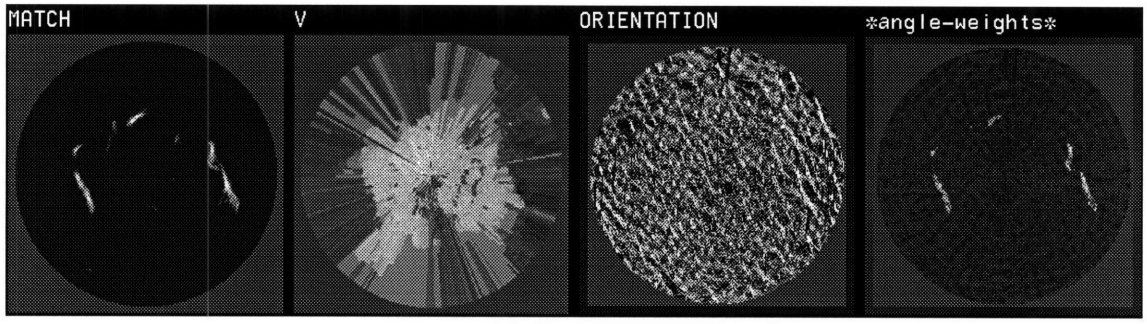


Figure 4-11: Sample Interest, Velocity, Orientation, and Orientation Weights



putation. The tables and index are stored in a list and assigned to the global variable ***angle-template***.

Currently, there are two functions for creating orientation weights. A function for reflectivity-based interest-images does not incorporate the sharp angular dependencies that Doppler-velocity-based interest-images do.

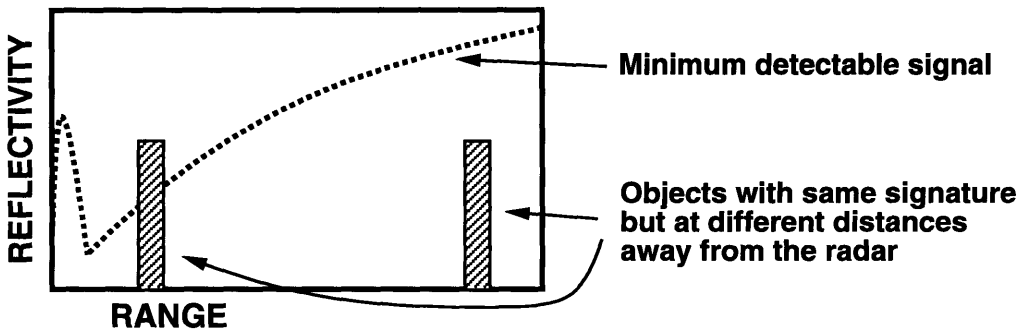
Although the system diagram seems to suggest that each step is completed on an entire image before the next step begins, the actual C function **weight_angle** completes the performance of all three steps one pixel at a time as it moves along the rows and columns of the original orientation-image, computes the final orientation weight for the current pixel, and then stores the result in an orientation-weight-image. This scheme takes advantage of many run-time computation optimizations.

4.3 Range

4.3.1 Range Considerations for Gust Fronts

A radar's sensitivity varies over range. Figure 4-12 is a graph of the minimum detectable signal as a function of distance away from the radar. As explained earlier,

Figure 4-12: Range sensitivity effects on gust front detection



a gust front that is close is very likely to be seen by the radar. Something that is very close is less likely to be seen clearly because of ground clutter, while events that occur very far from the radar are harder to see because the radar signal degrades over distance.

4.3.2 Computing Weights

The diagram in Figure 4-13 outlines the algorithm used to compute range weights from an interest image. For each pixel, the range is found by looking at a range index image, which has precomputed distance from the center values. Then, by looking at the interest image being weighted, we choose the confirming or disconfirming weight function. The pixel's range weight is indexed, and then stored in the range-weight image.

As we think about the two weighting functions, we realize that very close to the radar, both confirming and disconfirming evidence can be viewed with only moderate confidence because of ground clutter. A little further away, both confirming and disconfirming evidence can be trusted with very high confidence. This is reflected in Figure 4-13 with both confirming and disconfirming functions increasing. After a certain point, the sensitivity of the radar begins to gradually drop. This again is reflected in the gradual decrease of both the confirming and disconfirming functions.

At the furthest range values, the weight given to confirming evidence is higher

Figure 4-13: Compute Range Weights

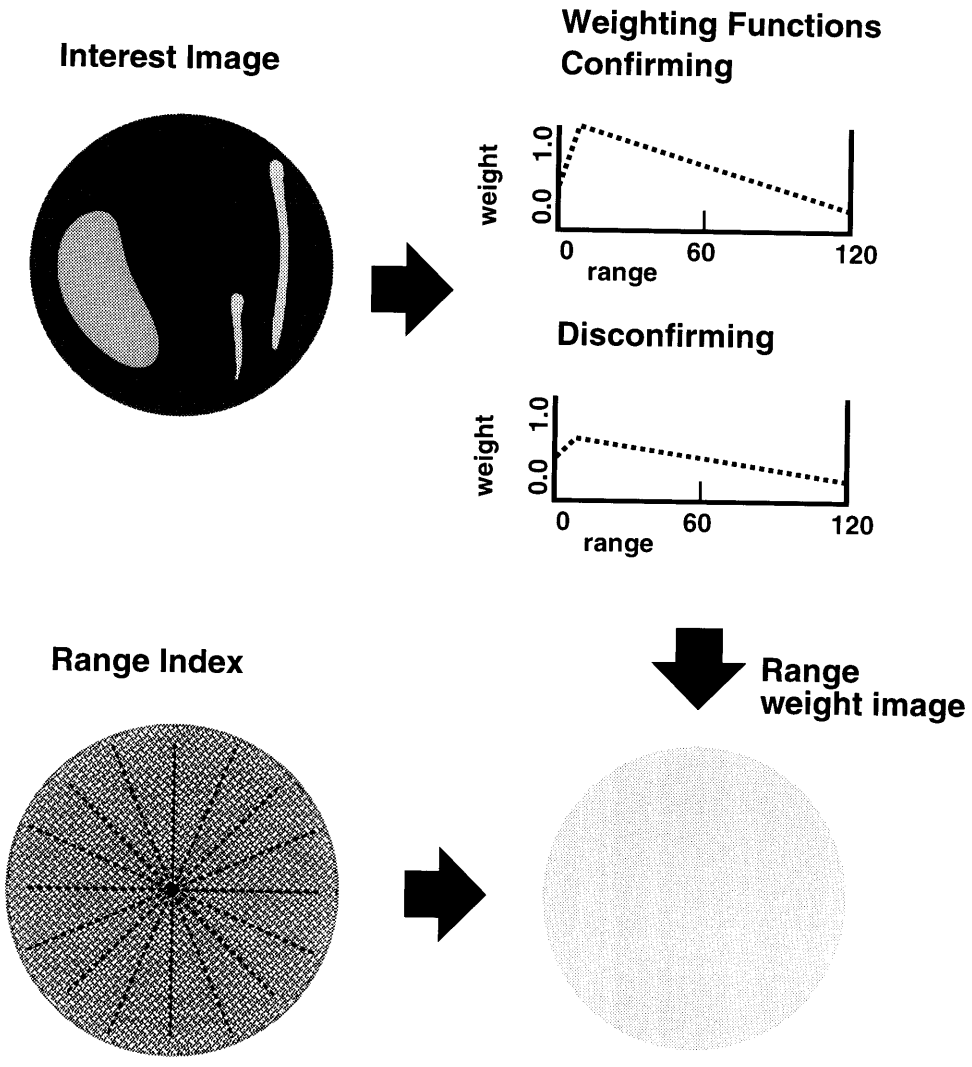
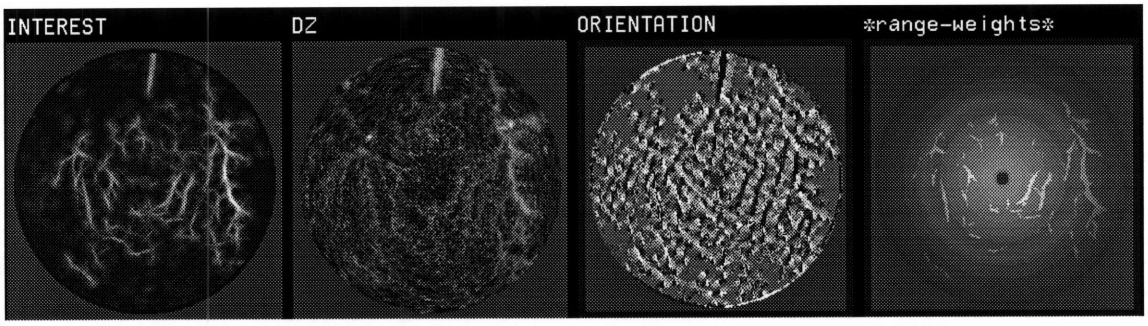


Figure 4-14: Sample Reflectivity, Interest, and Range Weights



than the weight given to disconfirming evidence, because the detection of a strong gust front far away from the radar is still very likely. On the other hand, disconfirming evidence far away cannot be trusted with as much confidence because a weak gust front could be present there while avoiding detection because of the radar's reduced sensitivity.

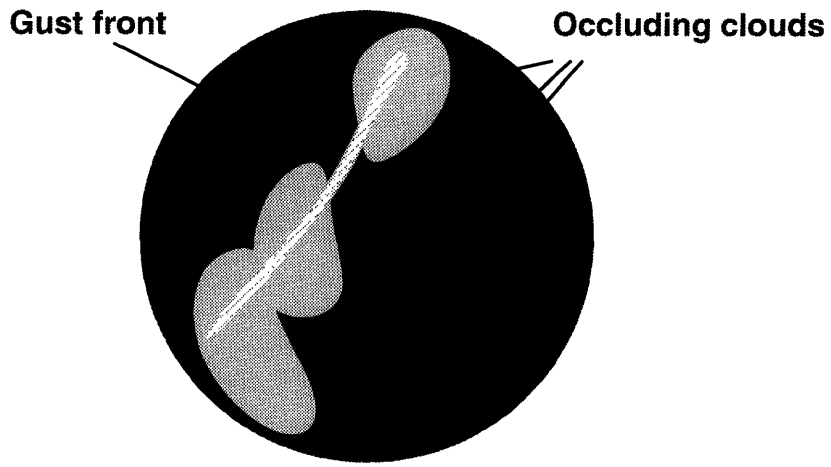
At moderate ranges, weight given to confirming evidence is higher than the weight given to disconfirming evidence in order to bias MIGFA toward greater sensitivity. A gust front that is seen by only a few detectors will have the positive evidence counted more heavily than negative evidence thereby increasing the chances of detection. (Note: This is just a current preference that can be changed later.)

Figure 4-14 shows a reflectivity scan and its corresponding interest-image and range weights images. Once again, we note that both confirming and disconfirming weights are stored in same range weight image.

4.3.3 Implementation

Appendix A lists the code used to implement the range weight computation. A **range-table** is created for confirming and disconfirming weight functions using the tool

Figure 4-15: Occlusion caused by clouds



`build-function-table`. A range-index is built simply as an image with each pixel containing a number indicating how far away that pixel is from the center, where the radar is supposed to be (this too reduces run time computational requirements). Both of these are stored in a list and assigned to the global variable `*range-template*`.

The actual range weighting is performed by the C function `weight_range`. Just like the orientation weighting function, the entire computation is completed one pixel at a time. The program moves along the rows and columns of the original orientation-image, computes the final orientation weight for the current pixel (whether it be confirming or disconfirming), and then stores the result in a range-weight image.

4.4 Occlusion

4.4.1 Occlusion Considerations for Gust Fronts

Large cloud formations and heavy rain areas can hide a passing gust front (Figure 4-15). For this reason, disconfirming evidence should not be given much weight in areas with a lot of occlusion. But in areas where there is very little occlusion, disconfirming

evidence should be given more weight.

Confirming evidence in regions where there is little or moderate occlusion should be given a lot of weight. Moderate levels of occlusion may still permit a gust front to reveal itself.

4.4.2 Computing Weights

The block diagram in Figure 4-16 outlines the algorithm used to compute occlusion weights from a reflectivity image and an interest image. We can create an occlusion image from the reflectivity image, which will give each pixel a score reflecting the level of occlusion surrounding that particular pixel. Then the interest image we are weighting can be used to select a confirming or disconfirming weight function.

An occlusion image score of zero indicates the existence of almost no scatterers in that region. Confirming evidence for a region with an occlusion score of zero or very near zero, is given a very low weight because it is very difficult to find true confirming evidence for a gust front in a region with no scatters for a radar beam to bounce off of. With a moderate to high concentration of scatterers, confirming evidence is given moderate weight.

Disconfirming evidence for an area with an occlusion image score of zero or near zero, is given a high weight, because it is very difficult for a gust front to hide in a region with no occluding scatterers.[2] Disconfirming evidence for a region with a high occlusion image score is given a low weight because we remember that a gust front can be hidden in regions with a high concentration of scatterers.

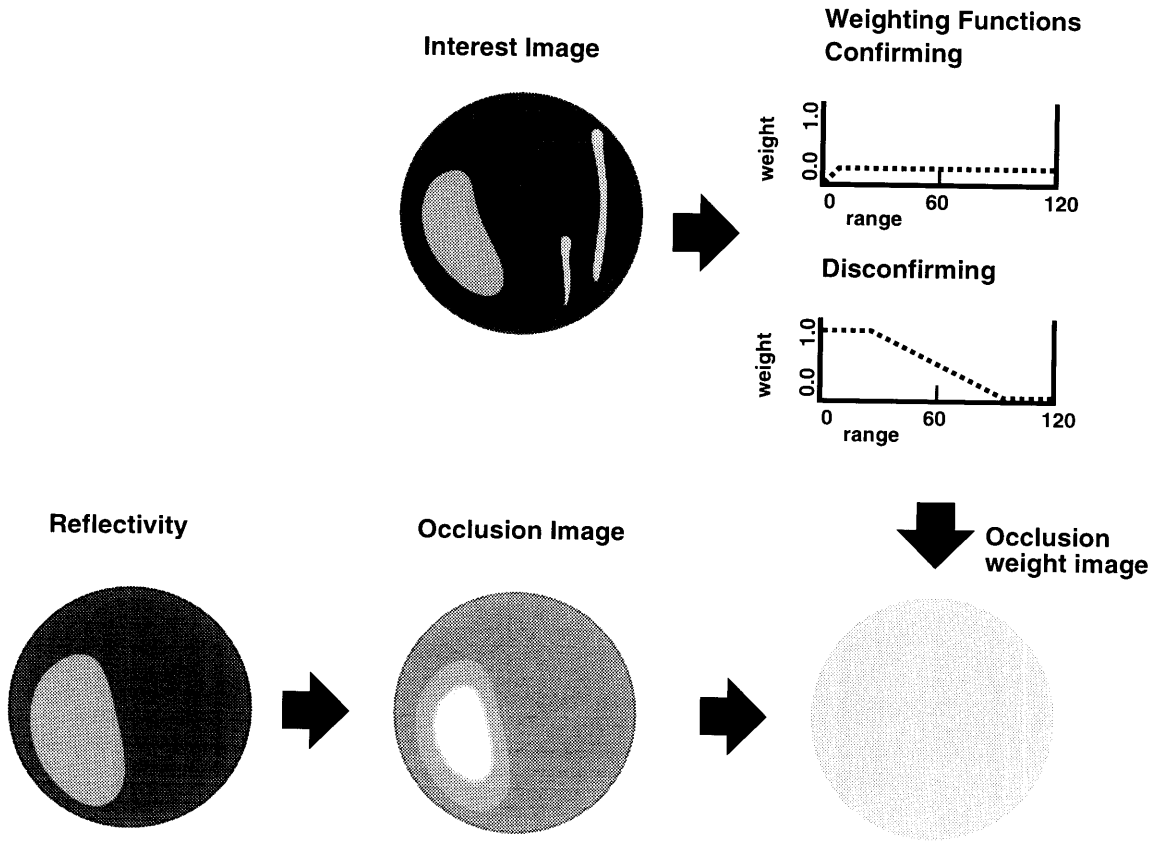
The various weights are stored in an occlusion-weights image.

4.4.3 Implementation

To compute an occlusion image, we can use Functional Template Correlation with a small circular kernel. The functional template is designed so that a gust front will not be marked as an occlusive object. This is achieved by using a kernel that is wider than

[2]The assumption here is that gust fronts will usually carry some scatterers with them.

Figure 4-16: Compute Occlusion Weights



the width of a gust front, and by using a scoring function that expects reflectivity values greater than those exhibited by gust fronts. The functional template desired is so similar to the one used for stratiform rain, that the interest image produced by the stratiform rain template is used for the occlusion image.

Functional Template for stratiform rain and occlusion

```
(defun build-tdwr-strat-rain-fuzzy-template (&aux dz-index)
  (let ((size 8))
    (setq dz-index (create-ellipsoid-kernel size size 1))
    (set-elements-in-interval dz-index dz-index 0 1 1 nil)

    (setq *strat-rain-template*
  (build-fuzzy-template
   dz-index
   '(0)     ;;; angle increment
   '(
     ((0 -512) (40 -256) (60 256) (255 256))
     )
   4     ;;; pyramid level
   (list size size)  ;;; center
   nil t))

  *strat-rain-template*))
```

Appendix A lists the rest of the code used to implement the occlusion weight computation. The confirming and disconfirming weight functions are again stored in table form.

4.5 Putting it all together

Now that the individual pieces have been discussed, we can focus on the big picture of how everything is put together. For each of the interest images generated by MIGFA's feature detectors, a corresponding pixel-level weight image must be computed. The diagram in Figure 4-17 depicts how pixel-level weights are computed for each interest image.

First, we must decide which weighting scheme(s) to use for a particular interest image. For example, reflectivity thin line templates are rotated but may not need to be assigned orientation weights because reflectivity based sensors do not have strong angular dependencies. Other interest images mark areas with a lot of occlusive clutter with nil values and do not allow those pixels to contribute to the final interest image.

Figure 4-17: Compute a combined pixel-level weights image

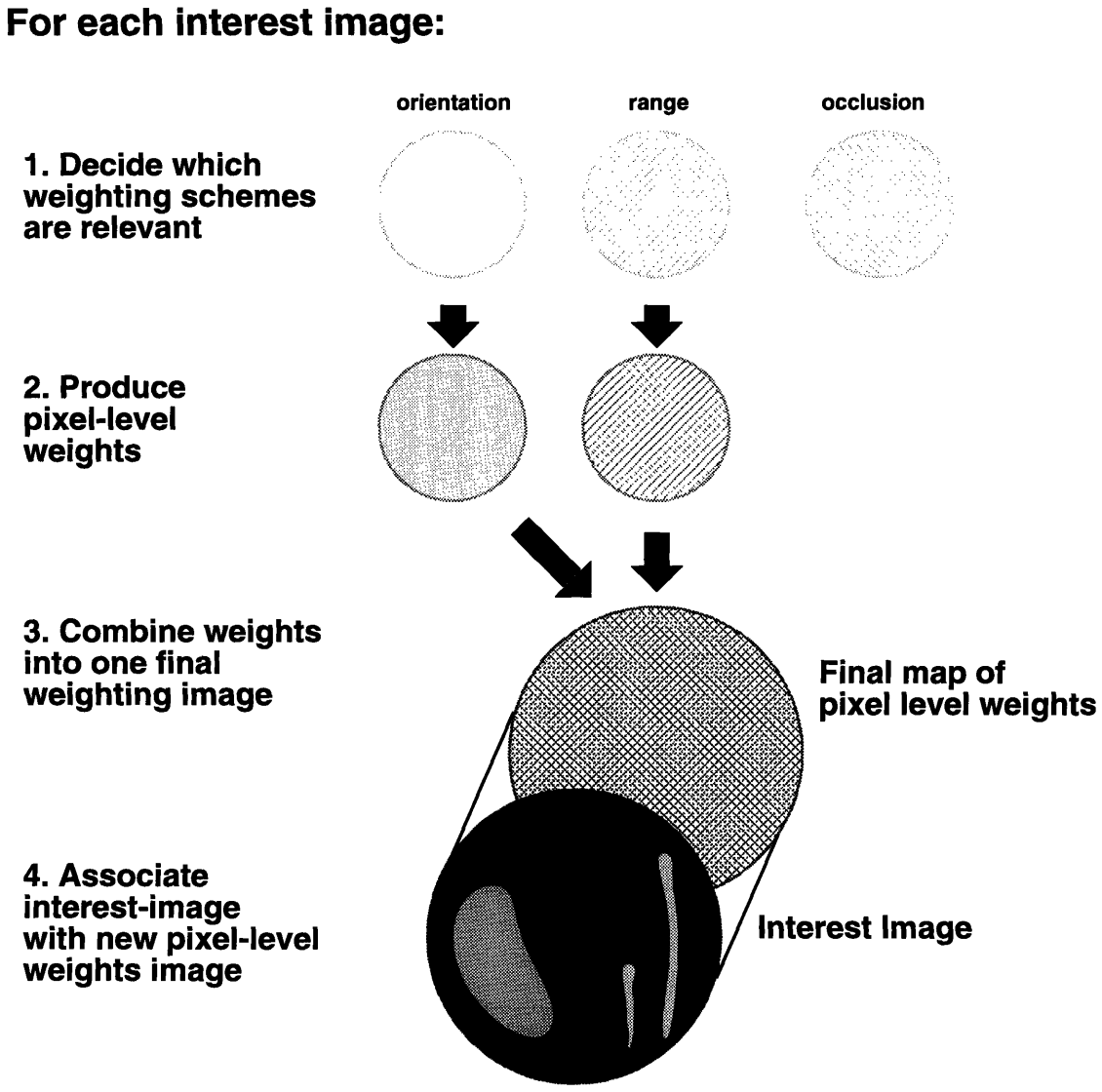
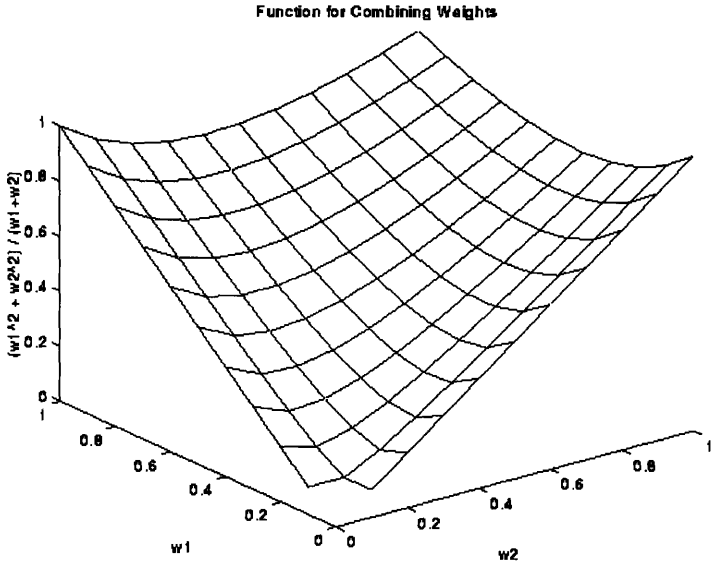


Figure 4-18: Graph of the squared average for two values



The addition of occlusion weights may not affect the non-nil areas significantly, so we could consider omitting occlusion weights. For most interest images, all three are used.

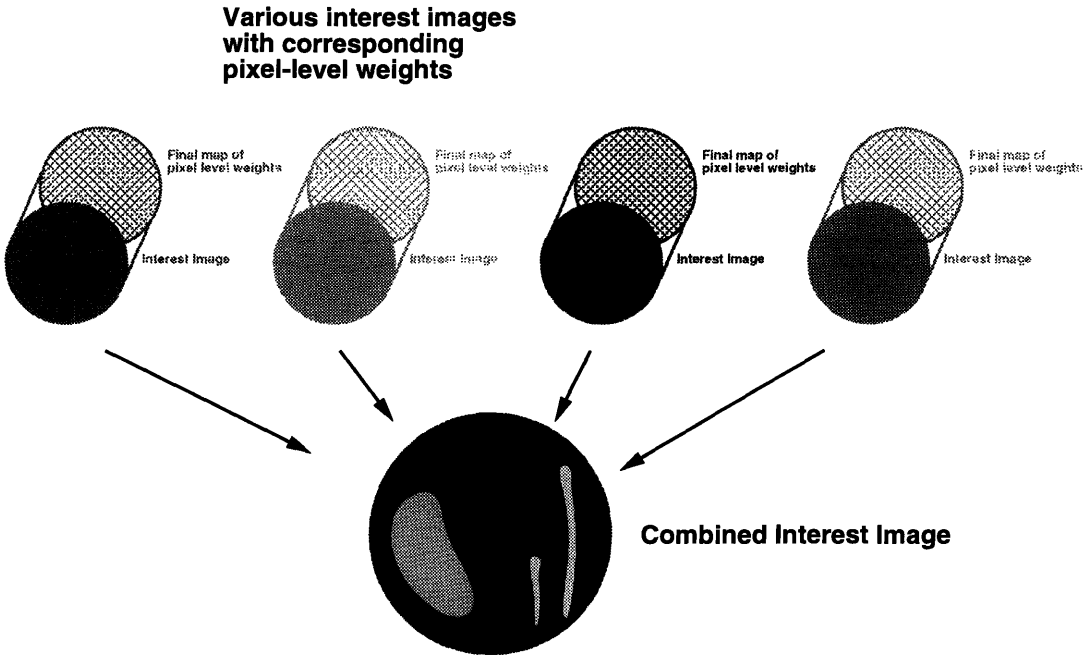
Second, the pixel-level weights must be computed. These computations for the various weighting schemes have already been described above.

Third, the various pixel-level weight images must be combined somehow. One simple way to do this is through the use of a squared average (Figure 4-18).

$$wt_f = \frac{\sum wt_n^2}{\sum wt_n}$$

When all of the weighting schemes agree about the weight of a certain pixel (either high or low) then a final weight value computed with a squared average will be in the same range of the individual weights. If there is some discrepancy among the weighting schemes, a squared average will tend to produce a final weight closer in value to the higher weights. For this reason, squared averages were used instead of some other method like a straight average.

Figure 4-19: Compute combined-interest-image



Fourth, the resultant final weight image must be paired with the appropriate interest image.

4.5.1 Combine Interest Images

The diagram in Figure 4-19 shows what is done after all the interest images and their respective weights have been computed. They are fused into one final combined-interest-image.

For each pixel, a weighted average between all the interest images is computed and stored in the `combined-interest-image`

$$I(x,y) = \frac{\sum i_n(x,y) * wt_n(x,y)}{\sum wt_n(x,y)}$$

One adaptation is worth mentioning here. The old confirming and disconfirming weight pairs for entire interest images are still accessible. One could ignore these and experiment only with the newly computed pixel-level weights. On the other hand,

one could try to benefit from the work that went into determining these pairs by scaling the pixel-level weights by one of these old weights.

$$wt_n(x,y) = conf\_wt * wt_n(x,y)$$

or

$$wt_n(x,y) = disconf\_wt * wt_n(x,y)$$

This approach is used in the current implementation.

In the future, one might consider using only pixel-level weights and consolidating the wisdom stored in the old weights with the new scheme by creating appropriately pre-scaled weighting functions for each feature detector.

4.5.2 Implementation Issues

The code for computing the final combined-interest-image using pixel-level weights, can be found in Appendix A under the section **Computing an Average Weighted Interest Image.**

The following changes to the MIGFA system are necessary for the addition of pixel-level weight based data fusion.

1. Incorporate tools for pixel-level data fusion

The files test.lsp, fusion_comb-interest.lsp, fusion.lsp, and fusion_librarytools.lsp contain tools supporting pixel-level data fusion. they must be included in the list of files to load in file **`exp_xgft.lsp`**.

2. Cause pixel-level weights to be computed with interest images

The feature detectors defined in file **`xgft_library.lsp`**, require the addition of a line like

```
(setf (has-weights match) (weight-interest-image match orient))
```

to cause the generation of pixel-level weights for each appropriate interest image.

3. Define and call alternate procedures for fusing interest images

The file **exp_xgft.lsp**:

- Create functions **compute-weighted-interest-image,**

combine-weighted-interest.

- Modify the function **compute-xgft** by replacing the line

(setq *interest* (compute-interest-image *interest-image-set*))

with the (setq *interest* (compute-weighted-interest-image *interest-image-set*))

4. Add a few data fusion specific flags and variables

The variable **default-pixel-weight** references an integer value used for the comparison and the building of weight images. It can be placed in file **xgft_control.lsp**.

The flag ***pixel-weights-off-dbg*** determines whether or not pixel-level weights should be computed. The flag ***fusion-dbg*** determines whether or not certain intermediate images should be displayed. Both of the flags are set in **xgft_mode.lsp** and declared in **xgft_control.lsp**.

Chapter 5

Results, Conclusions, and Recommendations

5.1 Results

Figure 5-1 shows a snapshot of MIGFA without pixel-level weights processing some data. Figure 5-2 shows a snapshot of the MIGFA with pixel-level weights processing the same set of data. A visual comparison of the two pictures shows MIGFA with pixel-level weights tracing the lower left gust front a little more completely.

Both combined-interest images of MIGFA highlight regions where the lower left gust front exists, but the values of the pixels for MIGFA without pixel-level weights are not enough to trigger a detection. The combined-interest image of MIGFA with pixel-level weights has those same regions highlighted but with slightly greater intensity. The pixel-level weights probably allowed confirming evidence to have greater influence in the generation of the combined-interest image.

The performance of MIGFA with and without pixel-level weights on a common set of data was evaluated against a human's interpretation of the same radar data, using a automatic scoring procedure described by D. Klingle-Wilson in [6]. Table 5.1 summarizes the results of MIGFA performing on sets of data with and without pixel-level weights.

Figure 5-1: MIGFA without pixel-level weights

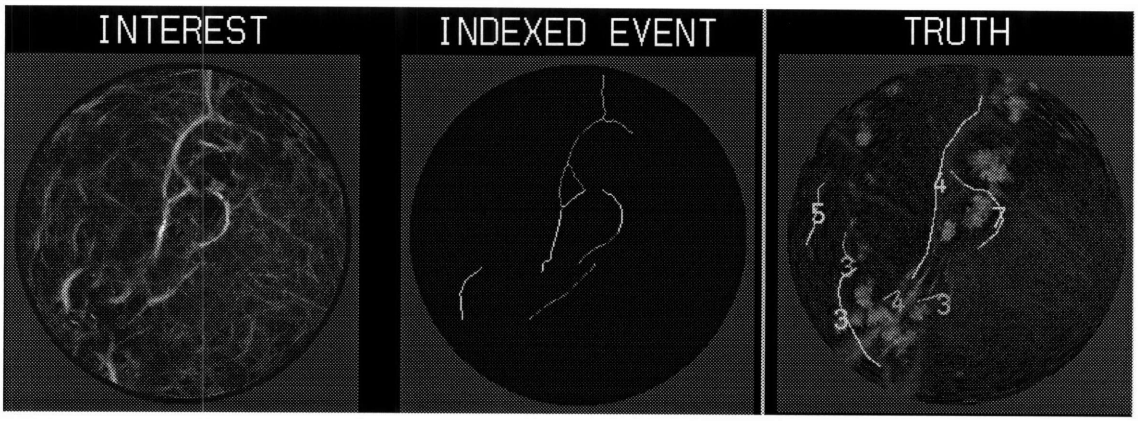


Figure 5-2: MIGFA with pixel-level weights

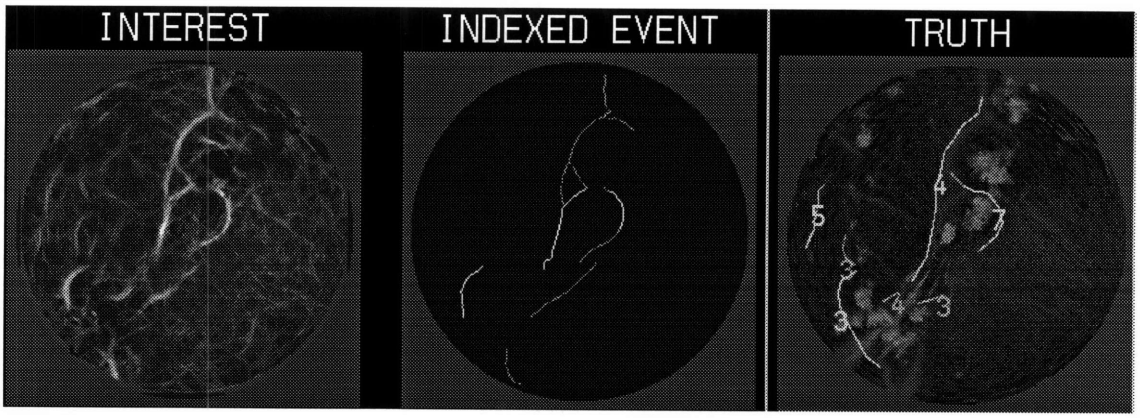


Table 5.1: MIGFA Performance without Pixel-level Weights

DATE	Percent Length Detected PLD	Percent False Length Detected PFD
08/03/93	54.4	16.2
08/04/93	75.9	15.2

Table 5.2: MIGFA Performance with Pixel-level Weights

DATE	Percent Length Detected PLD	Percent False Length Detected PFD
08/03/93	59.5	15.8
08/04/93	76.6	16.6

5.2 Evaluation

In general, MIGFA with pixel-level weights provided improved detection. It was also possible to decrease the number of false detections, but there were cases where the number of false detections increased as well.

The weighting functions require careful construction. Both the shapes of the weighting functions and their relative values can influence the final output noticeably (significantly negative effects are more easily noticed and produced than the positive ones).

Depending on what characteristic is desired, pixel-level weights can be used to push MIGFA's performance characteristics in one direction. Careful adjustment of MIGFA's weights can result in higher detection rates, while only slightly raising the number of false detections. Or, MIGFA's weights can be used to filter out a greater number of false detections while only slightly lowering the detection rate.

The effect of using pixel-level weights on overall TDWR MIGFA performance is small. However, TDWR by itself is not the final testbed. The goal is to fuse data from multiple radars, which for any gust front will have differing viewpoints on orientation, distance, and occlusion.

With the completion of Federal Aviation Administration's Integrated Terminal

Weather System approaching, future work would involve determining how interest-images from a number of different radars ought to be combined.

5.3 Conclusion

The method of using pixel-level weights, enables us to have varying degrees of confidence in data from the same image. It provides us with a good mechanism for encoding and applying object- and context-dependent knowledge. For gust front detection through MIGFA, pixel-level weights have shown that they can be a valuable addition that shows promise for future uses.

Appendix A

Code Listing

A.1 Set Up Templates to be used by Weighting Functions

A.1.1 Lisp Code

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Builds templates used for data fusion
;;; Assigns to global variables *angle-template* and *range-template*
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun build-fusion-templates()
  (setq range-index (an-array has-sizes *cart-sizes*
 has-element-type a-short
 has-exponent 0))
  (setq angle-index (an-array has-sizes *cart-sizes*
 has-element-type a-short
 has-exponent 0))
  (allocate-array angle-index)
  (allocate-array range-index)
  (build-range-index-c range-index)
  (build-angle-index-c angle-index)

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;; Set up global weight variables ;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  (setq *range-weights* (copy-of-array angle-index))
  (setq *angle-weights* (copy-of-array angle-index))
  (setq *occlusion-weights* (copy-of-array angle-index))

  ;;;;;;;;;;;;;;;;;;;;;;;;
  ;;; Build Templates ;;;
  ;;;;;;;;;;;;;;;;;;;;;;;;
  (setq *angle-template*
(list
 angle-index
;;; confirming weight
 (build-function-table '(((0 128) (90 256) (180 128)))) ; weighted
;  (build-function-table '(((0 128) (90 128) (180 128)))) ; neutral
```

```
;;; disconfirming weight
 (build-function-table '(((0 128) (90 150) (180 128)))) ; weighted
;  (build-function-table '(((0 128) (90 128) (180 128)))) ; neutral
;;; confirming weight for weak angle correlation like DZ
 (build-function-table '(((0 150) (90 190) (180 150)))) ; weighted
;  (build-function-table '(((0 128) (90 128) (180 128)))) ; neutral
;;; disconfirming weight for weak angle correlation DZ
 (build-function-table '(((0 128) (90 150) (180 128)))) ; weighted
;  (build-function-table '(((0 128) (90 128) (180 128)))) ; neutral
 ))
  (setq *range-template*
(list
 range-index
;;; confirming weight
 (build-function-table '(((0 128) (10 350) (120 80)))) ;confirm
;  (build-function-table '(((0 128) (120 128)))) ; neutral
;;; disconfirming weight
 (build-function-table '(((0 128) (10 200) (120 60)))) ;disconfirm
;  (build-function-table '(((0 128) (120 128)))) ; neutral
 ))
  (setq *occlusion-template*
(list
 (build-function-table '(((0 0) (10 180) (180 180)))) ;confirm
 (build-function-table '(((0 200) (20 180)
  (160 20) (180 20)))) ;disconfirm
 ))
)
```

A.1.2 C Code

```
/*************************************/
/* builds index for angle weighting */
/*************************************/
int build_angle_index(angle_index)
  object angle_index;
{
  short * index = sar_sbase(angle_index);
  short * temp;
  int i, j;
  double angle, xx, yy;
  int x = angle_index->sar_xsize;   int halfx = (int) x/2;
  int y = angle_index->sar_ysize;   int halfy = (int) y/2;

  printf("ANGLE_INDEX has size: (%d, %d)\n", x, y);
  /******************/
  /* print index */
  /******************/
/*  print_array_sketch(angle_index);*/
  /***************************/
  /* calculate angle weights */
  /***************************/
  for(j=0; j<y; j++) {
    for(i=0; i<x; i++) {
      xx = (double) i-halfx;     /* calculate x distance away from center */
      yy = (double) j-halfy;     /* calculate y distance away from center */
      angle = 180*atan(yy/xx)/3.14159;
                                 /* calculate angle in degrees from center */
      if (angle < 0.0)
        *(index+j*x+i) = (short) (180 + ((short)(angle)));
      else
        *(index+j*x+i) = (short)(angle);
    }
  }
  *(index+halfy*x+halfx) = 0; /* manually set center value to zero */
  /******************/
  /* print index */
  /******************/
  printf("* Finished building angle_index. *\n");
```

```
}
/*************************************/
/* builds index for range weighting */
/*************************************/
int build_range_index(range_index)
  object range_index;
{
  short * index = sar_sbase(range_index);
  short * temp;
  int i, j;
  double range, xx, yy;
  int x = range_index->sar_xsize;   int halfx = (int) x/2;
  int y = range_index->sar_ysize;   int halfy = (int) y/2;

  printf("RANGE_INDEX has size: (%d, %d)\n", x, y);
  /*******************/
  /* calculate range */
  /*******************/
  for(j=0; j<y; j++) {
    for(i=0; i<x; i++) {
      xx = (double) i-halfx;      /* calculate x distance away from center */
      yy = (double) j-halfy;      /* calculate y distance away from center */
      range = sqrt(xx*xx+yy*yy); /* calculate absolute distance from center */
      *(index+j*x+i) = (short) range;
    }
  }
  /******************/
  /* print index */
  /******************/
  printf("* Finished building range_index. *\n");
}
```

A.2 Orientation Weight Computation

A.2.1 Lisp Code

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Computes weights for given input image base on orientation.
;;; Stores weights in global variable *angle-weights*.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun weight-interest-image-angle (img orient)
  (assert (object-is an-array img) nil
  "interest images list contains a non-array element")
  (assert (eq (sk-array-has-element-type img) a-short) nil
  "interest images list contains an array that is not a-short")
  (assert (object-is an-array orient) nil
  "interest images list contains a non-array element")
  (assert (eq (sk-array-has-element-type orient) a-short) nil
  "interest images list contains an array that is not a-short")
  (allocate-array img)
  (allocate-array orient)

  (weight-angle-c img orient *angle-weights*
  (car *angle-template*)     ; angle indices
  (cadr *angle-template*)    ; angle table-confirming
  (caddr *angle-template*)   ; angle table-disconfirming
  (cadr *range-template*)    ; unused
  *default-pixel-weight*)
  *angle-weights*
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Computes weights for given input image base on orientation
;;;  for interest images with weak angle correlation like
;;;  interest images generated from DZ.
;;; Stores weights in global variable *angle-weights*.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun weight-interest-image-angle-DZ (img orient)
  (assert (object-is an-array img) nil
  "interest images list contains a non-array element")
  (assert (eq (sk-array-has-element-type img) a-short) nil
  "interest images list contains an array that is not a-short")
  (assert (object-is an-array orient) nil
  "interest images list contains a non-array element")
  (assert (eq (sk-array-has-element-type orient) a-short) nil
  "interest images list contains an array that is not a-short")
  (allocate-array img)
  (allocate-array orient)

  (weight-angle-c img orient *angle-weights*
  (car *angle-template*)     ; angle indices
  (cadddr *angle-template*)    ; weak angle table-confirming
  (cadddr (cdr *angle-template*)); weak angle table-disconfming
  (cadr *range-template*)    ; unused
  *default-pixel-weight*)
  *angle-weights*
)
```

A.2.2 C Code

```
/*****************************************
 * Based on angle considerations
 *  Add confirming weights to conf.
 * Indices found in ind.
 * Lookup Table found in tab.
 *****************************************/
int weight_angle(img, orient, conf, angle_ind, angle_tab,
            angle_tab2, range_tab,
     default_pixel_weight)
```

```
  object img, orient, conf, angle_ind, angle_tab, angle_tab2, range_tab;
  int default_pixel_weight;
{
  short * image = sar_sbase(img);
  short * orientation = sar_sbase(orient);
  short * confirm = sar_sbase(conf);
  short * angle_index = sar_sbase(angle_ind);
  short * angle_table = sar_sbase(angle_tab);
  short * angle_table2 = sar_sbase(angle_tab2);
  short * range_table = sar_sbase(range_tab);
  register short * temp, *otemp;
  register short offset;
  int val;

  int x = orient->sar_xsize;
  int y = orient->sar_ysize;
  register int i, j;


  printf("***Weighting based on angle.***\n");
/************************************************
 * Initialize weights to 0
 ************************************************/

  for(temp=confirm, otemp=orientation, j=0; j<y; j++) {
    for(i=0; i<x; i++) {
      if (!sat_smissing(*otemp)) {
        *temp = 0;
      } else *temp = *otemp;   /* set to nil */
      temp++; otemp++;
    }
  }
/************************************************
 * compute weight using angle template
 ************************************************/
  for(temp=confirm, otemp=orientation, j=0; j<y; j++) {
    for(i=0; i<x; i++) {
      if (!sat_smissing(*otemp)) {
      /*** angle with respect to origin = abs(*otemp - *angle_index + 90) ***/
        offset = abs(*otemp - *angle_index + 90);
if (offset > 180)
         offset -= 180;
if ( (*image) >= 128) {
  /*******************************************/
  /* pixel weight when evidence is confirming */
  /*******************************************/
  *temp = *(angle_table + offset);
} else {
  /***********************************************/
  /* pixel weight when evidence is disconfirming */
  /***********************************************/
         *temp = *(angle_table2 + offset);
}
      }
      temp++; otemp++; angle_index++; image++;
    }
  }
/*  printf("****Orientation****\n"); */
/*  print_array_sketch(orient);*/
/*  printf("****Confirming weight:****\n");*/
/*  print_array_sketch(conf);*/
}
```

A.3 Range Weight Computation

A.3.1 Lisp Code

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Computes weights for given input image base on range.
;;; Stores weights in global variable *range-weights*.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun weight-interest-image-range (img orient)
  (assert (object-is an-array img) nil
  "interest images list contains a non-array element")
  (assert (eq (sk-array-has-element-type img) a-short) nil
  "interest images list contains an array that is not a-short")
  (assert (object-is an-array orient) nil
  "interest images list contains a non-array element")
  (assert (eq (sk-array-has-element-type orient) a-short) nil
  "interest images list contains an array that is not a-short")
  (allocate-array img)
  (allocate-array orient)

  (weight-range-c img orient *range-weights*
  (car *range-template*)      ; range indices
  (cadr *range-template*)     ; range table-confirming
  (caddr *range-template*)    ; range table-disconfirming
  (cadr *angle-template*)     ; unused
  *default-pixel-weight*)
  *range-weights*
)
```

A.3.2 C Code

```
/*********************************************
 * Based on range considerations
 *  Add confirming weights to conf.
 * Indices found in ind.
 * Lookup Table found in tab.
 *********************************************/
int weight_range(img, orient, conf, range_ind, range_tab,
              range_tab2, angle_tab,
      default_pixel_weight)
  object img, orient, conf, range_ind, range_tab, range_tab2, angle_tab;
  int default_pixel_weight;
{
  short * image = sar_sbase(img);
  short * orientation = sar_sbase(orient);
  short * confirm = sar_sbase(conf);
  short * range_index = sar_sbase(range_ind);
  short * range_table = sar_sbase(range_tab);
  short * range_table2 = sar_sbase(range_tab2);
  short * angle_table = sar_sbase(angle_tab);
  register short * temp, *otemp;
  register short offset;
  int val;

  int x = orient->sar_xsize;
  int y = orient->sar_ysize;
  register int i, j;

  printf("***Weighting based on range.***\n");
/*********************************************
 * Initialize weights to 0
 *********************************************/
  for(temp=confirm, otemp=orientation, j=0; j<y; j++) {
    for(i=0; i<x; i++) {
      if (!sat_smissing(*otemp)) {
        *temp = 0;
      } else *temp = *otemp;   /* set to nil */
```

```
      temp++; otemp++;
    }
  }
/***********************************************
 * compute weight using range template
 ***********************************************/
  for(temp=confirm, otemp=orientation, j=0; j<y; j++) {
    for(i=0; i<x; i++) {
      if (!sat_smissing(*otemp)) {
        if ( (*image) >= 128) {
  /*****************************************/
  /* pixel weight when evidence is confirming */
  /*****************************************/
  *temp = *(range_table + (*range_index));
} else {
  /*********************************************/
  /* pixel weight when evidence is disconfirming */
  /*********************************************/
  *temp = *(range_table2 + (*range_index));
}
      }
      temp++; otemp++; range_index++; image++;
    }
  }
/*  printf("****Confirming weight:****\n"); */
/*  print_array_sketch(conf);*/
}
```

A.4 Occlusion Weight Computation

A.4.1 Lisp Code

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Computes weights for given input image base on occlusion.
;;; Stores weights in global variable *occlusion-weights*.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun weight-interest-image-occlusion (img orient)
  (assert (object-is an-array img) nil
  "interest images list contains a non-array element")
  (assert (eq (sk-array-has-element-type img) a-short) nil
  "interest images list contains an array that is not a-short")
  (assert (object-is an-array orient) nil
  "interest images list contains a non-array element")
  (assert (eq (sk-array-has-element-type orient) a-short) nil
  "interest images list contains an array that is not a-short")
  (allocate-array img)
  (allocate-array orient)

  (setq *occlusion-weights* *stratiform-rain*)
  *occlusion-weights*
)
```

A.5 Combining Weights

A.5.1 Lisp Code

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Combines weights.  num-weights is number of maps of weights
;;;  to be combined.
;;; Returns an array containing the combined weights.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun combine-weights (num-weights)
  (let* ((comb-w (copy-of-array orient)))
    (combine-weights-c comb-w
```

```
      *angle-weights*
      *range-weights*
      *occlusion-weights*
      num-weights)
    comb-w)
  )
```

A.5.2 C Code

```
/*******************************************
 * Method of combining weights
 *******************************************/
int combine_weights(conf, angle_weights, range_weights, occlusion_weights,
  num_weights)
  object conf, angle_weights, range_weights, occlusion_weights;
  int num_weights;
{
  short * confirm = sar_sbase(conf);
  short * a_w = sar_sbase(angle_weights);
  short * r_w = sar_sbase(range_weights);
  short * o_w = sar_sbase(occlusion_weights);
  register short * temp, *otemp;
  double val, div;
  register short a_val, r_val, o_val;
  int testing=0;
  int notesting = 0;

  int x = conf->sar_xsize;
  int y = conf->sar_ysize;
  register int i, j;

  if (num_weights == 2)
    printf("***Combining Angle and Range weights***\n");
/***********************************************
 * Combine weights
 ***********************************************/
  for(temp=confirm, j=0; j<y; j++) {
    for(i=0; i<x; i++) {
      if (!sat_smissing(*temp)) {
        if (!sat_smissing(*o_w)) {
          if (*o_w > 128) testing++;
    else notesting++;
} else notesting++;

a_val = *a_w;  r_val = *r_w;  o_val = *o_w;
if (num_weights == 2) {
          val =  (double) ((a_val*a_val)+(r_val*r_val)) / (a_val+r_val);
  *temp = (int)ROUND(val);
}

      }
      temp++; a_w++; r_w++; o_w++;
    }
  }

  printf("*****occlusion testing:%d no:%d*****\n", testing, notesting);
}
```

A.6 Top Level Function for requesting the computation of pixel-level weights for one interest image

A.6.1 Lisp Code

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Computes weights for given input image and orientation
;;;  based on angle and distance from radar.
;;; Returns an array containing weights.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun weight-interest-image (img orient)
  (weight-interest-image-angle img orient)
  (weight-interest-image-range img orient)
  (weight-interest-image-occlusion img orient)

  (combine-weights 2)
)

(defun weight-interest-image-DZ (img orient)
  (weight-interest-image-angle-DZ img orient)
  (weight-interest-image-range img orient)
  (weight-interest-image-occlusion img orient)

  (combine-weights 2)
)
```

A.7 Calling for computation of pixel-level weights in MIGFA

A.7.1 Lisp Code fragments

The following fragments are found in the file `xgff_library.lsp` in functions like `detect-tdwr-non-cell-conv-motion`. They create the structural links between the interest image and pixel-level weight map.

```
.
(setf (has-weights match) (weight-interest-image match orient))
.
(setf (has-weights match) (weight-interest-image-DZ match orient))
.
(setf (has-weights match) (weight-interest-image match orient))
.
(setf (has-weights match) (weight-interest-image match orient))
.
(setf (has-weights match) (weight-interest-image match orient))
.
```

A.8 Computing an Average Weighted Interest Image

A.8.1 Lisp Code

These functions found in file **exp_xgff.lsp**. They are among the upper level MIGFA functions which call the necessary procedures to perform pixel-level weight based data fusion.

```
(defun compute-weighted-interest-image
  (&optional (interest-image-set *interest-image-set*))
  (setq *interest-images* nil)
  (dolist (detector interest-image-set)
    (push (funcall (intern (concat "DETECT-" detector)))
          *interest-images*))
  (setq interest-imgs
        (ecase *radar-system*
          (asr9 (assemble-asr9-evidence))
          (tdwr (assemble-tdwr-evidence))))
  (setq *interest* (combine-weighted-interest interest-imgs))
  (when *comb-dbg*
    (display-interest-images)
    (when *append-figures* (append-figure))
    (conditional-break "done"))
  *interest*)

(defun combine-weighted-interest (&optional (interest-imgs *interest-images*))
  (if *pixel-weights-off-dbg*
      (let* ((interest (average-interest-images interest-imgs)))
(print "COMBINING INTEREST")
(print "COMBINING INTEREST")
(setq interest
      (average-interest-images
       (list interest (get-interest-image 'anticipation))))
(setq *interest* interest))
    (let* ((interest (average-weighted-interest-images interest-imgs)))
      (print "COMBINING ***WEIGHTED*** INTEREST")
      (print "COMBINING ***WEIGHTED*** INTEREST")
      (setq interest
    (average-interest-images
     (list interest (get-interest-image 'anticipation))))
      (setq *interest* interest))
))
```

average-weighted-interest-images

This is file **fusion_comb-interest.lsp**. It has the function that performs the fusion of interest images using averaged weighting at the pixel level.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; create a set of default weights = 60 ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *default-pixel-weights-array* (copy-of-array angle-index))
(set-elements-in-interval *default-pixel-weights-array*
                          *default-pixel-weights-array*
                          *default-pixel-weight*
                          0 500 nil)
(eval-when (compile load eval)
           (define-attribute 'has-weights))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Input: List of interest images.
;;; Effects: Combines several interest images into one interest image
;;;        based on pixel-level weights.
;;; Output: A combined interest image
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun average-weighted-interest-images (intrinsic-images)
  (print "In average-weighted-interest-images")
  (dolist (img intrinsic-images)
     (assert (object-is an-array img) nil
             "interest images list contains a non-array element")
     (assert (eq (sk-array-has-element-type img) a-short) nil
             "interest images list contains an array that is not a-short")
     (allocate-array img))

  (let* ((confirming-weights (cons nil nil))
         (disconfirming-weights (cons nil nil))
         (pixel-weights (cons nil nil))
         (number-images (length intrinsic-images))
         (output (copy-of-array (car intrinsic-images))))

    (assert (<= number-images 10) nil
            "average-interest-images - max number of images is 10")

    (unless (= number-images 1)
       (dolist (img intrinsic-images)
          (let ((pix-wt (has-weights img))
                (con-wt (has-confirming-weight img))
                (dis-wt (has-disconfirming-weight img)))

       (if pix-wt (print "Using *default-pixel-weights-array*"))
       (print "Confirming and disconfirming weights")
       (print con-wt)
       (print dis-wt)

            (tconc pixel-weights (if pix-wt
                                     pix-wt
                                   *default-pixel-weights-array*))
            (tconc confirming-weights (if con-wt con-wt 1.0))
            (tconc disconfirming-weights (if dis-wt dis-wt 1.0))))


       (ssu_avg_weighted_interest-c output
                                    intrinsic-images
                                    (car pixel-weights)
                                    (car confirming-weights)
                                    (car disconfirming-weights)
                                    *default-pixel-weight*
                                    number-images))

    output))
```

A.8.2 C Code

This function is found in file test.lsp with the other data fusion tools. It is grouped with other compilable C functions.

```
ssu_avg_weighted_interest (output, imgs, pixel_weights, conf_weights, disconf_weights,
   default_pixel_weight, num_imgs)
     object output, imgs, pixel_weights, conf_weights, disconf_weights;
     int default_pixel_weight;
     int num_imgs;

{ short *xop = sar_sbase(output);
```

```
  double c_wts[MAX_NUMBER_IMGS_TO_AVG];
  double dc_wts[MAX_NUMBER_IMGS_TO_AVG];
  short *p_wts[MAX_NUMBER_IMGS_TO_AVG];
  short *xip[MAX_NUMBER_IMGS_TO_AVG];
  int j,k;
  int jsize = output->sar_xsize * output->sar_ysize;
  double sum, count, val, avg, wt, pwt, d_p_w;

  printf("Using ssu_avg_weighted_interest-c\n");

  d_p_w = (double) default_pixel_weight;
  for(k = 0;
      k < num_imgs;
      ++k, imgs = imgs->sat_cdr,
      pixel_weights = pixel_weights->sat_cdr,
      conf_weights = conf_weights->sat_cdr,
      disconf_weights = disconf_weights->sat_cdr) {
    xip[k] = sar_sbase(imgs->sat_car);
    p_wts[k] = sar_sbase(pixel_weights->sat_car);
    c_wts[k] = conf_weights->sat_car->sat_ldouble;
    dc_wts[k] = disconf_weights->sat_car->sat_ldouble; }

  for(j = 0; j < jsize; ++j) {
    for(k = 0, sum = 0.0, count = 0.0; k < num_imgs; ++k) {
      val = *(xip[k]);
      pwt = *(p_wts[k]);
      xip[k]++;  p_wts[k]++;
      if (!sat_smissing(val)) {

if (pwt < 0.0) {
          pwt = 0.0;  /* don't allow negative values */
}

/*************************************************************/
/* incorporate pixel weights by altering wt only */
/*************************************************************/
        if (val >= 128) {
  wt = c_wts[k]*pwt/128.0;
} else {
  wt = dc_wts[k]*pwt/128.0;
}

sum += val * wt;
count += wt; } }
    avg = (count == 0.0) ? SAT_SMISSING : sum / count;
    *xop = (int)ROUND(avg);
    xop++; }

  return(0); }
```

A.9 Lines to enable Lisp to access C functions

```
(defentry build-range-index-c (object)
                                (int build_range_index))
(defentry build-angle-index-c (object)
                                (int build_angle_index))
(defentry build-angle-index2-c (object)
                                (int build_angle_index2))
(defentry print-array-c (object)
                        (int print_array_sketch))
(defentry weight-range-c (object object object object object
   object object int)
  (int weight_range))
(defentry combine-weights-c (object object object object int)
  (int combine_weights))
(defentry weight-angle-c (object object object object object
   object object int)
```

```
  (int weight_angle))
(defentry weight-angle-and-range-c (object object object object object
   object object int)
  (int weight_angle_and_range))
(defentry copy-elements-of-array-c (object object int int int int)
  (int copy_elements_of_array))
(defentry add-elements-of-array-c (object object int int int int)
  (int add_elements_of_array))
(defentry ssu_avg_interest-c (object object object object int)
                             (int ssu_avg_interest))
(defentry ssu_avg_weighted_interest-c (object object object object object
      int int)
                             (int ssu_avg_weighted_interest))


(build-fusion-templates)
```

Bibliography

[1] Richard L. Delanoy and Seth W. Troxel. Machine intelligent gust front detection. *The Lincoln Laboratory Journal*, 6(1), 1993.

[2] Richard L. Delanoy, Jacques G. Verly, and Dan E. Dudgeon. Pixel-level fusion using interest images. In *Proceedings of the 4th National Symposium on Sensor Fusion*, Orlando, Florida, April 1991.

[3] Richard L. Delanoy, Jacques G. Verly, and Dan E. Dudgeon. Functional templates and their application to 3-d object recognition. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, San Francisco, California, March 1992.

[4] M. Eilts, S. Olson, G. Stumpf, L. Hermes, A. Abrevaya, J. Culbert, K. Thomas, K. Hondl, and D. Klingle-Wilson. An improved gust front detection algorithm for the tdwr. In *Proc. 4th Intl. conf. on the Aviation Weather System*, June 1991.

[5] William T. Freeman. Steerable filters and local analysis of image structure. *Thesis (Ph.D.)–Massachusetts Institute of Technology*, 1992.

[6] D.L. Klingle-Wilson, M.F. Donovan, S.H. Olson, and F.W. Wilson. A comparison of the performance of two gust front detection algorithms using a length-based scoring technique. In *Project Report ATC-185*. MIT Lincoln Laboratory, May 1992.

[7] M.W. Merrit, D. Klingle-Wilson, and S.D. Campbell. Wind shear detection with pencil-beam radars. *The Lincoln Laboratory Journal*, 2(483), 1989.

[8] H. Uyeda and D.S. Zrnic. Automated detection of gust fronts. *J. Atmost, Oceanic Tech.*, 3(36), 1986.