IMPROVING NEXRAD DATA
Data Quality Algorithm Progress
FY2007 Annual Report

Prepared for: WSR-88D Radar Operations Center
By: Mike Dixon, Greg Meymaris, Scott Ellis, and John C. Hubbert
PI: John C. Hubbert
National Center for Atmospheric Research

5 March 2008
# Contents

Executive Summary 2

1 Spectrum Width Censoring with SZ 4  
1.1 Introduction .................................................. 4  
1.2 Short PRT $R_1/R_2$ ............................................. 5  
1.3 Conclusions .................................................. 11

2 Staggered PRT Clutter Filtering 16  
2.1 Introduction .................................................. 16  
2.2 Requirements .................................................. 17  
2.3 Methodology .................................................. 18  
2.4 Results ...................................................... 18  
2.5 Conclusions .................................................. 93

3 REC Updates 95  
3.1 Results ...................................................... 95  
3.2 Summary of Results ......................................... 101

References 104

Appendix 105  
A. Single polarization CMD (AEL and C-code)  
B. Dual polarization CMD (AEL and C-code)  
C. CMD conference publications  
   1. AMS Annual Meeting, IIPS, San Antonio, TX  
   2. AMS Radar Meteorology Conference, Cairns, AU
Executive Summary

This report provides continued analysis of the algorithms for NEXRAD Data Quality and Range-Velocity ambiguity mitigation.

A decision briefing was given by NCAR in conjunction with RS Information Systems and SI International at the March 2007 NEXRAD TAC meeting. The TAC recommended CMD for deployment on the WSR-88Ds. In September the NEXRAD SREC approved CMD for WSR-88D deployment. This represents the culmination of many years of effort by NCAR in the development of AP clutter mitigation strategies.

The CMD provides real time identification of ground clutter contaminated data and the subsequent clutter filtering (with GMAP) of the clutter contaminated data. Zero velocity weather echoes are not identified by CMD and thus such weather echoes are left intact, i.e., they are not processed by a clutter filter. Thus, CMD offers a final solution to the long standing problem of AP ground clutter contamination. During FY2007, NCAR continued to develop the new CMD input parameter CPA (Clutter Phase Alignment). Theory was developed and simulations were carried out. CPA is a direct measure of the defining characteristic of non-moving ground clutter targets: constant backscatter phase angle. In contrast, weather is a distributed target and the absolute phase of the received backscatter signal usually varies significantly over the dwell time required for time series collection.

An attractive feature of CPA is that it is calculated in the time domain and should be equally effective for uniform PRT data as well as staggered PRT data. Also, CPA is effective in identifying clutter targets that may be apparent only in part of the received time series due to the radar antenna rotation. For more discussion of CPA, see the the 2007 AMS Radar Meteorology Conference paper on CMD in the Appendix. The CMD AEL and C-code were delivered in separate documents during FY2007 and they are attached as appendices.

A version of CMD has also been developed for dual polarization NEXRAD data. The best feature fields for identifying clutter are the spatial texture of differential reflectivity ($Z_{dr}$) and differential phase ($\phi_{dp}$). The copolar correlation coefficient, $\rho_{hv}$, is not an effective discriminator between ground clutter and weather. Even though the dual polarization feature fields improve CMD performance, CPA continues to be an important and indispensable part of the algorithm.

NCAR investigated and developed an improved SZ-2 spectrum width censoring algorithm. The algorithm was developed specifically for use with the SZ phase coding but has broader implications for general spectrum width estimation. Problems of standard spectrum width estimators was discussed in the NCAR 2006 Annual Report to the ROC. Standard spectrum width estimators typically use the ratio $|R(0)|/|R(1)|$ or $|R(1)|/|R(2)|$ where $R(*)$ is the auto correlation function at lag $\text{"}*$\text{"}. The first estimator works well for wide spectrum widths where as the second estimator performs better for narrow spectrum widths. The improved spectrum width censoring algorithm utilizes this information.

Clutter filtering for staggered PRT times series is investigated via simulations. The algorithm tested is reported in Torres et al. (2005). The clutter filtering technique used
creates five replicates of the true underlying spectrum. Thus, the clutter filter must eliminate unwanted clutter echo not just from one area of the spectrum (i.e, at zero ms\(^{-1}\)) but from five locations in the spectrum. Our analysis shows that the WSR-88D System Specification (ROC, 2007, Excerpt 3.7.2.7) is only met for \(T_1 = 1\) ms and \(T_1 = 785\) \(\mu\)s and length 64 time series. Additionally, when “narrower” weather spectrum widths are present, the performance of the clutter filter degrades (the WSR-88D clutter suppression specification are only given for 4 ms\(^{-1}\)). The results presented here indicate that this clutter filter is not ready for operational use.

Improvement in the Radar Echo Classifier (REC) performance is also documented herein. The original design REC contains two separate Fuzzy Logic algorithms for the identification and separation of ground clutter and precipitation: 1) the APDA (Anomalous Propagation Clutter Detection Algorithm) and 2) the PDA (Precipitation Detection Algorithm). These two algorithms are intended to work in conjunction to provide the most robust identification and separation of AP clutter and precipitation echo. The APDA identifies radar data that is most likely ground clutter echoes. The PDA identifies radar data that is most likely precipitation echo. Thus, the final identification of contaminated data becomes, “if APDA identified or if \textit{not} PDA identified”. The combination of the two algorithms within the REC provides better classification of precipitation data. This was presented in the 2006 NCAR Annual Report to the ROC. The improved REC, with the PDA embedded, was presented to the TAC in 2006 and more rigorous tests of the improved REC performance was recommended, i.e., longer integration periods were requested. This analysis was performed in 2007 and the results were presented as a Decision Briefing to the TAC in March 2007 meeting. The TAC approved the upgraded REC and recommended that the ROC proceed to deployment. This reports shows the improvements in precipitation estimation when using the improved REC/PDA for longer integration periods. NCAR recommends that both the APDA corrections and PDA be implemented in the REC for optimum RPG precipitation accumulations.
1 Spectrum Width Censoring with SZ

1.1 Introduction

As described in Zrnić et al. (2007), different spectrum width estimators are used in SZ-2 depending on the overlaid echo situation. If there are no overlaid echoes, then the $R_0/R_1$ estimator (Doviak and Zrnić, 1993, p. 136) is used. If there are overlaid echoes, then the $R_1/R_2$ estimator (Doviak and Zrnić, 1993, p. 138) is used for the strong trip, and the $R_0/R_1$ estimator applied to the long PRT surveillance scan is used for the second weakest trip. One problem with this approach is that downstream processes will not know which estimator was used on a particular gate without trying to reconstruct the decision process that leads to the choice of estimators. This is an issue because downstream processes must now not only guard against 3 sets of failure modes, but also try to determine which set of failure modes is applicable for a certain gate. In other words, understanding the errors (biases and standard deviations) in the spectrum width is now much more complicated. This is further complicated by the fact that downstream processes will sometimes perform spatial averages on spectrum widths, in which case it would be very difficult to understand the errors on the averaged spectrum width estimator. Major failure modes for the different estimators include:

1. Long PRT $R_0/R_1$ (2 or more trips, weak trip): Saturates at very low spectrum widths. $R_0/R_1$ estimator saturates at about 55% of the Nyquist velocity, which is 8.5 m/s for the long PRT scan, which leads to a saturation level of about 4.8 m/s. See Figure 1 for a plot of the mean of the estimator as a function of true spectrum width. For the SZ-2 algorithm as detailed in Zrnić et al. (2007) (as well as Zrnić et al. (2004, 2005, 2006)), estimated spectrum widths, from the long PRT scan, larger than $2w_{n,max}v_{a,L}$ ($w_{n,max} = 0.25$ and $v_{a,L}$ is the unambiguous Nyquist velocity of the long PRT data) are censored. Figure 2 shows the 2-D histogram of input (true) spectrum width versus the estimated spectrum width. Only non-censored data are shown. Observe that there are a significant amount of spectrum width values that are saturated but not censored. For example, there are a large number of values where the input spectrum width is greater than 6 m/s but the estimated value is between 3 and 4 m/s. In Figure 3, the percentage of spectrum widths censored as a function of input spectrum width is shown. Even for very large true spectrum widths, about 17% of the saturated estimates are not censored.

2. Short PRT $R_0/R_1$ (1 trip): Saturates at fairly high spectrum widths (at about 19 m/s), but significant biases show up for very narrow spectrum widths. We can see in Figure 4 the very poor performance of the estimator for narrow spectrum widths. In particular, we can see very large number of 0’s, indicating that $|\hat{R}_1| > |\hat{R}_0|$ and very large outliers for narrow spectrum widths.
Figure 1: Mean of $R_0/R_1$ spectrum width estimator on long PRT data for varying input “true” spectrum widths. Generated from simulated I&Q time series with simulation parameters emulating SZ-2 style VCP: $\lambda = 10.5$ cm, PRT = 3306 $\mu$s, and $N = 16$. SNR = 20 dB.

3. Short PRT $R_1/R_2$ (2 or more trips, strong trip): Saturates at moderately high spectrum widths (at about 10 $m/s$). See Figure 5. These spectrum widths may be seen in powerful thunderstorms. The primary problem is that when saturation occurs, it does so in a badly behaved way. Note the large number of zeros and the extremely large spread as the true spectrum width becomes large. There are also some moderately large outliers for very narrow spectrum widths, but certainly the performance is better in this regime than for the short PRT $R_0/R_1$ estimator.

In the long term an improved spectrum width estimator should be incorporated into SZ-2, but in the short term it may be preferable to attempt to detect when $R_1/R_2$ is saturating and censor it. Thus, while all concerns should be addressed at some point, the issues discussed here relate to mitigating item 3.

1.2 Short PRT $R_1/R_2$

The $R_1/R_2$ spectrum width estimator (Doviak and Zrnić, 1993, p. 138) is given by

$$\hat{w} = \frac{2}{\pi \sqrt{6} v_a} \left( \log \left( \frac{|\hat{R}_1|}{|\hat{R}_2|} \right) \right)^{1/2}$$

if $|\hat{R}_1| > |\hat{R}_2|$, otherwise $\hat{w} = 0$, where $v_a$ is the unambiguous Nyquist velocity, and $\hat{R}_1$ and $\hat{R}_2$ are the complex unbiased autocorrelation estimates of the complex I&Q time series.
Figure 2: 2-D histogram for $R_0/R_1$ spectrum width estimator on long PRT data, including censoring of large spectrum widths, for varying input “true” spectrum widths. Generated from simulated I&Q time series with simulation parameters emulating SZ-2 style VCP: $\lambda = 10.5 \text{ cm}$, $\text{PRT} = 3306 \mu s$, and $N = 16$. SNR = 20 dB.

Figure 3: Percent censored for $R_0/R_1$ spectrum width estimator on long PRT data for varying input “true” spectrum widths. Generated from simulated I&Q time series with simulation parameters emulating SZ-2 style VCP: $\lambda = 10.5 \text{ cm}$, $\text{PRT} = 3306 \mu s$, and $N = 16$. SNR = 20 dB.
Figure 4: 2D histogram of true versus estimated $R_0/R_1$ spectrum width estimator on short PRT data for varying input “true” spectrum widths. Generated from simulated I&Q time series with simulation parameters emulating SZ-2 style VCP: $\lambda = 10.5 \text{ cm}$, PRT = 785 $\mu$s, and $N = 64$. SNR = 20 dB.

Figure 5: 2D histogram of true versus estimated $R_1/R_2$ spectrum width estimator on short PRT data for varying input “true” spectrum widths. Generated from simulated I&Q time series with simulation parameters emulating SZ-2 style VCP: $\lambda = 10.5 \text{ cm}$, PRT = 785 $\mu$s, and $N = 64$. SNR = 20 dB.
\( \hat{R}_i = (n - i)^{-1} \sum_{n=0}^{N-i-1} V^*(n) V(n + i) \) where \( V \) is the complex I&Q time series of length \( N \). This formula represents a 2 point fit of a Gaussian magnitude autocorrelation function to the estimated magnitude autocorrelation function. It is helpful to recall that narrow spectrums translate into wide autocorrelation functions and vice versa.

The three main advantages of this estimator is that an estimate of the noise level is not necessary, performance is better for true narrow spectrum widths than \( R_0/R_1 \) and, in the case of SZ, power from off trip echoes does not significantly impact the performance of the estimator until quite small strong trip to weak trip power ratios arise.

The disadvantages are for narrow spectrum widths (relative to the Nyquist velocity), \( |\hat{R}_2| \) can be greater than \( |\hat{R}_1| \) (thus \( \hat{w} = 0 \)) causing a bimodal distribution of estimates, and for large spectrum widths there is extremely poor performance of the estimator as it saturates. The latter issue occurs because as \( w \) increases \( |R_2| \) becomes small enough that random errors dominate the estimate, \( |\hat{R}_2| \), and thus \( \langle |\hat{R}_2| \rangle \) (\( \langle \cdot \rangle \) indicating average) “saturates” at a small value above 0. Simultaneously \( \langle |\hat{R}_1| \rangle \) continues to decrease, thus causing the ratio to approach 1. This in turn leads to decreasing spectrum widths. This can be seen as the downward turn of the main distribution of estimates as the input spectrum width increases in Figure 5. With \( \langle |\hat{R}_1| \rangle \) approaching \( \langle |\hat{R}_2| \rangle \), the chances that \( |\hat{R}_2| > |\hat{R}_1| \) increases, thus leading to an increasing number of 0 spectrum width estimates. These zeros can be seen for large spectrum widths at the bottom right of the figure. This saturation behavior is unlike that for the \( R_0/R_1 \) estimator because while \( \langle |\hat{R}_1| \rangle \) does saturate at a value above 0 for increasing spectrum widths, \( \langle R_0 \rangle \) remains constant since by definition it is the average power of the signal. Because of this, the \( R_0/R_1 \) estimator saturates in a more desirable way, namely as the true spectrum width increases the estimator also increases until a certain point and then saturates at a maximum value without displaying a very high variance.

The obvious question is whether this failure mode is something to worry about. Firstly, there have been observed cases in which there were spectrum width values around 8 m/s with reflectivities less than 35 dBZ. It is possible then that an aircraft may fly through this region, and yet the spectrum width values are indicating extreme turbulence (for an example of this, see NEXRAD level 2 data from KPAH on 20:49 UTC on 6 August 2003, which can be downloaded from NCDC). NEXRAD Turbulence Detection Algorithm (NTDA) does detect the turbulence, based on \( R_0/R_1 \). If the spectrum width was only slightly larger, it would have been in the saturation regime for the \( R_1/R_2 \) had that algorithm been used.

Secondly, consider the May 3, 1999 Oklahoma City tornado case, figures 6 and 7. The SNR is shown in Figure 6 and both spectrum width estimators (\( R_0/R_1 \) and \( R_1/R_2 \)) are shown in Figure 7. Notice that there are several regions, including where the the hook echo is occurring (40 km West and 20 km south of the radar), where the \( R_1/R_2 \) estimate (7(b)) is significantly smaller than the \( R_0/R_1 \) estimate (7(a)) and these locations correspond to large values of \( R_0/R_1 \), just as the simulations predict. Inspection of the spectra reveal very wide signals with the \( R_0/R_1 \) estimator fitting the data better. The point here is that the \( R_1/R_2 \) estimator is giving misleading information.

Ideally, improved spectrum width estimators would be incorporated into SZ-2. However,
Figure 6: SNR PPI from KOUN, May 3, 1999, 23:28:02 Z. \( N = 64 \).

Figure 7: Spectrum width PPI's from KOUN, May 3, 1999, 23:28:02 Z. \( N = 64 \).
as a stop-gap measure, the approach taken here is to attempt to detect when $R_1/R_2$ is saturating and censor it. Note that if a saturation failure mode is detected, we set $\hat{w}$ to the saturation level, rather than censoring. This is perhaps preferable since saturation does not occur very often, and when it does the saturated value conveys more information than just censoring the data. See Figure 8 for a cartoon of the censoring methodology. Note that the censoring scheme is independent of the Nyquist velocity with the only exception being the value that $\hat{w}$ is set to when designated as saturated. Here is the censoring scheme:

1. if $|\hat{R}_0|/|\hat{R}_1| > 2.7518$ then set $\hat{w} = 0.2615 \times v_a$ (saturated value): If this ratio is bigger than the saturated value, then the true spectrum width must be saturated (or close to it). This corresponds to the dark red region in Figure 8.

2. if $|\hat{R}_0|/|\hat{R}_1| \geq 1.3499$ and $|\hat{R}_1|/|\hat{R}_2| \leq 1.1618$ then censor $\hat{w}$: If the $R_0/R_1$ estimator says large but the $R_1/R_2$ says small, then censor. We do not really know what is going on in this case. It could either be a very narrow or very wide true spectrum width. This corresponds to the purple region in Figure 8.

3. if $|\hat{R}_0|/|\hat{R}_1| \geq 1.3499$ and $1.1618 \leq |\hat{R}_1|/|\hat{R}_2| \leq 2.7518$ then set $\hat{w} = 0.2615 \times v_a$ (saturated value): If the $R_0/R_1$ estimator says large but the $R_1/R_2$ says medium, then designate the data as saturated. This corresponds to the light red region in Figure 8.

The censoring algorithm was studied via simulation and the results are shown in Figures 9-17. An I&Q simulator, described by Frehlich and Yadlowsky (1994) was used to generate
10000 I&Q time series of length 64 with both a strong and weak trip, phase-coded using the SZ(8/64). Constant simulation parameters for this analysis were $\lambda = 10.5 \text{ cm}$, PRT = 780 $\mu$s, $v_1 = 0 \text{ m/s}$, SNR$_2 = 50$ dB, $v_2 = 0 \text{ m/s}$, $w_2 = 2 \text{ m/s}$ (where $v_1$ is the mean velocity of the strong trip and SNR$_2$, $v_2$ and $w_2$ are the weak trip signal-to-noise ratio, mean velocity, and spectrum width, resp.). Power ratios (PR) tested were 5 (Figures 9-11), 10 (Figures 12-14), and 20 dB (Figures 15-17), and true (simulation input) spectrum width varied from 0.25 to 16 $m/s$. In Figures 9, 12, and 15, 2-D histograms of true versus estimated spectrum width using the $R_1/R_2$ estimator is shown both before censoring, in the left panel, and after censoring in the right. The biases of the $R_1/R_2$ estimator before and after censoring, are shown in Figures 10, 13, and 16. Since each “gate” is designated as “good”, “saturated”, or “censored”, the designation breakdowns for varying true spectrum widths are shown in Figures 11, 14, and 17.

The results are positive with some issues to be resolved. The algorithm does a good job of labeling as censored or saturated when the true spectrum width is large, whether for 5, 10, or 20 dB PR. Ideally, large true spectrum widths should be designated as saturated, but having some labeled as censored is preferable to having them labeled as good. One problem is that for narrow true spectrum widths for all PR’s but especially at smaller PR’s, a small number of “gates” are incorrectly designated as censored or saturated. Even though the percentages are not large, because of the large numbers of gates with narrow spectrum widths in the operational setting, this will translate to a small but visible amount of speckle in censored spectrum width PPI’s. Now some of the estimates designated as censored/saturated were clearly of very poor quality. Looking at Figures 9 and 12, there are some large estimates for the very narrow true spectrum widths that have been censored/saturated. Labeled as censored would be OK, but the saturated designation is not desirable. Also visible in Figure 9 are gates with true spectrum widths in the 3 to 4 $m/s$ range and the censoring algorithm is designating as saturated. The reason for this is not known, currently, but will be studied in the future.

The bias plots show that the censoring scheme increases the bias for larger spectrum widths. This is expected however, since the censoring scheme pushes any estimate larger than $0.2615 \times v_a$ to $0.2615 \times v_a$, effectively shifting the distribution of estimates downward. For some values of true spectrum width, we see biases of as much as $1 \text{ m/s}$, but relative to the true spectrum width, this is not large, and certainly the estimated spectrum widths are still large.

1.3 Conclusions

While it would be preferable to improve the spectrum width estimators used in SZ-2, in the short term, it may be preferable to identify when the current algorithms are failing and censor the data to mitigate the problems. To that end, we propose a censoring algorithm to address problems in the short PRT $R_1/R_2$ estimator. It performs very well, in general. There are a couple of regimes in which the performance is not ideal, and these should be
Figure 9: True versus $R_1/R_2$ estimated spectrum width. $\lambda = 10.5\, cm$, PRT = 780 $\mu$s, $N = 64$, PR = 5 dB, SNR$_2$ = 50 dB, $w_2 = 2\, m/s$.

Figure 10: Bias of $R_1/R_2$ estimated spectrum width. $\lambda = 10.5\, cm$, PRT = 780 $\mu$s, $N = 64$, PR = 5 dB, SNR$_2$ = 50 dB, $w_2 = 2\, m/s$. 

12
Figure 11: Designation breakdown of censoring rules for varying true spectrum widths. 
\( \lambda = 10.5 \, \text{cm}, \text{PRT} = 780 \, \mu\text{s}, N = 64, \text{PR} = 5 \, \text{dB}, \text{SNR}_2 = 50 \, \text{dB}, w_2 = 2 \, \text{m/s}. \)

Figure 12: True versus \( R_1/R_2 \) estimated spectrum width. 
\( \lambda = 10.5 \, \text{cm}, \text{PRT} = 780 \, \mu\text{s}, N = 64, \text{PR} = 10 \, \text{dB}, \text{SNR}_2 = 50 \, \text{dB}, w_2 = 2 \, \text{m/s}. \)
Figure 13: Bias of $R_1/R_2$ estimated spectrum width. $\lambda = 10.5\, cm$, PRT = 780 $\mu$s, $N = 64$, PR = 10 dB, SNR$_2$ = 50 dB, $w_2 = 2\, m/s$.

Figure 14: Designation breakdown of censoring rules for varying true spectrum widths. $\lambda = 10.5\, cm$, PRT = 780 $\mu$s, $N = 64$, PR = 10 dB, SNR$_2$ = 50 dB, $w_2 = 2\, m/s$. 
Figure 15: True versus $R_1/R_2$ estimated spectrum width. $\lambda = 10.5\, cm$, PRT = 780 $\mu s$, $N = 64$, PR = 20 dB, SNR$_2$ = 50 dB, $w_2 = 2\, m/s$.

Figure 16: Bias of $R_1/R_2$ estimated spectrum width. $\lambda = 10.5\, cm$, PRT = 780 $\mu s$, $N = 64$, PR = 20 dB, SNR$_2$ = 50 dB, $w_2 = 2\, m/s$. 
addressed. Namely, for lower PR’s, narrow true spectrum widths are sometimes labeled as censored or saturated: some censored data is OK, but saturated is not desirable; and for 5 dB PR, there are also some gates with 3 to 4 m/s true spectrum width that are being designated as saturated. For large true spectrum widths, the performance is excellent. In short, the algorithm cleans up the larger true spectrum widths very well, but causes some problems for very narrow spectrum widths. However, it should be noted that the censoring scheme has not been finely tuned, and it may be that non-rectangularly shaped regions in Figure 8 would perform better. In particular, tuning the algorithm so that it performs less well for wide true spectrum widths, but better for narrow is probably preferable given that narrow occur much more frequently than wide.

2 Staggered PRT Clutter Filtering

2.1 Introduction

The staggered PRT technique is a range-velocity ambiguity mitigation scheme that has been described in other places (Zrnić and Mahapatra, 1985; Sachidananda et al., 1999; Torres et al., 2004). Integrating effective clutter filtering into the staggered PRT technique is proving challenging because of the difficult nature of unevenly spaced data. In this report, the staggered PRT clutter filtering technique as described in Torres et al. (2005) with small modifications as received in a white paper from the ROC is quantitatively evaluated via simulation techniques. This is done for a wide range of NEXRAD operational regimes.
Minimum Usable Mean Radial Velocity, $V_{\text{min}}$, in m/s | Required Ground Clutter Suppression Capability in dB (weather spectrum width = 4 m/s) |
---|---|
2 | 20 |
3 | 28 |
4 | 50 |

Table 1: The clutter suppression requirements for Doppler channel

| Weather spectrum width $w$ in m/s | Maximum allowable bias in reflectivity estimate (dB) |
---|---|
1 | 10 |
2 | 2 |
$\geq 3$ | 1 |

Table 2: Maximum allowable bias in reflectivity estimates due to clutter filter

2.2 Requirements

The WSR-88D System Specification (ROC, 2007, Excerpt 3.7.2.7), referred to here as WSS, describes two clutter models, so-called Clutter Models A and B. As indicated by the ROC, Clutter Model A was used, in which the clutter is modeled as a Gaussian random process with zero mean velocity and spectrum width a combination of the clutter width (0.1 m/s) and the spectrum width resulting from the antenna rotation rate. Thus a 0.28 m/s spectrum width is used in the clutter model. Also as dictated by the WSS, the radar transmit frequency used in the study is 2.85 GHz. To meet the requirement, the suppression in the reflectivity channel needs to be at least 30 dB. For the Doppler channel, the suppression requirements are listed in table 1. The suppression level is calculated as the ratio of the power of the input time series and that of the output time series, in dB.

Also dictated by the WSS are the errors, both biases and standard deviations, associated with the estimates. They can be broken into 2 pieces, one set defines the requirements when little to no clutter is present, and the other set when there is between 10 to 15 dB clutter to signal ratio (CSR). In this study, only the former set of requirements were evaluated. Namely:

- The bias of the reflectivity (equivalently SNR) should be less than or equal to the values in table 2 when the CSR $\leq -30$ dB, for all mean radial velocities, $v$. There does not appear to be a specification for the standard deviation of the reflectivity.

- The bias and standard deviation of $\hat{v}$ and $\hat{w}$ (spectrum width) should be less than 2 m/s for all $|v| \geq V_{\text{min}}$, for the suppression levels listed in table 1, input signal to noise ratio (SNR) of at least 20 dB, CSR $\leq -30$, and $w = 4$ m/s.
2.3 Methodology

One difficulty with applying the WSS to this particular clutter filter is that in many regimes, there really is not a well-defined meaning to $V_{\text{min}}$. When applying a high-pass filter, whether spectral (e.g. GMAP), or time-domain (e.g. 5-pole elliptic IIR) on evenly spaced time series (and thus a well defined Nyquist) there is only one interval in the Nyquist interval that is suppressed, somewhere around $-2$ to $2$ m/s. Because of the nature of staggered PRT, there is not 1 Nyquist but 2. In particular, in the spectral domain, because of the nature of the clutter filter algorithm, there are 5 intervals (which are referred to as the ‘fifths’ in this report) in which the clutter filter affects the spectrum. Thus there is some ambiguity as to how to fit the criterion detailed in the WSS to this filter. Now a related issue comes up even in the evenly spaced time series case since velocity aliasing must be considered. Because of velocity aliasing in the long PRT scans, biases on $\hat{\bar{Z}} (\hat{\bar{v}}, \text{and} \hat{\bar{w}}$ also, although not generally used) are incurred around $\pm 16n$ m/s as well as at $0$ m/s where $n$ in any integer. Analogously, perhaps the staggered PRT clutter filter should judge the ‘fifths’ in the same way as for the velocity interval around $0$ m/s. This may be “fair” with respect to the algorithm. On the other hand, if the Nyquist velocity of the staggered PRT algorithm is purported as 26.25 m/s, operators may not realize that the clutter filter will have deleterious effects for intervals around $\pm 10.5n$ m/s. From that standpoint, it may be better to judge the algorithm in the same way as the clutter filters for evenly spaced time series are. Namely, the clutter filter has to perform “well” on the entire Nyquist interval except around a small interval about zero. It may be very challenging for the staggered PRT clutter filter to satisfy the latter methodology. Both methodologies are discussed in this report. The former is referred to as the ‘relaxed’ methodology and the latter as the ‘stringent’ one.

With regards to the testing procedure, a simulator (Frehlich and Yadlowsky, 1994) was used to generate 10000 I&Q time series for all combinations of the parameter values in table 3.

In order to do this, the simulator was run using the PRT $T_u = T_2 - T_1$ and number of samples, $N = 5M/2$, and the subsequent time series were down-sampled. The time series were then processed using the method described in Torres et al. (2005) with small modifications as received in a white paper from the ROC, and with the further modification that the clutter suppression width, $q$, was fixed rather than be determined dynamically. This fixing of $q$ was done because that part of the algorithm was being modified by NSSL.

2.4 Results

Two types of plots are included in this report. The first set of plots, Figures 18-53 show the bias and standard deviation of clutter filtered SNR as a function of CSR with varying values for $T_1$, $M$, $v$ and $w$ (in order from slowest changing parameter, with regard to the order of the plots, to the fastest):

- $T_1$: 785 µs, 1000 µs, and 2000 µs
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>wavelength, $\lambda$</td>
<td>10.5 cm</td>
</tr>
<tr>
<td>PRT $T_1$ ($T_2 = 3/2 T_1$)</td>
<td>785, 1000, 2000 $\mu$s</td>
</tr>
<tr>
<td>$M$ (number of pulses)</td>
<td>16, 32, 64</td>
</tr>
<tr>
<td>Noise power $N$</td>
<td>$-80$ dB</td>
</tr>
<tr>
<td>Clutter CSR</td>
<td>$-30$ dB</td>
</tr>
<tr>
<td>Clutter spectrum width</td>
<td>0.28 $m/s$</td>
</tr>
<tr>
<td>SNR</td>
<td>20 $dB$</td>
</tr>
<tr>
<td>$v$</td>
<td>$-v_a, -0.98v_a, \ldots, 0.98v_a, v_a$ (where $v_a$ is the unambiguous velocity, $v = \lambda/(4(T_2 - T_1))$)</td>
</tr>
<tr>
<td>$w$</td>
<td>0.5, 1, 2, 4, 6, 8 $m/s$</td>
</tr>
<tr>
<td>clutter suppression width, $q$</td>
<td>3, 4, 5, 6 $^1$</td>
</tr>
</tbody>
</table>

Table 3: Maximum allowable bias in reflectivity estimates due to clutter filter

$^1$4 is the maximum value for $q$ when $M = 16$.

- $M$: 16, 32, and 64
- $v$: 0.4$v_a$ (on one of the ‘fifths’ where performance is likely to at its poorest outside of the interval between $-V_{\text{min}}$ and $V_{\text{min}}$) and 0.6 (not on any of the ‘fifths’).
- $w$: 1, 4 $m/s$

As can be seen from Figures 18-21, 30-33, and 42-45, the performance of the clutter filter for $M = 16$ is unacceptable. Often times the standard deviations are larger than 10 $m/s$. It should be noted that technically, these results still satisfy the WSS since nothing is stated about the standard deviation of the SNR. The biases are less than 2 dB in the cases shown, which satisfies the WSS. The suppression values for the $M = 32$ and 64 cases appear to satisfy the WSS in the cases where the mean velocity of the weather signal is not on the ‘fifths’. When the signal is on a ‘fifth’, the algorithm performs poorly and does not satisfy the WSS under the ‘stringent’ methodology, as can be seen, in Figures 22, 23, 26, 34, 38, 46, 50, 51. Additionally, when the $T_1$ is 2000 $\mu$s and $M = 64$, the suppression level for the narrower clutter notch widths do not satisfy either ‘stringent’ or ‘relaxed’ methodologies (Figures 52 and 53).
Figure 18: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.4v_o$ (on a ‘fifth’) and $w = 1$ m/s. The different lines represents different clutter filter widths, $q$. $T_1 = 785 \mu s$, $M = 16$, SNR = 20 dB.
Figure 19: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.4v_a$ (on a ‘fifth’) and $w = 4 \text{ m/s}$. The different lines represents different clutter filter widths, $q$. $T_1 = 785 \mu s$, $M = 16$, SNR = 20 dB.
Figure 20: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.6v_a$ (not on a ‘fifth’) and $w = 1$ m/s. The different lines represents different clutter filter widths, $q$. $T_1 = 785 \mu s$, $M = 16$, SNR = 20 dB.
Figure 21: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.6v_a$ (not on a ‘fifth’) and $w = 4 \text{ m/s}$. The different lines represents different clutter filter widths, $q$. $T_1 = 785 \mu\text{s}$, $M = 16$, SNR = 20 dB.
Figure 22: Bias and standard deviation of the clutter filtered SNR as a function of CSR for \(v = 0.4v_a\) (on a ‘fifth’) and \(w = 1\) m/s. The different lines represent different clutter filter widths, \(q\). \(T_1 = 785\) µs, \(M = 32\), SNR = 20 dB.
Figure 23: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.4v_0$ (on a ‘fifth’) and $w = 4$ m/s. The different lines represents different clutter filter widths, $q$. $T_1 = 785\,\mu s$, $M = 32$, SNR = 20 dB.
Figure 24: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.6v_a$ (*not on* a ‘fifth’) and $w = 1 \text{ m/s}$. The different lines represents different clutter filter widths, $q$. $T_1 = 785 \mu s$, $M = 32$, SNR = 20 dB.
Figure 25: Bias and standard deviation of the clutter filtered SNR as a function of CSR for \( v = 0.6v_a \) (not on a ‘fifth’) and \( w = 4 \) m/s. The different lines represents different clutter filter widths, \( q \). \( T_1 = 785 \mu s \), \( M = 32 \), SNR = 20 dB.
Figure 26: Bias and standard deviation of the clutter filtered SNR as a function of CSR for \(v = 0.4v_a\) (on a ‘fifth’) and \(w = 1\ m/s\). The different lines represent different clutter filter widths, \(q\). \(T_1 = 785\ \mu s\), \(M = 64\), SNR = 20 dB.
Figure 27: Bias and standard deviation of the clutter filtered SNR as a function of CSR for
$v = 0.4v_o$ (on a ‘fifth’) and $w = 4$ m/s. The different lines represents different clutter filter
widths, $q$. $T_1 = 785 \mu s$, $M = 64$, SNR = 20 dB.
Figure 28: Bias and standard deviation of the clutter filtered SNR as a function of CSR for \( v = 0.6v_a \) (not on a ‘fifth’) and \( w = 1 \) m/s. The different lines represents different clutter filter widths, \( q \). \( T_1 = 785 \mu s \), \( M = 64 \), SNR = 20 dB.
Figure 29: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.6v_a$ (not on a ‘fifth’) and $w = 4 \text{ m/s}$. The different lines represents different clutter filter widths, $q$. $T_1 = 785 \mu\text{s}$, $M = 64$, SNR = 20 dB.
Figure 30: Bias and standard deviation of the clutter filtered SNR as a function of CSR for \( v = 0.4v_a \) (\textit{on a ‘fifth’}) and \( w = 1 \text{ m/s} \). The different lines represents different clutter filter widths, \( q \). \( T_1 = 1000 \mu s, M = 16, \text{SNR} = 20 \text{ dB} \).
Figure 31: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.4v_a$ (on a ‘fifth’) and $w = 4 \, m/s$. The different lines represents different clutter filter widths, $q$. $T_1 = 1000 \, \mu s$, $M = 16$, SNR = 20 dB.
Figure 32: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.6v_a$ (not on a ‘fifth’) and $w = 1 \text{ m/s}$. The different lines represents different clutter filter widths, $q$. $T_1 = 1000 \mu s$, $M = 16$, SNR = 20 dB.
Figure 33: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.6v_a$ (not on a ‘fifth’) and $w = 4 \text{ m/s}$. The different lines represent different clutter filter widths, $q$. $T_1 = 1000 \mu s$, $M = 16$, SNR = 20 dB.
Figure 34: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.4v_a$ (on a ‘fifth’) and $w = 1$ m/s. The different lines represents different clutter filter widths, $q$. $T_1 = 1000\mu s$, $M = 32$, SNR = 20 dB.
Figure 35: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.4v_a$ (on a ‘fifth’) and $w = 4 \text{ m/s}$. The different lines represent different clutter filter widths, $q$. $T_1 = 1000 \mu s$, $M = 32$, SNR = 20 dB.
Figure 36: Bias and standard deviation of the clutter filtered SNR as a function of CSR for \( v = 0.6v_a \) (not on a ‘fifth’) and \( w = 1 \text{ m/s} \). The different lines represents different clutter filter widths, \( q \). \( T_1 = 1000 \text{ µs}, M = 32, \text{SNR}=20 \text{ dB} \).
Figure 37: Bias and standard deviation of the clutter filtered SNR as a function of CSR for \(v = 0.6v_a\) (not on a ‘fifth’) and \(w = 4\ m/s\). The different lines represents different clutter filter widths, \(q\). \(T_1 = 1000\ \mu s\), \(M = 32\), \(SNR = 20\ dB\).
Figure 38: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.4v_o$ (on a ‘fifth’) and $w = 1 \text{ m/s}$. The different lines represent different clutter filter widths, $q$. $T_1 = 1000 \mu s$, $M = 64$, SNR = 20 dB.
Figure 39: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.4v_a$ (on a ‘fifth’) and $w = 4$ m/s. The different lines represents different clutter filter widths, $q$. $T_1 = 1000 \mu s$, $M = 64$, SNR = 20 dB.
Figure 40: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.6v_a$ (not on a ‘fifth’) and $w = 1 \text{ m/s}$. The different lines represents different clutter filter widths, $q$. $T_1 = 1000 \mu s$, $M = 64$, SNR = 20 dB.
Figure 41: Bias and standard deviation of the clutter filtered SNR as a function of CSR for \( v = 0.6v_a \) (not on a ‘fifth’) and \( w = 4 \) m/s. The different lines represents different clutter filter widths, \( q \). \( T_1 = 1000 \mu s \), \( M = 64 \), SNR = 20 dB.
Figure 42: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.4v_a$ (on a ‘fifth’) and $w = 1 \text{ m/s}$. The different lines represents different clutter filter widths, $q$. $T_1 = 2000 \mu\text{s}$, $M = 16$, SNR = 20 dB.
Figure 43: Bias and standard deviation of the clutter filtered SNR as a function of CSR for \( v = 0.4v_0 \) (on a ‘fifth’) and \( w = 4 \text{ m/s} \). The different lines represents different clutter filter widths, \( q \). \( T_1 = 2000 \mu s \), \( M = 16 \), SNR = 20 dB.
Figure 44: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.6v_a$ (not on a ‘fifth’) and $w = 1 \text{ m/s}$. The different lines represents different clutter filter widths, $q$. $T_1 = 2000 \mu s$, $M = 16$, SNR = 20 dB.
Figure 45: Bias and standard deviation of the clutter filtered SNR as a function of CSR for \( v = 0.6v_a \) (not on a ‘fifth’) and \( w = 4 \text{ m/s} \). The different lines represents different clutter filter widths, \( q \). \( T_1 = 2000 \mu s \), \( M = 16 \), SNR = 20 dB.
Figure 46: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.4v_a$ (on a ‘fifth’) and $w = 1 \text{ m/s}$. The different lines represents different clutter filter widths, $q$. $T_1 = 2000 \mu s$, $M = 32$, SNR = 20 dB.
Figure 47: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.4v_a$ (on a ‘fifth’) and $w = 4 \text{ m/s}$. The different lines represent different clutter filter widths, $q$. $T_1 = 2000 \mu s$, $M = 32$, SNR = 20 dB.
Figure 48: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.6v_a$ (not on a ‘fifth’) and $w = 1 \text{ m/s}$. The different lines represents different clutter filter widths, $q$. $T_1 = 2000 \mu s$, $M = 32$, SNR = 20 dB.
Figure 49: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.6v_a$ (*not on* a ‘fifth’) and $w = 4$ m/s. The different lines represents different clutter filter widths, $q$. $T_1 = 2000\,\mu s$, $M = 32$, SNR = 20 dB.
Figure 50: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.4v_o$ (on a ‘fifth’) and $w = 1 \text{ m/s}$. The different lines represents different clutter filter widths, $q$. $T_1 = 2000 \mu s$, $M = 64$, SNR = 20 dB.
Figure 51: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.4v_a \ (\text{on a ‘fifth’})$ and $w = 4 \text{ m/s}$. The different lines represents different clutter filter widths, $q$. $T_1 = 2000 \mu s$, $M = 64$, SNR = 20 dB.
Figure 52: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.6v_a$ (not on a ‘fifth’) and $w = 1\ m/s$. The different lines represent different clutter filter widths, $q$. $T_1 = 2000\ \mu s$, $M = 64$, SNR = 20 dB.
Figure 53: Bias and standard deviation of the clutter filtered SNR as a function of CSR for $v = 0.6v_a$ (*not on a ‘fifth’*) and $w = 4 \text{ m/s}$. The different lines represents different clutter filter widths, $q$. $T_1 = 2000 \mu s$, $M = 64$, SNR = 20 dB.
The second type of plots, Figures 54-89, show bias (top panel) and standard deviations (bottom panel) of SNR, \( \hat{v} \), or \( \hat{w} \) (depending on the plot) as a function of the mean velocity of the weather signal. Different lines on the plot represent different values of \( w \). Each plot has fixed values for \( T_1 \), \( M \), CSR, and \( q \). For all the conditions tested, this creates a total of 720 plots, and thus only a subset of the plots are included in this report. In particular, only plots for the following combinations are shown: CSR = -30 dB, \( M = 32, 64 \), and \( q = 3, 6 \).

For the case where \( T_1 = 785 \mu s \) and \( M = 32 \) (Figures 54 and 55), the SNR satisfies the WSS for both \( q = 3, 6 \), although the performance for the standard deviations is very poor and erratic for \( w \) narrow. Recall that the WSS only specifies the bias for SNR when \( w = 4 \, m/s \). The \( V_{min} \) appears to be quite large for both \( q = 3 \) and 6, (\( \sim 5 \) and 10 \, m/s, respectively). Also, the biases for narrow spectrum widths (e.g. \( w = 1 \, m/s \)) when \( q = 6 \) approach 2.5 \, m/s with standard deviations going well above 10 \, m/s. For the velocity estimate (Figures 56 and 57), the \( q = 3 \) case satisfies the WSS, but the \( q = 6 \) case only satisfies under the ‘relaxed’ methodology, incurring biases at the ‘fifths’ that are slightly over 2 \, m/s and standard deviations exceeding 5 \, m/s. But it should be noted that about 1/2 of the Nyquist interval would fall outside of the WSS specification if the ‘relaxed’ methodology is used. It should also be noted that even for the \( q = 3 \) case, even though the WSS is satisfied, the performance is quite poor for narrow or wide values of \( w \) (e.g. 1, 2, and 6 \, m/s) on the ‘fifths’. The spectrum width estimates (Figures 58 and 59), as it turns out, are quite reasonable for both \( q = 3 \) and 6. The biases and standard deviations for the \( q = 6 \) case have some values that exceed 2 \, m/s but not by much when \( w = 4 \, m/s \).

For the case where \( T_1 = 785 \mu s \) and \( M = 64 \) (Figures 60 and 61), the SNR satisfies the WSS for both \( q = 3, 6 \), although the performance for the standard deviations is very poor for \( w \) narrow. Also, the standard deviations for narrow spectrum widths (e.g. \( w = 1 \, m/s \)) when \( q = 6 \) go well above 10 \, m/s. For the velocity estimate (Figures 62 and 63), both the \( q = 3 \) and 6 cases satisfy the WSS. It should also be noted that for both \( q = 3 \) and 6 cases, even though the WSS is satisfied, the performance is quite poor for narrow values of \( w \) on the ‘fifths’, incurring significant biases and in the case of \( q = 6 \), significant standard deviations. The spectrum width estimates (Figures 64 and 65), are quite reasonable for both \( q = 3 \) and 6.

For the \( T_1 = 1000 \mu s \), \( M = 32 \) case, the SNR (Figures 66 and 67) satisfies the WSS, although the standard deviations are quite bad for narrow spectrum widths. For the velocity estimates (Figures 68 and 69), the WSS is satisfied for the \( q = 3 \) case but the performance is poor for narrow spectrum widths. The WSS is not satisfied for \( q = 6 \). The spectrum widths (Figures 70 and 71) appear to perform fairly well, satisfying the WSS.

For the \( T_1 = 1000 \mu s \), \( M = 64 \) case, the SNR (Figures 72 and 73) satisfies the WSS, although the standard deviations are quite bad for narrow spectrum widths. For, the velocity estimates (Figures 74 and 75), the WSS is satisfied for the both \( q = 3 \) and 6 cases but the performance is poor for narrow and wide spectrum widths. The spectrum widths (Figures 76 and 77) appear to perform fairly well, satisfying the WSS.
Figure 54: Bias and standard deviation of the clutter filtered SNR as a function of $v$. The different lines represent different values for $w$. $T_1 = 785 \mu s$, $M = 32$, SNR = 20 dB, CSR = $-30$ dB, $q = 3$.

For the $T_1 = 2000 \mu s$, $M = 32$ case, the SNR (Figures 78 and 79) satisfies the WSS, although the standard deviations are quite bad for narrow spectrum widths. For the velocity estimates (Figures 80 and 81), the WSS is not satisfied for the $q = 3$ or 6 cases. The spectrum widths (Figures 82 and 83) satisfy the WSS but performance is very poor for larger spectrum widths.

For the $T_1 = 2000 \mu s$, $M = 64$ case, the SNR (Figures 84 and 85) satisfies the WSS, although the standard deviations are quite bad for narrow spectrum widths. For the velocity estimates (Figures 86 and 87), the WSS is not satisfied for the $q = 3$ or 6 cases. The spectrum widths (Figures 88 and 89) satisfy the WSS but performance is very poor for larger spectrum widths.
Figure 55: Bias and standard deviation of the clutter filtered SNR as a function of $v$. The different lines represents different values for $w$. $T_1 = 785 \mu s$, $M = 32$, SNR = 20 dB, CSR = $-30$ dB, $q = 6$. 
Figure 56: Bias and standard deviation of the clutter filtered V as a function of $v$. The different lines represents different values for $w$. $T_1 = 785\,\mu s$, $M = 32$, SNR = 20 dB, CSR = −30 dB, $q = 3$. 
Figure 57: Bias and standard deviation of the clutter filtered V as a function of $v$. The different lines represent different values for $w$. $T_1 = 785 \mu s$, $M = 32$, $\text{SNR} = 20$ dB, $\text{CSR} = -30$ dB, $q = 6$. 
Figure 58: Bias and standard deviation of the clutter filtered $W$ as a function of $v$. The different lines represents different values for $w$. $T_1 = 785 \, \mu s$, $M = 32$, SNR = 20 dB, CSR = $-30$ dB, $q = 3$. 
Figure 59: Bias and standard deviation of the clutter filtered $W$ as a function of $v$. The different lines represents different values for $w$. $T_1 = 785 \mu s$, $M = 32$, SNR = 20 dB, CSR = $-30$ dB, $q = 6$. 
Figure 60: Bias and standard deviation of the clutter filtered SNR as a function of $v$. The different lines represents different values for $w$. $T_1 = 785 \mu s$, $M = 64$, SNR = 20 dB, CSR = −30 dB, $q = 3$. 
Figure 61: Bias and standard deviation of the clutter filtered SNR as a function of $v$. The different lines represent different values for $w$. $T_1 = 785\ \mu s$, $M = 64$, $SNR = 20\ dB$, $CSR = -30\ dB$, $q = 6$. 
Figure 62: Bias and standard deviation of the clutter filtered V as a function of $v$. The different lines represent different values for $w$. $T_1 = 785\, \mu s$, $M = 64$, SNR = 20 dB, CSR = $-30$ dB, $q = 3$. 
Figure 63: Bias and standard deviation of the clutter filtered V as a function of $v$. The different lines represents different values for $w$. $T_1 = 785 \mu s$, $M = 64$, SNR = 20 dB, CSR = $-30$ dB, $q = 6$. 
Figure 64: Bias and standard deviation of the clutter filtered $W$ as a function of $v$. The different lines represent different values for $w$. $T_1 = 785 \mu s$, $M = 64$, SNR = 20 dB, CSR = $-30$ dB, $q = 3$. 
Figure 65: Bias and standard deviation of the clutter filtered $W$ as a function of $v$. The different lines represent different values for $w$. $T_1 = 785\,\mu s$, $M = 64$, SNR = 20 dB, CSR = $-30$ dB, $q = 6$. 
Figure 66: Bias and standard deviation of the clutter filtered SNR as a function of \( v \). The different lines represent different values for \( w \). \( T_1 = 1000 \, \mu s, M = 32, \text{SNR} = 20 \, \text{dB}, \text{CSR} = -30 \, \text{dB}, \, q = 3 \).
Figure 67: Bias and standard deviation of the clutter filtered SNR as a function of $v$. The different lines represents different values for $w$. $T_1 = 1000 \mu s$, $M = 32$, SNR = 20 dB, CSR = $-30$ dB, $q = 6$. 
Figure 68: Bias and standard deviation of the clutter filtered V as a function of $v$. The different lines represent different values for $w$. $T_1 = 1000 \mu s$, $M = 32$, SNR = 20 dB, CSR = $-30$ dB, $q = 3$. 
Figure 69: Bias and standard deviation of the clutter filtered V as a function of $v$. The different lines represents different values for $w$. $T_1 = 1000 \mu s$, $M = 32$, SNR = 20 dB, CSR = $-30$ dB, $q = 6$. 
Figure 70: Bias and standard deviation of the clutter filtered $W$ as a function of $v$. The different lines represents different values for $w$. $T_1 = 1000\mu s$, $M = 32$, SNR = 20 dB, CSR = $-30$ dB, $q = 3$. 
Figure 71: Bias and standard deviation of the clutter filtered $W$ as a function of $v$. The different lines represents different values for $w$. $T_1 = 1000 \mu s$, $M = 32$, $SNR = 20$ dB, $CSR = -30$ dB, $q = 6$. 
Figure 72: Bias and standard deviation of the clutter filtered SNR as a function of $v$. The different lines represent different values for $w$. $T_1 = 1000 \mu s$, $M = 64$, SNR = 20 dB, CSR = $-30$ dB, $q = 3$. 
Figure 73: Bias and standard deviation of the clutter filtered SNR as a function of $v$. The different lines represents different values for $w$. $T_1 = 1000 \mu s$, $M = 64$, SNR = 20 dB, CSR = $-30$ dB, $q = 6$. 
Figure 74: Bias and standard deviation of the clutter filtered V as a function of $v$. The different lines represent different values for $w$. $T_1 = 1000 \mu s$, $M = 64$, SNR = 20 dB, CSR = $-30$ dB, $q = 3$. 
Figure 75: Bias and standard deviation of the clutter filtered V as a function of $v$. The different lines represents different values for $w$. $T_1 = 1000 \, \mu s$, $M = 64$, SNR = 20 dB, CSR = $-30$ dB, $q = 6$. 
Figure 76: Bias and standard deviation of the clutter filtered $W$ as a function of $v$. The different lines represent different values for $w$. $T_1 = 1000 \, \mu s$, $M = 64$, $\text{SNR} = 20$ dB, $\text{CSR} = -30$ dB, $q = 3$. 

$p_{79}$
Figure 77: Bias and standard deviation of the clutter filtered $W$ as a function of $v$. The different lines represent different values for $w$. $T_1 = 1000 \mu s$, $M = 64$, SNR = 20 dB, CSR = $-30$ dB, $q = 6$. 
Figure 78: Bias and standard deviation of the clutter filtered SNR as a function of $v$. The different lines represents different values for $w$. $T_1 = 2000\mu s$, $M = 32$, SNR = 20 dB, CSR = $-30$ dB, $q = 3$. 
Figure 79: Bias and standard deviation of the clutter filtered SNR as a function of $v$. The different lines represents different values for $w$. $T_1 = 2000 \mu s$, $M = 32$, SNR = 20 dB, CSR = $-30$ dB, $q = 6$. 
Figure 80: Bias and standard deviation of the clutter filtered V as a function of $v$. The different lines represent different values for $w$. $T_1 = 2000 \mu s$, $M = 32$, SNR = 20 dB, CSR = $-30$ dB, $q = 3$. 
Figure 81: Bias and standard deviation of the clutter filtered $V$ as a function of $v$. The different lines represent different values for $w$. $T_1 = 2000 \mu s$, $M = 32$, SNR = 20 dB, CSR = $-30$ dB, $q = 6$. 
Figure 82: Bias and standard deviation of the clutter filtered $W$ as a function of $v$. The different lines represents different values for $w$. $T_1 = 2000\,\mu s$, $M = 32$, SNR = 20 dB, CSR = −30 dB, $q = 3$. 
Figure 83: Bias and standard deviation of the clutter filtered $W$ as a function of $v$. The different lines represent different values for $w$. $T_1 = 2000 \mu s$, $M = 32$, SNR = 20 dB, CSR = $-30$ dB, $q = 6$. 
Figure 84: Bias and standard deviation of the clutter filtered SNR as a function of $v$. The different lines represents different values for $w$. $T_1 = 2000\,\mu s$, $M = 64$, SNR = 20 dB, CSR = $-30$ dB, $q = 3$. 
Figure 85: Bias and standard deviation of the clutter filtered SNR as a function of $v$. The different lines represents different values for $w$. $T_1 = 2000 \mu s$, $M = 64$, SNR = 20 dB, CSR = −30 dB, $q = 6$. 
Figure 86: Bias and standard deviation of the clutter filtered V as a function of $v$. The different lines represent different values for $w$. $T_1 = 2000\,\mu s$, $M = 64$, SNR = 20 dB, CSR = $-30$ dB, $q = 3$. 

89
Figure 87: Bias and standard deviation of the clutter filtered $V$ as a function of $v$. The different lines represent different values for $w$. $T_1 = 2000\,\mu s$, $M = 64$, $\text{SNR} = 20\,\text{dB}$, $\text{CSR} = -30\,\text{dB}$, $q = 6$. 
Figure 88: Bias and standard deviation of the clutter filtered W as a function of $v$. The different lines represents different values for $w$. $T_1 = 2000\,\mu s$, $M = 64$, $SNR = 20$ dB, $CSR = -30$ dB, $q = 3$. 
Figure 89: Bias and standard deviation of the clutter filtered $W$ as a function of $v$. The different lines represents different values for $w$. $T_1 = 2000\,\mu s$, $M = 64$, SNR = 20 dB, CSR = $-30$ dB, $q = 6$. 
2.5 Conclusions

Figure 90 is an attempt to summarize the results from the analysis, most of which is included in this report. There are several import caveats:

- Errors due to clutter residue have not yet been analyzed.
- All results are preliminary and, in particular, configurations that are green or yellow could get downgraded upon further analysis.
- Green values do not necessarily indicate “good” performance.

From the summary table (Figure 90) we see that only for $M = 64$ and $T_1 = 785$ and 1000 $\mu s$ is the WSS satisfied, thus far, for all clutter notch widths, $q$. If only $q = 3$ is used, then $M = 32$ also satisfies the WSS for the same values of $T_1$. However, for $M = 32$, the performance of the SNR and velocity estimators for narrow (but nominal) spectrum widths is very poor. In fact, on closer inspection, although $M = 64$ and $T_1 = 785$, 1000 $\mu s$, and $q = 3, 4, 5, 6$ satisfies the WSS (thus far), the performance of the SNR and velocity estimates are poor for narrow spectrum widths.

In conclusion, the performance of the estimates of the clutter filtered time series has serious deficiencies, even in cases that satisfy the WSS. This says more about the relaxed nature of the WSS rather than the quality of the clutter filter. In general, the performance of the clutter filter is poor for “narrower” spectrum widths. However, it should be noted that “narrower” in this context are very common spectrum width values, e.g. 1 and 2 $m/s$. This clutter filter does not appear to be ready for operational use, at least with the regimes that were tested in this analysis.
Figure 90: Summary table for results of staggered PRT clutter filter analysis. The columns represent different PRT’s (T₁ is listed), and values of M. The rows correspond to different values of q. Some results appearing in the table are not discussed in the report. All results are preliminary and, in particular, configurations that are green or yellow could get downgraded upon further analysis (errors due to clutter residue have not yet been analyzed). Green = satisfies WSS, yellow = mostly satisfies WSS, red = does not satisfy WSS, black = N/A. Green values do not necessarily indicate “good” performance.


3 REC Updates

The primary task for the Radar Echo Classifier (REC) included testing the recent improvements over long rainfall accumulation times as recommended by the NEXRAD Technical Advisory Committee (TAC) in the fall of 2006. The REC improvements included corrections and improvements to the Anomalous Propagation Detection Algorithm (APDA), and the addition of the Precipitation Detection Algorithm (PDA) as described in Hubbert et al. (2005). For the APDA, the calculation of the SPIN variable was corrected and the SIGN variable was removed. These two corrections to the APDA alone resulted in significant improvement in rainfall estimates. The PDA was added to REC and the logic within EPRE was modified to accommodate this change. The motivation to add PDA to the REC was to increase the skill in identifying conditions of ground clutter mixed with precipitation echoes, a particularly difficult problem. Details are described in the NCAR 2005i Annual Report Hubbert et al. (2005).

The improved REC was tested using several data sets with long accumulation times (up to 32 hours). The Archive II (A2) data sets obtained from the National Climatic Data Center (NCDC) were recommended by the Radar Operations Center (ROC) and included examples of strong contamination of rainfall estimates by ground clutter (reported by field offices) as well as pure weather cases. The data sets included stratiform and convective rain, as well as snow, from 9 cases at 8 different radar sites and are summarized in Table 4. The convective precipitation presents the challenge of more variability in the moment data, which can be mistakenly classified as clutter by automated algorithms. There were examples of clutter mixed with precipitation echoes which demonstrated the benefit of adding the PDA to REC. The data also included examples containing censored Doppler data to test the updated REC’s robustness to missing radial velocity and spectrum width. This combination provided a rigorous test of the improved REC algorithm to ensure both improved clutter contamination removal and retention of valid weather data. The A2 data sets were processed using the Common Operations and Development Environment (CODE) using the operational version of the REC in addition to the improved REC, allowing direct comparisons between the two.

The results were presented to the TAC as a decision briefing in March of 2007. The TAC approved the presented science and recommended that the improved REC move forward in the implementation process. The next steps would have been the Independent Validation and Verification (IV and V) and presentation to the SREC. The results presented to the TAC are summarized below.

3.1 Results

The first result is from Houston, Texas (KHGX). In this case, extensive AP clutter appeared after the passage of a convective line through the area. The base reflectivity and radial velocity after the passage of the convection are shown in Fig. 91. It can be seen that
Table 4: Data sets

<table>
<thead>
<tr>
<th>Station</th>
<th>Date</th>
<th>Precip Type</th>
<th>Accumulation Time (hr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>KBOX</td>
<td>07/14/03</td>
<td>Stratiform Rain</td>
<td>8</td>
</tr>
<tr>
<td>KMLB</td>
<td>03/25/92</td>
<td>Convective Rain</td>
<td>6</td>
</tr>
<tr>
<td>KHGX</td>
<td>10/18/94</td>
<td>Conv/Strat Rain</td>
<td>9</td>
</tr>
<tr>
<td>KLWX</td>
<td>04/29/03</td>
<td>Conv/Strat Rain</td>
<td>4</td>
</tr>
<tr>
<td>KLWX</td>
<td>01/25/04</td>
<td>Snow</td>
<td>13</td>
</tr>
<tr>
<td>KBUF</td>
<td>01/11/03</td>
<td>Snow</td>
<td>24</td>
</tr>
<tr>
<td>KCCX</td>
<td>02/16/03</td>
<td>Snow</td>
<td>24</td>
</tr>
<tr>
<td>KFTG</td>
<td>03/18/03</td>
<td>Snow</td>
<td>32</td>
</tr>
<tr>
<td>KSRX</td>
<td>02/24/07</td>
<td>Conv/Strat Rain</td>
<td>24</td>
</tr>
</tbody>
</table>

the trailing stratiform rain echoes are mixed with AP clutter echoes. A comparison of 9 hour rainfall accumulation between the operational REC (left panel) and the updated REC (right panel) is shown in Fig. 92. Strong ground clutter contamination that is mixed with valid rain estimates is evident in the operational REC (demarked by the ovals in Fig. 92). The overestimates of rainfall due to the clutter exceed 5 inches in these areas. The rainfall estimates from the updated REC do not suffer from these ground clutter contaminations indicating that the algorithm was able to identify more of the ground clutter than the operational REC. Importantly, the rain estimates outside of the contaminated regions are similar for the operational and updated REC. This indicates that the updated algorithm is not removing more data from valid echoes than the operational version. This is a particularly important result because of the convective nature of the precipitation. An expanded view of the 9 hour accumulation is presented in Fig. 93. It can be seen that the operational REC has numerous false rainfall accumulations due to ground clutter contamination near the radar. The maximum false rainfall exceeds 6 inches. The updated REC removes most of the clutter contamination while preserving the legitimate rainfall accumulations.

The next example is from Sterling, Virginia (KLWX) and included mostly AP ground clutter contamination with some light rain in the northern part of the radar domain. The base reflectivity (dBZ) and radial velocity (kt) are shown in Fig. 94. The main features are strong AP ground clutter in the nearest 50 nautical miles of the radar and a small band of precipitation that moved from the west to the east passing about 100 nautical miles to the north. This example is a good test because the precipitation echo to the northwest of KLWX is censored in the Doppler scan due to overlaid echo. Without radial velocity information the separation of clutter and weather is more difficult. A plot of the 4 hour rainfall accumulation is shown in Fig. 95 with a zoom level to highlight the ground clutter region in the region near the radar. The operational REC allows strong ground clutter to accumulate false rainfall totals with a maximum over 9 inches. The updated REC does not remove all of the clutter contamination, but results in a significant reduction in false accumulations. Fig. 96 shows...
Figure 91: *Base reflectivity (dBZ) and radial velocity (kt) from KHGX.*

Figure 92: *Nine hour rainfall accumulation (in) from KHGX.*
the same 4 hour accumulation as Fig. 95, but with a zoom level to highlight the precipitation north of KLWX. It can be seen that the overall patterns are similar between the operational and updated versions of REC. One noticeable difference is that some rain removed by the operational REC is restored by the upgraded version, highlighted in Fig. 96 with white ovals. There are also a small number of pixels in which the updated REC produced less rain accumulation than the operational version.

The next example is a severe squall line observed with the Fort Smith, Arkansas radar (KSRX). Fig. 97 shows the base reflectivity scans as the squall line is approaching at 15:22 UTC (left panel) and after it has passed at 19:28 UTC (right panel). Isolated strong ground clutter contamination is apparent near the radar. This case is a strict test of the REC given the convective nature of the weather and the fact that the clutter is mixed with weather for long periods of time. The results of a 24 hour rainfall accumulation are shown in Fig. 98. The operational REC shows numerous large erroneous rainfall totals resulting from undetected ground clutter contamination. The color scale in Fig. 98 represents any amount greater than 15 inches as the pink color, and in fact the maximum erroneous rainfall using the operational REC exceeded 40 inches. The updated REC removes the majority of the spurious rainfall and the maximum is just over 4 inches, an improvement of about a factor of 10 over the operational REC. Importantly, the valid rainfall accumulations between the two versions are nearly identical with only a few pixels differing.

The next example is a snow storm from State College Pennsylvania (KCCX). The base reflectivity (dBZ) and radial velocity (kt) are shown in Fig. 99. In this case there is shallow, but strong, stratiform snow over a wide region. These shallow stratiform systems are very
Figure 94: Base reflectivity (dBZ) and radial velocity (kt) from KLWX.

Figure 95: Nine hour rainfall accumulation (in) from KLWX.
Figure 96: *Nine hour rainfall accumulation (in)* from KLWX.

Figure 97: *Base reflectivity (dBZ) at 15:22 UTC and 19:28 UTC* from KSRX.
Figure 98: Twenty four hour rainfall accumulation totals from KSRX.

sensitive to false clutter detections because there are little or no data at higher elevation scans to use to fill in data within the Hybrid Scan Reflectivity (HSR). The clutter filters were invoked at all bins resulting in there being no noticeable ground clutter contamination. The 24 hour snow accumulations are shown in Fig. 100. The updated REC results in higher snowfall amounts in many regions of the domain. This is an indication the updated REC removed less weather echoes than the operational REC.

3.2 Summary of Results

Several long integration time data sets recommended by the ROC were used to demonstrate the improvements in ground clutter mitigation afforded through implementation of the updated REC. These data sets represented difficult tests for automated algorithms such as the REC, including: Convective storms, mixed clutter and weather and censored Doppler data due to overlaid echo. The results show that updating the REC by correcting the APDA and adding the PDA results in significantly reduced contamination of precipitation estimates by ground clutter. At the same time the upgraded REC was shown to retain the legitimate precipitation, and in some instances resulted in less removal of valid data than the operational REC.

Based on this study the NEXRAD Technical Advisory Committee (TAC) approved of the scientific support of the updated REC and recommended the changes move forward in the implementation process.
Figure 99: Base reflectivity (dBZ) and radial velocity (kt) from KCCX.

Figure 100: Twenty four hour snowfall accumulations from KCCX.
References


Functional Description
for the
Single Polarization
Clutter Mitigation Decision (CMD) system
for the
ORDA
Version 4.1
Prepared for:
WSR-88D Radar Operations Center
US National Weather Service
Norman, Oklahoma
Prepared by:
The National Center for Atmospheric Research
Boulder, Colorado
Mike Dixon
Scott Ellis
November 17, 2007
1. Overview..................................................................................................................1

2. Revision history .......................................................................................................1
   2.1 Version 1 summary
   2.2 Version 2 changes
   2.3 Version 3 changes
   2.4 Version 4 changes

3. Applicability ............................................................................................................2

4. Algorithmic description .............................................................................................3

5. Feature fields...........................................................................................................3
   5.1 Overview
   5.2 TDBZ - DBZ texture
   5.3 SPIN - DBZ spin change
   5.4 CPA - clutter phase alignment
   5.5 Handling censored data in the feature fields
   5.6 Converting feature fields to interest fields

6. Flow charts ................................................................................................................6
   6.1 Computing the clutter flag
   6.2 Using the CMD clutter flag as a dynamic bypass map with SZ2
   6.3 Applying clutter filter with no range unfolding

7. Constants, known quantities and parameters .......................................................8
   7.1 Constants
   7.2 Parameters
   7.3 Interest map parameters
   7.4 Membership function plots

8. Algorithm objects ....................................................................................................10
   8.1 Pulse object
   8.2 Beam object
   8.3 Interest Map object

9. Computing the individual feature fields .................................................................18
   9.1 Computing DBZ Texture - TDBZ
   9.2 Computing DBZ SPIN
   9.3 Computing CPA
   9.4 Computing clutter probability and clutter flag
   9.5 Spatial in-fill filter for the CMD clutter flag field
   9.6 NEXRAD spike filter
Functional Description for the Clutter Mitigation Decision (CMD) system for the ORDA
Version 4.1

1. Overview

The goal of CMD is to provide guidance on where to apply an adaptive clutter filter such as Sigmet’s Gaussian Model Adaptive Processing (GMAP) filter. CMD will set a clutter probability flag for each gate in a radar beam at which (a) clutter is considered likely and (b) an adaptive filter such as GMAP is unlikely to remove weather power in error. The intention is that GMAP would then only be applied at the gates which have been flagged in this way.

2. Revision history

2.1 Version 1 summary

The Version 1 Functional Description was delivered to the ROC in January 2006.

Version 1 used the following feature fields:
- TDBZ - reflectivity texture
- SPIN - reflectivity spin
- SDVE - standard deviation of velocity
- Clutter Ratio Wide

Version 1 used a computation kernel which spanned 5 consecutive azimuths, requiring that beam information be buffered to provide access to the data over the kernel. This buffering requirement has implications for the implementation in the RVP8.

2.2 Version 2 changes

Version 2 differs from version 1 in the following ways:
- Clutter Ratio Wide has been dropped as a feature field.
- Clutter Phase Alignment (CPA) has been added as a feature field.
- The computation kernel is now limited to a single beam, so that beam buffering is no longer necessary.
- The length of the computation kernel is set individually for each field, rather than a single kernel length parameter being applied to all fields.
- A spatial (in-fill) filter is applied to the CMD flag field to fill in short gaps in range surrounded by significant clutter.
The feature fields used in version 2 are:
- TDBZ - reflectivity texture
- SPIN - reflectivity spin
- SDVE - standard deviation of velocity
- CPA - clutter phase alignment

The SDVE field is ‘turned off’ in this version, by setting the weight to 0, but it kept in the algorithm should it prove useful during tuning on further data sets.

2.3 Version 3 changes

Version 3 differs from version 2 in the following ways:
- SDVE has been dropped completely.
- Clutter Ratio Narrow and Wide have been dropped completely.
- Only SNR is used for thresholding.
- TDBZ and SPIN are combined into a single interest field, which is the maximum of the individual interest fields.

The feature fields used in version 3 are:
- TDBZ - reflectivity texture
- SPIN - reflectivity spin
- Max of TDBZ interest and SPIN interest
- CPA - clutter phase alignment

2.4 Version 4 changes

Version 4 differs from version 3 in the following ways:
- the SPIN computation has been changed to be symmetrical with respect to range. In other words, SPIN should have the same value whether it is computed with increasing range or decreasing range.
- the thresholds have been modified to handle this change in the formulation of SPIN.
- Version 4.1 has a modified interest map for SPIN.

3. Applicability

This version of CMD is intended for use with data which has the following characteristics:
- constant PRT;
- single polarization;
• non-phase coded;
• at least 16 samples per beam.

The most likely application of this version would be to generate a ‘dynamic clutter map’ using the long-PRT data of the first 2 tilts in a Volume Coverage Pattern (VCP). This dynamic clutter map would detect situations such as AP which are missed by the static clutter map. This dynamic would provide clutter information to algorithms such as SZ2.

The algorithm could also be used stand-alone with any constant-PRT data for deciding where to safely apply a clutter map.

## 4. Algorithmic description

In this document, the algorithm is described using a number of techniques: (a) flow charts; (b) step-by-step logic in text and graphics and (c) C code examples. C code is considered an accurate and unambiguous way to present the information because the RVP8 implementation will be coded in C.

The intention is that the flow charts and other explanations should allow an understanding of the algorithm by those readers who are not familiar with C code.

Please note that the C code was derived from a C++ application which has been tested on real data cases. The code compiles, but has not actually been run and tested in this form.

## 5. Feature fields

### 5.1 Overview

CMD uses a fuzzy logic approach to combine the information from a number of so-called ‘Feature Fields’ into a single decision-making field.

Some of the fields (TDBZ, SPIN) are computed using information from a small region in range (the ‘kernel’) around the gate. Other fields, such as CPA, are computed from information at the gate only.

In the version current, the kernel is 1-dimensional, along the beam in range. The length of the kernel is specified individually for each relevant feature field.
5.2 TDBZ - DBZ texture

Regions of clutter exhibit rapidly changing reflectivity from gate to gate. DBZ texture is a measure of the gate-to-gate change in value of dBZ. It is computed over a kernel. First, the gate-to-gate dbz difference is computed for each gate, and then squared. TDBZ is the mean of this squared difference over the kernel.

5.3 SPIN - DBZ spin change

Like TDBZ, SPIN is designed to find regions of rapidly changing reflectivity. SPIN is based on the number of significant changes in sign in the gradient of the reflectivity field over a kernel. A gate is considered to be a spin change point if (a) the mean absolute differences in dBZ, from the previous gate to this gate, and this gate to the next gate, exceeds a threshold \( \text{SPIN\_THRESHOLD} \) and (b) the sign of the slope has changed since the last spin point. The number of spin points is computed and then expressed as a percentage of the number of possible spin changes.

5.4 CPA - clutter phase alignment

In clutter the phase of each pulse in the time series for a particular gate is almost constant since the clutter does not move significantly and is at a constant distance from the radar. In noise, the phase from pulse to pulse is random. In weather, the phase from pulse to pulse will vary depending on the velocity of the targets within the illumination volume.

CPA is computed as the length of the cumulative phasor vector, normalized by the sum of the magnitudes for the pulses. CPA is computed at a single gate. It ranges from 0 to 1. In clutter, CPA is typically above 0.95. In weather, CPA is often close to 0, but increases in weather having a velocity close to 0 and a narrow spectrum width. In noise, CPA is typically less than 0.05.

5.5 Handling censored data in the feature fields

In computing the feature fields, moments data which has been censored are ignored.

In addition to the censoring which occurs upstream of the algorithm, CMD sets the clutter flag at a gate to FALSE if the SNR value at the date is below a set threshold. This prevents the application of a clutter filter at that gate, which improves efficiency.
5.6 Converting feature fields to interest fields

The values in the feature fields are dependent on what the field physically represents. In order to combine fields in a meaningful manner, feature fields are first converted into so-called ‘Interest Fields’ by applying a so-called membership transfer function to the feature field. The function converts feature values into interest values between 0 and 1. These interest fields may then be manipulated by the fuzzy logic system to produce the final decision-making field - in this case, the clutter flag.
6. Flow charts

6.1 Computing the clutter flag

- check for new pulse
  - NO
  - are pulses available to form new beam?
    - NO
    - group pulses into beam
    - for each gate in beam
      - compute SNR, dBZ, CPA
      - for each gate in beam
        - compute dbzDiffSq, dbzSpinFlag
        - for each gate in beam
          - compute TDBZ, SPIN
          - convert TDBZ, SPIN, CPA to interest values
          - compute MAX(TDBZ interest, SPIN interest)
          - for each gate in beam
            - set clutterFlag = FALSE
            - is (SNR > SNR_THRESHOLD)?
              - NO
              - combine interest values into clutterProbability
              - NO
              - is (clutterProbability > PROB_THRESHOLD)?
                - NO
                - apply IN-FILL filter to clutter flag field
                - apply clutter filter based on clutter flag
                - YES
              - YES
              - set clutterFlag = TRUE
            - YES
          - YES
        - YES
      - YES
    - YES
  - YES
- compute TDBZ, SPIN
- convert TDBZ, SPIN, CPA to interest values
- compute MAX(TDBZ interest, SPIN interest)
- for each gate in beam
  - NO
  - is (SNR > SNR_THRESHOLD)?
    - NO
    - set clutterFlag = FALSE
    - YES
  - YES
- set clutterFlag = TRUE
- apply IN-FILL filter to clutter flag field
- apply clutter filter based on clutter flag
6.2 Using the CMD clutter flag as a dynamic bypass map with SZ2

for each indexed beam in long-prt PPI

compute CMD clutter flag at each gate

Write clutter flag field to buffer, indexed by azimuth

for each indexed beam in short-prt PPI

Read clutter flag field from buffer, for this azimuth

Using clutter flag fields as dynamic bypass map, compute SZ2

apply NEXRAD spike filter to filtered fields

6.3 Applying clutter filter with no range unfolding

for each gate in beam

is clutterFlag set?

NO

YES

apply clutter filter

apply NEXRAD spike filter to filtered fields
7. Constants, known quantities and parameters

7.1 Constants

The following values are constant for a particular implementation of CMD:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Example value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_GATES</td>
<td>integer</td>
<td>2048</td>
<td>Maximum number of gates in a beam.</td>
</tr>
</tbody>
</table>

7.2 Parameters

These values are set at the start of the algorithm:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Suggested value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGATES_KERNEL_TDBZ</td>
<td>integer</td>
<td>9</td>
<td>Number of gates in kernel for TDBZ</td>
</tr>
<tr>
<td>NGATES_KERNEL_SPIN</td>
<td>integer</td>
<td>11</td>
<td>Number of gates in kernel for SPIN</td>
</tr>
<tr>
<td>SPIN_THRESHOLD</td>
<td>floating point</td>
<td>6.5</td>
<td>Difference threshold used in computation of SPIN, in dB.</td>
</tr>
<tr>
<td>SNR_THRESHOLD</td>
<td>floating point</td>
<td>3.0</td>
<td>Signal-to-noise ratio threshold for censoring on total power, in dB.</td>
</tr>
<tr>
<td>PROB_THRESHOLD</td>
<td>floating point</td>
<td>0.5</td>
<td>Probability threshold. If the clutter probability exceeds the threshold the clutter flag is set TRUE.</td>
</tr>
<tr>
<td>CMD_MAX_GATES_INFILLED</td>
<td>integer</td>
<td>3</td>
<td>Maximum number of gates to be in-filled in the CMD flag field</td>
</tr>
</tbody>
</table>
7.3 Interest map parameters

The following parameters define the interest maps for the CMD fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Weight</th>
<th>Membership function definition points</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDBZ</td>
<td>0.0</td>
<td>(20, 0), (40, 1)</td>
</tr>
<tr>
<td>SPIN</td>
<td>0.0</td>
<td>(15, 0), (30, 1)</td>
</tr>
<tr>
<td>MAX_TDBZ_SPIN</td>
<td>1.0</td>
<td>(0, 0), (1, 1) (identity)</td>
</tr>
<tr>
<td>CPA</td>
<td>1.01</td>
<td>(0.6, 1), (0.9, 1)</td>
</tr>
</tbody>
</table>

7.4 Membership function plots

The following plots show the membership functions graphically.

1Interest value

Feature value

<table>
<thead>
<tr>
<th>Interest value</th>
<th>Feature value</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>TDBZ</td>
</tr>
<tr>
<td></td>
<td>(20, 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(40, 1)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>SPIN</td>
</tr>
<tr>
<td></td>
<td>(15, 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(30, 1)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>CPA</td>
</tr>
<tr>
<td></td>
<td>(0.6, 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.9, 1)</td>
<td></td>
</tr>
</tbody>
</table>
8. Algorithm objects

In Object-Oriented (OO) programming, an object is a useful abstraction which groups together some data and some defined behavior associated with that data. There are a number of aspects of the CMD algorithm which are well described by the Object model. These items do not necessarily need to be coded as formal objects. However they can be thought of as objects for this discussion.

This section defines a number of objects which are used by the algorithm. It is useful to define these first and then refer to them later in the document.

8.1 Pulse object

The Pulse object consists of header data and an array of floating point IQ data.

8.1.1 Pulse implementation example

The Pulse object may be coded as a C structure.

```c
typedef struct {
    /* meta data */
    int nGates; /* number of gates */
    double time; /* time in secs and fractions from 1 Jan 1970 */
    double prt; /* pulse repetition time (secs) */
    double el; /* elevation angle (deg) */
    double az; /* azimuth angle (deg) */

    /* IQ data */

    float iq[MAX_GATES * 2];
} Pulse;
```

8.2 Beam object

A Beam is made up of an array of Pulses. The Pulses are often referred to as ‘samples’. There are `nSamples` pulses in a Beam.

A Beam object consists of header data, along with an array of pulses and arrays for storing the moments (Z, V, W) and relevant feature fields and interest fields.
8.2.1 Beam implementation example

The Beam object may be coded as a C structure.

```c
/* Beam implementation example */

typedef struct {

    int nSamples; /* number of pulse samples in beam */
    int nGates;  /* number of gates */
    double time; /* time for the center pulse of beam */
    double prt; /* pulse repetition time (secs) */
    double el; /* elevation angle for center of beam (deg) */
    double az; /* azimuth angle for center of beam (deg) */

    Pulse **pulses; /* Pointers to pulses with IQ data */

    /* arrays for moments */
    float *snr; /* SNR in dB */
    float *dbz; /* reflectivity in dBZ */
    float *vel; /* velocity in m/s */
    float *width; /* spectrum width in m/s */

    float *dbzf; /* filtered dBZ */

    /* arrays for intermediate fields */
    float *dbzDiffSq;
    float *dbzSpinFlag;

    /* arrays for CMD fields */
    float *tdbz;
    float *spin;
    float *cpa;

    /* array for clutter probability */
    float *clutterProb;

    /* array for clutter decision flag:
```
int *clutterFlag;

} Beam;

/*
* create a beam
*
* (a) set meta data
* (b) allocate array memory
* (c) initialize values to censored or 0
*
* Pulse memory is not owned by the beam. It should be
* allocated and freed elsewhere.
*/

Beam *create_beam(int nSamples,
                  int nGates,
                  double time,
                  double prt,
                  double el,
                  double az,
                  Pulse **pulses) {

  int isample, igate;

  /* allocate space for beam */
  Beam *beam = (Beam *) malloc(sizeof(Beam));

  /* set meta data */
  beam->nSamples = nSamples;
  beam->nGates = nGates;
  beam->time = time;
  beam->prt = prt;
  beam->el = el;
  beam->az = az;

  /* allocate space for arrays */
  beam->pulses = (Pulse **) malloc(nSamples * sizeof(Pulse *));

  beam->snr = (float *) malloc(nGates * sizeof(float));
  beam->dbz = (float *) malloc(nGates * sizeof(float));
  beam->vel = (float *) malloc(nGates * sizeof(float));
  beam->width = (float *) malloc(nGates * sizeof(float));
beam->dbzf = (float *) malloc(nGates * sizeof(float));
beam->dbzDiffSq = (float *) malloc(nGates * sizeof(float));
beam->dbzSpinFlag = (float *) malloc(nGates * sizeof(float));
beam->tdbz = (float *) malloc(nGates * sizeof(float));
beam->spin = (float *) malloc(nGates * sizeof(float));
beam->cpa = (float *) malloc(nGates * sizeof(float));

beam->clutterProb = (float *) malloc(nGates * sizeof(float));
beam->clutterFlag = (int *) malloc(nGates * sizeof(int));

/* initialize arrays */

for (isample = 0; isample < nSamples; isample++) {
    beam->pulses[isample] = pulses[isample];
}

for (igate = 0; igate < nGates; igate++) {
    beam->snr[igate] = CENSORED;
    beam->dbz[igate] = CENSORED;
    beam->vel[igate] = CENSORED;
    beam->width[igate] = CENSORED;
    beam->dbzf[igate] = CENSORED;
    beam->dbzDiffSq[igate] = CENSORED;
    beam->dbzSpinFlag[igate] = CENSORED;
    beam->tdbz[igate] = CENSORED;
    beam->spin[igate] = CENSORED;
    beam->cpa[igate] = CENSORED;

    beam->clutterProb[igate] = CENSORED;
    beam->clutterFlag[igate] = 0;
}

return beam;
}

/*
 * free memory associated with a beam
 */

void free_beam(Beam *beam) {

    free(beam->pulses);
    free(beam->snr);
    free(beam->dbz);
    free(beam->vel);
    free(beam->width);
    free(beam->dbzf);
    free(beam->dbzDiffSq);
    free(beam->dbzSpinFlag);
}
free(beam->tdbz);
free(beam->spin);
free(beam->cpa);
free(beam->clutterProb);
free(beam->clutterFlag);
free(beam);

}  

8.3  Interest Map object

8.3.1  Overview

CMD uses fuzzy logic to combine information from different fields. Prior to computing the fuzzy combination, the feature fields are converted to so-called ‘interest fields’ by applying a piece-wise linear membership function which converts the feature values to an ‘interest value’ between 0 and 1.

The figure above shows an example of a membership transfer function. For each feature value there exists a unique interest value: interest = f(feature).

The piece-wise linear function can be defined by specifying the coordinates of the points a, b, c, d, and e. The function is linear between these points, and constant outside the range of the first and last points.

For example, the membership function above may be defined by the array of points in the following table.

<table>
<thead>
<tr>
<th>Point</th>
<th>x-coord</th>
<th>y-coord</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>-1.5</td>
<td>0.0</td>
</tr>
<tr>
<td>b</td>
<td>-1.0</td>
<td>0.5</td>
</tr>
</tbody>
</table>
For all feature values below -1.5, the interest value is constant, at 0.0, while for all feature values above 1.5 the interest value is constant at 1.0.

Also associated with the Interest Map is the **weight** which will be applied to the interest field when the individual fields are combined to form a weighted-mean interest field.

### 8.3.2 Interest Map implementation example

The Interest Map object can be efficiently implemented as a lookup table, where the lookup only occurs for feature values between the minimum and maximum specified values. For feature values less than the minimum specified, the interest value is constant at the value for the first point. For feature values above the maximum specified, the interest value is constant at the value for the last point.

```c
/******************************/
* Interest Map implementation example
*
* Implemented as a lookup table.
*/

/* struct for points which define interest map */

typedef struct {
    double feature;
    double interest;
} ImPoint;

/* interest map struct */

typedef struct {
    int nPoints;
    ImPoint *points;
    double minFeature;
    double maxFeature;
    double deltaFeature;
    float *lut;
} InterestMap;

/*
 * Initialize map by passing in the points which define the map
 * along with the weight for the map as a whole
 */
```

<table>
<thead>
<tr>
<th>Point</th>
<th>x-coord</th>
<th>y-coord</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>-0.5</td>
<td>0.25</td>
</tr>
<tr>
<td>d</td>
<td>0.5</td>
<td>0.25</td>
</tr>
<tr>
<td>e</td>
<td>1.5</td>
<td>1.0</td>
</tr>
</tbody>
</table>
* Returns NULL on failure.
*/

InterestMap *create_map(ImPoint *pts,
   int npts) {

    int ii, jj;
    double slope;
    InterestMap *map = NULL;

    if (npts < 2) {
        /* error - must have at least 2 points to define function */
        return NULL;
    }

    /* allocate map itself */
    map = malloc(sizeof(InterestMap));

    /* clear map */
    memset(map, 0, sizeof(InterestMap));

    /* set meta data */
    map->nPoints = npts;

    /* allocate space for points which define map */
    map->points = malloc(npts * sizeof(ImPoint));

    /* copy point data */
    memcpy(map->points, pts, npts * sizeof(ImPoint));

    /* allocate space for lookup table */
    map->lut = (float*) malloc(N_MAP_LOOKUP * sizeof(float));

    /* compute min, max and delta */
    map->minFeature = map->points[0].feature;
    map->maxFeature = map->points[npts-1].feature;
    map->deltaFeature =
        (map->maxFeature - map->minFeature) / (N_MAP_LOOKUP - 1.0);

    /* compute slope for initial map segment */
    jj = 1;
    slope = ((map->points[jj].interest - map->points[jj-1].interest) /
(map->points[jj].feature - map->points[jj-1].feature));

/* load lookup table */
for (ii = 0; ii < N_MAP_LOOKUP; ii++) {
    double interest;
    double val = map->minFeature + ii * map->deltaFeature;
    if ((val > map->points[jj].feature) && (jj < npts-1)) {
        /* move along by one segment */
        jj++;
        slope = ((map->points[jj].interest - map->points[jj-1].interest) / (map->points[jj].feature - map->points[jj-1].feature));
    }
    /* compute interest, add to lookup table */
    interest = map->points[jj-1].interest + (val - map->points[jj-1].feature) * slope;
    map->lut[ii] = interest;
}

return map;

/* create map */

/* free map memory */

void free_map(InterestMap *map) {
    free(map->points);
    free(map->lut);
    free(map);
}

/*
 * look up interest given the feature value
 */

double getInterest(InterestMap *map, double feature)
{
    int jj = (int) floor((feature - map->minFeature) / map->deltaFeature + 0.5);
    if (jj < 0) {
        jj = 0;
    } else if (jj > N_MAP_LOOKUP-1) {
        jj = N_MAP_LOOKUP-1;
    }
    return map->lut[jj];
}
Set up interest maps for each feature field

```c
InterestMap *tdbzMap;
InterestMap *spinMap;
InterestMap *cpaMap;

ImPoint tdbzPoints[2] = {{20.0, 0.0}, {40.0, 1.0}};
ImPoint spinPoints[3] = {{15.0, 0.0}, {30.0, 1.0}};
ImPoint cpaPoints[2] = {{0.6, 0.0}, {0.9, 1.0}};
```

```c
void create_interest_maps() {

    tdbzMap = create_map(tdbzPoints, 2);
    spinMap = create_map(spinPoints, 3);
    cpaMap = create_map(cpaPoints, 2);
}
```

9. Computing the individual feature fields

9.1 Computing DBZ Texture - TDBZ

TDBZ is a measure of the gate-to-gate change in value of the dBZ values. It is computed over the kernel.

First, the squared differences in dBZ values from gate to gate are computed. Then, the mean value of these squared differences over the kernel is computed.

\[
TDBZ = \left( \sum_{i}^{\text{ngates}} (dBZ_{i,j} - dBZ_{i-1,j})^2 \right) / N
\]

9.1.1 TDBZ implementation example

First, we need to compute the squared difference in dBZ from gate to gate in the beam.

```c
/*
 * Prepare for computing TDBZ by computing
 * the gate-to-gate squared difference in Dbz
 */

void prepare_tdbz(Beam *beam) {

    int igate;
```
/*
 * compute the squared difference in dBZ from gate to gate in
 * the beam.
 */

for (igate = 1; igate < beam->nGates; igate++) {

    float prevDbz = beam->dbz[igate-1];
    float dbz = beam->dbz[igate];

    if (prevDbz != CENSORED && dbz != CENSORED) {
        float dbzDiff = dbz - prevDbz;
        beam->dbzDiffSq[igate] = dbzDiff * dbzDiff;
    }
}

/* use gate 1 value for gate 0 */
beam->dbzDiffSq[0] = beam->dbzDiffSq[1];

Then, we compute the mean squared difference over the kernel, setting the value for the center
beam of the beam queue.

void compute_tdbz(Beam *beam) {

    int igate;
    double sum = 0.0;
    double nn = 0.0;
    double mean = 0.0;

    for (igate = 0; igate < beam->nGates; igate++) {

        int startGate, endGate;
        int jbeam, jgate;

        /*
         * compute the start and end index, checking to make sure we
         * do not go off either end
         */

        startGate = igate - NGATES_KERNEL_TDBZ/2;
        if (startGate < 0) {
            startGate = 0;
        }

        endGate = igate + NGATES_KERNEL_TDBZ/2;
if (endGate > beam->nGates - 1) {
    endGate = beam->nGates - 1;
}

/*
 * iterate over the kernel
 * ignore censored gates
 */

for (jgate = startGate; jgate <= endGate; jgate++) {
    if (beam->dbzDiffSq[jgate] != CENSORED) {
        sum += beam->dbzDiffSq[jgate];
        nn += 1.0;
    }
} /* jgate */

if (nn > 0) {
    mean = sum / nn;
    beam->tdbz[igate] = mean;
}

} /* igate */

} /* compute_tdbz() */

9.2 Computing DBZ SPIN

SPIN is a measure of how frequently the slope of dBZ changes sign with range.

See the following figure as an example. The slope changes sign at points 1, 2, 3, 4 and 5.

First we compute the spin change at each of these points. The spin change is defined as the mean of the absolute difference in reflectivity between the previous gate and this gate, and this gate and the next gate.
So the spin change for each of the points marked above is:

\[
\begin{align*}
\text{spin}_\text{change}(1) &= \frac{\text{abs}(A) + \text{abs}(B)}{2} \\
\text{spin}_\text{change}(2) &= \frac{\text{abs}(C) + \text{abs}(D)}{2} \\
\text{spin}_\text{change}(3) &= \frac{\text{abs}(D) + \text{abs}(E)}{2} \\
\text{spin}_\text{change}(4) &= \frac{\text{abs}(F) + \text{abs}(G)}{2} \\
\text{spin}_\text{change}(5) &= \frac{\text{abs}(G) + \text{abs}(H)}{2}
\end{align*}
\]

A point with a change in sign is only counted if the spin change exceeds a set threshold.

### 9.2.1 SPIN implementation example

First, we set a flag at a gate if an inflection point exists. This flag field will then be used later to compute SPIN.

```c
/******************************************
* SPIN implementation example
* Compute SPIN feature field
*/

/*
* Prepare for computing spin by setting the spin flag for
* each gate
*/

void prepare_spin(Beam *beam) {
    int igate;

    /* first set spin flag for each gate */
    for (igate = 1; igate < beam->nGates; igate++) {
        float dbzPrev = beam->dbz[igate-1];
        float dbzThis = beam->dbz[igate];
        float dbzNext = beam->dbz[igate+1];

        if (dbzPrev != CENSORED &&
            dbzThis != CENSORED &&
            dbzNext != CENSORED) {
            beam->dbzSpinFlag[igate] = 0;

            float prevDiff = dbzThis - dbzPrev;
            float nextDiff = dbzNext - dbzThis;

            if (prevDiff * nextDiff < 0) { /* sign change */
                float spinChange = (fabs(prevDiff) + fabs(nextDiff)) / 2.0;
                if (spinChange > SPIN_THRESHOLD) {
```

```c
```
beam->dbzSpinFlag[igate] = 1;
} else {
   beam->dbzSpinFlag[igate] = 0;
}
} /* if (beam->dbz[igate] != CENSORED ... */
} /* igate */

/* use nearest values for end gates */
beam->dbzSpinFlag[0] = beam->dbzSpinFlag[1];
beam->dbzSpinFlag[beam->nGates-1] = beam->dbzSpinFlag[beam->nGates-2];
}

Then, we compute the SPIN as a percentage over the kernel, setting the value in the center beam of the beam queue.

void compute_spin(Beam *beam) {
    double sum = 0.0;
    double nn = 0.0;
    double mean = 0.0;
    int igate;

    for (igate = 0; igate < beam->nGates; igate++) {
        int startGate, endGate;
        int jgate;

        /*
        * compute the start and end index, checking to make sure we
        * do not go off either end
        */
        startGate = igate - NGATES_KERNEL_SPIN/2;
        if (startGate < 0) {
            startGate = 0;
        }
        endGate = igate + NGATES_KERNEL_SPIN/2;
        if (endGate > beam->nGates - 1) {
            endGate = beam->nGates - 1;
        }

        /*
        * iterate over the kernel
        * ignore censored gates
        */
for (jgate = startGate; jgate <= endGate; jgate++) {
    float spinFlag = beam->dbzSpinFlag[jgate];
    if (spinFlag != CENSORED) {
        sum += spinFlag;
        nn += 1.0;
    }
} /* jgate */

if (nn > 0) {
    mean = sum / nn;
    beam->spin[igate] = mean * 100.0;
}

} /* igate */

} /* compute_spin */

9.3 Computing CPA

The following figure demonstrates how CPA is computed.

![Phasor Diagram with CPA Calculation](image)

CPA is computed as the length of the combined phasor vector, normalized by the sum of the pulse magnitudes.

9.3.1 CPA implementation example

```c
void compute_cpa(Beam *beam) {
```

```c
    Noise or weather with non-zero velocity
    Weather mixed with clutter
    Adding I and Q for a gate in a phasor diagram
```

```c
}
```
int igate, isample;

for (igate = 0; igate < beam->nGates; igate++) {
    Pulse *pulse = beam->pulses[igate];
    double sumMag = 0.0;
    double sumI = 0.0, sumQ = 0.0;
    double phasorLen;

    for (isample = 0; isample < beam->nSamples; isample++) {
        float ii = pulse->iq[isample * 2];
        float qq = pulse->iq[isample * 2 + 1];
        sumI += ii;
        sumQ += qq;
        sumMag += sqrtf(ii * ii + qq * qq);
    } /* isample */

    phasorLen = sqrtf(sumI * sumI + sumQ * sumQ);
    beam->cpa[igate] = phasorLen / sumMag;
} /* igate */

} /* compute_cpa */

9.4 Computing clutter probability and clutter flag

To compute the clutter probability:

- At each gate:
  - convert each CMD field into an interest value.
  - compute maxTdbzSpinInterest = MAX(TDBZ interest, SPIN interest).
  - compute clutterProbability, as a weighted mean of the CPA interest and the maxTdbzSpinInterest.
  - if clutterProbability exceeds PROB_THRESHOLD, set clutterFlag to TRUE. Otherwise, set clutterFlag to FALSE.

9.4.1 Clutter probability implementation example

/***************************************************************************/
* Clutter probability implementation example
* *
*   (a) compute clutter probability
*   (b) set the clutter flag
* /

void compute_clut_prob(Beam *beam) {
    int igate;
float tdbz, spin, cpa;

double tdbzInterest, spinInterest;
double maxTdbzSpinInterest;
double cpaInterest;

double clutterProb;
int clutterFlag;

for (igate = 0; igate < beam->nGates; igate++) {
    /* get the feature fields */
    tdbz = beam->tdbz[igate];
    spin = beam->spin[igate];
    cpa = beam->cpa[igate];

    /* convert feature fields to interest values */
    tdbzInterest = getInterest(tdbzMap, tdbz);
    spinInterest = getInterest(spinMap, spin);
    cpaInterest = getInterest(cpaMap, cpa);

    /* compute max of tdbz and spin interest */
    maxTdbzSpinInterest = MAX(tdbzInterest, spinInterest);

    /* compute weighted sum */
    double sum = 0.0;
    double sumWt = 0.0;

    sum =
        maxTdbzSpinInterest * MAX_TDBZ_SPIN_WEIGH +
        cpaInterest * CPA_WEIGHT;

    sumWt = MAX_TDBZ_SPIN_WEIGH + CPA_WEIGHT;

    clutterProb = sum / sumWt;

    if (clutterProb > PROB_THRESHOLD) {
        clutterFlag = 1;
    } else {
        clutterFlag = 0;
    }

    beam->clutterProb[igate] = clutterProb;
    beam->clutterFlag[igate] = clutterFlag;
9.5 Spatial in-fill filter for the CMD clutter flag field

When the CMD flag field is computed, sometimes small gaps exist in range. If a clutter filter is applied in such a ‘spotty’ manner, it can lead to speckles in the weather field.

It appears unlikely that applying a clutter filter across these gaps will cause serious harm, since the adjacent gates will be filtered anyway. Filling in the gaps leads to a smoother result.

The following figure shows the circumstance under which this filter is applied.

![Diagram showing the in-fill filter application](image)

The in-fill filter is designed to fill in gaps of the type shown above. Specifically, it will fill in the gaps of the following type:

- 1 un-flagged gate between adjacent flagged gates;
- 2 un-flagged gates with at least 2 flagged gates on either side;
- 3 un-flagged gates with at least 3 flagged gates on either side.

9.5.1 In-fill filter implementation example

```c
void applyCmdInfillFilter(Beam *beam)
{
    int *countSet = (int *) malloc(beam->nGates * sizeof(int));
    int *countNot = (int *) malloc(beam->nGates * sizeof(int));

    /*
     * compute the running count of gates which have the flag set and
     * those which do not
     * Go forward through the gates, counting up the number of gates set
     * or not set and assigning that number to the arrays as we go.
     */
```
int nSet = 0;
int nNot = 0;
int igate, jgate;

for (igate = 0; igate < beam->nGates; igate++) {
    if (beam->clutterFlag[igate]) {
        nSet++;
        nNot = 0;
    } else {
        nSet = 0;
        nNot++;
    }
    countSet[igate] = nSet;
    countNot[igate] = nNot;
}

/*
 * Go in reverse through the gates, taking the max non-zero
 * values and copying them across the set or not-set regions.
 * This makes all the counts equal in the gaps and set areas.
 */

for (igate = beam->nGates - 2; igate >= 0; igate--) {
    if (countSet[igate] != 0 &&
        countSet[igate] < countSet[igate + 1]) {
        countSet[igate] = countSet[igate + 1];
    }
    if (countNot[igate] != 0 &&
        countNot[igate] < countNot[igate + 1]) {
        countNot[igate] = countNot[igate + 1];
    }
}

/*
 * fill in gaps
 */

for (igate = 1; igate < beam->nGates - 1; igate++) {

    /*
     * is the gap small enough?
     */
    nNot = countNot[igate];
    if (nNot > 0 && nNot <= CMD_MAX_NGATES_INFILLED) {

        /*
        * is it surrounded by regions at least as large as the gap?
        */
int minGateCheck = igate - nNot;
if (minGateCheck < 0) {
    minGateCheck = 0;
}
int maxGateCheck = igate + nNot;
if (maxGateCheck > beam->nGates - 1) {
    maxGateCheck = beam->nGates - 1;
}

int nAdjacentBelow = 0;
for (jgate = igate - 1; jgate >= minGateCheck; jgate--) {
    nSet = countSet[jgate];
    if (nSet != 0) {
        nAdjacentBelow = nSet;
        break;
    }
} /* jgate */

int nAdjacentAbove = 0;
for (jgate = igate + 1; jgate <= maxGateCheck; jgate++) {
    nSet = countSet[jgate];
    if (nSet != 0) {
        nAdjacentAbove = nSet;
        break;
    }
} /* jgate */

int minAdjacent = nAdjacentBelow;
minAdjacent = MIN(minAdjacent, nAdjacentAbove);

if (minAdjacent >= nNot) {
    beam->clutterFlag[igate] = 1;
}
} /* igate */

free(countSet);
free(countNot);

9.6 NEXRAD spike filter

The NEXRAD spike filter is designed to remove spikes of reflectivity 1 or 2 gates wide. This helps to remove speckle in the output fields.

Clearly the ROC is aware of how the NEXRAD filter works and how it might be implemented. Nevertheless, this implementation is included for completeness.
9.6.1 NEXRAD spike filter implementation

/*****************************/
* NEXRAD spike filter implementation example
* This routine filters the reflectivity data according to the
* NEXRAD specification DV1208621F, section 3.2.1.2.2, page 3-15.
* The algorithm is stated as follows:
* Clutter detection:
* The nth bin is declared to be a point clutter cell if its power value
* exceeds those of both its second nearest neighbors by a threshold
* value TCN. In other words:
* if P(n) exceeds TCN * P(n-2)
* and P(n) exceeds TCN * p(n+2)
* where
* TCN is the point clutter threshold factor, which is always
* greater than 1, and typically has a value of 8 (9 dB)
* P(n) if the power sum value for the nth range cell
* n is the range gate number
* Clutter censoring:
* The formulas for censoring detected strong point clutter in an
* arbitrary array A via data substitution are as follows. If the nth
* range cell is an isolated clutter cell (i.e., it is a clutter cell but
* neither of its immediate neighboring cells is a clutter cell) then the
* replacement scheme is as follows:
* Replace A(n-1) with A(n-2)
* Replace A(n) with 0.5 * A(n-2) * A(n+2)
* Replace A(n+1) with A(n+2)
* If the nth and (n+1)th range bins constitute an isolated clutter pair,
* the bin replacement scheme is as follows:
* Replace A(n-1) with A(n-2)
* Replace A(n) with A(n+2)
* Replace A(n+1) with A(n+3)
* Replace A(n+2) with A(n+3)
* Note that runs of more than 2 successive clutter cells cannot occur
* because of the nature of the algorithm.

NCAR. November 17, 2007 29
void applyNexradSpikeFilter(Beam *beam)
{
    int ii;

    /*
     * set clutter threshold
     */
    double tcn = 9.0;

    /*
     * loop through gates
     */
    for (ii = 2; ii < beam->nGates - 3; ii++) {
        /*
         * check for clutter at ii and ii + 1
         */
        int this_gate = 0, next_gate = 0;

        if ((beam->dbzf[ii] - beam->dbzf[ii - 2]) > tcn &&
            (beam->dbzf[ii] - beam->dbzf[ii + 2]) > tcn) {
            this_gate = 1;
        }
        if ((beam->dbzf[ii + 1] - beam->dbzf[ii - 1]) > tcn &&
            (beam->dbzf[ii + 1] - beam->dbzf[ii + 3]) > tcn) {
            next_gate = 1;
        }

        if (this_gate) {
            if (!next_gate) {
                /*
                 * only gate ii has clutter, substitute accordingly
                 */

                beam->dbzf[ii - 1] = beam->dbzf[ii - 2];
                beam->dbzf[ii + 1] = beam->dbzf[ii + 2];
                if (beam->dbzf[ii - 2] == CENSORED ||
                    beam->dbzf[ii + 2] == CENSORED) {
                    beam->dbzf[ii] = CENSORED;
                } else {
                    beam->dbzf[ii] = beam->dbzf[ii - 2];
                }
            }
        }
    }
} else {

    /*
     * both gate ii and ii+1 has clutter, substitute accordingly
     */

    beam->dbzf[ii - 1] = beam->dbzf[ii - 2];
    beam->dbzf[ii] = beam->dbzf[ii - 2];
    beam->dbzf[ii + 1] = beam->dbzf[ii + 3];
    beam->dbzf[ii + 2] = beam->dbzf[ii + 3];

} /* ii */
#include <stdlib.h>
#include <string.h>
#include <math.h>

#ifndef MAX
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#endif

#define MAX_GATES 2048
#define N_MAP_LOOKUP 10001

int NGATES_KERNEL_TDBZ = 9;
int NGATES_KERNEL_SPIN = 11;

float SPIN_THRESHOLD = 6.5;
float SNR_THRESHOLD = 3.0;
float PROB_THRESHOLD = 0.5;

float TDBZ_WEIGHT = 0.0;
float SPIN_WEIGHT = 0.0;
float MAX_TDBZ_SPIN_WEIGHT = 1.0;
float CPA_WEIGHT = 1.01;

int CMD_MAX_NGATES_INFILLED = 3;
float CENSORED = -9999;

typedef struct {

/* meta data */

int nGates; /* number of gates */
double time; /* time in secs and fractions from 1 Jan 1970 */
double prt; /* pulse repetition time (secs) */
double el; /* elevation angle (deg) */
double az; /* azimuth angle (deg) */

/* IQ data */

float iq[MAX_GATES * 2];

} Pulse;

/******************************
* Beam implementation example
*
* Note that the meta-data time, prt, el and az are not actually
* used in this code, they are just included for context.
*/
typedef struct {

/* meta data */

int nSamples; /* number of pulse samples in beam */
int nGates; /* number of gates */
double time; /* time for the center pulse of beam */
double prt; /* pulse repetition time (secs) */
double el; /* elevation angle for center of beam (deg) */
double az; /* azimuth angle for center of beam (deg) */

/* Array of Pulse pointers */
Pulse **pulses; /* Pointers to pulses with IQ data */

/* arrays for moments */

float *snr; /* SNR in dB */
float *dbz; /* reflectivity in dBZ */
float *vel; /* velocity in m/s */
float *width; /* spectrum width in m/s */

float *dbzf; /* filtered dBZ */

/* arrays for intermediate fields */

float *dbzDiffSq;
float *dbzSpinFlag;

/* arrays for CMD fields */

float *tdbz;
float *spin;
float *cpa;
/* array for clutter probability */
float *clutterProb;

/* array for clutter decision flag:
  * 1 means apply clutter filter
  * 0 means no not apply clutter filter
  */
int *clutterFlag;

} Beam;

Beam *create_beam(int nSamples,
                  int nGates,
                  double time,
                  double prt,
                  double el,
                  double az,
                  Pulse **pulses) {

  int isample, igate;

  Beam *beam = (Beam *) malloc(sizeof(Beam));
  /* set meta data */
  beam->nSamples = nSamples;
  beam->nGates = nGates;
  beam->time = time;
  beam->prt = prt;
  beam->el = el;
  beam->az = az;

  /* allocate space for arrays */
  beam->pulses = (Pulse **) malloc(nSamples * sizeof(Pulse *));
beam->dbzSpinFlag = (float *) malloc(nGates * sizeof(float));
beam->tdbz = (float *) malloc(nGates * sizeof(float));
beam->spin = (float *) malloc(nGates * sizeof(float));
beam->cpa = (float *) malloc(nGates * sizeof(float));
beam->clutterProb = (float *) malloc(nGates * sizeof(float));
beam->clutterFlag = (int *) malloc(nGates * sizeof(int));
/* initialize arrays */
for (isample = 0; isample < nSamples; isample++) {
    beam->pulses[isample] = pulses[isample];
}
for (igate = 0; igate < nGates; igate++) {
    beam->snr[igate] = CENSORED;
    beam->dbz[igate] = CENSORED;
    beam->vel[igate] = CENSORED;
    beam->width[igate] = CENSORED;
    beam->dbzf[igate] = CENSORED;
    beam->dbzDiffSq[igate] = CENSORED;
    beam->dbzSpinFlag[igate] = CENSORED;
    beam->tdbz[igate] = CENSORED;
    beam->spin[igate] = CENSORED;
    beam->cpa[igate] = CENSORED;
    beam->clutterProb[igate] = CENSORED;
    beam->clutterFlag[igate] = 0;
}
return beam;
}   

/*
 * free memory associated with a beam
 */
void free_beam(Beam *beam) {
    free(beam->pulses);
    free(beam->snr);
    free(beam->dbz);
    free(beam->vel);
    free(beam->width);
    free(beam->dbzf);
    free(beam->dbzDiffSq);
    free(beam->dbzSpinFlag);
    free(beam->tdbz);
    free(beam->spin);
    free(beam->cpa);
    free(beam->clutterProb);
    free(beam->clutterFlag);
    free(beam);
}
/***** Interest Map implementation example *****
* Implemented as a lookup table.
*/

/* struct for points which define interest map */

typedef struct {
    double feature;
    double interest;
} ImPoint;

/* interest map struct */

typedef struct {
    int nPoints;
    ImPoint *points;
    double minFeature;
    double maxFeature;
    double deltaFeature;
    float *lut;
} InterestMap;

/* Initialize map by passing in the points which define the map
* along with the weight for the map as a whole
* Returns NULL on failure.
*/

InterestMap *create_map(ImPoint *pts,
                        int npts) {

    int ii, jj;
    double slope;
    InterestMap *map = NULL;

    if (npts < 2) {
        /* error - must have at least 2 points to define function */
        return NULL;
    }

    /* allocate map itself */
    map = malloc(sizeof(InterestMap));

    /* clear map */
    memset(map, 0, sizeof(InterestMap));

    /* set meta data */
map->nPoints = npts;
/* allocate space for points which define map */
map->points = malloc(npts * sizeof(ImPoint));
/* copy point data */
memcpy(map->points, pts, npts * sizeof(ImPoint));
/* allocate space for lookup table */
map->lut = (float*) malloc(N_MAP_LOOKUP * sizeof(float));
/* compute min, max and delta */
map->minFeature = map->points[0].feature;
map->maxFeature = map->points[npts-1].feature;
map->deltaFeature =
    (map->maxFeature - map->minFeature) / (N_MAP_LOOKUP - 1.0);
/* compute slope for initial map segment */
jj = 1;
slope = ((map->points[jj].interest - map->points[jj-1].interest) /
            (map->points[jj].feature - map->points[jj-1].feature));
/* load lookup table */
for (ii = 0; ii < N_MAP_LOOKUP; ii++) {
    double interest;
    double val = map->minFeature + ii * map->deltaFeature;
    if ((val > map->points[jj].feature) && (jj < npts-1)) {
        /* move along by one segment */
        jj++;
        slope = ((map->points[jj].interest - map->points[jj-1].interest) /
            (map->points[jj].feature - map->points[jj-1].feature));
    }
    /* compute interest, add to lookup table */
    interest = map->points[jj-1].interest +
        (val - map->points[jj-1].feature) * slope;
    map->lut[ii] = interest;
} /* ii */
return map;
} /* create map */
/* free map memory */
void free_map(InterestMap *map) {
free(map->points);
free(map->lut);
free(map);

/**
 * look up interest given the feature value
 */
double getInterest(InterestMap *map, double feature) {
    int jj = (int) floor((feature - map->minFeature) / map->deltaFeature + 0.5);
    if (jj < 0) {
        jj = 0;
    } else if (jj > N_MAP_LOOKUP - 1) {
        jj = N_MAP_LOOKUP - 1;
    }
    return map->lut[jj];
}

/**
 * Set up interest maps for each feature field
 */
InterestMap *tdbzMap;
InterestMap *spinMap;
InterestMap *cpaMap;

ImPoint tdbzPoints[2] = {{20.0, 0.0}, {40.0, 1.0}};
ImPoint spinPoints[3] = {{15.0, 0.0}, {30.0, 1.0}};
ImPoint cpaPoints[2] = {{0.6, 0.0}, {0.9, 1.0}};

void create_interest_maps() {
    tdbzMap = create_map(tdbzPoints, 2);
    spinMap = create_map(spinPoints, 3);
    cpaMap = create_map(cpaPoints, 2);
}

/******************************
 * TDBZ implementation example
 * Compute TDBZ feature field
 */

/*
 * Prepare for computing TDBZ by computing
 * the gate-to-gate squared difference in Dbz
 */
void prepare_tdbz(Beam *beam) {
    int igate;
    /*
for (igate = 1; igate < beam->nGates; igate++) {
    float prevDbz = beam->dbz[igate-1];
    float dbz = beam->dbz[igate];
    if (prevDbz != CENSORED && dbz != CENSORED) {
        float dbzDiff = dbz - prevDbz;
        beam->dbzDiffSq[igate] = dbzDiff * dbzDiff;
    }
}
/* use gate 1 value for gate 0 */
beam->dbzDiffSq[0] = beam->dbzDiffSq[1];

void compute_tdbz(Beam *beam) {
    int igate;
    double sum = 0.0;
    double nn = 0.0;
    double mean = 0.0;
    for (igate = 0; igate < beam->nGates; igate++) {
        int startGate, endGate;
        int jbeam, jgate;
        /*
        * compute the start and end index, checking to make sure we
        * do not go off either end
        */
        startGate = igate - NGATES_KERNEL_TDBZ/2;
        if (startGate < 0) {
            startGate = 0;
        }
        endGate = igate + NGATES_KERNEL_TDBZ/2;
        if (endGate > beam->nGates - 1) {
            endGate = beam->nGates - 1;
        }
        /*
        * iterate over the kernel
* ignore censored gates
 */

for (jgate = startGate; jgate <= endGate; jgate++) {
    if (beam->dbzDiffSq[jgate] != CENSORED) {
        sum += beam->dbzDiffSq[jgate];
        nn += 1.0;
    }
} /* jgate */

if (nn > 0) {
    mean = sum / nn;
    beam->tdbz[igate] = mean;
}

} /* igate */
} /* compute_tdbz() */

/*****************************/
* SPIN implementation example
* Compute SPIN feature field
*/

/* Prepare for computing spin by setting the spin flag for
* each gate
*/

void prepare_spin(Beam *beam) {
    int igate;

    /* first set spin flag for each gate */
    for (igate = 1; igate < beam->nGates; igate++) {
        float dbzPrev = beam->dbz[igate-1];
        float dbzThis = beam->dbz[igate];
        float dbzNext = beam->dbz[igate+1];

        if (dbzPrev != CENSORED &&
            dbzThis != CENSORED &&
            dbzNext != CENSORED) {
            beam->dbzSpinFlag[igate] = 0;

            float prevDiff = dbzThis - dbzPrev;
            float nextDiff = dbzNext - dbzThis;

            if (prevDiff * nextDiff < 0) { /* sign change */
                float spinChange = (fabs(prevDiff) + fabs(nextDiff)) / 2.0;
                if (spinChange > SPIN_THRESHOLD) {
beam->dbzSpinFlag[igate] = 1;
  } else {
    beam->dbzSpinFlag[igate] = 0;
  }
  }
  } /* if (beam->dbz[igate] != CENSORED ... */

} /* igate */
/* use nearest values for end gates */
beam->dbzSpinFlag[0] = beam->dbzSpinFlag[1];
beam->dbzSpinFlag[beam->nGates-1] = beam->dbzSpinFlag[beam->nGates-2];
}

void compute_spin(Beam *beam) {
  double sum = 0.0;
  double nn = 0.0;
  double mean = 0.0;
  int igate;

  for (igate = 0; igate < beam->nGates; igate++) {
    int startGate, endGate;
    int jgate;

    /*
    * compute the start and end index, checking to make sure we
    * do not go off either end
    */
    startGate = igate - NGATES_KERNEL_SPIN/2;
    if (startGate < 0) {
      startGate = 0;
    }
    endGate = igate + NGATES_KERNEL_SPIN/2;
    if (endGate > beam->nGates - 1) {
      endGate = beam->nGates - 1;
    }

    /*
    * iterate over the kernel
    * ignore censored gates
    */
    for (jgate = startGate; jgate <= endGate; jgate++) {
      float spinFlag = beam->dbzSpinFlag[jgate];
      sum += spinFlag;
      nn++;
    }
    mean = sum / nn;
  }
}
if (spinFlag != CENSORED) {
    sum += spinFlag;
    nn += 1.0;
} /* jgate */
if (nn > 0) {
    mean = sum / nn;
    beam->spin[igate] = mean * 100.0;
}
/* igate */
} /* compute_spin */

/**************************
* CPA implementation example
* Compute CPA feature field
*/
void compute_cpa(Beam *beam) {
    int igate, isample;
    for (igate = 0; igate < beam->nGates; igate++) {
        Pulse *pulse = beam->pulses[igate];
        double sumMag = 0.0;
        double sumI = 0.0, sumQ = 0.0;
        double phasorLen;
        for (isample = 0; isample < beam->nSamples; isample++) {
            float ii = pulse->iq[isample * 2];
            float qq = pulse->iq[isample * 2 + 1];
            sumI += ii;
            sumQ += qq;
            sumMag += sqrt(ii * ii + qq * qq);
        } /* isample */
        phasorLen = sqrt(sumI * sumI + sumQ * sumQ);
        beam->cpa[igate] = phasorLen / sumMag;
    } /* igate */
} /* compute_cpa */

/**************************
* Clutter probability implementation example
* (a) compute clutter probability
* (b) set the clutter flag
*/
void compute_clut_prob(Beam *beam) {
int igate;

float tdbz, spin, cpa;

double tdbzInterest, spinInterest;
double maxTdbzSpinInterest;
double cpaInterest;

double clutterProb;
int clutterFlag;

for (igate = 0; igate < beam->nGates; igate++) {
    /* get the feature fields */
    tdbz = beam->tdbz[igate];
    spin = beam->spin[igate];
    cpa = beam->cpa[igate];

    /* convert feature fields to interest values */
    tdbzInterest = getInterest(tdbzMap, tdbz);
    spinInterest = getInterest(spinMap, spin);
    cpaInterest = getInterest(cpaMap, cpa);

    /* compute max of tdbz and spin interest */
    maxTdbzSpinInterest = MAX(tdbzInterest, spinInterest);

    /* compute weighted sum */
    double sum = 0.0;
    double sumWt = 0.0;
    sum =
        maxTdbzSpinInterest * MAX_TDBZ_SPIN_WEIGH +
        cpaInterest * CPA_WEIGHT;
    sumWt = MAX_TDBZ_SPIN_WEIGH + CPA_WEIGHT;

    clutterProb = sum / sumWt;

    if (clutterProb > PROB_THRESHOLD) {
        clutterFlag = 1;
    } else {
        clutterFlag = 0;
    }

    beam->clutterProb[igate] = clutterProb;
    beam->clutterFlag[igate] = clutterFlag;
}

} /* compute_clut_prob */
void applyCmdInfillFilter(Beam *beam) {
    int *countSet = (int *) malloc(beam->nGates * sizeof(int));
    int *countNot = (int *) malloc(beam->nGates * sizeof(int));

    /*
    * compute the running count of gates which have the flag set and
    * those which do not
    * Go forward through the gates, counting up the number of gates set
    * or not set and assigning that number to the arrays as we go.
    */

    int nSet = 0;
    int nNot = 0;
    int igate, jgate;
    for (igate = 0; igate < beam->nGates; igate++) {
        if (beam->clutterFlag[igate]) {
            nSet++;
            nNot = 0;
        } else {
            nSet = 0;
            nNot++;
        }
        countSet[igate] = nSet;
        countNot[igate] = nNot;
    }

    /*
    * Go in reverse through the gates, taking the max non-zero
    * values and copying them across the set or not-set regions.
    * This makes all the counts equal in the gaps and set areas.
    */
    for (igate = beam->nGates - 2; igate >= 0; igate--) {
        if (countSet[igate] != 0 &&
            countSet[igate] < countSet[igate+1]) {
            countSet[igate] = countSet[igate+1];
        }
        if (countNot[igate] != 0 &&
            countNot[igate] < countNot[igate+1]) {
            countNot[igate] = countNot[igate+1];
        }
    }

    /*
    * fill in gaps
    */
    for (igate = 1; igate < beam->nGates - 1; igate++) {

nNot = countNot[igate];
if (nNot > 0 && nNot <= CMD_MAX_NGATES_INFILLED) {
    
    int minGateCheck = igate - nNot;
    if (minGateCheck < 0) {
        minGateCheck = 0;
    }
    int maxGateCheck = igate + nNot;
    if (maxGateCheck > beam->nGates - 1) {
        maxGateCheck = beam->nGates - 1;
    }

    int nAdjacentBelow = 0;
    for (jgate = igate - 1; jgate >= minGateCheck; jgate--) {
        nSet = countSet[jgate];
        if (nSet != 0) {
            nAdjacentBelow = nSet;
            break;
        }
    } /* jgate */

    int nAdjacentAbove = 0;
    for (jgate = igate + 1; jgate <= maxGateCheck; jgate++) {
        nSet = countSet[jgate];
        if (nSet != 0) {
            nAdjacentAbove = nSet;
            break;
        }
    } /* jgate */

    int minAdjacent = nAdjacentBelow;
    minAdjacent = MIN(minAdjacent, nAdjacentAbove);

    if (minAdjacent >= nNot) {
        beam->clutterFlag[igate] = 1;
    }
} /* igate */

free(countSet);
free(countNot);

/*********************************************
* NEXRAD spike filter implementation example
* This routine filters the reflectivity data according to the
* NEXRAD specification DV1208621F, section 3.2.1.2.2, page 3-15.
* The algorithm is stated as follows:
* 
* Clutter detection:
* 
* The nth bin is declared to be a point clutter cell if its power value
* exceeds those of both its second nearest neighbors by a threshold value TCN. In other words:
*   
*   if   \( P(n) \) exceeds \( TCN \times P(n-2) \)
*   and \( P(n) \) exceeds \( TCN \times p(n+2) \)
* 
*   where
*   
*   \( TCN \) is the point clutter threshold factor, which is always greater than 1, and typically has a value of 8 (9 dB)
*   
*   \( P(n) \) if the poower sum value for the nth range cell
*   
*   \( n \) is the range gate number
* 
* Clutter censoring:
* 
* The formulas for censoring detected strong point clutter in an arbitrary array \( A \) via data substitution are as follows. If the nth range cell is an isolated clutter cell (i.e., it si a clutter cell but neither of its immediate neighboring cells is a clutter cell) then the replacement scheme is as follows:
*   
*   Replace \( A(n-1) \) with \( A(n-2) \)
*   Replace \( A(n) \) with \( 0.5 \times A(n-2) \times A(n+2) \)
*   Replace \( A(n+1) \) with \( A(n+2) \)
*   
*   If the nth and \( (n+1) \)th range bins constitute an isolated clutter pair, the bin replacement scheme is as follows:
*   
*   Replace \( A(n-1) \) with \( A(n-2) \)
*   Replace \( A(n) \) with \( A(n+2) \)
*   Replace \( A(n+1) \) with \( A(n+3) \)
*   Replace \( A(n+2) \) with \( A(n+3) \)
* 
*   Note that runs of more than 2 successive clutter cells cannot occur because of the nature of the algorithm.
* /

```c
void applyNexradSpikeFilter(Beam *beam)
{

    int ii;

    /*
     * set clutter threshold
```
double tcn = 9.0;

/*
 * loop through gates
 */

for (ii = 2; ii < beam->nGates - 3; ii++) {

/*
 * check for clutter at ii and ii + 1
 */

int this_gate = 0, next_gate = 0;

if ((beam->dbzf[ii] - beam->dbzf[ii - 2]) > tcn &&
   (beam->dbzf[ii] - beam->dbzf[ii + 2]) > tcn) {
   this_gate = 1;
}
if ((beam->dbzf[ii + 1] - beam->dbzf[ii - 1]) > tcn &&
   (beam->dbzf[ii + 1] - beam->dbzf[ii + 3]) > tcn) {
   next_gate = 1;
}

if (this_gate) {
   if (!next_gate) {
      /*
       * only gate ii has clutter, substitute accordingly
       */

      beam->dbzf[ii - 1] = beam->dbzf[ii - 2];
      beam->dbzf[ii + 1] = beam->dbzf[ii + 2];
      if ((beam->dbzf[ii - 2] == CENSORED ||
          beam->dbzf[ii + 2] == CENSORED) {
         beam->dbzf[ii] = CENSORED;
      } else {
         beam->dbzf[ii] = beam->dbzf[ii - 2];
      }
   } else {
      /*
       * both gate ii and ii+1 has clutter, substitute accordingly
       */

      beam->dbzf[ii - 1] = beam->dbzf[ii - 2];
      beam->dbzf[ii] = beam->dbzf[ii - 2];
      beam->dbzf[ii + 1] = beam->dbzf[ii + 3];
      beam->dbzf[ii + 2] = beam->dbzf[ii + 3];
   }
}
} /* ii */

}
Functional Description

for the

Dual Polarization
Clutter Mitigation Decision (CMD) system
for the
ORDA

Version 4.1

Prepared for:

WSR-88D Radar Operations Center
US National Weather Service
Norman, Oklahoma

Prepared by:

The National Center for Atmospheric Research
Boulder, Colorado

Mike Dixon
Scott Ellis

November 17, 2007
1. Overview....................................................................................................................1
2. Revision history .........................................................................................................1
   2.1 Version 1, single polarization, summary
   2.2 Version 2, single polarization, changes
   2.3 Version 3, single polarization, changes
   2.4 Version 4, single polarization, changes
   2.5 Version 4, dual polarization, extra fields
3. Applicability ................................................................................................................3
4. Algorithmic description ............................................................................................3
5. Feature fields...............................................................................................................3
   5.1 Overview
   5.2 TDBZ - DBZ texture
   5.3 SPIN - DBZ spin change
   5.4 CPA - clutter phase alignment
   5.5 SDEV_ZDR - standard deviation of ZDR in range
   5.6 SDEV_PHIDP - standard deviation of PHIDP in range
   5.7 Handling censored data in the feature fields
   5.8 Converting feature fields to interest fields
6. Flow charts.................................................................................................................6
   6.1 Computing the clutter flag
   6.2 Using the CMD clutter flag as a dynamic bypass map with SZ2
   6.3 Applying clutter filter with no range unfolding
7. Constants, known quantities and parameters .........................................................8
   7.1 Constants
   7.2 Parameters
   7.3 Interest map parameters
   7.4 Single-polarization membership function plots
   7.5 Dual-polarization membership function plots
8. Algorithm objects .......................................................................................................11
   8.1 Pulse object
   8.2 Beam object
   8.3 Interest Map object
9. Computing the individual feature fields ....................................................................19
   9.1 Computing DBZ Texture - TDBZ
   9.2 Computing DBZ SPIN
   9.3 Computing CPA
   9.4 Computing SDEV of ZDR
   9.5 Computing SDEV of PHIDP
   9.6 Computing clutter probability and clutter flag
   9.7 Spatial in-fill filter for the CMD clutter flag field
   9.8 NEXRAD spike filter
1. Overview

The goal of CMD is to provide guidance on where to apply an adaptive clutter filter such as Sigmet’s Gaussian Model Adaptive Processing (GMAP) filter. CMD will set a clutter probability flag for each gate in a radar beam at which (a) clutter is considered likely and (b) an adaptive filter such as GMAP is unlikely to remove weather power in error. The intention is that GMAP would then only be applied at the gates which have been flagged in this way.

2. Revision history

2.1 Version 1, single polarization, summary

The Version 1 Functional Description was delivered to the ROC in January 2006.

Version 1 used the following feature fields:

- TDBZ - reflectivity texture
- SPIN - reflectivity spin
- SDVE - standard deviation of velocity
- Clutter Ratio Wide

Version 1 used a computation kernel which spanned 5 consecutive azimuths, requiring that beam information be buffered to provide access to the data over the kernel. This buffering requirement has implications for the implementation in the RVP8.

2.2 Version 2, single polarization, changes

Version 2 differs from version 1 in the following ways:

- Clutter Ratio Wide has been dropped as a feature field.
- Clutter Phase Alignment (CPA) has been added as a feature field.
- The computation kernel is now limited to a single beam, so that beam buffering is no longer necessary.
- The length of the computation kernel is set individually for each field, rather than a single kernel length parameter being applied to all fields.
- A spatial (in-fill) filter is applied to the CMD flag field to fill in short gaps in range surrounded by significant clutter.
The feature fields used in version 2 are:

- TDBZ - reflectivity texture
- SPIN - reflectivity spin
- SDVE - standard deviation of velocity
- CPA - clutter phase alignment

The SDVE field is ‘turned off’ in this version, by setting the weight to 0, but it kept in the algorithm should it prove useful during tuning on further data sets.

### 2.3 Version 3, single polarization, changes

Version 3 differs from version 2 in the following ways:

- SDVE has been dropped completely.
- Clutter Ratio Narrow and Wide have been dropped completely.
- Only SNR is used for thresholding.
- TDBZ and SPIN are combined into a single interest field, which is the maximum of the individual interest fields.

The feature fields used in version 3 are:

- TDBZ - reflectivity texture
- SPIN - reflectivity spin
- Max of TDBZ interest and SPIN interest
- CPA - clutter phase alignment

### 2.4 Version 4, single polarization, changes

Version 4 differs from version 3 in the following ways:

- the SPIN computation has been changed to be symmetrical with respect to range. In other words, SPIN should have the same value whether it is computed with increasing range or decreasing range.
- the thresholds have been modified to handle this change in the formulation of SPIN.

### 2.5 Version 4, dual polarization, extra fields

The dual polarization CMD algorithm differs from the single polarization version only in the addition of two extra feature fields derived from dual-polarization variables:

- sdevZdr: the standard deviation of ZDR in range.
- sdevPhidp: the standard deviation of PHIDP in range.
- version 4.1 has a modified interest map for SPIN
3. Applicability

This version of CMD is intended for use with data which has the following characteristics:

- constant PRT;
- dual polarization;
- non-phase coded;
- at least 16 samples per beam.

The most likely application of this version would be to generate a ‘dynamic clutter map’ using the long-PRT data of the first 2 tilts in a Volume Coverage Pattern (VCP). This dynamic clutter map would detect situations such as AP which are missed by the static clutter map. This dynamic would provide clutter information to algorithms such as SZ2.

The algorithm could also be used stand-alone with any constant-PRT data for deciding where to safely apply a clutter map.

4. Algorithmic description

In this document, the algorithm is described using a number of techniques: (a) flow charts; (b) step-by-step logic in text and graphics and (c) C code examples. C code is considered an accurate and unambiguous way to present the information because the RVP8 implementation will be coded in C.

The intention is that the flow charts and other explanations should allow an understanding of the algorithm by those readers who are not familiar with C code.

Please note that the C code was derived from a C++ application which has been tested on real data cases. The code compiles, but has not actually been run and tested in this form.

5. Feature fields

5.1 Overview

CMD uses a fuzzy logic approach to combine the information from a number of so-called ‘Feature Fields’ into a single decision-making field.

Some of the fields (TDBZ, SPIN) are computed using information from a small region in range (the ‘kernel’) around the gate. Other fields, such as CPA, are computed from information at the gate only.

In the version current, the kernel is 1-dimensional, along the beam in range. The length of the kernel is specified individually for each relevant feature field.
5.2 TDBZ - DBZ texture

Regions of clutter exhibit rapidly changing reflectivity from gate to gate. DBZ texture is a measure of the gate-to-gate change in value of dBZ. It is computed over a kernel. First, the gate-to-gate dbz difference is computed for each gate, and then squared. TDBZ is the mean of this squared difference over the kernel.

5.3 SPIN - DBZ spin change

Like TDBZ, SPIN is designed to find regions of rapidly changing reflectivity. SPIN is based on the number of significant changes in sign in the gradient of the reflectivity field over a kernel. A gate is considered to be a spin change point if (a) the mean absolute differences in dBZ, from the previous gate to this gate, and this gate to the next gate, exceeds a threshold (\texttt{SPIN\_THRESHOLD}) and (b) the sign of the slope has changed since the last spin point. The number of spin points is computed and then expressed as a percentage of the number of possible spin changes.

5.4 CPA - clutter phase alignment

In clutter the phase of each pulse in the time series for a particular gate is almost constant since the clutter does not move significantly and is at a constant distance from the radar. In noise, the phase from pulse to pulse is random. In weather, the phase from pulse to pulse will vary depending on the velocity of the targets within the illumination volume.

CPA is computed as the length of the cumulative phasor vector, normalized by the sum of the magnitudes for the pulses. CPA is computed at a single gate. It ranges from 0 to 1. In clutter, CPA is typically above 0.95. In weather, CPA is often close to 0, but increases in weather having a velocity close to 0 and a narrow spectrum width. In noise, CPA is typically less than 0.05.

5.5 SDEV_ZDR - standard deviation of ZDR in range

In clutter, the ZDR field is very noisy and varies widely from gate to gate. Therefore, the standard deviation of ZDR in range is a good indicator of the likelihood of clutter. The standard deviation is computed over a small number of gates, 7 in this version.

One advantage of using the standard deviation of ZDR, rather than ZDR itself, is that the absolute calibration of ZDR is difficult to obtain, whereas the standard deviation is not sensitive to the absolute calibration.
5.6 SDEV_PHIDP - standard deviation of PHIDP in range

In clutter, the PHIDP field is very noisy and varies widely from gate to gate. Therefore, the standard deviation of PHIDP in range is a good indicator of the likelihood of clutter. The standard deviation is computed over a small number of gates, 7 in this version.

One advantage of using the standard deviation of PHIDP, rather than PHIDP itself, is that the absolute value of PHIDP depends on the system PHIDP value, whereas the standard deviation is independent of the system value.

5.7 Handling censored data in the feature fields

In computing the feature fields, moments data which has been censored are ignored.

In addition to the censoring which occurs upstream of the algorithm, CMD sets the clutter flag at a gate to FALSE if the SNR value at the date is below a set threshold. This prevents the application of a clutter filter at that gate, which improves efficiency.

5.8 Converting feature fields to interest fields

The values in the feature fields are dependent on what the field physically represents. In order to combine fields in a meaningful manner, feature fields are first converted into so-called ‘Interest Fields’ by applying a so-called membership transfer function to the feature field. The function converts feature values into interest values between 0 and 1. These interest fields may then be manipulated by the fuzzy logic system to produce the final decision-making field - in this case, the clutter flag.
6. Flow charts

6.1 Computing the clutter flag

check for new pulse

are pulses available to form new beam?

NO

YES

group pulses into beam

for each gate in beam

compute SNR, dBZ, CPA, ZDR, PHIDP

for each gate in beam

compute dbzDiffSq, dbzSpinFlag

for each gate in beam

compute TDBZ, SPIN

convert TDBZ, SPIN, CPA, ZDR, PHIDP to interest values

compute MAX(TDBZ interest, SPIN interest)

for each gate in beam

set clutterFlag = FALSE

NO

YES

is (SNR > SNR_THRESHOLD)?

combine interest values into clutterProbability

NO

YES

is (clutterProbability > PROB_THRESHOLD)?

apply IN-FILL filter to clutter flag field

apply clutter filter based on clutter flag
6.2 Using the CMD clutter flag as a dynamic bypass map with SZ2

- for each indexed beam in long-prt PPI
  - compute CMD clutter flag at each gate
  - Write clutter flag field to buffer, indexed by azimuth
- for each indexed beam in short-prt PPI
  - Read clutter flag field from buffer, for this azimuth
  - Using clutter flag fields as dynamic bypass map, compute SZ2
    - apply NEXRAD spike filter to filtered fields

6.3 Applying clutter filter with no range unfolding

- for each gate in beam
  - is clutterFlag set?
    - NO
    - YES
      - apply clutter filter
      - apply NEXRADR spike filter to filtered fields
7. Constants, known quantities and parameters

7.1 Constants

The following values are constant for a particular implementation of CMD:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Example value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_GATES</td>
<td>integer</td>
<td>2048</td>
<td>Maximum number of gates in a beam.</td>
</tr>
</tbody>
</table>

7.2 Parameters

These values are set at the start of the algorithm:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Suggested value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGATES_KERNEL_TDBZ</td>
<td>integer</td>
<td>9</td>
<td>Number of gates in kernel for TDBZ</td>
</tr>
<tr>
<td>NGATES_KERNEL_SPIN</td>
<td>integer</td>
<td>11</td>
<td>Number of gates in kernel for SPIN</td>
</tr>
<tr>
<td>NGATES_KERNEL_ZDR_SDEV</td>
<td>integer</td>
<td>7</td>
<td>Number of gates in kernel for ZDR SDEV</td>
</tr>
<tr>
<td>NGATES_KERNEL_PHIDP_SDEV</td>
<td>integer</td>
<td>7</td>
<td>Number of gates in kernel for PHIDP SDEV</td>
</tr>
<tr>
<td>SPIN_THRESHOLD</td>
<td>floating point</td>
<td>6.5</td>
<td>Difference threshold used in computation of SPIN, in dBZ</td>
</tr>
<tr>
<td>SNR_THRESHOLD</td>
<td>floating point</td>
<td>3.0</td>
<td>Signal-to-noise ratio threshold for censoring on total power, in dB</td>
</tr>
<tr>
<td>PROB_THRESHOLD</td>
<td>floating point</td>
<td>0.5</td>
<td>Probability threshold. If the clutter probability exceeds the threshold the clutter flag is set TRUE.</td>
</tr>
<tr>
<td>CMD_MAX_GATES_INFILLED</td>
<td>integer</td>
<td>3</td>
<td>Maximum number of gates to be in-filled in the CMD flag field</td>
</tr>
</tbody>
</table>
### 7.3 Interest map parameters

The following parameters define the interest maps for the CMD fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Weight</th>
<th>Membership function definition points</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDBZ</td>
<td>0.0</td>
<td>(20, 0), (40, 1)</td>
</tr>
<tr>
<td>SPIN</td>
<td>0.0</td>
<td>(15, 0), (30, 1)</td>
</tr>
<tr>
<td>MAX_TDBZ_SPIN</td>
<td>1.0</td>
<td>(0, 0), (1, 1) (identity)</td>
</tr>
<tr>
<td>CPA</td>
<td>1.01</td>
<td>(0.6, 1), (0.9, 1)</td>
</tr>
<tr>
<td>ZDR_SDEV</td>
<td>0.5</td>
<td>(1.2, 1), (2.4, 1)</td>
</tr>
<tr>
<td>PHIDP_SDEV</td>
<td>0.5</td>
<td>(10, 1), (15, 1)</td>
</tr>
</tbody>
</table>

### 7.4 Single-polarization membership function plots

The following plots show the single-polarization membership functions graphically.

1. **TDBZ**
   - Interest value: 1
   - Feature value: 0, 20, 40
   - Membership function definition points: (20, 0), (40, 1)

2. **SPIN**
   - Interest value: 1
   - Feature value: 0, 15, 30
   - Membership function definition points: (15, 0), (30, 1)

3. **CPA**
   - Interest value: 1
   - Feature value: 0, 0.6, 0.9
   - Membership function definition points: (0.6, 0), (0.9, 1)
7.5 Dual-polarization membership function plots

The following plots show the dual-polarization membership functions graphically.

![ZDR-SDEV Membership Function](image1)

![PHIDP-SDEV Membership Function](image2)
8. Algorithm objects

In Object-Oriented (OO) programming, an object is a useful abstraction which groups together some data and some defined behavior associated with that data. There are a number of aspects of the CMD algorithm which are well described by the Object model. These items do not necessarily need to be coded as formal objects. However they can be thought of as objects for this discussion.

This section defines a number of objects which are used by the algorithm. It is useful to define these first and then refer to them later in the document.

8.1 Pulse object

The Pulse object consists of header data and an array of floating point IQ data.

8.1.1 Pulse implementation example

The Pulse object may be coded as a C structure.

```c
typedef struct {
    /* meta data */
    int nGates; /* number of gates */
    double time; /* time in secs and fractions from 1 Jan 1970 */
    double prt; /* pulse repetition time (secs) */
    double el; /* elevation angle (deg) */
    double az; /* azimuth angle (deg) */

    /* IQ data */
    float iq[MAX_GATES * 2];
} Pulse;
```

8.2 Beam object

A Beam is made up of an array of Pulses. The Pulses are often referred to as ‘samples’. There are nSamples pulses in a Beam.

A Beam object consists of header data, along with an array of pulses and arrays for storing the moments (Z, V, W) and relevant feature fields and interest fields.
8.2.1 Beam implementation example

The Beam object may be coded as a C structure.

/*****************************
* Beam implementation example
*
* Note that the meta-data time, prt, el and az are not actually
* used in this code, they are just included for context.
*/

typedef struct {
    /* meta data */
    int nSamples; /* number of pulse samples in beam */
    int nGates; /* number of gates */
    double time; /* time for the center pulse of beam */
    double prt; /* pulse repetition time (secs) */
    double el; /* elevation angle for center of beam (deg) */
    double az; /* azimuth angle for center of beam (deg) */

    /* Array of Pulse pointers */
    Pulse **pulses; /* Pointers to pulses with IQ data */

    /* arrays for moments */
    float *snr; /* SNR in dB */
    float *dbz; /* reflectivity in dBZ */
    float *vel; /* velocity in m/s */
    float *zdr; /* zdr in dB */
    float *phidp; /* phidp in deg */

    float *dbzf; /* filtered dBZ */

    /* arrays for intermediate fields */
    float *dbzDiffSq;
    float *dbzSpinFlag;

    /* arrays for CMD fields */
    float *tdbz;
    float *spin;
    float *sdevZdr;
    float *sdevPhidp;
    float *cpa;

    /* array for clutter probability */
}
float *clutterProb;

/* array for clutter decision flag:
 *  1 means apply clutter filter
 *  0 means not apply clutter filter
 */

int *clutterFlag;

_beam;

/*
 * create a beam
 *
 * (a) set meta data
 * (b) allocate array memory
 * (c) initialize values to censored or 0
 *
 * Pulse memory is not owned by the beam. It should be
 * allocated and freed elsewhere.
 */

Beam *create_beam(int nSamples,
    int nGates,
    double time,
    double prt,
    double el,
    double az,
    Pulse **pulses) {

int isample, igate;

/* allocate space for beam */

Beam *beam = (Beam *) malloc(sizeof(Beam));

/* set meta data */

beam->nSamples = nSamples;
beam->nGates = nGates;
beam->time = time;
beam->prt = prt;
beam->el = el;
beam->az = az;

/* allocate space for arrays */

beam->pulses = (Pulse **) malloc(nSamples * sizeof(Pulse *));

beam->snr = (float *) malloc(nGates * sizeof(float));
**Functional Description for CMD for the ORDA, Dual-polarization, Version 4.1**

```c
beam->dbz = (float *) malloc(nGates * sizeof(float));
beam->vel = (float *) malloc(nGates * sizeof(float));
beam->zdr = (float *) malloc(nGates * sizeof(float));
beam->phidp = (float *) malloc(nGates * sizeof(float));
beam->dbzf = (float *) malloc(nGates * sizeof(float));
beam->dbzDiffSq = (float *) malloc(nGates * sizeof(float));
beam->dbzSpinFlag = (float *) malloc(nGates * sizeof(float));
beam->tdbz = (float *) malloc(nGates * sizeof(float));
beam->spin = (float *) malloc(nGates * sizeof(float));
beam->sdevZdr = (float *) malloc(nGates * sizeof(float));
beam->sdevPhidp = (float *) malloc(nGates * sizeof(float));
beam->cpa = (float *) malloc(nGates * sizeof(float));
beam->clutterProb = (float *) malloc(nGates * sizeof(float));
beam->clutterFlag = (int *) malloc(nGates * sizeof(int));

/* initialize arrays */

for (isample = 0; isample < nSamples; isample++) {
    beam->pulses[isample] = pulses[isample];
}

for (igate = 0; igate < nGates; igate++) {
    beam->snr[igate] = CENSORED;
    beam->dbz[igate] = CENSORED;
    beam->vel[igate] = CENSORED;
    beam->zdr[igate] = CENSORED;
    beam->phidp[igate] = CENSORED;
    beam->dbzf[igate] = CENSORED;
    beam->dbzDiffSq[igate] = CENSORED;
    beam->dbzSpinFlag[igate] = CENSORED;
    beam->tdbz[igate] = CENSORED;
    beam->spin[igate] = CENSORED;
    beam->sdevZdr[igate] = CENSORED;
    beam->sdevPhidp[igate] = CENSORED;
    beam->cpa[igate] = CENSORED;
    beam->clutterProb[igate] = CENSORED;
    beam->clutterFlag[igate] = 0;
}

return beam;
```

/*
 * free memory associated with a beam
 */

```c
void free_beam(Beam *beam) {
```

NCAR. November 17, 2007 14
free(beam->pulses);
free(beam->snr);
free(beam->dbz);
free(beam->vel);
free(beam->zdr);
free(beam->phidp);
free(beam->dbzf);
free(beam->dbzDiffSq);
free(beam->dbzSpinFlag);
free(beam->tdbz);
free(beam->spin);
free(beam->sdevZdr);
free(beam->sdevPhidp);
free(beam->cpa);
free(beam->clutterProb);
free(beam->clutterFlag);
free(beam);

}

8.3 Interest Map object

8.3.1 Overview

CMD uses fuzzy logic to combine information from different fields. Prior to computing the fuzzy combination, the feature fields are converted to so-called ‘interest fields’ by applying a piece-wise linear membership function which converts the feature values to an ‘interest value’ between 0 and 1.

The figure above shows an example of a membership transfer function. For each feature value there exists a unique interest value: interest = f(feature).

The piece-wise linear function can be defined by specifying the coordinates of the points a, b, c, d, and e. The function is linear between these points, and constant outside the range of the first and last points.
For example, the membership function above may be defined by the array of points in the following table.

<table>
<thead>
<tr>
<th>Point</th>
<th>x-coord</th>
<th>y-coord</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>-1.5</td>
<td>0.0</td>
</tr>
<tr>
<td>b</td>
<td>-1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>c</td>
<td>-0.5</td>
<td>0.25</td>
</tr>
<tr>
<td>d</td>
<td>0.5</td>
<td>0.25</td>
</tr>
<tr>
<td>e</td>
<td>1.5</td>
<td>1.0</td>
</tr>
</tbody>
</table>

For all feature values below -1.5, the interest value is constant, at 0.0, while for all feature values above 1.5 the interest value is constant at 1.0.

Also associated with the Interest Map is the weight which will be applied to the interest field when the individual fields are combined to form a weighted-mean interest field.

### 8.3.2 Interest Map implementation example

The Interest Map object can be efficiently implemented as a lookup table, where the lookup only occurs for feature values between the minimum and maximum specified values. For feature values less than the minimum specified, the interest value is constant at the value for the first point. For feature values above the maximum specified, the interest value is constant at the value for the last point.

```c
/
	* Interest Map implementation example
	* 
	* Implemented as a lookup table.
	*/

/* struct for points which define interest map */

typedef struct {
    double feature;
    double interest;
} ImPoint;

/* interest map struct */

typedef struct {
    int nPoints;
    ImPoint *points;
    double minFeature;
    double maxFeature;
    double deltaFeature;
```
float *lut;
) InterestMap;

/*
 * Initialize map by passing in the points which define the map
 * along with the weight for the map as a whole
 *
 * Returns NULL on failure.
 */
InterestMap *create_map(ImPoint *pts,
    int npts) {

    int ii, jj;
    double slope;
    InterestMap *map = NULL;

    if (npts < 2) {
        /* error - must have at least 2 points to define function */
        return NULL;
    }

    /* allocate map itself */
    map = malloc(sizeof(InterestMap));

    /* clear map */
    memset(map, 0, sizeof(InterestMap));

    /* set meta data */
    map->nPoints = npts;

    /* allocate space for points which define map */
    map->points = malloc(npts * sizeof(ImPoint));

    /* copy point data */
    memcpy(map->points, pts, npts * sizeof(ImPoint));

    /* allocate space for lookup table */
    map->lut = (float*) malloc(N_MAP_LOOKUP * sizeof(float));

    /* compute min, max and delta */
    map->minFeature = map->points[0].feature;
    map->maxFeature = map->points[npts-1].feature;}
map->deltaFeature =
    (map->maxFeature - map->minFeature) / (N_MAP_LOOKUP - 1.0);

/* compute slope for initial map segment */
jj = 1;
slope = ((map->points[jj].interest - map->points[jj-1].interest) /
    (map->points[jj].feature - map->points[jj-1].feature));

/* load lookup table */
for (ii = 0; ii < N_MAP_LOOKUP; ii++) {
    double interest;
    double val = map->minFeature + ii * map->deltaFeature;
    if ((val > map->points[jj].feature) && (jj < npts-1)) {
        /* move along by one segment */
        jj++;
        slope = ((map->points[jj].interest - map->points[jj-1].interest) /
            (map->points[jj].feature - map->points[jj-1].feature));
    }
    /* compute interest, add to lookup table */
    interest = map->points[jj-1].interest +
        (val - map->points[jj-1].feature) * slope;
    map->lut[ii] = interest;
}
/* ii */
return map;
}

/* free map memory */
void free_map(InterestMap *map) {
    free(map->points);
    free(map->lut);
    free(map);
}

/* look up interest given the feature value */

double getInterest(InterestMap *map, double feature) {
    int jj = (int) floor((feature - map->minFeature) / map->deltaFeature + 0.5);
    if (jj < 0) {

Functional Description for CMD for the ORDA, Dual-polarization, Version 4.1

void create_interest_maps() {
    tdbzMap = create_map(tdbzPoints, 2);
    spinMap = create_map(spinPoints, 3);
    cpaMap = create_map(cpaPoints, 2);
}

9. Computing the individual feature fields

9.1 Computing DBZ Texture - TDBZ

TDBZ is a measure of the gate-to-gate change in value of the dBZ values. It is computed over the kernel.

First, the squared differences in dBZ values from gate to gate are computed. Then, the mean value of these squared differences over the kernel is computed.

\[
TDBZ = \frac{1}{N} \sum_{i}^{n_{\text{gates}}} (dBZ_{i,j} - dBZ_{i-1,j})^2
\]

9.1.1 TDBZ implementation example

First, we need to compute the squared difference in dBZ from gate to gate in the beam.

/*
Prepare for computing TDBZ by computing the gate-to-gate squared difference in Dbz

```c
void prepare_tdbz(Beam *beam) {
    int igate;
    
    /*
     * compute the squared difference in dBZ from gate to gate in the beam.
     */
    for (igate = 1; igate < beam->nGates; igate++) {
        float prevDbz = beam->dbz[igate-1];
        float dbz = beam->dbz[igate];
        
        if (prevDbz != CENSORED && dbz != CENSORED) {
            float dbzDiff = dbz - prevDbz;
            beam->dbzDiffSq[igate] = dbzDiff * dbzDiff;
        }
    }
    /* use gate 1 value for gate 0 */
    beam->dbzDiffSq[0] = beam->dbzDiffSq[1];
}
```

Then, we compute the mean squared difference over the kernel, setting the value for the center beam of the beam queue.

```c
void compute_tdbz(Beam *beam) {
    int igate;
    double sum = 0.0;
    double nn = 0.0;
    double mean = 0.0;
    
    for (igate = 0; igate < beam->nGates; igate++) {
        int startGate, endGate;
        int jbeam, jgate;
        
        /*
         * compute the start and end index, checking to make sure we do not go off either end
         */
        ```
startGate = igate - NGATES_KERNEL_TDBZ/2;
if (startGate < 0) {
    startGate = 0;
}

endGate = igate + NGATES_KERNEL_TDBZ/2;
if (endGate > beam->nGates - 1) {
    endGate = beam->nGates - 1;
}

/*
 * iterate over the kernel
 * ignore censored gates
 */

for (jgate = startGate; jgate <= endGate; jgate++) {
    if (beam->dbzDiffSq[jgate] != CENSORED) {
        sum += beam->dbzDiffSq[jgate];
        nn += 1.0;
    }
} /* jgate */

if (nn > 0) {
    mean = sum / nn;
    beam->tdbz[igate] = mean;
}

} /* igate */

} /* compute_tdbz() */

9.2 Computing DBZ SPIN

SPIN is a measure of how frequently the slope of dBZ changes sign with range.

See the following figure as an example. The slope changes sign at points 1, 2, 3, 4 and 5.
First we compute the spin change at each of these points. The spin change is defined as the mean of the absolute difference in reflectivity between the previous gate and this gate, and this gate and the next gate.

So the spin change for each of the points marked above is:

\[
\begin{align*}
\text{spin\_change}(1) &= \frac{(\text{abs}(A) + \text{abs}(B))}{2} \\
\text{spin\_change}(2) &= \frac{(\text{abs}(C) + \text{abs}(D))}{2} \\
\text{spin\_change}(3) &= \frac{(\text{abs}(D) + \text{abs}(E))}{2} \\
\text{spin\_change}(4) &= \frac{(\text{abs}(F) + \text{abs}(G))}{2} \\
\text{spin\_change}(5) &= \frac{(\text{abs}(G) + \text{abs}(H))}{2}
\end{align*}
\]

A point with a change in sign is only counted if the spin change exceeds a set threshold.

### 9.2.1 SPIN implementation example

First, we set a flag at a gate if an inflection point exists. This flag field will then be used later to compute SPIN.

```c
/********************************/
/* SPIN implementation example */
/* Compute SPIN feature field */
/*
/* Prepare for computing spin by setting the spin flag for
/* each gate
/*

void prepare_spin(Beam *beam) {
    int igate;

    /* first set spin flag for each gate */
    for (igate = 1; igate < beam->nGates; igate++) {
```
float dbzPrev = beam->dbz[igate-1];
float dbzThis = beam->dbz[igate];
float dbzNext = beam->dbz[igate+1];

if (dbzPrev != CENSORED &&
    dbzThis != CENSORED &&
    dbzNext != CENSORED) {

    beam->dbzSpinFlag[igate] = 0;

    float prevDiff = dbzThis - dbzPrev;
    float nextDiff = dbzNext - dbzThis;

    if (prevDiff * nextDiff < 0) {
        /* sign change */
        float spinChange = (fabs(prevDiff) + fabs(nextDiff)) / 2.0;
        if (spinChange > SPIN_THRESHOLD) {
            beam->dbzSpinFlag[igate] = 1;
        } else {
            beam->dbzSpinFlag[igate] = 0;
        }
    }
}

} /* if (beam->dbz[igate] != CENSORED ... */

} /* igate */

/* use nearest values for end gates */

beam->dbzSpinFlag[0] = beam->dbzSpinFlag[1];
beam->dbzSpinFlag[beam->nGates-1] = beam->dbzSpinFlag[beam->nGates-2];

}

Then, we compute the SPIN as a percentage over the kernel, setting the value in the center beam of the beam queue.

void compute_spin(Beam *beam) {

    double sum = 0.0;
    double nn = 0.0;
    double mean = 0.0;
    int igate;

    for (igate = 0; igate < beam->nGates; igate++) {

        int startGate, endGate;
        int jgate;

        /*
         * compute the start and end index, checking to make sure we
* do not go off either end
*/

startGate = igate - NGATES_KERNEL_SPIN/2;
if (startGate < 0) {
    startGate = 0;
}

degate = igate + NGATES_KERNEL_SPIN/2;
if (endGate > beam->nGates - 1) {
    endGate = beam->nGates - 1;
}

/* iterate over the kernel
* ignore censored gates
*/

for (jgate = startGate; jgate <= endGate; jgate++) {
    float spinFlag = beam->dbzSpinFlag[jgate];
    if (spinFlag != CENSORED) {
        sum += spinFlag;
        nn += 1.0;
    }
} /* jgate */

if (nn > 0) {
    mean = sum / nn;
    beam->spin[igate] = mean * 100.0;
} /* igate */

} /* compute_spin */
9.3 Computing CPA

The following figure demonstrates how CPA is computed.

CPA is computed as the length of the combined phasor vector, normalized by the sum of the pulse magnitudes.

9.3.1 CPA implementation example

```c
void compute_cpa(Beam *beam) {
    int igate, isample;

    for (igate = 0; igate < beam->nGates; igate++) {
        Pulse *pulse = beam->pulses[igate];
        double sumMag = 0.0;
        double sumI = 0.0, sumQ = 0.0;
        double phasorLen;

        for (isample = 0; isample < beam->nSamples; isample++) {
            float ii = pulse->iq[isample * 2];
            float qq = pulse->iq[isample * 2 + 1];
            sumI += ii;
            sumQ += qq;
            sumMag += sqrt(ii * ii + qq * qq);
        } /* isample */

        phasorLen = sqrt(sumI * sumI + sumQ * sumQ);
    } /* igate */
}
```
beam->cpa[igate] = phasorLen / sumMag;

} /* igate */
} /* compute_cpa */

9.4 Computing SDEV of ZDR

The standard deviation of ZDR is computed over a series of gates in range.

9.4.1 SDEV of ZDR example

/***************************************************************************/
/* SDEV_ZDR implementation example */
/* Compute SDEV_ZDR */
/* SDEV_ZDR is the standard deviation of ZDR over the kernel. */
/* It is assumed that ZDR has already been computed for the beam. */
/***************************************************************************/

void compute_sdev_zdr(Beam *beam) {

    int igate;
    double sum = 0.0;
    double sumSq = 0.0;
    double nn = 0.0;
    double sdevZdr = 0.0;

    for (igate = 0; igate < beam->nGates; igate++) {

        int startGate, endGate;
        int jbeam, jgate;

        /*
         * compute the start and end index, checking to make sure we
         * do not go off either end
         */

        startGate = igate - NGATES KERNEL SDEV ZDR/2;
        if (startGate < 0) {
            startGate = 0;
        }

        endGate = igate + NGATES KERNEL SDEV ZDR/2;
        if (endGate > beam->nGates - 1) {
            endGate = beam->nGates - 1;
        }

        /*
         * iterate over the kernel
         * ignore censored gates
         */

    }
}
for (jgate = startGate; jgate <= endGate; jgate++) {
    float zdr = beam->zdr[jgate];
    if (zdr != CENSORED) {
        sum += zdr;
        sumSq += (zdr * zdr);
        nn += 1.0;
    }
} /* jgate */

if (nn >= 2) {
    double mean = sum / nn;
    double term1 = sumSq / nn;
    double term2 = mean * mean;
    if (term1 >= term2) {
        sdevZdr = sqrt(term1 - term2);
    } else {
        sdevZdr = 0.0;
    }
    beam->sdevZdr[igate] = sdevZdr;
}

} /* compute_sdev_zdr */

9.5 Computing SDEV of PHIDP

The standard deviation of PHIDP is computed over a series of gates in range.

9.5.1 SDEV of PHIDP example

/***************************************************************************/
/* SDEV_PHIDP implementation example */
/* Compute SDEV_PHIDP */
/* SDEV_PHIDP is the standard deviation of PHIDP over the kernel. */
/* It is assumed that PHIDP has already been computed for the beam. */
/***************************************************************************/

void compute_sdev_phidp(Beam *beam) {

    int igate;
    double sum = 0.0;
    double sumSq = 0.0;
    double nn = 0.0;
    double sdevPhidp = 0.0;

    for (igate = 0; igate < beam->nGates; igate++) {
int startGate, endGate;
int jbeam, jgate;

/*
 * compute the start and end index, checking to make sure we
 * do not go off either end
 */

startGate = igate - NGATES_KERNEL_SDEV_PHIDP/2;
if (startGate < 0) {
  startGate = 0;
}

endGate = igate + NGATES_KERNEL_SDEV_PHIDP/2;
if (endGate > beam->nGates - 1) {
  endGate = beam->nGates - 1;
}

/*
 * iterate over the kernel
 * ignore censored gates
 */

for (jgate = startGate; jgate <= endGate; jgate++) {
  float phidp = beam->phidp[jgate];
  if (phidp != CENSORED) {
    sum += phidp;
    sumSq += (phidp * phidp);
    nn += 1.0;
  }
} /* jgate */

if (nn >= 2) {
  double mean = sum / nn;
  double term1 = sumSq / nn;
  double term2 = mean * mean;
  if (term1 >= term2) {
    sdevPhidp = sqrt(term1 - term2);
  } else {
    sdevPhidp = 0.0;
  }
  beam->sdevPhidp[igate] = sdevPhidp;
}

} /* igate */

} /* compute_sdev_phidp */
9.6 Computing clutter probability and clutter flag

To compute the clutter probability:

- At each gate:
  - convert each CMD field into an interest value.
  - compute \( \text{maxTdbzSpinInterest} = \text{MAX}(\text{TDBZ interest}, \text{SPIN interest}) \).
  - compute \( \text{clutterProbability} \), as a weighted mean of the CPA interest and the \( \text{maxTdbzSpinInterest} \).
  - if \( \text{clutterProbability} \) exceeds \( \text{PROB\_THRESHOLD} \), set \( \text{clutterFlag} \) to TRUE. Otherwise, set \( \text{clutterFlag} \) to FALSE.

9.6.1 Clutter probability implementation example

```c
/********************************************
* Clutter probability implementation example
*
*    (a) compute clutter probability
*    (b) set the clutter flag
*/

void compute_clut_prob(Beam *beam) {

    int igate;
    float tdbz, spin, sdevZdr, sdevPhidp, cpa;
    double tdbzInterest, spinInterest;
    double sdevZdrInterest, sdevPhidpInterest;
    double maxTdbzSpinInterest;
    double cpaInterest;
    double clutterProb;
    int clutterFlag;

    for (igate = 0; igate < beam->nGates; igate++) {
        /* get the feature fields */
        tdbz = beam->tdbz[igate];
        spin = beam->spin[igate];
        sdevZdr = beam->sdevZdr[igate];
        sdevPhidp = beam->sdevPhidp[igate];
        cpa = beam->cpa[igate];

        /* convert feature fields to interest values */
        tdbzInterest = getInterest(tdbzMap, tdbz);
        spinInterest = getInterest(spinMap, spin);
```
sdevZdrInterest = getInterest(sdevZdrMap, sdevZdr);
sdevPhidpInterest = getInterest(sdevPhidpMap, sdevPhidp);
cpaInterest = getInterest(cpaMap, cpa);

/* compute max of tdbz and spin interest */
maxTdbzSpinInterest = MAX(tdbzInterest, spinInterest);

/* compute weighted sum */

double sum = 0.0;
double sumWt = 0.0;

sum =
    maxTdbzSpinInterest * MAX_TDBZ_SPIN_WEIGHT +
    sdevZdrInterest * SDEV_ZDR_WEIGHT +
    sdevPhidpInterest * SDEV_PHIDP_WEIGHT +
    cpaInterest * CPA_WEIGHT;

sumWt =
    MAX_TDBZ_SPIN_WEIGHT +
    SDEV_ZDR_WEIGHT +
    SDEV_PHIDP_WEIGHT +
    CPA_WEIGHT;

clutterProb = sum / sumWt;

if (clutterProb > PROB_THRESHOLD) {
    clutterFlag = 1;
} else {
    clutterFlag = 0;
}

beam->clutterProb[igate] = clutterProb;
beam->clutterFlag[igate] = clutterFlag;

} /* igate */

} /* compute_clut_prob */

9.7 Spatial in-fill filter for the CMD clutter flag field

When the CMD flag field is computed, sometimes small gaps exist in range. If a clutter filter is
applied in such a ‘spotty’ manner, it can lead to speckles in the weather field.

It appears unlikely that applying a clutter filter across these gaps will cause serious harm, since
the adjacent gates will be filtered anyway. Filling in the gaps leads to a smoother result.

The following figure shows the circumstance under which this filter is applied.
The in-fill filter is designed to fill in gaps of the type shown above. Specifically, it will fill in the gaps of the following type:

- 1 un-flagged gate between adjacent flagged gates;
- 2 un-flagged gates with at least 2 flagged gates on either side;
- 3 un-flagged gates with at least 3 flagged gates on either side.

9.7.1 In-fill filter implementation example

```c
void applyCmdInfillFilter(Beam *beam) {
    int *countSet = (int *) malloc(beam->nGates * sizeof(int));
    int *countNot = (int *) malloc(beam->nGates * sizeof(int));

    /*
     * compute the running count of gates which have the flag set and
     * those which do not
     *
     * Go forward through the gates, counting up the number of gates set
     * or not set and assigning that number to the arrays as we go.
     */

    int nSet = 0;
    int nNot = 0;
    int igate, jgate;

    for (igate = 0; igate < beam->nGates; igate++) {
        if (beam->clutterFlag[igate]) {
            nSet++;
            nNot = 0;
        } else {
            nSet = 0;
            nNot++;
        }
        countSet[igate] = nSet;
        countNot[igate] = nNot;
    }
}
```
for (igate = beam->nGates - 2; igate >= 0; igate--) {
    if (countSet[igate] != 0 &&
        countSet[igate] < countSet[igate+1]) {
        countSet[igate] = countSet[igate+1];
    }
    if (countNot[igate] != 0 &&
        countNot[igate] < countNot[igate+1]) {
        countNot[igate] = countNot[igate+1];
    }
}

/*
 * fill in gaps
*/

for (igate = 1; igate < beam->nGates - 1; igate++) {
    
    /*
    * is the gap small enough?
    */
    
    nNot = countNot[igate];
    if (nNot > 0 && nNot <= CMD_MAX_NGATES_INFILLED) {
        
        /*
        * is it surrounded by regions at least as large as the gap?
        */
        
        int minGateCheck = igate - nNot;
        if (minGateCheck < 0) {
            minGateCheck = 0;
        }
        int maxGateCheck = igate + nNot;
        if (maxGateCheck > beam->nGates - 1) {
            maxGateCheck = beam->nGates - 1;
        }

        int nAdjacentBelow = 0;
        for (jgate = igate - 1; jgate >= minGateCheck; jgate--) {
            nSet = countSet[jgate];
            if (nSet != 0) {
                nAdjacentBelow = nSet;
            }
break;
}
} /* jgate */

int nAdjacentAbove = 0;
for (jgate = igate + 1; jgate <= maxGateCheck; jgate++) {
    nSet = countSet[jgate];
    if (nSet != 0) {
        nAdjacentAbove = nSet;
        break;
    }
} /* jgate */

int minAdjacent = nAdjacentBelow;
minAdjacent = MIN(minAdjacent, nAdjacentAbove);

if (minAdjacent >= nNot) {
    beam->clutterFlag[igate] = 1;
}
} /* igate */

free(countSet);
free(countNot);
}

9.8  NEXRAD spike filter

The NEXRAD spike filter is designed to remove spikes of reflectivity 1 or 2 gates wide. This helps to remove speckle in the output fields.

Clearly the ROC is aware of how the NEXRAD filter works and how it might be implemented. Nevertheless, this implementation is included for completeness.

9.8.1 NEXRAD spike filter implementation

/*******************************************************
 * NEXRAD spike filter implementation example
 *
 * This routine filters the reflectivity data according to the
 * NEXRAD specification DV1208621F, section 3.2.1.2.2, page 3-15.
 *
 * The algorithm is stated as follows:
 *
 * Clutter detection:
 *
 * The nth bin is declared to be a point clutter cell if its power value
 * exceeds those of both its second nearest neighbors by a threshold
 */
* value TCN. In other words:
*   if   $P(n)$ exceeds $TCN \times P(n-2)$
*       and  $P(n)$ exceeds $TCN \times p(n+2)$
* where
*   TCN is the point clutter threshold factor, which is always
*       greater than 1, and typically has a value of 8 (9 dB)
*   $P(n)$ if the power sum value for the nth range cell
*   n is the range gate number
* Clutter censoring:
* The formulas for censoring detected strong point clutter in an
* arbitrary array A via data substitution are as follows. If the nth
* range cell is an isolated clutter cell (i.e., it is a clutter cell but
* neither of its immediate neighboring cells is a clutter cell) then the
* replacement scheme is as follows:
*   Replace $A(n-1)$ with $A(n-2)$
*   Replace $A(n)$   with  $0.5 \times A(n-2) \times A(n+2)$
*   Replace $A(n+1)$ with $A(n+2)$
* If the nth and (n+1)th range bins constitute an isolated clutter pair,
* the bin replacement scheme is as follows:
*   Replace $A(n-1)$ with $A(n-2)$
*   Replace $A(n)$   with $A(n+2)$
*   Replace $A(n+1)$ with $A(n+3)$
*   Replace $A(n+2)$ with $A(n+3)$
* Note that runs of more than 2 successive clutter cells cannot occur
* because of the nature of the algorithm.
*/

```c
void applyNexradSpikeFilter(Beam *beam)
{

    int ii;

    /*
       * set clutter threshold
       */

    double tcn = 9.0;
```
/*
 * loop through gates
 */

for (ii = 2; ii < beam->nGates - 3; ii++) {

    /*
     * check for clutter at ii and ii + 1
     */

    int this_gate = 0, next_gate = 0;

    if ((beam->dbzf[ii] - beam->dbzf[ii - 2]) > tcn &&
        (beam->dbzf[ii] - beam->dbzf[ii + 2]) > tcn) {
        this_gate = 1;
    }

    if ((beam->dbzf[ii + 1] - beam->dbzf[ii - 1]) > tcn &&
          (beam->dbzf[ii + 1] - beam->dbzf[ii + 3]) > tcn) {
        next_gate = 1;
    }

    if (this_gate) {

        if (!next_gate) {
            /*
             * only gate ii has clutter, substitute accordingly
             */

            beam->dbzf[ii - 1] = beam->dbzf[ii - 2];
            beam->dbzf[ii + 1] = beam->dbzf[ii + 2];
            if (beam->dbzf[ii - 2] == CENSORED ||
                beam->dbzf[ii + 2] == CENSORED) {
                beam->dbzf[ii] = CENSORED;
            } else {
                beam->dbzf[ii] = beam->dbzf[ii - 2];
            }
        } else {
            /*
             * both gate ii and ii+1 has clutter, substitute accordingly
             */

            beam->dbzf[ii - 1] = beam->dbzf[ii - 2];
            beam->dbzf[ii] = beam->dbzf[ii - 2];
            beam->dbzf[ii + 1] = beam->dbzf[ii + 3];
            beam->dbzf[ii + 2] = beam->dbzf[ii + 3];
        }
    }
    }
/********************************************************
 * CLUTTER MITIGATION DECISION SYSTEM
 * DUALPOL VERSION 4.1
 *
 * C code as part of Functional Description
 *
 * Mike Dixon, NCAR, Boulder, CO, 80301
 *
 * November 2007
 *
 *********************************************************/

#include <stdlib.h>
#include <string.h>
#include <math.h>

#ifndef MAX
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#endif

#define MAX_GATES 2048
#define N_MAP_LOOKUP 10001

int NGATES_KERNEL_TDBZ = 9;
int NGATES_KERNEL_SPIN = 11;
int NGATES_KERNEL_SDEV_ZDR = 7;
int NGATES_KERNEL_SDEV_PHIDP = 7;

float SPIN_THRESHOLD = 6.5;
float SNR_THRESHOLD = 3.0;
float PROB_THRESHOLD = 0.5;

float TDBZ_WEIGHT  = 0.0;
float SPIN_WEIGHT = 0.0;
float MAX_TDBZ_SPIN_WEIGHT = 1.0;
float CPA_WEIGHT = 1.01;
float SDEV_ZDR_WEIGHT = 0.5;
float SDEV_PHIDP_WEIGHT = 0.5;

int CMD_MAX_NGATES_INFILLED = 3;

float CENSORED = -9999;

/******************************
 * Pulse implementation example
 *
 * Note that the meta-data time, prt, el and az are not actually
typedef struct {
    /* meta data */
    int nGates; /* number of gates */
    double time; /* time in secs and fractions from 1 Jan 1970 */
    double prt; /* pulse repetition time (secs) */
    double el; /* elevation angle (deg) */
    double az; /* azimuth angle (deg) */

    /* IQ data */
    float iq[MAX_GATES * 2];
} Pulse;

/*****************************/
/* Beam implementation example */
/* Note that the meta-data time, prt, el and az are not actually */
/* used in this code, they are just included for context. */
/*****************************/

typedef struct {
    /* meta data */
    int nSamples; /* number of pulse samples in beam */
    int nGates; /* number of gates */
    double time; /* time for the center pulse of beam */
    double prt; /* pulse repetition time (secs) */
    double el; /* elevation angle for center of beam (deg) */
    double az; /* azimuth angle for center of beam (deg) */

    /* Array of Pulse pointers */
    Pulse **pulses; /* Pointers to pulses with IQ data */

    /* arrays for pulses */
    float *snr; /* SNR in dB */
    float *dbz; /* reflectivity in dBZ */
    float *vel; /* velocity in m/s */
    float *zdr; /* zdr in dB */
    float *phidp; /* phidp in deg */
    float *dbzf; /* filtered dBZ */

    /* arrays for intermediate fields */
    float *dbzDiffSq;
    float *dbzSpinFlag;
/* arrays for CMD fields */

float *tdbz;
float *spin;
float *sdevZdr;
float *sdevPhidp;
float *cpa;

/* array for clutter probability */

float *clutterProb;

/* array for clutter decision flag:
*   1 means apply clutter filter
*   0 means no not apply clutter filter
*/

int *clutterFlag;

)

Beam;

/>

* create a beam
*
* (a) set meta data
* (b) allocate array memory
* (c) initialize values to censored or 0
*
* Pulse memory is not owned by the beam. It should be
* allocated and freed elsewhere.
*/

Beam *create_beam(int nSamples,
                  int nGates,
                  double time,
                  double prt,
                  double el,
                  double az,
                  Pulse **pulses) {

int isample, igate;

/* allocate space for beam */

Beam *beam = (Beam *) malloc(sizeof(Beam));

/* set meta data */

beam->nSamples = nSamples;
beam->nGates = nGates;
beam->time = time;
beam->prt = prt;
beam->el = el;
beam->az = az;

/* allocate space for arrays */
beam->pulses = (Pulse **) malloc(nSamples * sizeof(Pulse *));
beam->snr = (float *) malloc(nGates * sizeof(float));
beam->dbz = (float *) malloc(nGates * sizeof(float));
beam->vel = (float *) malloc(nGates * sizeof(float));
beam->zdr = (float *) malloc(nGates * sizeof(float));
beam->phidp = (float *) malloc(nGates * sizeof(float));
beam->dbzf = (float *) malloc(nGates * sizeof(float));
beam->dbzDiffSq = (float *) malloc(nGates * sizeof(float));
beam->dbzSpinFlag = (float *) malloc(nGates * sizeof(float));
beam->tdbz = (float *) malloc(nGates * sizeof(float));
beam->spin = (float *) malloc(nGates * sizeof(float));
beam->sdevZdr = (float *) malloc(nGates * sizeof(float));
beam->sdevPhidp = (float *) malloc(nGates * sizeof(float));
beam->cpa = (float *) malloc(nGates * sizeof(float));
beam->clutterProb = (float *) malloc(nGates * sizeof(float));
beam->clutterFlag = (int *) malloc(nGates * sizeof(int));

/* initialize arrays */

for (isample = 0; isample < nSamples; isample++) {
    beam->pulses[isample] = pulses[isample];
}

for (igate = 0; igate < nGates; igate++) {
    beam->snr[igate] = CENSORED;
    beam->dbz[igate] = CENSORED;
    beam->vel[igate] = CENSORED;
    beam->zdr[igate] = CENSORED;
    beam->phidp[igate] = CENSORED;
    beam->dbzf[igate] = CENSORED;
    beam->dbzDiffSq[igate] = CENSORED;
    beam->tdbz[igate] = CENSORED;
    beam->spin[igate] = CENSORED;
    beam->sdevZdr[igate] = CENSORED;
    beam->sdevPhidp[igate] = CENSORED;
    beam->cpa[igate] = CENSORED;
    beam->clutterProb[igate] = CENSORED;
    beam->clutterFlag[igate] = 0;
}

return beam;
}

/*
 * free memory associated with a beam
 */

void free_beam(Beam *beam) {
    free(beam->pulses);
    free(beam->snr);
    free(beam->dbz);
}
free(beam->vel);
free(beam->zdr);
free(beam->phidp);
free(beam->dbzf);
free(beam->dbzDiffSq);
free(beam->dbzSpinFlag);
free(beam->tdbz);
free(beam->spin);
free(beam->sdev2dr);
free(beam->sdevPhidp);
free(beam->cpa);
free(beam->clutterProb);
free(beam->clutterFlag);
free(beam);
}

/********************************************
 * Interest Map implementation example
 * *
 * Implemented as a lookup table.
 */

/* struct for points which define interest map */
typedef struct {
  double feature;
  double interest;
} ImPoint;

/* interest map struct */
typedef struct {
  int nPoints;
  ImPoint *points;
  double minFeature;
  double maxFeature;
  double deltaFeature;
  float *lut;
} InterestMap;

/* Initialize map by passing in the points which define the map
 * along with the weight for the map as a whole
 * *
 * Returns NULL on failure.
 */

InterestMap *create_map(ImPoint *pts,
                         int npts) {
  int ii, jj;
double slope;
InterestMap *map = NULL;

if (npts < 2) {
    /* error - must have at least 2 points to define function */
    return NULL;
}

/* allocate map itself */
map = malloc(sizeof(InterestMap));
/* clear map */
memset(map, 0, sizeof(InterestMap));
/* set meta data */
map->nPoints = npts;
/* allocate space for points which define map */
map->points = malloc(npts * sizeof(ImPoint));
/* copy point data */
memcpy(map->points, pts, npts * sizeof(ImPoint));
/* allocate space for lookup table */
map->lut = (float*) malloc(N_MAP_LOOKUP * sizeof(float));
/* compute min, max and delta */
map->minFeature = map->points[0].feature;
map->maxFeature = map->points[npts-1].feature;
map->deltaFeature =
    (map->maxFeature - map->minFeature) / (N_MAP_LOOKUP - 1.0);
/* compute slope for initial map segment */
jj = 1;
slope = ((map->points[jj].interest - map->points[jj-1].interest) /
    (map->points[jj].feature - map->points[jj-1].feature));
/* load lookup table */
for (ii = 0; ii < N_MAP_LOOKUP; ii++) {
    double interest;
    double val = map->minFeature + ii * map->deltaFeature;
    if ((val > map->points[jj].feature) && (jj < npts-1)) {
        /* move along by one segment */
        jj++;
        slope = ((map->points[jj].interest - map->points[jj-1].interest) /
            (map->points[jj].feature - map->points[jj-1].feature));
    }
}
/* compute interest, add to lookup table */

interest = map->points[jj-1].interest + 
    (val - map->points[jj-1].feature) * slope;

map->lut[ii] = interest;

} /* ii */

return map;

} /* create map */

/* free map memory */

void free_map(InterestMap *map) {
    free(map->points);
    free(map->lut);
    free(map);
}

/* look up interest given the feature value */

double getInterest(InterestMap *map, double feature) {
    int jj = (int) floor((feature - map->minFeature) / map->deltaFeature + 0.5);
    if (jj < 0) {
        jj = 0;
    } else if (jj > N_MAP_LOOKUP-1) {
        jj = N_MAP_LOOKUP-1;
    }
    return map->lut[jj];
}

/* Set up interest maps for each feature field */

InterestMap *tdbzMap;
InterestMap *spinMap;
InterestMap *sdevZdrMap;
InterestMap *sdevPhidpMap;
InterestMap *cpaMap;

ImPoint tdbzPoints[2] = {{20.0, 0.0}, {40.0, 1.0}};
ImPoint spinPoints[3] = {{15.0, 0.0}, {30.0, 1.0}};
ImPoint sdevZdrPoints[2] = {{1.2, 0.0}, {2.4, 1.0}};
ImPoint sdevPhidpPoints[2] = {{10.0, 0.0}, {15.0, 1.0}};
ImPoint cpaPoints[2] = {{0.6, 0.0}, {0.9, 1.0}};

void create_interest_maps() {

tdbzMap = create_map(tdbzPoints, 2);
spinMap = create_map(spinPoints, 3);
sdevZdrMap = create_map(sdevZdrPoints, 2);
sdevPhidpMap = create_map(sdevPhidpPoints, 2);
cpaMap = create_map(cpaPoints, 2);
}

/****************************************************************************
* TDBZ implementation example
* Compute TDBZ feature field
*/

/* Prepare for computing TDBZ by computing
* the gate-to-gate squared difference in Dbz
*/

void prepare_tdbz(Beam *beam) {
    int igate;
    
    /*
    * compute the squared difference in dBZ from gate to gate in
    * the beam.
    */
    for (igate = 1; igate < beam->nGates; igate++) {
        float prevDbz = beam->dbz[igate-1];
        float dbz = beam->dbz[igate];

        if (prevDbz != CENSORED && dbz != CENSORED) {
            float dbzDiff = dbz - prevDbz;
            beam->dbzDiffSq[igate] = dbzDiff * dbzDiff;
        }
    }
    /* use gate 1 value for gate 0 */
    beam->dbzDiffSq[0] = beam->dbzDiffSq[1];
}

/*
* Computing the TDBZ feature field
* 
* compute the mean squared difference over the kernel
*/

void compute_tdbz(Beam *beam) {
    int igate;
    double sum = 0.0;
    double nn = 0.0;
    double mean = 0.0;
for (igate = 0; igate < beam->nGates; igate++) {
    int startGate, endGate;
    int jbeam, jgate;

    /*
     * compute the start and end index, checking to make sure we
     * do not go off either end
     */
    startGate = igate - NGATES_KERNEL_TDBZ/2;
    if (startGate < 0) {
        startGate = 0;
    }

    endGate = igate + NGATES_KERNEL_TDBZ/2;
    if (endGate > beam->nGates - 1) {
        endGate = beam->nGates - 1;
    }

    /*
     * iterate over the kernel
     * ignore censored gates
     */
    for (jgate = startGate; jgate <= endGate; jgate++) {
        if (beam->dbzDiffSq[jgate] != CENSORED) {
            sum += beam->dbzDiffSq[jgate];
            nn += 1.0;
        }
    } /* jgate */

    if (nn > 0) {
        mean = sum / nn;
        beam->tdbz[igate] = mean;
    }

} /* igate */
} /* compute_tdbz() */

/******************************************************************************
 * SPIN implementation example
 * Compute SPIN feature field
 */

/*
 * Prepare for computing spin by setting the spin flag for
 * each gate
 */
void prepare_spin(Beam *beam) {

int igate;

/* first set spin flag for each gate */
for (igate = 1; igate < beam->nGates; igate++) {

    float dbzPrev = beam->dbz[igate-1];
    float dbzThis = beam->dbz[igate];
    float dbzNext = beam->dbz[igate+1];

    if (dbzPrev != CENSORED &&
        dbzThis != CENSORED &&
        dbzNext != CENSORED) {

        beam->dbzSpinFlag[igate] = 0;

        float prevDiff = dbzThis - dbzPrev;
        float nextDiff = dbzNext - dbzThis;

        if (prevDiff * nextDiff < 0) {
            /* sign change */
            float spinChange = (fabs(prevDiff) + fabs(nextDiff)) / 2.0;
            if (spinChange > SPIN_THRESHOLD) {
                beam->dbzSpinFlag[igate] = 1;
            } else {
                beam->dbzSpinFlag[igate] = 0;
            }
        }
    }
} /* if (beam->dbz[igate] != CENSORED ... */

} /* igate */

/* use nearest values for end gates */
beam->dbzSpinFlag[0] = beam->dbzSpinFlag[1];
beam->dbzSpinFlag[beam->nGates-1] = beam->dbzSpinFlag[beam->nGates-2];

} /* compute_spin */

/*
 * Compute the SPIN as a percentage over the kernel
 */

void compute_spin(Beam *beam) {

    double sum = 0.0;
    double nn = 0.0;
    double mean = 0.0;
    int igate;

    for (igate = 0; igate < beam->nGates; igate++) {

        int startGate, endGate;
        int jgate;

        ...
/* 
* compute the start and end index, checking to make sure we 
* do not go off either end 
*/

startGate = igate - NGATES_KERNEL_SPIN/2;
if (startGate < 0) {
    startGate = 0;
}

endGate = igate + NGATES_KERNEL_SPIN/2;
if (endGate > beam->nGates - 1){
    endGate = beam->nGates - 1;
}

/*
* iterate over the kernel
* ignore censored gates
*/

for (jgate = startGate; jgate <= endGate; jgate++) {
    float spinFlag = beam->dbzSpinFlag[jgate];
    if (spinFlag != CENSORED) {
        sum += spinFlag;
        nn += 1.0;
    }
} /* jgate */

if (nn > 0) {
    mean = sum / nn;
    beam->spin[igate] = mean * 100.0;
}

} /* igate */

} /* compute_spin */

/**************************************************************************
* CPA implementation example 
* Compute CPA feature field 
**************************************************************************/

void compute_cpa(Beam *beam) {
    int igate, isample;

    for (igate = 0; igate < beam->nGates; igate++) {
        Pulse *pulse = beam->pulses[igate];
        double sumMag = 0.0;
        double sumI = 0.0, sumQ = 0.0;
        double phasorLen;

        for (isample = 0; isample < beam->nSamples; isample++) {
float ii = pulse->iq[isample * 2];
float qq = pulse->iq[isample * 2 + 1];
sumI += ii;
sumQ += qq;
sumMag += sqrt(ii * ii + qq * qq);
} /* isample */
phasorLen = sqrt(sumI * sumI + sumQ * sumQ);
beam->cpa[igate] = phasorLen / sumMag;
} /* igate */
} /* compute_cpa */

/*****************************
 * SDEV_ZDR implementation example
 * Compute SDEV_ZDR
 * SDEV_ZDR is the standard deviation of ZDR over the kernel.
 * It is assumed that ZDR has already been computed for the beam.
 */
void compute_sdev_zdr(Beam *beam) {
  int igate;
  double sum = 0.0;
  double sumSq = 0.0;
  double nn = 0.0;
  double sdevZdr = 0.0;
  for (igate = 0; igate < beam->nGates; igate++) {
    int startGate, endGate;
    int jbeam, jgate;
    /*
     * compute the start and end index, checking to make sure we
     * do not go off either end
     */
    startGate = igate - NGATES KERNEL SDEV ZDR/2;
    if (startGate < 0) {
      startGate = 0;
    }
    endGate = igate + NGATES KERNEL SDEV ZDR/2;
    if (endGate > beam->nGates - 1) {
      endGate = beam->nGates - 1;
    }
    /*
     * iterate over the kernel
     * ignore censored gates
     */
    for (jgate = startGate; jgate <= endGate; jgate++) {
      float zdr = beam->zdr[jgate];
      
      /*
       * compute the start and end index, checking to make sure we
       * do not go off either end
       */
      startGate = igate - NGATES KERNEL SDEV ZDR/2;
      if (startGate < 0) {
        startGate = 0;
      }
      endGate = igate + NGATES KERNEL SDEV ZDR/2;
      if (endGate > beam->nGates - 1) {
        endGate = beam->nGates - 1;
      }
      /*
       * iterate over the kernel
       * ignore censored gates
       */
      for (jgate = startGate; jgate <= endGate; jgate++) {
        float zdr = beam->zdr[jgate];
      } /* jgate */
    } /* endGate */
  } /* igate */
} /* compute_sdev_zdr */
if (zdr != CENSORED) {
    sum += zdr;
    sumSq += (zdr * zdr);
    nn += 1.0;
}
} /* jgate */

if (nn >= 2) {
    double mean = sum / nn;
    double term1 = sumSq / nn;
    double term2 = mean * mean;
    if (term1 >= term2) {
        sdevZdr = sqrt(term1 - term2);
    } else {
        sdevZdr = 0.0;
    }
    beam->sdevZdr[igate] = sdevZdr;
}
} /* igate */
} /* compute_sdev_zdr */

/******************************************************************************
 * SDEV_PHIDP implementation example
 * Compute SDEV_PHIDP
 * SDEV_PHIDP is the standard deviation of PHIDP over the kernel.
 * It is assumed that PHIDP has already been computed for the beam.
 */

void compute_sdev_phidp(Beam *beam) {
    int igate;
    double sum = 0.0;
    double sumSq = 0.0;
    double nn = 0.0;
    double sdevPhidp = 0.0;
    for (igate = 0; igate < beam->nGates; igate++) {
        int startGate, endGate;
        int jbeam, jgate;

        /*
         * compute the start and end index, checking to make sure we
         * do not go off either end
         */
        startGate = igate - NGATES_KERNEL_SDEV_PHIDP/2;
        if (startGate < 0) {
            startGate = 0;
        }
        endGate = igate + NGATES_KERNEL_SDEV_PHIDP/2;
        if (endGate > beam->nGates - 1) {
            endGate = beam->nGates - 1;
        }
    }
/
* iterate over the kernel
* ignore censored gates
*/

for (jgate = startGate; jgate <= endGate; jgate++) {
    float phidp = beam->phidp[jgate];
    if (phidp != CENSORED) {
        sum += phidp;
        sumSq += (phidp * phidp);
        nn += 1.0;
    }
} /* jgate */

if (nn >= 2) {
    double mean = sum / nn;
    double term1 = sumSq / nn;
    double term2 = mean * mean;
    if (term1 >= term2) {
        sdevPhidp = sqrt(term1 - term2);
    } else {
        sdevPhidp = 0.0;
    }
    beam->sdevPhidp[igate] = sdevPhidp;
} /* igate */

} /* compute_sdev_phidp */

/**************************
* Clutter probability implementation example
* (a) compute clutter probability
* (b) set the clutter flag
*/

void compute_clut_prob(Beam *beam) {
    int igate;

    float tdbz, spin, sdevZdr, sdevPhidp, cpa;
    double tdbzInterest, spinInterest;
    double sdevZdrInterest, sdevPhidpInterest;
    double maxTdbzSpinInterest;
    double cpainterest;
    double clutterProb;
    int clutterFlag;

    for (igate = 0; igate < beam->nGates; igate++) {
        /* get the feature fields */
        tdbz = beam->tdbz[igate];

        /* get the feature fields */
        tdbz = beam->tdbz[igate];
spin = beam->spin[igate];
sdevZdr = beam->sdevZdr[igate];
sdevPhidp = beam->sdevPhidp[igate];
cpa = beam->cpa[igate];

/* convert feature fields to interest values */
tdbzInterest = getInterest(tdbzMap, tdbz);
spinInterest = getInterest(spinMap, spin);
sdevZdrInterest = getInterest(sdevZdrMap, sdevZdr);
sdevPhidpInterest = getInterest(sdevPhidpMap, sdevPhidp);
cpaInterest = getInterest(cpaMap, cpa);

/* compute max of tdbz and spin interest */
maxTdbzSpinInterest = MAX(tdbzInterest, spinInterest);

/* compute weighted sum */
double sum = 0.0;
double sumWt = 0.0;

sum =
  maxTdbzSpinInterest * MAX_TDBZ_SPIN_WEIGHT +
  sdevZdrInterest * SDEV_ZDR_WEIGHT +
  sdevPhidpInterest * SDEV_PHIDP_WEIGHT +
  cpaInterest * CPA_WEIGHT;

sumWt =
  MAX_TDBZ_SPIN_WEIGHT +
  SDEV_ZDR_WEIGHT +
  SDEV_PHIDP_WEIGHT +
  CPA_WEIGHT;

clutterProb = sum / sumWt;

if (clutterProb > PROB_THRESHOLD) {
    clutterFlag = 1;
} else {
    clutterFlag = 0;
}

beam->clutterProb[igate] = clutterProb;
beam->clutterFlag[igate] = clutterFlag;

} /* igate */

} /* compute_clut_prob */

/***********************************************************/
/* CMD in-fill filter implementation example */
/***********************************************************/

void applyCmdInfillFilter(Beam *beam) {

int *countSet = (int *) malloc(beam->nGates * sizeof(int));
int *countNot = (int *) malloc(beam->nGates * sizeof(int));

/*
* compute the running count of gates which have the flag set and
* those which do not
* Go forward through the gates, counting up the number of gates set
* or not set and assigning that number to the arrays as we go.
*/

int nSet = 0;
int nNot = 0;
int igate, jgate;

for (igate = 0; igate < beam->nGates; igate++) {
    if (beam->clutterFlag[igate]) {
        nSet++;
        nNot = 0;
    } else {
        nSet = 0;
        nNot++;
    }
    countSet[igate] = nSet;
    countNot[igate] = nNot;
}

/*
* Go in reverse through the gates, taking the max non-zero
* values and copying them across the set or not-set regions.
* This makes all the counts equal in the gaps and set areas.
*/

for (igate = beam->nGates - 2; igate >= 0; igate--) {
    if (countSet[igate] != 0 &&
        countSet[igate] < countSet[igate+1]) {
        countSet[igate] = countSet[igate+1];
    }
    if (countNot[igate] != 0 &&
        countNot[igate] < countNot[igate+1]) {
        countNot[igate] = countNot[igate+1];
    }
}

/*
* fill in gaps
*/

for (igate = 1; igate < beam->nGates - 1; igate++) {
    /*
    * is the gap small enough?
    */
    nNot = countNot[igate];
    if (nNot > 0 && nNot <= CMD_MAX_NGATES_INFILLED) {
/ * is it surrounded by regions at least as large as the gap? */
int minGateCheck = igate - nNot;
if (minGateCheck < 0) {
  minGateCheck = 0;
}
int maxGateCheck = igate + nNot;
if (maxGateCheck > beam->nGates - 1) {
  maxGateCheck = beam->nGates - 1;
}
int nAdjacentBelow = 0;
for (jgate = igate - 1; jgate >= minGateCheck; jgate--) {
  nSet = countSet[jgate];
  if (nSet != 0) {
    nAdjacentBelow = nSet;
    break;
  }
} /* jgate */
int nAdjacentAbove = 0;
for (jgate = igate + 1; jgate <= maxGateCheck; jgate++) {
  nSet = countSet[jgate];
  if (nSet != 0) {
    nAdjacentAbove = nSet;
    break;
  }
} /* jgate */
int minAdjacent = nAdjacentBelow;
minAdjacent = MIN(minAdjacent, nAdjacentAbove);
if (minAdjacent >= nNot) {
  beam->clutterFlag[igate] = 1;
}
} /* igate */
free(countSet);
free(countNot);

/*********************************************
* NEXRAD spike filter implementation example
* This routine filters the reflectivity data according to the
* NEXRAD specification DV1208621F, section 3.2.1.2.2, page 3-15.
* The algorithm is stated as follows:
* Clutter detection:
* The nth bin is declared to be a point clutter cell if its power value
* exceeds those of both its second nearest neighbors by a threshold
* value TCN. In other words:

*    if   \( P(n) \) exceeds TCN * \( P(n-2) \)
*    and  \( P(n) \) exceeds TCN * \( P(n+2) \)
* where

*    TCN is the point clutter threshold factor, which is always
*    greater than 1, and typically has a value of 8 (9 dB)
*    \( P(n) \) if the power sum value for the nth range cell
*    \( n \) is the range gate number

Clutter censoring:

* The formulas for censoring detected strong point clutter in an
* arbitrary array \( A \) via data substitution are as follows. If the \( n \)th
* range cell is an isolated clutter cell (i.e., it is a clutter cell but
* neither of its immediate neighboring cells is a clutter cell) then
* the replacement scheme is as follows:

*    Replace \( A(n-1) \) with  \( A(n-2) \)
*    Replace \( A(n) \)   with  0.5 * \( A(n-2) \) * \( A(n+2) \)
*    Replace \( A(n+1) \) with  \( A(n+2) \)

* If the \( n \)th and \( (n+1) \)th range bins constitute an isolated clutter
* pair,  
* the bin replacement scheme is as follows:

*    Replace \( A(n-1) \) with  \( A(n-2) \)
*    Replace \( A(n) \)   with  \( A(n+2) \)
*    Replace \( A(n+1) \) with  \( A(n+3) \)
*    Replace \( A(n+2) \) with  \( A(n+3) \)

* Note that runs of more than 2 successive clutter cells cannot occur
* because of the nature of the algorithm.

*/

void applyNexradSpikeFilter(Beam *beam)
{
    int ii;

    /*
     * set clutter threshold
     */

double tcn = 9.0;

    /*
     * loop through gates
     */
for (ii = 2; ii < beam->nGates - 3; ii++) {
  /*
   * check for clutter at ii and ii + 1
   */
  int this_gate = 0, next_gate = 0;

  if ((beam->dbzf[ii] - beam->dbzf[ii - 2]) > tcn &&
      (beam->dbzf[ii] - beam->dbzf[ii + 2]) > tcn) {
    this_gate = 1;
  }
  if ((beam->dbzf[ii + 1] - beam->dbzf[ii - 1]) > tcn &&
      (beam->dbzf[ii + 1] - beam->dbzf[ii + 3]) > tcn) {
    next_gate = 1;
  }

  if (this_gate) {
    if (!next_gate) {
      /*
       * only gate ii has clutter, substitute accordingly
       */
      beam->dbzf[ii - 1] = beam->dbzf[ii - 2];
      beam->dbzf[ii + 1] = beam->dbzf[ii + 2];
      if (beam->dbzf[ii - 2] == CENSORED ||
          beam->dbzf[ii + 2] == CENSORED) {
        beam->dbzf[ii] = CENSORED;
      } else {
        beam->dbzf[ii] = beam->dbzf[ii - 2];
      }
    } else {
      /*
       * both gate ii and ii+1 has clutter, substitute accordingly
       */
      beam->dbzf[ii - 1] = beam->dbzf[ii - 2];
      beam->dbzf[ii] = beam->dbzf[ii - 2];
      beam->dbzf[ii + 1] = beam->dbzf[ii + 3];
      beam->dbzf[ii + 2] = beam->dbzf[ii + 3];
    }
  }
}
} /* ii */
5B.6 REAL TIME CLUTTER IDENTIFICATION AND MITIGATION FOR NEXRAD

John C. Hubbert,* Mike Dixon and Cathy Kessinger
National Center for Atmospheric Research, Boulder CO

1. INTRODUCTION

Mitigation of anomalous propagation clutter (AP) is an ongoing problem in radar meteorology. AP clutter routinely contaminates radar data masking weather returns causing poor data quality. The problem is typically mitigated by applying a clutter filter to all radar data but this also eliminates weather data at zero velocity. With the advent of fast digital receivers capable of real time spectral processing, the real time identification and elimination of AP clutter is now possible. A fuzzy logic algorithm is used to distinguish between clutter echo and precipitation echoes. Based on this classification, clutter filters can be applied to only those radar resolution volumes where clutter is present, in real time. In this way weather echoes are preserved while clutter echoes are mitigated. The clutter filters used in this paper are spectral based, i.e., they are applied in the spectral domain. In many cases, after the clutter echoes are filtered, the underlying weather echo signatures are revealed thereby significantly increasing the visibility of weather echo. This paper describes the Fuzzy Logic algorithm for clutter echo identification and the technique is illustrated with experimental data from the Denver NEXRAD KFTG and S-Pol, NCAR’s (National Center for Atmospheric Research) S-band polarimetric radar.

2. BACKGROUND

Clutter filters for weather radars typically operate in the time domain, i.e., the digitized I and Q samples (in phase and quadrature) are passed through some type of IIR (Infinite Impulse Response) filter. These filters work fairly well typically yielding 50 dB of clutter rejection or better. Different stop-band filter widths are possible but the clutter filter is typically applied to all of the radar data, i.e., the clutter filters are either on or off all the time. If the clutter filters are on all the time, weather signals along the zero velocity isodop are also be removed. Additionally, as weather conditions vary, AP clutter can appear and subsequently disappear. Radar operators have attempted to monitor AP clutter and then turn on a clutter filter when the AP conditions were significant. Later the clutter filter is turned off. Such human driven decisions are prone to error. Ideally one wants to filter only those radar gates that are clutter-contaminated and only when the clutter power dominates any overlaying weather signal. This is why trying to apply a clutter filter to those gates that lie on a previously constructed map of NP (Normal Propagation) clutter can eliminate important weather signals. For example, there may be a radar gate that shows 35 dBZ on a clutter map but then it is overlaid with a 40 dBZ or greater weather signal. In this case, a clutter filter likely should not be applied. The solution is to use signal processing to identify those gates that are dominated by clutter and then to apply a clutter filter to those gates in real time.

2.1 Spectral Clutter Filters

The new generation of radar processors now have enough processing power to calculate spectra, process them, and subsequently apply a clutter filter if needed. The I and Q samples can be put into a buffer while the data is processed and clutter affected gates are identified. After identification, the buffered data is clutter filtered. The additional processing power also allows for cluttering filtering in the “frequency” or velocity domain, i.e., the I and Q samples are Fourier transformed via a FFT algorithm after which the signal spectrum can be processed. The new spectral clutter filters not only remove power around zero velocity in the spectra but also use an interpolation scheme to fill in the spectral points that were “notched” out (i.e., set to zero) (Siggia and Passarelli 2004). Such filters can first adaptively set a filter notch width according to the characteristics of the spectrum, and then fit a Gaussian shaped curve to the remaining assumed weather signal. The fitted Gaussian curve is used to interpolate across the notch left in the spectrum due to the clutter filter. Figures 1 and 2 illustrate this process. Shown in Fig. 1 are a weather spectrum (red), a clutter signal spectrum (green) and the combination of these two spectra (blue). Fig. 2 shows the same spectra but with the Gaussian fitted curve in red dots and the resultant clutter filtered spectrum in black. If the interpolated area (around zero velocity) were replaced with a notch (set to zero), an obvious bias in both power and velocity estimates of the weather echo would occur. Such an adaptive spectral based clutter filter is used in this paper.

3. FUZZY LOGIC IDENTIFICATION

To identify the gates that are contaminated with clutter, a Fuzzy Logic based algorithm termed CMD (Clutter Mitigation Decision) is employed. First it it noted that narrow spectrum width, zero velocity weather echoes (such as from stratiform rain) are

* Correspondence to: hubbert@ucar.edu
very difficult to distinguish from clutter based solely on their spectra. To identify clutter echoes and distinguish them from narrow spectrum width, zero velocity weather echoes, three variables are used: 1) spatial texture of reflectivity, 2) the so-called SPIN of reflectivity (Steiner and Smith 2002) and 3) the Clutter Phase Alignment (CPA). The texture of the reflectivity (TDBZ) is computed as the mean of the squared reflectivity difference between adjacent gates,

\[
TDBZ = \left[ \frac{1}{N} \sum_{i=1}^{L} \left( \sum_{j=1}^{M} (dBZ_{i,j} - dBZ_{i,j-1})^2 \right) \right]^{1/2}
\]

where \(dBZ\) is the reflectivity, \(L\) is the number of radar beams or rays used, \(M\) is the number gates used and \(N = L \times M\). TDBZ is computed at each gate along the radial with the computation centered on the gate of interest. SPIN is a measure of how often the reflectivity gradient changes sign along a direction in space (in this case the radar radial) (Steiner and Smith 2002).

The Clutter Phase Alignment (CPA) is calculated from a length \(L\) time series \(x_i\) as

\[
CPA = \left| \frac{1}{L} \sum_{i=1}^{L} x_i \right| / \left[ \frac{1}{L} \sum_{i=1}^{L} |x_i| \right]^{1/2}
\]

If the arg\{\(x_i\)\} is the same for all \(x_i\) then \(CPA = 1\). For clutter, \(CPA\) is usually greater than 0.95 and is about zero for noise. Only weather echoes with velocity magnitude < 0.3 m/s (approximately) and spectrum widths less than about 0.5 m/s have CPA values that are in the 0.95 range. Thus, the texture fields are needed to distinguish these echoes from clutter. In any event, CPA is nearly always significantly less than 0.95 for weather time series that are collected over times that are significantly longer than the decorrelation time of the precipitation particles in the radar resolution volume. This variable was first used in (Fabry 2004) as a measure of clutter phase stability for the calculation of refractivity from radar returns.

CMD also uses a power ratio calculated from the spectra termed Power Ratio Narrow (PRN). PRN is the ratio of power from the spectrum at and close to 0 velocity, typically the 0 velocity point and one point on either side, to the power from two points adjacent on each side of this (four total spectral points). If this ratio is large, clutter is possibly present, if not, then a clutter filter is not applied. Thus, this ratio is used as a hard decision boundary. The other hard decision boundary used is the SNR (signal-to-noise ratio). The Fuzzy Logic inputs are centered around the radar resolution volume of interest (Dixon et al. 2006). The general steps of the CMD algorithm are as follows:

1. Check SNR > 3 dB, otherwise no filtering at this gate.
2. Compute clutter ratio narrow (CRN).
3. Check CRN > 6 dB, otherwise no filtering at this gate.
4. Compute feature fields, using a kernel of 1 beam wide by 9 gates long of dBZ texture (TDBZ), dBZ SPIN and clutter phase alignment (CPA)
5. Apply interest mapping to convert features to interest fields.
6. Compute CMD field by applying Fuzzy Logic to interest fields.
7. Threshold CMD at 0.5 to produce CMD clutter flag.
8. Apply clutter filter where CMD flag is set.

4. EXPERIMENTAL DATA

The following one half degree elevation angle PPI data were gathered with KFTG on 26 October 2006 in a wide-spread snow storm along the Eastern Foothills of the Rocky Mountains in Colorado. The times series data were gathered and the CMD algorithm was run during post processing. However, the algorithm is designed to run in real time and does so now on S-Pol. Figs. 3 and 4 show unfiltered reflectivity and velocity, respectively. The x- and y-axes span 250 km and the range rings are in 30 km increments. The Rocky Mountains are easily seen in the left portion of the PPIs. Peak reflectivities are about 40 dBZ in the storm (wet snow) while the reflectivity due to the mountain clutter is in excess of 65 dBZ. The velocity plot shows a clear 0 velocity isodop through the center of the plot (in gray). It is this 0 velocity weather that the CMD should not identify as clutter. Fig. 5 is clear air reflectivity which shows the location of clutter, i.e., it is a clutter map for the region displayed in Figs. 3 and 4 and is shown for reference. Fig. 5 shows the resulting reflectivity when a clutter filter is simply applied everywhere. Note the reflectivity that has been eliminated along the 0 velocity isodop. This shows the problem when clutter filters are applied everywhere.

Figures 7, 8, 9 and 10 show the feature fields of TDBZ, SPIN, CPA and CRN for the data of Fig. 3. Note how well the CPA feature field identifies the clutter. These feature fields are used by CMD to create the clutter map shown in Fig. 11 with yellow marking the regions to be clutter filtered and this can be compared to Fig. 5. A spectral based clutter filter is now applied to the data at the gates indicated by Fig. 11 and the resulting reflectivity and velocity PPIs are shown in Fig. 12 and 13, respectively, and should be compared to Figs. 3 and 4. Both the reflectivity and velocity fields are by-in-large preserved while the clutter has been eliminated.

4.1 Dual polarization data

The above clutter mitigation algorithm was designed for use with single polarization data, however, provision was also made for the inclusion of dual polarization parameters. The dual polarization inputs can be either "activated" or "deactivated" depending on the type of data processed. It was found that the spatial texture of copolar differential reflectivity
\( Z_{dr} \) and copolar differential phase \( \phi_{dp} \) are excellent discriminators of clutter and weather. In our tests we found that 7 or 9 gates of data along a radial produced good discrimination between weather and clutter echoes. It was also found that the copolar correlation coefficient, \( \rho_{hv} \), was not as good a discriminator as the standard deviations of \( Z_{dr} \) and \( \phi_{dp} \) and therefore is not included in the Fuzzy Logic algorithm.

The following data was gathered by S-Pol on 5 May 2006 in dual polarization mode. Figure 14 shows non-clutter filtered reflectivity. Figure 15 shows the clutter map created by the dual polarization CMD algorithm. Clutter filters are then applied to region designated in Fig.15 and the resulting clutter filtered reflectivity is shown in Fig. 16. As can be seen the clutter is nicely mitigated leaving the meteorological echoes of interest.

5. CONCLUSIONS

The Fuzzy Logic based Clutter Mitigation Decision (CMD) algorithm can effectively identify clutter in radar data. If a high speed processor is used, the CMD identified clutter contaminated data can be clutter filtered in real time. After the clutter filter has been applied to the data the radar moments can then be recalculated so that any remaining weather data may be revealed. The CMD algorithm used here employs a new feature field called Clutter Phase Alignment (CPA) which is an excellent discriminator of clutter.

Acknowledgment This research was supported by the ROC (Radar Operations Center) of Norman OK.

References

Dixon, M., C. Kessinger, and J.C. Hubbert, 2006: Echo classification within the spectral domain to discriminate ground clutter from meteorological targets. 86^{th} AMS Annual Meeting, IIPS, Atlanta, GA.


Figure 3: Unfiltered reflectivity.

Figure 4: Unfiltered velocity.
Figure 5: PPI clutter map for KFTG.

Figure 6: Reflectivity filtered everywhere for data of Fig. 3.
Figure 7: The texture of the reflectivity field (TDBZ) shown in Fig. 3.

Figure 8: The feature field SPIN for data of Fig. 3.
Figure 9: The feature field Clutter Phase Alignment (CPA) for data of Fig. 3.

Figure 10: The field Clutter Ratio Narrow for data of Fig. 3.
Figure 11: The CMD flag, i.e., the clutter filter is applied at those gates in the yellow regions.

Figure 12: CMD filtered dBZ for data of Fig. 3.
Figure 13: CMD filtered velocity for data of Fig. 3.

Figure 14: Dual pol. case, reflectivity.
Figure 15: CMD dual pol. clutter flag.

Figure 16: Dual pol. reflectivity filtered.
1. INTRODUCTION

Echoes from normal propagation ground clutter (NP) as well as anomalous propagation ground clutter (AP) contaminate weather echoes so that precipitation estimates based on such contaminated echoes are biased. The problem is typically mitigated by applying a clutter filter to all radar data but this also eliminates weather data at zero velocity. With the advent of fast digital receivers, the real-time identification and filtering of clutter is now possible. To do this, fuzzy logic is used to distinguish between clutter echoes and precipitation echoes. Based on this classification, clutter filters can be applied to only those radar resolution volumes where clutter is present, in real time. In this way weather echoes are preserved while clutter echoes are mitigated. The clutter filters used in this paper are spectral based, i.e., they are applied in the spectral domain. In many cases, after the clutter echoes are filtered, the underlying weather echo signatures are revealed thereby significantly increasing the visibility of weather echo. This paper describes the Fuzzy Logic algorithm for clutter echo identification and the technique is illustrated with experimental data from the Denver NEXRAD KFTG and S-Pol, NCAR’s (National Center for Atmospheric Research) S-band polarimetric radar.

2. BACKGROUND

Clutter filters for weather radars typically operate in the time domain, i.e., the digitized I and Q samples (in-phase and quadrature) are passed through some type of IIR (Infinite Impulse Response) filter. These filters work fairly well typically yielding 50 dB of clutter rejection or better. Different stop-band filter widths are possible but the clutter filter is typically applied to all of the radar data, i.e., the clutter filters are either on or off all the time. If the clutter filters are on all the time, weather signals along the zero velocity isodop are also removed. Additionally, as weather conditions vary, AP clutter can appear and subsequently disappear. To avoid necessary clutter filter usage, radar operators have attempted to monitor AP clutter and then turn on a clutter filter when the AP conditions were significant. Later the clutter filter is turned off. Such human-driven decisions are prone to error. Ideally the clutter filter is applied only to radar gates that are clutter-contaminated and only when the clutter power can affect the radar signatures due to an overlaying weather signal. Applying a clutter filter to those gates that lie on a previously constructed map of NP (Normal Propagation) clutter can eliminate important weather signals. For example, there may be a radar gate that shows 35 dBZ on a clutter map that is overlaid with a 45 dBZ or greater weather signal. In this case, a clutter filter likely should not be applied, particularly if the weather velocity is close to zero. The solution is to use signal processing to identify those gates that are dominated by clutter and then to apply a clutter filter to those gates in real time.

2.1 Spectral Clutter Filters

The new generation of radar processors now have enough processing power to calculate spectra, process them, and subsequently apply a clutter filter if needed. The I and Q samples can be put into a buffer while the data is processed and clutter-affected gates are identified. After identification, the buffered data is filtered. The additional processing power also allows for cluttering filtering in the “frequency” or velocity domain, i.e., the I and Q samples are Fourier transformed via an FFT algorithm after which the signal spectrum can be processed. The new spectral clutter filters not only remove power around zero velocity in the spectra but also use an interpolation scheme to fill in the spectral points that were “notched” out (i.e., set to zero) (Siggia and Passarelli 2004). Such filters can first adaptively set a filter notch width according to the characteristics of the spectrum, and then fit a Gaussian shaped curve to the remaining assumed weather signal. The fitted Gaussian curve is used to interpolate across the notch left in the spectrum due

*NCAR/EOL, Boulder, Colorado 80307, email: hubbert@ucar.edu
to the clutter filter. Figures 1 and 2 illustrate this process. Shown in Fig. 1 are a weather spectrum (red), a clutter signal spectrum (green) and the combination of these two spectra (blue). Fig. 2 shows the same spectra but with the Gaussian-fitted curve in red dots and the resultant clutter filtered spectrum in black. If the interpolated area (around zero velocity) were replaced with a notch (set to zero), an obvious bias in both power and velocity estimates of the weather echo would occur. Such an adaptive spectral based clutter filter is used in this paper.

3. FUZZY LOGIC IDENTIFICATION

To identify the gates that are contaminated with clutter, a Fuzzy Logic based algorithm termed CMD (Clutter Mitigation Decision) is employed. First it is noted that narrow spectrum width, zero velocity weather echoes (such as from stratiform rain) are very difficult to distinguish from clutter based solely on their spectra and thus spatial textures of various radar measureables are used in order to make this distinction.

3.1 Single Polarization Algorithm

To identify clutter echoes and distinguish them from narrow spectrum width, zero velocity weather echoes, three variables, or feature fields, are used in the single polarization case: 1) spatial texture of reflectivity, 2) the so-called SPIN of reflectivity (Steiner and Smith 2002) and 3) the Clutter Phase Alignment (CPA). The texture of the reflectivity (TDBZ) is computed as the mean of the squared reflectivity difference between adjacent gates,

$$TDBZ = \left[ \frac{1}{N} \sum_{i=1}^{L} \sum_{j=1}^{M} (dBZ_{i,j} - dBZ_{i-1,j})^2 \right] / N$$

where dBZ is the reflectivity, L is the number of radar beams or rays used, M is the number gates used and $N = L \times M$. In this paper, L is equal to one, i.e., only data along a single radar radial is used to calculate TDBZ (and SPIN), which eliminates the need to buffer adjacent beam information into memory and significantly reduces the algorithm complexity over the use of 2-D computations. The TDBZ feature field is computed at each gate along the radial with the computation centered on the gate of interest. The SPIN feature field is a measure of how often the reflectivity gradient changes sign along a direction in space (in this case the radar radial) (Steiner and Smith 2002).

3.2 Clutter Phase Alignment

The Clutter Phase Alignment (CPA) is calculated from a length N time series $x_i$ as

$$CPA = \left[ \sum_{i=1}^{N} x_i \right] / \left[ \sum_{i=1}^{N} |x_i| \right]$$

If the $\arg\{x_i\}$ is the same for all $x_i$ then $CPA = 1$. Thus, CPA is the magnitude of the vector sum of the individual time series members divided by the sum of the magnitudes of the $x_i$. CPA is an excellent indicator/identifier of clutter since by definition it is a measure of the primary characteristic of a stationary ground clutter target, i.e., constant backscatter phase. In fact, if the phase of the $x_i$ is a constant, CPA will be one regardless of the behavior of the magnitude of the $x_i$. CPA is a measure of how constant the absolute return phase (i.e., the phase of a received I and Q sample) is for a resolution volume. For a fixed, non moving target, CPA is 1. If the target is not completely stationary over the measurement period, the mean velocity will differ from 0 ms$^{-1}$ and/or the width of the spectrum of the radar return signal will increase. Both nonzero mean velocity and increased spectrum width will decrease CPA to below 1. The more constant the absolute phase is, the more likely it is that the gate contains clutter. For pure clutter, CPA is usually greater than 0.95 while it is about zero for noise and weather with mean velocity magnitude greater than 1 ms$^{-1}$ (ignoring the possibility of velocities that “wrap” back to 0 ms$^{-1}$). Only weather echoes with velocity magnitude < 0.3 ms$^{-1}$ (approximately) and spectrum widths less than about 0.5 ms$^{-1}$ have CPA values close to 0.95. Thus, the reflectivity texture and SPIN fields are needed to distinguish these echoes from clutter. In any event, CPA is nearly always significantly less than 0.95 for weather time series that are collected over times that are significantly longer than the decorrelation time of the precipitation particles in the radar resolution volume.

There is a close relationship between CPA and the velocity and spectrum width. Figure 3 show this relationship. Time series simulations were made for velocities from $-0.6$ ms$^{-1}$ to $0.6$ ms$^{-1}$ at 0.1 ms$^{-1}$ steps, 100 simulations per step. The simulation parameters are: $PRT=1$ ms, $\sigma_v = 0.1$ ms$^{-1}$, 64 points, and SNR=60 dB. For each simulated time series the velocity is estimated via the pulse pair algorithm, CPA is calculated and the result is shown in Figure 3. As can be seen, as velocity magnitude increases,
the value of CPA decreases rather quickly. When the velocity magnitude is greater than 0.2 ms\(^{-1}\), the average CPA values are below 0.9. Both velocity and spectrum affect CPA: 1) as the velocity departs from zero, CPA decreases rapidly, 2) as the spectrum width increases the value of CPA decreases. There are well known variances associated with the pulse pair velocity estimator and especially the width estimator, furthermore, no particular spectrum shape is assumed for the CPA calculation as is done with the pulse pair estimators. Thus, CPA is a more robust indicator of clutter than radial velocity and spectrum width.

There is another interesting relationship between CPA and the spectrum of the signal. The numerator of CPA is identical to the 0 velocity component of the discrete Fourier transform (DFT) the signal (within a scale factor N). Thus, it is natural to compare CPA and the spectrum of the signal. The numerator from zero, CPA decreases rapidly, 2) as the spectrum width increases the value of CPA decreases. There are well known variances associated with the pulse pair velocity estimator and especially the width estimator, furthermore, no particular spectrum shape is assumed for the CPA calculation as is done with the pulse pair estimators. Thus, CPA is a more robust indicator of clutter than radial velocity and spectrum width.

The inequality we hypothesize is

\[
CPA \geq \sqrt{PR}
\]

The right hand side is rewritten in terms of \(x_i\) as (using Parseval)

\[
\frac{\sum_{i=1}^{N} x_i}{\sum_{i=1}^{N} |x_i|} \geq \sqrt{\frac{\sum_{m=1}^{N} |X_m|^2}{N \sum_{m=1}^{N} |X_m|^2}}
\]

The numerators are equal and the inequality can be rewritten as

\[
\left[ \sum_{i=1}^{N} x_i \right] \leq \sqrt{\sum_{i=1}^{N} |x_i|^2}
\]

Dividing both sides by N gives

\[
\frac{\sum_{i=1}^{N} x_i}{N} \leq \sqrt{\frac{\sum_{i=1}^{N} |x_i|^2}{N}}
\]

This simply states that the root mean square average is greater than the arithmetic mean, a well known result. Thus we have proven the inequality of Eq. (5). The result indicates that CPA should be a better discriminator of clutter than the PR. To illustrate this, examine Fig. 4 which shows a scatter plot of CPA versus \(PR^{0.5}\) for a low level PPI scan that contains both clutter and weather echoes collected by S-Pol. As can be seen, there exist many resolution volumes where the \(PR^{0.5}\) is in the 0.5 to 0.7 range while CPA is close to one. Figures 5 and 6 show the times series of \(I^2 + Q^2\)dBm and \(\tan^{-1}(Q/I)\) degrees for a resolution volume with clutter where CPA=0.96 but \(PR^{0.5} = 0.59\) (time series length 64). As can be seen the clutter target only becomes visible to the scanning radar about half-way through the collected time series. At this point the phase becomes fairly stable at around \(-135^\circ\). The difference between CPA and \(PR^{0.5}\) can be explained as follows. Since the ground clutter target is only visible through about half of the I and Q samples, the power time series varies dramatically from -85 dBm to -55 dBm during which the phase remains fairly constant. This sharp gradient in the power time series spreads power away from the zero velocity component in the spectrum of the signal thereby reducing the the power ratio \(PR^{0.5}\). This characteristic makes CPA a better discriminator of clutter for such scanned ground clutter targets.

### 3.2 Fuzzy Logic Algorithm

The Fuzzy Logic membership functions used in CMD are shown in Fig. 7. The membership functions are applied to TDBZ, SPIN and CPA resulting in so-called interest fields varying from 0 to 1 with 1 indicating the strongest clutter signal for the given variable. The TDBZ and SPIN interest field are then combined with a fuzzy “or” rule: the maximum interest value of TDBZ and SPIN is selected. The two remaining interest fields, MAX(TDBZ,SPIN) and CPA are multiplied by a-priori weights of 1.0 and 1.01 respectively and normalized by the sum of the weights. The weighted sum of interest values yields a probability of clutter between 0 and 1. Finally gates with values greater than 0.5 probability are classified as clutter. The chosen weight of 1.01 for CPA can be justified as follows. For certain cases MAX(TDBZ,SPIN) is zero while the CPA interest is one. In this case the normalized weighted sum (or probability) would be 0.5 if both interest fields had weights of one, and the point would be classified as not clutter. By giving CPA a weight of 1.01, such cases are classified as clutter. Through experience, this has proven to
improve the performance of the CMD algorithm.

The general steps of the CMD algorithm are as follows:

1. Check if SNR > 3 dB, otherwise no filtering applied at this gate.
2. Compute feature fields texture of dBZ (TDBZ), dBZ SPIN and clutter phase alignment (CPA). The kernel for TDBZ computation is 1 beam by 9 range gates and for SPIN is 1 beam by 11 range gates.
3. Apply interest mapping to convert feature fields to interest values.
4. Compute CMD field by applying Fuzzy Logic to interest fields.
5. Combine TDBZ and SPIN interest fields using fuzzy or rule (maximum interest).
6. Compute normalized weighted sum of interest values.
7. Threshold CMD at 0.5 to produce CMD clutter flag.
8. Apply clutter filter where CMD flag is set.

3.3 Dual polarization Fuzzy Logic

The above clutter mitigation algorithm was designed for use with single polarization data; however, provision was also made for the inclusion of dual polarization parameters. The dual polarization inputs can be either “activated” or “deactivated” depending on the type of data processed. It was found that the spatial texture of copolar differential reflectivity ($Z_{dr}$) and copolar differential phase ($\phi_{dp}$) are excellent discriminators of clutter and weather. In our tests we found that kernels of 7 to 11 gates of data along a radial produced good discrimination between weather and clutter echoes, depending on the parameter. It was also found that the copolar correlation coefficient, $\rho_{hv}$, was not as good a discriminator as the textures of $Z_{dr}$ and $\phi_{dp}$ and therefore is not included in the Fuzzy Logic algorithm. The parameters used in the dual polarization CMD algorithm are summarized in Table 1 while the membership functions and the weights are summarized in Table 2. The membership functions for $\sigma_{zdr}$ and $\sigma_{\phi_{ap}}$ are given in Fig. 8.

4. EXPERIMENTAL DATA

4.1 Single polarization data

The following one half degree elevation angle PPI data were gathered with KFTG on 26 October 2006 in a wide-spread snow storm along the Eastern Foothills of the Rocky Mountains in Colorado. The times series data were gathered and the CMD algorithm was run during post processing. (However, the algorithm is designed to run in real time and currently does so on S-Pol.) Figs. 9 and 10 show unfiltered reflectivity and velocity, respectively. The x- and y-axes span 250 km and the range rings are in 30 km increments. The Rocky Mountains are easily seen in the left portion of the PPIs. Peak reflectivities are about 40 dBZ in the storm (wet snow) while the reflectivity due to the mountain clutter is in excess of 65 dBZ. The velocity plot shows a clear 0 velocity isodop through the center of the plot (in gray). The reflectivity shows areas marked by black lines that indicate the location of the zero velocity isodop. The velocity field shows areas where the velocity has folded back to 0 ms$^{-1}$ (again in gray). The power in these areas will be severely attenuated if a clutter filter is applied. It is these 0 velocity weather areas that the CMD should not identify as clutter. Fig. 11 is a 0.5$^\circ$ clear air PPI scan and thus is a clutter map for the region displayed in Figs. 9 and 10. It is shown for reference. Fig. 12 shows the resulting reflectivity when a clutter filter is simply applied everywhere. Note that the reflectivity has been eliminated not only along the 0 velocity isodop but also in the areas of 0 velocity folded echoes as indicated in Fig. 10. Figure 12 is actual Archive 2 (A2) data downloaded from the NWS archive site: i.e., this is the data that was actually viewed by NWS forecasters and used by algorithms including the precipitation accumulation. Large errors are apparent due to the clutter filters being applied everywhere. This shows the problem when clutter filters are applied everywhere.

The CMD algorithm was used to create the clutter map shown in Fig. 13 with yellow marking the regions to be clutter filtered and this can be compared to Fig. 11. A spectral based clutter filter is now applied to the data at the gates indicated by Fig. 13 and the resulting reflectivity PPI is shown in Fig. 14. This should be compared to Fig. 9. The CMD clutter filtered reflectivity demonstrates the enormous improvement in data quality when compared to the actual A2 data of Fig. 12.

4.2 Performance Evaluation

To evaluate the performance of the CMD algorithm in identifying clutter, the SNR (signal-to-clutter ratio) is calculated for each gate. The
clutter power is estimated as the power removed by
the clutter filter. The remaining power is considered
to be weather power. Gates with near zero velocity
(i.e., $|\text{vel}| \leq 2$ ms$^{-1}$) are excluded. The fraction of
range gates CMD identifies for filtering as a function
of the clutter to signal ratio (CSR) is determined.
The results are shown in Fig. 15. The red line
indicates the fraction of gates that were unfiltered
whereas the blue line indicates the fraction of gates
where the clutter filter was applied. One minus the
blue line value gives the red line value. As can be
seen the crossover point is located at about -8 dB
CSR, that is about 50% of the gates with with
CSR= -8 dB are clutter filtered. Since clutter with
CSR of -10 dB can bias weather velocity estimates
by 2 to 3 ms$^{-1}$ (depending on the true velocity
of the weather) being able to identify weather
contaminated by clutter with CSR down to -10 dB
is a desirable goal. Ideally the crossover point would
be at a CSR less than -10 dB but identifying clutter
at such low CSR is difficult: the weather echoes
dominate the calculated feature fields. Nevertheless,
the CMD performance is excellent especially when
considered against the alternative of applying the
clutter filter everywhere.

4.2 Dual Polarization Data

The following data was gathered by S-Pol on 24
April 2007 in dual polarization mode. Figure 16
shows non-clutter filtered reflectivity while Fig. 17
shows non-clutter filtered velocity. Figure 18 shows
clear air reflectivity and thus provides a NP clutter
map for reference. To illustrate the discrimination
capability of each of the Fuzzy Logic input variables,
Figs. 19, 20, 21 and 22 show the interest fields of
CPA, $\sigma_{\text{zdr}}$, $\sigma_{\phi_{dp}}$ and maximum(TDBZ, SPIN), re-
psectively. All interests fields are very good clutter
indicators with some being better than others in par-
ticular regions. Taken together, the interest fields
provide a robust indicator of clutter.

Figure 23 shows the clutter map created by the
dual polarization CMD algorithm. Clutter filters
are then applied to the region designated in Fig. 23
and the resulting clutter filtered reflectivity is shown
in Fig. 24. As can be seen the clutter is nicely
mitigated leaving the meteorological echoes intact.

5. CONCLUSIONS

The Fuzzy Logic based Clutter Mitigation Deci-
sion (CMD) algorithm can effectively identify clut-
er in radar data. If a high speed processor is used,
the CMD identified clutter contaminated data can be
clutter filtered in real time. After the clutter filter
has been applied to the data the radar moments can
then be recalculated so that any remaining weather
data may be revealed. The CMD algorithm used
here employs a new feature field called Clutter Phase
Alignment (CPA) which is an excellent discrimina-
tor of clutter. CMD has been designed to work with
both both single as well as dual polarization data.

Acknowledgment

This research was supported by the ROC (Radar Operations Center) of Norman OK. The National Center for Atmospheric Research is sponsored by the National Science Foundation. Any opinions, findings and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

Dixon, M., C. Kessinger, and J.C. Hubbert, 2006:
Echo classification within the spectral domain
to discriminate ground clutter from meteorologi-
cal targets. 86th AMS Annual Meeting, IIPS,
Atlanta, GA.

Siggia, A. and R. Passarelli, 2004: Gaussian
model adaptive processing (GMAP) for im-
proved ground clutter cancellation and moment
calculation. Pro. of Third Euro. Conf. on
Radar in Meteo. and Hydro (ERAD 2004). 67–
73, Visby, Gotland, Sweden.

Steiner, M. and J.A. Smith, 2002: Use of three-
dimensional reflectivity structure for automated
detection and removal of non-precipitating
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGATES_KERNEL_TDBZ</td>
<td>integer</td>
<td>9</td>
<td>Number of gates in kernel for TDBZ</td>
</tr>
<tr>
<td>NGATES_KERNEL_SPIN</td>
<td>integer</td>
<td>11</td>
<td>Number of gates in kernel for SPIN</td>
</tr>
<tr>
<td>NGATES_KERNEL_(\sigma_{zdr})</td>
<td>integer</td>
<td>7</td>
<td>Number of gates in kernel for (\sigma_{zdr})</td>
</tr>
<tr>
<td>NGATES_KERNEL_(\sigma_{dp})</td>
<td>integer</td>
<td>7</td>
<td>Number of gates in kernel for (\sigma_{dp})</td>
</tr>
<tr>
<td>SPIN_THRESHOLD</td>
<td>floating point</td>
<td>5.0</td>
<td>dBZ difference threshold used in SPIN calculation</td>
</tr>
<tr>
<td>SNR_THRESHOLD</td>
<td>floating point</td>
<td>3.0</td>
<td>Signal-to-noise ratio threshold for censoring total power, in dB</td>
</tr>
<tr>
<td>PROB_THRESHOLD</td>
<td>floating point</td>
<td>0.5</td>
<td>If the clutter probability exceeds this threshold the clutter flag is set</td>
</tr>
</tbody>
</table>

Table 1: Description of the dual polarization CMD parameters.

<table>
<thead>
<tr>
<th>Field</th>
<th>Weights</th>
<th>Membership function break points</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDBZ</td>
<td>N.A.</td>
<td>(20, 0), (40, 1)</td>
</tr>
<tr>
<td>SPIN</td>
<td>N.A.</td>
<td>(10, 0), (25, 1)</td>
</tr>
<tr>
<td>MAX_TDBZ_SPIN</td>
<td>1.0</td>
<td>N.A.</td>
</tr>
<tr>
<td>CPA</td>
<td>1.01</td>
<td>(0.6, 1), (0.9, 1)</td>
</tr>
<tr>
<td>(\sigma_{zdr})</td>
<td>0.5</td>
<td>(1.2, 1), (2.4, 1)</td>
</tr>
<tr>
<td>(\sigma_{dp})</td>
<td>0.5</td>
<td>(10, 1), (15, 1)</td>
</tr>
</tbody>
</table>

Table 2: Description of the dual polarization CMD parameters.
Figure 1: Example of weather and clutter spectra.

Figure 2: Example of spectral filter.
Figure 3: Scatter plot of CPA versus Doppler velocity for simulated data

Figure 4: Scatter plot of CPA versus PR$^{0.5}$ for experimental data
Figure 5: Time series of $I^2 + Q^2$ (dB) for a resolution volume with a clutter target.

Figure 6: Time series of $\tan^{-1}(Q/I)$ (phase of $I$ and $Q$) for a resolution volume with a clutter target corresponding to Fig. 5.
Figure 7: CMD single polarization membership functions.

Figure 8: CMD dual polarization membership functions.
Figure 9: KFTG Unfiltered reflectivity.

Figure 10: KFTG Unfiltered velocity.
Figure 11: PPI clutter map for KFTG.

Figure 12: Reflectivity filtered everywhere for data of Fig. 9.
Figure 13: The CMD flag for the KFTG data, i.e., the clutter filter is applied at those gates in the yellow regions.

Figure 14: CMD filtered dBZ for data of Fig. 9.
Figure 15: CMD performance.
Figure 16: Dual polarization S-Pol case, unfiltered reflectivity.

Figure 17: Dual polarization S-Pol case, unfiltered velocity.
Figure 18: S-Pol Clear air reflectivity.
Figure 19: Dual polarization case, CPA interest field.

Figure 20: Dual polarization case, standard deviation of $Z_{dr}$ interest field.
Figure 21: Dual polarization case, standard deviation of $\phi_{dp}$ interest field.

Figure 22: Dual pol. case, TDBZ and SPIN combined interest field.
Figure 23: CMD dual pol. clutter flag.

Figure 24: Dual pol. reflectivity filtered. Compare to Fig. 16.