Functional Description
for the
SUNSCAN Analysis Method
for NEXRAD
Version 1.1

Prepared for:

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

Prepared by:

Mike Dixon

Earth Observing Laboratory (EOL)
The National Center for Atmospheric Research (NCAR)
Boulder, Colorado

2016-12-01

Table of contents

1 Introduction 3

2 Measurement system 3

3 Scanning the sun 5

4 Parameters and constants 8

5 Computations overview 13

5.1 Grid centered on sun 13

5.2 Indexing the beams on the azimuth grid 13

5.3 Computing moments over gates in range 14

5.4 Flow-chart of main computational steps 15

6 Details of computations and code 16

6.1 Introduction 16

6.2 Computing the current solar position relative to the radar 16

6.3 Pulse object 17

6.4 Beam object 18

6.5 Reading the time series data into a Pulse queue 19

6.6 Finding the indexed beams 19

6.7 Computing moments for a beam 23

6.8 Storing the beams 26

6.9 Interpolating moments onto regular grid centered on sun 27

6.10 Correct the powers for noise 29

6.11 Compute the maximum sun power, and dbBelowPeak 30

6.12 Compute the mean sun location for the scan 30

6.13 Compute the estimated sun centroid from the power data 31

6.14 Perform quadratic fit to power data 35

6.15 Compute the mean ZDR and SS for a given solid angle 37

6.16 Compute the receiver gain, given a solar flux measurement 38

Functional Description for the SUNSCAN Analysis Method for NEXRAD Version 1.1

1 Introduction

NCAR/EOL has developed a method for processing data from a sun sector scan, using a data grid centered on the theoretical sun location.

The analysis produces the following principle results:

- The location of the centroid of the 2-D power patterns, for assessing pointing accuracy.
- Estimates of received power from the sun, in the H and V channels, using two quadratic fits to the 2-D power pattern, one in the horizontal sense (azimuth angle) and the other in the vertical sense (elevation angle).
- 2-D patterns of power, H/V correlation, H/V phase difference, ZDR and SS (for ZDR crosspolar calibration analysis).

This paper provides a description of the scan strategy for scanning the sun, and a functional description of the algorithms used for analysis.

2 Measurement system

The solar scan procedure makes use of the sun as a non-polarized radiation source, i.e., H and V radiated powers are assumed equal and uncorrelated.

Figure 1 shows a simplified conceptual diagram of the NEXRAD measurement system.

For the purposes of solar scan analysis, the receiver is of primary importance.

If only antenna pointing analysis is required, the transmitter may be off.

If power-differentials are important, say for solar ZDR analysis, the transmitter should be on during solar scans, to ensure that RF components such as the circulators remain at normal operating temperature.

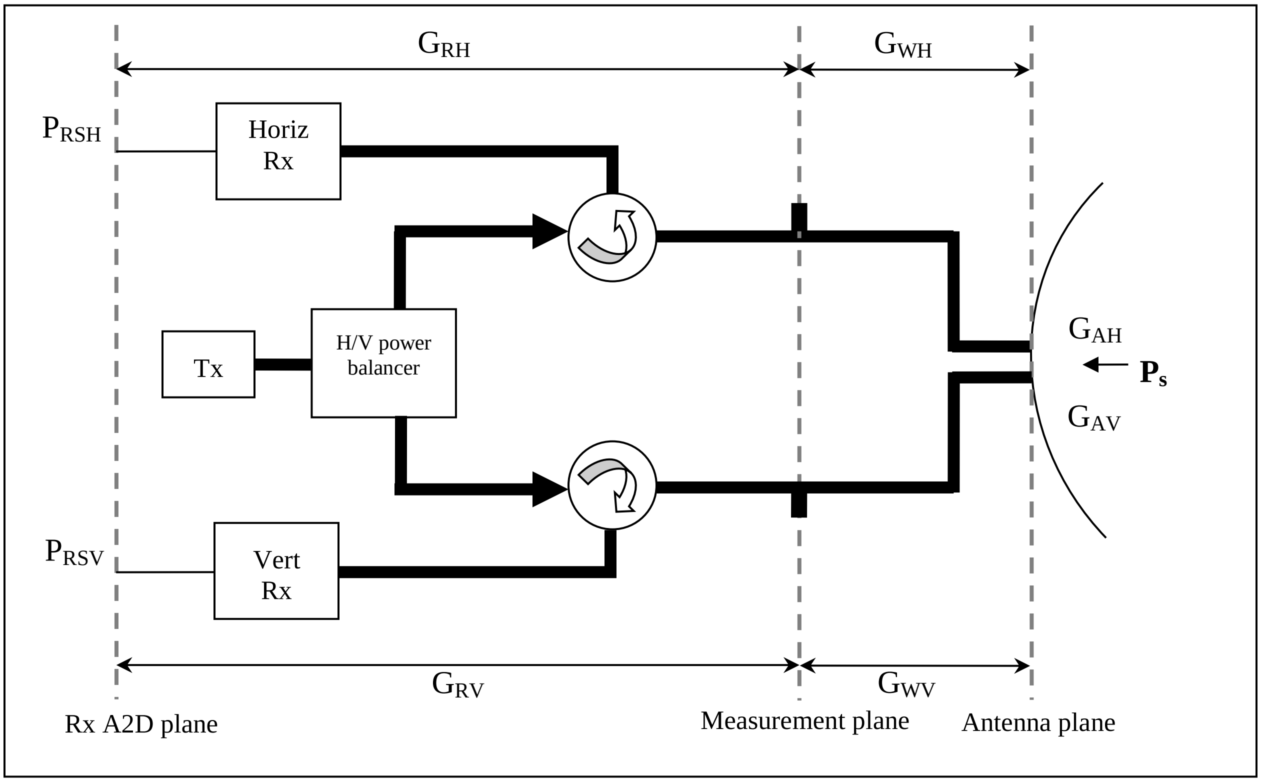


Figure 1: Conceptual measurement diagram.

Table 1 below lists the relevant receiver parameters:

Horizontal channel	**Vertical channel**	**Description**
P_S		Power from sun at antenna plane
P_{RSH}	P_{RSV}	Received sun power as measured by A2D in digital receiver
G_{AH}	G_{AV}	Gain in antenna, one-way
G_{WH}	G_{WV}	Gain in waveguide from measurement plane to antenna, one-way (actually a loss, so < 1)
G_{RH}	G_{RV}	Gain in receiver chain, from measurement plane to A2D

Table 1: receiver parameters.

3 Scanning the sun

A sector-type PPI scan should be set up to follow the sun as it moves. Primary scanning is in azimuth, and secondary scanning in elevation.

The scan should extend 3 degrees on either side of the solar centroid in azimuth, and 2 degrees above and below the solar position in elevation.

Note that because of the geometry of a radar pedestal, the azimuth limits must be corrected for the cosine of the elevation angle. The higher the elevation angle, the wider the azimuth limits.

corrected azimuth limit = (3 degrees) / cosine (elevation).

The scan rate should be relatively slow – i.e. around 1 degree per second.

In the elevation direction, each sweep should be 0.2 degrees apart or less.

Radar moments are computed for beams (dwells) of 128 pulse samples.

Figures 2 (a) through (g) below examples results from KOUN, at 17:00 UTC on 2012/12/25.

The orange grid shows the theoretical sun location. The white cross shows the measured location of the solar centroid. The white circles are at 1 degree and 2 degrees diameter.

The difference between the white cross and orange grid is the estimated antenna angular error. If the sun appears high, the antenna elevation angles are reading low. If the sun appears to the right, the antenna azimuth angles are low.

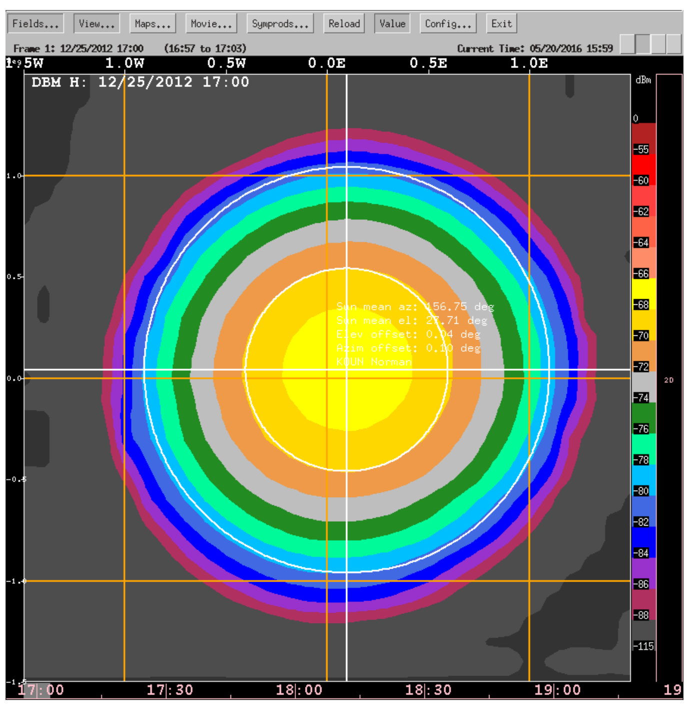


Figure 2a: KOUN, noise-corrected power (dBm) for H channel

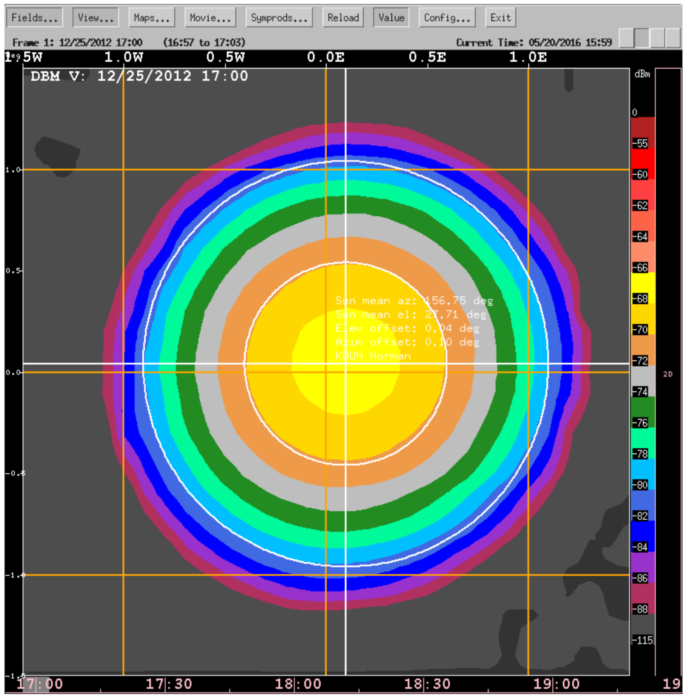


Figure 2b: KOUN, noise-corrected power (dBm) for V channel

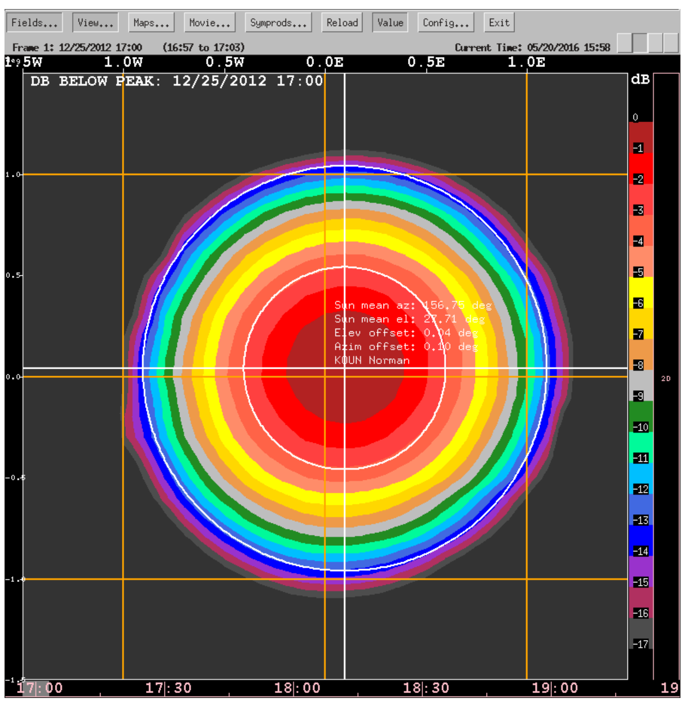


Figure 2c: KOUN, power below peak (dB)

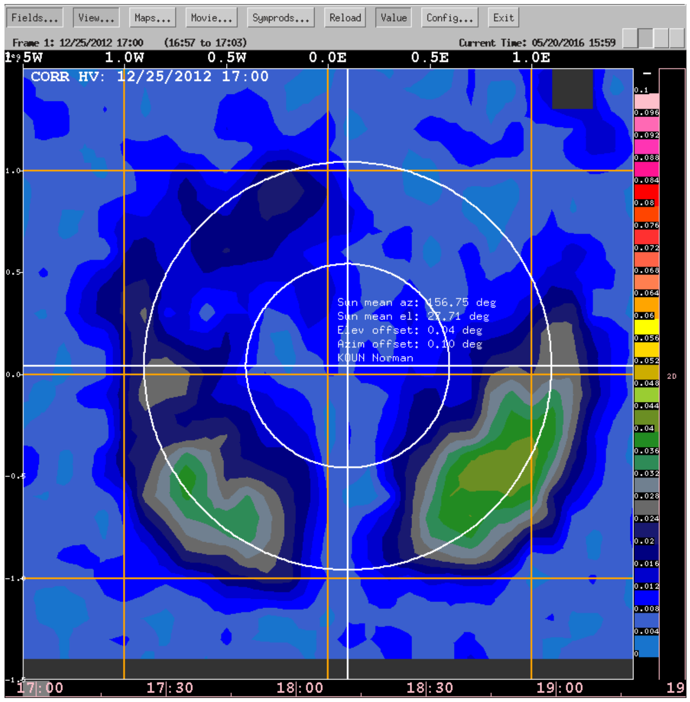


Figure 2d: KOUN, cross-correlation H to V

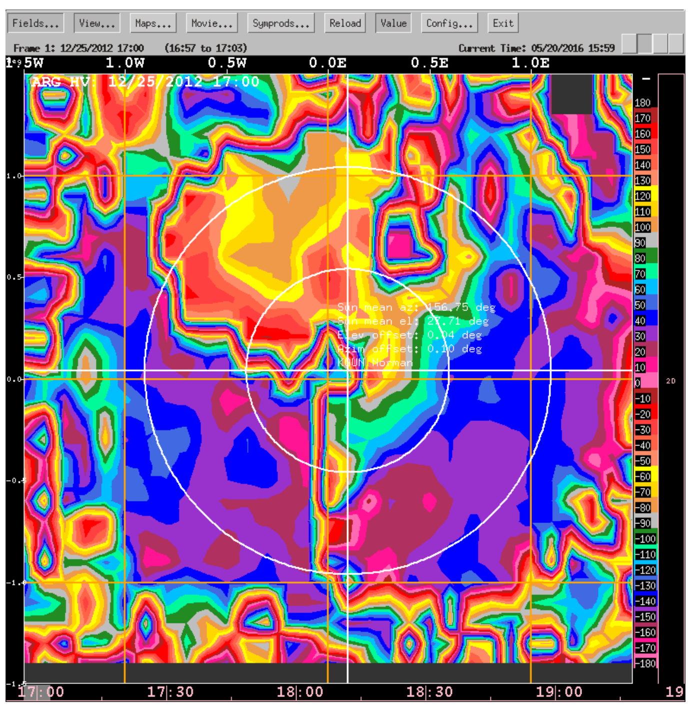


Figure 2e: KOUN, phase difference H to V

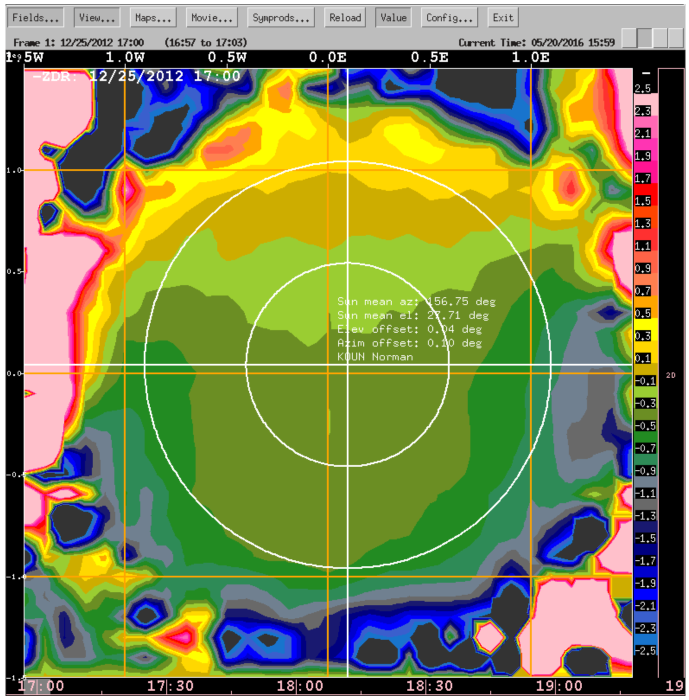


Figure 2f: KOUN, ratio V/H ratio (negative ZDR), (dB)

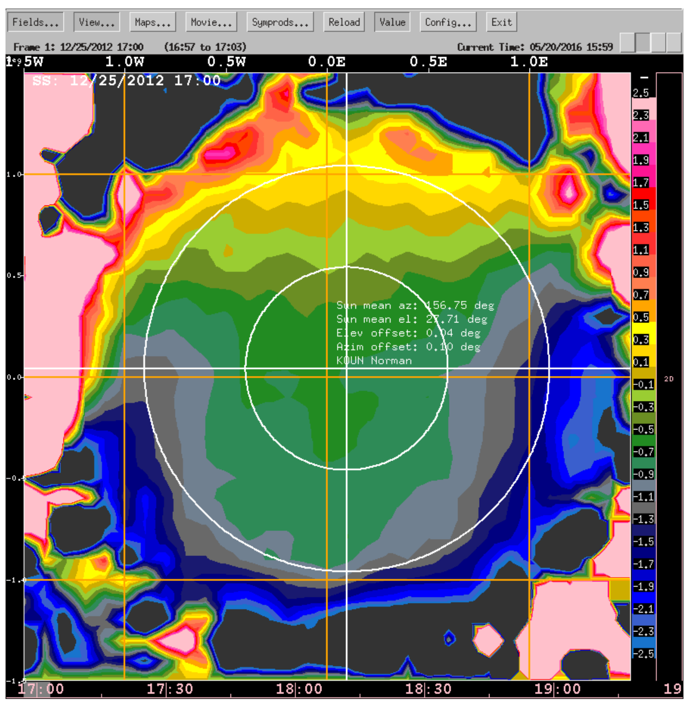


Figure 2g: KOUN, SS

4 Parameters and constants

Table 2 below lists the processing parameters:

Name	Type	Suggested value	Description
nSamples	integer	128	Number of pulse samples in a Beam
gridNAz	integer	31	Number of grid cells in azimuth
gridNEl	integer	21	Number of grid cells in elevation
gridDeltaAz	double	0.2	Grid resolution in azimuth
gridDeltaEl	double	0.2	Grid resolution in elevation
gridStartAz	double	-3.0	Azimuth for starting grid cell
gridStartEl	double	-2.0	Elevation for starting grid cell
nGates	int	-	Number of gates in data
startGate	integer	400	Start gate for computing moments
endGate	integer	800	End gate for computing moments
maxValidDrxPowerDbm	double	-60	Max measured power likely to be observed from the sun. Allows us to censor interference. This is the power at the digital receiver, i.e. P_{RSH} and P_{RSV}, not corrected for receiver gain.
validEdgeBelowPeakDb	double	8	Power at the edge of the valid measurement area, relative to the peak (dB). Only powers above this value are used in the computations.
noiseDbmH	double	-	Noise in H channel (dBm)
noiseDbmV	double	-	Noise in V channel (dBm)
solidAngleForMeanStats	double	1.0	Solid angle for computing mean ZDR and SS

Table 2: analysis parameters

Table 3 below lists the constants used.

Name	Type	Value
`RAD_TO_DEG`	double	57.29577951308092
`DEG_TO_RAD`	double	0.01745329251994372

Table 3: constants

Table 4 lists the global scope variables that are referred to in the code in section 6.

Name	Type	Value
`_latitude`	double	Location of radar in latitude (deg)
`_longitude`	double	Location of radar in longitude (deg)
`_altitudeM`	double	Location of radar in altitude (m)
`_prevSunTime`	double	Previous time.(unix secs) used for computing sun location. Initialize to 0.
`_prevAzOffset`	double	Previous azimuth found in beam indexing. Initialize to -9999.
`_pulseQueue`	Pulse[nSamples]	Queue for storing incoming pulses.
`_rawBeamArray`	Beam [gridNAz][]	Array of raw beams (not interpolated in elevation)
`_interpBeamArray`	Beam [gridNAz][gridNEl]	2D array of interpolated beams (regular grid)
`_interpDbmH`	double [gridNAz][gridNEl]	2D array of H channel power interpolated onto regular grid

Name	Type	Value
`_interpDbmV`	double [gridNAz][gridNEl]	2D array of V channel power interpolated onto regular grid
`_interpDbm`	double [gridNAz][gridNEl]	2D array of mean H/V power interpolated onto regular grid
`_maxPowerDbmH`	double	Maximum H channel power computed from the interpolated grid
`_maxPowerDbmV`	double	Maximum V channel power computed from the interpolated grid
`_maxPowerDbm`	double	Maximum mean power (H+V/2) computed from the interpolated grid
`_quadPowerDbmH`	double	H channel peak power computed from 2-D quadratic fit
`_quadPowerDbmV`	double	V channel peak power computed from 2-D quadratic fit
`_quadPowerDbm`	double	Peak power (mean of H and V) computed from 2-D quadratic fit
`_meanTime`	double	Mean time for Beams with power in excess of validEdgeBelowPeakDb
`_meanSunEl`	double	Sun elevation at _meanTime
`_meanSunAz`	double	Sun azimuth at _meanTime
`_pwrWtCentroidAzErrorH`	double	Estimated azimuth error from H-channel power-weighted centroid
`_pwrWtCentroidElErrorH`	double	Estimated elevation error from H-channel power-weighted centroid

Name	Type	Value
`_pwrWtCentroidAzErrorV`	double	Estimated azimuth error from V-channel power-weighted centroid
`_pwrWtCentroidElErrorV`	double	Estimated elevation error from V-channel power-weighted centroid
`_pwrWtCentroidAzError`	double	Estimated azimuth error from mean power-weighted centroid
`_pwrWtCentroidElError`	double	Estimated elevation error from mean power-weighted centroid
`_quadFitCentroidAzErrorH`	double	Estimated sun azimuth error from H-channel power quadratic fit
`_quadFitCentroidElErrorH`	double	Estimated sun elevation error from H-channel power quadratic fit
`_quadFitCentroidAzErrorV`	double	Estimated sun azimuth error from V-channel power quadratic fit
`_quadFitCentroidElErrorV`	double	Estimated sun elevation error from V-channel power quadratic fit
`_quadFitCentroidAzError`	double	Estimated sun azimuth error from mean power quadratic fit
`_quadFitCentroidElError`	double	Estimated sun elevation error from mean power quadratic fit
`_meanZdr`	double	Mean ZDR for solar pattern for specified solid angle
`_meanSS`	double	Mean SS for solar pattern for specified solid angle

Table 4: Global scope variables

5 Computations overview

5.1 Grid centered on sun

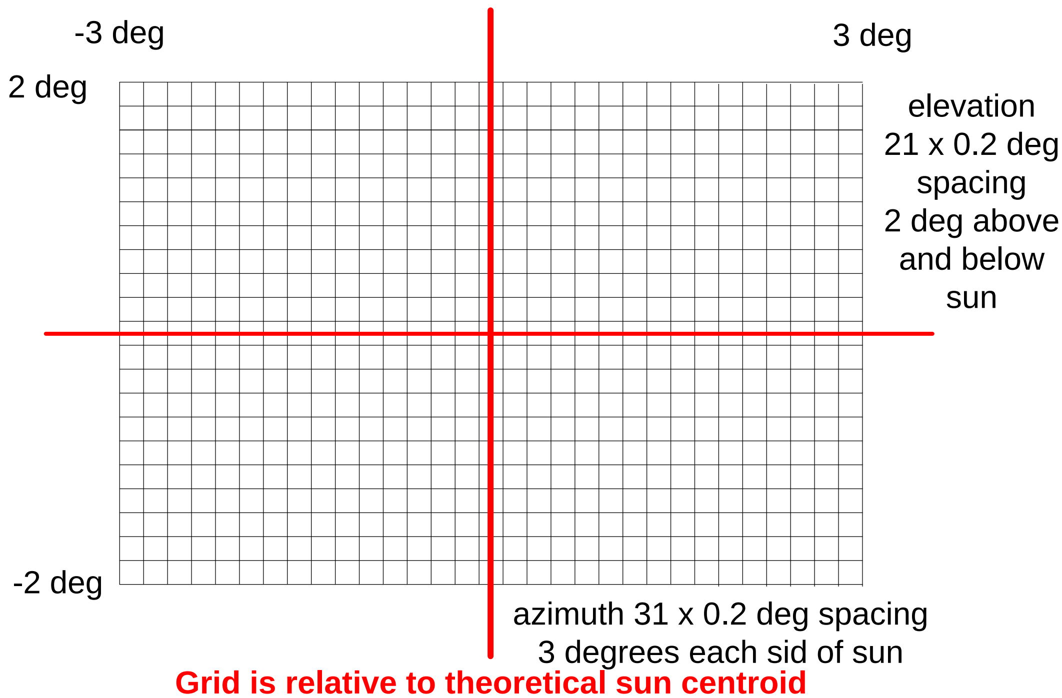
This analysis is based on moments computed for a grid centered on the sun. In populating the grid, we compute the angular offset of each beam relative to the sun at the time of the beam, and place the beam in the grid at the sun-relative location.

Figure 3 shows the grid details.

The resolution in azimuth and elevation is 0.2 degrees.

In azimuth we scan 3 degrees on either side of the sun.

In elevation we scan 2 degrees below and 2 degrees above the sun.



.

Figure 3: sun-centered grid

5.2 Indexing the beams on the azimuth grid

The analysis assumes that we have access to time series data. This allows us to form the beams by selecting pulses from the time series such that they are indexed to the grid in azimuth.

This simplifies mapping the data to the regular grid, and interpolation need only be performed in elevation. The interpolation step is 1-dimensional, in elevation, since all of the beams already line up on grid locations in azimuth.

5.3 Computing moments over gates in range

The signal received from the sun is incoherent relative to the radar, and should be equal in H and V, since the sun is regarded as an un-polarized source. Since this is a CW signal, the powers can be computed by averaging over a large number of gates in range.

If the transmitter is on during the solar scan, care must be used to avoid including power from side-lobe returns. Therefore, only gates beyond the range of side-lobe clutter should be used. A reasonable approach would be to use all gates between the ranges of say 100km to 200 km. See startGate and endGate in table 2.

5.4 Flow-chart of main computational steps

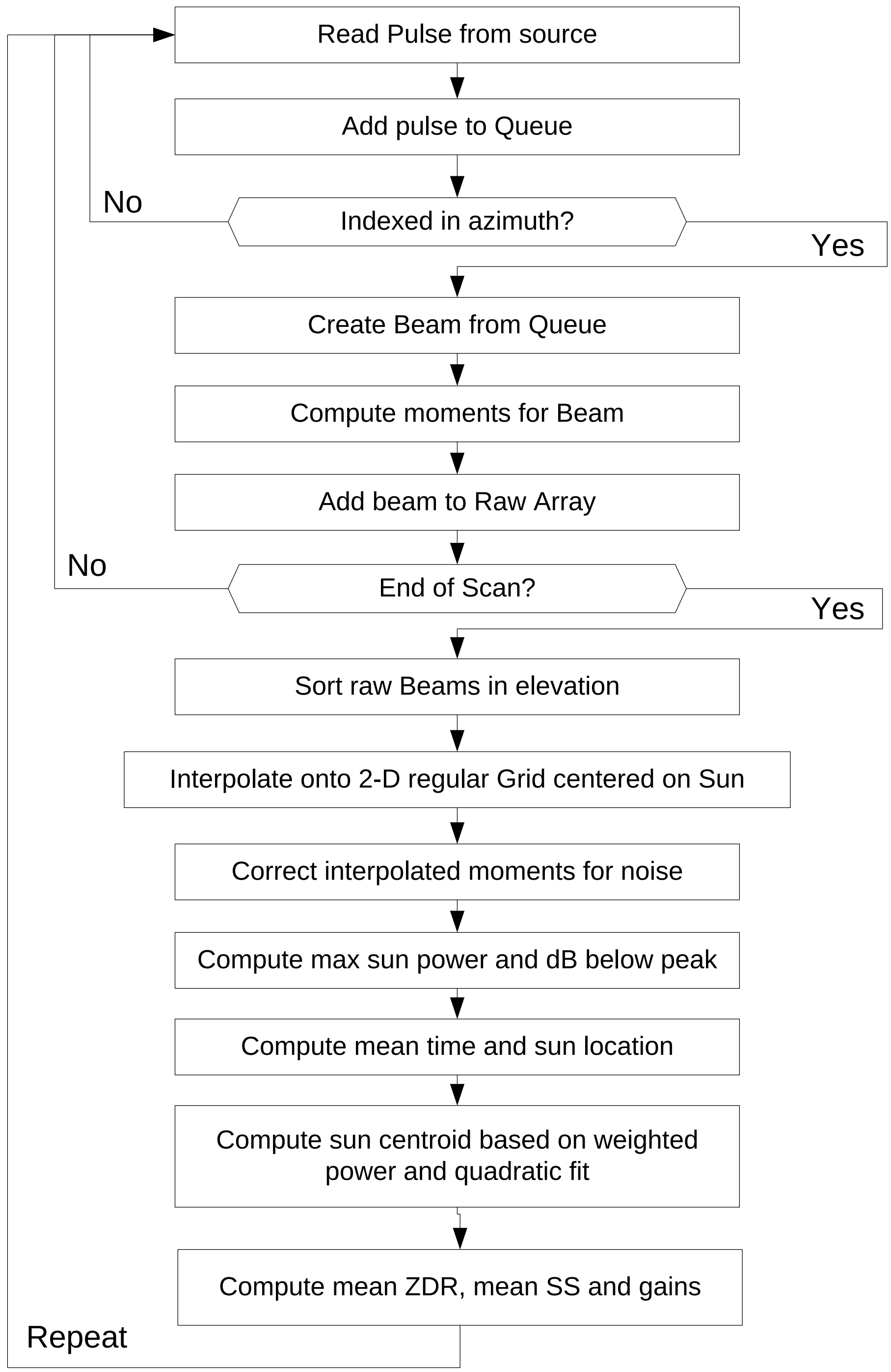


Figure 4: processing flow chart

6 Details of computations and code

6.1 Introduction

The following sections contain details about the implementation, including code fragments where applicable.

We have used C and C++ syntax for the code, since this document is based on the NCAR SunCal C++ application.

6.2 Computing the current solar position relative to the radar

For computing the sun orientation relative to the radar and at a specified time, NCAR makes use of the NOVAS-C software from the U.S.. Naval Observatory. See:

http://aa.usno.navy.mil/software/novas/novas_c/novasc_info.php

In this package, the file rsts_sun_pos.c contains the following function:

```
void rsts_SunNovasComputePos
    (site_info here, double deltat,
     double *SunAz, double *SunEl, double *distanceAU);
```

We use this function to compute the azimuth and elevation angle of the sun, from the radar location. The rsts package must be initialized.

setLocation() in an initialization routine, and sets up the latitude, longitude and altitude for the radar. It must be called at startup.

```
/////////////////////////////////////////////////////////////////
// initialize the lat/lon/alt for which sun position is computed
// lat/lon in degrees, alt_m in meters

void setLocation(double lat, double lon, double alt_m)

{
  _prevSunTime = 0.0;
  _latitude = lat;
  _longitude = lon;
  _altitudeM = alt_m;
}
```

computePosnNova() computes the sun location given the time, and the previously stored location. The sun elevation and azimuth are set.

```
/////////////////////////////////////////////
// compute sun position using NOVA routines

void computePosnNova(double now, double &el, double &az)
```

```
{
  // check if time has changed
  // if not do not recalculate

  double now = time(NULL);
  if (fabs(now - _prevSunTime) < 1) {
    // time has not changed more than 1 sec
    return;
  }
  _prevSunTime = now;

  // set up site info
  double tempC = 20;
  double pressureMb = 1013;

  site_info site = { _latitude, _longitude, _altitudeM, tempC, pressureMb };

  // set time
  time_t tnow = (time_t) now;
  double deltat = -0.45;

  // compute sun posn
  rsts_SunNovasComputePosAtTime(site, deltat, &az, &el, tnow);
}
```

6.3 Pulse object

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

6.3.1 Pulse implementation code

The Pulse object may be visualized as a C structure.

```
/******************************
 * Pulse implementation example
 */

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[nGates * 2];

} Pulse;
```

6.4 Beam object

A Beam is based on 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 computed moments.

6.4.1 Beam implementation code

The Beam object may be visualized 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) */
  double elOffset; /* elevation offset to theoretical sun center (deg) */
  double azOffset; /* azimuth offset to theoretical sun center (deg) */

  /* Array of Pulses */

  Pulse pulses[nSamples]; /* pulses for this beam */

  /* moments */

  double powerH; /* power for H channel I*I+Q*Q */
  double powerV; /* power for V channel I*I+Q*Q */
  double dbmH;   /* power for H channel in dBm */
  double dbmV;   /* power for V channel in dBm */
  double dbm;    /* mean of dbmH and dbmV */
  double corrHV;  /* correlation between H and V */
  double phaseHV; /* mean phase between H and V */
  double dbBelowPeak; /* peak sun power minus mean dbm */
  double zdr; /* dbmH minus dbmV */
  double SS; /* 1.0 / (zdr^2) */

} Beam;
```

6.5 Reading the time series data into a Pulse queue

We read the time series data, constructing Pulse objects and inserting them into a queue, of length nSamples.

Pulse _pulseQueue[nSamples]

As each pulse is added to the front of the queue, a pulse is discarded from the back of the queue.

The pulses in the queue represent the data for 1 beam, consisting of nSamples pulses.

6.6 Finding the indexed beams

After each pulse is added to the queue, we check the queue contents to determine whether the mid-azimuth lines up with the grid – i.e. is the beam represented in the queue indexed to the grid in azimuth?

We use the function isBeamIndexedToGrid(), see below, to check whether the current queue of pulses forms a Beam that is indexed to the grid in azimuth.

If the beam is indexed (return value 0), we accept it, create a beam, compute the moments and save the Beam in the _rawBeamArray.

If the beam is not indexed (return value -1), we read in another pulse, adjust the queue, and continue.

A Beam is computed from nSamples Pulse objects.

6.6.1 Function *isBeamIndexedToGrid()*

This function checks whether the beam in the queue is indexed to the grid. We do this by checking whether the pulses at the center of the queue straddle one of the grid cells in azimuth. If they do, then we form a beam from the pulse queue.

The function returns 0 on success (i.e. is indexed) and -1 on failure (i.e. is not indexed).

```
////////////////////////////////////////
// returns 0 on success, -1 on failure

// globals
// We keep track of the previous valid indexed azimuth found
// Initialize _prevAzOffset.
double _prevAzOffset = -9999;

int isBeamIndexedToGrid()

{

  // find pulses either side of mid point of queue

  int midIndex0 = nSamples / 2;
  int midIndex1 = midIndex0 + 1;
  Pulse pulse0 = _pulseQueue[midIndex0];
  Pulse pulse1 = _pulseQueue[midIndex1];
```

```
// compute angles at mid queue ? i.e. in center of beam

double az0 = pulse0.az;
double az1 = pulse1.az;

double el0 = pulse0.el;
double el1 = pulse1.el;

// adjust az angles if they cross north

adjustForNorthCrossing(az0, az1);

// order the azimuths

if (az0 > az1) {
  double tmp = az0;
  az0 = az1;
  az1 = tmp;
}

// compute mean azimuth and elevation

double az = computeAngleMean(az0, az1);
if (az < 0) {
  az += 360.0;
}
double el = computeAngleMean(pulse0.el, pulse1.el);

    // compute cosine of elevation for correcting azimuth relative to sun

double cosel = cos(el * DEG_TO_RAD);

// compute angles relative to sun position

double midTime = (pulse0.time + pulse1.time) / 2.0;
double sunEl, sunAz;
computePosnNova(midTime, sunEl, sunAz);

// compute az offsets for 2 center pulses

double offsetAz0 = computeAngleDiff(az0, sunAz) * cosel;
double offsetAz1 = computeAngleDiff(az1, sunAz) * cosel;

// compute grid az closest to the offset az

double roundedOffsetAz =
     (floor (offsetAz0 / gridDeltaAz + 0.5)) * gridDeltaAz;

// have we moved at least half grid point since last beam?

if (fabs(offsetAz0 - _prevAzOffset) < gridDeltaAz / 2) {
  return -1;
}
```

```
  // is the azimuth correct?

  if (offsetAz0 > roundedOffsetAz || offsetAz1 < roundedOffsetAz) {
    return -1;
  }

  // is this azimuth contained in the grid?

  int azIndex = -1;
  azIndex = (int) ((roundedOffsetAz - gridStartAz) / gridDeltaAz + 0.5);
  if (azIndex < 0 || azIndex > gridNAz - 1) {
    // outside grid - failure
    return -1;
  }

  // save az offset
  _prevAzOffset = roundedOffsetAz;

  // success

  return 0;

}
```

6.6.2 Angle manipulation functions

The following are functions for performing arithmetic on angles. They are used by the function isBeamIndexedToGrid() above.

```
//////////////////////////
// check for north crossing
// and adjust accordingly

void adjustForNorthCrossing(double &az0, double &az1)
{
  if (az0 - az1 > 180) {
    az0 -= 360.0;
  } else if (az0 - az1 < -180) {
    az1 -= 360.0;
  }
}

/////////////////////////////////////
/// condition az to between 0 and 360

double conditionAz(double az)
{
  while (az < 0.0) {
    az += 360.0;
  }
  while (az > 360.0) {
    az -= 360.0;
  }
  return az;
```

```
}

///////////////////////////////////////////////////
/// condition el to between -180 and 180

double conditionEl(double el)
{
  while (el < -180.0) {
    el += 360.0;
  }
  while (el > 180.0) {
    el -= 360.0;
  }
  return el;
}

///////////////////////////////////////////////////
/// condition angle delta to between -180 and 180

double conditionAngleDelta(double delta)
{
  if (delta < -180.0) {
    delta += 360.0;
  } else if (delta > 180.0) {
    delta -= 360.0;
  }
  return delta;
}

///////////////////////////////////////////////////
/// compute diff between 2 angles: (ang1 - ang2)

double computeAngleDiff(double ang1, double ang2)
{
  double delta = conditionAngleDelta(ang1 - ang2);
  return delta;
}

////////////////////////////////////////////////////////////
///
/// compute mean of 2 angles: ang1 + ((ang2 - ang1)/2)

double computeAngleMean(double ang1, double ang2)
{
  double delta = conditionAngleDelta(ang2 - ang1);
  double mean = ang1 + delta / 2.0;
  if (ang1 > 180 || ang2 > 180) {
    mean = conditionAz(mean);
  } else {
    mean = conditionEl(mean);
  }
  return mean;
}
```

6.7 Computing moments for a beam

In normal radar operations, moments are computed and stored for each gate in the beam.

However, because the sun is a continuous-wave (CW) source, the powers and cross-correlations should be the same at all gates, except for those contaminated by side-lobe clutter or weather if the transmitter is running. Therefore, we can compute the gate moments and then average them over a large number of gates to compute a stable mean value.

6.7.1 Function computeMoments()

```
////////////////////////////////////////////////////
// compute sun moments in dual-pol simultaneous mode
// load up Beam with moments

int computeMoments(int startGate,
                   int endGate,
                   Beam &beam)

{

  // initialize summation quantities

  double sumPowerH = 0.0;
  double sumPowerV = 0.0;
  Complex_t sumRvh0(0.0, 0.0);
  double nn = 0.0;

  // loop through gates to be used for sun computations

  for (int igate = startGate; igate <= endGate; igate++, nn++) {

    // get the I/Q data time series for the gate
    // the getGateIq() functions must be provided externally.

    const Complex_t *iqh = getGateIqH(igate);
    const Complex_t *iqv = getGateIqV(igate);

    // compute lag 0 covariance = power

    double lag0_h = meanPower(iqh, nSamples - 1);
    double lag0_v = meanPower(iqv, nSamples - 1);

    // check power for interference

    double dbmH = 10.0 * log10(lag0_h);
    double dbmV = 10.0 * log10(lag0_v);

    if (dbmH > maxValidDrxPowerDbm) {
      // don't use this gate - probably interference
      continue;
    }

    // compute lag0 conjugate product, for correlation
```

```
    Complex_t lag0_hv =
      meanConjugateProduct(iqh, iqv, nSamples - 1);

    // sum up

    sumPowerH += lag0_h;
    sumPowerV += lag0_v;
    sumRvh0 = complexSum(sumRvh0, lag0_hv);

  } // igate

  // sanity check

  if (nn < 3) {
    cerr << "Warning - computeMoments" << endl;
    cerr << "  Insufficient good data found" << endl;
    cerr << "  az, el: " << beam.az << ", " << beam.el << endl;
    cerr << "  nn: " << nn << endl;
    return -1;
  }

  // compute mean moments

  beam.powerH = sumPowerH / nn;
  beam.powerV = sumPowerV / nn;

  beam.dbmH = 10.0 * log10(beam.powerH);
  beam.dbmV = 10.0 * log10(beam.powerV);
  beam.dbm = (beam.dbmH + beam.dbmV)/2.0;
  beam.zdr = beam.dbmH - beam.dbmV;
  beam.SS = 1.0 / (2.0 * beam.zdr);

  double corrMag = mag(sumRvh0) / nn;
  beam.corrHV = corrMag / sqrt(beam.powerH * beam.powerV);
  beam.phaseHV = argDeg(sumRvh0);

  // compute sun angle offset

  double sunEl, sunAz;
  computePosnNova(beam.time, sunEl, sunAz);
  double cosel = cos(beam.el * DEG_TO_RAD);
  beam.azOffset = computeAngleDiff(beam.az, sunAz) * cosel;
  beam.elOffset = computeAngleDiff(beam.el, sunEl);

  return 0;

}
```

6.7.2 Complex math implementation

The Complex class, below, shows the code for the implementation of the complex computations used above.

```
/////////////////////////////////////////////////////
// Complex math object

class Complex_t {
public:
  // default constructor - magnitude of 1, phase of 0
  Complex_t() : re(1.0), im(0.0) {}
  // constructor with values
  Complex_t(double re_, double im_) : re(re_), im(im_) {}
  // data
  double re;
  double im;
};

/////////////////////////////////////////////
// compute mean power of time series

double meanPower(const Complex_t *c1, int len)
{
  if (len < 1) {
    return 0.0;
  }
  double sum = 0.0;
  for (int ipos = 0; ipos < len; ipos++, c1++) {
    sum += ((c1->re * c1->re) + (c1->im * c1->im));
  }
  return sum / len;
}

/////////////////////////////////////////////
// compute mean conjugate product of series

Complex_t meanConjugateProduct(const Complex_t *c1,
                               const Complex_t *c2,
                               int len)
{

  double sumRe = 0.0;
  double sumIm = 0.0;

  for (int ipos = 0; ipos < len; ipos++, c1++, c2++) {
    sumRe += ((c1->re * c2->re) + (c1->im * c2->im));
    sumIm += ((c1->im * c2->re) - (c1->re * c2->im));
  }

  Complex_t meanProduct;
  meanProduct.re = sumRe / len;
  meanProduct.im = sumIm / len;

  return meanProduct;
```

```
}

///////////////////////////////////////////////
// compute sum

Complex_t complexSum(const Complex_t &c1,
                     const Complex_t &c2)
{
  Complex_t sum;
  sum.re = c1.re + c2.re;
  sum.im = c1.im + c2.im;
  return sum;
}

/////////////////////////////////////////
// mean of complex sum

Complex_t Complex::mean(Complex_t sum, double nn)
{
  Complex_t mean;
  mean.re = sum.re / nn;
  mean.im = sum.im / nn;
  return mean;
}

/////////////////////////////////////////
// mean of complex sum

double mag(Complex_t val)
{
  return sqrt(val.re * val.re + val.im * val.im);
}

/////////////////////////////////////////////////////////
// compute arg in degrees

double argDeg(const Complex_t &cc)

{
  double arg = 0.0;
  if (cc.re != 0.0 || cc.im != 0.0) {
    arg = atan2(cc.im, cc.re);
  }
  arg *= RAD_TO_DEG;
  return arg;
}
```

6.8 Storing the beams

Each time the antenna angle crosses a grid azimuth line, we create a Beam object and compute the moments for that object. We refer to these are the ‘raw’ Beam objects.

We need to store the raw Beam objects, to keep them available for interpolation onto the regular grid.

To do this, we need a 2-D array of Beam objects.

In C++ STL notation, we have the following global variable:

vector<vector<Beam> > _rawBeamArray;

The outer dimension will be gridNAz. The inner dimension will be variable, depending on how many Beams are found for each azimuth in the grid.

In 2-D array notation, this would be:

Beam _rawBeamArray[gridNAz][variable];

6.8.1 Populating the raw beam array

The code for adding a beam to the array would be as follows:

```
void addBeam(Beam beam)
{
  int azIndex = -1;
  azIndex = (int) ((beam.az - gridStartAz) / gridDeltaAz + 0.5);
  if (azIndex >= 0 || azIndex < gridNAz) {
    _rawBeamArray[azIndex].push_back(beam);
  }
}
```

6.9 Interpolating moments onto regular grid centered on sun

After all of the data from a single solar scan has been read in and handled, the _rawBeamArray will be fully populated. This array is indexed in azimuth (outer dimension), but the elevation angles are not yet indexed to the grid (inner dimension).

To index the data to the solar-relative grid in elevation, we need to perform a 1-D interpolation step for each azimuth grid position.

First we sort the raw moments by elevation, in case the scan was not truly bottom-up.

Then we perform the interpolation.

6.9.1 Sort raw moments in ascending elevation order, for each azimuth column

```
//////////////////////////////////////////////////
// sort the raw beam data by elevation

void sortRawBeamsByEl()
{
  for (int iaz = 0; iaz < (int) rawBeamArray.size(); iaz++) {
    vector<Beam> beam = rawBeamArray[iaz];
    sort (beam.begin(), beam.end(), compareBeamEl);
  }
}

////////////////////////////////////////
// Compare for sort on elevation angles

bool compareBeamEl(Beam lhs, Beam rhs)
{
```

```
  if (lhs.offsetEl < rhs.offsetEl) {
    return true;
  } else {
    return false;
  }
}
```

6.9.2 Interpolation code

```
////////////////////////////////////////////////
// interp ppi moments onto regular 2-D grid

// global 2D array of Beam objects to store the interpolated data:
Beam _interpBeamArray[gridNAz][gridNEl];
double _interpDbmH[gridNAz][gridNEl];
double _interpDbmV[gridNAz][gridNEl];
double _interpDbm[gridNAz][gridNEl];

void interpMoments()
{

  // loop through azimuths
  for (int iaz = 0; iaz < gridNAz; iaz++) {
    double azOffset = gridStartAz + iaz * gridDeltaAz;

    // find elevation straddle if available
    for (int iel = 0; iel < gridNEl; iel++) {
      double elOffset = gridStartEl + iel * gridDeltaEl;

      // find the raw moments which straddle this elevation
      vector<Beam> &raw = _rawBeamArray[iaz];
      for (size_t ii = 0; ii < raw.size() - 1; ii++) {

        Beam raw0 = raw[ii];
        Beam raw1 = raw[ii+1];
        double elOffset0 = raw0.elOffset;
        double elOffset1 = raw1.elOffset;

        // is the elevation between these two values
        if (elOffset0 > elOffset || elOffset1 < elOffset) {
          continue;
        }

        // compute interpolation weights
        double wt0 = 1.0;
        double wt1 = 0.0;
        if (elOffset0 != elOffset1) {
          wt1 = (elOffset - elOffset0) / (elOffset1 - elOffset0);
          wt0 = 1.0 - wt1;
        }

        // compute interpolated values

        Beam &interp = _interpBeamArray[iaz][iel];

        interp.time = (wt0 * raw0.time) + (wt1 * raw1.time);
```

```
        interp.az = (wt0 * raw0.az) + (wt1 * raw1.az);
        interp.el = (wt0 * raw0.el) + (wt1 * raw1.el);
        interp.elOffset = elOffset;
        interp.azOffset = azOffset;
        interp.powerH = (wt0 * raw0.powerH) + (wt1 * raw1.powerH);
        interp.powerV = (wt0 * raw0.powerV) + (wt1 * raw1.powerV);
        interp.dbmH = (wt0 * raw0.dbmH) + (wt1 * raw1.dbmV);
        interp.dbmV = (wt0 * raw0.dbmV) + (wt1 * raw1.dbmV);
        interp.dbm = (wt0 * raw0.dbm) + (wt1 * raw1.dbm);
        interp.zdr = (wt0 * raw0.zdr) + (wt1 * raw1.zdr);
        interp.SS = (wt0 * raw0.SS) + (wt1 * raw1.SS);
        interp.corrHV = (wt0 * raw0.corrHV) + (wt1 * raw1.corrHV);
        interp.phaseHV = (wt0 * raw0.phaseHV) + (wt1 * raw1.phaseHV);

        _interpDbmH[iaz][iel] = interp.dbmH;
        _interpDbmV[iaz][iel] = interp.dbmV;
        _interpDbm[iaz][iel] = interp.dbm;

        break;

      } // ii
    } // iel
  } // iaz
}
```

6.10 Correct the powers for noise.

We correct the power in the interpolated beams for noise.

```
/////////////////////////////////////////////////
// correct powers by subtracting the noise

void correctPowersForNoise()

{
  double noisePowerH = pow(10.0, noiseDbmH / 10.0);
  double noisePowerV = pow(10.0, noiseDbmV / 10.0);
  for (int iel = 0; iel < gridNEl; iel++) {
    for (int iaz = 0; iaz < gridNAz; iaz++) {
      Beam beam = _interpBeamArray[iaz][iel];
      beam.powerH -= noisePowerH;
      beam.powerV -= noisePowerV;
      if (beam.powerH <= 0) {
        beam.powerH = 1.0e-12;
      }
      if (beam.powerV <= 0) {
        beam.powerV = 1.0e-12;
      }
      beam.dbmH = 10.0 * log10(beam.powerH);
      beam.dbmV = 10.0 * log10(beam.powerV);
    } // iaz
  } // iel
}
```

6.11 Compute the maximum sun power, and dbBelowPeak

Compute the max sun power in dBm, and load the dbBelowPeak field.

```
/////////////////////////////////////////////////
// compute the maximum power

void computeMaxPower()
{

  // max power for each channel, and mean of channels

  _maxPowerDbmH = -120.0;
  _maxPowerDbmV = -120.0;
  _maxPowerDbm = -120.0;
  for (int iel = 0; iel < gridNEl; iel++) {
    for (int iaz = 0; iaz < gridNAz; iaz++) {
      Beam beam = _interpBeamArray[iaz][iel];
      if (beam.dbmH <= maxValidDrxPowerDbm) {
        _maxPowerDbmH = MAX(_maxPowerDbmH, beam.dbmH);
      }
      if (beam.dbmV <= maxValidDrxPowerDbm) {
        _maxPowerDbmV = MAX(_maxPowerDbmV, beam.dbmV);
      }
      if (beam.dbm <= maxValidDrxPowerDbm) {
        _maxPowerDbm = MAX(_maxPowerDbm, beam.dbm);
      }
    }
  }

  // compute dbm below peak
  for (int iel = 0; iel < gridNEl; iel++) {
    for (int iaz = 0; iaz < gridNAz; iaz++) {
      Beam beam = _interpBeamArray[iaz][iel];
      beam.dbBelowPeak = beam.dbm - _maxPowerDbm;
    }
  }
}
```

6.12 Compute the mean sun location for the scan

We compute the mean time from the interpolated beams.

```
/////////////////////////////////////////////////////////
// compute the sun location for the mean time of the scan

double computeMeanSunLocation()

{
  double sumTime = 0.0;
  double nn = 0.0;
  for (int iel = 0; iel < gridNEl; iel++) {
    for (int iaz = 0; iaz < gridNAz; iaz++) {
      Beam beam = _interpBeamArray[iaz][iel];
```

```
      if (beam.dbBelowPeak > validEdgeBelowPeakDb) {
        sumTime += beam.time;
        nn++;
      }
    } // iaz
  } // iel

  // compute mean time
  _meanTime = sumTime / nn;

  // compute mean sun location

  computePosnNova(_meanTime, _meanSunEl, _meanSunAz);

}
```

6.13 Compute the estimated sun centroid from the power data

The steps are as follows:

- Estimate power-weighted centroid, in elevation and azimuth.
- In azimuth, fit a parabola to the row of power data located at the power-weighted centroid in elevation.
- In elevation, fit a parabola to the column of power data located at the power-weighted centroid in azimuth.
- Use the parabolas to determine the measured centroid and power.
- The measured centroid indicates the antenna pointing errors. The errors are the negative of the centroid elevation and azimuth.

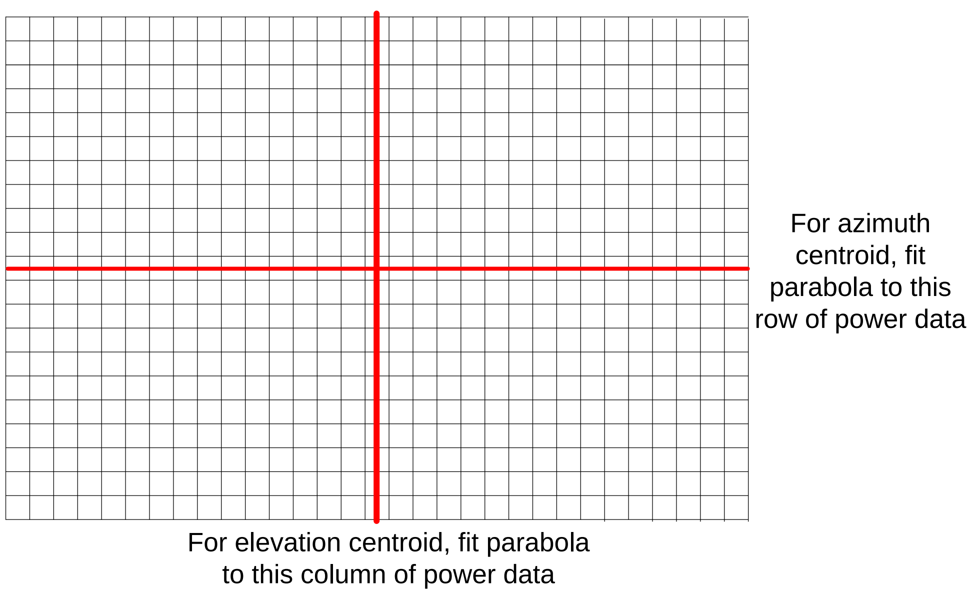


Figure 5: parabolas are fitted to the data in the column and row located at the power-weighted centroid

```
///////////////////////////////////////////////////////////////
//
// Compute sun centroid for mean, H and V channels

int computeSunCentroidAllChannels()

{
  // compute centroid for mean dbm (mean of H and V)

  computeSunCentroid(_interpDbm,
                     _maxPowerDbm,
                     _quadPowerDbm,
                     _pwrWtCentroidAzError,
                     _pwrWtCentroidElError,
                     _quadFitCentroidAzError,
                     _quadFitCentroidElError);

  // compute centroid for H channel

  computeSunCentroid(_interpDbmH,
                     _maxPowerDbmH,
                     _quadPowerDbmH,
                     _pwrWtCentroidAzErrorH,
                     _pwrWtCentroidElErrorH,
                     _quadFitCentroidAzErrorH,
                     _quadFitCentroidElErrorH);

  // compute centroid for V channel

  computeSunCentroid(_interpDbmV,
                     _maxPowerDbmV,
                     _quadPowerDbmV,
                     _pwrWtCentroidAzErrorV,
                     _pwrWtCentroidElErrorV,
                     _quadFitCentroidAzErrorV,
                     _quadFitCentroidElErrorV);

}

////////////////////////////////////////////////////////////////
// Compute sun centroid for given power array

int computeSunCentroid(double **interpDbm,
                       double maxPowerDbm,
                       double &quadPowerDbm,
                       double &pwrWtCentroidAzError,
                       double &pwrWtCentroidElError,
                       double &quadFitCentroidAzError,
                       double &quadFitCentroidElError)

{

  // initialize

  quadPowerDbm = -120.0;
  pwrWtCentroidAzError = 0.0;
  pwrWtCentroidElError = 0.0;
```

```
quadFitCentroidAzError = 0.0;
quadFitCentroidElError = 0.0;

// first estimate the 2-D power-weighted centroid

double sumWtAz = 0.0;
double sumWtEl = 0.0;
double sumPower = 0.0;
double count = 0.0;

double edgePowerThreshold = maxPowerDbm - validEdgeBelowPeakDb;

for (int iel = 0; iel < gridNEl; iel++) {
  for (int iaz = 0; iaz < gridNAz; iaz++) {
    Beam beam = _interpBeamArray[iaz][iel];
    double dbm = interpDbm[iaz][iel];
    double power = pow(10.0, dbm / 10.0);
    if (dbm >= edgePowerThreshold && dbm <= maxValidDrxPowerDbm) {
      double az = beam.azOffset;
      double el = beam.elOffset;
      sumPower += power;
      sumWtAz += az * power;
      sumWtEl += el * power;
      count++;
    } // if (dbm >= edgePowerThreshold ?
  } // iaz
} // iel

if (count == 0) {
  // no valid data
  cerr << "Cannot estimate solar centroid:" << endl;
  cerr << "  no measured power" << endl;
  return -1;
}

pwrWtCentroidAzError = sumWtAz / sumPower;
pwrWtCentroidElError = sumWtEl / sumPower;

double gridMaxAz = gridStartAz + gridNAz * gridDeltaAz;
double gridMaxEl = gridStartEl + gridNEl * gridDeltaEl;

if (pwrWtCentroidAzError < gridStartAz ||
    pwrWtCentroidAzError > gridMaxAz ||
    pwrWtCentroidElError < gridStartEl ||
    pwrWtCentroidElError > gridMaxEl) {
  cerr << "Estimated centroid outside grid:" << endl;
  cerr << "  pwrWtCentroidAzError: " << pwrWtCentroidAzError << endl;
  cerr << "  pwrWtCentroidElError: " << pwrWtCentroidElError << endl;
  cerr << "  Setting quad offsets to 0" << endl;
  return -1;
}

// compute the grid index location of the centroid
// in azimuth and elevation

int elCentroidIndex =
  (int) ((pwrWtCentroidElError - gridStartEl) / gridDeltaEl);
```

```
int azCentroidIndex =
  (int) ((pwrWtCentroidAzError - gridStartAz) / gridDeltaAz);

// fit parabola in azimuth to refine the azimuth centroid
// this is done for the grid row at the elevation centroid

bool fitIsGood = true;
vector<double> azArray;
vector<double> azDbm;

for (int iaz = 0; iaz < gridNAz; iaz++) {
  double dbm = interpDbm[iaz][elCentroidIndex]; // row for el centroid
  if (dbm >= edgePowerThreshold) {
    double az = gridStartAz + iaz * gridDeltaAz;
    azArray.push_back(az);
    // add 200 to dbm to ensure real roots
    azDbm.push_back(dbm + 200);
  }
}

double ccAz, bbAz, aaAz, errEstAz, rSqAz;
if (quadFit((int) azArray.size(),
            azArray, azDbm,
            ccAz, bbAz, aaAz,
            errEstAz, rSqAz) == 0) {
  double rootTerm = bbAz * bbAz - 4.0 * aaAz * ccAz;
  if (rSqAz > 0.9 && rootTerm >= 0) {
    // good fit, real roots, so override centroid
    double root1 = (-bbAz - sqrt(rootTerm)) / (2.0 * aaAz);
    double root2 = (-bbAz + sqrt(rootTerm)) / (2.0 * aaAz);
    quadFitCentroidAzError = (root1 + root2) / 2.0;
  } else {
    fitIsGood = false;
  }
} else {
  fitIsGood = false;
}

// fit parabola in elevation to refine the elevation centroid
// this is done for the grid column at the azimuth centroid

vector<double> elArray;
vector<double> elDbm;

for (int iel = 0; iel < gridNEl; iel++) {
  double dbm = interpDbm[azCentroidIndex][iel]; // column for az centroid
  if (dbm >= edgePowerThreshold) {
    double el = gridStartEl + iel * gridDeltaEl;
    elArray.push_back(el);
    // add 200 to dbm to ensure real roots
    elDbm.push_back(dbm + 200);
  }
}

double ccEl, bbEl, aaEl, errEstEl, rSqEl;
if (quadFit((int) elArray.size(),
```

```
               elArray, elDbm,
               ccEl, bbEl, aaEl,
               errEstEl, rSqEl) == 0) {
    double rootTerm = bbEl * bbEl - 4.0 * aaEl * ccEl;
    if (rSqEl > 0.9 && rootTerm >= 0) {
      // good fit, real roots, so override centroid
      double root1 = (-bbEl - sqrt(rootTerm)) / (2.0 * aaEl);
      double root2 = (-bbEl + sqrt(rootTerm)) / (2.0 * aaEl);
      quadFitCentroidElError = (root1 + root2) / 2.0;
    } else {
      fitIsGood = false;
    }
  } else {
    fitIsGood = false;
  }

  // set power from quadratic fits

  if (fitIsGood) {
    quadPowerDbm = (ccAz + ccEl) / 2.0 - 200.0;
  }

}
```

6.14 Perform quadratic fit to power data

This is a least-squares quadratic fit.

```
/////////////////////////////////////////////////////////////////
// quadFit : fit a quadratic to a data series
//
//  n: number of points in (x, y) data set
//  x: array of x data
//  y: array of y data
//  a? - quadratic coefficients (cc - bias, bb - linear, aa - squared)
//  std_error - standard error of estimate
//  r_squared - correlation coefficient squared
//
// Returns 0 on success, -1 on error.
//
/////////////////////////////////////////////////////////////////

int quadFit(int n,
            const vector<double> &x,
            const vector<double> &y,
            double &cc,
            double &bb,
            double &aa,
            double &std_error_est,
            double &r_squared)

{

  long i;

  double sumx = 0.0, sumx2 = 0.0, sumx3 = 0.0, sumx4 = 0.0;
```

```
double sumy = 0.0, sumxy = 0.0, sumx2y = 0.0;
double dn;
double term1, term2, term3, term4, term5;
double diff;
double ymean, sum_dy_squared = 0.0;
double sum_of_residuals = 0.0;
double xval, yval;

if (n < 4)
  return (-1);

dn = (double) n;

// sum the various terms

for (i = 0; i < n; i++) {

  xval = x[i];
  yval = y[i];

  sumx = sumx + xval;
  sumx2 += xval * xval;
  sumx3 += xval * xval * xval;
  sumx4 += xval * xval * xval * xval;
  sumy += yval;
  sumxy += xval  * yval;
  sumx2y += xval * xval * yval;

}

ymean = sumy / dn;

// compute the coefficients

term1 = sumx2 * sumy / dn - sumx2y;
term2 = sumx * sumx / dn - sumx2;
term3 = sumx2 * sumx / dn - sumx3;
term4 = sumx * sumy / dn - sumxy;
term5 = sumx2 * sumx2 / dn - sumx4;

aa = (term1 * term2 / term3 - term4) / (term5 * term2 / term3 - term3);
bb = (term4 - term3 * aa) / term2;
cc = (sumy - sumx * bb  - sumx2 * aa) / dn;

// compute the sum of the residuals

for (i = 0; i < n; i++) {
  xval = x[i];
  yval = y[i];
  diff = (yval - cc - bb * xval - aa * xval * xval);
  sum_of_residuals += diff * diff;
  sum_dy_squared += (yval - ymean) * (yval - ymean);
}

// compute standard error of estimate and r-squared

std_error_est = sqrt(sum_of_residuals / (dn - 3.0));
```

```
  r_squared = ((sum_dy_squared - sum_of_residuals) /
        sum_dy_squared);

  return 0;

}
```

6.15 Compute the mean ZDR and SS for a given solid angle

We compute the mean ZDR and SS for a circle of a specified diameter (in degrees), centered on the sun location from the quadratic fit.

```
/////////////////////////////////////////////////
// compute mean ZDR and SS ratio

int computeMeanZdrAndSS(double solidAngle)

{

  double sumZdr = 0.0;
  double sumSS = 0.0;
  double nn = 0.0;

  double searchRadius = solidAngle / 2.0;

  // for points within the required solid angle,
  // sum up stats

  for (int iel = 0; iel < gridNEl; iel++) {
    double el = gridStartEl + iel * gridDeltaEl;
    double elOffset = el - _quadFitCentroidElError;
    for (int iaz = 0; iaz < gridNAz; iaz++) {
      double az = gridStartAz + iaz * gridDeltaAz;
      double azOffset = az - _quadFitCentroidAzError;
      double offset = sqrt(elOffset * elOffset + azOffset * azOffset);
      if (offset <= searchRadius) {
        Beam beam = _interpBeamArray[iaz][iel];
        sumZdr += beam.zdr;
        sumSS += beam.SS;
        nn++;
      }
    } // iaz
  } // iel

  // if too few points, cannot compute mean

  if (nn < 1) {
    _meanSS = -9999; // missing
    _meanZdr = -9999; // missing
    return -1;
  }

  _meanZdr = sumZdr / nn;
  _meanSS = sumSS / nn;
```

```
  return 0;

}
```

6.16 Compute the receiver gain, given a solar flux measurement

This assumes that the solar flux measurement (observed flux) is available from the Penticton observatory in Canada, and that the antenna gain is known. (An alternative is to use a known receiver gain and compute the antenna gain).

```
/////////////////////////////////////////////////
// compute receiver gain
// based on solar flux from Penticton
//
// Reference: On Measuring WSR-88D Antenna Gain Using Solar Flux.
//            Dale Sirmans, Bill Urell, ROC Engineering Branch
//            2001/01/03.

int computeReceiverGain()

{

  // beam width correction for solar obs - Penticton

  double solarRadioWidth = 0.57;
  double radarBeamWidth = 0.92; // example
  double kk =
    pow((1.0 + 0.18 * pow((solarRadioWidth / radarBeamWidth), 2.0)), 2.0);

  // frequency of radar and solar observatory

  double radarFreqMhz = 2809.0; // example
  double solarFreqMhz = 2800.0;

  // estimated received power given solar flux

  double beamWidthRad = radarBeamWidth * DEG_TO_RAD;
  double radarWavelengthM = (2.99735e8 / (radarFreqMhz * 1.0e6));

  // antenna gains - from previous cal

  double antennaGainHdB = 44.95; // example
  double antennaGainH = pow(10.0, antennaGainHdB / 10.0);
  double antennaGainVdB = 45.32; // example
  double antennaGainV = pow(10.0, antennaGainVdB / 10.0);

  // waveguide gains - from previous cal

  double waveguideGainHdB = -1.16; // example
  double waveguideGainH = pow(10.0, waveguideGainHdB / 10.0);
  double waveguideGainVdB = -1.44; // example
```

```
  double waveguideGainV = pow(10.0, waveguideGainVdB / 10.0);

  // Observed flux
  // 'fluxobsflux' column from Penticton flux table

  double fluxSolarFreq = 135.0; // example
  double fluxRadarFreq =
    (0.0002 * fluxSolarFreq - 0.01) *
      (radarFreqMhz - solarFreqMhz) + fluxSolarFreq;

  // noise bandwidth from pulse width

  double pulseWidthUs = 1.5; // example
  double noiseBandWidthHz = 1.0e6 / pulseWidthUs;

  // gain H

  double PrHWatts = ((antennaGainH * waveguideGainH *
                      radarWavelengthM * radarWavelengthM * fluxRadarFreq *
                      1.0e-22 * noiseBandWidthHz) /
                     (4 * M_PI * 2.0 * kk));
  double PrHdBm = 10.0 * log10(PrHWatts) + 30.0;
  double rxGainHdB = _quadPowerDbmH - PrHdBm;

  // gain V

  double PrVWatts = ((antennaGainV * waveguideGainV *
                      radarWavelengthM * radarWavelengthM * fluxRadarFreq *
                      1.0e-22 * noiseBandWidthHz) /
                     (4 * M_PI * 2.0 * kk));
  double PrVdBm = 10.0 * log10(PrVWatts) + 30.0;
  double rxGainVdB = _quadPowerDbmV - PrHdBm;

}
```