Edge Node to Remote Node Topology Optimization in the Rapidly Deployable Radio Network (RDRN)

Shane M. Haas
Dr. David W. Petr
John Paden

May 31, 1997

Abstract

This report explains how the Rapidly Deployable Radio Network (RDRN) establishes the connections between an edge node and its associated remote nodes. The RDRN tries to maximize the minimum signal-to-interference ratio of the remote nodes by optimally deciding the number radio beams to form, the direction to steer each beam, the relative power of each beam, and the assignment of beams to remote nodes. This report documents how the RDRN makes these decisions.

Contents

1 Introduction

This report explains how the RDRN [1] establishes the connections between an edge node and its remote nodes [1]. The RDRN makes these connections by deciding on the following:

1. The assignment or mapping of radio beams to remote nodes. This assingment is the edge node's server map.
2. The number of radio beams to form.
3. The direction or steering angle of each beam.
4. The relative power of each beam.

The RDRN makes these decisions to maximize the minimum signal-to-interference ratio of the remote nodes.

This report documents how the RDRN makes these decisions. Section 2 explains the edge node to remote node topology problem in more detail. Section 3 shows how the RDRN determines the server map, number of beams, steering angles, and beam powers. Section 4 gives an example of an edge node to remote node configuration. Finally, Appendix A explains the integration of this software and lists the C code that implements these algorithms.

2 Background

This section describes the general structure of the RDRN and the signal-to-interference ratios (SIRs) of the remote nodes.

2.1 RDRN's Structure

As Figure 1 illustrates, the RDRN consists of two types of nodes: edge nodes and remote nodes. The edge nodes are ATM switches that form the network backbone. Each edge node serves a set of remote nodes by forming multiple steerable radio beams.

When the RDRN powers up, when a new edge node is introduced into the network, or when an edge node moves, the RDRN establishes a high speed

[1] Switch nodes and user nodes are alternative names for edge nodes and remote nodes, respectively.

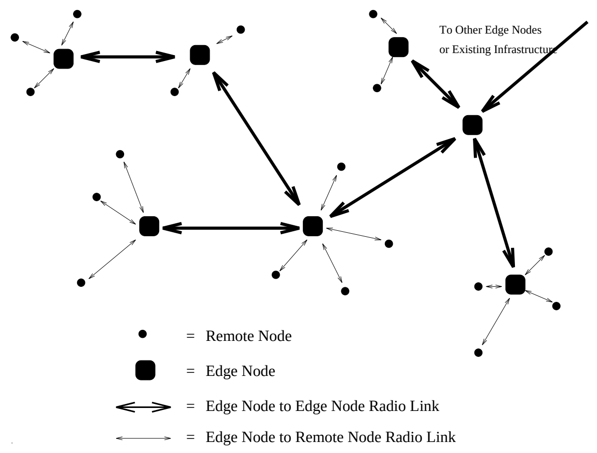
To Other Edge Nodes
or Existing Infrastructure
● = Remote Node
■ = Edge Node
= Edge Node to Edge Node Radio Link
= Edge Node to Remote Node Radio Link

Figure 1: The RDRN consists of edge nodes and remote nodes.

communication path between all edge nodes based on a consistent labeling problem algorithm [2].

After establishing the edge node backbone, the RDRN assigns each remote node to an edge node. Each edge node has an antenna capable of forming multiple electronically steerable radio beams to establish a communication pathway from itself to the remote nodes that it serves. Furthermore, each radio beam can serve several remote nodes simultaneously through TDMA or CDMA multiplexing. Figure 2 shows an edge node forming two radio beams to serve five remote nodes. This figure illustrates the problem scope for this report.

2.2 Signal-to-Interference Ratios

Because each beam operates on the same carrier frequency, they will interfere with each other when transmitting. In other words, a remote node receives information from not only its serving beam, but from every other beam as well.

The signal-to-interference ratio (SIR) quantifies this interference at each remote node. The SIR of a remote node is

$$SIR = \frac{\text{Power from serving beam}}{\sum \text{Power from all other beams}}$$

In words, the SIR of a remote node is the ratio of its serving beam's power

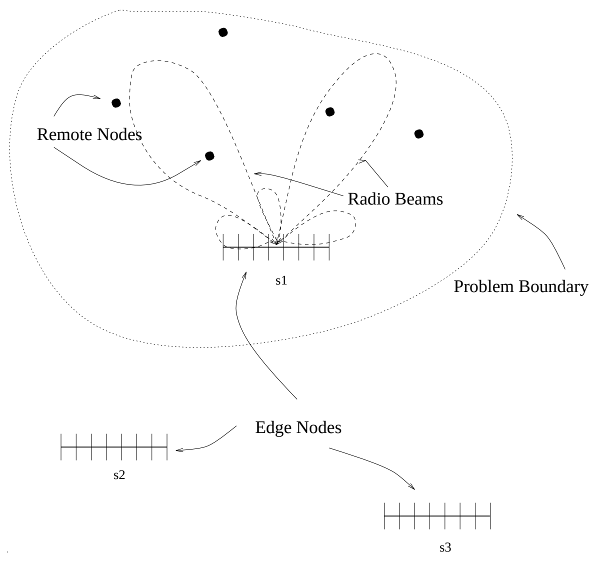

Figure 2: An edge node forms two radio beams to communicate with five remote nodes, illustrating the problem scope of this report.

to the sum of the other beams' powers. A higher SIR corresponds to better reception at a node.

The SIR is dependent upon the power from all the beams formed by an edge node at the angular location of the remote node. The SIR is not dependent upon the physical distance of the remote node from the edge node.

The edge node chooses the server map, number of beams, steering angles, and beam powers to increase the least SIR of its remote nodes.

3 Optimizing the Signal-To-Interference Ratios

An edge node must decide the appropriate server map, number of beams, beam steering angles, and beam powers to maximize the minimum SIR of its remote nodes. This section describes each of these decisions in detail.

3.1 Choosing the Server Map

A server map is a mapping of the edge node radio beams to the remote nodes that each beam communicates with or serves. An edge node will create every possible server map, decide if each server map is redundant, and perform steering angle and power optimization on those that are not redundant to decide which server map allows for the largest minimum SIR of its remote nodes. This subsection describes how an edge node creates all non-repetitive server maps.

3.1.1 Representing a Server Map as a Number

We can describe the server map for N remote nodes as a positive N-digit number whose ith digit is the beam that serves node i. For example, the server map 110 means that an edge node serves three remote nodes and that beam 1 serves remote nodes 0 and 1 and beam 0 serves node 2. The server map 0120 means that beam 0 serves remote nodes 0 and 3, beam 1 serves node 1, and beam 2 serves node 3.

3.1.2 Creating all Possible Server Maps

If N is the number of remote nodes and B is the maximum number of beams, an edge node can create all possible server maps by incrementing an N digit number of base B from zero to $B^N - 1$. For example, the exhaustive list of server maps for an edge node serving two remote nodes and a maximum of two beams is 00, 10, 01, and 11. For four remote nodes and three beams it is 0000, 1000, 2000, 0100, 1100, 2100, 0200, . . ., 2222.

An edge node creates a server map by converting a base ten number x into a base B number y. We can write x as

$$x = y_0 + y_1 B + y_2 B^2 + \cdots + y_{N-1} B^{N-1}$$

where y_0 and y_{N-1} is the least and most significant digit of y, respectively [3]. In our context, y_0 is the beam that serves remote node 0, y_1 is the beam that serves remote node 1, etc. In other words, the base N number y is our server map. The beam that serves remote node 0, y_0, is the remainder of the division x/B. The next digit, y_1, is the remainder of the previous quotient divided again by B. We can find every digit of y by continuing this process.

For example, if we wanted to convert $x = 7$ into a three remote node served by a maximum of three beams server map then we convert 7 into a base 3 number. First, divide 7 by 3 yielding 2 with a remainder of 1. This remainder is the beam that serves remote node 0 ($y_0 = 1$). Next, divide the previous quotient 2 by 3 producing 0 with a remainder of 2 ($y_1 = 2$). The last digit of the server map is the remainder of 0 divided by 3 which is 0 ($y_2 = 0$). In summary, the base ten number 7 represents the three remote node server map 120.

3.1.3 Redundant Server Maps

The number of possible server maps increases exponentially (B^N) as the number of remote nodes and maximum number of beams increases; however, the edge node does not need to perform power and steering angle optimization on all of these server maps because many of them are redundant, or contain the same information. In other words, two server maps can contain the same general information but with the beams numbered differently.

If we create a list of possible server maps using the counting technique presented in the previous section, then a server map is redundant if and only if the beam numbers of the first can be renumbered to create the second. For example, consider the listing of possible server maps 000, 100, 010, 110, 001, 101, 011, and 111. The fifth server map (001) is redundant because by substituting 1 for the 0's and 0 for the 1, we can create the fourth server map (110). Likewise, the sixth server map (101) contains the same information as the third server map (010). Furthermore, the seventh server map (011) and the eighth server map (111) repeat the same information contained in the second server map (100) and the first server map (000), respectively. This examples illustrates that the second half of the server maps are all redundant. We will show later that 110 is the largest possible non-redundant server map for three remote nodes and a maximum of three possible beams.

Given that the previous counting algorithm created a list of possible server maps, we can determine if a server map is redundant by using the following algorithm. Let the server map y be an positive N digit, base B number represented as a string of characters, and $y[i]$ be the ith digit where $y[0]$ is the least significant digit. This algorithm returns TRUE if the server map y is redundant and FALSE if it is not.

```
BOOLEAN IsRedundant(int N, char *y) {

   int MaxDigitValue = 0,
       i             = 0;

   if (y[N-1] != 0)
      return (TRUE);

   for (i = N-1; i >= 0; i--) {
      if (y[i] > MaxDigitValue + 1)
         return (TRUE);
      else if (y[i] > MaxDigitValue)
         MaxDigitValue = y[i];
   };

   return (FALSE)
}
```

A server map is redundant if the most significant digit is not equal to zero. If it passes this first test, then starting with the right most digit (most significant digit), each digit is compared to the previous maximum ($MaxDigitValue$). If the digit is greater than the previous maximum by more than one, then the map is redundant. If all the digits are checked and they do not violate this condition, then the map is non-redundant.

This algorithm implies that the maximum number of non-redundant server maps for a maximum of B beams and N remote nodes is

$$\sum_{i=1}^{N-1} min(i, B-1)B^{N-1-i} \tag{1}$$

Using the IsRedudant algorithm, the largest non-redundant server map for B beams is the number composed of the digits "$B-1, B-1, \cdots, B-1, B-2, B-3, \cdots, 2, 1, 0$". For example, the largest non-redundant server map for $B = 4$ and $N = 5$ is 33210, and for $B = 2$ and $N = 3$ it is 110 as previously shown.

3.2 Choosing the Number of Beams

Although we specify a maximum number of beams when creating server maps, the actual number of beams is represented directly in the server map. For a non-redundant server map the number of beams is the server map's largest digit plus one. For example, the server map 2010 has three beams, the server map 100 has two beams, and 000 has one beam.

To exhaust all possible server maps, the edge node sets the maximum number of beams to the number of nodes. The server maps created (0 to $N^N - 1$) represent the cases where one beam serves all the nodes to each beam serves exactly one node.

3.3 Choosing the Beams' Steering Angles

For a given server map, the edge node places its beams, ideally, to maximize the minimum signal-to-interference ratio of its remote nodes. This task is difficult, however, because the SIR is a function of several non-linear functions (the far-field antenna patterns).

An approximation to this ideal solution is to consider each beam separately, and maximize its power to the nodes it serves while minimizing its power to the nodes it does not serve. In other words, for each beam maximize the useful power while minimizing the interfering power. This solution becomes the optimal solution when the server map contains only one beam.

To determine the "best" position for a beam, the edge node calculates a rank or score for all beam positions between -90^o and 90^o in one degree increments by subtracting the power to the node receiving the maximum interference from the power to the node receiving the worst service by the beam. The edge node then places the beam in the angular location that has the highest score.

A 181 x 181 table of complex numbers containing antenna patterns sampled at one degree increments over a steering angle range from -90^o to 90^o allows the edge node to quickly calculate the ranks of all possible beam positions for each beam. This suboptimal solution is fast and computationally efficient requiring 181 times the number of nodes minus one arithmetic operations and 181 comparisons per beam.

3.4 Choosing the Beams' Relative Powers

Once the edge node determines the steering angles for a given server map, it adjusts the relative or normalized power per beam as follows. Power optimization occurs when the minimum SIRs from each beam are equal because any further increase in a beam's power will result in a decrease in the SIRs of the nodes that it does not serve, and any further decrease in a beam's power will result in a decrease of the SIRs that it does serve; therefore, the power is optimized when the minimum SIRs from each beam are equal.

Two possible solutions exist for solving this power optimization problem: an algebraic closed-form solution and a numerical solution. First, the system of quadratic equations resulting from this equilibrium can be solved algebraically to find the optimal solution. This approach becomes cumbersome for more than two beams. Furthermore, a general solution requires a numerical technique for computing the solution to a system of quadratic equations. As a result, the edge node uses the second approach, a numerical solution, to find the optimal beam powers.

The numerical algorithm works on the following observations. An increase in a beam's power will result in an increase in the SIRs of the nodes served by that beam and a decrease in the SIRs of all other nodes. Likewise, a decrease in a beam's power will result in a decrease in the SIRs of the nodes served by that beam and an increase in the SIRs of all other nodes. Using these observations, the beams' powers are adjusted incrementally until the minimum SIRs of nodes served in each beam are equal.

For multiple beam server maps, the node with the minimum SIR in first beam is the reference for establishing this equilibrium. The next beam's power is then adjusted by an initial increment until the minimum SIR of that beam oscillates around the reference SIR. Each subsequent beam is adjusted by this same increment before the increment is reduced by a configurable factor before repeating the process. This loop stops after a specified number of iterations or when the SIRs are within a specified tolerance.

These configurable parameters are stored in a global variable in a structure called *OptimizationPrefs*. The initial power increment is called *mPwrInitialRes*. The factor by which the increment is reduced after each loop is called *mPwrFactor*. The loop stops when the number of iterations exceeds the value stored in *PwrStop* or the minimum SIRs differ by an amount less than *mPwrFinalRes*.

3.5 Optimization Summary

In summary, an edge node creates all possible server maps from zero to the maximum described by Equation 1 and then determines if each server map is redundant. If the server map is redundant then, the edge node proceeds to the next server map; however, if the server map is not redundant, then the edge node optimizes the steering angles and relative powers of the beams. After optimization, the minimum SIR is compared to previous minimums. The edge node chooses the configuration that has largest minimum SIR.

4 Example

Consider the node layout in Figure 3 with remote nodes located at (5,-61^o), (4,-41^o), and (5.5,62^o). Out of the $3^3 = 27$ possible server maps (corre-

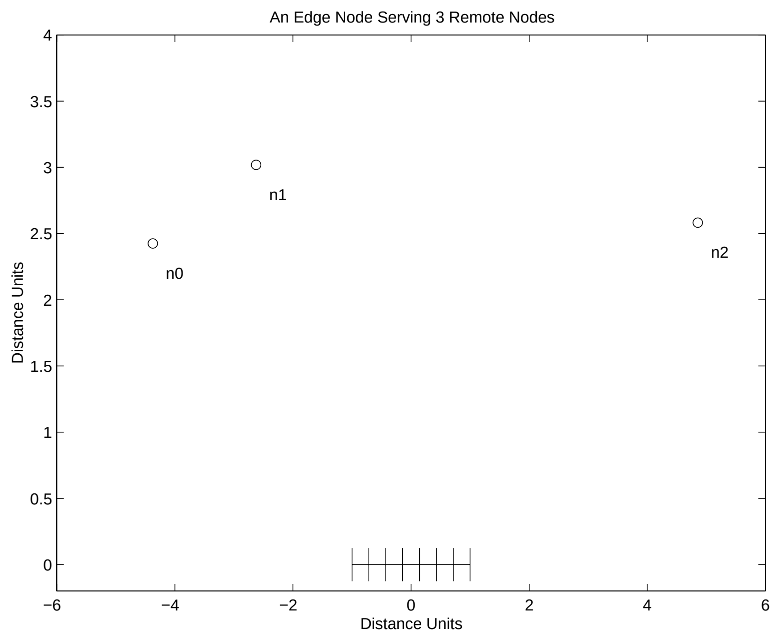

Figure 3: An example of an edge node serving three remote nodes

sponding to the cases where one beam serves all three nodes to each node having its own beam), only 5 are unique. These non-redundant server maps are 000, 100, 010, 110, and 210. After steering angle and power optimization, the resulting minimum SIRs for these server maps are 2.96, 4.32, 4.75, 3.33, and 3.53, respectively; therefore, the "best" configuration is the server map 010 with beams at -85^o and -40^o having relative powers 1 and 0.93, respectively. These results are summarized in Table 1.

Server Map	Beam 0 [degrees] [rel pwr]	Beam 1 [degrees] [rel pwr]	Beam 2 [degrees] [rel pwr]	Min SIR []
000	-51.00 1.00			2.96
100	-40.00 1.00	-65.00 0.15		4.30
010	-85.00 1.00	-40.00 0.93		4.75
110	62.00 1.00	-52.00 0.93		3.33
210	62.00 1.00	-40.00 0.45	-65.00 0.86	3.53

Table 1: The 5 non-redundant server maps with their optimal steering angles and relative powers, and their worst SIR

These results are not intuitive. The "natural" way to group these three nodes is the server map 110 because nodes 0 and 1 are close together while node 2 is more isolated; however, this server map is only the fourth best choice. Furthermore, the best server map (010) divides nodes 0 and 1, but still produces the best minimum SIR. These suprising results are due to the non-linear antenna patterns. The next two subsections will explore the server maps 110 and 010 in more detail.

4.1 The Server Map 110

Figures 4 and 5 show the two beams of server map 110 after angle and power optimization. The vertical dashed lines in each figure show the angular location of the three remote nodes. Notice that nodes 0 and 1 sit on beam 1's main lobe and fall in beam 0's nulls. Likewise, node 2 is on beam 0's main lobe and on one of beam 1's sidelobes.

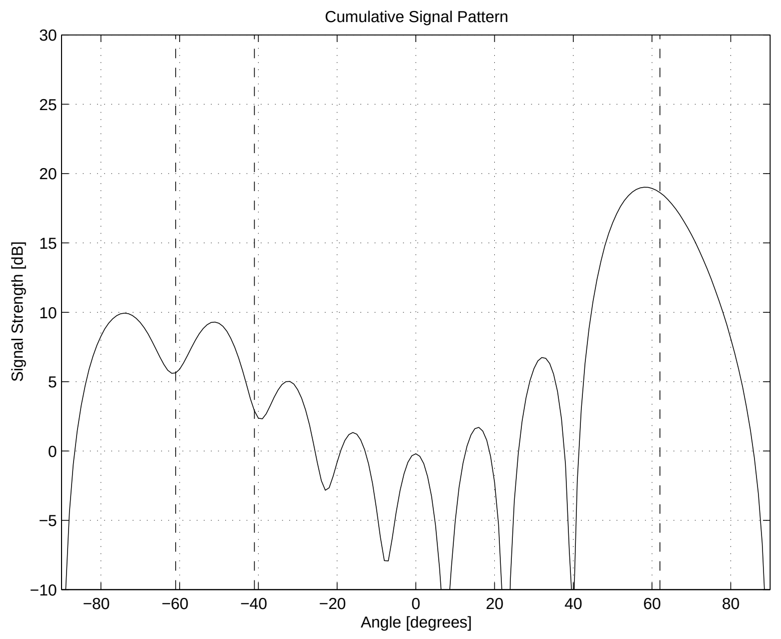

Figure 4: Beam 0 serves node 2 with a steering angle of 62^o and relative power 1.0

The edge node chose this particular set of steering angles by ranking each beam according to the difference between the beam's lowest serving power and highest interfering power. These ranks for each beam are shown in Figure 6. In this case, beam 0's highest score occurred at 62^o, and beam 1's occured at -52^o.

4.2 The Server Map 010

Figures 7 and 8 show the two beams of server map 010 after angle and power optimization. Notice that nodes 0 and 2 sit on beam 0's two main lobes and fall in beam 1's nulls. Likewise, node 2 is on beam 1's main lobe and on one of beam 0's sidelobes.

The ranking scores for each beam are shown in Figure 9. In this case, beam 0's highest score occured at -85^o, and beam 1's occured at -40^o.

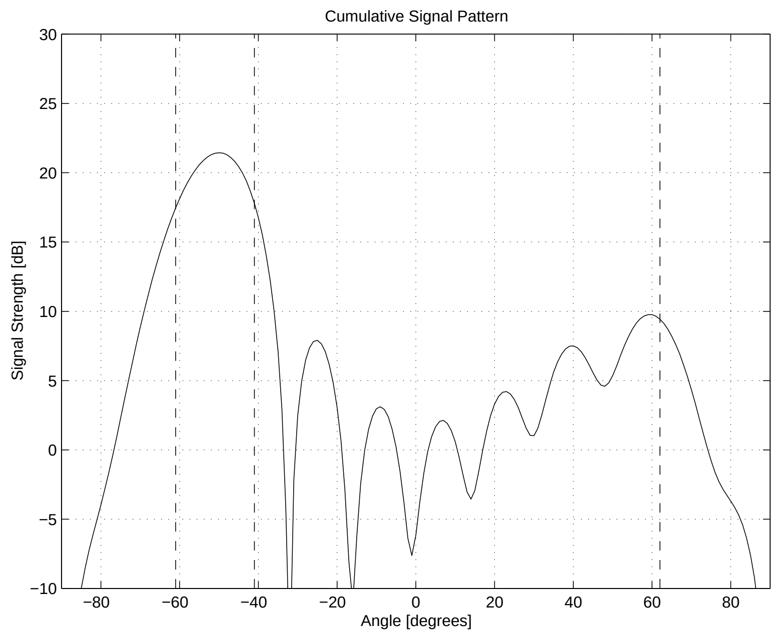

Figure 5: Beam 1 serves nodes 0 and 1 with a steering angle of -52^o and relative power 0.93

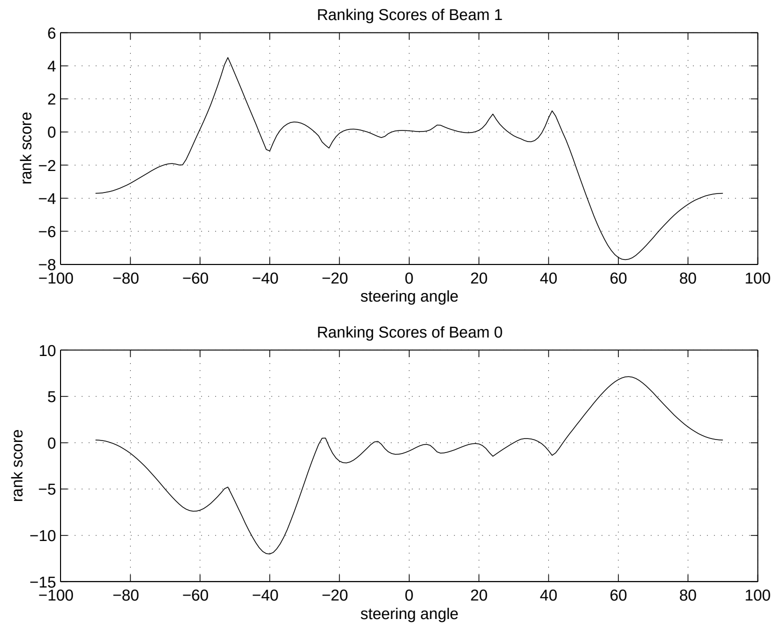

Figure 6: The edge node chooses the beam position with the highest ranking score.

5 Conclusions

The edge node tries to maximize the minimum signal-to-interference ratio among its remote nodes. To accomplish this optimization, the edge node creates all possible non-redundant server maps then chooses the one that has the best minimum SIR after beam angle and power optimization.

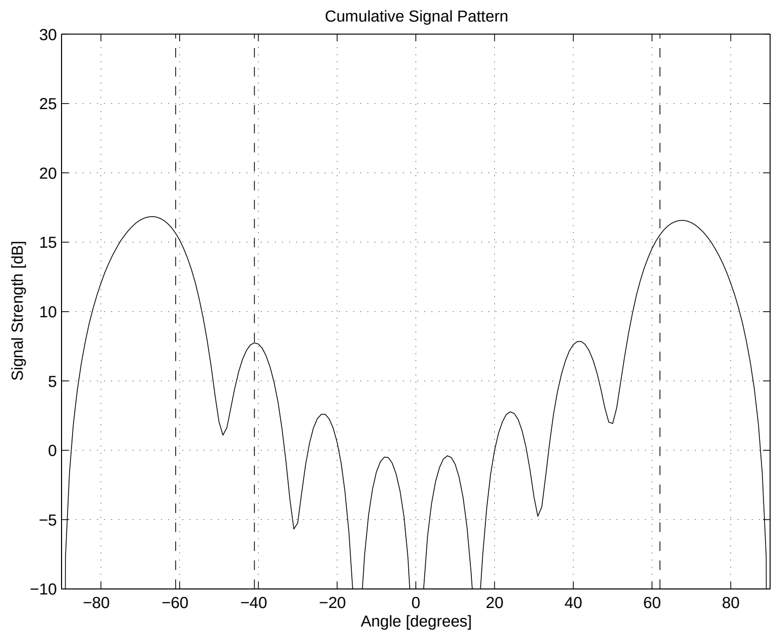

Figure 7: Beam 0 serves nodes 1 and 2 with a steering angle of -85^o and relative power 1.0

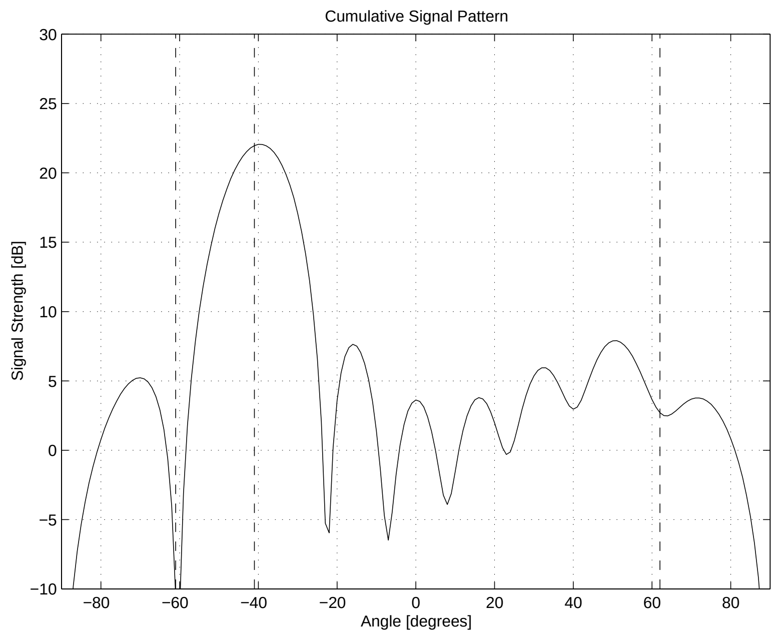

Figure 8: Beam 1 serves node 1 with a steering angle of -40^o and relative power 0.93

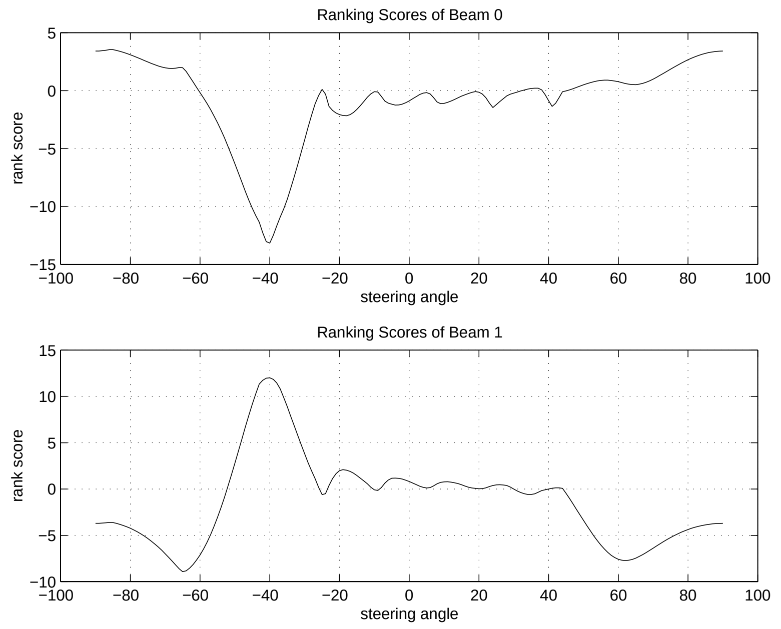

Figure 9: The edge node chooses the beam position with the highest ranking score.

References

[1] K. Sam Shanmugan Joseph Evans Benjamin Ewy, Craig Sparks and Glenn Prescott. An overview of the rapidly deployable radio network proof of concept system. TISL Technical Report TISL-10920-16, April 1996.

[2] Shane M. Haas and Dr. David Petr. A consistent labeling algorithm for the frequency/code assignments in a rapidly deployable radio network (rdrn). TISL Technical Report TISL-10920-04, January 1995.

[3] John F. Wakerly. *Digital Design: Principles and Practices.* Printice Hall, 2nd edition, 1994.

A Code

A.1 BeamTable.h

```
#ifndef __BEAMTABLE_H__
#define __BEAMTABLE_H__

#include "gps.h"

#define NO_ERROR 0
#define MEM_ERROR 1
#define OPEN_WATM_ERROR 2
#define DLTABLE_ERROR 3
#define CALIB_ERROR 4
#define DIG_ERROR 5

#define ELEMENT 8
#define BEAM 3
#define MAX_BEAM 4
#define SAMPLE_RATE 4
#define MODULATION 4
#define MAX_ELEMENT 16
#define MAX_CASE 32
#define MAX_WORD 255
#define ELEMENT_SIZE 49152
#define MASK_SIZE 1532
#define WORD_SIZE 12

#define LOWER_BOUND -1.5707963
#define UPPER_BOUND 1.5707963
#define CLOSE_ANGLE 0.52359877
#define HALF_ANGLE 0.261799383
#define MARGIN_ANGLE 0.017453292


struct GPSLocation
{
```

```
struct gps * mOrigin;
double mX;
double mY;
double mAngle;
double mSIR;
int mAngleIndex;
char mMask;
};

struct BeamInformation
{
double mAngle;
double mPower;
int mAngleIndex;
struct gps **mCover;
};

struct SampleType
{
double Double;
int Quantized;
};

struct TableElement
{
struct SampleType * mSymbol; /* Array of NUM_SAMPLE */
};

struct TableWord
{
struct TableElement * mElement; /* Array of size ELEMENT */
};

struct TableCase
{
struct TableWord * mWord; /* Array of size 4(qpsk) or 2(bpsk) ^ NUM_BEAM */
};
```

```
struct Table
{
int mNumNode;
int mNumBeam;
struct TableCase * mCase; /* Array of each possible UserMask setting(2 ^ NUM_NODE)
};

struct AngleListType
{
double mAngle;
int mStart;
int mCover;
};

struct AnglePathType
{
int mCover;
struct AngleListType *mAngleList;
int mAngleListLength;
};

int CreateBeamTable(
int aNumNode, /* Number of nodes - User fills */
struct gps * aGPSLocation, /* An array of gps locations - User fills */
struct gps * aBaseNode, /* Pointer to the base node -User fills */
int * aNumBeam, /* Filled by SetNode() from AdaptSA.h */
struct BeamInformation * aBeam, /* Filled by SetNode(), AdaptSA(), and AdaptPwr()
int * aNumBad, /* Filled by SetNode() from AdaptSA.h */
struct gps ** aBadGPS, /* Pointer to an Array of pointers to uncovered nodes. Fil
struct Table ** aTable); /* Returns the pointer to the Table */

int TestDownloadBeamTable(struct Table * aTable, int aMaskValue);
/* This function always returns NO_ERROR */

int DownloadBeamTable(struct Table * aTable, int aMaskValue);
/* Must have the driver in place... otherwise use TestDownloadBeamtable() */

/* PRIVATE FUNCTIONS */
```

```
void DeleteTable(struct Table * aTable);
/* Deletes and resets elements in gTable */

void RemoveBaseNode(
int * aNumNode, /* If the base node occurs with in the aGPS list then aNumNode is
struct gps * aGPS, /* An array of gps locations */
struct gps * aBaseNode, /* A pointer to the basenode */
struct GPSLocation ** aSortGPS); /* Allocates as an array and copies aGPS into aSo
/* This function copies aGPS into aSortGPS, but not copying the aBaseNode if it oc
within the aGPS array. */

int SortGPS(
int aNumNode, /* Number of nodes */
struct GPSLocation * aSortGPS); /* An array of GPSLocation locations */
/* This function uses a bubble sort to arrange the aGPSLocation array from smalles

int CreateBeamFormTable(
int aNumNode, /* Number of nodes */
struct GPSLocation * aSortGPS, /* An array of GPSLocation locations */
int aNumBeam, /* Number of beams */
struct BeamInformation * aBeam); /* An array of BeamInformation */
/* This function calls the AdaptSA, AdaptPwr, and Digradio code for each possible
In the process it creates the BeamFormTable(struct Table gTable) */

int Power(int aBase, int aPower);

int SetMask(
int aNumNode, /* Number of nodes */
struct GPSLocation *aSortGPS, /* An array of GPSLocation locations */
register int aMaskValue); /* A decimal representation of the bit mask for the aSor
/* Sets the bit mask in the aSortGPS for each possible case */

unsigned char * ConvertTable(struct Table * aTable, int aMaskValue);

char * Int2BinStr(int aQuantized);

void PrintArguments(
int aNumNode, /* Number of nodes - User fills */
struct gps * aGPSLocation, /* An array of gps locations - User fills */
struct gps * aBaseNode, /* Pointer to the base node -User fills */
int * aNumBeam, /* Filled by SetNode() from AdaptSA.h */
struct BeamInformation * aBeam, /* Filled by SetNode(), AdaptSA(), and AdaptPwr()
int * aNumBad, /* Filled by SetNode() from AdaptSA.h */
struct gps ** aBadGPS, /* Pointer to an Array of pointers to uncovered nodes. Fil
struct Table ** aTable); /* Returns the pointer to the Table */

#endif
#ifndef __COMPLEX_H__
#define __COMPLEX_H__
```

A.2 complex.h

```
/*
 * proj : RDRN
 * file : complex.h
 * prog : Shane M. Haas
 * desc : provides structs and functions for storing and manipulating
 *        complex numbers
 */

/*
 * global includes
 */

/* #include <stdio.h> */

/*
 * typedefs
 */

typedef struct
{
   double Real;
   double Imag;
} CplxRect;
```

```
typedef struct
{
   double Mag;
   double Arg;
} CplxPol;


/*
 * global functions
 */

void PrintCplxPol(CplxPol Z);
   /* Input   : Z = complex number in polar form
    * Output  : -
    * Returns : -
    * Desc    : Prints a number in polar notation to stdout
    */

void PrintCplxRect(CplxRect Z);
   /* Input   : Z = complex number in rectangular form
    * Output  : -
    * Returns : -
    * Desc    : Prints a number in rectangular form to stdout
    */

CplxPol Rect2Pol(CplxRect Z1);
   /* input   : Z1 = complex number in rectangular form
    * returns : complex number in polar form
    * output  : -
    * desc    : converts a complex number from rect to polar form
    */

CplxRect Pol2Rect(CplxPol Z1);
   /* input   : Z1 = complex number in polar form
    * returns : complex number in rectangular form
    * output  : -
    * desc    : converts a complex number from polar to rectangular form
    */

CplxRect CplxAdd(CplxRect Z1, CplxRect Z2);
   /* input   : Z1,Z2 = two complex numbers in rectangular form
    * returns : sum of Z1 and Z2 in rectangular form
    * output  : -
    * desc    : addition of two complex numbers
    */

CplxPol CplxMult(CplxPol Z1, CplxPol Z2);
   /* input   : Z1,Z2 = two complex numbers in polar form
    * returns : product of Z1 and Z2 in polar form
    * output  : -
    * desc    : multiplication of two complex numbers
    */
#endif
```

A.3 gps.h

```
#ifndef __GPS__
#define __GPS__

#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
/* #include <arpa/inet.h> */
#include <unistd.h>
#include <fcntl.h>
#include <netdb.h>

/* $Id: gps.h,v 2.0.5.5 1996/04/03 19:11:11 sbush Exp sbush $ */

/* HEIGHT STRUCTURE */

struct height {
int             ht; /* height */
```

```
char            unit; /* units of height */
};

/* POSITION DATA STRUCTURE */

struct pos {
int             deg; /* degrees */
#ifdef MAISIE
float           minutes;/* minutes */
#else
float           min; /* minutes */
#endif
char            dir; /* direction */
float           x; /* x - coordinate */
float           y; /* y - cooorditate */
};

/* VELOCITY STRUCTURE */

struct vel {
float           spd; /* speed [dist/sec] */
float           dir; /* bearing as measured cw from north [deg] */
};


/* GPS RAW DATA STRUCTURE */

struct gps {
/*
 * Maisie pulls structure declaration each time its used within a
 * structure, thus cannot have the same stuct declared multiple
 * times, e.g. must put all similar structs on same line !!
 */
char            callsign[64]; /* callsign of source */
char            ipaddr[64]; /* IP address of host */
struct tm       time; /* time */
struct pos      lat, lon, xy; /* latitude */
/* struct pos      lon; *//* longitude */
```

```
int             av; /* gps availability */
int             nsat; /* number of satellites used */
int             hdop; /* horizontal dilution of position */
struct height   el, ght;/* elevation */
/* struct height   ght; *//* geoidal height */
/* struct pos      xy; *//* (x,y) coordinates */
struct vel      v; /* velocity */
int             sw_index; /* index into switch table */
};
#endif
#ifndef __OPT_STEER_ANG_H__
#define __OPT_STEER_ANG_H__
```

A.4 opt-steer-ang.h

```
/*
 * proj : RDRN
 * file : opt_steer_ang.h
 * prog : Shane M. Haas and John Paden
 * desc : steering angle optimization algorithm
 */

/* Local Includes */
#include "complex.h"
#include "BeamTable.h"

/* Global Variables */
#define NOK 1
#define OK  0

#define BEAM_ANGLES 181
#define NODE_ANGLES 181

/* Structures */

struct OptimizationPrefs {
   double mPwrInitialRes;           /* Initial power resolution */
   double mPwrFinalRes;            /* Final power resulution */
   double mPwrFactor;              /* Factor to reduct power in each iter */
   int mPwrStop;                   /* Max number of Pwr opt iterations */
};

/* Prototypes */

int LoadBeamTable(
   char *File,
   CplxPol ***BeamTable);
   /*
    * Input   : File = filename of the table of beam patterns created in
    *              Matlab by the make_beam_table.m function
    * Output  : BeamTable = 181 x 181 table of beam patterns.  Each
    *              column corresponds a the steering angle and each row
    *        corresponds to the angle to evaluate the pattern at.  For
    *              example BeamTable[Angle-90][BeamAngle-90].Mag is the
    *              magnitude of the power pattern at Angle degrees when the
    *              beam is pointed at an angle of BeamAngle degrees off of
    *              the antenna array normal.
    * Returns : 1 if file was found and 0 if file was not
    * Desc    : Loads the table of antenna patterns into memory
    */

double  OptSteerAngle(
unsigned char * aServerMap,
struct GPSLocation * aGPS,
unsigned char aNumNode,
struct BeamInformation * aBeam,
unsigned char aNumBeam);
   /*
    * Input:  aServerMap;  aServerMap[i] = Beam number that serves node i
    *         aGPS = location of nodes
    *         aNumNode = number of nodes
    *         aNumBeam = number of beams
    * Output: aBeam = optimimal beam positions
    * Return: minimum SIR of the nodes
    * Desc:   For each beam, find the angle that maximizes the power to
    *         all the nodes it serves while minimizing the power to all
    *         nodes that it doesn't serve.  This is an approximate solution
    *         to finding the maximizing the minimum SIR
    */

double RankBeamAngle(
unsigned char * aServerMap,
unsigned char aNumNode,
struct GPSLocation * aGPS,
int aCurBeam,
int aBeamAngle);
   /* Input:  aServerMap = aServerMap[i] is the beam that serves node i
    *         aNumNode = number of nodes
    *         aGPS = location of nodes
    *         aCurBeam = current beam that is being optimized
    *         aBeamAngle = current beam angle being considered
    * Output: -
    * Return: difference between the minimum serving power and the maximum
    *         interfering power.
    */

double CalcSIR(
   unsigned char *aServerMap,
struct GPSLocation * aGPS,
unsigned char aNode,
struct BeamInformation * aBeam,
unsigned char aNumBeam);
   /*
    * Input:  aServerMap = aServerMap[i] is the beam serving node i
    *         aGPS = node locations
    *         aNode = node whose SIR is being calculated
    *         aBeam = beam angles, relative powers, etc
    *         aNumBeam = number of beams
    * Output: -
    * Return: SIR of aNode
    * Desc:   Calculates the SIR of aNode by dividing the power from its
    *         serving beam by the sum of the powers from the other beams
    */
```

```
void CreateServerMap(
   int aCount,
   int aBase,
   unsigned char * aServerMap);
   /*
    * Input:  aCount = base 10 integer to be converted
    *         aBase = base to be converted to
    * Output: aServerMap = base aBase number
    * Return: -
    * Desc:   creates a servermap by converting a base 10 number
    *         into a base aBase (number of beams) number
    * How it works: to convert a base 10 number (x) into a base b number,
    *         divide x by b.  The remainder is the least sig digit of the base
    *         b number.  Divide the quotient by b again to get the next
    *         least sig digit, etc
    */

int IsRedundant(
   int aNumNode,
   unsigned char * aServerMap,
   int *aNumActiveBeams);
   /*
    * Input : aNumNode = number of nodes
    *         aServerMap = server map to be checked for redundancy
    * Output: aNumActiveBeams = number of beams used in the server map
    * Return: 1 if redundant, 0 if not redundant
    * Desc  : determines if aServerMap is equivalent to previous
    *         server maps created by CreateServerMap() passed values
    *         of aCount ascending from 0.
    */

void PrintServerMap(
   int aNumNode,
   unsigned char * aServerMap);
   /* Input:  aNumNode = number of nodes
    *         aServerMap = server map
    * Output: -
    * Return: -
    * Desc  : prints a server map to the screen
    */

int OptENtoRN(
   unsigned char ** aServerMap,
   struct GPSLocation * aGPS,
   unsigned char aNumNode,
   struct BeamInformation ** aBeam,
   unsigned char aNumBeamMax,
   unsigned char * aNumBeam);
   /*
    * Input   : aGPS = position of nodes
    *           aNumNode = number of nodes
    *           aNumBeamMax = maximum number of beams
    * Output  : aServerMap = beam to node assignment
    *           aBeam = steering angles and relative powers of the beams
    *           aNumBeam = number of beams used
    * Returns: -1 = could not allocate mem; 0 = everything is fine
    * Desc    : finds the optimal EN to RN configuration based on RN SIRs by
    *           determining
    *             1) number of beams to use
    *             2) beam to node assignment
    *             3) steering angle of beams
    *             4) relative power of beams (not yet implement)
    */

double OptBeamPwr(
   unsigned char * aServerMap,
   struct GPSLocation * aGPS,
   unsigned char aNumNode,
   struct BeamInformation * aBeam,
   unsigned char aNumBeam);
   /*
    * Input : aServerMap;  aServerMap[i] = Beam number that serves node i
    *          aGPS = location of nodes
    *          aNumNode = number of nodes
    *          aNumBeam = number of beams
    * Output: aBeam = optimimal beam powers
    * Return: minimum SIR of the nodes
    * Desc  : For each beam, adjust its relative power until the min SIR of
    *         the nodes it serves is equal the the minimum SIR of the first
    *         beam.
    */

double FindMinSIR(
   unsigned char *aServerMap,
   struct GPSLocation * aGPS,
   int aNumNode,
   struct BeamInformation * aBeam,
   unsigned char aNumBeam);
   /*
    * Input:  aServerMap = aServerMap[i] is the beam serving node i
    *         aGPS = node locations
    *         aBeamIndex = beam number to examine
    *         aBeam = beam angles, relative powers, etc
    *         aNumBeam = number of beams
    * Output: -
    * Return: minimum SIR of all the nodes
    * Descs : finds the minimum SIR among all the nodes
    */


double FindMinSIRofBeam(
   unsigned char *aServerMap,
   struct GPSLocation * aGPS,
   int NumNode,
   unsigned char aBeamIndex,
   struct BeamInformation * aBeam,
   unsigned char aNumBeam);
   /*
    * Input:  aServerMap = aServerMap[i] is the beam serving node i
    *         aGPS = node locations
    *         aBeamIndex = beam number to examine
    *         aBeam = beam angles, relative powers, etc
    *         aNumBeam = number of beams
    * Output: -
    * Return: minimum SIR of aBeamIndex's nodes
    * Desc:   finds the minimum SIR among the nodes that a beam serves
    */

#endif
```

A.5 complex.c

```
/*
 * proj : RDRN
 * file : complex.c
 * prog : Shane M. Haas
 * desc : provides structs and functions for storing and manipulating
 *        complex numbers
 */

/*
 * global includes
 */

#include <math.h>
#include <stdio.h>

/*
 * local includes
 */

#include "complex.h"

/*
 * global functions
 */

void PrintCplxPol(CplxPol Z)
{
   /* Input   : Z = complex number in polar form
    * Output  : -
    * Returns : -
    * Prints a number in polar notation to the screen
    */

   printf("Z = %f*exp(j*%f)\n",Z.Mag,Z.Arg);
}

void PrintCplxRect(CplxRect Z)
{
   /* Input   : Z = complex number in rectangular form
    * Output  : -
    * Returns : -
    * Desc    : Prints a number in rectangular form to stdout
    */

   printf("Z = %f + j*%f\n",Z.Real,Z.Imag);
}


CplxPol Rect2Pol(CplxRect Z1)
{
   /* input   : Z1 = complex number in rectangular form
    * returns : complex number in polar form
    * output  : -
    * desc    : converts a complex number from rect to polar form
    */

   CplxPol Tmp;

/*    printf("Input to Rect2Pol: ");PrintCplxRect(Z1); */
   Tmp.Mag = sqrt(Z1.Imag*Z1.Imag+Z1.Real*Z1.Real);
   Tmp.Arg = atan2(Z1.Imag,Z1.Real);

/*    printf("Output of Rect2Pol: ");PrintCplxPol(Tmp);  */
   return Tmp;
}

CplxRect Pol2Rect(CplxPol Z1)
```

```
{
   /* input   : Z1 = complex number in polar form
    * returns : complex number in rectangular form
    * output  : -
    * desc    : converts a complex number from polar to rectangular form
    */

   CplxRect Tmp;

   Tmp.Real = Z1.Mag*cos(Z1.Arg);
   Tmp.Imag = Z1.Mag*sin(Z1.Arg);

/*    printf("Output of Pol2Rect: ");PrintCplxRect(Tmp); */
   return Tmp;
}

CplxRect CplxAdd(CplxRect Z1, CplxRect Z2)
{
   /* input   : Z1,Z2 = two complex numbers in rectangular form
    * returns : sum of Z1 and Z2 in rectangular form
    * output  : -
    * desc    : addition of two complex numbers
    */

    CplxRect Sum;

    Sum.Real = Z1.Real + Z2.Real;
    Sum.Imag = Z1.Imag + Z2.Imag;


/*     printf("Sum: ");PrintCplxRect(Sum); */
    return Sum;
}

CplxPol CplxMult(CplxPol Z1, CplxPol Z2)
{
   /* input   : Z1,Z2 = two complex numbers in polar form
    * returns : product of Z1 and Z2 in polar form
    * output  : -
    * desc    : multiplication of two complex numbers
    */

    CplxPol Prod;

    Prod.Mag = Z1.Mag*Z2.Mag;
    Prod.Arg = Z1.Arg + Z2.Arg;

    return Prod;
}
```

A.6 opt-steer-ang.c

```
/*
 * proj : RDRN
 * file : opt_steer_ang.c
 * prog : Shane M. Haas and John Paden
 * desc : steering angle optimization algorithm
 */

/*
 * global includes
 */

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

/*
 * local includes
 */

#include "opt_steer_ang.h"
#include "BeamTable.h"
```

```
/*
 * global variables
 */

 extern CplxPol ** gBeamTable;
 extern struct OptimizationPrefs gOptPrefs;

/*
 * macros
 */

#define MIN(x,y) (x > y ? y : x)

/*
 * functions
 */

int LoadBeamTable(char *File, CplxPol ***BeamTable)
{
   /* Input   : File = filename of the table of beam patterns created in
    *              Matlab by the make_beam_table.m function
    * Output  : BeamTable = 181 x 181 table of beam patterns.
    *              Each column corresponds a the steering angle and each row
    *              corresponds to the angle to evaluate the pattern at.  For
    *              example BeamTable[Angle-90][BeamAngle-90].Mag is the
    *              magnitude of the power pattern at Angle degrees when the
    *              beam is pointed at an angle of BeamAngle degrees off of
    *              the antenna array normal.
    * Returns : OK if file was found and NOK if file was not
    * Desc    : Loads the table of antenna patterns into memory
    */
   FILE *fd;
   double mag, arg;
   int BeamAngle, NodeAngle;


   /* open the file */
   if (!(fd = fopen(File,"r"))) {
      fprintf(stderr,"ERROR: could not open file '%s'\n",File);
      return NOK;
   }

   /* load table of complex numbers */
   printf("Loading Beam Table:\n");
   (*BeamTable) = malloc(NODE_ANGLES*sizeof(CplxPol*));
   for (NodeAngle = 0;NodeAngle < NODE_ANGLES;NodeAngle++) {
      (*BeamTable)[NodeAngle] = malloc(BEAM_ANGLES*sizeof(CplxPol));
      for (BeamAngle = 0;BeamAngle < BEAM_ANGLES;BeamAngle++) {
         fscanf(fd,"%lf %lf",&mag,&arg);
         (*BeamTable)[NodeAngle][BeamAngle].Mag = mag;
         (*BeamTable)[NodeAngle][BeamAngle].Arg = arg;
      };
      printf("Progress: %1.0f\n",100.0*(NodeAngle+1)/NODE_ANGLES);
   };

   /* close the file */
   fclose(fd);
   return OK;
}

double  OptSteerAngle(
unsigned char * aServerMap,
struct GPSLocation * aGPS,
unsigned char aNumNode,
struct BeamInformation * aBeam,
unsigned char aNumBeam)

   /*
 * Input:  aServerMap;  aServerMap[i] = Beam number that serves node i
 *         aGPS = location of nodes
 *         aNumNode = number of nodes
 *         aNumBeam = number of beams
 * Output: aBeam = optimimal beam positions
 * Return: minimum SIR of the nodes
 * Desc:   For each beam, find the angle that maximizes the power to
 *         all the nodes it serves while minimizing the power to all
 *         nodes that it doesn't serve.  This is an approximate solution
 *         to finding the maximizing the minimum SIR
 */
{
double lMaxRank,
         lMinSIR = 0.0;
int lBeamAngle,
lMaxBeamAngle,
lNode,
lCurBeam;

for (lCurBeam = 0; lCurBeam < aNumBeam; lCurBeam++) {
for (lBeamAngle = 0; lBeamAngle < BEAM_ANGLES; lBeamAngle++) {
double lRank = RankBeamAngle(aServerMap, aNumNode, aGPS, lCurBeam, lBeamAngle);
if (!lBeamAngle ||
lRank > lMaxRank) {
lMaxRank = lRank;
lMaxBeamAngle = lBeamAngle;
}
}
aBeam[lCurBeam].mAngleIndex = lMaxBeamAngle;
aBeam[lCurBeam].mAngle = lMaxBeamAngle - 90;
}

   /* find the minimum SIR for this set of nodes */
for (lNode = 0;lNode < aNumNode;lNode++) {
  double lNodeSIR = CalcSIR(aServerMap,aGPS,lNode,aBeam,aNumBeam);
  if (lNodeSIR < lMinSIR || !lNode)
     lMinSIR = lNodeSIR;
   }

   return (lMinSIR);

}

double RankBeamAngle(
unsigned char * aServerMap,
unsigned char aNumNode,
struct GPSLocation * aGPS,
int aCurBeam,
int aBeamAngle)
{
   /* Input:  aServerMap = aServerMap[i] is the beam that serves node i
 *         aNumNode = number of nodes
 *         aGPS = location of nodes
 *         aCurBeam = current beam that is being optimized
 *         aBeamAngle = current beam angle being considered
 * Output: -
 * Return: difference between the minimum serving power and the maximum
    *         interfering power.
 */

double lNodePower,
lMinServer = 0,
lMaxInterference = 0;
int lCurNode,
lFlag1 = 0,
lFlag2 = 0;

for (lCurNode = 0; lCurNode < aNumNode; lCurNode++) {
lNodePower = gBeamTable[aGPS[lCurNode].mAngleIndex][aBeamAngle].Mag;
if (aServerMap[lCurNode] == aCurBeam) {
if ((lNodePower < lMinServer) || !lFlag1++) {
lMinServer = lNodePower;
}
   }
else
if ((lNodePower < lMaxInterference) || !lFlag2++) {
lMaxInterference = lNodePower;
}
}
return(lMinServer - lMaxInterference);
}

double CalcSIR(
   unsigned char *aServerMap,
struct GPSLocation * aGPS,
unsigned char aNode,
struct BeamInformation * aBeam,
unsigned char aNumBeam)
{
   /* Input:  aServerMap = aServerMap[i] is the beam serving node i
 *         aGPS = node locations
 *         aNode = node whose SIR is being calculated
 *         aBeam = beam angles, relative powers, etc
 *         aNumBeam = number of beams
 * Output: -
 * Return: SIR of aNode
 * Desc:   Calculates the SIR of aNode by dividing the power from its
 *         serving beam by the sum of the powers from the other beams
 */

   int lBeam = 0;
double lSIRnum = 0.0,
       lSIRdenom = 0.0;

   for (lBeam = 0;lBeam < aNumBeam;lBeam++) {
      if ( lBeam == aServerMap[aNode] ) /* beam serves this node */
   lSIRnum = aBeam[lBeam].mPower *
            gBeamTable[aGPS[aNode].mAngleIndex][aBeam[lBeam].mAngleIndex].Mag;
else /* beam interferes with this node */
   lSIRdenom += aBeam[lBeam].mPower *
            gBeamTable[aGPS[aNode].mAngleIndex][aBeam[lBeam].mAngleIndex].Mag;
}

return ((lSIRdenom != 0 ? lSIRnum/lSIRdenom : lSIRnum));
}

void CreateServerMap(
   int aCount,
   int aBase,
   unsigned char * aServerMap)
{
   /*
    * Input:  aCount = base 10 integer to be converted
    *         aBase = base to be converted to
    * Output: aServerMap = base aBase number
    * Return: -
    * Desc:   creates a servermap by converting a base 10 number
    *         into a base aBase (number of beams) number
    * How it works: to convert a base 10 number (x) into a base b number,
    *         divide x by b.  The remainder is the least sig digit of the base
    *         b number.  Divide the quotient by b again to get the next
    *         least sig digit, etc
    */
   int lIndex = 0;

   while (aCount) {
      aServerMap[lIndex++] = aCount % aBase;
      aCount = aCount / aBase;
   }
}

int IsRedundant(
   int aNumNode,
   unsigned char * aServerMap,
   int *aNumActiveBeams)
{
   /*
    * Input : aNumNode = number of nodes
    *         aServerMap = server map to be checked for redundancy
    * Output: aNumActiveBeams = number of beams used in the server map
    * Return: 1 if redundant, 0 if not redundant
    * Desc  : determines if aServerMap is equivalent to previous
    *         server maps created by CreateServerMap() passed values
    *         of aCount ascending from 0.
    */

   int lIndex = 0,
       lMax = 0;
   for (lIndex = aNumNode-1;lIndex >= 0;lIndex--) {
      if (aServerMap[lIndex] > lMax+1)
         return (1);
      else if (aServerMap[lIndex] > lMax)
         lMax = aServerMap[lIndex];
   }
   *aNumActiveBeams = lMax+1;
   return (0);
}

void PrintServerMap(
   int aNumNode,
   unsigned char * aServerMap)
{
   /* Input:  aNumNode = number of nodes
    *         aServerMap = server map
    * Output: -
    * Return: -
    * Desc  : prints a server map to the screen
    */

   int lIndex;
   for (lIndex = 0;lIndex < aNumNode; lIndex++)
      printf("%d", aServerMap[lIndex]);
}

int OptENtoRN(
   unsigned char ** aServerMap,
   struct GPSLocation * aGPS,
   unsigned char aNumNode,
   struct BeamInformation ** aBeam,
   unsigned char aNumBeamMax,
   unsigned char * aNumBeam)
{
   /*
    * Input  : aGPS = position of nodes
    *          aNumNode = number of nodes
    *          aNumBeamMax = maximum number of beams
```

```
 * Output : aServerMap = beam to node assignment
 *          aBeam = steering angles and relative powers of the beams
 *          aNumBeam = number of beams used
 * Returns: -1 = could not allocate mem; 0 = everything is fine
 * Desc   : finds the optimal EN to RN configuration based on RN SIRs by
 *          determining
 *            1) number of beams to use
 *            2) beam to node assignment
 *            3) steering angle of beams
 *            4) relative power of beams
 */

int    lIndex = 0,
       lFinal = 0,
       lCount = 0,
       lNumBeam = aNumBeamMax;
double lMinSIR = 0,
       lBestSIR;
unsigned char          lServerMap[aNumNode];
struct BeamInformation lBeam[aNumBeamMax];

/* calculate the largest non-redundant server map */
for (lIndex = 1; lIndex < aNumNode; lIndex++)
   lFinal += MIN(lIndex, aNumBeamMax - 1) *
             pow(aNumBeamMax, aNumNode - 1 - lIndex);

/* initialize the servermap to zeros */
memset(lServerMap, 0, aNumNode);
if (!(*aServerMap = (unsigned char*)malloc(aNumNode*sizeof(unsigned char))))
   return (-1);
memset(*aServerMap, 0, aNumNode);

/* find the best servermap by maximizing the minimum SIR */
  for (lCount = 0;lCount <= lFinal; lCount++) {
       CreateServerMap(lCount, aNumBeamMax, lServerMap);
       printf("----\n");
       PrintServerMap(aNumNode, lServerMap);
       printf(" : %d\n",IsRedundant(aNumNode,lServerMap,&lNumBeam));
```

```
            if (!IsRedundant(aNumNode,lServerMap,&lNumBeam)) {
               /* optimize steering angle and power */
               for (lIndex = 0; lIndex < aNumBeamMax; lIndex++)
                  lBeam[lIndex].mPower = 1.0;
               lMinSIR = OptSteerAngle(lServerMap,aGPS,aNumNode,lBeam,lNumBeam);
               lMinSIR = OptBeamPwr(lServerMap,aGPS,aNumNode,lBeam,lNumBeam);
               /* is this the best configuration found so far? */
               if (lMinSIR > lBestSIR){
                  lBestSIR = lMinSIR;
                  *aNumBeam = lNumBeam;
                  free (*aBeam);
                  if (!(*aBeam = (struct BeamInformation*)malloc(lNumBeam*
                      sizeof(struct BeamInformation))))
                     return (-1);
                  for (lIndex = 0;lIndex < aNumNode;lIndex++)
                       (*aServerMap)[lIndex] = lServerMap[lIndex];
                  for (lIndex = 0;lIndex < lNumBeam;lIndex++){
                       (*aBeam)[lIndex].mAngle = lBeam[lIndex].mAngle;
                       (*aBeam)[lIndex].mAngleIndex = lBeam[lIndex].mAngleIndex;
                       (*aBeam)[lIndex].mPower = 1.0;
                  }
               }
               for (lIndex = 0;lIndex < lNumBeam;lIndex++) {
                  printf("lBeam[%d].mAngleIndex = %d\n",lIndex,
                         lBeam[lIndex].mAngleIndex);
                  printf("lBeam[%d].mPower = %f\n",lIndex,
                         lBeam[lIndex].mPower);
               }
               printf("lMinSIR = %f\n",lMinSIR);
            }
      }
   return (0);
}

double OptBeamPwr(
   unsigned char * aServerMap,
   struct GPSLocation * aGPS,
   unsigned char aNumNode,
   struct BeamInformation * aBeam,
   unsigned char aNumBeam)

{
   /*
    * Input : aServerMap;  aServerMap[i] = Beam number that serves node i
    *         aGPS = location of nodes
    *         aNumNode = number of nodes
    *         aNumBeam = number of beams
    * Output: aBeam = optimimal beam powers
    * Return: minimum SIR of the nodes
    * Desc  : For each beam, adjust its relative power until the min SIR of
    *         the nodes it serves is equal the the minimum SIR of the first
    *         beam.
    */

   double lMinSIR = 0.0,
          lSIR1,
          lPwrRes;
   int    lThisAdj = 1,
          lLastAdj = 1,
          lBeam = 0,
          lCount = 1;

   if (aNumBeam == 1){
      /* if number of beams equals one...then set its power to 1 */
      lSIR1 = FindMinSIRofBeam(aServerMap,aGPS,aNumNode,0,aBeam,aNumBeam);
      aBeam[0].mPower = 1.0;
      lMinSIR = lSIR1;
   }
   else {
      /* adjust each beam so its min SIR is equal to SIR1 */
      lPwrRes = gOptPrefs.mPwrInitialRes;
      while (lPwrRes >= gOptPrefs.mPwrFinalRes){
       for (lBeam = 1;lBeam < aNumBeam;lBeam++){
         lCount = 0;
         printf("\nBeamIndex = %d   PwrRes = %f\n",lBeam,lPwrRes);
         printf("%10s %10s %10s %10s\n","Ref SIR","min[SIR]",
                "Adj","Pwr");
         printf("%10s %10s %10s %10s \n","-------","--------","---","---");
         while ((lThisAdj == lLastAdj || (lCount < 2)) &&
                (lCount <= gOptPrefs.mPwrStop)){
            /* find the reference SIR */
            lSIR1 = FindMinSIRofBeam(aServerMap,aGPS,aNumNode,0,aBeam,aNumBeam);
            lCount++;
            lLastAdj = lThisAdj;
            lMinSIR = FindMinSIRofBeam(aServerMap,aGPS,aNumNode,lBeam,
                        aBeam,aNumBeam);
            if (lMinSIR > lSIR1){  /* decrease the beam's power */
               /* dividing by zero is very bad */
               if (aBeam[lBeam].mPower >= 2*lPwrRes) {
                  aBeam[lBeam].mPower -= lPwrRes;
                  lThisAdj = -1;
               }
               else
                  lThisAdj = -lLastAdj;
            }
            else {                 /* increase the beam's power */
               aBeam[lBeam].mPower += lPwrRes;
               lThisAdj = 1;
            }
            /* print some progress info */
            lSIR1 = FindMinSIRofBeam(aServerMap,aGPS,aNumNode,0,aBeam,aNumBeam);
            lMinSIR = FindMinSIRofBeam(aServerMap,aGPS,aNumNode,lBeam,
                        aBeam,aNumBeam);
            printf("%10.5f %10.5f %10d %10.3f\n",lSIR1,lMinSIR,lThisAdj,
                   aBeam[lBeam].mPower);
         }
       }
       lPwrRes *= gOptPrefs.mPwrFactor;
      }
   }
   lMinSIR = FindMinSIR(aServerMap,aGPS,aNumNode,aBeam,aNumBeam);
   return (lMinSIR);
}
```

```
double FindMinSIR(
   unsigned char *aServerMap,
   struct GPSLocation * aGPS,
   int aNumNode,
   struct BeamInformation * aBeam,
   unsigned char aNumBeam)
{
   /*
    * Input:  aServerMap = aServerMap[i] is the beam serving node i
    *         aGPS = node locations
    *         aBeamIndex = beam number to examine
    *         aBeam = beam angles, relative powers, etc
    *         aNumBeam = number of beams
    * Output: -
    * Return: minimum SIR of all the nodes
    * Descs : finds the minimum SIR among all the nodes
    */

   double lMinSIR = 0.0;
   int lNode;


   /* find the minimum SIR for this set of nodes */
   for (lNode = 0;lNode < aNumNode;lNode++) {
     double lNodeSIR = CalcSIR(aServerMap,aGPS,lNode,aBeam,aNumBeam);
     if (lNodeSIR < lMinSIR || !lNode)
        lMinSIR = lNodeSIR;
   }

   return (lMinSIR);

}



double FindMinSIRofBeam(
   unsigned char *aServerMap,
   struct GPSLocation * aGPS,
   int aNumNode,
   unsigned char aBeamIndex,
   struct BeamInformation * aBeam,
   unsigned char aNumBeam)
{
   /*
    * Input:  aServerMap = aServerMap[i] is the beam serving node i
    *         aGPS = node locations
    *         aBeamIndex = beam number to examine
    *         aBeam = beam angles, relative powers, etc
    *         aNumBeam = number of beams
    * Output: -
    * Return: minimum SIR of aBeamIndex's nodes
    * Desc:   finds the minimum SIR among the nodes that a beam serves
    */

   int    lNode  = 0,
          Flag   = 0;
   double lMinSIR = 0.0,
          lSIR    = 0.0;

   for (lNode = 0; lNode < aNumNode; lNode++){
      if (aServerMap[lNode] == aBeamIndex){
         lSIR = CalcSIR(aServerMap,aGPS,lNode,aBeam,aNumBeam);
         if (lSIR < lMinSIR || !Flag++)
            lMinSIR = lSIR;
      }
   }
   return (lMinSIR);
}

void PrintPrefs(void)
{
   /*
    * Input:   -
    * Output:  -
    * Returns: -
    * Desc:    Prints the optimization preferences
    */


}
```

A.7 optimize.c

```
/*
 * proj : RDRN
 * file : optimize.c
 * prog : Shane M. Haas and John Paden
 * desc : main program for steering angle optimiztion code
 *
 */

/*
 * global includes
 */

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

/*
 * local includes
 */

#include "opt_steer_ang.h"
#include "BeamTable.h"

/*
 * global variables
 */

   CplxPol **gBeamTable;
   struct OptimizationPrefs gOptPrefs = {0.2,0.0001,0.4,100};
```

```
/*
 * main program
 */

void main(int argc, char **argv)
{
if (argc != 4)
{
printf("%s: First Argument = Number of Beams\n", argv[0]);
printf("%s: Second Argument = Number of Nodes\n", argv[0]);
printf("%s: Third Argument = Number of Times to Run\n", argv[0]);
return;
}
   if(LoadBeamTable("beam_table.tbl",&gBeamTable))
   {
      printf("ERROR: Beam table not loaded");
   };
/* Test */
{
unsigned char * lServerMap;
struct GPSLocation lGPS[6];
struct BeamInformation * lBeam;
unsigned char   lNumBeamMax,
          lNumNode,
          lNumBeam;
   int  lIndex,
lTotal,
lTime,
lTimeFinish;

      /* Arguments */
lNumBeamMax = atoi(argv[1]);
lNumNode = atoi(argv[2]);
lTotal = atoi(argv[3]);

/* Node Positions */
lGPS[0].mAngleIndex = 29;
lGPS[1].mAngleIndex = 49;
lGPS[2].mAngleIndex = 152;
lGPS[3].mAngleIndex = 15;
lGPS[4].mAngleIndex = 60;
lGPS[5].mAngleIndex = 175;

/* Optimize EN to RN */
lTime = time(NULL);
for (lIndex = 0; lIndex < lTotal; lIndex++) {
         if (OptENtoRN(&lServerMap,lGPS,lNumNode,&lBeam,
             lNumBeamMax,&lNumBeam))
            printf("ERROR: Could not allocate memory in OptENtoRN\n");
}
lTimeFinish = time(NULL);
printf("Average Time: %f Time: %d\n",
       (double)(lTimeFinish - lTime)/atoi(argv[3]), lTimeFinish - lTime);
printf("\n\n\nBest Configuration Found:\n");
PrintServerMap(lNumNode, lServerMap);
printf("\n");
for (lIndex = 0;lIndex < lNumBeam;lIndex++)
printf("lBeam[%d].mAngleIndex = %d\n",lIndex,
        lBeam[lIndex].mAngleIndex);

}
}
```