GBT MEMO 178

The "ray" ray tracing package

Don Wells*

March 6, 1998

Abstract

The "ray" package[1] and program rayMain trace sets of rays representing wavefronts through systems of rotationally-symmetric aspheric optical elements. The starting sets of rays can represent either plane or spherical wavefronts, with feedhorn tapering. The optical elements can be de-centered and/or tilted conic sections (planes, spheres, ellipsoids, paraboloids, hyperboloids) with additional superimposed radially-symmetric aspheric terms, and they can be mirrors as well as refracting surfaces. Both foci and nearly-plane wavefronts can be analyzed.

Contents

*mailto:dwells@nrao.edu

[1] Source code of the ray package is available at ftp://fits.cv.nrao.edu/pub/gbt_dwells_ray.tar.gz (140 KB). The ray package is supplied separately from other GBT code because it is potentially useful in a wide range of other applications.

1 The ray library

An unusual feature of this ray tracing package is that it is available as the *library* `libray.a` of ray-tracing routines in ANSI-C. The individual functions of this library can be called by custom application programs coded in C; a good example is the program `srFocusTrackingTable1` which is used to verify that the GBT focus tracking algorithm [Wel98a] will produce high quality imaging as the GBT distorts due to gravity as a function of elevation. In order to do this, `srFocusTrackingTable1` must call the structural model and BFP functions [WK95b, WK95a] in addition to the `ray` functions; no conventional standalone ray tracing program could do this.

For conventional ray tracing applications a user can use program `rayMain` (Section 2, p.10) to invoke the individual ray tracing functions from a script. Program `rayMain` is analogous to commercial standalone ray tracing programs. For example, in the analysis of the imaging properties of the GBT subreflector [Wel98b], the following terse script named `srEllipsoidCase2.in` commands `rayMain` to trace 29 spherical waves of 13 rays each through the off-axis ellipsoidal mirror:

```
Digits 4 0.00001
System GBT_Subreflector
rayAddSurface "subreflector"  -0.133110 0.528 0 0 -1\
                        [4.91667,0,0]&[0,0,0] \
                        "cone"@[0,0,0]&[1,0,0]1
rayAddSurface "prime_plane" 0 0 0 0 1                   \
                        [0,0,0]&[0,0,0.798]                     \
                        "cylinder"@[0,0,0]&[1,0,0] 1
rayGenerator "spherical"@[-11,0,0]&[+0.902411,+0.312393,0]0.261677 \
                        0.02,3,6,2      0.261677,-13.0  2,13 "bundle"
rayTrace
rayGetFoci
rayPrtFoci
rayPltSystem
rayPltPS [12,12] 0.15 [-11.0,0.01,0.] orthographic srEllipsoidCase2b.ps
Quit
```

The commands shown in the example above define two surfaces, generate and trace rays representing a set of spherical wavefronts originating from a grid of 29 points near the second ellipsoid focal point, determine the locations of the foci near the first focal point, print the results to standard output and send a Postscript plot to file `srEllipsoidCase2b.ps`. These operations, which invoke the individual functions in `libray.a`, are documented in this report.

Copyright: The source files of library `ray` and program `rayMain` are being made available under a GNU "copyleft" license.[2] Each of the source files begins with a copyright notice[3] which is suppressed in the source file listings of this memo.

Acknowledgements: It is a pleasure to thank R. Fisher, R. Norrod and P. Napier for a number of valuable discussions and R.G. Tull (Univ. Texas) for supplying a listing of the original Fortran version of the tracing algorithm.

[2] See file "`GNU_GPL_2.txt`" in the `ray` distribution kit.

[3] "*Copyright (C) 1995 Associated Universities, Inc. Washington DC, USA. This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA. Correspondence concerning GBT software should be addressed as follows: GBT Operations, National Radio Astronomy Observatory, P. O. Box 2, Green Bank, WV 24944-0002 USA*".

1.1 Tracing Bundles of Rays through a System: rayTrace()

```
struct Node *rayTrace (                        /* returns traced RayBundleSet */
                       struct Node *RayBundleSet, /* the lists of input rays */
                       struct Node *System,       /* the list of surfaces    */
                       double tolerance,          /* for aspheric intercepts */
                       struct Node *Segments)     /* init this list! appends */
```

Function `rayTrace()` traces a list of lists of rays through a list of surfaces, returning a new list of lists of rays. It uses Feder's algorithm [Fed51] for tracing a ray through a centered rotationally-symmetric aspheric surface,[4] augmented by extensions [AS52] for de-centering and tilting of the surface. Feder's technique for tracing aspheric surfaces has been extended to use the numerical eccentricity [SM62, p.675], so that arbitrary conic sections can be traced, including superimposed radially-symmetric polynomial deviations from the conic sections.[5]

The surfaces to be traced are conic sections plus polynomial terms, and are computed by `rayTrace()` as

$$x = \frac{cs^2}{1+\sqrt{1-c^2s^2(1-\varepsilon^2)}} + A_2 s^2 + A_4 s^4 \qquad (1)$$

where s is the radius of ray intercept ($s^2 = y^2 + z^2$), c is the osculating curvature at the vertex ($c = 1/r$), ε is the numerical eccentricity of the conic and the A_i are the radially-symmetric polynomical coefficients. Spheres are specified by setting $\varepsilon = 0$ and curvature $c = 1/r$. The `ray` package follows Feder's convention [Fed51] that the X axis is the main optical axis of a system, and the sign convention for c is that it is positive if the center-of-curvature is to the right of the vertex ($x_{\rm coc} > x_{vertex}$). Planes are specified as spheres with curvature $c = 0$ ($r = \infty$). Paraboloids with focal length f are specified as $\varepsilon = 1$ and $c = 1/2f$. Ellipsoids with major axis a and minor axis b are specified as $\varepsilon = \sqrt{a^2 - b^2}/a$ and $c = a/b^2$. The reader can verify by inspection that equation (1) computes a paraboloid ($\varepsilon = 1$) of focal length $f = r/2$ with $A_2 = A_4 = 0$ as $x_{\rm paraboloid} = s^2/2f$.[6] A plane with $\varepsilon = 0$, $A_2 = A_4 = 0$ and $c = 0$ is computed as $x_{\rm plane} = 0$. A simple sphere with radius r is computed as $x_{\rm sphere} = s^2/r(1 + \sqrt{1 - s^2/r^2})$, where the sign of $x_{\rm sphere}$ varies with the sign of $c = 1/r$ (i.e., positive if $x_{\rm coc} > x_{vertex}$).

The concept of the extensions for de-centering and tilting is that the coordinates of both surfaces and rays are defined relative to the same origin, and rays are transformed to local coordinates just before tracing them through a surface, and then are transformed back to the original coordinate system afterward.

Argument `Segments` is a list to which items of type `Segment` (see `ray.h` in Appendix A.1) are appended by the ray tracing algorithm for plotting by function `rayPltPS()` (section 1.5, p.9); see Figure 3 for an example of plotted ray segments.

The local variable declarations within `rayTrace()` include temporary variables used by Feder in [Fed51]. `M` is the vector from the vertex of the surface and perpendicular to the incident ray and having its terminus on the ray. `M_1_2` is M_1^2, the square of the length of this vector, and `M_1x` is its X-component M_{1x}. `xi_1` (ξ_1) is the cosine of the angle of incidence, `xi_1_p` (ξ_1') is the cosine of the angle of refraction. `L` is the length of the ray intercepted between the two surfaces.

[4] If toric (non-rotationally-symmetric) surfaces, such as the tertiary which was proposed [Sri91] for the GBT, need to be traced, this algorithm will need further extensions [SM62, p.675].

[5] The basic algorithm used in `rayTrace()` was developed by the author during the summer of 1967 as part of the design work for the Coude spectrograph of the McDonald Observatory 2.7-meter telescope [Tul69, Tul72], under the direction of Robert G. Tull (Univ. Texas), the designer of the spectrograph. The algorithm was embodied in a program named `COUDE`. In 1993, the author contacted Tull in order to obtain a 26-year-old line printer listing of `COUDE`, and then translated the code from Fortran-66 to ANSI-C while typing in the first version of function `rayTrace()`. Program `COUDE` was used to analyze a spectrograph which has optical complexity comparable to the GBT: the collimator is an off-axis paraboloid, the camera is a Schmidt with the diffraction grating at the entrance pupil, and a *tilted* Schmidt corrector in the collimated beam is used to generate the necesary elliptically-symmetric Schmidt-type wavefront correction to compensate the spherical abberation of the camera.

[6] Equation (1) implies that a paraboloid can also be specified as $c = 0$, $\varepsilon = 0$, $A_2 = 1/2f$.

Function `rayTrace()` performs many of its calculations using macros defined in `mathVectorMatrix.h`, which is `included` by `ray.h`. Application programs which call `rayTrace()` and other functions of the library may also find these macros useful. The macros expand operations on three-element vectors and 3×3 matrices as *inline* scalar operations (no `for`-loops are used), in hopes that scalar optimization in compilers will improve performance.

1.2 Defining the Surfaces of the System: `rayAddSurface()`

```
struct Node *rayAddSurface(                    /* returns ptr to new node     */
                           struct Node *list,  /* list of surfaces            */
                           char surfname[],    /* descriptive string          */
                           double curvature,   /* 1/r                         */
                           double epsilon,     /* eccentricity of conic       */
                           double A_2,         /* deformation terms           */
                           double A_4,
                           double index_ratio, /* N/N_1, -1 means mirror      */
                           double S[],         /* XYZ of vertex               */
                           double E[],         /* Euler angles of vertex tilt */
                           enum VignetteType
                                  vign_type,   /* VIGN_CYLINDER | VIGN_CONE   */
                           double VO[],        /* vignette origin XYZ         */
                           double VV[],        /* vignette direction cosines  */
                           double VR)          /* radius, linear | radians    */
```

Argument `epsilon` is the numerical eccentricity ε; `curvature` is the inverse of the radius of the sphere which osculates to the surface at the vertex point. The `index_ratio` argument is the ratio of the indicies of refraction $\mu_1 = N/N_1$, where N is the index to the left of the surface and N_1 is the index to the right. For a focal plane or aperture stop, which neither refracts nor reflects, set $\mu_1 = 1$; for a mirror, set $\mu_1 = -1$.

The position of the vertex of the surface is given by argument `S[]`, relative to an arbitrary origin chosen by the user. For example, in Table 3 the GBT's prime focal point has been chosen as the origin, and so the vertex of the paraboloidal primary is at $X = -60$ meters.[7] The vertex point of a plane is undefined, and the vertex point of a sphere can be varied by varying the tilt `E[]`; the user should supply whatever values seem reasonable in such cases.

Although the comment in the function prototype above asserts that argument `E[]` is the "*Euler* angles of vertex tilt" of the surface, *this is not correct* — the algorithm currently implemented in `rayTrace()` uses `E[0]` to rotate about the X axis, `E[1]` to rotate about the Y axis and `E[2]` to rotate about the Z axis, *in that order.* For example, in Table 3 (p.12) the tilt of the GBT's subreflector is specified as `E[2]`, rotation about Z, with value -0.097214 radians ($-5.57°$), and the vertex is specified to be (4.8935,-0.4772,0.0) meters. It is likely that future versions of `rayTrace()` will implement a true Euler angle convention for `E[]`.

The vignetting arguments `vign_type`, `VO[]`, `VV[]` and `VR` are provided to define the edge of the optical element, however *the current implementation of function* `rayTrace()` *does not use them.* When the vignetting logic is added to `rayTrace()` it will be necessary to implement an output list for the rays which are vignetted; this will support "spillover" calculations in radio telescopes. The author's present thinking is that this will be done by adding a pointer to that list of bundles of rays to the `Surface` struct; tools will be provided to sum the fluxes of rays in the various lists. The concept of the vignetting arguments is that two types of aperture stops will be supported: surfaces delimited by cylinders and surfaces delimited by cones. The GBT paraboloidal primary is an example of the cylindrical case and its ellipsoidal secondary is

[7] The `ray` package does not define the unit used for positions of surfaces, but it does provide arguments `tolerance` and `d` (number of digits after decimal point) in functions `rayTrace()`, `rayPrtSystem()`, *etc.* Users can consider the positions and tolerances to be inches, centimeters or meters as desired. Angle arguments are always radians.

steps	items 1-D	2-D	3-D
0	1	1	1
1	3	5	7
2	5	13	33
3	7	29	123
4	9	49	257
5	11	81	515
6	13	113	925
7	15	149	1419
8	17	197	
9	19	253	
10	21	317	
11	23	377	
12	25	441	
13	27	529	

Table 1: Items (rays,cases) as function of number of "steps"

an example of the cone case. The cylinder or cone is defined by a point, a vector and a radius. An example of the use of these arguments in the cylinder case is shown in Table 3, for the case of the "`main_mirror`", where an aperture stop of 50 meters radius is specified to be located at (-60,-54,0) meters, normal to the X-axis of the system. In practice an optical system like the GBT can be traced with reasonable accuracy by the present implementation of `rayTrace()`, because function `rayGenerator()` can produce bundles of rays which will suffer negligible vignetting.

1.3 Generating Bundles of Rays: `rayGenerator()`

```
void rayGenerator(                          /* no value returned            */
                  struct Node *RayBundleSet, /* appends this list of Bundles */
                  char bundle_name[],       /* default used if NULL         */
                  char wave_type[],         /* "plane" || "spherical"       */
                  double wave_point[],      /*         XYZ                  */
                  double wave_direction[],  /*      unit_vector             */
                  double wave_radius,       /* linear  || angular           */
                  double case_step,         /* angular || linear            */
                  int    case_steps,        /*    #steps off-axis           */
                  int    axis_mask,         /* 7=XYZ,4=X,2=Y,6=XY,..        */
                  int    ray_steps,         /*    #rays off-axis            */
                  double taper_angle,       /* of feed horn (radians)       */
                  double taper_db,          /* down at taper_angle          */
                  int ColorCode_1,          /*    first ColorCode           */
                  int ColorCode_2,          /*    last  ColorCode           */
                  enum ColorType assign_by) /* COLOR_BUNDLE|COLOR_RAY       */
```

The concept is that a "bundle" of rays can be generated and appended to an existing list of bundles. Each bundle is intended to represent an individual wavefront case; sets of bundles can be traced, analyzed, printed and plotted as sets, with single calls to functions, because the sets are represented as lists of lists of rays and lists of computed results.

Two types of bundles can be generated, plane and spherical; character string argument `wave_type[]` specifies

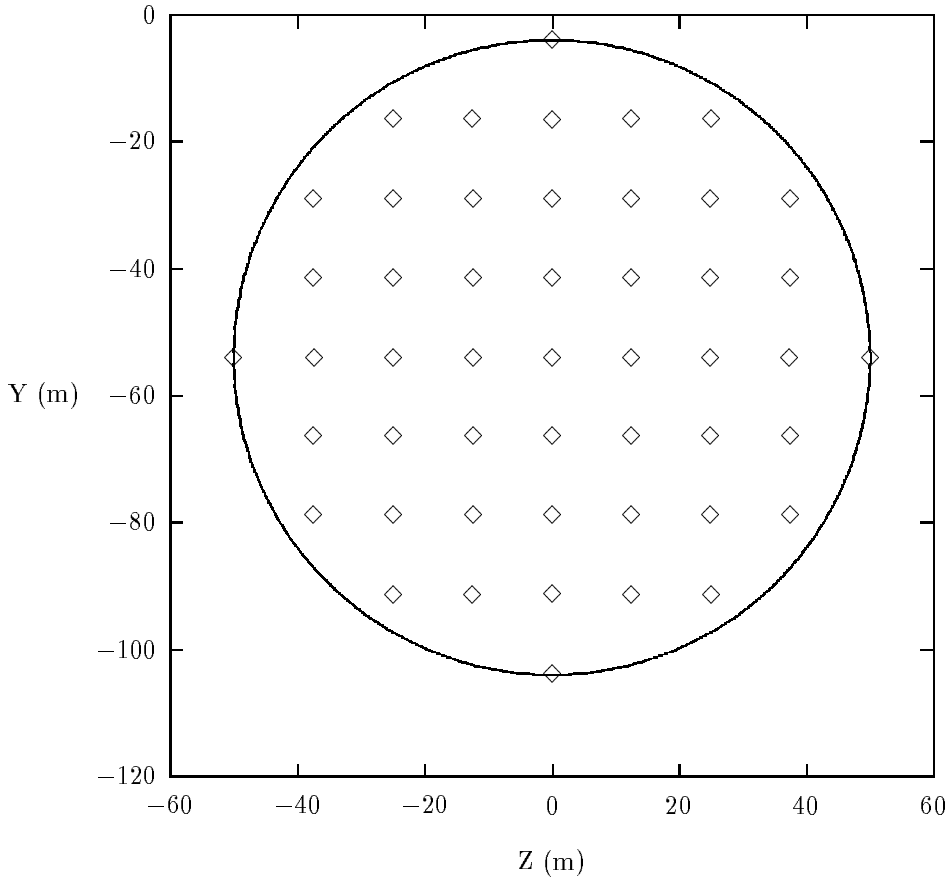

Figure 1: Ray pattern in 100 meter aperture (`ray_steps` = 4 ↦ 49 rays)

this. A bundle of rays will originate at `wave_point[]` and will proceed in `wave_direction[]`. Actually, `rayGenerator()` appends a *set* of ray bundles with a single call; the arguments and algorithms provided are intended to support typical analytical problems in optical systems, such as tracing plane wavefronts with the angle off-axis being stepped, or tracing spherical wavefronts with the position of the feedhorn being stepped relative to the on-axis focal point. Argument `axis_mask` specifies which axes will be stepped. Argument `case_steps` specifies the number of steps ± off-axis, and `case_step` is the step size. For each step a bundle of rays will be generated of size `wave_radius` and with `ray_steps` rays ± about the axis of the bundle. Function `rayGenerator()` can produce vast numbers of rays from innocuous-looking values of these few arguments! For example, in Table 3 (p. 12) the four arguments "`0.0001388,2,2,4`" generate five sets of 49 rays each, 245 rays in total. This is a plane wave case, and so `case_step`=0.0001388 is interpreted as 0.14 mr (29 arcsec). The `case_steps` argument is 2, so five cases (-2, -1, 0, +1, +2) will be computed for each active axis. The `axis_mask` is 2, which we see is the Y axis only, so only five cases will be generated. The `ray_steps` argument is 4; the number of rays to be generated for this value is given as 49 in the "2-D" column of Table 1.[8] With a few keystrokes we could change this argument to 6, and thereby generate 113 rays, more than doubling the sampling of the wavefront. In practice, when debugging the setup of a new problem, it is convenient to start with a small value for `ray_steps` so that the calculation will go rapidly, and then increase the value to improve accuracy when the problem appears to be operating correctly.

Arguments `taper_angle` and `taper_db` implement radio-style beam tapering (in optical systems aperture

[8] The number of items is simply $2n + 1$ for the 1-D case, but for 2-D and 3-D it is limited by the `wave_radius` argument; Figure 1 shows how the 49 rays fit into the circular area.

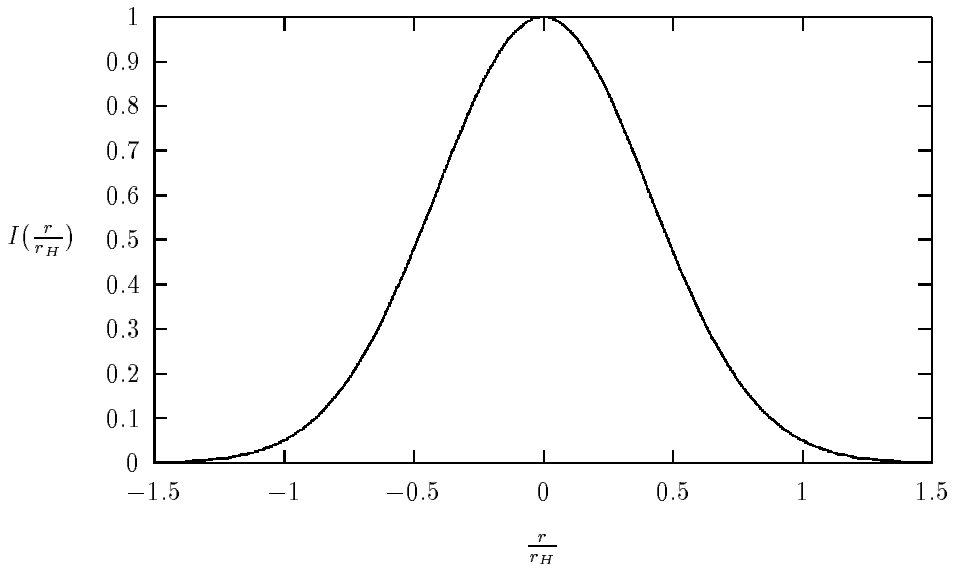

Figure 2: Power-weighting of rays ($q = 43.17, \theta_H = 15°$)

tapering is called "apodization"). The relative E-field pattern of the feedhorn in decibels is

$$E(\theta) = 20 \log(\cos\theta)^q, \tag{2}$$

where θ is the angle off-axis [RSCWL81]. Parameter q is specified implicitly by giving the value of $E(\theta)$ at the edge of the beam. If θ_H is the subtended half-angle of the subreflector, we have

$$E(\theta_H) = 20q \log\cos\theta_H, \tag{3}$$

and we can solve for the exponent q as

$$q = \frac{E(\theta_H)}{20 \log\cos\theta_H}. \tag{4}$$

For example, in [Nor90b] we have $\theta_H = 15°$ and $E(\theta_H) = -12$ dB, which implies $q = 43.17$. The relative power of the rays in the generated wave is computed as

$$I(r) = \cos^{2q}(\frac{r}{r_H}\theta_H), \tag{5}$$

where r_H is the radius in the wavefront corresponding to angle θ_H, and this intensity (power) is used in the `ray` library as a weighting factor for analyzing foci and wavefronts. For example, a taper of -12 dB is a weight of 0.0630 for marginal rays (see Figure 2). An un-apodized (un-tapered, uniformly weighted) optical system can be traced by setting the `taper_db` parameter to zero.

The three color arguments will assign color codes to individual rays or to individual bundles of rays. In Table 3 these three arguments are "`2,13 "bundle"`"; `"bundle"` means that the rays of the five cases (angle off-axis stepping by 29 arcsec) will have five different colors, starting with color code "2" (i.e., codes 2, 3, 4, 5 and 6 will be used). The colors assigned to the codes are arbitrary.

1.4 Analyzing foci and wavefronts: `rayGetFoci()` and `rayGetPlanes()`

```
struct Node *rayGetFoci(                          /* returns list of Focus */
                        struct Node *RayBundleSet, /* list of lists of rays */
                        double tolerance,          /* position accuracy     */
                        struct Node *Segments)     /* appends to Segments   */
```

Function `rayGetFoci()` uses a least-squares regression to solve for the XYZ coordinates of a focal-point about which a nearly spherical wavefront converges to the smallest RMS volume. The regression also produces the pathlength of the wavefront for the focal point. The algorithm then selects a delta pathlength which is large ($50\times$) compared to the 3-D RMS of the ray points on the wavefront about the focal point solution. It subtracts this delta pathlength from the pathlength produced by the regression and computes the XYZ coordinates of this pathlength for each ray, and then computes the difference between the delta pathlength and the Euclidean distance between these XYZ coordinates and the XYZ position of the focal point; *i.e.*, it forms the radial residual between the ray points and a sphere centered on the focal point. The RMS of these residuals is the RMS phase deviation of the nearly spherical wavefront. The rays are weighted by their intensity (radio power), as discussed in Section 1.3. This calculation of the focal point location ignores wave interference effects, which will be significant at low frequencies (< 5GHz?) but are less and less important as frequency is increased.

Function `rayGetFoci()` adds segments to the segment list to draw a $\pm 3\sigma$ 3-D box (sides $6\sigma_x$, $6\sigma_y$, $6\sigma_z$) centered on each focal point that it computes (see Figure 4 for examples).

```
struct Node *rayGetPlanes(                          /* return list of planes */
                          struct Node *RayBundleSet, /* list of lists of rays */
                          double yz_axis[],          /* Zernike analysis axis */
                          double yz_radius,          /* Zernike radius        */
                          char print_mode[])         /* "verbose"|"silent"    */
```

For the analysis of nearly-planar wavefronts, we need to fit the pathlengths of rays across the aperture with Zernike polynomials [BW59], and identify the terms which represent the primary (Seidel) aberrations. The argument `nord` to function `mathZernike()` in `rayGetPlanes()` is

```
int nord[]=4,3,2,-1; /* max_orders for Seidel abberations */,
```

which will command `mathZernike()`[9] to compute the nine terms shown in Table 2 (*cf.* Table 6).

During the fitting process the y, z coordinates of each ray intercepting the plane are converted to polar coordinates ρ, θ relative to `yz_axis[]`. The formulæ are

$$r = \sqrt{\Delta y^2 + \Delta z^2}, \tag{6}$$

$$\cos\theta = \Delta y / r, \tag{7}$$

[9] The function prototype is:

```
int mathZernike(                    /* returns #coeffs in u[] & ui[]    */
                double r,           /* radius in range 0-->1.0          */
                double cost,        /* cosine of theta                  */
                double sint,        /* sine   of theta                  */
                int    nord[],      /* order limits                     */
                double u[],         /* Zernike coefficients return here */
                int    ui[])        /* Indicies of coeffs return here   */
```

It is likely that the details of this C function will be discussed in a future GBT Memo, as a part of the active surface servo development effort. A similar set of polynomials was used earlier in the GBT project [Nel90] to analyze the shapes of the GBT panels.

index	Zernike term	n	m	Function	Seidel Abberation
0	A_{0000}	0	0	1	Zero Point (Pathlength)
1	A_{2000}	2	0	$2\rho^2 - 1$	Curvature (Defocus)
2	A_{4000}	4	0	$6\rho^4 - 6\rho^2 + 1$	Spherical Abberation
3	A_{1010}	1	1	$\rho\cos\theta$	Tilt (Pointing Error)
4	A_{1011}	1	1	$\rho\sin\theta$	*[sine term]*
5	A_{3010}	3	1	$(3\rho^3 - 2\rho)\cos\theta$	Coma
6	A_{3011}	3	1	$(3\rho^3 - 2\rho)\sin\theta$	*[sine term]*
7	A_{2020}	2	2	$\rho^2\cos 2\theta$	Astigmatism
8	A_{2021}	2	2	$\rho^2\sin 2\theta$	*[sine term]*

Table 2: Seidel (3^{rd} order) terms fitted to wavefronts by `rayGetPlanes()`

$$\sin\theta = \Delta z / r \tag{8}$$

and

$$\rho = r / r_{\max}, \tag{9}$$

where $r_{\max}$ is argument `yz_radius`.

If argument `print_mode[]` has value `"verbose"`, `rayGetPlanes()` will print a listing of the least squares solution for the coefficients, the RMS of fit and the correlation matrix, *for the first bundle only.* This is mainly intended as a debug aid, because `rayPrtPlanes()` produces a more readable listing of the coefficients, and does it for all of the bundles which were traced.

1.5 Printing & plotting from lists (`rayPrtPlanes()`, `rayPltPS()`, *etc*)

```
void rayPrtSystem(struct Node *list,          /* list of surfaces           */
                  int d)                      /* digits after decimal point */
```

A good example of the output produced by this command is shown in the middle part of Table 4 (p.14). The curvatures are printed in "`g`" format with six significant figures, the eccentricities with five figures, the A_i with two figures and index ratios μ_i with five figures. The positions `S[`i`]` are printed with `d` digits after the decimal point, while the `E[`i`]` angles are always printed with 5 decimal places ($10\mu r \approx 2$ arcsec precision). The primary purpose of this listing is to facilitate debugging of optical prescriptions.

```
void rayPrtFoci (struct Node *foci_list,      /* list of Focus structs      */
                 int d)                       /* digits after decimal point */
```

A good example of the output produced by this command is shown in the lower part of Table 4 (p.14). The focal point solution is printed for each bundle of rays that was traced. The "`n`" column is the number of rays in the i^{th} bundle. Columns (`xc`,`yc`,`zc`) are the positions of the phase centers of the converging nearly-spherical wavefronts, `lc` is the path length to the phase center, (`xs`,`ys`,`zs`) are the mean errors (σ_x,σ_y,σ_z) of the phase center coordinates and `ls` is the mean error of the pathlengths `lc`. These values are printed with `d` digits precision.

```
void rayPrtPlanes (struct Node *planes_list, /* list of Plane structs      */
                   int d)                    /* digits after decimal point */
```

A good example of the output produced by this command is shown in the lower part of Table 6 (p.17), where the values of the Seidel terms are printed for each bundle of rays which was traced. The quantity being fitted

is the pathlength to the final aperture stop; the mean pathlength of the wavefront is given as A_{0000}. Only those coefficients which are non-zero to `d` digits precision are printed—negligible values are printed as blanks. The values are printed with `d` digits after the decimal point. `rayPrtPlanes()` prints the values of terms A_{1010} and A_{1011} (Wavefront Tilt, Pointing Error) in *milliradians* rather than the wavefront amplitude; it does this by multiplying the amplitudes by `1000.0/yz_radius` (the latter is an argument supplied to `rayGetPlanes()` which is transferred in `struct Plane`). `rayPrtPlanes()` prints a warning message if the RMS wavefront residual after fitting the 3^{rd}-order Zernike terms is greater than zero to `d` digits precision; *i.e.,* the residual represents the 5^{th} order terms of the wavefront.[10]

```
void rayPrtBundles(struct Node *set,          /* list of lists of rays      */
                   int d)                     /* digits after decimal point */
```

This is a debug utility.

```
void rayPrtSegments(struct Node *list,        /* list of ray segments       */
                    int d)                    /* digits after decimal point */
```

This is a debug utility.

```
void rayPltSystem(                       /* returns lines in segments list */
                  struct Node *system,   /* list of optical elements       */
                  struct Node *segments) /* list of line segments          */
```

The current implementation of this function merely plots the axes of the optical system; a future version will walk through the list of surfaces `system` and will plot a mesh of lines for each of them, delimited by their respective vignetting parameters. Lines plotted are appended to the list of `segments`.

```
void rayPltPS(                        /* returns PS in psname[]         */
              struct Node *segments,  /* list of line segments          */
              double height_cm,       /* height of PS in centimeters    */
              double width_cm,        /* width  of PS in centimeters    */
              double width,           /* width in units of System       */
              double to_point[],      /* XYZ of point in center of PS   */
              char plt_mode[],        /* "Orthographic"|"Perspective"   */
              char psname[])          /* Postscript output file         */
```

A view of the list of `segments` is computed as the Postscript file `psname[]`. The size of the Postscript image in centimeters is (`width_cm`,`height_cm`), and argument `width` specifies the width in the units of the optical system. The `to_point[]` will appear in the center of the image. Although `plt_mode` is provided, it is not implemented in the current version of `rayPltPS()`; the implementation always computes `"Orthographic"` projections with $+X$ to the right and $+Y$ pointing up, looking toward $-Z$.

2 The ray-tracing main program `rayMain`

Program `rayMain` contains a simple command language parser which recognizes function names, parses argument values and passes the arguments to the functions. The algorithm of the parser is the following steps:

[10] It is shown in [Wel98a] that the RMS wavefront residual (the combined 5^{th} order terms) of the GBT Gregorian optical system for $E = 0°$ and for $E = 90°$ is about 15μ, whereas the 3^{rd}-order terms have typical amplitudes of a millimeter, $50\times$ or more larger than the RMS residual. *The GBT is a* 3^{rd}*-order system to high accuracy.*

1. Read next input line and append it to the command line being accumulated
2. If the last character of the input line is the character "\", remove it and control goes back to step 1.
3. If the first character of the command line is "#", it is a comment; it is echoed to the standard output, the command line is cleared, and control goes back to step 1.
4. The command line is scanned for occurrences of any of the characters ",()[]"&@" and tab; they are changed to blanks.
5. The command line is scanned for occurrences of two or more blanks; they are changed to single blanks.
6. If the first character of the command line is a blank, the blank is deleted.
7. If the last character of the command line is a blank, the blank is deleted.
8. If the command line is empty (all blank), control goes back to step 1.
9. The transformed command line is echoed to standard output; if longer than 65 characters it is printed on more than one line in segments of 65 characters.
10. The command code is parsed as the initial string of characters up to the first blank. If the code is "Quit", rayMain exits.
11. The command code is tested against the list of valid codes: Digits, SetName, System, rayAddSurface, rayGenerator, rayGetFoci, rayGetPlanes, rayPltPS, rayPltSystem, rayPrtBundles, rayPrtFoci, rayPrtPlanes, rayPrtSegments, rayPrtSystem and rayTrace. If the code is recognized, the blank-delimited arguments are parsed by sscanf() formats in which the successive format codes are appropriate for the data types of the arguments of the functions to be invoked. Then the specified function is called with the arguments. After the function returns, control goes back to step 1.

The "Digits" command wants two arguments: the d value and the tolerance value to be used in calls to the functions. The print commands, whose functions take argument "d", will get the number of digits from the "Digits" command, and rayTrace will use the tolerance.

The "System" command wants one argument, the name to be given to the optical system which will be defined by the rayAddSurface() calls.

The "SetName" command wants one argument, the name to be given to the bundles of rays which will be generated by subsequent rayGenerator() calls. A default name choice will be used if this command is not supplied. A good example of the use of this command is shown in five occurrences in the lower part of Table 5 (p.16), and the result of the operation is shown in the bundle_name column of the rayPrtPlanes output in the lower part of Table 6 (p.17).

All string arguments must be single strings—they cannot contain embedded blanks. Underscores can be used to separate words; good examples of this technique are the system name and three of the surface names in the command script shown in Table 3 (p.12). The double-quotes around the surface name main_mirror in the script are irrelevant—the command parser strips them, and would regard main and mirror as two arguments rather than one if a blank were present instead of the underscore. The quotes are used as punctuation in these cases because the author judged that they make the script more readable.

Other punctuation symbols and the backslashes are allowed to make scripts more readable, but they are irrelevant to command interpretation because rayMain strips them at steps 2 and 4 above. The author uses the punctuation characters to group arguments into logical, mnemonic patterns.

The function arguments for lists of bundles of rays, lists of surface elements and lists of line segments are not supplied with the commands to rayMain. These lists are created and managed by rayMain automatically. There is no command which will delete these lists and re-initialize them—rayMain is intended to used to execute one problem at a time. To calculate two problems, run rayMain twice with separate scripts.

```
# Testcase1: plane waves entering GBT Gregorian at rigging angle
# D.Wells, NRAO-CV, 1995-07-06,08-29.

[GNU GPL copyright notice omitted]

Digits 4 0.00001

System GBT_Gregorian_at_44d
#       unit is meters. origin at prime focus.
#       eps=0.528, e=5.5m, a=e/eps=10.4167m, b=sqrt(a^2-e^2)=8.8463m
#       r0=b^2/a=7.5126m, c=1/r0=0.133110, d=a-e=4.9167m
#       alpha=17.89878d=0.312393r, beta=5.56996d=0.097214r
#       vertex=(F1+d*cos(beta),F1-d*sin(beta),0)&(0,0,-beta)
#       the_big_mirror: fp=60.0m, r0=120.0m, c=1/r0=0.0083333
#       greg_focus=(F1-2e*cos(beta),F1+2e*sin(beta),0),
rayAddSurface "main_mirror" +0.0083333333 1 0 0 -1              \
                        [-60,0,0]&[0,0,0]               \
                        "cylinder"@[-60,-54,0]&[1,0,0] 50
#       tilt prime focus plane to 45.7d (prime focus box orientation):
rayAddSurface prime_plane 0 0 0 0 1                     \
                        [0,0,0]&[0,0,0.798]                     \
                        "cylinder"@[0,0,0]&[1,0,0] 1
rayAddSurface subreflector  -0.133110 0.528 0 0 -1\
                        4.8935 -0.4772 0     0 0 -0.097214 \
                        "cone"@[0,0,0]&[1,0,0]1
rayAddSurface greg_plane 0 0 0 0 1 \
                        [-10.9481,1.0677,0]&[0,0,+0.312414] \
                        "cylinder"@[0,0,0]&[1,0,0]1
rayPrtSystem

# Generate plane wave at origin (prime focus), 100m diameter 54m offaxis
# grid spacing 0.1388mr=28.63asec corresponds to 2.1in-spacing Q-band horns.
# feed taper is -13db at 15d=0.26rad
rayGenerator "plane"@[0,-54,0]&[1,0,0]50 \
                        0.0001388,2,2,4 0.26,-13.0      2,13 "bundle"

rayTrace
rayGetFoci
rayPrtFoci
rayPltSystem
rayPltPS [12,10] 100.0 [-20,-47,0] orthographic rayTestCase1a.ps
rayPltPS [10,10] 0.10 [-10.9481,1.0677,0] orthographic rayTestCase1b.ps
Quit
```

Table 3: Input file for Foci example

2.1 Example 1: Multiple plane waves imaged to GBT Gregorian foci

The script shown in Table 3 defines the GBT surfaces for $E = 44°$ (the rigging angle) in the order in which an incident plane wave from the sky encounters them; the subreflector geometry parameters are taken from [Nor90a]. The comments in this script give the details of calculation of a number of key quantities of interest which were used to compute the arguments to the ray tracing functions. The `prime_plane` and `greg_plane` surfaces are defined in order to make these important locations explicit in the optical prescription—they have $\mu_i = 1$, and so do not deflect the rays, even though function `rayTrace` does do the full tracing calculation on them. The origin is at the prime focal point, and the unit is meters, so the vertex of the `main_mirror` is at $x = -60$ and the `prime_plane` is at $x = 0$. The arguments to `rayGenerator` create five plane waves starting at the origin ($x = 0$), with tilts stepping by about 29 arcsec and with each wave represented as 49 rays. Because the `main_mirror` is to the left of the origin, the wave moves to left initially.

Figure 3 shows how the five sets of rays proceed through the GBT (because `rayPltSystem` does not yet plot the surfaces, we see only the points where the rays change direction). The width of this scene is 100 meters. Figure 4 shows the region around the Gregorian focus greatly enlarged (width 100 mm, 1:1 scale). We see that the on-axis wave focusses perfectly but that the off-axis waves focus to caustic surfaces.

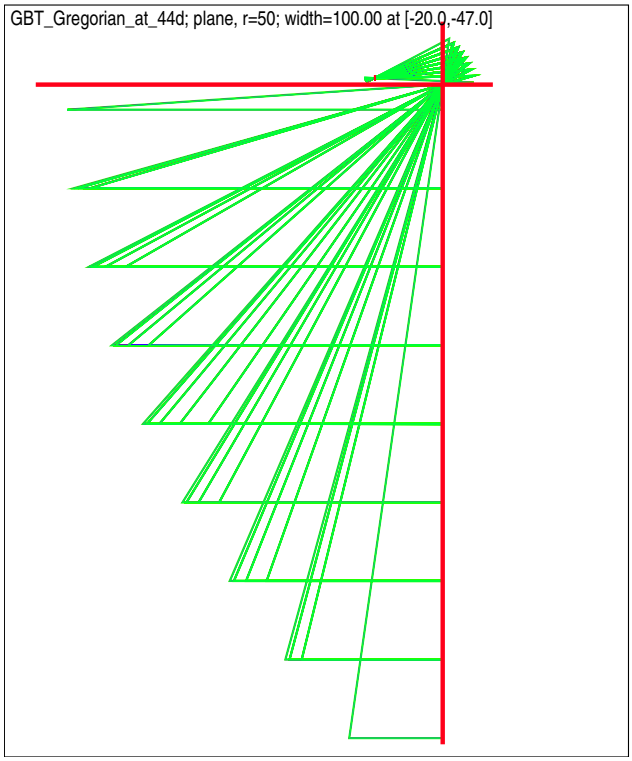

Figure 3: Overview of rays traced through the GBT (`rayTestCase1a.ps`)

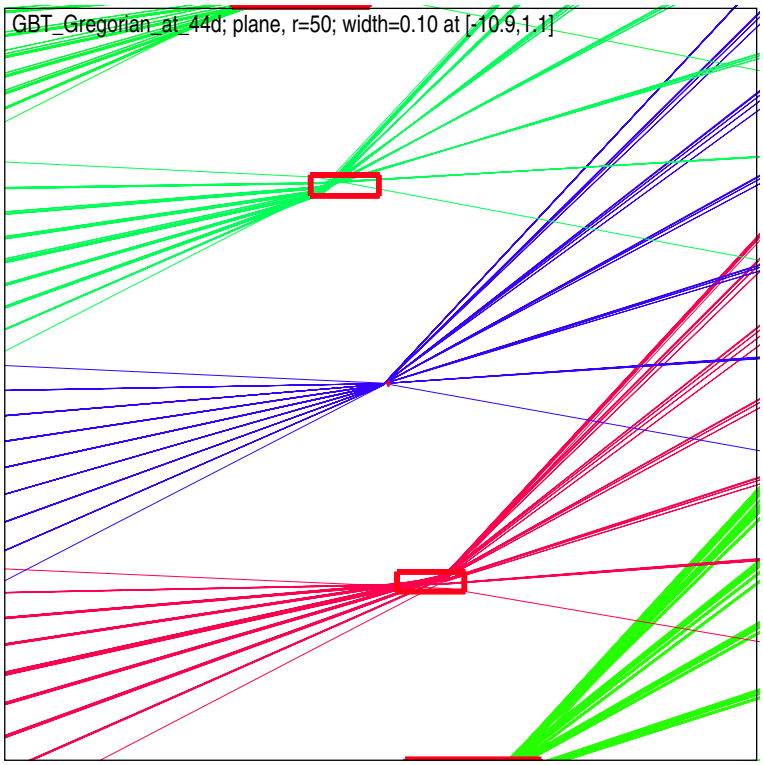

Figure 4: Foci at the Gregorian focal plane (`rayTestCase1b.ps`)

```
# Testcase1: plane waves entering GBT Gregorian at rigging angle
# D.Wells, NRAO-CV, 1995-07-06,08-29.
[GNU GPL copyright notice omitted]
# Command=<Digits 4 0.00001>
# Command=<System GBT_Gregorian_at_44d>
#       unit is meters. origin at prime focus.
#       eps=0.528, e=5.5m, a=e/eps=10.4167m, b=sqrt(a^2-e^2)=8.8463m
#       r0=b^2/a=7.5126m, c=1/r0=0.133110, d=a-e=4.9167m
#       alpha=17.89878d=0.312393r, beta=5.56996d=0.097214r
#       vertex=(F1+d*cos(beta),F1-d*sin(beta),0)&(0,0,-beta)
#       the_big_mirror: fp=60.0m, r0=120.0m, c=1/r0=0.0083333
#       greg_focus=(F1-2e*cos(beta),F1+2e*sin(beta),0),
# Command=<rayAddSurface main_mirror +0.0083333333 1 0 0 -1 -60 0 0 0 0 0 cy|
#         |linder -60 -54 0 1 0 0 50>
#       tilt prime focus plane to 45.7d (prime focus box orientation):
# Command=<rayAddSurface prime_plane 0 0 0 0 1 0 0 0 0 0 0.798 cylinder 0 0 |
#         |0 1 0 0 1>
# Command=<rayAddSurface subreflector -0.133110 0.528 0 0 -1 4.8935 -0.4772 |
#         |0 0 0 -0.097214 cone 0 0 0 1 0 0 1>
# Command=<rayAddSurface greg_plane 0 0 0 0 1 -10.9481 1.0677 0 0 0 +0.31241|
#         |4 cylinder 0 0 0 1 0 0 1>
# Command=<rayPrtSystem>

        -=-< GBT_Gregorian_at_44d >-=-
Surface Properties:
 i         name          curv    eps     A_2    A_4       mu
--      --------        -----   ----    ----   ----     ----
 1   main_mirror   0.00833333       1      0      0       -1
 2   prime_plane            0       0      0      0        1
 3  subreflector     -0.13311   0.528      0      0       -1
 4    greg_plane            0       0      0      0        1
Vertex Positions and Tilts:
 i          name      S[0]    S[1]    S[2]      E[0]     E[1]     E[2]
--     ---------      ----    ----    ----     -----    -----    -----
 1   main_mirror  -60.0000  0.0000  0.0000   0.00000  0.00000  0.00000
 2   prime_plane    0.0000  0.0000  0.0000   0.00000  0.00000  0.79800
 3  subreflector    4.8935 -0.4772  0.0000   0.00000  0.00000 -0.09721
 4    greg_plane  -10.9481  1.0677  0.0000   0.00000  0.00000  0.31241

# Generate plane wave at origin (prime focus), 100m diameter 54m offaxis
# grid spacing 0.1388mr=28.63asec corresponds to 2.1in-spacing Q-band horns.
# feed taper is -13db at 15d=0.26rad
# Command=<rayGenerator plane 0 -54 0 1 0 0 50 0.0001388 2 2 4 0.26 -13.0 2 |
#         |13 bundle>
# Command=<rayTrace>
# Command=<rayGetFoci>
# Command=<rayPrtFoci>

               -=-< plane, r=50 >-=-
  i          bundle_name   n       xc       yc       zc       lc     xs     ys     zs     ls
 --     ---------------- ---   ------   ------   ------  -------  -----  -----  -----  -----
  1    -0.00028  0.00000  49 -10.9361   1.0151   0.0000 140.8236 0.0030 0.0009 0.0004 0.0031
  2    -0.00014  0.00000  49 -10.9416   1.0414  -0.0000 140.8289 0.0015 0.0004 0.0002 0.0015
  3     0.00000  0.00000  49 -10.9473   1.0676   0.0000 140.8342 0.0000 0.0000 0.0000 0.0000
  4     0.00014  0.00000  49 -10.9530   1.0939   0.0000 140.8394 0.0015 0.0005 0.0002 0.0016
  5     0.00028  0.00000  49 -10.9589   1.1202   0.0000 140.8447 0.0030 0.0009 0.0004 0.0031

# Command=<rayPltSystem>
# Command=<rayPltPS 12 10 100.0 -20 -47 0 orthographic rayTestCase1a.ps>
# Command=<rayPltPS 10 10 0.10 -10.9481 1.0677 0 orthographic rayTestCase1b.|
#         |ps>
# Command=<Quit>
```

Table 4: `rayMain` output for Foci example

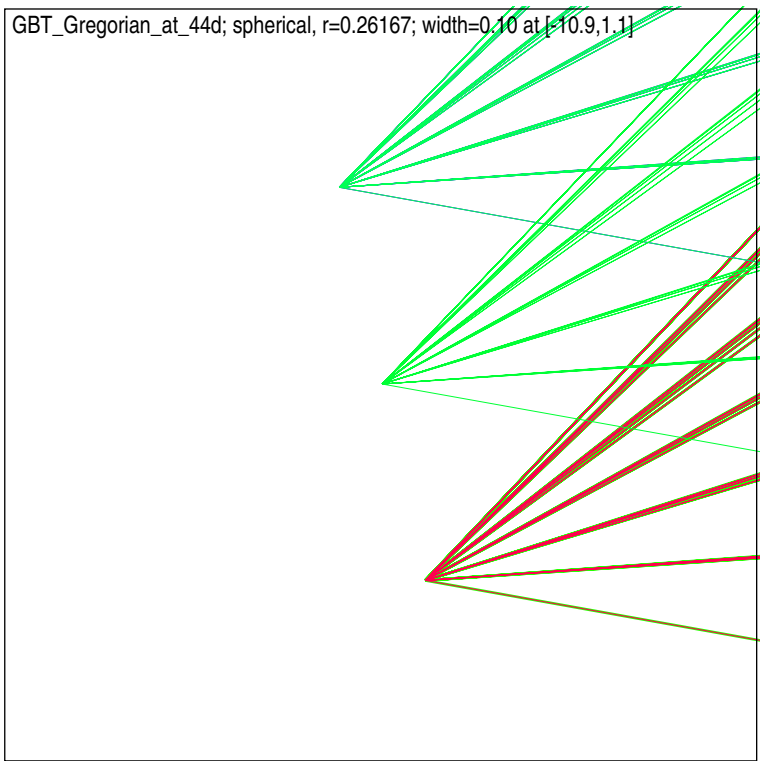

Figure 5: Q-band horns emitting spherical waves (`rayTestCase2a.ps`)

2.2 Example 2: Spherical waves from Q-band horns produce plane waves

In this example we trace cones of rays (spherical waves) originating from the phase centers of the four horns proposed for the 40-50 GHz Q-band receiver system [WS95] and we analyze the resulting nearly-plane wavefronts at the GBT aperture plane. Figure 5 is a view of the Gregorian focal point with width 100 mm, reproduced at 1:1 scale (these feedhorns are tiny!). The upper and lower ray cones are actually two cones projected one on top of the other. The central cone of rays is the on-axis case which is computed as a reference. The input script shown in Table 5 defines the GBT at the rigging angle in the same manner as in the first example, but in opposite order, and it defines the `prime_plane` as the last intercept point for the rays. Because the arrangement of the four feedhorns does not correspond to any combination of the `rayGenerator()` arguments, the five ray bundles are computed as five separate executions of `rayGenerator` with different bundle names.

The output produced in this example calculation is in Table 6. The interesting feature is the `rayPrtPlanes` output at the bottom of the table. The five nearly-plane wavefronts have been analyzed, and their 3^{rd}-order Zernike coefficients, the Seidel abberations, are tabulated. Most of the cells of the tabular listing are blank, meaning that those coefficients are *zero* in the 4^{th} decimal place (note `Digits 4` at the top of Table 6). The A_{0000} values are the pathlengths for the four horns in meters, differing by only 100μ. The non-zero A_{1010} and A_{1011} coefficients are the *tilts* of the four wavefronts produced by the off-axis feedhorns. The A_{1011} (sine) terms are wavefront tilt in azimuth, out of the GBT's plane of symmetry, and they are all the same value, ± 0.1388. The unit for these tilt values is *milliradians*, as was mentioned in Section 1.5 in the discussion of `rayPrtPlanes()`. So, four beams are off-axis in aximuth by ± 0.1388 mr $= \pm 28.63$ arcsec. The A_{1010} tilts are in elevation, and they differ by about one percent—the scale factor between linear focal plane displacement and angle on the sky in elevation is varying significantly. Because only the tilt terms are non-blank in this listing, and `rayPrtPlanes` has not warned us about significant RMS wavefront errors (5^{th}-order terms), we can conclude that these four 100 meter-wide wavefronts are *planes* to better than 100μ.

```
# Testcase#2: spherical waves from proposed 40-50_GHz feedhorns through GBT
# D.Wells, NRAO-CV, 1995-07-28,08-29.
[GNU GPL copyright notice omitted]
Digits 4 0.00001
System GBT_Gregorian_at_44d
#       unit is meters. origin at prime focus. cone from feedhorn to right.
#       eps=0.528, e=5.5m, a=e/eps=10.4167m, b=sqrt(a^2-e^2)=8.8463m
#       r0=b^2/a=7.5126m, c=1/r0=0.133110, d=a-e=4.91667m
#       alpha=17.89878d=0.312393r, beta=5.56996d=0.097214r
#       vertex=(F1+d*cos(beta),F1-d*sin(beta),0)&(0,0,-beta)
#       the_big_mirror: fp=60.0m, r0=120.0m, c=1/r0=0.0083333
rayAddSurface "subreflector"  -0.133110 0.528 0 0 -1\
                        4.89346 -0.477217 0     0 0 -0.097214 \
                        "cone"@[0,0,0]&[1,0,0]1
#       tilt prime focus plane to 45.7d (prime focus box orientation):
rayAddSurface "prime_plane" 0 0 0 0 1                   \
                        [0,0,0]&[0,0,0.798]                     \
                        "cylinder"@[0,0,0]&[1,0,0] 1
rayAddSurface "main_mirror" +0.0083333333 1 0 0 -1              \
                        [-60,0,0]&[0,0,0]               \
                        "cylinder"@[-60,-54,0]&[1,0,0] 50
rayAddSurface "prime_plane" 0 0 0 0 1                   \
                        [0,0,0]&[0,0,0]                 \
                        "cylinder"@[0,0,0]&[1,0,0] 1
#       Generate spherical waves at Gregorian feedhorn(s):
#       greg_focus=(F1-2e*cos(beta),F1+2e*sin(beta),0),
#               cone tilted by (alpha-beta)=(17.89878d-5.56996d)=12.32882d
#               cone half-angle=14.993d=0.261677r
#       feed taper is -13db at 15d=0.26rad; sphere_wave_origins in 26.67_mm grid
# gx=-10.94806m, gy=  1.06767m, delta=  0.02667m, (alpha-beta)=12.32882d
SetName "1L/R_-26.67_+26.67"
rayGenerator "spherical"@[-10.94237,  1.04162,  0.02667]&[+0.976938,+0.213522,0]0.261677 \
                0.02667,0,3,4   0.261677,-13.0  2,13 "bundle"
SetName "2L/R_-26.67_-26.67"
rayGenerator "spherical"@[-10.94237,  1.04162, -0.02667]&[+0.976938,+0.213522,0]0.261677 \
                0.02667,0,3,4   0.261677,-13.0  2,13 "bundle"
SetName "3L/R_+26.67_+26.67"
rayGenerator "spherical"@[-10.95376,  1.09373,  0.02667]&[+0.976938,+0.213522,0]0.261677 \
                0.02667,0,3,4   0.261677,-13.0  2,13 "bundle"
SetName "4L/R_+26.67_-26.67"
rayGenerator "spherical"@[-10.95376,  1.09373, -0.02667]&[+0.976938,+0.213522,0]0.261677 \
                0.02667,0,3,4   0.261677,-13.0  2,13 "bundle"
SetName "On_Axis_reference"
rayGenerator "spherical"@[-10.94806,  1.06767,  0.00000]&[+0.976938,+0.213522,0]0.261677 \
                0.02667,0,3,4   0.261677,-13.0  2,13 "bundle"
rayTrace
rayGetPlanes [-54,0]50 "silent"
rayPrtPlanes
rayPltPS [10,10] 0.10 [-10.9481,1.0677,0] orthographic rayTestCase2a.ps
Quit
```

Table 5: Input file for Planes example

```
# Testcase#2: spherical waves from proposed 40-50_GHz feedhorns through GBT
# D.Wells, NRAO-CV, 1995-07-28,08-29.
[GNU GPL copyright notice omitted]
# Command=<Digits 4 0.00001>
# Command=<System GBT_Gregorian_at_44d>
#       unit is meters. origin at prime focus. cone from feedhorn to right.
#       eps=0.528, e=5.5m, a=e/eps=10.4167m, b=sqrt(a^2-e^2)=8.8463m
#       r0=b^2/a=7.5126m, c=1/r0=0.133110, d=a-e=4.91667m
#       alpha=17.89878d=0.312393r, beta=5.56996d=0.097214r
#       vertex=(F1+d*cos(beta),F1-d*sin(beta),0)&(0,0,-beta)
#       the_big_mirror: fp=60.0m, r0=120.0m, c=1/r0=0.0083333
# Command=<rayAddSurface subreflector -0.133110 0.528 0 0 -1 4.89346 -0.4772|
#         |17 0 0 0 -0.097214 cone 0 0 0 1 0 0 1>
#       tilt prime focus plane to 45.7d (prime focus box orientation):
# Command=<rayAddSurface prime_plane 0 0 0 0 1 0 0 0 0 0 0.798 cylinder 0 0 |
#         |0 1 0 0 1>
# Command=<rayAddSurface main_mirror +0.0083333333 1 0 0 -1 -60 0 0 0 0 0 cy|
#         |linder -60 -54 0 1 0 0 50>
# Command=<rayAddSurface prime_plane 0 0 0 0 1 0 0 0 0 0 0 cylinder 0 0 0 1 |
#         |0 0 1>
#       Generate spherical waves at Gregorian feedhorn(s):
#       greg_focus=(F1-2e*cos(beta),F1+2e*sin(beta),0),
#               cone tilted by (alpha-beta)=(17.89878d-5.56996d)=12.32882d
#               cone half-angle=14.993d=0.261677r
#       feed taper is -13db at 15d=0.26rad; sphere_wave_origins in 26.67_mm grid
# gx=-10.94806m, gy=  1.06767m, delta=  0.02667m, (alpha-beta)=12.32882d
# Command=<SetName 1L/R_-26.67_+26.67>
# Command=<rayGenerator spherical -10.94237 1.04162 0.02667 +0.976938 +0.213|
#         |522 0 0.261677 0.02667 0 3 4 0.261677 -13.0 2 13 bundle>
# Command=<SetName 2L/R_-26.67_-26.67>
# Command=<rayGenerator spherical -10.94237 1.04162 -0.02667 +0.976938 +0.21|
#         |3522 0 0.261677 0.02667 0 3 4 0.261677 -13.0 2 13 bundle>
# Command=<SetName 3L/R_+26.67_+26.67>
# Command=<rayGenerator spherical -10.95376 1.09373 0.02667 +0.976938 +0.213|
#         |522 0 0.261677 0.02667 0 3 4 0.261677 -13.0 2 13 bundle>
# Command=<SetName 4L/R_+26.67_-26.67>
# Command=<rayGenerator spherical -10.95376 1.09373 -0.02667 +0.976938 +0.21|
#         |3522 0 0.261677 0.02667 0 3 4 0.261677 -13.0 2 13 bundle>
# Command=<SetName On_Axis_reference>
# Command=<rayGenerator spherical -10.94806 1.06767 0.00000 +0.976938 +0.213|
#         |522 0 0.261677 0.02667 0 3 4 0.261677 -13.0 2 13 bundle>
# Command=<rayTrace>
# Command=<rayGetPlanes -54 0 50 silent>
# Command=<rayPrtPlanes>

               -=-< spherical, r=0.26167 >-=-
                              A_0000 A_2000 A_4000 A_1010 A_1011 A_3010 A_3011 A_2020 A_2021
  i      bundle_name      n  Zero_Pt  Defoc Sph_Ab   Tilt  <sin>   Coma  <sin> Astigm  <sin>
 -- -------------------- --  -------  -----  -----  -----  -----  -----  -----  -----  -----
  1   1L/R_-26.67_+26.67 49 140.8334               -.1381  .1388
  2   2L/R_-26.67_-26.67 49 140.8334               -.1381 -.1388
  3   3L/R_+26.67_+26.67 49 140.8334                .1395  .1388
  4   4L/R_+26.67_-26.67 49 140.8334                .1395 -.1388
  5    On_Axis_reference 49 140.8333

# Command=<rayPltPS 10 10 0.10 -10.9481 1.0677 0 orthographic rayTestCase2a.|
#         |ps>
# Command=<Quit>
```

Table 6: `rayMain` output for Planes example

A Appendices

A.1 Include file ray.h

The function prototypes which are at the end of ray.h have been suppressed in the following listing:

```
/* ray.h -- Include file for the 'ray' (ray-tracing) package.
   D.Wells, NRAO-CV
   97-05-03 many name changes
   97-05-30 removed changes to rayGenerator() */
```

[GNU GPL copyright notice omitted]

```
#ifndef RAY_H
#define RAY_H

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mathVectorMatrix.h"

#define TRUE  1
#define FALSE 0
#define NAMEMAX 21
#define FORMMAX 100
#define AMNIMAX 9
enum VignetteType {VIGN_CYLINDER, VIGN_CONE};
enum ColorType    {COLOR_BUNDLE,  COLOR_RAY};
enum PlotType     {ORTHOGRAPHIC, ORTHOGRAPHIC_X, PERSPECTIVE};

struct Surface {
    char name[NAMEMAX]; /* user-supplied descriptive string */
    double c_1;         /* curvature c=1/r; positive if center of curvature is
                           to right of surface. c_1 for following surface */
    double eps;         /* numerical eccentricity of the conic section. */
    double a_2;         /* coefficients of radially-symmetric */
    double a_4;         /*       aspheric deformation series. */
    double mu_1;        /* $mu_1\equiv N/N_1$, N_1 is index to right
                           of following surface */
    double S[3];        /* absolute vector to vertex of this surface. */
    double E[3];        /* tilt of next vertex, 3 Euler Angles (see
                           "General Ray-Tracing Procedure",
                           G.H.Spencer and M.V.R.K.Murty, JOSA 52,
                           pp.672-678, (June 1962)). */
    enum VignetteType vign_type;
    double vign_origin[3];
    double vign_vector[3];
    double vign_radius;
};

struct Ray {
    double T[3];        /* $T \equiv (x, y, z)$ is the vector from the
                           vertex of the first surface to the point of
                           incidence of the ray on this surface.          */
    double Q[3];        /* $Q \equiv (X, Y, Z)$ is the unit vector along
```

```
                        the ray to the right of the first surface. X,Y,Z
                        are direction cosines; ray path is $T + l * Q$ */
    double direction;   /* experimental, +1->+X, -1-->-X                */
    double Length;      /* cumulative path length as ray traverses system */
    double Intensity;   /* same as power for radio                      */
    int ColorCode;      /* color code to use when plotting this ray     */
};

struct Segment {
    double T1[3];       /* x,y,z position of starting point of ray */
    double T2[3];       /* x,y,z position of   ending point of ray */
    int ColorCode;      /* color code to use when plotting this ray */
};

struct Focus {
    char name[NAMEMAX]; /* name_string for the ray bundle */
    int n;              /* number of rays in the bundle   */
    double xyz[3];      /* position of focus              */
    double xyzs[3];     /* std_err of focus position      */
    double lc;          /* mean pathlength to focus       */
    double ls;          /* spherical wave rms wrt focus   */
};

struct Plane {
    char name[NAMEMAX];    /* name_string for the ray bundle   */
    int n;                 /* number of rays in the bundle     */
    int nu;                /* number of Seidel terms           */
    double Amni[AMNIMAX];  /* Seidel terms for pathlengths     */
    double Asig[AMNIMAX];  /* Sigmas for Amni[AMNIMAX]         */
    int     mni[AMNIMAX];  /* indicies of Seidel terms         */
    double sig;            /* rms of (pathlength-sum(Amni[]))  */
    double yz_radius;      /* Zernike radius in rayGetPlanes() */
};

#ifdef SEIDEL
struct Seidel_item {
    int mni;               /* A_mni index                     */
    char *name;            /* names of Seidel aberrrations    */
};
static struct Seidel_item Seidel_list[] = {
    {0000, "Zero_Pt"},
    {2000, "Defoc"},
    {4000, "Sph_Ab"},
    {1010, "Tilt"},
    {1011, " <sin>"},
    {3010, "Coma"},
    {3011, " <sin>"},
    {2020, "Astigmatism"},
    {2021, " <sin>"}
    };
const static int Seidel_n = sizeof(Seidel_list) / sizeof(struct Seidel_item);
#endif

struct Node {
    void *item;         /* pointer to an "item" struct */
    struct Node *next;  /* pointer to next item in list */
};
```

A.2 List structures – initializing, inserting and deleting

The `ray` package deals with *lists* of objects, not with fixed-dimension arrays.[11] A simple list processing package is sufficient. This one is based on algorithms given by Sedgewick [Sed90, p.20].

The lists are composed of "nodes". Each `Node structure` consists of a pointer to another node plus a pointer to some data object (see definition in `ray.h`, Appendix A.1). A list consists of a chain of nodes, with the head node containing a pointer to the second node, which points to the third node, ..., an so forth until the tail node, whose "`next`" pointer points to itself. The head node always exists, and the data item associated with it is a descriptive character string supplied when the list is initialized by calling `listInitialize()`. The tail node does not point to a data object. The minimum (empty) list is a head node with its descriptive string and a tail node. The six function prototypes shown below are in `ray.h`.

```
struct Node *listInitialize(char *name)        /* descriptive string for list */
```

A pointer to the head of the new list is returned.

```
struct Node *listInsertAfter (                 /* returns ptr to new node */
                              void *newitem,   /* ptr to new item struct */
                              struct Node *t)  /* ptr to node in a list  */
```

This function will `malloc()` a new instance of `Node`, set the item pointer of the new node to the pointer `newitem`, copy the pointer to the following node from the prior node into the new node and change the pointer of the prior node to point to the new node.

```
struct Node *listAppend (                      /* returns ptr to new node */
                         void *newitem,        /* ptr to new item struct  */
                         struct Node *list)    /* ptr to existing list    */
```

This function reads the list until it finds the node just before the tail node, then it calls `listInsertAfter()`. The terseness of the idioms for searching and manipulating these list structures is exemplified by this slightly simplified version of the function:

```
struct Node *listAppend (                      /* returns ptr to new node */
                         void *newitem,        /* ptr to new item struct  */
                         struct Node *list) {  /* ptr to existing list    */
  struct Node *t;
  t = list;
  while (t->next->next != t->next) { t = t->next; }
  return(listInsertAfter (newitem, t));
}
```

[11] Use of list techniques in ray tracing is not merely a question of programming convenience and notational elegance — it is very nearly a matter of necessity, because the number of output rays produced for a given number of input rays is not predictable *a priori*. First, when rays are vignetted by one of the optical elements of a system, they need not be traced through the remaining elements, and they need not (probably should not) even appear in the output list of rays. (Of course, in radio telescopes "vignetting" is "spillover", and vignetted radio rays are interesting in their own right.) Secondly, when a ray is traced through a refractive surface, it becomes *two* rays, one reflected and one refracted. Each of these rays can then encounter another refractive surface (*e.g.*, by an internal reflection) at which it too can become two rays. In general, each individual input ray passing through a refractive system produces an infinite cascade of output rays. In practice, many of the rays in the cascade will be vignetted by the optical elements and others can be discarded when their intensity falls below some specified sensitivity level. The cascading of refracted and reflected rays has actually been implemented and used for tracing special radomes, in which rays can even be reflected from the inside of one part of the radome and then refracted through an entirely different part (in [PPL95] these cases are called "flash lobes"). The `ray` package does not currently support tracing of rays reflected from refractive surfaces because the author only needs to trace reflective surfaces for the GBT analysis.

```
int listDeleteNext(struct Node *t)              /* ptr to node in a list */
```

This function does `free()` on the data object to which the node points, cuts the node out of the chain of pointers and then does `free()` on the `Node` structure itself.

```
void listDeleteList (struct Node *list)         /* list of items */
```

This function walks through the list doing `listDeleteNext()` on each node, then does `free()` on the tail node, the descriptive character string and the head node.

```
void listDeleteListList (struct Node *list)     /* list of lists of items */
```

The `item` pointer of a `Node` can contain a pointer to the head node of another list. This list-of-lists concept is very powerful, and is heavily used in the `ray` package. This function walks through the list of lists and calls `listDeleteList()` and `listDeleteNext()` to `free()` all storage associated with each node. It finishes the operation by deleting the tail node, descriptive string and head node of the list of lists.

A.3 Features expected to be added in future releases of `ray`

- `rayTrace`
  - Implement vignetting functions. Is ray intercept point *inside* the CONE or CYLINDER? If test fails, append ray to bundle-set of vignetted rays, marked by the vignetting surface. This will support spillover calculations in radio telescopes.
  - Implement polarization of rays, including refraction/reflection calculations at surfaces. This will support cross-polarization calculations in radio telescopes.
  - Change to Euler angle rotations for surfaces
- `rayGenerator`
  - Implement "generalized spiral points" [SK97] for ray origins on spherical waves, and probably for plane waves too (use equal-area mapping from sphere to plane). This equal-area distribution will facilitate numerical integrations, such as for cross-polarization calculations in radio telescopes. The quasi-hexagonal pattern of the generalized spiral points will fit circular aperture stops better than does the square pattern generated by the present algorithm.
- `rayPltSystem`
  - Implement plotting of vignetting-bounded surfaces, with axes.
- `rayPltPS`
  - Implement "`Perspective`" mode; may need to add `from_point` and `up_vector` arguments.

References

[AS52] William A. Allen and John R. Snyder. Ray tracing through uncentered and aspheric surfaces. *JOSA*, 42(4):243–249, April 1952. "The generalized equations.. developed herein reduce to those used in.. [Fed51] for the special case of a centered optical system".

[BW59] Max Born and Emil Wolf. *Principles of Optics*. Pergamom Press, London, first edition, 1959. "The circle polynomials of Zernike" is the title of Section 9.2.1; in Section 9.2.2 these polynomials are used to construct an expansion of the aberration function. LOC=QC355.B63.

[Fed51] Donald P. Feder. Optical calculations with automatic computing machinery. *JOSA*, 41(9):630–635, September 1951.

[Nel90] Jerry Nelson. Panel surfaces for the Green Bank Telescope. GBT Memo 31, National Radio Astronomy Observatory, January 1990. This memo uses a set of polynomials analogous to the Zernike polynomials [BW59] to analyze the GBT panels, and concludes that the panels can be approximated by quadratic surfaces. This memo is also Keck Observatory Technical Note 291.

[Nor90a] Roger D. Norrod. Increase in distance between large and small subreflectors. GBT Memo 40, National Radio Astronomy Observatory, March 1990. This memo quotes the formulæ for the α, β and θ_H angles of the subreflector and Gregorian feedhorn geometry. It discusses the cross-polarization justification for β. Table 1 gives the official values for the parameters of the GBT geometry.

[Nor90b] Roger D. Norrod. Spillover blockage in the GBT design. GBT Memo 41, National Radio Astronomy Observatory, March 1990. This memo discusses feed spillover, spillover efficiency (feedhorn power patterns) and shields that could be deployed to reduce scattering.

[PPL95] B. Philips, E. A. Parker, and R. J. Langley. Ray tracing analysis of the transmission performance of curved FSS. *IEE Proc. Microw. Antennas Propag.*, 142(3):193–200, June 1995. "Finite frequency selective surfaces (FSS) embedded in a curved dielectric are analysed using the ray-tracing technique.. [by means of] complex reflection and transmission coefficients subjected to multiple interactions with rays internally reflected within the dielectric..".

[RSCWL81] Y. Rahmat-Samii, P. Cramer, K. Woo, and S. W. Lee. Realizable feed-element patterns for multibeam reflector antenna analysis. *IEEE Trans. Antennas Propagat.*, AP-29(6):961–963, 1981.

[Sed90] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, 1990. LOC=QA76.73.C15S43, ISBN=0-201-51425-7.

[SK97] E. B. Saff and A. B. J. Kuijlaars. Distributing many points on a sphere. *The Mathematical Intelligencer*, 19(1):5–11, January 1997. This paper describes the 'generalized spiral points', an algorithm for producing an approximately equal-area distribution of N points on the sphere.

[SM62] G. H. Spencer and M. V. R. K. Murty. General ray tracing procedure. *JOSA*, 52(6):672–678, June 1962. Discusses tracing toric surfaces.

[Sri91] S. Srikanth. Concept of quasi-optical beam-switching scheme on the GBT. GBT Memo 67, NRAO, September 1991.

[Tul69] R. G. Tull. Planetary spectroscopy with the 107-inch telescope. *Sky & Telescope*, 38(3):156, September 1969.

[Tul72] R. G. Tull. The coude spectrograph and echelle scanner of the 2.7-m telescope at McDonald Observatory. In *ESO/CERN Conf. on Auxiliary Instrumentation for Large Telescopes*, page 259. European Southern Observatory, Geneva (now in Munich), 1972.

[Wel98a] Don Wells. GBT Gregorian focus tracking in C. GBT Memo *[tbd]*, NRAO, *[in preparation]* 1998. The GBT Gregorian subreflector, an off-axis portion of an ellipsoid, and the feedroom with the feedhorn move relative to the prime focal point of the main paraboloidal mirror. The subreflector must be maneuvered relative to the prime focal point and the feedhorn to maintain nearly stigmatic imaging (maximum gain, minimum sidelobes). An algorithm is described which computes the required actuator motions as a function of elevation, with *no* focus error, spherical abberation or coma, and with only 0.5 mm of residual astigmatism. The focus tracking algorithm is expressed in C.

[Wel98b] Don Wells. Imaging properties of the GBT subreflector. GBT Memo *[tbd]*, NRAO, *[in preparation]* 1998. The.. subreflector.. images points in the neighborhood of its first focus onto points in the neighborhood of its second focus.. nearly-stigmatic imaging.. can be obtained for a variety of tilts and displacements.. cases.. computed by ray tracing.. minimum phase error.. if.. ellipsoid is tilted slightly.. results.. fitted with polynomials.. expressed in C.

[WK95a] Don Wells and Lee King. GBT Best-Fitting Paraboloid [BFP] in C. GBT Memo 131, NRAO, June 1995. Abstract: The gravitational displacements of the GBT actuators have been fitted with a paraboloid. The parameters of the paraboloid for various elevations have been fitted with polynomials and expressed as C code which computes the parameters of this best-fitting-paraboloid [BFP] as a function of elevation. The BFP will be used by the control software modules for the pointing, focus-tracking and active-surface subsystems of the GBT. We give a description of this C-code version of the BFP and two examples of its application to practical problems. We also give a function in C which fetches node data from the structural model and transforms it to a coordinate system tied to the BFP. The predicted gravitational term of the GBT's traditional pointing model and the predicted prime focus focus-tracking formula of the GBT are given. See `ftp://fits.cv.nrao.edu/pub/gbt_dwells_doc.tar.gz` for the current revision of this memo (131.2 as of 1997-06-23).

[WK95b] Don Wells and Lee King. The GBT Tipping-Structure Model in C. GBT Memo 124, NRAO, March 1995. Abstract: The finite element model of the GBT tipping structure has been translated into executable code expressed in the C language, so that it can be used by the control software modules for the pointing, focus-tracking, quadrant detector, active-surface and laser-rangefinder subsystems of the GBT. We give a description of this C-code version of the tipping structure model and two examples of its application to practical problems. See `ftp://fits.cv.nrao.edu/pub/gbt_dwells_doc.tar.gz` for the current revision of this memo (124.3 as of 1997-06-23).

[WS95] E. Wollack and S. Srikanth. Design considerations for a GBT Q-band array receiver. GBT Scientific Working Group Memo, NRAO, April 1995. See `http://info.gb.nrao.edu/GBT/swg/gbtmme.ps`.