

Appendix A Example Lenses from Solution Space Results

A.1. Introduction

Four specific lenses derived from the solution space for zero oblique astigmatism and zero mean oblique error are shown. The lens has a +10 diopter power, a base index of $n=1.525$, an f -number of 10, and the stop and exit pupil are located 25 mm from the back surface. The output from Code V is given in each case. The lens is also optimized to show how much the radii change when compensating for non-zero lens thickness and non- paraxial rays. Each system is illustrated by a preceding figure (Figure A.1, Figure A.2, Figure A.3, Figure A.4, Figure A.5, Figure A.6, Figure A.7, and Figure A.8).

A.2. Zero Oblique Astigmatism for Power Radial Gradient

A.2.1. Compute Astigmatism using Radii from Solution Space

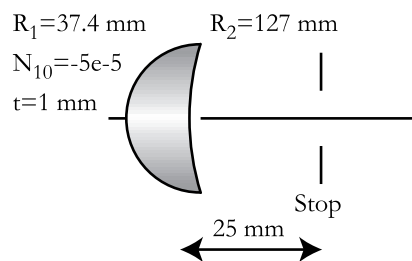


Figure A.1. Schematic of lens specified by Code V code in example A.2.1.

$$N_{10}=-5e-5$$

```

sur sa
RDY      THI      GLA
OBJ:     INFINITY INFINITY
> 1:     37.38300 1.000000 'grin_ex'
2:     126.69500 25.000000
S:       INFINITY 73.702323
IMG:     INFINITY 0.000000

INFINITE CONJUGATES
EFL      99.6230
BFL      73.7023
FFL      -99.8904
FNO      10.0000
IMG DIS  73.7023
OAL      26.0000
PARAXIAL IMAGE
HT       8.7159
ANG      5.0000
ENTRANCE PUPIL
DIA      9.9623
THI      34.7694
EXIT PUPIL
DIA      7.3702
THI      0.0000

SA      TCO      TAS      SAS      PTB      DST      AX      LAT      PTZ
1 -0.026602 -0.003663 -0.017658 -0.017546 -0.017490 -0.000805 0.000000 0.000000 -0.009209
2 -0.000119 0.003060 -0.021129 -0.003603 0.005161 0.030955 0.000000 0.000000 0.002717
0.023312 -0.042453 0.025446 0.008427 -0.000082 -0.005018 GRADIENT CONTRIBUTIONS
S 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
Sum
-0.003409 -0.043057 -0.013341 -0.012721 -0.012411 0.025132 0.000000 0.000000 -0.006535
Astigmatism = (SAS)-(PTB) = -0.0003101153875
    
```

A.2.2. Minimize Astigmatism by Optimizing Surface Curvature

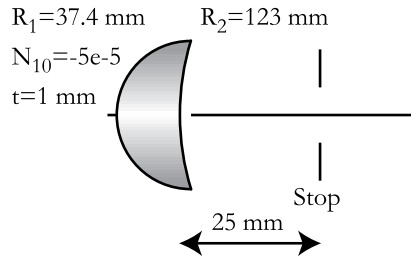


Figure A.2. Schematic of lens specified by Code V code in example A.2.2.

```

aut
@bl == ((THI S2) + (THI S3))
@bl = 1000.0/(10)
@sig3 == ((SAS) - (PTB))
@sig3 = 0
go

RDY      THI      RMD      GLA      CCY      THC      GLC
> OBJ:     INFINITY INFINITY 100 100
1:     37.37873 1.000000 'grin_ex' 0 100
2:     122.75196 25.000000 0 100
S:       INFINITY 75.000000 100 PIM
IMG:     INFINITY 0.000000 100 100

SA      TCO      TAS      SAS      PTB      DST      AX      LAT      PTZ
1 -0.028039 -0.004023 -0.018147 -0.018019 -0.017955 -0.000862 0.000000 0.000000 -0.009210
2 -0.000083 0.002516 -0.019990 -0.003018 0.005467 0.030538 0.000000 0.000000 0.002805
0.024249 -0.043406 0.025566 0.008466 -0.000084 -0.004953 GRADIENT CONTRIBUTIONS
S 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
    
```

```
Sum
-0.003872 -0.044913 -0.012571 -0.012571 -0.012571 0.024723 0.000000 0.000000 -0.006449
Astigmatism = (SAS)-(PTB) = -0.867361738e-17
```

A.3. Zero Oblique Astigmatism for Shallow Radial Gradient

A.3.1. Compute Astigmatism using Radii from Solution Space

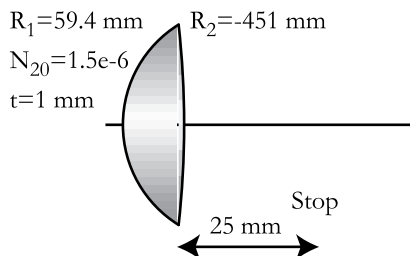


Figure A.3. Schematic of lens specified by Code V code in example A.3.1.

$N_{20} = 1.5e-6$

```
RDY      THI      RMD      GLA
OBJ:     INFINITY INFINITY
1:       59.41620 1.000000 'grin_ex'
> 2:     -451.02100 25.000000
S:       INFINITY 74.487694
IMG:     INFINITY 0.000000

INFINITE CONJUGATES
EFL      100.0675
BFL      74.4877
FFL      -99.9911
FNO      10.0000
IMG DIS  74.4877
OAL      26.0000
PARAXIAL IMAGE
HT       8.7548
ANG      5.0000
ENTRANCE PUPIL
DIA      10.0067
THI      34.4405
EXIT PUPIL
DIA      7.4488
THI      0.0000
Lens #1 (8)

SA      TCO      TAS      SAS      PTB      DST      AX      LAT      PTZ
1 -0.006745 -0.008837 -0.014961 -0.012389 -0.011102 -0.005410 0.000000 0.000000 -0.005794
2 -0.005742 0.035012 -0.072627 -0.025184 -0.001463 0.051189 0.000000 0.000000 -0.000763
0.074336 -0.133023 0.079350 0.026450 0.000000 -0.015778 GRADIENT CONTRIBUTIONS
S 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
Sum
0.061850 -0.106848 -0.008239 -0.011123 -0.012565 0.030001 0.000000 0.000000 -0.006557
Astigmatism = (SAS)-(PTB) = 0.0014419262174
```

A.3.2. Minimize Astigmatism by Optimizing Surface Curvature

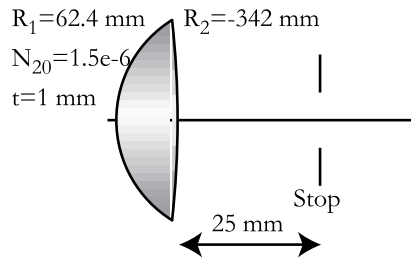


Figure A.4. Schematic of lens specified by Code V code in example A.3.2.

```

RDY      THI      RMD      GLA      CCY      THC      GLC
OBJ:     INFINITY INFINITY 100 100
1:       62.36161 1.000000 'grin_ex' 0 100
> 2:     -342.10271 25.000000 0 100
S:       INFINITY 75.000000 100 PIM
IMG:     INFINITY 0.000000 100 100

SA      TCO      TAS      SAS      PTB      DST      AX      LAT      PTZ
1 -0.005948 -0.008693 -0.014917 -0.012093 -0.010681 -0.005892 0.000000 0.000000 -0.005520
2 -0.006638 0.038795 -0.077520 -0.027138 -0.001947 0.052865 0.000000 0.000000 -0.001006
0.075837 -0.134747 0.079808 0.026603 0.000000 -0.015757 GRADIENT CONTRIBUTIONS
S 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
Sum
0.063251 -0.104645 -0.012628 -0.012628 -0.012628 0.031216 0.000000 0.000000 -0.006527
Astigmatism = (SAS)-(PTB) = -0.69388939e-17
    
```

A.4. Zero Mean Oblique Error for Shallow Radial Gradient

A.4.1. Compute Oblique Error using Radii from Solution Space

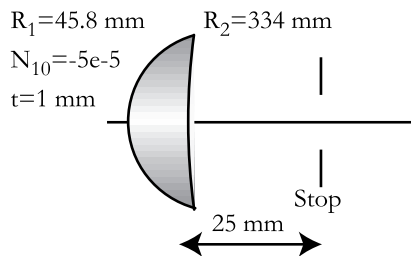


Figure A.5. Schematic of lens specified by Code V code in example A.4.1.

$N_{10} = -5e-5$

```

RDY      THI      RMD      GLA
OBJ:     INFINITY INFINITY
1:       45.77270 1.000000 'grin_ex'
> 2:     334.45700 25.000000
S:       INFINITY 74.130711
IMG:     INFINITY 0.000000
    
```

```

INFINITE CONJUGATES
EFL      99.8852
BFL      74.1307
FFL     -99.9848
FNO      10.0000
IMG DIS   74.1307
OAL      26.0000
PARAXIAL IMAGE
HT       8.7388
ANG      5.0000
ENTRANCE PUPIL
DIA      9.9885
THI     34.6026
EXIT PUPIL
DIA      7.4131
THI      0.0000
Lens #1 (8)

SA      TCO      TAS      SAS      PTB      DST      AX      LAT      PTZ
1 -0.014645 -0.008597 -0.016041 -0.014920 -0.014359 -0.002919 0.000000 0.000000 -0.007521
2 -0.001543 0.014353 -0.042534 -0.012868 0.001965 0.039895 0.000000 0.000000 0.001029
0.023398 -0.042194 0.025037 0.008291 -0.000082 -0.004886 GRADIENT CONTRIBUTIONS
S 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
Sum
0.007210 -0.036438 -0.033539 -0.019497 -0.012476 0.032090 0.000000 0.000000 -0.006535

(-(HCY SI)*(HCY SI)*(UMY SI)/((THI S2)-(EFL))) = -0.0509894044811
((SAS) + (TAS)) = -0.0530357028427
    
```

Power Error = -0.0530357028427

A.4.2. Minimize Error by Optimizing Surface Curvature

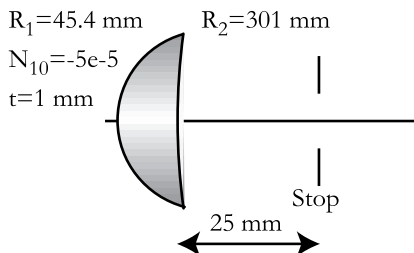


Figure A.6. Schematic of lens specified by Code V code in example A.4.2.

```

aut
@bl == ((THI S2) + (THI S3))
@bl = 1000.0/(10)
@curv1 == ((SAS) + (TAS))
@curv2 == (-(HCY SI)*(HCY SI)*(UMY SI)/((THI S2)-(EFL)))
@target == @curv2 - @curv1
@target = 0
dsp @curv1
dsp @curv2
dsp @target
go

RDY      THI      RMD      GLA      CCY      THC      GLC
OBJ:     INFINITY INFINITY 100 100
1:       45.43267 1.000000 'grin_ex' 0 100
> 2:     300.93568 25.000000 0 100
S:       INFINITY 75.000000 100 PIM
IMG:     INFINITY 0.000000 100 100

SA      TCO      TAS      SAS      PTB      DST      AX      LAT      PTZ
1 -0.015512 -0.008829 -0.016398 -0.015281 -0.014723 -0.002899 0.000000 0.000000 -0.007577
2 -0.001379 0.013358 -0.040904 -0.012153 0.002223 0.039234 0.000000 0.000000 0.001144
    
```

```

0.024028 -0.042839 0.025127 0.008320 -0.000084 -0.004846 GRADIENT CONTRIBUTIONS
S 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
Sum
0.007137 -0.038309 -0.032175 -0.019114 -0.012584 0.031489 0.000000 0.000000 -0.006476

((SAS) + (TAS)) = -0.0512896453759
(-(HCY SI)*(HCY SI)*(UMY SI)/((THI S2)-(EFL))) = -0.0512896453759
Power Error = -0.69388939e-17
    
```

A.5. Zero Mean Oblique Error for Shallow Radial Gradient

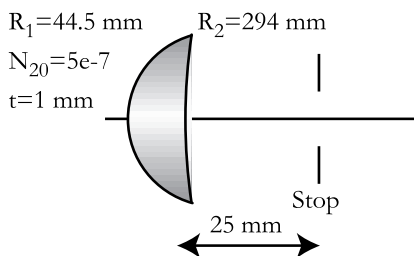


Figure A.7. Schematic of lens specified by Code V code in example A.5.1.

$N_{20} = 5e-7$

A.5.1. Compute Oblique Error using Radii from Solution Space

```

RDY      THI      RMD      GLA
OBJ:     INFINITY INFINITY
1:       44.54980 1.000000 'grin_ex'
> 2:     294.18900 25.000000
S:       INFINITY 74.090615
IMG:     INFINITY 0.000000

INFINITE CONJUGATES
EFL      99.8623
BFL      74.0906
FFL      -99.9792
FNO      10.0000
IMG DIS  74.0906
OAL      26.0000
PARAXIAL IMAGE
HT       8.7368
ANG      5.0000
ENTRANCE PUPIL
DIA      9.9862
THI      34.6193
EXIT PUPIL
DIA      7.4091
THI      0.0000
Lens #1 (8)

SA      TCO      TAS      SAS      PTB      DST      AX      LAT      PTZ
1 -0.015870 -0.008284 -0.016188 -0.015227 -0.014747 -0.002650 0.000000 0.000000 -0.007728
2 -0.001345 0.013134 -0.040507 -0.012014 0.002233 0.039093 0.000000 0.000000 0.001170
0.024481 -0.044129 0.026516 0.008839 0.000000 -0.005311 GRADIENT CONTRIBUTIONS
S 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
Sum
0.007266 -0.039278 -0.030180 -0.018402 -0.012513 0.031133 0.000000 0.000000 -0.006557

(-(HCY SI)*(HCY SI)*(UMY SI)/((THI S2)-(EFL))) = -0.0509816115916
((SAS) + (TAS)) = -0.0485818206077
    
```

Power Error = -0.0023997909839

A.5.2. Minimize Error by Optimizing Surface Curvature

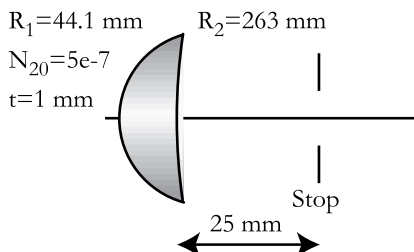


Figure A.8. Schematic of lens specified by Code V code in example A.5.2.

```

RDY      THI      RMD      GLA      CCY      THC      GLC
OBJ:     INFINITY INFINITY 100 100
1:       44.11369 1.000000 'grin_ex' 0 100
> 2:     263.20089 25.000000 0 100
S:       INFINITY 75.000000 100      PIM
IMG:     INFINITY 0.000000 100 100

SA      TCO      TAS      SAS      PTB      DST      AX      LAT      PTZ
1 -0.016959 -0.008473 -0.016580 -0.015640 -0.015169 -0.002605 0.000000 0.000000 -0.007804
2 -0.001164 0.011981 -0.038574 -0.011163 0.002542 0.038312 0.000000 0.000000 0.001308
0.021751 -0.038736 0.022996 0.007665 0.000000 -0.004551 GRADIENT CONTRIBUTIONS
S 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
Sum
0.003629 -0.035229 -0.032159 -0.019138 -0.012627 0.031156 0.000000 0.000000 -0.006496

((SAS) + (TAS)) = -0.0512966567202
(-(HCY SI)*(HCY SI)*(UMY SI)/((THI S2)-(EFL))) = -0.0512963601321
    
```

Power Error = 0.29658803746e-6

Appendix B Indicial Derivatives of B-Spline Basis Functions

B.1. Introduction

These are the second derivatives of the transformed B-spline basis functions, as described in section 5.4.6.

B.2. Variable Substitutions

$$\begin{aligned}\bar{x} &= x - (x_o + l\Delta x) & \delta x &= \bar{x} - \Delta x \\ \bar{y} &= y - (y_o + m\Delta y) & \delta y &= \bar{y} - \Delta y\end{aligned}\tag{B.1}$$

$$\Gamma = \frac{1}{6(x^2 + y^2)\Delta x^3 \Delta y^3}$$

B.3. “Tangential” Direction Second Derivatives

$$\frac{d^2 \bar{b}_{11}}{dy^2} = \Gamma \delta x \delta y \left[\begin{array}{l} y^2 \delta x^2 + 3xy \delta x \Delta y + x^2 \Delta y^2 \\ + \bar{x}\bar{y}(3y \delta x - 2x \Delta y + \bar{x}\bar{y}) \end{array} \right]\tag{B.2}$$

$$\frac{d^2 \bar{b}_{12}}{dy^2} = \Gamma \delta y \left[\begin{array}{l} -3y^2 \bar{x}^3 + 3y\bar{x}^2 (y\Delta x + 3x\Delta y - 3\bar{x}\bar{y}) \\ + 3x\delta y\bar{x} (4y\Delta x + x\Delta y - \bar{x}\bar{y}) \\ + 2\Delta x (-2y^2 \Delta x^2 + x^2 \Delta y^2 + x^2 \bar{y}(-2\Delta y + \bar{y})) \end{array} \right]\tag{B.3}$$

$$\frac{d^2 \bar{b}_{13}}{dy^2} = -\Gamma \delta y \left[\begin{array}{l} -3y^2 \bar{x}^3 + 3y\bar{x}^2 (y\Delta x + 3x\Delta y - 3\bar{x}\bar{y}) \\ + 3\bar{x} \left(\begin{array}{l} y^2 \Delta x^2 - 2xy \Delta x \Delta y - x^2 \Delta y^2 \\ + \bar{x}\bar{y} (2y\Delta x + 2x\Delta y - \bar{x}\bar{y}) \end{array} \right) \\ + \Delta x \left(\begin{array}{l} y^2 \Delta x^2 - 3xy \Delta x \Delta y + x^2 \Delta y^2 \\ + \bar{x}\bar{y} (3y\Delta x - 2x\Delta y + \bar{x}\bar{y}) \end{array} \right) \end{array} \right]\tag{B.4}$$

$$\frac{d^2 \bar{b}_{14}}{dy^2} = -\Gamma \bar{x} \delta y \left[x^2 \delta y^2 + y^2 \bar{x}^2 + 3xy\bar{x}\delta y \right] \quad (\text{B.5})$$

$$\frac{d^2 \bar{b}_{21}}{dy^2} = \Gamma \delta x \left[\begin{array}{l} 2 y^2 \delta x^2 \Delta y - 4 x^2 \Delta y^3 \\ -3 \bar{y} (y \delta x (y \delta x - 4 x \Delta y) + \bar{x} \bar{y} (3 y \delta x - 2 x \Delta y + \bar{x} \bar{y})) \end{array} \right] \quad (\text{B.6})$$

$$\frac{d^2 \bar{b}_{22}}{dy^2} = \Gamma \left[\begin{array}{l} 3 y^2 \bar{x}^3 (-2 \Delta y + 3 \bar{y}) + \\ 3 y \bar{x}^2 (4 y \Delta x \Delta y + 3 \bar{y} (-2 y \Delta x - 4 x \Delta y + 3 \bar{x} \bar{y})) \\ -2 \Delta x \left(\begin{array}{l} 4 (y^2 \Delta x^2 \Delta y + x^2 \Delta y^3) + \\ 4 \bar{y} (-2 y^2 \Delta x^2 + x^2 \bar{y} (-2 \Delta y + \bar{y})) \end{array} \right) \\ + 3 x \bar{x} \left(\begin{array}{l} 4 x \Delta y^3 + \bar{y} \left(\begin{array}{l} 16 y \Delta x \Delta y + \\ 3 \bar{y} (-4 y \Delta x - 2 x \Delta y + \bar{x} \bar{y}) \end{array} \right) \end{array} \right) \end{array} \right] \quad (\text{B.7})$$

$$\frac{d^2 \bar{b}_{23}}{dy^2} = \Gamma \left[\begin{array}{l} \left(\begin{array}{l} \Delta x^2 (y \Delta x - 4 x \Delta y) \\ 3 \bar{y} \left(\begin{array}{l} \Delta x (3 y \Delta x - 8 x \Delta y) \\ + \bar{x} \left(\begin{array}{l} \Delta x (3 y \Delta x - 2 x \Delta y) \\ + 3 \bar{x} (2 y \Delta x + 2 x \Delta y - 3 y \bar{x}) \end{array} \right) \end{array} \right) \\ + 3 \bar{x} \bar{y}^2 \left(\begin{array}{l} \Delta x (3 y \Delta x - 2 x \Delta y) \\ + 3 \bar{x} (2 y \Delta x + 2 x \Delta y - 3 y \bar{x}) \end{array} \right) \\ + x^2 \bar{y}^3 (\Delta x - 3 \bar{x}) - 2 y^2 \Delta x^3 \Delta y + 4 x^2 \Delta x \Delta y^3 \\ - 6 \Delta y \bar{x} (y^2 \Delta x^2 + 2 x^2 \Delta y^2 - y^2 \delta x \bar{x}) \end{array} \right) \end{array} \right] \quad (\text{B.8})$$

$$\frac{d^2 \bar{b}_{24}}{dy^2} = \Gamma \bar{x} \left[\begin{array}{l} y^2 \bar{x}^2 (3 \bar{y} - 2 \Delta y) - 3 xy \bar{x} \bar{y} (4 \Delta y - 3 \bar{y}) \\ + x^2 (4 \Delta y^3 + 3 \bar{y}^2 (\bar{y} - 2 \Delta y)) \end{array} \right] \quad (\text{B.9})$$

$$\frac{d^2 \bar{b}_{31}}{dy^2} = \Gamma \delta x \left[\begin{array}{l} -\Delta y (y^2 \delta x^2 + 3 xy \delta x \Delta y + x^2 \Delta y^2) \\ + 3 \bar{y} \left(\begin{array}{l} y^2 \delta x^2 - 2 xy \delta x \Delta y \\ -x^2 \Delta y^2 + \bar{x} \bar{y} (3 y \delta x - x \Delta y + \bar{x} \bar{y}) \end{array} \right) \end{array} \right] \quad (\text{B.10})$$

$$\frac{d^2 \bar{b}_{32}}{dy^2} = \Gamma \left[\begin{array}{l} 4 y^2 \Delta x^3 \Delta y - 2 x^2 \Delta x \Delta y^3 + 3 y^2 \bar{x}^3 (\Delta y - 3 \bar{y}) \\ -6 \Delta x \bar{y} (2 y^2 \Delta x^2 + x^2 \Delta y^2 + x^2 \Delta y^2 - x^2 \delta \bar{y}) \\ -3 y \bar{x}^2 \left(\begin{array}{l} \Delta y (2 y \Delta x - 3 x \Delta y) \\ -6 \bar{y} (y \Delta x + x \Delta y) + 9 x \bar{y}^2 \end{array} \right) \\ +3 x \bar{x} \left(\begin{array}{l} \Delta y^2 (x \Delta y - 4 y \Delta x) \\ + \bar{y} \left(\begin{array}{l} \Delta y (3 x \Delta y - 8 y \Delta x) \\ +3 \bar{y} (4 y \Delta x + x \Delta y - x \bar{y}) \end{array} \right) \end{array} \right) \end{array} \right] \quad (\text{B.11})$$

$$\frac{d^2 \bar{b}_{33}}{dy^2} = \Gamma \left[\begin{array}{l} \Delta x \Delta y (y^2 \Delta x^2 + 3 x y \Delta x \Delta y + x^2 \Delta y^2) \\ -3 x^2 \bar{y}^3 (\Delta x - 3 \bar{x}) \\ +3 \Delta y \bar{x} \left(\begin{array}{l} y^2 \Delta x^2 + 2 x y \Delta x \Delta y - x^2 \Delta y^2 \\ + \bar{x} (y \Delta x - 3 x \Delta y - y \bar{x}) \end{array} \right) \\ -3 \bar{y} \Delta x (y^2 \Delta x^2 - 2 x y \Delta x \Delta y - x^2 \Delta y^2) \\ -3 y \bar{x} \left(\begin{array}{l} 3 y^2 \Delta x^2 - 4 x y \Delta x \Delta y + 3 x^2 \Delta y^2 \\ +3 \bar{x} (y \Delta x + 2 x \Delta y - y \bar{x}) \end{array} \right) \\ +3 x \bar{y}^2 (\Delta x (x \Delta y - 3 y \Delta x) + 3 \bar{x} (3 y \bar{x} - 2 y \Delta x - x \Delta y)) \end{array} \right] \quad (\text{B.12})$$

$$\frac{d^2 \bar{b}_{34}}{dy^2} = \Gamma \bar{x} \left[\begin{array}{l} y^2 \bar{x}^2 (\Delta y - 3 \bar{y}) - 3 x y \delta \bar{y} (\Delta y + 3 \bar{y}) \\ + x^2 (\Delta y^3 + 3 \bar{y} (\Delta y^2 - \delta \bar{y})) \end{array} \right] \quad (\text{B.13})$$

$$\frac{d^2 \bar{b}_{41}}{dy^2} = -\Gamma \delta x \bar{y} [y^2 \delta x^2 + x \bar{y} (3 y \delta x + x \bar{y})] \quad (\text{B.14})$$

$$\frac{d^2 \bar{b}_{42}}{dy^2} = \Gamma \bar{y} \left[\begin{array}{l} 4 y^2 \Delta x^3 + 3 y^2 \bar{x}^2 (\bar{x} - 2 \Delta x) \\ +3 x y \bar{x} \bar{y} (3 \bar{x} - 4 \Delta x) + x^2 \bar{y}^2 (3 \bar{x} - 2 \Delta x) \end{array} \right] \quad (\text{B.15})$$

$$\frac{d^2 \bar{b}_{43}}{dy^2} = \Gamma \bar{y} \left[\begin{array}{l} y^2 (\Delta x^3 + 3 \bar{x} (\Delta x^2 - \delta x \bar{x})) \\ -3 x y \delta x \bar{y} (\Delta x + 3 \bar{x}) + x^2 \bar{y}^2 (\Delta x - 3 \bar{x}) \end{array} \right] \quad (\text{B.16})$$

$$\frac{d^2 \bar{b}_{44}}{dy^2} = \Gamma \bar{xy} (y^2 \bar{x}^2 + 3xy \bar{xy} + x^2 \bar{y}^2) \quad (\text{B.17})$$

B.4. “Sagittal” Direction Second Derivatives

$$\frac{d^2 \bar{b}_{11}}{dx^2} = \Gamma \delta x \delta y \left[\begin{array}{l} x^2 \delta x^2 + 3xy \delta x \Delta y + y^2 \Delta y^2 \\ + \bar{yy} (\bar{yy} - 3x \delta x - 2 y \Delta y) \end{array} \right] \quad (\text{B.18})$$

$$\frac{d^2 \bar{b}_{12}}{dx^2} = -\Gamma \delta y \left[\begin{array}{l} 4x^2 \Delta x^3 - 2 y^2 \Delta x \Delta y^2 + 3x^2 \bar{x}^3 \\ + 2 y^2 \Delta x \bar{xy} (2 \Delta y - \bar{y}) \\ - 3 y \delta y \bar{x} (y \Delta y - 4x \Delta x - \bar{yy}) \\ - 3x \bar{x}^2 (2x \Delta x - 3 y \Delta y + 3 \bar{yy}) \end{array} \right] \quad (\text{B.19})$$

$$\frac{d^2 \bar{b}_{13}}{dx^2} = -\Gamma \delta y \left[\begin{array}{l} -3x^2 \bar{x}^3 + 3x \bar{x}^2 (x \Delta x - 3 y \Delta y + 3 \bar{yy}) \\ + \Delta x \left(\begin{array}{l} x^2 \Delta x^2 + 3xy \Delta x \Delta y + y^2 \Delta y^2 \\ + \bar{yy} (\bar{yy} - 3x \Delta x - 2 y \Delta y) \end{array} \right) \\ + 3\bar{x} \left(\begin{array}{l} x^2 \Delta x^2 + 2xy \Delta x \Delta y - y^2 \Delta y^2 \\ - \bar{yy} (2x \Delta x - 2 y \Delta y + \bar{yy}) \end{array} \right) \end{array} \right] \quad (\text{B.20})$$

$$\frac{d^2 \bar{b}_{14}}{dx^2} = -\Gamma \delta y \bar{x} (y^2 \delta y^2 - 3xy \delta y \bar{x} + x^2 \bar{x}^2) \quad (\text{B.21})$$

$$\frac{d^2 \bar{b}_{21}}{dx^2} = \Gamma \delta x \left[\begin{array}{l} 2x^2 \delta x^2 \Delta y - 4 y^2 \Delta y^3 \\ + 3 \bar{y} \left(\begin{array}{l} \bar{yy} (3x \delta x + 2 y \Delta y - \bar{yy}) \\ - x \delta x (x \delta x + 4 y \Delta y) \end{array} \right) \end{array} \right] \quad (\text{B.22})$$

$$\frac{d^2 \bar{b}_{22}}{dx^2} = \Gamma \left[\begin{array}{l} 3x^2 \bar{x}^3 (3\bar{y} - 2\Delta y) \\ + 3x\bar{x}^2 (4x\Delta x\Delta y - 3\bar{y}(2x\Delta x - 4y\Delta y + 3\bar{y})) \\ - 2\Delta x \left(\begin{array}{l} 4(x^2\Delta x^2\Delta y + y^2\Delta y^3) \\ + 3\bar{y}(y^2\bar{y}(\bar{y} - 2\Delta y) - 2x^2\Delta x^2) \end{array} \right) \\ + 3y\bar{x} \left(\begin{array}{l} 4y\Delta y^3 \\ + \bar{y}(3\bar{y}(4x\Delta x - 2y\Delta y + \bar{y}) - 16x\Delta x\Delta y) \end{array} \right) \end{array} \right] \quad (\text{B.23})$$

$$\frac{d^2 \bar{b}_{23}}{dx^2} = \Gamma \left[\begin{array}{l} 4y^2\Delta x\Delta y^3 - 2x^2\Delta x^3\Delta y \\ - 6\Delta y\bar{x} (x^2\Delta x^2 + 2y^2\Delta y^2 - x^2\delta x\bar{x}) \\ + 3x\bar{y}\Delta x^2 (x\Delta x + 4y\Delta y) \\ + 3x\bar{x}\bar{y} \left(\begin{array}{l} \Delta x(3x\Delta x + 8y\Delta y) \\ - 3\bar{x}(4y\Delta y - x\Delta x + x\bar{x}) \end{array} \right) \\ - 3\bar{y}^2 \left(\begin{array}{l} \Delta x(3x\Delta x + 2y\Delta y) \\ - 3\bar{x}(2y\Delta y - 2x\Delta x + 3x\bar{x}) \end{array} \right) \\ + 3y^2\bar{y}^3 (\Delta x - 3\bar{x}) \end{array} \right] \quad (\text{B.24})$$

$$\frac{d^2 \bar{b}_{24}}{dx^2} = \Gamma \bar{x} \left[\begin{array}{l} 3xy\bar{x}\bar{y}(4\Delta y - 3\bar{y}) + x^2\bar{x}^2(3\bar{y} - 2\Delta y) \\ + y^2(4\Delta y^3 + 3\bar{y}^2(\bar{y} - 2\Delta y)) \end{array} \right] \quad (\text{B.25})$$

$$\frac{d^2 \bar{b}_{31}}{dx^2} = \Gamma \delta x \left[\begin{array}{l} 3\bar{y} \left(\begin{array}{l} x^2\delta x^2 + 2xy\delta x\Delta y \\ - y^2\Delta y^2 + \bar{y}(\bar{y} - 3x\delta x - y\Delta y) \end{array} \right) \\ - \Delta y(x^2\delta x^2 - 3xy\delta x\Delta y + y^2\Delta y^2) \end{array} \right] \quad (\text{B.26})$$

$$\frac{d^2 \bar{b}_{32}}{dx^2} = \Gamma \left[\begin{array}{l} 4x^2 \Delta x^3 \Delta y - 2y^2 \Delta x \Delta y^3 \\ + 3x^2 \bar{x}^3 (\Delta y - 3\bar{y}) \\ - 6\Delta x \bar{y} (2x^2 \Delta x^2 + y^2 \Delta y^2 - y^2 \delta \bar{y}) \\ + 3x \bar{x}^2 \left(\begin{array}{l} 3\bar{y} (2x \Delta x - 2y \Delta y + 3\bar{y}) \\ - \Delta y (2x \Delta x + 3y \Delta y) \end{array} \right) \\ + 3y \bar{x} \left(\begin{array}{l} \Delta y^2 (4x \Delta x + y \Delta y) \\ + \bar{y} \left(\begin{array}{l} \Delta y (8x \Delta x + 3y \Delta y) \\ - 3\bar{y} (4x \Delta x - y \Delta y + \bar{y}) \end{array} \right) \end{array} \right) \end{array} \right] \quad (\text{B.27})$$

$$\frac{d^2 \bar{b}_{33}}{dx^2} = \Gamma \left[\begin{array}{l} \Delta x \Delta y (x^2 \Delta x^2 - 3xy \Delta x \Delta y + y^2 \Delta y^2) \\ + 3\Delta y \bar{x} \left(\begin{array}{l} x^2 \Delta x^2 - 2xy \Delta x \Delta y - y^2 \Delta y^2 \\ + x \bar{x} (x \Delta x + 3y \Delta y - x \bar{x}) \end{array} \right) \\ - 3\bar{y} \Delta x (x^2 \Delta x^2 + 2xy \Delta x \Delta y - y^2 \Delta y^2) \\ - 3\bar{x} \bar{y} \left(\begin{array}{l} 3x^2 \Delta x^2 + 4xy \Delta x \Delta y + 3y^2 \Delta y^2 \\ - 3x \bar{x} (2y \Delta y - x \Delta x + x \bar{x}) \end{array} \right) \\ + 3\bar{y}^2 (\Delta x (3x \Delta x + y \Delta y) - 3\bar{x} (y \Delta y - 2x \Delta x + 3x \bar{x})) \\ - 3y^2 \bar{y}^3 (\Delta x - 3\bar{x}) \end{array} \right] \quad (\text{B.28})$$

$$\frac{d^2 \bar{b}_{34}}{dx^2} = \Gamma \bar{x} \left[\begin{array}{l} x^2 \bar{x}^2 (\Delta y - 3\bar{y}) \\ + 3xy \delta \bar{y} (\Delta y + 3\bar{y}) \\ + y^2 (\Delta y^3 + 3\bar{y} (\Delta y^2 - \delta \bar{y})) \end{array} \right] \quad (\text{B.29})$$

$$\frac{d^2 \bar{b}_{41}}{dx^2} = -\Gamma \delta \bar{x} \bar{y} [x^2 \delta x^2 + \bar{y} (-3x \delta x + \bar{y})] \quad (\text{B.30})$$

$$\frac{d^2 \bar{b}_{42}}{dx^2} = \Gamma \bar{y} \left[\begin{array}{l} x^2 (4\Delta x^3 + 3\bar{x}^2 (\bar{x} - 2\Delta x)) \\ + 3xy \bar{x} \bar{y} (4\Delta x - 3\bar{x}) - y^2 \bar{y}^2 (2\Delta x - 3\bar{x}) \end{array} \right] \quad (\text{B.31})$$

$$\frac{d^2 \bar{b}_{43}}{dx^2} = \Gamma \bar{y} \left[\begin{array}{l} x^2 (\Delta x^3 + 3\bar{x} (\Delta x^2 - \delta x \bar{x})) \\ + 3xy \delta \bar{x} y (\Delta x + 3\bar{x}) + y^2 \bar{y}^2 (\Delta x - 3\bar{x}) \end{array} \right] \quad \text{(B.32)}$$

$$\frac{d^2 \bar{b}_{44}}{dx^2} = \Gamma \bar{x} \bar{y} (x^2 \bar{x}^2 - 3xy \bar{x} \bar{y} + y^2 \bar{y}^2) \quad \text{(B.33)}$$

Appendix C Interferometer Description

A phase-shifting Mach-Zehnder interferometer was constructed using the hardware components detailed in Table C.1.

Laser	632.8 nm He-Ne
Spatial Filter	
Objective	40x (0.65 NA) objective
Pinhole	15 μm pinhole
Imaging Sub-System	
First Lens	160 mm f.l. achromatic doublet, 40 mm O.D.
Second Lens	60 mm f.l. achromatic doublet, 30 mm O.D.
Detector	Kodak KAF 1600a CCD chip

Table C.1. Hardware used for interferometer.

The collimated beam after the beam splitter is an oval approximately 45 mm (h) x 32 mm (w), so that it illuminates the entire height and 80% of the width of the sample window.

The image detector is a Kodak KAF 1600a CCD chip on a KAF Evaluation board. The board is capable of 12-bit digital acquisition, though noise and other effects limit practical resolution to 10 bits or so. The CCD has 1534x1024 pixels, spaced by 9 μm , and can acquire a maximum of 4 frames per second from the evaluation board. An electro-optic shutter by Meadowlark optics shutters the CCD during communication between the detector and the frame grabber (a Bitflow Data Raptor).

The laser is a linearly polarized red-orange Helium-Neon, made by Research Electro-Optics. It lases at 632.8 nm (red) and 612.0 nm (orange) wavelengths; an excitation filter made for the 632.8 nm (red) wavelength (by Chroma) provides wavelength selection (by blocking the orange wavelength).

The phase stepping of the reference mirror is achieved with a Burleigh 2” PZAT. The voltage signals are generated with a ComputerBoards PCI-DAS1602/16 card, and the Burleigh RG-93 Ramp generator amplifies this signal.³⁵

The hardware was controlled with the computer program LabView.

Appendix D Polymer Fabrication

D.1. Sample Mold Assembly

The sample mold is detailed in Appendix F. A simplified diagram is in Figure D.1. The mold is cleaned by wiping the inside surfaces of both halves with purified acetone. The glass windows are likewise cleaned and Teflon tape is wrapped around their edges. The glass windows are likewise cleaned and Teflon tape is wrapped around their edges. The window-slot edges are coated with vacuum grease, and the windows carefully inserted into the recessed area, so that the Teflon tape remains wrapped around the windows, providing a tight seal. This mounting technique secures the windows in the mold, creates a barrier against vapor flow through the window-mold interface, and allows for simple removal and cleanup after the experiment.

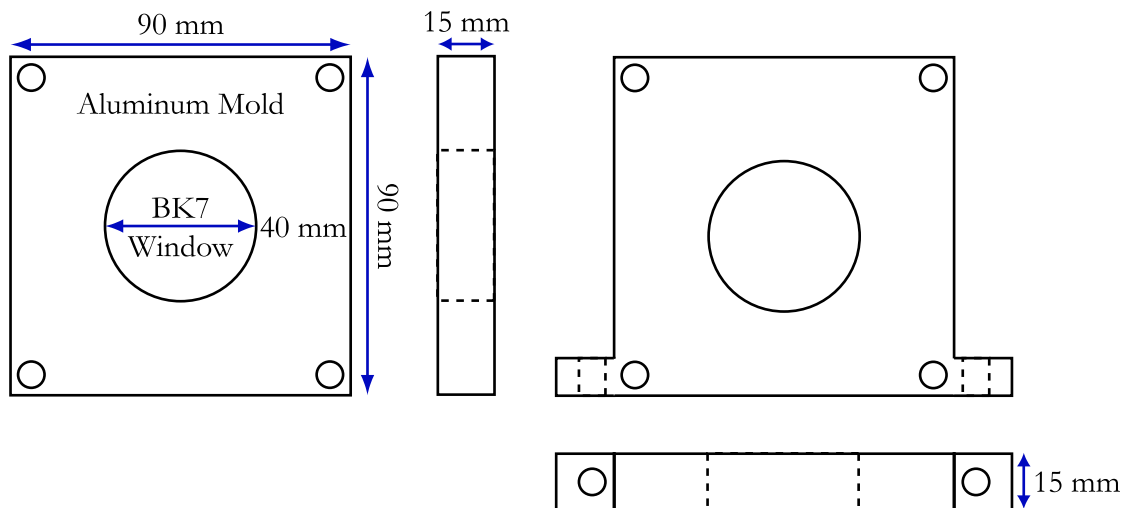


Figure D.1. Diffusion Mold

Next, the mold is assembled. (See Figure D.2) A length of 3 mm outer-diameter Teflon tubing, approximately 400 mm long, is held in a U-shape and

sandwiched between the two mold halves, with the opening of the “U” at the top of the mold. Four ¼-20 screws are inserted into the corners and finger tightened. The tubing is adjusted to be straight along the sides and so that the tubing will just close the top of the mold. The screws are then tightened sufficiently to hold the tubing in place. The sample, inside the assembled mold, will be approximately 2 mm thick.

After assembly, the mold is adjusted for measurements. It is filled with purified (18 MΩ) water, and mounted in the interferometer. The mold is nulled by adjusting the tightness of the corner screws to remove any tilt fringes. Finally, the top tubing is opened sufficiently to pour out the water, and the mold is rinsed with purified acetone. The mold is set aside until the polymer is prepared.

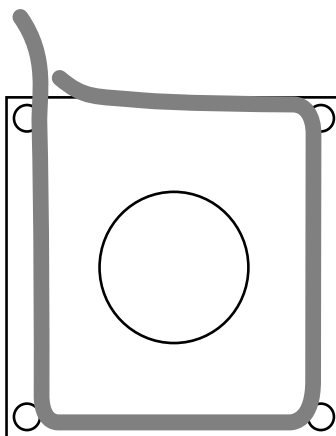


Figure D.2. Teflon™ tubing used to separate the mold halves.

D.2. Polymer Sample Fabrication

The chemicals used are detailed in Table D.1. The initial monomer, methylmethacrylate (MMA), is prepared according to the recipe in Table D.2. For this research 2000 g MMA was mixed with 20.00 g ethylene glycol dimethacrylate (EDMA)

and 2.000 g benzoyl peroxide (BPO). This quantity was sufficient for all experiments, and ensured the same mixture for all samples. The initiated MMA is kept refrigerated (10 °C) at all times; the styrene (St) is also refrigerated.

Chemical Name	Abbr.	Assay	Stabilizer	CAS No.
Methyl-methacrylate	MMA	99%	Hydroquinone	80-62-6
Ethylene glycol dimethacrylate	EDMA	98%,	100 ppm MEHQ	97-90-5
Benzoyl peroxide	BPO	97%	--	94-36-0
Styrene	St	99%	20 ppm 4-tert-Butylpyrocatechol	100-42-5

Table D.1. Specifications of chemicals used in research.

Purified MMA
1% EDMA (by weight)
0.1% BPO (by weight)

Table D.2. Recipe for initiated methyl-methacrylate (MMA).

To a 150ml flask is added 20.00 g of prepared MMA, which is first allowed to reach room temperature (22 °C). It is closed with a stopper with attached hose, which is connected to a vacuum pump. The pump is turned on and the flask agitated for 3 minutes to remove the dissolved oxygen from the monomer mixture. The hose is then clamped, detached from the pump, and the flask placed in a self-regulated 60 °C oven for 40 minutes. The flask is then removed from the oven and the stopper immediately removed. The partially polymerized MMA is carefully extracted by syringe and injected slowly into the sample mold, until it reaches a level of 3 mm below top of the glass window. The Teflon tubing is closed and covered with vinyl tape, sealing with mold. It is placed back into the 60 °C oven for the desired final polymerization time.

After the second polymerization step, the sample is immediately cooled, halting the polymerization process. This is done by placing it in a sealed plastic bag and immersing it in cold, running water for 15 minutes. The mold is removed from bag and let rest at room temperature until the sample has returned to room temperature (approximately 1 hour).

D.3. Experimental Procedure

The mold is then carefully mounted in the interferometer and nulled (by adjusting the corner screws). Phase measurements of the homogeneous sample are taken every 10 minutes for one hour.

Following the homogeneous measurements, the Teflon tubing, sealing the top of the mold, is opened slightly. Inhibited styrene, which is at room temperature, is injected until the mold is filled. The mold is quickly closed and sealed. The diffusion measurements are immediately begun, acquiring data every 30 minutes for 16 hours.

Appendix E Phase Unwrap Pseudo-Code

Select unwrap start position

1st Order - Function Continuity:

Find all points where phase changes by more than π

Add 2π cumulatively beyond all points those found

Cumulatively sum all phase to unwrap

2nd Order - Slope Continuity (and physical constraint):

Compute slope for unwrapped phase

Find all points where phase change is greater than $\pi/2$ and slope is negative

Add 2π cumulatively beyond all points those found

3rd Order - Curvature Continuity (and physical constraint):

Compute curvature for unwrapped phase

Find all points where phase change is greater than $\pi/2$, slope is positive, and curvature is not zero

Add 2π cumulatively beyond all points those found

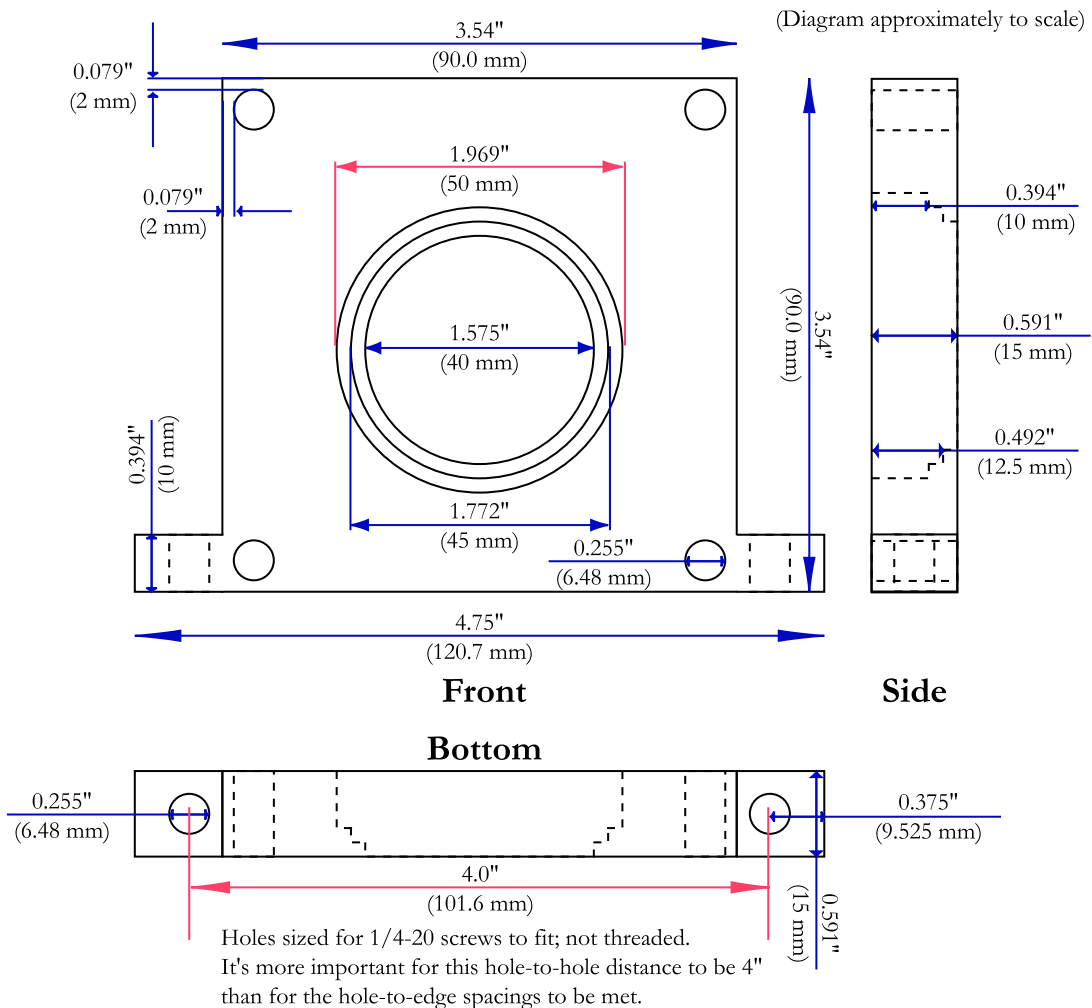
4th Order:

Compute slope of unwrapped phase

Find all points where phase change < -3 rad, for five points consecutively

Add 2π cumulatively beyond all points found

Appendix F Diffusion Sample Mold Design

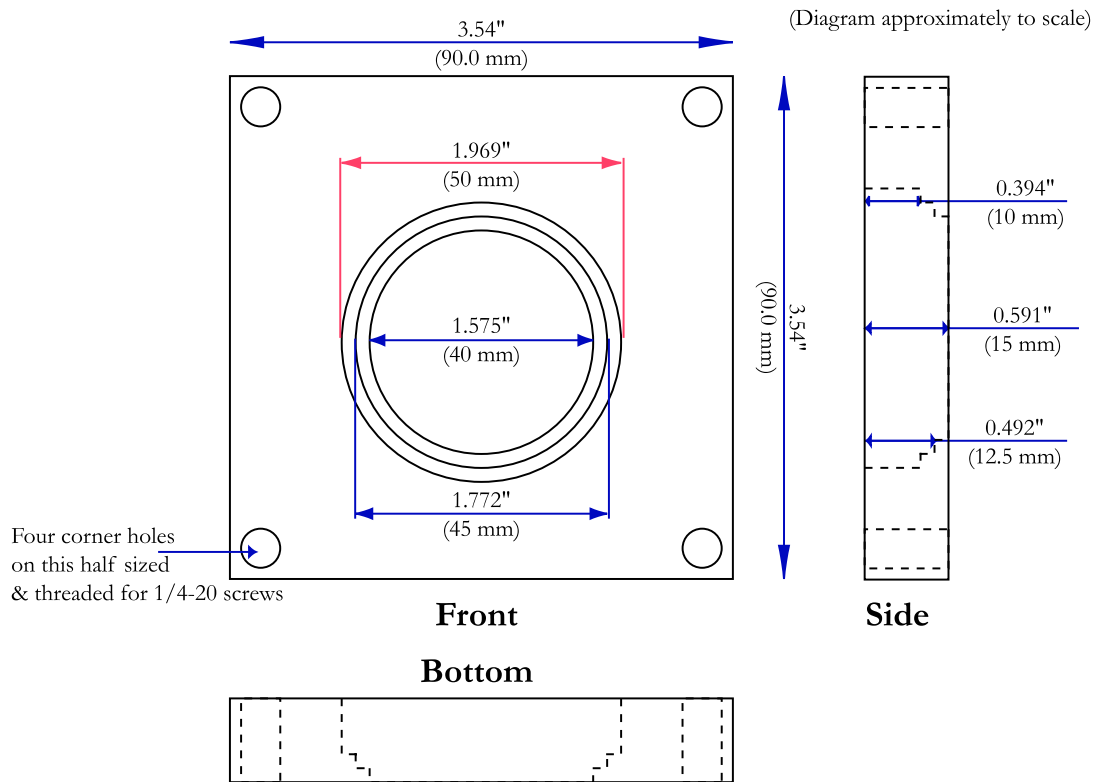


Blue = $\pm 0.040"$ (± 1 mm) tolerance
 Red = $-0.002" / +0.040"$ (-0.05 mm / $+1$ mm) tolerance

These are the two halves of a mold.
 The four corner holes must be registered in order for them to be screwed together

On this half, the four corner holes are not threaded; 1/4-20 screws fit smoothly

Figure F.1. First half of polymer fabrication and diffusion mold.



Blue = $\pm 0.040"$ (± 1 mm) tolerance
 Red = $-0.002" / +0.040"$ (-0.05 mm / $+1$ mm) tolerance

These are the two halves of a mold.
 The four corner holes must be registered in order for them to be screwed together

On this half, the four corner holes are sized & threaded for 1/4-20 screws

Figure F.2. Second half of polymer fabrication and diffusion mold.

Appendix G Computational Code for Ray-Tracing Program

G.1. Variable.h

```

#include <time.h>
/***** CONSTANTS *****/
#define M_ROOT_PI 1.7724538509055160273
#define M_PI 3.14159265358979323846
#define M_PI_2 M_PI/2
#define M_ROOT_PI 1.7724538509055160273
#define INFINITY 1e8/*Object distances greater than this is considered to be at Infinity*/
#define FILENUM 12 /*Number of File pointers to open*/
#define ZERO 1e-300
/***** DATA TYPES *****/
/*
 * NONE : No error
 * GENERAL : catch all
 * MEMORY : memory could not be allocated
 * PUPIL : the exit pupil ray (tangential or sagittal ray) could not be found
 * NAN : computed value was NaN
 * NUMERICAL: numerical computation error
 * APERTURE : ray misses surface
 * TIR : total internal reflection -- sqrt(-val) in refract
 * VARIABLE : a variable pointer has a NULL value, and so cannot be used
 * NEGEDG : negative edge thickness encountered in ray-propagation
 */
typedef enum errorCode_ {NONE=0, GENERAL, MEMORY, PUPIL, APERTURE, NUMERICAL, NAN, TIR, VARIABLE, NEGEDG, IO}
errorCode;
/* RES : Results file (MOE, OA, power, etc)
 * SPECS : Lens, System, Optimization specifications
 * DEBUG : Debug information file
 * AXIS : Gaze-angle and distance Axes file
 * MERIT : Merit-space file
 */
typedef enum fileType_ {F_SPECS=0, F_AXIS, F_TAN, F_SAG, F_AST, F_MOE, F_INDEX, F_DIST, F_MERIT, F_INDPW,
F_INDAST, F_INDMOE, F_DEBUG} fileType;
/*
 * O_NONE : No optimization
 * O_AUTO : Normal optimization : runs until optimizer completes
 * O_USER : Allows the user to interrupt Multi-Dim opt after each major sequence
 * other : Any value >= 1 instructs the optimizer to run the full opt sequence that many times
 */
typedef enum optMode_ {O_NONE=0, O_USER=-1, O_AUTO=1};
/* DEG : Degrees
 * RAD : Radians
 * TANX : Tan( theta )
 * DC : Direction Cosine values
 * RELZ : Relative to the z-axis
 */
typedef enum angleMode_ {DEG=0, RAD, TANX, DC, CV, PARTIAL=10, RELZ=20} angleMode;
/* TAN : Tangential- (x-) axis of the Exit Pupil
 * SAG : Sagittal- (y-) axis of the Exit Pupil
 * TANN : Negative Tangential axis
 * SAGN : Negative Sagittal axis
 */
typedef enum epAxisMode_ {TAN=0, SAG, TANN, SAGN} epAxisMode;
/* G_H : Homogeneous Media n(r) = n
 * G_R : Polynomial representation n(r) = Sum[ai * r^i, i=0..N]
 * G_RSQ : Polynomial representation n(r) = Sum[ai * r^2i, i=0..N]
 * G_XY : Polynomial representation n(x,y) = Sum[aij * x^i y^j, i=0..I, j=0..J]
 * G_BSP : Bicubic B-spline representation
 * G_BSPFIT : Begin with a non-G_BSP representation; generate a B-spline representation;and work with that
 */
typedef enum grinMode_ {G_H=0, G_R, G_RSQ, G_XY, G_BSP,G_BSPFIT=20} grinMode;
/* S_SPH : Spherical Surface
 * S_PLANE : Planar Surface
 * S_PAL : Polynomial PAL surface
 */
typedef enum surfMode_ {S_SPH=0, S_PLANE, S_PAL } surfMode;
/* Power expressions are functions of gaze-angle alpha (radians)
 * P_POLY : Polynomial expression for power
 * P_HARD : "Hard" expression for power: Linear addition followed by constant
 * P_EXP : Exponential form of power addition (modified from Patent 5,042,936, c.5 (74))
 * P_GAUSS : Gaussian form of power addition

```

```

*          Taken from the desiredPower() documentation:
*          P_POLY : pcoeff[i] is the coeff to the ith power for alpha (a^i)
*          P_HARD : this mode describes a linear add, followed by constant power
*                   0 -> base power, 1 -> add power, 2 -> alpha end of 'add'
*          P_EXP : 0 -> base power term (base)
*                   1 -> add power term (add), 2 -> rate of add term (k), 3 -> midpoint of add term
*
*          P_GAUSS: 0 -> base power term (base), 1 -> add power term (add), 2 -> width of add (w)
*/
typedef enum powerMode_ {P_POLY=0, P_HARD, P_EXP, P_GAUSS } powerMode;
typedef struct vector_ *vectorPtr;
typedef struct vector_ {double x; double y; double z;} vector;
typedef struct dirCos_ *dirCosPtr;
typedef struct dirCos_ {double alpha; double beta; double gamma;} dirCos;
typedef struct lensDef_ *lensDefPtr;
typedef struct palSurf_ *palPtr;
/*Defines a Progressive (Aspherical) Surface*/
typedef struct palSurf_ {
double radiusFar; /*radius of far section*/
double (*radiusPro)(double, palPtr); /*function giving radius along MM of progressive section*/
double radiusNear; /*radius of near section*/
double startPro; /*dist from 0 (zero) to start of progressive section*/
double startFar; /*dist from 0 (zero) to start of far section*/
double slope; /*these 2 variables to unrotate, and recenter surface*/
double yintercept;
double rp1, rp2, rp3; /*radius = rp1 / (rp2 + rp3*x)*/
double qp1, qp2, qp3; /*qPro = qp1 + qp2*x + qp3*x*x*/
double up1, up2, up3, up4, up5; /*uPro = (up1 + up2*x + up3*x*x)/(up4 + up5*x)*/
vector origin;} palSurf;
/* The equation of the plane is given by: mx (x-xi) + my (y-yi) + mz (z-zi) = 0*/
typedef struct planeSurf_ {double xi, yi, zi; double mx, my, mz;} planeSurf;
typedef struct sphereSurf_ {
double radius; /*radius*/
double rmerit[3]; /*array of merit space values: min, max, step*/
int ropt;
int na; /*Number of coefficients in a (aspheric) array*/
double *a; /*Aspheric coefficients*/
/*a[0] = conic constant*/
/*a[1..na-1] = aspheric coeffs*/
/*Corresponds to: a[0], a[1], a[2], a[3], ...*/
/* : k, r^4, r^5, r^6, ...*/
int *aopt; /*Flags for which asphere coeff to opt on*/
/*0 = no opt; 1 = opt*/
double *amerit[3]; /*array of merit space values: min, max, step*/
vector origin; /*the position of the center of the sphere*/
} sphereSurf;
/*General Surface type*/
typedef struct surface_ {
surfMode mode; /*Surface representation mode*/
double sa; /*Semi-aperture size*/
sphereSurf sph; /*Spherical surface*/
palSurf pal; /*Progressive surface*/
planeSurf pla; /*Planar surface*/
} surface;
/*Bicubic B-spline Variable*/
typedef struct _bspline {
double xo, yo, /*Minimum position on axis*/
xf, yf, /*Maximum position on axis*/
dx, dy; /*Width of patch with 4 control pts*/
/*This is also the B-spline width / 4*/
int nl, nm, /*Total number of (x,y) control points*/
L, M, /*Total number of patches*/
lorigin, morigin, /*Patch # for x-y origin*/
llo, mlo, /*Optimized control point # low*/
lhi, mhi; /*Optimized control point # high*/
int *zopt; /*(i)->(control point #) map for spline*/
double *zc; /*Control points (v/m/y-major order)*/
} bspline;
/*Polynomial Variable*/
typedef struct _poly {
/******
int nx; /*Number of coefficients in n array for GRIN modes G_R, G_RSQ*/
/*Maximum order+1 of x for G_XY (ie x^3 -> nl=4)*/
int ny; /*Maximum order+1 of y for G_XY (ie y^3 -> n2=4), Unused otherwise*/
int *zopt; /*Flags for which index coeff to opt on*/
/*0 = no opt; 1 = opt*/
/*must be one flag for each index coeff*/
double *zmerit[3]; /*array of merit space values: min, max, step*/
double xo; /*x-offset to allow for series expansion about x=xo*/
double yo; /*y-offset to allow for series expansion about y=yo*/
double *zc; /*Linear array of polynomial coefficients describing the index*/
/*for G_XY data is saved in y-row-major order*/
/******
} poly;
/*Polynomial Variable*/
typedef struct _homog {
int zopt; /*Flag for opt: 0 = no opt; 1 = opt*/
double zmerit[3]; /*array of merit space values: min, max, step*/
double ni; /*Index value*/
} homog;

```

```

typedef struct grinDef_ *grinDefPtr;
typedef struct grinDef_ {
    int steps; /*Nominal number of steps to use in tracing through GRIN medium*/
    grinMode mode; /*GRIN representation mode*/
    homog nh; /*index for homogeneous mode*/
    poly np; /*Polynomial representation of index*/
    bspline nbs; /*B-spline representation of index*/
    int nx, ny; /*Number of points to sample along x, y-axes for outputting index*/
} grinDef;
/*Lens Specifications*/
typedef struct lensDef_ {
    surface surf1; /*Front Surface*/
    surface surf2; /*Back Surface*/
    grinDef grin; /*GRIN description*/
    double t; /*lens thickness*/
    double power; /*Center power (BFL) of lens*/
} lensDef;
/*System Specifications*/
typedef struct sysDef_ {
    double q; /*Center of Rotation of Eye, Relative to back lens surface*/
    /*q > 0 : q is to right of lens*/
    double p; /*Pupil Distance from q*/
    /*p < 0 : p is to left of q*/
    double pRad; /*pupil radius*/
    double dAngle; /*delta-(gaze angle) used in ray tracing*/
    int maxAlphaCount; /*maximum count for alpha gaze-angle steps*/
    int minAlphaCount; /*minimum count for alpha gaze-angle steps*/
    int maxBetaCount; /*maximum count for beta gaze-angle steps*/
    int minBetaCount; /*minimum count for beta gaze-angle steps*/
    double alphaOffset; /*alpha angle offset needed to trace correct range*/
    double betaOffset; /*beta angle offset needed to trace correct range*/
    double object; /*object distance*/
    sphereSurf *vertex; /*Vertex Sphere of eye geometry*/
    char prefix[100]; /*prefix for output files*/
    char suffix[100]; /*suffix for output files*/
    int trace; /*Ray-Trace flag*/
    double traceDim[4]; /*Gaze-angle limits for ray-tracing*/
    /*[0] = alpha min [1] = alpha max*/
    /*[2] = beta min [3] = beta max*/
    double time; /*Program run-time*/
    double *alpha; /*Alpha axis gaze-angles*/
    double *beta; /*Beta axis gaze-angles*/
    double *od; /*Object distances*/
    double *dp; /*Desired Power*/
    double *T; /*Tangential power*/
    double *S; /*Sagittal power*/
    double *A; /*Oblique Astigmatism*/
    double *M; /*Mean Oblique Error*/
    double *D; /*Distortion Error*/
    double *iA; /*Indicial Oblique Astigmatism*/
    double *iM; /*Indicial Mean Oblique Error*/
    double *iP; /*Indicial Power*/
} sysDef;
/*Optimization Specifications*/
typedef struct optDef_ {
    int opt; /*Optimization flag: Number of Loops*/
    int merit; /*Merit Space flag*/
    double startMerit; /*Initial Merit Value*/
    double endMerit; /*Ending Merit Value*/
    optDim[4]; /*Gaze-angle limits for ray-tracing: [0,1,2,3] = alpha min, max, beta min, beta max*/
    dAngle, /*delta-(gaze Angle) used in optimization*/
    weight[3]; /*Optimization weights for Ast, Moe, Dist*/
    powerMode power; /*Desired power flag*/
    double *pcoeff; /*Desired Power Coeff*/
    int np; /*Number of coeffs in above array*/
    int n_s1, /*Number of free parameters on surface 1*/
        n_s2, /*Number of free parameters on surface 2*/
        n_n; /*Number of free parameters on index*/
    double tollD, /*Tolerance factor for 1D Optimization*/
        tolMD; /*Tolerance factor for Multi-D Optimization*/
    int meritWidth; /*Number of ± patches to compute merit function with*/
} optDef;

```

G.2. B-spline.c

```

#define CP(ii,jj) *(bcs->zc + (jj)*(bcs->nl) + (ii))
#define CPidx(ii,jj) ((jj)*(bcs->nl) + (ii))
/*****
* Name: freeBSpline
* Input: bcs : B-spline structure
* Modifies: bcs
* Purpose: Free the control array in the B-spline structure
* Requires: Control-Point array must be NULL if it has not been allocated
*****/
void freeBSpline(bspline *bcs) {if(bcs->zc != NULL) {free(bcs->zc);bcs->zc = NULL;}return;}
/*****

```

```

* Name:   initBSpline
* Input:  bcs : B-spline structure. L : number of x-patches. xo : minimum x-position for B-spline
*         xf : maximum x-position for B-spline. M : number of y-patches. yo : minimum y-position for B-
spline
*         yf : maximum y-position for B-spline
* Modifies: bcs. Global error is memory not available
* Purpose: Set values for all necessary parameters in B-spline and allocated memory for control points array
* Requires: L, M > 0, Control-Point array must be NULL if it has not been allocated
*****
void initBSpline(bspline *bcs, int L, double xo, double xf, int M, double yo, double yf) {
    bcs->L = L; bcs->nL = L+3; bcs->xo = xo; bcs->xf = xf; bcs->dx = (xf-xo)/L;
    bcs->M = M; bcs->nM = M+3; bcs->yo = yo; bcs->yf = yf; bcs->dy = (yf-yo)/M;
    freeBSpline(bcs);
    bcs->zc = (double *)calloc(bcs->nL*bcs->nM, sizeof(double));
    if(bcs->zc == NULL) {return;}
    return;}
/*****
* Name:   bsp
* Input:  bcs : B-spline structure. R : position to evaluate B-spline
* Returns: B-spline value at R
* Purpose: Calculate B-spline value at input position
* Requires: bcs is allocated and defined. Input position is within bounds of B-spline domain: (xo<=x<=xf)
(yo<=y<=yf)
*****
double bsp(bspline *bcs, vector *R) {
    int l, m;
    double x, dx, xo, xf, u, ou, b0u, blv, b2u, b3u, y, dy, yo, yf, v, ov, b0v, blv, b2v, b3v, bval;
    y = R->y; yo = bcs->yo; yf = bcs->yf; dy = bcs->dy;
    x = R->x; xo = bcs->xo; xf = bcs->xf; dx = bcs->dx;
    if( x < xo || y < yo || x > xf || y > yf ) return 0;
    l = (int)floor((x-xo)/dx); if(l==bcs->L) l--;m = (int)floor((y-yo)/dy); if(m==bcs->M) m--;
    u = (x - (xo+l*dx))/dx;v = (y - (yo+m*dy))/dy;ov = 1.-v;
    b0v = ov*ov*ov/6.;blv = (4 + v*v*(-6 + 3*v))/6.;b2v = (1. + 3*v*(1. + v*(1.-v)))/6.;b3v = v*v*v/6.;
    ou = 1.-u;
    b0u = ou*ou*ou/6.;blu = (4 + u*u*(-6 + 3*u))/6.;b2u = (1. + 3*u*(1. + u*(1.-u)))/6.;b3u = u*u*u/6.;
    bval = 0;
    bval += CP(1+0,m+0)*b0u*b0v + CP(1+1,m+0)*blu*b0v + CP(1+2,m+0)*b2u*b0v + CP(1+3,m+0)*b3u*b0v;
    bval += CP(1+0,m+1)*b0u*b1v + CP(1+1,m+1)*blu*b1v + CP(1+2,m+1)*b2u*b1v + CP(1+3,m+1)*b3u*b1v;
    bval += CP(1+0,m+2)*b0u*b2v + CP(1+1,m+2)*blu*b2v + CP(1+2,m+2)*b2u*b2v + CP(1+3,m+2)*b3u*b2v;
    bval += CP(1+0,m+3)*b0u*b3v + CP(1+1,m+3)*blu*b3v + CP(1+2,m+3)*b2u*b3v + CP(1+3,m+3)*b3u*b3v;
    return bval;}
/*****
* Name:   diffBspY
* Input:  bcs : B-spline structure. R : position to evaluate B-spline
* Returns: B-spline derivative value (with respect to y) at R
* Purpose: Calculate B-spline derivative value (wrt/y) at input position
* Requires: bcs is allocated and defined. Input position is within bounds of B-spline domain:
(xo<=x<=xf) (yo<=y<=yf)
*****
double diffBspY(bspline *bcs, vector *R) {
    int l, m;
    double x, dx, xo, xf, u, ou, b0dv, b1dv, b2dv, b3dv, y, dy, yo, yf, v, ov, b0u, blv, b2u, b3u, bval;
    y = R->y; yo = bcs->yo; yf = bcs->yf; dy = bcs->dy;x = R->x; xo = bcs->xo; xf = bcs->xf; dx = bcs->dx;
    if( x < xo || y < yo || x > xf || y > yf ) return 0;
    l = (int)floor((x-xo)/dx); if(l==bcs->L) l--;m = (int)floor((y-yo)/dy); if(m==bcs->M) m--;
    u = (x - (xo+l*dx))/dx;v = (y - (yo+m*dy))/dy;
    ov = 1.-v;
    b0dv = -ov*ov/2.;b1dv = (-4 + 3*v)*v/2.;b2dv = 1/2.+v*(1.-3/2.*v);b3dv = v*v/2.;
    ou = 1.-u;
    b0u = ou*ou*ou/6.;blu = (4 + u*u*(-6 + 3*u))/6.;b2u = (1. + 3*u*(1. + u*(1.-u)))/6.;
    b3u = u*u*u/6.;
    bval = 0;
    bval += CP(1+0,m+0)*b0u*b0dv + CP(1+1,m+0)*blu*b0dv + CP(1+2,m+0)*b2u*b0dv + CP(1+3,m+0)*b3u*b0dv;
    bval += CP(1+0,m+1)*b0u*b1dv + CP(1+1,m+1)*blu*b1dv + CP(1+2,m+1)*b2u*b1dv + CP(1+3,m+1)*b3u*b1dv;
    bval += CP(1+0,m+2)*b0u*b2dv + CP(1+1,m+2)*blu*b2dv + CP(1+2,m+2)*b2u*b2dv + CP(1+3,m+2)*b3u*b2dv;
    bval += CP(1+0,m+3)*b0u*b3dv + CP(1+1,m+3)*blu*b3dv + CP(1+2,m+3)*b2u*b3dv + CP(1+3,m+3)*b3u*b3dv;
    bval /= dy;
    return bval;}
/*****
* Name:   diffBspX
* Input:  bcs : B-spline structure. R : position to evaluate B-spline
* Returns: B-spline derivative value (with respect to x) at R
* Purpose: Calculate B-spline derivative value (wrt/x) at input position
* Requires: bcs is allocated and defined. Input position is within bounds of B-spline domain: (xo<=x<=xf)
(yo<=y<=yf)
*****
double diffBspX(bspline *bcs, vector *R) {
    int l, m;
    double x, dx, xo, xf, u, ou, b0du, b1du, b2du, b3du, y, dy, yo, yf, v, ov, b0v, blv, b2v, b3v,bval;
    y = R->y; yo = bcs->yo; yf = bcs->yf; dy = bcs->dy;x = R->x; xo = bcs->xo; xf = bcs->xf; dx = bcs->dx;
    if( x < xo || y < yo || x > xf || y > yf ) return 0;
    l = (int)floor((x-xo)/dx); if(l==bcs->L) l--;m = (int)floor((y-yo)/dy); if(m==bcs->M) m--;
    u = (x - (xo+l*dx))/dx;v = (y - (yo+m*dy))/dy;
    ov = 1.-v;b0v = ov*ov*ov/6.;blv = (4 + v*v*(-6 + 3*v))/6.;b2v = (1. + 3*v*(1. + v*(1.-v)))/6.;b3v =
v*v*v/6.;
    ou = 1.-u;b0du = -ou*ou/2.;b1du = (-4 + 3*u)*u/2.;b2du = 1/2.+u*(1.-3/2.*u);b3du = u*u/2.;
    bval = 0;
    bval += CP(1+0,m+0)*b0du*b0v + CP(1+1,m+0)*b1du*b0v + CP(1+2,m+0)*b2du*b0v + CP(1+3,m+0)*b3du*b0v;
    bval += CP(1+0,m+1)*b0du*b1v + CP(1+1,m+1)*b1du*b1v + CP(1+2,m+1)*b2du*b1v + CP(1+3,m+1)*b3du*b1v;
    bval += CP(1+0,m+2)*b0du*b2v + CP(1+1,m+2)*b1du*b2v + CP(1+2,m+2)*b2du*b2v + CP(1+3,m+2)*b3du*b2v;

```

```

    bval += CP(1+0,m+3)*b0du*b3v + CP(1+1,m+3)*b1du*b3v + CP(1+2,m+3)*b2du*b3v + CP(1+3,m+3)*b3du*b3v;
    bval /= dx;
    return bval;}
/*****
* Name:    diffBspYY
* Input:  bcs : B-spline structure. R : position to evaluate B-spline
* Returns: B-spline 2nd-derivative value (with respect to y) at R
* Purpose: Calculate B-spline 2nd-derivative value (wrt/y) at input position
* Requires: bcs is allocated and defined. Input position is within bounds of B-spline domain: (xo<=x<=xf)
(yo<=y<=yf)
*****/
double diffBspYY(bspline *bcs, vector *R) {
    int l, m;
    double x, dx, xo, xf, u, ou, b0u, blu, b2u, b3u, y, dy, yo, yf, v, b0ddv, b1ddv, b2ddv, b3ddv, bval;
    y = R->y; yo = bcs->yo; yf = bcs->yf; dy = bcs->dy; x = R->x; xo = bcs->xo; xf = bcs->xf; dx = bcs->dx;
    if( x < xo || y < yo || x > xf || y > yf ) return 0;
    l = (int)floor((x-xo)/dx); if(l==bcs->L) l--; m = (int)floor((y-yo)/dy); if(m==bcs->M) m--;
    u = (x - (xo+l*dx))/dx; v = (y - (yo+m*dy))/dy;
    b0ddv = 1-v;b1ddv = 3*v-2;b2ddv = 1-3*v;b3ddv = v;
    ou = 1.-u;
    b0u = ou*ou*ou/6.;blu = (4 + u*u*(-6 + 3*u))/6.;b2u = (1. + 3*u*(1. + u*(1.-u)))/6.;b3u = u*u*u/6.;
    bval = 0;
    bval += CP(1+0,m+0)*b0u*b0ddv + CP(1+1,m+0)*blu*b0ddv + CP(1+2,m+0)*b2u*b0ddv + CP(1+3,m+0)*b3u*b0ddv;
    bval += CP(1+0,m+1)*b0u*b1ddv + CP(1+1,m+1)*blu*b1ddv + CP(1+2,m+1)*b2u*b1ddv + CP(1+3,m+1)*b3u*b1ddv;
    bval += CP(1+0,m+2)*b0u*b2ddv + CP(1+1,m+2)*blu*b2ddv + CP(1+2,m+2)*b2u*b2ddv + CP(1+3,m+2)*b3u*b2ddv;
    bval += CP(1+0,m+3)*b0u*b3ddv + CP(1+1,m+3)*blu*b3ddv + CP(1+2,m+3)*b2u*b3ddv + CP(1+3,m+3)*b3u*b3ddv;
    bval /= dx*dx;
    return bval;}
/*****
* Name:    diffBspXX
* Input:  bcs : B-spline structure. R : position to evaluate B-spline
* Returns: B-spline 2nd-derivative value (with respect to x) at R
* Purpose: Calculate B-spline 2nd-derivative value (wrt/x) at input position
* Requires: bcs is allocated and defined. Input position is within bounds of B-spline domain: (xo<=x<=xf)
(yo<=y<=yf)
*****/
double diffBspXX(bspline *bcs, vector *R) {
    int l, m;
    double x, dx, xo, xf, u,b0ddu, b1ddu, b2ddu, b3ddu, y, dy, yo, yf, v, ov, b0v,b1v,b2v,b3v, bval;
    y = R->y; yo = bcs->yo; yf = bcs->yf; dy = bcs->dy; x = R->x; xo = bcs->xo; xf = bcs->xf; dx = bcs->dx;
    if( x < xo || y < yo || x > xf || y > yf ) return 0;
    l = (int)floor((x-xo)/dx); if(l==bcs->L) l--; m = (int)floor((y-yo)/dy); if(m==bcs->M) m--;
    u = (x - (xo+l*dx))/dx; v = (y - (yo+m*dy))/dy;
    ov = 1.-v;b0v = ov*ov*ov/6.;b1v = (4 + v*v*(-6 + 3*v))/6.;b2v = (1. + 3*v*(1. + v*(1.-v)))/6.;b3v =
    v*v*v/6.;
    b0ddu = 1-u;b1ddu = 3*u-2;b2ddu = 1-3*u;b3ddu = u;
    bval = 0;
    bval += CP(1+0,m+0)*b0ddu*b0v + CP(1+1,m+0)*b1ddu*b0v + CP(1+2,m+0)*b2ddu*b0v + CP(1+3,m+0)*b3ddu*b0v;
    bval += CP(1+0,m+1)*b0ddu*b1v + CP(1+1,m+1)*b1ddu*b1v + CP(1+2,m+1)*b2ddu*b1v + CP(1+3,m+1)*b3ddu*b1v;
    bval += CP(1+0,m+2)*b0ddu*b2v + CP(1+1,m+2)*b1ddu*b2v + CP(1+2,m+2)*b2ddu*b2v + CP(1+3,m+2)*b3ddu*b2v;
    bval += CP(1+0,m+3)*b0ddu*b3v + CP(1+1,m+3)*b1ddu*b3v + CP(1+2,m+3)*b2ddu*b3v + CP(1+3,m+3)*b3ddu*b3v;
    bval /= dy*dy;
    return bval;}
/*****
* Name:    bSpline
* Input:  x : linear (non-normalized) position. dx : path width
* Returns: Bell-Curve value at x. Zero for x <0 and x > 4*dx
* Purpose: Calculate value of B-spline basis function (bell curve) at x.
* Requires: 0 <= x <= 4*dx
*****/
double bSpline(double x, double dx) {
    double u, bval;
    u = x/dx;
    if(u < 0)bval = 0;
    else if(u < 1) {bval = u*u*u/6.;}
    else if(u < 2) {u -= 1; bval = (1. + 3*u*(1. + u*(1.-u)))/6.;}
    else if(u < 3) {u -= 2; bval = (4 + u*u*(-6 + 3*u))/6.;}
    else if(u <= 4) {u -= 3; u = 1 - u; bval = u*u*u/6.;}
    else bval=0;
    return bval;}
/*****
* Name:    fitBSpline
* Input:  bcs : B-spline structure. grin: grin structure.
*       num : number of points along x, y axes to calculate function values
* Modifies: B-spline control points
* Purpose: Least Squares Fit B-spline to index profile over range determined by B-spline domain
*       parameters. The number of functions values computed in the fit along the x-,y-axes
*       is determined by num. If the number of points along an axis is less than the number
*       of control points along that axis, the the number of points used is (# control points) + 10
* Requires: bcs parameters are defined and control point array is allocated
* Dependencies: index, bSpline, transpose, multiply, gaussj
*****/
void fitBSpline(bspline *bcs, grinDef *grin, int num) {
    int nl, nm;
    double *A, *At, *d, x, dx, xo, xs,*B, *Bt, *c, y, dy, yo, ys;
    int numx, numy, i, j;
    vector R;
    nl = bcs->nl;nm = bcs->nm;
    if(num < nl) num = nl+10;
    numx = num; xs = (bcs->xf - bcs->xo)/(numx-1); dx = bcs->dx; xo = bcs->xo;

```

```

numy = num; ys = (bcs->yf - bcs->yo)/(numy-1); dy = bcs->dy; yo = bcs->yo;
A = (double *)calloc((numx)*(nm), sizeof(double)); if(A == (double *)NULL) exit(-1);
At = (double *)calloc((nm) *(numx), sizeof(double)); if(At == (double *)NULL) exit(-1);
d = (double *)calloc((numx)*(numy), sizeof(double)); if(d == (double *)NULL) exit(-1);
for(i=0; i<numx; i++) {for(j=0; j<nm; j++) {y = i*ys + yo;A[i*nm + j] = bspline(y-yo-(j-3)*dy, dy);}}
for(i=0; i<numx; i++) {for(j=0; j<numy; j++) {R.x = j*xs+xo;R.y = i*ys+yo;d[i*numx + j] = index(grin,
&R);}}

/*Matrix [rows,cols]*/
/*A [numx,nm]*/
/*d [numx,numy]*/
transpose(A, At, numx, nm); /*At -> Tranpose(A) [nm,numx]*/
multiply(At, nm, numx, A, numx, nm, A); /*A -> At.A [nm,nm]*/
multiply(At, nm, numx, d, numx, numy, d);/*d -> At.d [nm,numy]*/
free(At);
gaussj(A, nm, d, numy); /*d -> Inverse(A).d [nm,numy]*/
free(A);
B = (double *)calloc((numy)*(nl), sizeof(double)); if(B == (double *)NULL) exit(-1);
Bt = (double *)calloc((nl) *(numy), sizeof(double)); if(Bt == (double *)NULL) exit(-1);
c = (double *)calloc((numx)*(numy), sizeof(double)); if(c == (double *)NULL) exit(-1);
for(i=0; i<numy; i++) {for(j=0; j<nl; j++) {x = i*xs + xo;B[i*nl + j] = bspline(x-xo-(j-3)*dx, dx);}}
/*Matrix [rows,cols]*/
/*B [numy,nl]*/
transpose(d, c, nm, numy); /*c -> Tranpose(d) [numy,nm]*/
free(d);
transpose(B, Bt, numy, nm); /*Bt -> Tranpose(B) [nl,numy]*/
multiply(Bt, nm, numy, B, numy, nl, B);/*B -> Bt.B [nl,nl]*/
multiply(Bt, nm, numy, c, numy, nm, c);/*c -> Bt.c [nl,nm]*/
free(Bt);
gaussj(B, nl, c, nm); /*c -> Inverse(B).c [nl,nm]*/
free(B);
for(i=0; i<nl; i++) for(j=0; j<nm; j++) CP(i,j) = c[i*nm + j];
free(c);}
/*****
* Name: bsplineFitError
* Input: bcs : B-spline structure. grin: grin structure. N : number of points along x, y axes to calculate error
* Returns: Variance (RMS Error / Sqrt[# points])
* Purpose: Computes error (variance) between the B-spline representation and the currently used
index definition (as defined by grin.mode)
* Requires: bcs parameters are defined and control point array is allocated index function is not B-spline
* Dependencies: index
*****/
double bsplineFitError(bspline *bcs, grinDef *grin, int N) {
double x, y, xs, ys, err, m;
vector R;
surface s;
s.sa = bcs->xf;
xs = (bcs->xf - bcs->x0)/(N-1);ys = (bcs->yf - bcs->y0)/(N-1);
err = 0;
for(x=bcs->x0; x<=0; x+=xs) {
for(y=bcs->y0; y<=bcs->yf; y+=ys) {
R.x = x; R.y = y;aperture(&R, &s);
if(error == NONE) {m = index(grin, &R) - bsp(bcs, &R);m *= m;err += m;}
else RESET_ERROR;}}
err = sqrt(err)/N;
return err;}
/*****
* Name: adaptiveFitBSpline
* Input: bcs : B-spline structure. grin: grin structure. xo : minimum x-position for B-spline
* xf : maximum x-position for B-spline. yo : minimum y-position for B-spline
* yf : maximum y-position for B-spline. tol : maximum error between fit and original index function
* Returns: Variance (RMS Error / Sqrt[# points]) in fit
* Modifies: B-spline parameters (including control point values)
* Purpose: Fit the B-spline to the index function using the minimum number of control points
possible. The spline parameters are set and control points allocated/assigned
* Requires: Index function is not B-spline
* Dependencies: initBSpline, bsplineFitError
*****/
#define MAXIT 25
double adaptiveFitBSpline(bspline *bcs, grinDef *grin, double xo, double xf, double yo, double yf, double tol) {
int i, L, M, num;
double err;
bcs->zc = NULL;
L=0; M=0; i=0; num=11;
do { i++; L++; M++;
printf(" %2d Patches", L);fprintf(outfps[F_DEBUG], " %2d Patches", L);
initBSpline(bcs, L, xo, xf, M, yo, yf);if(bcs->nl > num) num *= 2;
fitBSpline(bcs, grin, num);err = bsplineFitError(bcs, grin, 2*num);
printf(" : RMS Fit Err = %lf\n", err);
fprintf(outfps[F_DEBUG], " : RMS Fit Err = %lf\n", err);} while(err > tol && i < MAXIT);
return err;}
#undef MAXIT
/*****
* Name: reflectBSplineControlPoints
* Input: bcs : B-spline structure
* Modifies: B-spline control point values
* Purpose: Reflect the y<0 control points optimized about the center onto the y>0 control point values
This ensures symmetry in the index profile
* Requires: bcs is allocated and defined
* Dependencies: Nothing
*****/

```

```

void reflectBSplineControlPoints(bspline *bcs) {
    int i, j;
    for(j=bcs->mlo; j<=bcs->mhi; j++) for(i=bcs->llo; i<=bcs->lhi; i++) CP(bcs->nl-(i+1), j) = CP(i,j);
    return;}
/*****
* Name:    updateBSplineControlPoints
* Input:   bcs : B-spline structure. cp : Array of new control point values. nc : Number of new control point
values
* Modifies: B-spline control point values
* Purpose: Update the control points being optimized with the current (modified) values from optimizer.
* Requires: bcs is allocated and defined (and the map bcs->zopt is properly defined)
* Dependencies: Nothing
*****/
void updateBSplineControlPoints(bspline *bcs, double *cp, int nc) {
    int i;
    /*Update Optimized Control Points*/
    for(i=0; i<nc; i++) bcs->zc[bcs->zopt[i]] = cp[i+1]; reflectBSplineControlPoints(bcs);
    return;}
/*****
* Name:    optimizeBSpline
* Input:   sys : System structure. lens: Lens structure. opt : Optimization structure
* Modifies: B-spline control point values. Does not affect lens surface parameters
* Purpose: Optimize the appropriate control points in order to improve the lenses performance optimizer.
* Works best when # x-control points is odd
* Requires: All lens system, and optimization parameters are properly defined
* Dependencies: setTraceRange, allocateStorage, computeDesiredPower, freeStorage,
minimizeMDMerit, reflectControlPoints, meritFunction
*****/
#define EPS 3.0e-8
void optimizeBSpline(sysDef *sys, lensDef *lens, optDef *opt) {
    int l, lmin, lmax, lmid, m, mmin, mmax, i, j, k, o, iter, mw, oflag;
    char cont[2];
    double xo, dx, xmin, xmax, yo, dy, ymin, ymax, sysdAngle, dim[4], dAngle, *pos, **ui, sm, em;
    /*Multi-Dimensional Opt parameters*/
    bspline *bcs;
    bcs = &lens->grin.nbs;
    mw = opt->meritWidth;
    /*Temporarily store the system trace settings*/
    sysdAngle = sys->dAngle; dAngle = sys->dAngle = opt->dAngle;
    for(i=0; i<4; i++) dim[i] = sys->traceDim[i];
    for(i=0; i<4; i++) sys->traceDim[i] = opt->optDim[i];
    setTraceRange(lens, sys);
    allocateStorage(sys); if(error != NONE) return;
    computeDesiredPower(lens, sys);
    opt->startMerit = meritVal();
    opt->endMerit = 0;
    freeStorage(sys);
    /*Count the number of parameters to be optimized; allow space for n(n+1)*/
    opt->n_s1 = 0; opt->n_s2 = 0; opt->n_n = 0;
    /*Determine Patches to be optimized*/
    xo = bcs->x0; dx = bcs->dx; yo = bcs->y0; dy = bcs->dy;
    xmin = -sys->q*tan(M_PI/180.0*opt->optDim[3]); xmax = -sys->q*tan(M_PI/180.0*opt->optDim[2]);
    ymin = sys->q*tan(M_PI/180.0*opt->optDim[0]); ymax = sys->q*tan(M_PI/180.0*opt->optDim[1]);
    lmid = bcs->L/2 - 1;
    lmin = (int)ceil((xmin-xo)/dx-EPS)-0; if(lmin < 0) lmin = 0;
    lmax = (int)ceil((xmax-xo)/dx-EPS)-0; if(lmax >= bcs->L) lmax = bcs->L-1;
    mmin = (int)floor((ymin-yo)/dy); if(mmin < 0) mmin = 0;
    mmax = (int)floor((ymax-yo)/dy); if(mmax >= bcs->M) mmax = bcs->M-1;
    // to allow 1D optimizing in a simple way
    if(lmin > lmax) lmin = lmax;
    if(mmax < mmin) mmax = mmin;
    o=1; oflag = 1;
    do {
        printf("\nOptimization Run %d\n", o);
        fprintf(outFps[F_DEBUG], "\nOptimization Run %d\n", o);
        for(l=lmax; l>=lmin; l--) {
            for(m=mmin; m<=mmax; m++) {
                printf(" Bicubic B-spline Patch (%d,%d)", l, m);
                fprintf(outFps[F_DEBUG], " Bicubic B-spline Patch (%d,%d)", l, m);
                if(l==lmid && m==0) {opt->n_n = 12;}
                else if(l==lmid && m!=0) {opt->n_n = 3;}
                else if(l!=lmid && m==0) {opt->n_n = 4;}
                else {/*l!=lmid && m!=0*/opt->n_n = 1;}
                /*allocate memory for position variable*/
                lens->grin.nbs.zopt = (int *)calloc(opt->n_n, sizeof(int));
                pos = (double *)calloc(opt->n_n + 1, sizeof(double));
                /*allocate memory for 2D basis vector matrix*/
                ui = (double **)calloc(opt->n_n + 1, sizeof(double *));
                for(i=0; i<opt->n_n + 1; i++)
                    ui[i] = (double *)calloc(opt->n_n + 1, sizeof(double));
                /*Set the optimization parameters.
                * Note: all ui[][] terms are the axis vectors of the
                * subspace of the optimization parameter and so they are
                * scaled relative to the starting position pos[]. This gives
                * greater optimization stability.
                */
                k = 1;
                if(l==lmid && m==0) {
                    for(i=0; i<3; i++) {
                        for(j=0; j<4; j++) {

```

```

        lens->grin.nbs.zopt[k-1] = CPidx(1+i,m+j);
        pos[k] = bcs->zc[lens->grin.nbs.zopt[k-1]];
        ui[k][k] = pos[k]/1000;k++;}}
    bcs->llo = 1; bcs->lhi = 1+2;bcs->mlo = m; bcs->mhi = m+3;}
    else if(l!=lmid && m!=0) {
        for(i=0;i<3;i++) {
            lens->grin.nbs.zopt[k-1] = CPidx(1+i,m+3);
            pos[k] = bcs->zc[lens->grin.nbs.zopt[k-1]];
            ui[k][k] = pos[k]/1000;k++;}
        bcs->llo = 1; bcs->lhi = 1+2;bcs->mlo = m+3; bcs->mhi = m+3;}
    else if(l1=lmid && m=0) {
        for(j=0;j<4;j++) {
            lens->grin.nbs.zopt[k-1] = CPidx(1,m+j);
            pos[k] = bcs->zc[lens->grin.nbs.zopt[k-1]];
            ui[k][k] = pos[k]/1000; k++;}
        bcs->llo = 1; bcs->lhi = 1;bcs->mlo = m; bcs->mhi = m+3;}
    else {/*l!=lmid && m!=0*/
        lens->grin.nbs.zopt[k-1] = CPidx(1,m+3);
        pos[k] = bcs->zc[lens->grin.nbs.zopt[k-1]];
        ui[k][k] = pos[k]/1000;k++;
        bcs->llo = 1; bcs->lhi = 1; bcs->mlo = m+3; bcs->mhi = m+3;}
    sys->traceDim[0] = ceil (atan2((m-0-mw)*dy + yo, sys->q)*180/M_PI / dAngle)*dAngle;
    sys->traceDim[1] = floor(atan2((m+1+mw)*dy + yo, sys->q)*180/M_PI / dAngle)*dAngle;
    sys->traceDim[3] = -ceil (atan2((1-0-mw)*dx + xo, sys->q)*180/M_PI / dAngle)*dAngle;
    sys->traceDim[2] = -floor(atan2((1+1+mw)*dx + xo, sys->q)*180/M_PI / dAngle)*dAngle;
    if(sys->traceDim[0]<dim[0]) sys->traceDim[0] = dim[0];
    if(sys->traceDim[1]>dim[1] || sys->traceDim[1] < sys->traceDim[0]) sys->traceDim[1] =
dim[1];
    if(sys->traceDim[2]<dim[2]) sys->traceDim[2] = dim[2];
    if(sys->traceDim[3]>dim[3] || sys->traceDim[3] < sys->traceDim[2]) sys->traceDim[3] =
dim[3];

    if(sys->traceDim[0]>sys->traceDim[1]) sys->traceDim[0] = sys->traceDim[1];
    if(sys->traceDim[1]<sys->traceDim[0]) sys->traceDim[1] = sys->traceDim[0];
    if(sys->traceDim[2]>sys->traceDim[3]) sys->traceDim[2] = sys->traceDim[3];
    if(sys->traceDim[3]<sys->traceDim[2]) sys->traceDim[3] = sys->traceDim[2];
    printf(" %+-5.1f, %+-5.1f; %+-5.1f, %+-5.1f):\n", sys->traceDim[0],sys->traceDim[1],sys-
>traceDim[2],sys->traceDim[3]);
    fprintf(outfps[F_DEBUG], " %+-5.1f, %+-5.1f; %+-5.1f, %+-5.1f):\n", sys->traceDim[0],sys-
>traceDim[1],sys->traceDim[2],sys->traceDim[3]);
    setTraceRange(lens, sys);
    allocateStorage(sys); if(error != NONE) return;
    computeDesiredPower(lens, sys);
    minimizeMDEmerit(pos, ui, opt->n_n, opt->tolMD, opt->tolLD, &iter, &sm, &em, meritFunction);
    opt->endMerit += em*em;
    freeStorage(sys);free(pos);
    for(i=0; i<opt->n_n + 1; i++) free(ui[i]);
    free(ui); free(lens->grin.nbs.zopt);}}
    o++;
    if(opt->opt == O_USER) {
        printf(" Continue? (y/n): ");
        scanf("%s", &cont);
        if(cont[0] == 'n' || cont[0] == 'N') oflag = 0;
        else if(o > opt->opt)oflag = 0;} while(oflag);
    opt->endMerit = sqrt(opt->endMerit);
    sys->dAngle = sysdAngle;
    for(i=0; i<4; i++) sys->traceDim[i] = dim[i];
    return;}
#undef EPS
#undef CP
#undef CPidx

```

G.3. Compute.c

```

/*****
* Name: computeAxes
* Input: lens : Lens defn. struct with the lens info. sys : System defn. struct with the system info
* Modifies: Assigns values to axis system array
* Purpose: Calculates the alpha gaze angle used in the ray trace and assigns the values to storage array
* Requires: lens, sys be allocated & defined
*****/
void computeAxes(lensDef *lens, sysDef *sys) {
    int alphaCount,betaCount,minA,maxA,minB,maxB;
    minA = sys->minAlphaCount;maxA = sys->maxAlphaCount;minB = sys->minBetaCount;maxB = sys->maxBetaCount;
    for(alphaCount = minA; alphaCount <= maxA; alphaCount++) *(sys->alpha + alphaCount-minA) = alphaCount *
sys->dAngle;
    for(betaCount = minB; betaCount <= maxB; betaCount++) *(sys->beta + betaCount-minB) = betaCount * sys-
>dAngle;
    return;}
/*****
* Name: computeDesiredPower
* Input: lens : Lens defn. struct with the lens info. sys : System defn. struct with the system info
* Modifies: Assigns Desired Power to system arrays
* Purpose: Computes Desired Power
* Requires: lens, sys be allocated & defined
* Dependencies: desiredPower
*****/

```

```

void computeDesiredPower(lensDef *lens, sysDef *sys) {
    vector R;
    dirCos U;
    double alpha, beta, L;
    int alphaCount, betaCount, cols, minA, maxA, minB, maxB;
    minA = sys->minAlphaCount; maxA = sys->maxAlphaCount; minB = sys->minBetaCount; maxB = sys->maxBetaCount;
    cols = (maxA - minA + 1);
    /*Compute Desired Power over permitted angles*/
    for(betaCount = minB; betaCount <= maxB; betaCount++) {
        for(alphaCount = minA; alphaCount <= maxA; alphaCount++) {
            alpha = alphaCount * sys->dAngle; beta = betaCount * sys->dAngle;
            /*Place Chief Ray at center of rotation of eye*/
            setVector(0.0, 0.0, sys->q+lens->t, &R);
            /*Rotate exit pupil according to Gaze Direction and find chief ray direction*/
            setDirCos(alpha, beta, 0, DEG+PARTIAL+RELZ, &U); reverseGamma(&U);
            /*Find intersection with back surface of lens*/
            L = intersectSurfaceDist(&R, &U, &lens->surf2);
            propagateSameRay(&R, &U, L); aperture(&R, &lens->surf2);
            if(error == NONE) *(sys->dp + (alphaCount-minA) + (betaCount-
minB)*cols)=desiredPower(alpha,beta);
            else {(sys->dp + (alphaCount-minA) + (betaCount-minB)*cols) = 0; RESET_ERROR}}
        return;}
    /*****
    * Name: computeAst
    * Input: lens : Lens defn. struct with the lens info. sys : System defn. struct with the system info
    * Modifies: Assigns Ast power values to system arrays
    * Purpose: Computes Oblique Astigmatism Error
    * Requires: lens, sys be allocated & defined; Tan, Sag Powers computed
    *****/
void computeAst(lensDef *lens, sysDef *sys) {
    double T, S; /*Infinitesimal Tan & Sag powers (1/mm)*/
    int alphaCount, betaCount, cols, minA, maxA, minB, maxB;
    minA = sys->minAlphaCount; maxA = sys->maxAlphaCount; minB = sys->minBetaCount; maxB = sys->maxBetaCount;
    cols = (maxA - minA + 1);
    /*Compute infinitesimal Oblique Astigmatism Error over permitted angles*/
    for(betaCount = minB; betaCount <= maxB; betaCount++) {
        for(alphaCount = minA; alphaCount <= maxA; alphaCount++) {
            T = *(sys->T + (alphaCount-minA) + (betaCount-minB)*cols);
            S = *(sys->S + (alphaCount-minA) + (betaCount-minB)*cols);
            *(sys->A + (alphaCount-minA) + (betaCount-minB)*cols) = T-S;}}
    return;}
    /*****
    * Name: computeMOE
    * Input: lens : Lens defn. struct with the lens info. sys : System defn. struct with the system info
    * Modifies: Assigns Mean Oblique (Power) error to system arrays
    * Purpose: Computes Mean Oblique (Power) Error
    * Requires: lens, sys be allocated & defined; Tan, Sag, Desired Powers computed
    *****/
void computeMOE(lensDef *lens, sysDef *sys) {
    double T, S, dp; /*Infinitesimal Tan & Sag powers (1/mm). Desired Power at each gaze position*/
    int alphaCount, betaCount, cols, minA, maxA, minB, maxB;
    minA = sys->minAlphaCount; maxA = sys->maxAlphaCount; minB = sys->minBetaCount; maxB = sys->maxBetaCount;
    cols = (maxA - minA + 1);
    /*Compute infinitesimal Mean Oblique Error over permitted angles*/
    for(betaCount = minB; betaCount <= maxB; betaCount++) {
        for(alphaCount = minA; alphaCount <= maxA; alphaCount++) {
            T = *(sys->T + (alphaCount-minA) + (betaCount-minB)*cols);
            S = *(sys->S + (alphaCount-minA) + (betaCount-minB)*cols);
            dp = *(sys->dp + (alphaCount-minA) + (betaCount-minB)*cols);
            *(sys->M + (alphaCount-minA) + (betaCount-minB)*cols) = (T+S)/2-dp;}}
    return;}
    /*****
    * Name: computeDistortion
    * Input: lens : Lens defn. struct with the lens info. sys : System defn. struct with the system info
    * Modifies: Assigns Distortion values to system arrays
    * Purpose: Computes distortion values and assigns them to the system array (%/100 ratio values)
    * Requires: lens, sys be allocated & defined
    * Dependencies: desiredPower
    *****/
void computeDistortion(lensDef *lens, sysDef *sys) {
    double alpha, beta, nz; /*Gaze Angles. Z-Position of Back Nodal Point*/
    int alphaCount, betaCount, cols, minA, maxA, minB, maxB;
    planeSurf image; /*Image plane for computing distortion*/
    vector R; /*Back Nodal point pos for distortion*/
    dirCos U; /*Back Nodal point dir for distortion*/
    minA = sys->minAlphaCount; maxA = sys->maxAlphaCount; minB = sys->minBetaCount; maxB = sys->maxBetaCount;
    cols = (maxA - minA + 1);
    /*Calculate Back Nodal Point relative to origin; using homogeneous lens equ*/
    nz = backNodalPoint(lens);
    /*Compute infinitesimal Oblique Astigmatism Error over permitted angles*/
    for(betaCount = minB; betaCount <= maxB; betaCount++) {
        for(alphaCount = minA; alphaCount <= maxA; alphaCount++) {
            double ih, oh, dist; /*image height & object height & distortion value*/
            alpha = alphaCount * sys->dAngle; beta = betaCount * sys->dAngle;
            /*Set Image Plane at desired power position - allows for progressive power*/
            setPlaneSurface(0,0,1./desiredPower(alpha, beta) + lens->t, 0,0,1, &image);
            /*Real image height*/
            setVector(0,0,lens->t + sys->q, &R);
            setDirCos(alpha, beta, 0, DEG+PARTIAL+RELZ, &U);
            reverseGamma(&U);

```

```

propagateSameRay(&R, &U, intersectPlaneDist(&R, &U, &image));
ih = magnitudeZPlane(&R);
/*Paraxial image height*/
setVector(0,0,nz, &R);
setDirCos(0,*(sys->D + (alphaCount-minA) + (betaCount-minB)*cols),0, DC+PARTIAL, &U);
propagateSameRay(&R, &U, intersectPlaneDist(&R, &U, &image));
oh = magnitudeZPlane(&R);
/*Calculate Distortion*/
dist = (ih-oh)/oh;
if(dist*2 == dist) *(sys->D + (alphaCount-minA) + (betaCount-minB)*cols) = 0.;
else *(sys->D + (alphaCount-minA) + (betaCount-minB)*cols) = dist; /*end of Loop over alpha*/
} /*end of Loop over beta*/
return;}
/*****
* Name: intersectLineDist
* Input: R1 : Ray 1 position. U1 : Ray 1 direction. R2 : Ray 2 position. U2 : Ray 2 direction
* a : distance along Ray 1 to point of closest approach. b : distance along Ray 2 to point of closest
approach
* t : minimum distance between lines
* Modifies: a, b, t
* Purpose: Finds point of closest approach of the two lines defined by (R1,U1) & (R2,U2).
* Requires: R1, U1, R2, U2 are allocated and defined
*****/
void intersectLineDist(vector *R1, dirCos *U1, vector *R2, dirCos *U2, double *a, double *b, double *t) {
dirCos *norm;
double den, Nx, Ny, Nz;
double R1x, R1y, R1z, U1x, U1y, U1z,R2x, R2y, R2z, U2x, U2y, U2z,a11, a12, a13, a21, a22, a23, a31, a32,
a33;
norm = crossProduct(U1, U2);Nx = norm->alpha;Ny = norm->beta;Nz = norm->gamma;free(norm);
R1x = R1->x;R1y = R1->y;R1z = R1->z;U1x = U1->alpha;U1y = U1->beta;U1z = U1->gamma;
R2x = R2->x;R2y = R2->y;R2z = R2->z;U2x = U2->alpha;U2y = U2->beta;U2z = U2->gamma;
den = (Nz*U1y*U2x - Ny*U1z*U2x - Nz*U1x*U2y + Nx*U1z*U2y + Ny*U1x*U2z - Nx*U1y*U2z);
a11 = Nz*U2y - Ny*U2z;a12 = -Nz*U2x + Nx*U2z;a13 = Ny*U2x - Nx*U2y;
a21 = Nz*U1y - Ny*U1z;a22 = -Nz*U1x + Nx*U1z;a23 = Ny*U1x - Nx*U1y;
a31 = U1z*U2y - U1y*U2z;a32 = -U1z*U2x + U1x*U2z;a33 = U1y*U2x - U1x*U2y;
*a = a11*(R1x - R2x) + a12*(R1y - R2y) + a13*(R1z - R2z);
*b = a21*(R1x - R2x) + a22*(R1y - R2y) + a23*(R1z - R2z);
*t = a31*(R1x - R2x) + a32*(R1y - R2y) + a33*(R1z - R2z);
*a /= den;*b /= den;*t /= den;
return;}
/*****
* Name: intersectPlaneDist
* Input: R : Ray position. U : Ray direction. s : plane to intersect with
* Returns: Distance along Ray to intersection with surface
* Purpose: Determine the distance to intersection of the ray with the surface
* Requires: R, U allocated & defined
*****/
double intersectPlaneDist(vector *R, dirCos *U, planeSurf *surf) {
double dist;
dist = surf->mx*(surf->xi - R->x);dist += surf->my*(surf->yi - R->y);dist += surf->mz*(surf->zi - R->z);
dist /= (surf->mx*U->alpha + surf->my*U->beta + surf->mz*U->gamma);
return dist;}
/*****
* Name: intersectSphereDist
* Input: R : Ray position. U : Ray direction. s : plane to intersect with
* Returns: Distance along Ray to intersection with surface
* Purpose: Determine the distance to intersection of the ray with the surface
* Requires: R, U allocated & defined. surf->a[0] must be defined (surf->na == 0 not valid)
*****/
#define NRANSI
#define ITMAX 100
#define EPS 3.0e-8
double intersectSphereDist(vector *R, dirCos *U, sphereSurf *surf) {
double dist, RdotU, Rsq, Uz, Rz, b, c, d, k, num, den;
vector Rtemp;
c = 1.0/surf->radius;
k = surf->a[0];
Rtemp.x = R->x;Rtemp.y = R->y;Rtemp.z = R->z - (surf->origin.z - surf->radius); /*so sphere is at coord
origin*/
Rz = Rtemp.z;RdotU = dotProduct((dirCosPtr)&Rtemp, U);Rsq = magnitudeSquared(&Rtemp);Uz = U->gamma;
/*solution that allows small values of c*/
num = c*Rsq - 2.0*Rz + c*k*Rz*Rz;
b = 1.0 - c*k*Rz - c/Uz*RdotU;d = b*b - c*(k + 1.0/(Uz*Uz))*(c*Rsq - 2.0*Rz + c*k*Rz*Rz);
if(d < 0) {error = NAN;return INFINITY;}
den = Uz*((1.0 - c*k*Rz - c/Uz*RdotU) + sqrt(d));
dist = num/den;
if(surf->na > 1) { /*Aspheric Terms*/
int iter;
double a, b, c, d, e, min1, min2, tol=1e-8,fa, fb, fc, p, q, r, s, toll, xm;
a=b=c=dist;
do{ a *= 0.9;b *= 1.1;c = b;
fa = sphereSurfaceCondition(R, U, surf, a);
fb = sphereSurfaceCondition(R, U, surf, b);}while((fa > 0.0 && fb > 0.0) || (fa<0.0 && fb<0.0));
fc=fb;
for (iter=1;iter<=ITMAX;iter++) {
if ((fb > 0.0 && fc > 0.0) || (fb < 0.0 && fc < 0.0)) {c=a;fc=fa;e=d=b-a;}
if (fabs(fc) < fabs(fb)) {a=b;b=c;c=a;fa=fb;fb=fc;fc=fa;}
toll=2.0*EPS*fabs(b)+0.5*tol;
xm=0.5*(c-b);
if (fabs(xm) <= toll || fb == 0.0) return b;
}
}
}

```

```

    if (fabs(e) >= toll && fabs(fa) > fabs(fb)) {
        s=fb/fa;
        if (a == c) {p=2.0*xm*s;q=1.0-s;}
        else {q=fa/fc;r=fb/fc;p=s*(2.0*xm*q*(q-r)-(b-a)*(r-1.0));q=(q-1.0)*(r-1.0)*(s-1.0);}
        if (p > 0.0) q = -q;
        p=fabs(p);min1=3.0*xm*q-fabs(toll*q);
        min2=fabs(e*q);
        if (2.0*p < (min1 < min2 ? min1 : min2)) {e=d;d=p/q;}
        else {d=xm;e=d;}
        a=b;fa=fb;
        if (fabs(d) > toll) b += d;
        else b += SIGN2(toll,xm);
        fb = sphereSurfaceCondition(R, U, surf, b);}
    error = NUMERICAL;
    return INFINITY;}
return dist;}
#undef EPS
#undef ITMAX
#undef NRANSI
/*****
* Name: computeRadius1ForZeroAst1
* Input: n : index of refraction. objectdist : object distance. phi : lens power
* stop : stop distance from back surface (not currently implemented)
* Returns: Radius of front surface which gives a zero astigmatism solution
* Purpose: Calculate the Ostwalt solution (in reference to Tscherning Ellipses - the lower surface
* power solutions) for zero astigmatism for a thin lens
* Requires: Nothing
*****/
double computeRadius1ForZeroAst1(double n, double stop, double objectDist, double phi) {
    double a, b, b1, b2, b3, c, q, q2, n2, d, d2, phil;
    q = stop;q2 = q*q;n2 = n*n; d = objectDist;d2 = d*d; a = 2*(-1 + n2)*q + (2 + n)*q2*phi;
    b1 = -1 + n + q*phi; b1*= b1;b2 = -2 + 2*n2 + 2*q*phi + n*q*phi; b2*=b2;b3 = -4*n*(2+n)*b1 + b2;
    if(b3 < 0) {error = NAN;return INFINITY;}
    b = sqrt(q2*b3); c = (2.*(2. + n)*q2); phil = (a-b)/c;
    if(phil == 0) {error = NUMERICAL;return INFINITY;}
    else return (n-1)/phil;}
/*****
* Name: computeRadius1ForZeroAst2
* Input: n : index of refraction. objectdist : object distance. phi : lens power
* stop : stop distance from back surface (not currently implemented)
* Returns: Radius of front surface which gives a zero astigmatism solution
* Purpose: Calculate the Wollaston solution (in reference to Tscherning Ellipses - the higher
* surface power solutions) for zero astigmatism for a thin lens
* Requires: Nothing
*****/
double computeRadius1ForZeroAst2(double n, double stop, double objectDist, double phi) {
    double a, b, b1, b2, c, q, q2, n2, d, d2, phil;
    q = stop;q2 = q*q;n2 = n*n;d = objectDist;d2 = d*d;a = 2*(-1 + n2)*q + (2 + n)*q2*phi;
    b1 = -1 + n + q*phi; b1*= b1;b2 = -2 + 2*n2 + 2*q*phi + n*q*phi; b2*=b2;b = q2*(-4*n*(2+n)*b1 + b2);
    if(b < 0) {error = NAN;return INFINITY;}
    else b = sqrt(b);
    c = (2.*(2. + n)*q2);phil = (a+b)/c;
    return (n-1)/phil;}
return;}

```

G.4. Grin.c

```

/*****
* Name: homogeneous
* Input: a : homogeneous structure. b : placeholder for position to evaluate Polynomial
* Returns: Homogeneous value
* Purpose: Return Homogeneous value
* Requires: homog (a) is allocated and defined.
*****/
#define homogeneous(a,b) ((a)->n)
/*****
* Name: diffHomogX
* Input: a : homogeneous structure. b : placeholder for position to evaluate Polynomial
* Returns: Derivative of Homogeneous wrt x
* Purpose: Return dn/dx value
* Requires: Nothing
*****/
#define diffHomogX(a,b) (0)
/*****
* Name: diffHomogY
* Input: a : homogeneous structure. b : placeholder for position to evaluate Polynomial
* Returns: Derivative of Homogeneous wrt y
* Purpose: Return dn/dy value
* Requires: Nothing
*****/
#define diffHomogY(a,b) (0)
/*****
* Name: sech
* Input: x : evaluation point
* Returns: value of sech(x)
* Purpose: Sech() evaluation

```

```

* Requires: Nothing
*****/
double sech(double x) {
    return 2.0 / (exp(x) + exp(-x));
}
int numIndexMeritSpaceParams(grinDef *g) {
    int i, count;
    switch(g->mode) {
        case G_R : /*Radial Polynomial Gradient*/
        case G_RSQ : /*Radial r-Squared Polynomial Gradient*/
            count = 0; for(i=0; i<g->np.ny; i++) if(*(g->np.zmerit[2]+i)) count++; break;
        case G_H : case G_XY : case G_BSP: count = 0; break;
    }
    return count;
}
int numIndexParam(grinDef *g) {
    int count;
    switch(g->mode) {
        case G_H : count = 1; break;
        case G_R : case G_RSQ : case G_XY : count = g->np.nx * g->np.ny; break;
        case G_BSP: count = g->nbs.nl * g->nbs.nm; break;
    }
    return count;
}
/*****/
* Name: index
* Input: R : Position vector. g : GRIN information
* Returns: Index of refraction at R
* Purpose: Compute index of refraction within the lens, the method of which depends on the grin mode.
* mode = G_R : Calculate as a polynomial of r. mode = G_RSQ : Calculate as a polynomial of r^2
* Requires: g be allocated & defined; g->n are coeffs of poly description of n(r) grin mode must be G_R or G_RSQ
*****/
double index(grinDef *g, vector *R) {
    double index;
    switch(g->mode) {
        case G_H : /*Homogeneous Gradient*/index = homogeneous(&g->nh, R);break;
        case G_R : /*Radial Polynomial Gradient*/index = radial(&g->np, R);break;
        case G_RSQ : /*Radial r-Squared Polynomial Gradient*/index = radialSquared(&g->np, R);break;
        case G_XY : /*Cartesian (XY) Polynomial Gradient*/index = polyXY(&g->np, R);break;
        case G_BSP:index = bsp(&(g->nbs), R);break;
    }
    return index;
}
/*****/
* Name: diffIndexX
* Input: R : Position vector. g : GRIN information
* Returns: X-derivative of index of refraction at R
* Purpose: Compute x-derivative of index of refraction within the lens, the method of which depends on the grin mode
* mode = G_R : Calculate as a polynomial of r. mode = G_RSQ : Calculate as a polynomial of r^2
* Requires: g be allocated & defined; g->n are coeffs of polynomial description of n(r)
*****/
double diffIndexX(grinDef *g, vector *R) {
    double deriv;
    switch(g->mode) {
        case G_H : /*Homogeneous Gradient*/deriv = diffHomogX(&g->nh, R);break;
        case G_R : /*Radial Polynomial Gradient*/deriv = diffRadialX(&(g->np), R);break;
        case G_RSQ : /*Radial r-Squared Polynomial Gradient*/deriv = diffRadialSquaredX(&(g->np), R);break;
        case G_XY : /*Cartesian (XY) Polynomial Gradient*/deriv = diffPolyXYX(&(g->np), R);break;
        case G_BSP:/*Bicubic B-spline Gradient*/deriv = diffBspX(&(g->nbs), R);break;
        default :deriv = 0; break;}
    return deriv;
}
/*****/
* Name: diffIndexY
* Input: R : Position vector. g : GRIN information
* Returns: Y-derivative of index of refraction at R
* Purpose: Compute y-derivative of index of refraction within the lens, the method of which depends on the grin mode
* mode = G_R : Calculate as a polynomial of r. mode = G_RSQ : Calculate as a polynomial of r^2
*****/
double diffIndexY(grinDef *g, vector *R) {
    double deriv;
    switch(g->mode) {
        case G_H : /*Homogeneous Gradient*/deriv = diffHomogY(&g->nh, R);break;
        case G_R : /*Radial Polynomial Gradient*/deriv = diffRadialY(&(g->np), R);break;
        case G_RSQ : /*Radial r-Squared Polynomial Gradient*/deriv = diffRadialSquaredY(&(g->np), R);break;
        case G_XY : /*Cartesian (XY) Polynomial Gradient*/deriv = diffPolyXYX(&(g->np), R);break;
        case G_BSP:/*Bicubic B-spline Gradient*/deriv = diffBspY(&(g->nbs), R);break;
        default :deriv = 0; break;}
    return deriv;
}
/*****/
* Name: diffIndexXX
* Input: R : Position vector. g : GRIN information
* Returns: X-2nd-derivative of index of refraction at R
* Purpose: Compute x-2nd-derivative of refractive index in the lens, the method of which depends on the grin mode
* mode = G_R : Calculate as a polynomial of r. mode = G_RSQ : Calculate as a polynomial of r^2
* Requires: g be allocated & defined; g->n are coeffs of polynomial description of n(r)
*****/
double diffIndexXX(grinDef *g, vector *R) {
    double deriv;
    deriv = 0.0;
    switch(g->mode) {case G_BSP: deriv = diffBspXX(&(g->nbs), R);break;}
    return deriv;
}
/*****/
* Name: diffIndexYY
* Input: R : Position vector. g : GRIN information

```

```

* Returns: Y-2nd-derivative of index of refraction at R
* Purpose: Compute y-2nd-derivative of index within the lens, the method of which depends on the grin mode
* mode = G_R : Calculate as a polynomial of r. mode = G_RSQ : Calculate as a polynomial of r^2
* Requires: g be allocated & defined; g->n are coeffs of polynomial description of n(r)
*****
double diffIndexYY(grinDef *g, vector *R) {
    double deriv;deriv = 0.0;switch(g->mode) {case G_BSP:deriv = diffBspYY(&(g->nbs), R);break;}return deriv;}
/*****
* Name: diffIndexZ
* Input: R : Position vector. g : GRIN information
* Returns: Z-derivative of index of refraction at R
* Purpose: Compute z-derivative of index of refraction within the lens
* Requires: g be allocated & defined; g->n are coeffs of polynomial description of n(r)
* r^2 = (x^2 + y^2) (no z-dependence)
*****
double diffIndexZ(grinDef *g, vector *R) {return 0.0;}
/*****
* Name: stepRay
* Input: R : Position vector. T : Direction cosine vector. l : Lens information. dt : optical path step size
* Nm : n(r)^2 at new position. Sm : Optical path distance from previous to new position
* Modifies: R : modified for new position. T : modified for new direction cosines. Nm : modified for new n(r)^2
* Sm : modified for new Optical path distance (from previous to new position)
* Purpose: Compute next position in ray-trace through GRIN media according to Sharma's method
* Requires: R, T, l be allocated & defined; index is defined; 2D GRIN profile
* Dependencies: diffIndexX, diffIndexY
*****
void stepRay(vector *R, dirCos *T, lensDef *l, double dt, double *Nm, double *Sm) {
    vector A, B, C, D, Rtmp;
    double n, a, b, g;
    a = T->alpha;b = T->beta;g = T->gamma;
    Rtmp.x = R->x;Rtmp.y = R->y;Rtmp.z = R->z;
    n = index(&l->grin, &Rtmp);
    A.x = dt*n*diffIndexX(&l->grin, &Rtmp);A.y = dt*n*diffIndexY(&l->grin, &Rtmp);
    Rtmp.x = R->x + dt/2.0 * a + 0.125*dt*A.x;Rtmp.y = R->y + dt/2.0 * b + 0.125*dt*A.y;Rtmp.z = R->z + dt/2.0 *
g;
    n = index(&l->grin, &Rtmp);
    B.x = dt*n*diffIndexX(&l->grin, &Rtmp);B.y = dt*n*diffIndexY(&l->grin, &Rtmp);
    Rtmp.x = R->x + dt*a + 0.5*dt*B.x;Rtmp.y = R->y + dt*b + 0.5*dt*B.y;Rtmp.z = R->z + dt*g;
    n = index(&l->grin, &Rtmp);
    C.x = dt*n*diffIndexX(&l->grin, &Rtmp);C.y = dt*n*diffIndexY(&l->grin, &Rtmp);
    R->x += dt*(a + 1.0*(A.x + 2.0*B.x)/6.0);->y += dt*(T->beta + 1.0*(A.y + 2.0*B.y)/6.0);R->z += dt*(T->
>gamma);
    T->alpha += 1.0*(A.x + 4.0*B.x + C.x)/6.0; /*meridional plane axis direction*/
    T->beta += 1.0*(A.y + 4.0*B.y + C.y)/6.0; /*skew plane axis direction*/
    D.x = diffIndexX(&l->grin, R);D.y = diffIndexY(&l->grin, R);
    *Nm = index(&l->grin, R);*Sm = 2*(Nm)*(D.x*T->alpha + D.y*T->beta);*Nm *= *Nm;
    return;}
/*****
* Name: stepRay3D
* Input: R : Position vector. T : Direction cosine vector. l : Lens information. dt : optical path step size
* Nm : n(r)^2 at new position. Sm : Optical path distance from previous to new position
* Modifies: R : modified for new position. T : modified for new direction cosines. Nm : modified for new n(r)^2
* Sm : modified for new Optical path distance (from previous to new position)
* Purpose: Compute next position in ray-trace through GRIN media according to Sharma's method
* Requires: R, T, l be allocated & defined; index is defined
* Dependencies: diffIndexX, diffIndexY, diffIndexZ
*****
void stepRay3D(vector *R, dirCos *T, lensDef *l, double dt, double *Nm, double *Sm) {
    vector A, B, C, D, Rtmp;
    Rtmp.x = R->x; Rtmp.y = R->y; Rtmp.z = R->z;
    A.x = dt*index(&l->grin, &Rtmp)*diffIndexX(&l->grin, &Rtmp);
    A.y = dt*index(&l->grin, &Rtmp)*diffIndexY(&l->grin, &Rtmp);
    A.z = dt*index(&l->grin, &Rtmp)*diffIndexZ(&l->grin, &Rtmp);
    Rtmp.x = R->x + dt/2.0 * T->alpha + 0.125*dt*A.x;
    Rtmp.y = R->y + dt/2.0 * T->beta + 0.125*dt*A.y;
    Rtmp.z = R->z + dt/2.0 * T->gamma + 0.125*dt*A.z;
    B.x = dt*index(&l->grin, &Rtmp)*diffIndexX(&l->grin, &Rtmp);
    B.y = dt*index(&l->grin, &Rtmp)*diffIndexY(&l->grin, &Rtmp);
    B.z = dt*index(&l->grin, &Rtmp)*diffIndexZ(&l->grin, &Rtmp);
    Rtmp.x = R->x + dt*T->alpha + 0.5*dt*B.x;
    Rtmp.y = R->y + dt*T->beta + 0.5*dt*B.y;
    Rtmp.z = R->z + dt*T->gamma + 0.5*dt*B.z;
    C.x = dt*index(&l->grin, &Rtmp)*diffIndexX(&l->grin, &Rtmp);
    C.y = dt*index(&l->grin, &Rtmp)*diffIndexY(&l->grin, &Rtmp);
    C.z = dt*index(&l->grin, &Rtmp)*diffIndexZ(&l->grin, &Rtmp);
    R->x += dt*(T->alpha + 1.0*(A.x + 2.0*B.x)/6.0);
    R->y += dt*(T->beta + 1.0*(A.y + 2.0*B.y)/6.0);
    R->z += dt*(T->gamma + 1.0*(A.z + 2.0*B.z)/6.0);
    T->alpha += 1.0*(A.x + 4.0*B.x + C.x)/6.0; /*meridional plane axis direction*/
    T->beta += 1.0*(A.y + 4.0*B.y + C.y)/6.0; /*skew plane axis direction*/
    T->gamma += 1.0*(A.z + 4.0*B.z + C.z)/6.0; /*optical axis direction*/
    D.x = diffIndexX(&l->grin, R);D.y = diffIndexY(&l->grin, R);D.z = diffIndexZ(&l->grin, R);
    *Nm = index(&l->grin, R);*Sm = 2*(Nm)*(D.x*T->alpha + D.y*T->beta + D.z*T->gamma);*Nm *= *Nm;
    return;}

```

G.5. Main.c

```

#include <time.h>
/*****
* Name:    allocateStorage
* Input:  sys : system parameters
* Modifies: sys->axis, od, dp, T, S, A, P, D, iA, iP
* Purpose: Allocates memory for the system value arrays and sets all values to zero (0).
*         If memory can't be allocated error = MEMORY
*         The amount of memory is rows*cols where
*         cols = sys->minAlphaCount + sys->maxAlphaCount + 1
*         rows = sys->minBetaCount + sys->maxBetaCount + 1
* Requires: sys->minAlphaCount, maxAlphaCount, minBetaCount, maxBetaCount are set
*****/
void allocateStorage(sysDef *sys) {
    int rows, cols;
    cols = (sys->maxAlphaCount - sys->minAlphaCount + 1); rows = (sys->maxBetaCount - sys->minBetaCount + 1);
    if((sys->alpha = (double *)calloc(sizeof(double), cols)) == NULL) {error = MEMORY;return;}
    if((sys->beta = (double *)calloc(sizeof(double), rows)) == NULL) {error = MEMORY;return;}
    if((sys->od = (double *)calloc(sizeof(double), cols)) == NULL) {error = MEMORY;return;}
    if((sys->dp = (double *)calloc(sizeof(double), rows*cols)) == NULL) {error = MEMORY;return;}
    if((sys->T = (double *)calloc(sizeof(double), rows*cols)) == NULL) {error = MEMORY;return;}
    if((sys->S = (double *)calloc(sizeof(double), rows*cols)) == NULL) {error = MEMORY;return;}
    if((sys->A = (double *)calloc(sizeof(double), rows*cols)) == NULL) {error = MEMORY;return;}
    if((sys->M = (double *)calloc(sizeof(double), rows*cols)) == NULL) {error = MEMORY;return;}
    if((sys->D = (double *)calloc(sizeof(double), rows*cols)) == NULL) {error = MEMORY;return;}
    if((sys->iP = (double *)calloc(sizeof(double), rows*cols)) == NULL) {error = MEMORY;return;}
    if((sys->iA = (double *)calloc(sizeof(double), rows*cols)) == NULL) {error = MEMORY;return;}
    if((sys->iM = (double *)calloc(sizeof(double), rows*cols)) == NULL) {error = MEMORY;return;}
    return;}
/*****
* Name:    nullStorage
* Input:  sys : system parameters
* Modifies: sys->axis, od, dp, T, S, A, P, D, iA, iP
* Purpose: Sets system variables to NULL
* Requires: Nothin
*****/
void nullStorage(sysDef *sys) {
    sys->alpha = NULL;sys->beta = NULL;sys->od = NULL;sys->dp = NULL;sys->T = NULL;sys->S = NULL;sys->A = NULL;
    sys->M = NULL;sys->D = NULL;sys->iP = NULL;sys->iA = NULL;sys->iM = NULL;
    return;}
/*****
* Name:    freeStorage
* Input:  sys : system parameters
* Modifies: sys->axis, od, dp, T, S, A, P, D, iA, iP
* Purpose: Frees memory allocated by allocateStorage for the system value arrays. Checks to
*         see if they are NULL first. Variables set to NULL
* Requires: All variables must be NULL if they were not allocated
*****/
void freeStorage(sysDef *sys) {
    if(sys->alpha != NULL){free(sys->alpha); sys->alpha = NULL;}if(sys->beta != NULL){free(sys->beta); sys->beta = NULL;}
    if(sys->od != NULL) {free(sys->od); sys->od = NULL;}if(sys->dp != NULL) {free(sys->dp); sys->dp = NULL;}
    if(sys->T != NULL) {free(sys->T); sys->T = NULL;}if(sys->S != NULL) {free(sys->S); sys->S = NULL;}
    if(sys->A != NULL) {free(sys->A); sys->A = NULL;}if(sys->M != NULL) {free(sys->M); sys->M = NULL;}
    if(sys->D != NULL) {free(sys->D); sys->D = NULL;}if(sys->iP != NULL) {free(sys->iP); sys->iP = NULL;}
    if(sys->iA != NULL) {free(sys->iA); sys->iA = NULL;}if(sys->iM != NULL) {free(sys->iM); sys->iM = NULL;}
    nullStorage(sys);
    return;}
/*****
* Name:    setLensThickness
* Input:  lens : Lens information
*         t : thickness
* Purpose: set given lens values
* Requires: Nothing
*****/
void setLensThickness(lensDef *lens, double t) {lens->t = t;return;}
int palTrace(int argc, char *argv[]) {
    sysDef sys; /*System Parameters*/
    optDef opt; /*Optimization Parameters*/
    lensDef lens; /*Lens Parameters*/
    clock_t start, stop; /*Program start & stop time*/
    nullStorage(&sys);
    /*Read lens & system data*/
    readLensParam(&lens, &sys, &opt, &argc, argv);
    while (error == IO) {printf(" Lens info not found.\n\n");RESET_ERROR readLensParam(&lens, &sys, &opt,
    &argc, argv);}
    if(error != NONE) {printError(stdout);freeStorage(&sys);closeFiles(outfps);}
    /*Open the output files*/
    openFiles(outfps, sys.prefix);
    if(error != NONE) {printError(stdout);return error;closeFiles(outfps);}
    /*fit Spline to index polynomial*/
    if(lens.grin.mode >= G_BSPFIT) { /*fit B-spline*/
        double minX, maxX, minY, maxY;
        printf("Constructing B-spline:\n");fprintf(outfps[F_DEBUG], "Constructing B-spline:\n");
        lens.grin.nbs.zc = NULL;
    }
}

```

```

minX = -lens.surfl.sa;minY = -lens.surfl.sa;maxX = +lens.surfl.sa;maxY = +lens.surfl.sa;
lens.grin.mode -= G_BSPFIT;
adaptiveFitBSpline(&(lens.grin.nbs), &(lens.grin), minX, maxX, minY, maxY, 1e-4);
lens.grin.mode = G_BSP;}
/*Start timer*/
start = clock();
setOptimizeGlobals(&sys, &lens, &opt);
if(opt.merit != O_NONE) {
    printf("Computing Merit Space:\n");
    fprintf(outfps[F_DEBUG], "Computing Merit Space:\n");
    meritSpaceFunction(outfps, &sys, &lens);}
if(error != NONE) {printError(stdout);printError(outfps[F_DEBUG]);closeFiles(outfps);return error;}
/*Optimize the Lens*/
if(opt.opt != O_NONE) {
    printf("Optimizing:\n");fprintf(outfps[F_DEBUG], "Optimizing:\n");optimize();}
if(error != NONE) {
    printError(stdout);printError(outfps[F_DEBUG]);closeFiles(outfps);
    return error;}
/*RayTrace the Lens*/
if(sys.trace) {
    printf("Tracing Lens:\n");fprintf(outfps[F_DEBUG], "Tracing Lens:\n");
    setTraceRange(&lens, &sys);allocateStorage(&sys);
    if(error != NONE) {printError(stdout);printError(outfps[F_DEBUG]);closeFiles(outfps);return error;}
    traceLens(&lens, &sys, outfps[F_DEBUG]);
    printf("Computing Aberrations:\n");fprintf(outfps[F_DEBUG], "Computing Aberrations:\n");
    computeDesiredPower(&lens, &sys);computeAst(&lens, &sys);computeMOE(&lens, &sys);
    computeDistortion(&lens, &sys);computeAxes(&lens, &sys);setObjectDistances(&lens, &sys);}
/*Stop timer*/
stop = clock();
/*Compute run-time*/
sys.time = ((double)(stop - start))/CLOCKS_PER_SEC;
/*Write Lens Information*/
printf("Writing Results:\n");
fprintf(outfps[F_DEBUG], "Writing Results:\n");
writeLensParam(outfps[F_SPECS], &lens, &sys, &opt);
if(sys.trace) writeResults(outfps, &lens, &sys);
freeStorage(&sys);
/*Close the output files*/
closeFiles(outfps);
/*Quit. Go home and eat popcorn.*/
return error;}
/*****
* Name: main
* Input: argc : Number of command line arguments
*         argv : command line arguments. If argc > 1, argv[1] should be the input filename
* Purpose: Controls the PalTrace program
* Requires: Nothing
* Dependencies: readLensData, openFiles, computeMeritSpace, optimize, traceLens, printSpecs, closeFiles
*****/
void main(int argc, char *argv[]) {palTrace(argc, argv);return;} /*end of main*/

```

G.6. Matrix.c

```

void transpose(double *A, double *At, int row, int col) {
    int i, j;for(i=0; i<row; i++)for(j=0; j<col; j++){At+j*row+i} = *(A+i*col+j);}
void multiply(double *A, int ar, int ac, double *B, int br, int bc, double *C) {
    int i, j, k;
    double *t;
    if(ac != br) return;
    t = (double *)calloc(ar*bc, sizeof(double));
    if(t == (double*)NULL) exit(-1);
    for(i=0; i<ar; i++){for(j=0; j<bc; j++){*(t+i*bc+j) = 0; for(k=0; k<ac; k++){(t+i*bc+j) += *(A+i*ac+k) *
*(B+k*bc+j);}}
    for(i=0; i<ar; i++) for(j=0; j<bc; j++) *(C+i*bc+j) = *(t+i*bc+j);
    free(t);}
void nrerror(char error_text[]) {
    fprintf(stderr, "Numerical Recipes run-time error...\n");fprintf(stderr, "%s\n", error_text); return;}
#define NR_END 1
#define FREE_ARG char
double *array(int nl, int nh){
    double *v;
    v = (double *)malloc((nh-nl+1+NR_END)*sizeof(double));
    if (!v) nrerror("allocation failure in array()");
    return v-nl+NR_END;}
void free_array(double *v, long nl, long nh) {free((FREE_ARG) (v+n1-NR_END));}
#undef NR_END
#undef FREE_ARG
#define NR_END 0
int *ivector(long nl, long nh) { /*allocate an int vector with subscript range v[nl..nh] */
    int *v;
    v=(int *)calloc(nh-nl+1+NR_END, sizeof(int));
    if (!v) nrerror("allocation failure in ivector()");
    return v-nl+NR_END;}
void free_ivector(int *v, long nl) { /*free an int vector allocated with ivector() */free(v+n1-NR_END);}
#undef NR_END
#define SWAP(a,b) {temp=(a);(a)=(b);(b)=temp;}

```

```

void gaussj(double *a, int n, double *b, int m) {
    int *indx,*indxr,*ipiv;
    int i,icol,irow,j,k,l,ll;
    double big,dum,pivinv,temp;
    indx=ivector(0,n);
    indxr=ivector(0,n);
    ipiv=ivector(0,n);
    for (j=0;j<n;j++) ipiv[j]=0;
    for (i=0;i<n;i++) {big=0.0;for (j=0;j<n;j++) if (ipiv[j] != 1) for (k=0;k<n;k++) {if (ipiv[k] == 0) {if
(fabs(*(a+j*n+k)) >= big) {big = fabs(*(a+j*n+k));irow = j;icol = k;}}else if (ipiv[k] > 1) nrerror("gaussj:
Singular Matrix-1");}
        ++(ipiv[icol]);
        if (irow != icol) {for (l=0; l<n; l++) SWAP(*(a+irow*n+l), *(a+icol*n+l));for (l=0; l<m; l++)
SWAP(*(b+irow*m+l), *(b+icol*m+l))}
        indxr[i]=irow;indx[i]=icol;
        if (*(a+icol*n+icol) == 0.0)nrerror("gaussj: Singular Matrix-2");
        pivinv=1.0/ (*(a+icol*n+icol));
        *(a+icol*n+icol) = 1.0;
        for (l=0; l<n; l++) *(a+icol*n+l) *= pivinv;
        for (l=0; l<m; l++) *(b+icol*m+l) *= pivinv;
        for (ll=0; ll<n; ll++)if (ll != icol) {dum = *(a+ll*n+icol);*(a+ll*n+icol) = 0.0;for (l=0; l<n; l++)
*(a+ll*n+l) -= *(a+icol*n+l)*dum;for (l=0; l<m; l++) *(b+ll*m+l) -= *(b+icol*m+l)*dum;}}
        for (l=n-1;l>=0;l--) {if (indxr[l] != indx[l])for (k=0; k<n; k++)SWAP(*(a+k*n+indxr[l]),
*(a+k*n+indx[l]));}
        free_ivector(ipiv,0);free_ivector(indxr,0);free_ivector(indxc,0);}
#undef SWAP

```

G.7. Optimize.c

```

static lensDef *locLens;
static sysDef *locSys;
static optDef *locOpt;
/*****
* Name: setOptimizeGlobals
* Input: lens: System lens definition. sys : System definition. opt : System optimization definition
* Modifies: Local global variables: locLens, locSys, locOpt. locLens = lens. locSys = sys. locOpt = opt
* Purpose: Assigns (local) global system variables so the optimization routines can use them
* Requires: This function must be called before any optimization functions are called.
*****/
void setOptimizeGlobals(sysDef *sys, lensDef *lens, optDef *opt) {
    locLens = lens;locSys = sys;locOpt = opt;return;}
/*****
* Name: desiredPower
* Input: alpha : Vertical Gaze-Angle
        beta : Horizontal Gaze-Angle
* Returns: Power in 1/mm (not dpt)
* Purpose: Calculate the desired power of the lens for the given gaze-direction. The power is
        calculated using the polynomial expansion coefficients in the optimization variable
* The power expression is a function of gaze angle alpha (radians)
* Requires: Global variable locOpt be allocated & defined equal to optimization structure. alpha, beta in
degrees
        There are requirements on the data in locOpt->pcoeff, depending on locOpt->power:
* P_POLY : pcoeff[i] is the coeff to the ith power for alpha (a^i)
* P_HARD : 0 -> base power. 1 -> add power. 2 -> alpha (degrees) at which 'add' ends and constant region
begins
* P_EXP : 0 -> base power term. 1 -> add power term. 2 -> rate of add term. 3 -> midpoint of add term
* P_GAUSS : 0 -> base power term (base). 1 -> add power term (add). 2 -> width of add (w)
*****/
double desiredPower(double alpha, double beta) {
    double pow=0;
    switch(locOpt->power) {
        case P_POLY : { /*Polynomial Description*/
            int i;
            double a;
            if (alpha > 0) alpha = 0; else alpha *= -1.;
            a = 1;
            for(i=0; i<locOpt->np; i++) {pow += locOpt->pcoeff[i]*a; a *= alpha;}} break;
        case P_HARD : { /*Discrete Function*/
            double offset, a1, add;
            offset = locOpt->pcoeff[0];add = locOpt->pcoeff[1];a1 = locOpt->pcoeff[2];
            if(alpha < a1) pow = fabs(alpha)/a1 * add + offset;
            else pow = add + offset;} break;
        case P_EXP : { /*Exponential Function*/
            double y, base, add, k, ys;
            y = locSys->q*tan(alpha*M_PI/180);
            base = locOpt->pcoeff[0];
            add = locOpt->pcoeff[1];
            k = locOpt->pcoeff[2];
            ys = locOpt->pcoeff[3];
            pow = base + add/( 1. + exp(k*(y-ys)) );} break;
        case P_GAUSS : { /*Exponential Function*/
            double base, add, w, r;
            if(alpha < 0) alpha *= -1;
            else alpha *= 1;
            base = locOpt->pcoeff[0];add = locOpt->pcoeff[1];w = locOpt->pcoeff[2];r=0;
            pow = base + add*(1. - exp(-(alpha*alpha/w/w)));} break;
    }
}

```

```

    }
    return pow;}
/*****
* Name: meritVal
* Input: None
* Returns: Merit value of lens
* Modifies: Nothing (though called functions modify lens aberration arrays values)
* Purpose: Calculate the lens' goodness for the optimization routines. The merit function is defined as:
*   Sqrt[Sum[w0*Ast^2 + w1*Moe^2 + w2*Dist^2]] where w0..w2 are opt.weight[0..2]. Currently, Ast and Dist
*   are summed over the optimization range; Moe is summed only along the MM (linear: bottom-half of lens
* Requires: sys->A, sys->M, sys->D must be allocated in accordance with the optimization range
*   The trace range must be set in accordance with the optimization range
* Dependencies: traceLens, computeAst, computeMOE, computeDist
*****/
double meritVal() {
    double merit,m0, m1, m2, w0, w1, w2, maxCount;
    int alphaCount, betaCount, cols,minA,maxA,minB,maxB;
    /*Initialize the merit value, and set the temp variables*/
    erit = 0.0;m0 = 0; w0 = locOpt->weight[0];m1 = 0; w1 = locOpt->weight[1];m2 = 0; w2 = locOpt->weight[2];
    minA = locSys->minAlphaCount;maxA = locSys->maxAlphaCount;minB = locSys->minBetaCount;maxB = locSys->
    >maxBetaCount;
    cols = (maxA - minA + 1);
    /*Trace the system using the new parameters*/
    traceLens(locLens, locSys, NULL);
    /*Compute the relevant errors*/
    if(w0) computeAst(locLens, locSys);
    if(w1) computeMOE(locLens, locSys);
    if(w2) computeDistortion(locLens, locSys);
    /*Compute the merit value*/
    if(w0 || w2) {
        for(betaCount = minB; betaCount <= maxB; betaCount++) {
            for(alphaCount = minA; alphaCount <= maxA; alphaCount++) {
                m0 = *(locSys->A + (alphaCount-minA) + (betaCount-minB)*cols); m0 *= m0;
                m2 = *(locSys->D + (alphaCount-minA) + (betaCount-minB)*cols); m2 *= m2;
                merit += w0*m0 + w2*m2;}}
        if(minB <= 0 && maxB >= 0 && w1) {
            for(betaCount = minB; betaCount <= maxB; betaCount++) {
                for(alphaCount = minA; alphaCount <= maxA; alphaCount++) {
                    double t;
                    m1 = *(locSys->M + (alphaCount-minA) + (betaCount-minB)*cols); m1 *= m1;
                    t=1.0; merit += w1*m1 * t*t*t;}}
            merit = sqrt(merit);
        }
    }
    return merit;}
/*****
* Name: meritFunction
* Input: p : array of optimization parameters
* Returns: Merit value of lens
* Modifies: All lens parameters with a non-zero optimization flag value
* Purpose: This function updates all parameters being optimized with the new values stored in p
*   It then calls meritVal() to actually compute the merit value. The value is then returned.
* Requires: Global variables locLens, locSys, locOpt be defined to be equal to the lens, system, and
    optimization
    structures, respectively. p[1] is the start of the array The index coeffs are first taken from p[], then
    R1,
    then k1 terms, then k2 terms The values of p[] must then correspond to that order when p[] is initialized
* Dependencies: updateBSplineControlPoints, updatePolyControlPoints, meritVal
*****/
double meritFunction(double *p) {
    int i, j; /*loop variables*/
    /*Copy the optimizable parameters to the appropriate lens parameters*/
    j = 1;
    /*Index Coefficients*/
    switch(locLens->grin.mode) {
        case C_BSP : updateBSplineControlPoints(&locLens->grin.nbs, p, locOpt->n_n);break;
        default : updatePolyControlPoints(&locLens->grin.np, p, 0); break;}
    j += locOpt->n_n;
    /*Radius 1*/
    if(locLens->surf1.sph.ropt) { setLensRadius2(locLens, p[j]);j++;}
    /*Surface 1 Aspheric Coeffs*/
    for(i=0; i<locLens->surf1.sph.na; i++) {if(locLens->surf1.sph.aopt[i]) {locLens->surf1.sph.a[i] =
    p[j];j++;}}
    /*Surface 2 Aspheric Coeffs*/
    for(i=0; i<locLens->surf2.sph.na; i++) {if(locLens->surf2.sph.aopt[i]) {locLens->surf2.sph.a[i] =
    p[j];j++;}}
    return meritVal();}
#define NR_END 1
#define FREE_ARG char
static double sqrarg;
#define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);
#define NRANSI
#define GOLD 1.618034
#define CGOLD 0.3819660
#define GLIMIT 100.0
#define TINY 1.0e-20
#define ZEPS 1.0e-10
/*****
* Name: bracketMinMerit (Based on mnbrak from Numerical Recipes)
* Input: ax : left most point of min. bx : right of ax, left of min. cx : does not need to be defined on input

```

```

*      fa : need not be defined on input. fb : need not be defined on input. fc : need not be defined on
input
*      func: function to bracket the minimum
* Modifies: Modifies ax, bx, cx such that: f(ax) > f(bx) > f(cx). fa is given the value f(ax).
*      fb is given the value f(bx). fc is given the value f(cx)
* Purpose: Brackets the minimum of the function passed to it.
* Requires: The function to be minimized must expect an array of values, with the starting index of 1
*****/
void bracketMinMerit(double *ax,double *bx,double *cx,double *fa,double *fb,double *fc,double (*func)(double *))
{
    int iter=1;
    double ulim,u,r,q,fu,dum;
    *fa=(*func)(ax-1);
    *fb=(*func)(bx-1);
    if (*fb > *fa) {SHFT(dum,*ax,*bx,dum) SHFT(dum,*fb,*fa,dum)}
    *cx=(*bx)+GOLD*( *bx-*ax);
    *fc=(*func)(cx-1);
    while (*fb > *fc) {
        iter++;r=( *bx-*ax)*( *fb-*fc);q=( *bx-*cx)*( *fb-*fa);
        u=( *bx)-(( *bx-*cx)*q-( *bx-*ax)*r)/(2.0*SIGN2(FMAX(fabs(q-r),TINY),q-r)); ulim=( *bx)+GLIMIT*( *cx-*bx);
        if (( *bx-u)*(u-*cx) > 0.0) {
            fu=(*func)( &u-1);
            if (fu < *fc) { *ax=( *bx); *bx=u; *fa=( *fb); *fb=fu; return;}
            else if (fu > *fb) { *cx=u; *fc=fu; return;} u=( *cx)+GOLD*( *cx-*bx); fu=(*func)( &u-1);
            else if (( *cx-u)*(u-ulum) > 0.0) {
                fu=(*func)( &u-1);
                if (fu < *fc) {SHFT( *bx,*cx,u,*cx+GOLD*( *cx-*bx)) SHFT( *fb,*fc,fu,(*func)( &u-1))}
            }
            else if ((u-ulum)*(ulum-*cx) >= 0.0) {u=ulim; fu=(*func)( &u-1);
            }
            else {u=( *cx)+GOLD*( *cx-*bx); fu=(*func)( &u-1);
            SHFT( *ax,*bx,*cx,u) SHFT( *fa,*fb,*fc,fu)}
        }
    }
}
/*****
* Name: minimizeLDMerit (based on Brent from Numerical Recipes)
* Input: ax : Bracketed point left of min. bx : Bracketed point, right of min.
*      cx : Bracketed point between ax, bx, s.t. f(ax) > f(bx) > f(cx). func : function to minimize.
*      tol : tolerance on minimum value; no less than sqrt(machine-res). xmin : does not need to be defined on
input
* Returns: Minimum value of function if the function succeeds. -1 if it exceeds ITMAX iterations in the search
* Modifies: xmin is given the position for the minimum value of the function
* Purpose: Finds position of min and min value for function: for min bracketed by ax, bx, cx
* Requires: Function minimum is bracketed by ax, bx, cx.
* The function to be minimized must expect an array of values, with the starting index of 1
*****/
#define ITMAX 100
double minimizeLDMerit(double ax, double bx, double cx, double (*func)(double *), double tol, double *xmin) {
    int iter;
    double a,b,d,etemp,fu,fv,fw,fx,p,q,r,toll,tol2,u,v,w,x, xm, e=0.0;
    a=(ax < cx ? ax : cx);b=(ax > cx ? ax : cx);x=w=v=bx;fw=fv=fx=(*func)( &x-1);
    for (iter=1; iter<=ITMAX; iter++) {
        xm = 0.5*(a+b);tol2 = 2.0*(tol=fabs(x)+ZEPS);
        if (fabs(x-xm) <= (tol2 - 0.5*(b-a))) { *xmin = x; return fx;}
        if (fabs(e) > toll) {
            r = (x-w)*(fx-fv);q = (x-v)*(fx-fw);p = (x-v)*q-(x-w)*r;q = 2.0*(q-r);
            if (q > 0.0) p = -p;
            q = fabs(q);etemp = e;e = d;
            if (fabs(p) >= fabs(0.5*q*etemp) || p <= q*(a-x) || p >= q*(b-x))d=CGOLD*(e=(x >= xm ? a-x : b-
x));
            else {d=p/q;u=x+d;if (u-a < tol2 || b-u < tol2) d = SIGN2(toll, xm-x);}
            e=(fabs(d) >= toll ? x+d : x+SIGN2(toll, d));
            fu=(*func)( &u-1);
            if (fu <= fx) {if (u >= x) a=x; else b=x;SHFT(v,w,x,u) SHFT(fv,fw,fx,fu)}
            else {
                if (u < x) a=u; else b=u;
                if (fu <= fw || w == x) {v = w;w = u;fv = fw;fw = fu;}
                else if (fu <= fv || v == x || v == w) {v = u;fv = fu;}
            }
            nrerror("Too many iterations in minimizeLDMerit");printf("\n");fprintf(outfps[F_DEBUG], "\n");
            return -1;}
    }
}
#undef ITMAX
/*****
* Name: fldim & linmin
* Purpose: These functions work together to minimize on a 1D vector, in a Multi-D optimization.
* They are used by minimizeMDMerit.
* Requires: Global variables ncom, pcom, xicom, nrfunc
* Dependencies: bracketMinMerit, minimizeLDMerit, array, free_array
*****/
static int ncom;
static double *pcom,*xicom,(*nrfunc)(double []);
double fldim(double *x) {
    int j;
    double f,*xt;
    xt = array(1,ncom);for (j=1; j<=ncom; j++) xt[j] = pcom[j] + (x[1])*xicom[j];
    f = (*nrfunc)(xt);free_array(xt, 1, ncom);
    return f;}
void linmin(double p[], double xi[], int n, double tol, double *fret, double (*func)(double [])) {
    int j;
    double xx,xmin,fx,fb,fa,bx,ax;
    ncom = n;pcom = array(1, n);
    xicom = array(1, n);nrfunc = func;
    for (j=1; j<=n; j++) {pcom[j] = p[j];xicom[j] = xi[j];}
    ax=0.0;xx=1.0;
}

```

```

    bracketMinMerit(&ax, &xx, &bx, &fa, &fx, &fb, fldim);*fret = minimize1DMerit(ax, xx, bx, fldim, tol,
&xmin);
    for (j=1; j<=n; j++) {xi[j] *= xmin;p[j] += xi[j];}
    free_array(xicom, 1, n);free_array(pcom, 1, n);}
/*****
* Name: minimizeMDMerit (based on powell from Numerical Recipes)
* Input: p : p[1..n] are the starting positions for optimization. func : does not need to be defined on input
*        xi : xi[1..n][1..n] is the matrix basis vectors of n-dimensional space.
*           Each vector is a row of xi. All vectors should be unique.
*        n : number of variables (dimensions) to operate on
*        tolMD : tolerance in finding minimum value. Shouldn't be less than machine limit
*        tollD : tolerance for 1D linear optimization. Shouldn't be less than machine limit
*        iter : Doesn't need to be set on input. smerit : beginning merit value. emerit : ending merit
value
* Modifies: iter : number of iterations taken. fret : minimum value found. xi : position of min found
* Purpose: Finds position and value of minimum for multi-dimensional function. If opt. Mode is O_USER, it queries
* the user before every major sequence if the opt. should be continued. An input of 'n' or 'N' will halt the
* optimization and return with the current settings.
* Requires: p is indexed from 1..n. xi is indexed xi[1..n][1..n]. The function minimized must expect an array
* of values, with the starting index of 1 and must return a real number.
* Dependencies: linmin, array, free_array
*****/
void minimizeMDMerit(double p[], double **xi, int n, double tolMD, double tollD, int *iter, double *smerit,
double *emerit, double (*func)(double [])) {
    int i, ibig, j;
    char cont[2];
    double del, fp, fptt, t, *pt, *ptt, *xit;
    pt=array(1,n);ptt=array(1,n);xit=array(1,n);*emerit=(*func)(p);*smerit = *emerit;*iter = 0;
    for (j=1; j<=n; j++) pt[j]=p[j];
    for (*iter=1; **iter) {
        fp=(*emerit);
        printf(" %2d Minimizing: %2d (%11.9lf)", n, *iter, fp);
        fprintf(outfps[F_DEBUG], " %2d Minimizing: %2d (%11.9lf)", n, *iter, fp);
        if(locOpt->opt == O_USER) {
            printf(" Continue? (y/n): ");
            scanf("%s", &cont);
            if(cont[0] == 'n' || cont[0] == 'N') {
                free_array(xit,1,n);free_array(ptt,1,n);free_array(pt,1,n);return;}
            else {printf("\n");fprintf(outfps[F_DEBUG], "\n");}
            ibig=0;del=0.0;
            for (i=1; i<=n; i++) {
                for (j=1; j<=n; j++) xit[j] = xi[j][i];
                fptt=(*emerit);linmin(p, xit, n, tollD, emerit, func);
                if (fabs(fptt-(*emerit)) > del) {del = fabs(fptt-(*emerit));ibig = i;}
            }
            if (2.0*fabs(fp-(*emerit)) <= tolMD*(fabs(fp)+fabs(*emerit))) {
                free_array(xit, 1, n);free_array(ptt, 1, n);free_array(pt, 1, n);
                printf(" Final: (%11.9lf)\n", fp); return;}
            for (j=1; j<=n; j++) {ptt[j] = 2.0*p[j] - pt[j];xit[j] = p[j] - pt[j];pt[j] = p[j];}
            fptt=(*func)(ptt);
            if (fptt < fp) {
                t = 2.0*(fp-2.0*(emerit)+fptt)*SQR(fp-(*emerit)-del)-del*SQR(fp-fptt);
                if (t < 0.0) {
                    linmin(p, xit, n, tollD, emerit, func);
                    for (j=1; j<=n; j++) {xi[j][ibig] = xi[j][n];xi[j][n] = xit[j];}}}}
}
#undef NRANSI
#undef GOLD
#undef CGOLD
#undef GLIMIT
#undef TINY
#undef ZEPS
void meritSpaceVals(FILE **outfps, int z, int nal, int na2, int nn, double *param, int pc) {
    double v, *vp, min, max, step, m;
    step = 0;
    if(z == 0) {
        min = locLens->surf1.sph.rmerit[0];max = locLens->surf1.sph.rmerit[1];step= locLens-
>surf1.sph.rmerit[2];
        vp = &locLens->surf1.sph.radius;}
    else if(z <= nal) {
        min = *(locLens->surf1.sph.amerit[0]+z-1);max = *(locLens->surf1.sph.amerit[1]+z-1);
        step= *(locLens->surf1.sph.amerit[2]+z-1);vp = &locLens->surf1.sph.a[z-1];}
    else if(z <= nal+na2) {
        min = *(locLens->surf2.sph.amerit[0]+z-(1+nal));max = *(locLens->surf2.sph.amerit[1]+z-(1+nal));
        step= *(locLens->surf2.sph.amerit[2]+z-(1+nal));vp = &locLens->surf2.sph.a[z-(1+nal)];}
    else if(z <= nal+na2+nn) {
        min = *(locLens->grin.np.zmerit[0]+z-(1+nal+na2));max = *(locLens->grin.np.zmerit[1]+z-(1+nal+na2));
        step= *(locLens->grin.np.zmerit[2]+z-(1+nal+na2));vp = &locLens->grin.np.zc[z-(1+nal+na2)];}
    else {/*z > nl+na2+nn */
        int i;
        static int iter = 0;
        m = meritVal();
        for(i=0; i<=pc; i++)fprintf(outfps[F_MERIT], "%+-8.6g ", param[i]);
        fprintf(outfps[F_MERIT], "%+-8.6g\n", m);fflush(outfps[F_MERIT]);return;}
    if(step > ZERO) {
        pc++;
        for(v=min; v<=max; v+=step) {
            if(z==0) printf("."); *vp = v; param[pc]=v;
            updateLensRadius2(locLens); meritSpaceVals(outfps, z+1, nal, na2, nn, param, pc);}
        if(z==0) printf(".");}
    else
        meritSpaceVals(outfps, z+1, nal, na2, nn, param, pc);
}

```

```

    return;}
void meritSpaceFunction(FILE **outfps, sysDef *sys, lensDef *lens) {
    int i, na1, na2, nn, num;
    double *vals;
    setTraceRange(lens, sys);allocateStorage(sys);computeDesiredPower(lens, sys);
    /*Count total possible params to work with*/
    na1 = locLens->surf1.sph.na;na2 = locLens->surf2.sph.na;
    nn = numIndexMeritSpaceParams(&locLens->grin); //numIndexParam(&locLens->grin);
    /*Count free params for merit space calculation*/
    num = 0;
    if(locLens->surf1.sph.rmerit[2]) num++;/*Radius 1*/
    for(i=0; i<locLens->surf1.sph.na; i++)/*Surf 1 Aspherics*/
    if(*(locLens->surf1.sph.amerit[2]+i)) num++;
    for(i=0; i<locLens->surf2.sph.na; i++)/*Surf 1 Aspherics*/
    if(*(locLens->surf2.sph.amerit[2]+i)) num++;
    num += numIndexMeritSpaceParams(&locLens->grin);/*GRIN coeffs*/
    vals = (double *)calloc(num, sizeof(double));
    locLens->power = desiredPower(0,0);
    meritSpaceVals(outfps, 0, na1, na2, nn, vals, -1);
    printf("\n");fprintf(outfps[F_DEBUG], "\n");
    freeStorage(sys);
    return;}
/*****
* Name: optimize
* Input: None
* Modifies: lens, the meritVal variable of opt
* Purpose: Optimize the lens by minimizing the merit function.
*          Calls the appropriate function depending on the index representation
* Requires: Global variables locLens, locSys, locOpt exist and defined via setOptimizeGlobals
* Dependencies:optimizeBSpline, optimizePoly
*****/
void optimize() {
    switch(locLens->grin.mode) {
        case C_BSP : optimizeBSpline(locSys, locLens, locOpt); break;
        default : optimizePoly(locSys, locLens, locOpt); break;}
    return;}

```

G.8. Polynomial.c

```

/*****
* Name: radial
* Input: ply : polynomial structure. R : position to evaluate Polynomial
* Returns: Polynomial value at R
* Purpose: Calculate Polynomial value at input position for a radial f(r) function
* Requires: ply is allocated and defined.
*****/
double radial(poly *ply, vector *R) {
    int i;
    double x, y, r, rv, rsq, index;
    x = R->x;y = R->y;index = 0;rv = 1;rsq = (x*x + y*y);r = sqrt(rsq);
    for(i=0; i<ply->ny; i++) {index += ply->zc[i]*rv;rv *= r;}
    return index;}
/*****
* Name: diffRadialX
* Input: ply : polynomial structure. R : position to evaluate polynomial
* Returns: Polynomial derivative value (with respect to x) at R
* Modifies: error if R outside of aperture
* Purpose: Calculate Polynomial derivative value (wrt/x) at R for a radial f(r) function
* Requires: ply is allocated and defined.
*****/
double diffRadialX(poly *ply, vector *R) {
    int i;
    double x, y, r, rv, rsq, deriv;
    deriv = 0;x = R->x;y = R->y;rsq = x*x+y*y;r = sqrt(rsq);rv = 1.0/(r + ZERO); /*to avoid 1/0*/
    for(i=1; i<ply->ny; i++) {deriv += i*ply->zc[i]*rv;rv *= r;}
    deriv *= x;
    return deriv;}
/*****
* Name: diffRadialY
* Input: ply : polynomial structure. R : position to evaluate polynomial
* Returns: Polynomial derivative value (with respect to y) at R
* Modifies: error if R outside of aperture
* Purpose: Calculate Polynomial derivative value (wrt/y) at R for a radial f(r) function
* Requires: ply is allocated and defined.
*****/
double diffRadialY(poly *ply, vector *R) {
    int i;
    double x, y, r, rv, rsq, deriv;
    deriv = 0;x = R->x;y = R->y;rsq = x*x+y*y;r = sqrt(rsq);rv = 1.0/(r + ZERO);
    for(i=1; i<ply->ny; i++) {deriv += i*ply->zc[i]*rv;rv *= r;}
    deriv *= y;
    return deriv;}
/*****
* Name: radialSquared
* Input: ply : polynomial structure. R : position to evaluate B-spline
* Returns: Polynomial value at R

```

```

* Purpose: Calculate Polynomial value at input position for a radial squared f(r^2) function
* Requires: ply is allocated and defined.
*****/
double radialSquared(poly *ply, vector *R) {
    int i;
    double x, y, rv, rsq, index;
    rv = 1; x = R->x; y = R->y; index = 0; rsq = (x*x + y*y);
    for(i=0; i<ply->ny; i++) {index += ply->zc[i]*rv; rv *= rsq;}
    return index;}
/*****/
* Name:    diffRadialSquaredX
* Input:  ply : polynomial structure. R : position to evaluate polynomial
* Returns: Polynomial derivative value (with respect to x) at R
* Modifies: error if R outside of aperture
* Purpose: Calculate derivative value (wrt/x) at R for a radial squared f(r^2) function
* Requires: ply is allocated and defined.
*****/
double diffRadialSquaredX(poly *ply, vector *R) {
    int i;
    double x, y, rv, rsq, deriv;
    rv = 1; x = R->x; y = R->y; deriv = 0; rsq = (x*x + y*y); rsq = x*x+y*y; rv = 1;
    for(i=1; i<ply->ny; i++) {deriv += i*ply->zc[i]*rv; rv *= rsq;}
    deriv *= 2*x;
    return deriv;}
/*****/
* Name:    diffRadialSquaredY
* Input:  ply : polynomial structure. R : position to evaluate polynomial
* Returns: Polynomial derivative value (with respect to y) at R
* Modifies: error if R outside of aperture
* Purpose: Calculate derivative value (wrt/y) at R for a radial squared f(r^2) function
* Requires: ply is allocated and defined.
*****/
double diffRadialSquaredY(poly *ply, vector *R) {
    int i;
    double x, y, rv, rsq, deriv;
    rv = 1;
    x = R->x; y = R->y; deriv = 0; rsq = (x*x + y*y); rsq = x*x+y*y; rv = 1;
    for(i=1; i<ply->ny; i++) {deriv += i*ply->zc[i]*rv; rv *= rsq;}
    deriv *= 2*y;
    return deriv;}
/*****/
* Name:    polyXY
* Input:  ply : polynomial structure. R : position to evaluate polynomial
* Returns: Polynomial value at R
* Purpose: Calculate Polynomial value at input position for a X-Y f(x,y) function
* Requires: ply is allocated and defined.
*****/
double polyXY(poly *ply, vector *R) {
    int i, j;
    double x, y, xv, yv, index;
    x = R->x - ply->xo; y = R->y - ply->yo; index = 0; xv = 1;
    for(i=0; i<ply->nx; i++) {
        yv = 1; for(j=0; j<ply->ny; j++) {index += ply->zc[j*ply->nx + i]*xv*yv; yv *= y;} xv *= x;}
    return index;}
/*****/
* Name:    diffPolyXYX
* Input:  ply : polynomial structure. R : position to evaluate polynomial
* Returns: Polynomial derivative value (with respect to x) at R
* Modifies: error if R outside of aperture
* Purpose: Calculate derivative value (wrt/x) at R for an X-Y f(x,y) function
* Requires: ply is allocated and defined.
*****/
double diffPolyXYX(poly *ply, vector *R) {
    int i, j;
    double x, xp, xv, y, yp, yv, deriv;
    x = R->x - ply->xo; y = R->y - ply->yo; deriv = 0; xp = x; yp = y; xv = 1;
    for(i=1; i<ply->nx; i++) {
        yv = 1; for(j=0; j<ply->ny; j++) {deriv += i*ply->zc[j*ply->nx + i]*xv*yv; yv *= yp;} xv *= xp;}
    return deriv;}
/*****/
* Name:    diffPolyXYX
* Input:  ply : polynomial structure. R : position to evaluate polynomial
* Returns: Polynomial derivative value (with respect to y) at R
* Modifies: error if R outside of aperture
* Purpose: Calculate derivative value (wrt/y) at R for an X-Y f(x,y) function
* Requires: ply is allocated and defined.
*****/
double diffPolyXYX(poly *ply, vector *R) {
    int i, j;
    double x, xp, xv, y, yp, yv, deriv;
    x = R->x - ply->xo; y = R->y - ply->yo; deriv = 0; xp = x; yp = y; xv = 1;
    deriv = 0; xp = x; yp = y; xv = 1;
    for(i=0; i<ply->nx; i++) {
        yv = 1; for(j=1; j<ply->ny; j++) {deriv += j*ply->zc[j*ply->nx + i]*xv*yv; yv *= yp;} xv *= xp;}
    return deriv;}
/*****/
* Name:    updatePolyControlPoints
* Input:  ply : Polynomial structure. cp : Array of new control point values. nc : Number of new control point values
* Modifies: Polynomial control point values

```

```

* Purpose: Update the control points being optimized with the current (modified) values from optimizer.
* Requires: ply is allocated and defined
* Dependencies: Nothing
*****/
void updatePolyControlPoints(poly *ply, double *cp, int nc) {
    int i, j;
    j=1;
    for(i=0; i<ply->nx*ply->ny; i++) {if(ply->zopt[i]) {ply->zc[i] = cp[j];j++;}}
    return;}
/*****
* Name: optimizePoly
* Input: sys : System structure. lens: Lens structure. opt : Optimization structure
* Modifies: May modify either/or Polynomial control point values, lens surface parameters
* Purpose: Optimize the appropriate control points in order to improve the lenses performance optimizer.
* Requires: All lens system, and optimization parameters are properly defined
* Dependencies: setTraceRange, allocateStorage, computeDesiredPower, freeStorage,minimizeMDMerit,
reflectControlPoints, * computeDesiredPower, setLensRadius2, bracketMinMerit, minimize1DMerit, meritFunction
*****/
void optimizePoly(sysDef *sys, lensDef *lens, optDef *opt) {
    double Ma, Mb, Mc, Pa, Pb, Pc, *pos, **ui, tol; /*Merit function, 1D Starting; Multi-Dim Opt values*/
    int i, j, n, iter, nc; /*Multi-Dimensional Opt parameters*/
    double dAngle, dim[4]; /*Temp variable for sys->dAngle, Temp Variable for sys->traceDim*/
    /*Temporarily change the system trace settings to the optimization settings*/
    printf(" Setup\n");fprintf(outfps[F_DEBUG], " Setup\n");
    dAngle = sys->dAngle;
    for(i=0; i<4; i++) dim[i] = sys->traceDim[i];
    for(i=0; i<4; i++) sys->traceDim[i] = opt->optDim[i];
    sys->dAngle = opt->dAngle;
    setTraceRange(lens, sys);allocateStorage(sys);computeDesiredPower(lens, sys);
    /*Set power to the desired Power & set the back surface radius*/
    lens->power = desiredPower(0,0);
    if(lens->surf2.sph.ropt) setLensRadius2(lens, lens->surf1.sph.radius);
    nc = lens->grin.np.nx*lens->grin.np.ny;
    /*Count the number of parameters to be optimized; allow space for n(n+1)*/
    n = 1; opt->n_s1 = 0; opt->n_s2 = 0; opt->n_n = 0;
    printf(" Preparing Coefficients\n"); fprintf(outfps[F_DEBUG], " Preparing Coefficients\n");
    if(lens->surf1.sph.ropt) n++;
    for(i=0; i<lens->surf1.sph.na; i++) opt->n_s1 += lens->surf1.sph.aopt[i];
    n += opt->n_s1;
    for(i=0; i<lens->surf2.sph.na; i++) opt->n_s2 += lens->surf2.sph.aopt[i];
    n += opt->n_s2;
    for(i=0; i<nc; i++) opt->n_n += lens->grin.np.zopt[i];
    printf(" Allocating memory\n"); fprintf(outfps[F_DEBUG], " Allocating memory\n");
    n += opt->n_n;
    lens->grin.nbz.zopt = (int *)calloc(opt->n_n, sizeof(int));
    /*allocate memory for position variable*/
    pos = (double *)calloc(n, sizeof(double));
    /*allocate memory for 2D basis vector matrix*/
    ui = (double **)calloc(n, sizeof(double *));
    for(i=0; i<n; i++) ui[i] = (double *)calloc(n, sizeof(double));
    /*Set the optimization parameters.
    * Note: all ui[][] terms are the axis vectors of the
    * subspace of the optimization parameter and so they are
    * scaled relative to the starting position pos[]. This gives
    * greater optimization stability .
    */
    j = 1;
    printf(" Setting Parameters\n"); fprintf(outfps[F_DEBUG], " Setting Parameters\n");
    /*Set initial basis vectors and starting guess for index parameters*/
    for(i=0; i<nc; i++) {
        if(lens->grin.np.zopt[i]) {
            pos[j] = lens->grin.np.zc[i];
            if(pos[j] != 0) ui[j][j] = pos[j]/100;
            else ui[j][j] = 1e-6/pow(10,i*(1+lens->grin.mode));
            j++;}
    /*Set initial basis vector and starting guess for Surface 1 Radius*/
    if(lens->surf1.sph.ropt) {pos[j] = lens->surf1.sph.radius;ui[j][j] = pos[j]/100;j++;}
    /*Set initial basis vectors and starting guess for Surface 1 Aspheric Terms*/
    for(i=0; i<lens->surf1.sph.na; i++) {
        if(lens->surf1.sph.aopt[i]) {
            pos[j] = lens->surf1.sph.a[i];
            if(pos[j] != 0) ui[j][j] = pos[j]/100;
            else ui[j][j] = 1./pow(10,2*(i+1)+3);
            j++;}
    /*Set initial basis vectors and starting guess for Surface 2 Aspheric Terms*/
    for(i=0; i<lens->surf2.sph.na; i++) {
        if(lens->surf2.sph.aopt[i]) {
            pos[j] = lens->surf2.sph.a[i];if(pos[j] != 0) ui[j][j] = pos[j]/100;
            else ui[j][j] = 1./pow(10,2*(i+1)+3);j++;}
    printf(" Running\n"); fprintf(outfps[F_DEBUG], " Running\n");
    tol = 1e-3;
    /*Determine Optimization process based on number of Degrees of Freedom*/
    switch(n) {
    case 1 : /*Zero DOF : Do nothing*/ break;
    case 2 : /*One DOF : 1D Optimization*/
    /*If starting position is non-zero, use as initial guess, else use the vector length determined
    earlier*/
        if(pos[1] != 0) Pc = pos[1]; else Pc = ui[1][1];
        /*Guess at range to straddle minima*/
        if(SIGN(Pc) > 0) {Pa = 0.95*Pc;Pb = 0.99*Pc;}else {Pa = 1.05*Pc;Pb = 1.01*Pc;}

```

```

/*Bracket the local minima, and then find the minima*/
bracketMinMerit(&Pa, &Pb, &Pc, &Ma, &Mb, &Mc, meritFunction);
opt->startMerit = meritVal();
opt->endMerit = minimizeDMerit(Pa, Pb, Pc, meritFunction, opt->tolLD, &pos[1]);
break;
default : /*More than One DOF : Multi-Dimensional Optimization*/
minimizeMDMerit(pos, ui, n-1, opt->tolMD, opt->tolLD, &iter, &opt->startMerit, &opt->endMerit,
meritFunction);
break;}
/*Set the system parameters back to their trace values*/
printf(" Finished\n"); fprintf(outfps[F_DEBUG], " Finished\n");
freeStorage(sys); sys->dAngle = dAngle;
for(i=0; i<4; i++) sys->traceDim[i] = dim[i];
return;}

```

G.9. RayTrace.c

```

/*****
* Name: propagateSameRay
* Input: R : ray position. U : ray direction. L : propagation distance
* Modifies: R
* Purpose: Propagate a ray a distance : R += L*U
* Requires: R, U must be allocated & defined
*****/
void propagateSameRay(vector *R, dirCos *U, double L) {
    R->x = R->x + L*U->alpha; R->y = R->y + L*U->beta; R->z = R->z + L*U->gamma;
    return;}
/*****
* Name: propagateSameRay
* Input: R : ray position. U : ray direction. L : propagation distance. Rp : propagated ray (need not be
defined)
* Modifies: Rp
* Purpose: Propagate a ray a distance : Rp = R + L*U
* Requires: R, U must be allocated & defined, Rp must be allocated
*****/
void propagateRay(vector *R, dirCos *U, double L, vector *Rp) {
    Rp->x = R->x + L*U->alpha; Rp->y = R->y + L*U->beta; Rp->z = R->z + L*U->gamma;
    return;}
/*****
* Name: propagateSameRay
* Input: R : initial ray position. U : initial ray direction. l : lens. OPL: optical path length (need not be
defined)
* Modifies: R : has final ray position after propagation. U : has final ray direction after propagation
* OPL: has optical path length ray travels during propagation
* Purpose: Propagate ray through lens via Sharma's method. If U->gamma > 0 it is propagated to second surface.
* If U->gamma < 0 it is propagated to the first surface. Unless an error occurs, the ray position will be
within
* 1e-9 of the output surface. If an error occurs, the ray position & direction are not expected to be
meaningful.
* Negative edge thickness is not considered
* Requires: R, U, l must be allocated & defined
* Dependencies: stepRay, sphereSurfaceCondition, l->grin()
*****/
void propagateInGRIN(vector *R, dirCos *U, lensDef *l, double *OPL) {
    int i=0, Tsign, complete;
    double dt, n, extra, thresh;
    double Nm, Sm;
    dirCos T, oldT;
    vector oldR;
    surface *surf;
    thresh = 5.0e-10;
    n = index(&l->grin, R);
    T.alpha = U->alpha * n; T.beta = U->beta * n; T.gamma = U->gamma * n;
    Tsign = (int)SIGN(T.gamma);
    if(Tsign > 0) surf = &l->surf2; else surf = &l->surf1;
    if(l->grin.mode == G_H) { propagateSameRay(R, U, intersectSurfaceDist(R, U, surf)); return;}
    dt = l->t/l->grin.steps/(Tsign * T.gamma);
    complete=0;
    while(!complete && error == NONE) {
        oldR.x = R->x; oldR.y = R->y; oldR.z = R->z;
        oldT.alpha = T.alpha; oldT.beta = T.beta; oldT.gamma = T.gamma;
        stepRay(R, &T, l, dt, &Nm, &Sm); if(error != NONE) return;
    }
    /*
    * Tsign is < 0 if the ray is traveling from left to right
    * ___SurfaceCondition is < 0 if the ray has not crossed going from left to right
    * ___SurfaceCondition is > 0 if the ray has crossed going from left to right
    * Thus: Tsign * ___SurfaceCondition > 0 if the ray has crossed from left to right
    * Likewise: Tsign * ___SurfaceCondition > 0 if the ray has crossed from right to left
    */
    extra = Tsign*surfaceCondition(R, &T, surf, 0.0);
    if(2*extra == extra+1. || error != NONE || dt < thresh/100) { /*Detect if extra is Infinity*/
        error = NUMERICAL; extra = 0; complete = 1;}
    if(extra > -thresh || extra <= -INFINITY) {
        if(extra > thresh || extra > INFINITY || extra < -INFINITY) {
            double ddt;
            R->x = oldR.x; R->y = oldR.y; R->z = oldR.z;

```

```

        T.alpha = oldT.alpha; T.beta = oldT.beta; T.gamma = oldT.gamma;
        /*take a better estimate of correct dt to hit surface*/
        ddt = extra/(Tsign*T.gamma);
        if(ddt < dt) dt -= ddt;
        else dt /= 2.;
        else complete = 1;}
        else if(extra > -dt) {dt = -extra/(Tsign*T.gamma);}}
    n = index(&l->grin, R);
    U->alpha = T.alpha / n;U->beta = T.beta / n; U->gamma = T.gamma / n;
    return;}
/*****
* Name:    exitPupilNorm
* Input:  alpha : vertical gaze-angle. beta : horizontal gaze-angle.
* mode :  angle definition used for alpha, beta. norm : exit pupil normal (not required to be defined)
* Modifies: norm
* Purpose: Determine the normal vector of the exit pupil, defined by the gaze-direction.
* Requires: norm must be allocated
* Dependencies: setDirCos
*****/
void exitPupilNorm(double alpha, double beta, angleMode mode, dirCos *norm) {
    if(mode == DEG) {alpha *= M_PI/180.0;beta *= M_PI/180.0;}
    setDirCos(M_PI_2-beta, M_PI_2-alpha, 0, RAD+PARTIAL, norm); return;}
/*****
* Name:    exitPupilAxis
* Input:  alpha : vertical gaze-angle. beta : horizontal gaze-angle.
* mode :  angle definition used for alpha, beta. epMode : which exit pupil axis to generate
* axis :  the exit pupil axis generated (not required to be defined)
* Modifies: axis : defines it according to epMode and the gaze-direction
* Purpose: Calculate exit pupil axis direction cosine vector according to gaze-direction and which axis is
desired.
* Requires: axis must be allocated
* Dependencies: setDirCos, linearDirCosCombination
*****/
void exitPupilAxis(double alpha, double beta, angleMode mode, epAxisMode epMode, dirCos *axis) {
    double theta;
    dirCos xtemp, ytemp;
    if(mode == DEG) {alpha *= M_PI/180.0;beta *= M_PI/180.0;}
    theta = atan2(cos(alpha), sin(beta));
    setDirCos(sin(alpha)*sin(beta), cos(alpha), sin(alpha)*cos(beta), DC, &xtemp);
    setDirCos(beta, 0, beta+M_PI_2, RAD, &ytemp);
    switch(epMode) {
        case TAN : linearDirCosCombination(&xtemp, &ytemp, theta, axis, RAD); break;
        case SAG : linearDirCosCombination(&xtemp, &ytemp, theta+M_PI_2, axis, RAD); break;
        case TANN : linearDirCosCombination(&xtemp, &ytemp, theta+M_PI, axis, RAD); break;
        case SAGN : linearDirCosCombination(&xtemp, &ytemp, theta-M_PI_2, axis, RAD); break;
    }
    return;}
/*****
* Name:    aperture
* Input:  R : position vector. s : surface
* Modifies: error = APERTURE if R is outside of aperture of s
* Purpose: Determine whether R is within the aperture of s
* Requires: R, s must be allocated and the semi-ape defined
* Dependencies: Nothing
*****/
void aperture(vector *R, surface *s) {
    double radius = s->sa;if(R->x*R->x + R->y*R->y > radius*radius) error = APERTURE;return;}
/*****
* Name:    searchDirection
* Input:  U1 : ray direction. V1 : vector perpendicular to U1. V2 : vector perpendicular to U1, V1
* phi : angle relative to U1 of V3 ray. theta : angle relative to V1 of V3 ray. V3 : modified vector
* Modifies: V3
* Purpose: Used in searching the exit-pupil for the tangential and sagittal rays. It is necessary to create
search rays on cone surface, angular half-width phi, about chief ray. Phi determines the cone surface.
Theta
* specifies the specific line on the cone. Since U1, V1, V2 are mutually perpendicular, they form basis set
in
* 3D. V3 is offset from U1 by phi. Projected onto the V1-V2 plane, V3 is at an angle of theta from V1.
* Requires: U1, V1, V2 are perpendicular. V3 allocated
* Dependencies: linearDirCosCombination
*****/
void searchDirection(dirCos *U1, dirCos *V1, dirCos *V2, double phi, double theta, dirCos *V3) {
    /*get position of ray in search circle*/
    linearDirCosCombination(V1, V2, theta, V3, RAD);
    /*determine scaling factors so V3 is normalized*/
    /*perform scaling now to (hopefully) avoid roundoff error*/
    V3->alpha = cos(phi)*U1->alpha + sin(phi)*V3->alpha;
    V3->beta = cos(phi)*U1->beta + sin(phi)*V3->beta;
    V3->gamma = cos(phi)*U1->gamma + sin(phi)*V3->gamma;
    return;}
/*****
* Name:    refract
* Input:  ui : incident ray direction. norm : surface normal. n1 : incident index of refraction
* n2 : second index of refraction. ur : ray direction after refraction
* Modifies: ur
* Purpose: Refract a ray in 3D
* Requires: ui, norm allocated & defined; ur allocated;
* Dependencies: dotProduct
*****/
void refract(dirCos *ui, dirCos *norm, double n1, double n2, dirCos *ur) {
    double udotn, co, n2sq, n1sq, tg, sg;

```

```

/* The normal must be in the direction of the incident ray
 * For a ray propagating in the -z direction, I think the
 * following will reverse the normal's direction
 */
if((sg=SIGN(ui->gamma)) < 0) reverseDirection(norm);
udotn = dotProduct(ui, norm); n2sq = n2*n2; nlsq = n1*n1 * (1.0 - udotn*udotn);
if(n2sq < nlsq) {error = TIR;return;}
co = sqrt(n2sq - nlsq) - n1*udotn;
/*I don't what causes a ray direction reversal, but it can't be good*/
tg = (n1*ui->gamma + co*norm->gamma) / n2;
if(SIGN(tg)*sg != 1.) {error = TIR;return;}
ur->alpha = (n1*ui->alpha + co*norm->alpha)/n2;
ur->beta = (n1*ui->beta + co*norm->beta) / n2;
ur->gamma = tg;
return;}
double intersectSurfaceDist(vector *R, dirCos *U, surface *s) {
double dist=0;
switch(s->mode) {
case S_SPH : dist = intersectSphereDist(R, U, &s->sph);break;
case S_PLANE : dist = intersectPlaneDist(R, U, &s->pla);break;}
return dist;}
void surfaceNorm(vector *R, surface *s, dirCos *norm) {
switch(s->mode) {
case S_SPH : sphereNorm(R, &s->sph, norm);break;
case S_PLANE : planeNorm(R, &s->pla, norm); break;}
return;}
double surfaceCondition(vector *R, dirCos *U, surface *s, double L) {
double dist;
switch(s->mode) {
case S_SPH : dist = sphereSurfaceCondition(R, U, &s->sph, L); break;
case S_PLANE : dist = planeSurfaceCondition(R, U, &s->pla, L); break;}
return dist;}
/*****
* Name: tracePupilRay
* Input: alpha : vertical gaze-angle. beta : horizontal gaze-angle. axis : which pupil axis to find
* pupilRad: radius of pupil. Rchief : chief ray position in object space.
* Uchief : chief ray direction in object space; oriented towards lens
* epOrigin: origin of Exit Pupil. epNorm : normal vector of Exit Pupil at current gaze-direction
* Rvertex : vertex sphere of system. lens : system lens being traced
* od : object distance from front surface. pow : calculated infinitesimal power of pupil ray
* Modifies: pow, if no error occurs. If an error occurs, pow is INFINITY(*2) for tan(sag) rays
* Purpose: Find pupil ray and calculate infinitesimal power. If the power > 40dpt, this will fail.
* Requires: Uchief oriented towards lens
* Dependencies: lineNorm, crossProduct, setPlaneSurface, exitPupilAxis, lineDirCosCombination, multiplyVector,
* addVector, copyDirCos, copyVector, searchDirection, propagateSameRay, sphereNorm, refract, propagateInGRIN
* subtractVectors,dotProduct, intersectLineDist, propagateRay, magnitude
*****/
void tracePupilRay(double alpha, double beta, epAxisMode axis, double pupilRad, vector *Rchief, dirCos *Uchief,
vector *epOrigin, dirCos *epNorm, vector *Rvertex, lensDef *lens, double od, double *pow) {
int dir;
double searchAngle, searchOffset,theta, pdThresh=0.99, diff, a, b, d, OPL, L;
planeSurf pupil;
vector Rtry, Roffset, epIntersect, epPos, imagePos;
dirCos V3, Utry, norm, epDir, epAxis, dirCos *V1, *V2;
/*Search a circle -- looking for Tangential and Sagittal Rays*/
V1 = lineNorm(Uchief); V2 = crossProduct(Uchief, V1);
setPlaneSurface(epOrigin->x, epOrigin->y, epOrigin->z, epNorm->alpha, epNorm->beta, epNorm->gamma, &pupil);
exitPupilAxis(alpha, beta, DEG, axis, &epAxis);
switch(axis) {
case TAN : dir = 0; break;
case SAG : dir = +1; break;
case TANN: dir = 2; break;
case SAGN: dir = -1; break;}
theta = -atan2(sin(M_PI/180.0*beta), tan(M_PI/180.0*alpha)) + dir*M_PI_2;
searchAngle = 0.000001;
searchOffset = 0.001;
/* Try the guessTheta angle first. If it works, then the subsequent
 * search-loop is skipped. If it doesn't, then do the search like normal.
 */
if(-od > INFINITY) { /*Infinite Object : Objects to left of front surface*/
lineDirCosCombination(V1, V2, theta, &V3, RAD); copyDirCosToVector(&V3, &Rtry);
multiplyVector(searchOffset, &Rtry); addVector(Rchief, &Rtry); copyDirCos(Uchief, &Utry);}
else { /*Finite Object*/
copyVector(Rchief, &Rtry); /*Rtry is modified each time; so recopy it*/
searchDirection(Uchief, V1, V2, searchAngle, theta, &Utry);}
/*trace attempted-ray back to exit pupil*/
/*Trace to front Surface*/
L = intersectSurfaceDist(&Rtry, &Utry, &lens->surf1); if(error != NONE) return;
propagateSameRay(&Rtry, &Utry, L);
/*Refract into lens*/
surfaceNorm(&Rtry, &lens->surf1, &norm);
refract(&Utry, &norm, 1.0, index(&lens->grin, &Rtry), &Utry); if(error != NONE) return;
/*Trace to back Surface*/
propagateInGRIN(&Rtry, &Utry, lens, &OPL);
/*Refract out of Lens*/
surfaceNorm(&Rtry, &lens->surf2, &norm);
refract(&Utry, &norm, index(&lens->grin, &Rtry), 1.0, &Utry); if(error != NONE) return;
/*Trace to pupil*/
propagateSameRay(&Rtry, &Utry, intersectPlaneDist(&Rtry, &Utry, &pupil));
/*Calculate intersection vector within exit pupil*/

```

```

subtractVectors(&Rtry, epOrigin, &epIntersect);
normalizeVector(&epIntersect); copyVector(&Rtry, &epPos); copyDirCos(&Utry, &epDir);
/*If the eye-pupil is non-zero, trace finite rays through pupil edge*/
if(pupilRad != 0.0 && error == NONE) {
    searchOffset = pupilRad;
    linearDirCosCombination(V1, V2, theta, (dirCosPtr)&Roffset, RAD);
    diff = 0.0;
    do { searchOffset -= diff;
        copyVector(&Roffset, &Rtry); multiplyVector(searchOffset, &Rtry);
        ctor(Rchief, &Rtry); copyDirCos(Uchief, &Utry);
        /*trace attempted-ray back to exit pupil*/
        L = intersectSurfaceDist(&Rtry, &Utry, &lens->surf1); if(error != NONE) return;
        propagateSameRay(&Rtry, &Utry, L); surfaceNorm(&Rtry, &lens->surf1, &norm);
        refract(&Utry, &norm, 1.0, index(&lens->grin, &Rtry), &Utry); if(error != NONE) return;
        propagateInGRIN(&Rtry, &Utry, lens, &OPL); if(error != NONE) continue;
        surfaceNorm(&Rtry, &lens->surf2, &norm);
        refract(&Utry, &norm, index(&lens->grin, &Rtry), 1.0, &Utry); if(error != NONE) return;
        if(error != NONE) continue;
        propagateSameRay(&Rtry, &Utry, intersectPlaneDist(&Rtry, &Utry, &pupil));
        /*calculate intersection vector within exit pupil*/
        subtractVectors(&Rtry, epOrigin, &epIntersect);
        diff = magnitude(&epIntersect) - pupilRad; while(1.0-fabs(diff/pupilRad) < pdThresh);
        copyVector(&Rtry, &epPos);
        copyDirCos(&Utry, &epDir);}
    free(V1); free(V2);
    /*compute local power*/
    /*Image Position : find intersection with O.A. and chief ray*/
    intersectLineDist(epOrigin, epNorm, &epPos, &epDir, &a, &b, &d);
    if(2*a != a) {
        propagateRay(epOrigin, epNorm, a, &imagePos);
        imagePos.x -= Rvertex->x; imagePos.y -= Rvertex->y; imagePos.z -= Rvertex->z;
        *pow = SIGN(imagePos.z) / magnitude(&imagePos);}
    else
        error = NUMERICAL;
    return;}
/*****
* Name: setLensRadii
* Input: l : lens. R1 : radius of first surface. R2 : radius of second surface
* Modifies: l : set the radii of both surfaces; and their origins
* Purpose: Set the radii of curvature of both surfaces. Takes care of necessary bookkeeping
* Requires: l allocated & defined
*****/
void setLensRadii(lensDef *l, double R1, double R2) {
    l->surf1.sph.radius = R1; l->surf2.sph.radius = R2; l->surf1.sph.origin.z = R1; l->surf2.sph.origin.z = R2+l->t;
    return;}
void forceLensRadius1(lensDef *l, double R1) {l->surf1.sph.radius = R1; l->surf1.sph.origin.z = R1; return;}
void forceLensRadius2(lensDef *l, double R2) {l->surf2.sph.radius = R2; l->surf2.sph.origin.z = R2+l->t; return;}
/*****
* Name: setLensRadius1
* Input: l : lens. R2 : radius of second surface
* Modifies: l (sets radii of both surfaces)
* Purpose: Calculate front surface radius, given back surface radius. Uses lens. power, R2, N2
*           to calculate R1 necessary to maintain the power
* Requires: l & l->grin allocated & defined (including l->power)
* Dependencies: newVector, index
*****/
void setLensRadius1(lensDef *l, double R2) {
    double R1, f, n, n2, t;
    vector *v;
    switch(l->grin.mode) {
        case G_R : if(l->grin.np.nx > 1) n2 = l->grin.np.zc[2]; break;
        case G_RSQ : if(l->grin.np.nx > 0) n2 = l->grin.np.zc[1]; break;
        default: n2 = 0; break;}
    f = l->power; t = l->t; v = newVector(0,0,0); n = index(&l->grin, v);
    if(n2 == 0) R1 = ((n-1.)*(n*(R2 + t) + t*(R2*f-1.)))/(n*(n-1.+R2*f));
    else {
        double a, b, c;
        a = sqrt(SIGN(n2)*n2/2./n);
        b = ((n-1.)*(2*n*R2*a*cos(2*t*a) + (n-1.+R2*f)*sin(2*t*a)));
        c = (2*n*a*(-(n-1.+R2*f)*cos(2*t*a) + 2*n*R2*a*sin(2*t*a)));
        R1 = -b/c;}
    l->surf1.sph.radius = R1; l->surf2.sph.radius = R2;
    l->surf1.sph.origin.z = R1; l->surf2.sph.origin.z = R2+t;
    return;}
/*****
* Name: setLensRadius2
* Input: l : lens. R1 : radius of first surface
* Modifies: l (sets radii of both surfaces)
* Purpose: Calculate the back surface radius, given the front surface radius. It uses the lens
*           power, R1, and N2 to calculate R2 necessary to maintain the desired power
* Requires: l & l->grin allocated & defined (including l->power)
* Dependencies: newVector, index
*****/
void setLensRadius2(lensDef *l, double R1) {
    double R2, f, n, n2, t;
    vector *v;
    switch(l->grin.mode) {
        case G_R : if(l->grin.np.nx > 1) n2 = l->grin.np.zc[2]; break;
        case G_RSQ : if(l->grin.np.nx > 0) n2 = l->grin.np.zc[1]; break;

```

```

        default: n2 = 0; break;}
    f = l->power; t = l->t; v = newVector(0,0,0); n = index(&l->grin, v);
    if(n2 == 0) R2 = ((n-1.)*(n*(R1-t) + t))/(n*n - t*f + n*(t*f-R1*f-1.));
    else {
        double a, b, c;
        a = sqrt(SIGN(n2)*2*n2/n);
        b = ((n-1.)*(n*R1*a*cos(t*a) - (n-1.)*sin(t*a)));
        c = (n*a*(n-1.-R1*f)*cos(t*a) + (n*n*R1*a*a - f + n*f)*sin(t*a));
        R2 = b/c;
    }
    l->surf1.sph.radius = R1; l->surf2.sph.radius = R2;
    l->surf1.sph.origin.z = R1; l->surf2.sph.origin.z = R2+t;
    return;}
/*****
* Name:    updateLensRadius2
* Input:  l : lens
* Modifies: l (sets radii of back surface)
* Purpose: Calculate the back surface radius, using the lens information already set. It uses
*          the lens power, R1, and N2 to calculate R2 necessary to maintain the desired power
* Requires: l & l->grin allocated & defined (including l->power)
* Dependencies: newVector, index
*****/
void updateLensRadius2(lensDef *l) {
    double R1, R2, f, n, n2, t;
    vector *v;
    R1 = l->surf1.sph.radius;
    switch(l->grin.mode) {
        case G_R : if(l->grin.np.nx > 1) n2 = l->grin.np.zc[2]; break;
        case G_RSQ : if(l->grin.np.nx > 0) n2 = l->grin.np.zc[1]; break;
        default: n2 = 0; break;}
    f = l->power; t = l->t; v = newVector(0,0,0); n = index(&l->grin, v);
    if(n2 == 0) R2 = ((n-1.)*(n*(R1-t) + t))/(n*n - t*f + n*(t*f-R1*f-1.));
    else {
        double a, b, c;
        a = sqrt(SIGN(n2)*2*n2/n);
        b = ((n-1.)*(n*R1*a*cos(t*a) - (n-1.)*sin(t*a)));
        c = (n*a*(n-1.-R1*f)*cos(t*a) + (n*n*R1*a*a - f + n*f)*sin(t*a));
        R2 = b/c;
    }
    l->surf1.sph.radius = R1; l->surf2.sph.radius = R2;
    l->surf1.sph.origin.z = R1; l->surf2.sph.origin.z = R2+t;
    return;}
/*****
* Name:    backFocalLength
* Input:  l : lens parameters
* Returns: Back focal length of lens
* Purpose: Calculate the BFL of thick lens, given surface radii, thickness, index, & N2 index
* Requires: Nothing
*****/
double backFocalLength(lensDef *l) {
    double a, b, alp;
    double R1, R2, t, n, n2, bf1;
    vector *v;
    R1 = l->surf1.sph.radius; R2 = l->surf2.sph.radius;
    v = newVector(0,0,0); n = index(&l->grin, v); t = l->t; n2 = 0;
    switch(l->grin.mode) {
        case G_R : if(l->grin.np.nx > 1) n2 = l->grin.np.zc[2]; break;
        case G_RSQ : if(l->grin.np.nx > 0) n2 = l->grin.np.zc[1]; break;}
    if(n2 == 0) {a = 1. - t/n*(n-1)/R1; b = (n-1)*(1./R1 - 1./R2 + t/n*(n-1.)/R1/R2); bf1 = a/b;}
    else {
        alp = sqrt(SIGN(n2)*2*(n2/n)); a = -cos(alp*t) - (n-1)*sin(alp*t)/(n*alp*R1);
        b = (n-1)*(1./R1 - 1./R2)*cos(alp*t) - ((-2*n*n2 + (n-1)*(n-1)/(R1*R2))*sin(alp*t))/(n*alp);
        bf1 = a/b;}
    return bf1;}
/*****
* Name:    backNodalPoint
* Input:  l : lens parameters
* Returns: Back nodal point (relative to origin) of lens
* Purpose: Calculate N' of thick lens, given its surface radii, thickness, index, & N2 index
* Requires: l is allocated & defined
*****/
double backNodalPoint(lensDef *l) {
    double R1, R2, t, n, n2, nz, alp;
    vector *v;
    R1 = l->surf1.sph.radius;
    R2 = l->surf2.sph.radius;
    v = newVector(0,0,0);
    n = index(&l->grin, v);
    t = l->t;
    n2 = 0;
    switch(l->grin.mode) {
        case G_R : if(l->grin.np.nx > 1) n2 = l->grin.np.zc[2]; break;
        case G_RSQ : if(l->grin.np.nx > 0) n2 = l->grin.np.zc[1]; break;}
    if(n2 == 0) {
        nz = (R2*t)/(n*(R1 - R2 - t) + t); /*distance relative to back surface*/
        nz += t; /*distance relative to origin*/
    }
    else {
        alp = sqrt(SIGN(n2)*(n2/2./n));
        nz = 2*R2*sin(t*alp)*((n-1)*cos(t*alp) + 2*n*R1*alp*sin(t*alp));
        nz /= 2*(n-1)*n*(R1-R2)*alp*cos(2*t*alp) - (1. - 2*n + n*n*(1. + 4*R1*R2*alp*alp))*sin(2*t*alp);
        nz += t;}
    return nz;}

```

```

/*****
* Name:      setTraceRange
* Input:    lens : lens parameters. sys : system parameters
* Modifies: sys->maxAlphaCount, minAlphaCount, maxBetaCount, minBetaCount, alphaOffset, betaOffset
* Purpose:  Sets the number of gaze angle loop steps along the alpha & beta axes given the
*          traceDim[] values. If a calculated count range exceeds the max angle (from the SA)
*          then the max angle value is used. The traceDim values are used as follows:
*          traceDim[0] -> Min Alpha Range. traceDim[1] -> Max Alpha Range
*          traceDim[2] -> Min Beta Range. traceDim[3] -> Max Beta Range
*          It also sets the angle offset. e.g., tracing 1°..9° w/ dAngle 2° requires 1° offset
* Requires: sys & lens are allocated and defined
*****/
void setTraceRange(lensDef *lens, sysDef *sys) {
    double maxAngle, maxCount, count; /*Maximum Gaze Angle, Max Gaze Angle Loop count, Temp Loop Count*/
    maxAngle = 180.0/M_PI * atan(lens->surf2.sa / sys->q); maxCount = (int)(maxAngle / sys->dAngle);
    /*alpha min*/
    count = (int)(sys->traceDim[0] / sys->dAngle);
    sys->minAlphaCount = (int)SIGN(count)*MIN(abs(count), abs(maxCount));
    sys->alphaOffset = sys->traceDim[0] - sys->minAlphaCount*sys->dAngle;
    /*alpha max*/
    count = (int)(sys->traceDim[1] / sys->dAngle);
    sys->maxAlphaCount = (int)SIGN(count)*MIN(abs(count), abs(maxCount));
    /*beta min*/
    count = (int)(sys->traceDim[2] / sys->dAngle);
    sys->minBetaCount = (int)SIGN(count)*MIN(abs(count), abs(maxCount));
    sys->betaOffset = sys->traceDim[2] - sys->minBetaCount*sys->dAngle;
    /*beta max*/
    count = (int)(sys->traceDim[3] / sys->dAngle);
    sys->maxBetaCount = (int)SIGN(count)*MIN(abs(count), abs(maxCount));
    return;
}
/*****
* Name:      traceRay
* Input:    lens : Lens defn. struct with the lens info. sys : System defn. struct with the system info
*          object : object plane to trace to. alpha : Vertical gaze angle at which to trace ray
*          beta : Horizontal gaze angle at which. T, Tn : Tangential power above & below chief ray
*          S, Sn : Sagittal power right & left of chief ray
* Modifies: Calculated powers assigned to T, Tn, S, Sn. If error occurs, powers set to INFINITY(*2) for Tan
(Sag) rays
* Purpose:  Traces a single ray at gaze direction (alpha, beta)
* Requires: lens, sys, object be allocated & defined
* Dependencies: setVector, exitPupilNorm, copyDirCos, propagateRay, propagateSameRay, aperture, sphereNorm,
*          refract, propagateInGRIN, intersectSphereDist, reverseDirection, tracePupilRayInfinity
*****/
void traceRay(lensDef *lens, sysDef *sys, planeSurf *object, double alpha, double beta, double *T,
             double *S, double *Tn, double *Sn, dirCos *oray) {
    vector Rchief, Rvertex, epOrigin; /*Chief Ray Positio, Vertex Sphere intersection, Exit Pupil Origin*/
    dirCos Uchief, norm, epNorm; /*Chief Ray Slope, Normal vector for surfaces, Normal vector for Exit Pupil*/
    double L; /*temp variable for ray distances*/
    errorCode e1, e2, e3, e4; /*Error codes from Pupil Ray Search*/
    e1 = e2 = e3 = e4 = error;
    /*initially, if an error occurs, it is an aperture error*/
    *T = -2*INFINITY; *Tn = -2*INFINITY;
    *S = -4*INFINITY; *Sn = -4*INFINITY;
    /*Place Chief Ray at center of rotation of eye*/
    setVector(0.0, 0.0, sys->q+lens->t, &Rchief);
    /*Rotate exit pupil according to Gaze Direction and find chief ray direction*/
    exitPupilNorm(alpha, beta, DEG, &epNorm); reverseGamma(&epNorm); copyDirCos(&epNorm, &Uchief);
    propagateRay(&Rchief, &epNorm, sys->p, &epOrigin);
    /*Determine intersection with vertex sphereSurf*/
    L = intersectSphereDist(&Rchief, &Uchief, sys->vertex); if(error != NONE) return;
    propagateRay(&Rchief, &Uchief, L, &Rvertex);
    /*Find intersection with back surface of lens*/
    L = intersectSurfaceDist(&Rchief, &Uchief, &lens->surf2); if(error != NONE) return;
    propagateSameRay(&Rchief, &Uchief, L);
    if(error != NONE) return;
    /*Traces ray if it is within aperture of lens*/
    aperture(&Rchief, &lens->surf2);
    if(error != NONE) return;
    /*all errors are ray-failure errors; assign lens powers default values*/
    *T = INFINITY; *Tn = INFINITY; *S = 2*INFINITY; *Sn = 2*INFINITY;
    /*Refract at back surface into lens*/
    surfaceNorm(&Rchief, &lens->surf2, &norm);
    refract(&Uchief, &norm, 1.0, index(&lens->grin, &Rchief), &Uchief); if(error != NONE) return;
    /*Propagate to front surface*/
    propagateInGRIN(&Rchief, &Uchief, lens, &L); if(error != NONE) return;
    /*Refract at front surface out of lens*/
    surfaceNorm(&Rchief, &lens->surf1, &norm);
    refract(&Uchief, &norm, index(&lens->grin, &Rchief), 1.0, &Uchief); if(error != NONE) return;
    /*Propagate to object point*/
    if(-object->zi >= INFINITY) {
        planeSurf *tp = newPlaneSurface(0.0,-10.0,0.0,1);
        propagateSameRay(&Rchief, &Uchief, intersectPlaneDist(&Rchief, &Uchief, tp));
    } else propagateSameRay(&Rchief, &Uchief, intersectPlaneDist(&Rchief, &Uchief, object));
    reverseDirection(&Uchief); oray->beta = Uchief.beta;
    /*Find infinitesimal power*/
    tracePupilRay(alpha,beta,TAN,sys->pRad, &Rchief, &Uchief, &epOrigin, &epNorm, &Rvertex, lens, object->zi,
T);
    e1 = error;
    if(beta > 0)

```

```

    tracePupilRay(alpha,beta,SAG,sys->pRad, &Rchief, &Uchief, &epOrigin, &epNorm, &Rvertex, lens, object->zi,
S);
    else
    tracePupilRay(alpha,beta,SAGN,sys->pRad,&Rchief,&Uchief, &epOrigin, &epNorm, &Rvertex, lens, object->zi,
S);
    e2 = error;
    error = MAX(MAX(MAX(MAX(error, e1), e2), e3), e4);
    return;}
/*****
* Name:    traceLens
* Input:  lens : Lens defn. struct with the lens info. sys : System defn. struct with the system info
* Modifies: Assigns T & S powers to system arrays
* Purpose: Performs ray-tracing the lens over the entire trace region
* Requires: lens, sys be allocated & defined
* Dependencies: generateObjectPlanes, traceRay
*****/
void traceLens(lensDef *lens, sysDef *sys, FILE *fp_debug) {
    planeSurf *objectPlanes;/*Object Planes for various gaze angles*/
    double alpha, beta, T, S, Tn, Sn;/*infinitesimal Tan & Sag powers (1/mm)*/
    int alphaCount, betaCount, cols, minA, maxA, minB, maxB;
    /*assign temp variables*/
    minA = sys->minAlphaCount;maxA = sys->maxAlphaCount;minB = sys->minBetaCount;maxB = sys->maxBetaCount;
    cols = (maxA - minA + 1);
    /*Object Plane - vertical plane some distance from front of lens*/
    objectPlanes = generateObjectPlanes(-sys->object, sys->dAngle, minA, maxA);
    /*Loop over all permitted Gaze Angles*/
    for(betaCount = minB; betaCount <= maxB; betaCount ++){
        for(alphaCount = minA; alphaCount <= maxA; alphaCount ++){
            dirCos oray;
            alpha = alphaCount*sys->dAngle+sys->alphaOffset;beta = betaCount * sys->dAngle + sys-
>betaOffset;
            /*Trace the ray at the current gaze-angle and determine its infinitesimal powers*/
            traceRay(lens, sys, &objectPlanes[alphaCount-minA], alpha, beta, &T, &S, &Tn, &Sn, &oray);
            RESET_ERROR
            /*Assign the powers to the storage arrays*/
            *(sys->T + (alphaCount-minA) + (betaCount-minB)*cols) = T;
            *(sys->S + (alphaCount-minA) + (betaCount-minB)*cols) = S;
            *(sys->D + (alphaCount-minA) + (betaCount-minB)*cols) = oray.beta;}}
        free(objectPlanes);
    return;}

```

G.10. Surface.c

```

/*****
* Name:    setPlaneSurface
* Input:  x, y, z : plane origin. alpha, beta gamma : plane normal (direction cosines). p : plane to define
* Modifies: p : defines it according to the input parameters
* Purpose: Defines the plane parameters
* Requires: p must be allocated
*****/
void setPlaneSurface(double x, double y, double z, double alpha, double beta, double gamma, planeSurf *p) {
    p->mx = alpha;p->xi = x;p->my = beta;p->yi = y;p->mz = gamma;p->zi = z;return;}
/*****
* Name:    newPlaneSurface
* Input:  x, y, z : plane origin. alpha, beta, gamma : plane normal (direction cosines)
* Returns: New plane
* Modifies: Nothin
* Purpose: Allocates & defines the plane
* Requires: Nothing
*****/
planeSurf* newPlaneSurface(double x, double y, double z, double alpha, double beta, double gamma) {
    planeSurf *p;
    p = (planeSurf *)malloc(sizeof(planeSurf));
    p->mx = alpha;p->xi = x;p->my = beta;p->yi = y;p->mz = gamma;p->zi = z;return p;}
/*****
* Name:    newSphereSurface
* Input:  x, y, z: sphere origin. radius : sphere radius. k : sphere conic constant
* Returns: New sphere
* Purpose: Allocates & defines the sphere
* Requires: Nothing
*****/
sphereSurf* newSphereSurface(double x, double y, double z, double r, int ropt, int na, double *a, int *aopt) {
    sphereSurf *s;
    s = (sphereSurf *)malloc(sizeof(sphereSurf));
    s->radius = r;s->ropt = ropt;
    if(na == 0) {s->na = 1; s->a = (double *)calloc(1, sizeof(double)); s->a[0] = 0; s->aopt = aopt;}
    else {s->a = a; s->aopt = aopt;}
    s->origin.x = x; s->origin.y = y; s->origin.z = z;
    return s;}
/*****
* Name:    sphereSurfaceCondition
* Input:  R : Ray position. U : Ray direction. s : sphere to check
          L : distance from R
* Returns: Distance along z-axis of propagated ray from surface. This is:
          > 0 if ray is to the right of the surface. < 0 if ray is to the left of the surface
          INFINITY if the value can't be computed (ray outside of surface)
*****/

```

```

* Purpose: Determines whether the ray, propogated a distance L, will be to the left or the right of the surface
* Requires: R, U, s allocated & defined. surf->a[0] must be defined
*****/
double sphereSurfaceCondition(vector *R, dirCos *U, sphereSurf *s, double L) {
    int i;
    double r, rv, rsq, c, d, asp, dist;
    vector Rp;
    propagateRay(R, U, L, &Rp);
    rsq = Rp.x*Rp.x + Rp.y*Rp.y; r = sqrt(rsq);
    c = 1.0/s->radius; d = 1.0 - c*c*rsq*(s->a[0] + 1.0);
    if(d < 0) { error = NAN;dist = INFINITY;}
    else {rv = rsq*rsq;asp = 0; /*compute aspheric contribution*/
        for(i=1; i<s->na; i++) {asp += rv*s->a[i];rv *= r;}
        dist = (Rp.z - (s->origin.z - s->radius)) - c*rsq / (1.0 + sqrt(d)) - asp;}
    return dist;}
/*****
* Name: planeSurfaceCondition
* Input: R : Ray position. U : Ray direction. s : plane to check. L : distance from R
* Returns: Distance along z-axis of propagated ray from surface. This is:
* > 0 if ray is to the right of the surface. < 0 if ray is to the left of the surface
* Purpose: Determines whether the ray, propogated a distance L, will be to the left or the right of the surface
* Requires: R, U, s allocated & defined
*****/
double planeSurfaceCondition(vector *R, dirCos *U, planeSurf *s, double L) {
    vector Rp;
    propagateRay(R, U, L, &Rp);
    return s->mx*(Rp.x - s->xi) + s->my*(Rp.y - s->yi) + s->mz*(Rp.z - s->zi);}
/*****
* Name: planeNorm
* Input: R : Ray position. surf : surface to find normal of. norm : normal of surface at R
* Modifies: norm : set to be the surface normal at R
* Purpose: Determine surface normal vector at a given point
* Requires: R, surf, norm allocated & defined; R is a point on the surface
*****/
void planeNorm(vector *R, planeSurf *surf, dirCos *norm) {
    double a, b, g, mag;
    a = surf->mx;b = surf->my;g = surf->mz;mag = sqrt(a*a + b*b + g*g);
    norm->alpha = a / mag;norm->beta = b / mag;norm->gamma = g / mag;
    return;}
/*****
* Name: sphereNorm
* Input: R : Ray position. surf : surface to find normal of. norm : normal of surface at R
* Modifies: norm : set to be the surface normal at R
* Purpose: Determine surface normal vector at a given point
* Requires: R, surf, norm allocated & defined; R is a point on the surface. surf->a[0] must be defined
*****/
void sphereNorm(vector *R, sphereSurf *surf, dirCos *norm) {
    int i;
    double a, b, g, c, k, mag, asp, x, y, r, rsq, rv;
    c = 1.0/surf->radius; k = surf->a[0];
    x = R->x;y = R->y;rsq = x*x + y*y; r = sqrt(rsq);
    /*Conic Terms*/
    a = 1.0 - c*c*(rsq*(1.0 + k));
    if(a < 0) {error = NAN;return;} else a = -c*x / sqrt(a);
    /*Aspheric terms*/
    asp = 0; rv = rsq;
    for(i=1; i<surf->na; i++) { asp += (i+3)*surf->a[i]*rv; rv *= r;//rsq}
    asp *= x; a -= asp;
    /*Conic Terms*/
    b = 1.0 - c*c*(rsq*(1.0 + k));
    if(b < 0) {error = NAN;return;} else b = -c*y/sqrt(b);
    /*Aspheric terms : d/dy r^n = n y r^(n-2)*/
    asp = 0; rv = rsq;
    for(i=1; i<surf->na; i++) { asp += (i+3)*surf->a[i]*rv; rv *= r;//rsq}
    asp *= y; b -= asp; g = +1.0; mag = sqrt(a*a + b*b + g*g);
    norm->alpha = a / mag; norm->beta = b / mag; norm->gamma = g / mag;
    return;}
/*****
* Name: generateObjectPlanes
* Input: dist : object distance. dAngle : delta-angle for vertical gaze directions
* dist < 0 all objects planes have INF for distance. dist > 0 object planes at "convenient distance" for a
PAL
* maxstep, minstep: minstep-maxstep+1 = number of object planes needed
* -minstep * dAngle = min gaze angle. maxstep * dAngle = max gaze angle
* Returns: pointer to array of planes
* Purpose: Allocates & defines array of 2*maxstep+1 planes.
* The ith plane corresponds to gaze-angle: (i - (gaze angle)/dAngle) * dAngle
* Requires: Nothing
* Dependencies: setPlaneSurface
*****/
planeSurf* generateObjectPlanes(double dist, double dAngle, int minstep, int maxstep) {
    int i;
    planeSurf *p;
    p = (planeSurf *)calloc(sizeof(planeSurf), maxstep-minstep+1);
    if(dist < 0)/*Finite & Infinite Objects*/
    for(i=0; i<maxstep-minstep+1; i++) setPlaneSurface(0,0,dist, 0,0,1, &p[i]);
    else for(i=0; i<maxstep-minstep+1; i++) { /*Convenient Distances*/
        double alpha = dAngle*(i+minstep);
        if(alpha >= 0) setPlaneSurface(0,0,-INFINITY, 0,0,1, &p[i]);
        else if(alpha >= -25.0) setPlaneSurface(0,0,10.0/alpha * 1000, 0,0,1, &p[i]);
    }
}

```

```

        else setPlaneSurface(0,0,-1/2.5 * 1000, 0,0,1, &p[i]);
    return p;}
/*****
* Name:    setObjectDistances
* Input:  lens : lens defn. struct with the lens info. sys : System defn. struct with the system info
* Modifies: Assigns values to od (object distance) system array
* Purpose: Calculates the object distances used in the ray trace and assigns the values to storage array
* Requires: lens, sys be allocated & defined
* Dependencies: generateObjectPlanes
*****/
void setObjectDistances(lensDefn *lens, sysDefn *sys) {
    planeSurf *objectPlanes;/*Object Planes for various gaze angles*/
    int alphaCount, minA,maxA;
    minA = sys->minAlphaCount; maxA = sys->maxAlphaCount;
    /*Object Plane - vertical plane some distance from front of lens*/
    objectPlanes = generateObjectPlanes(-sys->object, sys->dAngle, minA, maxA);
    for(alphaCount = -minA; alphaCount <= maxA; alphaCount ++){
        *(sys->od + alphaCount-minA) = -1.0/(objectPlanes+alphaCount-minA)->zi;
        free(objectPlanes);
    }
    return;}

```

G.11. Vector.c

```

/*****
* Name:    dotProduct
* Input:  U1 : vector 1. U2 : vector 2
* Returns: The vector dot product of U1 and U2
* Purpose: Calculate the vector dot product of two direction cosine vectors
* Requires: Nothing
*****/
double dotProduct(dirCos *U1, dirCos *U2) {return U1->alpha * U2->alpha + U1->beta * U2->beta + U1->gamma * U2->gamma;}
/*****
* Name:    dotProductV
* Input:  R1 : vector 1. R2 : vector 2
* Returns: The vector dot product of R1 and R2
* Purpose: Calculate the vector dot product of two vectors
* Requires: Nothing
*****/
double dotProductV(vector *R1, vector *R2) {return R1->x * R2->x + R1->y * R2->y + R1->z * R2->z;}
/*****
* Name:    magnitudeZPlane
* Input:  R : input vector
* Returns: The magnitude of R projected into the x-y plane
* Purpose: Calculate the vector magnitude of the input vector projected onto an x-y plane
* Requires: Nothing
*****/
double magnitudeZPlane(vector *R) {double x, y; x = R->x; y = R->y; return sqrt(x*x + y*y);}
/*****
* Name:    magnitude
* Input:  R : input vector
* Returns: The magnitude of R
* Purpose: Calculate the vector magnitude of the input vector
* Requires: Nothing
*****/
double magnitude(vector *R) {double x, y, z;x = R->x;y = R->y;z = R->z;return sqrt(x*x + y*y + z*z);}
/*****
* Name:    magnitudeSquared
* Input:  R : input vector
* Returns: The squared magnitude of R
* Purpose: Calculate the vector squared magnitude of the input vector
* Requires: Nothing
*****/
double magnitudeSquared(vector *R) {double x, y, z;x = R->x;y = R->y;z = R->z;return x*x + y*y + z*z;}
/*****
* Name:    normalizeVector
* Input:  R : input vector
* Modifies: R
* Purpose: Normalizes the vector
* Requires: Nothing
*****/
void normalizeVector(vector *R) {
    double x, y, z, mag, small = 0.001, big=1000.0;
    x = R->x;y = R->y;z = R->z;
    /*try and avoid round-off error when normalizing small vectors*/
    if( x<small || y<small || z<small ) {x*=big;y*=big;z*=big;}
    mag = sqrt(x*x + y*y + z*z); R->x = x/mag;R->y = y/mag;R->z = z/mag;
    return;}
/*****
* Name:    newVector
* Input:  x, y, z: components of new vector
* Returns: Pointer to new vector
* Modifies: Return vector
* Purpose: Allocate & define a vector
* Requires: Nothing
*****/

```

```

vector *newVector(double x, double y, double z) {
    vector *r: r = (vectorPtr)malloc(sizeof(vector)); r->x = x;r->y = y;r->z = z;return r;}
/*****
* Name: setVector
* Input: x, y, z: components of vector. R : vector to modify
* Modifies: Return vector
* Purpose: Modify the parameters of an allocated vector
* Requires: R must be allocated
*****/
void setVector(double x, double y, double z, vector *R) {R->x = x;R->y = y;R->z = z;return;}
/*****
* Name: copyVector
* Input: Rorig : vector to copy
* Rcopy : vector to copy to (does not need to be defined)
* Modifies: Rcopy : sets its parameters equal to those of Rorig
* Purpose: Copy one vector to another
* Requires: Rorig must be allocated & defined. Rcopy must be allocated
*****/
void copyVector(vector *Rorig, vector *Rcopy) {Rcopy->x = Rorig->x;Rcopy->y = Rorig->y;Rcopy->z = Rorig->z;return;}
/*****
* Name: copyVectorToDirCos
* Input: Rorig : vector 1. Ucopy : vector 2
* Modifies: Vcopy
* Purpose: Copy one direction cosine vector (Uorig) to a vector (Rcopy)
* Requires: Uorig, Ucopy must be allocated & defined
*****/
void copyVectorToDirCos(vector *Rorig, dirCos *Ucopy) {
    Ucopy->alpha = Rorig->x;Ucopy->beta = Rorig->y;Ucopy->gamma = Rorig->z;return;}
/*****
* Name: copyDirCosToVector
* Input: Uorig : vector 1. Rcopy : vector 2
* Modifies: Vcopy
* Purpose: Copy one direction cosine vector (Uorig) to a vector (Rcopy)
* Requires: Uorig, Ucopy must be allocated & defined
*****/
void copyDirCosToVector(dirCos *Uorig, vector *Rcopy) {
    Rcopy->x = Uorig->alpha;Rcopy->y = Uorig->beta;Rcopy->z = Uorig->gamma;return;}
/*****
* Name: multiplyVector
* Input: factor : multiplicative factor. Rres : vector to be multiplied
* Modifies: Rres : multiplies its parameters by factor
* Purpose: Scalar multiplication of a vector
* Requires: Rres must be allocated & defined
*****/
void multiplyVector(double factor, vector *Rres) {
    Rres->x *= factor;Rres->y *= factor;Rres->z *= factor;return;}
/*****
* Name: addVector
* Input: Rmod : vector to add. Rres : vector to be added to
* Modifies: Rres : adds Rmod to Rres
* Purpose: Vector addition: Rmod += Rres
* Requires: Rres & Rmod must be allocated & defined
*****/
void addVector(vector *Rmod, vector *Rres) {
    Rres->x += Rmod->x;Rres->y += Rmod->y;Rres->z += Rmod->z;return;}
/*****
* Name: subtractVector
* Input: Rmod : vector to subtract. Rres : vector to be subtracted from
* Modifies: Rres : subtracts Rmod from Rres
* Purpose: Vector subtraction: Rmod -= Rres
* Requires: Rres & Rmod must be allocated & defined
*****/
void subtractVector(vector *Rmod, vector *Rres) {
    Rres->x -= Rmod->x;Rres->y -= Rmod->y;Rres->z -= Rmod->z;return;}
/*****
* Name: addVectors
* Input: Ra : vector 1. Rb : vector 2. Rres : vector result
* Modifies: Rres
* Purpose: Vector addition result is in Rres
* Requires: Ra, Rb, Rres must be allocated & defined
*****/
void addVectors(vector *Ra, vector *Rb, vector *Rres) {
    Rres->x = Ra->x + Rb->x;Rres->y = Ra->y + Rb->y;Rres->z = Ra->z + Rb->z;}
/*****
* Name: subtractVectors
* Input: Ra : vector 1. Rb : vector 2. Rres : vector result
* Modifies: Rres : subtraction result is in Rres
* Purpose: Vector addition: Rres = Ra - Rb
* Requires: Ra, Rb, Rres must be allocated & defined
*****/
void subtractVectors(vector *Ra, vector *Rb, vector *Rres) {
    Rres->x = Ra->x - Rb->x;Rres->y = Ra->y - Rb->y;Rres->z = Ra->z - Rb->z;}
/*****
* Name: copyDirCos
* Input: Uorig : vector 1. Ucopy : vector 2
* Modifies: Ucopy
* Purpose: Copy one direction cosine vector (Uorig) to another (Ucopy)
* Requires: Uorig, Ucopy must be allocated & defined
*****/

```

```

void copyDirCos(dirCos *Uorig, dirCos *Ucopy) {
    Ucopy->alpha = Uorig->alpha;Ucopy->beta = Uorig->beta;Ucopy->gamma = Uorig->gamma; return;}
/*****
* Name:    newDirCos
* Input:  alpha : direction cosine parameter. beta : direction cosine parameter. gamma : direction cosine
parameter
*
* mode : which angle definition is being used for alpha, beta, gamma
* Returns: Pointer to new direction cosine vector
* Purpose: Define a new direction cosine ray
* Requires: Nothing
*****/
dirCos *newDirCos(double alpha, double beta, double gamma, angleMode mode) {
    dirCos *u;
    u = (dirCosPtr)malloc(sizeof(dirCos));
    switch(mode) {
        case DC:u->alpha = alpha;u->beta = beta;u->gamma = gamma;break;
        case RAD:u->alpha = cos(alpha);u->beta = cos(beta);u->gamma = cos(gamma);break;
        case DEG:u->alpha = cos(M_PI/180.0*alpha);u->beta = cos(M_PI/180.0*beta);u->gamma =
cos(M_PI/180.0*gamma);break;}
    return u;}
/*****
* Name:    setDirCos
* Input:  a : direction cosine parameter. b : direction cosine parameter. g : direction cosine parameter
* mode: which angle definition is being used for alpha, beta, gamma. U : direction cosine vector to modify
* Modifies: U
* Purpose: Change the parameters of a direction cosine ray
* Requires: U be allocated
*****/
void setDirCos(double a, double b, double g, angleMode mode, dirCos *U) {
    int partial=0, relz=0;
    double tnz, ctz, den;
    if(mode >= RELZ) { relz = 1; mode -= RELZ;}
    if(mode >= PARTIAL) { partial = 1; mode -= PARTIAL;}
    if(mode == TANX) { a = atan(a);b = atan(b);g = atan(g);mode = RAD;}
    switch(mode) {
        case DC: U->alpha = a;U->beta = b;U->gamma = g;break;
        case RAD: if(relz) {a = M_PI_2 - a;b = M_PI_2 - b;g = M_PI_2 - g;}
            U->alpha = cos(a);U->beta = cos(b);U->gamma = cos(g); break;
        case DEG:
            if(relz) {a = 90.0 - a;b = 90.0 - b;g = 90.0 - g;}
            U->alpha = cos(M_PI/180.0*a);U->beta = cos(M_PI/180.0*b);U->gamma = cos(M_PI/180.0*g);break;
        case CV:
            g *= M_PI/180.; tnz = tan(g); ctz = 1.0/tnz; den = sqrt( a*a + b*b + ctz*ctz );
            U->alpha = a / den;U->beta = b / den;U->gamma = ctz / den;break;}
    if(partial) {U->gamma = 0; g = 1. - U->alpha*U->alpha - U->beta*U->beta;if(g<0) g=0;U->gamma = sqrt(g);}
    normalizeDirCos(U);
    return;}
/*****
* Name:    normalizeDirCos
* Input:  U : direction cosine
* Modifies: U
* Purpose: Normalizes the direction cosine
* Requires: Nothing
*****/
void normalizeDirCos(dirCos *U) {
    double a, b, g, mag, rab, rag, rbg, small = 1e-10, big=1e5;
    a = U->alpha;b = U->beta;g = U->gamma;
    /*try and avoid round-off error when normalizing small vectors*/
    if( (a<small && a!= 0) || (b<small && b!= 0) || (g<small && g!= 0) ) {a *= big;b *= big;g *= big;}
    rab = fabs(a/b);rag = fabs(a/g);rbg = fabs(b/g);
    if(rab > 1e15) b=0;if(rag > 1e15) g=0;if(rbg > 1e15) g=0;
    if(1./rab > 1e15) a=0;if(1./rag > 1e15) a=0;if(1./rbg > 1e15) b=0;
    mag = sqrt(a*a + b*b + g*g); U->alpha = a/mag;U->beta = b/mag;U->gamma = g/mag;
    return;}
/*****
* Name:    crossProduct
* Input:  U1 : direction cosine vector 1. U2 : direction cosine vector 2
* Returns: Direction cosine vector equal to the cross product of U1 & U2
* Purpose: Allocates & defines a direction cosine vector U3. U3 = U1 x U2
* Requires: U1, U2 allocated and defined
* Dependencies: Nothing
*****/
dirCos *crossProduct(dirCos *U1, dirCos *U2) {
    double a, b, g, mag;
    dirCos *U3;
    U3 = (dirCosPtr)malloc(sizeof(dirCos));
    a = U1->beta *U2->gamma - U1->gamma*U2->beta;
    b = -U1->alpha*U2->gamma + U1->gamma*U2->alpha;
    g = U1->alpha*U2->beta - U1->beta *U2->alpha;
    mag = sqrt(a*a + b*b + g*g); U3->alpha = a / mag;U3->beta = b / mag;U3->gamma = g / mag;
    return U3;}
/*****
* Name:    reverseDirection
* Input:  U : direction cosine vector to reverse
* Modifies: U
* Purpose: Modifies U to be anti-parallel with its initial direction
* Requires: U allocated and defined
* Dependencies: Nothing
*****/
void reverseDirection(dirCos *U) {U->alpha = -U->alpha;U->beta = -U->beta;U->gamma = -U->gamma;return;}

```

```

/*****
* Name: reverseGamma
* Input: U : gamma value of direction cosine vector reversed
* Modifies: U
* Purpose: Modifies U to be anti-parallel with its initial direction
* Requires: U allocated and defined
* Dependencies: Nothing
*****/
void reverseGamma(dirCos *U) {U->gamma = -U->gamma;return;}
/*****
* Name: lineNorm
* Input: U : direction cosine vector to find a normal to
* Returns: Normalized direction cosine vector perpendicular to U
* Purpose: Allocates & defines a direction cosine vector perpendicular to U
* Requires: U
* Dependencies: Nothing
*****/
dirCos *lineNorm(dirCos *U) {
    dirCos *norm;
    double b, g;
    norm = (dirCosPtr)malloc(sizeof(dirCos));
    b = U->beta;g = U->gamma;norm->alpha = 0;norm->beta = g/sqrt(b*b + g*g);norm->gamma = -b/sqrt(b*b + g*g);
    return norm;}
/*****
* Name: linearDirCosCombination
* Input: V1 : axis 1. V2 : axis 2 (perpendicular to V1). theta : angle relative to V1.
* V3 : vector to be a linear combination of V1 & V2. mode : angle mode of theta
* Modifies: V3
* Purpose: Define V3 to be a linear combination of V1 & V2. In other words: Given the
* coordinate axes V1 & V2, define V3 to be the vector at angle theta relative to V1
* Requires: V1, V2 are perpendicular. V3 allocated
* Dependencies: Nothing
*****/
void linearDirCosCombination(dirCos *V1, dirCos *V2, double theta, dirCos *V3, angleMode mode) {
    double a, b, g;
    if(mode == DEG) theta *= M_PI/180.0;
    a = cos(theta)*V1->alpha + sin(theta)*V2->alpha;
    b = cos(theta)*V1->beta + sin(theta)*V2->beta;
    g = cos(theta)*V1->gamma + sin(theta)*V2->gamma;
    V3->alpha = a;V3->beta = b;V3->gamma = g;normalizeDirCos(V3);
    return;}

```