Mie Scattering

Version 2-6-3

1. Equations for Mie scattering.

The index of refraction m of the sphere may be complex,

$$m = n(1 - i\kappa)$$

The imaginary part of the complex refractive index $n\kappa$ is the damping factor while κ is called the index of absorption or the index of attenuation. Note that the sign of the imaginary part of the index of refraction is negative. The complex index of refraction may also be written in terms of the conductivity σ, the dielectric constant ε and the circular frequency ω as

$$m = \sqrt{\varepsilon - i\frac{4\pi\sigma}{\omega}}$$

The other important parmeter governing scattering by a sphere is the size parameter x of the sphere, which is given by

$$x = 2\pi a/\lambda$$

Sometimes the value ρ is used to indicate the size of the sphere and it is defined as

$$\rho = 2x(m - 1)$$

and really only makes sense if the sphere does not absorb light.

The absorption coefficient from Beer's law is defined as

$$I = I_0 \exp(-\mu_a z)$$

and thus

$$\mu_a = \frac{4\pi n\kappa}{\lambda_0} = \frac{4\pi\kappa}{\lambda}$$

where λ_0 is the wavelength in a vacuum [Kerker, p. 15].

Now to reprise some nomenclature. The extinction efficiency may be separated into

$$Q_{\text{ext}} = Q_{\text{sca}} + Q_{\text{abs}}$$

where Q_{sca} is the scattering efficiency and Q_{abs} is the absorption efficiency. Typically Q_{sca} and Q_{ext} are determined by the Mie scattering program and Q_{abs} is obtained by subtraction.

The radiation pressure is given by

$$Q_{\text{pr}} = Q_{\text{ext}} - gQ_{\text{sca}}$$

The pressure exerted on the particle of cross-sectional area πr_0^2 is

$$P = \frac{F}{\pi r_0^2} = \frac{Q_{\text{ext}}}{c}$$

were c is the velocity of the radiation in the medium [Kerker, p. 94].

The relation between the efficiency factor for scattering and the cross section for scattering are obtained by dividing by the actual geometrical cross section

$$Q_{\text{sca}} = \frac{C_{\text{sca}}}{\pi r_0^2}$$

where r_0 is the radius of the sphere.

The scattering cross section may be related to the transmission of a beam through a dispersion of scatterers of equal size. For ρ particles per unit volume, the attenuation due to scattering is

$$-\frac{dI}{dx} = \rho C_{\text{sca}} I$$

The transmission is

$$T = I/I_0 = \exp(-\rho C_{\text{sca}} x) = \exp(-\mu_s x)$$

or

$$\mu_s = \rho C_{\text{sca}} = \rho\pi r_0^2 Q_{\text{sca}}$$

[Kerker, p. 38].

2. Double Array Routines.
Here are a bunch of routines to deal arrays of doubles. This file will create three files when run through `ctangle` — the usual `.c` and `.h`, as well as a testing driver.

3. Here, then, is an overview of document structure

⟨ `mie_array.c` 3 ⟩ ≡
```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <float.h>
#include "mie_array.h"
```
⟨ Definition for *array_error* 7 ⟩
⟨ Definition for *new_darray* 9 ⟩
⟨ Definition for *free_darray* 11 ⟩
⟨ Definition for *copy_darray* 13 ⟩
⟨ Definition for *set_darray* 15 ⟩
⟨ Definition for *min_max_darray* 17 ⟩
⟨ Definition for *sort_darray* 20 ⟩
⟨ Definition for *print_darray* 23 ⟩

4. Each function has its prototype exported to a header file.

⟨ `mie_array.h` 4 ⟩ ≡
⟨ Prototype for *new_darray* 8 ⟩;
⟨ Prototype for *free_darray* 10 ⟩;
⟨ Prototype for *copy_darray* 12 ⟩;
⟨ Prototype for *set_darray* 14 ⟩;
⟨ Prototype for *min_max_darray* 16 ⟩;
⟨ Prototype for *sort_darray* 19 ⟩;
⟨ Prototype for *print_darray* 22 ⟩;

5. Allocation.

6. A simple error routine.

⟨ Prototype for *array_error* 6 ⟩ ≡
```
static void array_error(char *s)
```
This code is used in section 7.

7. ⟨ Definition for *array_error* 7 ⟩ ≡
```
⟨ Prototype for array_error 6 ⟩
{
  printf("Array␣--␣%s\n", s);
  exit(1);
}
```
This code is used in section 3.

8. Double array routines.

⟨ Prototype for *new_darray* 8 ⟩ ≡
```
double *new_darray(long size)
```
This code is used in sections 4 and 9.

9. ⟨ Definition for *new_darray* 9 ⟩ ≡
```
⟨ Prototype for new_darray 8 ⟩
{
  double *a;
  if (size ≤ 0) array_error("Non-positive␣double␣array␣size␣chosen");
  a = (double *) calloc(sizeof(double), (unsigned long) size + 2);
  if (a ≡ Λ) array_error("Insufficient␣space␣to␣allocate␣array");
  a[0] = DBL_MIN;
  a[size + 1] = DBL_MAX;
  return a + 1;
}
```
This code is used in section 3.

10. ⟨ Prototype for *free_darray* 10 ⟩ ≡
```
void free_darray(double *a)
```
This code is used in sections 4 and 11.

11. ⟨ Definition for *free_darray* 11 ⟩ ≡
```
⟨ Prototype for free_darray 10 ⟩
{
  if (a ≠ Λ) free(a − 1);
}
```
This code is used in section 3.

12. This allocates a new double array data structure and copies the contents of a into it.

⟨ Prototype for *copy_darray* 12 ⟩ ≡
```
double *copy_darray(double *a, long size)
```
This code is used in sections 4 and 13.

13. ⟨ Definition for *copy_darray* 13 ⟩ ≡
```
⟨ Prototype for copy_darray 12 ⟩
{
  double *b = Λ;
  if (a ≡ Λ) return b;
  b = new_darray(size + 2);
  if (b ≡ Λ) array_error("Insufficient␣space␣to␣duplicate␣array");
  memcpy(b, a − 1, sizeof(double) * (size + 2));
  return b + 1;
}
```
This code is used in section 3.

14. This sets all the entries in the array $a[\,]$ to x.

⟨ Prototype for *set_darray* 14 ⟩ ≡
```
void set_darray(double *a, long size, double x)
```
This code is used in sections 4 and 15.

15. ⟨ Definition for *set_darray* 15 ⟩ ≡

```
⟨ Prototype for set_darray 14 ⟩
{
  long j;
  if (a ≡ Λ) array_error("Attempt␣to␣set␣elements␣in␣a␣NULL␣array");
  for (j = 0; j < size; j++) a[j] = x;
}
```

This code is used in section 3.

16. *min_max_darray* finds the minimum and maximum of the array *a*.

⟨ Prototype for *min_max_darray* 16 ⟩ ≡

```
void min_max_darray(double *a, long size, double *min, double *max)
```

This code is used in sections 4 and 17.

17. ⟨ Definition for *min_max_darray* 17 ⟩ ≡

```
⟨ Prototype for min_max_darray 16 ⟩
{
  long j;
  if (a ≡ Λ) array_error("A␣NULL␣array␣does␣not␣have␣a␣min␣or␣max");
  if (size ≡ 0) array_error("An␣array␣with␣no␣elements␣does␣not␣have␣a␣min␣or␣max");
  *min = a[0];
  *max = *min;
  for (j = 1; j < size; j++) {
    if (a[j] > *max) *max = a[j];
    if (a[j] < *min) *min = a[j];
  }
}
```

This code is used in section 3.

18. Sorting.

19. *sort_darray* will sort an array a into ascending numerical order using the Heapsort algorithm. Adapted to work with zero-based arrays from *Numerical Recipes*. This could certainly use some sprucing up, but I can't quite seem to figure out how to do it. It is kinda tricky.

```
⟨ Prototype for sort_darray 19 ⟩ ≡
  void sort_darray(double *a, long size)
```

This code is used in sections 4 and 20.

20.

```
⟨ Definition for sort_darray 20 ⟩ ≡
  ⟨ Prototype for sort_darray 19 ⟩
  {
    long i, ir, j, l;
    double aa;
    if (a ≡ Λ) array_error("Can't␣sort␣a␣NULL␣array");
    if (size < 2) return;
    l = (size ≫ 1) + 1;
    ir = size;
    for ( ; ; ) {
      if (l > 1) {
        aa = a[--l − 1];
      }
      else {
        aa = a[ir − 1];
        a[ir − 1] = a[0];
        if (--ir ≡ 1) {
          a[0] = aa;
          break;
        }
      }
      i = l;
      j = l + l;
      while (j ≤ ir) {
        if (j < ir ∧ a[j − 1] < a[j]) j++;
        if (aa < a[j − 1]) {
          a[i − 1] = a[j − 1];
          i = j;
          j ≪= 1;
        }
        else j = ir + 1;
      }
      a[i − 1] = aa;
    }
  }
```

This code is used in section 3.

21. Printing.

22. *print_darray* prints the elements of the array a from *ilow* through *ihigh*.

```
⟨ Prototype for print_darray 22 ⟩ ≡
  void print_darray(double *a, long size, long ilow, long ihigh)
```

This code is used in sections 4 and 23.

23.

```
⟨ Definition for print_darray 23 ⟩ ≡
  ⟨ Prototype for print_darray 22 ⟩
  {
    long j;
    if (a ≡ Λ) array_error("Can't␣print␣a␣NULL␣array");
    if (ilow < 0) ilow = 0;
    if (ihigh > size − 1) ihigh = size − 1;
    for (j = ilow; j ≤ ihigh; j++) printf("x[%ld]=␣%-10.5g␣\n", j, a[j]);
  }
```

This code is used in section 3.

24. Testing.

25. Here are driver routines to test the routines in this file.

```
⟨ test_mie_array.c 25 ⟩ ≡
#include <stdio.h>
#include "mie_array.h"
  void main( )
  {
    double *x;
    double *y;
    long i, size;
    double min, max;
    size = 10;
    printf("starting\n");
    fflush(stdout);
    x = new_darray(size);
    ⟨ Test Set Routine 26 ⟩
    ⟨ Test Copy Routine 27 ⟩
    ⟨ Test Sort Routine 28 ⟩
    ⟨ Test Min/Max Routine 29 ⟩
    printf("done\n");
    fflush(stdout);
  }
```

26. ⟨ Test Set Routine 26 ⟩ ≡

```
  printf("Testing␣set_darray\n");
  printf("All␣entries␣should␣be␣3.0\n");
  set_darray(x, size, 3.0);
  print_darray(x, size, 0, size − 1);
  fflush(stdout);
  printf("\n");
```

This code is used in section 25.

27. ⟨ Test Copy Routine 27 ⟩ ≡

```
  printf("Testing␣copy_darray\n");
  for (i = 0; i < size; i++) x[i] = size − i;
  printf("The␣original␣vector␣was:\n");
  print_darray(x, size, 0, size − 1);
  fflush(stdout);
  y = copy_darray(x, size);
  printf("The␣copied␣vector␣is:\n");
  print_darray(y, size, 0, size − 1);
  fflush(stdout);
  printf("\n");
```

This code is used in section 25.

28. ⟨ Test Sort Routine 28 ⟩ ≡

```
printf("Testing␣sort_darray\n");
printf("The␣original␣vector␣is:\n");
print_darray(x, size, 0, size − 1);
fflush(stdout);
sort_darray(x, size);
printf("The␣sorted␣vector␣is:\n");
print_darray(x, size, 0, size − 1);
fflush(stdout);
printf("\n");
```

This code is used in section 25.

29. ⟨ Test Min/Max Routine 29 ⟩ ≡

```
printf("Testing␣min_max_darray\n");
min_max_darray(x, size, &min, &max);
printf("min=%g␣␣max=%g\n", min, max);
fflush(stdout);
printf("\n");
```

This code is used in section 25.

30. Complex Number Routines.

Here are a bunch of routines to deal with complex numbers. The functions are pretty straightforward, but there are some subtle points in some of the functions. This could use some more error checking.

Changed names to not conflict with c++ routines

31. Here, then, is an overview of document structure

⟨ mie_complex.c 31 ⟩ ≡
```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <float.h>
#include "mie_complex.h"
```
⟨ Definition for *complex_error* 35 ⟩
⟨ Definition for *c_set* 37 ⟩
⟨ Definition for *c_polarset* 39 ⟩
⟨ Definition for *c_abs* 41 ⟩
⟨ Definition for *c_arg* 45 ⟩
⟨ Definition for *c_norm* 47 ⟩
⟨ Definition for *c_sqrt* 49 ⟩
⟨ Definition for *c_sqr* 51 ⟩
⟨ Definition for *c_inv* 53 ⟩
⟨ Definition for *c_conj* 43 ⟩
⟨ Definition for *c_add* 56 ⟩
⟨ Definition for *c_sub* 58 ⟩
⟨ Definition for *c_mul* 60 ⟩
⟨ Definition for *c_div* 62 ⟩
⟨ Definition for *c_rdiv* 64 ⟩
⟨ Definition for *c_rmul* 66 ⟩
⟨ Definition for *c_sadd* 71 ⟩
⟨ Definition for *c_sdiv* 73 ⟩
⟨ Definition for *c_smul* 69 ⟩
⟨ Definition for *c_sin* 76 ⟩
⟨ Definition for *c_cos* 78 ⟩
⟨ Definition for *c_tan* 80 ⟩
⟨ Definition for *c_asin* 82 ⟩
⟨ Definition for *c_acos* 84 ⟩
⟨ Definition for *c_atan* 86 ⟩
⟨ Definition for *c_sinh* 91 ⟩
⟨ Definition for *c_cosh* 89 ⟩
⟨ Definition for *c_tanh* 93 ⟩
⟨ Definition for *c_atanh* 95 ⟩
⟨ Definition for *c_asinh* 97 ⟩
⟨ Definition for *c_exp* 100 ⟩
⟨ Definition for *c_log* 102 ⟩
⟨ Definition for *c_log10* 104 ⟩
⟨ Definition for *new_carray* 107 ⟩
⟨ Definition for *free_carray* 109 ⟩
⟨ Definition for *copy_carray* 111 ⟩
⟨ Definition for *set_carray* 113 ⟩

32. Each function has its prototype exported to a header file along with a couple of structure definitions.

```
⟨ mie_complex.h 32 ⟩ ≡
  struct c_complex {
    double re;
    double im;
  };
  ⟨ Prototype for c_set 36 ⟩;
  ⟨ Prototype for c_polarset 38 ⟩;
  ⟨ Prototype for c_abs 40 ⟩;
  ⟨ Prototype for c_arg 44 ⟩;
  ⟨ Prototype for c_sqr 50 ⟩;
  ⟨ Prototype for c_conj 42 ⟩;
  ⟨ Prototype for c_norm 46 ⟩;
  ⟨ Prototype for c_sqrt 48 ⟩;
  ⟨ Prototype for c_inv 52 ⟩;
  ⟨ Prototype for c_add 55 ⟩;
  ⟨ Prototype for c_sub 57 ⟩;
  ⟨ Prototype for c_mul 59 ⟩;
  ⟨ Prototype for c_div 61 ⟩;
  ⟨ Prototype for c_rdiv 63 ⟩;
  ⟨ Prototype for c_rmul 65 ⟩;
  ⟨ Prototype for c_sadd 70 ⟩;
  ⟨ Prototype for c_sdiv 72 ⟩;
  ⟨ Prototype for c_smul 68 ⟩;
  ⟨ Prototype for c_sin 75 ⟩;
  ⟨ Prototype for c_cos 77 ⟩;
  ⟨ Prototype for c_tan 79 ⟩;
  ⟨ Prototype for c_asin 81 ⟩;
  ⟨ Prototype for c_acos 83 ⟩;
  ⟨ Prototype for c_atan 85 ⟩;
  ⟨ Prototype for c_sinh 90 ⟩;
  ⟨ Prototype for c_cosh 88 ⟩;
  ⟨ Prototype for c_tanh 92 ⟩;
  ⟨ Prototype for c_atanh 94 ⟩;
  ⟨ Prototype for c_asinh 96 ⟩;
  ⟨ Prototype for c_exp 99 ⟩;
  ⟨ Prototype for c_log 101 ⟩;
  ⟨ Prototype for c_log10 103 ⟩;
  ⟨ Prototype for new_carray 106 ⟩;
  ⟨ Prototype for free_carray 108 ⟩;
  ⟨ Prototype for copy_carray 110 ⟩;
  ⟨ Prototype for set_carray 112 ⟩;
```

33. Basic routines.

34. A simple error routine.

⟨ Prototype for *complex_error* 34 ⟩ ≡
```
static void complex_error(char *s)
```
This code is used in section 35.

35. ⟨ Definition for *complex_error* 35 ⟩ ≡
```
⟨ Prototype for complex_error 34 ⟩
{
  printf("%s\n", s);
  exit(1);
}
```
This code is used in section 31.

36. This is shorthand for setting a complex number. It just returns a complex equal to $a + bi$

⟨ Prototype for *c_set* 36 ⟩ ≡
```
struct c_complex c_set(double a, double b)
```
This code is used in sections 32 and 37.

37. ⟨ Definition for *c_set* 37 ⟩ ≡
```
⟨ Prototype for c_set 36 ⟩
{
  struct c_complex c;
  c.re = a;
  c.im = b;
  return c;
}
```
This code is used in section 31.

38. A variation on *c_set* in which the complex number is specifed using polar coordinates.

⟨ Prototype for *c_polarset* 38 ⟩ ≡
```
struct c_complex c_polarset(double r, double theta)
```
This code is used in sections 32 and 39.

39. ⟨ Definition for *c_polarset* 39 ⟩ ≡
```
⟨ Prototype for c_polarset 38 ⟩
{
  return c_set(r * cos(theta), r * sin(theta));
}
```
This code is used in section 31.

40. This routine returns the absolute value of a complex number $\sqrt{zz^*}$. To avoid unnecessary loss of accuracy as explained in §5.4 of *Numerical Recipes in C*

⟨ Prototype for *c_abs* 40 ⟩ ≡
```
double c_abs(struct c_complex z)
```
This code is used in sections 32 and 41.

41. ⟨ Definition for *c_abs* 41 ⟩ ≡

```
⟨ Prototype for c_abs 40 ⟩
{
  double x, y, temp;
  x = fabs(z.re);
  y = fabs(z.im);
  if (x ≡ 0.0) return y;
  if (y ≡ 0.0) return x;
  if (x > y) {
    temp = y/x;
    return (x * sqrt(1.0 + temp * temp));
  }
  temp = x/y;
  return (y * sqrt(1.0 + temp * temp));
}
```

This code is used in section 31.

42. Returns the conjugate of the complex number z

⟨ Prototype for *c_conj* 42 ⟩ ≡

```
struct c_complex c_conj(struct c_complex z)
```

This code is used in sections 32 and 43.

43. ⟨ Definition for *c_conj* 43 ⟩ ≡

```
⟨ Prototype for c_conj 42 ⟩
{
  return c_set(z.re, -z.im);
}
```

This code is used in section 31.

44. ⟨ Prototype for *c_arg* 44 ⟩ ≡

```
double c_arg(struct c_complex z)
```

This code is used in sections 32 and 45.

45. ⟨ Definition for *c_arg* 45 ⟩ ≡

```
⟨ Prototype for c_arg 44 ⟩
{
  return atan2(z.im, z.re);
}
```

This code is used in section 31.

46. Returns the square of the modulus of the complex number zz^*

⟨ Prototype for *c_norm* 46 ⟩ ≡

```
double c_norm(struct c_complex z)
```

This code is used in sections 32 and 47.

47. ⟨ Definition for *c_norm* 47 ⟩ ≡

```
⟨ Prototype for c_norm 46 ⟩
{
  return (z.re * z.re + z.im * z.im);
}
```

This code is used in section 31.

48. ⟨ Prototype for *c_sqrt* 48 ⟩ ≡
```
struct c_complex c_sqrt(struct c_complex z)
```
This code is used in sections 32 and 49.

49. ⟨ Definition for *c_sqrt* 49 ⟩ ≡
```
⟨ Prototype for c_sqrt 48 ⟩
{
  double a, b;
  if ((z.re ≡ 0.0) ∧ (z.im ≡ 0.0)) return c_set(0.0, 0.0);
  a = sqrt((fabs(z.re) + c_abs(z)) ∗ 0.5);
  if (z.re ≥ 0) b = z.im/(a + a);
  else {
    b = z.im < 0 ? −a : a;
    a = z.im/(b + b);
  }
  return c_set(a, b);
}
```
This code is used in section 31.

50. Returns the product of a complex number with itself $z \cdot z$. If you want $z \cdot z^*$ then use *c_norm*.

⟨ Prototype for *c_sqr* 50 ⟩ ≡
```
struct c_complex c_sqr(struct c_complex z)
```
This code is used in sections 32 and 51.

51. ⟨ Definition for *c_sqr* 51 ⟩ ≡
```
⟨ Prototype for c_sqr 50 ⟩
{
  return c_mul(z, z);
}
```
This code is used in section 31.

52. Returns the reciprocal of z.

⟨ Prototype for *c_inv* 52 ⟩ ≡
```
struct c_complex c_inv(struct c_complex w)
```
This code is used in sections 32 and 53.

53. ⟨ Definition for *c_inv* 53 ⟩ ≡
```
⟨ Prototype for c_inv 52 ⟩
{
  double r, d;
  if ((w.re ≡ 0) ∧ (w.im ≡ 0)) complex_error("Attempt to invert 0+0i");
  if (fabs(w.re) ≥ fabs(w.im)) {
    r = w.im/w.re;
    d = 1/(w.re + r ∗ w.im);
    return c_set(d, −r ∗ d);
  }
  r = w.re/w.im;
  d = 1/(w.im + r ∗ w.re);
  return c_set(r ∗ d, −d);
}
```
This code is used in section 31.

54. Two complex numbers.

55. Returns the sum of the two complex numbers a and b

```
⟨ Prototype for c_add 55 ⟩ ≡
  struct c_complex c_add(struct c_complex z, struct c_complex w)
```

This code is used in sections 32 and 56.

56.

```
⟨ Definition for c_add 56 ⟩ ≡
  ⟨ Prototype for c_add 55 ⟩
  {
    struct c_complex c;
    c.im = z.im + w.im;
    c.re = z.re + w.re;
    return c;
  }
```

This code is used in section 31.

57. Returns the difference of two complex numbers $z - w$

```
⟨ Prototype for c_sub 57 ⟩ ≡
  struct c_complex c_sub(struct c_complex z, struct c_complex w)
```

This code is used in sections 32 and 58.

58.

```
⟨ Definition for c_sub 58 ⟩ ≡
  ⟨ Prototype for c_sub 57 ⟩
  {
    struct c_complex c;
    c.im = z.im - w.im;
    c.re = z.re - w.re;
    return c;
  }
```

This code is used in section 31.

59. Returns the product of two complex numbers $z \cdot w$

```
⟨ Prototype for c_mul 59 ⟩ ≡
  struct c_complex c_mul(struct c_complex z, struct c_complex w)
```

This code is used in sections 32 and 60.

60.

```
⟨ Definition for c_mul 60 ⟩ ≡
  ⟨ Prototype for c_mul 59 ⟩
  {
    struct c_complex c;
    c.re = z.re * w.re - z.im * w.im;
    c.im = z.im * w.re + z.re * w.im;
    return c;
  }
```

This code is used in section 31.

61. Returns the quotient of two complex numbers z/w see §5.4 of *Numerical Recipes in C*

⟨ Prototype for *c_div* 61 ⟩ ≡
```
struct c_complex c_div(struct c_complex z, struct c_complex w)
```
This code is used in sections 32 and 62.

62. ⟨ Definition for *c_div* 62 ⟩ ≡
```
⟨ Prototype for c_div 61 ⟩
{
  struct c_complex c;
  double r, denom;
  if ((w.re ≡ 0) ∧ (w.im ≡ 0)) complex_error("Attempt to divide by 0+0i");
  if (fabs(w.re) ≥ fabs(w.im)) {
    r = w.im/w.re;
    denom = w.re + r * w.im;
    c.re = (z.re + r * z.im)/denom;
    c.im = (z.im − r * z.re)/denom;
  }
  else {
    r = w.re/w.im;
    denom = w.im + r * w.re;
    c.re = (z.re * r + z.im)/denom;
    c.im = (z.im * r − z.re)/denom;
  }
  return c;
}
```
This code is used in section 31.

63. Returns the real part of the quotient of two complex numbers $\mathrm{Re}(z/w)$. Note how this is a special case of *c_div* above

⟨ Prototype for *c_rdiv* 63 ⟩ ≡
```
double c_rdiv(struct c_complex z, struct c_complex w)
```
This code is used in sections 32 and 64.

64. ⟨ Definition for *c_rdiv* 64 ⟩ ≡
```
⟨ Prototype for c_rdiv 63 ⟩
{
  double r, c, denom;
  if ((w.re ≡ 0) ∧ (w.im ≡ 0)) complex_error("Attempt to find real part with divisor 0+0i");
  if (fabs(w.re) ≥ fabs(w.im)) {
    r = w.im/w.re;
    denom = w.re + r * w.im;
    c = (z.re + r * z.im)/denom;
  }
  else {
    r = w.re/w.im;
    denom = w.im + r * w.re;
    c = (z.re * r + z.im)/denom;
  }
  return c;
}
```
This code is used in section 31.

65. Returns the real part of the product of two complex numbers $\mathrm{Re}(z \cdot w)$

```
⟨ Prototype for c_rmul 65 ⟩ ≡
  double c_rmul(struct c_complex z, struct c_complex w)
```

This code is used in sections 32 and 66.

66.

```
⟨ Definition for c_rmul 66 ⟩ ≡
  ⟨ Prototype for c_rmul 65 ⟩
  {
    return z.re * w.re - z.im * w.im;
  }
```

This code is used in section 31.

67. A scalar and a complex number.

68. Returns the product of a scalar with a complex number

⟨ Prototype for *c_smul* 68 ⟩ ≡
```
struct c_complex c_smul(double x, struct c_complex z)
```
This code is used in sections 32 and 69.

69. ⟨ Definition for *c_smul* 69 ⟩ ≡
```
⟨ Prototype for c_smul 68 ⟩
{
  struct c_complex c;
  c.re = z.re * x;
  c.im = z.im * x;
  return c;
}
```
This code is used in section 31.

70. Returns the sum of a scalar and a complex number

⟨ Prototype for *c_sadd* 70 ⟩ ≡
```
struct c_complex c_sadd(double x, struct c_complex z)
```
This code is used in sections 32 and 71.

71. ⟨ Definition for *c_sadd* 71 ⟩ ≡
```
⟨ Prototype for c_sadd 70 ⟩
{
  struct c_complex c;
  c.re = x + z.re;
  c.im = z.im;
  return c;
}
```
This code is used in section 31.

72. Returns the quotient of real number by a complex number z. Again a special case of *c_div*

⟨ Prototype for *c_sdiv* 72 ⟩ ≡
```
struct c_complex c_sdiv(double x, struct c_complex w)
```
This code is used in sections 32 and 73.

73. ⟨ Definition for *c_sdiv* 73 ⟩ ≡

```
⟨ Prototype for c_sdiv 72 ⟩
{
  struct c_complex c;
  double r, factor;
  if ((w.re ≡ 0) ∧ (w.im ≡ 0)) complex_error("Attempt to divide scalar by 0+0i");
  if (fabs(w.re) ≥ fabs(w.im)) {
    r = w.im/w.re;
    factor = x/(w.re + r * w.im);
    c.re = factor;
    c.im = −r * factor;
  }
  else {
    r = w.re/w.im;
    factor = x/(w.im + r * w.re);
    c.im = −factor;
    c.re = r * factor;
  }
  return c;
}
```

This code is used in section 31.

74. Trigonometric Functions.

75. The complex sine.

⟨ Prototype for *c_sin* 75 ⟩ ≡
```
struct c_complex c_sin(struct c_complex z)
```
This code is used in sections 32 and 76.

76. ⟨ Definition for *c_sin* 76 ⟩ ≡
```
⟨ Prototype for c_sin 75 ⟩
{
  return c_set(sin(z.re) * cosh(z.im), cos(z.re) * sinh(z.im));
}
```
This code is used in section 31.

77. The complex cosine.

⟨ Prototype for *c_cos* 77 ⟩ ≡
```
struct c_complex c_cos(struct c_complex z)
```
This code is used in sections 32 and 78.

78. ⟨ Definition for *c_cos* 78 ⟩ ≡
```
⟨ Prototype for c_cos 77 ⟩
{
  return c_set(cos(z.re) * cosh(z.im), -(sin(z.re) * sinh(z.im)));
}
```
This code is used in section 31.

79. The complex tangent.

$$\tan(a+bi) = \frac{\sin 2a + i \sinh 2b}{\cos 2a + \cosh 2b}$$

or

$$\tan(a+bi) = \frac{2\sin 2a + i\exp(2b) - i\exp(-2b)}{2\cos 2a + \exp(2b) + \exp(-2b)}$$

it is easy to see that if $2b$ is large, then problems arise.

The number `DBL_MAX_10_EXP` is the value c such that 10^c can be represented by a double precision variable. Now we are interested in the maximum exponential, one would just multiply c by $\ln 10 = 2.3$ to get such an exponential. This could then be compared against the value of $2b$ to figure out when an approximation should be used. Slightly more conservatively, one could just test to see when

$$2b > 2c$$

and adjust accordingly.

⟨ Prototype for *c_tan* 79 ⟩ ≡
```
struct c_complex c_tan(struct c_complex z)
```
This code is used in sections 32 and 80.

80. ⟨ Definition for *c_tan* 80 ⟩ ≡

```
⟨ Prototype for c_tan 79 ⟩
{
  double t, x, y;
  if (z.im ≡ 0) return c_set(tan(z.re), 0.0);
  if (z.im > DBL_MAX_10_EXP) return c_set(0.0, 1.0);
  if (z.im < −DBL_MAX_10_EXP) return c_set(0.0, −1.0);
  x = 2 ∗ z.re;
  y = 2 ∗ z.im;
  t = cos(x) + cosh(y);
  if (t ≡ 0) complex_error("Complex␣tangent␣is␣infinite");
  return c_set(sin(x)/t, sinh(y)/t);
}
```

This code is used in section 31.

81. The complex inverse sine.

⟨ Prototype for *c_asin* 81 ⟩ ≡

```
struct c_complex c_asin(struct c_complex z)
```

This code is used in sections 32 and 82.

82. ⟨ Definition for *c_asin* 82 ⟩ ≡

```
⟨ Prototype for c_asin 81 ⟩
{
  struct c_complex x;
  x = c_log(c_add(c_set(−z.im, z.re), c_sqrt(c_sub(c_set(1.0, 0.0), c_mul(z, z)))));
  return c_set(x.im, −x.re);
}
```

This code is used in section 31.

83. The complex inverse cosine

⟨ Prototype for *c_acos* 83 ⟩ ≡

```
struct c_complex c_acos(struct c_complex z)
```

This code is used in sections 32 and 84.

84. ⟨ Definition for *c_acos* 84 ⟩ ≡

```
⟨ Prototype for c_acos 83 ⟩
{
  struct c_complex x;
  x = c_log(c_add(z, c_mul(c_set(0.0, 1.0), c_sqrt(c_sub(c_set(1.0, 0.0), c_sqr(z))))));
  return c_set(x.im, −x.re);
}
```

This code is used in section 31.

85. The complex inverse tangent

⟨ Prototype for *c_atan* 85 ⟩ ≡

```
struct c_complex c_atan(struct c_complex z)
```

This code is used in sections 32 and 86.

86. ⟨ Definition for *c_atan* 86 ⟩ ≡

```
⟨ Prototype for c_atan 85 ⟩
{
  struct c_complex x;
  x = c_log(c_div(c_set(z.re, 1 + z.im), c_set(−z.re, 1 − z.im)));
  return c_set(−x.im/2, x.re/2);
}
```

This code is used in section 31.

87. Hyperbolic functions.

88. ⟨ Prototype for *c_cosh* 88 ⟩ ≡
```
struct c_complex c_cosh(struct c_complex z)
```
This code is used in sections 32 and 89.

89. ⟨ Definition for *c_cosh* 89 ⟩ ≡
```
⟨ Prototype for c_cosh 88 ⟩
{
  return c_set(cosh(z.re) * cos(z.im), sinh(z.re) * sin(z.im));
}
```
This code is used in section 31.

90. ⟨ Prototype for *c_sinh* 90 ⟩ ≡
```
struct c_complex c_sinh(struct c_complex z)
```
This code is used in sections 32 and 91.

91. ⟨ Definition for *c_sinh* 91 ⟩ ≡
```
⟨ Prototype for c_sinh 90 ⟩
{
  return c_set(sinh(z.re) * cos(z.im), cosh(z.re) * sin(z.im));
}
```
This code is used in section 31.

92. ⟨ Prototype for *c_tanh* 92 ⟩ ≡
```
struct c_complex c_tanh(struct c_complex z)
```
This code is used in sections 32 and 93.

93. ⟨ Definition for *c_tanh* 93 ⟩ ≡
```
⟨ Prototype for c_tanh 92 ⟩
{
  double x = 2 * z.re;
  double y = 2 * z.im;
  double t = 1.0/(cosh(x) + cos(y));
  return c_set(t * sinh(x), t * sin(y));
}
```
This code is used in section 31.

94. ⟨ Prototype for *c_atanh* 94 ⟩ ≡
```
struct c_complex c_atanh(struct c_complex z)
```
This code is used in sections 32 and 95.

95. ⟨ Definition for *c_atanh* 95 ⟩ ≡
```
⟨ Prototype for c_atanh 94 ⟩
{
  return c_atan(c_set(-z.im, z.re));
}
```
This code is used in section 31.

96. ⟨ Prototype for *c_asinh* 96 ⟩ ≡
```
struct c_complex c_asinh(struct c_complex z)
```
This code is used in sections 32 and 97.

97. ⟨ Definition for *c_asinh* 97 ⟩ ≡

```
⟨ Prototype for c_asinh 96 ⟩
{
  return c_asin(c_set(−z.im, z.re));
}
```

This code is used in section 31.

98. Exponentials and logarithms.

99. ⟨ Prototype for *c_exp* 99 ⟩ ≡
```
struct c_complex c_exp(struct c_complex z)
```
This code is used in sections 32 and 100.

100. ⟨ Definition for *c_exp* 100 ⟩ ≡
```
⟨ Prototype for c_exp 99 ⟩
{
  double x = exp(z.re);
  return c_set(x * cos(z.im), x * sin(z.im));
}
```
This code is used in section 31.

101. ⟨ Prototype for *c_log* 101 ⟩ ≡
```
struct c_complex c_log(struct c_complex z)
```
This code is used in sections 32 and 102.

102. ⟨ Definition for *c_log* 102 ⟩ ≡
```
⟨ Prototype for c_log 101 ⟩
{
  return c_set(log(c_abs(z)), c_arg(z));
}
```
This code is used in section 31.

103. ⟨ Prototype for *c_log10* 103 ⟩ ≡
```
struct c_complex c_log10(struct c_complex z)
```
This code is used in sections 32 and 104.

104. ⟨ Definition for *c_log10* 104 ⟩ ≡
```
⟨ Prototype for c_log10 103 ⟩
{
  return c_set(0.2171472409516259 * log(c_norm(z)), c_arg(z));
}
```
This code is used in section 31.

105. Arrays of complex numbers.
This assumes zero based arrays.

106. ⟨ Prototype for *new_carray* 106 ⟩ ≡
```
struct c_complex *new_carray(long size)
```
This code is used in sections 32 and 107.

107. ⟨ Definition for *new_carray* 107 ⟩ ≡
```
⟨ Prototype for new_carray 106 ⟩
{
  struct c_complex *a;
  if (size ≤ 0) complex_error("Non-positive␣complex␣array␣size␣chosen");
  a = (struct c_complex *) calloc(sizeof(struct c_complex), (unsigned long) size);
  if (a ≡ Λ) complex_error("Can't␣allocate␣complex␣array");
  return a;
}
```
This code is used in section 31.

108. ⟨ Prototype for *free_carray* 108 ⟩ ≡
```
void free_carray(struct c_complex *a)
```
This code is used in sections 32 and 109.

109. ⟨ Definition for *free_carray* 109 ⟩ ≡
```
⟨ Prototype for free_carray 108 ⟩
{
  if (a ≠ Λ) free(a);
}
```
This code is used in section 31.

110. This allocates a new complex array and copies the contents of a into it.

⟨ Prototype for *copy_carray* 110 ⟩ ≡
```
struct c_complex *copy_carray(struct c_complex *a, long size)
```
This code is used in sections 32 and 111.

111. ⟨ Definition for *copy_carray* 111 ⟩ ≡
```
⟨ Prototype for copy_carray 110 ⟩
{
  struct c_complex *b = Λ;
  if (a ≡ Λ) complex_error("Can't␣duplicate␣a␣NULL␣complex␣array");
  b = new_carray(size);
  if (b ≠ Λ) memcpy(b, a, size * sizeof(struct c_complex));
  return b;
}
```
This code is used in section 31.

112. This puts z in all the entries in a complex array.

⟨ Prototype for *set_carray* 112 ⟩ ≡
```
void set_carray(struct c_complex *a, long size, struct c_complex z)
```
This code is used in sections 32 and 113.

113. ⟨ Definition for *set_carray* 113 ⟩ ≡

```
⟨ Prototype for set_carray 112 ⟩
{
  long j;
  if (a ≡ Λ) complex_error("Can't␣operate␣on␣a␣NULL␣complex␣array");
  for (j = 0; j < size; j++) a[j] = z;
}
```

This code is used in section 31.

114. Mie Scattering Algorithms.

This is a Mie scattering implementation. Several resources were used in creating this program. First, the Fortran listing in Bohren and Huffman's book was used. This listing was translated into Pascal and refined using various suggestions by Wiscombe. This version was used for a couple of years and later translated by me into C and then into CWeb with the documentation you see here.

Finally, consider using *ez_Mie* for problems that involve non-absorbing spheres and you don't care about the scattering phase function.

A short to do list includes (1) use Wiscombe's trick to find the scattering functions, (2) add code to deal with near zero entries in the Lentz routine, (3) allow calculation of extinction efficiencies with zero angles.

115. There are seven basic functions that are defined.

⟨ mie.c 115 ⟩ ≡
```
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mie_array.h"
#include "mie_complex.h"
#include "mie.h"
```
⟨ Definition for *mie_error* 119 ⟩
⟨ Definition for *Lentz_Dn* 124 ⟩
⟨ Definition for *Dn_down* 133 ⟩
⟨ Definition for *Dn_up* 130 ⟩
⟨ Definition for *small_Mie* 136 ⟩
⟨ Definition for *small_conducting_Mie* 145 ⟩
⟨ Definition for *Mie* 148 ⟩
⟨ Definition for *ez_Mie* 164 ⟩
⟨ Definition for *ez_Mie_Full* 166 ⟩

116. Only the main function *Mie* is available for calling.

⟨ mie.h 116 ⟩ ≡
```
#define MIE_VERBOSE_ERROR_REPORTING 0
```
⟨ Prototype for *Lentz_Dn* 123 ⟩;
⟨ Prototype for *Dn_down* 132 ⟩;
⟨ Prototype for *Dn_up* 129 ⟩;
⟨ Prototype for *small_Mie* 135 ⟩;
⟨ Prototype for *small_conducting_Mie* 144 ⟩;
⟨ Prototype for *Mie* 147 ⟩;
⟨ Prototype for *ez_Mie* 163 ⟩;
⟨ Prototype for *ez_Mie_Full* 165 ⟩;

117. Some prototypes for the library interface.

⟨ libmie.h 117 ⟩ ≡
⟨ Prototype for *ez_Mie* 163 ⟩;
⟨ Prototype for *ez_Mie_Full* 165 ⟩;

118. A simple error routine that contains the only printf statement used in the program.

⟨ Prototype for *mie_error* 118 ⟩ ≡
static void *mie_error*(**char** ∗*s*, **int** *n*)

This code is used in section 119.

119. ⟨ Definition for *mie_error* 119 ⟩ ≡

```
⟨ Prototype for mie_error 118 ⟩
{
  if (MIE_VERBOSE_ERROR_REPORTING) {
    fprintf(stderr, "Mie Error %d -- %s\n", n, s);
    exit(n);
  }
}
```

This code is used in section 115.

120. The logarithmic derivative D_n.

121. This routine uses a continued fraction method to compute $D_n(z)$ proposed by Lentz.* This method eliminates many weaknesses in previous algorithms using forward recursion.

I should add code to deal with $\alpha_{j,1} \approx 0$.

The logarithmic derivative D_n is defined as

$$D_n = -\frac{n}{z} + \frac{J_{n-1/2}(z)}{J_{n+1/2}(z)}$$

Equation (5) in Lentz's paper can be used to obtain

$$\frac{J_{n-1/2}(z)}{J_{n+1/2}(z)} = \frac{2n+1}{z} + \cfrac{1}{-\frac{2n+3}{z} + \cfrac{1}{\frac{2n+5}{z} + \cfrac{1}{-\frac{2n+7}{z} + \cdots}}}$$

Now if

$$\alpha_{i,j} = [a_i, a_{i-1}, \ldots, a_j] = a_i + \cfrac{1}{a_{i-1} + \cfrac{1}{a_{i-2} + \cdots \cfrac{1}{a_j}}}$$

we seek to create

$$\alpha = \alpha_{1,1}\,\alpha_{2,1} \cdots \alpha_{j,1} \qquad \beta = \alpha_{2,2}\,\alpha_{3,2} \cdots \alpha_{j,2}$$

since Lentz showed that

$$\frac{J_{n-1/2}(z)}{J_{n+1/2}(z)} \approx \frac{\alpha}{\beta}$$

122. The whole goal is to iterate until the α and β are identical to the number of digits desired. Once this is achieved, then use equations this equation and the first equation for the logarithmic derivative to calculate $D_n(z)$.

123. ⟨ Prototype for *Lentz_Dn* 123 ⟩ ≡

```
struct c_complex Lentz_Dn(struct c_complex z, long n)
```

This code is used in sections 116 and 124.

124. ⟨ Definition for *Lentz_Dn* 124 ⟩ ≡

```
⟨ Prototype for Lentz_Dn 123 ⟩
{
  struct c_complex alpha_j1, alpha_j2, zinv, aj;
  struct c_complex alpha, result, ratio, runratio;
  ⟨ Calculate first alpha and beta 125 ⟩
  do ⟨ Calculate next ratio 126 ⟩ while (fabs(c_abs(ratio) − 1.0) > 1 · 10^−12);
  result = c_add(c_sdiv((double) −n, z), runratio);
  return result;
}
```

This code is used in section 115.

* Lentz uses the notation A_n instead of D_n, but I prefer the notation used by Bohren and Huffman.

125. Here I initialize for looping. Of course it is kind of tricky, but what else would you expect. The value of a_j is given by,

$$a_j = (-1)^{j+1} \frac{2n + 2j - 1}{z}$$

The first terms for α and *beta* are

$$\alpha = a_1 \left(a_2 + \frac{1}{a_1} \right) \qquad \beta = a_2$$

⟨ Calculate first *alpha* and *beta* 125 ⟩ ≡

```
zinv = c_sdiv(2.0, z);
alpha = c_smul(n + 0.5, zinv);
aj = c_smul(−n − 1.5, zinv);
alpha_j1 = c_add(aj, c_inv(alpha));
alpha_j2 = aj;
ratio = c_div(alpha_j1, alpha_j2);
runratio = c_mul(alpha, ratio);
```

This code is used in section 124.

126. To calculate the next α and β, I use

$$a_{j+1} = -a_j + (-1)^j \frac{2}{z}$$

to find the next a_j and

$$\alpha_{j+1} = a_j + \frac{1}{\alpha_j}, \qquad \text{and} \qquad \beta_{j+1} = a_j + \frac{1}{\beta_j}$$

and

⟨ Calculate next *ratio* 126 ⟩ ≡

```
{
  aj.re = zinv.re − aj.re;
  aj.im = zinv.im − aj.im;
  alpha_j1 = c_add(c_inv(alpha_j1), aj);
  alpha_j2 = c_add(c_inv(alpha_j2), aj);
  ratio = c_div(alpha_j1, alpha_j2);
  zinv.re ∗= −1;
  zinv.im ∗= −1;
  runratio = c_mul(ratio, runratio);
}
```

This code is used in section 124.

127. D_n **by upward recurrence.**

Calculating the logarithmic derivative $D_n(\rho)$ using the upward recurrence relation,

$$D_n(z) = \frac{1}{n/z - D_{n-1}(z)} - \frac{n}{z}$$

128. To calculate the initial value we must figure out $D_0(z)$. This is

$$D_0(z) = \frac{d}{dz} \ln \psi_0(z) = \frac{d}{dz} \ln \sin(z) = \frac{\cos z}{\sin z}$$

The only tricky part is finding the tangent of a complex number, but this is all stuck in `complex.w`.

Finally, note that the returned array $*D$ is set-up so that $D_n(z) = D[n]$. Therefore the first value for $D_1(z)$ will be found not in $D[0]$, but rather in $D[1]$.

129. ⟨ Prototype for *Dn_up* 129 ⟩ ≡

```
void Dn_up(struct c_complex z, long nstop, struct c_complex *D)
```

This code is used in sections 116 and 130.

130. ⟨ Definition for *Dn_up* 130 ⟩ ≡

```
⟨ Prototype for Dn_up 129 ⟩
{
  struct c_complex zinv, k_over_z;
  long k;
  D[0] = c_inv(c_tan(z));
  zinv = c_inv(z);
  for (k = 1; k < nstop; k++) {
    k_over_z = c_smul((double) k, zinv);
    D[k] = c_sub(c_inv(c_sub(k_over_z, D[k − 1])), k_over_z);
  }
}
```

This code is used in section 115.

131. D_n **by downwards recurrence.**

Start downwards recurrence using Lentz method, then find earlier terms of the logarithmic derivative $D_n(z)$ using the recurrence relation,

$$D_{n-1}(z) = \frac{n}{z} - \frac{1}{D_n(z) + n/z}$$

This is a pretty straightforward procedure.

Finally, note that the returned array $*D$ is set-up so that $D_n(z) = D[n]$. Therefore the first value for $D_1(z)$ will be found not in $D[0]$, but rather in $D[1]$.

132. ⟨ Prototype for *Dn_down* 132 ⟩ ≡

```
void Dn_down(struct c_complex z, long nstop, struct c_complex *D)
```

This code is used in sections 116 and 133.

133. ⟨ Definition for *Dn_down* 133 ⟩ ≡

```
⟨ Prototype for Dn_down 132 ⟩
{
  long k;
  struct c_complex zinv, k_over_z;
  D[nstop − 1] = Lentz_Dn(z, nstop);
  zinv = c_inv(z);
  for (k = nstop − 1; k ≥ 1; k−−) {
    k_over_z = c_smul((double) k, zinv);
    D[k − 1] = c_sub(k_over_z, c_inv(c_add(D[k], k_over_z)));
  }
}
```

This code is used in section 115.

134. Small Spheres.

This calculates everything accurately for small spheres. This approximation is necessary because in the small particle or Rayleigh limit $x \to 0$ the Mie formulas become ill-conditioned. The method was taken from Wiscombe's paper and has been tested for several complex indices of refraction. Wiscombe uses this when

$$x|m| \le 0.1$$

and says this routine should be accurate to six places.

If $nangles \equiv 0$ or $s1 \equiv \Lambda$ or $s2 \equiv \Lambda$ then this routine will do the right thing—it will calculate the efficiencies and the anisotropy, but will not calculate any of the scattering amplitudes.

Since it is not obvious $z0 = i(m^2 - 1)$

135. ⟨ Prototype for *small_Mie* 135 ⟩ ≡

```
void small_Mie(double x, struct c_complex m, double *mu, long nangles, struct c_complex
    *s1, struct c_complex *s2, double *qext, double *qsca, double *qback, double *g)
```

This code is used in sections 116 and 136.

136. ⟨ Definition for *small_Mie* 136 ⟩ ≡

```
⟨ Prototype for small_Mie 135 ⟩
{
  struct c_complex ahat1, ahat2, bhat1;
  struct c_complex z0, m2, m4;
  double x2, x3, x4;
  if ((s1 ≡ Λ) ∨ (s2 ≡ Λ)) nangles = 0;
  m2 = c_sqr(m);
  m4 = c_sqr(m2);
  x2 = x * x;
  x3 = x2 * x;
  x4 = x2 * x2;
  z0.re = −m2.im;
  z0.im = m2.re − 1;
  ⟨ Calculate â1 137 ⟩
  ⟨ Calculate b̂1 138 ⟩
  ⟨ Calculate â2 139 ⟩
  ⟨ Calculate small Mie efficiencies and asymmetry 141 ⟩
  ⟨ Calculate small Mie scattering amplitudes 142 ⟩
}
```

This code is used in section 115.

137. The formula for $\hat{a}_1$ is

$$\hat{a}_1 = 2i\frac{m^2-1}{3}\frac{1-0.1x^2+\frac{4m^2+5}{1400}x^4}{D}$$

where

$$D = m^2+2+(1-0.7m^2)x^2-\frac{8m^4-385m^2+350}{1400}x^4+2i\frac{m^2-1}{3}x^3(1-0.1x^2)$$

Note that I have disabled the case when the sphere has no index of refraction. The perfectly conducting sphere equations are

⟨ Calculate $\hat{a}_1$ 137 ⟩ ≡

```
{
  struct c_complex z1, z2, z3, z4, D;
  z1 = c_smul(2.0/3.0, z0);
  z2.re = 1.0 - 0.1 * x2 + (4.0 * m2.re + 5.0) * x4/1400.0;
  z2.im = 4.0 * x4 * m2.im/1400.0;
  z3 = c_mul(z1, z2);
  z4 = c_smul(x3 * (1.0 - 0.1 * x2), z1);
  D.re = 2.0 + m2.re + (1 - 0.7 * m2.re) * x2 - (8.0 * m4.re - 385.0 * m2.re + 350.0)/1400.0 * x4 + z4.re;
  D.im = m2.im + (-0.7 * m2.im) * x2 - (8.0 * m4.im - 385.0 * m2.im)/1400.0 * x4 + z4.im;
  ahat1 = c_div(z3, D);
}
```

This code is used in section 136.

138. The formula for $\hat{b}_1$ is

$$\hat{b}_1 = ix^2\frac{m^2-1}{45}\frac{1+\frac{2m^2-5}{70}x^2}{1-\frac{2m^2-5}{30}x^2}$$

⟨ Calculate $\hat{b}_1$ 138 ⟩ ≡

```
{
  struct c_complex z2, z6, z7;
  z2 = c_smul(x2/45.0, z0);
  z6.re = 1.0 + (2.0 * m2.re - 5.0) * x2/70.0;
  z6.im = m2.im * x2/35.0;
  z7.re = 1.0 - (2.0 * m2.re - 5.0) * x2/30.0;
  z7.im = -m2.im * x2/15.0;
  bhat1 = c_mul(z2, c_div(z6, z7));
}
```

This code is used in section 136.

139. The formula for $\hat{a}_2$ is

$$\hat{a}_2 = ix^2 \frac{m^2-1}{15} \frac{1-\frac{1}{14}x^2}{2m^2+3-\frac{2m^2-7}{14}x^2}$$

⟨ Calculate $\hat{a}_2$ 139 ⟩ ≡

```
{
  struct c_complex z3, z8;
  z3 = c_smul((1.0 - x2/14.0) * x2/15.0, z0);
  z8.re = 2.0 * m2.re + 3.0 - (m2.re/7.0 - 0.5) * x2;
  z8.im = 2.0 * m2.im - m2.im/7.0 * x2;
  ahat2 = c_div(z3, z8);
}
```

This code is used in section 136.

140. The scattering and extinction efficiencies are given by

$$Q_{\rm ext} = 6x \,{\rm Re}\left[\hat{a}_1 + \hat{b}_1 + \frac{5}{3}\hat{a}_2\right]$$

and

$$Q_{\rm sca} = 6x^4 T$$

with

$$T = |\hat{a}_1|^2 + |\hat{b}_1|^2 + \frac{5}{3}|\hat{a}_2|^2$$

and the anisotropy (average cosine of the phase function) is

$$g = \frac{1}{T}\,{\rm Re}\left[\hat{a}_1(\hat{a}_2 + \hat{b}_1)^*\right]$$

I also calculate the backscattering efficiency so that it will be calculated correctly even when *nangles* ≡ 0. The backscattering efficiency $Q_{\rm back}$ is defined as

$$Q_{\rm back} = \frac{\sigma_{\rm back}}{\pi a^2} = \frac{|S_1(-1)|^2}{x^2}$$

where $\sigma_{\rm back}$ is the backscattering cross section. The expression for $S_1(\mu)$ given in the chunk below yields

$$\frac{S_1(-1)}{x} = \frac{3}{2}x^2\left[\hat{a}_1 - \hat{b}_1 - \frac{5}{3}\hat{a}_2\right]$$

This only remains to be squared before the efficiency for backscattering is obtained.

141. ⟨ Calculate small Mie efficiencies and asymmetry 141 ⟩ ≡

```
{
  struct c_complex ss1;
  double T;
  T = c_norm(ahat1) + c_norm(bhat1) + (5.0/3.0) * c_norm(ahat2);
  *qsca = 6.0 * x4 * T;
  *qext = 6.0 * x * (ahat1.re + bhat1.re + (5.0/3.0) * ahat2.re);
  *g = (ahat1.re * (ahat2.re + bhat1.re) + ahat1.im * (ahat2.im + bhat1.im))/T;
  ss1.re = 1.5 * x2 * (ahat1.re - bhat1.re - (5.0/3.0) * ahat2.re);
  ss1.im = 1.5 * x2 * (ahat1.im - bhat1.im - (5.0/3.0) * ahat2.im);
  *qback = 4 * c_norm(ss1);
}
```

This code is used in section 136.

142. Here is where the scattering functions get calculated according to

$$S_1(\mu) = \frac{3}{2}x^3\left[\hat{a}_1 + \left(\hat{b}_1 + \frac{5}{3}\hat{a}_2\right)\mu\right] \qquad S_2(\mu) = \frac{3}{2}x^3\left[\hat{b}_1 + \hat{a}_1\mu + \frac{5}{3}\hat{a}_2(2\mu^2 - 1)\right]$$

Since this is the last thing to get calculated, I take the liberty of mucking around with the variables $\hat{a}_1$, $\hat{b}_1$, $\hat{a}_2$, and x^3

⟨ Calculate small Mie scattering amplitudes 142 ⟩ ≡

```
{
  double muj, angle;
  long j;
  x3 *= 1.5;
  ahat1.re *= x3;
  ahat1.im *= x3;
  bhat1.re *= x3;
  bhat1.im *= x3;
  ahat2.re *= x3 * (5.0/3.0);
  ahat2.im *= x3 * (5.0/3.0);
  for (j = 0; j < nangles; j++) {
    muj = mu[j];
    angle = 2.0 * muj * muj - 1.0;
    s1[j].re = ahat1.re + (bhat1.re + ahat2.re) * muj;
    s1[j].im = ahat1.im + (bhat1.im + ahat2.im) * muj;
    s2[j].re = bhat1.re + ahat1.re * muj + ahat2.re * angle;
    s2[j].im = bhat1.im + ahat1.im * muj + ahat2.im * angle;
  }
}
```

This code is used in section 136.

143. Small Perfectly Conducting Spheres.

144. ⟨ Prototype for *small_conducting_Mie* 144 ⟩ ≡

```
void small_conducting_Mie(double x, struct c_complex m, double *mu, long nangles, struct
    c_complex *s1, struct c_complex *s2, double *qext, double *qsca, double *qback, double *g)
```

This code is used in sections 116 and 145.

145. ⟨ Definition for *small_conducting_Mie* 145 ⟩ ≡

```
⟨ Prototype for small_conducting_Mie 144 ⟩
{
  struct c_complex ahat1, ahat2, bhat1, bhat2;
  struct c_complex ss1;
  double x2, x3, x4, muj, angle;
  long j;
  if ((s1 ≡ Λ) ∨ (s2 ≡ Λ)) nangles = 0;
  m.re += 0.0;      /* suppress warning */
  x2 = x * x;
  x3 = x2 * x;
  x4 = x2 * x2;
  ahat1 = c_div(c_set(0.0, 2.0/3.0 * (1.0 − 0.2 * x2)), c_set(1.0 − 0.5 * x2, 2.0/3.0 * x3));
  bhat1 = c_div(c_set(0.0, (x2 − 10.0)/30.0), c_set(1 + 0.5 * x2, −x3/3.0));
  ahat2 = c_set(0.0, x2/30.);
  bhat2 = c_set(0.0, −x2/45.);
  *qsca = 6.0 * x4 * (c_norm(ahat1) + c_norm(bhat1) + (5.0/3.0) * (c_norm(ahat2) + c_norm(bhat2)));
  *qext = *qsca;
  *g = 6.0 * x4 * (ahat1.im * (ahat2.im + bhat1.im) + bhat2.im * (5.0/9.0 * ahat2.im + bhat1.im) +
      ahat1.re * bhat1.re)/(*qsca);
  ss1.re = 1.5 * x2 * (ahat1.re − bhat1.re);
  ss1.im = 1.5 * x2 * (ahat1.im − bhat1.im − (5.0/3.0) * (ahat2.im + bhat2.im));
  *qback = 4 * c_norm(ss1);
  x3 *= 1.5;
  ahat1.re *= x3;
  ahat1.im *= x3;
  bhat1.re *= x3;
  bhat1.im *= x3;
  ahat2.im *= x3 * (5.0/3.0);
  bhat2.im *= x3 * (5.0/3.0);
  for (j = 0; j < nangles; j++) {
    muj = mu[j];
    angle = 2.0 * muj * muj − 1.0;
    s1[j].re = ahat1.re + (bhat1.re) * muj;
    s1[j].im = ahat1.im + (bhat1.im + ahat2.im) * muj + bhat2.im * angle;
    ;
    s2[j].re = bhat1.re + (ahat1.re) * muj;
    s2[j].im = bhat1.im + (ahat1.im + bhat2.im) * muj + ahat2.im * angle;
  }
}
```

This code is used in section 115.

146. Arbitrary Spheres.

Calculates the amplitude scattering matrix elements and efficiencies for extinction, total scattering and backscattering for a given size parameter and relative refractive index. The basic algorithm follows Bohren and Huffman originally written in Fortran. The code was translated into CWeb and documented by Scott Prahl.

Many improvements suggested by Wiscombe have been incorporated. In particular, either upward or downward iteration will be used to calculate the lograthmic derivative $D_n(z)$.

Routine preliminary checking suggests that everything is being calculated ok except g.

Space must have been allocated for the scattering amplitude angles *s1* and *s2* before this routine is called.

147. ⟨ Prototype for *Mie* 147 ⟩ ≡

```
void Mie(double x, struct c_complex m, double *mu, long nangles, struct c_complex *s1, struct
    c_complex *s2, double *qext, double *qsca, double *qback, double *g)
```

This code is used in sections 116 and 148.

148. ⟨ Definition for *Mie* 148 ⟩ ≡

```
⟨ Prototype for Mie 147 ⟩
{
  ⟨ Declare variables for Mie 149 ⟩
  ⟨ Catch bogus input values 150 ⟩
  ⟨ Deal with small spheres 151 ⟩
  ⟨ Calculate nstop 153 ⟩
  ⟨ Mie allocate and initialize angle arrays 152 ⟩
  if (m.re > 0) ⟨ Calculate the logarithmic derivatives 154 ⟩
  ⟨ Prepare to sum over all nstop terms 155 ⟩
  for (n = 1; n ≤ nstop; n++) {
    ⟨ Establish a_n and b_n 156 ⟩
    ⟨ Calculate phase function for each angle 157 ⟩
    ⟨ Increment cross sections 158 ⟩
    ⟨ Prepare for the next iteration 159 ⟩
  }
  ⟨ Calculate Efficiencies 160 ⟩
  ⟨ Free allocated memory 161 ⟩
}
```

This code is used in section 115.

149.

⟨ Declare variables for *Mie* 149 ⟩ ≡

```
struct c_complex *D;
struct c_complex z1, an, bn, bnm1, anm1, qbcalc;
double *pi0, *pi1, *tau;
struct c_complex xi, xi0, xi1;
double psi, psi0, psi1;
double alpha, beta, factor;
long n, k, nstop, sign;
*qext = -1;
*qsca = -1;
*qback = -1;
*g = -1;
```

This code is used in section 148.

150.

⟨ Catch bogus input values 150 ⟩ ≡

```
if (m.im > 0.0) {
  mie_error("This␣program␣requires␣m.im>=0", 1);
  return;
}
if (x ≤ 0.0) {
  mie_error("This␣program␣requires␣positive␣sphere␣sizes", 2);
  return;
}
if (nangles < 0) {
  mie_error("This␣program␣requires␣non-negative␣angle␣sizes", 3);
  return;
}
if (nangles < 0) {
  mie_error("This␣program␣requires␣non-negative␣angle␣sizes", 4);
  return;
}
if ((nangles > 0) ∧ (s1 ≡ Λ)) {
  mie_error("Space␣must␣be␣allocated␣for␣s1␣if␣nangles!=0", 5);
  return;
}
if ((nangles > 0) ∧ (s2 ≡ Λ)) {
  mie_error("Space␣must␣be␣allocated␣for␣s2if␣nangles!=0", 6);
  return;
}
if (x > 20000) {
  mie_error("Program␣not␣validated␣for␣spheres␣with␣x>20000", 7);
  return;
}
```

This code is used in section 148.

151.

⟨ Deal with small spheres 151 ⟩ ≡

```
if ((m.re ≡ 0) ∧ (x < 0.1)) {
  small_conducting_Mie(x, m, mu, nangles, s1, s2, qext, qsca, qback, g);
  return;
}
if ((m.re > 0.0) ∧ (c_abs(m) ∗ x < 0.1)) {
  small_Mie(x, m, mu, nangles, s1, s2, qext, qsca, qback, g);
  return;
}
```

This code is used in section 148.

152. ⟨ Mie allocate and initialize angle arrays 152 ⟩ ≡

```
if (nangles > 0) {
  set_carray(s1, nangles, c_set(0.0, 0.0));
  set_carray(s2, nangles, c_set(0.0, 0.0));
  pi0 = new_darray(nangles);
  pi1 = new_darray(nangles);
  tau = new_darray(nangles);
  set_darray(pi0, nangles, 0.0);
  set_darray(tau, nangles, 0.0);
  set_darray(pi1, nangles, 1.0);
}
```

This code is used in section 148.

153. Calculate number of terms to be summed in series after Wiscombe

⟨ Calculate *nstop* 153 ⟩ ≡

```
nstop = floor(x + 4.05 * pow(x, 0.33333) + 2.0);
```

This code is used in section 148.

154. Allocate and initialize the space for the arrays. One noteworthy aspect is that the complex array D is allocated from 0 to *nstop*. This allows D to be a one-based array from 1 to *nstop* instead of a zero-based array from 0 to $nstop - 1$. Therefore $D[n]$ will directly correspond to D_n in Bohren. Furthermore, *an* and *bn* will correspond to a_n and b_n. The angular arrays are still zero-based.

Use formula 7 from Wiscombe's paper to figure out if upwards or downwards recurrence should be used. Namely if

$$m_{\mathrm{Im}} x \leq 13.78 m_{\mathrm{Re}}^2 - 10.8 m_{\mathrm{Re}} + 3.9$$

the upward recurrence would be stable.

⟨ Calculate the logarithmic derivatives 154 ⟩ ≡

```
{
  struct c_complex z;
  z = c_smul(x, m);
  D = new_carray(nstop + 1);
  if (D ≡ Λ) {
    mie_error("Cannot␣allocate␣log␣array", 8);
    return;
  }
  if (m.re < 1 ∨ m.re > 10 ∨ fabs(m.im) > 10 ∨ fabs(x * m.im) ≥ 3.9 − 10.8 * m.re + 13.78 * m.re * m.re)
    Dn_down(z, nstop, D);
  else Dn_up(z, nstop, D);
}
```

This code is used in section 148.

155. OK, Here we go. We need to start up the arrays. First, recall (page 128 Bohren and Huffman) that

$$\psi_n(x) = x j_n(x) \qquad \text{and} \qquad \xi_n(x) = x j_n(x) + i x y_n(x)$$

where j_n and y_n are spherical Bessel functions. The first few terms may be worked out as,

$$\psi_0(x) = \sin x \qquad \text{and} \qquad \psi_1(x) = \frac{\sin x}{x} - \cos x$$

and

$$\xi_0(x) = \psi_0 + i \cos x \qquad \text{and} \qquad \xi_1(x) = \psi_1 + i \left[\frac{\cos x}{x} + \sin x \right]$$

⟨ Prepare to sum over all *nstop* terms 155 ⟩ ≡

```
psi0 = sin(x);
psi1 = psi0/x − cos(x);
xi0 = c_set(psi0, cos(x));
xi1 = c_set(psi1, cos(x)/x + sin(x));
*qsca = 0.0;
*g = 0.0;
*qext = 0.0;
sign = 1;
qbcalc = c_set(0.0, 0.0);
anm1 = c_set(0.0, 0.0);
bnm1 = c_set(0.0, 0.0);
```

This code is used in section 148.

156. The main equations for a_n and b_n in Bohren and Huffman Equation (4.88).

$$a_n = \frac{\Big[D_n(mx)/m + n/x\Big]\psi_n(x) - \psi_{n-1}(x)}{\Big[D_n(mx)/m + n/x\Big]\xi_n(x) - \xi_{n-1}(x)}$$

and

$$b_n = \frac{\Big[mD_n(mx) + n/x\Big]\psi_n(x) - \psi_{n-1}(x)}{\Big[mD_n(mx) + n/x\Big]\xi_n(x) - \xi_{n-1}(x)}$$

```
⟨ Establish a_n and b_n 156 ⟩ ≡
  if (m.re ≡ 0.0) {
    an = c_sdiv(n * psi1/x - psi0, c_sub(c_smul(n/x, xi1), xi0));
    bn = c_sdiv(psi1, xi1);
  }
  else if (m.im ≡ 0.0) {
    z1.re = D[n].re/m.re + n/x;
    an = c_sdiv(z1.re * psi1 - psi0, c_sub(c_smul(z1.re, xi1), xi0));
    z1.re = D[n].re * m.re + n/x;
    bn = c_sdiv(z1.re * psi1 - psi0, c_sub(c_smul(z1.re, xi1), xi0));
  }
  else {
    z1 = c_div(D[n], m);
    z1.re += n/x;
    an = c_div(c_set(z1.re * psi1 - psi0, z1.im * psi1), c_sub(c_mul(z1, xi1), xi0));
    z1 = c_mul(D[n], m);
    z1.re += n/x;
    bn = c_div(c_set(z1.re * psi1 - psi0, z1.im * psi1), c_sub(c_mul(z1, xi1), xi0));
  }
```

This code is used in section 148.

157. The scattering matrix is given by Equation 4.74 in Bohren and Huffman. Namely,

$$S_1 = \sum_n \frac{2n+1}{n(n+1)}(a_n\pi_n + b_n\tau_n)$$

and

$$S_2 = \sum_n \frac{2n+1}{n(n+1)}(a_n\tau_n + b_n\pi_n)$$

Furthermore, equation 4.47 in Bohren and Huffman states

$$\pi_n = \frac{2n-1}{n-1}\mu\pi_{n-1} - \frac{n}{n-1}\pi_{n-2}$$

and

$$\tau_n = n\mu\pi_n - (n+1)\pi_{n-1}$$

⟨ Calculate phase function for each angle 157 ⟩ ≡

```
for (k = 0; k < nangles; k++) {
  factor = (2.0 * n + 1.0)/(n + 1.0)/n;
  tau[k] = n * mu[k] * pi1[k] - (n + 1) * pi0[k];
  alpha = factor * pi1[k];
  beta = factor * tau[k];
  s1[k].re += alpha * an.re + beta * bn.re;
  s1[k].im += alpha * an.im + beta * bn.im;
  s2[k].re += alpha * bn.re + beta * an.re;
  s2[k].im += alpha * bn.im + beta * an.im;
}
for (k = 0; k < nangles; k++) {
  factor = pi1[k];
  pi1[k] = ((2.0 * n + 1.0) * mu[k] * pi1[k] - (n + 1.0) * pi0[k])/n;
  pi0[k] = factor;
}
```

This code is used in section 148.

158. From page 120 of Bohren and Huffman the anisotropy is given by

$$Q_{\rm sca}\langle\cos\theta\rangle = Q_{\rm sca}\cdot g = \frac{4}{x^2}\left[\sum_{n=1}^{\infty}\frac{n(n+2)}{n+1}\,{\rm Re}\{a_n a_{n+1}^* + b_n b_{n+1}^*\} + \sum_{n=1}^{\infty}\frac{2n+1}{n(n+1)}\,{\rm Re}\{a_n b_n^*\}\right]$$

For computation purposes, this must be rewritten as

$$Q_{\rm sca}\cdot g = \frac{4}{x^2}\left[\sum_{n=2}^{\infty}\frac{(n^2-1)}{n}\,{\rm Re}\{a_{n-1} a_n^* + b_{n-1} b_n^*\} + \sum_{n=1}^{\infty}\frac{2n+1}{n(n+1)}\,{\rm Re}\{a_n b_n^*\}\right]$$

From page 122 we find an expression for the backscattering efficiency

$$Q_{\rm back} = \frac{\sigma_b}{\pi a^2} = \frac{1}{x^2}\left|\sum_{n=1}^{\infty}(2n+1)(-1)^n(a_n - b_n)\right|^2$$

From page 103 we find an expression for the scattering cross section

$$Q_{\rm sca} = \frac{\sigma_s}{\pi a^2} = \frac{2}{x^2}\sum_{n=1}^{\infty}(2n+1)(|a_n|^2 + |b_n|^2)$$

The total extinction efficiency is also found on page 103

$$Q_{\rm ext} = \frac{\sigma_t}{\pi a^2} = \frac{2}{x^2}\sum_{n=1}^{\infty}(2n+1)\,{\rm Re}(a_n + b_n)$$

⟨ Increment cross sections 158 ⟩ ≡

```
factor = 2.0 * n + 1.0;
*g += (n * n - 1.0)/n * (anm1.re * an.re + anm1.im * an.im + bnm1.re * bn.re + bnm1.im * bn.im);
*g += factor/n/(n + 1.0) * (an.re * bn.re + an.im * bn.im);
*qsca += factor * (c_norm(an) + c_norm(bn));
*qext += factor * (an.re + bn.re);
sign *= -1;
qbcalc.re += sign * factor * (an.re - bn.re);
qbcalc.im += sign * factor * (an.im - bn.im);
```

This code is used in section 148.

159. The recurrence relations for ψ and ξ depend on the recursion relations for the spherical Bessel functions (page 96 equation 4.11)

$$z_{n-1}(x) + z_{n+1}(x) = \frac{2n+1}{x} z_n(x)$$

where z_n might be either j_n or y_n. Thus

$$\psi_{n+1}(x) = \frac{2n+1}{x}\psi_n(x) - \psi_{n-1}(x) \qquad \text{and} \qquad \xi_{n+1}(x) = \frac{2n+1}{x}\xi_n(x) - \xi_{n-1}(x)$$

Furthermore,

⟨ Prepare for the next iteration 159 ⟩ ≡
```
factor = (2.0 * n + 1.0)/x;
xi = c_sub(c_smul(factor, xi1), xi0);
xi0 = xi1;
xi1 = xi;
psi = factor * psi1 - psi0;
psi0 = psi1;
psi1 = xi1.re;
anm1 = an;
bnm1 = bn;
```
This code is used in section 148.

160. ⟨ Calculate Efficiencies 160 ⟩ ≡
```
*qsca *= 2/(x * x);
*qext *= 2/(x * x);
*g *= 4/(*qsca)/(x * x);
*qback = c_norm(qbcalc)/(x * x);
```
This code is used in section 148.

161. ⟨ Free allocated memory 161 ⟩ ≡
```
if (m.re > 0) free_carray(D);
if (nangles > 0) {
  free_darray(pi0);
  free_darray(pi1);
  free_darray(tau);
}
```
This code is used in section 148.

162. Easy Mie.

Given the size and real index of refraction, calculate the scattering efficiency and the anisotropy for a non-absorbing sphere. If the sphere is totally reflecting, then let the index of refraction be equal to zero.

To recover the scattering coefficient μ_s from the efficiency *qsca* just multiply *qsca* by the geometric cross sectional area and the density of scatterers.

163. The function *ez_Mie*.

```
⟨ Prototype for ez_Mie 163 ⟩ ≡
  void ez_Mie(double x, double n, double *qsca, double *g)
```

This code is used in sections 116, 117, and 164.

164.

```
⟨ Definition for ez_Mie 164 ⟩ ≡
  ⟨ Prototype for ez_Mie 163 ⟩
  {
    long nangles = 0;
    double *mu = Λ;
    struct c_complex *s1 = Λ;
    struct c_complex *s2 = Λ;
    struct c_complex m;
    double qext, qback;

    m.re = n;
    m.im = 0.0;
    Mie(x, m, mu, nangles, s1, s2, &qext, qsca, &qback, g);
  }
```

This code is used in section 115.

165. The function *ez_Mie_Full*. This is a simple interface to that provides complete access to Mie calculations. This function will return the scattering functions $S_1(\mu)$ and $S_2(\mu)$ for each of the angles in the array *mu*. Note that these are the cosines of the angles and not the angles in radians.

This routine assumes that memory has been allocated for the arrays *mu*, *s1_real*, *s1_imag*, *s2_real* and, *s2_imag*. The number of elements in these arrays is specified by *nangles*.

If you do not want to mess with angles then you probably want to call *ez_Mie* instead of this function.

```
⟨ Prototype for ez_Mie_Full 165 ⟩ ≡
  void ez_Mie_Full(double x, double m_real, double m_imag, long nangles, double *mu, double
      *s1_real, double *s1_imag, double *s2_real, double *s2_imag, double *qext, double *qsca, double
      *qback, double *g)
```

This code is used in sections 116, 117, and 166.

166. ⟨ Definition for *ez_Mie_Full* 166 ⟩ ≡

```
⟨ Prototype for ez_Mie_Full 165 ⟩
{
  struct c_complex *s1 = Λ;
  struct c_complex *s2 = Λ;
  struct c_complex m;
  int i;
  m.re = m_real;
  m.im = m_imag;
  s1 = new_carray(nangles);
  s2 = new_carray(nangles);
  Mie(x, m, mu, nangles, s1, s2, qext, qsca, qback, g);
  for (i = 0; i < nangles; i++) {
    s1_imag[i] = s1[i].im;
    s1_real[i] = s1[i].re;
    s2_imag[i] = s2[i].im;
    s2_real[i] = s2[i].re;
  }
  free_carray(s1);
  free_carray(s2);
}
```

This code is used in section 115.

167. A driver program for spherical Mie scattering.

This program assumes a sphere in a medium with index of refraction 1.0. If this is not the case then the index of refraction of the sphere should be divided by the index of refraction of the medium. The wavelength should also be divided by the index of refraction of the medium as well.

This program is intended to provide a convenient means for calculating Mie scattering parameters. It reads from *stdin* and writes to *stdout*. Each line of *stdin* should contain one set of Mie parameters arranged as follows

`radius wavelength index.real index.imag density num.angles`

where

`radius` is the radius of the sphere [μm]
`wavelength` is the wavelength in the medium [μm]
`index.real` is the real refractive index
`index.imag` is the imaginary refraction index
`density` is the sphere density per cubic micron [μm^{-3}]
`num.angles` is the number of angles to generate

168. The real program is here

⟨ `mie_main.c` 168 ⟩ ≡
```
⟨ the include files 169 ⟩
⟨ print version function 177 ⟩
⟨ print usage function 178 ⟩
int main(int argc, char **argv)
{
  ⟨ Declare Mie variables 170 ⟩
  ⟨ Handle options 171 ⟩
  ⟨ Allocate angle based arrays 172 ⟩
  ⟨ Print header 174 ⟩
  mm.re = m.re/n_medium;
  mm.im = m.im/n_medium;
  lambda = lambda_vac/n_medium;
  x = 2 * 3.1415926 * radius * n_medium/lambda_vac;
  Mie(x, mm, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
  ⟨ Print summary 175 ⟩
  ⟨ Print phase function 176 ⟩
  ⟨ Free angle based arrays 173 ⟩
  return 0;
}
```

169. ⟨ the include files 169 ⟩ ≡
```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include "mie_array.h"
#include "mie_complex.h"
#include "mie.h"
#include "mygetopt.h"
#include "version.h"
  extern char *optarg;
  extern int optind;
```
This code is used in section 168.

170. Variables.

⟨ Declare Mie variables 170 ⟩ ≡

```
char *g_out_name = Λ;
double pi = 3.14159265358979;
double area;
long i;
struct c_complex m, mm;
struct c_complex *s1 = Λ;
struct c_complex *s2 = Λ;
double *parallel = Λ;
double *perpen = Λ;
double *phasefn = Λ;
double *mu = Λ;
double x, qext, qsca, qback, g;
int machine_readable_output = 0;
int quiet = 0;
double radius = 0.525;
double lambda_vac = 0.6328;
double lambda_vac_last = 0.6328;
double lambda = 0.6328;
long nangles = 0;
long nlambda = 0;
double density = 1;
double n_medium = 1.0;
m.re = 1.55;
m.im = 0.00;
```

This code is used in section 168.

171. use the *mygetop* to process options.

```
⟨ Handle options 171 ⟩ ≡
  {
    char c;
    double xopt;
    while ((c = my_getopt(argc, argv, "h?qvm:l:L:n:r:i:o:d:p:P:")) ≠ EOF) {
      switch (c) {
      case 'r': sscanf(optarg, "%lf", &xopt);
        if (xopt > 0) radius = xopt;
        break;
      case 'm': sscanf(optarg, "%lf", &xopt);
        if (xopt > 0) n_medium = xopt;
        break;
      case 'n': sscanf(optarg, "%lf", &xopt);
        if (xopt > 0) m.re = xopt;
        break;
      case 'l': sscanf(optarg, "%lf", &xopt);
        if (xopt > 0) lambda_vac = xopt;
        break;
      case 'L': sscanf(optarg, "%lf", &xopt);
        if (xopt > 0) lambda_vac_last = xopt;
        break;
      case 'i': sscanf(optarg, "%lf", &xopt);
        if (xopt ≤ 0) m.im = xopt;
        break;
      case 'p': sscanf(optarg, "%lf", &xopt);
        if (xopt ≥ 0) nangles = (long) xopt;
        nlambda = 0;
        break;
      case 'P': sscanf(optarg, "%lf", &xopt);
        if (xopt ≥ 0) nlambda = (long) xopt;
        nangles = 0;
        break;
      case 'd': sscanf(optarg, "%lf", &xopt);
        if (xopt ≥ 0) density = xopt;
        break;
      case 'o': g_out_name = strdup(optarg);
        break;
      case 'q': machine_readable_output = 1;
        quiet = 1;
        break;
      case 'v': print_version( );
        break;
      default: case 'h': case '?': print_usage( );
        break;
      }
    }
    argc -= optind;
    argv += optind;
    if (argc > 0) {
      fprintf(stderr, "No␣file␣support␣in␣this␣version.␣␣Sorry.\n");
      exit(1);
```

```
    }
    if (g_out_name ≠ Λ) {
      if (freopen(g_out_name, "w", stdout) ≡ Λ) {
        fprintf(stderr, "Could␣not␣open␣file␣<%s>␣for␣output", g_out_name);
        exit(1);
      }
    }
  }
```

This code is used in section 168.

172.

⟨ Allocate angle based arrays 172 ⟩ ≡

```
  if (nangles > 0) {
    mu = new_darray(nangles);
    for (i = 0; i < nangles; i++) mu[i] = cos(2 * pi/nangles * i);
    parallel = new_darray(nangles);
    perpen = new_darray(nangles);
    phasefn = new_darray(nangles);
    s1 = new_carray(nangles);
    s2 = new_carray(nangles);
  }
```

This code is used in section 168.

173.

⟨ Free angle based arrays 173 ⟩ ≡

```
  if (nangles > 0) {
    free_darray(mu);
    free_darray(parallel);
    free_darray(perpen);
    free_darray(phasefn);
    free_carray(s1);
    free_carray(s2);
  }
```

This code is used in section 168.

174. Print a header then the angles. Make sure everything lines up.

⟨ Print header 174 ⟩ ≡

```
  printf("#␣Mie␣Scattering␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣Version␣%s\n", Version);
  printf("#␣Oregon␣Medical␣Laser␣Center␣␣␣#␣https://omlc.org/software/mie\n");
  printf("#␣by␣Scott␣Prahl␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣scott.prahl@oit.edu\n");
  printf("#\n");
```

This code is used in section 168.

175. ⟨ Print summary 175 ⟩ ≡

```
{
  double mut, mus, musp;
  area = 3.14159265358979 * radius * radius;
  mut = density * qext * area * 1000;
  mus = density * qsca * area * 1000;
  musp = mus * (1.0 - g);
  printf("# radius      \t%9.5f\t [microns]      (sphere radius)\n", radius);
  printf("# n_medium    \t%9.5f\t [---]          (refractive index of medium)\n", n_medium);
  printf("# n_real      \t%9.5f\t [---]          (refractive index of sphere)\n", m.re);
  printf("# n_imag      \t%9.5f\t [---]          (absorption of sphere)\n", m.im);
  printf("# lambda_vac  \t%9.5f\t [microns]      (wavelength in vacuum)\n", lambda_vac);
  printf("# density     \t%9.5f\t [#/micron^3]   (spheres per cubic micron)\n", density);
  printf("#\n");
  printf("# lambda      \t%9.5f\t [microns]      (wavelength in medium)\n", lambda);
  printf("# X           \t%9.5f\t [---]          (size parameter)\n", x);
  printf("# g           \t%9.5g\t [---]          (average cosine of phase function)\n", g);
  printf("# Qsca        \t%9.5g\t [---]          (scattering efficiency)\n", qsca);
  printf("# Qext        \t%9.5g\t [---]          (extinction efficiency)\n", qext);
  printf("# Qback       \t%9.5g\t [---]          (backscattering efficiency)\n", qback);
  printf("# Csca        \t%9.5g\t [micron^2]     (scattering cross section)\n", qsca * area);
  printf("# Cext        \t%9.5g\t [micron^2]     (extinction cross section)\n", qext * area);
  printf("# Cback       \t%9.5g\t [micron^2]     (backscattering cross section)\n",
      qback * area);
  printf("# mu_s        \t%9.5g\t [1/mm]         (scattering coefficient)\n", mus);
  printf("# mu_s'       \t%9.5g\t [1/mm]         (reduced scattering coefficient)\n", musp);
  printf("# mu_t        \t%9.5g\t [1/mm]         (total attenuation coefficient)\n", mut);
}
```

This code is used in section 168.

176. ⟨ Print phase function 176 ⟩ ≡

```
if (nangles > 0) {
  int j;
  double max_natural, max_perpen, max_parallel;
  for (i = 0; i < nangles; ++i) {
    parallel[i] = c_norm(s2[i])/(x * x * qsca)/3.14159;
    perpen[i] = c_norm(s1[i])/(x * x * qsca)/3.14159;
    phasefn[i] = (parallel[i] + perpen[i])/2.0;
  }
  max_natural = phasefn[0];
  max_perpen = perpen[0];
  max_parallel = parallel[0];
  for (i = 0; i < nangles; ++i) {
    if (phasefn[i] > max_natural) max_natural = phasefn[i];
    if (parallel[i] > max_parallel) max_parallel = parallel[i];
    if (perpen[i] > max_perpen) max_perpen = perpen[i];
  }
  printf("#\n");
  printf("#␣␣␣The␣second␣column␣is␣normalized␣so␣that␣the␣integral␣of␣it␣over␣\n");
  printf("#␣␣␣4*pi␣steradians␣will␣be␣unity.␣␣The␣average␣of␣the␣3rd␣&␣4th\n");
  printf("#␣␣␣columns␣is␣the␣second.␣␣The␣next␣three␣columns␣are␣normalized\n");
  printf("#␣␣␣to␣the␣value␣at␣0␣degrees.␣\n");
  printf("#\n");
  printf("#␣␣␣␣␣␣␣natural␣␣␣␣␣␣=␣(|S1|^2+|S2|^2)/2*1/(pi␣X^2␣Qsca)\n");
  printf("#␣␣␣␣␣␣␣perpen␣␣␣␣␣␣␣=␣|S1|^2/(pi␣X^2␣Qsca)\n");
  printf("#␣␣␣␣␣␣␣parallel␣␣␣␣␣=␣|S2|^2/(pi␣X^2␣Qsca)\n");
  printf("#␣␣␣␣␣␣␣polarization␣=␣(|S1|^2-|S2|^2)/(|S1|^2+|S2|^2)\n");
  printf("#␣␣␣␣␣␣␣S33␣␣␣␣␣␣␣␣␣␣=␣␣Real(S2␣*␣S1^*)\n");
  printf("#␣␣␣␣␣␣␣S34␣␣␣␣␣␣␣␣␣␣=␣-Imag(S2␣*␣S1^*)\n");
  printf("#\n");
  printf("##theta\t␣natural␣\t␣perpen␣\t␣parallel");
  printf("\t␣natural␣\t␣perpen␣\t␣parallel\t␣polarization␣\t␣␣␣S33␣␣␣\t␣␣␣S34\n");
  for (j = 0; j < nangles; j++) {
    double angle, d, polar, s33, s34;
    struct c_complex t;
    i = j + nangles/2 + 1;
    if (i ≥ nangles) i -= nangles;
    if (i ≤ nangles/2) angle = 180.0/3.1415926 * acos(mu[i]);
    else angle = -180.0/3.1415926 * acos(mu[i]);
    t = c_mul(s2[i], c_conj(s1[i]));
    d = (c_norm(s1[i]) + c_norm(s2[i]))/2.0;
    polar = (c_norm(s1[i]) - c_norm(s2[i]))/2.0/d;
    s33 = (t.re)/d;
    s34 = -(t.im)/d;
    printf("%6.3f\t%8.5f\t%8.5f\t%8.5f", angle, phasefn[i], perpen[i], parallel[i]);
    printf("\t%␣6.5f\t%␣6.5f\t%␣6.5f", phasefn[i]/max_natural, perpen[i]/max_perpen,
        parallel[i]/max_parallel);
    printf("\t%␣8.5f\t%␣8.5f\t%␣8.5f\n", polar, s33, s34);
  }
}
```

This code is used in section 168.

177. ⟨ print version function 177 ⟩ ≡

```
static void print_version(void)
{
  fprintf(stderr, "mie␣%s\n\n", Version);
  fprintf(stderr, "Copyright␣2012-23␣Free␣Software␣Foundation,␣Inc.\n");
  fprintf(stderr,
      "This␣is␣free␣software;␣see␣the␣source␣for␣copying␣conditions.␣␣There␣is␣NO\n");
  fprintf(stderr,
      "warranty;␣not␣even␣for␣MERCHANTABILITY␣or␣FITNESS␣FOR␣A␣PARTICULAR␣PURPOSE.");
  fprintf(stderr, "\n\nWritten␣by␣Scott␣Prahl\n");
  exit(0);
}
```

This code is used in section 168.

178. ⟨ print usage function 178 ⟩ ≡

```
static void print_usage(void)
{
  fprintf(stderr, "mie␣%s\n\n", Version);
  fprintf(stderr, "Calculates␣spherical␣Mie␣scattering␣phase␣function\n\n");
  fprintf(stderr, "Usage:\n");
  fprintf(stderr, "␣␣␣␣mie␣[-l␣lambda]␣[-r␣radius]␣[-n␣index]␣");
  fprintf(stderr, "[-i␣imag␣index]␣[-d␣density]␣[-p␣phase␣angles]\n\n");
  fprintf(stderr, "Options:\n");
  fprintf(stderr, "␣␣␣␣-o␣filename␣␣␣␣␣␣␣␣␣␣#␣explicitly␣specify␣filename␣for␣output\n");
  fprintf(stderr, "␣␣␣␣-q␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣quiet␣---␣omit␣output␣to␣stderr\n\n");
  fprintf(stderr,
      "␣␣␣␣-d␣␣density␣␣␣␣␣␣␣␣␣␣#␣density␣(spheres/micron^3)␣␣[default=1.000]\n");
  fprintf(stderr,
      "␣␣␣␣-i␣␣imag_index␣␣␣␣␣␣␣␣#␣imag␣index␣of␣refraction␣␣␣␣[default=0.000]\n");
  fprintf(stderr,
      "␣␣␣␣-l␣␣lambda_vac␣␣␣␣␣␣␣␣#␣wavelength␣in␣vacuum␣␣␣␣␣␣␣␣[default=0.633]\n");
  fprintf(stderr,
      "␣␣␣␣-L␣␣last_lambda␣␣␣␣␣␣␣␣#␣last␣wavelength␣in␣vacuum␣␣␣[default=0.633]\n");
  fprintf(stderr,
      "␣␣␣␣-m␣␣index_of_medium␣␣␣␣#␣refractive␣index␣of␣medium␣␣[default=1.000]\n");
  fprintf(stderr,
      "␣␣␣␣-n␣␣real_index␣␣␣␣␣␣␣␣#␣real␣index␣of␣refraction␣␣␣␣[default=1.550]\n");
  fprintf(stderr,
      "␣␣␣␣-p␣␣num_of_angles␣␣␣␣␣␣#␣number␣of␣angles␣␣␣␣␣␣␣␣␣␣␣␣[default=0␣␣␣␣]\n");
  fprintf(stderr,
      "␣␣␣␣-P␣␣num_of_lambda␣␣␣␣␣␣#␣number␣of␣wavelengths␣␣␣␣␣␣␣␣[default=0␣␣␣␣]\n");
  fprintf(stderr,
      "␣␣␣␣-r␣␣radius␣␣␣␣␣␣␣␣␣␣␣␣#␣sphere␣radius␣[microns]␣␣␣␣␣␣[default=0.525]\n\n");
  fprintf(stderr, "␣␣␣␣-h␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣display␣help\n");
  fprintf(stderr, "␣␣␣␣-v␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣version␣information\n\n");
  fprintf(stderr, "Examples:\n");
  fprintf(stderr, "␣␣␣␣mie␣-p␣40␣␣␣␣␣␣␣␣␣#␣Bohren␣&␣Huffman␣Appendix␣A\n");
  fprintf(stderr, "␣␣␣␣mie␣-d␣1␣-i␣0␣-l␣0.6328␣-m␣1␣-n␣1.55␣-p␣40␣-r␣0.525\n\n");
  fprintf(stderr, "Report␣bugs␣to␣scott.prahl@oit.edu\n\n");
  exit(0);
}
```

This code is used in section 168.

179. Cylindrical Mie Algorithms.
Routines to calculate scattering form an infinitely long cylinder. Original Fortran version written by D. Mackowski. This version was translated by me into Pascal and then into C.

180. Here, then, is an overview of document structure

```
⟨ mie_cylinder.c 180 ⟩ ≡
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include "mie_array.h"
#include "mie_complex.h"
#include "mie_cylinder.h"
#define PI 3.14159265358979
  ⟨ Definition for bessj0 183 ⟩
  ⟨ Definition for bessj1 187 ⟩
  ⟨ Definition for bessy0 185 ⟩
  ⟨ Definition for bessy1 189 ⟩
  ⟨ Definition for jn_real 191 ⟩
  ⟨ Definition for jn_complex 193 ⟩
  ⟨ Definition for MieCylinderCoefficients 195 ⟩
  ⟨ Definition for MieCylinder 197 ⟩
```

181. And the header file

```
⟨ mie_cylinder.h 181 ⟩ ≡
  ⟨ Prototype for MieCylinderCoefficients 194 ⟩;
  ⟨ Prototype for MieCylinder 196 ⟩;
```

182. Bessel function $J_0(x)$ for real x

```
⟨ Prototype for bessj0 182 ⟩ ≡
  static double bessj0(double x)
```

This code is used in section 183.

183. ⟨ Definition for *bessj0* 183 ⟩ ≡

```
⟨ Prototype for bessj0 182 ⟩
{
  double ax, z;
  double xx, y, ans, ans1, ans2;
  if ((ax = fabs(x)) < 8.0) {
    y = x * x;
    ans1 = 57568490574.0 + y * (−13362590354.0 + y * (651619640.7 + y * (−11214424.18 + y *
        (77392.33017 + y * (−184.9052456)))));
    ans2 = 57568490411.0 + y * (1029532985.0 + y * (9494680.718 + y * (59272.64853 + y * (267.8532712 +
        y * 1.0))));
    ans = ans1 / ans2;
  }
  else {
    z = 8.0/ax;
    y = z * z;
    xx = ax − 0.785398164;
    ans1 = 1.0 + y * (−0.1098628627 · 10^−2 + y * (0.2734510407 · 10^−4 + y * (−0.2073370639 · 10^−5 + y *
        0.2093887211 · 10^−6)));
    ans2 = −0.1562499995 · 10^−1 + y * (0.1430488765 · 10^−3 + y * (−0.6911147651 · 10^−5 + y *
        (0.7621095161 · 10^−6 − y * 0.934935152 · 10^−7)));
    ans = sqrt(0.636619772/ax) * (cos(xx) * ans1 − z * sin(xx) * ans2);
  }
  return ans;
}
```

This code is used in section 180.

184. Bessel function $Y_0(x)$ for positive x

```
⟨ Prototype for bessy0 184 ⟩ ≡
  static double bessy0(double x)
```

This code is used in section 185.

185. ⟨ Definition for *bessy0* 185 ⟩ ≡

```
⟨ Prototype for bessy0 184 ⟩
{
  double z;
  double xx, y, ans, ans1, ans2;
  if (x < 8.0) {
    y = x * x;
    ans1 = -2957821389.0 + y * (7062834065.0 + y * (-512359803.6 + y * (10879881.29 + y *
        (-86327.92757 + y * 228.4622733))));
    ans2 = 40076544269.0+y*(745249964.8+y*(7189466.438+y*(47447.26470+y*(226.1030244+y*1.0))));
    ans = (ans1 / ans2) + 0.636619772 * bessj0(x) * log(x);
  }
  else {
    z = 8.0/x;
    y = z * z;
    xx = x - 0.785398164;
    ans1 = 1.0 + y * (-0.1098628627 · 10^-2 + y * (0.2734510407 · 10^-4 + y * (-0.2073370639 · 10^-5 + y *
        0.2093887211 · 10^-6)));
    ans2 = -0.1562499995 · 10^-1 + y * (0.1430488765 · 10^-3 + y * (-0.6911147651 · 10^-5 + y *
        (0.7621095161 · 10^-6 + y * (-0.934945152 · 10^-7))));
    ans = sqrt(0.636619772/x) * (sin(xx) * ans1 + z * cos(xx) * ans2);
  }
  return ans;
}
```

This code is used in section 180.

186. Bessel function $J_1(x)$ for real x

⟨ Prototype for *bessj1* 186 ⟩ ≡

```
static double bessj1(double x)
```

This code is used in section 187.

187. ⟨ Definition for *bessj1* 187 ⟩ ≡

```
⟨ Prototype for bessj1 186 ⟩
{
  double ax, z;
  double xx, y, ans, ans1, ans2;
  if ((ax = fabs(x)) < 8.0) {
    y = x * x;
    ans1 = x * (72362614232.0 + y * (−7895059235.0 + y * (242396853.1 + y * (−2972611.439 + y *
        (15704.48260 + y * (−30.16036606))))));
    ans2 = 144725228442.0 + y * (2300535178.0 + y * (18583304.74 + y * (99447.43394 + y * (376.9991397 +
        y * 1.0))));
    ans = ans1 / ans2;
  }
  else {
    z = 8.0/ax;
    y = z * z;
    xx = ax − 2.356194491;
    ans1 = 1.0 + y * (0.183105 · 10^−2 + y * (−0.3516396496 · 10^−4 + y * (0.2457520174 · 10^−5 + y *
        (−0.240337019 · 10^−6))));
    ans2 = 0.04687499995 + y * (−0.2002690873 · 10^−3 + y * (0.8449199096 · 10^−5 + y * (−0.88228987 · 10^−6 +
        y * 0.105787412 · 10^−6)));
    ans = sqrt(0.636619772/ax) * (cos(xx) * ans1 − z * sin(xx) * ans2);
    if (x < 0.0) ans = −ans;
  }
  return ans;
}
```

This code is used in section 180.

188. Bessel function $Y_1(x)$ for positive x

⟨ Prototype for *bessy1* 188 ⟩ ≡

```
static double bessy1(double x)
```

This code is used in section 189.

189. ⟨ Definition for *bessy1* 189 ⟩ ≡

```
⟨ Prototype for bessy1 188 ⟩
{
  double z;
  double xx, y, ans, ans1, ans2;
  if (x < 8.0) {
    y = x * x;
    ans1 = x * (−0.4900604943 · 10^13 + y * (0.1275274390 · 10^13 + y * (−0.5153438139 · 10^11 + y *
        (0.7349264551 · 10^9 + y * (−0.4237922726 · 10^7 + y * 0.8511937935 · 10^4)))));
    ans2 = 0.2499580570 · 10^14 + y * (0.4244419664 · 10^12 + y * (0.3733650367 · 10^10 + y *
        (0.2245904002 · 10^8 + y * (0.1020426050 · 10^6 + y * (0.3549632885 · 10^3 + y)))));
    ans = (ans1 / ans2) + 0.636619772 * (bessj1(x) * log(x) − 1.0/x);
  }
  else {
    z = 8.0/x;
    y = z * z;
    xx = x − 2.356194491;
    ans1 = 1.0 + y * (0.183105 · 10^−2 + y * (−0.3516396496 · 10^−4 + y * (0.2457520174 · 10^−5 + y *
        (−0.240337019 · 10^−6))));
    ans2 = 0.04687499995+y*(−0.2002690873 · 10^−3+y*(0.8449199096 · 10^−5+y*(−0.88228987 · 10^−6+
        y * 0.105787412 · 10^−6)));
    ans = sqrt(0.636619772/x) * (sin(xx) * ans1 + z * cos(xx) * ans2);
  }
  return ans;
}
```

This code is used in section 180.

190. Compute $(J_0[x], \ldots, J_n[x])$ and $(Y_0[x], \ldots, Y_n[x])$ for real x.

⟨ Prototype for *jn_real* 190 ⟩ ≡

```
static void jn_real(double x, int n, double *BesselJn, double *BesselYn)
```

This code is used in section 191.

191. ⟨ Definition for *jn_real* 191 ⟩ ≡

```
⟨ Prototype for jn_real 190 ⟩
{
  const double iacc = 40;
  const double bigno = 1.0 · 10^{+20};
  const double bigni = 1.0 · 10^{−20};
  int j, m, jsum;
  double tox, bj, bjp, bjm, sum;
  BesselJn[0] = bessj0(x);
  BesselJn[1] = bessj1(x);
  BesselYn[0] = bessy0(x);
  BesselYn[1] = bessy1(x);
  tox = 2./x;
  for (j = 1; j < n; j++) BesselYn[j + 1] = j * tox * BesselYn[j] − BesselYn[j − 1];
  if (x > n) {
    for (j = 1; j < n; j++) BesselJn[j + 1] = j * tox * BesselJn[j] − BesselJn[j − 1];
    return;
  }
  m = 2 * floor((n + sqrt(iacc * n))/2);
  jsum = 0;
  sum = 0.0;
  bjp = 0.0;
  bj = 1.0;
  for (j = m; j > 0; j−−) {
    bjm = j * tox * bj − bjp;
    bjp = bj;
    bj = bjm;
    if (fabs(bj) > bigno) {
      int i;
      bj = bj * bigni;
      bjp = bjp * bigni;
      sum = sum * bigni;
      for (i = j + 1; i ≤ n; i++) BesselJn[i] *= bigni;
    }
    if (jsum) sum = sum + bj;
    jsum = 1 − jsum;
    if (j ≤ n ∧ j ≥ 2) BesselJn[j] = bjp;
  }
  sum = 2.0 * sum − bj;
  for (j = 2; j ≤ n; j++) BesselJn[j] /= sum;
}
```

This code is used in section 180.

192. Compute $(J_0[z], \ldots, J_n[z])$ for complex z.

⟨ Prototype for *jn_complex* 192 ⟩ ≡

```
static void jn_complex(struct c_complex z, int n, struct c_complex *Jn)
```

This code is used in section 193.

193. ⟨ Definition for *jn_complex* 193 ⟩ ≡

```
⟨ Prototype for jn_complex 192 ⟩
{
  struct c_complex a, *JnTmp;
  int nd, i;
  nd = 2 * floor((pow(101 + c_abs(z), 0.499) + n)/2);
  JnTmp = new_carray((long) nd + 1);
  JnTmp[nd] = c_set(0.0, 0.0);
  JnTmp[nd - 1] = c_set(1.0 · 10^-32, 0.0);
  a = c_set(0.0, 0.0);
  for (i = nd - 1; i ≥ 3; i -= 2) {
    JnTmp[i - 1] = c_sub(c_smul(2.0 * i, c_div(JnTmp[i], z)), JnTmp[i + 1]);
    JnTmp[i - 2] = c_sub(c_smul(2.0 * (i - 1), c_div(JnTmp[i - 1], z)), JnTmp[i]);
    a.re += JnTmp[i - 1].re;
    a.im += JnTmp[i - 1].im;
  }
  JnTmp[0] = c_sub(c_smul(2.0, c_div(JnTmp[1], z)), JnTmp[2]);
  a.re = 2.0 * a.re + JnTmp[0].re;
  a.im = 2.0 * a.im + JnTmp[0].im;
  for (i = 0; i < n; i++) Jn[i] = c_div(JnTmp[i], a);
  free_carray(JnTmp);
}
```

This code is used in section 180.

194. Calculates *n_terms* multipole coefficients for EM scattering by an infinite cylinder. Case 1 coefficients, *an1* and *bn1*, corresponds to the incident electric field parallel to the x-z plane. Case 2 coefficients, *an2* and *bn2*, corresponds to the case when the electric field is prependicular to the x-z plane. See §8.4 of Bohren and Huffman for details.

x is the usual size parameter and m is the relative (complex) index of refraction of the cylinder. *zeta* is the incident angle of radiation with respect to the z-axis (in radians). Normal incidence is when *zeta*$= \pi/2$.

⟨ Prototype for *MieCylinderCoefficients* 194 ⟩ ≡

```
void MieCylinderCoefficients(double x, struct c_complex m, double zeta, int n_terms, struct
    c_complex an1[], struct c_complex bn1[], struct c_complex an2[], struct c_complex bn2[])
```

This code is used in sections 181 and 195.

195. ⟨ Definition for *MieCylinderCoefficients* 195 ⟩ ≡

```
⟨ Prototype for MieCylinderCoefficients 194 ⟩
{
  struct c_complex *jn1, ci, eta, feta, m2xi;
  double *BesselJn, *BesselYn, jnp, ynp, sin_zeta, xi, cos_zeta;
  int i;
  long size = n_terms + 1;
  jn1 = new_carray(size);
  BesselJn = new_darray(size);
  BesselYn = new_darray(size);
  cos_zeta = cos(zeta);
  sin_zeta = sqrt((1.0 + cos_zeta) * (1.0 - cos_zeta));
  ci = c_set(0.0, 1.0);
  eta = c_smul(x, c_sqrt(c_sub(c_sqr(m), c_set(cos_zeta * cos_zeta, 0.0))));
  xi = x * sin_zeta;
  feta = c_mul(c_smul(cos_zeta, eta), c_sadd(-1.0, c_sdiv(xi * xi, c_sqr(eta))));
  jn_real(xi, n_terms + 1, BesselJn, BesselYn);
  jn_complex(eta, n_terms + 1, jn1);
  m2xi = c_smul(xi, c_sqr(m));
  for (i = 0; i < n_terms; i++) {
    struct c_complex hn, hnp, dn1, an, bn, cn, dn, vn, wn, den;
    dn1 = c_div(c_sub(c_div(c_smul((double) i, jn1[i]), eta), jn1[i + 1]), jn1[i]);
    jnp = i * BesselJn[i]/xi - BesselJn[i + 1];
    ynp = i * BesselYn[i]/xi - BesselYn[i + 1];
    hn = c_set(BesselJn[i], BesselYn[i]);
    hnp = c_set(jnp, ynp);
    dn = c_smul((double) i, c_mul(hn, feta));
    cn = c_smul((double) i, c_smul(BesselJn[i], feta));
    bn = c_smul(xi, c_sub(c_smul(BesselJn[i], c_mul(m2xi, dn1)), c_smul(jnp, eta)));
    vn = c_smul(xi, c_sub(c_mul(hn, c_mul(m2xi, dn1)), c_mul(hnp, eta)));
    wn = c_mul(c_smul(xi, ci), c_sub(c_mul(hnp, eta), c_smul(xi, c_mul(dn1, hn))));
    an = c_mul(c_smul(-xi, ci), c_sub(c_smul(jnp, eta), c_smul(xi * BesselJn[i], dn1)));
    den = c_add(c_mul(wn, vn), c_mul(ci, c_sqr(dn)));
    an1[i] = c_div(c_sub(c_mul(cn, vn), c_mul(bn, dn)), den);
    bn1[i] = c_div(c_add(c_mul(wn, bn), c_mul(c_mul(ci, dn), cn)), den);
    an2[i] = c_div(c_sub(c_mul(c_mul(ci, dn), cn), c_mul(an, vn)), den);
    bn2[i] = c_mul(c_add(c_mul(cn, wn), c_mul(an, dn)), c_div(ci, den));
    bn2[i] = c_smul(-1.0, bn2[i]);
  }
  free_carray(jn1);
  free_darray(BesselJn);
  free_darray(BesselYn);
}
```

This code is used in section 180.

196. Here we calculate Mie scattering for a cylinder.

x is the usual size parameter and m is the relative (complex) index of refraction of the cylinder.

zeta is the incident angle (cf. figure 8.3 in Bohren and Huffman) of radiation with respect to the z-axis (in radians). Normal incidence is when $zeta = \pi/2$.

The vector *theta* is a list of length *nangles* that has the angles (in radians) for calculating the scattering phase function. If *theta* is Λ or if *nangles* is zero, no phase function angles are calculated.

The scattering matrix elements *t1*, *t2*, and *t3* are described on page 202 of Bohren and Huffman. Specifically

$$\begin{pmatrix} E_{\parallel s} \\ E_{\perp s} \end{pmatrix} = e^{i3\pi/4}\sqrt{\frac{2}{\pi kr\sin\zeta}}e^{ik(r\sin\zeta - z\cos\zeta)}\begin{pmatrix} T_1 & T_4 \\ T_3 & T_2 \end{pmatrix}\begin{pmatrix} E_{\parallel i} \\ E_{\perp i} \end{pmatrix}$$

⟨ Prototype for *MieCylinder* 196 ⟩ ≡

void *MieCylinder*(**double** x, **struct c_complex** m, **double** *zeta*, **const double** $*$*theta*, **int** *nangles*, **struct c_complex** $*$*t1*, **struct c_complex** $*$*t2*, **struct c_complex** $*$*t3*, **struct c_complex** $*$*qexpar*, **struct c_complex** $*$*qexper*, **double** $*$*qscpar*, **double** $*$*qscper*)

This code is used in sections 181 and 197.

197. ⟨ Definition for *MieCylinder* 197 ⟩ ≡

```
⟨ Prototype for MieCylinder 196 ⟩
{
  struct c_complex *an1, *bn1, *an2, *bn2;
  int i, k, n_stop_terms;
  long size;
  n_stop_terms = x + 4.0 * pow(x, 1.0/3.0) + 2.0;
  size = n_stop_terms + 1;
  an1 = new_carray(size);
  bn1 = new_carray(size);
  an2 = new_carray(size);
  bn2 = new_carray(size);
  MieCylinderCoefficients(x, m, zeta, n_stop_terms, an1, bn1, an2, bn2);
  *qexpar = c_smul(0.5, bn1[0]);
  *qexper = c_smul(0.5, an2[0]);
  *qscpar = 0.5 * c_norm(bn1[0]);
  *qscper = 0.5 * c_norm(an2[0]);
  for (i = 1; i < n_stop_terms; i++) {
    qexpar→re += bn1[i].re;
    qexpar→im += bn1[i].im;
    qexper→re += an2[i].re;
    qexper→im += an2[i].im;
    *qscpar += c_norm(an1[i]) + c_norm(bn1[i]);
    *qscper += c_norm(an2[i]) + c_norm(bn2[i]);
  }
  *qscpar *= 4.0/x;
  *qscper *= 4.0/x;
  qexpar→re *= 4.0/x;
  qexpar→im *= 4.0/x;
  qexper→re *= 4.0/x;
  qexper→im *= 4.0/x;
  if (¬theta) nangles = 0;
  for (i = 0; i < nangles; i++) {
    double t = theta[i];
    t1[i] = c_smul(0.5, bn1[0]);
    t2[i] = c_smul(0.5, an2[0]);
    t3[i] = c_set(0.0, 0.0);
    for (k = 1; k ≤ n_stop_terms; k++) {
      double ct = cos(k * t);
      double st = sin(k * t);
      t1[i].re += ct * bn1[k].re;
      t1[i].im += ct * bn1[k].im;
      t2[i].re += ct * an2[k].re;
      t2[i].im += ct * an2[k].im;
      t3[i].re += st * an1[k].re;
      t3[i].im += st * an1[k].im;
    }
  }
  free_carray(an1);
  free_carray(an2);
  free_carray(bn1);
```

```
    free_carray(bn2);
  }
```

This code is used in section 180.

198. A driver program for the cylindrical Mie scattering code.

199. Here, then, is an overview of document structure

⟨ mie_cylinder_main.c 199 ⟩ ≡
```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include "mie_array.h"
#include "mie_complex.h"
#include "mie_cylinder.h"
#define PI 3.14159265358979
```
⟨ Definition for *mie_cylinder_main* 200 ⟩

200. ⟨ Definition for *mie_cylinder_main* 200 ⟩ ≡

```
int main(int argc, char **argv)
{
  struct c_complex m, *t1, *t2, *t3, qexpar, qexper;
  double qscpar, qscper, x, zeta, lambda, diameter, qsca, t1norm, *theta;
  int i, nangles;
  long size;
  double n_medium, n_cylinder, k_cylinder;
  nangles = 21;
  size = nangles;
  t1 = new_carray(size);
  t2 = new_carray(size);
  t3 = new_carray(size);
  theta = new_darray(size);
  for (i = 0; i < nangles; i++) theta[i] = (double) i * PI/(nangles - 1.0);
  lambda = 632.8;
  diameter = 1050;
  x = PI * diameter/lambda;
  n_cylinder = 1.55;
  n_medium = 1.0;
  k_cylinder = 0.0;
  m = c_set(n_cylinder/n_medium, k_cylinder/n_medium);
  zeta = 90.0 * PI/180.0;
  MieCylinder(x, m, zeta, theta, nangles, t1, t2, t3, &qexpar, &qexper, &qscpar, &qscper);
  qsca = 0.5 * (qscpar + qscper);
  printf("zeta␣␣␣␣␣␣␣=␣%8.4f␣degrees\n", zeta * 180/PI);
  printf("n_medium␣␣␣=␣%8.4f\n", n_medium);
  printf("n_cylinder␣=␣%8.4f␣+␣%8.4fi\n", n_cylinder, k_cylinder);
  printf("lambda␣␣␣␣␣=␣%8.1f␣nm\n", lambda);
  printf("diameter␣␣␣=␣%8.1f␣nm\n", diameter);
  printf("\n");
  printf("x␣␣␣␣␣␣␣␣␣␣␣=␣%8.4f\n", x);
  printf("qexpar␣␣␣␣␣=␣%8.4f␣+␣%8.4fi\n", qexpar.re, qexpar.im);
  printf("qscpar␣␣␣␣␣=␣%8.4f\n", qscpar);
  printf("qexper␣␣␣␣␣=␣%8.4f␣+␣%8.4fi\n", qexper.re, qexper.im);
  printf("qscper␣␣␣␣␣=␣%8.4f\n", qscper);
  printf("\n");
  t1norm = 0.5 * c_norm(t1[0]) + 0.5 * c_norm(t2[0]);
  for (i = 0; i < nangles; i++) {
    double tpar, tper, t11, t12, pol, s1, s2, s3, t33, t34;
    tpar = c_norm(t1[i]);
    tper = c_norm(t2[i]);
    t11 = 0.5 * (tpar + tper);
    t12 = 0.5 * (tpar - tper);
    t33 = t1[i].re * t2[i].re + t1[i].im * t2[i].im;
    t34 = t1[i].im * t2[i].re - t1[i].re * t2[i].im;
    t33 /= t11;
    t34 /= t11;
    pol = t12/t11;
    s1 = 4 * c_norm(t1[i])/qsca;
    s2 = 4 * c_norm(t2[i])/qsca;
    s3 = 4 * c_norm(t3[i])/qsca;
```

```
        printf("%8.3f␣\t␣%8.5f␣\t␣%8.5f␣\t␣%8.5f␣\t␣%8.5f\n", theta[i] * 180/PI, t11/t1norm, pol,
            t33, t34);
    }
    free_carray(t1);
    free_carray(t2);
    free_carray(t3);
    return (0);
}
```

This code is used in section 199.

201. Index. Here is a cross-reference table for the Mie scattering program. All sections in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in section names are not indexed. Underlined entries correspond to where the identifier was declared. Error messages and a few other things like "ASCII code dependencies" are indexed here too.

⟨ Allocate angle based arrays 172 ⟩ Used in section 168.
⟨ Calculate $\hat{a}_1$ 137 ⟩ Used in section 136.
⟨ Calculate $\hat{a}_2$ 139 ⟩ Used in section 136.
⟨ Calculate $\hat{b}_1$ 138 ⟩ Used in section 136.
⟨ Calculate Efficiencies 160 ⟩ Used in section 148.
⟨ Calculate first *alpha* and *beta* 125 ⟩ Used in section 124.
⟨ Calculate next *ratio* 126 ⟩ Used in section 124.
⟨ Calculate phase function for each angle 157 ⟩ Used in section 148.
⟨ Calculate small Mie efficiencies and asymmetry 141 ⟩ Used in section 136.
⟨ Calculate small Mie scattering amplitudes 142 ⟩ Used in section 136.
⟨ Calculate the logarithmic derivatives 154 ⟩ Used in section 148.
⟨ Calculate *nstop* 153 ⟩ Used in section 148.
⟨ Catch bogus input values 150 ⟩ Used in section 148.
⟨ Deal with small spheres 151 ⟩ Used in section 148.
⟨ Declare Mie variables 170 ⟩ Used in section 168.
⟨ Declare variables for *Mie* 149 ⟩ Used in section 148.
⟨ Definition for *Dn_down* 133 ⟩ Used in section 115.
⟨ Definition for *Dn_up* 130 ⟩ Used in section 115.
⟨ Definition for *Lentz_Dn* 124 ⟩ Used in section 115.
⟨ Definition for *MieCylinderCoefficients* 195 ⟩ Used in section 180.
⟨ Definition for *MieCylinder* 197 ⟩ Used in section 180.
⟨ Definition for *Mie* 148 ⟩ Used in section 115.
⟨ Definition for *array_error* 7 ⟩ Used in section 3.
⟨ Definition for *bessj0* 183 ⟩ Used in section 180.
⟨ Definition for *bessj1* 187 ⟩ Used in section 180.
⟨ Definition for *bessy0* 185 ⟩ Used in section 180.
⟨ Definition for *bessy1* 189 ⟩ Used in section 180.
⟨ Definition for *c_abs* 41 ⟩ Used in section 31.
⟨ Definition for *c_acos* 84 ⟩ Used in section 31.
⟨ Definition for *c_add* 56 ⟩ Used in section 31.
⟨ Definition for *c_arg* 45 ⟩ Used in section 31.
⟨ Definition for *c_asinh* 97 ⟩ Used in section 31.
⟨ Definition for *c_asin* 82 ⟩ Used in section 31.
⟨ Definition for *c_atanh* 95 ⟩ Used in section 31.
⟨ Definition for *c_atan* 86 ⟩ Used in section 31.
⟨ Definition for *c_conj* 43 ⟩ Used in section 31.
⟨ Definition for *c_cosh* 89 ⟩ Used in section 31.
⟨ Definition for *c_cos* 78 ⟩ Used in section 31.
⟨ Definition for *c_div* 62 ⟩ Used in section 31.
⟨ Definition for *c_exp* 100 ⟩ Used in section 31.
⟨ Definition for *c_inv* 53 ⟩ Used in section 31.
⟨ Definition for *c_log10* 104 ⟩ Used in section 31.
⟨ Definition for *c_log* 102 ⟩ Used in section 31.
⟨ Definition for *c_mul* 60 ⟩ Used in section 31.
⟨ Definition for *c_norm* 47 ⟩ Used in section 31.
⟨ Definition for *c_polarset* 39 ⟩ Used in section 31.
⟨ Definition for *c_rdiv* 64 ⟩ Used in section 31.
⟨ Definition for *c_rmul* 66 ⟩ Used in section 31.
⟨ Definition for *c_sadd* 71 ⟩ Used in section 31.
⟨ Definition for *c_sdiv* 73 ⟩ Used in section 31.
⟨ Definition for *c_set* 37 ⟩ Used in section 31.
⟨ Definition for *c_sinh* 91 ⟩ Used in section 31.

⟨ Definition for *c_sin* 76 ⟩ Used in section 31.
⟨ Definition for *c_smul* 69 ⟩ Used in section 31.
⟨ Definition for *c_sqrt* 49 ⟩ Used in section 31.
⟨ Definition for *c_sqr* 51 ⟩ Used in section 31.
⟨ Definition for *c_sub* 58 ⟩ Used in section 31.
⟨ Definition for *c_tanh* 93 ⟩ Used in section 31.
⟨ Definition for *c_tan* 80 ⟩ Used in section 31.
⟨ Definition for *complex_error* 35 ⟩ Used in section 31.
⟨ Definition for *copy_carray* 111 ⟩ Used in section 31.
⟨ Definition for *copy_darray* 13 ⟩ Used in section 3.
⟨ Definition for *ez_Mie_Full* 166 ⟩ Used in section 115.
⟨ Definition for *ez_Mie* 164 ⟩ Used in section 115.
⟨ Definition for *free_carray* 109 ⟩ Used in section 31.
⟨ Definition for *free_darray* 11 ⟩ Used in section 3.
⟨ Definition for *jn_complex* 193 ⟩ Used in section 180.
⟨ Definition for *jn_real* 191 ⟩ Used in section 180.
⟨ Definition for *mie_cylinder_main* 200 ⟩ Used in section 199.
⟨ Definition for *mie_error* 119 ⟩ Used in section 115.
⟨ Definition for *min_max_darray* 17 ⟩ Used in section 3.
⟨ Definition for *new_carray* 107 ⟩ Used in section 31.
⟨ Definition for *new_darray* 9 ⟩ Used in section 3.
⟨ Definition for *print_darray* 23 ⟩ Used in section 3.
⟨ Definition for *set_carray* 113 ⟩ Used in section 31.
⟨ Definition for *set_darray* 15 ⟩ Used in section 3.
⟨ Definition for *small_Mie* 136 ⟩ Used in section 115.
⟨ Definition for *small_conducting_Mie* 145 ⟩ Used in section 115.
⟨ Definition for *sort_darray* 20 ⟩ Used in section 3.
⟨ Establish a_n and b_n 156 ⟩ Used in section 148.
⟨ Free allocated memory 161 ⟩ Used in section 148.
⟨ Free angle based arrays 173 ⟩ Used in section 168.
⟨ Handle options 171 ⟩ Used in section 168.
⟨ Increment cross sections 158 ⟩ Used in section 148.
⟨ Mie allocate and initialize angle arrays 152 ⟩ Used in section 148.
⟨ Prepare for the next iteration 159 ⟩ Used in section 148.
⟨ Prepare to sum over all *nstop* terms 155 ⟩ Used in section 148.
⟨ Print header 174 ⟩ Used in section 168.
⟨ Print phase function 176 ⟩ Used in section 168.
⟨ Print summary 175 ⟩ Used in section 168.
⟨ Prototype for *Dn_down* 132 ⟩ Used in sections 116 and 133.
⟨ Prototype for *Dn_up* 129 ⟩ Used in sections 116 and 130.
⟨ Prototype for *Lentz_Dn* 123 ⟩ Used in sections 116 and 124.
⟨ Prototype for *MieCylinderCoefficients* 194 ⟩ Used in sections 181 and 195.
⟨ Prototype for *MieCylinder* 196 ⟩ Used in sections 181 and 197.
⟨ Prototype for *Mie* 147 ⟩ Used in sections 116 and 148.
⟨ Prototype for *array_error* 6 ⟩ Used in section 7.
⟨ Prototype for *bessj0* 182 ⟩ Used in section 183.
⟨ Prototype for *bessj1* 186 ⟩ Used in section 187.
⟨ Prototype for *bessy0* 184 ⟩ Used in section 185.
⟨ Prototype for *bessy1* 188 ⟩ Used in section 189.
⟨ Prototype for *c_abs* 40 ⟩ Used in sections 32 and 41.
⟨ Prototype for *c_acos* 83 ⟩ Used in sections 32 and 84.
⟨ Prototype for *c_add* 55 ⟩ Used in sections 32 and 56.