1. 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.

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

⟨ `array.c` 2 ⟩ ≡
```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <float.h>
#include "array.h"
```
⟨ Definition for *array_error* 6 ⟩
⟨ Definition for *new_darray* 8 ⟩
⟨ Definition for *free_darray* 10 ⟩
⟨ Definition for *copy_darray* 12 ⟩
⟨ Definition for *set_darray* 14 ⟩
⟨ Definition for *min_max_darray* 16 ⟩
⟨ Definition for *sort_darray* 19 ⟩
⟨ Definition for *print_darray* 22 ⟩

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

⟨ `array.h` 3 ⟩ ≡
⟨ Prototype for *new_darray* 7 ⟩;
⟨ Prototype for *free_darray* 9 ⟩;
⟨ Prototype for *copy_darray* 11 ⟩;
⟨ Prototype for *set_darray* 13 ⟩;
⟨ Prototype for *min_max_darray* 15 ⟩;
⟨ Prototype for *sort_darray* 18 ⟩;
⟨ Prototype for *print_darray* 21 ⟩;

4. Allocation.

5. A simple error routine.

```
⟨ Prototype for array_error 5 ⟩ ≡
  static void array_error(char *s)
```

This code is used in section 6.

6. ⟨ Definition for *array_error* 6 ⟩ ≡

```
⟨ Prototype for array_error 5 ⟩
{
  printf("Array␣--␣%s\n", s);
  exit(1);
}
```

This code is used in section 2.

7. Double array routines.

⟨ Prototype for *new_darray* 7 ⟩ ≡
double ∗*new_darray*(**long** *size*)
This code is used in sections 3 and 8.

8. ⟨ Definition for *new_darray* 8 ⟩ ≡
⟨ Prototype for *new_darray* 7 ⟩
{
double ∗*a*;
if (*size* ≤ 0) *array_error*("Non-positive␣double␣array␣size␣chosen");
size += 2;
a = (**double** ∗) *malloc*(**sizeof**(**double**) ∗ *size*);
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 2.

9. ⟨ Prototype for *free_darray* 9 ⟩ ≡
void *free_darray*(**double** ∗*a*)
This code is used in sections 3 and 10.

10. ⟨ Definition for *free_darray* 10 ⟩ ≡
⟨ Prototype for *free_darray* 9 ⟩
{
if (*a* ≠ Λ) *free*(*a* − 1);
}
This code is used in section 2.

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

⟨ Prototype for *copy_darray* 11 ⟩ ≡
double ∗*copy_darray*(**double** ∗*a*, **long** *size*)
This code is used in sections 3 and 12.

12. ⟨ Definition for *copy_darray* 12 ⟩ ≡
⟨ Prototype for *copy_darray* 11 ⟩
{
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 2.

13. This sets all the entries in the array *a*[ ] to *x*.

⟨ Prototype for *set_darray* 13 ⟩ ≡
void *set_darray*(**double** ∗*a*, **long** *size*, **double** *x*)
This code is used in sections 3 and 14.

14. ⟨ Definition for *set_darray* 14 ⟩ ≡

```
⟨ Prototype for set_darray 13 ⟩
{
  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 2.

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

⟨ Prototype for *min_max_darray* 15 ⟩ ≡

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

This code is used in sections 3 and 16.

16. ⟨ Definition for *min_max_darray* 16 ⟩ ≡

```
⟨ Prototype for min_max_darray 15 ⟩
{
  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 2.

17. Sorting.

18. *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* 18 ⟩ ≡

```
void sort_darray(double *a, long size)
```

This code is used in sections 3 and 19.

19. ⟨ Definition for *sort_darray* 19 ⟩ ≡

```
⟨ Prototype for sort_darray 18 ⟩
{
  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 2.

20. Printing.

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

```
⟨ Prototype for print_darray 21 ⟩ ≡
  void print_darray(double *a, long size, long ilow, long ihigh)
```

This code is used in sections 3 and 22.

22.

```
⟨ Definition for print_darray 22 ⟩ ≡
  ⟨ Prototype for print_darray 21 ⟩
  {
    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 2.

23. Testing.

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

```
⟨ arraytest.c 24 ⟩ ≡
#include <stdio.h>
#include "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 25 ⟩
    ⟨ Test Copy Routine 26 ⟩
    ⟨ Test Sort Routine 27 ⟩
    ⟨ Test Min/Max Routine 28 ⟩
    printf("done\n");
    fflush(stdout);
  }
```

25. ⟨ Test Set Routine 25 ⟩ ≡

```
  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 24.

26. ⟨ Test Copy Routine 26 ⟩ ≡

```
  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 24.

27. ⟨ Test Sort Routine 27 ⟩ ≡

```
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 24.

28. ⟨ Test Min/Max Routine 28 ⟩ ≡

```
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 24.

29. 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.

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

⟨ `complex.c` 30 ⟩ ≡

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "complex.h"
```

⟨ Definition for *complex_error* 34 ⟩
⟨ Definition for *cset* 36 ⟩
⟨ Definition for *cpolarset* 38 ⟩
⟨ Definition for *cabs* 40 ⟩
⟨ Definition for *carg* 44 ⟩
⟨ Definition for *cnorm* 46 ⟩
⟨ Definition for *csqrt* 48 ⟩
⟨ Definition for *csqr* 50 ⟩
⟨ Definition for *cinv* 52 ⟩
⟨ Definition for *conj* 42 ⟩
⟨ Definition for *cadd* 55 ⟩
⟨ Definition for *csub* 57 ⟩
⟨ Definition for *cmul* 59 ⟩
⟨ Definition for *cdiv* 61 ⟩
⟨ Definition for *crdiv* 63 ⟩
⟨ Definition for *crmul* 65 ⟩
⟨ Definition for *csadd* 70 ⟩
⟨ Definition for *csdiv* 72 ⟩
⟨ Definition for *csmul* 68 ⟩
⟨ Definition for *csin* 75 ⟩
⟨ Definition for *ccos* 77 ⟩
⟨ Definition for *ctan* 79 ⟩
⟨ Definition for *casin* 81 ⟩
⟨ Definition for *cacos* 83 ⟩
⟨ Definition for *catan* 85 ⟩
⟨ Definition for *csinh* 90 ⟩
⟨ Definition for *ccosh* 88 ⟩
⟨ Definition for *ctanh* 92 ⟩
⟨ Definition for *catanh* 94 ⟩
⟨ Definition for *casinh* 96 ⟩
⟨ Definition for *cexp* 99 ⟩
⟨ Definition for *clog* 101 ⟩
⟨ Definition for *clog10* 103 ⟩
⟨ Definition for *new_carray* 106 ⟩
⟨ Definition for *free_carray* 108 ⟩
⟨ Definition for *copy_carray* 110 ⟩
⟨ Definition for *set_carray* 112 ⟩

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

⟨ `complex.h` 31 ⟩ ≡
```
struct complex {
  double re, im;
};
⟨ Prototype for cset 35 ⟩;
⟨ Prototype for cpolarset 37 ⟩;
⟨ Prototype for cabs 39 ⟩;
⟨ Prototype for carg 43 ⟩;
⟨ Prototype for csqr 49 ⟩;
⟨ Prototype for conj 41 ⟩;
⟨ Prototype for cnorm 45 ⟩;
⟨ Prototype for csqrt 47 ⟩;
⟨ Prototype for cinv 51 ⟩;
⟨ Prototype for cadd 54 ⟩;
⟨ Prototype for csub 56 ⟩;
⟨ Prototype for cmul 58 ⟩;
⟨ Prototype for cdiv 60 ⟩;
⟨ Prototype for crdiv 62 ⟩;
⟨ Prototype for crmul 64 ⟩;
⟨ Prototype for csadd 69 ⟩;
⟨ Prototype for csdiv 71 ⟩;
⟨ Prototype for csmul 67 ⟩;
⟨ Prototype for csin 74 ⟩;
⟨ Prototype for ccos 76 ⟩;
⟨ Prototype for ctan 78 ⟩;
⟨ Prototype for casin 80 ⟩;
⟨ Prototype for cacos 82 ⟩;
⟨ Prototype for catan 84 ⟩;
⟨ Prototype for csinh 89 ⟩;
⟨ Prototype for ccosh 87 ⟩;
⟨ Prototype for ctanh 91 ⟩;
⟨ Prototype for catanh 93 ⟩;
⟨ Prototype for casinh 95 ⟩;
⟨ Prototype for cexp 98 ⟩;
⟨ Prototype for clog 100 ⟩;
⟨ Prototype for clog10 102 ⟩;
⟨ Prototype for new_carray 105 ⟩;
⟨ Prototype for free_carray 107 ⟩;
⟨ Prototype for copy_carray 109 ⟩;
⟨ Prototype for set_carray 111 ⟩;
```

32. Basic routines.

33. A simple error routine.

```
⟨ Prototype for complex_error 33 ⟩ ≡
  static void complex_error(char *s)
```

This code is used in section 34.

34.
```
⟨ Definition for complex_error 34 ⟩ ≡
  ⟨ Prototype for complex_error 33 ⟩
  {
    printf("%s\n", s);
    exit(1);
  }
```

This code is used in section 30.

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

```
⟨ Prototype for cset 35 ⟩ ≡
  struct complex cset(double a, double b)
```

This code is used in sections 31 and 36.

36.
```
⟨ Definition for cset 36 ⟩ ≡
  ⟨ Prototype for cset 35 ⟩
  {
    struct complex c;
    c.re = a;
    c.im = b;
    return c;
  }
```

This code is used in section 30.

37. A variation on *cset* in which the complex number is specifed using polar coordinates.

```
⟨ Prototype for cpolarset 37 ⟩ ≡
  struct complex cpolarset(double r, double theta)
```

This code is used in sections 31 and 38.

38.
```
⟨ Definition for cpolarset 38 ⟩ ≡
  ⟨ Prototype for cpolarset 37 ⟩
  {
    return cset(r * cos(theta), r * sin(theta));
  }
```

This code is used in section 30.

39. 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 cabs 39 ⟩ ≡
  double cabs(struct complex z)
```

This code is used in sections 31 and 40.

40. ⟨ Definition for *cabs* 40 ⟩ ≡

```
⟨ Prototype for cabs 39 ⟩
{
  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 30.

41. Returns the conjugate of the complex number z

⟨ Prototype for *conj* 41 ⟩ ≡

```
struct complex conj(struct complex z)
```

This code is used in sections 31 and 42.

42. ⟨ Definition for *conj* 42 ⟩ ≡

```
⟨ Prototype for conj 41 ⟩
{
  return cset(z.re, -z.im);
}
```

This code is used in section 30.

43. ⟨ Prototype for *carg* 43 ⟩ ≡

```
double carg(struct complex z)
```

This code is used in sections 31 and 44.

44. ⟨ Definition for *carg* 44 ⟩ ≡

```
⟨ Prototype for carg 43 ⟩
{
  return atan2(z.im, z.re);
}
```

This code is used in section 30.

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

⟨ Prototype for *cnorm* 45 ⟩ ≡

```
double cnorm(struct complex z)
```

This code is used in sections 31 and 46.

46. ⟨ Definition for *cnorm* 46 ⟩ ≡

```
⟨ Prototype for cnorm 45 ⟩
{
  return (z.re * z.re + z.im * z.im);
}
```

This code is used in section 30.

47. ⟨ Prototype for *csqrt* 47 ⟩ ≡
```
struct complex csqrt(struct complex z)
```
This code is used in sections 31 and 48.

48. ⟨ Definition for *csqrt* 48 ⟩ ≡
```
⟨ Prototype for csqrt 47 ⟩
{
  double a, b;
  if ((z.re ≡ 0.0) ∧ (z.im ≡ 0.0)) return cset(0, 0);
  a = sqrt((fabs(z.re) + cabs(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 cset(a, b);
}
```
This code is used in section 30.

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

⟨ Prototype for *csqr* 49 ⟩ ≡
```
struct complex csqr(struct complex z)
```
This code is used in sections 31 and 50.

50. ⟨ Definition for *csqr* 50 ⟩ ≡
```
⟨ Prototype for csqr 49 ⟩
{
  return cmul(z, z);
}
```
This code is used in section 30.

51. Returns the reciprocal of z.

⟨ Prototype for *cinv* 51 ⟩ ≡
```
struct complex cinv(struct complex w)
```
This code is used in sections 31 and 52.

52. ⟨ Definition for *cinv* 52 ⟩ ≡
```
⟨ Prototype for cinv 51 ⟩
{
  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 cset(d, −r ∗ d);
  }
  r = w.re/w.im;
  d = 1/(w.im + r ∗ w.re);
  return cset(r ∗ d, −d);
}
```
This code is used in section 30.

53. Two complex numbers.

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

```
⟨ Prototype for cadd 54 ⟩ ≡
  struct complex cadd(struct complex z, struct complex w)
```

This code is used in sections 31 and 55.

55.

```
⟨ Definition for cadd 55 ⟩ ≡
  ⟨ Prototype for cadd 54 ⟩
  {
    struct complex c;
    c.im = z.im + w.im;
    c.re = z.re + w.re;
    return c;
  }
```

This code is used in section 30.

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

```
⟨ Prototype for csub 56 ⟩ ≡
  struct complex csub(struct complex z, struct complex w)
```

This code is used in sections 31 and 57.

57.

```
⟨ Definition for csub 57 ⟩ ≡
  ⟨ Prototype for csub 56 ⟩
  {
    struct complex c;
    c.im = z.im - w.im;
    c.re = z.re - w.re;
    return c;
  }
```

This code is used in section 30.

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

```
⟨ Prototype for cmul 58 ⟩ ≡
  struct complex cmul(struct complex z, struct complex w)
```

This code is used in sections 31 and 59.

59.

```
⟨ Definition for cmul 59 ⟩ ≡
  ⟨ Prototype for cmul 58 ⟩
  {
    struct 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 30.

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

⟨ Prototype for *cdiv* 60 ⟩ ≡
```
struct complex cdiv(struct complex z, struct complex w)
```
This code is used in sections 31 and 61.

61. ⟨ Definition for *cdiv* 61 ⟩ ≡
```
⟨ Prototype for cdiv 60 ⟩
{
  struct 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 30.

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

⟨ Prototype for *crdiv* 62 ⟩ ≡
```
double crdiv(struct complex z, struct complex w)
```
This code is used in sections 31 and 63.

63. ⟨ Definition for *crdiv* 63 ⟩ ≡
```
⟨ Prototype for crdiv 62 ⟩
{
  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 30.

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

```
⟨ Prototype for crmul 64 ⟩ ≡
  double crmul(struct complex z, struct complex w)
```

This code is used in sections 31 and 65.

65.

```
⟨ Definition for crmul 65 ⟩ ≡
  ⟨ Prototype for crmul 64 ⟩
  {
    return z.re * w.re - z.im * w.im;
  }
```

This code is used in section 30.

66. A scalar and a complex number.

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

```
⟨ Prototype for csmul 67 ⟩ ≡
  struct complex csmul(double x, struct complex z)
```

This code is used in sections 31 and 68.

68.

```
⟨ Definition for csmul 68 ⟩ ≡
  ⟨ Prototype for csmul 67 ⟩
  {
    struct complex c;
    c.re = z.re * x;
    c.im = z.im * x;
    return c;
  }
```

This code is used in section 30.

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

```
⟨ Prototype for csadd 69 ⟩ ≡
  struct complex csadd(double x, struct complex z)
```

This code is used in sections 31 and 70.

70.

```
⟨ Definition for csadd 70 ⟩ ≡
  ⟨ Prototype for csadd 69 ⟩
  {
    struct complex c;
    c.re = x + z.re;
    c.im = z.im;
    return c;
  }
```

This code is used in section 30.

71. Returns the quotient of real number by a complex number z. Again a special case of *cdiv*

```
⟨ Prototype for csdiv 71 ⟩ ≡
  struct complex csdiv(double x, struct complex w)
```

This code is used in sections 31 and 72.

72. ⟨ Definition for *csdiv* 72 ⟩ ≡

```
⟨ Prototype for csdiv 71 ⟩
{
  struct 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 30.

73. Trigonometric Functions.

74. The complex sine.

```
⟨ Prototype for csin 74 ⟩ ≡
  struct complex csin(struct complex z)
```

This code is used in sections 31 and 75.

75.

```
⟨ Definition for csin 75 ⟩ ≡
  ⟨ Prototype for csin 74 ⟩
  {
    return cset(sin(z.re) * cosh(z.im), cos(z.re) * sinh(z.im));
  }
```

This code is used in section 30.

76. The complex cosine.

```
⟨ Prototype for ccos 76 ⟩ ≡
  struct complex ccos(struct complex z)
```

This code is used in sections 31 and 77.

77.

```
⟨ Definition for ccos 77 ⟩ ≡
  ⟨ Prototype for ccos 76 ⟩
  {
    return cset(cos(z.re) * cosh(z.im), -(sin(z.re) * sinh(z.im)));
  }
```

This code is used in section 30.

78. The complex tangent.

```
⟨ Prototype for ctan 78 ⟩ ≡
  struct complex ctan(struct complex z)
```

This code is used in sections 31 and 79.

79.

```
⟨ Definition for ctan 79 ⟩ ≡
  ⟨ Prototype for ctan 78 ⟩
  {
    double x = 2 * z.re;
    double y = 2 * z.im;
    double t = cos(x) + cosh(y);
    if (t ≡ 0) complex_error("Complex␣tangent␣is␣infinite");
    return cset(sin(x)/t, sinh(y)/t);
  }
```

This code is used in section 30.

80. The complex inverse sine.

```
⟨ Prototype for casin 80 ⟩ ≡
  struct complex casin(struct complex z)
```

This code is used in sections 31 and 81.

81. ⟨ Definition for *casin* 81 ⟩ ≡

```
⟨ Prototype for casin 80 ⟩
{
  struct complex x;
  x = clog(cadd(cset(−z.im, z.re), csqrt(csub(cset(1, 0), cmul(z, z)))));
  return cset(x.im, −x.re);
}
```

This code is used in section 30.

82. The complex inverse cosine

⟨ Prototype for *cacos* 82 ⟩ ≡

```
struct complex cacos(struct complex z)
```

This code is used in sections 31 and 83.

83. ⟨ Definition for *cacos* 83 ⟩ ≡

```
⟨ Prototype for cacos 82 ⟩
{
  struct complex x;
  x = clog(cadd(z, cmul(cset(0, 1), csqrt(csub(cset(1, 0), csqr(z))))));
  return cset(x.im, −x.re);
}
```

This code is used in section 30.

84. The complex inverse tangent

⟨ Prototype for *catan* 84 ⟩ ≡

```
struct complex catan(struct complex z)
```

This code is used in sections 31 and 85.

85. ⟨ Definition for *catan* 85 ⟩ ≡

```
⟨ Prototype for catan 84 ⟩
{
  struct complex x;
  x = clog(cdiv(cset(z.re, 1 + z.im), cset(−z.re, 1 − z.im)));
  return cset(−x.im/2, x.re/2);
}
```

This code is used in section 30.

86. Hyperbolic functions.

87. ⟨ Prototype for *ccosh* 87 ⟩ ≡
```
struct complex ccosh(struct complex z)
```
This code is used in sections 31 and 88.

88. ⟨ Definition for *ccosh* 88 ⟩ ≡
```
⟨ Prototype for ccosh 87 ⟩
{
  return cset(cosh(z.re) * cos(z.im), sinh(z.re) * sin(z.im));
}
```
This code is used in section 30.

89. ⟨ Prototype for *csinh* 89 ⟩ ≡
```
struct complex csinh(struct complex z)
```
This code is used in sections 31 and 90.

90. ⟨ Definition for *csinh* 90 ⟩ ≡
```
⟨ Prototype for csinh 89 ⟩
{
  return cset(sinh(z.re) * cos(z.im), cosh(z.re) * sin(z.im));
}
```
This code is used in section 30.

91. ⟨ Prototype for *ctanh* 91 ⟩ ≡
```
struct complex ctanh(struct complex z)
```
This code is used in sections 31 and 92.

92. ⟨ Definition for *ctanh* 92 ⟩ ≡
```
⟨ Prototype for ctanh 91 ⟩
{
  double x = 2 * z.re;
  double y = 2 * z.im;
  double t = 1.0/(cosh(x) + cos(y));
  return cset(t * sinh(x), t * sin(y));
}
```
This code is used in section 30.

93. ⟨ Prototype for *catanh* 93 ⟩ ≡
```
struct complex catanh(struct complex z)
```
This code is used in sections 31 and 94.

94. ⟨ Definition for *catanh* 94 ⟩ ≡
```
⟨ Prototype for catanh 93 ⟩
{
  return catan(cset(-z.im, z.re));
}
```
This code is used in section 30.

95. ⟨ Prototype for *casinh* 95 ⟩ ≡
```
struct complex casinh(struct complex z)
```
This code is used in sections 31 and 96.

96. ⟨ Definition for *casinh* 96 ⟩ ≡

```
⟨ Prototype for casinh 95 ⟩
{
  return casin(cset(−z.im, z.re));
}
```

This code is used in section 30.

97. Exponentials and logarithms.

98. ⟨ Prototype for *cexp* 98 ⟩ ≡
```
struct complex cexp(struct complex z)
```
This code is used in sections 31 and 99.

99. ⟨ Definition for *cexp* 99 ⟩ ≡
```
⟨ Prototype for cexp 98 ⟩
{
  double x = exp(z.re);
  return cset(x * cos(z.im), x * sin(z.im));
}
```
This code is used in section 30.

100. ⟨ Prototype for *clog* 100 ⟩ ≡
```
struct complex clog(struct complex z)
```
This code is used in sections 31 and 101.

101. ⟨ Definition for *clog* 101 ⟩ ≡
```
⟨ Prototype for clog 100 ⟩
{
  return cset(log(cabs(z)), carg(z));
}
```
This code is used in section 30.

102. ⟨ Prototype for *clog10* 102 ⟩ ≡
```
struct complex clog10(struct complex z)
```
This code is used in sections 31 and 103.

103. ⟨ Definition for *clog10* 103 ⟩ ≡
```
⟨ Prototype for clog10 102 ⟩
{
  return cset(0.2171472409516259 * log(cnorm(z)), carg(z));
}
```
This code is used in section 30.

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

105. ⟨ Prototype for *new_carray* 105 ⟩ ≡
```
struct complex *new_carray(long size)
```
This code is used in sections 31 and 106.

106. ⟨ Definition for *new_carray* 106 ⟩ ≡
```
⟨ Prototype for new_carray 105 ⟩
{
  struct complex *a;
  if (size ≤ 0) complex_error("Non-positive␣complex␣array␣size␣chosen");
  a = (struct complex *) malloc(size * sizeof(struct complex));
  if (a ≡ Λ) complex_error("Can't␣allocate␣complex␣array");
  return a;
}
```
This code is used in section 30.

107. ⟨ Prototype for *free_carray* 107 ⟩ ≡
```
void free_carray(struct complex *a)
```
This code is used in sections 31 and 108.

108. ⟨ Definition for *free_carray* 108 ⟩ ≡
```
⟨ Prototype for free_carray 107 ⟩
{
  if (a ≠ Λ) free(a);
}
```
This code is used in section 30.

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

⟨ Prototype for *copy_carray* 109 ⟩ ≡
```
struct complex *copy_carray(struct complex *a, long size)
```
This code is used in sections 31 and 110.

110. ⟨ Definition for *copy_carray* 110 ⟩ ≡
```
⟨ Prototype for copy_carray 109 ⟩
{
  struct 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 complex));
  return b;
}
```
This code is used in section 30.

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

⟨ Prototype for *set_carray* 111 ⟩ ≡
```
void set_carray(struct complex *a, long size, struct complex z)
```
This code is used in sections 31 and 112.

112. ⟨ Definition for *set_carray* 112 ⟩ ≡

```
⟨ Prototype for set_carray 111 ⟩
{
  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 30.

113. Legendre Polynomials.

To calculate the values of the Legendre polynomial using the recurrence relations given by H. H. Michels, in "Abscissas and weight coefficients for Lobatto quadrature," *Math Comp*, **17**, 237-244 (1963).

These were checked for $n = 10$ for $-1 <= x <= 1$ using the basic differential equation

$$(1 - x^2)P_n''(x) - 2xP_n'(x) + n(n + 1)P_n(x) = 0$$

$P_n(x)$ and $P_n'(x)$ were also checked against Abromowitz and Stegun.

⟨ `legendre.c` 113 ⟩ ≡

```
#
include "legendre.h" ⟨ Definition for Pn 116 ⟩
⟨ Definition for Pn_and_Pnm1 118 ⟩
⟨ Definition for Pnd 120 ⟩
⟨ Definition for Pndd 122 ⟩
```

114. ⟨ `legendre.h` 114 ⟩ ≡

```
⟨ Prototype for Pn 115 ⟩;
⟨ Prototype for Pn_and_Pnm1 117 ⟩;
⟨ Prototype for Pnd 119 ⟩;
⟨ Prototype for Pndd 121 ⟩;
```

115. Basic Legendre functions.
Returns the Legendre polynomial $P_n(x)$ using the recurrence formulas from Michel's paper,

$$P_{n+1}(x) = \left(\frac{2n+1}{n+1}\right) xP_n(x) - \left(\frac{n}{n+1}\right) P_{n-1}(x)$$

and

$$P_0(x) = 1 \qquad \text{and} \qquad P_1(x) = x$$

Also added special cases for $x = \pm 1$. These are $P_n(1) = 1$ and $P_n(-1) = (-1)^n$ and for what it is worth

$$P_n(0) = \frac{1 \cdot 3 \cdot 5 \cdots (2n-1)}{2 \cdot 4 \cdot 6 \cdots 2n}$$

If the argument is out of range ($|x| > 1$) then instead of complaining, I just return the value at $x = \pm 1$ as appropriate. If $n < 0$ then I just return 1.0 and don't complain either.

Finally, I decided not to

⟨ Prototype for *Pn* 115 ⟩ ≡
```
double Pn(long n, double x)
```
This code is used in sections 114 and 116.

116. ⟨ Definition for *Pn* 116 ⟩ ≡
```
⟨ Prototype for Pn 115 ⟩
{
  double pk, pkp1, pkm1;
  long k;
  if (n ≤ 0) return 1.0;
  if (n ≡ 1) return x;
  if (x ≥ 1.0) return 1.0;
  if (x ≤ −1.0) return (n % 2) ? −1.0 : 1.0;
  pk = x;
  pkm1 = 1.0;
  for (k = 1; k < n; k++) {
    pkp1 = ((2 ∗ k + 1) ∗ x ∗ pk − k ∗ pkm1)/(k + 1);
    pkm1 = pk;
    pk = pkp1;
  }
  return pk;
}
```
This code is used in section 113.

117. *Pn_and_Pnm1* returns $P_n(x)$ and $P_{n-1}(x)$

⟨ Prototype for *Pn_and_Pnm1* 117 ⟩ ≡
```
void Pn_and_Pnm1(long n, double x, double ∗Pnm1, double ∗Pn)
```
This code is used in sections 114 and 118.

118. ⟨ Definition for *Pn_and_Pnm1* 118 ⟩ ≡

```
⟨ Prototype for Pn_and_Pnm1 117 ⟩
{
  long k;
  double Pk, Pkp1;
  double Pkm1 = 1.0;
  *Pnm1 = 1.0;
  *Pn = 1.0;
  if (x ≥ 1.0) x = 1.0;
  if (x ≤ −1.0) x = −1.0;
  Pk = x;
  for (k = 1; k < n; k++) {
    Pkp1 = ((2 * k + 1) * x * Pk − k * Pkm1)/(k + 1);
    Pkm1 = Pk;
    Pk = Pkp1;
  }
  *Pnm1 = Pkm1;
  *Pn = Pk;
}
```

This code is used in section 113.

119. First derivative.

Returns the first derivative of the Legendre polynomial $P'_n(x)$ using the recurrence formulas from Michel's paper,

$$P'_{n+1}(x) = \left(\frac{2n+1}{n}\right) xP'_n(x) - \left(\frac{n+1}{n}\right) P'_{n-1}(x)$$

and

$$P'_0(x) = 0 \qquad \text{and} \qquad P'_1(x) = 1$$

⟨ Prototype for *Pnd* 119 ⟩ ≡

```
double Pnd(long n, double x)
```

This code is used in sections 114 and 120.

120. ⟨ Definition for *Pnd* 120 ⟩ ≡

```
⟨ Prototype for Pnd 119 ⟩
{
  double p, pminus, pplus;
  long i;
  if (n ≤ 0) return 0;
  if (n ≡ 1) return 1;
  if (x > 1.0) x = 1.0;
  if (x < −1.0) x = −1.0;
  pminus = 0;
  p = 1;
  for (i = 1; i < n; i++) {
    pplus = ((2 ∗ i + 1) ∗ x ∗ p − (i + 1) ∗ pminus)/i;
    pminus = p;
    p = pplus;
  }
  return p;
}
```

This code is used in section 113.

121. Second derivative.
Returns the second derivative of the Legendre polynomial $P_n''(x)$ using the recurrence formulas

$$P_{n+1}''(x) = \left(\frac{2n+1}{n-1}\right) xP_n''(x) - \left(\frac{n+2}{n-1}\right) P_{n-1}''(x)$$

and

$$P_1''(x) = 0 \qquad \text{and} \qquad P_2''(x) = 3$$

⟨ Prototype for *Pndd* 121 ⟩ ≡
```
double Pndd(long n, double x)
```
This code is used in sections 114 and 122.

122. ⟨ Definition for *Pndd* 122 ⟩ ≡
```
⟨ Prototype for Pndd 121 ⟩
{
  double p, pminus, pplus;
  long m;
  if (n ≤ 1) return 0;
  if (n ≡ 2) return 3;
  if (x > 1.0) x = 1.0;
  if (x < −1.0) x = −1.0;
  pminus = 0;
  p = 3;
  for (m = 2; m < n; m++) {
    pplus = ((2 ∗ m + 1) ∗ x ∗ p − (m + 2) ∗ pminus)/(m − 1);
    pminus = p;
    p = pplus;
  }
  return p;
}
```
This code is used in section 113.

123. Lobatto Quadrature. These routines are useful in when both ends of the interval must be calculated for some other reason. Lobatto quadrature also works well when both ends of the interval are equal to zero.

This global variable is needed because the degree of the Legendre Polynomial must be known. The routine *Lobatto* stores the correct value in this. I assume the Numerical Recipes version of arrays in this program.

The *Lobatto* routine was tested for several values of n and compared with the values from the paper by Michel.

```
#define NSLICES 1000
#define EPS 1 · 10^-16
```

⟨ Lobatto.c 123 ⟩ ≡

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include "array.h"
#include "legendre.h"
#include "lobatto.h"
#include "saferoot.h"
#include "bracketroot.h"
  ⟨ Preprocessor definitions ⟩
  static long Lobatto_n_minus_1;
  ⟨ Definition for Lobatto_error 126 ⟩
  ⟨ Definition for Lobatto_fn1 129 ⟩
  ⟨ Definition for Lobatto_fn2 131 ⟩
  ⟨ Definition for Lobatto 137 ⟩
```

124. ⟨ lobatto.h 124 ⟩ ≡

```
  ⟨ Prototype for Lobatto 136 ⟩;
```

125. A simple error routine.

⟨ Prototype for *Lobatto_error* 125 ⟩ ≡

```
  static void Lobatto_error(char *s)
```

This code is used in section 126.

126. ⟨ Definition for *Lobatto_error* 126 ⟩ ≡

```
  ⟨ Prototype for Lobatto_error 125 ⟩
  {
    printf("%s\n", s);
    exit(1);
  }
```

This code is used in section 123.

127. Lobatto functions.
These functions rely on the local variable *Lobatto_n_minus_1*.

128. This funtion is used to bracket all the roots. It just returns $P_n'(x)$.

```
⟨ Prototype for Lobatto_fn1 128 ⟩ ≡
  static double Lobatto_fn1(double x)
```

This code is used in section 129.

129.

```
⟨ Definition for Lobatto_fn1 129 ⟩ ≡
  ⟨ Prototype for Lobatto_fn1 128 ⟩
  {
    return Pnd(Lobatto_n_minus_1, x);
  }
```

This code is used in section 123.

130. This funtion is used to find each root. Since Newton's method is used both the function and the first derivative are needed. This routine returns both $f = P_n'(x)$ and $df = P_n''(x)$.

```
⟨ Prototype for Lobatto_fn2 130 ⟩ ≡
  static void Lobatto_fn2(double x, double *f, double *df)
```

This code is used in section 131.

131.

```
⟨ Definition for Lobatto_fn2 131 ⟩ ≡
  ⟨ Prototype for Lobatto_fn2 130 ⟩
  {
    *f = Pnd(Lobatto_n_minus_1, x);
    *df = Pndd(Lobatto_n_minus_1, x);
  }
```

This code is used in section 123.

132. Lobatto Tables.
Here is a selection of commonly used number of quadrature points.

133. ⟨ Values for $n \equiv 4$ 133 ⟩ ≡
```
x[2] = 0.4472135954999579;
w[2] = 0.8333333333333333;
break;
```
This code is used in section 137.

134. ⟨ Values for $n \equiv 8$ 134 ⟩ ≡
```
x[6] = 0.8717401485096066;
x[5] = 0.5917001814331423;
x[4] = 0.2092992179024789;
w[6] = 0.2107042271435061;
w[5] = 0.3411226924835043;
w[4] = 0.4124587946587038;
break;
```
This code is used in section 137.

135. ⟨ Values for $n \equiv 16$ 135 ⟩ ≡
```
x[14] = 0.9695680462702180;
x[13] = 0.8992005330934720;
x[12] = 0.7920082918618151;
x[11] = 0.6523887028824931;
x[10] = 0.4860594218871376;
x[9] = 0.2998304689007632;
x[8] = 0.1013262735219495;
w[14] = 0.0508503610059200;
w[13] = 0.0893936973259308;
w[12] = 0.1242553821325141;
w[11] = 0.1540269808071643;
w[10] = 0.1774919133917041;
w[9] = 0.1936900238252036;
w[8] = 0.2019583081782299;
break;
```
This code is used in section 137.

136. Lobatto. *Lobatto* calculates the n quadrature points x_i and weights w_i over the interval (a, b). The basis fo this is Lobatto quadrature

$$\int_a^b f(x)dx = w_0 f(a) + \sum_{k=1}^{n-2} w_k f(x_k) + w_{n-1} f(b)$$

where the quadrature points x_k $(k = 1, 2, \cdots, n-2)$ are the zeros of

$$P'_{n-1}(x_k) = 0$$

and the weights are given by

$$w_k = \frac{2}{n(n-1)[P_{n-1}(x_k)]^2}$$

Finally

$$w_0 = w_{n-1} = \frac{2}{n(n-1)}$$

⟨ Prototype for *Lobatto* 136 ⟩ ≡
void *Lobatto*(**double** a, **double** b, **double** $*x$, **double** $*w$, **long** n)
This code is used in sections 124 and 137.

137. ⟨ Definition for *Lobatto* 137 ⟩ ≡
⟨ Prototype for *Lobatto* 136 ⟩
{
long *nby2*, *n_odd*, *i*;
double *xm*, *xl*, *pnval*;
if ($n < 3$) *Lobatto_error*("Number␣of␣Lobatto␣quadrature␣points␣less␣than␣3");
if ($x \equiv \Lambda$) *Lobatto_error*("NULL␣value␣passed␣for␣x␣array␣to␣Lobatto");
if ($w \equiv \Lambda$) *Lobatto_error*("NULL␣value␣passed␣for␣w␣array␣to␣Lobatto");
$x[n-1] = 1.0$;
$w[n-1] = 2.0/n/(n-1)$;
$nby2 = n/2 - 1$;
$n\_odd = n \,\%\, 2$;
switch (n) {
case 4: ⟨ Values for $n \equiv 4$ 133 ⟩
case 8: ⟨ Values for $n \equiv 8$ 134 ⟩
case 16: ⟨ Values for $n \equiv 16$ 135 ⟩
default: ⟨ Values for arbitrary n 141 ⟩
}
if (*n_odd*) ⟨ Do middle value 138 ⟩
⟨ Do negative values 139 ⟩
⟨ Scale values 140 ⟩
}
This code is used in section 123.

138. If the number of quadrature points in odd, then the center value x_i will always be zero. Furthermore $w_{n-i-1} = w_k$.

```
⟨ Do middle value 138 ⟩ ≡
  {
    i = nby2 + 1;
    x[i] = 0.0;
    pnval = Pn(n − 1, 0);
    w[i] = 2/(n ∗ (n − 1) ∗ pnval ∗ pnval);
  }
```

This code is used in section 137.

139. The quadrature points are symmetric with $x_{n-i-1} = -x_i$ and $w_{n-i-1} = w_i$. Just copy the top half of the array into the bottom half.

```
⟨ Do negative values 139 ⟩ ≡
  for (i = 0; i ≤ nby2; i++) {
    w[i] = w[n − i − 1];
    x[i] = −x[n − i − 1];
  }
```

This code is used in section 137.

140. The code to scale values is easy. Lobatto quadrature is defined over the range -1 to 1. To modify to the range a to b, I just linearly scale the width of each interval and weight as appropriate.

$$x_i = \frac{a+b}{2} + \frac{a-b}{2} x_i$$

and

$$w_i = \frac{b-a}{2} w_i$$

```
⟨ Scale values 140 ⟩ ≡
  if ((a ≠ −1.0) | (b ≠ 1.0)) {
    xm = (b + a)/2.0;
    xl = (b − a)/2.0;
    for (i = 0; i < n; i++) {
      x[i] = xm − xl ∗ x[i];
      w[i] = xl ∗ w[i];
    }
  }
```

This code is used in section 137.

141. Here is the method for finding Lobatto quadrature points for non-tabulated values. There will be $n/2$ roots located between 0 and 1.

The only strange part is that the local variable *Lobatto_n_minus_1* is set here so that it will be set correctly when *Lobatto_fn1* and *Lobatto_fn2* get called.

```
⟨ Values for arbitrary n 141 ⟩ ≡
  {
    long nb, ndiv;
    double z, *xb1, *xb2;
    Lobatto_n_minus_1 = n − 1;
    xb1 = new_darray(NSLICES);
    xb2 = new_darray(NSLICES);
    ⟨ Bracket roots 142 ⟩
    ⟨ Find roots and weights 143 ⟩
    free_darray(xb1);
    free_darray(xb2);
    break;
  }
```

This code is used in section 137.

142. Bracket $n/2$ roots, double *ndiv* if not enough roots are found. I make every effort to find all the damn roots.

```
⟨ Bracket roots 142 ⟩ ≡
  ndiv = nby2;
  do {
    ndiv *= 2;
    if (ndiv ≥ NSLICES) ndiv = NSLICES − 1;
    nb = nby2;
    bracketroot(Lobatto_fn1, 0, 1.0, ndiv, xb1, xb2, &nb);
  } while (nb < nby2 ∧ ndiv < NSLICES − 1);
  if (nb < nby2) Lobatto_error("Cannot␣find␣enough␣roots␣for␣Lobatto␣quadrature");
```

This code is used in section 141.

143. Find the roots with an accuracy `EPS` and store them in the array x. Put them in backwards so that $x[n-1] = -1$ is in the correct spot.

```
⟨ Find roots and weights 143 ⟩ ≡
  for (i = 0; i < nby2; i++) {
    z = saferoot(Lobatto_fn2, xb1[i], xb2[i], EPS);
    x[n − nby2 + i − 1] = z;
    pnval = Pn(n − 1, z);
    w[n − nby2 + i − 1] = w[n − 1]/(pnval ∗ pnval);
  }
```

This code is used in section 141.

144. Testing Lobatto.

```
⟨ lobatto-main.c 144 ⟩ ≡
#include <stddef.h>
#include <stdio.h>
#include "array.h"
#include "lobatto.h"
  void main( )
  {
    double *x, *w, z;
    double sum;
    long i, n;
    printf("testing␣n=10␣---␣sum␣should␣be␣2\n");
    sum = 0;
    n = 10;
    x = new_darray(n);
    w = new_darray(n);
    Lobatto(−1, 1, x, w, n);
    printf("The␣x_i␣are\n");
    print_darray(x, n, 0, n − 1);
    printf("The␣w_i␣are\n");
    print_darray(w, n, 0, n − 1);
    for (i = 0; i < n; i++) sum = sum + w[i];
    printf("sum␣␣␣%20.15f\n", sum);
    free_darray(x);
    free_darray(w);
    printf("\n");
    printf("testing␣n=9␣---␣sum␣should␣be␣2\n");
    sum = 0;
    n = 9;
    x = new_darray(n);
    w = new_darray(n);
    Lobatto(−1, 1, x, w, n);
    printf("The␣x_i␣are\n");
    print_darray(x, n, 0, n − 1);
    printf("The␣w_i␣are\n");
    print_darray(w, n, 0, n − 1);
    for (i = 0; i < n; i++) sum = sum + w[i];
    printf("sum␣␣␣%20.15f\n", sum);
    free_darray(x);
    free_darray(w);
    printf("\n");
  }
```

145. 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

- use Wiscombe's trick to find the scattering functions
- add code to deal with near zero entries in the Lentz routine
- allow calculation of extinction efficiencies with zero angles.

146. There are seven basic functions that are defined.

```
⟨ mie.c 146 ⟩ ≡
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "array.h"
#include "complex.h"
#include "mie.h"
  ⟨ Definition for mie_error 150 ⟩
  ⟨ Definition for Lentz_Dn 155 ⟩
  ⟨ Definition for Dn_down 164 ⟩
  ⟨ Definition for Dn_up 161 ⟩
  ⟨ Definition for small_Mie 167 ⟩
  ⟨ Definition for Mie 176 ⟩
  ⟨ Definition for ez_Mie 191 ⟩
```

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

```
⟨ mie.h 147 ⟩ ≡
  ⟨ Prototype for Lentz_Dn 154 ⟩;
  ⟨ Prototype for Dn_down 163 ⟩;
  ⟨ Prototype for Dn_up 160 ⟩;
  ⟨ Prototype for small_Mie 166 ⟩;
  ⟨ Prototype for Mie 175 ⟩;
  ⟨ Prototype for ez_Mie 190 ⟩;
```

148. This is the testing program

```
⟨ mietest.c 148 ⟩ ≡
  ⟨ Definition for MieTest 193 ⟩
```

149. A simple error routine that is not used elsewhere.

```
⟨ Prototype for mie_error 149 ⟩ ≡
  static void mie_error(char *s)
```

This code is used in section 150.

150. ⟨ Definition for *mie_error* 150 ⟩ ≡

```
⟨ Prototype for mie_error 149 ⟩
{
  printf("Mie -- %s\n", s);
  exit(1);
}
```

This code is used in section 146.

151. The logarithmic derivative D_n.

152. 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

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

153. 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 $D_n(z)$.

154. ⟨ Prototype for *Lentz_Dn* 154 ⟩ ≡

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

This code is used in sections 147 and 155.

155. ⟨ Definition for *Lentz_Dn* 155 ⟩ ≡

```
⟨ Prototype for Lentz_Dn 154 ⟩
{
  struct complex alpha_j1, alpha_j2, zinv, aj;
  struct complex alpha, result, ratio, runratio;
  ⟨ Calculate first alpha and beta 156 ⟩
  do ⟨ Calculate next ratio 157 ⟩ while (fabs(cabs(ratio) − 1.0) > 1 · 10^−12);
  result = cadd(csdiv(−n, z), runratio);
  return result;
}
```

This code is used in section 146.

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

156. 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* 156 ⟩ ≡

```
zinv = csdiv(2.0, z);
alpha = csmul(n + 0.5, zinv);
aj = csmul(−n − 1.5, zinv);
alpha_j1 = cadd(aj, cinv(alpha));
alpha_j2 = aj;
ratio = cdiv(alpha_j1, alpha_j2);
runratio = cmul(alpha, ratio);
```

This code is used in section 155.

157. 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* 157 ⟩ ≡

```
{
  aj.re = zinv.re − aj.re;
  aj.im = zinv.im − aj.im;
  alpha_j1 = cadd(cinv(alpha_j1), aj);
  alpha_j2 = cadd(cinv(alpha_j2), aj);
  ratio = cdiv(alpha_j1, alpha_j2);
  zinv.re *= −1;
  zinv.im *= −1;
  runratio = cmul(ratio, runratio);
}
```

This code is used in section 155.

158. D_n **by upward recurrence.**

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

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

159. 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 $D[1]$.

160. ⟨ Prototype for *Dn_up* 160 ⟩ ≡

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

This code is used in sections 147 and 161.

161. ⟨ Definition for *Dn_up* 161 ⟩ ≡

```
⟨ Prototype for Dn_up 160 ⟩
{
  struct complex zinv, k_over_z;
  long k;
  D[0] = cinv(ctan(z));
  zinv = cinv(z);
  for (k = 1; k < nstop; k++) {
    k_over_z = csmul(k, zinv);
    D[k] = csub(cinv(csub(k_over_z, D[k − 1])), k_over_z);
  }
}
```

This code is used in section 146.

162. 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 $D[1]$.

163. ⟨ Prototype for *Dn_down* 163 ⟩ ≡

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

This code is used in sections 147 and 164.

164. ⟨ Definition for *Dn_down* 164 ⟩ ≡

```
⟨ Prototype for Dn_down 163 ⟩
{
  long k;
  struct complex zinv, k_over_z;
  D[nstop − 1] = Lentz_Dn(z, nstop);
  zinv = cinv(z);
  for (k = nstop − 1; k ≥ 1; k−−) {
    k_over_z = csmul(k, zinv);
    D[k − 1] = csub(k_over_z, cinv(cadd(D[k], k_over_z)));
  }
}
```

This code is used in section 146.

165. 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.

166. ⟨ Prototype for *small_Mie* 166 ⟩ ≡

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

This code is used in sections 147 and 167.

167. ⟨ Definition for *small_Mie* 167 ⟩ ≡

```
⟨ Prototype for small_Mie 166 ⟩
{
  struct complex ahat1, ahat2, bhat1;
  struct complex z0, m2, m4;
  double x2, x3, x4;
  if ((s1 ≡ Λ) ∨ (s2 ≡ Λ)) nangles = 0;
  m2 = csqr(m);
  m4 = csqr(m2);
  x2 = x * x;
  x3 = x2 * x;
  x4 = x2 * x2;
  z0.re = −m2.im;
  z0.im = m2.re − 1;
  ⟨ Calculate â1 168 ⟩
  ⟨ Calculate b̂1 169 ⟩
  ⟨ Calculate â2 170 ⟩
  ⟨ Calculate small Mie efficiencies and asymmetry 172 ⟩
  ⟨ Calculate small Mie scattering amplitudes 173 ⟩
}
```

This code is used in section 146.

168. 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

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

⟨ Calculate $\hat{a}_1$ 168 ⟩ ≡

```
{
  struct complex z1, z2, z3, z4, D;
  if (m.re ≡ 0) {
    z3 = cset(0.0, 2.0/3.0 * (1.0 - 0.2 * x2));
    D = cset(1.0 - 0.5 * x2, 2.0/3.0 * x3);
  }
  else {
    z1 = csmul(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 = cmul(z1, z2);
    z4 = csmul(x3 * (1.0 - 0.1 * x2), z1);
    D.re = 2.0 + m2.re + (1 - 0.7 * m2.re) * x2 + (8 * m4.re - 385 * m2.re + 350.0)/1400 * x4 + z4.re;
    D.im = (-0.7 * m2.im) * x2 + (8 * m4.im - 385 * m2.im)/1400 * x4 + z4.im;
  }
  ahat1 = cdiv(z3, D);
}
```

This code is used in section 167.

169. 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 $\hat{b}_1$ 169 ⟩ ≡

```
{
  struct complex z2, z6, z7;
  if (m.re ≡ 0) {
    bhat1 = cdiv(cset(0.0, -(1.0 - 0.1 * x2)/3.0), cset(1 + 0.5 * x2, -x3/3));
  }
  else {
    z2 = csmul(x2/45, z0);
    z6.re = 1.0 + (2.0 * m2.re - 5.0) * x2/70.0;
    z6.im = m2.im * x2/35;
    z7.re = 1.0 - (2.0 * m2.re - 5.0) * x2/30.0;
    z7.im = -m2.im * x2/15;
    bhat1 = cmul(z2, cdiv(z6, z7));
  }
}
```

This code is used in section 167.

170. 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 â2 170 ⟩ ≡
  {
    struct complex z3, z8;
    if (m.re ≡ 0) {
      ahat2 = cset(0, x2/30.);
    }
    else {
      z3 = csmul((1.0 − x2/14) ∗ 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 = cdiv(z3, z8);
    }
  }
```

This code is used in section 167.

171. The scattering and extinction efficiencies are given by

$$\begin{aligned} Q_{\text{ext}} &= 6x \,\text{Re}\left[\hat{a}_1 + \hat{b}_1 + \frac{5}{3}\hat{a}_2\right] \\ Q_{\text{sca}} &= 6xT \\ g &= \frac{1}{T}\,\text{Re}\left[\hat{a}_1(\hat{a}_2 + \hat{b}_1)^*\right] \\ T &= |\hat{a}_1|^2 + |\hat{b}_1|^2 + \frac{5}{3}|\hat{a}_2|^2 \end{aligned}$$

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

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

where σ_{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.

172. ⟨ Calculate small Mie efficiencies and asymmetry 172 ⟩ ≡

```
{
  struct complex ss1;
  double T;
  T = cnorm(ahat1) + cnorm(bhat1) + (5/3) * cnorm(ahat2);
  *qsca = 6 * x4 * T;
  *qext = 6 * x * (ahat1.re + bhat1.re + (5/3) * 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 - (5/3) * ahat2.re);
  ss1.re = 1.5 * x2 * (ahat1.im - bhat1.im - (5/3) * ahat2.im);
  *qback = cnorm(ss1);
}
```

This code is used in section 167.

173. 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 173 ⟩ ≡

```
{
  double muj, angle;
  long j;
  x3 *= 1.5;
  ahat1.re *= x3;
  ahat1.im *= x3;
  bhat1.re *= x3;
  bhat1.im *= x3;
  ahat2.re *= x3 * 5/3;
  ahat2.im *= x3 * 5/3;
  for (j = 0; j < nangles; j++) {
    muj = mu[j];
    angle = 2 * muj * muj - 1;
    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 167.

174. 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.

175. ⟨ Prototype for *Mie* 175 ⟩ ≡

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

This code is used in sections 147 and 176.

176. ⟨ Definition for *Mie* 176 ⟩ ≡

```
⟨ Prototype for Mie 175 ⟩
{
  ⟨ Declare variables for Mie 177 ⟩
  ⟨ Catch bogus input values 178 ⟩
  ⟨ Deal with small spheres 179 ⟩
  ⟨ Mie allocate and initialize angle arrays 180 ⟩
  ⟨ Calculate the logarithmic derivatives 181 ⟩
  ⟨ Prepare to sum over all nstop terms 182 ⟩
  for (n = 1; n ≤ nstop; n++) {
    ⟨ Establish a_n and b_n 183 ⟩
    ⟨ Calculate phase function for each angle 184 ⟩
    ⟨ Increment cross sections 185 ⟩
    ⟨ Prepare for the next iteration 186 ⟩
  }
  ⟨ Calculate Efficiencies 187 ⟩
  ⟨ Free allocated memory 188 ⟩
}
```

This code is used in section 146.

177.

⟨ Declare variables for *Mie* 177 ⟩ ≡

```
struct complex *D;
struct complex z1, an, bn, bnm1, anm1, qbcalc;
double *pi0, *pi1, *tau;
struct complex xi, xi0, xi1;
double psi, psi0, psi1;
double alpha, beta, factor;
long n, k, nstop, sign;
```

This code is used in section 176.

178.

⟨ Catch bogus input values 178 ⟩ ≡

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

This code is used in section 176.

179.

⟨ Deal with small spheres 179 ⟩ ≡

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

This code is used in section 176.

180. ⟨ Mie allocate and initialize angle arrays 180 ⟩ ≡

```
if (nangles > 0) {
  set_carray(s1, nangles, cset(0.0, 0.0));
  set_carray(s2, nangles, cset(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 176.

181. Calculate number of terms to be summed in series. Allocate the space. Initialize 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

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

the upward recurrence would be stable.

⟨ Calculate the logarithmic derivatives 181 ⟩ ≡

```
{
  struct complex z;
  z = csmul(x, m);
  nstop = floor(x + 4.05 * pow(x, 0.33333) + 2.0);
  D = new_carray(nstop + 1);
  if (D ≡ Λ) mie_error("Cannot␣allocate␣log␣array");
  if (fabs(m.im * x) < ((13.78 * m.re - 10.8) * m.re + 3.9)) Dn_up(z, nstop, D);
  else Dn_down(z, nstop, D);
}
```

This code is used in section 176.

182. 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 182 ⟩ ≡

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

This code is used in section 176.

183. 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 183 ⟩ ≡

```
if (m.re ≡ 0.0) {
  an = csdiv(n * psi1/x - psi0, csub(csmul(n/x, xi1), xi0));
  bn = csdiv(psi1, xi1);
}
else if (m.im ≡ 0.0) {
  z1.re = D[n].re/m.re + n/x;
  an = csdiv(z1.re * psi1 - psi0, csub(csmul(z1.re, xi1), xi0));
  z1.re = D[n].re * m.re + n/x;
  bn = csdiv(z1.re * psi1 - psi0, csub(csmul(z1.re, xi1), xi0));
}
else {
  z1 = cdiv(D[n], m);
  z1.re += n/x;
  an = cdiv(cset(z1.re * psi1 - psi0, z1.im * psi1), csub(cmul(z1, xi1), xi0));
  z1 = cmul(D[n], m);
  z1.re += n/x;
  bn = cdiv(cset(z1.re * psi1 - psi0, z1.im * psi1), csub(cmul(z1, xi1), xi0));
}
```

This code is used in section 176.

184. 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 184 ⟩ ≡

```
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 176.

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

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

For computation purposes, this must be rewritten as

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

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 185 ⟩ ≡

```
factor = 2.0 * n + 1.0;
*g += (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 * (cnorm(an) + cnorm(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 176.

186. 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 186 ⟩ ≡
```
factor = (2.0 * n + 1.0)/x;
xi = csub(csmul(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 176.

187. ⟨ Calculate Efficiencies 187 ⟩ ≡
```
*qsca *= 2/(x * x);
*qext *= 2/(x * x);
*g *= 4/(*qsca)/(x * x);
*qback = cnorm(qbcalc)/(x * x);
```
This code is used in section 176.

188. ⟨ Free allocated memory 188 ⟩ ≡
```
if (nangles > 0) {
  free_carray(D);
  free_darray(pi0);
  free_darray(pi1);
  free_darray(tau);
}
```
This code is used in section 176.

189. 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.

190. ⟨ Prototype for *ez_Mie* 190 ⟩ ≡

```
void ez_Mie(double x, double n, double *qsca, double *g)
```

This code is used in sections 147 and 191.

191. ⟨ Definition for *ez_Mie* 191 ⟩ ≡

```
⟨ Prototype for ez_Mie 190 ⟩
{
  long nangles = 0;
  double *mu = Λ;
  struct complex *s1 = Λ;
  struct complex *s2 = Λ;
  struct 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 146.

192. Mie Testing.

193. Here is the obligatory program to test this unit.

⟨ Definition for *MieTest* 193 ⟩ ≡
```
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "array.h"
#include "complex.h"
#include "lobatto.h"
#include "mie.h"
  void main( )
  {
    ⟨ Test Logarithmic derivative 194 ⟩
    ⟨ Test Small Mie 195 ⟩
    ⟨ First Mie Test 196 ⟩
    ⟨ Second Mie Test 199 ⟩
    ⟨ Third Mie Test 200 ⟩
    ⟨ Fourth Mie Test 201 ⟩
    ⟨ Fifth Mie Test 202 ⟩
    ⟨ Sixth Mie Test 208 ⟩
  }
```
This code is used in section 148.

194. First, I make sure that the logarithmic derivatives are calculated correctly.

⟨ Test Logarithmic derivative 194 ⟩ ≡
```
  {
    struct complex y, z, *D;
    long nstop;
    printf("Testing the logarithmic derivative functions\n");
    printf("The result should for D_9(1.0) = 9.95228198\n");
    z = cset(1, 0.0);
    y = Lentz_Dn(z, 9);
    printf("The actual value                = %11.8f +i%12.8f\n\n", y.re, y.im);
    z = cset(62 * 1.28, -62 * 1.37);
    nstop = 50;
    D = new_carray(nstop);
    printf("For n = %ld \n", nstop);
    printf("For j = %ld \n", 10 L);
    printf("For z = %10.5f +i %10.5f\n", z.re, z.im);
    printf("Mathematica                  gives %10.6f +i %10.6f\n", 0.004087, 1.0002620);
    y = Lentz_Dn(z, 10);
    printf("Dn[10] continued fraction    gives %10.6f +i %10.6f\n", y.re, y.im);
    Dn_up(z, nstop, D);
    printf("Dn[10] upwards recurrence    gives %10.6f +i %10.6f\n", D[10].re, D[10].im);
    Dn_down(z, nstop, D);
    printf("Dn[10] downwards recurrence gives %10.6f +i %10.6f\n", D[10].re, D[10].im);
    free_carray(D);
  }
```
This code is used in section 193.

195. ⟨ Test Small Mie 195 ⟩ ≡
{ }
This code is used in section 193.

196. For the first Mie test, I chose the sample output values from the book by Bohren and Huffman. This is the only test that includes $Q_{\rm back}$. The angle stuff is not used.

```
⟨ First Mie Test 196 ⟩ ≡
  {
    double pi = 3.14159265358979;
    long nangles, i;
    struct complex *s1, *s2, refractive_index;
    double *weights, *mu;
    double x, rho, qext, qsca, qback, g;
    printf("\nFirst Mie Test -- Bohren and Huffman pg 482\n");
    ⟨ Initialize Mie Test variables 197 ⟩
    ⟨ Print header 198 ⟩
    Mie(x, refractive_index, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
    printf("%7.3f %10.6f %10.6f %10.6f \n", 5.213, 3.10543, 3.10543, 2.92534);
    printf("%7.3f %10.6f %10.6f %10.6f %10.6f \n", x, qsca, qext, qback, g);
    free_darray(mu);
    free_darray(weights);
    free_carray(s1);
    free_carray(s2);
  }
```
This code is used in section 193.

197. Allocate the memory needed for the quadrature angles and weights, then use Lobatto quadrature to fill them appropriately.

```
⟨ Initialize Mie Test variables 197 ⟩ ≡
  x = 5.213;
  refractive_index = cset(1.55, 0);
  rho = 2 * x * (refractive_index.re − 1);
  nangles = 10;
  mu = new_darray(nangles);
  weights = new_darray(nangles);
  s1 = new_carray(nangles);
  s2 = new_carray(nangles);
  Lobatto(0, pi, mu, weights, nangles);
  for (i = 0; i < nangles; ++i) {
    weights[i] *= sin(mu[i])/2;
    mu[i] = cos(mu[i]);
  }
```
This code is used in section 196.

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

⟨ Print header 198 ⟩ ≡

```
printf("*␣Mie␣Scattering␣\n");
printf("*␣The␣real␣index␣of␣refraction␣of␣the␣sphere␣is␣%7.4g\n", refractive_index.re);
printf("*␣The␣imag␣index␣of␣refraction␣of␣the␣sphere␣is␣%7.4g\n", refractive_index.im);
printf("*␣The␣number␣of␣quadrature␣angles␣is␣%ld\n", nangles);
printf("*\n");
printf("*␣␣␣x␣␣␣␣␣␣␣␣Qsca␣␣␣␣␣␣␣Qext␣␣␣␣␣␣␣Qback␣␣␣␣g\n");
```

See also section 215.

This code is used in sections 196 and 209.

199. Another test that includes absorbing spheres is from the paper by Dave. His table 2 tabulates the absorption efficiency.

```
⟨ Second Mie Test 199 ⟩ ≡
  {
    double pi = 3.14159265358979;
    long nangles = 0;
    struct complex *s1 = Λ;
    struct complex *s2 = Λ;
    double *mu = Λ;
    struct complex m;
    double x = 50.0 * pi;
    double qext, qsca, qback, g;
    printf("\nSecond␣Mie␣Test␣--␣Dave␣Table␣2\n");
    printf("␣␣␣␣␣␣␣␣␣␣n␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Qa␣␣␣␣␣␣␣␣␣␣␣Dave\n");
    m = cset(1.342, 0);
    Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
    printf("%10.5g%-+7.4fi␣␣␣␣%10.5f␣␣␣␣%10.5f\n", m.re, m.im, qext - qsca, 0.0);
    m = cset(1.342, -0.0001);
    Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
    printf("%10.5g%-+7.4fi␣␣␣␣%10.5f␣␣␣␣%10.5f\n", m.re, m.im, qext - qsca, 0.0535);
    m = cset(1.342, -0.01);
    Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
    printf("%10.5g%-+7.4fi␣␣␣␣%10.5f␣␣␣␣%10.5f\n", m.re, m.im, qext - qsca, 0.9649);
    m = cset(1.342, -0.2);
    Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
    printf("%10.5g%-+7.4fi␣␣␣␣%10.5f␣␣␣␣%10.5f\n", m.re, m.im, qext - qsca, 0.9542);
    m = cset(1.342, -0.4);
    Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
    printf("%10.5g%-+7.4fi␣␣␣␣%10.5f␣␣␣␣%10.5f\n", m.re, m.im, qext - qsca, 0.9221);
    m = cset(1.342, -0.6);
    Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
    printf("%10.5g%-+7.4fi␣␣␣␣%10.5f␣␣␣␣%10.5f\n", m.re, m.im, qext - qsca, 0.8808);
    m = cset(1.342, -0.8);
    Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
    printf("%10.5g%-+7.4fi␣␣␣␣%10.5f␣␣␣␣%10.5f\n", m.re, m.im, qext - qsca, 0.8369);
    m = cset(1.342, -1.0);
    Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
    printf("%10.5g%-+7.4fi␣␣␣␣%10.5f␣␣␣␣%10.5f\n", m.re, m.im, qext - qsca, 0.7910);
    printf("\n");
  }
```

This code is used in section 193.

200. The third test uses tabulated values from van de Hulst to check the anisotropy calculation.

⟨ Third Mie Test 200 ⟩ ≡

```
{
  double pi = 3.14159265358979;
  long nangles = 0;
  struct complex *s1 = Λ;
  struct complex *s2 = Λ;
  double *mu = Λ;
  struct complex m;
  double x = 20;
  double qext, qsca, qback, g;
  printf("\nThird␣Mie␣Test␣--␣van␣de␣Hulst␣page␣161\n");
  printf("␣x␣␣␣␣␣␣␣␣␣Qs␣␣␣␣vdH␣␣␣␣␣␣␣Qs*g␣␣␣vdH\n");
  m = cset(0, 0);
  x = 0.3;
  Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
  printf("%4.1f␣␣␣%7.3f␣%7.3f␣␣␣%7.3f␣%7.3f\n", x, qsca, 0.028, qsca * g, −0.011);
  ez_Mie(x, 0, &qsca, &g);
  printf("%4.1f␣␣␣%7.3f␣%7.3f␣␣␣%7.3f␣%7.3f\n", x, qsca, 0.028, qsca * g, −0.011);
  x = 1.0;
  Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
  printf("%4.1f␣␣␣%7.3f␣%7.3f␣␣␣%7.3f␣%7.3f\n", x, qsca, 2.036, qsca * g, −0.385);
  x = 1.5;
  Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
  printf("%4.1f␣␣␣%7.3f␣%7.3f␣␣␣%7.3f␣%7.3f\n", x, qsca, 2.155, qsca * g, 0.156);
  x = 5.0;
  Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
  printf("%4.1f␣␣␣%7.3f␣%7.3f␣␣␣%7.3f␣%7.3f\n", x, qsca, 2.116, qsca * g, 0.965);
  printf("\n");
}
```

This code is used in section 193.

201. This test includes one vale for totally reflecting spheres.

```
⟨ Fourth Mie Test 201 ⟩ ≡
  {
    double pi = 3.14159265358979;
    long nangles = 0;
    struct complex *s1 = Λ;
    struct complex *s2 = Λ;
    double *mu = Λ;
    struct complex m;
    double x = 20;
    double qext, qsca, qback, g;
    printf("\nFourth␣Mie␣Test␣--␣van␣de␣Hulst␣page␣277\n");
    printf("␣x␣␣␣␣␣␣␣␣␣Qs␣␣␣␣vdH␣␣␣␣␣␣␣Qs*g␣␣␣vdH\n");
    m = cset(3.41, -1.94);
    x = 1.3;
    Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
    printf("%4.1f␣␣␣%7.3f␣%7.3f␣␣␣%7.2f␣%7.2f\n", x, qsca, 1.669, qsca * g, 0.30);
    m = cset(7.20, -2.65);
    x = 1.3;
    Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
    printf("%4.1f␣␣␣%7.3f␣%7.3f␣␣␣%7.2f␣%7.2f\n", x, qsca, 1.860, qsca * g, 0.31);
    m = cset(0, 0);
    x = 1.3;
    Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
    printf("%4.1f␣␣␣%7.3f␣%7.3f␣␣␣%7.2f␣%7.2f\n", x, qsca, 2.266, qsca * g, -0.05);
    printf("\n");
  }
```

This code is used in section 193.

202. Well I found a complete set of results posted by Wiscombe. This allows complete code coverage.

```
⟨ Fifth Mie Test 202 ⟩ ≡
  {
    long nangles = 0;
    struct complex *s1 = Λ;
    struct complex *s2 = Λ;
    double *mu = Λ;
    struct complex m;
    double x;
    double qext, qsca, qback, g;
    printf("\nFifth␣Mie␣Test␣--␣Wiscombe\n");
    ⟨ Wiscombe Non-absorbing spheres 203 ⟩
    ⟨ Wiscombe Absorbing water spheres 204 ⟩
    ⟨ Wiscombe Absorbing spheres 205 ⟩
    ⟨ Wiscombe Yet More Absorbing spheres 206 ⟩
    ⟨ Wiscombe perfectly conducting spheres 207 ⟩
    printf("\n");
  }
```

This code is used in section 193.

203. ⟨ Wiscombe Non-absorbing spheres 203 ⟩ ≡

```
printf("\nNon-Absorbing␣Spheres␣m=(0.75+0.0i)\n");
printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Calc.␣␣␣␣␣Wiscombe␣␣␣␣␣Calc␣␣␣␣␣Wiscombe\n");
printf("␣␣␣␣x␣␣␣␣␣␣␣␣␣␣␣Qs␣␣␣␣␣␣␣␣␣␣␣Qs␣␣␣␣␣␣␣␣␣␣␣g␣␣␣␣␣␣␣␣␣␣␣g\n");
m = cset(0.75, 0.0);
x = 0.099;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 0.000007, g, 0.001448);
ez_Mie(x, 0.75, &qsca, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 0.000007, g, 0.001448);
m = cset(0.75, 0.0);
x = 0.101;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 0.000008, g, 0.001507);
m = cset(0.75, 0.0);
x = 10.0;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 2.232265, g, 0.896473);
m = cset(0.75, 0.0);
x = 1000.0;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 1.997908, g, 0.844944);
ez_Mie(x, 0.75, &qsca, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 1.997908, g, 0.844944);
```

This code is used in section 202.

204. ⟨ Wiscombe Absorbing water spheres 204 ⟩ ≡

```
printf("\nAbsorbing␣Water␣Spheres␣m=(1.33-0.00001i)\n");
printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Calc.␣␣␣␣␣Wiscombe␣␣␣␣␣Calc␣␣␣␣␣Wiscombe\n");
printf("␣␣␣␣x␣␣␣␣␣␣␣␣␣␣␣Qs␣␣␣␣␣␣␣␣␣␣␣Qs␣␣␣␣␣␣␣␣␣␣␣g␣␣␣␣␣␣␣␣␣␣␣g\n");
m = cset(1.33, -0.00001);
x = 1.0;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 0.093923, g, 0.184517);
x = 100.0;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 2.096594, g, 0.868959);
x = 10000.0;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 1.723857, g, 0.907840);
```

This code is used in section 202.

205. ⟨ Wiscombe Absorbing spheres 205 ⟩ ≡

```
printf("\nAbsorbing␣Spheres␣m=(1.5-i)\n");
printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Calc.␣␣␣␣␣Wiscombe␣␣␣␣␣Calc␣␣␣␣␣Wiscombe\n");
printf("␣␣␣x␣␣␣␣␣␣␣␣␣␣Qs␣␣␣␣␣␣␣␣␣␣Qs␣␣␣␣␣␣␣␣␣␣g␣␣␣␣␣␣␣␣␣␣g\n");
m = cset(1.5, -1.00);
x = 0.055;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 0.000011, g, 0.000491);
x = 0.056;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 0.000012, g, 0.000509);
x = 1.0;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 0.6634538, g, 0.192136);
x = 100.0;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 1.283697, g, 0.850252);
x = 10000.0;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 1.236574, g, 0.846310);
```

This code is used in section 202.

206. ⟨ Wiscombe Yet More Absorbing spheres 206 ⟩ ≡

```
printf("\n␣Yet␣More␣Absorbing␣Spheres␣m=(10-10i)\n");
printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Calc.␣␣␣␣␣Wiscombe␣␣␣␣␣Calc␣␣␣␣␣Wiscombe\n");
printf("␣␣␣x␣␣␣␣␣␣␣␣␣␣Qs␣␣␣␣␣␣␣␣␣␣Qs␣␣␣␣␣␣␣␣␣␣g␣␣␣␣␣␣␣␣␣␣g\n");
m = cset(10, -10.00);
x = 1.0;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 2.049405, g, -0.110664);
x = 100.0;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 1.836785, g, 0.556215);
x = 10000.0;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 1.795393, g, 0.548194);
```

This code is used in section 202.

207. ⟨ Wiscombe perfectly conducting spheres 207 ⟩ ≡

```
printf("\nPerfectly␣Conducting␣Spheres\n");
printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Calc.␣␣␣␣␣Wiscombe␣␣␣␣␣Calc␣␣␣␣␣Wiscombe\n");
printf("␣␣␣␣x␣␣␣␣␣␣␣␣␣␣␣Qs␣␣␣␣␣␣␣␣␣␣␣Qs␣␣␣␣␣␣␣␣␣␣␣g␣␣␣␣␣␣␣␣␣␣␣g\n");
m = cset(0.0, 0.0);
x = 0.099;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 0.000321, g, −0.397357);
x = 0.101;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 0.000348, g, −0.397262);
x = 100;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 2.008102, g, 0.500926);
ez_Mie(x, 0.0, &qsca, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 2.008102, g, 0.500926);
x = 10000;
Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
printf("%9.3f␣␣␣%11.7f␣%11.7f␣␣␣%11.7f␣%11.7f\n", x, qsca, 2.000289, g, 0.500070);
```

This code is used in section 202.

208.

⟨ Sixth Mie Test 208 ⟩ ≡

```
{
  double pi = 3.14159265358979;
  long nangles = 7;
  struct complex *s1 = Λ;
  struct complex *s2 = Λ;
  double *mu = Λ;
  struct complex m;
  double x;
  double qext, qsca, qback, g;
  char *form = "%7.4f␣%8.5f%+-8.5fi␣␣␣␣%8.5f%+-8.5fi␣␣Calc\n";
  char *form2 = "%7.4f␣%8.5f%+-8.5fi␣␣␣␣%8.5f%+-8.5fi␣␣␣Wiscombe\n\n";
  long i;
  s1 = new_carray(nangles);
  s2 = new_carray(nangles);
  mu = new_darray(nangles);
  for (i = 0; i < nangles; i++) mu[i] = cos(pi * i/6.0);
  printf("\nSixth␣Mie␣Test␣--␣Wiscombe\n");
  printf("␣␣␣angle␣␣␣␣␣␣␣S1␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣S2␣␣␣␣␣␣␣␣␣\n");
  x = 1.0;
  m = cset(1.5, -1);
  Mie(x, m, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
  printf(form, mu[0], s1[0].re, s1[0].im, s2[0].re, s2[0].im);
  printf(form2, mu[0], 5.84080 · 10^-01, 1.90515 · 10^-01, 5.84080 · 10^-01, 1.90515 · 10^-01);
  printf(form, mu[1], s1[1].re, s1[1].im, s2[1].re, s2[1].im);
  printf(form2, mu[1], 5.65702 · 10^-01, 1.87200 · 10^-01, 5.00161 · 10^-01, 1.45611 · 10^-01);
  printf(form, mu[2], s1[2].re, s1[2].im, s2[2].re, s2[2].im);
  printf(form2, mu[2], 5.17525 · 10^-01, 1.78443 · 10^-01, 2.87964 · 10^-01, 4.10540 · 10^-02);
  printf(form, mu[3], s1[3].re, s1[3].im, s2[3].re, s2[3].im);
  printf(form2, mu[3], 4.56340 · 10^-01, 1.67167 · 10^-01, 3.62285 · 10^-02, -6.18265 · 10^-02);
  printf(form, mu[4], s1[4].re, s1[4].im, s2[4].re, s2[4].im);
  printf(form2, mu[4], 4.00212 · 10^-01, 1.56643 · 10^-01, -1.74875 · 10^-01, -1.22959 · 10^-01);
  printf(form, mu[5], s1[5].re, s1[5].im, s2[5].re, s2[5].im);
  printf(form2, mu[5], 3.62157 · 10^-01, 1.49391 · 10^-01, -3.05682 · 10^-01, -1.43846 · 10^-01);
  printf(form, mu[6], s1[6].re, s1[6].im, s2[6].re, s2[6].im);
  printf(form2, mu[6], 3.48844 · 10^-01, 1.46829 · 10^-01, -3.48844 · 10^-01, -1.46829 · 10^-01);
  printf("\n");
}
```

This code is used in section 193.

209. A driver program for Mie scattering.

This implementation has not been tested. In fact it currently does not work. Careful comparison with the Pascal version is needed to find the problems.

```
⟨ mie-main.c 209 ⟩ ≡
#include <stdio.h>
#include <math.h>
#include "array.h"
#include "complex.h"
#include "Legendre.h"
#include "Lobatto.h"
#include "Mie.h"
  void main( )
  {
    ⟨ Declare Mie Variables 210 ⟩
    ⟨ Initialize Mie variables 213 ⟩
    ⟨ Allocate angle based arrays 211 ⟩
    ⟨ Initialize quadrature angles and weights 212 ⟩
    ⟨ Calculate sphere size and number 214 ⟩
    ⟨ Print header 198 ⟩
    Mie(x, refractive_index, mu, nangles, s1, s2, &qext, &qsca, &qback, &g);
    ⟨ Print summary 216 ⟩
    ⟨ Print moments 217 ⟩
    ⟨ Print phase function 218 ⟩
  }
```

210. Variables.

```
⟨ Declare Mie Variables 210 ⟩ ≡
  double pi = 3.14159265358979;
  double n_sphere, n_medium;
  double sphere_density, medium_density;
  double sphere_volume;
  double diameter, lambda;
  double concentration;
  long max_moments;
  long nangles;
  struct complex *s1;
  struct complex *s2;
  struct complex refractive_index;
  double *parallel, *perpen, *phasefn, *mu, *weights;
  double moments[11];
  double x, qext, qsca, qback, g;
  long i;
  long j;
  double number_per_cc;
  double sphere_area;
```

This code is used in section 209.

211.

```
⟨ Allocate angle based arrays 211 ⟩ ≡
  mu = new_darray(nangles);
  weights = new_darray(nangles);
  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 209.

212. Use Lobatto quadrature quadrature to initialize the angles and weights,

```
⟨ Initialize quadrature angles and weights 212 ⟩ ≡
  Lobatto(0.0, pi, mu, weights, nangles);
  for (i = 0; i < nangles; i++) {
    weights[i] *= sin(mu[i])/2;
    mu[i] = cos(mu[i]);
  }
  print_darray(weights, nangles, 0, nangles);
  print_darray(mu, nangles, 0, nangles);
```

This code is used in section 209.

213. Initialize all the global variables and allocate those necessary.

```
⟨ Initialize Mie variables 213 ⟩ ≡
  nangles = 20;
  max_moments = 2;
  lambda = 543.5;      /* in nm */
  diameter = 1070;     /* in nm */
  n_sphere = 1.59;
  n_medium = 1.34;
  sphere_density = 1.05;      /* in gm/cc */
  medium_density = 1.00;      /* in gm/cc */
  concentration = 0.01;       /* in gm spheres/ gm fluid */
  refractive_index.im = 0;
  refractive_index.re = n_sphere/n_medium;
```

This code is used in section 209.

214. Calculating the number of spheres per cubic centimeter is straightforward, if onerous. The first constraint is that

$$1 \text{ cc} = N_s V_s + V_f$$

where the subscript s refers to the sphere and f to the surrounding fluid. N_s is the number of spheres per milliliter, and V is the volume. The next equation is that the concentration c (in gm spheres/gm fluid) is

$$c = \frac{N_s w_s}{w_f} = \frac{N_s \rho_s V_s}{\rho_f V_f}$$

where w is the weight and ρ is the density. Putting these two equations together leads to

$$1 \text{ cc} = N_s V_s \frac{\rho_f c + \rho_s}{\rho_f c}$$

or

$$N_s = \frac{1 \text{ cc}}{V_s} \frac{1}{1 + \rho_s/(\rho_f c)}$$

⟨ Calculate sphere size and number 214 ⟩ ≡
  $x = (diameter * pi)/(lambda/refractive\_index.re)$;
  $sphere\_area = pi * diameter * diameter/4.0$;
  $sphere\_volume = pi * diameter * diameter * diameter/6.0/1 \cdot 10^{21}$; /* in cc */
  $number\_per\_cc = 1/sphere\_volume/(1 + sphere\_density/medium\_density/concentration)$;
This code is used in section 209.

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

⟨ Print header 198 ⟩ +≡
  printf("* Mie Scattering -- Non-absorbing spheres\n");
  printf("* Sphere concentration is %5.2g%% gm spheres/gm medium\n", *concentration*);
  printf("* Medium refractive index is %7.4g \n", *n_medium*);
  printf("* Sphere refractive index is %7.4g \n", *n_sphere*);
  printf("* The density of the spheres is %10g gm/ml\n", *sphere_density*);
  printf("* The density of the medium is %10g gm/ml\n", *medium_density*);
  printf("* The number of spheres per cc is %10g gm/ml\n", *medium_density*);
  printf("* The number of quadrature angles is %ld\n", *nangles*);
  printf("*\n");
  printf("* d lambda x N Qsca g s s(1-g)\n");
  printf("* [nm] [nm] #/ml [1/cm] [1/cm] \n");

216. ⟨ Print summary 216 ⟩ ≡
  printf("*%7.1f %7.1f ", *diameter*, *lambda*);
  printf("%7.2f %7.2e %7.4f %7.5f ", x, *number_per_cc*, *qsca*, g);
  printf("%7.3f ", *qsca* * *sphere_area* * *number_per_cc*);
  printf("%7.3f \n", $(1 - g) * qsca * sphere\_area * number\_per\_cc$);
This code is used in section 209.

217. ⟨ Print moments 217 ⟩ ≡
  for ($i = 0$; $i \le max\_moments$; $++i$) {
    $moments[i] = 0$;
    for ($j = 0$; $j < nangles$; j++) $moments[i] = moments[i] + weights[j] * Pn(i, mu[j]) * phasefn[j]$;
    printf("moment %ld is %10.5f\n", i, $moments[i]$);
  }
This code is used in section 209.

218. ⟨ Print phase function 218 ⟩ ≡

```
for (i = 0; i < nangles; ++i) {
  parallel[i] = 4 * cnorm(s2[i])/(x * x * qsca);
  perpen[i] = 4 * cnorm(s1[i])/(x * x * qsca);
  phasefn[i] = (parallel[i] + parallel[i])/2;
}
printf("*\n");
printf("*␣␣␣␣␣␣␣␣␣␣␣␣␣␣natural␣␣␣␣␣␣␣␣␣␣␣␣perpen␣␣␣␣␣parallel\n");
printf("*␣␣mu␣␣␣␣(|s1|^2+|S2|^2)/2␣␣␣␣␣|S1|^2␣␣␣␣␣|S2|^2\n");
for (i = 0; i < nangles; i++)
  printf("%7.4f␣%10.5f␣%10.5f␣%10.5f\n", mu[i], phasefn[i], perpen[i], parallel[i]);
```

This code is used in section 209.

219. 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.

Mie Scattering

(Version 1.0)