

2 Introduction to C – part II

2.1 Loops

The program, `planck.c` that you compiled and ran last time can be used to compute the Planck function, or blackbody spectrum for a given temperature by inputting different values for the wavelength while holding the temperature constant. Instead of running the program multiple times, it is easy to do this by introducing a loop into your program. Please read over the section in your textbook on Loops and Conditionals. Let us see how to modify `planck.c` to include a loop to make our life easier.

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

int main()
{
    double wave,T;
    double p = 1.19106e+27;
    double p1;

    printf('\nEnter the temperature in Kelvin > ');
    scanf('%lf',&T);

    wave = 500.0;

    while(wave <= 12000.0) {
        p1 = pow(wave,5.0)*(exp(1.43879e+08/(wave*T)) - 1.0);
        printf('The Planck function at %7.2f A and %7.2f K is %e\n',
            wave,T,p/p1);
        wave += 500.0;
    }
    getchar();
    return(0);
}
```

Here we have introduced a *while loop* to evaluate the Planck function at values of the wavelength between 500 and 12000Å. The *while loop* begins with the statement

```
while(wave <= 12000.0) {
```

and ends with the close bracket

```
}
```

five lines down. What this means is that the statements inbetween these two lines are executed in order multiple times until the condition `wave <= 12000.0` is no longer true. Notice that the *while loop* contains the statement `wave += 500.0`; which increments `wave` by 500 everytime the statement is executed. Eventually `wave` will exceed 12000.0 and then the program will exit the loop.

Another type of loop is called a *do while loop*. We could replace the *while loop* above with the following statements:

```
do {
    p1 = pow(wave,5.0)*(exp(1.43879e+08/(wave*T)) - 1.0);
    printf('The Planck function at %7.2f A and %7.2f K is %e\n',
           wave,T,p/p1);
    wave += 500.0;
} while(wave <= 12000.0)
```

This loop functions very similarly to the *while loop*, except that the conditional `wave <= 12000.0` is evaluated at the end of the loop instead of the beginning. It turns out that *do while* loops are almost never used. However, every once in a while it proves to be useful to have the conditional evaluated at the end of the loop, for instance when you want to guarantee that the loop will be executed at least once.

Another way of writing this program is to initialize `wave` as a vector and to use a *for loop*, as follows:

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

int main()
```

```

{
double T;
double p = 1.19106e+27;
double p1;
double wave[24] = { 500.0, 1000.0, 1500.0, 2000.0, 2500.0,
                    3000.0, 3500.0, 4000.0, 4500.0, 5000.0,
                    5500.0, 6000.0, 6500.0, 7000.0, 7500.0,
                    8000.0, 8500.0, 9000.0, 9500.0,10000.0,
                    10500.0,11000.0,11500.0,12000.0};

int i;

printf("\nEnter the temperature in Kelvin > ");
scanf("%lf",&T);

for(i=0;i<24;i++) {
    p1 = pow(wave[i],5.0)*(exp(1.43879e+08/(wave[i]*T)) - 1.0);
    printf("The Planck function at %7.2f A and %7.2f K is %e\n",
          wave[i],T,p/p1);
}
getchar();
return(0);
}

```

Notice how we have initialized the values for the vector `wave` in this example. The *for loop* begins with the `for(i=0;i<24;i++)` statement which requires some explanation. This `for` statement says “initialize `i` to 0 and for each passage through the loop, increment `i` by 1 (`i++`). When `i` equals or is greater than 24, exit the loop.” Notice inside the loop that the different values in the `wave` vector are accessed with `wave[i]` as `i` is different for each passage through the loop.

All three examples should give identical output. In this application, the *for loop* example is probably unnecessarily complicated, but this method of initializing a vector can be quite useful, especially if the values of the vector are not readily computed. For instance, let us suppose we wanted to evaluate the Planck function at the wavelengths: 515.0, 912.0, 3123.0, 6615.0 In such a case we would have been forced to use a *for loop* because these values cannot be computed with an increment statement such as `wave += 500`. In

addition, *for loops* are used when we want to control exactly the number of times the loop is executed.

Exercise 2.1: Write a program containing a *while loop* which will output the velocity of a ball thrown upwards with an initial velocity v_0 (which should be input to the program) moving under the influence of gravity. The program should output the velocity of the ball for times $t = 0$ to $t = 100$ seconds with an interval of 5 seconds.

Exercise 2.2: Accomplish the task for **Exercise 2.1** using a *for loop*.

2.2 Conditionals

In the previous section we were exposed to conditionals. For instance, we saw the use of a conditional in the *while* statement. Generally speaking, a conditional is a statement which, when evaluated to be true causes the program to execute a certain block of statements, and when evaluated to be false, to execute another block of statements. The simplest form of conditional is the *if* statement and its extension, the *if else* statement. Consider the following fragment of code to see how the *if* statement works:

```
if(n == 0) T = 5000;
```

This code states that if n is equal to 0, then T should be assigned the value 5000. But, what if n is not equal to 0? This is taken care of by the *if else* statement:

```
if(n == 0) T = 5000;
else T = 4000;
```

This clearly indicates that if n is not 0, then T should be assigned the value 4000. An *if else* statement can be extended with *else if* statements such as:

```
if(n == 0) T = 5000;
else if(n < 20) T = 4000;
else T = 3000;
```

If an entire block of statements needs to be executed as the result of an *if*, *else* or *else if* statement, this block can be enclosed in parentheses:

```
if(n == 0) {
    T = 5000;
    x = 62.5*T;
}
```

Table 1: Comparison Expressions

Operator	Meaning
==	is equal to
!=	is not equal to
>	is greater than
>=	is greater than or equal to
<	is less than
<=	is less than or equal to

The expression `n == 0` is called a Boolean expression. A Boolean expression evaluates to 1 if it is true and to 0 if it is false. Table 1 lists the comparison operators that can be used in Boolean expressions.

Boolean expressions can be combined using Boolean algebra. For instance, Boolean expressions can be combined using `&&` (and) and `||` (or). If two expressions are combined with `&&`, then both must be true for the entire expression to be true. If the two expressions are combined with `||` then only one of the expressions need be true. More than two expressions can be combined using `&&` and `||` to give quite complex Boolean expressions. In the following examples, let `A = 10`, `B = 1` and `C = 5`. The character “!” is used to negate a statement.

```
A == 10 || B == 5           True
!(A == 10 || B == 5)       False
A == 10 && B == 5           False
A != 3 && B == 1            True
A >= C && B >= A            False
(A != 5 && B == 1) || C == 3 True
B                           True
!B                           False
```

Exercise 2.3: Random numbers can be generated between 0.0 and 1.0 using the function `ran1` from the file `comphys.c`. The random number generator needs to be initialized, and this code fragment shows you how to do it:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "comphys.h"
```

```

int main()
{
    long idum = -1;
    time_t now;
    float x,rn;

    /* Use the time function to supply a different seed to the
       random number generator every time the program is run */

    now = time(NULL);

    idum = -1*now;

    /* Initialize random number generator */

    rn = ran1(&idum);

    /* Now you can generate any number of random numbers between 0.0
       and 1.0 with statements like the following: */

    x = ran1(&idum);

```

Write a program using this random number generator that will sort the random numbers x that you generate into 4 bins, one bin with $0 \leq x < 0.25$, the next with $0.25 \leq x < 0.50$, etc. Generate 1000 random numbers, and have the program count the number in each bin. If you call your program `ranbin.c`, you will need to compile it with the following command:

```
gcc -o ranbin ranbin.c comphys.c -lm
```

Exercise 2.4: In many applications, whether they are directly related to physics or not, one needs to sort data based on some criterion. This need arises quite often and, if not approached correctly, could consume a considerable amount of CPU time. There are many sorting applications out there, all of which rely on conditional statements to sort the data. A highly efficient and commonly used sorting routine is “quicksort.c”.

The quicksort routine uses recursion to divide a given array into smaller and smaller partitions, sorting as it goes. Quicksort and its subroutine, `partition.c`, are provided for you within the `comphys.c` library in two forms: `qsint`, which applies quicksort to integers, and `qsfloat`, which applies quicksort to floating point

values:

```
void qsint( int[], int, int);  
void qfloat( float[], int, int);
```

The qsint routine is shown below, but remember, you don't need to add this code to your programs, all you need to do is place:

```
#include "comphys.h"  
#include "comphys.c"
```

at the beginning of your program and then call qsint or qfloat as you need them. For example, the following demo sorts integers. To compile it, just run:

```
gcc -o sort_demo sort_demo.c -lm
```

```
#include <stdio.h>  
#include "comphys.h"  
#include "comphys.c"
```

```
int main()  
{  
    int a[9]={7,12,1,-2,0,15,4,11,9};  
    int a0[9];  
    int i=0;  
  
    for (i=0;i<=8;i++) a0[i]=a[i];  
  
    /* Here is where you call quicksort (for integers): */  
  
    qsint(a,0,8);  
  
    /* Notice: you pass the array to be sorted, a, the lower limit index, 0  
       and the upper limit index, 8 (in this case).  
    */  
  
    printf("\nQuicksort demo:\n");  
    printf("\nOriginal    Sorted\n");  
    for(i=0;i<9;++i) printf("%4d    %4d\n",a0[i],a[i]);  
    return(0);  
}
```

Here is the qsint code for your reference.

```

/* quicksort integers */
void qsint( int a[], int l, int r)
{
    int partint( int a[], int l, int r)
    int j;
    if( l < r )
    {
        j = partint( a, l, r);
        qsint( a, l, j-1);
        qsint( a, j+1, r);
    }
}
/* partition integers */
int partint( int a[], int l, int r) {
    int pivot, i, j, t;
    pivot = a[l];
    i = l; j = r+1;

    while( 1) {
        do ++i; while( a[i] <= pivot && i <= r );
        do --j; while( a[j] > pivot );
        if( i >= j ) break;
        t = a[i]; a[i] = a[j]; a[j] = t;
    }
    t = a[l]; a[l] = a[j]; a[j] = t;
    return j;
}

```

As an example of the use of quicksort, in earth surface physics, one often needs to determine the grain size distribution of grains in a given population. This is usually done to model the boundary layer behavior between a bed of grains and a fluid that is moving relative to that bed. The grain sizes beneath a bed, known as "granular surface roughness", will determine the drag experienced by the fluid as it passes overhead. Large grains induce turbulence, draining energy from the flow and slowing it down. Very small grains (or a smooth surface) may allow laminar flow, speeding it up.

In the following assignment, you will sort 50 grains whose size [in mm] is provided in the file grains.dat. You must sort the grains in two ways:

1. Write to screen the list of 50 grains sorted by size from smallest to largest.
2. Generate the number of grains that fall within each size bin, where the bins

correspond to sizes 1.0-1.1mm, 1.1-1.2mm, ... , 1.9-2.0mm. For grains that fall on a boundary, place them in the smaller size bin (for example, place a grain with size 1.3mm into the 1.2-1.3mm bin). You will output to the screen the number of grains per size bin.

3. Print the average grain size of the distribution.

4. Print the standard deviation of the size distribution, defined as

$$SD = \sqrt{(1/N * \sum((gs(i) - avegs)^2))}$$

where gs =grainsize, $avegs$ =average grain size, N =number of grains, and i is the grain index ranging from 1 to N .

FYI: Fluid models would then use the average grain size and the standard deviation to model the boundary layer. You don't need to use quicksort to obtain these statistics.

5. Do (2.) as stated above but you must print them out in descending order, starting with the bin with the most grains in it. For example, if the bin 1.2-1.3 has the most grains in it, say 14, followed by bin 1.8-1.9 that has, say 13, followed by bin 1.5-1.6 that has, say 8, you would print out:

bin 1.2 to 1.3 has 14 grains

bin 1.8 to 1.9 has 13 grains

bin 1.5 to 1.6 has 8 grains

.
. .
.

GRADUATE STUDENTS ONLY

6. Create a new routine `qsdouble` which applies quicksort to arguments that are of double precision. Include this in your versions of `comphys.h` and `comphys.c` and run part (1.) of this assignment with the grain sizes defined as double.

2.3 An Aside – Command-line input and Piping

Sometimes it is useful to write your program so that all the input information can be introduced on the *command line*. For instance, take the example program in § 2.1. This program prompts the user for a temperature, and then prints out the value of the Planck function for wavelengths between 500

and 12,000Å. It would be nice to get the program to accept its input from the command line, because this then makes it possible to redirect the output from the program from the screen to a file. For instance, with the command line (entered at the Linux prompt):

```
./planck 7500.0 > planck.out
```

a suitably modified planck program would read in 7500K for the temperature, and then would *pipe* the output into the file planck.out. To make this possible, modify the program in § 2.1 to read as follows:

```
#include <stdio.h>
#include <stdlib.h> /* Required for the function atof */
#include <math.h>

int main(int argc, char *argv[])
{
    double wave,T;
    double p = 1.19106e+27;
    double p1;

    if(argc != 2) {
        printf("\nEnter the temperature in Kelvin > ");
        scanf("%lf",&T);
    } else T = atof(argv[1]);

    wave = 500.0;

    while(wave <= 12000.0) {
        p1 = pow(wave,5.0)*(exp(1.43879e+08/(wave*T)) - 1.0);
        printf("%7.2f %e\n",wave,p/p1); /* Note modification here */
        wave += 500.0;
    }
    /* Take out the getchar() */
    return(0);
}
```

How does this work? The parameters to the main function `int argc`, `char *argv[]` allow the program to read input from the command line. If

the program sees no input on the command line (for instance, if the command line looks simply like `./planck`), then `argc` is set to 1. If there is one input item on the command line (for instance, `./planck 7500`, then `argc` is set to 2, etc. So, the above program reads the number of input items on the command line. If `argc == 2`, then the program reads the temperature `T` from `argv[1]` (`argv[0]` is, by default, always set to the name of the program, in this case `planck`). If, however, `argc != 2`, the program prompts the user for the temperature. Try it! Then use `gnuplot` to plot the Planck function using the output file `planck.out`, using the `gnuplot` command:

```
gnuplot> plot "planck.out" using 1:2 with lines
```