Chapter 4

Improving the FDTD Code

4.1 Introduction

The C code presented in the previous chapter adequately served its purpose—it implemented the desired FDTD functionality in a relatively straightforward way. However, as the algorithms get more involved, for instance, when implementing simulations of two- or three-dimensional problems, the readability, portability, and maintainability will be improved if our implementation is slightly more sophisticated. In this chapter we will implement the algorithms of the previous chapter in a way which may seem, at first, to be overly complicated. However, as the complexity of the algorithms increases in coming chapters, we will see that the effort required to understand this more sophisticated approach will have been worth it.

4.2 Arrays and Dynamic Memory Allocation

In C an array is stored in a block of contiguous memory. Memory itself is fundamentally a one-dimensional quantity since memory is ultimately accessed with a single memory address. Let us consider a one-dimensional array of doubles `ez` where the size is specified at run-time rather than at compile-time. The compiler needs to know about the existence of this array, but at compile-time it does not need to know the amount or the location of memory where the array will be stored. We can specify the size of the array when the program is run and we can allow the computer to decide the location of the necessary amount of memory. The compiler is told about the potential existence of the array with a pointer. Once the pointer is associated with the appropriate block of memory, it is virtually indistinguishable from a one-dimensional array.

The code shown in Fragment 4.1 demonstrates how the array `ez` can be created at run-time. In line 1 `ez` is declared as a pointer—it can store an address but when the program initially starts it does not point to anything meaningful. Line 2 declares two integer variables: `num_elements` which will contain the number of elements in the array, and `mm` which will be used as a loop counter. Lines 4 and 5 determine the number of elements the user desires.

Lecture notes by John Schneider. `fdtd-improved-code.tex`

Fragment 4.1 Fragment of code demonstrating how the size of the array `ez` can be set at run-time. The header file `stdlib.h` would typically have to be included to provide the prototype for the `calloc()` function.

```
1   double *ez;
2   int num_elements, mm;
3
4   printf("Enter the size of the array: ");
5   scanf("%d", &num_elements);
6
7   ez = calloc(num_elements, sizeof(double));
8
9   for (mm=0; mm < num_elements; mm++)
10    ez[mm] = 3.0 * mm;
```

Line 7 is the key to getting `ez` to behave as an array. In this line the pointer `ez` is set equal to the memory address that is returned by the function `calloc()`.[*] `calloc()` takes two arguments. The first specifies the number of elements in the array while the second specifies the amount of memory needed for a single element. Here the `sizeof()` operator is used to obtain the size of a double variable. (Although not shown in this fragment, when using the `calloc()` function one typically has to include the header file `stdlib.h` to provide the function prototype.) If `calloc()` is unable to provide the requested memory it will return `NULL`.[†]

After the call of `calloc()` in line 7, `ez` points to the start of a contiguous block of memory where the array elements can be stored. To demonstrate this, lines 9 and 10 write the value of three times the array index to each element of the array (so that `ez[0]` would be 0.0, `ez[1]` would be 3.0, `ez[2]` would be 6.0, and so on).

Some compilers will actually complain about the code as it is written in Fragment 4.1. The "problem" is that technically `calloc()` returns a void pointer—it is simply the address of the start of a block of memory, but we have not said what is stored at that memory. We want a pointer to doubles since we will be storing double precision variables in this memory. The compiler really already knows this since we are storing the address in `ez` which is a pointer to doubles. Nevertheless, some compilers will give a warning because of line 7. Therefore, to ensure that compilers do not complain, it would be best to replace line 7 with

```
ez = (double *)calloc(num_elements, sizeof(double));
```

In this way the void pointer returned by `calloc()` is converted (or cast) to a pointer to doubles.

[*]`calloc()` is closely related to the function `malloc()` which also allocates a block of memory and returns the address of the start of that block. However `calloc()` returns memory which has been cleared, i.e., set to zero, while `malloc()` returns memory which may contain anything. Since we want the field arrays initially to be zero, it is better to use `calloc()` than `malloc()`.

[†]Robust code would check the return value and take appropriate measures if `NULL` were returned. For now we will assume that `calloc()` succeeded.

4.3 Macros

C provides a preprocessor which "processes" your code prior to the compiler itself. Preprocessor directives start with the pound sign (`#`) and instruct the preprocessor to do things such as include a header file (with an `#include` statement) or substitute one string for another. Program 3.1 had three preprocessor directives. Two were used to include the files `stdio.h` and `math.h` and one used a `#define` statement to tell the preprocessor to substitute `200` for all occurrences of the string `SIZE`.

Compilers allow you to see what the source code is after passing through the preprocessor. Using the GNU C compiler, one adds the `-E` flag to the compiler command to obtain the output from the preprocessor. So, for example, one can see the source code as it appears after passing through the preprocessor with the command

```
gcc -E 1DbareBones.c
```

In this case you will observe that there are many, many more lines of output than there are in your original program. This is because of the inclusion of the header files. Your original program will appear at the end of the output except now `SIZE` does not appear anywhere. Instead, any place it had appeared you will now see `200`.

The `#define` statement can also be used to create a macro. Unlike the simple string substitution used before, a macro can take one or more arguments. These arguments dictate how strings given as arguments should re-appear in the output. For example, consider the following macro

```
#define SQR(X)  ((X) * (X))
```

This tells the preprocessor that every time `SQR` appears in the source code, the preprocessor should take whatever appeared as an argument and replace that with the argument multiplied by itself. Here the argument `X` is just serving as a place holder. Consider the following code

```
a = 6.0 * SQR(3.0 + 4.0);
```

After passing through the preprocessor, this code would appear as

```
a = 6.0 * ((3.0 + 4.0) * (3.0 + 4.0));
```

The result would be that `a` would be set equal to $6 \times 7^2 = 294$.

It may seem that there are an excess number of parentheses in this macro, but one must be careful with macros to ensure the desired results are obtained. Consider this macro

```
#define BAD_SQR(X)  X * X
```

If a program then contained this statement

```
a = 6.0 * BAD_SQR(3.0 + 4.0);
```

the preprocessor translates this to

```
a = 6.0 * 3.0 + 4.0 * 3.0 + 4.0;
```

Since multiplication has a higher precedence than addition, in this case a would be set equal to $18 + 12 + 4 = 34$. There is no harm in using extra parentheses, so do not hesitate to use them to ensure you are getting precisely what you want.

It is also worth noting that the macro definition itself will typically not be terminated by a semicolon. If one includes a semicolon, it could easily produce unintended results. As an example, consider

```
#define BAD_SQR1(X)  ((X) * (X));
```

If a program then contained this statement

```
a = BAD_SQR1(4.0) + 10.0;
```

the preprocessor would translate this to

```
a = ((4.0) * (4.0)); + 10;
```

Note that the "`+ 10`" is consider a separate statement since there is a semicolon between it and the squared term.[‡] Thus, a will be set to 16.0.

Macros may have any number of arguments. As an example, consider

```
#define FOO(X, Y) (Y) * cos((X) * (Y))
```

Given this macro, the following code

```
a = FOO(cos(2.15), 7.0 * sqrt(4.0));
```

would be translated by the preprocessor to

```
a = (7.0 * sqrt(4.0)) * cos((cos(2.15)) * (7.0 * sqrt(4.0)))
```

Macros can span multiple lines provided the newline character at the end of each line is "quoted" with a backslash. Said another way, a backslash must be the last character on a line if the statement is to continue on the next line.

There is another "trick" that one can do with macros. One can say they want the string version of an argument in the preprocessor output, essentially the argument will appear enclosed in quotes in the output. To understand why this is useful, it first helps to recall that in C, if one puts two string next to each other, the compiler treats the two separate strings as a single string. Thus, the following commands are all equivalent

```
printf("Hello world.\n");
printf("Hello "  "world.\n");
printf("Hello "
       "world.\n");
```

Each of these produce the output

```
Hello world.
```

[‡]When using the GNU C compiler, this "bad" code will compile without error. If one adds the `-Wall` flag when compiling, the GNU compiler will provide a warning that gives the line number and a statement such as "`warning: statement with no effect`." Nevertheless, the code will compile.

Note that there is no comma between the separated strings in the `printf()` statements and the amount of whitespace between the strings is irrelevant.

If we want the preprocessor to produce the string version of an argument in the output, we affix # to the argument name in the place where it should appear in the output. For example, we could use a macro to print what is being calculated and show the result of the calculation as well:

```
#define SHOW_CALC(X) \
        printf(#X " = %g\n", X)
```

The first line of this macro is terminated with a backslash telling the preprocessor that the definition is continued on the next line. Now, if the code contained

```
SHOW_CALC(6.0 + 24.0 / 8.0);
```

the preprocessor would convert this to

```
printf("6.0 + 24.0 / 8.0" " = %g\n", 6.0 + 24.0 / 8.0);
```

When the program is run, the following would be written to the screen

```
6.0 + 24.0 / 8.0 = 9
```

We are now at a point where we can construct a fairly sophisticated macro to do memory allocation. The macro will check if the allocation of memory was successful. If not, the macro will report the problem and terminate the program. The following fragment shows the desired code.

Fragment 4.2 Macro for allocating memory for a one-dimensional array. The trailing backslashes must appear directly before the end of the line. (By "quoting" the newline character at the end of the line we are telling the preprocessor the definition is continued on the next line.)

```
#define ALLOC_1D(PNTR, NUM, TYPE)                                \
    PNTR = (TYPE *)calloc(NUM, sizeof(TYPE));                    \
    if (!PNTR) {                                                 \
      perror("ALLOC_1D");                                        \
      fprintf(stderr,                                            \
          "Allocation failed for " #PNTR ".  Terminating...\n");\
      exit(-1);                                                  \
    }
```

The macro `ALLOC_1D()` takes three arguments: a pointer (called `PNTR`), an integer specifying the number of elements (`NUM`), and a data type (`TYPE`). The macro uses `calloc()` to obtain the desired amount of memory. It then checks if the allocation was successful. Recall that `calloc()` will return `NULL` if it failed to allocate the requested memory. In C, the exclamation mark is also the "not operator." So, if the value of `PNTR` is `NULL` (or, thought of another way, "false"), then `!PNTR` would be true. If the memory allocation failed, two error messages are printed (one message is generated using the system function `perror()`, the other uses `fprintf()` and writes the output to `stderr`, which is typically the screen). The program is then terminated with the `exit()` command.

With this macro in our code we could create an array `ez` with 200 elements with statements such as

```
double *ez;

ALLOC_1D(ez, 200, double);
```

Technically the semicolon in the second statement is not necessary (since this macro translates to a block of code that ends with a close-brace), but there is no harm in having it.

4.4 Structures

In C one can group data together, essentially create a compound data type, using what is known as a structure. A program can have different types of structures, i.e., ones that bundle together different kinds of data. For example, a program might use structures that bundle together a person's age, weight, and name as well as structures that bundle together a person's name, social security number, and income. Just as we can have multiple variables of a given type and each variable is a unique instance of that data type, we can have multiple variables that corresponds to a given structure, each of which is a unique instance of that structure.

Initially, when declaring a structure, we first tell the compiler the composition of the structure. As an example, the following command defines a `person` structure that contains three elements: a character pointer called `name` that will be used to store the name of a person, an integer called `age` that will correspond to the person's age, and an integer called `weight` that will correspond to the person's weight.

```
  struct person {
    char *name;
    int age;
    int weight;
  };
```

This statement is merely a definition—no structure has been created yet. To create a `person` structure called `bob` and another one called `sue`, a command such as the following could be used:

```
  struct person bob, sue;
```

Actually, one could combine these two statements and create `bob` and `sue` with the following:

```
  struct person {
    char *name;
    int age;
    int weight;
  }  bob, sue;
```

However, in the code to come, we will use a variant of the two-statement version of creating structures. It should also be pointed out that elements do not need to be declared with individual statements of their own. For example, we could write

```
  int age, weight;
```

instead of the two separate lines shown above.

To access the elements of a structure we write the name of the structure, a dot, and then the name of the element. For example, `bob.age` would be the `age` element of the structure `bob`, and `sue.age` would be the `age` element of the structure `sue` (despite the fact that these are both `age` elements, they are completely independent variables).

Now, let's write a program that creates two `person` structures. The program has a function called `showPerson1()` to show the elements of a `person` and another function called `changePerson1()` that reduces the person's age by two and decreases the weight by five. Each of these functions takes a single argument, a `person` structure. The program is shown in Program 4.3.

Program 4.3 `struct-demo1.c`: Program to demonstrate the basic use of structures.

```c
/* Program to demonstrate the use of structures.  Here structures are
passed as arguments to functions.*/

#include <stdio.h>

struct person {
  char *name;
  int age;
  int weight;
};

void changePerson1(struct person p);
void showPerson1(struct person p);

int main() {
  struct person bob, sue;

  sue.name = "Sue";
  sue.age = 21;
  sue.weight = 120;

  bob.name = "Bob";
  bob.age = 62;
  bob.weight = 180;

  showPerson1(sue);

  printf("*** Before changePerson1() ***\n");
  showPerson1(bob);
  changePerson1(bob);
  printf("*** After changePerson1() ***\n");
32   showPerson1(bob);
33
34   return 0;
35 }
36
37 /* Function to display the elements in a person. */
38 void showPerson1(struct person p) {
39   printf("name: %s\n", p.name);
40   printf("age: %d\n", p.age);
41   printf("weight: %d\n", p.weight);
42   return;
43 }
44
45 /* Function to modify the elements in a person. */
46 void changePerson1(struct person p) {
47   p.age = p.age - 2;
48   p.weight = p.weight - 5;
49   printf("*** In changePerson1() ***\n");
50   showPerson1(p);
51   return;
52 }
```

In line 16 of the `main()` function we declare the two `person` structures `bob` and `sue`. This allocates the space for these structures, but as yet their elements do not have meaningful values. In 18 we set the `name` of element of `sue`. The following two lines set the `age` and `weight`. The elements for `bob` are set starting at line 22. In line 26 the `showPerson1()` function is called with an argument of `sue`. This function, which starts on line 38, shows the `name`, `age`, and `weight` of a `person`.

After showing the elements of `sue`, in line 29 of `main()`, `showPerson1()` is used to show the elements of `bob`. In line 30 the function `changePerson1()` is called with an argument of `bob`. This function, which is given starting on line 46, subtracts two from the `age` and five from the `weight`. After making these modifications, in line 50, `showPerson1()` is called to display the modified person.

Finally, returning to line 32 of the `main()` function, the `showPerson1()` function is called once again with an argument of `bob`. Note that this is after `bob` has supposedly been modified by the `changePerson1()` function. The output produced by Program 4.3 is

```
name: Sue
age: 21
weight: 120
*** Before changePerson1() ***
name: Bob
age: 62
weight: 180
*** In changePerson1() ***
name: Bob
```

```
age: 60
weight: 175
*** After changePerson1() ***
name: Bob
age: 62
weight: 180
```

Here we see that the initial values of `sue` and `bob` are as we would anticipate. Also, when `showPerson1()` is called from within `changePerson1()`, we see the modified values for `bob`, i.e., his age has been reduced by two and his weight has been reduced by five. *However,* the last three lines of output show that, insofar as the `bob` structure in the `main()` function is concerned, nothing has changed! `bob` has not been modified by `changePerson1()`.

Although this behavior may not have been what we would have anticipated, it is correct. When a structure is given as an argument, the function that is called is given a complete copy of the original structure. Thus, when that function modifies the elements of the structure, it is modifying a copy of the original—it is not affecting the original itself.

If we want a function that we call to be able to modify a structure, we must pass a pointer to that structure. We will modify the code to permit this but let use first introduce the `typedef` statement that allows to write slightly cleaner code. Having to write `struct person` everywhere we want to specify a `person` structure is slightly awkward. C allows us to use the `typedef` statement to define an equivalent. The following statement tells the compiler that `Person` is the equivalent of `struct person`

```
  typedef struct person Person;
```

Note that there is no requirement that we use the same word for the structure and the `typedef`-equivalent. Additionally, even when using the same word, there is no need to use different capitalization (since the structure and its equivalent are maintained in different name spaces).

Now, let us assume we want to create two *pointers* to structures, one named `susan` and one `robert`. These can be created with

```
  struct person *susan, *robert;
```

Assuming the `typedef` statement given above has already appeared in the program, we could instead write:

```
  Person *susan, *robert;
```

`susan` and `robert` are pointers to structures but, initially, they do not actually point to any structures. We cannot set their elements because there is is no memory allocated for the storage of these elements. Thus, we must allocate memory for these structures and ensure the pointers point to the memory.

To accomplish this, we can include in our program a statement such as

```
  ALLOC_1D(susan, 1, Person);
```

Recalling the `ALLOC_1D()` macro presented in Sec. 4.3, this statement will allocate the memory for one `Person` and associate that memory with the pointer `susan`. We can now set the element associated with `susan`. However, accessing the elements of a *pointer* to a `Person` is different than directly accessing the elements of a `Person`. To set the `age` element of `susan` we would have to write either

```
  (*susan).age = 21;
```

or

```
  susan->age = 21;
```

Program 4.4 is similar to Program 4.3 in many respects except here, rather than using structures directly, the program primarily deals with pointers to structures. As will be shown, this allows other functions to change the elements within any given structure—the function merely has to be passed a pointer to the structure rather than (a copy of) the structure.

Program 4.4 `struct-demo2.c`: Program to demonstrate the basic use of pointers to structures.

```
/* Program to demonstrate the use of pointers to structures. */

#include <stdlib.h>
#include <stdio.h>

#define ALLOC_1D(PNTR, NUM, TYPE)                                \
    PNTR = (TYPE *)calloc(NUM, sizeof(TYPE));                    \
    if (!PNTR) {                                                 \
      perror("ALLOC_1D");                                        \
      fprintf(stderr,                                            \
          "Allocation failed for " #PNTR ".  Terminating...\n");\
      exit(-1);                                                  \
    }

struct person {
  char *name;
  int age;
  int weight;
};

typedef struct person Person;

void changePerson2(Person *p);
void showPerson2(Person *p);

int main() {
  Person *robert, *susan;

  ALLOC_1D(susan, 1, Person);
  ALLOC_1D(robert, 1, Person);

  susan->name = "Susan";
  susan->age = 21;
34    susan->weight = 120;
35
36    robert->name = "Robert";
37    robert->age = 62;
38    robert->weight = 180;
39
40    showPerson2(susan);
41
42    printf("*** Before changePerson2() ***\n");
43    showPerson2(robert);
44    changePerson2(robert);
45    printf("*** After changePerson2() ***\n");
46    showPerson2(robert);
47
48    return 0;
49  }
50
51  /* Function to display the elements in a person. */
52  void showPerson2(Person *p) {
53    printf("name: %s\n", p->name);
54    printf("age: %d\n", p->age);
55    printf("weight: %d\n", p->weight);
56    return;
57  }
58
59  /* Function to modify the elements in a person. */
60  void changePerson2(Person *p) {
61    p->age = p->age - 2;
62    p->weight = p->weight - 5;
63    printf("*** In changePerson2() ***\n");
64    showPerson2(p);
65    return;
66  }
```

The `typedef` statement in line 21 allows us to write simply `Person` instead of `struct person`. This is followed by the prototypes for functions `showPerson2()` and `changePerson2()`. These functions are similar to the corresponding functions in the previous program except now the arguments are pointers to structures instead of structures. Thus, the syntactic changes are necessary in the functions themselves (e.g., we have to write `p->age` instead of `p.age`). Starting on line 29 the `ALLOC_1D()` macro is used to allocate the memory for the `susan` and `robert` pointers that were declared in line 27. The values of the elements are then set, the contents of the structures displayed, `robert` is modified, and the contents or `robert` are shown again.

The output produced by Program 4.4 is

```
name: Susan
age: 21
```

```
weight: 120
*** Before changePerson2() ***
name: Robert
age: 62
weight: 180
*** In changePerson2() ***
name: Robert
age: 60
weight: 175
*** After changePerson2() ***
name: Robert
age: 60
weight: 175
```

Note that in this case, the changes made by `changePerson2()` are persistent—when the `main()` function shows the elements of `robert` we see the modified values (unlike with the previous program).

4.5 Improvement Number One

Now let us use some of the features discussed in the previous sections in a revised version of the "bare-bones" program that appeared in Sec. 3.5. Here we will bundle much of the data associated with an FDTD simulation into a structure that we will call a `Grid` structure. (We will often refer to this structure as simply the `Grid` or a `Grid`.) Here will define the `Grid` as follows:

```
struct Grid {
  double *ez;         // electric field array
  double *hy;         // magnetic field array
  int sizeX;          // size of computational domain
  int time, maxTime;  // current and max time step
  double cdtds;       // Courant number
};
```

In the "improved" program that we will soon see, there will not appear to be much reason for creating this structure. Why bother? The motivation will be more clear when we start to modularize the code so that different functions handle different aspects of the FDTD simulation. By bundling all the relevant information about the simulation into a structure, we can simply pass as an argument to each function a pointer to this `Grid`.

First, let us create a header file `fdtd1.h` in which we will put the definition of a `Grid` structure. In this file we will also include the (1D) macro for allocating memory. Finally, in a quest to keep the code as clean as possible, we will also define a few additional preprocessor directives: a macro for accessing the `ez` array elements, a macro for accessing the `hy` array elements, and some simple `#define` statements to facilitate references to `sizeX`, `time`, `maxTime`, and `cdtds`. The complete header file is shown in Program 4.5.

Program 4.5 `fdtd1.h`: Header file for the first "improved" version of a simple 1D FDTD program.

```
 1 #ifndef _FDTD1_H
 2 #define _FDTD1_H
 3
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6
 7 struct Grid {
 8   double *ez;
 9   double *hy;
10   int sizeX;
11   int time, maxTime;
12   double cdtds;
13 };
14
15 typedef struct Grid Grid;
16
17 /* memory allocation macro */
18 #define ALLOC_1D(PNTR, NUM, TYPE)                                \
19     PNTR = (TYPE *)calloc(NUM, sizeof(TYPE));                   \
20     if (!PNTR) {                                                \
21       perror("ALLOC_1D");                                       \
22       fprintf(stderr,                                           \
23           "Allocation failed for " #PNTR ".  Terminating...\n");\
24       exit(-1);                                                 \
25     }
26
27 /* macros for accessing arrays and such */
28 /* NOTE!!!!  Here we assume the Grid structure is g. */
29 #define Hy(MM)    g->hy[MM]
30 #define Ez(MM)    g->ez[MM]
31 #define SizeX     g->sizeX
32 #define Time      g->time
33 #define MaxTime   g->maxTime
34 #define Cdtds     g->cdtds
35
36 #endif  /* matches #ifndef _FDTD1_H */
```

Line 1 checks if this header file, i.e., `fdtd1.h`, has previously been included. Including multiple copies of a header file can cause errors or warnings (such as when a term that was previously defined in a `#define` statement is again mentioned in a different `#define` statement). In the simple code with which we are working with now, multiple inclusions are not a significant concern, but we want to develop the techniques to ensure multiple inclusions do not cause problems in the future. Line 1 checks for a previous inclusion by testing if the identifier (technically a compiler directive) `_FDTD1_H` is not defined. If it is not defined, the next statement on line 2 defines it. Thus, `_FDTD1_H` serves as something of a flag. It will be defined if this file has been previously included and will not be defined otherwise. Therefore, owing to the `#ifndef` statement at the start of the file, if this header file has previously been included the preprocessor will essentially ignore the rest of the file. The `#ifndef` statement on line 1 is paired with the `#endif` statement on line 36 at the end of the file.

Assuming this file has not previously been included, next, the header files `stdio.h` and `stdlib.h` are included. Since the macro `ALLOC_1D()` uses both `calloc()` and some printing functions, both these headers have to be included anywhere `ALLOC_1D()` is used.

The commands for defining the `Grid` structure start on line 7. Following that, the `ALLOC_1D()` macro begins on line 18. The macros given on lines 29 and 30 allow us to access the field arrays in a way that is easier to read. (Importantly, note that these statements assume that the `Grid` in the program has been given the name `g`! We will, in fact, make a habit of using the variable name `g` for a `Grid` in the code to follow.) So, for example, the macro on line 30 allows us to write `Ez(50)` instead of writing the more cumbersome `g->ez[50]`. Note that the index for the array element is now enclosed in parentheses and not square brackets. The definitions in lines 31 through 34 allow us to write `SizeX` instead of `g->sizeX`, `Time` instead of `g->time`, `MaxTime` instead of `g->maxTime`, and `Cdtds` instead of `g->cdtds`.

An improved version of Program 3.1 is shown in Program 4.6.

Program 4.6 `improved1.c`: Source code for an improved version of the bare-bones 1D FDTD program.

```c
/* Improved bare-bones 1D FDTD simulation. */

#include "fdtd1.h"
#include <math.h>

int main()
{
  Grid *g;
  double imp0 = 377.0;
  int mm;

  ALLOC_1D(g, 1, Grid);

  SizeX = 200;     // size of grid
  MaxTime = 250;   // duration of simulation
  Cdtds = 1.0;     // Courant number (unused)

  ALLOC_1D(g->ez, SizeX, double);
  ALLOC_1D(g->hy, SizeX, double);

  /* do time stepping */
  for (Time = 0; Time < MaxTime; Time++) {
```

```
23
24      /* update magnetic field */
25      for (mm = 0; mm < SizeX - 1; mm++)
26        Hy(mm) = Hy(mm) + (Ez(mm + 1) - Ez(mm)) / imp0;
27
28      /* update electric field */
29      for (mm = 1; mm < SizeX; mm++)
30        Ez(mm) = Ez(mm) + (Hy(mm) - Hy(mm - 1)) * imp0;
31
32      /* hardwire a source node */
33      Ez(0) = exp(-(Time - 30.0) * (Time - 30.0) / 100.0);
34
35      printf("%g\n", Ez(50));
36    } /* end of time-stepping */
37
38    return 0;
39  }
```

In line 8 the pointer to the `Grid` structure `g` is declared. Because we have just declared a pointer to a `Grid`, we next need to obtain the memory to store the elements of the structure itself. This allocation is done in line 12.

Because the `Grid` contains pointers `ez` and `hy`, we do not need to declare those field arrays separately. The size of the computational domain is set in line 14 while the duration of the simulation is set in 15. After the preprocessor has processed the code, these lines will actually be

```
  g->sizeX = 200;
  g->maxTime = 250;
```

At this point in the program the pointers `ez` and `hy` do not have any memory associated with them—we cannot store anything yet in the field arrays. Therefore the next step is the allocation of memory for the field arrays which is accomplished in lines 18 and 19.

The rest of the code is largely the same as given in Program 3.1. The only difference being a slight change in notation/syntax. Program 3.1 and Program 4.6 produce identical results.

This new version of the bare-bones simulation is split into two files: the header file `fdtd1.h` and the source file `improved1.c`. This is depicted in Fig. 4.1. The `main()` function in file `improved1.c` handles all the calculations associated with the FDTD simulation. Since there is only a single source file, there is no obvious advantage to creating a header file. However, as we further modularize the code, the header file will provide a convenient way to ensure that different source files share common definitions of various things. For example, many source files may need to know about the details of a `Grid` structure. These details can be written once in the header file and then the header file can be included into different source files as appropriate. Examples of this will be provided in the following sections.

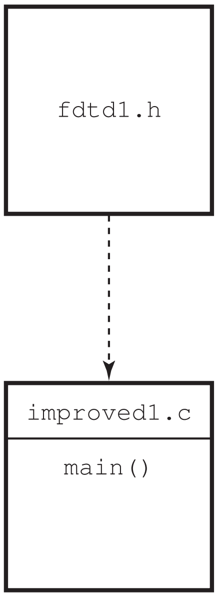

Figure 4.1: The files associated with the first improved version of the FDTD code. The header file `fdtd1.h` is included in the source file `improved1.c` as indicated by the dashed line. The source file `improved1.c` contains the `main()` function which performs all the executable statements associated with the simulation.

4.6 Modular Design and Initialization Functions

Thus far all the programs have been written using a single source file that contains a single function (the `main()` function), or in the case of the previous section, one source file and one header file. As the complexity of the FDTD simulation increases this approach becomes increasingly unwieldy and error prone. A better approach is to modularize the program so that different functions perform specific tasks—not everything is done within `main()`. For example, we may want to use one function to update the magnetic field, another to update the electric field, another to introduce energy into the grid, and another to handle the termination of the grid.

Additionally, with C it is possible to put different functions in different source files and compile the source files separately. By putting different functions in different source files, it is possible to have different functions that are within one source file share variables which are "hidden" from all functions that do not appear in that file. You can think of these variables as being private, known only to the functions contained in the file. As will be shown, such sharing of variables can be useful when one wants to initialize the behavior of a function that will be called multiple times (but the initialization only needs to be done once).

To illustrate how functions within a file can share variables that are otherwise hidden from other parts of a program, assume there is a function that we want to call several times. Further assume this function performs a calculation based on some parameters but these parameters only need to be set once—they will not vary from the value to which they are initially set. For this type of scenario, it is often convenient to split the function into two functions: one function handles the initialization of the parameters (the "initialization function") and the other function handles the calculation based on those parameters (the "calculation function"). Now, the question is: How can the initialization function make the parameters visible to the calculation function and how can the values of these parameters persist between one invocation of the function and the next? The answer lies in global variables.

Generally the use of global variables is discouraged as they can make programs hard to understand and difficult to debug. However, global variables can be quite useful when used properly. To help minimize the problems associated with global variables, we will further modularizing the program so that the initialization function and calculation function mentioned above are stored in a separate file from the rest of the program. In this way the global variables that these functions share are not visible to any other function.

As a somewhat contrived example of this approach to setting parameters, assume we want to write a program that will calculate the values of a harmonic function where the user can specify the amplitude and the phase, i.e., we want to calculate $f(x) = a\cos(x + \phi)$ where a is the amplitude and ϕ is the phase. Note that we often just write $f(x)$ for a function like this even though the function depends on a and ϕ as well as x. We usually do ***not*** write $f(x, a, \phi)$ because we think of a and ϕ as fixed values (even though they have to be specified at some point) while x is the value we are interested in varying. Assume we want to write our program so that there is a function `harmonic1()` that is the equivalent of $f(x) = a\cos(x+\phi)$. `harmonic1()` should take a single argument that corresponds to the value of x. We will use a separate function, `harmonicInit1()` that will set the amplitude and phase.

A file that contains a suitable the `main()` function and the associated statements to realize the parameter setting discussed above is shown in Program 4.7. The function prototypes for `harmonicInit1()` and `harmonic1()` are given in lines 12 and 13, respectively. Note, however, that these functions do not appear in this file. Between lines 19 and 23 the user is prompted for the amplitude and phase (and the phase is converted from degrees to radians). These values are passed as arguments to the `harmonicInit1()` function. As we will see a little later, this function sets persistent global parameters to these values so that they are visible to the function `harmonic1()` whenever it is called. The for-loop that starts on line 28 generates the desired output values. Here we set the variable `x` to values between 0 and 2π. In line 30 the value of `x` is printed together with the value of `harmonic1(x)`. Note that the `harmonic1()` function is not passed the amplitude or phase as an argument.

Program 4.7 `param-demo1.c`: File containing the `main()` function and appropriate header material that is used to demonstrate the setting of persistent parameters in an auxilliary function. Here `harmonicInit1()` and `harmonic1()` are serving this auxilliary role. The code associated with those functions is in a separate file (see Program 4.8).

```
1  /* param-demo1.c:  Program that demonstrates the setting of
2   * "persistent" parameters via the arguments of an initialization
3   * function.  Here the parameters control the amplitude and phase of a
4   * harmonic function f(x) = a cos(x + phi).  This program generates
5   * num_points of the harmonic function with the "x" value of the
6   * varying between zero and 2*pi.
7   */
```

```

#include <stdio.h>
#include <math.h>  // To obtain M_PI, i.e., 3.14159...

void harmonicInit1(double amp, double phase);
double harmonic1(double x);

int main() {
  double amp, phase, x;
  int mm, num_points = 100;

  printf("Enter the amplitude: ");
  scanf(" %lf", &amp);
  printf("Enter the phase [in degrees]: ");
  scanf(" %lf", &phase);
  phase *= M_PI / 180.0;

  /* Set the amplitude and phase. */
  harmonicInit1(amp, phase);

  for (mm = 0; mm < num_points; mm++) {
    x = 2.0 * M_PI * mm / (float)(num_points - 1);
    printf("%f %f\n", x, harmonic1(x));
  }

  return 0;
}
```

The file containing the functions `harmonicInit1()` and `harmonic1()` is shown in Program 4.8. In line 12 two static double variables are declared: `amp` is the amplitude and `phase` is the phase. These variables are visible to all the functions in this file but are not visible to any other functions (despite the common name, these variables are distinct from those with the same name in the `main()` function). Furthermore, the value of these variables will "persist." They will remain unchanged (unless we explicitly change them) and available for our use through the duration of the running of the program. (Note that, it may perhaps be somewhat confusing, but the "`static`" qualifier in line 12 does not mean constant. The value of these variables can be changed. Rather, it means these global variables are local to this file.)

The `harmonicInit1()` function starts on line 17. It takes two arguments. Here those arguments are labeled `the_amp` and `the_phase`. We must distinguish these variable names from the corresponding global variables. We accomplish this by putting the prefix `the_` on the corresponding global variable name. The global variables are set to the desired values in lines 18 and 19. The `harmonic1()` function that begins on line 25 then uses these global values in the calculation of $a\cos(x+\phi)$.

Program 4.8 `harmonic-demo1.c`: File containing the functions `harmonicInit1()` and harmonic1().

```
/*
 * harmonic-demo1.c: Functions to calculate a harmonic function of a
 * given amplitude and phase.  The desired amplitude and phase are
 * passed as arguments to harmonicInit1().  harmonicInit1() sets the
 * corresponding static global variables so that these values will be
 * available for the harmonic1() function to use whenever it is
 * called.
 */

#include <math.h> // for cos() function

/* Global static variables that are visible only to functions inside
   this file. */
static double amp, phase;

// initialization function
void harmonicInit1(double the_amp, double the_phase) {
  amp = the_amp;
  phase = the_phase;

  return;
}

// calculation function
double harmonic1(double x) {
  return amp * cos(x + phase);
}
```

Now, let us change these programs in order to further separate the `main()` function from the harmonic function. There is no reason that the `main()` function should have to prompt the user for the amplitude or phase. These values are simply passed along to the harmonic initialization function and never actually used in `main()`. Thus, a better approach would be to let the harmonic initialization function prompt the user for whatever input it needs. The `main()` function would merely call the initialization function and leave all the details up to it. So in the future, if one wanted to change the harmonic function so the user could specify a frequency as well as the amplitude and phase, that code could be added to the harmonic functions but the `main()` function would not have to be changed in any way.

The new version of the `main()` function is shown in Program 4.9. Note that there is now no mention of amplitude or phase in `main()`. That information has all been relegated to the harmonic functions themselves.

Program 4.9 `param-demo2.c`: Modified file containing the `main()` function which demonstrates the use of an initialization function to set parameters. In this version of the code the initilization function `harmonicInit2()` takes no arguments. The code for `harmonicInit2()` and `harmonic2()` is given in Program 4.10.

```
/* param-demo2.c: Program that demonstrates the setting of
 * "persistent" parameters via an initialization function.  Here the
 * initialization function handles all the details of obtaining the
 * parameters associated with the harmonic function.
 */

#include <stdio.h>
#include <math.h>  // To obtain M_PI, i.e., 3.14159...

void harmonicInit2();
double harmonic2(double x);

int main() {
  double x;
  int mm, num_points = 100;

  /* Initialize the harmonic function. */
  harmonicInit2();

  for (mm = 0; mm < num_points; mm++) {
    x = 2.0 * M_PI * mm / (float)(num_points - 1);
    printf("%f %f\n", x, harmonic2(x));
  }

  return 0;
}
```

The file containing `harmonicInit2()` and `harmonic2()` is shown in Program 4.10. As before, the amplitude and phase are global static variables that are declared in line 13. Note that `harmonicInit2()` takes no arguments. Instead, this function prompts the user for the amplitude and phase and sets the global variables appropriately. Having done this, these values are visible to the function `harmonic2()` (which is unchanged from the function `harmonic1()` given previously).

Program 4.10 `harmonic-demo2.c`: File containing the functions `harmonicInit2()` and `harmonic2()`.

```
1  /*
2   * harmonic-demo2.c: Functions to calculate a harmonic function of a
3   * given amplitude and phase.  harmonicInit2() prompts the user for
4   * the amplitude and phase and sets the global variables so these
```

```
 * values will be available for the harmonic2() function to use
 * whenever it is called.
 */

#include <math.h> // for cos() function

/* Global static variables that are visible only to functions inside
   this file. */
static double amp, phase;

// initialization function
void harmonicInit2() {

  printf("Enter the amplitude: ");
  scanf(" %lf", &amp);
  printf("Enter the phase [in degrees]: ");
  scanf(" %lf", &phase);
  phase *= M_PI / 180.0;

  return;
}

// calculation function
double harmonic2(double x) {
  return amp * cos(x + phase);
}
```

In Sec. 4.8 we will discuss the compilation of multi-file programs such as these.

4.7 Improvement Number Two

Let us consider a further refinement to the simple bare-bones FDTD simulation. In this version of the code the updating of the electric and magnetic fields will be handled by separate functions. The grid arrays will be initialized with a separate function and the source function will be calculated using a separate function. The arrangement of the functions among the various files is depicted in Fig. 4.2.

Program 4.11 shows the contents of the file `improved2.c`. As indicated in Fig. 4.2, `main()` calls `gridInit2()`. This function initializes the `Grid` structure `g`. The function `gridInit2()` is contained in the separate file `gridinit2.c`. The magnetic fields are updated using the function `updateH2()` while the electric fields are updated using `updateE2()`. Both these functions are contained in the file `update2.c`. The source function is calculated using the function `ezInc()` that is contained in the file `ezinc2.c`.

Program 4.11 `improved2.c`: Further code improvement of the bare-bones 1D FDTD simulation. Here the initialization of the grid as well as the updating of the fields are handled by separate

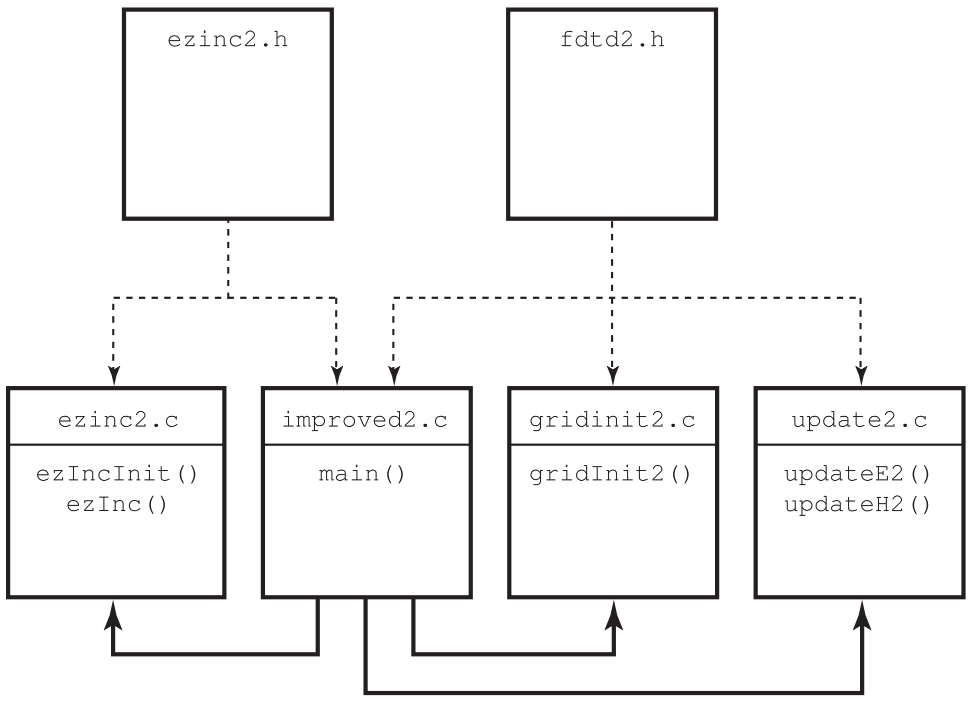

Figure 4.2: The files associated with the second improved version of the FDTD code. The header files `fdtd2.h` and `ezinc2.h` are included in the source files to which they are joined by a dashed line. The file `improved2.c` contains the `main()` function but the initialization of the grid, the calculation of the source function, and the updating of the fields are no longer done in `main()`. Instead, other functions are called to accomplish these tasks. The heavy black lines indicates which functions call which other functions. In this case, `main()` originates calls to all the other functions.

functions. The argument for these functions is merely the `Grid` pointer `g`. Additionally, the source function is initialized and calculated by separate functions.

```
1  /* Version 2 of the improved bare-bones 1D FDTD simulation. */
2
3  #include "fdtd2.h"
4  #include "ezinc2.h"
5
6  int main()
7  {
8    Grid *g;
9
10   ALLOC_1D(g, 1, Grid);        // allocate memory for Grid
11   gridInit2(g);                // initialize the grid
12
13   ezIncInit(g);                // initialize source function
14
15   /* do time stepping */
16   for (Time = 0; Time < MaxTime; Time++) {
17     updateH2(g);               // update magnetic field
18     updateE2(g);               // update electric field
19     Ez(0) = ezInc(Time, 0.0); // apply source function
20     printf("%g\n", Ez(50));    // print output
21   } // end of time-stepping
22
23   return 0;
24 }
```

Line 8 declares `g` to be a pointer to a `Grid`. Since a `Grid` structure has as elements the field arrays, the time step, the duration of the simulations, and the maximum number of time steps, none of these variables need to be declared explicitly. Note, however, that line 8 merely creates a pointer but as yet this pointer does not point to anything meaningful. Line 10 uses `ALLOC_1D()` to allocated memory for the `Grid` and ensures `g` points to that memory. Assuming there were no errors in the allocation, this line is effectively the equivalent of

```
  g = calloc(1, sizeof(Grid));
```

As shown in lines 11, 17, and 18, `gridInit2()`, `updateH2()`, and `updateE2()` each have a single argument: `g` (a pointer to the `Grid`). The parameters of the source function are initialized by calling `ezIncInit()` in line 13.

The header file `fdtd2.h` is shown in Program 4.12. This file is largely the same as `fdtd1.h`. The only significant difference is the function prototypes that are provided in lines 36–38 for three of the functions called by `main()` (note that the prototypes for the functions related to the source are provided in `ezinc2.h`).

Program 4.12 `fdtd2.h`: Header file to accompany the second version of the improved code showin in Program 4.11. The differences between this file and `fdtd1.h` are shown in bold.

```
#ifndef _FDTD2_H
#define _FDTD2_H

#include <stdio.h>
#include <stdlib.h>

struct Grid {
  double *ez;
  double *hy;
  int sizeX;
  int time, maxTime;
  double cdtds;
};

typedef struct Grid Grid;

/* memory allocation macro */
#define ALLOC_1D(PNTR, NUM, TYPE)                                  \
    PNTR = (TYPE *)calloc(NUM, sizeof(TYPE));                      \
    if (!PNTR) {                                                   \
      perror("ALLOC_1D");                                          \
      fprintf(stderr,                                              \
          "Allocation failed for " #PNTR ".  Terminating...\n");\
      exit(-1);                                                    \
    }

/* macros for accessing arrays and such */
#define Hy(MM)     g->hy[MM]
#define Ez(MM)     g->ez[MM]
#define SizeX      g->sizeX
#define Time       g->time
#define MaxTime    g->maxTime
#define Cdtds      g->cdtds

/* function prototypes */
void gridInit2(Grid *g);
void updateH2(Grid *g);
void updateE2(Grid *g);

#endif  /* matches #ifndef _FDTD2_H */
```

Program 4.13 shows the contents of the file `update2.c`. The static global variable `imp0` represents the characteristic impedance of free space and is set to 377.0 in line 6. This variable is never changed throughout the program. The magnetic field is updated with the function `updateH2()` which is given between lines 9 and 16. Note that the update equation uses `Hy()` and `Ez()` to refer to the elements of the field arrays. The macros in `fdtd2.h` translate these to the necessary syntax (which is essentially `g->hy[]` and `g->ez[]`). The electric field is updated using `updateE2()` which is given between lines 19 and 26.

Program 4.13 `update2.c`: Source code for the functions `updateH2()` and `updateE2()`.

```
 1 /* Functions to update the electric and magnetic fields. */
 2
 3 #include "fdtd2.h"
 4
 5 /* characteristic impedance of free space */
 6 static double imp0 = 377.0;
 7
 8 /* update magnetic field */
 9 void updateH2(Grid *g) {
10   int mm;
11
12   for (mm = 0; mm < SizeX - 1; mm++)
13     Hy(mm) = Hy(mm) + (Ez(mm + 1) - Ez(mm)) / imp0;
14
15   return;
16 }
17
18 /* update electric field */
19 void updateE2(Grid *g) {
20   int mm;
21
22   for (mm = 1; mm < SizeX - 1; mm++)
23     Ez(mm) = Ez(mm) + (Hy(mm) - Hy(mm - 1)) * imp0;
24
25   return;
26 }
```

Program 4.14 shows the source code for the function `gridInit2()`. This function is used to set the value of various elements of the `Grid`. (For this rather simple simulation, this program is itself quite simple.) Line 6 sets the size of the Grid, `SizeX`, to 200. Actually, after the preprocessor has processed the code, this line will be

```
  g->sizeX = 200;
```

Line 7 sets the duration of the simulation to 250 time-steps. Line 8 sets the Courant number to unity. Lines 10 and 11 use the `ALLOC_1D()` macro to allocate the necessary memory for the electric and magnetic field arrays.

Program 4.14 `gridinit2.c`: Source code for the function `gridInit2()`.

```
/* Function to initialize the Grid structure. */

#include "fdtd2.h"

void gridInit2(Grid *g) {
  SizeX = 200;                     // set the size of the grid
  MaxTime = 250;                   // set duration of simulation
  Cdtds = 1.0;                     // set Courant number

  ALLOC_1D(g->ez, SizeX, double); // allocate memory for Ez
  ALLOC_1D(g->hy, SizeX, double); // allocate memory for Hy

  return;
}
```

Finally, Program 4.15 shows the contents of the file `ezinc2.c` which contains the code to implement the functions `ezIncInit()` and `ezInc()`. The function `ezinc()` is Gaussian pulse whose width and delay are parameters that are set by `ezIncInit()`. The implementation of this source function is slightly different than in the original bare-bones code in that here the user is prompted for the width and delay. Additionally, the source function `ezInc()` takes two arguments, the time and the location, so that this function can be used in a TFSF formulation. When `ezInc()` is called from `main()`, the location is simply hardwired to zero (ref. line 19 of Program 4.11).

Program 4.15 `ezinc2.c`: File for the functions `ezIncInit()` and `ezInc()`. `ezInc()` is a traveling-wave implementation of a Gaussian pulse. There are three private static global variables in this code. These variables, representing the width and delay of the pulse as well as the Courant number, are determined and set by the initialization function `ezIncInit()`. The Courant number is taken from the `Grid` pointer that is passed as an argument while the user is prompted to provide the width and delay.

```
/* Functions to calculate the source function (i.e., the incident
 * field). */

#include "ezinc2.h"

/* global variables -- but private to this file */
static double delay, width = 0, cdtds;

/* prompt user for source-function width and delay. */
void ezIncInit(Grid *g){

```

```
12   cdtds = Cdtds;
13   printf("Enter delay: ");
14   scanf(" %lf", &delay);
15   printf("Enter width: ");
16   scanf(" %lf", &width);
17
18   return;
19 }
20
21 /* calculate source function at given time and location */
22 double ezInc(double time, double location) {
23   if (width <= 0) {
24     fprintf(stderr,
25        "ezInc: must call ezIncInit before ezInc.\n"
26        "       Width must be positive.\n");
27     exit(-1);
28   }
29   return exp(-pow((time - delay - location / cdtds) / width, 2));
30 }
```

In Program 4.15 the variable `width` is used to determine if initialization has been done. `width` is initialized to zero when it is declared in line 7. If it is not positive when `ezInc()` is called, an error message is printed and the program terminates. In practice one should check that all the parameters have been set to reasonable values, but here we only check the width.

The private variable `cdtds` declared in line 7 is distinct from `Cdtds` which the preprocessor expands to `g->cdtds`. That is to say the Courant number `cdtds` that is an element within the `Grid` is different from the private variable `cdtds` in this file. But, of course, we want these values to be the same and line 12 assures this.

The header file to accompany `ezinc2.c` is shown in Program `ezinc2.h`. As well as providing the necessary function prototypes, this file ensures the inclusion of `fdtd2.h` (which provides the description a `Grid` structure).

Program 4.16 `ezinc2.h`: Header file that accompanies `ezinc2.c` and is also included in the file that specifies `main()` (i.e., the file `improved2.c`.

```
/* Header file to accompany ezinc2.c. */

#ifndef _EZINC2_H
#define _EZINC2_H

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "fdtd2.h"

```

```
void ezIncInit(Grid *g);
double ezInc(double time, double location);

#endif  /* matches #ifndef _EZINC2_H */
```

4.8 Compiling Modular Code

When a program is divided between multiple files it is typically not necessary to recompile every file if there was a change in only some of them. Each source file can be compiled individually to an "object file" or object code. An object file, by itself, is not executable. (To create executable code, all the object code must be linked together.) To create an object file with the GNU C compiler, one uses the `-c` flag. Thus, for example, to obtain the object file for `ezinc2.c`, one would issue a command such as

```
gcc -Wall -O -c ezinc2.c
```

The flag `-Wall` means to show all warnings, while `-O` tells the compiler to optimize the code (greater optimization can be obtained by instead using `-O2` or `-O3`—the greater optimization usually comes at the cost of slower compilation and larger object and executable files). When this command is issued the compiler will create the object file `ezinc2.o`.

The object files for all the components of the program can be created separately by reissuing commands similar to the one shown above, e.g.,

```
gcc -Wall -O -c ezinc2.c
gcc -Wall -O -c improved2.c
gcc -Wall -O -c update2.c
gcc -Wall -O -c gridinit2.c
```

These commands would create the object files `ezinc2.o`, `improved2.o`, `update2.o`, and `gridinit2.o`. Alternatively, one can create all the object files at once using a command such as

```
gcc -Wall -O -c ezinc2.c improved2.c update2.c gridinit2.c
```

No matter how the object files are created, they need to be linked together to obtain an executable. The command to accomplish this is

```
gcc ezinc2.o improved2.o update2.o gridinit2.o -lm -o improved2
```

The flag `-lm` tells the compiler to link to the math library (which is necessary because of the math functions used in `ezinc2.c`). The `-o` flag allows one to specify the name of the output/executable file, in this case `improved2`.

For small programs there is not much advantage to incremental compilation. However, as the software increases in size and complexity, there may be a significant savings in time realized by recompiling only the code that has changed. For example, assume a change was made in the file `gridinit2.c` but in no other. Also assume that the object code for each file has previously been created. To obtain an executable that reflects this change one merely needs to recompile this one file and then link the resulting object file to the others, i.e.,

```
gcc -Wall -O -c gridinit2.c
gcc ezinc2.o improved2.o update2.o gridinit2.o -lm -o improved2
```

The details of incremental compilations can actually be handled by utilities that detect the files that need to be recompiled and react accordingly. Those who are interested in an example of such a utility may be interested in the `make` utility which is available on most Unix-based machines. Another helpful utility, which goes hand-in-hand with `make` is `makedepend` which sorts out the dependence of all the source files on the header files. `make` is a rather old utility—going back to the early days of Unix—and there are alternatives available such as SCons available from `www.scons.org`.

4.9 Improvement Number Three

Now let us modularize the program that contained the matched lossy layer (Program 3.8). As shown in Fig. 4.3, we will use separate functions for initializing the grid, taking snapshots, applying the TFSF boundary, updating the grid, and applying the ABC. Additionally, the calculation of the incident source function will be handled by a separate function. This source function will be called by the function that implements the TFSF boundary, but no other. Each box in Fig. 4.3 represents a separate file that contains one or more functions. The dashed lines indicate the inclusion of header files and the heavy lines indicate function calls from one file to another.

The source code `improved3.c` which contains the `main()` function is shown in Program 4.17. Note that nearly all activities associated with the FDTD simulation are now done by separate functions. `main()` merely serves to ensure that proper initialization is done and then implements the time-stepping loop in which the various functions are called.

Program 4.17 `improved3.c`: Source code containing the `main()` function. Nearly all the FDTD related activities have been relegated to separate functions which are called from `main()`.

```
/* FDTD simulation where main() is primarily used to call other
 * functions that perform the necessary operations. */

#include "fdtd3.h"

int main()
{
  Grid *g;

  ALLOC_1D(g, 1, Grid);  // allocate memory for Grid

  gridInit3(g);         // initialize the grid
  abcInit(g);           // initialize ABC
  tfsfInit(g);          // initialize TFSF boundary
  snapshotInit(g);      // initialize snapshots
```

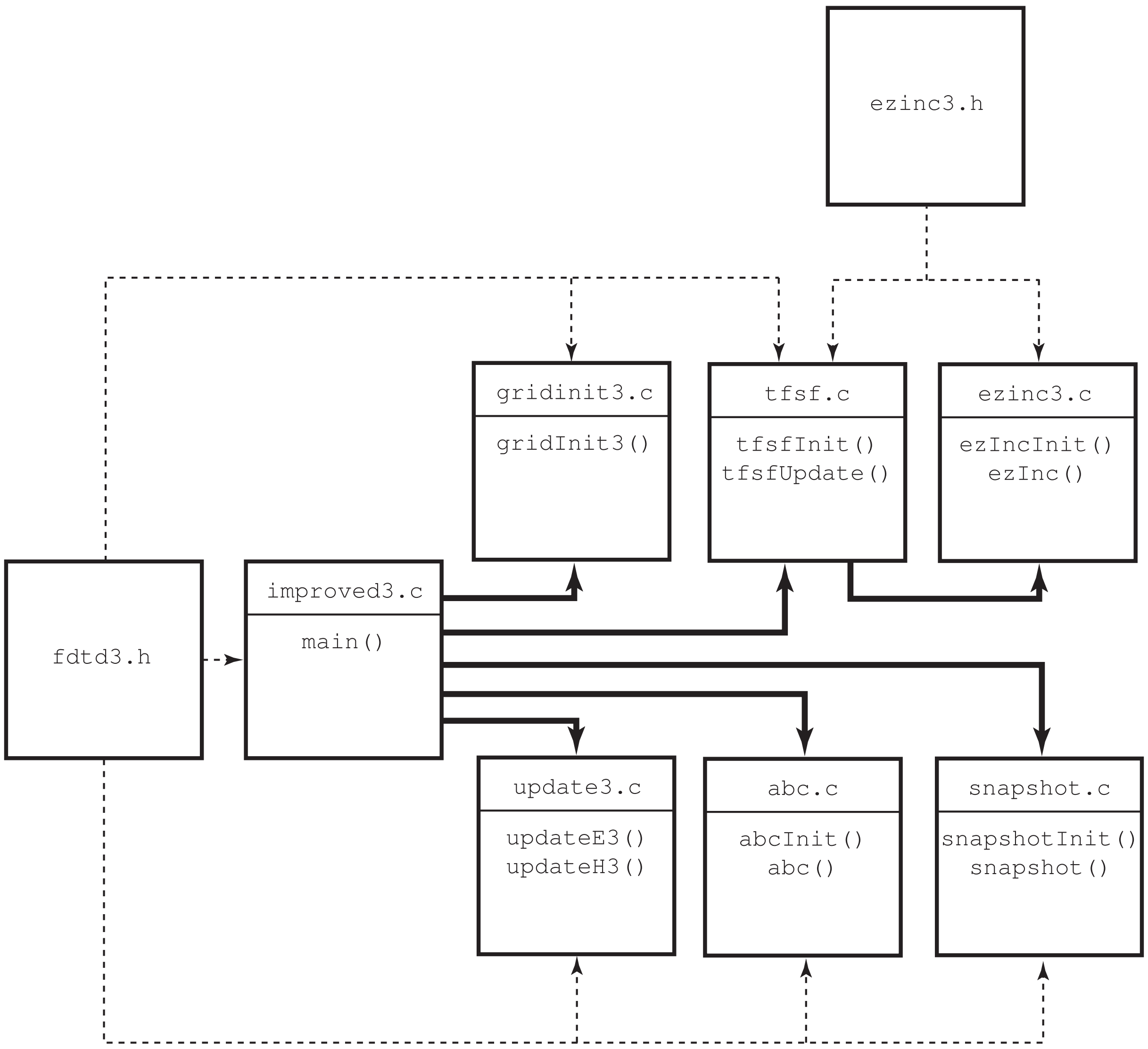

Figure 4.3: The files associated with the third improvement of the FDTD code. The header file `fdtd3.h` is explicitly included in the source files to which it is joined by a dashed line. (Wherever `ezinc3.h` appears it also ensures `fdtd3.h` is included.) The file `improved3.c` contains the `main()` function but the initialization of the grid, the application of the absorbing boundary condition, the calculation of the source function, the updating of the fields, and the taking of the snapshots is no longer done in `main()`. Instead, other functions are called to accomplish these tasks.

```
16
17    /* do time stepping */
18    for (Time = 0; Time < MaxTime; Time++) {
19      updateH3(g);    // update magnetic field
20      tfsfUpdate(g); // correct field on TFSF boundary
21      abc(g);         // apply ABC
22      updateE3(g);    // update electric field
23      snapshot(g);    // take a snapshot (if appropriate)
24    } // end of time-stepping
25
26    return 0;
27  }
```

We have any number of options in terms of how functions should be initialized or what arguments they should be passed. Thus, one should not consider this code to be optimum in any way. Rather this code is being used to illustrate implementation options.

As in the previous program, `main()` starts by defining a `Grid` pointer and allocating space for the actual structure. Then, in lines 12–15, four initialization functions are called. `gridInit3()` initializes the `Grid` (this will be discussed in more detail shortly). `abcInit()` handles any initialization associated with the ABC (as we will see, in this particular case there is nothing for this initialization function to do). `tfsfInit()` initializes the TFSF boundary while `snapshotInit()` does the necessary snapshot initialization. Following these initialization steps is the time-stepping loop where the various functions are called that do the actual calculations.

The header `fdtd3.h` is shown in Program 4.18. Looking just at `improved3.c` you would be unaware that the `Grid` structure has changed. However, four pointers have been added that will contain the update-equation coefficients. As can be seen in lines 8 and 9 of Program 4.18, these pointers are named `ceze`, `cezh`, `chyh`, and `chye`. Besides this and besides providing the function prototypes for the new functions, this header file is largely the same as `fdtd2.h`.

Program 4.18 `fdtd3.h`: Header file to accompany `improved3.c`. Differences from `fdtd2.h` are shown in bold.

```
1   #ifndef _FDTD3_H
2   #define _FDTD3_H
3
4   #include <stdio.h>
5   #include <stdlib.h>
6
7   struct Grid {
8     double *ez, *ceze, *cezh;
9     double *hy, *chyh, *chye;
10    int sizeX;
11    int time, maxTime;
12    double cdtds;
};

typedef struct Grid Grid;

/* macros for accessing arrays and such */
#define Hy(MM)    g->hy[MM]
#define Chyh(MM)  g->chyh[MM]
#define Chye(MM)  g->chye[MM]

#define Ez(MM)    g->ez[MM]
#define Ceze(MM)  g->ceze[MM]
#define Cezh(MM)  g->cezh[MM]

#define SizeX     g->sizeX
#define Time      g->time
#define MaxTime   g->maxTime
#define Cdtds     g->cdtds

/* memory allocation macro */
#define ALLOC_1D(PNTR, NUM, TYPE)                                   \
    PNTR = (TYPE *)calloc(NUM, sizeof(TYPE));                       \
    if (!PNTR) {                                                    \
      perror("ALLOC_1D");                                           \
      fprintf(stderr,                                               \
          "Allocation failed for " #PNTR ".  Terminating...\n");\
      exit(-1);                                                     \
    }

/* Function prototypes */
void abcInit(Grid *g);
void abc(Grid *g);

void gridInit3(Grid *g);

void snapshotInit(Grid *g);
void snapshot(Grid *g);

void tfsfInit(Grid *g);
void tfsfUpdate(Grid *g);

void updateE3(Grid *g);
void updateH3(Grid *g);

#endif  /* matches #ifndef _FDTD3_H */
```

The function `gridInit3()` is contained in the file `gridinit3.c` shown in Program 4.19. Keep in mind that it does not matter what file names are—file names do not have to match the contents of the file in any way, but, of course, it is best to use names that are descriptive of the contents.

The preprocessor directives in lines 5–7 are simply to provide convenient names to the amount of loss, the starting location of the lossy layer, and the relatively permittivity of the half space. These parameters are the same as they were in Program 3.8.

Program 4.19 `gridinit3.c`: The `gridInit3()` function to initialize the `Grid`.

```
1  /* Function initialize Grid structure. */
2
3  #include "fdtd3.h"
4
5  #define LOSS 0.02
6  #define LOSS_LAYER 180  // node at which lossy layer starts
7  #define EPSR 9.0
8
9  void gridInit3(Grid *g) {
10   double imp0 = 377.0;
11   int mm;
12
13   SizeX = 200;    // size of domain
14   MaxTime = 450; // duration of simulation
15   Cdtds = 1.0;    // Courant number
16
17   ALLOC_1D(g->ez,   SizeX, double);
18   ALLOC_1D(g->ceze, SizeX, double);
19   ALLOC_1D(g->cezh, SizeX, double);
20   ALLOC_1D(g->hy,   SizeX - 1, double);
21   ALLOC_1D(g->chyh, SizeX - 1, double);
22   ALLOC_1D(g->chye, SizeX - 1, double);
23
24   /* set electric-field update coefficients */
25   for (mm = 0; mm < SizeX; mm++)
26     if (mm < 100) {
27       Ceze(mm) = 1.0;
28       Cezh(mm) = imp0;
29     } else if (mm < LOSS_LAYER) {
30       Ceze(mm) = 1.0;
31       Cezh(mm) = imp0 / EPSR;
32     } else {
33       Ceze(mm) = (1.0 - LOSS) / (1.0 + LOSS);
34       Cezh(mm) = imp0 / EPSR / (1.0 + LOSS);
35     }
36
37   /* set magnetic-field update coefficients */
38   for (mm = 0; mm < SizeX - 1; mm++)
39      if (mm < LOSS_LAYER) {
40        Chyh(mm) = 1.0;
41        Chye(mm) = 1.0 / imp0;
42      } else {
43        Chyh(mm) = (1.0 - LOSS) / (1.0 + LOSS);
44        Chye(mm) = 1.0 / imp0 / (1.0 + LOSS);
45      }
46
47    return;
48  }
```

Lines 13–15 set the size of the `Grid`, the duration of the simulation, and the Courant number. Lines 17–22 allocate the necessary memory for the various arrays. The coefficient arrays are then set as they were in Program 3.8.

The functions to update the electric and magnetic fields, i.e., `updateE3()` and `updateH3()` are contained in the file `update3.c`. The contents of this file are shown in Program 4.20. The functions are largely unchanged from those that appeared in Program 4.13. The only significant differences are the appearance of the coefficient arrays in the update equations that start on lines 10 and 21.

Program 4.20 `update3.c`: Functions to update the electric and magnetic fields.

```
1   /* Functions to update the electric and magnetic fields. */
2
3   #include "fdtd3.h"
4
5   /* update magnetic field */
6   void updateH3(Grid *g) {
7     int mm;
8
9     for (mm = 0; mm < SizeX - 1; mm++)
10      Hy(mm) = Chyh(mm) * Hy(mm) +
11               Chye(mm) * (Ez(mm + 1) - Ez(mm));
12
13    return;
14  }
15
16  /* update electric field */
17  void updateE3(Grid *g) {
18    int mm;
19
20    for (mm = 1; mm < SizeX - 1; mm++)
21      Ez(mm) = Ceze(mm) * Ez(mm) +
22               Cezh(mm) * (Hy(mm) - Hy(mm - 1));
23
```

```
  return;
}
```

The function to apply the absorbing boundary conditions is rather trivial and is shown in Program 4.21. Also shown in Program 4.21 is the initiliztaion function `abcInit()`. In this particular case the ABC is so simple that there is no initializataion that needs to be done and hence this function simply returns. In Chap. 6 we will begin to consider more sophisticated ABC's that do indeed require some initialization. Thus, this call the initialization function is done in anticipation of that. As was the case in Program 3.8, the ABC is only applied to the left side of the grid. The right side of the grid is terminated with a lossy layer.

Program 4.21 `abc.c`: Absorbing boundary condition used by `improved3.c`. For this particular simple ABC there is nothing for the initialization function to do and hence it simply returns.

```
/* Functions to terminate left side of grid. */

#include "fdtd3.h"

// Initialize the ABC -- in this case, there is nothing to do.
void abcInit(Grid *g) {

  return;
}

// Apply the ABC -- in this case, only to the left side of grid.
void abc(Grid *g) {

  /* simple ABC for left side of grid */
  Ez(0) = Ez(1);

  return;
}
```

The code associated with the TFSF boundary is shown in Program 4.22. Line 7 declares a static global variable `tfsfBoundary` which specifies the location of the TFSF boundary. This variable is initialized to zero with the understanding that when the code is initialized it will be set to some meaningful (positive) value.

Program 4.22 `tfsf.c`: Code to implement the TFSF boundary.

```
1  /* Function to implement a 1D FDTD boundary. */
2
```

```
3  #include <math.h>
4  #include "fdtd3.h"
5  #include "ezinc3.h"
6
7  static int tfsfBoundary = 0;
8
9  void tfsfInit(Grid *g) {
10
11   printf("Enter location of TFSF boundary: ");
12   scanf(" %d", &tfsfBoundary);
13
14   ezIncInit(g); // initialize source function
15
16   return;
17 }
18
19 void tfsfUpdate(Grid *g) {
20   /* check if tfsfInit() has been called */
21   if (tfsfBoundary <= 0) {
22     fprintf(stderr,
23       "tfsfUpdate: tfsfInit must be called before tfsfUpdate.\n"
24       "            Boundary location must be set to positive value.\n");
25     exit(-1);
26   }
27
28   /* correct Hy adjacent to TFSF boundary */
29   Hy(tfsfBoundary) -= ezInc(Time, 0.0) * Chye(tfsfBoundary);
30
31   /* correct Ez adjacent to TFSF boundary */
32   Ez(tfsfBoundary + 1) += ezInc(Time + 0.5, -0.5);
33
34   return;
35 }
```

The initialization function `tfsfInit()` begins on line 9. The user is prompted to enter the location of the TFSF boundary. Then the initialization function for the source `ezIncInit()` is called to set the parameters that control the shape of the pulse.

The code to calculate the source function, i.e., the functions `ezIncInit()` and `ezInc()` is identical to that shown in Program 4.15. However, there would be one slight change to the file: instead of including `ezinc2.h`, line 4 of Program 4.15 would be changed to include `ezinc3.h`. We assume this modified program is in the file `ezinc3.c` which is not shown. The header file `ezinc3.h` would be nearly identical to the code shown in Program 4.16 except the "2" in lines 3, 4, and 9, would be changed to "3" (thus ensuring the proper inclusion of the header file `fdtd3.h` instead of `fdtd2.h`). Again, since this is such a minor change, the contents of `ezinc3.h` are not shown.

The function `tfsfUpdate()` which begins on line 19 is called once every time-step. This function applies the necessary correction to the nodes adjacent to the TFSF boundary. Because of the implicit timing within this file, this function needs to be called after the magnetic-field update, but before the electric-field update. The function first checks that the boundary location is positive. If it is not, an error message is printed and the program terminates, otherwise the fields adjacent to the boundary are corrected.

Finally, the code associated with taking snapshots of the field is shown in Program 4.23. `snapshotInit()` allows the user to specify the time at which snapshots should start, the temporal stride between snapshots, the node at which a snapshot should start, the node at which it should end, the spatial stride between nodes, and the base name of the output files. Assuming the user entered a base name of `sim`, then, as before, the output files would be named `sim.0`, `sim.1`, `sim.2`, and so on. If the user said the `startTime` is 105 and the `temporalStride` is 10, then snapshots would be taken at time-steps 105, 115, 125, and so on. Similarly, if the user specified that the `startNode` and `endNode` are 0 and 180, respectively, and the `spatialStride` is 1, then the value of every node between 0 and 180, inclusive, would be recorded to the snapshot file. If the `spatialStride` were 2, every other node would be recorded. If it were 3, every third node would be recorded. (Because of this, the `endNode` only corresponds to the actual last node in the snapshot file if its offset from the `startNode` is an even multiple of the spatial stride.)

Program 4.23 `snapshot.c`: Code for taking snapshots of the electric field.

```c
/* Function to take a snapshot of a 1D grid. */

#include "fdtd3.h"

static int temporalStride = 0, spatialStride, startTime,
  startNode, endNode, frame = 0;
static char basename[80];

void snapshotInit(Grid *g) {

  printf("For the snapshots:\n");
  printf("  Duration of simulation is %d steps.\n", MaxTime);
  printf("  Enter start time and temporal stride: ");
  scanf(" %d %d", &startTime, &temporalStride);
  printf("  Grid has %d total nodes (ranging from 0 to %d).\n",
         SizeX, SizeX-1);
  printf("  Enter first node, last node, and spatial stride: ");
  scanf(" %d %d %d", &startNode, &endNode, &spatialStride);
  printf("  Enter the base name: ");
  scanf(" %s", basename);

  return;
}

void snapshot(Grid *g) {
```

```
26   int mm;
27   char filename[100];
28   FILE *snapshot;
29
30   /* ensure temporal stride set to a reasonable value */
31   if (temporalStride <= 0) {
32     fprintf(stderr,
33       "snapshot: snapshotInit must be called before snapshot.\n"
34       "          Temporal stride must be set to positive value.\n");
35     exit(-1);
36   }
37
38   /* get snapshot if temporal conditions met */
39   if (Time >= startTime &&
40       (Time - startTime) % temporalStride == 0) {
41     sprintf(filename, "%s.%d", basename, frame++);
42     snapshot = fopen(filename, "w");
43     for (mm = startNode; mm <= endNode; mm += spatialStride)
44       fprintf(snapshot, "%g\n", Ez(mm));
45     fclose(snapshot);
46   }
47
48   return;
49 }
```

As shown in line 9, `snapshotInit()` takes a single argument, a pointer to a `Grid` structure. This function prompts the user to set the appropriate snapshot control values. The snapshot files themselves are created by the function `snapshot()` which starts on line 25. This function starts by checking that the temporal stride has been set to a reasonable value. If not, an error message is printed and the program terminates. The program then checks if time is greater than or equal to `startTime` and the difference between the current time and the `startTime` is a multiple of `temporalStride`. If not, the function returns. If those conditions are met, then, as we have seen before, a snapshot file is written and the frame counter is advanced. However, now there is some control over which nodes are actually written. Note that the function `snapshot()` is called every time-step. We could easily have checked in the `main()` function if the time-step was such that a snapshot should be generated and only then called `snapshot()`. Reasons for adopting either approach could be made but be aware that the overhead for calling a function is typically small. Thus, the savings realized by calling `snapshot()` less often (but still ultimately generating the same number of snapshots) is likely to be trivial. The approach used here keeps all the snapshot-related data in the snapshot code itself.

After compiling all this code (and linking it), the executable will produce the same results as were obtained from Program 3.8 (assuming, of course, the user enters the same parameters as were used in Program 3.8). With the new version of the code there are several improvements we could potentially use to our advantage. Assume, for instance, we wanted to do simulations of two different scenarios. We could create two `Grid` structures, one for scenario. Each grid would have its own `gridInit()` function, but other than that all the code could be used by either grid. The update functions could be applied, without any modification, to any grid. The snapshot function could be applied to any grid, and so on.