Recall from class that we can predict the weather by covering the region of interest (e.g., the Bay Area, California, or the United States) with a grid and then predicting the weather (i.e., temperature, pressure, wind, etc.) at each vertex of the grid for each time of interest. For programming assignment 1, we’re going to use a similar approach to solve a much simpler problem.

Suppose we have a metal bar which we’ve heated. Further suppose that we place the bar in a well-insulated sheath and then place ice at the ends of the bar. What will happen to the temperature in the bar as time goes by? Your intuition should tell you that eventually the bar will cool to nearly freezing.

By using a grid, we can simulate the temperature in the bar and check our intuition. In order to get the grid, we divide up the length of the bar into equal-sized segments and the time of interest into equal-sized intervals. To be explicit, suppose the length of the bar is 1 meter and the time of interest is 1 second. Further suppose that we divide the bar into $m$ segments of length $h = 1/m$, meters and we divide the time into $n$ intervals of length $d = 1/n$ seconds. This gives us the grid shown in Figure 1.

We’ll call the horizontal direction the $x$-direction. It corresponds to the bar: the left end of the bar is at 0, and the right end is at 1. We’ll call the vertical direction the $t$-direction, since it corresponds to time. Furthermore, we’ll call $ih = x_i$ and $jd = t_j$. We’ll also call the temperature at $x_i$ and $t_j$ $u(x_i, t_j)$.\footnote{We’re using $u$ since $t$ is already taken.} If we know the temperatures in the bar at time $t_j$, we can estimate the temperatures at time $t_{j+1}$ by using the following formula:

$$u(x_i, t_{j+1}) = u(x_i,t_j) + \frac{d}{h^2} [u(x_{i-1},t_j) - 2u(x_i,t_j) + u(x_{i+1},t_j)].$$

This looks amazingly complicated, but it’s really not too bad. It says that to get the temperature at any location at any time ($u(x_i, t_{j+1})$), we look at the temperature at the same location at the previous time ($u(x_i, t_j)$) and add to it a sort of “average” of the temperatures at the surrounding points at the previous time:

$$\frac{d}{h^2} [u(x_{i-1},t_j) - 2u(x_i,t_j) + u(x_{i+1},t_j)].$$

If you look at our formula closely, you’ll see that there are a couple of “gotchas.” First, if $i = 0$, then $x_i = 0 \cdot h = 0$, and $x_{i-1} = (-1) \cdot h = -h$, and $-h$ isn’t in our grid. We run
Figure 1: Grid for Computing Temperatures
into a similar problem, when \( i = m \). However, recall that we already know the temperatures when \( i = 0 \) and \( i = m \): these points correspond to the ends of the bar, which have ice next to them. So their temperatures are always 0 degrees Celsius. That is,

\[
u(x_0, t_j) = u(0, t_j) = 0 = u(1, t_j) = u(x_m, t_j),
\]

regardless of what \( t_j \) is.

The second gotcha is similar. When \( j + 1 = 0 \), \( t_{j+1} = 0 \) and in order to compute \( u(x_i, t_0) \) we need to know \( u(x_i, t_{-1}) = u(x_i, -d) \), and \( t = -d \) is also not in our grid. But remember that we initially heated the bar: presumably we did this so that we knew the temperature at each point \( x_i \) when \( t_j = 0 \).

**Program 1**

We can now give a more formal statement of programming assignment 1: You should write a C program that reads in

- \( m \): the number of segments into which the bar is divided
- \( n \): the number of intervals into which the time between 0 and 1 is divided
- The initial conditions: the values

\[
u(x_0, 0) = u(0, 0), u(x_1, 0), u(x_2, 0), \ldots, u(x_{m-1}, 0), u(x_m, 0) = u(1, 0).
\]

Your program should then print the times and values of \( u(x_i, t_j) \) for each point \( x_i \) on the bar and for each time \( t_j \). For example, if \( m = 5 \), then the length of each segment on the bar is \( 1/5 = 0.2 \) and a typical line of output might look something like this

\[
0.020 \ 0.000 \ 0.476 \ 0.769 \ 0.769 \ 0.476 \ 0.000
\]

The first value (0.020) is the time, the remaining values are \( u(x_i, 0.020) \). More explicitly,

\[
0.000 = u(x_0, 0.020) = u(0.0, 0.020) \\
0.476 = u(x_1, 0.020) = u(0.2, 0.020) \\
0.769 = u(x_2, 0.020) = u(0.4, 0.020) \\
0.769 = u(x_3, 0.020) = u(0.6, 0.020) \\
0.476 = u(x_4, 0.020) = u(0.8, 0.020) \\
0.000 = u(x_5, 0.020) = u(1.0, 0.020)
\]

(By the way, the values were printed with the format specifier \( \% .3f \).)

**Details**

In writing this program you should use two arrays of double:
double new_u[MAX], old_u[MAX];

The array new_u will store the temperatures that are being currently computed (i.e., \( u(x_i, t_{j+1}) \)) and old_u will store the temperatures corresponding to the previous time. So after reading and printing the initial data, the main loop of the program should look something like this:

```c
for (j = 1; j <= n; j++) {
    t_j = j*d;
    new_u[0] = new_u[m] = 0.0;
    for (i = 1; i < m; i++)
        new_u[i] = old_u[i] +
               d/(h*h)*(old_u[i-1] - 2*old_u[i] + old_u[i+1]);
    Print new_u values;
    Copy new_u into old_u for next pass;
}
```

The dimension for the two arrays, MAX, can be a global constant that’s defined before the main function with the statement

```c
const int MAX = 101;
```

The input value for \( m \), the number of segments, will not be greater than 100. So the arrays will always be large enough to store all the data.

**Testing and Debugging**

There are a couple of programs on the class website that you can use to help with testing and debugging. The first, `input_data.c` can be used to generate input data to the program \((m, n,\) and \(u(x_i, 0), i = 0, 1, \ldots, m)\). Furthermore, with the input data generated by `input_data.c`, the problem of finding the temperature of the bar has an exact solution. The second program, `exact_solution.c`, computes the exact solution, which you can compare to the output of your program.

Both `input_data.c` and `exact_solution.c` require that you input \( m \), the number of segments in the bar, \( n \), the number of time intervals, and a third integer \( k \). This third integer determines the “frequency” of the solution: at time 0, the solution is a wave and the integer \( k \) determines the number of crests and troughs in the wave. If \( k = 1 \), there’s a single crest. If \( k = 2 \), there’s a crest and a trough; if \( k = 3 \), there are two crests and one trough, etc.

**Note:** The output computed by `exact_solution.c` will bear no relation to the solution generated by your program unless the input to your program comes from `input_data.c`.

Your program can be run with the input generated by `input_data.c` by “redirecting” standard input. For example, if you tell `input_data.c` to call its output file `data`, and if your executable is called `solve_heat_eqn`, you can run it with the command

```
$ ./solve_heat_eqn < data
```
The dollar sign ($) is the shell prompt: you shouldn’t type it. The less than sign (<) tells the shell that it should take its input from the file data instead of the keyboard.

When your program is graded, it will be tested using Linux on one of the CS department systems. So before putting your final copy in your SVN repository, you should compile and test your program using Linux on one of the CS department systems.

A Caveat
The method you’re using is “unstable” if \( n < 2m^2 \). For practical purposes, this means that the solution it computes will be wildly incorrect unless \( n \), the number of time intervals is at least equal to \( 2m^2 \). So, for example, if you’re using \( m = 10 \) segments in the metal bar, you should use at least \( n = 200 \) time intervals.

Furthermore, even if \( n \geq 2m^2 \) the method isn’t very accurate. For example, in the program I wrote, the maximum error (the difference between the solution computed by my program and the exact solution) is about \( 6.2 \times 10^{-3} \) when I use input generated by input_data.c with \( m = 10, n = 200 \), and \( k = 1 \).

Input Errors
You can assume that the input to your program is correct. That is, you can assume that \( m \) and \( n \) are positive integers and the “initial condition” list will contain \( m + 1 \) doubles. So you don’t need to check the input for errors.

Submitting the Solution
After you’ve completed your program, you should copy it to a subdirectory of your submit directory called p1.

\[
\begin{align*}
$ scp my_prog.c stargate.cs.usfca.edu: \\
$ ssh stargate.cs.usfca.edu \\
$ mkdir /home/submit/cs220-01/<my_userid>/p1 \\
$ cp my_prog.c /home/submit/cs220-01/<my_userid>/p1
\end{align*}
\]

(The dollar-sign is the shell prompt: don’t type it.) The first command (scp) can be used to copy your program (which I’m calling my_prog.c) from your Mac or Linux computer to the CS server stargate. The second command (ssh) logs you on to stargate. These commands are typed in a terminal window. If you’re using Windows you can use the program winscp to copy your program to stargate, and the program putty to log on to stargate.

The third and fourth commands are typed while you’re logged on to stargate. The third command creates the p1 subdirectory of your submit directory. The fourth command copies your program (my_prog.c) from your home directory (where you’re typing the command) to the new p1 subdirectory.

Grading
1. Correctness will be 50% of your grade. Does your program find the correct solution given correct input?

2. Documentation will be 20% of your grade. Does your header documentation include the author’s name, the purpose of the program, and a description of how to use the program? Are the identifiers meaningful? Are any obscure constructs clearly explained? Does the function header documentation explain the purpose of the function, its arguments and its return value?

3. Source format will be 20% of your grade. Is the indentation consistent? Have blank lines been used so that the program is easy to read? Is the use of capitalization consistent? Are there any lines of source that are longer than 80 characters (i.e., wrap around the screen)?

4. Quality of solution will be 10% of your grade. Are any of your functions more than 40 lines (not including blank lines, curly braces, or comments)? Are there multipurpose functions? Is your solution too clever – i.e., has the solution been condensed to the point where it’s incomprehensible?

**Academic Honesty**

Remember that you can discuss the program with your classmates, but you cannot look at anyone else’s source code. (This includes source code that you might find on the internet.) If you have any doubt about whether what you’d like to do is legal, you should ask me first.