Energy Efficiency through Smart Wall Design

New Mexico
Supercomputing Challenge
Final Report
April 1, 2009

Team #65
Los Alamos Middle School

Team Members
Rachel Robey
Gabe Montoya
Teacher
Bob Dryja
Project Mentor
Bob Robey
Derrick Montoya

Table of Contents

Table of Figures

Executive Summary

We are modeling the heat transfer through walls to assess their energy efficiency, as an extension to a project from last year. Even with different team members, we were able to successfully extend our work.

- Quickly design and test any wall using interactive features
- Much faster run times for long simulations using implicit method
- Experimental model comparison for validation

One of our foremost goals was to enhance the interactivity of the input of the wall and the display. Any wall can be quickly designed, input, and tested, using the keyboard. The running program has an easy to read real-time display of temperatures, time, and outside temperature. This also makes it possible to easily change the resolution and size of the area modeled.

Simulations of long periods of time have much faster run times. We implemented a completely new computational method that makes these long simulations possible. This was done with an alternating direction implicit method.

An experimental model was built to be used to validate our computer model. Even though this we were just beginning validation, and there was still some guesswork at this point, there were amazing results. The inside temperature from our computational model closely matches the temperatures from the experimental model.

We have done significant additional work from our previous project. All interactive input is new, and the graphics have had considerable changes and improvements. We created an experimental model, which wouldn't have been possible previously. We also added the new method and began verification and validation of our model.

Introduction

Problem Statement

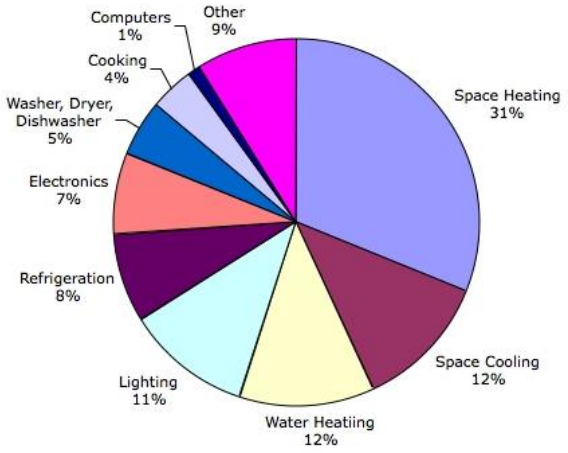

Figure 1 Break down of residential energy usage in 2006 (Revelle & Galland, 2008)

Energy conservation through building energy efficient homes has become an even more important issue in recent years. Both environmental and economical concerns have contributed to this interest. Space heating and cooling are the principal use of residential energy (Figure 1). Newer houses are being built to be more energy efficient as a result of this concern (Figure 2). Conserving heat already in a house by means of good wall design can cut down energy needs significantly, and, in turn, costs and fuel. There are countless wall designs available which have been created to be more energy efficient. When selecting a design it important to know how effective the design actually is, and how well it performs in individual climates. Our purpose is to create a versatile program that is capable of testing walls for efficiency in different environments.

This is a continuation of a project from last year. It was still somewhat incomplete at the end, and we were still interested in working with the problem more. Whereas last year's project was more focused on a specific wall design, this year was meant to be more flexible. We also took the opportunity to do some of the program improvements we had been interested in, such as better resolution and longer time steps.

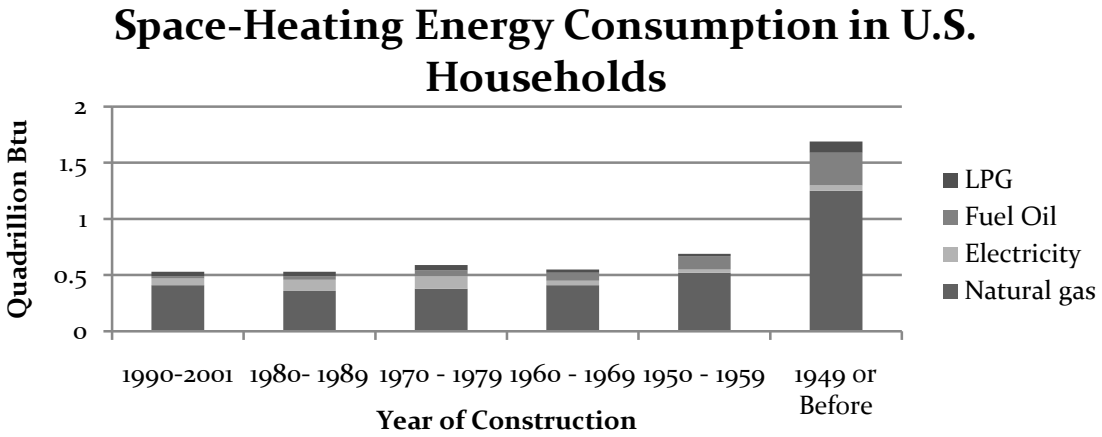

Figure 2 More recently constructed houses use less energy for space heating (Energy Information Administration, 2001)

Objective

Our objective was to extend our previous model to be used for a larger variety of walls and allow more extensive testing. Solving this problem computationally gives us the advantage of putting in changing conditions such as a thermal swing to simulate night and day. We added an 'interactive' wall design, which takes input from the keyboard to initialize the cell materials; this is very functional because we can easily change the wall that is being tested. We completely rewrote the code method, changing to Alternating Direction Implicit (ADI) which lets longer tests, such as months, be computed in shorter run times. We planned to compare data from our program runs, such as explicit versus implicit. We also built an experimental model, which we wanted to match in our program and compare results. This is the validation section of our project.

Wall Designs

There are many different wall designs proposed for energy efficiency. Our goal was to create a versatile program that was capable of testing a large range of walls, but we selected a few specific ones to try. Although solar gain is significant, especially in New Mexico, it requires radiation and the ability of glass to trap heat. This is more complex than we intended to make our program this year, so we did not choose any designs that were dependent on solar gain. We did want to try designs which take advantage of thermal swing, which we model on the outside boundary. This is also a good way to take advantage of the computer model, because it is hard to incorporate changing conditions in hand calculations.

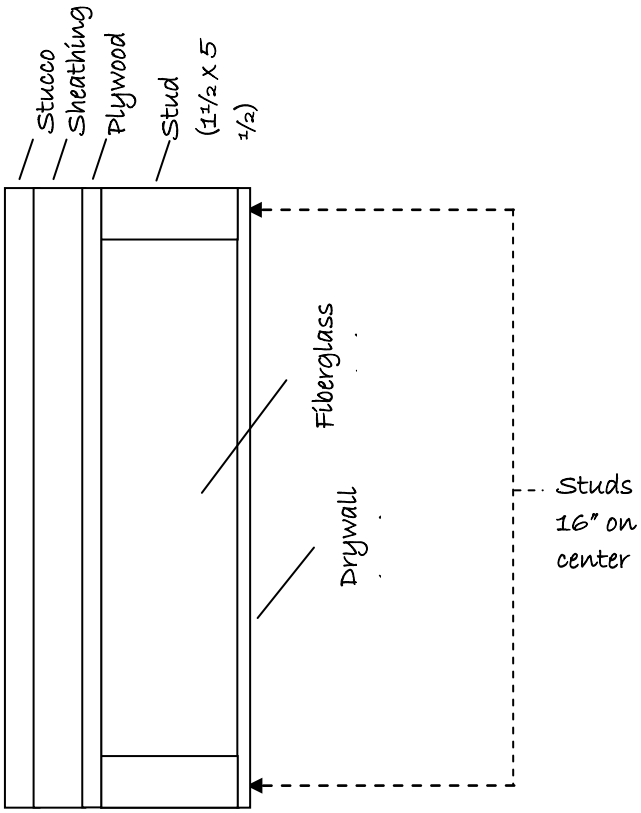

Figure 3 Standard wall design and materials

Our standard wall uses stucco, the foam sheathing, plywood, wooden studs, fiberglass insulation, and sheetrock (Figure 3). Stucco is common in building here in New Mexico. It consists of aggregate (sand, gravel, crushed stone or concrete), a binding material that works as a glue, and water. It is applied to the next layer, foam sheathing. The innermost layer sheetrock, which is a plaster, sometimes mixed with a fiber, and enclosed by heavy paper. Between the foam and sheetrock are wooden studs with fiberglass insulation. The studs are the framework and provide the structure of a building. The rest of the wall is built off the structure. However, there is a lot of heat loss through the studs because heat flows through the wood much more quickly than the insulation. The fiberglass insulation is meant to prevent the majority of the heat flow, and works best when it's not compressed.

There are many ways to make a wall more energy efficient. One approach is to add more mass so the energy travels through the wall more slowly. The wall can become unreasonably large though, and one application of our program would be to experiment with effectively adding mass without adding too much extra space. The use of water is also very common, because it has a large heat capacity. Energy is stored up in the water and emitted later. These are just a few possible design techniques that our program could be used to investigate. Since our goal was to make a program capable of testing these kinds of walls, we didn't spend very much time actually working with these designs.

Experimental Model

Building

We built an experimental model to be used for validation of our code. Our model consisted of a standard wall (Figure 4) and over-insulated foam walls for all of the other sides of the box (Figure 5). This minimized the influence from the other sides of the box, making it possible to test just the standard wall. We chose to use a concrete wall board in the place of stucco because it was easier to handle and they have similar heat transfer properties (Cooling and Heating: Load Calculation Manual, 1979). To build the wall, we started with a stud frame and screwed on the plywood and concrete board. Then we filled in the insulation and screwed the sheetrock into place. The foam walls were difficult to build because we did not want to use wood for a structure since it would not resist heat flow very well. We used a rigid foam for studs, with a thicker piece of soft foam on the outside, thinner on the inside, and insulation between. We positioned the pieces and pushed in nails so everything would be aligned for us to glue them together. We built each side separately, and later put them together. Since the box is front heavy, we put weights in the back to stabilize it. We put the box in the shadiest spot that we could find. We did this because we wanted to reduce solar radiation as much as possible. After setting the box up, we wrapped all the sides except the front in insulation and plastic. This was done to further reduce the heat lost from the sides.

Figure 5 Experimental test wall

Figure 4 Over-insulated foam walls

Testing

After researching different thermometers, we found a USB data logger that can be plugged directly into the computer (Figure 6).

Figure 6 Data logger used in experimental model

The time between samples, number of samples, etcetera can be set. We had created a rack to hold a light bulb and data logger inside of the box. The light bulb was put into the box to simulate the constant flow of energy in the model. We decided on using a 25 watt light bulb after learning from a dry run that a 40 watt bulb was bringing the box up over 100°F even with a very cold outside temperature. In between the light bulb and the data logger a piece of foil was placed to minimize radiation. The other data logger was taped onto a nearby window to take the outside temperature. In the first test the outside data logger was open to the elements; in the second test the data logger was wrapped in a plastic bag to reduce the influence of weather, such as wind, on the results. We were testing during late January and the very beginning[1], so the temperatures were fairly cold, but the weather more mild (no storms). Figure 7 shows the setup for our test, with the rack, the inside of the box, and the sides completely wrapped up and covered.

Figure 7 Set up for experimental testing

Results

We downloaded the data onto the computer from the data loggers after each test, and plotted the inside and outside temperatures together (Figure 8 and Figure 9).The very beginning of the plots should be discounted because they show the starting conditions, which are atypical. These are things such as placing the data logger or taking it out, and the light bulb warming up. The effect of putting the outside data logger in a bag is very clear in the difference between the two plots. The wind influenced the temperature readings and caused spikes in the plot, while second test has a much smoother line. The thermal swing is evident for the outside graph in both tests and is still visible on the inside. The inside remained well above the outside temperature, occasionally getting too warm, at 80°F or more.

[1] Specifically, our first test ran from January 24 3:45 pm to January 27 7:53 pm and our second test from February 1 2:23 pm to February 4 9:02 am.

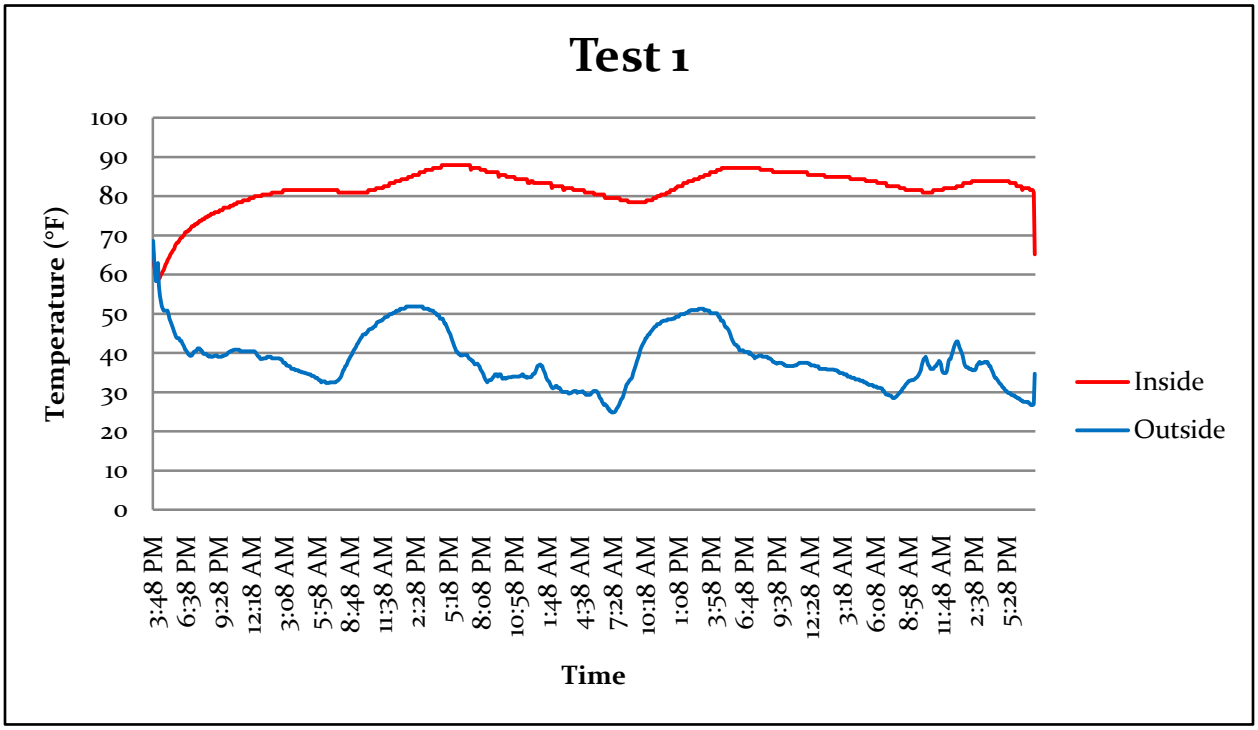

Figure 8 Temperature data from first experimental test

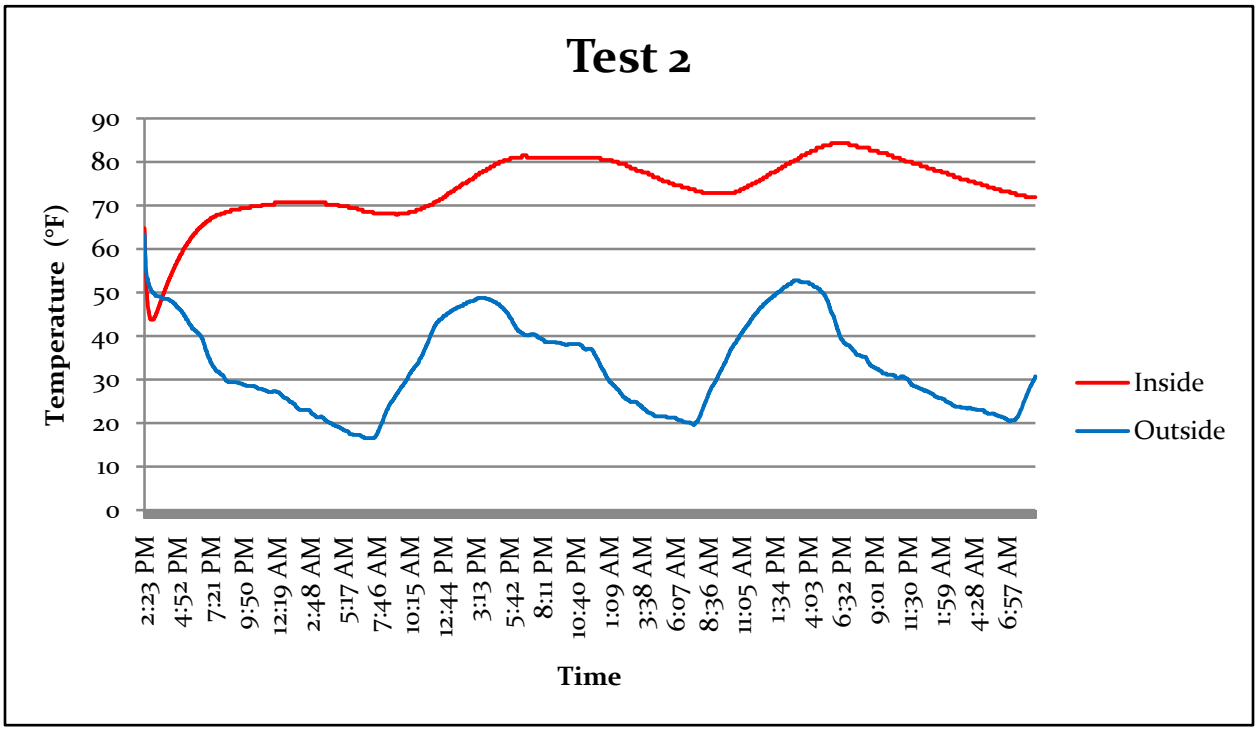

Figure 9 Temperature data from second experimental test

Mathematical Model

Heat Transfer

The flow of heat can be represented with three main equations (Kreith, Priciples of Heat Transfer, 1973), which make up the primary part of our mathematical model. Heat energy is transferred through conduction, convection, and radiation. Conduction is the movement of heat within a substance or between substances with a direct physical link. Energy is shifted from molecule to molecule. Because it does not require the molecules to move it occurs in all types of matter, including solids, and is found everywhere in our model. Convection is not strictly a type of heat transfer, but rather conduction combined with the movement of the molecules. Heat moves from a warm surface to and adjacent fluid or gas, which then rises. They lose heat to the colder molecules around them and sink back down, where they may be heated again, creating a circular flow. Convection occurs at the wall surfaces where it meets air. Radiation transfers heat without passing through all the molecules in a material. It travels in waves away from a warm body and is absorbed by objects it comes into contact with. The properties of each material were looked up in tables (Cooling and Heating: Load Calculation Manual, 1979) (Ashrae Handbook 1977 Fundamentals, 1977).

The rate of heat flow by conduction is equal to the product of the following values:

k, the conductivity of the material

A, the area through which heat is flowing, measured perpendicular to the direction of flow

$\frac{\Delta T}{\Delta x}$, the temperature gradient, or difference in temperature with respect to the distance and direction of heat flow

Since heat will always flow from an area of higher temperature to an area of lower temperature, the heat flow should be positive when the temperature gradient is negative, and a negative sign is included accordingly. This is written as an equation as shown below.

$$q_k = -kA\frac{\Delta T}{\Delta x}$$

In general, convection may be approximated by:

$$q_c = \bar{h}_c A \Delta T$$

where $\bar{h}_c$ = the convective heat transfer coefficient

A = the area through which heat is flowing, measured perpendicularly

ΔT = the difference between the surface and air temperatures

Convection is to just increase the rate of conduction, because that is its essential effect.
The net rate of radiant heat transfer from one blackbody, or ideal radiator, into another is given by:

$$q_r = \sigma A(T_1^4 - T_2^4)$$

where σ (sigma) = the Stefan-Boltzmann constant which is equal to 0.1714 x 10^{-8} BTU/hr ft^2 R^4

A = the surface area

$(T_1^4 - T_2^4)$ = the difference of the temperature of the emitting body to the fourth power and the temperature of the receiving body to the fourth power

In radiation, a blackbody is emits and absorbs the maximum amount of radiation at all wavelengths at any temperature. Real materials do not behave this way, and emit energy at a lower rate which is dependent on the properties of the surface. Radiation occurs both from the sun to the wall and from the wall into the night sky.

The equation of state shows the relationship between the temperature of a material and the energy that it contains. We use a fairly simple form of the equation, which is as follows:

$$E = c_v T$$

where E = energy

c_v = constant volume specific heat, or the energy required to raise a unit of mass one degree

T = the temperature of the material

Our problem involves only one conservation law, that of energy. There are no changes in density, mass, or momentum, so their conservation does not apply. Energy, however, is moved and changed in the process of heat transfer. “The law of conservation of energy states that energy can be neither created nor destroyed (Faires, 1970).” Since this heat transfer is not being applied

to a nuclear process, we may ignore the exception of the conversion of energy into mass or mass to energy. This law supports the equations:

$$E_{in} = \Delta E_s + E_{out}$$

$$E_{s1} + E_{in} - E_{out} = E_{s2}$$

where E_{in} and E_{out} are the energy entering and leaving the system respectively

ΔE_s is change in the energy stored in the system

E_{s1} and E_{s2} are the initial and final stored energy.

This law holds true for all of the equations, and any violation indicates an error of some kind.

Thermal Swing

We created a sinusoidal wave to be a general representation of the thermal swing of night and day. The outside boundary follows this wave and is a driving force in our program. We began with the equation for a general sine wave, which is shown as:

$$y(t) = A \cdot sin(\omega t + \theta)$$

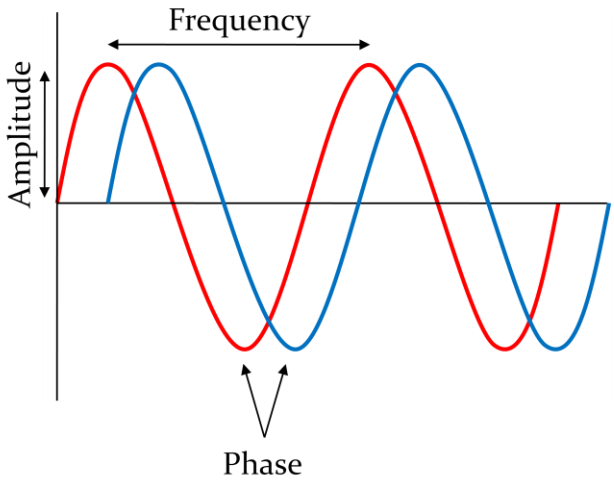

Figure 10 Parts of a sine wave

A is the amplitude, or distance from the center line

ω (omega) is the angular frequency, or radians per unit time

θ (theta) is the phase, or horizontal placement of the wave

With given minimum and maximum temperatures, the following formula can be used to create a sinusoidal wave.

$$T = \frac{max + min}{2} + \frac{max - min}{2} \sin\left(\left(t + t_{offset}\right)\frac{\pi}{720}\right)$$

Where *max* and *min* are the minimum and maximum temperatures

t and t_{offset} represent the current time and the time of a peak

$\frac{\pi}{720}$ sets how often the peak occurs, 720 is the number of minutes between each peak

Computational Model

Alternating Direction Implicit

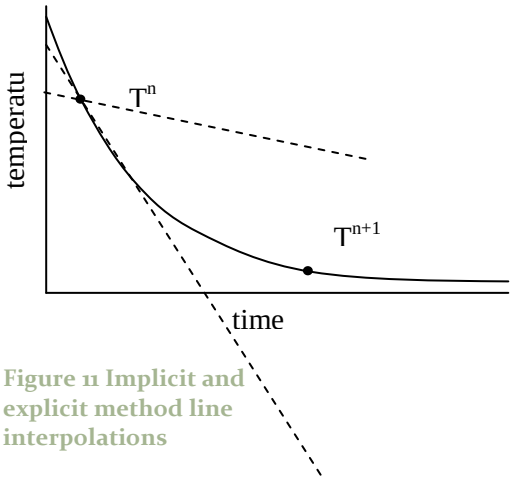

Figure 11 Implicit and explicit method line interpolations

The alternating direction implicit method (Peaceman & Rachford, 1955) (ADI) has a major advantage over our previous explicit method: the ability to remain stable with any size time step. This, in turn, affects the number of iterations. Combined with less work per iteration, it results in much shorter run times for of longer test periods, such as weeks or months. An explicit method determines the heat transfer and finds the energy in the cell for the next time step using the slope of a line at the current time step. If the line has a steep slope, and the time step is too long, the value can become negative. The implicit method, however, solves this problem by using the slope from the next time step. This assures that the result will always be positive, and while the error may be larger, the program will not become unstable or crash. This is a particularly good method for our heat transfer problem because the changes are gradual and the additional error will be relatively small.

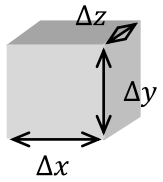

Figure 12 Cell dimensions and their variables

The equation used to find the heat transfer through conduction is as follows:

$$\frac{\Delta E}{\Delta t} = \Delta y \Delta z k_{[j][i-\frac{1}{2}]} \frac{\Delta T}{\Delta x} - \Delta y \Delta z k_{[j][i+\frac{1}{2}]} \frac{\Delta T}{\Delta x}$$

where ΔE = the change in energy

Δt = the change in time, or time step

$\Delta y \Delta z$ = the area, height times depth (Figure 12)

k = the conductivity[2]

[2] Note that time is indicated in the superscript and space in the subscript throughout the equations in the computational method section. For example, n is the current time step, and n+1 the next time step. The indices are denoted by i and j; boundaries are "half steps".

Since the ADI method uses multiple values from the next time step in its computation, and these values are unknown, a solver is used to solve for these values simultaneously. We are using a tridiagonal solver, which handles one row or column at a time, and uses the coefficients of T_{i-1}^{n+1}, T_i^{n+1}, T_{i+1}^{n+1}, and T_i^n. To find these coefficients we derive the equation into the form of:

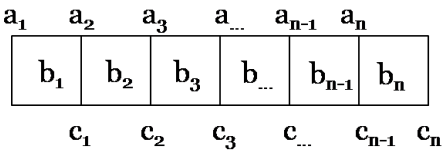

Figure 13 Locations and indices of flux coefficients

$$c_v\Delta x\Delta y\Delta z\rho T_i^{n+1} - \Delta y\Delta z\Delta t k_{i-\frac{1}{2}}\frac{T_i^{n+1}}{\Delta x} + \Delta y\Delta z\Delta t k_{i-\frac{1}{2}}\frac{T_{i-1}^{n+1}}{\Delta x} + \Delta y\Delta z\Delta t k_{i+\frac{1}{2}}\frac{T_{i+1}^{n+1}}{\Delta x}$$
$$- \Delta y\Delta z\Delta t k_{i+\frac{1}{2}}\frac{T_i^{n+1}}{\Delta x} = S_i\Delta t + c_v\Delta x\Delta y\Delta z\rho T_i^n$$

and find the value of each coefficient, which we then send to the solver:

$$a_i = \frac{\Delta y\Delta z\Delta t k_{i-\frac{1}{2}}}{720\Delta x}$$

$$c_i = \frac{\Delta y\Delta z\Delta t k_{i+\frac{1}{2}}}{720\Delta x}$$

$$b_i = \frac{C_v\Delta x\Delta y\Delta z\rho}{1728} - a_i - c_i$$

$$d_i = S_i\Delta t + \left(\frac{C_v\Delta x\Delta y\Delta z\rho}{1728}\right)T_i^n$$

The solver will return the solution, which is the temperature at *i.* In order to find the heat transfer through two dimensions, the solution of for each row is stored, and the flux from each column added.

Notice that d_i includes the term $S_i\Delta t$. This is a source term which adds energy directly to the cell. This can be used to represent radiation or a light bulb.

Tridiagonal Solver

A tridiagonal solver (Tridiagonal Matrix Algorithm, 2008) is used to solve tridiagonal systems of equations which may be written as[3]:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$$

or in matrix form, it is written as:

$$\begin{bmatrix} b_0 & c_0 & 0 & 0 & 0 \\ a_1 & b_1 & c_1 & 0 & 0 \\ 0 & a_2 & b_2 & \cdot & 0 \\ 0 & 0 & \cdot & \cdot & c_{n-1} \\ 0 & 0 & 0 & a_n & b_n \end{bmatrix}$$

This three diagonals formed by the a's, b's, and c's are what gives the solver and equations the "tridiagonal" part of their name. The diagonal pattern is a result of a cell's temperature being dependent on only itself, and the fluxes of its boundaries. a_0 and c_n are not shown because they are equal to zero and do not affect the cell's energy.

The first two equations complete a forward sweep which eliminates the a_i's; the second two perform back substitution, leaving x as the solution[4].

$$c'_i = \begin{cases} \dfrac{c_0}{b_0} & ; i = 0 \\ \dfrac{c_i}{b_i - c'_{i-1} a_i} & ; i = 1,\ 2, \dots, n-1 \end{cases}$$

$$d'_i = \begin{cases} \dfrac{d_0}{b_0} & ; i = 0 \\ \dfrac{d_i - d'_{i-1} a_i}{b_i - c'_{i-1} a_i} & ; i = 1,\ 2, \dots, n \end{cases}$$

$$x_n = d'_n \qquad ; i = 0$$

$$x_i = d'_i - c'_i x_{i+1} \qquad ; i = n-1, \dots, 1$$

[3] a_1 and c_n must equal zero. No energy enters through the boundaries of the mesh, so this is true.
[4] The solution can be obtained in O(n) operations rather than the $O(n^3)$ used for Gaussian elimination

Explicit Finite Difference

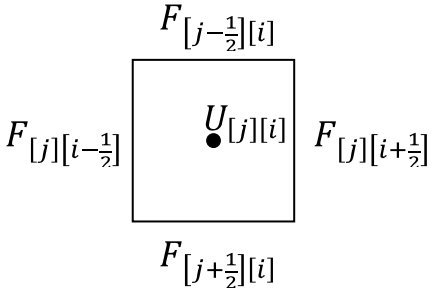

Figure 14 State variable and flux of cell with indices

We used an explicit central finite difference method in our code last year. This method also uses cells, and determines the energy in the cell in the next time step by using the flux at the boundaries. The values used in the calculation are from the current time step, which is what makes it, as stated, an explicit method. The energy of a cell in the next time step is equal to its current energy plus the flux energy entering and leaving on the right, left, bottom, and top boundaries. The mathematical equation for this is as shown below:

$$U_{[j][i]}^{n+1} = U_{[j][i]}^{n} + \frac{\Delta t}{\Delta x}\left(F_{[i+\frac{1}{2}][j]}^{n} - F_{[i-\frac{1}{2}][j]}^{n}\right) + \frac{\Delta t}{\Delta y}\left(F_{[i][j+\frac{1}{2}]}^{n} - F_{[i][j-\frac{1}{2}]}^{n}\right)$$

where U = the state variable, energy

Δt = the time step

Δx and Δy = the width and height of a cell

F = the flux

The term $\frac{\Delta t}{\Delta x}F$ is simplified, as result of cancelled units:

$$F\left(\frac{BTU}{\cancel{hr} - \cancel{ft^2}}\right)\Delta t(\cancel{hr})\frac{\cancel{\Delta y}\cancel{\Delta z}(\cancel{ft^2})}{\Delta x\cancel{\Delta y}\cancel{\Delta z}(ft^3)} = F\left(\frac{BTU\Delta t}{\Delta x ft}\right)$$

Energy is added or subtracted from the state variable value, depending on this flux. In our implementation of this method, we were first order in time and second order in space, which means we find the flux once every time step, and in two directions.

Boundary Conditions

The explicit and implicit methods also have different boundaries and boundary conditions. The explicit method uses ghost cells, a row of cells around the actual mesh. We give these special conditions. The top and bottom boundaries simply reflect the energy to ensure no energy is lost or gained through them. The inside boundary is maintained at 70°F, and the energy needed to maintain it is calculated. The outside boundary follows the sinusoidal wave for a given

maximum and minimum temperature. These side boundaries are also the driving force of the model; they provide a continual change in temperature to drive the heat transfer and prevent all the cells from reaching equilibrium.

The implicit method does not work the same way, and does not have any ghost cells. The cells farthest to the left are set to the outside temperature every iteration. The top, bottom, and right cells have no fluctuation on any boundary that is on the edge of the mesh. Heat is provided as a constant energy input through source cells, which are interactively set.

Assumptions

When creating the computer model, we made several assumptions that may not accurately emulate the real world. These can cause the output from our program to differ from actuality. There are limitations to what our program is capable of if it not capable of taking into account certain conditions. Although we added the option of reading data from a file to control the outside temperature, the alternative of the sinusoidal wave is a limitation. We assume a regular wave that has a maximum at noon and a minimum at midnight, and our approximation is acceptable, but differs from our real data by a large margin. We didn't have time to add radiation to our model, and this causes considerable solar gain and night radiation from the wall to be ignored. Also, two dimensions restrict the program from things that require the third dimension. Some examples are convection, which really moves in 3-D space, and walls, which have different layers. Similarly, we don't take into account air leaks and imperfections in the building and materials. Along with our sinusoidal wave, we estimate our convection and do not include weather. Realistically, the temperature, weather, and wind vary from day to day and are irregular.

Assumption/Limitation	Description
Sinusoidal wave	Adequate approximation, but doesn't give very accurate model of thermal swing
Radiation	Radiation wasn't added in, so solar gain is ignored
2-D	Not as important to our model, but some properties of a wall or actions need three dimensions
Uniform materials	Won't take into account imperfections such as air leakage, faulty building, etc.
Convection estimated	Convection acts as an constant increase in conductivity of air, rather than irregular wind
No weather	Weather has a huge influence on the outside conditions, and these affect the heat transfer

Figure 15 Description of assumptions and limitations in the program

Code

Program

We are using our code from last year, which was written off of the structure of the shallow water simulation code WAVE (Robey B. , 2007). The language we are using is C. We also borrowed code for our tridiagonal solver (Tridiagonal Matrix Algorithm, 2008), and for the interactive keys (Robey, Holland, Jacobs, & Shlachter, 2008) and integrated it into the rest of our program. For the graphics, we used the MPE library (Multi-Processing Environment) (Web Pages for MPI and MPE, 2004). Since WAVE was parallelized, there are still MPI (Message Passing Interface) calls in the code, but they have not been updated or run. Our code consists of a "main" file, a header, a display file with all of the display subroutines, and a file with the tridiagonal solver subroutine. In total, it is about 830 lines.

We were fortunate not to have to do too excessive debugging. We did, however, make extensive changes to our code and several bugs were caused by pieces of the code that were no longer valid. There were also some problems with the conduction after we wrote it with the implicit method.

Computation

In our code, the iteration loop calculates the heat transfer through all the cells for one time step. The structure of this loop is illustrated in Figure 16. In the first pass, we find the heat flow in the x and y directions. Both use the tridiagonal solver subroutine, and calculate the coefficients using the same 'old' array of temperatures. When calculating the coefficients we take advantage of a_i being equal to c_{i-1} and set the c array using the a array. The 'new' temperatures are stored in a separate array. In the second pass we update the 'old' temperature array. Also in the second pass, the display is called only called every set number of iterations.

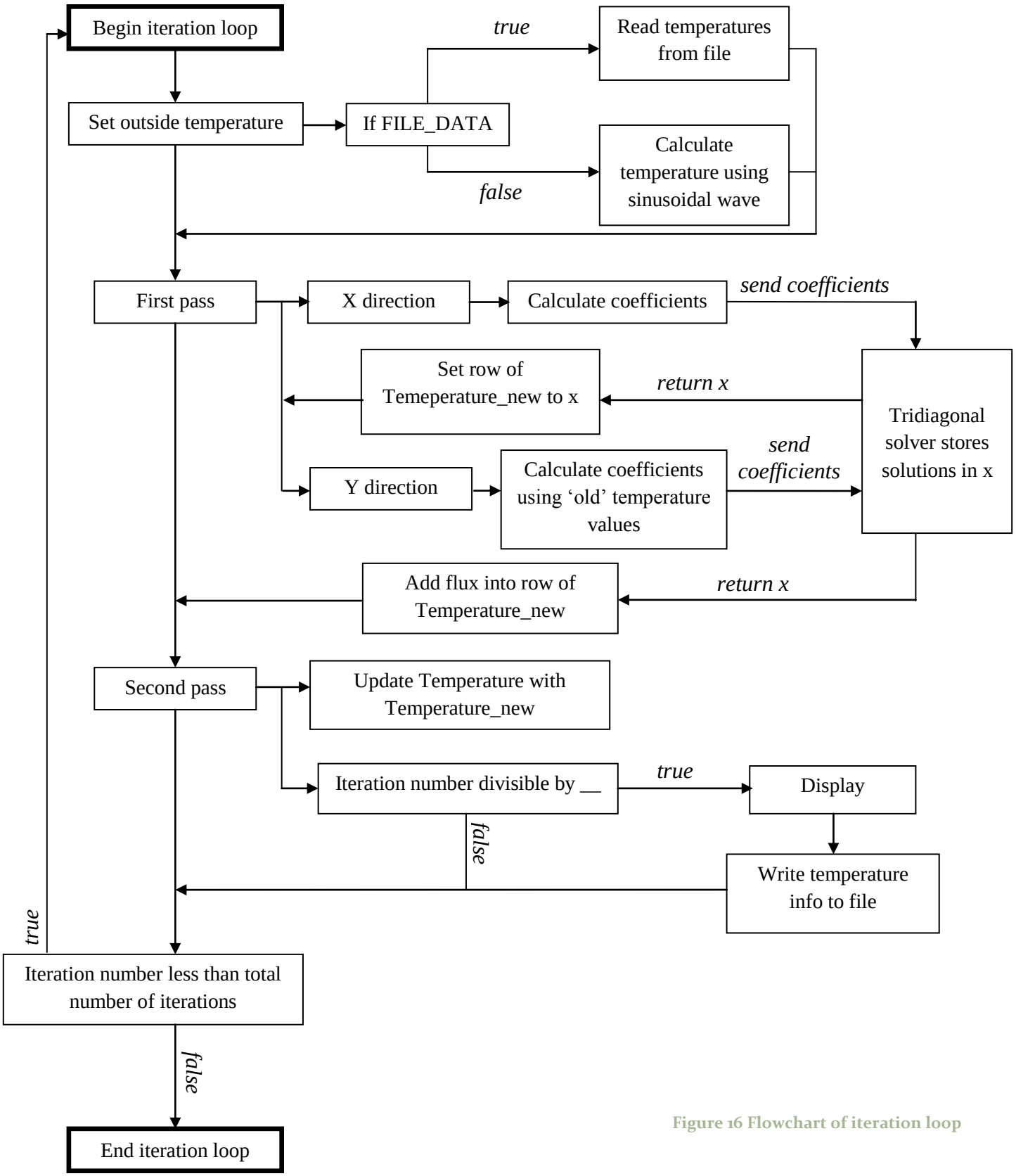

Figure 16 Flowchart of iteration loop

Interactive

We revised the program to take key input to initialize the matrix. The cells are all set to a default outside air, and cells under the cursor in the graphics window are changed to other materials by pressing a corresponding key. This allows cell size, number, and wall type to be easily changed, quickly and more easily than by changing the code itself.

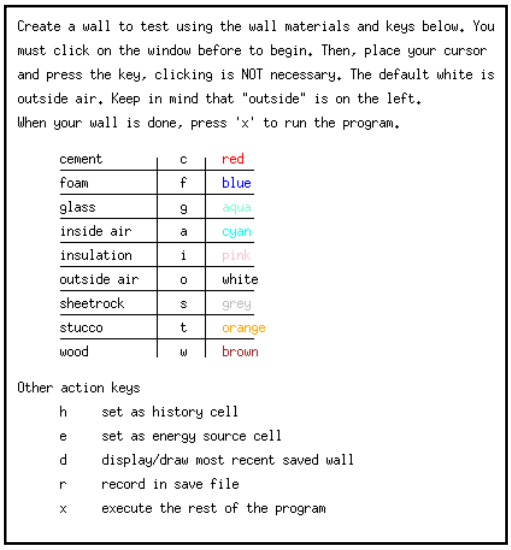

Figure 17 Instruction window from program display

There are other inputs besides those used to set cell materials. The 'x' key executes the rest of the program when it is pressed, and starts the heat transfer simulation through the wall. The 'l' key is used to set a source cell, where a yellow X will be drawn. Once a cell is assigned as a source cell, it cannot be changed back to normal. The 'h' key records the indices of the cell. When the program is run, the temperature of that cell will be written to a history file. Pressing it again will reset the indices. The 'r' key records the wall materials in a save file, and 'd' draws the most recently saved wall and sets all the materials.

Graphics

We spent a fair amount of time on the graphics. We wanted the display to be fairly easy to understand, and a visualization of the results. We also added an instruction window to guide a user through the steps of drawing a wall, and also to make the commands for each key more accessible (Figure 17).

The majority of the window shows the mesh of cells, colored to represent their current temperature. On the right side of the screen we print a scale of the colors and the temperature they correspond to (Figure 18). Below the scale is an updated printout of the present time in the simulation and the outside temperature on either a sun or moon (Figure 19). We thought that this was a good way signify to people what is happening. For example, it should be fairly clear that if there is blue in the window, and there is a moon, that it is a cold night.

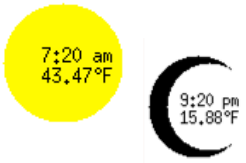

Figure 19 Sun/moon clock and outside temperature from display

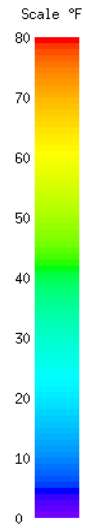

Figure 18 Temperature scale from display window

Results

Verification

"'Verification' ~ solving the equations right (Roache, 1988)." Verification is proving the program solves the equations correctly, and with minimal *mathematical* error. As part of verification, we solved a problem by hand to show that the conduction works properly, and compared the outputs of different time steps and cell sizes.

We invented a small test problem to do by hand and used print statements to follow the values through each section of code. The input is a 1-by-3 mesh of wood and insulation at 60°F, 65°F, and 70°F (Figure 20). To begin, we found the values of each of the coefficients using the properties of the materials (Figure 21). They are listed below in the order they were calculated:

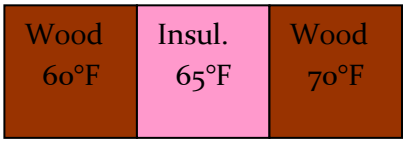

Figure 20 Test problem used in verification of the code

	Wood	Insulation
Conductivity	0.8	0.009636
Density	32	0.85
Specific Heat	0.33	0.2

Figure 21 Table of wood and insulation properties

$$a_0 = 0.0$$
$$a_1 = 0.000007$$
$$a_2 = 0.000007$$
$$c_0 = 0.000007$$
$$c_1 = 0.000007$$
$$c_2 = 0.0$$
$$b_0 = 0.006104$$
$$b_1 = 0.000084$$
$$b_2 = 0.006104$$
$$d_0 = 0.3\overline{6}$$
$$d_1 = 0.006395$$
$$d_2 = 0.42\overline{7}$$

$$c'_0 = 0.001103$$
$$c'_1 = 0.060313$$
$$d'_0 = 59.933830$$
$$d'_1 = 60.778430$$
$$d'_2 = 69.994486$$
$$x_2 = 69.994486$$
$$x_1 = 65.0$$
$$x_0 = 60.005514$$

After this single time step, the temperatures are: 60.005514, 65, and 69.994486 respectively[5]. Our hand calculated values matched with those output from the program, with the exception of small discrepancies caused by round off errors. We set the same problem vertically to check the program in the y direction and found the same results.

[5] Note that these have the same sum as the original temperatures, and therefore can be shown to follow the Law of Conservation of Energy.

Units are necessary to express all quantities and to give numbers physical value. There are several systems that are used in the field of heat transfer, and it is important to consistently use units from one system. We are working in U.S. engineering units. In hear transfer, the fundamental dimensions are time, length, mass, temperature, force, and heat. Derived dimensions are expressed in terms of these fundamental dimensions (Figure 22). In the context of our program, cells are measured in inches, and the time step in minutes.

Physical Quantity	U.S. Engineering Unit
Heat flow rate	BTU/hr
Specific heat	$BTU/lb\ F$
Thermal conductivity	$BTU/hr\ ft\ F$
Heat transfer coefficient	$BTU/hr\ ft^2\ F$
Density	lb_m/ft^3

Figure 22 Table of units

The units for the a_i and c_i coefficients are shown below[6]:

$$\frac{\cancel{in^2}\ \cancel{sec}\ BTU}{\cancel{hr}\ \cancel{in}\ F\ \cancel{in}}\left(\frac{\cancel{hr}}{60\cancel{sec}}\right) = \frac{BTU}{60\ F}$$

b_i should end up in the same units:

$$\frac{BTU\ \cancel{in^3}\ \cancel{lb}}{\cancel{lb}\ F\ \cancel{ft^3}}\left(\frac{\cancel{ft}}{12\cancel{in}}\right)\left(\frac{\cancel{ft}}{12\cancel{in}}\right)\left(\frac{\cancel{ft}}{12\cancel{in}}\right) = \frac{BTU}{1728\ F}$$

d_i is multiplied by the temperature, which leaves it in $BTUs$:

$$\frac{BTU\ \cancel{in^3}\ \cancel{lb_m}\ \cancel{F}}{\cancel{lb}\ \cancel{F}\ \cancel{ft^3}}\left(\frac{\cancel{ft}}{12\cancel{in}}\right)\left(\frac{\cancel{ft}}{12\cancel{in}}\right)\left(\frac{\cancel{ft}}{12\cancel{in}}\right) = \frac{BTU}{1728}$$

Any source terms added to d_i must also be in $BTUs$.

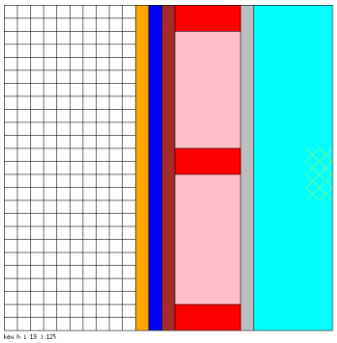

Figure 23 Input wall used for collecting data for comparisons of time steps and cell resolution

In order to assess the error created by using longer time steps, we ran our implicit program with three different time steps. Each run simulated the wall shown in Figure 23 for about seven days, recording the temperature at a given cell every ten minutes. We plotted the temperature output from the runs for each of the time steps in Figure 26. It shows that the shortest time step, 0.01 minutes, has a very defined swing, while the minute long time step is

[6] Conductivity has already been input to the program in Btu/hr in F, so no conversion from feet into inches is necessary

indistinct. This is because the bigger the time we are calculating for, the less detail it can pick up. The run times, shown in Figure 24, increase by roughly a factor of ten. This makes sense, as the time steps are decreasing at the same rate. A longer time step should moderate the fluctuations, and be able to come close to the average, which the plot confirms.

We also plotted the results of the runs with different cell dimensions (Figure 25). We are not sure what caused these results, and whether there was an error in the code. The two inch cells recorded generally higher temperatures, and the oscillation seems to decrease with a finer resolution. As the size of the cell is cut in half, the run time is about four times as long (Figure 24). This is reasonable, as it takes four cells to make one of the next size.

	Time Step			Cell Size		
	1.0 min	0.1 min	0.01 min	2in	1in	0.5in
Run Time	2 sec	20 sec	196 sec	5 sec	18 sec	72 sec
Average Temperature	37.12°F	36.15°F	41.58°F	41.10°F	36.55°F	36.16°F

Figure 24 Results from comparisons of time steps and cell size

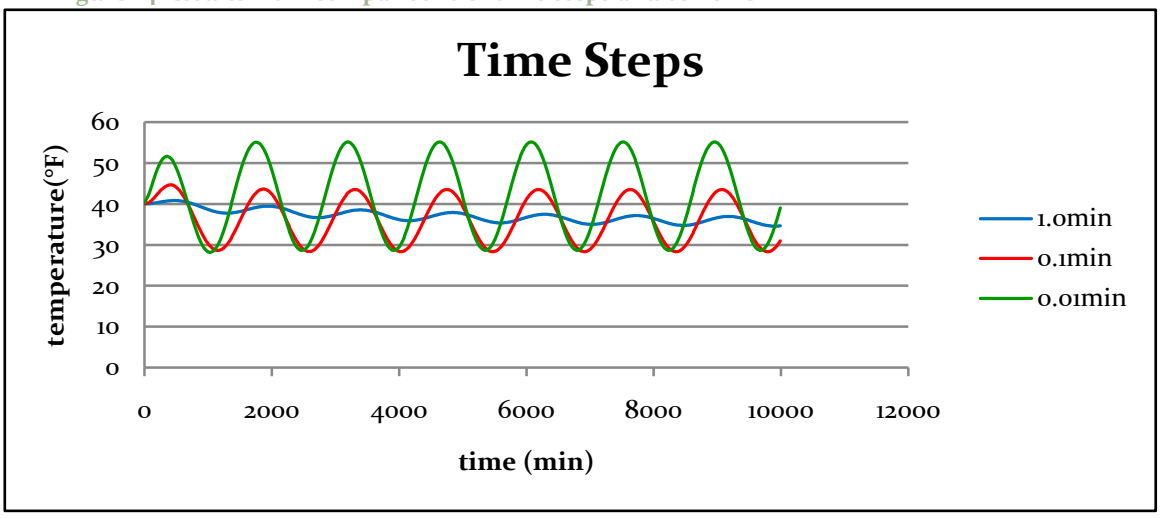

Figure 26 Plot of output from time step comparison

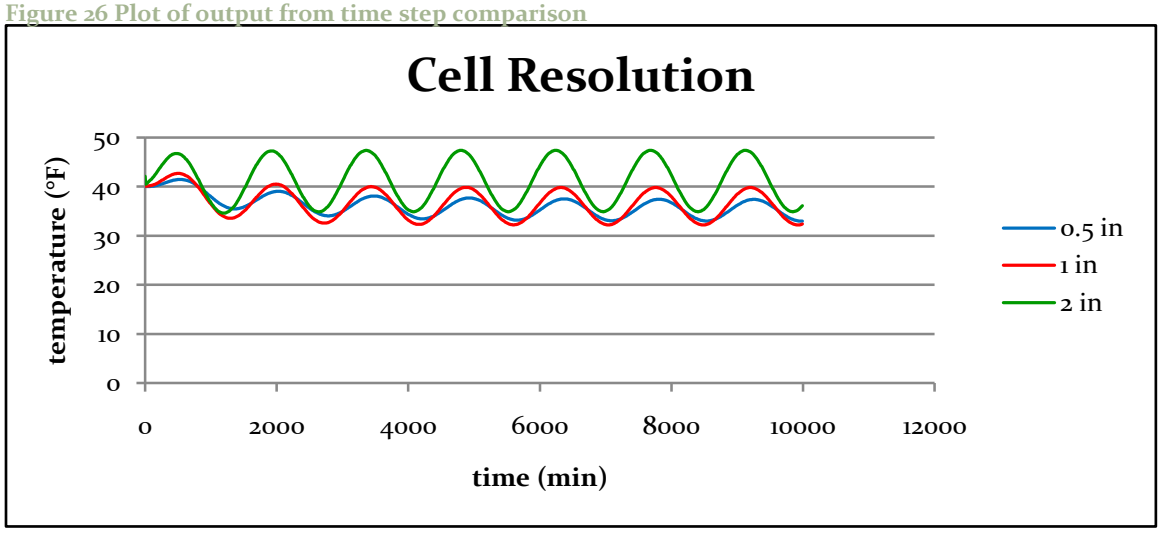

Figure 25 Plot of output from cell size comparison

Validation

"'Validation' ~ solving the right equations (Peaceman & Rachford, 1955)." Validation is the ability of our model to produce answers for intended applications that fall within an acceptable range of accuracy.

We tried to match the conditions in the experimental and computer model as closely as possible so we could compare the results to determine how closely our model comes to real life (Figure 27). We read in the test two experimental outside temperatures from a file to control the outside boundary. The wall we input to the program as an imitation of our experimental wall is shown in Figure 27. When running our program, we discovered a problem with the energy cells. An input of 25 watts did not heat the interior as we expected. We are uncertain whether this is caused by wrong energy input or an error in the method, as we did not have time to look into it.

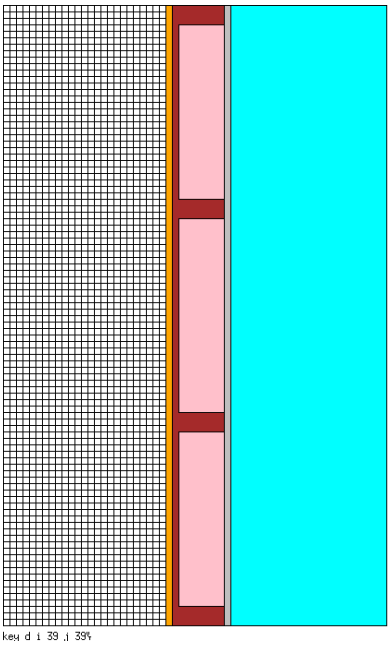

Figure 27 Imitation of experimental wall used as input for validation

Condition	Descriptions/Adjustments
Internal heat source	We put a light bulb inside t simulate the indoor heating. A steady input of 25 watts to generate enough energy to get close to comfortable temperatures
No solar gain	We placed the box in an area which is shady most of the day.
No internal radiation	The rack held tin foil between the light bulb and data logger to minimize direct radiation
Wind is irregular	Tested both with outside data logger exposed and in a plastic bag.
One wall	The rest of the box is over insulated to reduce the influence as much as possible and isolate the test to one test wall
Air leakage	The corners are sealed up with foam on the inside, and duck tape on the outside to prevent as much air leakage as possible.

Figure 28 Adjustments made in order to match computer and experimental model

We estimated the energy input through our observations of the effects of different amounts, and ran the program. The plot of the temperatures from the computer for inside and outside[7] is compared to the experimental results in Figure 29 below.

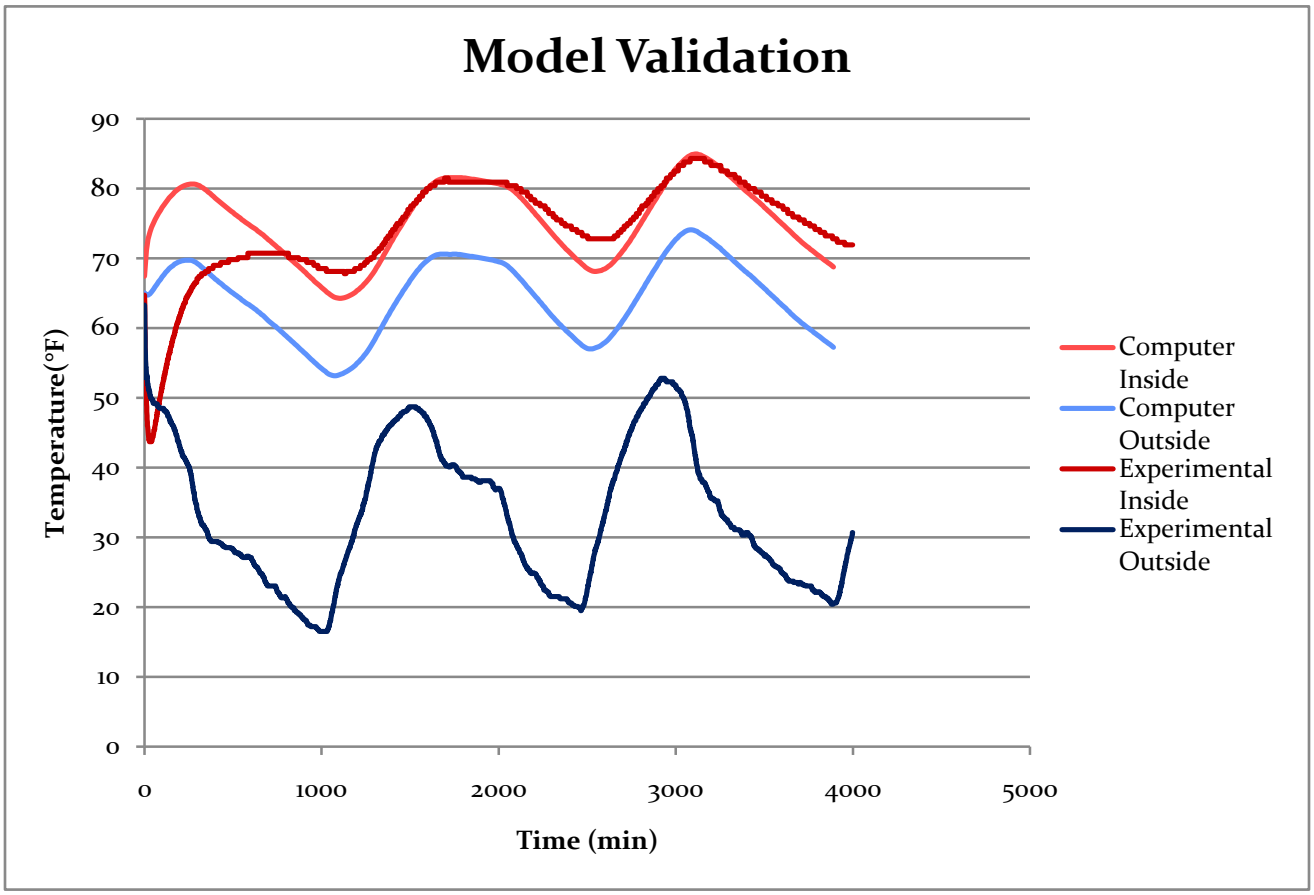

Figure 29 Validation comparison of experimental and computational results

We were very impressed by these results, as there was a fair amount of guess work done at this point. The starting conditions cause the discrepancy in the beginning of the graph, so this was ignored for the most part. The inside results match each other remarkably well, both in shape and in general temperature range. The outside temperatures are much farther apart, and while the computer model has the same general shape, it did not capture some of the more subtle changes.

[7] A history cell was set about where we estimated the data logger would be and on the outside boundary.

Conclusions

Model Capabilities

Our model was successful. The verification process found most, if not all, of the bugs in the conduction part of our code. When it tested walls its answers were reasonable. There are clearly still some bugs though, as we saw while trying to use a source of energy in the validation. The outside temperature may also be influenced by factors in the model, and more air space may be needed between the wall and boundary during validation runs. In summary, while there is still debugging to be done, our model is acceptable.

The display and key input improvements we were able to make were far beyond our original expectations. The input window works well and refreshes smoothly, and the instruction window is a good guide and reference. The display is very readable and provides a real-time display of temperatures.

Skills

As a result of this project, both of us gained many skills that we will continue to find useful in the future. This was Gabe's first year in the Challenge, and he learned a lot about the process of computer modeling. He started working with C, and hopes to learn more about programming this summer. After first starting to program last year, Rachel greatly expanded her understanding of programming, computational models and the process in general.

Teamwork

Our team consisted of two members this year. Rachel had worked on the project last year with Jessie. Jessie was unable to participate this year, so Gabe took her place. He has done an excellent job of stepping into the middle of the project.

It was difficult for us to meet regularly, on account of activities, and Gabe living farther away. We were still able to work together on some weekends and occasionally after school. We were generally very productive at these meetings, but also did a fairly good job of working on our own in between. We were able to talk to each other when at school and keep one another updated, or transfer information with a flash drive.

Gabe was responsible for most of the building and testing of our experimental model. We came up with some of the basic concepts together, but he did the more detailed design of how to build it, and chose and bought the materials. We met at his house on a weekend in early January to put together the sides of the box. Gabe had broken his arm and was somewhat incapacitated. He and his dad later finished building it and getting it ready for testing. Gabe tested it on their property and downloaded all the data for each test. He also did most of the photography of the box, which was a helpful record of the set-up.

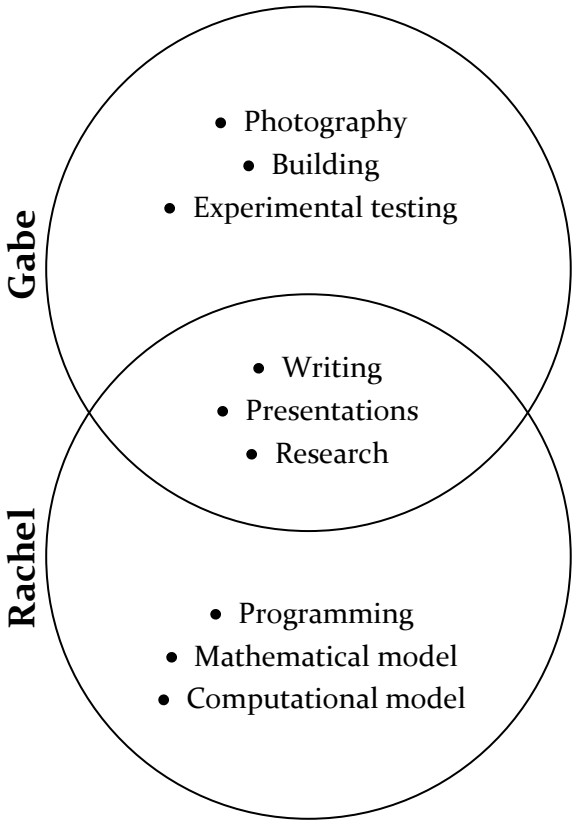

Figure 30 Teamwork Venn diagram

Rachel was in charge of the programming, including the mathematical and computational models. Gabe helped some with the programming, but it was a learning experience for him as he had never programmed before.

We have both worked together on most of the reports and presentations. We would try to meet more often when reports were due, and especially to create and practice presentations. Overall, we worked well together, and split up the work load efficiently.

Recommendations

Model

Our program has an incredible potential, and there are many ways to enhance and improve it. There are several in particular we considered would be useful, and would probably pursue in any continuing work on this project. Parallelizing our code has always been an option, but not realistic because there wasn't enough work for multiple processors and not worth it. Now, however, as we improve our program, there is more work, and it would be beneficial. The possibilities to add onto the interactive, display, and user interface part of our model are endless, and not the purpose of a supercomputing project. Even so, we are interested in adding rubber banding[8] to the interactive cell initialization. This would be very practical when trying to assign large numbers of small cells. We did make an attempt to do this, but we don't have any experience with x-windows, and even with help had trouble capturing and returning the location of the mouse button click's subsequent release. We also would like to go through and validate every part of the code. For example we would check that each material behaved the way it should. This would help us to understand the differences between experimental and computational models. We would also go through and solve the problems with the "light bulb" energy source. Even though we could do a lot more with this program, we did an amazing amount of coding and were able to accomplish most of our goals.

Wall Design

Writing the code and doing experimental tests in order to create a functional and valid program required the whole year. We ran out of time for testing energy efficient walls, and so were unable to analyze these designs. The next step would most likely be beginning to test these walls, and perhaps adding in components such as window, roof, floors, doors, etc. in order to expand the walls the program is capable of testing.

[8] Moving of object where one end is fixed in position. In our case this would be selecting a block of cells to assign with a click and release of the mouse button.

Appendix

Bibliography

Alme, M., Vold, E., Yilk, T., & Robey, B. (July 9, 2008). *Climate Modeling and Global Warming Simulation.* Los Alamos, NM: X/CCS Division Summer Workshop.

Ashrae Handbook 1977 Fundamentals. (1977). New York, New York: American Society of Heating, Refrigerating, and Air Conditioning Engineers, Inc.

Cooling and Heating: Load Calculation Manual. (1979). New York: American Society of Heating, Refrigerating, and Air Conditioning Engineers, Inc.

Energy Information Administration. (2001). *Space-Heating Energy Consumption in U.S. Households by Year of Construction*. Retrieved March 28, 2009, from www.eia.doe.gov/emeu/recs/recs2001/ce_pdf/spaceheat/ce2-2c_construction2001.pdf

Faires, V. M. (1970). *Thermodynamics* (5th ed.). New York, New York: Macmillan Publishing Co., Inc.

Iowa State University. (1992, March). *Building Energy Efficient New Houses.* Retrieved from University Extension: http://www.forestry.iastate.edu/publications/PM790.pdf

Kreith, F. (1973). *Priciples of Heat Transfer* (3rd ed.). New York, New York: Intext Educational Publishers.

Kreith, F., & Kreider, J. F. (1978). *Principles of Solar Engineering.* USA: Hemisphere Publishing Corporation.

Lemieux, D. J., & Totten, P. E. (2007, February 1). *Building Envelope Design Guide - Wall Systems.* Retrieved from Wiss, Jannev, Elstnew Associates, Inc. Whole Building Design Guide: http://wbdg.org/design/env_wall.php

Peaceman, D., & Rachford, J. H. (1955, March). The Numerical Solution of Parabolic and Elliptic Differential Equations. *vol. 3* . USA: J. Soc. Indust. Appl. Math.

Revelle, E., & Galland, E. (2008, September). *Understanding Energy Consumption*. Retrieved March 28, 2009, from revelle.net/lakeside/lakeside.new/understanding.html

Roache, P. J. (1988). *Verification and Validation in Computational Science and Engineering.* USA: Hermosa Publishers.

Robey, B. (2007, October). *Shallow Water Workshop*. Retrieved from www.challenge.nm.org/archive/07-08/kickoff/classes/#experienced

Robey, J., Holland, A., Jacobs, L., & Shlachter, D. (2008). *Modeling Spacecraft Reentry.*

Robey, R., & Bohn, J. (2008). *Turn Up the Heat: Energy Efficiency through Smart Wall Design.*

Tridiagonal Matrix Algorithm. (2008, November 27). Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm

Web Pages for MPI and MPE. (2004, August 4). Retrieved from Argonne National Laboratory Mathematics and Computer Science Divison: www.mcs.anl.gov/research/projects/mpi/www

Acknowledgements

Our mentors, Bob Robey and Derrick Montoya, for their ideas, help, and support to finish
Tom Laub, Larry Kilham, and Kathy Pallis, for reviewing our project at the evaluations
Our teacher, Bob Dryja
Jonathan Robey, for his help with math and programming questions
Victor Kuhns, Dale Henderson, for reviewing our proposal and interim and their suggestions

Source Code

```
#define WAIT_TIME   0    // Slows down run if too fast
#define DEBUG 0          // Turn on debug statements
#define DISPLAY_ON 100   // Turns on output and sets iterations between plots
#define FILE_DATA 1      // Chose sinusoidal wave or data file for outside
#define max(a, b) ((a) > (b) ? (a) : (b))

/**************************************
 * HEAT -- 2D Heat Transfer Model
 * Rachel Robey, Los Alamos Middle School
 * Copyright 2007-2009
 **************************************/

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <unistd.h>
#include <time.h>
#include "heat2.h"

#ifdef DMALLOC
#include "dmalloc.h"
#endif

#define Pi 3.14

/* Display routines */
void display_init(char *displayname, int matrix_size_x, int matrix_size_y,
      int iheight);
void display_instructions(void);
char get_key(int *i, int *j, int my_offset, int matrix_size_x, int
      matrix_size_y);
void display_one_d(int matrix_size_y, int matrix_size_x, double **temp, int
      my_offset,int mysize, double maxscale, double time, double Temp_max,
      double Temp_min, int **Material);
void display_setup(int matrix_size_x, int matrix_size_y, int **Color, int
      my_offset,int mysize, double time, double Temp_max, double Temp_min,
      int **Source);
void display_colors(void);
void set_label(char *text);
void display_close(void);

/* Calculation routines */
void TridiagonalSolve(const double *a, const double *b, double *c, double *d,
double *x, unsigned int n);
double source_data(double time);

/* Memory allocation routines */
double *dvector(int n);
```

```
double **dmatrix(int m, int n);
int **imatrix(int m, int n);

double **Mass, **Temperature, **Temperature_new, **Source_radiation,
      **Energy;                  //state variables
int **Source;                    //source cells - light bulb
double *a, *b, *c, *d, *x;       //tridiagonal solver coefficients
int **Material, **Color;;        //cell materials and colors

int main(int argc, char *argv[]) {
   int rank, size;
   int next, prev;
   int i, j, k, l;
   int matrix_size_x, matrix_size_y;
   int ntimes;
   int n;
   int mysize;
   int my_offset;
   char keyComm;
   double Energy_added;
   double deltat = 0.1;                     //hardwired timestep - minutes
   double deltax = 0.5, deltay = 0.5, deltaz = 1;  //size of cell - inches
   double maxScale;                         //display color scaling
   double time = 0.0;                       //computer simulation time
   double time1, time2, temp1, temp2;       //for reading temp info from file
   double totaltime, starttime;             // to calculate program run time
   double myTE, TotalEnergy, origTE;        //for checking conserve. of energy
   int ihistory = -1, jhistory = -1;        //indices of temp log
   double Temp_max=60, Temp_min=10;         //temp swing max and min
   FILE *fhistory, *fdata, *fsave;          //declare files
   char *desc;                              //variable for labels
   char string[80], numbers[80];
   char *displayname = ":0";

   /* Material properties */
   enum material{OUTSIDE_AIR, INSIDE_AIR, STUCCO, FOAM, WOOD, CEMENT, GLASS,
      INSULATION, SHEETROCK};
   double Material_density[9]= { .076, .076, 116., 2.2, 32., 64.65, 80, .85,
      50. };
      //density in pounds/ft cubed
   double Material_specific_heat[9]= { .24, .24, .22, .29, .33, .22, .22, .2,
      .26 };
      //specific heat in Btu/pound*mass*Farenheit
   double Material_conductivity[9]= { 6.0, 1.46, .5, .2, .8, .96, .32,
      .053/5.5, 1.78/.625 };
      //conductivity in Btu/hour*inch*Farenheit, 15mph wind factored into
      outside air
   int Material_color[9]={0, 5, 10, 6, 12, 2, 8, 13, 15};
      //white, cyan, orange, blue, brown, red, aqua, pink, grey

   MPI_Init(&argc, &argv);
   //Determine size and my rank in MPI_COMM_WORLD communicator
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (argc > 2 && strcmp(argv[1], "-display") == 0) {
      displayname = (char *)malloc(strlen(argv[2]) + 1);
      strcpy(displayname, argv[2]);
```

```
}

if (rank == 0)
   printf("Copyright 2008\n");
sleep(WAIT_TIME);

/* Determine the matrix sizes and # of iterations */
if (rank == 0) {
   /*
   printf("Matrix Size X : ");
   scanf("%d",&matrix_size_x);
   printf("Matrix Size Y : ");
   scanf("%d",&matrix_size_y);
   printf("Iterations : ") ;
   scanf("%d",&ntimes);
   */
   matrix_size_x = 96;
   matrix_size_y = 96;
   ntimes = 39000;
}

if (DISPLAY_ON) {
   display_init(displayname, matrix_size_x, matrix_size_y, 700);
   display_instructions();
}

//Broadcast the size and # of itertations to all processes
MPI_Bcast(&matrix_size_x, 1, MPI_INT, 0, MPI_COMM_WORLD) ;
MPI_Bcast(&matrix_size_y, 1, MPI_INT, 0, MPI_COMM_WORLD) ;
MPI_Bcast(&ntimes, 1, MPI_INT, 0, MPI_COMM_WORLD);
//Set neighbors
if (rank == 0)
   prev = MPI_PROC_NULL;
else
   prev = rank-1;
if (rank == size - 1)
   next = MPI_PROC_NULL;
else
   next = rank+1;
mysize = matrix_size_y/size + ((rank < (matrix_size_y % size)) ? 1 : 0 );
my_offset = rank * (matrix_size_y/size);
if (rank > (matrix_size_y % size))
   my_offset += (matrix_size_y % size);
else
   my_offset += rank;
if (DEBUG)
   printf("my rank is %d and mysize is %d\n", rank, mysize);

/* Allocate the memory dynamically for the matrix */
Mass = dmatrix(matrix_size_y, matrix_size_x);
Energy = dmatrix(matrix_size_y, matrix_size_x);
Temperature_new = dmatrix(matrix_size_y, matrix_size_x);
Source_radiation = dmatrix(matrix_size_y, matrix_size_x);
Temperature = dmatrix(matrix_size_y, matrix_size_x);
Material = imatrix(matrix_size_y, matrix_size_x);
Source = imatrix(matrix_size_y, matrix_size_x);
Color = imatrix(matrix_size_y, matrix_size_x);
```

```
if (rank == 0 && DEBUG)
   printf("Memory allocated\n");

for (j = 0; j < matrix_size_y; j++) {
   for (i = 0; i < matrix_size_x; i++) {
      Material[j][i] = OUTSIDE_AIR;
      Source[j][i] = 0;
   }
}

/* Initialize matrix */
keyComm = '\0';
while (keyComm != 'x') {
   keyComm = get_key(&i, &j, my_offset, matrix_size_x, matrix_size_y);
   if (keyComm != '\0') {
      sprintf(string, "key %c i %d j %d\n", keyComm, i, j);
         printf("key %c i %d j %d\n", keyComm, i, j);
         if (keyComm == 'w') {
            Material[j][i] = WOOD;
            Color[j][i]= (Material_color[ Material[j][i] ]);
         }
         if (keyComm == 'f') {
            Material[j][i] = FOAM;
            Color[j][i] = (Material_color[ Material[j][i] ]);
         }
         if (keyComm == 's') {
            Material[j][i] = SHEETROCK;
            Color[j][i] = (Material_color[ Material[j][i] ]);
         }
         if (keyComm == 'i') {
            Material[j][i] = INSULATION;
            Color[j][i] = (Material_color[ Material[j][i] ]);
         }
         if (keyComm == 't') {
            Material[j][i] = STUCCO;
            Color[j][i] = (Material_color[ Material[j][i] ]);
         }
         if (keyComm == 'o') {
            Material[j][i] = OUTSIDE_AIR;
            Color[j][i] = (Material_color[ Material[j][i] ]);
         }
         if (keyComm == 'a') {
            Material[j][i] = INSIDE_AIR;
            Color[j][i] = (Material_color[ Material[j][i] ]);
         }
         if (keyComm == 'c') {
            Material[j][i] = CEMENT;
            Color[j][i] = (Material_color[ Material[j][i] ]);
         }
         if (keyComm == 'g') {
            Material[j][i] = GLASS;
            Color[j][i] = (Material_color[ Material[j][i] ]);
         }
         if( keyComm == 'e' ){
            Source[j][i] = 1;
         }
         if( keyComm == 'h' ){
```

```
            ihistory = i;
            jhistory = j;
         }
         if( keyComm == 'r' ){     //save wall
                  if( ( fsave = fopen("wall", "wb") ) == NULL ) {
                        printf("Error--save file missing\n");
                        exit(0);
                  }
                  for(k = 0; k < matrix_size_y; k++){
                        if(fwrite(Material[k], sizeof(int), matrix_size_x,
                           fsave)< 1) {
                              printf("Write error occured.\n");
                              exit(0);
                        }
                  }
                  fclose(fsave);
         }
         if( keyComm == 'd' ){      //open saved wall
            if( ( fsave = fopen("wall", "rb") ) == NULL) {
               printf("Error--cannot open save file\n");
               exit(0);
            }
            for(k = 0; k < matrix_size_y; k++){
               if( (fread(Material[k], sizeof(int), matrix_size_x, fsave))
                     < 1){
                 printf("Read error occured\n");
                 exit(0);
               }
               for(l = 0; l < matrix_size_x; l++){
                 Color[k][l] = (Material_color[ Material[k][l] ]);
               }
            }
            fclose(fsave);
         }
      if (DISPLAY_ON) {
         set_label(string);
         display_setup(matrix_size_x, matrix_size_y, Color, my_offset,
         mysize, time, Temp_max, Temp_min, Source);
      }
   }
}
display_colors();

/* Set initial temperatures */
for (j = 0; j < matrix_size_y; j++) {
   for (i = 0; i <= 0; i++)
      Temperature[j][i] = 60.0;
}
for (j = 0; j < matrix_size_y; j++) {
   for (i = 1; i < matrix_size_x-1; i++)
      Temperature[j][i] = 65.0;
}
for (j = 0; j < matrix_size_y; j++) {
   for (i = matrix_size_x-1; i < matrix_size_x; i++)
      Temperature[j][i] = 70.0;
}
```

```
/* Initialize cell properties */
for (j = 0; j < matrix_size_y; j++) {
   for (i = 0; i < matrix_size_x; i++) {
      Mass[j][i] = (Material_density[ Material[j][i] ]* deltax * deltay *
         deltaz) / 1728.0;
         //multiply by 1 foot/12 inches three times, becomes pounds,mass
      Energy[j][i] = (Material_specific_heat[ Material[j][i] ]) *
         Temperature[j][i];
      Source_radiation[j][i] = 0.0;
         //no source term yet -- must be in BTUs
   }
}
if (rank == 0 && DEBUG)
   printf("initial values set\n");

myTE = 0.0;
for (j = 0; j < matrix_size_y; j++) {
   for (i = 0; i < matrix_size_x; i++)
      myTE += Energy[j][i];
}

MPI_Allreduce(&myTE, &origTE, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
if (rank == 0 && DEBUG)
   printf("initial values displayed\n");

if( ihistory != -1 && jhistory != -1 ) {
      fhistory = fopen("history", "w");     //open write file
}

if( FILE_DATA ) {
  if( ( fdata = fopen("input.csv", "r") ) == NULL) {
        printf("Error -- input file missing");
      exit(0);
  }
  time2 = -100.0;
  temp2 = 0;
}
starttime = clock()/CLOCKS_PER_SEC;

/* Begin the iteration loop */
for (n = 0; n < ntimes; n++) {
      if( FILE_DATA ) {
        while( time2 < time){
              time1 = time2;
              temp1 = temp2;
              fgets( numbers, 80, fdata );
              sscanf( numbers, "%lf, %lf", &time2, &temp2 );
              //printf( "%lf, %lf, %lf\n", time, time2, temp2 );
      }
   }
   /*
   MPI_Request req[8];
   MPI_Status status[8];
   //Send and receive boundary information
   MPI_Isend(Energy[1], matrix_size_x+2, MPI_DOUBLE, prev, 1,
   MPI_COMM_WORLD, req  );
```

```
MPI_Irecv(Energy[mysize+1], matrix_size_x+2, MPI_DOUBLE, next, 1,
MPI_COMM_WORLD, req+1);
MPI_Isend(Energy[mysize], matrix_size_x+2, MPI_DOUBLE, next, 2,
MPI_COMM_WORLD, req+2);
MPI_Irecv(Energy[0], matrix_size_x+2, MPI_DOUBLE, prev, 2,
MPI_COMM_WORLD, req+3);
if (rank == 0 && DEBUG)
   printf("values for energy communicated\n");
MPI_Isend(Temperature[1       ], matrix_size_x+2, MPI_DOUBLE, prev, 5,
MPI_COMM_WORLD, req+4);
MPI_Irecv(Temperature[mysize+1], matrix_size_x+2, MPI_DOUBLE, next, 5,
MPI_COMM_WORLD, req+5);
MPI_Isend(Temperature[mysize  ], matrix_size_x+2, MPI_DOUBLE, next, 6,
MPI_COMM_WORLD, req+6);
MPI_Irecv(Temperature[0       ], matrix_size_x+2, MPI_DOUBLE, prev, 6,
MPI_COMM_WORLD, req+7);
if (rank == 0 && DEBUG)
   printf("values for temperature communicated\n");
MPI_Waitall(8, req, status);
if (rank == 0 && DEBUG)
   printf("Communication successful\n");
*/

/* Set outside temperature */
for (j = 0; j < matrix_size_y; j++) {
  if( FILE_DATA ) {
      Temperature[j][0]= temp1 + (temp2-temp1)/(time2-time1)*( time -
      time1 );
  }
  else {
      Temperature[j][0] = ( ( (Temp_max + Temp_min)/2) )+( (Temp_max -
      Temp_min)/2 )*( sin( (time + 360) * Pi/720 ) );
  }
}

if (rank == 0 && DEBUG)
   printf("Boundary conditions set\n");

if (rank == 0 && DEBUG)
   printf("Before 1st pass\n");

/* First pass */
/* x direction */
a = (double *) malloc(sizeof(double) * max(matrix_size_x,
matrix_size_y));
b = (double *) malloc(sizeof(double) * max(matrix_size_x,
matrix_size_y));
c = (double *) malloc(sizeof(double) * max(matrix_size_x,
matrix_size_y));
d = (double *) malloc(sizeof(double) * max(matrix_size_x,
matrix_size_y));
x = (double *) malloc(sizeof(double) * max(matrix_size_x,
matrix_size_y));
for (j = 0; j < matrix_size_y; j++) {
   a[0] = 0.0;                              //boundary condition
   c[matrix_size_x-1] = 0.0;                //boundary condition
   for (i = 1; i < matrix_size_x; i++) {
```

```
      a[i] = -((Material_conductivity[Material[j][i]] +
      Material_conductivity[Material[j][i-1]])/2) * deltay * deltaz *
      deltat / deltax / 60.0;
         //average of material conductivity multiplied by the area
      multiplied mulitiplied by time step divided by distance between
      cells, unit conversion -- 60 sec / hr
   }
   for (i = 1; i <= matrix_size_x-1; i++) {
      c[i-1] = a[i];
   }

   for (i = 0; i < matrix_size_x; i++) {
       if(Source[j][i]==1){
            Energy_added = (1.42 / 8.0 * deltat) /
      Material_specific_heat[ Material[j][i] ]/150;
          //1.42 BTU/minute = 25 watts, division as needed
       }
       else {
           Energy_added = 0.0;
       }
      b[i] = ( Material_specific_heat[ Material[j][i] ] * deltax *
      deltay * deltaz * Material_density[ Material[j][i] ] / 1728.0 ) -
      a[i] - c[i];
         //unit conversion -- ft cubed / 1728 in cubed, also below
      d[i] = ( Material_specific_heat[ Material[j][i] ] * deltax *
      deltay * deltaz* Material_density[ Material[j][i]] / 1728.0 *
      Temperature[j][i]) + ( Source_radiation[j][i] * deltat ) +
      Energy_added;
         //BTU/F, cancelled when multiplied by temperature
   }
   TridiagonalSolve(a, b, c, d, x, matrix_size_x);  //set solution to x
   for (i = 0; i < matrix_size_x; i++)
      Temperature_new[j][i] = x[i];   //set row of Temperature_new to x
}

if (rank == 0 && DEBUG)
   printf("First pass x direction complete\n");

/* y direction */
for (i = 0; i < matrix_size_x; i++) {
   a[0] = 0.0;                          //boundary conditions
 c[matrix_size_y-1] = 0.0;              //boundary conditions
   for (j = 1; j < matrix_size_y; j++) {
      a[j] = -((Material_conductivity[ Material[j][i] ] +
      Material_conductivity[ Material[j-1][i] ])/2)* deltay * deltaz *
      deltat / deltax / 60.0;
         //average of material conductivity multiplied by the area
         //multiplied mulitiplied by time step divided by distance
         //between cells, unit conversion -- 60 sec / hr
   }
   for (j = 1; j < matrix_size_y; j++) {
      c[j-1] = a[j];
   }
   for(j = 0; j < matrix_size_y; j++) {
       if(Source[j][i]==1){
            Energy_added = (1.42 / 8.0 * deltat) /
            Material_specific_heat[ Material[j][i] ]/150;
```

```
            //1.42 BTU/minute = 25 watts, division as needed
          }
          else {
              Energy_added = 0.0;
          }
         b[j] = ( Material_specific_heat[ Material[j][i] ] * deltax *
         deltay * deltaz * Material_density[ Material[j][i]] / 1728.0 ) -
         a[j] - c[j];
         d[j] = ( Material_specific_heat[ Material[j][i] ] * deltax *
         deltay * deltaz * Material_density[ Material[j][i]] / 1728.0 *
         Temperature[j][i]) + (Source_radiation[j][i] * deltat) +
         Energy_added;
            //see x direction
      }
      TridiagonalSolve(a, b, c, d, x, matrix_size_y);  //solution in x
      for(j = 0; j < matrix_size_y; j++) {
         Temperature_new[j][i] += x[j] - Temperature[j][i];
         //adding change to Temperature_new
      }
   }
   if (rank == 0 && DEBUG)
      printf("First pass complete\n");

   /* Second pass */
   if (rank == 0 && DEBUG)
      printf("Second Pass started\n");

   for (j = 0; j < matrix_size_y; j++) {
      for (i = 0; i < matrix_size_x; i++) {
         Energy[j][i] = Material_specific_heat[ Material[j][i] ] *
         Temperature_new[j][i];
         Temperature[j][i] = Temperature_new[j][i];
      }
   }
   if (rank == 0 && DEBUG)
      printf("Second pass complete\n");
   if (rank == 0 && DEBUG)
      printf("Done calculations\n");
   time += deltat;         //increase time by time step

   if (DEBUG) {
      for (j = 0; j < matrix_size_y; j++) {
         for (i = 0; i < matrix_size_x; i++) {
            printf("end of cycle %d %d %lf %lf\n", i, j, Energy[j][i],
            Temperature[j][i]);
         }
      }
   }
   if (DISPLAY_ON) {
      maxScale = 85.0;
      desc = "Temperature";
      sprintf(string, "%s     iter %d      time %.2lf", desc, n, time);
      set_label(string);
      if (n%DISPLAY_ON == 0) {
         display_one_d(matrix_size_x, matrix_size_y, Temperature,
         my_offset, mysize, maxScale, time, Temp_max, Temp_min, Material);
      }
```

```
      }
      if( ihistory != -1 && jhistory != -1 && n % 100 == 0 )
        fprintf(fhistory, "%g, %g\n", time, Temperature[jhistory][ihistory]);

      myTE = 0.0;
      TotalEnergy = 0.0;
      for (j = 0; j < matrix_size_y; j++) {
         for (i = 0; i < matrix_size_x; i++) {
            if (isnan(Energy[j][i])) {
               printf("Error -- Energy[%d][%d]=%f\n", i, j, Energy[j][i]);
            }
         myTE += Energy[j][i] * deltax * deltay * deltaz;
         }
      }
      MPI_Allreduce(&myTE, &TotalEnergy, 1, MPI_DOUBLE, MPI_SUM,
      MPI_COMM_WORLD);

      /* Print iteration information */
      if (DISPLAY_ON && n%DISPLAY_ON == 0) {
         if (rank == 0) {
            printf ( "Iteration:%5.5d, Time:%f\n", n, time);
         }
      }
   }
/* End of iteration loop */

   totaltime = clock()/CLOCKS_PER_SEC - starttime;
   printf ("[%d] Flow finished in %lf seconds\n", rank,
      totaltime/(double)size);

   if( ihistory != -1 && jhistory != -1)
         fclose(fhistory);
   if (DISPLAY_ON)
      display_close();
   MPI_Finalize();
   //dmalloc_shutdown();
   exit(0);
}
```

DISPLAY

```
#define MPE_INTERNAL
#define MPE_GRAPHICS
#include <mpi.h>
#include "mpe.h"
#include <X11/Xlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <termios.h>
#include <time.h>
#include "heat2.h"

#define Pi 3.14

char Iget_key_press(int *xpos, int *ypos);
char get_key(int *i, int *j, int my_offset, int matrix_size_x, int
      matrix_size_y);
void display_scale(double maxscale, double x1, double xsize, double y1,
      double ysize);
void display_strings( char **strings, int size, int xloc, int yloc, MPE_Color
      color );
void colored_strings( char **strings, int size, int xloc, int yloc, MPE_Color
      *color );
int width = 750;
int height = 750;
struct tm time_structure;
time_t epochtime;           //seconds
MPE_XGraph graph;
MPE_XGraph graph2;
MPE_Color *color_array;
int ncolors = 256;
char *label;
char string[20];
int first = 1;
int i;

void display_init(char *displayname, int matrix_size_x, int matrix_size_y,
int iheight) {
   int rank;
   int ncolors2;

   /* Open the graphics display */
   width = matrix_size_x * ( iheight / matrix_size_y);
   height = iheight - iheight % matrix_size_y;    //evenly divided

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPE_Open_graphics( &graph, MPI_COMM_WORLD, displayname, -1, -1, width+80,
      height+18, 0 );
   MPE_Num_colors( graph, &ncolors2 );
   MPE_Draw_string( graph, 10, 20, MPE_BLACK, "Draw your wall here." );
   MPE_Open_graphics( &graph2, MPI_COMM_WORLD, displayname, -1, -1, 415, 445,
      0);
   MPE_Num_colors( graph2, &ncolors2 );

   /* time_structure to begin at ___ */
```

```
   time_structure.tm_year = 2009 - 1900;
   time_structure.tm_mon = 1;
   time_structure.tm_mday = 1;
   time_structure.tm_hour = 14;
   time_structure.tm_min = 23;
   time_structure.tm_sec = 0;
   time_structure.tm_isdst = 0;
   epochtime = mktime(&time_structure);
}
void display_instructions(void) {
   /* Print instruction window */
   char *string[5] = {"Create a wall to test using the wall materials and
      keys below. You", "must click on the window before to begin. Then,
      place your cursor", "and press the key, clicking is NOT necessary. The
      default white is", "outside air. Keep in mind that \"outside\" is on
      the left.", "When your wall is done, press 'x' to run the program."};
   char *material[9] = {"cement", "foam", "glass", "inside air",
      "insulation", "outside air","sheetrock", "stucco", "wood" };
   char *letter[9] = {"c", "f", "g", "a", "i", "o", "s", "t", "w" };
   char *command[5] = {"h", "e", "d", "r", "x"};
   char *action[5] = {"set as history cell", "set as energy source cell",
      "display/draw most recent saved wall","record in save file", "execute
      the rest of the program"};
   MPE_Color colors[9] = {MPE_RED, MPE_BLUE, MPE_AQUAMARINE, MPE_CYAN,
      MPE_PINK, MPE_BLACK,MPE_GRAY, MPE_ORANGE, MPE_BROWN};
   char *color[9] = {"red", "blue", "aqua", "cyan", "pink", "white", "grey",
      "orange", "brown"};
   int place = 45;

   for(i = 0; i < sizeof material/sizeof(char *)-1; i++)
         MPE_Draw_line( graph2, place, (i*20)+135, place+160, (i*20)+135,
      MPE_BLACK);
   MPE_Draw_line( graph2, place+80, 125, place+80, 290, MPE_BLACK );
   MPE_Draw_line( graph2, place+120, 125, place+120, 290, MPE_BLACK );
   display_strings( string, sizeof string/sizeof(char *), 10, 20, MPE_BLACK);
   display_strings( material, sizeof material/sizeof(char *), place, 130,
      MPE_BLACK);
   display_strings( letter, sizeof letter/sizeof(char *), place+=100, 130,
      MPE_BLACK);
   colored_strings( color, sizeof color/sizeof(char *), place+=35, 130,
      colors);

   MPE_Draw_string( graph2, 10, 320, MPE_BLACK, "Other action keys" );
   display_strings( command, sizeof command/sizeof(char *), place=45, 340,
      MPE_BLACK);
   for(i = 0; i <sizeof command/sizeof(char *)-1; i++)
   display_strings( action, sizeof action/sizeof(char *), place+=35, 340,
      MPE_BLACK);

   MPE_Update(graph2);
}
void display_strings( char **strings, int size, int xloc, int yloc, MPE_Color
color ) {
      int y;
      for( i = 0; i < size; i++ ) {
            y = yloc + (20 * i);
            MPE_Draw_string(graph2, xloc, y, color, strings[i]);
```

```
      }
      MPE_Update(graph2);
}
void colored_strings( char **strings, int size, int xloc, int yloc, MPE_Color
*color ) {
      //color array must be same size or > than string array
      int y;
      for( i = 0; i < size; i++ ) {
            y = yloc + (20 * i);
            MPE_Draw_string(graph2, xloc, y, color[i], strings[i]);
      }
      MPE_Update(graph2);
}
void display_setup(int matrix_size_x, int matrix_size_y, int **Color, int
      my_offset, int mysize,double time, double Temp_max, double Temp_min,
      int **Source) {
   int i, j;
   unsigned int plot_value;
   int xloc, yloc, xwid, ywid;

   for (j = 0; j < matrix_size_y; j++) {
      for (i = 0; i < matrix_size_x; i++) {
         xloc = ( i * (width) )/ matrix_size_x;
         yloc = ( j * (height) )/ matrix_size_y;
         xwid = ((width)/matrix_size_x);
         ywid = ((height)/matrix_size_y);
         if (plot_value < 2)
            plot_value = 2;
         if (plot_value > ncolors)
              plot_value = ncolors;
         MPE_Draw_line(graph, width, 0, width, height, MPE_BLACK);
            //right boundary
         MPE_Draw_line(graph, 0, height, width, height, MPE_BLACK);
            //lower boundary
         MPE_Draw_line(graph, 0, 0, width, 0, MPE_BLACK);
            //upper boundary
         MPE_Draw_line(graph, 0, 0, 0, height, MPE_BLACK);
            //left boundary
         MPE_Fill_rectangle(graph, xloc, yloc, xwid, ywid, Color[j][i]);

         if (j >= 0 && i >= 1) {
            if ( Color[j][i] != Color[j][i-1] || Color[j][i] == 0 )
               MPE_Draw_line( graph, xloc, yloc, xloc, ywid+yloc, MPE_BLACK
);
         }
         if (j >= 1 && i >= 0) {
            if ( Color[j][i] != Color[j-1][i] || Color[j][i] == 0 )
                     MPE_Draw_line( graph, xloc, yloc, xwid+xloc, yloc,
                        MPE_BLACK );
         }
         if( Source[j][i]  > 0 ){
             MPE_Draw_line( graph, xloc, yloc, xwid+xloc, ywid+yloc,
                  MPE_YELLOW );
             MPE_Draw_line( graph, xwid+xloc, yloc, xloc, ywid+yloc,
                  MPE_YELLOW );
         }
      }
```

```
   }

   MPE_Fill_rectangle(graph, 0, height+4, width, height+4, MPE_WHITE);
   if (my_offset == 0) MPE_Draw_string( graph, 0, height+16, MPE_BLACK,
      label);
   MPE_Update(graph);
   sleep(WAIT_TIME);
}

void display_colors() {
   int ierr;
   int rank;
   color_array = (MPE_Color *) malloc(sizeof(MPE_Color) * ncolors);

   ierr = MPE_Make_color_array(graph, ncolors, color_array);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (ierr && rank == 0) printf("Error(Make_color_array): ierr is %d\n",
      ierr);
}

void display_one_d(int matrix_size_x, int matrix_size_y, double **temp, int
      my_offset, int mysize, double maxscale, double time, double Temp_max,
      double Temp_min, int **Material) {
   int i, j;
   unsigned int plot_value;
   int xloc, yloc, xwid, ywid;
   char string[60], outside_temp[12];
   time_t currenttime;

   /* Fill cells to display temperature */
   for (j = 0; j < matrix_size_y; j++) {
      for (i = 0; i < matrix_size_x; i++) {
           xloc = (i * width)/matrix_size_x;
           yloc = (j * height)/matrix_size_y;
           xwid = (width/matrix_size_x);
           ywid = (height/matrix_size_y);
           plot_value = ncolors - ((double)ncolors * temp[j][i] / maxscale);
           if (plot_value < 2) plot_value = 2;
           if (plot_value >= ncolors) plot_value = ncolors-1;
           if( isnan( temp[j][i] ) )
             MPE_Fill_rectangle( graph, xloc, yloc, xwid, ywid, MPE_WHITE );
           else if( temp[j][i] < 0 )
             MPE_Fill_rectangle( graph, xloc, yloc, xwid, ywid, MPE_WHITE );
           else if( temp[j][i] > maxscale )
             MPE_Fill_rectangle( graph, xloc, yloc, xwid, ywid, MPE_BLACK );
           else
             MPE_Fill_rectangle( graph, xloc, yloc, xwid, ywid,
            color_array[plot_value] );
      }
   }
   for (j = 1; j < matrix_size_y; j++) {
      for (i = 1; i < matrix_size_x; i++) {
           xloc = (i * width)/matrix_size_x;
           yloc = (j * height)/matrix_size_y;
           xwid = xloc+(width/matrix_size_x);
           ywid = yloc+(height/matrix_size_y);
           if( Material[j][i] != Material[j][i-1] )
```

```
            MPE_Draw_line(graph, xloc, yloc, xloc, ywid, MPE_BLACK);
          if( Material[j][i] != Material[j-1][i] )
            MPE_Draw_line(graph, xloc, yloc, xwid, yloc, MPE_BLACK);
      }
   }
   if(DEBUG)
      printf("Color display complete\n");

   if(first) {       //draw the scale on first call
         display_scale( maxscale, width+15.0, 45.0, 15.0, 500.0 );
         first = 0;
   }

   /* Draw sun/moon clock and outside temp */
   currenttime = epochtime + (int)time * 60;
   localtime_r(&currenttime, &time_structure);
      //advance epochtime by simulation time
   if(DEBUG)
      printf("time = %s\n", asctime(&time_structure));

   if(time_structure.tm_hour >= 18 || time_structure.tm_hour < 6) {
      MPE_Fill_circle(graph, width + 40, .93*height, 35, MPE_BLACK);
        MPE_Fill_circle(graph, width + 48, .93*height, 30, MPE_WHITE);
   }
   else
      MPE_Fill_circle(graph, width + 40, .93*height, 35, 60);

   if (time_structure.tm_hour == 12)                      //noon
      sprintf(string, "12:%02d pm", time_structure.tm_min);
   else if (time_structure.tm_hour == 0)                  //midnight
      sprintf(string, "12:%02d am", time_structure.tm_min);
   else if (time_structure.tm_hour > 12)
      sprintf(string,"%d:%02d pm", time_structure.tm_hour%12,
      time_structure.tm_min);
   else
      sprintf(string,"%d:%02d am", time_structure.tm_hour,
      time_structure.tm_min);

   MPE_Draw_string(graph, width + 28, .93*height, MPE_BLACK, string);
   sprintf(outside_temp, "%.2lf%cF", temp[0][0], 0x00B0);
   MPE_Draw_string(graph, width + 28, .95*height, MPE_BLACK, outside_temp);
   sleep(WAIT_TIME);
}
void display_scale(double maxscale, double x1, double xsize, double y1,
double ysize) {
   //xsize should be greater than 15, ysize greater than 8
   int plot_value;
   double step, place;
   sprintf(string, "Scale %cF", 0x00B0);
   MPE_Draw_string(graph, x1+5, y1, MPE_BLACK, string);
   y1 += 8;
   ysize -= 8;
   ysize -= (int)ysize % (int)maxscale;
   step = ysize/maxscale;
   for( i = maxscale; i >= 0; i-- ) {
      plot_value = ncolors - ((double)ncolors * i / maxscale);
      place = y1+ysize-(step * i);
```

```
      MPE_Fill_rectangle( graph, x1+15, place, xsize-15, step,
      color_array[plot_value] );
        if( i % 10 == 0) {
         sprintf( string, "%d", i);
             MPE_Draw_string(graph, x1, place+5, MPE_BLACK, string);
      }
   }
}

char Iget_key_press(int*xpos, int*ypos) {  //from Sapient
   XEvent event;
   char keys[20], toReturn;
   int numChar;
   KeySym keysym;
   XComposeStatus compose;
   if (graph->Cookie != MPE_G_COOKIE) {
      fprintf(stderr, "Handle argument is incorrect or corrupted\n");
      return '\0';
   }
   XSelectInput(graph->xwin->disp,graph->xwin->win,
      MPE_XEVT_IDLE_MASK|ButtonPressMask|KeyPressMask);
   //add mouse press to events monitored
   if (XCheckWindowEvent(graph->xwin->disp,graph->xwin->win, KeyPressMask,
&event) == False) {
      return '\0';
   }
   //check once if mouse has been pressed
   numChar = XLookupString(&event, keys, 20, &keysym, &compose);
   toReturn = '\0';
   *xpos=event.xkey.x;
   *ypos=event.xkey.y;
   if ((((keysym>=XK_KP_Space) && (keysym<=XK_KP_9)) || ( (keysym>XK_space)
&& (keysym<XK_asciitilde)))) {
      toReturn = keys[0];
   }
   XSelectInput(graph->xwin->disp, graph->xwin->win, MPE_XEVT_IDLE_MASK);
   /*turn off all events*/
   return toReturn;
}

void display_close(void) {
   MPE_Close_graphics(&graph);
   MPE_Close_graphics(&graph2);
}

void set_label(char *text) {
   label = text;
}

char get_key(int *i, int *j, int my_offset, int matrix_size_x, int
matrix_size_y) {
   char ch;
   int xcor, ycor;

   ch = Iget_key_press(&xcor, &ycor);
   if (ch != '\0') {
      *i = (xcor - 2) * matrix_size_x / width;
```

```
      *j = (ycor - 2) * matrix_size_y / height - my_offset;
      if (*i < 0)
         *i = 0;
      if (*j < 0)
         *j = 0;
      if (*i > matrix_size_x)
         *i = matrix_size_x;
      if (*j > matrix_size_y)
         *j = matrix_size_y;
   }
   MPI_Bcast(&ch, 1, MPI_CHAR, 0, MPI_COMM_WORLD);
   MPI_Bcast(i, 1, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Bcast(j, 1, MPI_INT, 0, MPI_COMM_WORLD);
   return ch;
}
```

TRIDIAGONAL SOLVER

```
/* From Wikipedia */
/* Fills solution into x */
void TridiagonalSolve(const double *a, const double *b, double *c, double *d,
double *x, unsigned int n) {
      int i;

   double eps = 1.0e-20;
   /* Modify the coefficients - forward sweep */
   if ( b[0] != 0.0) {
      c[0] /= b[0];
        d[0] /= b[0];
   }
   else {
      c[0] /= (b[0] + eps);
      d[0] /= (b[0]+eps);
   }
   for (i = 1; i < n; i++) {
      double id = (b[i] - c[i-1] * a[i]); //Division by zero risk
      c[i] /= id;                         //Last value found is redundant
      d[i] = (d[i] - d[i-1] * a[i])/id;
   }

   /* Substitute - backward sweep */
   x[n - 1] = d[n - 1];
   for(i = n - 2; i >= 0; i--){
      x[i] = d[i] - c[i] * x[i + 1];
   }
}
```