Alternating Direct Implicit Method for Solving the 2D Heat Equation

Kevin Juan (kj89)

December 12th 2018

Introduction

Partial Differential Equations (PDEs) are a class of differential equations for which the solution is a function of several variables. These types of equations appear in many areas of study including physics, chemistry, engineering, biology, and finance due to their ability to model real world phenomena. In many scientific applications, PDEs often solve for functions of space and time, $f(x, y, z, t)$. One particular class of PDEs of interest is the parabolic PDE. If we consider a function $f(x, y)$ with independent variable x and y, the second order, linear, constant coefficient PDE has the form

$$Af_{xx} + 2Bf_{xy} + Cf_{yy} + Df_x + Ef_y + F = 0, \tag{1}$$

where $B^2 - AC = 0$ is the condition that the PDE meets to be classified as parabolic [1]. One of the most widely studied parabolic PDEs is the Heat Equation, which is derived from the conservation of energy as shown below.

$$\rho C_p \frac{\partial T}{\partial t} = k\nabla^2 T + \dot{q} \tag{2}$$

The term $\dot{q}$ is a heat flow term, which is 0 for many simple problems. The constants ρ, C_p, and k are the density, specific heat capacity, and thermal conductivity, all of which vary by the material of interest. When solved, the Heat Equation's solution gives a function for the temperature profile for the geometry of interest. Many simple problems look at flat, rectangular plates in Cartesian coordinates or cylindrical rods in cylindrical coordinates. For the purpose of this project, the 2D square plate was the geometry of choice due to its appearance in many practical applications including stove tops and heat sinks for computer components like CPUs and GPUs. Following some rearrangement and expansion of the gradient operator, our 2D Heat Equation takes on the form:

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) + \frac{\dot{q}}{\rho C_p}. \tag{3}$$

We have introduced a new constant, α, called the thermal diffusivity, which is a measure of the rate of heat transfer from a hot side to a cold side. The thermal diffusivity is defined below.

$$\alpha = \frac{k}{\rho C_p} \tag{4}$$

While many problems involving the Heat Equation can be solved analytically with Fourier Series, many problems involving PDEs in real world applications cannot be solved analytically, and must be done numerically. One popular method of solving such problems uses the finite difference method. Simply put, the finite difference method discretizes the area of interest into many nodes and calculates an average of the derivative of a node's nearest neighbors to estimate the value of the function at the given node. For steady state problems, the Gauss-Seidel with relaxation is an effective method for solving PDEs. For 1D transient problems The Crank-Nicolson method is an efficient way to turn the PDE into a simple matrix equation that can be efficiently solved with an error of $\mathcal{O}(\Delta t^2)$ and unconditional stability. However, higher dimension transient problems pose a major issue when using the Crank-Nicolson method. For the simple 1D case, the matrix equation is a simple tri-diagonal matrix. At higher dimensions the matrix equation becomes the solution to a band diagonal matrix, which can be prohibitively expensive to compute for large grids. A simple yet elegant way to solve such a complex problem is the Alternating Direction Implicit (ADI) method, which is the focus of this project and is discussed in the next section.

More on the Heat Equation, finite differences, Crank-Nicolson, and the ADI method can be found in the following resources: Jaluria and Torrance[1], Press et al[2], Özışık et al[3], Pletcher et al[4], and Majumdar et al[5].

Methods and Algorithms

The ADI method is a simple method for solving the 2D Heat Equation that solves the equation twice within one full time step between n and $n+1$. As the name implies, the finite difference equation is solved implicitly for each direction in single time step. In the first step from n to $n+\frac{1}{2}$, the x direction is solved implicitly while the y direction is solved explicitly. In the second time step from $n+\frac{1}{2}$ to $n+1$, the y direction is solved implicitly while the x direction is solved explicitly. This results in two different finite difference equations for the 2D Heat Equation as shown below:

$$\frac{T_{i,j}^{n+\frac{1}{2}} - T_{i,j}^{n}}{\frac{\Delta t}{2}} = \alpha\left(\frac{T_{i-1,j}^{n+\frac{1}{2}} - 2T_{i,j}^{n+\frac{1}{2}} + T_{i+1,j}^{n+\frac{1}{2}}}{(\Delta x)^2} + \frac{T_{i,j-1}^{n} - 2T_{i,j}^{n} + T_{i,j+1}^{n}}{(\Delta y)^2}\right) + \frac{1}{\rho C_p}\dot{q}_{i,j}^{n+\frac{1}{2}} \tag{5}$$

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n+\frac{1}{2}}}{\frac{\Delta t}{2}} = \alpha\left(\frac{T_{i-1,j}^{n+\frac{1}{2}} - 2T_{i,j}^{n+\frac{1}{2}} + T_{i+1,j}^{n+\frac{1}{2}}}{(\Delta x)^2} + \frac{T_{i,j-1}^{n+1} - 2T_{i,j}^{n+1} + T_{i,j+1}^{n+1}}{(\Delta y)^2}\right) + \frac{1}{\rho C_p}\dot{q}_{i,j}^{n+1} \quad (6)$$

To make computation easier, we can rearrange the two equations to obtain the two finite difference equations as follows:

$$-r_x T_{i-1,j}^{n+\frac{1}{2}} + (1+2r_x)T_{i,j}^{n+\frac{1}{2}} - r_x T_{i+1,j}^{n+\frac{1}{2}} + \frac{\Delta t}{2\rho C_p}\dot{q}_{i,j}^{n+\frac{1}{2}} = r_y T_{i,j-1}^{n} + (1-2r_y)T_{i,j}^{n} + r_y T_{i,j+1}^{n} \quad (7)$$

$$-r_y T_{i,j-1}^{n+1} + (1+2r_y)T_{i,j}^{n+1} - r_y T_{i,j+1}^{n+1} + \frac{\Delta t}{2\rho C_p}\dot{q}_{i,j}^{n+1} = r_x T_{i-1,j}^{n+\frac{1}{2}} + (1-2r_x)T_{i,j}^{n+\frac{1}{2}} + r_x T_{i+1,j}^{n+\frac{1}{2}} \quad (8)$$

We define r_x and r_y as

$$r_x = \frac{\alpha \Delta t}{2(\Delta x)^2} \text{ and} \quad (9)$$

$$r_y = \frac{\alpha \Delta t}{2(\Delta y)^2}. \quad (10)$$

It is noted that both of these values are identical for a square geometry with equal number of nodes in each dimension. As shown in Equation 7, we first perform a sweep of each column in the 2D matrix defining the square geometry. During this sweep, a tri-diagonal matrix equation is solved for each column with the solution obtained from the tri-diagonal matrix equation solver replacing the old column in the matrix that stores the temperature at each node. At the end of the column sweep, the matrix containing information of the temperature at all the nodes has been updated to carry the solution to Equation 7. After the column sweep, a similar routine is done whereby the tri-diagonal matrix equation is used on fixed rows. The two tri-diagonal matrix equations are visualized below:

$$\begin{bmatrix} b_0 & c_0 & & & & & & \\ a_1 & b_1 & c_1 & & & & & \\ & & a_2 & b_2 & c_2 & & & \\ & & & & \ddots & & & \\ & & & & a_{N_x-2} & b_{N_x-2} & c_{N_x-2} & \\ & & & & & & a_{N_x-1} & b_{N_x-1} \end{bmatrix} \begin{bmatrix} T_0 \\ T_1 \\ T_2 \\ \vdots \\ T_{N_x-2} \\ T_{N_x-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{N_x-2} \\ d_{N_x-1} \end{bmatrix} \quad (11)$$

$$\begin{bmatrix} b_0 & c_0 & & & & & & \\ a_1 & b_1 & c_1 & & & & & \\ & & a_2 & b_2 & c_2 & & & \\ & & & & \ddots & & & \\ & & & & a_{N_y-2} & b_{N_y-2} & c_{N_y-2} & \\ & & & & & & a_{N_y-1} & b_{N_y-1} \end{bmatrix} \begin{bmatrix} T_0 \\ T_1 \\ T_2 \\ \vdots \\ T_{N_y-2} \\ T_{N_y-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{N_y-2} \\ d_{N_y-1} \end{bmatrix} \quad (12)$$

The algorithm to solve the tri-diagonal matrix equations is listed as follows:

Algorithm 1 Tri-Diagonal Solver

$c_0 \leftarrow c_0/b_0$ and $d_0 \leftarrow d_0/b_0$
for $i = 1, 2, 3, ...N-2$ **do**
    $c_i \leftarrow c_i/(b_i - a_i c_{i-1})$
for $i = 1, 2, 3, ...N-1$ **do**
    $d_i \leftarrow (d_i - a_i d_{i-1})/(b_i - a_i c_{i-1})$
$T_{N-1} \leftarrow d_{N-1}$
for $i = N-2, ..., 0$ **do**
    $T_i \leftarrow d_i - c_i T_{i+1}$
return

For the tri-diagonal matrix solver, the left-hand sides of Equations 7 and 8 are stored in $\vec{d}$. The values stored $\vec{a}$, $\vec{b}$, and $\vec{c}$ are simply r, $1-2r$, and r, respectively for each equation. We assert that $T(x, y, t) = 0°\text{C}$ at $i = j = -1$, $i = N_x$, and $j = N_y$, thus imposing Dirichlet boundary conditions. The pseudocode below shows the structure for the ADI method. The number of nodes in the x direction is given by iMax and the number of nodes in the y direction is given by jMax. nMax is the number of time iterations.

Algorithm 2 ADI Method

Initialize $T(x, y, t = 0)$
Initialize $\vec{a}, \vec{b}, \vec{c}, \vec{d}, \mathbf{T}$
Begin big loop for time
for $n = 1, 2, 3, ...$nMax **do**
    Begin column sweep
    for $i = 0, 1, 2, 3, ...$iMax **do**
        for $j = 0, 1, 2, 3, ...$jMax **do**
            Assign $\vec{a}$, $\vec{b}$, $\vec{c}$, and $\vec{d}$
        Call `tridiag`
        Replace current column in $\mathbf{T}$ with $\vec{d}$
    $\mathbf{T}$ contains intermediate solution
    Begin row sweep
    for $j = 0, 1, 2, 3, ...$jMax **do**
        for $i = 0, 1, 2, 3, ...$iMax **do**
            Assign $\vec{a}$, $\vec{b}$, $\vec{c}$, and $\vec{d}$
        Call `tridiag`
        Replace current row in $\mathbf{T}$ with $\vec{d}$
$\mathbf{T}$ now contains the final solution

One of the benefits of the ADI method is that the method is accurate to $\mathcal{O}((\Delta x)^2, (\Delta y)^2, (\Delta t)^2)$.

Additionally, stability analysis as shown below proves that the method is unconditionally stable for any Δt, where we define $\Delta x = \Delta y = h$.

$$\frac{\epsilon^{n+\frac{1}{2}}}{\epsilon^n} = \frac{1 - 2\frac{\alpha\Delta t}{h^2}\sin^2\left(\frac{kh}{2}\right)}{1 + 2\frac{\alpha\Delta t}{h^2}\sin^2\left(\frac{mh}{2}\right)} \tag{13}$$

$$\frac{\epsilon^{n+1}}{\epsilon^{n+\frac{1}{2}}} = \frac{1 - 2\frac{\alpha\Delta t}{h^2}\sin^2\left(\frac{mh}{2}\right)}{1 + 2\frac{\alpha\Delta t}{h^2}\sin^2\left(\frac{kh}{2}\right)} \tag{14}$$

$$\frac{\epsilon^{n+1}}{\epsilon^n} = \left(\frac{1 - 2\frac{\alpha\Delta t}{h^2}\sin^2\left(\frac{kh}{2}\right)}{1 + 2\frac{\alpha\Delta t}{h^2}\sin^2\left(\frac{mh}{2}\right)}\right)\left(\frac{1 - 2\frac{\alpha\Delta t}{h^2}\sin^2\left(\frac{mh}{2}\right)}{1 + 2\frac{\alpha\Delta t}{h^2}\sin^2\left(\frac{kh}{2}\right)}\right) < 1 \tag{15}$$

Implementation and Results

The ADI Method that was written was designed to solve the 2D Heat Equation for a variety of scenarios. The program that was written solved the equation for a square plate with $L_x = 5$ m and $L_y = 5$ m. To limit the file size and computation time, 101 nodes were used for both the x and y dimension. The program asks the user to choose between two materials to simulate: aluminum and polyvinyl chloride (PVC). These were chosen due to their drastically different physical properties. Aluminum is highly conductive to heat transfer while PVC is highly resistant to heat transfer. Their properties are listed in the table below:

	Aluminum	PVC
α in m^2/s	9.7×10^{-5}	8.0×10^{-8}
ρ in g/m^3	2.71×10^6	1.38×10^6
C_p in $\mathrm{J}/(\mathrm{g}^\circ\mathrm{C})$	0.9	0.9

Table 1: Physical constants for aluminum and PVC (Data acquired from `wikipedia.org`).

The program then asks the user to select an initial temperature profile $T(x, y)$. Their names and function form are shown in the table below. If none is selected, the program defaults to an exponential profile. Based on the material, an amplitude, A, was chosen such that the initial temperature does not occur at a solid to liquid phase change or in a region where the material is non-solid.

Name	$T(x, y)$
Gaussian	$A\exp(-((x-2.5)^2/2 + (y-2.5)^2/2))$
Bump	$A\lvert\sin(5x)\cos(5y)\rvert$
Ripple	$A\lvert\sin((x-2.5)^2 + (y-2.5)^2)\rvert$
Elliptic Half-Gaussian	$A\exp(-(x^2/10 + (y-2.5)^2/2))$
Exponential	$A\exp(-x/2.5)$

Table 2: Initial temperature profiles that the user can select.

After selecting the initial profile, the program asks the user for a source term $S(t)$ and $\dot{q}_{Max}$, which is the maximum heat flow in Watts. The program defaults to no source if the user does not select one. The following table lists the source name and its functional form.

Name	$S(t) kevinjuan25 Rangers25!$
Oscillatory	$\dot{q}_{Max} \sin(2\pi t)$
Decaying Oscillatory	$\dot{q}_{Max} \exp(-t/10) \sin(2\pi t)$
Constant	$\dot{q}_{Max}$
Logistic	$\dot{q}_{Max}/(1 + \exp(-(t - t_{Max}/2))$

Table 3: Source terms that the user can select.

The final user input is the simulation time and the number of time iterations for the simulation. After all inputs have been defined, the ADI method is run. The program saves the profile at n equally spaced times based on the user input. This is done to visualize the temperature profile at different times and to create an animation of the temperature profile's change with time in Python. Generally, 10 profiles is enough to observe steady changes for a large enough simulation time.

For demonstration purposes, each material was simulated twice. Each material was simulated with no source term to demonstrate cooling. After a simulation with no source term, each material was simulated with a constant source term of $\dot{q}_{Max}$. Both simulations used an Elliptic Half-Gaussian initial profile as shown below.

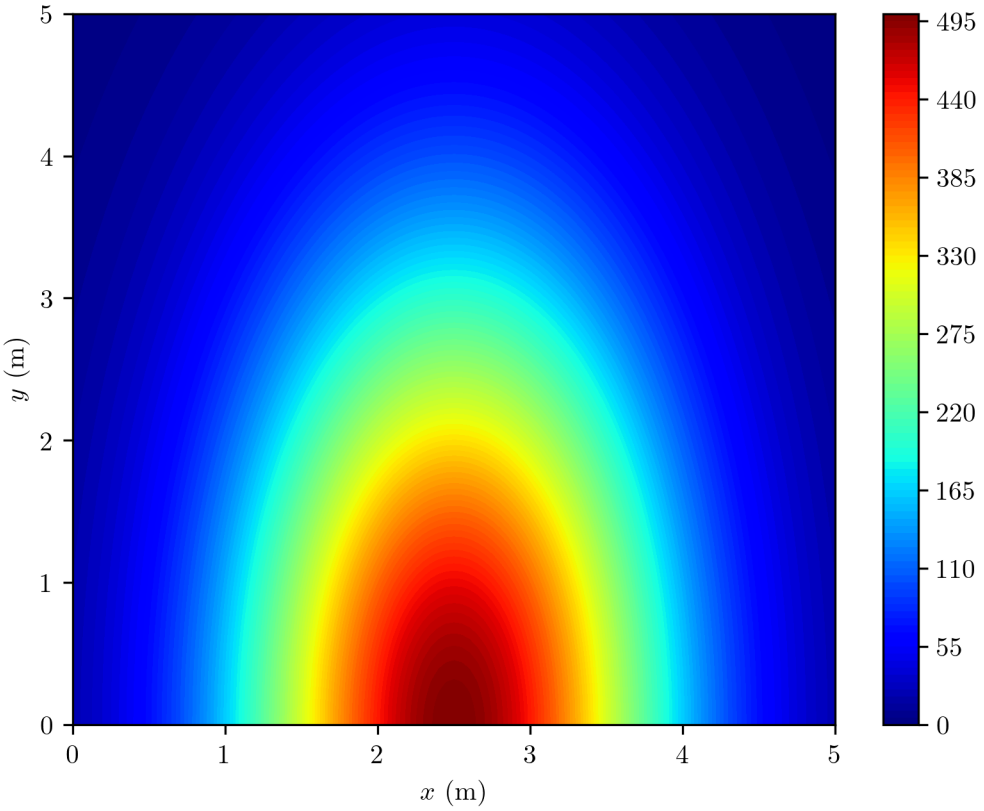

Figure 1: Initial temperature profile for Aluminum.

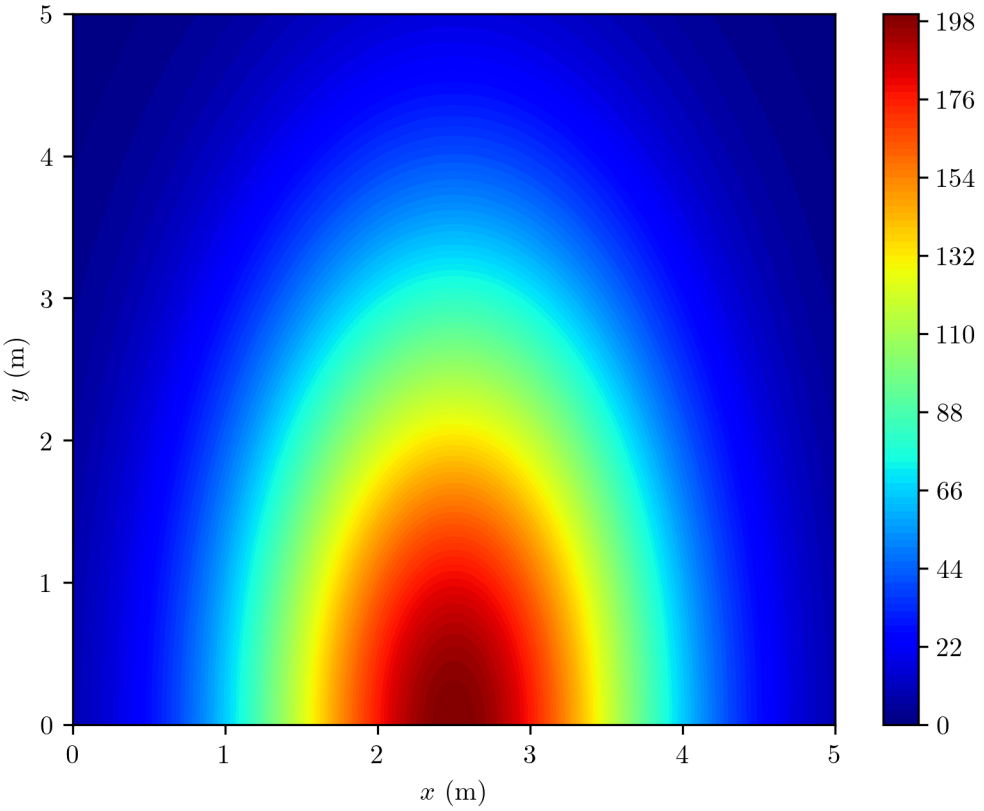

Figure 2: Initial temperature profile for PVC.

For the aluminum simulation, the total simulation time was 10 seconds with 101 time iterations. The PVC simulation was done for 10,000 seconds with 100,001 time iterations. Temperature profiles were saved at 5 and 10 seconds for aluminum and 5,000 and 10,000 seconds for PVC. A shorter simulation time was chosen for aluminum due to its ability to transfer heat quickly, while PVC required a longer simulation due to it inability to transfer heat. The following plots show profiles at the simulation mid-point and end when no source was used.

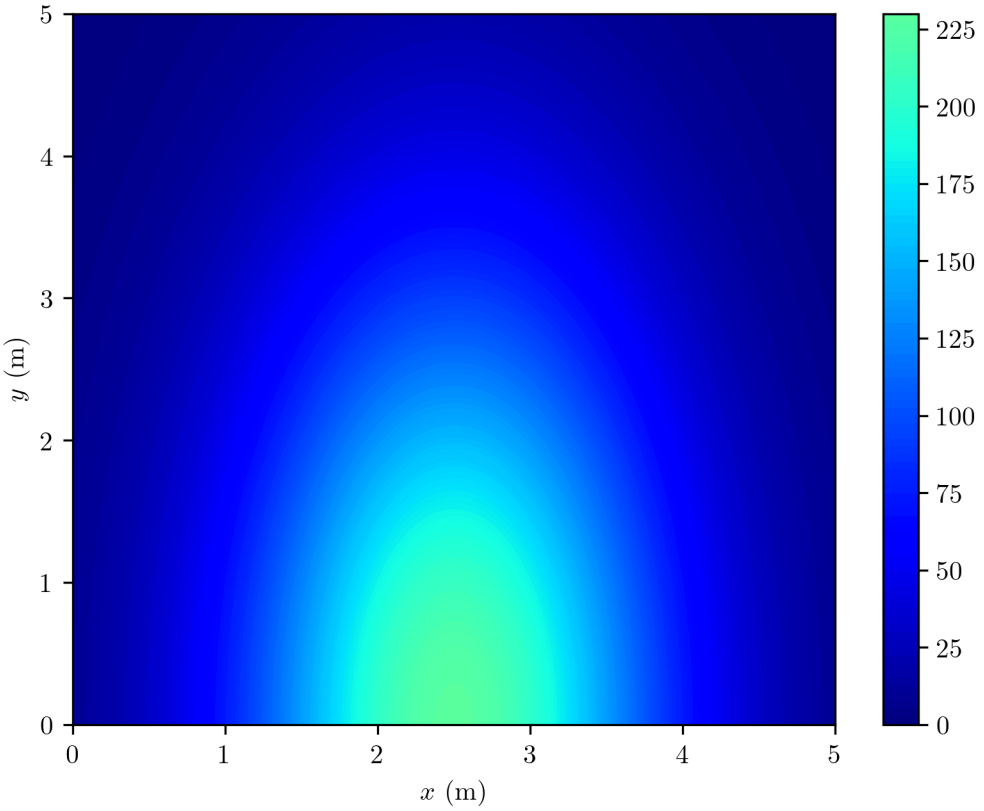

Figure 3: Temperature profile for Aluminum at 5 seconds with no source.

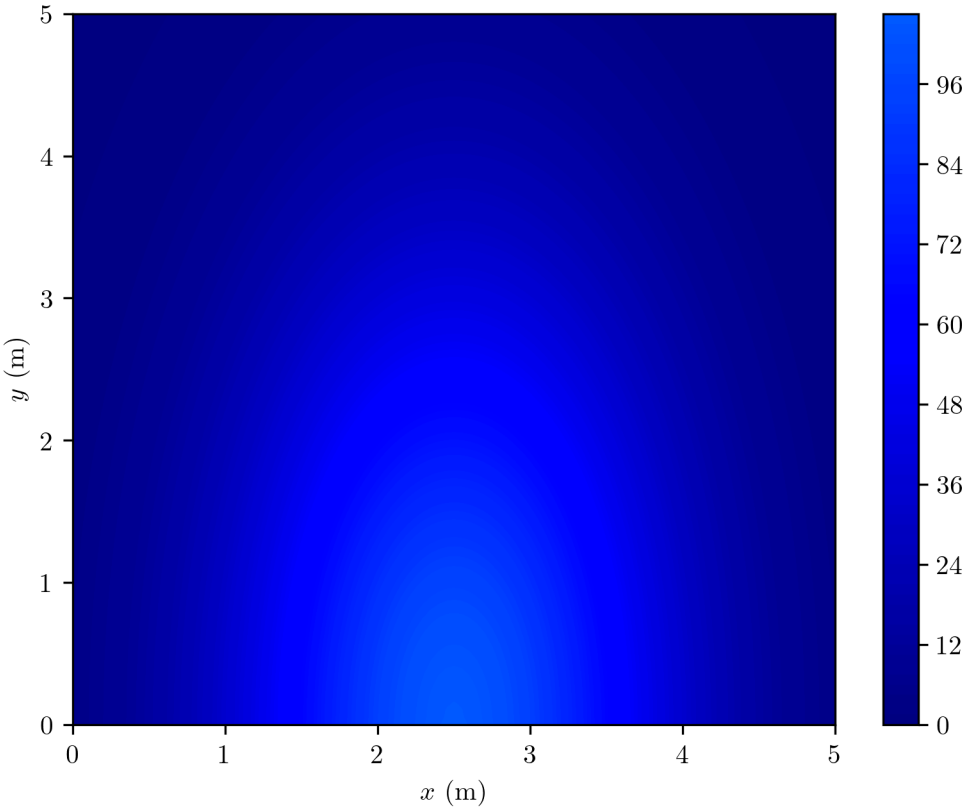

Figure 4: Temperature profile for Aluminum at 10 seconds with no source.

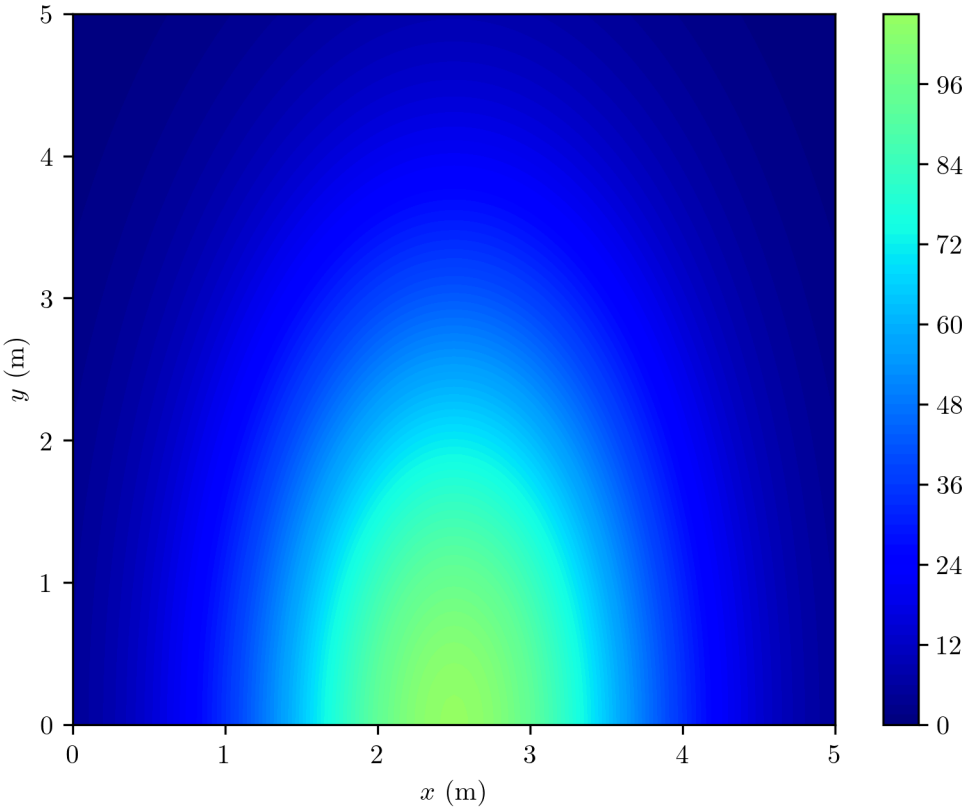

Figure 5: Temperature profile for PVC at 5,000 seconds with no source.

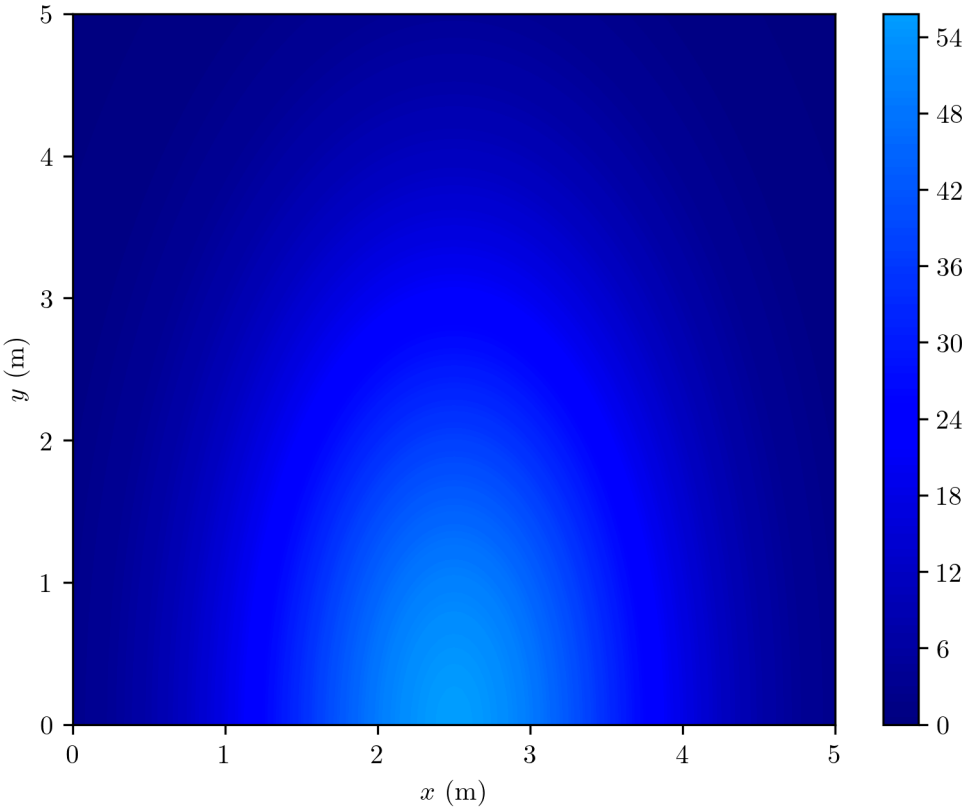

Figure 6: Temperature profile for PVC at 10,000 seconds with no source.

As expected, aluminum's conductive properties required a shorter simulation to detect significant temperature changes, while PVC required a much longer time to show significant changes in temperature due to its insulating properties. When a constant heat source was added, 50 MW was chosen for aluminum while 10 kW was chosen for PVC. These heat flows were chosen for the materials due to their material properties. Since the last simulation showed how effective aluminum was at transferring heat, we could input much more heat before significant temperature changes would be noticed between the two simulations. PVC's insulting properties made it such that less heat could be added to the system before the temperature rose to drastically high temperatures. The profiles recorded at the same times for both materials with a constant source is demonstrated below.

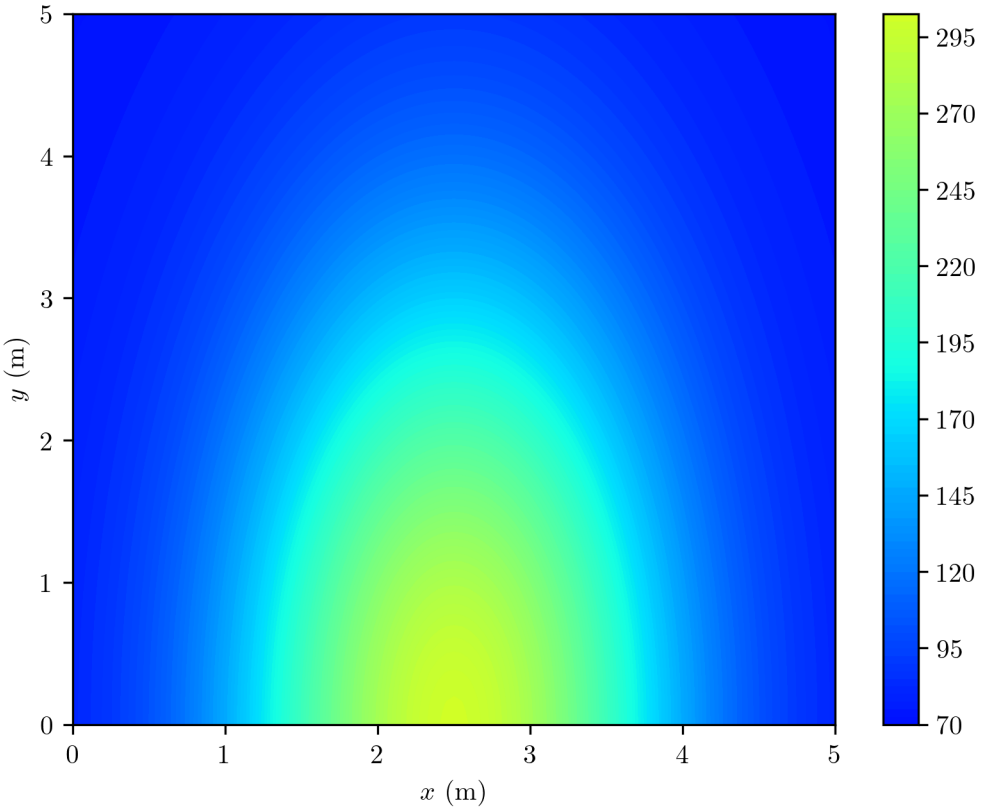

Figure 7: Temperature profile for Aluminum at 5 seconds with $\dot{q}_{Max} = 50$ MW.

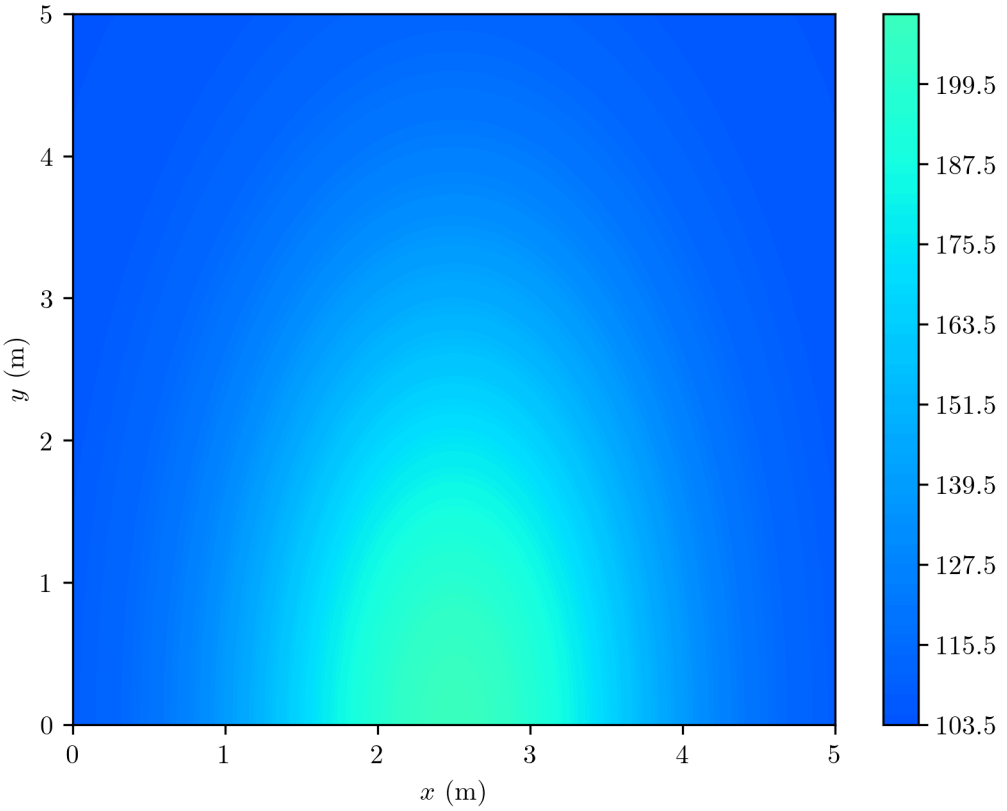

Figure 8: Temperature profile for Aluminum at 10 seconds with $\dot{q}_{Max} = 50$ MW.

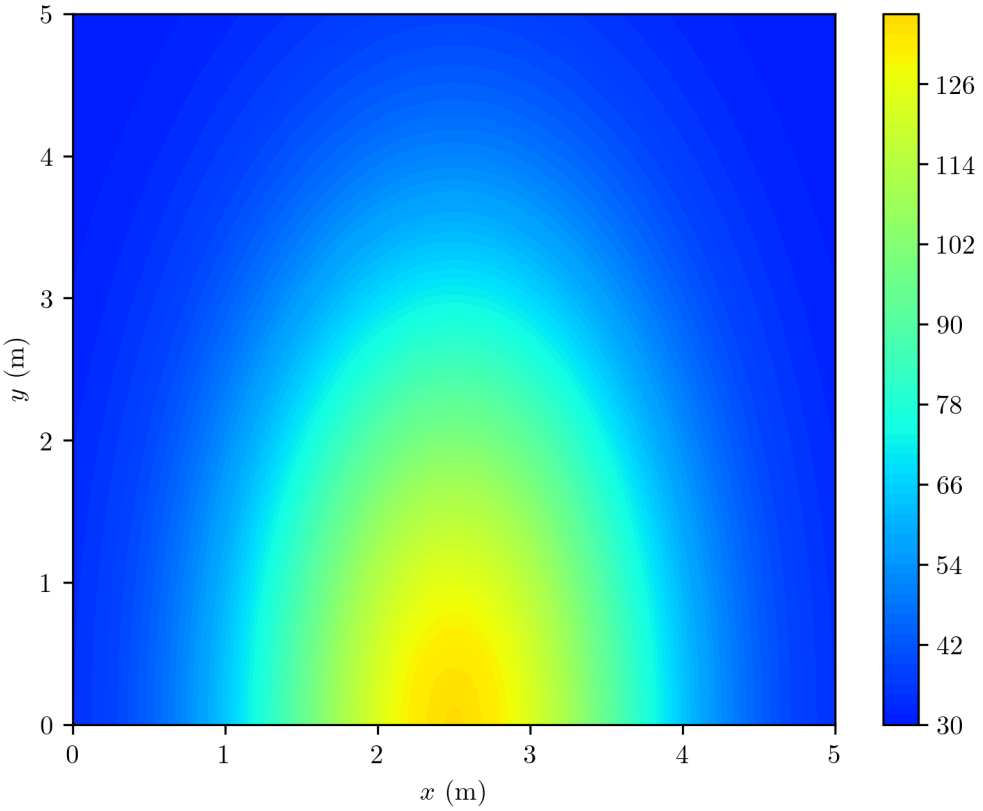

Figure 9: Temperature profile for PVC at 5,000 seconds with $\dot{q}_{Max} = 10$ kW.

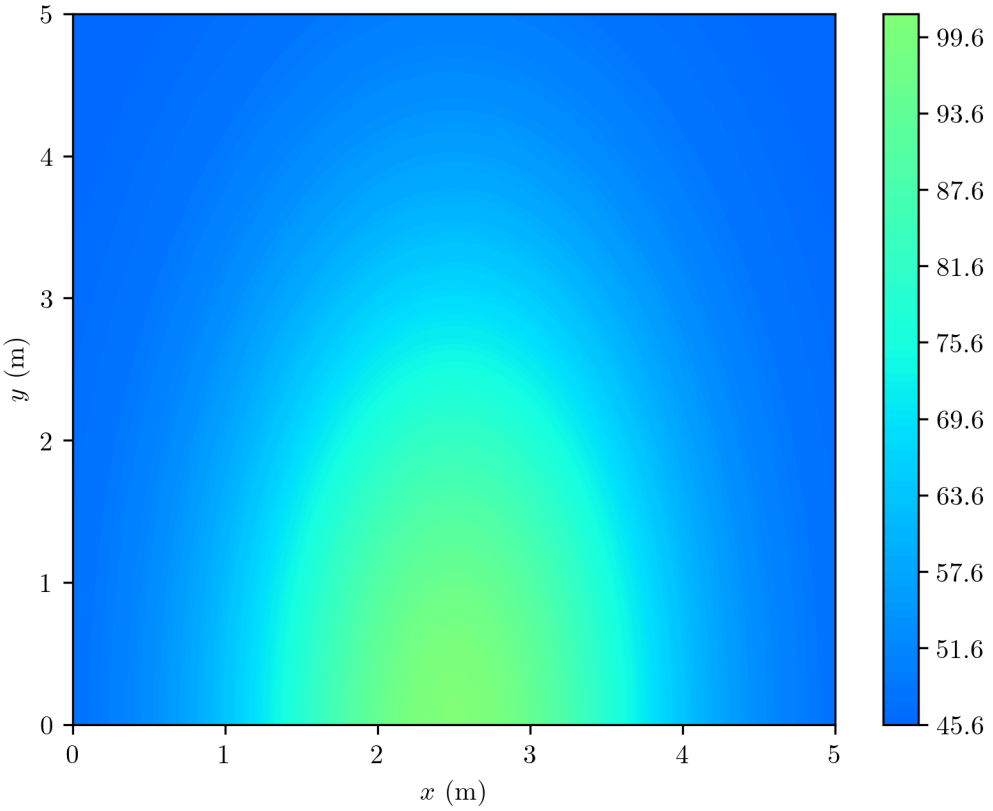

Figure 10: Temperature profile for PVC at 10,000 seconds with $\dot{q}_{Max} = 10$ kW.

We notice that with the addition of a source term, the temperature profiles at the same times as those recorded without a source for both materials are hotter due to the heat input. Animations for the 4 simulations have been included as supplemental material with this report, and demonstrate the temperature profile change over the course of several times.

Conclusions

It has been shown that the ADI method for solving the 2D Heat Equation is an efficient and accurate algorithm for solving transient PDEs in 2 spatial dimensions. Rather than solving a large and computationally expensive band diagonal matrix equation, we have shown that the problem can be reduced down to solving two tri-diagonal matrix equations in one full time step with each dimension being solved implicitly and explicitly exactly once. Due to the multiple nested loops needed to solve the two tri-diagonal equations, the ADI method can become slow for high resolution grids or large grids as was discovered during program testing. From a practical standpoint, these types of PDE problems are not well-suited for consumer grade computers beyond small, simple geometries. Because of this, many industries that

need to perform such calculations such as the automotive industry employ parallelization or run such simulations on much more powerful machines.

The results of running 4 different simulations with PVC and aluminum confirm what we expect based on real-life observation and intuition. When we looked at the case of cooling with no internal heat generation source, aluminum was able to show drastic temperature drops within seconds due to its ability to conduct heat. On the contrary, PVC required thousands of seconds to notice any significant temperature drop due to its thermal insulation properties, which can be observed from the order of magnitude of its thermal diffusivity. When a constant heat source term was introduced, both simulations showed that at similar times compared to the cooling case, the temperature profile was hotter as expected. Additionally, we showed that we could feed more heat into the aluminum while keeping its temperature under control. For PVC the amount of heat we could add while maintaining a controlled temperature was 3 orders of magnitude lower. Again, this demonstrates what we see in reality and confirms what we know to be true. The simulation gives us good mathematical and visual understanding of why metals like aluminum are used in applications like automobiles and computer components where the dissipation of heat is crucial to the product performance and why plastics and non-conducting materials are used in applications like homes and clothing where thermal insulation is desired.

One area of improvement that the program could use would be to introduce a search algorithm on the matrix storing the temperature data to allow the program to terminate when the temperature causes a phase change where material properties can drastically change. This made the simulation difficult to use for physical problems where such considerations play a large role as the temperature could often exceed phase boundaries if improper simulation conditions were used. Despite this, we have shown the program to be an effective general-use solver method for 2D parabolic PDEs.

Source Code

```
/* AEP 4380 Final Project

ADI Method tested on the 2D Heat Equation

Run on a core i7 using clang 1000.11.45.2 on macOS Mojave

Kevin Juan 12 December 2018
*/

#include <cmath>    // use math package
#include <cstdlib>  // plain C
#include <fstream>  // stream file IO
#include <iomanip>  // to format the output
#include <iostream> // stream IO
```

```
#define ARRAYT_BOUNDS_CHECK
#include "bigarrayt.hpp" // to use arrays
#define _USE_MATH_DEFINES

using namespace std;

const static double Lx = 5.0; // in m
const static double Ly = 5.0; // in m
const static double pi = M_PI;

template <class T>
void tridiag(arrayt<T> &a, arrayt<T> &b, arrayt<T> &c, arrayt<T> &d) {
    int k, kMax = b.n1();
    T bac;

    c(0) /= b(0);
    d(0) /= b(0);
    for (k = 1; k < kMax - 1; k++) {
        bac = b(k) - a(k) * c(k - 1);
        c(k) /= bac;
        d(k) = (d(k) - a(k) * d(k - 1)) / bac;
    }

    d(kMax - 1) = (d(kMax - 1) - a(kMax - 1) * d(kMax - 2)) /
                  (b(kMax - 1) - a(kMax - 1) * c(kMax - 2));

    for (k = kMax - 2; k >= 0; k--) {
        d(k) -= c(k) * d(k + 1);
    }
    return;
}

double TempInit(double x, double y, int initProfile, int material) {
    double TempInit;
    double xo = Lx / 2.0;
    double yo = Ly / 2.0;
    double dx = x - xo;
    double dy = y - yo;
    double A; // A should not exceed material melting point
    if (material == 1) {
        A = 500.0;
    } else if (material == 2) {
        A = 200.0;
    }
    if (initProfile == 1) {
```

```
        TempInit = A * exp(-(dx * dx / 2.0 + dy * dy / 2.0));
    } else if (initProfile == 2) {
        TempInit = A * abs(sin(5.0 * x) * cos(5.0 * y));
    } else if (initProfile == 3) {
        TempInit = A * abs(sin(dx * dx + dy * dy));
    } else if (initProfile == 4) {
        TempInit = A * exp(-(x * x / 10.0 + dy * dy / 2.0));
    } else {
        TempInit = A * exp(-x / xo);
    }
    return TempInit;
}

double S(double t, int sourceType, double tMax, double qMax) {
    // source is a heat flow in J/s
    // if qMax is positive, heat is added to the system
    // if qMax is negative, heat is removed from the system
    double source;
    if (sourceType == 1) {
        source = -qMax * sin(2.0 * pi * t);
    } else if (sourceType == 2) {
        source = -qMax * exp(-t / 10.0) * sin(2 * pi * t);
    } else if (sourceType == 3) {
        source = -qMax;
    } else if (sourceType == 4) {
        source = -qMax / (1 + exp(-(t - tMax / 2.0)));
    } else {
        source = 0.0;
    }
    return source / 2.0;
}

int main() {
    int i, j, n, iMax = 101, jMax = 101, nMax;
    double dx = Lx / (iMax - 1), dy = Ly / (jMax - 1);
    double currTime, midTime;
    int initProfile, sourceType, material;
    double x, y;
    double alpha; // in m^2/s
    double Cp;    // heat capacity in J/g*C
    double rho;   // density in g/m^3
    double t;     // time in seconds
    double qMax;  // max heat flow for the source in J/s
    int nSaves, tData = 0;
```

```
// asks user for initial profile, source type, and simulation time
cout << "What material should be used? (1 = Aluminum, 2 = PVC)" << endl;
cin >> material;
cout << "What initial profile should be used? (1 = Gaussian, 2 = Bump, 3 = "
        "Ripple, 4 = Elliptic Half-Gaussian, Exponential Otherwise)"
     << endl;
cin >> initProfile;
cout << "What source type should be used? (1 = Oscillatory, 2 = Decaying "
        "Oscillatory, 3 = Constant, 4 = Logistic, No Source Otherwise)"
     << endl;
cin >> sourceType;
cout << "What value should be use for maximum heat flow in J/s?" << endl;
cin >> qMax;
cout << "What simulation time should be used? (Input as Seconds)" << endl;
cin >> t;
cout << "How many time steps should be used? (Add 1 to account for t = 0)"
     << endl;
cin >> nMax;
cout << "How many profiles should be saved? (Excluding t = 0)" << endl;
cin >> nSaves;

double dt = t / (nMax - 1);

if (material == 1) {
    // Aluminum
    alpha = 9.7e-5; // in m^2/s
    Cp = 0.9;       // heat capacity in J/g*C
    rho = 2.71e6;   // density in g/m^3
} else if (material == 2) {
    // PVC
    alpha = 8.0e-8; // in m^2/s
    Cp = 0.9;       // heat capacity in J/g*C
    rho = 1.38e6;   // density in g/m^3
}

double rx = alpha * dt / (2.0 * dx * dx);
double ry = alpha * dt / (2.0 * dy * dy);

// we only need one set of vectors for an equal size area
arrayt<double> Temp(iMax, jMax);
arrayt<double> a(iMax);
arrayt<double> b(iMax);
arrayt<double> c(iMax);
arrayt<double> d(iMax);
arrayt<int> saves(nSaves);
```

```
// initialize array of iterations to save profiles
for (n = 0; n < nSaves; n++) {
    saves(n) = (n + 1) * t;
}

// output data for x
ofstream fpx;
fpx.open("x_vals.dat");
if (fpx.fail()) {
    cout << "cannot open file" << endl;
    return (EXIT_SUCCESS);
}
for (i = 0; i < iMax; i++) {
    fpx << dx * i << endl;
}
fpx.close();

// output data for y
ofstream fpy;
fpy.open("y_vals.dat");
if (fpy.fail()) {
    cout << "cannot open file" << endl;
    return (EXIT_SUCCESS);
}
for (j = 0; j < jMax; j++) {
    fpy << dy * j << endl;
}
fpy.close();

// output data for t = 0
ofstream fpTempInit;
fpTempInit.open("000.dat");
if (fpTempInit.fail()) {
    cout << "cannot open file" << endl;
    return (EXIT_SUCCESS);
}
for (i = 0; i < iMax; i++) {
    for (j = 0; j < jMax; j++) {
        fpTempInit << setw(15)
                   << TempInit(i * dx, j * dy, initProfile, material);
        Temp(i, j) = TempInit(i * dx, j * dy, initProfile, material);
    }
    fpTempInit << endl;
}
```

```
cout << "Iteration: 0, Time: 0, Temp @ Center: "
     << Temp((iMax - 1) / 2, (jMax - 1) / 2) << endl;

for (n = 1; n < nMax; n++) {

    currTime = n * dt;
    midTime = (n - 0.5) * dt;

    // begin column-wise sweep
    for (i = 0; i < iMax; i++) {
        for (j = 0; j < jMax; j++) {
            a(j) = -rx;
            b(j) = 1.0 + 2.0 * rx;
            c(j) = -rx;
            if (j == 0) {
                d(j) = (1.0 - 2.0 * ry) * Temp(i, j) - ry * Temp(i, j + 1) -
                       dt * S(midTime, sourceType, t, qMax) / (rho * Cp);
            } else if (j == jMax - 1) {
                d(j) = -ry * Temp(i, j - 1) +
                       (1.0 - 2.0 * ry) * Temp(i, j) -
                       dt * S(midTime, sourceType, t, qMax) / (rho * Cp);
            } else {
                d(j) = -ry * Temp(i, j - 1) +
                       (1.0 - 2.0 * ry) * Temp(i, j) - ry * Temp(i, j + 1) -
                       dt * S(midTime, sourceType, t, qMax) / (rho * Cp);
            }
        }

        // call to tridiag
        tridiag<double>(a, b, c, d);
        for (j = 0; j < jMax; j++) {
            Temp(i, j) = d(j);
        }
    }

    // begin row-wise sweep
    for (j = 0; j < jMax; j++) {
        for (i = 0; i < iMax; i++) {
            a(i) = -ry;
            b(i) = 1.0 + 2.0 * ry;
            c(i) = -ry;
            if (i == 0) {
                d(i) = (1.0 - 2.0 * rx) * Temp(i, j) - rx * Temp(i + 1, j) -
                       dt * S(currTime, sourceType, t, qMax) / (rho * Cp);
```

```
                } else if (i == iMax - 1) {
                    d(i) = -rx * Temp(i - 1, j) +
                           (1.0 - 2.0 * rx) * Temp(i, j) -
                           dt * S(currTime, sourceType, t, qMax) / (rho * Cp);
                } else {
                    d(i) = -rx * Temp(i - 1, j) +
                           (1.0 - 2.0 * rx) * Temp(i, j) - rx * Temp(i + 1, j) -
                           dt * S(currTime, sourceType, t, qMax) / (rho * Cp);
                }
            }

            // call to tridiag
            tridiag<double>(a, b, c, d);
            for (i = 0; i < iMax; i++) {
                Temp(i, j) = d(i);
            }
        }
        if (n == saves(tData)) {
            cout << "Iteration: " << n << ", Time: " << currTime
                 << ", Temp @ Center: " << Temp((iMax - 1) / 2, (jMax - 1) / 2)
                 << endl;
            // output data at different times to be used for animation
            ofstream fpTemp;
            fpTemp.open(to_string(n) + ".dat");
            if (fpTemp.fail()) {
                cout << "cannot open file" << endl;
                return (EXIT_SUCCESS);
            }

            for (i = 0; i < iMax; i++) {
                for (j = 0; j < jMax; j++) {
                    fpTemp << setw(15) << Temp(i, j);
                }
                fpTemp << endl;
            }
            tData += 1;
        }
    }
    return (EXIT_SUCCESS);
}
```

References

[1] Yogesh Jaluria and K. E. Torrance. *Computational heat transfer*. Series in computational methods in mechanics and thermal sciences. Taylor & Francis, New York, 2nd ed edition, 2003.

[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, editors. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, UK ; New York, 3rd ed edition, 2007. OCLC: ocn123285342.

[3] M. Necati Özışık, Helcio R. B. Orlande, Marcelo Jose Colaco, and Renato Machado Cotta. *Finite difference methods in heat transfer*. Taylor & Francis, CRC Press, Boca Raton, second edition edition, 2017.

[4] Richard H Pletcher, Dale A Anderson, and John C Tannehill. *Computational fluid mechanics and heat transfer*. 2016. OCLC: 1027194464.

[5] Pradip Majumdar. *Computational methods for heat and mass transfer*. 2011. OCLC: 897338030.