The Method of Lines

Rory Fox
Peng Liu

AM3064/AM6015 Written Report

School of Mathematical Sciences
University College Cork
Ireland
April 2023

This report is wholly the work of the author, except where explicitly stated otherwise. The source of any material which was not created by the author has been clearly cited.

Date: 04/04/2023

Signature: Rory Fox, Peng Liu

Contents

Chapter 1

Introduction and Motivation

1.1 What is the Method of Lines?

The Method of Lines (MOL), is a versatile numerical technique pioneered in the early twentieth century, widely employed for solving partial differential equations (PDEs). The method enables the solution to complex problems in a wide range of scientific and engineering disciplines by reducing PDE problems, which may be difficult to evaluate analytically, to solving a system of ordinary differential equations (ODEs), which can be evaluated by a number of established numerical techniques.[1]

In our exploration of the MOL, we will briefly present the history behind the method of lines. We will present several worked examples that showcase its applicability, focusing on the study of heat transfer, which is a field of both historical and contemporary significance to the MOL.

Despite its versatility and widespread use, the MOL has certain drawbacks that can affect its accuracy, stability, and computational efficiency. We will aim to address these and to compare the MOL to other viable alternatives, such as the finite element method (FEM). By addressing these limitations, we aim to provide a well-rounded perspective on the good use of this powerful numerical technique.

Finally, we will show an example of a novel research application using the MOL. We will detail how the conductivity of sand was measured by designing a simulation of the heating of a bucket of sand and comparing to measured experimental data.

1.2 History

The history of the method of lines can be traced back to the early 20th century when mathematicians started to develop numerical techniques for solving PDEs. In the 1920s and 1930s, finite difference methods (FDM) was widely used to discretize PDEs. However, FDM had some limitations, such as difficulty in handling boundary conditions and irregular geometries.[2] The method of lines was first introduced by John Crank and Phyllis Nicolson in their paper published in 1947. [3]

Later the method of lines gained popularity in the 1970s and 1980s due to the availability of powerful computers. The use of high-performance computing allowed the

simulation of more complex systems and the development of more sophisticated numerical methods. The method of lines became widely used in scientific computing, particularly in the field of computational fluid dynamics .In addition, the use of the method of lines expanded to other areas of applied mathematics, such as the numerical solution of wave equations, reaction-diffusion equations, Hamiltonian systems and etc.[4]-[6]

The method of lines was combined with other numerical techniques, such as finite element methods and spectral methods, to create hybrid methods that are more efficient and accurate in the 1990s.[7] In 1992, the method of lines was used to solve the Schrödinger equation in quantum mechanics, by Robert W. Carroll and Lutz Lehmann. This marked the first time that the method of lines was applied to a problem in quantum mechanics.

In the 2000s, the method of lines was extended to solve partial integro-differential equations, which arise in many areas of science and engineering.[8]

In recent years, the method of lines has been combined with other numerical techniques, such as machine learning and artificial intelligence, to create hybrid methods for solving PDEs. These hybrid methods show promise for solving complex PDEs that are difficult to solve by using traditional methods. The method of lines continues to be an active area of research, with ongoing efforts to improve its accuracy, efficiency, and applicability to more complex problems.[9]-[10]

Chapter 2

Worked Examples

2.1 The Maths of the MOL

Rather than a set algorithm, the MOL is a more general approach to solving PDEs, where the defining characteristic is the discretisation of the problem domain into a regular grid of evenly spaced points. A number of **finite difference schemes** (approximations derived from the Taylor Series) can then be used to approximate the solution function and its derivatives at these points.

With this in mind, one way to explain how the MOL works is to work through some illustrative examples of its implementation. Crank and Nicholson's paper dealt extensively with the solution of complex heat diffusion problems. Crank went on to spend much of his career studying numerical solutions to heat diffusion problems, publishing his book "The Mathematics of Diffusion" in 1975 [16]. It therefore seemed appropriate to select examples from the field of heat transfer.

2.2 1-D Simple Heat Equation

The simple heat equation in 1D is given by:

$$\frac{\partial U}{\partial t} = \alpha \frac{\partial^2 U}{\partial x^2} \tag{2.1}$$

$$U(0,t) = U(L,t) = 0$$
$$U(x,0) = f(x)$$

Where U is the temperature of the rod, and α is a quantity known as the diffusivity of the medium, which is closely related to its thermal conductivity.

This is a special case of the general heat equation, also known as Fourier's law, detailed in his 1822 paper "La Théorie analytique de la chaleur" [15]

Consider an infinitesimally thin rod of length L. The temperature of the rod $U(x,t)$ is plotted on the y-axis. The figure shows the initial temperature distribution of the rod

$f(x)$, which is a Gaussian distribution centered on the middle of the rod. Given the initial temperature distribution of the rod, we would like to examine how the temperature distribution evolves with time. We are assuming Dirichlet boundary conditions, which means that at the boundaries of the rod, the solution takes a fixed value (in this case zero) for all time.

To solve this problem via the MOL, we begin by discretising the rod into a set of N evenly spaced grid points. The grid spacing is given by $h = \frac{L}{N-1}$ with the points $x_j = jh$ for $j = 0, 1, ..N-1$ At each grid point x_j we define a function $U_j(t) = U(x_j, t)$

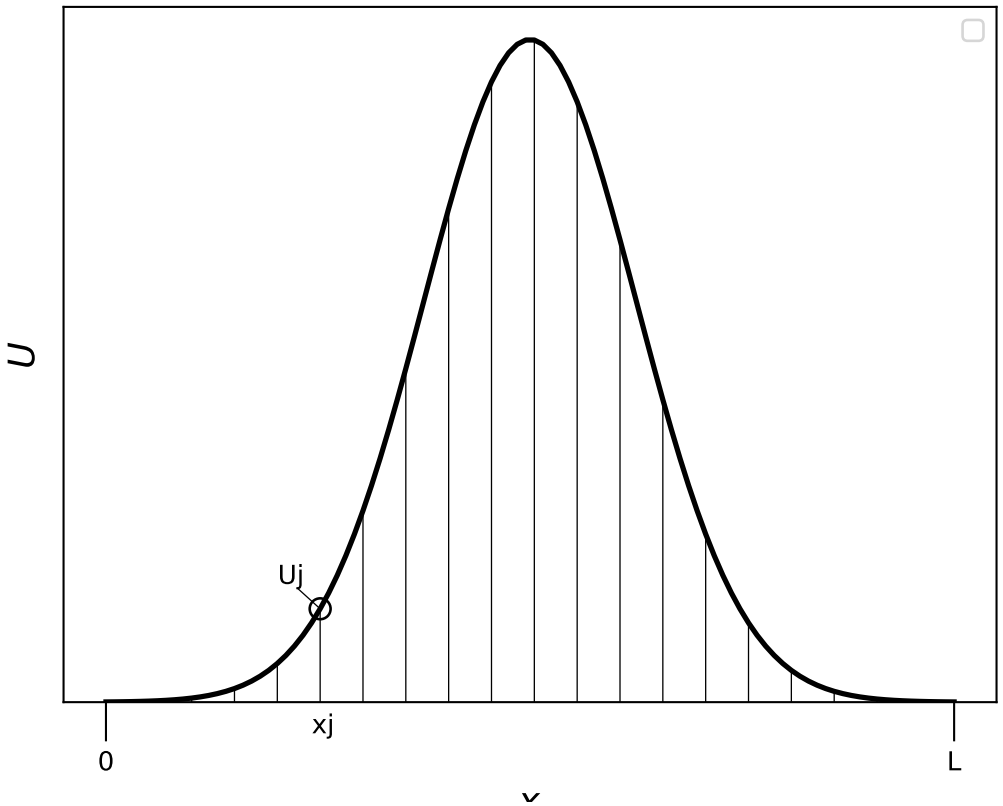


Figure 2.1: Initial temperature distribution of the rod, with value of solution at $U(x_j, t)$ labelled

Having discretised the spatial domain of the problem, we will approximate the partial derivative with respect to x using the central difference approximation:

$$\frac{\partial^2 U(x,t)}{\partial x^2} = \frac{U(x+h,t) - 2U(x,t) + U(x-h,t)}{h^2} \tag{2.2}$$

The central difference scheme can be derived by considering the Taylor Series Expansion of $U(x+h,t)$ and $U(x-h,t)$ about the point x.

$$U(x+h,t) = U(x,t) + \frac{\partial U(x,t)}{\partial x}h + \frac{1}{2!}\frac{\partial^2 U(x,t)}{\partial x^2}h^2 + \frac{1}{3!}\frac{\partial^3 U(x,t)}{\partial x^3}h^3 + ... + \frac{1}{n!}\frac{\partial^n U(x,t)}{\partial x^n}h^n + ...$$

$$U(x-h,t) = U(x,t) - \frac{\partial U(x,t)}{\partial x}h + \frac{1}{2!}\frac{\partial^2 U(x,t)}{\partial x^2}h^2 - \frac{1}{3!}\frac{\partial^3 U(x,t)}{\partial x^3}h^3 + ... + \frac{1}{n!}\frac{\partial^n U(x,t)}{\partial x^n}(-h)^n + ...$$

Assuming the step h to be sufficiently small, we can neglect terms in h of cubic order and higher. Thus, by taking the sum of the Taylor Series about $U(x+h,t)$ and $U(x-h,t)$, the linear terms in h will cancel, and the result can be rearranged to give Eq. (2.2)

If the point x is taken to be the grid point x_j, then the point $x + h = h(j+1) = x_{j+1}$ Similarly, $x - h = h(j-1) = x_{j-1}$

By definition above $U(x_j,t) = U_j(t)$, therefore, inserting Eq. (2.2) with $x = x_j$ into the heat equation Eq. (2.1) gives us an expression for the time derivative of the temperature at each grid point U_j

$$\frac{\partial U_j(t)}{\partial t} = \alpha \frac{U_{j+1}(t) - 2U_j(t) + U_{j-1}(t)}{h^2} \tag{2.3}$$

We can then use any numerical algorithm of our choice to solve the resulting ODEs in time, reducing the original problem from solving a PDE to solving a system of ODEs.

We can represent the system of ODEs by a matrix multiplication.

Let $\vec{u}(t) = \begin{pmatrix} U_1(t) \\ U_1(t) \\ ... \\ U_{N-2}(t) \end{pmatrix}$

The first and last grid points U_0 and U_{N-1} are excluded as they will remain 0 for all time due to the boundary conditions.

The time derivative of $\vec{U}$ can then be written as a tridiagonal matrix:

$$\frac{d}{dt}\begin{pmatrix} U_1(t) \\ U_1(t) \\ \vdots \\ U_{N-3}(t) \\ U_{N-2}(t) \end{pmatrix} = \begin{pmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & 1 & -2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -2 \end{pmatrix} \begin{pmatrix} U_1(t) \\ U_1(t) \\ \vdots \\ U_{N-3}(t) \\ U_{N-2}(t) \end{pmatrix} \tag{2.4}$$

The system of ordinary differential equations (ODEs) is now ready for numerical integration. We've chosen to use the fourth-order Runge-Kutta (RK4) algorithm for this purpose. The RK4 algorithm offers greater accuracy than first or second-order methods like the forward Euler method, as its truncation error is proportional to the step size h raised to the fifth power, which reduces the error significantly, especially for larger time steps, which the RK4 method handles much better than other simpler explicit methods.

Despite its higher accuracy, the RK4 algorithm remains computationally efficient. Although it needs four function evaluations per step, it's still an explicit method, relying only on the previous step's information. Many other accurate ODE solvers use implicit methods, which require solving an additional nonlinear equation that involves both the current and previous time steps. This adds to the computational cost.

All of this means that the RK4 method provides a good balance between accuracy and computational efficiency. As well as this, due to its widespread use, it is included as part of the SciPy package for Python making it an ideal choice for solving this system of ODEs.

The computation was executed in Python, employing Numpy for the rod's discretisation and calculating the matrix-vector product from Eq. (2.4), while utilizing SciPy to incorporate the RK4 algorithm. The simulation results are displayed below.

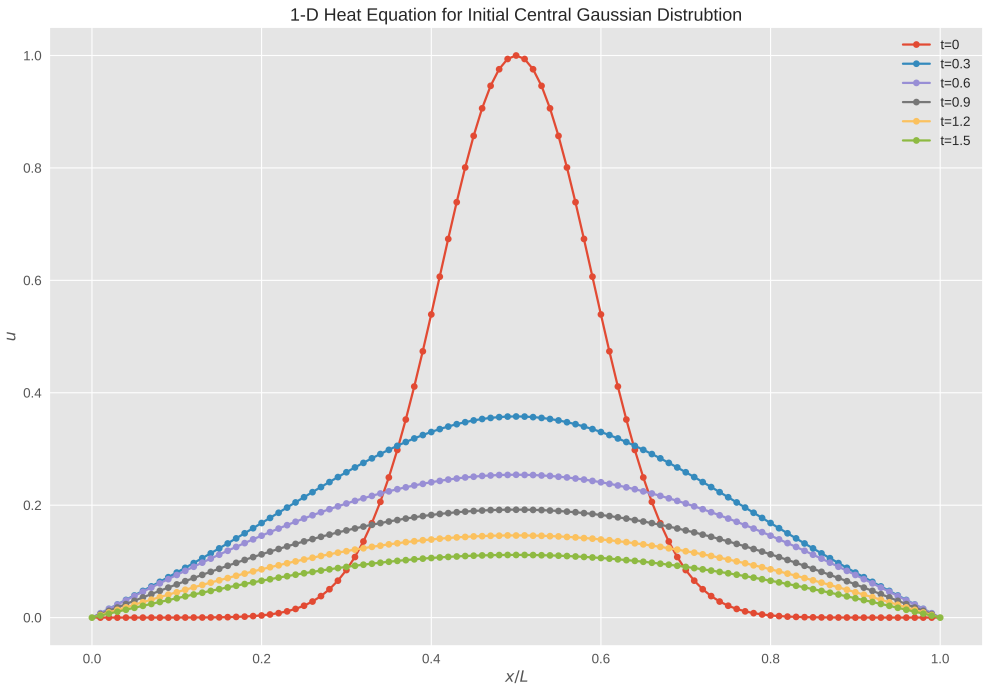


Figure 2.2: Evolution of Gaussian temperature distribution in time, diffusivity: 0.09, timestep: 0.001s, gridpoints: 100

The results of the simulation show the large central peak decreasing over time, as well as increasing in width. This indicates the transfer of heat away from the peak, being instead more evenly distributed over the length of the rod. This simulation agrees qualitatively with the diffusion of heat through a region of homogeneous diffusivity expected from the analytic solution. [16]

While this 1D case is a basic example, which can even be solved analytically (see Chapter 3) , it nevertheless effectively demonstrates the Method of Lines (MOL) applicability. The developed techniques can be readily extended to tackle more complex problems.

2.3 The MOL in Higher Dimensions - 2D Heat Equation

Despite the name, the MOL is not limited to merely solving problems along lines in 1-D, the concept generalises to any discretisation of the spatial domain of a problem

along a regular grid of points. We will demonstrate this by extending our earlier example of the 1-D heat equation to solving the 2-D heat equation on a square surface.

$$\frac{\partial U}{\partial t} = \alpha\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}\right) \tag{2.5}$$

$$\begin{aligned} U(0,y,t) &= U(L,y,t) = 0 \\ U(x,0,t) &= U(x,L,t) = 0 \\ U(x,y,0) &= f(x,y) \end{aligned}$$

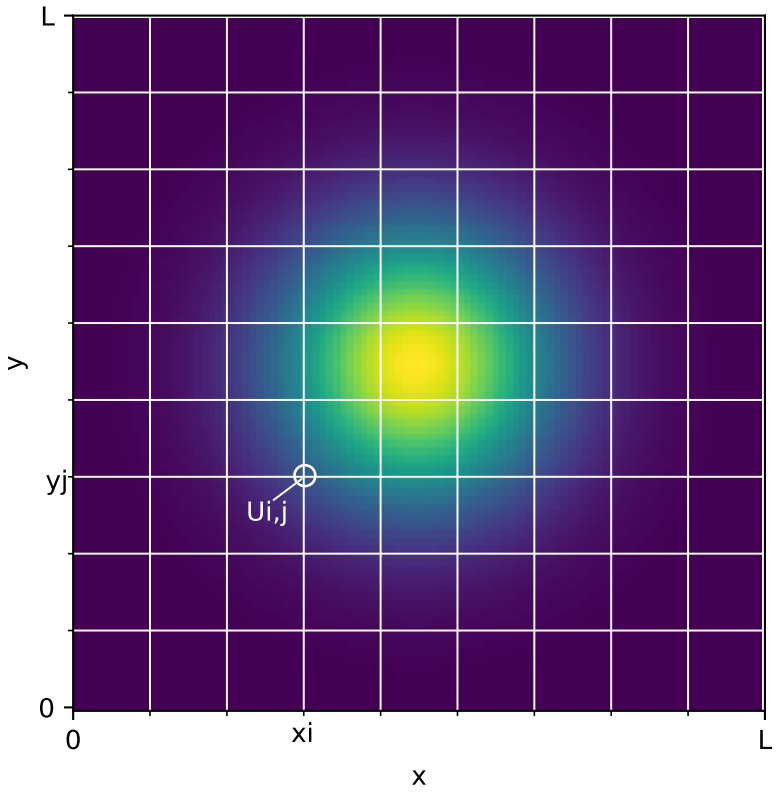


Figure 2.3: Value of temperature is evaluated at each node on the grid

Consider the following square surface of side length L, the initial temperature distribution is a 2D Gaussian. Similar to the 1D example, we begin by discretising the spatial domain, this time with a square grid. The grid spacing along each axis is given by $h = \frac{L}{N-1}$ with the x and y components $x_i = ih$ and $y_j = jh$ for $i, j = 0, 1, ..N-1$ At each grid point we define a function $U_{i,j}(t) = U(x_i, y_j, t)$ as shown in the diagram.

We will use the 2-D central difference scheme to approximate the two spatial derivatives. This is derived similarly to the 1D case, by considering the sum of the Taylor expansions about some point (x_i, y_j) of $U(x_i + h, y_j, t)$, $U(x_i - h, y_j, t)$, $U(x_i, y_j + h, t)$, $U(x_i, y_j - h, t)$ and neglecting the cubic terms and higher in h.

We defined the grid spacing h such that $x_i + h = x_{i+1}$, with similar expressions for $x_i - h$ and $y_j \pm h$, we can therefore write the 2-D central difference scheme in terms of the temperature at our known grid points

$$\frac{dU_{i,j}(t)}{dt} = \alpha \frac{U_{i+1,j}(t) + U_{i,j+1} - 4U_{i,j}(t) + U_{i-1,j}(t) + U_{i,j-1}}{h^2} \tag{2.6}$$

We will again cast the problem of computing the time derivatives at each grid point as a matrix multiplication.

First, the grid of solution points $U_{i,j}(t)$ is flattened in row order to a 1D vector, with the form:

$$\vec{u}(t) = \begin{pmatrix} U_{0,0}(t) \\ U_{1,0}(t) \\ \vdots \\ U_{N-1,0}(t) \\ U_{0,1}(t) \\ U_{1,1}(t) \\ \vdots \\ U_{N-1,N-1}(t) \end{pmatrix}$$

The matrix representation of the central difference scheme takes a similar form to before, with additional diagonals representing the dependence on $U_{i,j+1}$ and $U_{i,j-1}$

$$\frac{d}{dt}\begin{pmatrix} U_{0,0}(t) \\ U_{1,0}(t) \\ \vdots \\ U_{N-1,0}(t) \\ U_{0,1}(t) \\ U_{1,1}(t) \\ \vdots \\ U_{N-1,N-1}(t) \end{pmatrix} = \begin{pmatrix} -4 & 1 & 0 & \cdots & 1 & \cdots & \cdots & 0 \\ 1 & -4 & 1 & 0 & \cdots & 1 & \cdots & 0 \\ 0 & 1 & -4 & 1 & 0 & 0 & \cdots & 0 \\ \vdots & 0 & 1 & -4 & 1 & 0 & \cdots & 1 \\ 1 & 0 & \cdots & 1 & -4 & 1 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 1 & -4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & \cdots & 0 & \cdots & -4 \end{pmatrix} \begin{pmatrix} U_{0,0}(t) \\ U_{1,0}(t) \\ \vdots \\ U_{N-1,0}(t) \\ U_{0,1}(t) \\ U_{1,1}(t) \\ \vdots \\ U_{N-1,N-1}(t) \end{pmatrix} \tag{2.7}$$

The boundary conditions in this case were enforced by zeroing the elements of the vector corresponding to the edges of the square at the end of each time step.

The calculation was again performed in Python, using the RK4 algorithm.

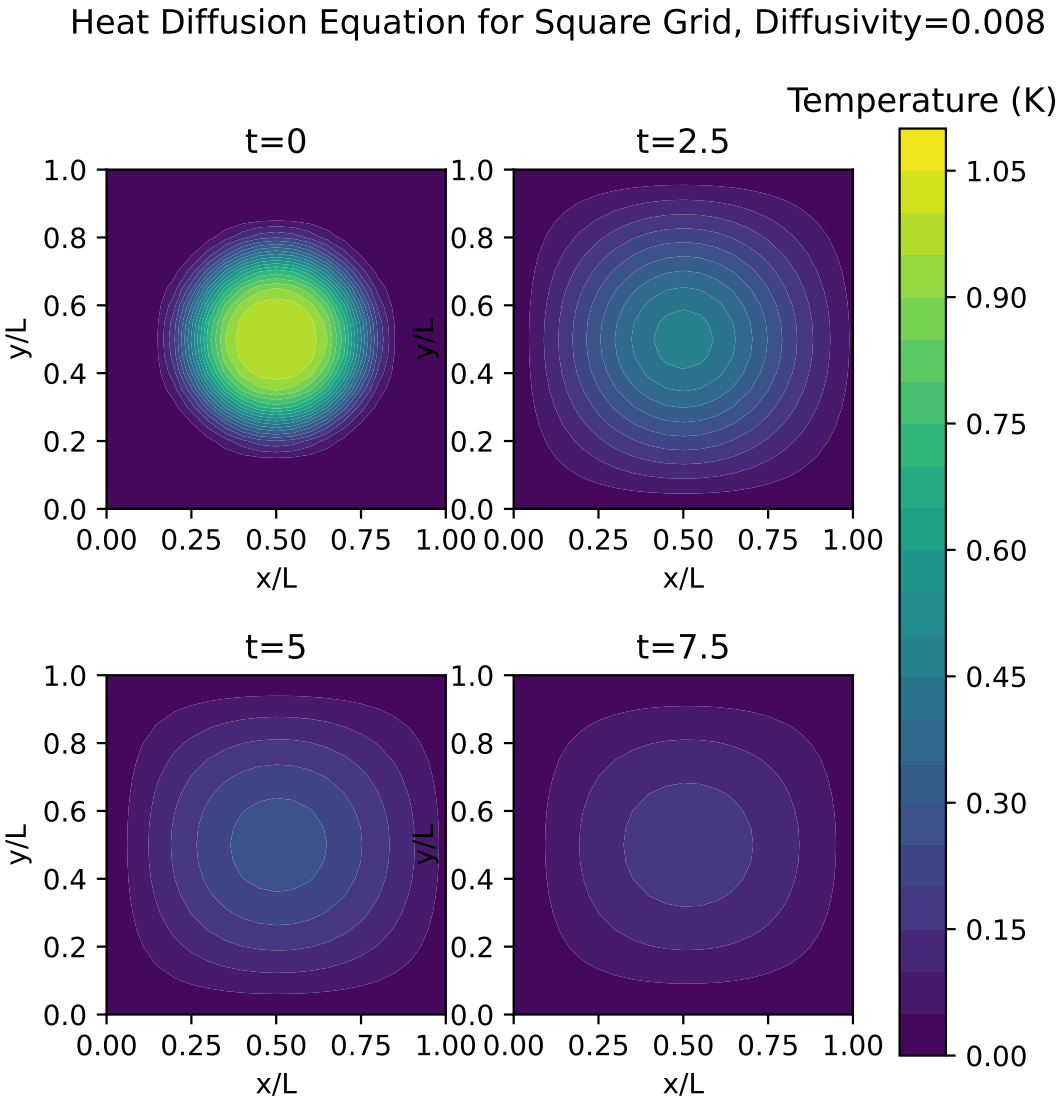


Figure 2.4: 2D Heat Equation with initial Gaussian distribution (mu: 0.5,0.5 sigma: 0.05), time step: 0.005s, grid size: (20x20)

Chapter 3

Limitations of the MOL & Comparison with related algorithms

3.1 Limitations of the MOL

While the Method of Lines (MOL) is a powerful and versatile numerical method that can be applied to a wide range of problems, it has some limitations that should be taken into account when choosing a numerical method. One limitation is that the MOL can be computationally expensive, particularly for problems with high spatial dimensions or complex geometries, as it requires the solution of a large system of ordinary differential equations at each time step.

3.2 Implementation of 1-D heat equation in python

Common methods for solving partial differential equations include finite element method, spectral method, boundary element method, finite volume method, etc. This report chooses finite element method and makes a simple comparison with the line method, based on the one-dimensional heat equation.

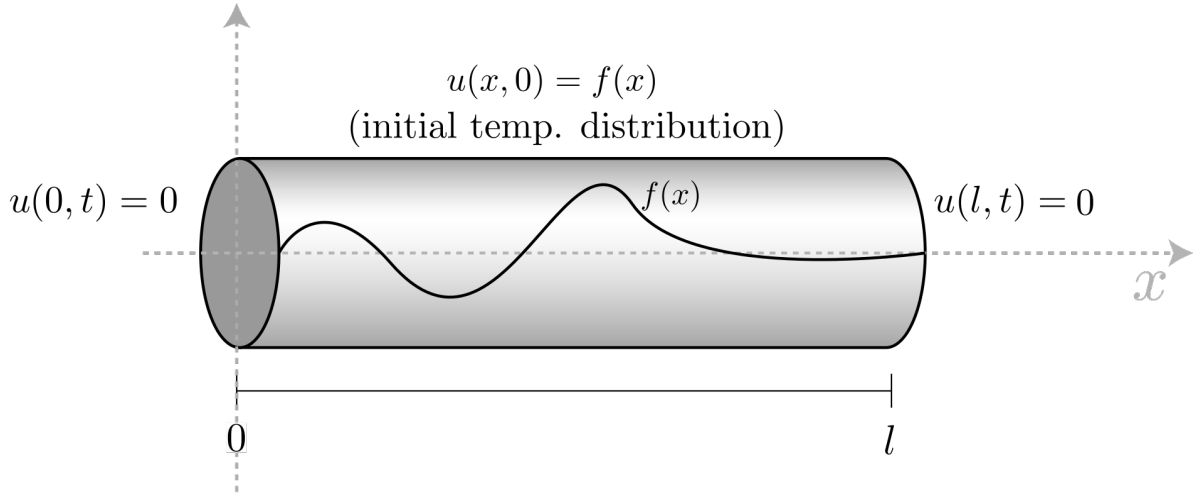


Figure 3.1: 1D heat equation model

1)MOL:

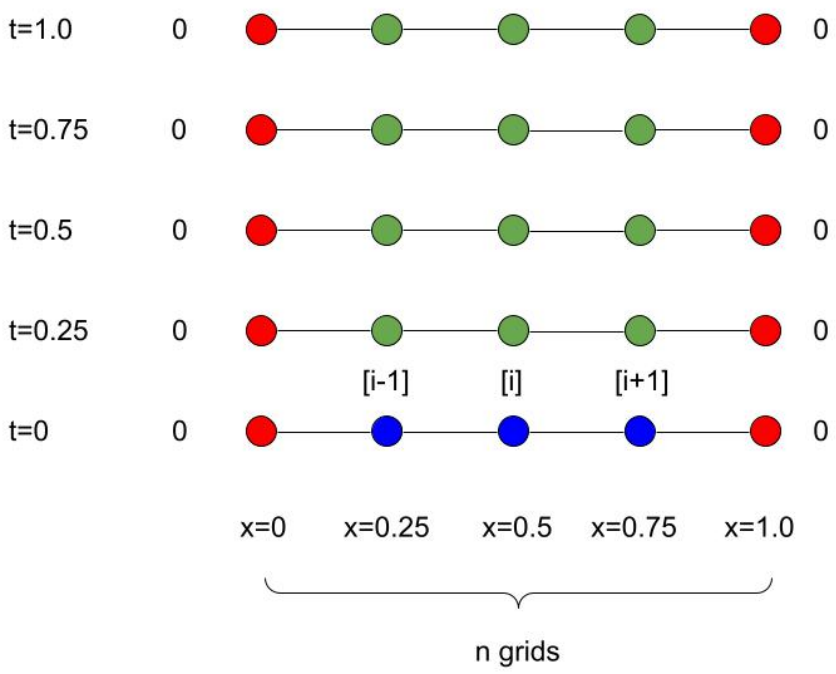


Figure 3.2: Discretization of MOL in a rod

As mentioned before, MOL uses a numerical method called finite differential method. Here we use central differential method, which is a special kind of finite differential method.

The Eq. (2.3) represents the heat equation in one dimension and can be used to simulate the behavior of temperature as it diffuses over time in a one-dimensional space. The resulting set of n ODEs is then solved using the 'solve ivp' function, which integrates the equations in time using a specified ODE solver. The initial temperature profile 'T0' is used as the initial condition for the ODE system.[11]

'solve ivp' solver uses the solution in the previous time step to find the solution in the next time step. Specifically, it uses the Runge-Kutta method which is the default method to numerically integrate the system of differential equations over time. At each time step, the method evaluates the derivative of the solution at the current time step and uses this information to predict the solution at the next time step.

2)FEM:

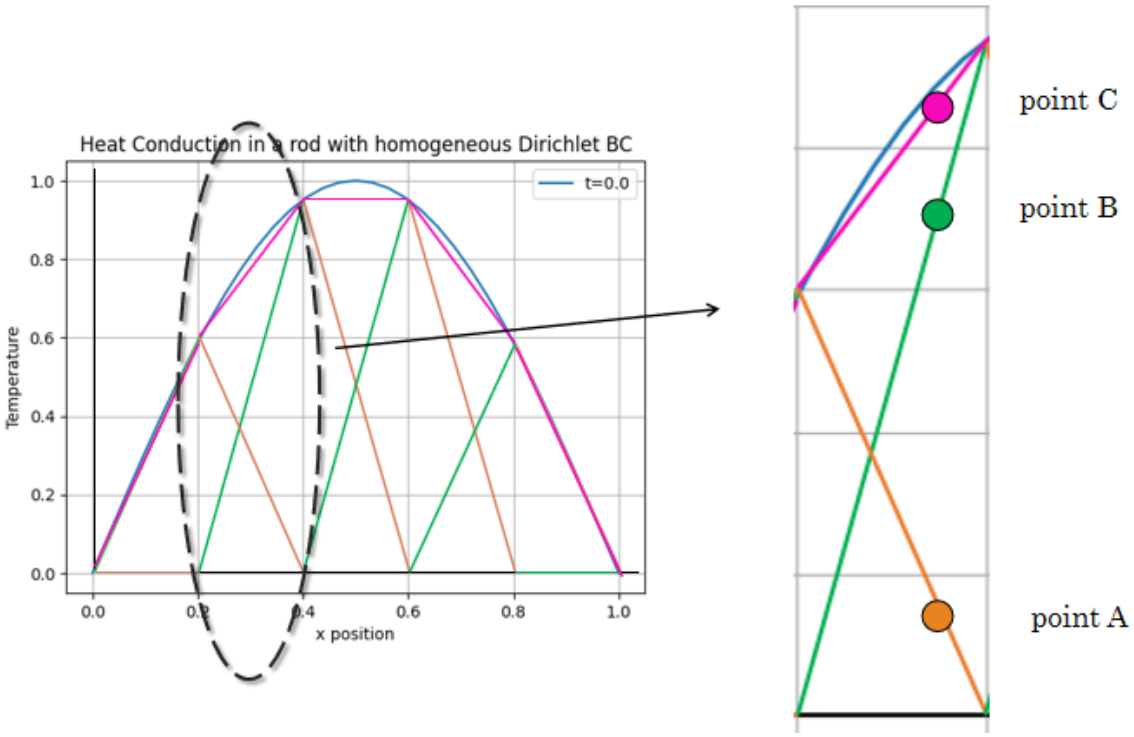


Figure 3.3: Discretization of FEM in a rod

FEM subdivides the domain into finite elements.Each finite element will be outfitted with a shape function. Here it is the superposition of two linear functions. For example, you get point A and point B, and add it up you get point C to approximate the solution. So when you approximate all solutions you will get a set of straight lines. This should then resemble the parabola which is the hypothetical analytical solution of 1-D heat equation. Of course these solution lines seems not perfect but you can imagine that if you have more elements it will be smoother. [12]-[13]

In fact, this is just one of probably hundreds ways of choosing finite elements. It's a type of finite elements which are called Lagrange element. 'Fenics', a popular finite element solver based on Python can be used to implement it. we uses a first-order Lagrange polynomial space and the homogeneous Dirichlet boundary condition.

In the Method of Lines (MOL), the spatial domain is typically discretized into a grid of points, whereas in the Finite Element Method (FEM), the domain is typically discretized into a mesh of small elements, such as triangles or quadrilaterals in 2D or tetrahedra or hexahedra in 3D.

3.3 Accuracy

We also investigated the accuracy of both methods via the L2 error which is computed by exact solution and numerical solution.

1)What is L2 error

L2 error is a measure of the difference between an estimated or predicted value and the true or actual value of a quantity of interest. It is commonly used in mathematics,

statistics, and machine learning to quantify the accuracy of a model or estimation.

$$L2error = \frac{1}{n}\sum_{n=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.1}$$

where n is the number of data points; yi is the actual value for the ith data point; ŷi is the predicted value for the ith data point.

2)Exact solution[14]

When it comes to exact solution, you may have doubts. The heat equation is a partial differential equation, and our goal is to approximate the solution of this problem. So how can it have an analytical solution? It seems a bit contradictory.

It is true that the 1D heat equation is a partial differential equation, and in general, there may not be an analytical solution to the equation. However, in some cases, an analytical solution may be available for specific initial and boundary conditions.

For example, in our experiment, the initial temperature profile is defined as a sin function. If the boundary conditions are also well-defined, then an analytical solution for this particular combination of initial and boundary conditions can be derived using separation of variables or other techniques. Then,let's see the process of generation of exact solution. The heat equation is given by formula Eq. (2.1).

First, by using the method of separation of variables we can assume that the solution has a separable form where X(x) depends only on position and T(t) depends only on time.

$$U(x,t) = X(x)T(t) \tag{3.2}$$

where X(x) depends on position x and T(t) depends on time t

Next, Substitute the expression Eq. (3.2). into the heat equation, we obtain formula:

$$\frac{1}{T(t)}\frac{\partial T}{\partial t} = \alpha\frac{1}{X(x)}\frac{\partial^2 X}{\partial x^2} \tag{3.3}$$

Because both sides depend only on one variable, they must be equal to a constant which is denoted by minus squared lambda.

$$\frac{1}{T(t)}\frac{\partial T}{\partial t} = -\lambda^2 \tag{3.4}$$

$$\alpha\frac{1}{X(x)}\frac{\partial^2 X}{\partial x^2} = -\lambda^2 \tag{3.5}$$

Then solve each equation separately by reasonable assumption

$$T(t) = Ae^{(-\alpha\lambda t)} \tag{3.6}$$

where A is a constant determined by the initial condition.

$$X(x) = B\sin(\lambda x) + C\cos(\lambda x) \tag{3.7}$$

where B and C are constants determined by the boundary conditions.

Combine the time and spatial solutions:

$$U(x,t) = (B\sin(\lambda x) + C\cos(\lambda x))Ae^{(-\alpha\lambda t)} \tag{3.8}$$

Assuming the initial temperature distribution is given by f(x), apply the initial and boundary conditions to determine the constants B, C, and A we have

$$U(x,0) = f(x) = (B\sin(\lambda x) + C\cos(\lambda x) \tag{3.9}$$

And we have sine coefficient Bn and cosine coefficient Cn by using Fourier series

$$B_n = \frac{2}{L}\int f(x)\sin(\frac{n\pi x}{L})dx \tag{3.10}$$

$$C_n = \frac{2}{L}\int f(x)\sin(\frac{n\pi x}{L})dx \tag{3.11}$$

Last, find the exact solution by summing over all possible values of λ

$$U(x,t) = \sum[(B_n\sin(\frac{n\pi x}{L}) + C_n\cos(\frac{n\pi x}{L}))e^{\frac{-n^2pi^2\alpha t}{L^2}}] \tag{3.12}$$

3.4 Result

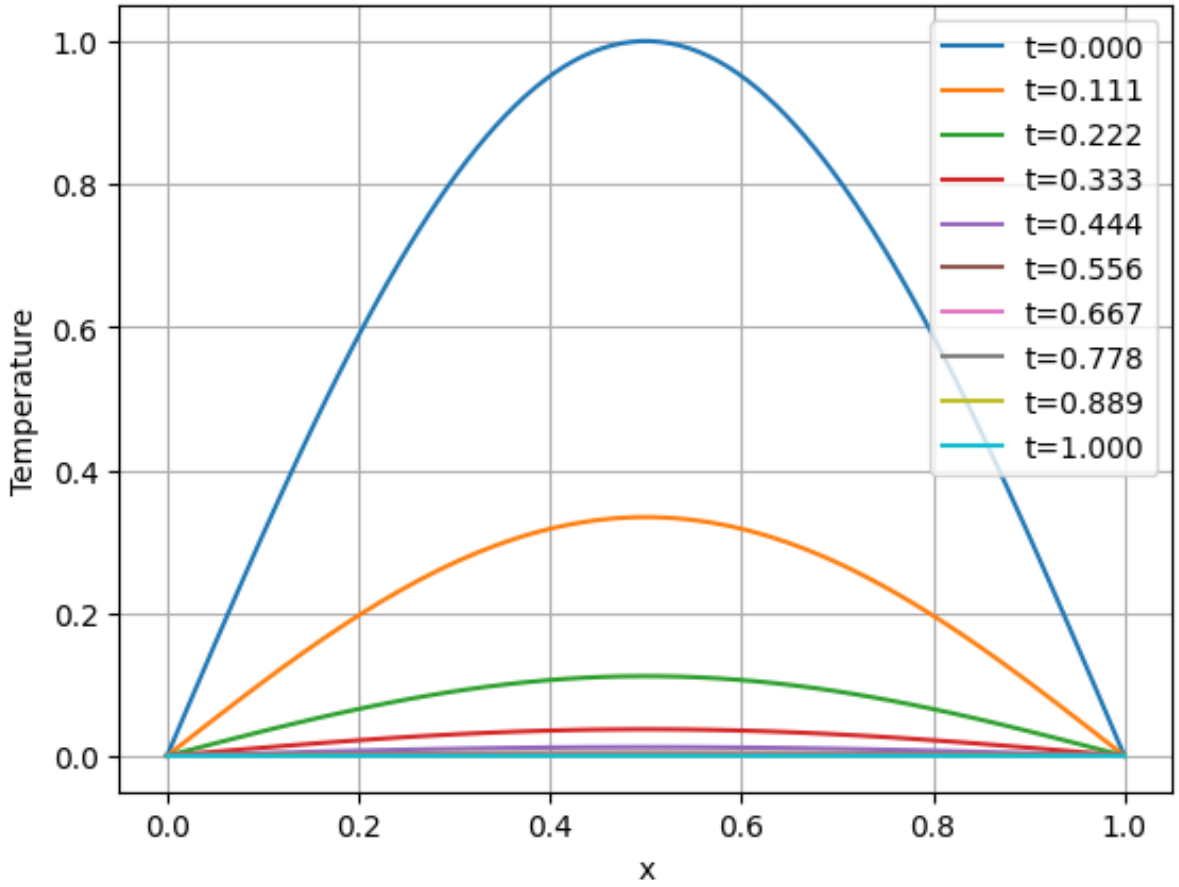


Figure 3.4: Solution of MOL

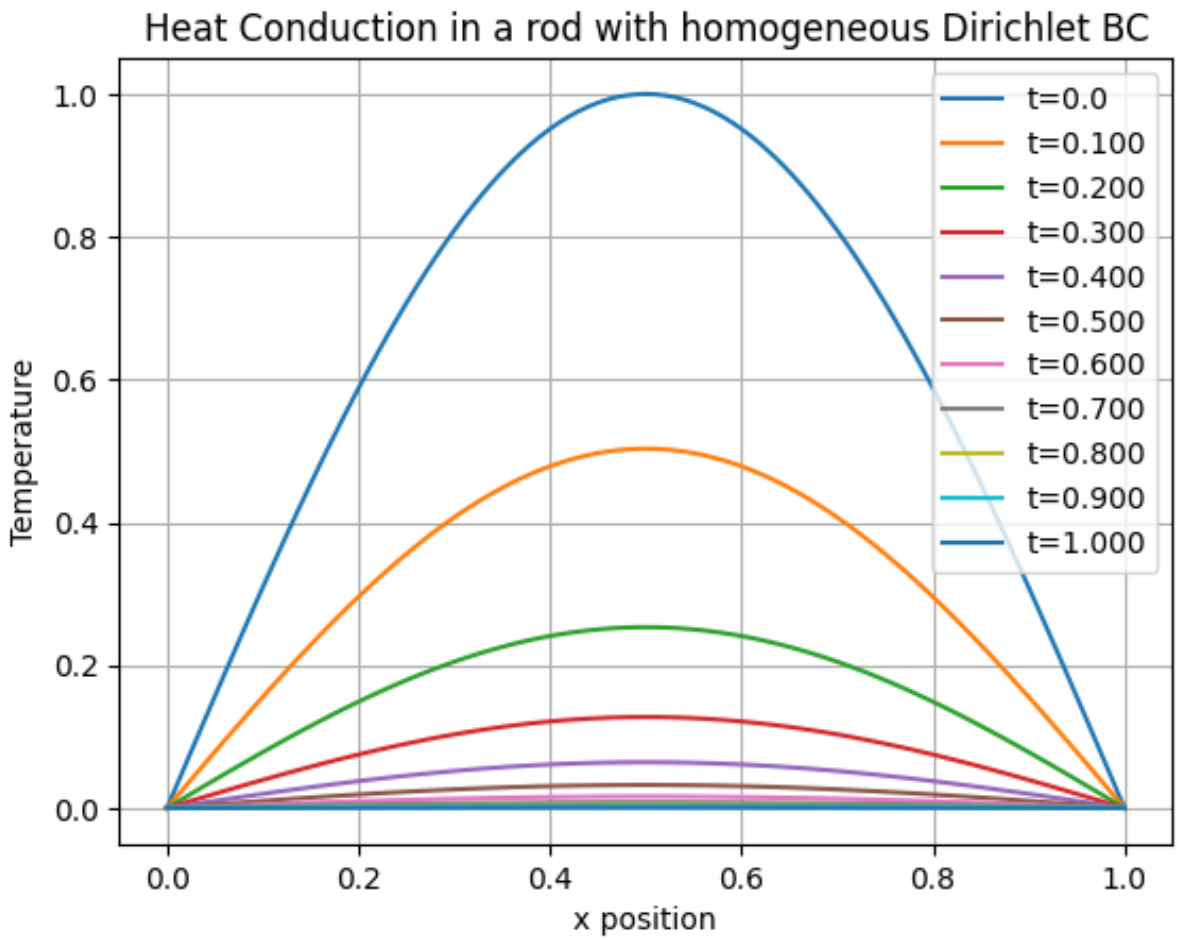


Figure 3.5: Solution of FEM

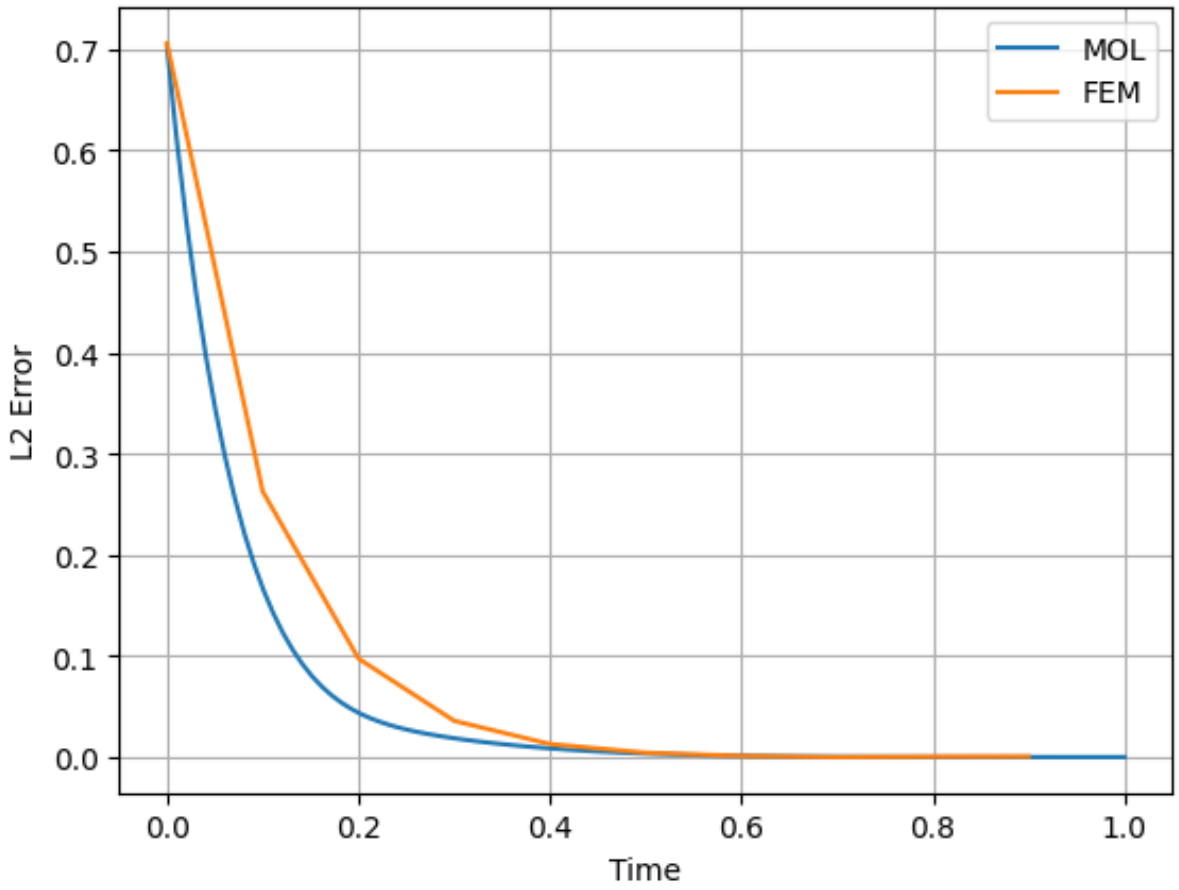


Figure 3.6: Comparison of L2error in two methods

It can be seen from the results that as time goes by, the L2 error of the two methods gradually converges to 0.

In the case of the Finite Element Method (FEM), the solution is approximated by piecewise polynomials defined over a finite element mesh. As we refine the mesh by increasing the number of elements, the accuracy of the solution increases, leading to a decrease in the L2 error.

In the case of the Method of Lines (MOL), the time domain is discretized into a set of time steps. As we decrease the time step size, the accuracy of the solution increases, leading to a decrease in the L2 error.

However, in the current implementation of the two methods, the time step size and the number of elements are fixed, which means that the error reduction is mainly due to the effect of numerical diffusion. Numerical diffusion is an inherent property of numerical methods that causes a smoothing of the solution, which can lead to a slower decrease in the error compared to the exact solution.

In this case, the error decrease is likely to be slower for FEM because it involves more complex calculations and introduces more numerical diffusion than MOL. Therefore, we expect the L2 error to decrease more slowly for FEM than MOL, which is consistent with the results obtained.

Chapter 4

Assessing Sand Conductivity Using the Method of Lines: A New Experimental Approach

Up to this point, we have demonstrated the method of lines (MOL) by providing worked examples of solving for the temperature distributions of various straightforward geometries. These basic cases were used to illustrate the MOL's main approaches and concepts. Nonetheless, the technique can also be employed for more complicated problems.

In a physics experiment conducted recently as part of another module, a cylindrical bucket filled with sand was heated by a heating element, and the radial and axial temperature profiles were recorded. By using this data, along with knowledge about the power supplied to the heating element, an analytical approximation was used to determine the sand's conductivity. However, this method has several sources of error and necessitates the sand being heated for several hours (more than six for ideal results) to achieve a steady state temperature, which is challenging to achieve in a laboratory environment with time constraints. Additionally, the analytical solution supposes an infinitely long heating element, disregarding the axial temperature dependence, whereas in actuality, the temperature varies considerably at various distances along the heating element's axis.

On the other hand, by simulating the sand's temperature in the container and adjusting the conductivity as a fitting parameter in the simulation, the sand's conductivity can be determined. This approach has numerous advantages over traditional analytical methods for determining conductivity, as it provides an accurate estimate without requiring an extended heating period. It also accounts for the finite length of the heating element, providing a more realistic image of the dependence of the temperature on axial depth in the bucket.

Before simulation, we had to establish the final parameters of the model. In earlier examples, after the inital conditions had been set, there were no factors affecting the temperature except for the diffusion of heat according to Fourier's law. However, in this experiment, heat was introduced at a constant rate due to joule heating in the metal element. This was accounted for by modelling the heating element as a sepa-

rate medium of higher diffusivity to the sand. Inside the element, as well as obeying Fourier's law, an extra term was introduced to the rate of increase of temperature with time, accounting for the electrical heating in this material.

$$\text{In sand: } \frac{\partial U}{\partial t} = \alpha_s\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + \frac{\partial^2 U}{\partial z^2}\right) \tag{4.1}$$

$$\text{In element: } \frac{\partial U}{\partial t} = \alpha_m\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + \frac{\partial^2 U}{\partial z^2}\right) + c_m\frac{dQ}{dt} \tag{4.2}$$

Where c_m is the heat capacity of the wire, α_s is the diffusivity of the sand and α_m is the diffusivity of the element and $\frac{dQ}{dt}$ is the rate of heat dissipation by the heating element.

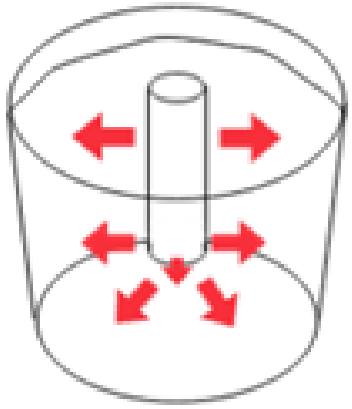


Figure 4.1: Diagram of Model setup, showing heating element of finite length

The extra term was taken to be a constant, as according to Joule's law the heat energy dissipated in the element is proportional to the electrical current density in the element, which is uniform [17]

The current (I) and voltage (V) flowing in the heating element were held constant throughout the heating of the sand, their values were measured to be $I = 0.89A$ and $V = 31.8V$ respectively [18]

Using Joule's Law, $P = VI$, the rate of heat energy dissipation in the element was determined to be $VI = (31.8)(0.89) = 28.3W$

The mass of the heating element was $0.076kg$, it is made from Nichrome, which has a specific heat of $450JKg^{-1}K^{-1}$, the value of the extra term in 4.2 inside the heating element is $\frac{28.3}{0.076*450} = 0.827Ks^{-1}$

Similar to the 1D and 2D examples, the central difference scheme is used to approximate the partial derivatives in the spatial co-ordinates. The functions at each grid point were defined as $U_{i,j,k}(t) = U(x_i, y_j, z_k, t)$ with $x_i = ih$, $y_j = jl$, $z_k = km$, where h,l,m are the grid spacings along the x,y and z axes respectively. This leads to the following central difference schemes for the sand and heating element:

$$\text{In sand: } \frac{dU_{i,j,k}(t)}{dt} = \alpha_s \frac{U_{i+1,j,k}(t) - 2U_{i,j,k}(t) + U_{i-1,j,k}(t)}{h^2} + \alpha_s \frac{U_{i,j+1,k}(t) - 2U_{i,j,k}(t) + U_{i,j-1,k}(t)}{l^2} + \alpha_s \frac{U_{i,j+1,k}(t) - 2U_{i,j,k}(t) + U_{i,j-1,k}(t)}{m^2} \quad (4.3)$$

$$\text{In element: } \frac{dU_{i,j,k}(t)}{dt} = \alpha_m \frac{U_{i+1,j,k}(t) - 2U_{i,j,k}(t) + U_{i-1,j,k}(t)}{h^2} + \alpha_m \frac{U_{i,j+1,k}(t) - 2U_{i,j,k}(t) + U_{i,j-1,k}(t)}{l^2} + \alpha_m \frac{U_{i,j+1,k}(t) - 2U_{i,j,k}(t) + U_{i,j-1,k}(t)}{m^2} + 0.827 \quad (4.4)$$

Unlike previous examples, the ODEs in this model were not expressed as a matrix multiplication, instead the values of $U_{i,j,k}$ were stored in a 4D numpy array (x,y,z,t) and operated on element wise.

Due to the increased number of total grid points, the RK4 algorithm was not used in this case. The forward Euler algorithm was thought to be more appropriate as it requires only one function evaluation to the 4 needed for RK4.

The calculation was carried out in Python, and the following results were obtained.

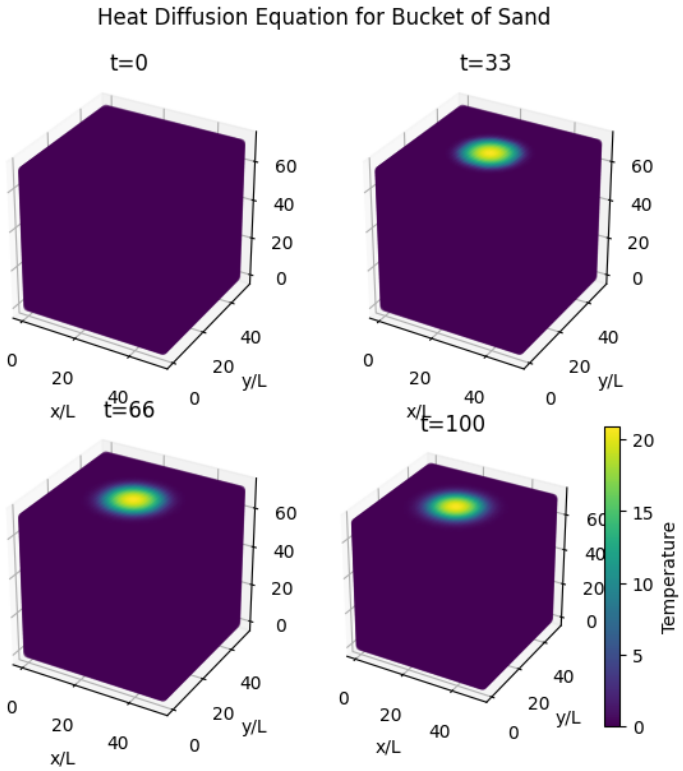


Figure 4.2: Python simulation of first 100 seconds of heating, z=60mm

While the model appeared to replicate the observed behaviour of the sand in the bucket, it had some major limitations, which highlight some of the wider issues with

the MOL. To adequately capture the cylindrical geometry, a large number of grid points is necessary. To hold this large number of points in memory is not feasible for large timescales. Indeed, we were unable to simulate beyond 700 seconds by this method. At the time of the experiment, the sand had been heating for two hours (7200 seconds). Therefore, to be able to make comparisons between the experiment and the simulation, longer timescales were needed and thus the memory issue had to be addressed.

Another major issue of the MOL is that due to the large number of grid points which had to be evaluated at each time step, the execution times were very long (around 20 minutes for 100 seconds of simulation). Many of these grid points were outside of the region of interest of the cylindrical bucket, but were obliged to be calculated due to the nature of the Cartesian grid typically used to discretise the spatial domain of a problem when applying the MOL.

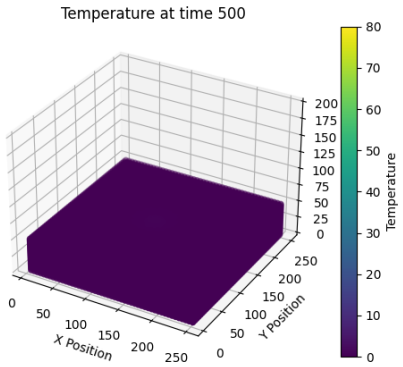
To improve the performance of the simulation, it was rewritten in C++, which is known to have better performance for computing many iterations with nested for loops. The C++ code was written in such a way as to allow for concurrent execution on all four of the cores of the CPU, this speeds up the simulation massively, indeed, this ability to be executed concurrently is a massive advantage of the MOL.

As well as this, the memory issue was circumvented by realising that the forward Euler method requires only that the previous time step be kept in memory at any time. Therefore, only one iteration is stored at any one time. The temperature data is written to the disk only once for every twenty seconds of simulation, this improves performance as well as reducing the amount of unnecessary data which must be stored and managed.

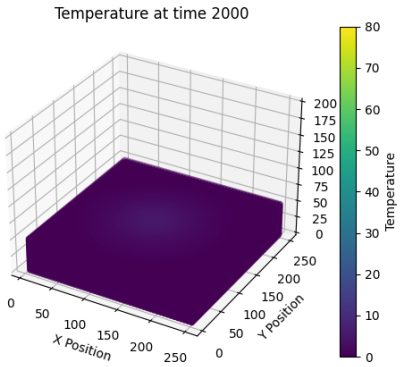
Finally, the scale of the simulation was changed to millimeters, by making this change, the value of the diffusivity did not become so small that floating point error became an issue.

Having made these changes, the performance of the simulation improved dramatically, and it was possible to simulate 7200 seconds of the bucket heating without memory issues.

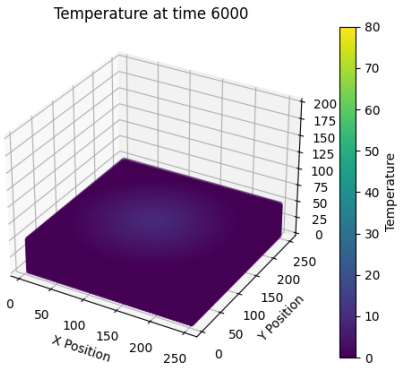
At this point, the dimensions of the bucket of the sand were measured (radius 300mm, height 200mm). The dimensions of the simulation were modified to agree with the physical dimensions to allow comparisons to be drawn between the data.



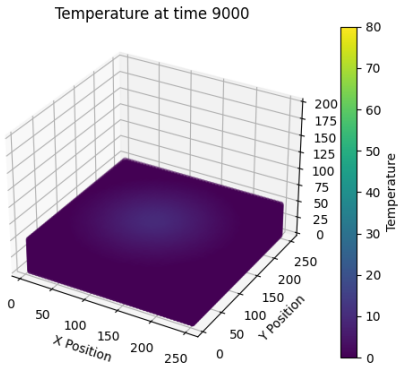
(a) $t = 500$



(b) $t = 2000$

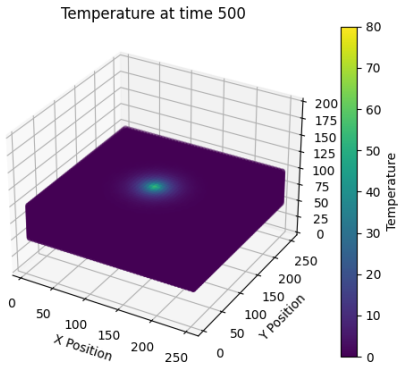


(c) $t = 6000$

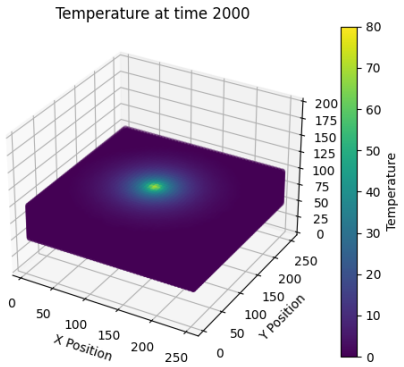


(d) $t = 9000$

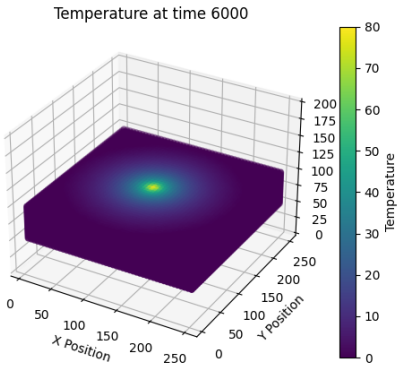
Figure 4.3: Data from z=0 to z=50mm, bottom of the bucket under the heating element



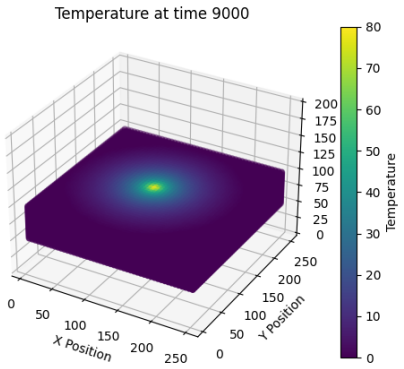
(a) $t = 500$



(b) $t = 2000$

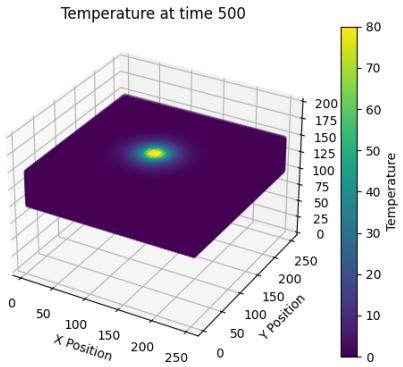


(c) $t = 6000$

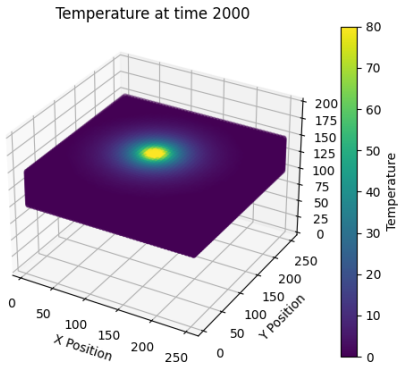


(d) $t = 9000$

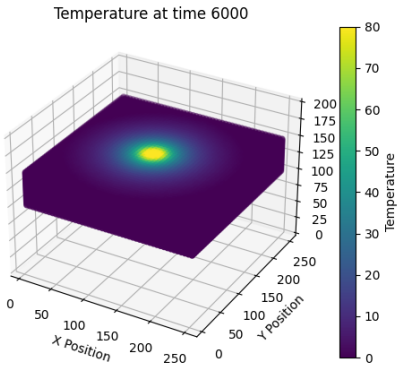
Figure 4.4: Data from z=50 to z=100mm, Sand close to end heating element, with some axial leak of heat observed



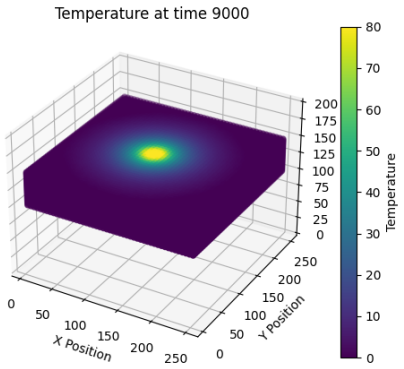
(a) $t = 500$



(b) $t = 2000$



(c) $t = 6000$



(d) $t = 9000$

Figure 4.5: Data from z=100 to z=150mm, Much more heating observed in this layer

The value of the diffusivity α was varied by hand in steps of 0.05 until the best fit between the experimental and simulated data was achieved. The data from the C++ simulations was saved as a .csv and plotted in Python. In this case, 0 in the simulation represents room temperature (approx $19.6°C$, which was the obvious initial condition for the sand in the bucket, the experimentally measured temepratures

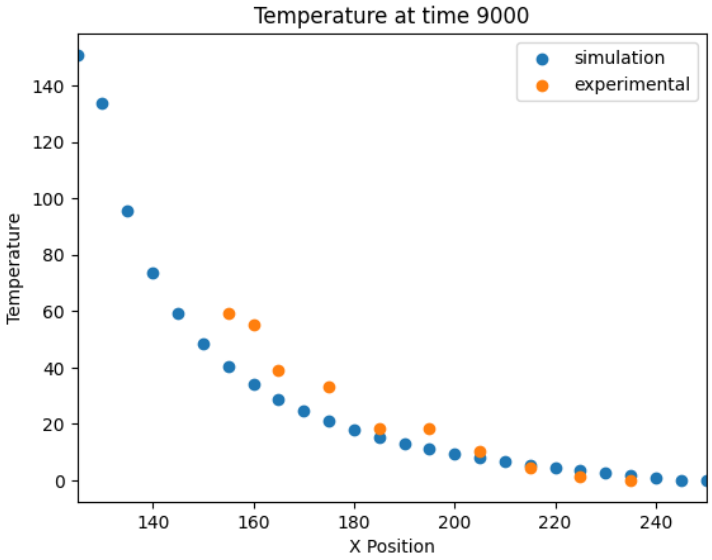


Figure 4.6: Best fit of simulation to experimental data

This fit was achieved for $\alpha = 1$. This corresponds to a conductivity of $1.33 Wm^{-2}$: this is within the range of conductivity's measured for sand in previous research, and it would indicate that the MOL can be used to determine the conductivity of sand in this manner. It was observed that even for the best fit to the experimental data, there is still a large discrepancy between the real and simulated data. While the reason for this is not currently known, using the numerical approach to this problem opens up many new explanations which could not easily be explored analytically. For example, anecdotal evidence from lab technicians reported that the conductivity of the sand is not in fact constant, and instead varies massively with moisture content. This will change during the experiment as in regions of high temperature evaporation will reduce the moisture content. A variable diffusivity is very difficult to capture in a simple analytic model, but could be included in a simulation such as this with some extra thought.

Chapter 5

Conclusions

In this report, we covered how the method of lines provides a simple to use approach to solving PDE problems. We applied the method successfully to solve the 1-D and 2-D simple heat equation, which served as a good foundation to begin work on more complex problems, culminating in our successfully simulating the conduction of heat in a bucket of sand and comparing with experimental data to rapidly determine the conductivity of the sand, which could serve as an interesting alternative to existing analytic solutions.

While exploring these different examples, we covered one of the most common approximations used to discretise problems with spatial derivatives, the central difference scheme as derived from the Taylor Series approximation. We encountered and utilised two popular numerical integration algorithms, the Forward Euler and Runge-Kutta Fourth Order, and briefly compared their respective utilities.

Finally, we covered some of the limitations of the method of lines, especially when applied to complex geometries. We discussed other, potentially more suitable methods for such problems such as the finite element method, which is similar to the FEM but due to its unstructured discretisation of space into a mesh is more suitable for capturing complex curved surfaces. We used the simple 1-D heat equation to compare the performance of the two methods.

The problem of capturing complex geometries became apparent when attempting to accurately capture the cylindrical bucket of sand. This example showed that while there are many problems with the method of lines in such cases (e.g. wasted computation calculating points outside of the region of interest), it is still a viable method for performing such simulations. Recent research has been conducted into using a method of lines style discretisation of space, but utilising adaptive methods to adjust the fineness of the grid points in regions where the local error is high. An approach like this may indeed be the "best of both worlds", and allow the simplicity of the approximations that the method of lines allows to be combined with the geometric complexity enabled by the finite element method, and may be an interesting area for future presentation topics.

Author contributions

Rory wrote Section 1.1, Peng wrote Section 1.2. Rory wrote Chapter 2. Peng wrote Chapter 3. Rory wrote Chapter 4. Chapter 5 was cowritten.

Code in Appendices A,B,D,E,F,G,H was written by Rory. Code in Appendix C was written by Peng

Bibliography

[1] M. N. O. Sadiku and C. N. Obiozor*"A simple introduction to the method of lines"*The International Journal of Electrical Engineering & Education. 2000;37(3):282-296. doi:10.7227/IJEEE.37.3.8

[2] Courant, Richard, Kurt Otto Friedrichs and Hans Lewy. *"On the Partial Difference Equations, of Mathematical Physics."*(2015).

[3] Crank, J., Nicolson, P. *"A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type."*Adv Comput Math **6**, 207–226 (1996).

[4] J.I Ramos. *"A review of some numerical methods for reaction-diffusion equations."*Mathematics and Computers in Simulation,Volume 25, Issue 6,1983,Pages 538-548,ISSN 0378-4754.

[5] Nobuyuki Satofuka, Koji Morinishi, Yusuke Nishida. *"Numerical Solution of Two-Dimensional Compressible Navier-Stokes Equations Using Rational Runge-Kutta Method."*Numerical Simulation of Compressible Navier-Stokes Flows, 1987, Volume 18,ISBN : 978-3-528-08092-1.

[6] Skeel, Robert D. *"Numerical Hamiltonian Problems (J. M. Sanz-Serna and M. P. Calvo)"*SIAM Review 37, no.2 (1995): 277-279.

[7] Claudio Canuto, M. Youssuff Hussaini, Alfio Quarteroni, Thomas A. Zang. *"Fundamentals in Single Domains"*Springer-Verlag Berlin Heidelberg 2006.

[8] Bahuguna, D., and Dabas, J.. *"Existence and uniqueness of a solution to a partial integro-differential equation by the method of lines"*Electronic Journal of Qualitative Theory of Differential Equations [electronic only] 2008 (2008): Paper No. 4, 12 p., electronic only-Paper No. 4, 12 p., electronic only.

[9] Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019)*"Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations"*Journal of Computational Physics, 378, 686–707. doi:10.1016/j.jcp.2018.10.045

[10] Niknam, A. R., Ghorbani, M., & Solaimani, M. (2020)*"Analysis of filamentation instability in a current-carrying plasma using meshless method of lines coupled with radial basis functions"*Physics Letters A, 384(33), 126839. doi:10.1016/j.physleta.2020.126839

[11] Zafarullah, A.(1970)"*Application of the Method of Lines to Parabolic Partial Differential Equations With Error Estimates*"J. ACM, 17(2), 294–302. doi:10.1145/321574.321583

[12] A. K. Aziz and Peter Monk"*Continuous finite elements in space and time for the heat equation*"Math. Comp. 52 (1989), 255-274

[13] Koehler,F.(n.d.)."*Machine Learning and Simulation*"Retrieved from https://github.com/ceyron/machine-learning-and-simulation

[14] Boyce, William E., Richard C. DiPrima, and Douglas B. Meade."*Elementary differential equations and boundary value problems.*"John Wiley & Sons, 2021.

[15] Fourier J, *"La Théorie analytique de Chaleur"*Paris : F. Didot 1822.

[16] Crank, J. (1975). The Mathematics of Diffusion Chapter 2: Methods of Solution When The Diffusion Coefficient is Constant p 12-13 Oxford University Press.

[17] Griffiths, D. (2017). Introduction to Electrodynamics (4th ed.). Cambridge: Cambridge University Press. doi:10.1017/9781108333511

[18] Fox, R (2023) Heat Conduction of Sand, UCC

Appendix A

Python For Section 2.2

[ ]:
```
import numpy as np
from scipy.sparse import diags
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Set up the problem parameters
L = 1.0
nx = 100
dx = L / (nx + 1)
nt = 1500
dt = 0.001
alpha = 0.09
mu1 = 0.5
mu2 = 0.7
sigma = 0.09

# Set up the initial condition
x = np.linspace(0, L, nx+1)
u0 = np.exp(-0.5*((x-mu1)/sigma)**2)
#u0 = np.sin(x*(np.pi/L))
u0[0] = 0
u0[-1] = 0
# Set up the spatial derivative discretization matrix
A=np.eye(nx+1,k=1)+np.eye(nx+1,k=-1)-2*np.eye(nx+1,k=0)
A[0,:]=np.zeros(nx+1)
A[-1,:]=np.zeros(nx+1)
A = A /(dx**2)
# Define the function for the RHS of the ODE system
def heat_eq(t,u):
    return alpha * A @ u
```

```
# Solve the ODE system using the runge-kutta method
sol = solve_ivp(heat_eq, [0, nt * dt], u0, t_eval=np.arange(0, nt * dt,␣
 ↪dt), method='RK45', vectorized=True)
u = sol.y.T


#u = np.apply_along_axis(dirichlet_bc, axis=1, arr=u)
#print(u)
# Plot the solution at different time points
plt.style.use('ggplot')

# Set figure size
fig, ax = plt.subplots(figsize=(15, 10))

# Set labels and titles
ax.set_xlabel(r'$x/L$', fontsize=14)
ax.set_ylabel(r'$u$', fontsize=14)
ax.set_title('1-D Heat Equation for Initial Central Gaussian␣
 ↪Distrubtion', fontsize=16)

# Plot the data with line and marker styles
ax.plot(x, u[0, :], linestyle='-', linewidth=2, marker='o',␣
 ↪markersize=6, label='t=0')
ax.plot(x, u[300, :], linestyle='-', linewidth=2, marker='o',␣
 ↪markersize=6, label='t=0.3')
ax.plot(x, u[600, :], linestyle='-', linewidth=2, marker='o',␣
 ↪markersize=6, label='t=0.6')
ax.plot(x, u[900, :], linestyle='-', linewidth=2, marker='o',␣
 ↪markersize=6, label='t=0.9')
ax.plot(x, u[1200, :], linestyle='-', linewidth=2, marker='o',␣
 ↪markersize=6, label='t=1.2')
ax.plot(x, u[1499, :], linestyle='-', linewidth=2, marker='o',␣
 ↪markersize=6, label='t=1.5')

# Configure legend, gridlines, and tick labels
ax.legend(fontsize=12, loc='best')
ax.grid(True)
ax.tick_params(axis='both', labelsize=12)

# Save the figure with a high resolution
fig.savefig('figure_name.png', dpi=900)

# Show the plot
plt.show()
```

Appendix B

Python For Section 2.3

[1]:
```
import numpy as np
from scipy.sparse import diags
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Set up the problem parameters
L = 1.0
nx = 20
dx = L / (nx + 1)
nt = 1500
dt = 0.005
alpha = 0.008
mux = 0.5
muy = 0.5
sigma = 0.05
```

```
Matplotlib is building the font cache; this may take a moment.
```

[2]:
```
# Set up the initial condition (Gaussian Centred on mux, muy)
x = np.linspace(0, 1, nx+1)
y = np.linspace(0, 1, nx+1)
xx, yy = np.meshgrid(x, y)
u0 = np.exp(-0.5*(((xx-mux)**2+(yy-muy)**2)/sigma)**2)
#flatten the 2-D array to a 1-D vector for computing time derivatives␣
 ↪by matrix multiplication
```

[3]:
```
# Set up the initial condition (Gaussian Centred on mux, muy)
x = np.linspace(0, 1, nx+1)
y = np.linspace(0, 1, nx+1)
xx, yy = np.meshgrid(x, y)
u0 = np.exp(-0.5*(((xx-mux)**2+(yy-muy)**2)/sigma)**2)
```

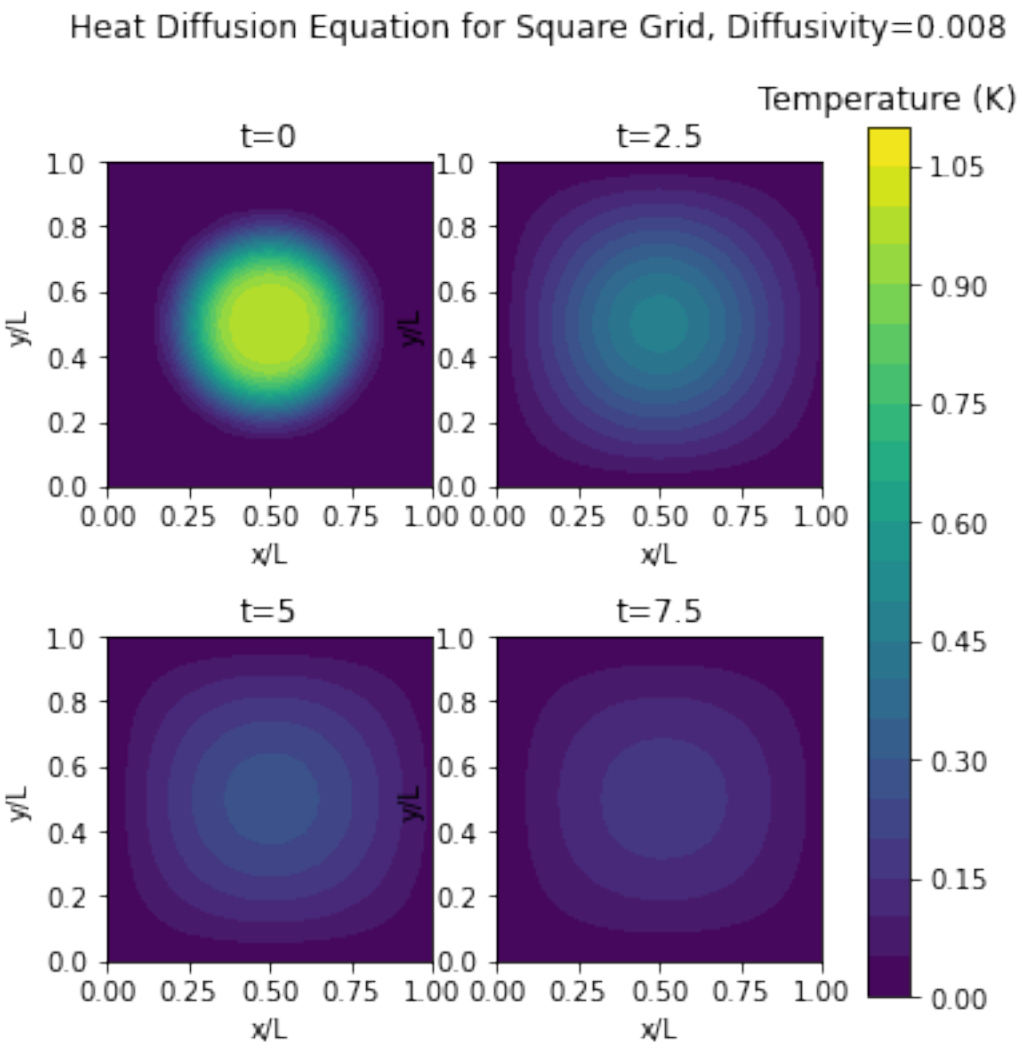
```
#flatten the 2-D array to a 1-D vector for computing time derivatives␣
 ↪by matrix multiplication

u0_flat = u0.flatten()
```

[4]:
```
# Set up the spatial discretization matrix, this time accounting for␣
 ↪two spatial derivatives
A=np.eye((nx+1)*(nx+1),k=1)+np.eye((nx+1)*(nx+1),k=-1)-4*np.
 ↪eye((nx+1)*(nx+1),k=0)+np.eye((nx+1)*(nx+1), k=nx+1)+np.
 ↪eye((nx+1)*(nx+1), k=-(nx+1))
#these for loops are used to zero elements of the matrix corresponding␣
 ↪to boundaries of square
for j in range(0,nx+1):
  A[j,:]=np.zeros((nx+1)*(nx+1))
  A[:,j]=np.zeros((nx+1)*(nx+1))
for j in range(1,nx+2):
  A[-j,:]=np.zeros((nx+1)*(nx+1))
  A[:,-j]=np.zeros((nx+1)*(nx+1))
for i in range(0, nx+2,1):
  A[min((i*(nx+1)),((nx+1)*(nx+1)-1)),:]=np.zeros((nx+1)*(nx+1))
  A[:,min((i*(nx+1)),((nx+1)*(nx+1)-1))]=np.zeros((nx+1)*(nx+1))
A = A /(dx**2)
```

[5]:
```
# Solve the ODE system using the Runge-Kutta 4 Algorithm
#the function heat_eq calculates the time derivatives for each␣
 ↪component of the system, by matrix multiplication in numpy
def heat_eq(t,u):
  return alpha * A @ u

sol = solve_ivp(heat_eq, [0, nt * dt], u0_flat, t_eval=np.arange(0, nt␣
 ↪* dt, dt), method='RK45', vectorized=True)
u = sol.y.T
```

[6]:
```
#plot a collection of time steps for figure
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(6, 6))
axs[0,0].set_aspect("equal")
axs[0,0].set_xlabel(r'x/L')
axs[0,0].set_ylabel(r'y/L')

axs[0,0].contourf(x, y, u[0,:].reshape(nx+1,nx+1), [0,0.05,0.1,0.15,0.
 ↪2,0.25,0.3,0.35,0.4,0.45,0.5,0.55,0.6,0.65,0.7,0.75,0.8,0.85,0.9,0.
 ↪95,1,1.05,1.1])
axs[0,1].set_aspect("equal")
axs[0,1].set_xlabel(r'x/L')
axs[0,1].set_ylabel(r'y/L')
```

```
axs[0,1].contourf(x, y, u[500,:].reshape(nx+1,nx+1), [0,0.05,0.1,0.15,0.
 ↪2,0.25,0.3,0.35,0.4,0.45,0.5,0.55,0.6,0.65,0.7,0.75,0.8,0.85,0.9,0.
 ↪95,1,1.05,1.1])
axs[1,0].set_aspect("equal")
axs[1,0].set_xlabel(r'x/L')
axs[1,0].set_ylabel(r'y/L')

axs[1,0].contourf(x, y, u[1000,:].reshape(nx+1,nx+1), [0,0.05,0.1,0.
 ↪15,0.2,0.25,0.3,0.35,0.4,0.45,0.5,0.55,0.6,0.65,0.7,0.75,0.8,0.85,0.
 ↪9,0.95,1,1.05,1.1])
axs[1,1].set_aspect("equal")
axs[1,1].set_xlabel(r'x/L')
axs[1,1].set_ylabel(r'y/L')

a=axs[1,1].contourf(x, y, u[1499,:].reshape(nx+1,nx+1), [0,0.05,0.1,0.
 ↪15,0.2,0.25,0.3,0.35,0.4,0.45,0.5,0.55,0.6,0.65,0.7,0.75,0.8,0.85,0.
 ↪9,0.95,1,1.05,1.1])
axs[0, 0].set_title('t=0')
axs[0, 1].set_title('t=2.5')
axs[1, 0].set_title('t=5')
axs[1, 1].set_title('t=7.5')

cbar = fig.colorbar(a, ax=axs)

# Add a title to the colorbar
cbar.ax.set_title('Temperature (K)')
fig.suptitle("Heat Diffusion Equation for Square Grid, Diffusivity=0.
 ↪008")

plt.savefig('2DHeatEquation.pdf',dpi=700)
```


Heat Diffusion Equation for Square Grid, Diffusivity=0.008
Temperature (K)
t=0
t=2.5
t=5
t=7.5
y/L
x/L
0.0
0.2
0.4
0.6
0.8
1.0
0.00
0.25
0.50
0.75
1.00
1.05
0.90
0.75
0.60
0.45
0.30
0.15
0.00

[ ]:

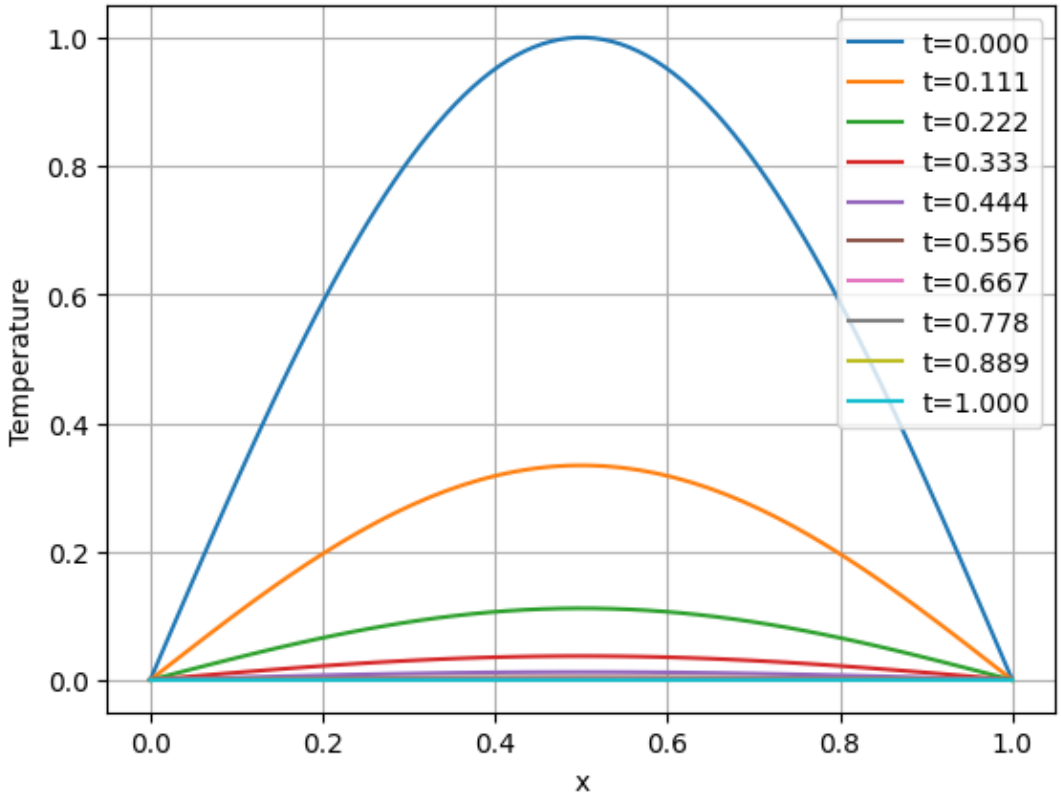
Appendix C

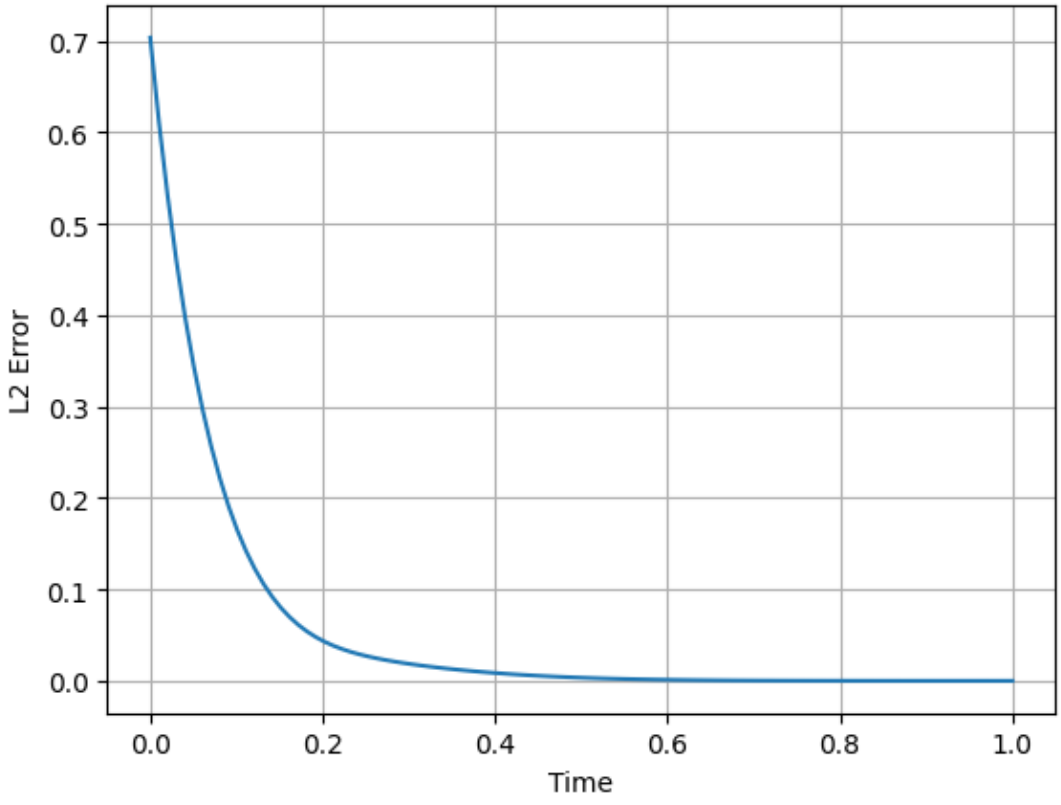
MOL and FEM in 1D heat equation

[1]:
```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
import time as time

# Define the equation to be solved
def heat_equation(t, y, dx, D):
    # y is a vector of the temperature at each point x
    # dx is the grid spacing
    # D is the diffusion coefficient
    n = len(y)
    dydx = np.zeros(n)
    dydx[0] = 0  # boundary condition at x=0
    dydx[n-1] = 0  # boundary condition at x=L
    for i in range(1, n-1):
        dydx[i] = D * (y[i+1] - 2*y[i] + y[i-1]) / dx**2
    return dydx

# Define the exact solution
def exact_solution(x, t, D, L):
    return np.sin(np.pi*x/L) * np.exp(-np.pi**2*D*t/L**2)

# Define the initial temperature profile
L = 1.0  # length of the rod
nx = 100  # number of grid points
x = np.linspace(0, L, nx)
T0 = np.sin(np.pi*x/L)

# Set the diffusion coefficient and time range
D = 1  # diffusion coefficient
tspan = [0, 1]

start_time = time.time()
```

```
# Solve the equation using the method of lines
sol = solve_ivp(heat_equation, tspan, T0, args=(L/(nx-1), D),
                t_eval=np.linspace(tspan[0], tspan[1], 100))
end_time = time.time()

cost_time1 = end_time - start_time
print('Cost time:', cost_time1)

# Compute the exact solution
exact = exact_solution(x, sol.t, D, L)

# Compute the L2 error between the numerical and exact solutions
errors2 = np.sqrt(np.sum((sol.y - exact)**2, axis=0) * L/nx)

# Plot the numerical and exact solutions for 5 different times
t_indices = np.linspace(0, len(sol.t)-1, 10, dtype=int)
t_values = sol.t[t_indices]

for i, t_index in enumerate(t_indices):
    plt.plot(x, sol.y[:, t_index], label='t={:.3f}'.format(t_values[i]))

plt.legend()
plt.xlabel('x')
plt.ylabel('Temperature')
plt.grid()
plt.show()

# Plot the L2 error vs time
plt.plot(sol.t, errors2)
plt.xlabel('Time')
plt.ylabel('L2 Error')
plt.grid()
plt.show()
```

```
Cost time: 6.393998384475708
```


t=0.000
t=0.111
t=0.222
t=0.333
t=0.444
t=0.556
t=0.667
t=0.778
t=0.889
t=1.000
Temperature
x
0.0
0.2
0.4
0.6
0.8
1.0


L2 Error
Time
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.0
0.2
0.4
0.6
0.8
1.0

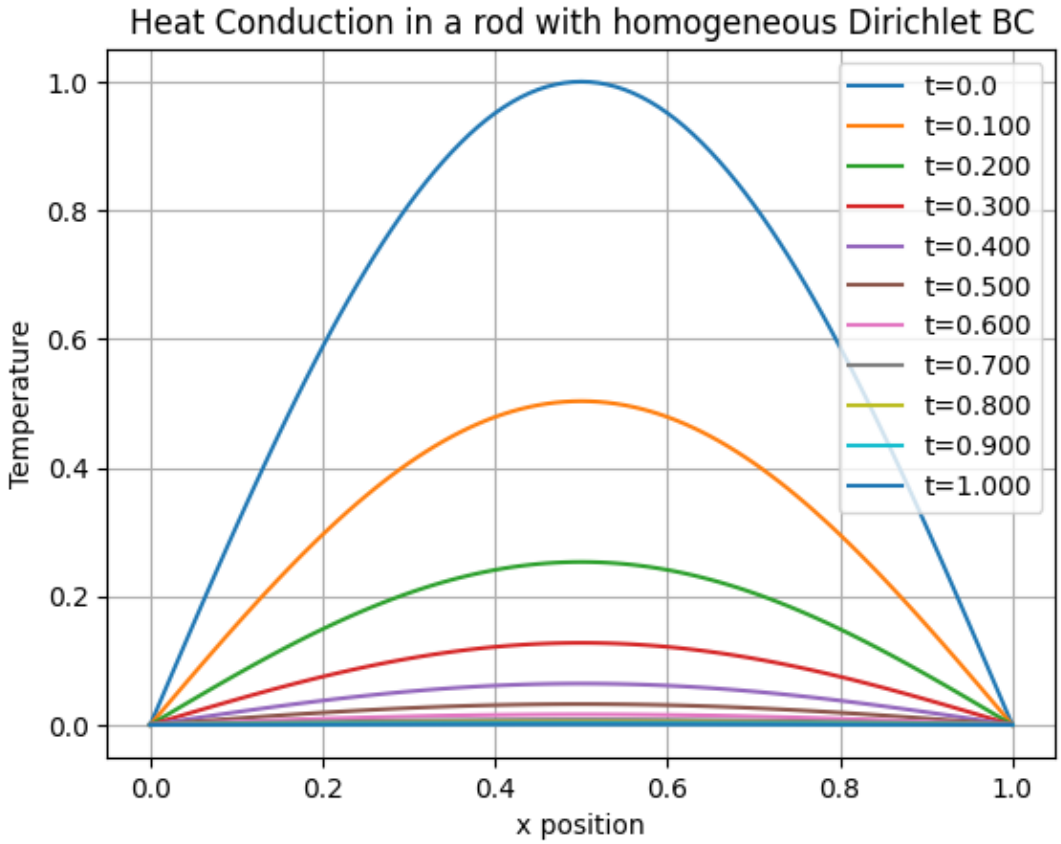
[2]:
```
import fenics as fe
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
import time as time

n_elements = 100
mesh = fe.UnitIntervalMesh(n_elements)

# Define a Function Space
lagrange_polynomial_space_first_order = fe.FunctionSpace(
    mesh,
    "Lagrange",
    1
)

# The value of the solution on the boundary
u_on_boundary = fe.Constant(0.0)

# A function to return whether we are on the boundary
def boundary_boolean_function(x, on_boundary):
    return on_boundary

# The homogeneous Dirichlet Boundary Condition
boundary_condition = fe.DirichletBC(
    lagrange_polynomial_space_first_order,
    u_on_boundary,
    boundary_boolean_function,
)

# The initial condition, u(t=0, x) = sin(pi * x)
initial_condition = fe.Expression(
    "sin(pi * x[0])",
    degree=1
)

# Discretize the initial condition
u_old = fe.interpolate(
    initial_condition,
    lagrange_polynomial_space_first_order
)
plt.figure()
fe.plot(u_old, label="t=0.0")

# Record start time
start_time = time.time()
```

```
# The time stepping of the implicit Euler discretization (=dt)
time_step_length = 0.1

# The forcing on the rhs of the PDE
heat_source = fe.Constant(0.0)

# Create the Finite Element Problem
u_trial = fe.TrialFunction(lagrange_polynomial_space_first_order)
v_test = fe.TestFunction(lagrange_polynomial_space_first_order)

weak_form_residuum = (
    u_trial * v_test * fe.dx
    +
    time_step_length * fe.dot(
        fe.grad(u_trial),
        fe.grad(v_test),
    ) * fe.dx
    -
    (
        u_old * v_test * fe.dx
        +
        time_step_length * heat_source * v_test * fe.dx
    )
)

# We have a linear PDE that is separable into a lhs and rhs
weak_form_lhs = fe.lhs(weak_form_residuum)
weak_form_rhs = fe.rhs(weak_form_residuum)

# The function we will be solving for at each point in time
u_solution = fe.Function(lagrange_polynomial_space_first_order)

# time stepping
n_time_steps = 10
time_current = 0.0
for i in range(n_time_steps):
    time_current += time_step_length

    # Finite Element Assembly, BC imprint & solving the linear system
    fe.solve(
        weak_form_lhs == weak_form_rhs,
        u_solution,
        boundary_condition,
    )

    u_old.assign(u_solution)
```

```
    fe.plot(u_solution, label=f"t={time_current:1.3f}")
# Record end time
end_time = time.time()

# Calculate cost time
cost_time2 = end_time - start_time

# Add the legend and show the plot
plt.legend()
plt.title("Heat Conduction in a rod with homogeneous Dirichlet BC")
plt.xlabel("x position")
plt.ylabel("Temperature")
plt.grid()
plt.show()

# Define the exact solution
def exact_solution(x, t):
    return np.sin(np.pi*x) * np.exp(-np.pi**2*t)

# Compute the L2 norm of the error at each time step
errors = []
for i in range(n_time_steps):
    time_current = i * time_step_length
    u_exact = fe.interpolate(
        fe.Expression(
            "sin(pi*x[0])*exp(-pi*pi*t)",
            degree=1,
            t=time_current
        ),
        lagrange_polynomial_space_first_order
    )
    error = fe.errornorm(u_exact, u_solution, "L2")
    errors.append(error)

# Plot the error as a function of time
plt.figure()
plt.plot(np.arange(n_time_steps)*time_step_length, errors)
plt.title("L2 Error vs. Time")
plt.xlabel("Time")
plt.ylabel("L2 Error")
plt.grid()
plt.show()
print("Total cost time:", cost_time2, "seconds")
```

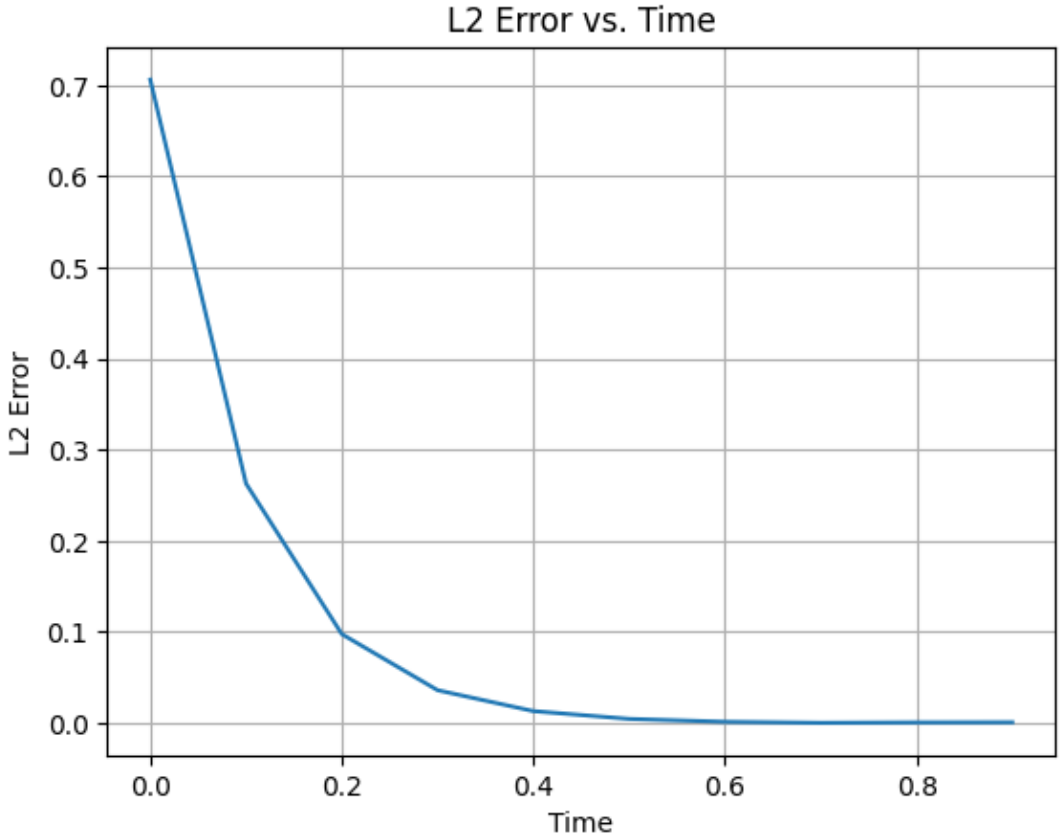
```
Solving linear variational problem.
Solving linear variational problem.
Solving linear variational problem.
```

```
Solving linear variational problem.
Solving linear variational problem.
Solving linear variational problem.
Solving linear variational problem.
Solving linear variational problem.
Solving linear variational problem.
Solving linear variational problem.
```




```
*** Warning: Degree of exact solution may be inadequate for accurate␣
 ↪result in
errornorm.
*** Warning: Degree of exact solution may be inadequate for accurate␣
 ↪result in
errornorm.
*** Warning: Degree of exact solution may be inadequate for accurate␣
 ↪result in
errornorm.
*** Warning: Degree of exact solution may be inadequate for accurate␣
 ↪result in
errornorm.
*** Warning: Degree of exact solution may be inadequate for accurate␣
 ↪result in
```

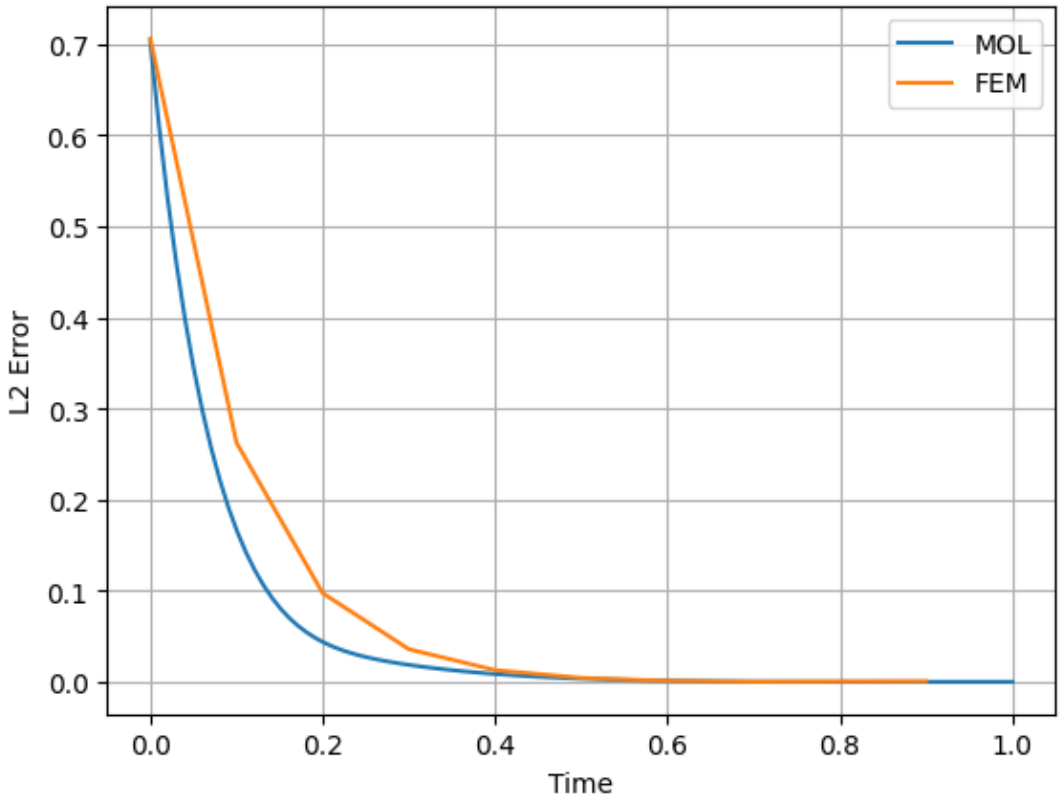
```
errornorm.
*** Warning: Degree of exact solution may be inadequate for accurate␣
 ↪result in
errornorm.
*** Warning: Degree of exact solution may be inadequate for accurate␣
 ↪result in
errornorm.
*** Warning: Degree of exact solution may be inadequate for accurate␣
 ↪result in
errornorm.
*** Warning: Degree of exact solution may be inadequate for accurate␣
 ↪result in
errornorm.
*** Warning: Degree of exact solution may be inadequate for accurate␣
 ↪result in
errornorm.
```




```
Total cost time: 0.10394048690795898 seconds
```

[3]:
```
# Plot the L2 error vs time for both methods
plt.plot(sol.t, errors2, label='MOL')
plt.plot(np.arange(n_time_steps)*time_step_length, errors, label='FEM')
```

```
plt.xlabel('Time')
plt.ylabel('L2 Error')
plt.legend()
plt.grid()
plt.show()
```



[4]:
```
print("Cost time of MOL:", cost_time1, "seconds")
print("Cost time of FEM:", cost_time2, "seconds")
```

```
Cost time of MOL: 6.393998384475708 seconds
Cost time of FEM: 0.10394048690795898 seconds
```

Appendix D

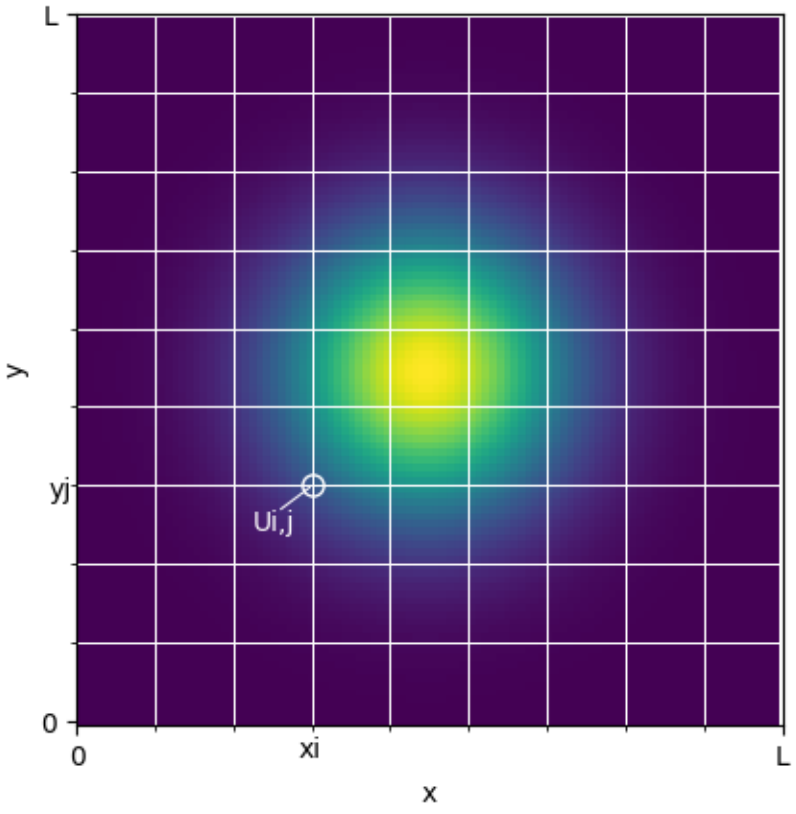
Python For Figure 2.3

[ ]:

[ ]:
```
#Import Dependencies
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
```

[ ]:
```
# Grid size
grid_size = 100

# Define the 2D Gaussian distribution
mean = np.array([grid_size / 2, grid_size / 2])
cov = np.array([[200, 0], [0, 200]])
gaussian = multivariate_normal(mean=mean, cov=cov)

# Create a square grid
x, y = np.meshgrid(np.linspace(0, grid_size, grid_size), np.linspace(0,␣
 ↪grid_size, grid_size))
pos = np.dstack((x, y))

# Evaluate the Gaussian distribution on the grid
z = gaussian.pdf(pos)

# Create the colormap
fig, ax = plt.subplots()
heatmap = ax.imshow(z, cmap='viridis', origin='lower')

# Add a visible white grid along the x and y axis
ax.set_xticks(np.linspace(0,grid_size,10), minor=True)
ax.set_yticks(np.linspace(0,grid_size,10), minor=True)
ax.grid(which='minor', linestyle='-', linewidth=0.75, color='white')
ax.plot(33.5, 33.5, marker='o', markersize=8, fillstyle='none',␣
 ↪color='white')
```

```
ax.text(25,27,"Ui,j", color='white', fontsize=10)
ax.text(31.5,-5,"xi", color='black', fontsize=10)
ax.text(-4,31.5,"yj", color='black', fontsize=10)
ax.plot([29,33],[30,33],color='white',linewidth=0.75)
# Configure the plot
ax.set_xticks([0,100])
ax.set_yticks([0,100])
ax.set_xlim(0,100)
ax.set_xlabel("x")
ax.set_ylabel("y")#
ax.set_xticklabels(["0","L"])
ax.set_yticklabels(["0","L"])
# Save the plot as an PDF file
fig.savefig('gaussian_heatmap_with_grid.pdf', format='pdf')

# Show the plot
plt.show()
```



[ ]:

[ ]:

Appendix E

Python for figure 2.1

[ ]:
```
#Generates graph for section 2.1
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
```

[ ]:
```
# Grid size and line intervals
grid_size = 100
line_interval = 5

# Define the Gaussian distribution
mean = grid_size / 2
std_dev = grid_size / 8
x_values = np.linspace(0, grid_size, grid_size)
y_values = norm.pdf(x_values, loc=mean, scale=std_dev)

# Create the plot
fig, ax = plt.subplots()

# Plot the Gaussian distribution in black
ax.plot(x_values, y_values, linewidth=2, color='black')
```

[ ]:
```
# Draw vertical lines at even intervals in black
for x_val, y_val in zip(x_values[::line_interval], y_values[::
 ↪line_interval]):
    ax.plot([x_val, x_val], [0, y_val], linestyle='-', linewidth=0.5,␣
 ↪color='black')
```

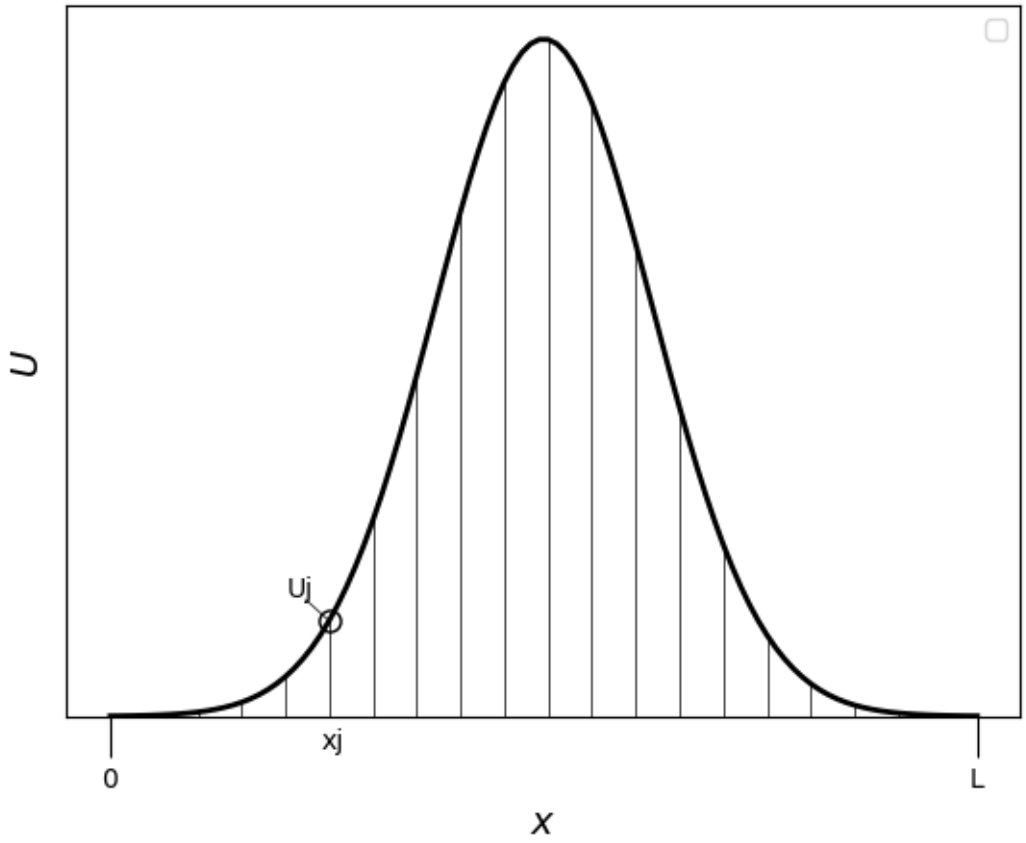
[ ]:
```
# Add an empty black circle at the intersection of a line with the␣
 ↪curve
selected_x = x_values[25]  # Choose the x value where the circle will␣
 ↪be placed
selected_y = y_values[25]
```

```
ax.plot(selected_x, selected_y, marker='o', markersize=8,␣
 ↪fillstyle='none', color='black')
ax.plot([(selected_x-2.7),selected_x], [selected_y+0.001,selected_y],␣
 ↪linestyle='-', linewidth=0.5, color='black')
```

[ ]:
```
# Label the point underneath the line as 'xj'
ax.text((selected_x-1), -0.0015, 'xj', fontsize=10)
ax.text((selected_x-5), selected_y+0.0012, 'Uj', fontsize=10)
```

[ ]:
```
# Configure the plot
ax.set_xlabel(r'$x$', fontsize=14)
ax.set_ylabel(r'$U$', fontsize=14)
ax.set_xticks([x_values[0], x_values[-1]])
ax.tick_params(axis="x",which="both", length=15)
ax.tick_params(axis="y",which="both", length=0)
ax.set_xticklabels(["0","L"])
ax.set_yticklabels([""])
# Set custom y limit
ax.set_ylim(bottom=0)

# Add legend
ax.legend()

# Save the plot as an SVG file
fig.savefig('gaussian_distribution_with_vertical_lines_bw_circle.pdf',␣
 ↪format='pdf')

# Show the plot
plt.show()
```

```
<ipython-input-70-fd0b67c4aab3>:40: UserWarning: FixedFormatter should␣
 ↪only be
used together with FixedLocator
  ax.set_yticklabels([""])
WARNING:matplotlib.legend:No artists with labels found to put in legend.␣
 ↪ Note
that artists whose label start with an underscore are ignored when␣
 ↪legend() is
called with no argument.
```


U
Uj
xj
0
L
x

Appendix F

Python Implementation of Sand Simulation Chapter 4

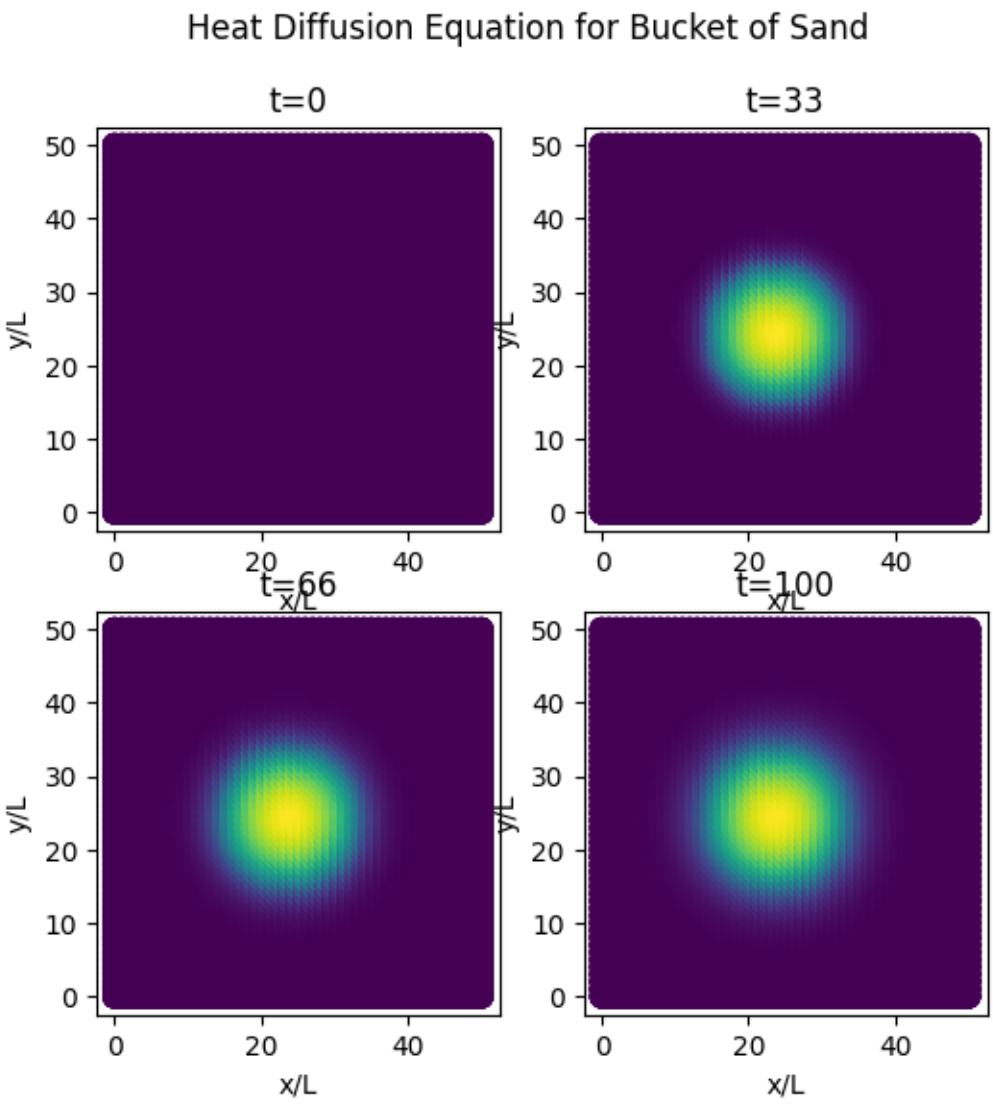
Projection was changed to 3-D after this latex document was compiled, I didn't have access to Jupyter and so couldn't edit it.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.stats import multivariate_normal
import matplotlib.animation as animation

# Define the parameters
x_max = 50
y_max = 50
z_max = 100
t_max = 100
dx = 1
dy = 1
dz = 1
dt = 0.1


Nx = int(x_max / dx) + 1
Ny = int(y_max / dy) + 1
Nz = int(z_max / dz) + 1
Nt = int(t_max / dt) + 1

alpha = 0.2
alpha_inner = 2.9  # diffusivity for the inner cylinder

inner_radius = 10  # radius of the inner cylinder
temperature_increase = 0.0827  # constant temperature increase per time␣
 ↪step
```

```
x = np.linspace(0, x_max, Nx)
y = np.linspace(0, y_max, Ny)
z = np.linspace(0, z_max, Nz)
t = np.linspace(0, t_max, Nt)

U = np.zeros((Nx, Ny, Nz, Nt))

Ax = alpha * dt / (2 * dx**2)
Ay = alpha * dt / (2 * dy**2)
Az = alpha * dt / (2 * dz**2)

Ax_inner = alpha_inner * dt / (2 * dx**2)
Ay_inner = alpha_inner * dt / (2 * dy**2)
Az_inner = alpha_inner * dt / (2 * dz**2)

for n in range(0, Nt - 1):
    U_prev = U[:, :, :, n].copy()
    for i in range(1, Nx - 1):
        for j in range(1, Ny - 1):
            for k in range(1, Nz - 1):
                # Check if the current grid point is inside the inner␣
 ↪cylinder
                if ((x[i] - x_max / 2)**2 + (y[j] - y_max / 2)**2) <␣
 ↪inner_radius**2:
                    U[i, j, k, n+1] = U_prev[i, j, k] + Ax_inner *␣
 ↪(U_prev[i+1, j, k] - 2 * U_prev[i, j, k] + U_prev[i-1, j, k]) +␣
 ↪Ay_inner * (U_prev[i, j+1, k] - 2 * U_prev[i, j, k] + U_prev[i, j-1,␣
 ↪k]) + Az_inner * (U_prev[i, j, k+1] - 2 * U_prev[i, j, k] +␣
 ↪U_prev[i, j, k-1]) + temperature_increase
                else:
                    U[i, j, k, n+1] = U_prev[i, j, k] + Ax *␣
 ↪(U_prev[i+1, j, k] - 2 * U_prev[i, j, k] + U_prev[i-1, j, k]) + Ay *␣
 ↪(U_prev[i, j+1, k] - 2 * U_prev[i, j, k] + U_prev[i, j-1, k]) + Az *␣
 ↪(U_prev[i, j, k+1] - 2 * U_prev[i, j, k] + U_prev[i, j, k-1])

# Visualize the results


X, Y, Z = np.meshgrid(x, y, z[:-30], indexing='ij')
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(6, 6))
axs[0,0].set_aspect("equal")
axs[0,0].set_xlabel(r'x/L')
axs[0,0].set_ylabel(r'y/L')
```

```
axs[0,0].scatter(X, Y, Z, c=U[:, :, :-30, 0].flatten(), cmap='viridis')
axs[0,1].set_aspect("equal")
axs[0,1].set_xlabel(r'x/L')
axs[0,1].set_ylabel(r'y/L')

axs[0,1].scatter(X, Y, Z, c=U[:, :, :-30, 330].flatten(),␣
 ↪cmap='viridis')
axs[1,0].set_aspect("equal")
axs[1,0].set_xlabel(r'x/L')
axs[1,0].set_ylabel(r'y/L')

axs[1,0].scatter(X, Y, Z, c=U[:, :, :-30, 660].flatten(),␣
 ↪cmap='viridis')
axs[1,1].set_aspect("equal")
axs[1,1].set_xlabel(r'x/L')
axs[1,1].set_ylabel(r'y/L')

a=axs[1,1].scatter(X, Y, Z, c=U[:, :, :-30, 990].flatten(),␣
 ↪cmap='viridis')
axs[0, 0].set_title('t=0')
axs[0, 1].set_title('t=33')
axs[1, 0].set_title('t=66')
axs[1, 1].set_title('t=100')


# Add a title to the colorbar
fig.suptitle("Heat Diffusion Equation for Bucket of Sand")
```

[ ]: Text(0.5, 0.98, 'Heat Diffusion Equation for Bucket of Sand')


Heat Diffusion Equation for Bucket of Sand
t=0
t=33
t=66
t=100
y/L
x/L
0
10
20
30
40
50

[ ]:

Appendix G

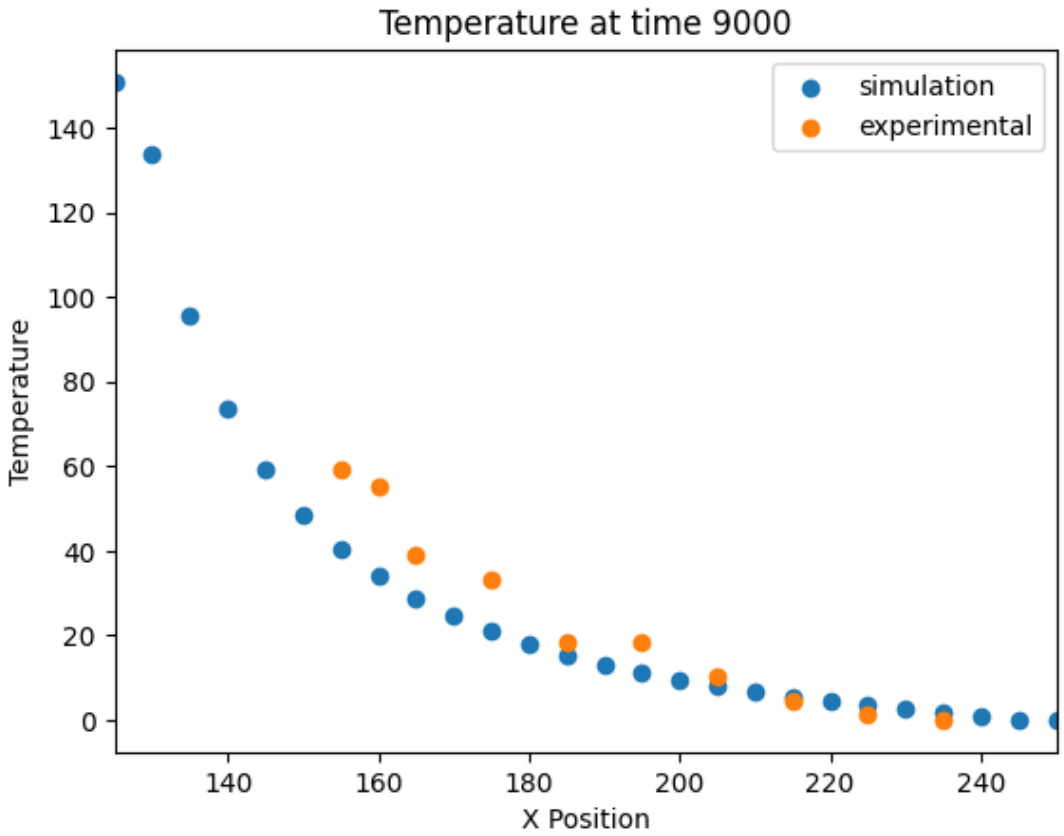
Python Code to Interpet C++ data Chapter 4

[ ]:

[ ]:

[13]:
```
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.patches import Circle
# Read the CSV file
csv_file = "/content/drive/MyDrive/BucketData3 (13).csv"
data = pd.read_csv(csv_file)

# Select data for an arbitrary time value
t_arbitrary = 9000  # Replace this with the desired time value
selected_data = data[(data['t'] == t_arbitrary) & (data['z'] == 128) &␣
 ↪(data['y'] == 125)]

# Create the 3D scatter plo
fig = plt.figure()
ax = fig.add_subplot(111)

x = selected_data['x']
c = selected_data['temperature']

x1= [155,160,165,175,185,195,205,215,225,235]
c1= [59.3,55.2,39.1,33,18.4,18.3,10.4,4.5,1.3,0]
simulation = ax.scatter(x,c)
experimental = ax.scatter(x1,c1)

# Add axis labels and colorbar
ax.set_xlabel('X Position')
```

```
ax.set_ylabel('Temperature')

ax.set_xlim(125,250)
simulation.set_label("simulation")
experimental.set_label("experimental")
# Set the title
ax.set_title(f"Temperature at time {t_arbitrary}")
ax.legend()
# Show the plot
plt.show()
```



[ ]:

Appendix H

C++ Code for Chapter 4

```cpp
// basic file operations
#include <thread>
#include <fstream>
#include <iostream>
#include <cmath>

using namespace std;
void foo(int t, double dx, double dy, double dt, double dz, ofstream& file, int zmin, int zmax, double***& Ucurr, double***& Uprev1, double***& Uprev2, double***& pos, double extra_heat, double Ax, double Ay, double Az, double Ax_inner, double Ay_inner, double Az_inner, int Nx, int Ny, int Nz)
{
    if (t % 2 == 0)
    {
        for (int i = 1;i <= Nx - 1;i++)
        {
            for (int j = 1;j <= Ny - 1;j++)
            {
                for (int k = zmin/dz;k <zmax/dz;k++)
                {
                        Ucurr[i][j][k] = 0;
                        if (pos[i][j][k] == 1)
                        {
                            Ucurr[i][j][k] = Uprev2[i][j][k] + Ax * (Uprev2[i + 1][j][k] + Uprev2[i - 1][j][k] - 2 * Uprev2[i][j][k]) + Ay * (Uprev2[i][j + 1][k] + Uprev2[i][j - 1][k] - 2 * Uprev2[i][j][k]) + Az * (Uprev2[i][j][k + 1] + Uprev2[i][j][k - 1] - 2 * Uprev2[i][j][k]);
                        }
                        else if (pos[i][j][k] == 2)
                        {
                            Ucurr[i][j][k] = Uprev2[i][j][k] + Ax_inner * (Uprev2[i + 1][j][k] + Uprev2[i - 1][j][k] - 2 * Uprev2[i][j][k]) + Ay_inner * (Uprev2[i][j + 1][k] + Uprev2[i][j - 1][k] - 2 * Uprev2[i][j][k]) + Az_inner * (Uprev2[i][j][k + 1] + Uprev2[i][j][k - 1] - 2 * Uprev2[i][j][k]) + extra_heat;
                        }



                    Uprev1[i][j][k] = Ucurr[i][j][k];
                    if (t % 10000 == 0)
                    {
                        file << t * dt << "," << i * dx << "," << j * dy << "," << k * dz << "," << Ucurr[i][j][k] << endl;
                    }

                }
            }
        }
```

```
    }
    else
    {
        for (int i = 1;i <= Nx - 1;i++)
        {
            for (int j = 1;j <= Ny - 1;j++)
            {
                for (int k = zmin/dz;k <(zmax/dz);k++)
                {

                    Ucurr[i][j][k] = 0;
                    if (pos[i][j][k] == 1)
                    {
                        Ucurr[i][j][k] = Uprev1[i][j][k] + Ax * (Uprev1[i + 1] ↵
                    [j][k] + Uprev1[i - 1][j][k] - 2 * Uprev1[i][j][k]) + Ay * ↵
                    (Uprev1[i][j + 1][k] + Uprev1[i][j - 1][k] - 2 * Uprev1[i] ↵
                    [j][k]) + Az * (Uprev1[i][j][k + 1] + Uprev1[i][j][k - 1] -↵
                     2 * Uprev1[i][j][k]);
                    }
                    else if (pos[i][j][k] == 2)
                    {
                        Ucurr[i][j][k] = Uprev1[i][j][k] + Ax_inner * (Uprev1[i↵
                     + 1][j][k] + Uprev1[i - 1][j][k] - 2 * Uprev1[i][j][k]) + ↵
                    Ay_inner * (Uprev1[i][j + 1][k] + Uprev1[i][j - 1][k] - 2 *↵
                     Uprev1[i][j][k]) + Az_inner * (Uprev1[i][j][k + 1] +      ↵
                    Uprev1[i][j][k - 1] - 2 * Uprev1[i][j][k]) + extra_heat;
                    }

                    Uprev2[i][j][k]=Ucurr[i][j][k];
                    if (t % 10000==0)
                    {
                        file << t * dt << "," << i * dx << "," << j * dy << ","↵
                     << k * dz << "," << Ucurr[i][j][k] << endl;
                    }
                }

            }
        }
    }
}
int main()
{
    int x_max = 250;
    int y_max = 250;
    int z_max = 200;
    int t_max = 9000;
    double dx = 5;
    double dy = 5;
    double dz = 2;
    double dt = 0.05;
    int Nx = int(x_max / dx) + 1;
    int Ny = int(y_max / dy) + 1;
    int Nz = int(z_max / dz) + 1;
```

```
    int Nt = int(t_max / dt) + 1;
    double*** Uprev1;
    double*** Uprev2;
    double*** Ucurr;
    double*** pos;
    Uprev1 = new double** [Nx];
    Uprev2 = new double** [Nx];
    Ucurr = new double** [Nx];
    pos = new double** [Nx];
    //Allocating the column space in heap dynamically
    for (int i = 0; i < Nx; i++) {

        Uprev1[i] = new double* [Ny];
        Uprev2[i] = new double* [Ny];
        Ucurr[i] = new double* [Ny];
        pos[i] = new double* [Ny];
        for (int j = 0; j < Ny; j++)
        {
            Uprev1[i][j] = new double[Nz];
            Uprev2[i][j] = new double[Nz];
            Ucurr[i][j] = new double[Nz];
            pos[i][j] = new double[Nz];
            for (int k = 0; k < Nx; k++)
            {
                Uprev1[i][j][k] = 0;
                Uprev2[i][j][k] = 0;
                Ucurr[i][j][k] = 0;
                pos[i][j][k] = 0;
            }
        }

    }

    int inner_radius = 5;
    int outer_radius = 120;
    double alpha = 0.85;
    double alpha_inner = 2.9;
    double extra_heat = 8.27 * dt;
    double Ax = (alpha * dt) / pow(dx, 2);
    double Ay = (alpha * dt) / pow(dy, 2);
    double Az = (alpha * dt) / pow(dz, 2);
    double Ax_inner = (alpha_inner * dt) / pow(dx, 2);
    double Ay_inner = (alpha_inner * dt) / pow(dy, 2);
    double Az_inner = (alpha_inner * dt) / pow(dz, 2);
    cout << "beepbopp";

    for (int i = 0;i < Nx;i++)
    {
        for (int j = 0;j <Ny;j++)
        {
            for (int k = 0;k <Nz ;k++)
            {
```

```
                if (pow((i * dx - x_max / 2), 2) + pow((j * dy - y_max / 2), 2)
                    <= pow(outer_radius, 2))
                {
                    pos[i][j][k] = 1;
                }
                if ((pow((i * dx - x_max / 2), 2) + pow((j * dy - y_max / 2),
                  2) <= pow(inner_radius, 2)) && (k*dz >= z_max/2))
                {
                    pos[i][j][k] = 2;
                }
                if (pow((i * dx - x_max / 2), 2) + pow((j * dy - y_max / 2), 2)
                    >= pow(outer_radius, 2))
                {
                    pos[i][j][k] = 0;
                }
            }
        }
    }




    ofstream file1;
    file1.open("BucketData1.csv");
    ofstream file2;
    file2.open("BucketData2.csv");
    ofstream file3;
    file3.open("BucketData3.csv");
    ofstream file4;
    file4.open("BucketData4.csv");
    file1 << "t,x,y,z,temperature" << endl;
    file2 << "t,x,y,z,temperature" << endl;
    file3 << "t,x,y,z,temperature" << endl;
    file4 << "t,x,y,z,temperature" << endl;
    for (int t = 1; t < Nt; t++)
    {
        cout << "t = " << t * dt;
        std::thread t1(foo, t, dx, dy, dt, dz, std::ref(file1), 1, 50, std::ref
          (Ucurr), std::ref(Uprev1), std::ref(Uprev2), std::ref(pos),
          extra_heat, Ax, Ay, Az, Ax_inner, Ay_inner, Az_inner, Nx, Ny, Nz);
        std::thread t2(foo, t, dx, dy, dt, dz, std::ref(file2), 50, 100,
          std::ref(Ucurr), std::ref(Uprev1), std::ref(Uprev2), std::ref(pos),
          extra_heat, Ax, Ay, Az, Ax_inner, Ay_inner, Az_inner, Nx, Ny, Nz);
        std::thread t3(foo, t, dx, dy, dt, dz, std::ref(file3), 100,150,
          std::ref(Ucurr), std::ref(Uprev1), std::ref(Uprev2), std::ref(pos),
          extra_heat, Ax, Ay, Az, Ax_inner, Ay_inner, Az_inner, Nx, Ny, Nz);
        std::thread t4(foo, t, dx, dy, dt, dz, std::ref(file4), 150, 198,
          std::ref(Ucurr), std::ref(Uprev1), std::ref(Uprev2), std::ref(pos),
          extra_heat, Ax, Ay, Az, Ax_inner, Ay_inner, Az_inner, Nx, Ny, Nz);
        t1.join();
        t2.join();
        t3.join();
```

```
        t4.join();
    }
    file1.close();
    file2.close();
    file3.close();
    file4.close();
    return 0;
}
```