Coursework 1 (resit): The heat equation

6CCE3EAL Engineering Algorithms

David Moxey, 29th June 2023

- This coursework will count for **30%** of your total grade in the course.
- Remember that all work you submit should be **your own work**. Although you can e.g. discuss your approach to the problem at hand with others, this coursework is not meant to be a team exercise. The College treats plagiarism very seriously.
- You should submit your report by **16:00, Thursday 27th July 2023** using the submission link on the KEATS site for 6CCE3EAL.
- If you have more general or administrative problems please use the online forum or consult during drop-in-sessions or support sessions. If an email is required, please include the course number (6CCE3EAL) in the subject line.

1 Overview

Modelling the transfer of heat, either within a body or between different physical systems, is a fundamental part of engineering. There are three different mechanisms that underpin heat transfer: conduction, convection and radiation. In this assignment, you will explore thermal conduction (or diffusion) and create a program to model this physical phenomenon.

! This coursework has a number of components that need to be completed. You should ensure that you allow enough time to complete them before the deadline!

2 Background

Imagine that we have a solid rod, and we apply heat to the rod: for example, by holding it under a flame. Intuitively, we know that if we hold the rod at one end, then eventually we will feel the temperature rising, depending on what the rod is made from: metal rods will heat quickly, whereas a glass rod might not heat at all.

At the molecular level, the *'hot'* molecules underneath the flame are vibrating rapidly, which causes their adjacent neighbours to also start oscillating. This process repeats, and the further we are from the flame, the more the temperature 'spreads out'. If we remove the flame, then the temperature of the rod will eventually decay until it reaches the temperature of the room. This process is called *diffusion*. Visually, you can think about adding a small amount of dye to a large container of fluid at rest. Introducing the dye via a syringe at a specific point, we would eventually expect the dye to diffuse amongst the fluid until it can no longer be seen.

To model this mathematically, consider a solid rod of length L and constant cross-section, and assume that the temperature is constant in each cross-section. Then the temperature can be expressed as a function of both one-dimensional location x, where $0 \leq x \leq L$, and time $t \geq 0$. We can therefore

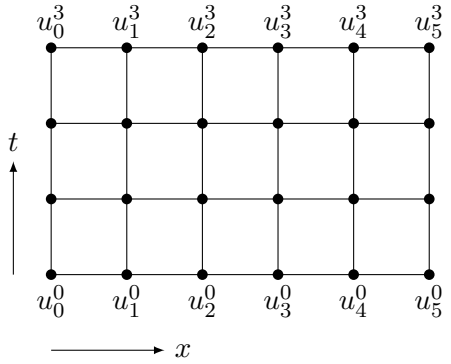

Figure 1: Grid of points (x_j, t_n) on which the solution $u_j^n = u(x_j, t_n)$ is represented.

express the temperature as a function $u(x, y)$. If the thermal conductivity of the material is given by a constant α, then we can show that u observes the *partial differential equation*

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial t^2}.$$

This is called the **heat equation**.

To solve the heat equation, we also need some additional information, in the same way that an ODE needs to known an initial value to give a unique answer. In particular, we need to know two things:

- an **initial condition**: what is the temperature at $t = 0$?
- the **boundary conditions**: what is the temperature at $x = 0$ and $x = L$?

For this assignment, we will assume the following:

- the initial temperature is given by some known function $f(x)$, so that $u(x, 0) = f(x)$;
- we fix the temperature at each end of the rod, so that $u(0, t) = a$ and $u(L, t) = b$, where a and b are constants.

2.1 Solving the heat equation numerically

Although analytic solutions of the heat equation do exist, they are not terribly easy to find. In fact, the solution to this equation was a topic of fundamental research in classical engineering, leading to the development of important techniques such as Fourier series. However, with a computer, we can simply brute force the problem! We have already seen that with finite differences we can compute derivatives in one dimension: it is a fairly trivial extension to consider derivatives in multiple dimensions.

Consider that we split the domain $[0, L]$ into $N + 1$ discrete points, so that $x_0 = 0$, $x_N = L$ and $x_j = j\Delta x$. We can do the same thing for time as well, so that $t_n = n\Delta t$, and Δt is called the **timestep**. This leads to a grid in terms of x and t that is shown in figure 1. As shorthand, let us write that $u(x_j, t_n) = u_j^n$.

2.1.1 Discretising in space

At a point (x_j, t_n), we can use a central difference to approximate the second-order derivative as

$$\frac{\partial^2 u}{\partial x^2}(x_j, t_n) = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2} + \mathcal{O}(\Delta x^2)$$

Then, if we represent the values of u at any time t as a vector $\mathbf{u}(t) = (u(x_1, t), \ldots, u(x_N, t))$, our PDE now reduces to a large system of ordinary differential equations

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \alpha \mathbf{A}\mathbf{u},$$

where the matrix $\mathbf{A}$ can be written as

$$\frac{1}{\Delta x^2}\begin{bmatrix} 0 & 0 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & -2 & 1 \\ 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix}.$$

This is called **semi-discrete form**: we have discretised in space, but not yet in time. Notice two things about $\mathbf{A}$:

- $\mathbf{A}$ is a **tridiagonal matrix**: it has entries only on the diagonal of the matrix, the entries directly above the diagonal and the entries directly below the diagonal. This will prove beneficial in terms of our solver, as we will discuss below.
- The top and bottom rows of the matrix are different than the rest of the matrix. This is so that we can impose the boundary conditions, which we'll see in the next step.

2.2 Discretising in time

We can now apply a number of techniques from the course to discretise the ODE system in time. Although forward Euler is the easiest, we can show that this has quite severe timestep restrictions. Instead, therefore, we will use the θ timestepping method.

Consider the column vector $\mathbf{U}^n := \mathbf{u}(t_n) = (u_0^n, \ldots, u_N^n)$. Then let θ be a constant where $0 \leq \theta \leq 1$. Then the θ scheme reads as:

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \alpha\Delta t \left[\theta \mathbf{A}\mathbf{U}^n + (1-\theta)\mathbf{A}\mathbf{U}^{n+1}\right] \tag{1}$$

Notice that varying θ recovers many of the schemes we have looked at:

- $\theta = 0$ recovers the backward Euler method;
- $\theta = 1$ recovers the forward Euler method;
- $\theta = \frac{1}{2}$ recovers the midpoint rule (also called the Crank-Nicolson rule).

Rearranging eq. (1) for $\mathbf{U}^{n+1}$, and letting $\lambda = \alpha\Delta t$, we must therefore solve the system

$$\underbrace{\left[\mathbf{I} - \lambda(1-\theta)\mathbf{A}\right]}_{\mathbf{M}}\mathbf{U}^{n+1} = \underbrace{\left[\mathbf{I} + \theta\lambda\mathbf{A}\right]}_{\mathbf{R}}\mathbf{U}^n. \tag{2}$$

This is a **matrix equation** $\mathbf{M}\mathbf{U}^{n+1} = \mathbf{b}$, where $\mathbf{b} = \mathbf{R}\mathbf{U}^n$, which we know how to solve!

2.3 Imposing boundary conditions

Until now we have not considered the application of the boundary conditions $u(0, t) = a$ and $u(L, t) = b$. In fact, if we are careful, these come for free: defining the first and last row of $\mathbf{A}$ as a zero matrix, the first row of $\mathbf{M}$ and $\mathbf{R}$ should be $(1, 0, \ldots, 0)$ and the last row $(0, \ldots, 0, 1)$. Therefore, whatever values

we insert into the vector $\mathbf{U}^0$ should be preserved as we integrate in time. However, if necessary then you may need to adjust the right-hand side vector $\mathbf{b}$ so that

$$\mathbf{b} = \mathbf{B}(\mathbf{R}\mathbf{U}^n)$$

where the $\mathbf{B}$ operator sets the boundary conditions at the first and last entry of the vector.

2.4 Putting it all together

To solve the system, we can therefore apply the following sequence of steps:

1. Initialise a vector $\mathbf{U}^0$ which contains an initial condition $u(x_j, 0) = f(x_j)$. Make sure that the boundary conditions are set, so that $u_0^0 = a$ and $u_N^0 = b$.
2. Set a counter $n = 0$.
3. Construct the matrix $\mathbf{M} = [\mathbf{I} - \lambda(1-\theta)\mathbf{A}]$.
4. Construct the vector $\mathbf{b} = [\mathbf{I} + \theta\lambda\mathbf{A}]\mathbf{U}^n$.
5. Solve the system $\mathbf{M}\mathbf{U}^{n+1} = \mathbf{b}$, which is equivalent to eq. (2).
6. Increment n and go to step 2, or terminate if we have reached the final desired time.

3 Tasks

To begin, download the MATLAB archive which contains a template for the functions that you should complete.

> **!** **Do not** adjust the function signature, i.e. the function inputs and outputs. These are designed to be **precisely** what you need for your coursework!

3.1 LU factorisation: `tridiag_factor`

You should first write a function `tridiag_factor`, which has the function signature

```
function [l, d, u] = tridiag_factor(T)
```

This takes as argument a general tridiagonal matrix $\mathbf{T}$. The matrix should be stored as a **normal** MATLAB matrix, i.e. something constructed by calling e.g. `eye(n,n)`. Your routine should compute the LU factorisation of $\mathbf{T}$ so that $\mathbf{T} = \mathbf{L}\mathbf{U}$. As with the Doolittle algorithm, we assume that the diagonal elements of the lower-diagonal matrix $\mathbf{L}$ are 1. Then, you should show that the matrix product

$$\underbrace{\begin{bmatrix} 1 & & & & \\ l_2 & 1 & & & \\ & l_3 & 1 & & \\ & & \ddots & \ddots & \\ & & & l_n & 1 \end{bmatrix}}_{\mathbf{L}} \underbrace{\begin{bmatrix} d_1 & u_1 & & & \\ & d_2 & u_2 & & \\ & & \ddots & \ddots & \\ & & & d_{n-1} & u_{n-1} \\ & & & & d_n \end{bmatrix}}_{\mathbf{U}}$$

is tridiagonal. Finally, set the product equal to the matrix $\mathbf{T}$ above, and equate coefficients to find a difference relation for each of the d_i, u_i and l_i. Your function should return three vectors:

- `l` and `u` will be vectors of length $n-1$ which contain the lower- and upper-diagonal entries l_i and u_i of the LU decomposition, respectively;
- `d` is a vector of length n which contains the diagonal entries d_i of the LU decomposition.

3.2 Solving an LU system: `tridiag_solve`

Next, write a function `tridiag_solve` which has the function signature

```
function x = tridiag_solve(l, d, u, c)
```

This function should take the vectors `l`, `d` and `u` from `tridiag_factor` and perform the forward- and back-substitution phase to compute the solution of the system $\mathbf{Tx} = \mathbf{c}$, where $\mathbf{c}$ is a given right-hand side vector.

3.3 Creating the matrices

In our algorithm to solve the heat equation, we discussed in the previous section that we require two matrices:

- $\mathbf{M} = [\mathbf{I} - \lambda(1-\theta)\mathbf{A}]$, which will be the matrix that will be factored with `tridiag_factor`;
- $\mathbf{R} = [\mathbf{I} + \theta\lambda\mathbf{A}]$, which is the matrix used to compute the right-hand side $\mathbf{b}$ for the solution of the system.

You should write the function `create_matrices` which computes these matrices, and has the signature

```
function [M, R] = create_matrices(n, lambda, theta, dx)
```

The matrices may be created as standard MATLAB matrices, as described in the `tridiag_factor` routine. As arguments the function requires:

- `n`: the number of points used in the space discretisation, so that the domain length $L = (n-1)\Delta x$.
- `lambda`: the value of $\lambda = \alpha\Delta t$;
- `theta`: the value of θ which should be between 0 and 1;
- `dx`: the grid spacing size Δx.

Ensure that the first row and last row of your $\mathbf{M}$ and $\mathbf{R}$ matrices are correct, otherwise your boundary conditions won't be imposed properly!

3.4 Solving the heat equation: `solve_heat`

Write a function `solve_heat` which solves the heat equation using all of the functions above. This has the signature

```
function x = solve_heat(a, b, dt, dx, alpha, theta, ic, t_final)
```

and accepts several arguments:

- `a`: The boundary condition at the left-hand side so that $u(0,t) = a$.
- `b`: The boundary condition at the right-hand side so that $u(L,t) = b$.

- `dt`: The timestep size Δt.
- `dx`: The grid spacing size Δx.
- `alpha`: The value of thermal conductivity α.
- `theta`: The value of θ which should be between 0 and 1.
- `ic`: A column vector that contains the initial condition $\mathbf{U}^0$;
- `t_final`: The final time T up to which to integrate.

The function should return `un`, the solution of the heat equation $\mathbf{U}^n$ at the time T according to the conditions above.

3.5 Validation and report

You should submit a short report which outlines an appropriate validation case for your routines to justify that the cases are working as intended for a selection of use cases. In particular, you should ensure that you validate the error you obtain in space and time for each of the schemes, following the examples in the lab sheets.

This should ideally be in the form of a MATLAB `.mtx` live script, as we have been using in the labs. Otherwise, supply a PDF. Ensure your report is no more than 4 pages in length including all references and citations with appropriate font size and margins.

4 Debugging tips

This project involves many different bits of code.

- Deliberately I have split this into many functions. That is because this is **easier to test**!
- Ensure you use MATLAB to help debug your programs: you can set breakpoints by clicking on the line numbers, which will allow you to stop the program and observe values of variables.
- Also ensure you make use of MATLAB's routines: for example, if you're struggling with the `tridiag_solve` routine, try just using the backslash operator.
- Some exact solutions for the heat equation exist and are reasonably easy to represent. For instance, if $a = b = 0$ and we define $\beta = k\pi/L$ for any integer k, then:

$$u(x, 0) = \sin(\beta x) \Rightarrow u(x, t) = \exp(-\alpha\beta^2 t)\sin(\beta x).$$

5 Marking scheme

Your submission will be assessed on the following criteria:

- Fully working implementation of the `tridiag_factor` function. **[15%]**
- Fully working implementation of the `tridiag_solve` function. **[15%]**
- Fully working implementation of the `create_matrix` function. **[15%]**
- Fully working implementation of the `solve_heat` function. **[15%]**
- Report that adopts appropriate use cases, demonstrates correct solver accuracy and expected convergence. **[25%]**

- Discretionary marks for good programming technique (in terms of efficiency, presentation of code, and structure) and appropriate use of commenting. **[15%]**

You should attempt all of the above routines; incomplete solutions will be given partial credit. Note that these routines will all be marked independently, and you can still receive full credit for complete solutions even if the preceding parts are not fully implemented.

! 'Fully working' in the above is left *deliberately* vague. You should try to consider the different inputs that your functions may receive and act accordingly. For your report, *appropriate use cases* are also vague: you should do some research to explore other possible solutions.

6 Submission

You should submit your report via the online submission point on KEATS by the assignment deadline: **16:00, Thursday 27th July 2023**.