Creation of computational problems for undergraduate Physics students

Aidan Fellows

Department of Physics, University of Bath, Bath, BA2 7AY, United Kingdom.

9 May 2023

Abstract.

A project has been undertaken to develop computational problems for physics students, with the aim of enhancing their computational skills and reinforcing their understanding of lecture-taught modules. The computational problems were designed to be challenging but accessible and cover a range of topics in physics, including condensed matter, thermal physics and statistical mechanics. The problems were created with clear and concise framework in mind and with feedback and guidance from lecturers and students.

1. Introduction

1.1. The importance of computation in physics

Computational methods have some key advantages over analytical methods as physics problems can often involve complex mathematical equations and models that are difficult or impossible to solve analytically, such as systems with many interacting variables. In such cases, computational methods can be used to solve these problems numerically, using numerical algorithms and simulations to approximate the solution. Computational methods can provide analysis for problems involving big data, being used to either verify or develop analytical models or even simulate the behavior of physical systems, which can be useful when experiments are difficult or expensive to carry out.

The field of physics is closely intertwined with mathematics and computation. As such, it is increasingly important for physics students to have a solid foundation in computational thinking and problem solving. Integrating computational problems into the curriculum not only helps students develop these skills, but also enhances their ability to analyze and interpret data, and to develop and test hypotheses.

1.2. The curriculum transformation

The physics course at the University of Bath is undergoing a curriculum transformation (CT) in which key decisions about how the course will be taught heading forwards are determined. A significant proportion of the CT is the "Suggested outline of future scientific computing provision in the post CT programmes" [1], which stresses the desired computing ability of physics students as well as the teaching and activities given to them. Some sections of the document outline the integration of classical lecture-taught modules with scientific computing such as "summative assessment to occur [...] via a set of exercises integrated in the physics and maths courses/problem sheets in semester 2", or "During year 2 taught courses will continue to expose problems whose solution requires numerical analysis beyond calculator usage and that provide practice in scientific computing skills". The document also outlines the coursework that students may be given "Students would undertake an assessed computational mini-project requiring the writing of code to solve a scientific problem [...]. This mini-project would improve student's computational and scientific skills in terms of modelling, coding (in Python) and testing, analysis and interpretation of data, and communication/report writing". Coursework may be required as part of "A course introducing a more formal high/mid-level compiler-based language will be provided (c/c++) and delivered in the Linux environment" like the current physics course, although the outline goes on to comment "There is an argument for this course being optional for some students".

These two areas, creating problems to be implemented in problem sheets and creating coursework to form summative assessment were the focus of this final year project.

2. Best practices

2.1. For creating computational problems

In order to write effective and useful computational problems-sheet questions and computational coursework, a clear and concise framework for how to set up the problems should be considered. Advice that lecturers could consider are:

(i) Any data used in computational problems should be realistic and relevant to the subject matter being studied. This is so the computational model is tested when put in real-world situations (the code doesn't only work when something is travelling faster than light), and that students understand the real-world applications of the computational model.

(ii) Every problem should be approachable to any student, even if to get full marks, or to consider the entirety of what is asked of them is challenging. More advanced students may take the solution further than the expectation regardless. In summative assessment, such as computational coursework, marks could be rewarded for coding solutions that are flexible, well-structured, well-annotated and demonstrate a non-trivial use of more advanced coding concepts while ensuring that less experienced students are not overwhelmed.

(iii) Before assigning computational problems to students, it is important to test the problems to determine their difficulty, how much time is required to complete them to check whether this is appropriate, and to familiarise one's self with approaches to the solution.

(iv) It is necessary to provide an introduction to the problems to ensure students understand why they are being asked to complete them and what is required of the students to complete the task. This includes any specific tools that they need to use, any input or output formats, and any specific requirements for the code (e.g., efficiency, accuracy, etc.). Many times the hardest part of a computational problem is understand what is being asked of you, removing roadblocks to students starting the problem allows them to spend more time concentrating on the important part of the exercise.

(v) Provide sample code or code snippets that demonstrate the key concepts and techniques needed to complete the task. This can be particularly helpful for students who are new to programming or who are not familiar with the specific language or tool being used.

(vi) Provide feedback and guidance to the students as they work on the programming task. This can include providing feedback on their code, answering questions, and providing additional resources or examples to help them better understand the concepts and techniques being used.

(vii) Stress how the programming task relates to the modules that students are studying or to physics as a whole. This will help to keep the students engaged and motivated to complete the task.

Creating computational problems to be implemented in problem sheets may be a tricky task for a number of reasons as many students will not done computer science as a GCSE or A-level, and might not be expecting to write much code, especially in year 1 non-computational lecture modules. Students may also have only written code for summative assessment, as such be unused to the process or importance of coding solutions to problem sheet questions. Lecturers cannot be expected to have significant programming experience or specific knowledge of python, and as such may be unable to help students struggling with computational questions, or be unwilling if helping students to code solutions requires a significant investment of their time.

Due to these concerns attempts were made to introduce the problems appropriately, provide resources, code snippets and advice and produce example solutions to give to students.

2.2. For answering computational problems

Recent studies have found that on average, scientists spend 30% or more of their time developing software, 40% or more of their time using scientific software [2, 3]. As such it is important to develop good programming habits, because they may be impactful in one's scientific career. Any mistakes in writing, validating and using scientific programs may cause serious errors in the conclusion of published papers [4] causing retractions, technical comments and corrections. There are lots of advice on writing good quality code online [5, 6], some advice that students could consider are:

(i) Write programs that other people can read and understand.
  (a) Use meaningful and distinctive names for functions and variables
  (b) Use a consistent code and formatting style
  (c) Use comments to document the design, purpose and implementation of one's code

(ii) Reduce the chance of human error
  (a) Write code to do the work for you and automate human input
  (b) Work in small steps, writing the program out of smaller functions that can be easily tested
  (c) Use version control so that mistakes can be reverted and versions compared

3. Problem sheet question on the Einstein and Debye models of heat capacity

Creating problems for condensed matter was a significant focus of the final year project for a couple of reasons. Condensed matter is an area well suited for computational approach, as it contains many models that can be simulated and the supervisor of this project, Dr. Simon Crampin has taught condensed matter 1, currently a second-year physics module, and as such it was easy to get advice and feedback on questions, working on condensed matter was key feedback in the oral presentation at the end of semester 1.

3.1. Motivation

One scientific computing activity could be on the Einstein and Debye models of heat capacity, for a number of reasons. The Einstein and Debye models are currently taught late in the semester in condensed matter 1 (week 10 of 11), and therefore students may not have spent much time trying to understand the models. It is also a topic that comes up on exam papers, giving students topics that are examinable may motivate them to spend time on the questions. The last time a question was asked on the topic it scored reasonably well, students averaged 60% but the exam feedback form [7] noted "some students maybe did not read the question well enough and failed to describe the assumptions [of the classical, Einstein and Debye model of heat capacity due to atomic motion] at all, whilst I was also surprised to see it stated a few times that the Debye theory predicts $C_v \sim T^{-3}$ at low temperatures. I have not seen that before".

This computational problem on the Einstein and Debye models of heat capacity requires students to read in, use and plot data from scientific experiments, as well as knowledge of fitting curves in python. These are key scientific skills that can be applied across a wide range of problems and topics.

3.2. Introduction to Einstein and Debye models of heat capacity

In 1901 Planck postulated that oscillators have quantised energies [8]: $E_n = n\hbar\omega$. In 1907 Einstein calculated the vibrational heat capacity [9] of a crystal, in particular diamond, assuming it comprised of 3N independent oscillators with Planck's energies (it was later shown $E_n = (n + \frac{1}{2})\hbar\omega$ for vibrating oscillators[10]). By considering the statistical mechanics of an oscillator at temperature T, Einstein proposed:

$$\langle E\rangle = 3N\langle E_{osc}\rangle = \frac{3N\hbar\omega}{e^{\hbar\omega/k_BT} - 1} \tag{1}$$

For which the heat capacity at constant volume follows as:

$$C_v = 3Nk_B\left(\frac{\theta_E}{T}\right)^2 \frac{e^{\theta_E/T}}{(e^{\theta_E/T} - 1)^2} \tag{2}$$

where N is the number of atoms in the solid, k_B is the Boltzmann constant, T is the temperature of the material and θ_E is now known as the Einstein temperature, a material dependent constant and equal to $\hbar\omega/k_B$.

Einstein assumed all oscillators have the same frequency ω, whereas in reality atoms vibrate in solids as collective oscillators (lattice vibrations) at a range of frequencies satisfying a dispersion relation $\omega = \omega(k)$, where k is the wave vector of the oscillation.

A monatomic solid has one longitudinal and two transverse acoustic modes. For low frequencies:

$$g(\omega) = \frac{3V\omega^2}{2\pi^2\bar{c}^3} \tag{3}$$

where $\bar{c}$ is an average speed and V the volume of the solid. Debye assumed this expression held for higher frequencies, dropping to zero at a cut-off frequency ω_D, known as the Debye frequency, that ensured the solid has 3N phonons/lattice vibrations. The average energy in the Debye model is given as:

$$\langle E\rangle = \int \langle E_{osc}\rangle g(\omega)d\omega \tag{4}$$

The heat capacity at constant volume in the Debye model is:

$$C_v = 9Nk_B\left(\frac{T}{\theta_D}\right)^3 \int_0^{\frac{\theta_D}{T}} x^4 \frac{e^x}{(e^x - 1)^2}dx \tag{5}$$

where θ_D, the Debye temperature, is also a material dependent constant.

3.3. Computational problem

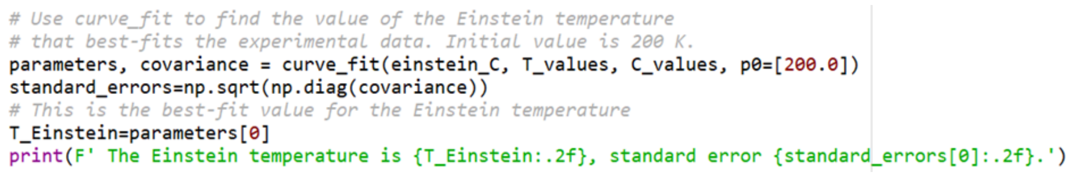
Using Python, read in the data provided in the text file si_c.txt [11] which provides experimental data for temperature (in kelvin) against the heat capacity (in joules) of silicon.

(a) Using the Einstein model and scipy's curve_fit function find the Einstein temperature of silicon and plot the experimental data and the line of best fit from the Einstein model on the same plot.

(b) Using the Debye model and scipy's curve_fit function find the Debye temperature of silicon and add the line of best fit from the Debye model to the plot. Bear in mind that the curvefit function would require an array to be returned if arrays are being passed in as data.

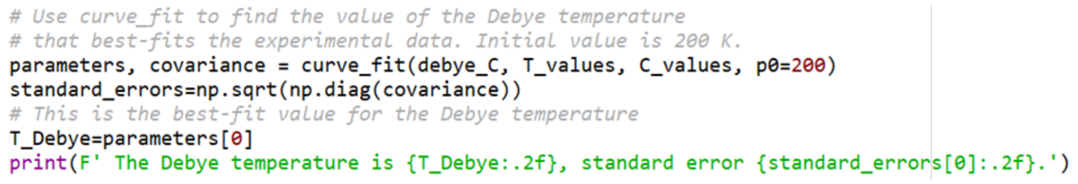
(c) Comment on the differences between the Debye and Einstein fits, and why that might be.

Code snippets shown in Fig. 1 and Fig. 2, the equations of heat capacity at constant volume for the Einstein and Debye models, values for well known constants, example code and the notes for the corresponding condensed matter 1 topic were given to physics students for feedback.



```
# Use curve_fit to find the value of the Einstein temperature
# that best-fits the experimental data. Initial value is 200 K.
parameters, covariance = curve_fit(einstein_C, T_values, C_values, p0=[200.0])
standard_errors=np.sqrt(np.diag(covariance))
# This is the best-fit value for the Einstein temperature
T_Einstein=parameters[0]
print(F' The Einstein temperature is {T_Einstein:.2f}, standard error {standard_errors[0]:.2f}.')
```

Figure 1: Code snippet supplied to students setting up the curve_fit function for the Einstein model



```
# Use curve_fit to find the value of the Debye temperature
# that best-fits the experimental data. Initial value is 200 K.
parameters, covariance = curve_fit(debye_C, T_values, C_values, p0=200)
standard_errors=np.sqrt(np.diag(covariance))
# This is the best-fit value for the Debye temperature
T_Debye=parameters[0]
print(F' The Debye temperature is {T_Debye:.2f}, standard error {standard_errors[0]:.2f}.')
```

Figure 2: Code snippet supplied to students setting up the curve_fit function for the Debye model

3.4. Example solution

Producing a solution to the question was quite challenging as while the coding the Einstein model was easy, there were lots of errors output for the Debye model. This may be due to the interaction between the curve_fit function and using an array in the integral for the Debye equation. To fix this an array is set up with the same size as the experimental data, the value for the specific heat is calculated at each temperature, added to the new array and returned to the curve_fit function. Students may need more guidance for this question as many of the errors seen were new, and the cause of the errors are in functions that they haven't written. Fig. 13 in the appendix is the code written to solve the question and can be given to students on the moodle page as an example and snippets taken from to assist students and Fig. 3 is the plot created by this code.

Using a different function such as scipy.optimize.minimize on the sum of the squares of the deviation between the model and the experimental data instead of scipy.optimize.curvefit gets around the errors produced with the Debye model but the function not been taught to students.

An answer to part (c) may be that the Einstein model assumes each atom in the solid lattice acts as an independent 3D quantum harmonic oscillator and all atoms oscillate with the same frequency, whereas the Debye model treats the vibrations of the atomic lattice of a solid as 3N quantised oscillators, with frequencies satisfying the dispersion relation $\omega = ck$ of long-wavelength acoustic vibrations.

3.5. Feedback

The question along with a couple of code snippets showing how to use the curvefit function were given to students for feedback, only one student gave feedback on this question with the numerical responses found in table 1. The student was in their fourth year and found the question quite difficult commenting "Really nice to see what we've been taught put into practice and give the results we expect", "I think some pointers on part c) would help - what sort of thing should we be looking for - open ended questions often get skipped by students (imo). Also, in the curve_fit code snippets shown, it would be helpful to explain what 'einstein_c' is (The function that calculated the C_v value at that temperature, for example)" and "If the main aim of the questions is to improve physics understanding, bit of programming such as reading from a file could be given to the students, if they want it. Also, perhaps there should be more questions linking back to the course content, to reinforce the exam knowledge, although I don't any specific suggestions".

The high rating for the difficulty of the question as well as the student spending just

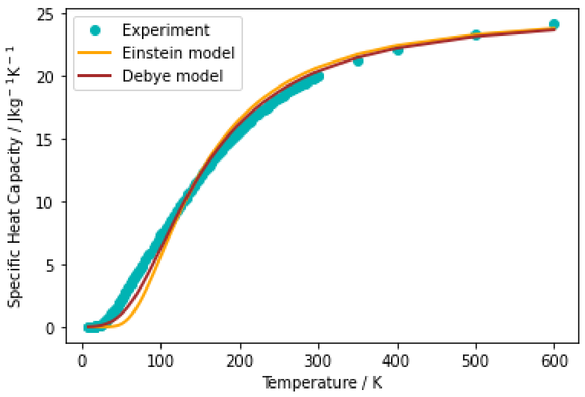


Figure 3: The figure the students completing the activity should produce, showing the experimental heat capacity data along with the best fit models according to the Einstein and Debye theories.

Table 1: Survey response for the Debye and Einstein question

Year of Study	Difficulty /5	How much time did it take you?	How relevant was the question to you? /5	How interesting did you find the problem? /5
4th year	4	40 minutes	1	3

over 40 minutes to answer a problem sheet question was worrisome. The student also marked the question a 1 out of 5 for relevance to them, although this may be because the student answered a second year question as a fourth year student. The feedback was very insightful such as giving more context around the code snippets, and to give students the code for reading data in from a file because as the student points out, ultimately all questions on problem sheets should be about the physics.

The feedback around part (c) was disappointing but understandable, the condensed matter notes were provided to students but better guidance, such as pointing students towards the assumptions that the Einstein and Debye models make, should have been given.

3.6. Improvement and implementation

To implement the question in problem sheets, example code on moodle, as well as help either explicitly in the question, or a different file that they can access should be given to students. Some help that could be provided would be code snippets showing students how to read in the data and separate it into temperature and heat capacity arrays, code snippets showing how to set up the curve_fit function with comments on what each parameter means, a walk-through or code on how to use the scipy.integrate.quad function and explicit help on setting up the debye_C function so that no errors are produced, and perhaps some simple code to plot a graph although this should be second nature to students. With this help, the time it would take for students to find a solution to the question should be greatly reduced, as time spent trying to remember how to read in data, or looking at documentation for integrals in python would be gone.

The question in part (c) should be more clear perhaps rephrasing it as "What are the essential assumptions of, and the differences between the predictions of the Einstein and the Debye theories of heat capacity due to atomic motion? Does this match your results from part (a) and (b)?"

4. Problem sheet question on a 1D diatomic chain and dispersion curves

4.1. Motivation

A suitable topic for a scientific computing activity is the linear chain of masses and the relevant dispersion diagram. Just like the Einstein and Debye models, it is currently taught late in the semester in condensed matter 1 (week 10 of 11), and therefore students may not have spent much time trying to understand the models. It is also a topic that comes up on exam papers.

The last time a question was asked on the topic it scored reasonably well, students averaged 61% but the exam feedback form [12] noted "[...] many students scored well. Nevertheless it felt like many students were answering from memory, rather than having understood the underlying idea, and then implementing in response to the question. [...] Optical vibrations and transverse vibrations were widely cited in part (ii), but degeneracy to explain there only being two acoustic/optical branches was not".

The computational problem that has been defined requires students to set up a matrix and find it's eigenvectors for each value of an array that the student creates. Being comfortable with solving matrix problems computationally is a key scientific computing skill.

4.2. Introduction to the monatomic and diatomic chain

As a simple model of lattice vibrations we can consider a chain of atoms, each with mass M, and with an average spacing a. Let the displacement of atom s from its equilibrium position $x_s = sa$ be the quantity u_s and assume interactions between neighbouring atoms can be described by a spring with spring constant K.

The forces on atom s are $K(u_s - u_{s-1})$ to the left and $K(u_{s+1} - u_s)$ to the right. Equating total force to the right to mass times acceleration gives the equation of motion for atom s:

$$M\frac{d^2u_s}{dt^2} = K(u_{s+1} + u_{s-1} - 2u_s) \tag{6}$$

Assuming a wavelike solution, $u_s = ue^{i(kx_s - \omega t)}$ and substituting produces:

$$\omega = \omega(k) = \sqrt{\frac{2K}{M}}\left|sin\frac{ka}{2}\right| \tag{7}$$

This is known as a dispersion relation, giving the frequency of lattice vibrations that have wavevector k. In total the chain has 3N different lattice vibrations, for a monatomic chain, this is one longitudinal and two transverse.

For a diatomic chain we end up with the equations of motion being two coupled equations that can be put into a matrix and solved:

$$\begin{bmatrix} \frac{2K}{m} & \frac{-K}{m}(1 + e^{-ika}) \\ \frac{-K}{M}(e^{ika} + 1) & \frac{2K}{M} \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \omega^2 \begin{bmatrix} u \\ v \end{bmatrix} \tag{8}$$

By finding the eigenvalues of the matrix for each wavevector, k, one can plot the dispersion curve.

4.3. Computational problem

Using Python, plot the longitudinal modes in the Brillouin zone for a diatomic linear chain by solving the matrix eigenvalue problem for each wavevector k. The functions numpy.linalg.eig and cmath.exp may be used to find the eigenvalues and for complex numbers respectively.

4.4. Example solution

Coding a solution to the question was quite easy, although looking through documentation on how to set up the matrix with complex numbers was required. I also initially made a plot of the eigenvalues against wavevector, k, the eigenvalues of the matrix are ω^2, not ω, this is a mistake that students may make.

In attempting to show that when $m = M$ the dispersion relation of the monatomic chain is recovered issues, were discoverd, this may be due to a reduced and an extended zone scheme, even when $m = M$, the monatomic chain was described with a lattice containing two atoms, therefore there were N/2 k-values forming two bands. As a result, students weren't to do this, instead students could first be asked to plot the dispersion relation for the monatomic chain by using the equation (7) before then being asked to solve the matrix eigenvalue problem for the diatomic chain.

Fig. 14 in the appendix is the code written to solve the question and can be given to students on the moodle page as an example and snippets taken from to assist students and Fig. 4 is the plot created by this code.

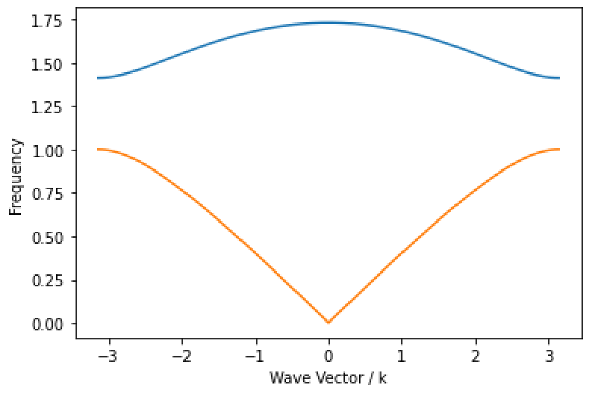


Figure 4: The figure the students completing the activity should produce, showing the dispersion relation of a linear diatomic chain.

4.5. Feedback

The question was given to students for feedback, only one student gave feedback on this question with the numerical responses found in table 2. The student was in their fourth year and found the question quite easy commenting "Very helpful being given the libraries to use for matrices and complex numbers, saves looking it up myself.",

Table 2: Survey response for the diatomic chain question

Year of Study	Difficulty /5	How much time did it take you?	How relevant was the question to you? /5	How interesting did you find the problem? /5
4th year	2	20-30 minutes	1	3

“It would be nice being told what parameters to use in the matrix library, although it seems reasonable that students should just look that up for themselves. Also, maybe give students a little tip that they’ll need to separate the returned eigen values into the two bands to be plotted.” and “[The question] Wasn’t relevant to me as I did this unit 2 years ago”.

While the student found being pointed towards numpy.linalg.eig and cmath.exp helpful they also commented asking to provide students with what parameters to use in the matrix library, with how the question was asked, students would have to spend time looking at documentation rather than thinking about the physics. More help could be given using code snippets to show how to use the two functions, for example, imaginary numbers such as -i are represented in python as -1j, I also gave no description of what cmath.exp does, while it should be apparent if the student looks the function up, this should just be provided to students.

I have mixed feelings about giving a tip to students about separating the eigenvalues into two arrays, at least in the framing of the question as it is a defining feature of the diatomic chain. For the monatomic chain this is not necessary as it has only one longitudinal mode, in other words if a student understands the physics, this tip is not needed, and if a student doesn’t understand the physics, it would be better if they struggle but gain a better understanding than have the correct answer with ease. Perhaps a compromise could be made where a document containing tips or a walk-through guide could be provided to students.

Another tip that could be given is the eigenvectors of the matrix are ω^2, whereas students are asked to plot ω. I also used the values a=1, m=1, M=2 and K=1, this could either be given to students so that the shape of the plots are replicated.

4.6. Extensions

The problem could be extended beyond the course content by looking at a triatomic linear chain, this is an example where solving the equations of motion analytically becomes a nightmare, and as such a computational approach may be preferred.

One could also look at the transverse vibrations for the diatomic chain, although this should not be any more complex than the longitudinal.

Finally an animation of the lattice vibrations could be made, with sliders to change the important variables. Both producing animations and sliders are outside the scope of coding taught to physics students but python is capable of both, and a code snippet could be provided for students to use in their solution.

4.7. Improvement and implementation

To implement the question in problem sheets, example code on moodle, as well as help either explicitly in the question, or a different file that they can access should be given to students. Some help that could be provided would be code snippets showing students how to use numpy.linalg.eig with comments on what each parameter means, explicit help with how to define complex numbers in python, a brief description of cmath.exp and perhaps some simple code to plot a graph. I would also be more clear that students will need to produce the array of k-values in order to find the eigenvalues for each wavevector, k.

Even without this help it only took the student who submitted feedback 20 to 30 minutes, this feels like a good amount of time for students to spend on a problem sheet question, although quicker may be better.

5. Percolation theory coursework

5.1. Motivation

As a part of this final year project, potential summative assessment to be given as coursework is outlined. This coursework could be used as an option for the python computational mini-project or for the module teaching c/c++.

Percolation theory was chosen because of the large scope of the topic, meaning a student could do a self-led investigation, it's relationship with modules students are currently taught such as statistical mechanics, diffusion-limited aggregation in computational physics B and PH40073 Mathematical physics, and because I find the topic interesting.

Students would be expected to spend around 25 hours doing the python mini-computational project which includes writing a report. A good practice would be the coursework should be set up to take less time than students are given as more able students can extend and polish their solution, but every student should have a solution.

There is a lot of python code about Percolation Theory written online [13, 14, 15], so one has to be careful setting coursework on it. However there are reasons why I am not too concerned with students using other people's code as the basis of their own. One reason is that many find it difficult to understand their own code, let alone understanding another's and editing it to solve the coursework, another reason is that it should be obvious if multiple students have copied the same code. If this is felt to be an issue the coursework can be edited such that students are asked a question that hasn't been answered before as the scope for questions in percolation theory is large.

5.2. Introduction

Percolation theory takes it's name from percolation, the movement and filtering of fluid through porous materials. One question that can describe percolation theory is: if a liquid is poured onto a porous material, will it make it's way from cavity to cavity and reach the bottom?

Percolation theory has a large number of applications, such as a model for forest fires, phase transitions, critical phenomena, distribution of oil or gas in porous rocks or diffusion in disordered media [16]. The study of percolation theory involves a range of mathematical techniques, including graph theory, probability theory, and statistical mechanics.

To model cavities in a porous medium, a plane can be split into an N by N grid. Assuming the occupation of sites is random and independent we can define p as the probability of a site being occupied. Fluid can flow from one occupied cell to another occupied cell if they are adjacent. A cluster can be defined as a group of occupied neighbour cells. Here neighbour means squares with one common side rather than at a corner, so called "nearest neighbour sites on the square lattice". If a cluster extends from the top to the bottom or from the left to the right of the grid, one says that the cluster percolates.

The percolation threshold, p_c, is defined as the probability of each site being occupied when a percolating cluster is formed for the first time. For a very large square lattice the value is well known.

5.3. Computational problem

In python for the mini-computation project or in c/c++ for the course introducing a more formal compiler-based language:

(a) Write a program that produces a grid of 5x5 cells in which each cell has the same probability of being occupied, p, and determines whether a percolating cluster exists, in any direction, through the grid.

(b) Edit your program so that it creates a plot of the chance that percolation occurs (run the code many times and take the radio of runs where percolation occurs over the total number of runs) against p for a 5x5, 10x10 and 15x15 grid. An example of the corresponding plot is shown in Fig. 6.

5.4. Example solution

Fig. 5 is an example of two plots created by code written to solve question (a). With an infinite square lattice the (site) percolation threshold p_c is about 0.59. Fig. 6 is the plot created by code written to solve question (b), from it one can see that as the array becomes larger, the first probability, p, when a percolating cluster is formed is tending towards 0.59. Fig. 15 and Fig. 16, in the appendix, together are the code that produces a solution for questions (a) and (b).

The code is quite tricky to produce as it relies on labeling (giving every cell an ID) all the occupied cells, relabeling them according to whether they are touching other occupied cells to form clusters and then relabeling them again so that any two clusters that touch have the same label (ID). This may require guidance from the module lead such that students understand labeling clusters.

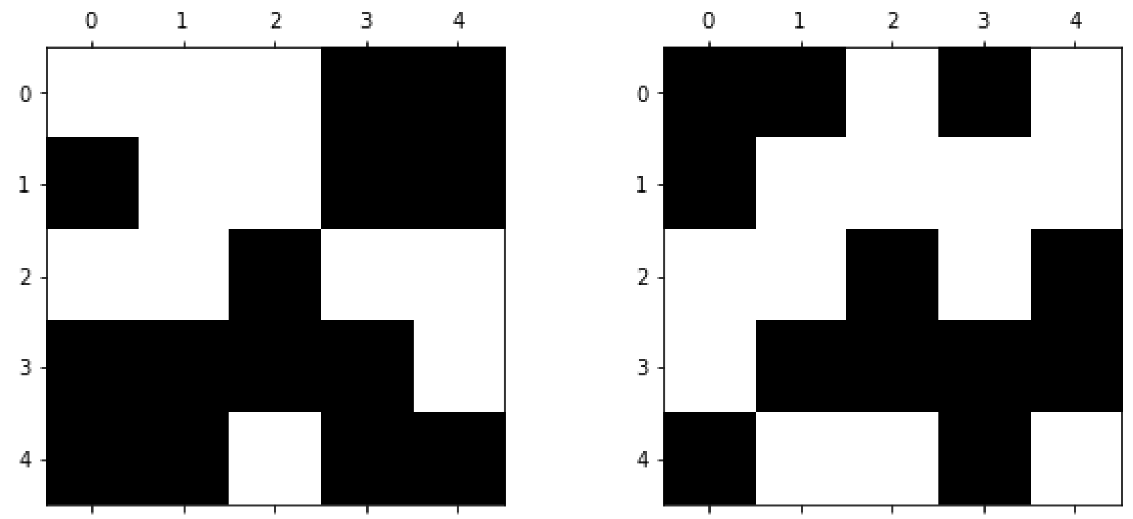


Figure 5: A figure showing two examples of a 5x5 grid that students completing the question (a) should produce, where the black cells represent occupied cells (cavities in the case of water flowing through a porous rock) and the white cells represent unoccupied cells. In the left image there is one cluster that percolates from left to right (or right to left) and in the right image there are no percolating clusters.

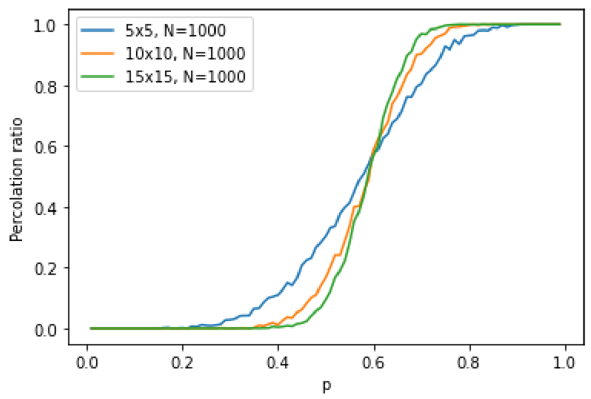


Figure 6: The figure the students completing question (b) should produce, showing percolation ratio (number of runs where percolation occurs over number of runs) against p for a 5x5, 10x10 and 15x15 grid, where p ranges from 0.01 to 0.99.

5.5. Extensions

Currently the model only has two types of cells, occupied and unoccupied, but this is not a requirement of percolation theory. Stauffer suggests modeling forest fires, by having the occupied cells made up of burnt and unburnt trees, meaning that a fire could die out and as such one could investigate the lifetime of forest fires against p. The definition of a cluster could also be changed such that a fire could jump small gaps between trees. Or the cells that make up a cluster must have two adjacent occupied cells.

Instead of using a square lattice, instead one could use a honeycomb, or a triangular or a simple cubic lattice or even higher dimensions. One could also instead look at what is called 'bond percolation' rather than what has been talked about here, the so called 'site percolation'.

6. Thermal transfer coursework

As part of the advanced problem solving module, PH40083, students are asked to come up with novel questions to investigate and solve for coursework. One of my friends, Mr James Brooks, was discussing possible questions and decided on "Is it possible to cook a bird out of the sky with a large mirror?". After attempting an analytical solution, he chose to solve the issue of thermal transfer computationally. After working through the logic of implementing my own solution to the thermal transfer problem, I thought it was a good question to ask students.

6.1. Motivation

Thermal transfer and the heat equation is a topic that many students will have an intuitive understanding of, because of this, creating and investigating a computational model might be an interesting activity for students as their understanding will guide the interpretation of their results, for example if a result is unphysical. The coursework will also help students to quantify thermal transfer and introduce them to concepts explored in simulation techniques and used in computational astrophysics such as the finite-difference method.

6.2. Introduction

The heat equation is given as:

$$\frac{\partial u}{\partial t} - \alpha \nabla u = 0 \tag{9}$$

In 2D this can be written as:

$$\frac{\partial u}{\partial t} - \alpha \left(\frac{\partial^2 u}{\partial x^2} \frac{\partial^2 u}{\partial y^2} \right) = 0 \tag{10}$$

where u is the quantity that we want to know, t, is time, and α is diffusivity constant. Using the finite-difference method (FDM) we find:

$$\frac{u_{i,j}^{k+1} - u_{i,j}^{k}}{\Delta t} - \alpha \left(\frac{u_{i+1,j}^{k} - 2u_{i,j}^{k} + u_{i-1,j}^{k}}{\Delta x^2} + \frac{u_{i,j+1}^{k} - 2u_{i,j}^{k} + u_{i,j-1}^{k}}{\Delta y^2} \right) = 0 \tag{11}$$

if $\Delta x = \Delta y$ then the equation simplifies to:

$$u_{i,j}^{k+1} = \gamma \left(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} - 4u_{i,j}^{k} \right) + u_{i,j}^{k} \tag{12}$$

where:

$$\gamma = \alpha \frac{\Delta t}{\Delta x^2} \tag{13}$$

This will be numerically stable when:

$$\Delta t \leq \frac{\Delta x^2}{4\alpha} \tag{14}$$

6.3. Computational problem

In python for the mini-computation project or in c/c++ for the course introducing a more formal compiler-based language:

(a) Model the thermal transfer across a square array with one hot edge, and a square array with four hot edges and plot the two after 100 iterations.

(b) Write code to model thermal transfer across a circle with a hot circumference. This is quite challenging as a 2d array will be in the shape of a square, and heat should not transfer outside the bounds of the circle.

(c) Edit your code to add heat to the circumference of the circle at each time step, and if the heat of a cell exceeds an 'ablation temperature' assume that the cell is vaporised and no longer conduct heat from or to this cell. The heat added at each time step must then be added to the circumference of the smaller circle.

6.4. Example solution

The question is very open, with few parameters or quantities being defined, this relies on students making sensible choices and producing outputs that best shows their work. An example solution to part (a) is shown in Fig. 7, with the figure caption quantifying some of the choices made in my solution. Students writing a report on the problem should specify and justify any choices that made, especially in regards to variables, such as α, the diffusivity constant or the number of iterations the code runs for.

I wrote two pieces of code, one for the thermal transfer across a square, shown in Fig. 17 in the appendix and one for the thermal transfer across a circle, shown in Fig. 18 in the appendix. Students could be asked to write one piece of code to perform both.

An example solution to part (b) is shown in Fig. 8 and an example solution to part (c) is shown in Fig. 9.

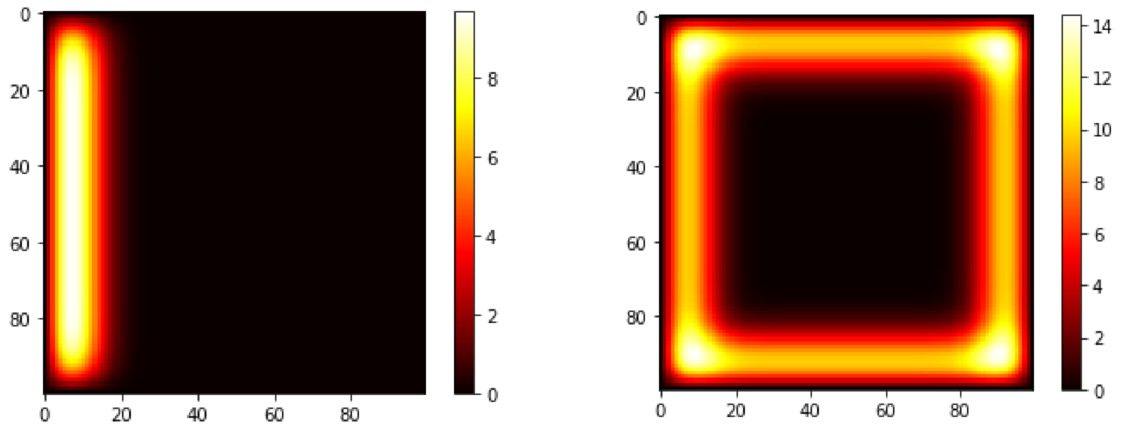


Figure 7: Figures the students completing question (a) could produce. Here thermal transfer is modelled across a 100x100 array after one-hundred iterations. For the left image, one side of the array is given a temperature of one-thousand, for the right image, all four sides of the array are given a temperature of one-thousand.

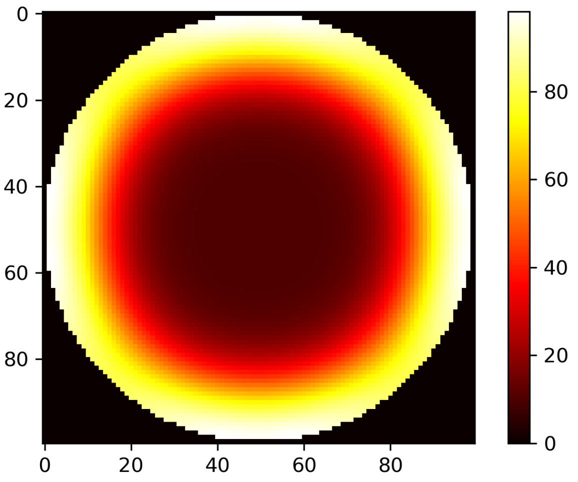


Figure 8: The figure the students completing question (b) could produce. Here thermal transfer is modelled across a 100x100 array after two-hundred iterations. Thermal transfer does not occur outside the bounds of the circle. Every cell of the circle is given an initial temperature of 10 to define the bounds of the circle (cells with a temperature of 0 are outside) and 1000 is added to the circumference.

6.5. Extension

The coursework could be extended into 3d or framed more heavily as an investigation into the interesting results that a simple thermal transfer model produces. For example investigating how much time it takes for a significant amount of heat reaches the centre of the array, or how long until thermal equilibrium is achieved for different sizes of array. Students could also attempt to answer a real-life problem with the model such as cooking things in the oven, or asteroids and spacecraft falling to earth. Brook's question

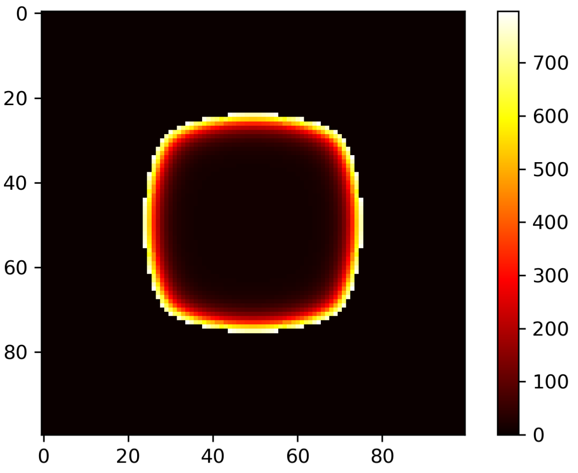


Figure 9: The figure the students completing question (c) could produce. Here thermal transfer is modelled across a 100x100 array after two-hundred iterations. Every cell of the circle is given an initial temperature of 10 to define the bounds of the circle (cells with a temperature of 0 are outside) and 100 is added to the circumference of the shrinking circle after every iteration. If the temperature of any cell is higher than 800, the temperature is set to 0 and thermal transfer no longer occurs on that cell.

was about shooting a bird out of the sky using a large mirror and seeing whether it was cooked before hitting the ground.

7. Incomplete questions

7.1. Planck's Radiation Law

One topic explored for problem sheet questions was Planck's Radiation Law:

$$B(\lambda, T) = \frac{2hc^2}{\lambda^5} \frac{1}{exp(hc/\lambda kT) - 1} \tag{15}$$

where students are asked to recreate a plot of intensity against wavelength for a 7000K blackbody. Ultimately while I did feel like it helped me understand the radiation law better, the code was just implementing an equation, and after asking Dr. Steve Davis about creating computational problems for Stars and Stellar evolution or Particles, Nuclei and Stars, was told that such modules just require a calculator, not particularly suitable for a computational approach. Fig. 10 is the graph created by the code, for your interest.

7.2. Planets and Exoplanets

After reaching out to many lecturers, Philippe Blondel gave an enthusiastic response and offered lots of help, however he requested the creation of python programs for use as learning aids. Students could run the code created, change parameters and see the

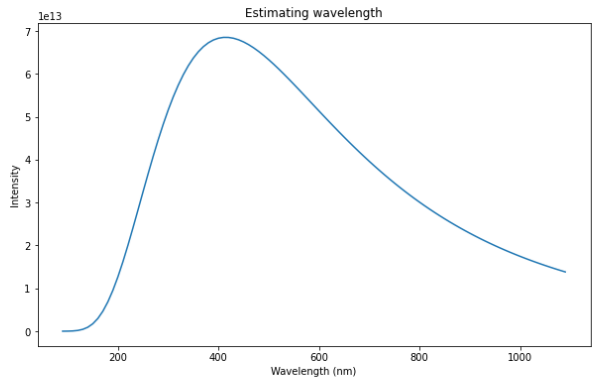


Figure 10: The figure students completing the Planck's Radiation Law question could produce showing intensity against wavelength for a 7000K blackbody.

results as a way of doing a deep dive into certain equations and therefore coming to understand them.

One program "allows students to select particular beam widths (for example for a radar) and see how much of an extended target they can see as it moves away. This would include geometric spreading and attenuation. Students can then add their own particular ranges of interest, and compare that with the problem I would write to accompany it. It could be possible to extend the scope by also including Doppler broadening as a function of a rotation of the object and its aspect angle relative to the imaging wave. I often have problem making people understand how important beam width is so any help there will be useful to all students".

The other program "would be to show 1-D and 2-D examples of fractal dimensions. Being able to slide the fractal dimension from 1 to 1.9, for example, would allow me to demonstrate how a profile becomes more and more complex. Same thing with a 2-D surface".

I felt that this was outside the scope of my final year project, but found the programs interesting and with more time would attempt to create them. I however found myself occupied with other aspects, for example feedback I received after my oral presentation was to work more closely with my supervisor Dr. Simon Crampin on problem sheet questions for condensed matter.

One aspect that did intrigue me was creating and investigating fractals in python see Fig. 11 and Fig. 12, which already forms the basis of coursework in computational physics A and B.

8. Future possibilities

Computational questions for problem sheets should be written for every module where the lecturer believes it's suitable. For this final year project, modules in year 2 semester 2 were prioritised, because as third-year students we took those modules recently and in order to get feedback, my partner Jacob Rae created problems for electromagnetism

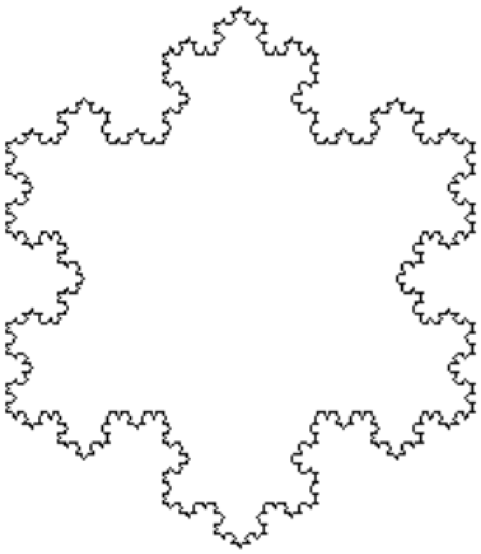


Figure 11: A Koch Curve created in python

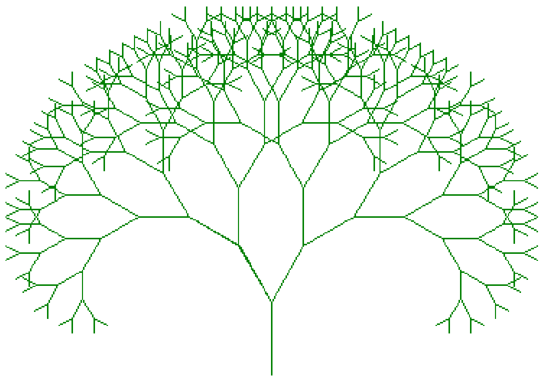


Figure 12: A fractal tree created in python

1 and the second half of planets and exoplanets. Working closely with lecturers is recommended, as they best know the content, the topics students commonly struggle with and are responsible for the problem sheets.

All of the code created as solutions to the problems discussed are accessible to Dr. Simon Crampin at `X:\Physics \StudentProjects \Final Year\2022-23\ScientificComputing` and on my github page[17].

References

[1] S. Crampin, S.R. Davis, M. Mucha-Kruczynski, V. Scowcroft, P. Salmon, and D. Skryabin. Suggested outline of future scientific computing provision in the post ct programmes, 2022.

[2] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson. How do scientists develop and use scientific software? *SECSE*, pages 1–8, 2009.

[3] P. Prabhu et al. A survey of the practice of computational science. *Supercomputing Conference*, pages 1–12, 2011.

[4] Z. Merali. Computational science: ...error. *Nature*, 467:775–777, 2010.

[5] R.H. Landau, M.J. Paez, and C.C. Bordeianu. *Computational Physics : Problem Solving with Python*. John Wiley and Sons, 2015.

[6] G. Wilson, D.A. Aruliah, C.T. Brown, N.P. Chue Hong, M. Davis, and R.T. Guy. Best practices for scientific computing. *PLoS Biol*, 12(1), jan 2014.

[7] S. Crampin. Ph20017 ph20063 - exam feedback form - s2 - 2017_2018.

[8] M. Planck. On the law of distribution of energy in the normal spectrum [translated]. *Annalen der Physik*, 309(3):553–563, 1901.

[9] A. Einstein. Planck's theory of radiation and the theory of specific heat [translated]. *Annalen der Physik*, 800(22):180–190, 1907.

[10] M. Planck. Über die begründung des gesetzes der schwarzen strahlung. *Annalen der Physik*, 342:642–656, 1912.

[11] D.R. Lide. *Handbook of Chemistry and Physics 72nd Edition*. CRC Press, 1991.

[12] S. Crampin. Ph20017 ph20063 - exam feedback form - s2 - 2018_2019.

[13] Tony Chouteau. Python percolation, 2023. `https://github.com/TonyChouteau/PythonPercolation`.

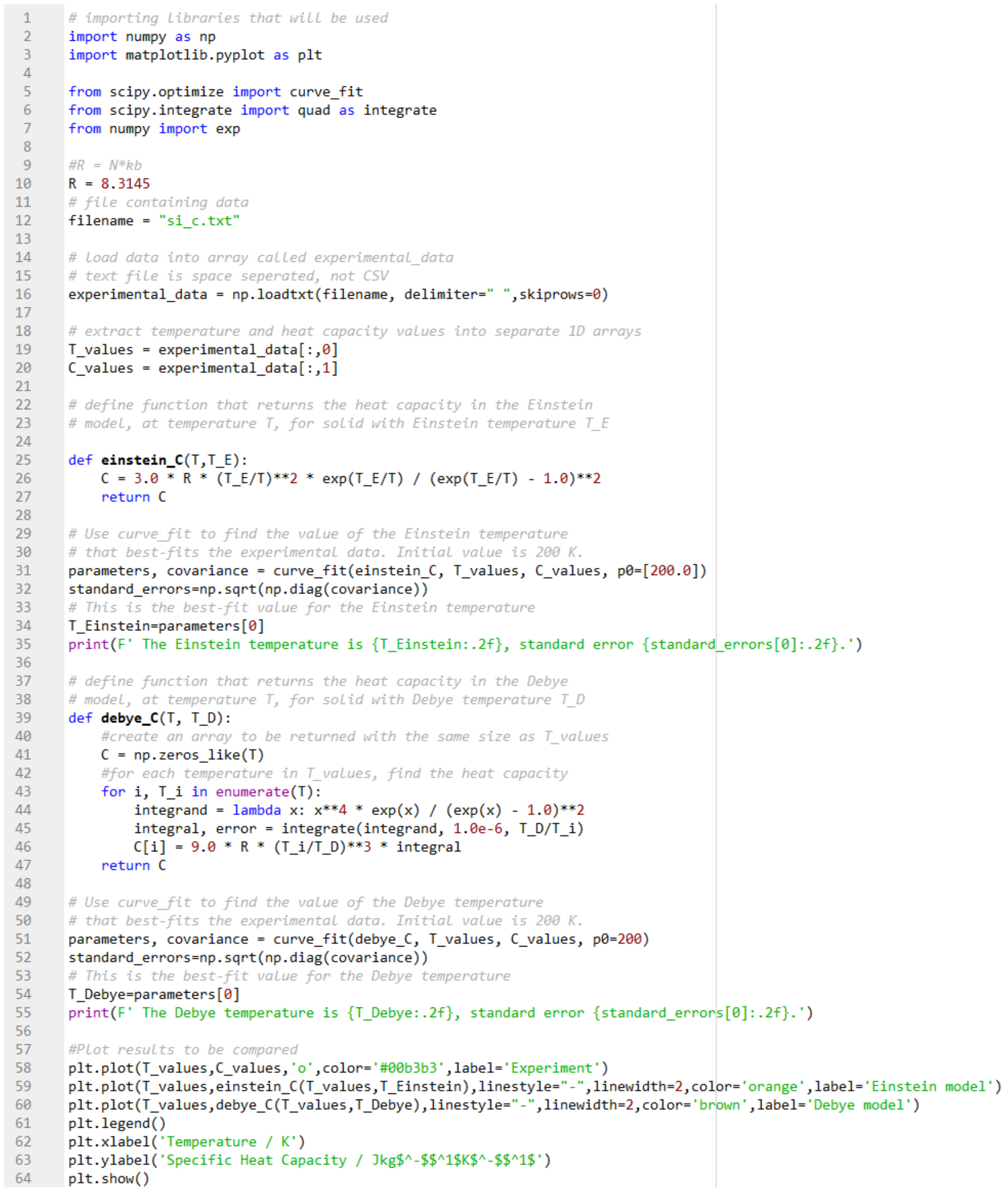
[14] Annika Monari. Percolation-theory, 2023. `https://github.com/annikamonari/Percolation-Theory`.

[15] Ussserrr. percolation-python, 2023. `https://github.com/ussserrr/percolation-python`.

[16] A. Aharony D., Stauffer. *Introduction To Percolation Theory*. London: Taylor and Francis, 1992.

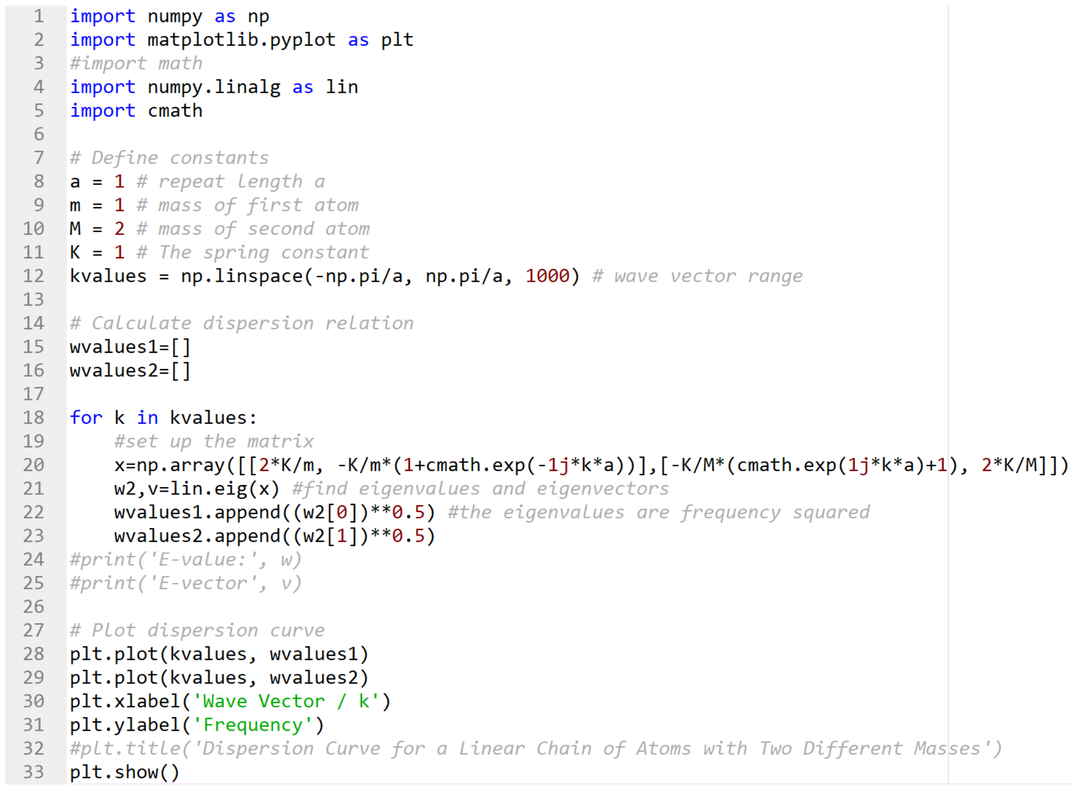
[17] Aidan Fellows. Finalyearproject, 2023. `https://github.com/AidanFe/FinalYearProject`.

9. Appendix



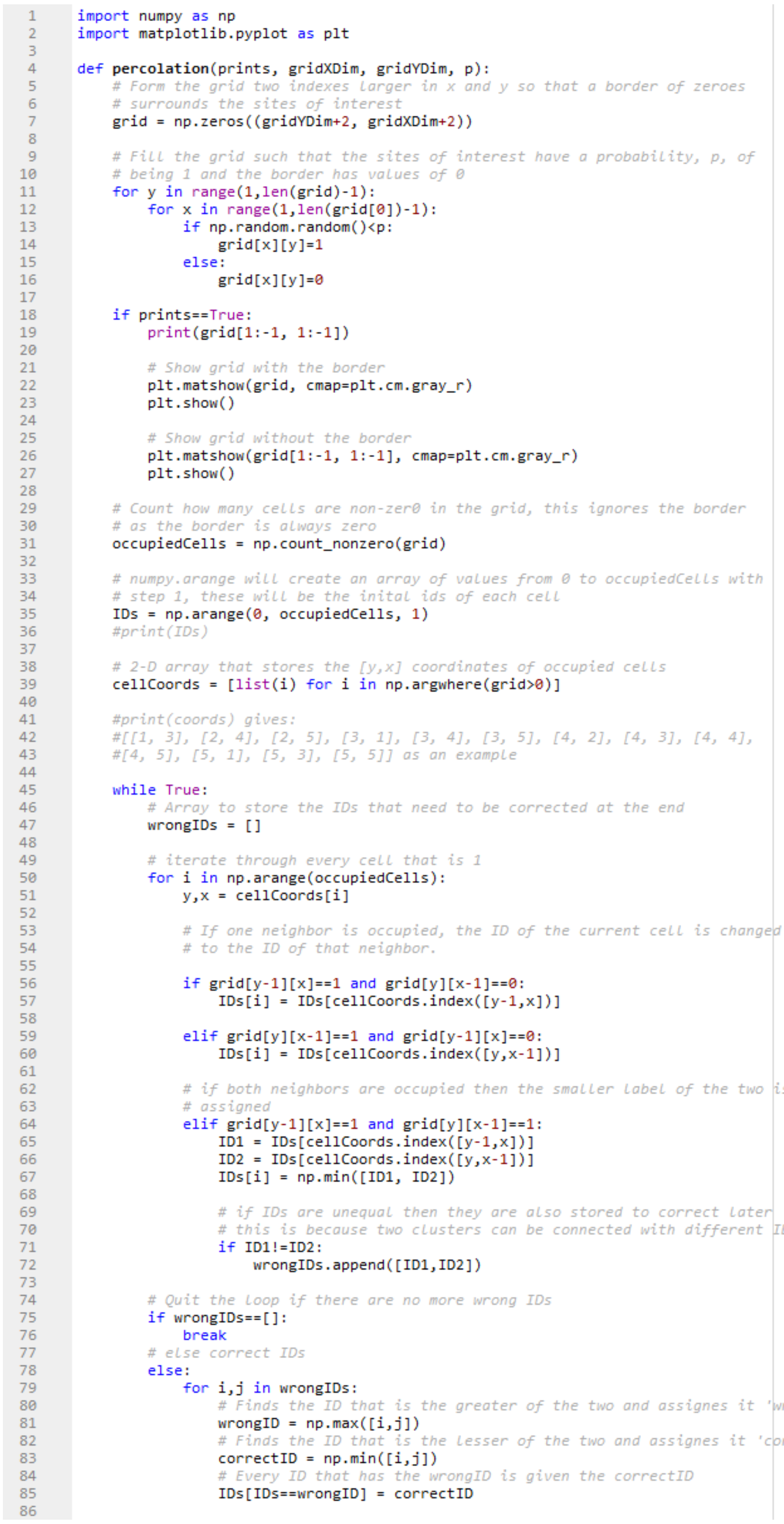
```python
# importing libraries that will be used
import numpy as np
import matplotlib.pyplot as plt

from scipy.optimize import curve_fit
from scipy.integrate import quad as integrate
from numpy import exp

#R = N*kb
R = 8.3145
# file containing data
filename = "si_c.txt"

# load data into array called experimental_data
# text file is space seperated, not CSV
experimental_data = np.loadtxt(filename, delimiter=" ",skiprows=0)

# extract temperature and heat capacity values into separate 1D arrays
T_values = experimental_data[:,0]
C_values = experimental_data[:,1]

# define function that returns the heat capacity in the Einstein
# model, at temperature T, for solid with Einstein temperature T_E

def einstein_C(T,T_E):
    C = 3.0 * R * (T_E/T)**2 * exp(T_E/T) / (exp(T_E/T) - 1.0)**2
    return C

# Use curve_fit to find the value of the Einstein temperature
# that best-fits the experimental data. Initial value is 200 K.
parameters, covariance = curve_fit(einstein_C, T_values, C_values, p0=[200.0])
standard_errors=np.sqrt(np.diag(covariance))
# This is the best-fit value for the Einstein temperature
T_Einstein=parameters[0]
print(F' The Einstein temperature is {T_Einstein:.2f}, standard error {standard_errors[0]:.2f}.')

# define function that returns the heat capacity in the Debye
# model, at temperature T, for solid with Debye temperature T_D
def debye_C(T, T_D):
    #create an array to be returned with the same size as T_values
    C = np.zeros_like(T)
    #for each temperature in T_values, find the heat capacity
    for i, T_i in enumerate(T):
        integrand = lambda x: x**4 * exp(x) / (exp(x) - 1.0)**2
        integral, error = integrate(integrand, 1.0e-6, T_D/T_i)
        C[i] = 9.0 * R * (T_i/T_D)**3 * integral
    return C

# Use curve_fit to find the value of the Debye temperature
# that best-fits the experimental data. Initial value is 200 K.
parameters, covariance = curve_fit(debye_C, T_values, C_values, p0=200)
standard_errors=np.sqrt(np.diag(covariance))
# This is the best-fit value for the Debye temperature
T_Debye=parameters[0]
print(F' The Debye temperature is {T_Debye:.2f}, standard error {standard_errors[0]:.2f}.')

#Plot results to be compared
plt.plot(T_values,C_values,'o',color='#00b3b3',label='Experiment')
plt.plot(T_values,einstein_C(T_values,T_Einstein),linestyle="-",linewidth=2,color='orange',label='Einstein model')
plt.plot(T_values,debye_C(T_values,T_Debye),linestyle="-",linewidth=2,color='brown',label='Debye model')
plt.legend()
plt.xlabel('Temperature / K')
plt.ylabel('Specific Heat Capacity / Jkg$^-$$^1$K$^-$$^1$')
plt.show()
```

Figure 13: Example code that produces a solution for question (a) and (b) on the Einstein and Debye models of heat capacity



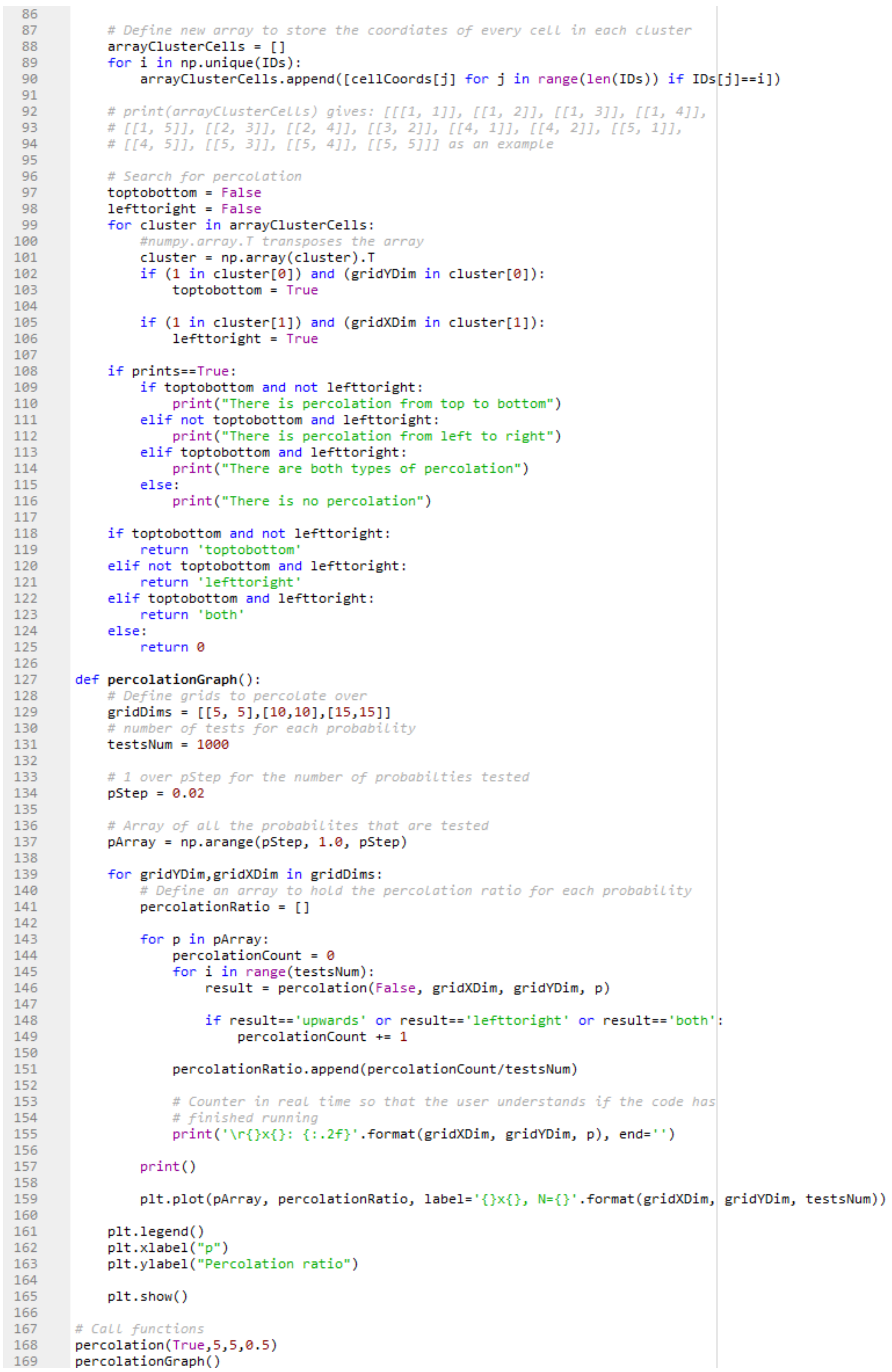
```python
import numpy as np
import matplotlib.pyplot as plt
#import math
import numpy.linalg as lin
import cmath

# Define constants
a = 1 # repeat length a
m = 1 # mass of first atom
M = 2 # mass of second atom
K = 1 # The spring constant
kvalues = np.linspace(-np.pi/a, np.pi/a, 1000) # wave vector range

# Calculate dispersion relation
wvalues1=[]
wvalues2=[]

for k in kvalues:
    #set up the matrix
    x=np.array([[2*K/m, -K/m*(1+cmath.exp(-1j*k*a))],[-K/M*(cmath.exp(1j*k*a)+1), 2*K/M]])
    w2,v=lin.eig(x) #find eigenvalues and eigenvectors
    wvalues1.append((w2[0])**0.5) #the eigenvalues are frequency squared
    wvalues2.append((w2[1])**0.5)
#print('E-value:', w)
#print('E-vector', v)

# Plot dispersion curve
plt.plot(kvalues, wvalues1)
plt.plot(kvalues, wvalues2)
plt.xlabel('Wave Vector / k')
plt.ylabel('Frequency')
#plt.title('Dispersion Curve for a Linear Chain of Atoms with Two Different Masses')
plt.show()
```

Figure 14: Example code that produces a solution for the question on the dispersion relation of the linear diatomic chain



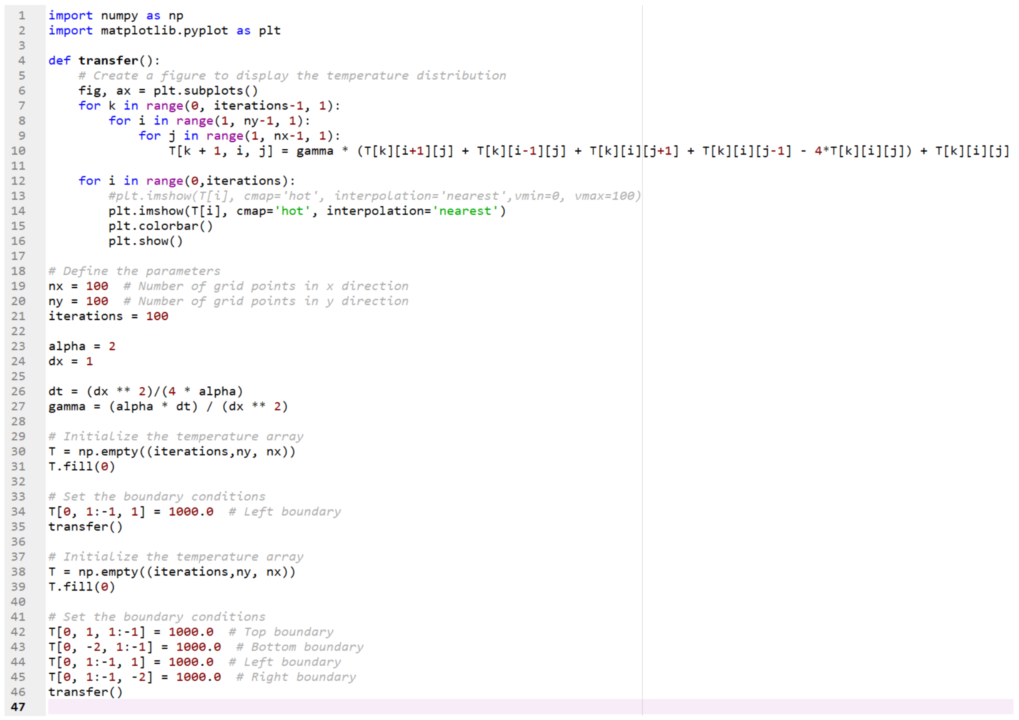
```python
import numpy as np
import matplotlib.pyplot as plt

def percolation(prints, gridXDim, gridYDim, p):
    # Form the grid two indexes larger in x and y so that a border of zeroes
    # surrounds the sites of interest
    grid = np.zeros((gridYDim+2, gridXDim+2))

    # Fill the grid such that the sites of interest have a probability, p, of
    # being 1 and the border has values of 0
    for y in range(1,len(grid)-1):
        for x in range(1,len(grid[0])-1):
            if np.random.random()<p:
                grid[x][y]=1
            else:
                grid[x][y]=0

    if prints==True:
        print(grid[1:-1, 1:-1])

        # Show grid with the border
        plt.matshow(grid, cmap=plt.cm.gray_r)
        plt.show()

        # Show grid without the border
        plt.matshow(grid[1:-1, 1:-1], cmap=plt.cm.gray_r)
        plt.show()

    # Count how many cells are non-zer0 in the grid, this ignores the border
    # as the border is always zero
    occupiedCells = np.count_nonzero(grid)

    # numpy.arange will create an array of values from 0 to occupiedCells with
    # step 1, these will be the inital ids of each cell
    IDs = np.arange(0, occupiedCells, 1)
    #print(IDs)

    # 2-D array that stores the [y,x] coordinates of occupied cells
    cellCoords = [list(i) for i in np.argwhere(grid>0)]

    #print(coords) gives:
    #[[1, 3], [2, 4], [2, 5], [3, 1], [3, 4], [3, 5], [4, 2], [4, 3], [4, 4],
    #[4, 5], [5, 1], [5, 3], [5, 5]] as an example

    while True:
        # Array to store the IDs that need to be corrected at the end
        wrongIDs = []

        # iterate through every cell that is 1
        for i in np.arange(occupiedCells):
            y,x = cellCoords[i]

            # If one neighbor is occupied, the ID of the current cell is changed
            # to the ID of that neighbor.

            if grid[y-1][x]==1 and grid[y][x-1]==0:
                IDs[i] = IDs[cellCoords.index([y-1,x])]

            elif grid[y][x-1]==1 and grid[y-1][x]==0:
                IDs[i] = IDs[cellCoords.index([y,x-1])]

            # if both neighbors are occupied then the smaller label of the two is
            # assigned
            elif grid[y-1][x]==1 and grid[y][x-1]==1:
                ID1 = IDs[cellCoords.index([y-1,x])]
                ID2 = IDs[cellCoords.index([y,x-1])]
                IDs[i] = np.min([ID1, ID2])

                # if IDs are unequal then they are also stored to correct later
                # this is because two clusters can be connected with different IDs
                if ID1!=ID2:
                    wrongIDs.append([ID1,ID2])

        # Quit the loop if there are no more wrong IDs
        if wrongIDs==[]:
            break
        # else correct IDs
        else:
            for i,j in wrongIDs:
                # Finds the ID that is the greater of the two and assignes it 'wrongID'
                wrongID = np.max([i,j])
                # Finds the ID that is the lesser of the two and assignes it 'correctID'
                correctID = np.min([i,j])
                # Every ID that has the wrongID is given the correctID
                IDs[IDs==wrongID] = correctID

```

Figure 15: Part one of example code that produces a solution for question (a) and (b) on percolation theory



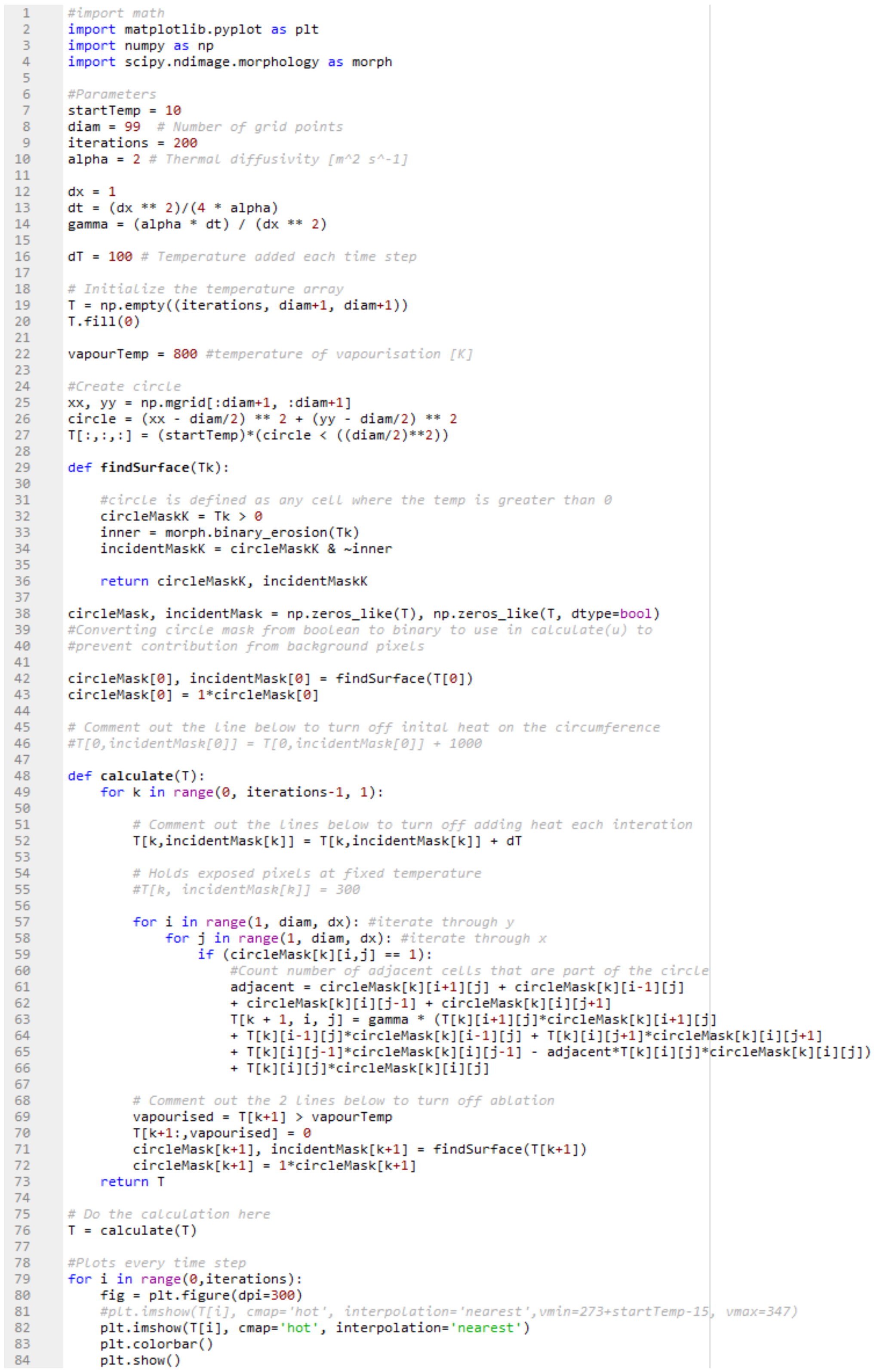
```python

    # Define new array to store the coordiates of every cell in each cluster
    arrayClusterCells = []
    for i in np.unique(IDs):
        arrayClusterCells.append([cellCoords[j] for j in range(len(IDs)) if IDs[j]==i])

    # print(arrayClusterCells) gives: [[[1, 1]], [[1, 2]], [[1, 3]], [[1, 4]],
    # [[1, 5]], [[2, 3]], [[2, 4]], [[3, 2]], [[4, 1]], [[4, 2]], [[5, 1]],
    # [[4, 5]], [[5, 3]], [[5, 4]], [[5, 5]]] as an example

    # Search for percolation
    toptobottom = False
    lefttoright = False
    for cluster in arrayClusterCells:
        #numpy.array.T transposes the array
        cluster = np.array(cluster).T
        if (1 in cluster[0]) and (gridYDim in cluster[0]):
            toptobottom = True

        if (1 in cluster[1]) and (gridXDim in cluster[1]):
            lefttoright = True

    if prints==True:
        if toptobottom and not lefttoright:
            print("There is percolation from top to bottom")
        elif not toptobottom and lefttoright:
            print("There is percolation from left to right")
        elif toptobottom and lefttoright:
            print("There are both types of percolation")
        else:
            print("There is no percolation")

    if toptobottom and not lefttoright:
        return 'toptobottom'
    elif not toptobottom and lefttoright:
        return 'lefttoright'
    elif toptobottom and lefttoright:
        return 'both'
    else:
        return 0

def percolationGraph():
    # Define grids to percolate over
    gridDims = [[5, 5],[10,10],[15,15]]
    # number of tests for each probability
    testsNum = 1000

    # 1 over pStep for the number of probabilties tested
    pStep = 0.02

    # Array of all the probabilites that are tested
    pArray = np.arange(pStep, 1.0, pStep)

    for gridYDim,gridXDim in gridDims:
        # Define an array to hold the percolation ratio for each probability
        percolationRatio = []

        for p in pArray:
            percolationCount = 0
            for i in range(testsNum):
                result = percolation(False, gridXDim, gridYDim, p)

                if result=='upwards' or result=='lefttoright' or result=='both':
                    percolationCount += 1

            percolationRatio.append(percolationCount/testsNum)

            # Counter in real time so that the user understands if the code has
            # finished running
            print('\r{}x{}: {:.2f}'.format(gridXDim, gridYDim, p), end='')

        print()

        plt.plot(pArray, percolationRatio, label='{}x{}, N={}'.format(gridXDim, gridYDim, testsNum))

    plt.legend()
    plt.xlabel("p")
    plt.ylabel("Percolation ratio")

    plt.show()

# Call functions
percolation(True,5,5,0.5)
percolationGraph()
```

Figure 16: Part two of example code that produces a solution for question (a) and (b) on percolation theory



```python
import numpy as np
import matplotlib.pyplot as plt

def transfer():
    # Create a figure to display the temperature distribution
    fig, ax = plt.subplots()
    for k in range(0, iterations-1, 1):
        for i in range(1, ny-1, 1):
            for j in range(1, nx-1, 1):
                T[k + 1, i, j] = gamma * (T[k][i+1][j] + T[k][i-1][j] + T[k][i][j+1] + T[k][i][j-1] - 4*T[k][i][j]) + T[k][i][j]

    for i in range(0,iterations):
        #plt.imshow(T[i], cmap='hot', interpolation='nearest',vmin=0, vmax=100)
        plt.imshow(T[i], cmap='hot', interpolation='nearest')
        plt.colorbar()
        plt.show()

# Define the parameters
nx = 100  # Number of grid points in x direction
ny = 100  # Number of grid points in y direction
iterations = 100

alpha = 2
dx = 1

dt = (dx ** 2)/(4 * alpha)
gamma = (alpha * dt) / (dx ** 2)

# Initialize the temperature array
T = np.empty((iterations,ny, nx))
T.fill(0)

# Set the boundary conditions
T[0, 1:-1, 1] = 1000.0  # Left boundary
transfer()

# Initialize the temperature array
T = np.empty((iterations,ny, nx))
T.fill(0)

# Set the boundary conditions
T[0, 1, 1:-1] = 1000.0  # Top boundary
T[0, -2, 1:-1] = 1000.0  # Bottom boundary
T[0, 1:-1, 1] = 1000.0  # Left boundary
T[0, 1:-1, -2] = 1000.0  # Right boundary
transfer()
```

Figure 17: Example code that produces a solution for question (a) on thermal transfer



```python
#import math
import matplotlib.pyplot as plt
import numpy as np
import scipy.ndimage.morphology as morph

#Parameters
startTemp = 10
diam = 99  # Number of grid points
iterations = 200
alpha = 2 # Thermal diffusivity [m^2 s^-1]

dx = 1
dt = (dx ** 2)/(4 * alpha)
gamma = (alpha * dt) / (dx ** 2)

dT = 100 # Temperature added each time step

# Initialize the temperature array
T = np.empty((iterations, diam+1, diam+1))
T.fill(0)

vapourTemp = 800 #temperature of vapourisation [K]

#Create circle
xx, yy = np.mgrid[:diam+1, :diam+1]
circle = (xx - diam/2) ** 2 + (yy - diam/2) ** 2
T[:,:,:] = (startTemp)*(circle < ((diam/2)**2))

def findSurface(Tk):

    #circle is defined as any cell where the temp is greater than 0
    circleMaskK = Tk > 0
    inner = morph.binary_erosion(Tk)
    incidentMaskK = circleMaskK & ~inner

    return circleMaskK, incidentMaskK

circleMask, incidentMask = np.zeros_like(T), np.zeros_like(T, dtype=bool)
#Converting circle mask from boolean to binary to use in calculate(u) to
#prevent contribution from background pixels

circleMask[0], incidentMask[0] = findSurface(T[0])
circleMask[0] = 1*circleMask[0]

# Comment out the line below to turn off inital heat on the circumference
#T[0,incidentMask[0]] = T[0,incidentMask[0]] + 1000

def calculate(T):
    for k in range(0, iterations-1, 1):

        # Comment out the lines below to turn off adding heat each interation
        T[k,incidentMask[k]] = T[k,incidentMask[k]] + dT

        # Holds exposed pixels at fixed temperature
        #T[k, incidentMask[k]] = 300

        for i in range(1, diam, dx): #iterate through y
            for j in range(1, diam, dx): #iterate through x
                if (circleMask[k][i,j] == 1):
                    #Count number of adjacent cells that are part of the circle
                    adjacent = circleMask[k][i+1][j] + circleMask[k][i-1][j]
                    + circleMask[k][i][j-1] + circleMask[k][i][j+1]
                    T[k + 1, i, j] = gamma * (T[k][i+1][j]*circleMask[k][i+1][j]
                    + T[k][i-1][j]*circleMask[k][i-1][j] + T[k][i][j+1]*circleMask[k][i][j+1]
                    + T[k][i][j-1]*circleMask[k][i][j-1] - adjacent*T[k][i][j]*circleMask[k][i][j])
                    + T[k][i][j]*circleMask[k][i][j]

        # Comment out the 2 lines below to turn off ablation
        vapourised = T[k+1] > vapourTemp
        T[k+1:,vapourised] = 0
        circleMask[k+1], incidentMask[k+1] = findSurface(T[k+1])
        circleMask[k+1] = 1*circleMask[k+1]
    return T

# Do the calculation here
T = calculate(T)

#Plots every time step
for i in range(0,iterations):
    fig = plt.figure(dpi=300)
    #plt.imshow(T[i], cmap='hot', interpolation='nearest',vmin=273+startTemp-15, vmax=347)
    plt.imshow(T[i], cmap='hot', interpolation='nearest')
    plt.colorbar()
    plt.show()
```

Figure 18: Example code that produces a solution for question (b) and (c) on thermal transfer. This code is heavily based on Brook's, but edited to be applicable for the question asked, rather than birds.