Scientific Computing Exercise Set 3

Sander Broos (11895616) & Nick van Santen (11857846)

I. Introduction

MANY methods for solving particular partial differential equations like the diffusion equation involve discretising time by iterating over an approximated finite difference function while the algorithm comes closer to a stable solution. However, for some partial differential equations, a stable solution can be found by only discretising space and representing the equation as a matrix-vector problem.

This report investigates a method to solve the 2D wave equation of an elastic material which is fixed along its edges, like a vibrating membrane of a drum. This is done by assuming the solutions have separated space and time dependencies, and discretising space while keeping the time dependent part continuous. A matrix is constructed whose eigenvectors are the space dependent parts of solutions to the wave equation. This report calculates solutions for three different shapes of the membrane, and investigates the impact of the size of these shapes on the frequencies of the oscillation of the solutions.

The diffusion equation is another partial differential equation that is investigated in this report. Again, a matrix is constructed, which is used in an equation together with the initial state of the diffusion field. This equation can be solved to find the stable solution of the diffusion equation for a specific initial setting.

Finally, the leapfrog method is implemented, which computes the speed and position in a system separately at half time steps using equations for the force. This report looks at Hookes law, which gives the force of a spring in a one-dimensional harmonic oscillator. Additionally, a time dependent driving force is added to the system.

II. Theory

A finite difference equation is derived for the squared nabla operator. This equation is then transformed into matrix form. This matrix is used to iteratively simulate 2-dimensional waves inside a domain, which follow the 2-dimensional wave equation. The eigenfrequencies and eigenvectors are then equated for these waves.

This report will also look into a direct method to solve the 2-dimensional diffusion equation given an initial configuration of the field.

At last the leap frog method will be described, which solves equations of motions through a central scheme.

A. Representing ∇^2 as a matrix

Consider a 2D function $f(x, y)$ of which the value depends on the 2D coordinates. Many 2D partial differential equations involve the squared nabla operator which performs the operation

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}. \tag{1}$$

These second derivatives can be numerically approximated by considering the function on a rectangular domain of size $l_x \times l_y$. The x- and y-axes of the domain are divided into slices of equal length δx, so that the environment is represented by a lattice. The points on the lattice are indicated by (i, j) where $x = i\delta x$ and $y = j\delta x$. This makes it convenient to use the new notation

$$f_{i,j} = f(i\delta x, j\delta x). \tag{2}$$

The second derivative of a 1D function $g(x)$ can be approximated by evaluating g at different points, using the Taylor series expansions

$$g(x + h) = g(x) + g'(x) * h + \frac{g''(x)}{2}h^2 + \cdots, \tag{3}$$

and

$$g(x - h) = g(x) - g'(x) * h + \frac{g''(x)}{2}h^2 + \cdots. \tag{4}$$

These series can be added together to form

$$g(x + h) + g(x - h) = 2 * g(x) + g''(x)h^2 + \cdots, \tag{5}$$

which can be rewritten by leaving out the higher order terms to the approximation

$$g''(x) \approx \frac{g(x + h) - 2 * g(x) + g(x - h)}{h^2}. \tag{6}$$

This method is more accurate for smaller values of h. It can be applied to 2D functions like $f(x, y)$ as well, to approximate the partial derivative with respect to x or y. By taking $h = \delta x$, equation 1 can be approximated at a point (i, j) by evaluating f at the lattice sites above, below, left and right of it. The value becomes

$$\begin{aligned}
&\frac{\partial^2 f_{i,j}}{\partial x^2} + \frac{\partial^2 f_{i,j}}{\partial y^2} \\
&\quad\approx \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{\delta x^2} + \frac{f_{i,j+1} - 2f_{i,j} + f_{i,j-1}}{\delta x^2} \\
&\quad= \frac{f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1} - 4f_{i,j}}{\delta x^2}.
\end{aligned} \tag{7}$$

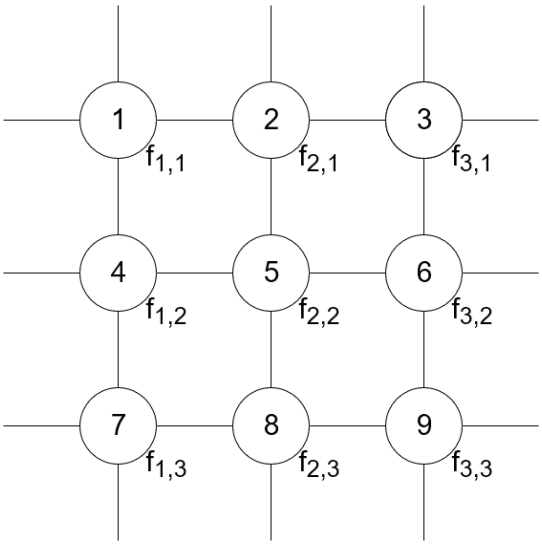


Fig. 1: A schematic drawing of the discretisation of a 3×3 environment, with the numbers in the circles showing at which indices the lattice sites are stored in the vector $\mathbf{v}$. Furthermore, the lines show for each lattice site which other elements are needed to calculate Equation 7. The lines on the outside are connected to the boundary of the environment, where the value of f is set to 0.

Since the function f is now stored as a 2D lattice, the result of the operation $\nabla^2 f$ should also result in a lattice of the same size where the value at lattice site (i, j) is Equation 7 applied to the point (i, j).

Because equation 7 uses only values from the lattice, it is possible to represent the ∇^2 operator on the lattice as a 2D matrix $\mathbf{M}$ applied on a 1D vector $\mathbf{v}$ which represents the values $f_{i,j}$. To store the 2D lattice in a 1D vector, the rows are linked after each other in succession. Consider a 3×3 lattice which is stored in $\mathbf{v}$ by linking the rows after each other, with as boundary conditions that all values f outside this domain are 0. This boundary condition is used for all environments that are used in this report. Figure 1 shows for each value $f_{i,j}$, at which index in $\mathbf{v}$ it is stored. Furthermore, the figure shows for each lattice site the connections that are needed to compute Equation 7 for that point. Some connections go outside the domain, so the boundary condition says that those values should be 0.

The vector $\mathbf{v}$ has a size of $3 \times 3 = 9$, so the matrix $\mathbf{M}$ has the size 9×9. A row in $\mathbf{M}$ then determines how the new value at a point is calculated using Equation 7. This means the row contains a 1 at the indices of the lattice sites above, below, left and right of the point, and a -4 at the index of the site itself. Finally all the values in the matrix are divided by δx^2. For the 3×3 lattice, this results in the matrix

$$\mathbf{M}_{3\times 3} = \begin{pmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{pmatrix}, \tag{8}$$

where the boundary condition means that the rows of the sites at the boundaries have fewer 1's than the other rows because the boundaries are 0 - the fifth row does have a 1 at four indices since that point is not at the boundary.

B. 2D wave equation

This report investigates solutions of the 2D wave equation. A wave solution in 2D can be thought of as a vibrating membrane or drum when this membrane is fixed along its edges. The 2D wave equation is given by

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u, \tag{9}$$

where $u(x, y, t)$ is the amplitude of the vibration as a function of space and time, c is a positive constant. A way to more easily find a valid solution is by looking for a solution where the space- and time-dependent parts are separated and multiplied, and the function u is written as

$$u(x, y, t) = v(x, y)T(t), \tag{10}$$

where v is the space dependent part and T is the time dependent part. In this way, the membrane retains its spatial "pattern" and only changes its total amplitude over time. Many solutions can not be written this way, for example if a ripple bounces through the environment. However, this separation of variables makes the wave equation easier to solve.

If Equation 10 is plugged into Equation 9, the first key to solving it is by moving all the time-dependent terms to one side of the equation and the space-dependent sides to the other, resulting in

$$\frac{1}{T(t)} \frac{\partial^2 T(t)}{\partial t^2} = c^2 \frac{1}{v(x, y)} \nabla^2 v(x, y), \tag{11}$$

where the left side only depends on t and the right side only depends on x and y. Thus, for these two sides to be

equal, they must both equal the same constant K. Setting the time-dependent part to K results in the equation

$$\frac{\partial^2 T(t)}{\partial t^2} = Kc^2 T(t). \tag{12}$$

This report investigates the cases where $K < 0$, for which this partial differential equation has the solutions

$$T(t) = A\cos(c\lambda t) + B\sin(c\lambda t), \tag{13}$$

where the frequency of the oscillation $\lambda = \sqrt{-K}$ and A and B can be any numbers to form a valid solution.

Setting the space-dependent part of Equation 11 equal to K as well results in the equation

$$\nabla^2 v(x, y) = Kv(x, y). \tag{14}$$

The function v can be discretised as shown in section II-A, with the matrix $\mathbf{M}$ being constructed to compute ∇^2. Now the equation can be written in the form

$$\mathbf{Mv} = K\mathbf{v}, \tag{15}$$

which is an eigenvalue problem where K become the eigenvalues and $\mathbf{v}$ the eigenvectors.

While section II-A only considered rectangular environments, another shape that is investigated in this paper is a circle of diameter L. To create the matrix $\mathbf{M}$ for such an environment, first a simple square environment with sides of length L is constructed as usual. Then, it should be checked what the indices are of the points that do not lie on the circle, since these should be kept at 0 according to the boundary condition. The rows that correspond to these indices are then filled with zeroes, except for the index of the point itself (at the diagonal). When the matrix $\mathbf{M}$ is then applied to the vector $\mathbf{v}$, the calculation at the row of a point outside the circle with index k becomes

$$0 * v_1 + \ldots + \frac{-4}{\delta x^2} * v_k + \ldots + 0 * v_N = K * v_k, \tag{16}$$

which is only possible if $v_k = 0$, which is the desired boundary condition.

Once $\mathbf{v}$ and K are found, the time-dependent equation can be recreated by multiplying the vector values with the continuous function $T(t)$.

C. Direct method

The 2-dimensional time dependent diffusion equation is given by

$$\frac{\partial c}{\partial t} = D\nabla^2 c, \tag{17}$$

where c is the concentration field. The steady-state solution of this concentration field can be found through a direct method. This is done by solving the matrix equation

$$\mathbf{Mc} = \mathbf{b}, \tag{18}$$

for c, where $\mathbf{M}$ is the diffusion matrix, $\mathbf{c}$ a column vector representing the concentration field and $\mathbf{b}$ a column vector representing the initial configuration of the field. Here, $\mathbf{M}$ not only depends on the shape of the environment but also on the initial configuration of the field.

D. Leapfrog method

The leapfrog method is a method used for integrating differential equations which have the form of

$$\frac{\partial x}{\partial t} = v \tag{19}$$

$$\frac{\partial v}{\partial t} = \frac{F(x)}{m}, \tag{20}$$

where x is the position, v the velocity, F the force, and m the mass.

Simple Euler integration uses a forward finite difference approximation to solve these equations. The leapfrog method, however, uses a central finite difference approximation, which increases the precision of the integration.

The leapfrog method implements its central scheme through calculating the updated position and velocity half a time-step apart. The updated positions are then calculated on steps n, $n+1$, $n+2$, ..., while the updated velocities are calculated on steps $n + 1/2$, $n + 3/2$, $n + 5/2$, ...

Equations 19 and 20 are then rewritten using the leapfrog method to

$$\frac{x_{n+1} - x_n}{\Delta t} = v_{n+1/2} \tag{21}$$

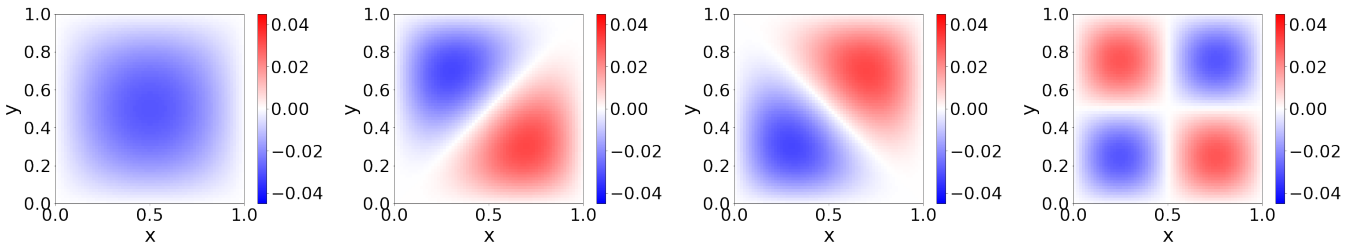
$$\frac{v_{n+3/2} - v_{n+1/2}}{\Delta t} = \frac{F(x_{n+1})}{m} \tag{22}$$

The updated positions and velocities are then calculated by rewriting these equations which give

$$x_{n+1} = x_n + v_{n+1/2}\Delta t \tag{23}$$

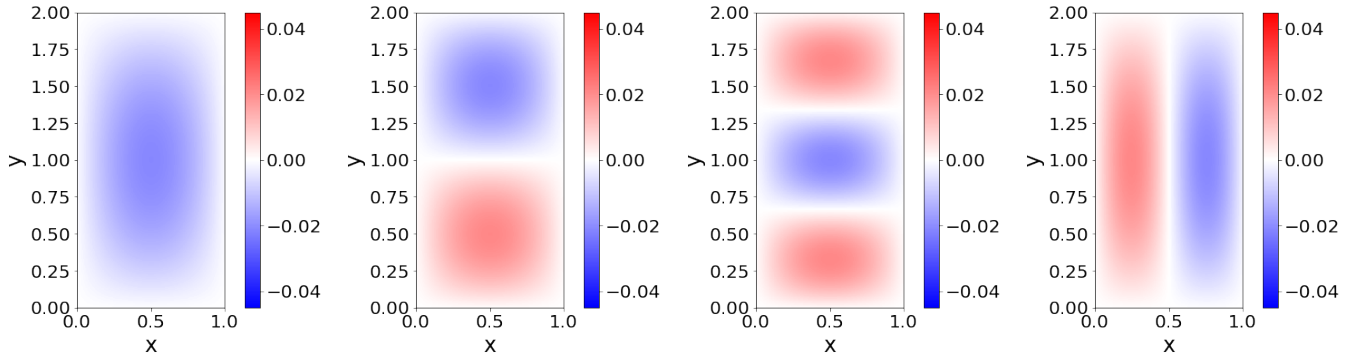
$$v_{n+3/2} = v_{n+1/2} + \frac{F(x_{n+1})}{m}\Delta t \tag{24}$$

The leapfrog method requires two initial values for the position x_{-1} and x_0. From these position values an initial velocity $v_{-1/2}$ is calculated by Equation 21. Then the updated positions and velocities are iteratively computed with equations 23 and 24 until a stopping condition is met or a maximum time is reached.



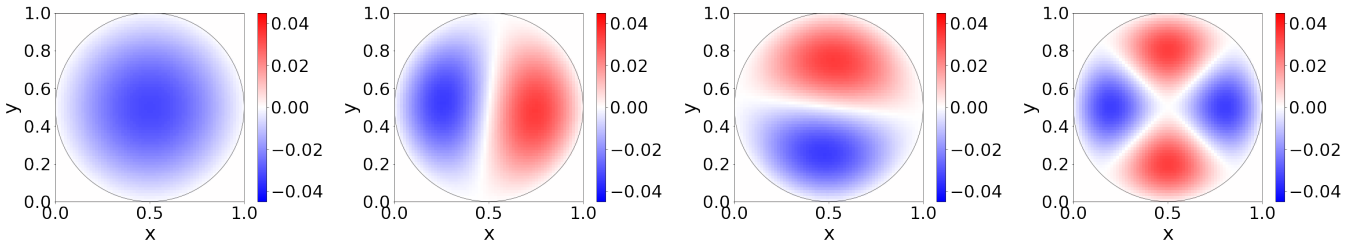
(a) $\lambda = 4.42037$ (b) $\lambda = 6.98769$ (c) $\lambda = 6.98769$ (d) $\lambda = 8.83832$

Fig. 2: The four eigenvectors $\mathbf{v}$ with the smallest eigenfrequencies λ for a square with side length 1 and $\delta x = 0.015$ using the non-sparse matrix function.



(a) $\lambda = 3.49466$ (b) $\lambda = 4.42037$ (c) $\lambda = 5.63445$ (d) $\lambda = 6.44211$

Fig. 3: The four eigenvectors $\mathbf{v}$ with the smallest eigenfrequencies λ for a rectangle with side lengths 1 and 2 and $\delta x = 0.015$ using the non-sparse matrix function.



(a) $\lambda = 4.79312$ (b) $\lambda = 7.63511$ (c) $\lambda = 7.63511$ (d) $\lambda = 10.21373$

Fig. 4: The four eigenvectors $\mathbf{v}$ with the smallest eigenfrequencies λ for a circle with diameter 1 and $\delta x = 0.015$ using the non-sparse matrix function. The edge of the circle is outlined.

III. Method

THE implementation for the computation of the eigenvalues and eigenvectors for several domains will be described in this section. Furthermore, the direct method will be described for a circular disk with a single source. Lastly, the leapfrog method is implemented for a spring which has two forces acting upon it: Hookes law and a sinusoidal driving force.

A. 2D wave equation

The model was implemented in Python. The eigenvalues K and eigenvectors $\mathbf{v}$ from Equation 15 are calculated using the `scipy.linalg.eig()` function. The eigenvalues and -vectors are then sorted by smallest eigenvalue.

Since each row in the matrix $\mathbf{M}$ contains at most 5 non-zero entries, $\mathbf{M}$ will contain mostly zeroes for very large shapes. Such a matrix is called a sparse matrix, and the eigenvectors and -values of such a matrix can be calculated using another function, `scipy.sparse.linalg.eigs()`, which should perform better for sparse matrices. This function can return the k eigenvalues with the largest real value. It can not return all the eigenvalues, like the `eig()`

function can. The largest real value corresponds to the smallest eigenfrequencies, since this report looks at eigenvalues $K < 0$.

Three different shapes are investigated in this report: a square with sides of length L, a rectangle with sides of length L and $2L$, and a circle of diameter L. The value L is then set differently for different investigations. This report uses $c = 1$.

B. Implementation direct method

The direct method is used to find a steady-state solution, where the domain is a circular disk with radius 2 centered on the origin. A single source is added at position (x, y).

The matrix M is initially constructed as described in Section II-A. The matrix is then updated at the point of the source. The position of the source is transformed into an index i, which represents the position in the column vector. Similarly to how the points outside the circle are fixed in the matrix, the ith row of M is then set to zeros, except for the diagonal, which is 1. This guarantees that value at the source point stays constant.

The column vector b consists of zeros, except at position i where it has a value of 1. Equation 18 is then solved for c by `scipy.linalg.solve`.[1]

C. Leapfrog method applied to a spring

The leapfrog method is studied by implementing a 1-dimensional spring. This spring has two forces acting upon it: Hookes law and a sinusoidal driving force.

Hookes law is given by

$$F_{hooke}(x) = -kx, \tag{25}$$

where k is a positive constant. The sinusoidal driving force is given by

$$F_{sine}(t) = A \sin \omega t \tag{26}$$

where A is the strength of the driving force and ω is the frequency of the frequency.

Thus the total force acting on the spring is

$$F(x, t) = F_{hooke} + F_{sine} = A \sin \omega t - kx. \tag{27}$$

Equations 23 and 24 are then used to evolve the spring over time.

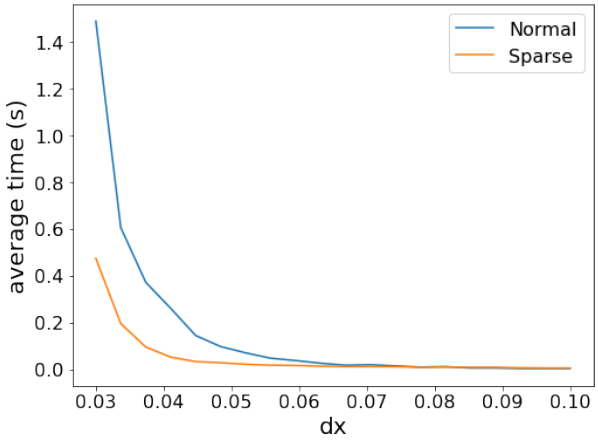


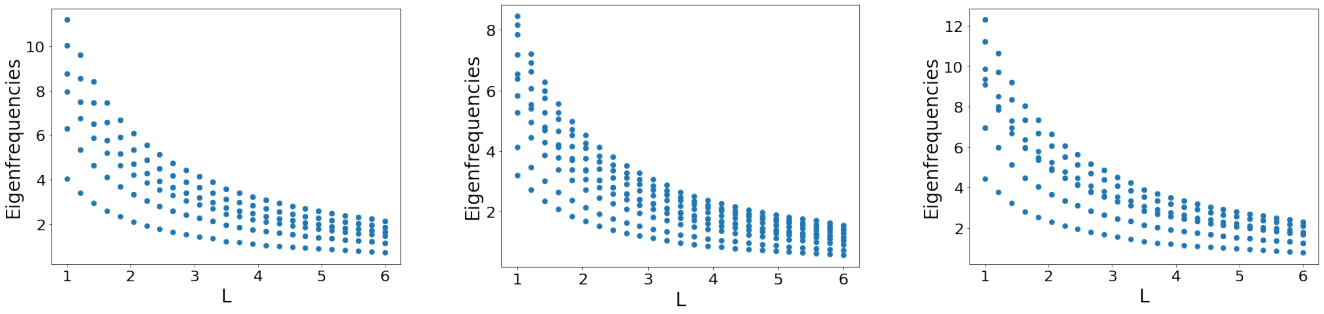
Fig. 5: The average duration over 50 runs of the sparse and normal function for obtaining the first 10 eigenvectors of a square with side length $L = 1$, for different values of δx.

IV. Results

THIS section will show the results of the computed eigenvalues and eigenvectors of a square, rectangle, and circular domain and will look into how the spectrum of eigenfrequencies depends on the size of the domains. Furthermore, it will compare the performance at which the eigenvalues and eigenvectors are computed for normal matrices and sparse matrices. Thereafter, the direct method for the diffusion equation is demonstrated for a circular disk with a single source and finally the leapfrog method is tested on a spring for which the impacts of k and ω are studied.

A. 2D wave equation

The eigenvectors were computed for different shapes using the parameter $L = 1$. Figure 2, 3 and 4 show the four eigenvectors with the smallest eigenvalues for the square, rectangle and circle shapes, respectively. Here, the vector $\mathbf{v}$ is re-mapped to a 2D grid by separating the rows of the lattice again. As shown by the colour bars, blue areas indicate a negative wave amplitude, while red areas indicate a positive wave amplitude. It looks like eigenvectors with higher frequencies generally have more ”divisions”, with more localised peaks. From the first few eigenvectors, it seems that any pattern that is not circularly symmetric will have two eigenvectors which are equal to each other but rotated $90°$. These will then both have the same eigenvalue. Something that stands out is that the first eigenvector of the square and the second eigenvector of the rectangle both have the same eigenfrequency λ. This is because the rectangle has the same pattern repeated side by side.



(a) Square with sides L (b) Rectangle with sides L and $2L$ (c) Circle with diameter L

Fig. 6: The spectrum of the first ten eigenfrequencies for different values of L and $\delta x = 0.1$ for three shapes using the sparse matrix function.

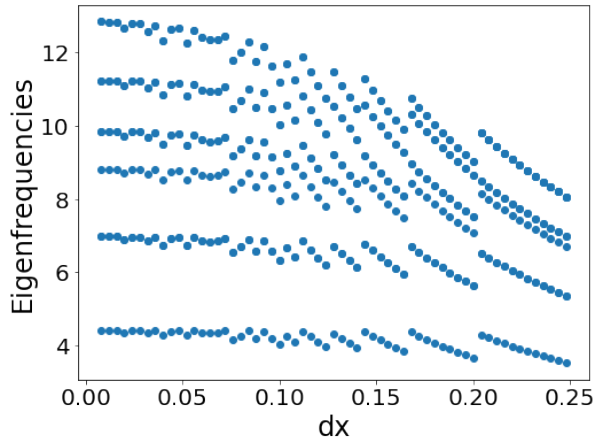


Fig. 7: The spectrum of the first ten eigenfrequencies for different values of δx for a square with side length $L = 1$ using the sparse matrix function.

The advantage of using the sparse matrix function was tested by measuring the average durations of both the sparse and normal function for obtaining the first 10 eigenvectors. This was done for different values of δx. The sparse solving function is expected to perform better for low values of δx, because then the matrix $\mathbf{M}$ has a higher proportion of zeroes and is thus "sparser". These results are shown in Figure 5. Indeed, the sparse function has a notable advantage over the regular function for low values of δ, although the speeds of the two functions become much closer for $\delta x > 0.08$.

Figure 6 shows how the spectrum of eigenfrequencies depends on the size L for the three shapes. Here, the first 10 eigenfrequencies are plotted at different values of L. For all three shapes, the spectrum of eigenfrequencies gets lower values as L increases. Thus, the time dependent solution will have slower oscillations for larger shapes. Figure 7 shows how the spectrum depends on δx for a square with $L = 1$. For low values of δx, the spectrum shows only subtle variations. However, when δx becomes too large, the spectrum starts to deviate more significantly. Thus, it is desirable to choose δx as low as possible. However, note that when δx is too low, the matrix $\mathbf{M}$ becomes too large to analyse by a standard computer.

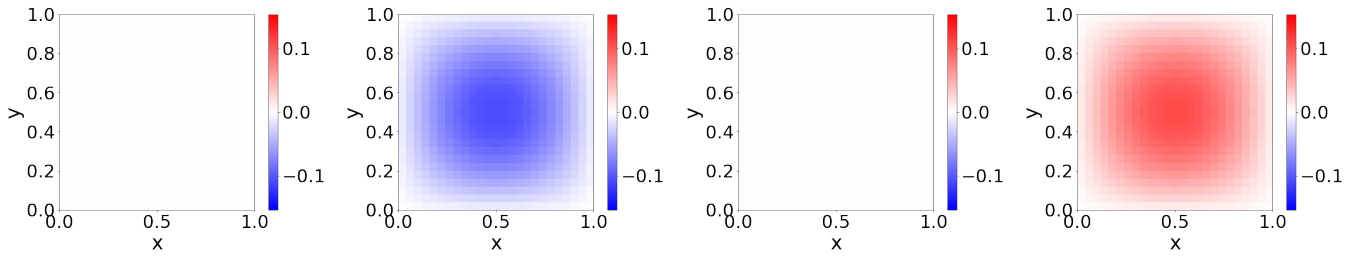
The time dependent solutions were then created according to Equation 10 by multiplying the vector $\mathbf{v}$ by $T(t)$ and using the obtained eigenfrequency as λ. This report takes $A = B = 1$. To show the behaviour in time, some solutions are plotted at different time points. Figure 8 shows the time dependence for the square, Figure 9 for the rectangle and Figure 10 for the circle, where for all shapes $L = 1$. The chosen time points here are $0.75 * \frac{\pi}{\lambda}$, $1.25 * \frac{\pi}{\lambda}$, $1.75 * \frac{\pi}{\lambda}$ and $2.25 * \frac{\pi}{\lambda}$, since those are the points where $T(t)$ has its peaks and zero intersections during a single full oscillation. All figures show how the amplitude of all points go from zero to a higher absolute amplitude, either positive or negative, and then go back through zero to the inverse pattern with the signs flipped (the red and blue areas switch colours). Then, the values of the points go back to zero again and repeat the same oscillation. All solutions show this same behaviour, since those are the solutions that result from the separation of the space- and time-dependent parts.

B. *Direct method*

A steady-state solution was computed with the direct method and is shown Figure 11. It had a single source with a position of (0.6, 1.2). The spatial resolution was set to $\delta x = 0.05$.

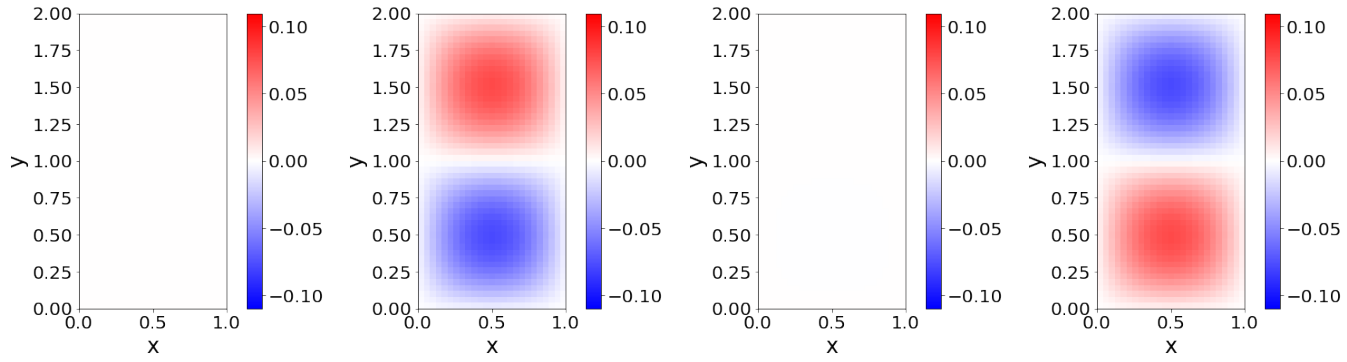
C. *Leapfrog method applied to a spring*

The impact of k on the spring was studied. This was performed for a spring with parameters $m = 1$, which uses time steps of $\Delta t = 0.01$ with force parameters $A =$



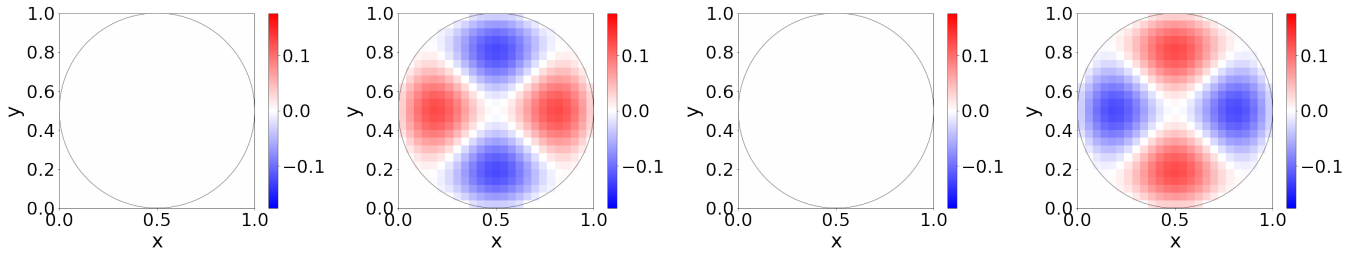
(a) t = 0.55 (b) t = 0.92 (c) t = 1.29 (d) t = 1.66

Fig. 8: The first time dependent solution with the eigenfrequency $\lambda = 4.42037$ for a square with side length $L = 1$ and $\delta x = 0.04$ at four different time points using the sparse matrix function.



(a) t = 0.55 (b) t = 0.91 (c) t = 1.28 (d) t = 1.64

Fig. 9: The second time dependent solution with the eigenfrequency $\lambda = 4.42037$ for a rectangle with side lengths 1 and 2 and $\delta x = 0.04$ at four different time points using the sparse matrix function.



(a) t = 0.32 (b) t = 0.53 (c) t = 0.74 (d) t = 0.95

Fig. 10: The fourth time dependent solution with the eigenfrequency $\lambda = 4.42037$ for a circle with diameter $L = 1$ and $\delta x = 0.04$ at four different time points using the sparse matrix function.

0. The initial positions of the spring are $x_{-1} = 0.49$ and $x_0 = 0.5$. Figure 12 shows how the position and velocity change over time for different values of k. It can be seen that the frequency of the spring increases for larger k values. This is caused by a stronger Hooke's force as k increases, which pulls the spring back with more force which increases its frequency.

The impact of the frequency of the sinusoidal driving force is also studied. This is performed for a spring with parameters $m = 1$, which uses timesteps of 0.01 with force parameters $k = 1$ and $A = 1$. The initial positions of the spring are $x_{-1} = 0.49$ and $x_0 = 0.5$. Figure 13 shows phase plots of the position and velocity for different values of ω. When ω approaches the natural frequency of the spring, resonance starts to occur, which increases the speed of the spring.

V. Discussion & Conclusion

THE results show that the separation of variables technique indeed makes it possible to find solutions to the 2D wave equation by solving an eigenvalue problem. The eigenfrequencies of the solutions decrease

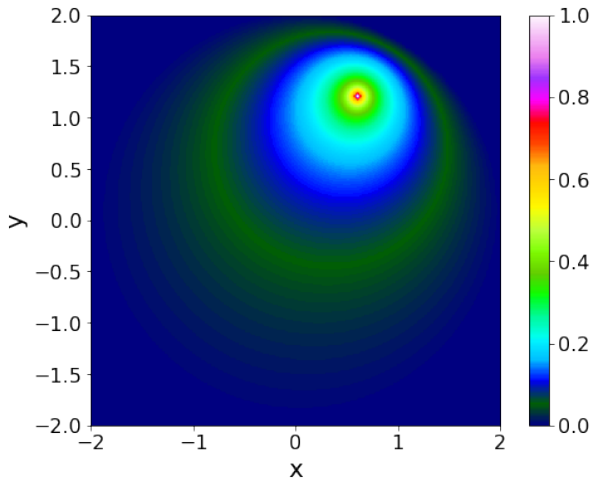


Fig. 11: A steady state solution computed with the direct method. It has a single source with a position of (0.6, 1.2). The spatial resolution was set to $\delta x = 0.05$. The color-map shows the concentration at each position.

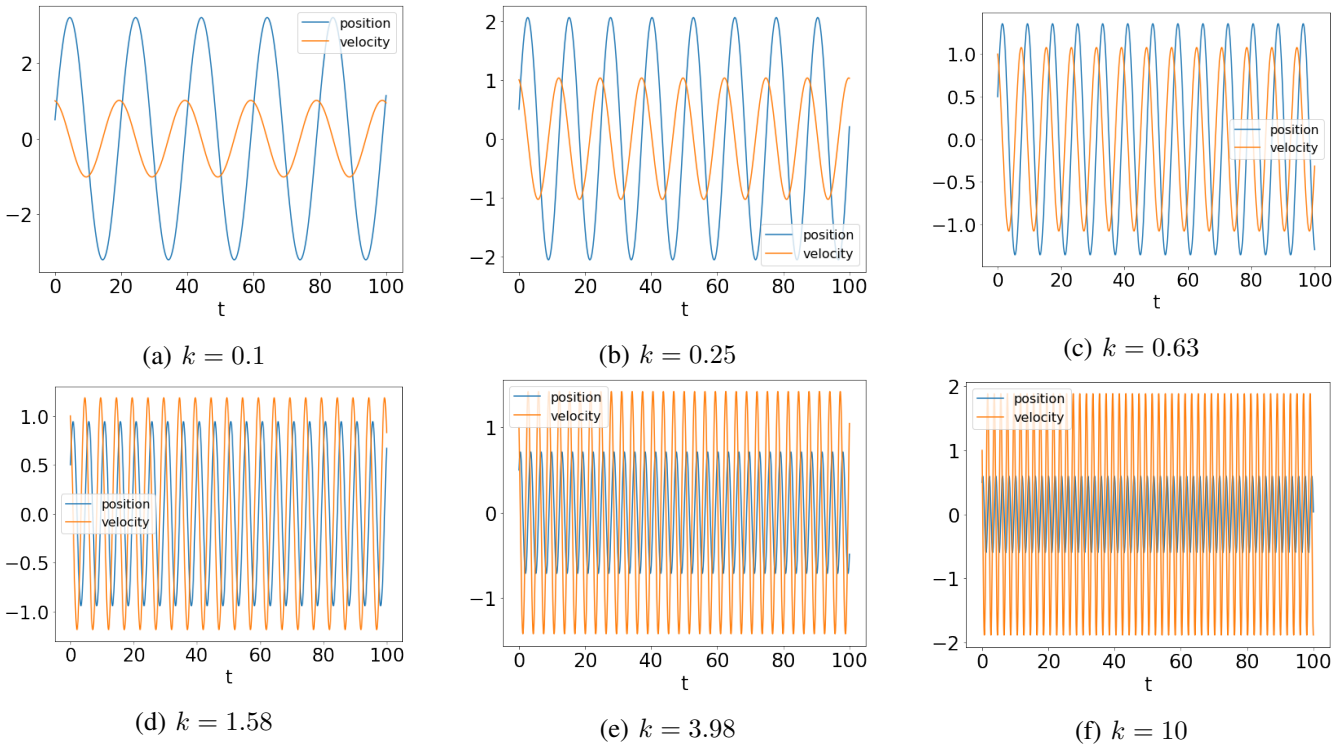
when the size of the shape L increases. Future research could investigate whether this also holds for other solutions to the 2D wave equation that do not satisfy Equation 10. The resulting eigenfrequencies seem to be more accurate for lower δx, although that results in very large matrices $\mathbf{M}$ which are hard to analyse using standard computers. In future research, the problem could be parallelised to allow for much larger computations.

The results showed that sparse matrices had a significant performance boost when computing the eigenvalues and eigenvectors. The same principle can be used to solve Equation 18, where instead of the general `scipy.linalg.solve` method, a specific method designed for solving sparse matrices is used. A possible method would be `scipy.sparse.linalg.spsolve`, which could provide a significant performance boost over the current implementation.

The results of the spring showed that the leapfrog method is a valid method to simulate a spring, which is acted upon by Hooke's law and a sinusoidal driving force. It was seen that the frequency of the spring increases as k increases. Furthermore, resonance starts to occur when the driving force starts to match with the natural spring frequency. The maximum velocity of the spring then starts to increase.

REFERENCES

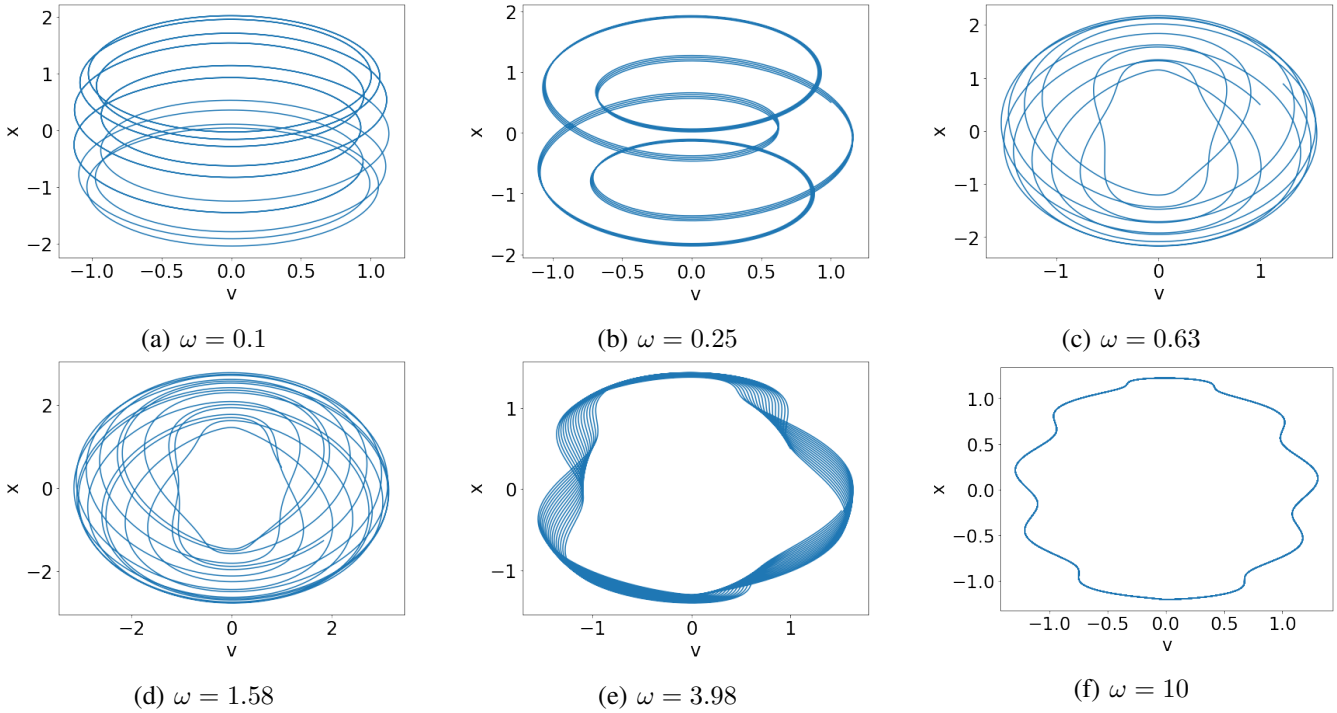
[1] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.



(a) $k = 0.1$ (b) $k = 0.25$ (c) $k = 0.63$

(d) $k = 1.58$ (e) $k = 3.98$ (f) $k = 10$

Fig. 12: These figures show the position and velocity of a spring computed with the leapfrog method. This is performed for different values of k with spring parameter $m = 1$ and timesteps of $\Delta t = 0.01$ with force parameters $A = 0$. The initial positions of the spring are $x_{-1} = 0.49$ and $x_0 = 0.5$.



(a) $\omega = 0.1$ (b) $\omega = 0.25$ (c) $\omega = 0.63$

(d) $\omega = 1.58$ (e) $\omega = 3.98$ (f) $\omega = 10$

Fig. 13: These figures show a phase plot of the position and velocity of a spring computed with the leapfrog method. This is performed for different values of ω with spring parameter $m = 1$ and timesteps of $\Delta t = 0.01$ with force parameters $k = 1$ and $A = 1$. The initial positions of the spring are $x_{-1} = 0.49$ and $x_0 = 0.5$. The spring is iterated for 10000 timesteps.