3 Assignment Set 3

3.1 Eigenmodes of drums or membranes of different shapes

In this exercise we solve the wave equation on a two-dimensional elastic material. We use boundary conditions that fix the membrane along its edge, and solve for different shapes of the membrane. This time, we will not explicitly solve the time development as in exercise 1, instead we are interested in the *eigenmodes* and the *eigenfrequencies* or resonance frequencies of the system.

A famous question related to this problem is: *Can you hear the shape of the drum*?, or more specifically, whether the membrane shape can be identified uniquely from its eigenvalue spectrum. The answer recently turned out to be no in general, but we'll still try to hear the difference between some simple membranes.

Consider the wave equation in 2 dimensions.

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \tag{19}$$

We look for a solution in the form

$$u(x, y, t) = v(x, y)T(t) \tag{20}$$

where the time and space dependencies can be separated in two independent functions. Not every $u(x, y, t)$ can be separated this way, rather we state that we are specifically looking for a u of this form due to the nice properties. Inserting this in the wave equation and moving all t-dependencies to the left and all x, y dependencies to the right, we get

$$\frac{1}{T(t)} \frac{\partial^2 T(t)}{\partial t^2} = c^2 \frac{1}{v(x, y)} \nabla^2 v(x, y) \tag{21}$$

The left hand side depends only on t, the right hand side only on x and y. Thus both sides must equal some constant K independent of x, y, and t. Now the left hand side and the right hand side can be treated independently. We'll start with the left hand side:

$$\frac{\partial^2 T(t)}{\partial t^2} = Kc^2 T(t). \tag{22}$$

If $K < 0$, it has an oscillating solution

$$T(t) = A\cos(c\lambda t) + B\sin(c\lambda t), \tag{23}$$

where $\lambda^2 = -K$. $\lambda > 0$ can be assumed without loss of generality. (If $K > 0$, the solution either grows or decays exponentially, and this case is not interesting at the moment. If $K = 0$, the solution is a constant or a linear function of x and y, and has to be $v = 0$ due to the boundary conditions.)

The right-hand side is

$$\nabla^2 v(x, y) = Kv(x, y), \tag{24}$$

an eigenvalue problem. The solutions to this problem give the eigenmodes v and eigenfrequencies λ. A way to find v numerically is to discretize Eq. (19), to obtain a set of linear equations, one for each discretization point. If this system of equations is written in the form $\boldsymbol{M}\boldsymbol{v} = K\boldsymbol{v}$, where $\boldsymbol{M}$ is a matrix, $\boldsymbol{v}$ a vector of $v(x, y)$ at the grid points and K is a scalar constant. This matrix eigenvalue problem can be solved efficiently with library methods e.g. in Python. Boundary conditions: $v(x, y) = 0$ on the boundary, i.e. the boundary is fixed. Take $c = 1$.

A. *(1 point)* Discretize eq. (24). Formulate a matrix version of the eigenvalue problem, taking the boundary conditions into account. Hint: draw a small example, number the elements and think about which elements are connected and which are not. Draw a figure showing the discretization points and their positions. Show and explain the shape of the matrix for a very small problem, maybe a 4x4 system.

B. *(3 points)* Consider the following three shapes.

1. a square with side length L
2. a rectangle with sides L and $2L$
3. a circle with diameter L

Solve the eigenvalue problem. Try `scipy.linalg.eig()`, `eigh()` or `eigs()`, (or something else). Which did you use, and why? Plot the eigenvectors v for some of the smallest eigenvalues, for $L = 1$. Label the plots with their frequencies.

C. *(0.5 point)* For higher performance you can try *sparse* matrices: `scipy.sparse.linalg.eigs()`. Show the difference in speed.

D. *(1 point)* How does the spectrum of eigenfrequencies depend on the size L? Plot the eigenfrequencies for each shape as a function of L. Do the frequencies depend on the number of discretization steps?

E. *(1 point)* Use eq. 20 and the solutions to eq. 24 to construct time-dependent solutions. Show how the first few eigenmodes behave in time. Try to make an animated plot of some eigenmodes for one of the three systems.

Hints for the circular domain: Grid points within the distance $R = L/2$ from the center belong to the domain. Points further away do not, and for all those $v_{ij} = 0$. There are two options for implementing these irregular boundary conditions:

1) The vector $\boldsymbol{v}$ contains only points inside the domain. You need to construct an indexing scheme mapping index in $\boldsymbol{v}$ to a point on the grid, and construct the matrix so that all connections and boundary conditions are satisfied.

2) The vector $\boldsymbol{v}$ contains points in a rectangular grid. Some of them do not belong to the domain, and for those, $v_k = 0$. This can be written in the matrix form by filling the k-th row of the matrix with 0:s. In the matrix eigenvalue equation $\boldsymbol{M}\boldsymbol{v} = K\boldsymbol{v}$, the k-th row will then be $0 = Kv_k$, forcing $v_k = 0$ (or $K = 0$, and these eigenvalues we can ignore).

3.2 Direct methods for solving steady state problems

In this exercise we will again find the steady state of the diffusion equation. In the previous exercises it was done using *iterative* methods. Now direct methods will be used, where a matrix for the ∇^2 is constructed, a matrix equation of the type $\boldsymbol{M}\boldsymbol{c} = \boldsymbol{b}$ is formed, and solved with standard library methods.

The domain is a circular disk with radius 2, centered on the origin. On and beyond the boundary of the disk, the concentration $c = 0$. At the point $(0.6, 1.2)$ a source is present, so that the concentration there is 1.

G. *(3 points)* Find the steady state concentration $c(x, y)$ by discretizing the diffusion equation as mentioned above. Plot the solution.

H. *(1 point)* Explain carefully how the matrix $\boldsymbol{M}$ and the vector $\boldsymbol{b}$ are constructed, and how the boundary conditions are taken into account.

Note that the hints about the circular domain from the preceding exercise still apply, and that the matrix M there might be quite similar to the one needed here.

For solving the matrix equation, you can use `scipy.linalg.solve()`. For larger systems and very little additional effort, sparse matrix operations can be used instead, `scipy.sparse.linalg.spsolve()`. For even larger systems, iterative sparse matrix solvers may be needed.

See Heath for a discussion about direct vs iterative methods.

3.3 The leapfrog method - efficient time integration

In the previous exercises we performed the time-stepping through a forward finite-difference which is a first-order method. This means that the truncation error is directly proportional to the stepsize. Better convergence can be obtained by using higher order methods such as the central difference we used for space discretization, which is second order. This becomes difficult in the case of time-stepping however, as the forward step is not known in advance. Implicit methods are required for which a system of equations has to be solved at each step, making the method more complex and expensive. Depending on the system being analysed it is possible to obtain higher order convergence without the corresponding increase in cost, by making smart use of the given differential equations.

The leapfrog method is one of these "smart" algorithms used for integrating the equations of motion. Commonly used for interacting systems with many particles such as in molecular dynamics or astronomical N-body simulations. From Newtons second law we know that force is directly proportional to the mass times acceleration of an object, acceleration being the first and second derivative of velocity and position respectively. For a system of N particles we thus obtain the following set of differential equations:

$$\frac{\partial x_i}{\partial t} = v_i \tag{25}$$

$$\frac{\partial v_i}{\partial t} = \frac{F_i(\{x\})}{m_i} = -\frac{\partial U(\{x\})}{\partial x_i} \tag{26}$$

With x_i, v_i, m_i indicating the position, velocity and mass of the i^{th} particle respectively $(i = 1, 2, ..., N)$. F and U are the force and potential which depend on $\{x\}$, the ensemble over all particles $\{x_1, x_2, ..., x_N\}$ Notable is that the two equations do not depend on themselves i.e. the positions depend only on the velocity and vice versa. This means that we can perform the time-stepping of the equations independent of each other. A smart choice would be to calculate our position and velocity half a time-step apart: $(x_n, x_{n+1}, x_{n+2}, ...)$ and $(v_{n+1/2}, v_{n+3/2}, v_{n+5/2}, ...)$. By using these positions or velocity for the next iteration we effectively perform a central finite difference approximation whilst using the same steps of the forward method.

$$\frac{x_{n+1} - x_n}{\Delta t} = v_{n+1/2} \tag{27}$$

$$\frac{v_{n+3/2} - v_{n+1/2}}{\Delta t} = \frac{F(x_{n+1})}{m} \tag{28}$$

I. *(bonus, 2 points)* Implement the leapfrog method for a simple one dimensional harmonic oscillator where the force is a function following Hookes law: $F(x) = -kx$ with k being a positive (spring) constant. Clearly explain how the alternating timesteps are discretized and how the initial velocity at the half step is calculated, does this affect the overall accuracy of the method? Take $m = 1$ for the mass of the object and plot the position and velocity for a few values of k.

Extra bonus *The leapfrog method is also a symplectic integrator which means that energy is conserved over long simulation times, demonstrate this by comparing it with a non-stable but higher order integrator e.g. RK45*

J. *(bonus, 2 points)* Add an external time-dependent sinusoidal driving force to the oscillator, separate from the from the restoring force giving the following velocity term:

$$\frac{dv}{dt} = \frac{F(t) - kx}{m} \tag{29}$$

What happens when the driving force is close to the original frequency of the oscillator? Show a phase plot (v, x) of various frequencies.

4 Appendix. Tool recommendations

Python
Always use Python ≥ 3.9.

The suggested distribution is https://anaconda.org/.
Note: don't overwrite your system's python (which might be older), install a new distribution next to it.

Python Modules - required
matplotlib
numpy
scipy
json

Python Modules - optional
numba
taichi
cupy

You can also take a look at Google Colab https://colab.research.google.com/, for instance for GPU programming.

Scientific code development practices Aim to follow good practices: write structured, well-documented code, apply unit test (e.g., look at https://docs.python.org/3/library/doctest.html), even if you use a notebook develop modular code in multiple files that are imported in the notebook.

For more details on good scientific coding practices please see the following workshop materials: https://github.com/JuliaDynamics/GoodScientificCodeWorkshop.

Code editor
There is a wealth of options, the default suggestion is https://code.visualstudio.com/.

AI tools
At all times the official guidelines of the university are in effect. Also see discussion during the intro lecture.

Version control
Use git, preferably https://github.com/. You UvA ID will give you a 'pro' account.

Drawing figure
Beyond matplotlib you might want to use TikZ (latex) or Inscape.

Other useful references
An introduction to Numpy and Scipy:
https://sites.engineering.ucsb.edu/~shell/che210d/numpy.pdf

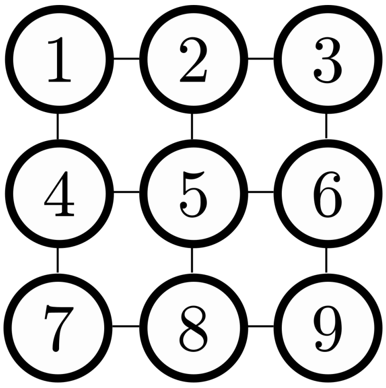
Python Scientific Lecture Notes
https://lectures.scientific-python.org/

Scipy getting started
https://projects.scipy.org/getting-started.html

Matplotlib documents
https://matplotlib.org/stable/

5 Appendix. Matrices

Example of a matrix encoding the connections of a 3 by 3 grid. The boundary conditions are “reflective” or “no-flow”, meaning that there is no diffusion over the edges of the system. See Heath (optional book), Chapter 11.



$$\begin{pmatrix} -2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -3 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -3 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \\ c_9 \end{pmatrix} \tag{30}$$