

Assignments for Scientific Computing

General guidelines

You are requested to write three concise reports of all the practical work you do for this course. The reports should contain a clear introduction stating the problem at hand, a theory section that introduces necessary theory, a methods section that describes the methods used, e.g. pseudo code of algorithms, a results section that presents the main results that you obtained, figures of the simulated results, and finally a short discussion that critically goes through the results in view of the original problem, e.g. did you solve the problem in a satisfactory and complete way. It is important that you also include information about the software that you create. That is, you also provide text that explains how you implemented the simulations. *Any reader of your report should be able to reproduce your results.* Use this as a guideline to decide the amount of detail that will go into your report.

Please note that a lab assistants are available during the lab sessions. If you have any questions regarding the practical work you should ask them for advice. We strongly advise you, if you have any questions, to attend the lab sessions.

You are asked to write the report in *groups of three* people. Use Canvas to set up groups, even if you work by yourself. Your report should be 6 pages maximum including figures, references, and appendices if applicable. Half a point per extra page is subtracted from the assignment mark. *Note:* You can use the assignment document as a valid reference, and refer to figures and equations in it to avoid text duplication.

For the programming language we strongly suggest Python or C. Other languages are possible on your own responsibility, the TAs might not be able to provide help with them.

Every set is graded and the combined result counts for 50% towards your final grade (≥ 5.0 is required). The other half of the grade is determined by the result of the exam. The lab reports are to be submitted through Canvas, and should be in the PDF format. Together with your lab report you must send (in a zip archive) all your source files (both a URL to the repository and a copy of the repository) and possibly graphs and or movies not presented in the report. We should be able to run your programs (and this will be tested).

You will be graded on the structure and content of your report. There is also the possibility to obtain bonus points through some optional exercises.

Assignment sets

Note: the deadlines will be at 8pm! For more details see the first introductory slides.

Set 1: Vibrating string, Time dependent diffusion, Jacobi iteration, Gauss-Seidel iteration, Successive over relaxation.

Deadline: End of 3rd week.

Set 2: Diffusion-limited aggregation, Random walk, Reaction-Diffusion System.

Deadline: End of 5th week.

Set 3: Eigenmodes of a circular drum, Direct methods.

Deadline: End of 7th week.

1 Assignment Set 1

1.1 Vibrating string

The problem at hand is the numerical solution of a one-dimensional wave equation, which might describe the vibrations of a uniform string, the transport of voltage and current along a lossless transmission line, the pressure and flow rate of a compressible liquid or gas in a pipe, sound waves in gases or liquids, and optical waves. In this example, the domain of the wave equation is a string.

The one-dimensional wave equation is

$$\frac{\partial^2 \Psi}{\partial t^2} = c^2 \frac{\partial^2 \Psi}{\partial x^2}$$

The solution $\Psi(x, t)$ is the vibration amplitude expressed as a function of position and time. The problem becomes fully posed with the addition of boundary conditions on the spatial domain, and initial position and velocity distributions. We attempt a numerical solution method by introducing a uniform discretization of the spatial and temporal domains, and approximating the partial differential equation by finite difference expressions. The grid points of this problem consist of discrete nodal points at which Ψ is to be evaluated.

A. (0.5 point) Discretize the wave equation, and write it in a form suitable for implementing in a computer program. Assume that the boundaries are fixed, $\Psi(x = 0, t) = 0$, $\Psi(x = L, t) = 0$. L is the length of the string. Take $L = 1$ for simplicity. Divide the string in N intervals, so that the interval length is $\Delta x = L/N$. Also consider the boundary cases.

If you use Euler's method, you need to use both $\Psi(x, t)$ and $\Psi'(x, t)$ as variables. Or use the stepping method from the lectures, which uses Ψ at the two most recent time points to calculate it at the next one.

B. (1 point) Implement the time stepping. Determine the time development of the string, with the following initial conditions. The string is at rest at $t = 0$, i.e. $\Psi'(x, t = 0) = 0$.

- i. $\Psi(x, t = 0) = \sin(2\pi x)$.
- ii. $\Psi(x, t = 0) = \sin(5\pi x)$.
- iii. $\Psi(x, t = 0) = \sin(5\pi x)$ if $1/5 < x < 2/5$, else $\Psi = 0$.

Take $c = 1$ and use the time step $\Delta t = 0.001$. Plot the result at several times in the same figure, e.g. varying the color of the curve.

C. (1 point) Make an animated plot of the time development. This can be done from within matplotlib, see the following reference:

<https://matplotlib.org/stable/users/explain/animations/animations.html>

With this technique, you can show the animation from within the Python program, or save it to a file in various video formats to use later, e.g. in presentations.

You can also use matplotlib to save individual images, e.g. in the .png format, and then pack the images into an animation using `ffmpeg` or `avconv`.

1.2 The Time Dependent Diffusion Equation

Consider the two-dimensional time dependent diffusion equation

$$\frac{\partial c}{\partial t} = D \nabla^2 c \tag{1}$$

here $c(x, y, t)$ is the concentration as a function of the coordinates x and y and time t , and D is the diffusion constant. In all the assignments we will consider a square

domain, and without loss of generality we assume that $0 \leq x, y \leq 1$. Furthermore, we always assume the following boundary conditions:

$$c(x, y = 1; t) = 1 \text{ and } c(x, y = 0; t) = 0 \quad (2)$$

So, on the top of the domain the concentration is always equal to one, and on the bottom of the domain the concentration is always equal to zero. Furthermore, in the x -direction we will always assume periodic boundary conditions:

$$c(x = 0, y; t) = c(x = 1, y; t) \quad (3)$$

These boundary conditions are once more drawn in fig. 1.

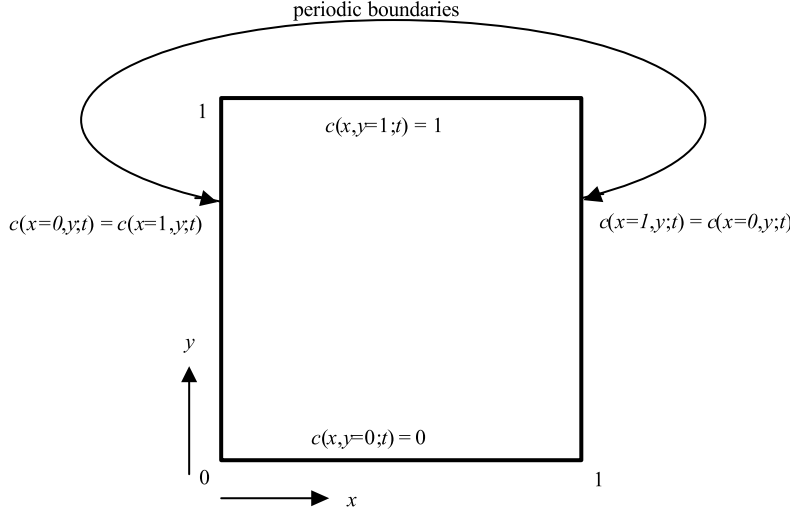


Figure 1: The computational domain and boundary conditions.

As an initial condition we take

$$c(x, y; t = 0) = 0 \text{ for } 0 \leq x \leq 1, 0 \leq y < 1. \quad (4)$$

Note that for this specific set of initial and boundary conditions, the solution depends only on the y -coordinate (because of symmetry) and on time. This is a very useful feature to test the correctness of your code. Furthermore, for this specific set of initial and boundary conditions, the diffusion equation can also be solved analytically. The analytical solutions, assuming $D = 1$, as a function of the y -coordinate are shown in fig. 2 at a number of time points.

Note that for $t \rightarrow \infty$ the concentration profile is simply a straight line,

$$\lim_{t \rightarrow \infty} c(y, t) = y. \quad (5)$$

You can easily derive this yourself from the time-independent diffusion equation (try it). This is also a powerful check of the correctness of your simulation. For finite times you can also compare the simulation results with the exact solution.

Next the spatial and temporal domain are discretized. The x - and y -axes are divided in intervals with length δx . Assuming that we have N interval in the x - and y -directions, we have $\delta x = 1/N$, and $x = i\delta x$, $y = j\delta x$ where $i, j \in (0, 1, 2, \dots, N)$. Furthermore, we assume a small time interval δt , so that $t = k\delta t$ where $k \in \mathbb{N}$.

We now want to find solution to the concentration at these discretized space and time points. With the definition

$$c(i\delta x, j\delta x; k\delta t) \equiv c_{i,j}^k \quad (6)$$

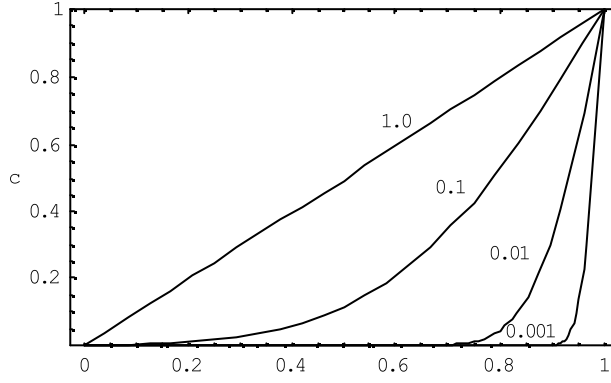


Figure 2: The analytical solution of the concentration, as a function of the y -coordinate, for t equal to 0.001, 0.01, 0.1, and 1 respectively. Here, $D = 1$ and the initial and boundary conditions are as in the text.

and discretizing the diffusion equation using standard finite difference methods the following explicit scheme is easily derived.

$$c_{i,j}^{k+1} = c_{i,j}^k + \frac{\delta t D}{\delta x^2} (c_{i+1,j}^k + c_{i-1,j}^k + c_{i,j+1}^k + c_{i,j-1}^k - 4c_{i,j}^k) \quad (7)$$

This scheme is stable if

$$\frac{4\delta t D}{\delta x^2} \leq 1 \quad (8)$$

This determines the maximum time step δt that you can take. In this assignment you implement a finite difference simulation to solve the time-dependent diffusion equation, using the finite difference scheme in eq. (7) and subject to the boundary and initial conditions in eqs. (2) to (4).

Due to the five-point *stencil* every grid point only needs local information to update for the next time step. Therefore, this calculation scheme is also suitable for parallel computations.

D. (0.5 point) Determine the equation to use at the boundaries of the domain. Clearly show the ranges of the indices of the grid. A figure is extremely helpful for figuring this out.

Write a program for the simulation of the two-dimensional time dependent diffusion equation discretized using the explicit finite difference formulation from eq. (7). You may want to write your data to a file (e.g. after every iteration, or maybe after every 100 iterations) so that you can analyze the data later on or plot them immediately.

E. (1 point) Test the correctness of your simulation. Compare to the analytic solutions, plot $c(y)$ for different times. The analytic solution is

$$c(x, t) = \sum_{i=0}^{\infty} \operatorname{erfc} \left(\frac{1-x+2i}{2\sqrt{Dt}} \right) - \operatorname{erfc} \left(\frac{1+x+2i}{2\sqrt{Dt}} \right). \quad (9)$$

F. (1 point) Plot the results, show the 2D domain, with a color representing the concentration at each point. Make a plot of the state of the system at several times: $t = \{0, 0.001, 0.01, 0.1, \text{ and } 1\}$.

G. (1 point) Make an animated plot of the time dependent diffusion equation until equilibrium.

1.3 The Time Independent Diffusion Equation

Now assume that we are not very much interested in the time development of the concentration profile (i.e. the transient behavior), but only in the steady state. In

that case it is possibly more effective to directly solve the time independent diffusion equation:

$$\nabla^2 c = 0. \quad (10)$$

This is the famous Laplace equation. We again assume the same boundary conditions as in the previous section. Taking the same spatial discretization as before, and again applying the same 5-points stencil for the second order derivatives, eq. (7) transforms to the following set of finite difference equations:

$$\frac{1}{4} (c_{i+1,j} + c_{i-1,j} + c_{i,j+1} + c_{i,j-1}) = c_{i,j} \quad (11)$$

for all values of (i, j) . Note that the superscript k is no longer present, because the time behavior is now suppressed. Many methods exist to solve the equations. We will here concentrate on iterative methods, because they are potentially efficient, and demand less memory than direct methods (and allow for relative easy parallelism). A direct method is tried in exercise set 3.

1.4 The Jacobi Iteration

Using now the superscript k to denote the k -th iteration, the Jacobi iteration is immediately suggested from eq. (11):

$$c_{i,j}^{k+1} = \frac{1}{4} (c_{i+1,j}^k + c_{i-1,j}^k + c_{i,j+1}^k + c_{i,j-1}^k) \quad (12)$$

Note that eq. (12) is easily derived from eq. (7) by putting

$$\frac{\delta t D}{\delta x^2} = \frac{1}{4} \quad (13)$$

i.e. the Jacobi iteration is nothing but the solution of the time-dependent equation using the scheme from eq. (7) with the maximum allowed time step. This suggests that more efficient iterative methods are needed.

There is one noticeable difference between the Jacobi iteration and the solution of the time dependent equation. In the time dependent case one defines the time step and the total time that should be simulated. In an iterative method one needs a stopping condition. This stopping condition typically is some global measure. Assume that the stopping condition is such that the solution is assumed to be converged if for all values of (i, j)

$$\delta \equiv \max_{i,j} |c_{i,j}^{k+1} - c_{i,j}^k| < \epsilon, \quad (14)$$

where ϵ is some small number, say 10^{-5} .

For implementing the Jacobi iteration, note that separate matrices are needed for $c_{i,j}^k$ and $c_{i,j}^{k+1}$, one cannot simply use one matrix and update it, as one would then overwrite values that are needed later.

1.5 The Gauss-Seidel Iteration

An improvement over the Jacobi iteration is the Gauss-Seidel iteration, where during the iteration a new value is used as soon as it has been calculated. Assuming that the iteration proceeds along the rows (i.e. incrementing i for fixed j), the Gauss-Seidel iteration reads

$$c_{i,j}^{k+1} = \frac{1}{4} (c_{i+1,j}^k + c_{i-1,j}^{k+1} + c_{i,j+1}^k + c_{i,j-1}^{k+1})$$

The Gauss-Seidel iteration is not a big improvement over the Jacobi iteration (in terms of the amount of iterations needed for convergence) and is only a first step in introducing the Successive Over Relaxation method (next section). However, the update can be performed *in place*.

1.6 Successive Over Relaxation

Now that you have the parallel Gauss-Seidel iteration in place, it is easy to take the next and final step. The Gauss-Seidel iteration did not provide a huge improvement over the Jacobi iteration. A next improvement comes from the Successive Over Relaxation (SOR). SOR is obtained from Gauss-Seidel by an over-correction of the new iterate, in formula

$$c_{i,j}^{k+1} = \frac{\omega}{4} (c_{i+1,j}^k + c_{i-1,j}^{k+1} + c_{i,j+1}^k + c_{i,j-1}^{k+1}) + (1 - \omega)c_{i,j}^k$$

This method converges only for $0 < \omega < 2$. For $\omega < 1$ the method is called under-relaxation. The new value is then the weighted average of the Gauss-Seidel method and the previous value. For $\omega = 1$ we recover the Gauss-Seidel iteration.

It turns out that for our diffusion problem the optimal ω (that minimizes the number of iterations) lies somewhere in the interval $1.7 < \omega < 2$. The exact value depends on the grid size N .

H. (1 point) Implement the Jacobi iteration, the Gauss-Seidel method and SOR. Try $N = 50$. Test the methods by comparing the result to the analytical result in eq. (5), i.e. the linear dependence of the concentration on y .

I. (1 point) Show how the convergence measure δ in eq. (14) depends on the number of iterations k for each of the methods. A log-lin plot may be suitable. For SOR, choose a few representative values for ω .

J. (1 point) In the SOR method, find the optimal ω . How does it depend on N ?

So far we have only looked at diffusion in a rather dull domain. Now that we have an efficient iterative solver available, it's time to start including some object into the domain. So, now we assume that within our computational domain we include say a square object. We assume that the object is a sink for the diffusion concentration, that is, the concentration is zero everywhere on the object.

K. (2 points) Implement the possibility to include objects into the computational domain. The objects should be sinks. Experiment a little bit with some objects in the computational domain (e.g. a rectangle or a few rectangles, ...). What is the influence on the number of iterations. What about the optimal ω , is it influenced by the presence of objects? Look at the resulting concentration fields, and try to interpret what happens. The implementation in this exercise will also be used for diffusion-limited aggregation in Set 2.

Hint: For the iterations, the presence of the objects is not complicated. If a point (i, j) is part of an object, the concentration is just 0, and an iteration is not necessary (i.e., the new value is also 0). Therefore, you must implement some easy encoding of the object in the computational grid, and during the iterations simply test if the grid point that you are updating is part of the object or not. If not, you apply the SOR rule, if yes, just put the new value to zero. The easiest encoding is just an extra array of integers, where e.g. a one-value would code for the presence of an object, and a zero value for the absence of an object.

Optional. (1 point) Think of a way to incorporate objects with insulating material in your domain. What changes in the time evolution of the system? And in the final state?

2 Assignment Set 2

2.1 Diffusion Limited Aggregation

The diffusion-limited aggregation (DLA) is a growth model based on diffusing particles. The growth is started with a single seed (just a small square object in a single lattice point, at the bottom of the computational domain). As described in detail in the lecture notes, DLA can be modeled by solving the time independent diffusion equation (i.e. the Laplace equation), locating growth candidates around the cluster, and assigning a growth probability p_g for each growth candidate, as a function of the concentration of diffusing nutrients at that growth location. Next, a single growth candidate is added to the cluster with probability p_g . After this growth step, the diffusion equation is again solved, and the process is iterated for a large number of growth steps. You currently have almost all the tools available for a simulation of the DLA growth model. Let us take a closer look at the growth model itself, allowing you to insert this into your programs. Figure 3 shows a possible configuration of an object (the filled dots) and the growth candidates (the open circles). A growth candidate is basically a lattice site that is not part of the object, but whose north, east, south, or west neighbor is part of the object.

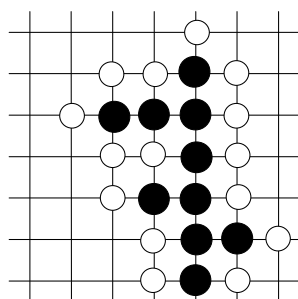


Figure 3: DLA cluster (filled circles) with growth candidate sites (open circles)

The probability for growth at each of the growth candidates is calculated by

$$p_g(i, j) = \frac{c_{i,j}^\eta}{\sum_{\text{growth candidates}} c_{i,j}^\eta}$$

The parameter η determines the shape of the object. For $\eta = 1$ we get the normal DLA cluster. For $\eta < 1$ the object becomes more compact (with $\eta = 0$ resulting in the Eden cluster), and for $\eta > 1$ the cluster becomes more open (and finally resembles say a lightning flash).

Modelling the growth is now a simple procedure. A set is created of all growth candidates with their associated weights, and a single candidate is chosen. *Hint: use `numpy.choice` with the `p` parameter for this*

A. (4 points) Implement the growth model, paying special attention to the calculation of the growth probabilities. Run a number of growth simulations. Try to do this for a domain of size 100×100 and investigate the influence of the η parameter. Can you still optimise by setting your ω parameter in the SOR iteration to a specific value?

Hint: the SOR iteration is run over and over again on a slowly growing object. As the growth step is constructed in such a way that on average only one lattice site is grown to the object, the concentration fields will hardly change. Therefore, it is advantageous to start a new SOR iteration with the solution of the previous growth step. Also, you can start the simulation with the analytical result for the empty system, the linear concentration gradient of eq. (5).

Optional: (1 point) Think of a way to reduce the time required to solve the diffusion equation. Compare your results for the 100×100 grid in question A and

try larger grid sizes. Some suggestions: parallelize one of the iteration schemes in the previous exercise set, and possibly use a GPU to do the calculations.

2.2 Monte Carlo simulation of DLA

DLA can also be simulated by releasing random walkers on a grid, and letting them walk until they hit the cluster. When they hit, the walkers are stopped and become part of the cluster.

One random walker at a time is released in the system. It moves in steps, which are randomly chosen to be one lattice point up, down, left, or right. If the walker reaches a cell neighboring the cluster, the walker is stopped there, so that the cell with the walker becomes part of the cluster.

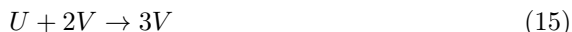
To simulate with the same boundary conditions as above, start the walkers on a randomly chosen point on the top boundary. If a walker walks out of the system on the top or bottom boundary it is removed and a new one created instead. If it walks across the left or right boundary, it should enter the system from the other side, as periodic boundary conditions were assumed in the horizontal direction.

B. (2 points) Implement the Monte Carlo version of DLA. Compare the resulting cluster to those obtained with the diffusion equation.

C. (1 point) In this model, the η parameter is no longer easily variable, it is fixed to 1. However, another parameter can be introduced, namely a sticking probability p_s . The sticking rule can then be stated in the following way: if the walker enters a cell which is a neighbor of the cluster, it stops there with probability p_s . If it does not stick, the walk continues as normal. The walker is however not allowed to move into a site belonging to the cluster. Run the simulation for different values of p_s , and plot the results. How does the cluster shape depend on p_s ?

2.3 The Gray-Scott model - A reaction-diffusion system

The Gray-Scott model (J. E. Pearson, *Science*, Vol 261, 5118, 189-192 (1993)) describes a system of chemical reactions, involving the two chemicals U and V. Both chemicals diffuse in the system, and also react with each other. The reaction rate at any point in space, is determined by the *local* concentrations of U and V. The reactions are:



U is continuously fed into the system. It then reacts with V to produce more V. V spontaneously decays into P, a reaction product which is not interacting with U and V. The first reaction is said to be autocatalytic, since the reaction product V enhances the production of itself.

If we let u and v denote the concentrations of U and V, the following equations can be formulated.

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u - uv^2 + f(1 - u), \quad (17)$$

$$\frac{\partial v}{\partial t} = D_v \nabla^2 v + uv^2 - (f + k)v. \quad (18)$$

Here, f controls the rate at which U is supplied, and $f + k$ controls the rate at which V decays. For different values of f and k a large variety of behaviors can be observed. Some result in stable patterns, while others remain time-dependent.

D. (3 points) Implement the Gray-Scott model in two dimensions. Explain the discretization and how the equations and boundary conditions are implemented in the program (you may choose which boundary conditions to use). Plot the resulting concentrations of U and/or V for several choices of the parameters. The

time-dependent diffusion program from Set 1 can be used as a base. In this case, there are two variables to keep track of. For parameters, start with $\delta t = 1, \delta x = 1, D_u = 0.16, D_v = 0.08, f = 0.035, k = 0.060$. For the initial conditions, you can take $u = 0.5$ everywhere in the system, and $v = 0.25$ in a small square in the center of the system, and 0 outside. Try adding a small amount of noise too.

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.

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \quad (19)$$

We look for a solution in the form

$$u(x, y, t) = v(x, y)T(t) \quad (20)$$

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

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

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:

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

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

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

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

$$\nabla^2 v(x, y) = Kv(x, y), \quad (24)$$

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 $\mathbf{M}\mathbf{v} = K\mathbf{v}$, where \mathbf{M} is a matrix, \mathbf{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 \mathbf{v} contains only points inside the domain. You need to construct an indexing scheme mapping index in \mathbf{v} to a point on the grid, and construct the matrix so that all connections and boundary conditions are satisfied.
- 2) The vector \mathbf{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 $\mathbf{M}\mathbf{v} = K\mathbf{v}$, 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 $\mathbf{M}\mathbf{c} = \mathbf{b}$ 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 \mathbf{M} and the vector \mathbf{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 Newton's 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:

$$\frac{\partial x_i}{\partial t} = v_i \quad (25)$$

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

With x_i, v_i, m_i indicating the position, velocity and mass of the i^{th} particle respectively ($i = 1, 2, \dots, N$). F and U are the force and potential which depend on $\{x\}$, the ensemble over all particles $\{x_1, x_2, \dots, x_N\}$. 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: $(x_n, x_{n+1}, x_{n+2}, \dots)$ and $(v_{n+1/2}, v_{n+3/2}, v_{n+5/2}, \dots)$. 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.

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

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

I. (bonus, 2 points) Implement the leapfrog method for a simple one dimensional harmonic oscillator where the force is a function following Hooke's 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 restoring force giving the following velocity term:

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

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/>. Your UvA ID will give you a 'pro' account.

Drawing figure

Beyond matplotlib you might want to use TikZ (latex) or Inkscape.

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

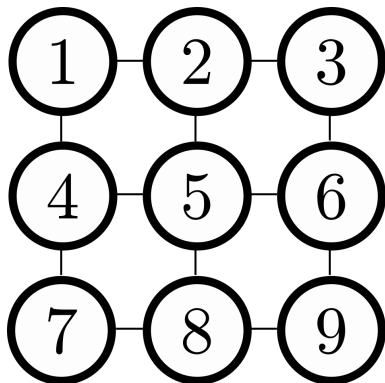
<https://projects.scipy.org/getting-started.html>

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.



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