Solving linear systems

Aksel Hiorth

University of Stavanger

Mar 31, 2023

Contents

List of inline comments

Most problems in nature are nonlinear. That means that the system response is not proportional to the system variables, e.g. doubling the CO_2 concentration in the atmosphere does not lead to a doubling of the earth surface temperature. Still, linear solvers lies at the heart of all grid based models describing e.g. the earths climate. The reason is that although the *global* model is nonlinear, the model can be formulated *locally* as a linear model. Typically the simulation code solves the nonlinear problem through a series of steps where each step is a solution of a linear problem. The topic of solving linear systems of equations have been extensively studied, and sophisticated linear equation solving packages have been developed. Python uses functions from the LAPACK[1] library.

In the next sections we will show in detail how differential equations can be solved as a linear problem. We will first start off by deriving one of the most useful differential equations describing conservation of a quantity, e.g. mass, energy, momentum, charge.

1 The Continuity Equation

The continuity equation is fundamental to all mathematical models describing a physical phenomenon. To gain more understanding of its origin we will take the time to derive it from first principles. We will do so in one dimension, consider a volume in space between $A(x)$ and $A(x + dx)$ in figure 1. To be concrete we will assume that the green arrows represents the flow of heat. Thus there are heat flowing into and out of the system, and also heat that can be generated within the system by e.g. chemical reactions. The conservation equation can be formulated with words

$$\frac{\text{heat into V(x)}}{\text{time}} - \frac{\text{heat out of V(x)}}{\text{time}} + \frac{\text{heat generated in V(x)}}{\text{time}} = \frac{\text{change of heat in V(x)}}{\text{time}}. \quad (1)$$

We formulate the conservation equation per time, because we would like to investigate the time dependency of heat flow. The next step is to replace the terms "heat into/out of" with a useful mathematical quantity. It turns out that the term *flux* is particularly useful, because it is an *intensive* quantity.

[1] `https://en.wikipedia.org/wiki/LAPACK`

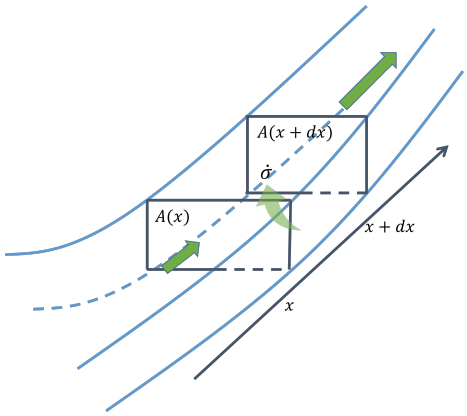

Figure 1: A closed volume, $V(x) = A(x)dx$, where a quantity flows in and out (illustrated by the green lines), there is also a possibility for generation or loss of the same quantity inside the volume.

An intensive quantity is a quantity that is *independent of the system size*, like density. The flux is denoted by the symbol J

$$J(x) = \frac{\text{quantity (heat)}}{\text{area} \cdot \text{time}}, \tag{2}$$

and was first introduced by Isaac Newton. Thus to find the amount of heat transported through a surface per time we simply multiply the flux with the surface area. Next, we define the heat per volume as $q(x)$, and the heat produced per volume as σ. Then equation (1) can be written

$$\begin{aligned}\frac{J(x)A(x)}{dt} - \frac{J(x+dx)A(x+dx)}{dt} + \frac{\sigma(t+dt)V(x) - \sigma(t)V(x)}{dt} \\ = \frac{q(t+dt)V(x) - q(t)V(x)}{dt}.\end{aligned} \tag{3}$$

Using Taylor expansion we can write

$$J(x+dx)A(x+dx) = J(x)A(x) + \frac{d(J(x)A(x)}{dx}dx + \mathcal{O}(dx^2), \tag{4}$$

$$\sigma(t+dt) = \sigma(t) + \frac{d\sigma}{dt}dt + \mathcal{O}(dt^2),$$

$$q(t+dt) = q(t) + \frac{dq}{dt}dt + \mathcal{O}(dt^2), \tag{5}$$

Inserting these equations into equation(3), using $V(x) = A(x)dx$, and taking the limit $dx, dt \to 0$ we arrive at

The continuity equation in 1 dimension.

$$-\frac{d(J(x)A(x))}{dx}+\frac{d\sigma(t)}{dt}A(x)=\frac{dq(t)}{dt}A(x). \tag{6}$$

We have kept the area in equation (6), because we are only considering flow of heat in one dimension and then we can allow for the area to change in the y and z dimension. When the continuity equation is derived in three dimensions, one consider a volume $V(x,y,z)=dxdydz$, then the area in equation (6) will drop out and $d/dx \to \nabla = [\partial/\partial x, \partial/\partial y, \partial/\partial z]$

The continuity equation in 3 dimensions.

$$-\nabla\cdot\mathbf{J}+\frac{d\sigma(t)}{dt}=\frac{dq(t)}{dt}. \tag{7}$$

2 Continuity Equation as a linear problem

How can a differential equation be formulated as a matrix problem? To see this we need to discretize equation (6). We will discretize the equation in one dimension, and we will use a regular grid, where we keep the same distance, h, between the points. Assume our system has dimension L, in figure 2, there are two examples of discretization.

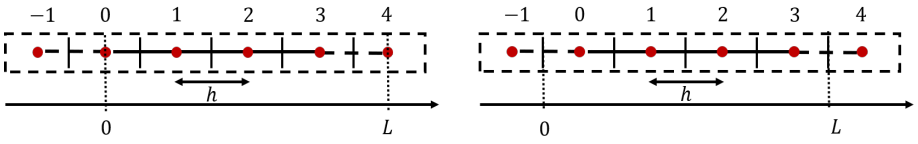

Figure 2: Examples of discretization of a system with length L (left) the boundaries lies exactly at the boundary nodes, (right) boundary nodes lies half-way between the grid nodes.

There are many things to consider when discretizing equations, but perhaps the most important are

1. Treat the boundary nodes correctly. In most cases the dominating numerical errors are introduced through the boundaries. Always draw a picture of the system, if the boundaries lies exactly at the grid nodes it is usually easier to find a good numerical representation. If the boundaries lies a distance from the nodes, e.g. to the right in figure 2, then one usually need to do some interpolation.

2. Should finite volume or finite difference approach be used? A finite volume approach is especially attractive for conservation equations.

Finite difference and finite volume.

We have already encountered finite difference discretization in the last chapter where we used various approximations to calculate derivatives, i.e. we calculate derivatives by calculating the *difference* between $f(x+h)$ and $f(x)$ (or $f(x+h)$ and $f(x-h)$). The finite volume formulation is also a finite difference scheme, but it is formulated such that we always ensure that the quantity we are simulating is conserved (regardless of numerical errors). Formally, one transforms the divergence term (the term that contains the flux $\nabla \mathbf{J}$) into a surface integral using the Gauss (divergence) theorem

$$\int_V \nabla \cdot \mathbf{J} = \int_S \mathbf{J} \cdot \hat{\mathbf{n}} \tag{8}$$

There are excellent books written on the finite volume method, see e.g. [1]. Here we will mainly focus on the key idea, which is to formulate a scheme that conserves the flux. The process of formulating a finite volume scheme is very close to the derivation of the continuum equation we did in the beginning of the chapter. We consider our numerical discretization as several boxes (exactly like the dotted lines in figure 2), the continuum equation is written down for each box an therefore we are ensured that the quantities are conserved *regardless of the size of the boxes*.

Example: Finite difference and volume discretization of the heat equation.

Let us consider the heat equation, where the heat flux is given as

$$J = -k\frac{dT}{dx}, \tag{9}$$

where k describes the thermal conductivity of the solid. We will further assume that there is a constant source term $d\sigma/dt = \kappa = const$, and steady state $dq/dt = 0$. Then equation (6) can be written

$$k\frac{d^2T}{dx^2} + \kappa = 0, \tag{10}$$

The finite difference discretization is now straight forward, just replace the term d^2T/dx^2 with a suitable finite difference formula for the second

derivative, e.g.

$$k\frac{T(x+h)+T(x-h)-2T(x)}{h^2}+\kappa=0,$$
$$k\frac{T_{i+1}+T_{i-1}-2T_i}{h^2}+\kappa=0. \tag{11}$$

Note that in the last equations we have introduced the short hand notation $T(x)\equiv T_i$, and $T(x\pm h)=T_{i\pm1}$.

The finite volume discretization approach is slightly different, we then operate with *cell averaged values*. The heat in the box is the volume averaged heat. Since the divergence term is replaced with a surface integral, equation (8), we calculate the flow of heat into the boundary $x-h/2$ and out of the boundary $x+h/2$ as

$$\frac{J_{x+h/2}-J_{x-h/2}}{h}+\kappa=0. \tag{12}$$

Note that this equation is exactly the same as equation (3), with the only exception that the point x is placed in the center of the box. The diffusive flux is $-kdT/dx$, and in order to be consistent with this law we have to write the flux between two cells as proportional to the difference between the cell average values

$$-\frac{k}{h}\left(\frac{T_{i+1}-T_i}{h}-\frac{T_i-T_{i-1}}{h}\right)+\kappa=0,$$
$$k\frac{T_{i+1}+T_{i-1}-2T_i}{h^2}+\kappa=0. \tag{13}$$

In this case we actually recover the same equation as we did for the finite difference approach equation (11).

Boundary conditions. Basically there are two types of boundary conditions i) the flux is known at the edges of the computational domain and/or ii) the physical quantity we are solving for is known. To be more specific, and to see how all connects, we will continue with the example above on the heat equation. Consider the outline of nodes as in figure 2, we will consider two possibilities i) where the physical boundary lies exactly between nodes, and ii) where the physical boundary is exactly at the grid nodes. In the finite volume scheme, we need to make sure that the flux over the surface is calculated correctly, and then we have to use the formulas in figure 3

$$\left.\frac{dT}{dx}\right|_{x=0}=\frac{T_{-1}-T_0}{h}+\mathcal{O}(h^2). \tag{14}$$

Note that if the boundary node lies exactly at $x=0$, we have to replace T_0 with T_1. A flux boundary condition is usually called Neumann boundary condition after Carl Neumann (1832–1925) a German mathematician, and the constant

value boundary condition is called Dirichlet boundary condition after another German mathematician, Peter Gustav Lejeune Dirichlet (1805–1859). If the boundary nodes lies exactly at the physical boundary, it is trivial to implement, just replace $T_N = T_b$ i.e. with the boundary value. On the other hand if the physical boundary lies a distance from the node, we have to interpolate the value from the physical coordinate to the simulation node.

$$\begin{aligned} T_N &= T(x+h) = T(x+h/2+h/2) \\ &= T_{N+1/2} + \left.\frac{dT}{dx}\right|_{x+h/2} + \mathcal{O}(h^2) = T_b + \frac{T_N - T_{N-1}}{h}\frac{h}{2} + \mathcal{O}(h^2), \text{ hence:} \\ T_N &= 2T_b - T_{N-1} + \mathcal{O}(h^2). \end{aligned} \tag{15}$$

Notice that the result make sense, $T_b = (T_N + T_{N-1})/2$, i.e. the value midway is the average of the values at the neighboring nodes.

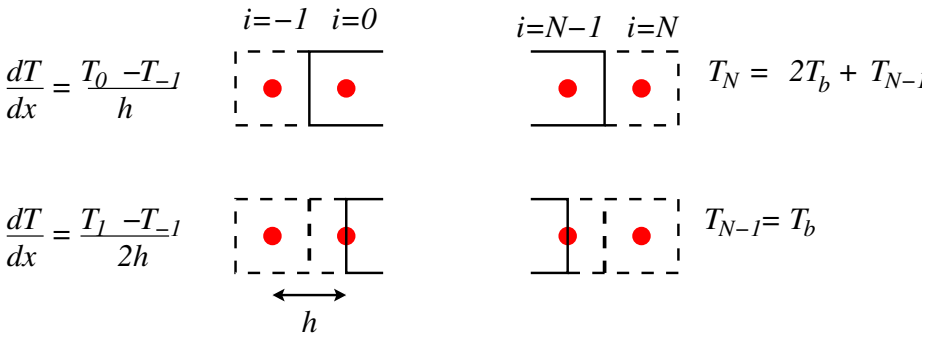

Figure 3: Flux boundary condition (Neumann), and value boundary condition (Dirichlet). For the upper right boundary condition we use Taylors formula to interpolate, see equation (15).

Example: Steady state heat equation as a linear problem.

Consider the case where we have 4 grid nodes and the outline of the simulation nodes are as in figure 2 to the left, i.e. nodes at the physical boundaries. Assume a zero flux boundary condition to the left, and a constant temperature,T_b, to the right. Write the heat equation

$$k\frac{d^2T}{dx^2} + \kappa = 0, \tag{16}$$

as a matrix equation.

Solution: First, we use the discrete version of equation (16) in equation (11) for $i = 0, 1, 2, 3$

$$\begin{aligned} T_{-1} + T_1 - 2T_0 &= -h^2\kappa/k, \\ T_0 + T_2 - 2T_1 &= -h^2\kappa/k \\ T_1 + T_3 - 2T_2 &= -h^2\kappa/k \\ T_2 + T_4 - 2T_3 &= -h^2\kappa/k. \end{aligned} \tag{17}$$

Now, we have four equations, but six unknowns $(T_{-1}, T_0, T_1, T_2, T_3, T_4)$. T_{-1}, and T_4 can be found from the boundary conditions. Using the formulas in figure 3 at the lower left and lower right, we get $dT/dx = 0$, and $T_{-1} = T_1$, and $T_4 = T_b$. Thus the first and last equation in equation (17), can be written

$$\begin{aligned} 2T_1 - 2T_0 &= -h^2\kappa/k, \\ T_2 - 2T_3 &= -h^2\kappa/k - T_b. \end{aligned} \tag{18}$$

Now, we can formulate equation (17) as a matrix problem, with the unknowns on the left side and the unknown on the right hand side.

$$\begin{pmatrix} -2 & 2 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} T_0 \\ T_1 \\ T_2 \\ T_3 \end{pmatrix} = \begin{pmatrix} -h^2\kappa/k \\ -h^2\kappa/k \\ -h^2\kappa/k \\ -h^2\kappa/k - T_b \end{pmatrix}. \tag{19}$$

In principle, to discretize an equation is straight forward, but there are some

First, we are going to consider a *steady state* solution. Steady state means that the solution does no longer change as a function of time, i.e. $dq/dt = 0$ in equation (6). We are also going to assume that the area is constant $A(x) = A$, thus the equation we want to solve is

3 Solving linear equations

There are a number of excellent books covering this topic, see e.g. [2, 6, 4, 5]. In most of the examples covered in this course we will encounter problems where we have a set of *linearly independent* equations and one equation for each unknown. For these type of problems there are a number of methods that can be used, and they will find a solution in a finite number of steps. If a solution cannot be found it is usually because the equations are not linearly independent, and our formulation of the physical problem is wrong.

Assume that we would like to solve the following set of equations:

$$2x_0 + x_1 + x_2 + 3x_3 = 1, \tag{20}$$
$$x_0 + x_1 + 3x_2 + x_3 = -3, \tag{21}$$
$$x_0 + 4x_1 + x_2 + x_3 = 2, \tag{22}$$
$$x_0 + x_1 + 2x_2 + 2x_3 = 1. \tag{23}$$

These equations can be written in matrix form as:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}, \tag{24}$$

where:

$$\mathbf{A} \equiv \begin{pmatrix} 2 & 1 & 1 & 3 \\ 1 & 1 & 3 & 1 \\ 1 & 4 & 1 & 1 \\ 1 & 1 & 2 & 2 \end{pmatrix} \qquad \mathbf{b} \equiv \begin{pmatrix} 1 \\ -3 \\ 2 \\ 1 \end{pmatrix} \qquad \mathbf{x} \equiv \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}. \tag{25}$$

You can easily verify that $x_0 = -4, x_1 = 1, x_2 = -1, x_3 = 3$ is the solution to the above equations by direct substitution. If we were to replace one of the above equations with a linear combination of any of the other equations, e.g. replace equation (23) with $3x_0 + 2x_1 + 4x_2 + 4x_3 = -2$, there would be no unique solution (infinite number of solutions). This can be checked by calculating the determinant of the matrix $\mathbf{A}$, if $\det \mathbf{A} = 0$, What is the difficulty in solving these equations? Clearly if none of the equations are linearly dependent, and we have N independent linear equations, it should be straight forward to solve them? Two major numerical problems are i) even if the equations are not exact linear combinations of each other, they could be very close, and as the numerical algorithm progresses they could at some stage become linearly dependent due to roundoff errors. ii) roundoff errors may accumulate if the number of equations are large [2].

3.1 Gauss-Jordan elimination

Let us continue the discussion by consider Gauss-Jordan elimination, which is a *direct* method. A direct method uses a final set of operations to obtain a solution. According to [2] Gauss-Jordan elimination is the method of choice if we want to find the inverse of $\mathbf{A}$. However, it is slow when it comes to calculate the solution of equation (24). Even if speed and memory use is not an issue, it is also not advised to first find the inverse, $\mathbf{A}^{-1}$, of $\mathbf{A}$, then multiply it with $\mathbf{b}$ to obtain the solution, due to roundoff errors (Roundoff errors occur whenever we subtract to numbers that are very close to each other). To simplify our notation, we write equation (25) as:

$$\left(\begin{array}{cccc|c} 2 & 1 & 1 & 3 & 1 \\ 1 & 1 & 3 & 1 & -3 \\ 1 & 4 & 1 & 1 & 2 \\ 1 & 1 & 2 & 2 & 1 \end{array}\right). \tag{26}$$

The numbers to the left of the vertical dash is the matrix $\mathbf{A}$, and to the right is the vector $\mathbf{b}$. The Gauss-Jordan elimination procedure proceeds by doing the same operation on the right and left side of the dash, and the goal is to get only zeros on the lower triangular part of the matrix. This is achieved by multiplying rows with the same (nonzero) number, swapping rows, adding a multiple of a row to another:

$$\left(\begin{array}{cccc|c} 2 & 1 & 1 & 3 & 1 \\ 1 & 1 & 3 & 1 & -3 \\ 1 & 4 & 1 & 1 & 2 \\ 1 & 1 & 2 & 2 & 1 \end{array}\right) \rightarrow \left(\begin{array}{cccc|c} 2 & 1 & 1 & 3 & 1 \\ 0 & 1/2 & 5/2 & -1/2 & -7/2 \\ 0 & 7/2 & 1/2 & -1/2 & 3/2 \\ 0 & 1/2 & 3/2 & 1/2 & 1/2 \end{array}\right) \rightarrow \tag{27}$$

$$\left(\begin{array}{cccc|c} 2 & 1 & 1 & 3 & 1 \\ 0 & 1/2 & 5/2 & -1/2 & -7/2 \\ 0 & 0 & -17 & 3 & 26 \\ 0 & 0 & 1 & -1 & 4 \end{array}\right) \rightarrow \left(\begin{array}{cccc|c} 2 & 1 & 1 & 3 & 1 \\ 0 & 1/2 & 5/2 & -1/2 & -7/2 \\ 0 & 0 & -17 & 3 & 26 \\ 0 & 0 & 0 & 14/17 & 42/17 \end{array}\right)$$

The operations done are: $(1 \rightarrow 2)$ multiply first row with $-1/2$ and add to second, third and the fourth row, $(2 \rightarrow 3)$ multiply second row with -7, and add to third row, multiply second row with -1 and add to fourth row, $(3 \rightarrow 4)$ multiply third row with $-1/17$ and add to fourth row. These operations can easily be coded into Python:

```
A = np.array([[2, 1, 1, 3],[1, 1, 3, 1],
              [1, 4, 1, 1],[1, 1, 2, 2 ]],float)
b = np.array([1,-3,2,1],float)
N=4
# Gauss-Jordan Forward Elimination
for i in range(1,N):
    fact    = A[i:,i-1]/A[i-1,i-1]
    A[i:,] -= np.outer(fact,A[i-1,])
    b[i:]  -= b[i-1]*fact
```

The python code is a bit compact, below there is an implementation using for loops

```
# Gauss-Jordan Forward Elimination - for loops
for i in range(N):
    for j in range(i+1,N):
        fact    = A[j,i]/A[i,i]
        for k in range(i+1,N):
            A[j,k] = A[j,k]- fact*A[i,k]
        b[j]  = b[j]- b[j-1]*fact
    A[j,i]= 0. # alternatively k=i,...,N
```

Number of (long) operations.

The code above reveals that that there are quite a few multiplications or divisions being performed in the forward elimination. Multiplications and divisions are more time consuming than addition and subtraction, and are usually termed *long* operations. Not all loops runs from zero to N, the innermost from $k = i + 1 \ldots N - 1$, i.e. a total of $N - i - 2$, the second contains $N - i - 2$ and one multiplication for the `b` vector. Hence we have number of long operations

$$\sum_{i=0}^{N-1} (N - i - 2)^2 + (N - i - 2) = \frac{N}{3}(N^2 - 3N + 2). \tag{28}$$

The important result is that when the system of equations becomes large $N^3 \gg N^2$ and the algorithm scales as N^3.

Notice that the final matrix has only zeros beyond the diagonal, such a matrix is called *upper triangular*. We still have not found the final solution, but from an upper triangular (or lower triangular) matrix it is trivial to determine the solution. The last row immediately gives us $14/17z = 42/17$ or $z = 3$, now we have the solution for z and the next row gives: $-17y + 3z = 26$ or $y = (26 - 3 \cdot 3)/(-17) = -1$, and so on. In a more general form, we can write our solution of the matrix $\mathbf{A}$ after making it upper triangular as:

$$\begin{pmatrix} a'_{0,0} & a'_{0,1} & a'_{0,2} & a'_{0,3} \\ 0 & a'_{1,1} & a'_{1,2} & a'_{1,3} \\ 0 & 0 & a'_{2,2} & a'_{2,3} \\ 0 & 0 & 0 & a'_{3,3} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \end{pmatrix} \tag{29}$$

The back substitution can then be written formally as:

$$x_i = \frac{1}{a'_{ii}} \left[b'_i - \sum_{j=i+1}^{N-1} a'_{ij} x_j \right], \quad i = N - 1, N - 2, \ldots, 0 \tag{30}$$

The back substitution can now easily be implemented in Python as:

```
# Back substitution
sol = np.zeros(N,float)
sol[N-1]=b[N-1]/A[N-1,N-1]
for i in range(2,N+1):
    sol[N-i]=(b[N-i]-np.dot(A[(N-i),],sol))/A[N-i,N-i]
```

Notice that in the Python implementation, we have used vector operations instead of for loops. This makes the code more efficient, but it could also be implemented with for loops:

```
# Back substitution - for loop
sol = np.zeros(N,float)
```

```
for i in range(N-1,-1,-1):
    sol[i]= b[i]
    for j in range(i+1,N):
        sol[i] -= A[i][j]*sol[j]
    sol[i] /= A[i][i]
```

Number of (long) operations.

As for the forward elimination, we can find how the backward substitution scales. Notice that here there are only two loops, hence we have number of long operations

$$\sum_{i=0}^{N-1}(N-i-1) = \frac{N}{2}(N-1). \tag{31}$$

Thus, the backward substitution scales as N^2.

There are at least two things to notice with our implementation:

- Matrix and vector notation makes the code more compact and efficient. In order to understand the implementation it is advised to put $i = 1, 2, 3, 4$, and then execute the statements in the Gauss-Jordan elimination and compare with equation (27).
- The implementation of the Gauss-Jordan elimination is not robust, in particular one could easily imagine cases where one of the leading coefficients turned out as zero, and the routine would fail when we divide by `A[i-1,i-1]`. By simply changing equation (21) to $2x_0+x_1+3x_2+x_3 = -3$, when doing the first Gauss-Jordan elimination, both x_0 and x_1 would be canceled. In the next iteration we try to divide next equation by the leading coefficient of x_1, which is zero, and the whole procedure fails.

3.2 Pivoting

The solution to the last problem is solved by what is called *pivoting*. The element that we divide on is called the *pivot element*. It actually turns out that even if we do Gauss-Jordan elimination *without* encountering a zero pivot element, the Gauss-Jordan procedure is numerically unstable in the presence of roundoff errors [2]. There are two versions of pivoting, *full pivoting* and *partial pivoting*. In partial pivoting we only interchange rows, while in full pivoting we also interchange rows and columns. Partial pivoting is much easier to implement, and the algorithm is as follows:

1. Find the row in $\mathbf{A}$ with largest absolute value in front of x_0 and change with the first equation, switch corresponding elements in $\mathbf{b}$

2. Do one Gauss-Jordan elimination, find the row in $\mathbf{A}$ with the largest absolute value in front of x_1 and switch with the second (same for $\mathbf{b}$), and so on.

For a linear equation we can multiply with a number on each side and the equation would be unchanged, so if we where to multiply one of the equations with a large value, we are almost sure that this equation would be placed first by our algorithm. This seems a bit strange as our mathematical problem is the same. Sometimes the linear algebra routines tries to normalize the equations to find the pivot element that would have been the largest element if all equations were normalized according to some rule, this is called *implicit pivoting*.

3.3 LU decomposition

As we have already seen, if the matrix $\mathbf{A}$ is reduced to a triangular form it is trivial to calculate the solution by using back substitution. Thus if it was possible to decompose the matrix $\mathbf{A}$ as follows:

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U} \tag{32}$$

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} = \begin{pmatrix} l_{0,0} & 0 & 0 & 0 \\ l_{1,0} & l_{1,1} & 0 & 0 \\ l_{2,0} & l_{2,1} & l_{2,2} & 0 \\ l_{3,0} & l_{3,1} & l_{3,2} & l_{3,3} \end{pmatrix} \cdot \begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} & u_{0,3} \\ 0 & u_{1,1} & u_{1,2} & u_{1,3} \\ 0 & 0 & u_{2,2} & u_{2,3} \\ 0 & 0 & 0 & u_{3,3} \end{pmatrix}. \tag{33}$$

The solution procedure would then be to rewrite equation (24) as:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{L} \cdot \mathbf{U} \cdot \mathbf{x} = \mathbf{b}, \tag{34}$$

If we define a new vector $\mathbf{y}$:

$$\mathbf{y} \equiv \mathbf{U} \cdot \mathbf{x}, \tag{35}$$

we can first solve for the $\mathbf{y}$ vector:

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b}, \tag{36}$$

and then for $\mathbf{x}$:

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y}. \tag{37}$$

Note that the solution to equation (36) would be done by *forward substitution*:

$$y_i = \frac{1}{l_{ii}} \left[b_i - \sum_{j=0}^{i-1} l_{ij} x_j \right], \quad i = 1, 2, \ldots N-1. \tag{38}$$

Why go to all this trouble? First of all it requires (slightly) less operations to calculate the LU decomposition and doing the forward and backward substitution than the Gauss-Jordan procedure discussed earlier. Secondly, and more importantly, is the fact that in many cases one would like to calculate the solution for different values of the $\mathbf{b}$ vector in equation (34). If we do the LU decomposition first we can calculate the solution quite fast using backward and forward substitution for any value of the $\mathbf{b}$ vector.

The NumPy function `solve`[2], uses LU decomposition and partial pivoting, and we can find the solution to our previous problem simply by the following code:

```
from numpy.linalg import solve
x=solve(A,b)
```

4 Iterative methods

The methods described so far are what is called *direct* methods. The direct methods for very large systems might suffer from round off errors. That means that even if the computer has found a solution, the solution is "polluted" by round off errors, or stated more clearly: your solution for $\mathbf{x}$, when entered into the original equation $\mathbf{Ax} \neq \mathbf{b}$. Below we will describe one trick, and two alternative methods to the direct methods.

4.1 Iterative improvement

The first method [3] assumes that we already have solved the matrix equation (24), and obtained an *estimate* $\hat{\mathbf{x}}$ of the true solution $\mathbf{x}$. Assume that $\hat{\mathbf{x}} = \mathbf{x} + \delta\mathbf{x}$, and that

$$\mathbf{A} \cdot \hat{\mathbf{x}} = \mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}, \tag{39}$$

subtracting equation (24) we get

$$\mathbf{A} \cdot \delta\mathbf{x} = \delta\mathbf{b}. \tag{40}$$

Solving equation (39) for $\delta\mathbf{b}$ an inserting in the equation above, we get

$$\mathbf{A} \cdot \delta\mathbf{x} = \mathbf{A} \cdot \hat{\mathbf{x}} - \mathbf{b}. \tag{41}$$

The usefulness of this method assumes that we have already obtained the LU decomposition of $\mathbf{A}$, and if possible one should use a higher precision to calculate the right hand side, since there will be a lot of cancellations. Then the whole computational process it is simply to calculate the right hand side and backsubstitute. The improved solution is then obtained by subtracting $\delta\mathbf{x}$ from $\hat{\mathbf{x}}$.

[2] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.solve.html`

4.2 The Jacobi method

A completely different approach is the Jacobian method, which is simply to decompose the $\mathbf{A}$ matrix in the following way

$$\mathbf{A} = \mathbf{D} + \mathbf{R} \tag{42}$$

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} = \begin{pmatrix} a_{0,0} & 0 & 0 & 0 \\ 0 & a_{1,1} & 0 & 0 \\ 0 & 0 & a_{2,2} & 0 \\ 0 & 0 & 0 & a_{3,3} \end{pmatrix} + \begin{pmatrix} 0 & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & 0 & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & 0 & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & 0 \end{pmatrix}. \tag{43}$$

We can then write equation (24) as

$$\mathbf{D}\mathbf{x} = \mathbf{b} - \mathbf{R} \cdot \mathbf{x}. \tag{44}$$

How does this help us? First of all, the matrix $\mathbf{D}$ is easy to invert as it is diagonal, the inverse can be found by simply replace $a_{ii} \rightarrow 1/a_{ii}$. But $\mathbf{x}$ is still present on the right hand side? This is where the *iterations* comes into play, we simply guess at an initial solution $\mathbf{x}^k$, and then we use equation (44) to calculate the next solution $\mathbf{x}^{k+1}$, and so on

$$\mathbf{x}^{k+1} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R} \cdot \mathbf{x}^k). \tag{45}$$

Lets write it out on component form for a 4×4 matrix to see what is going on

$$x_0^{k+1} = \frac{1}{a_{00}}(b_0 - a_{01}x_1^k - a_{02}x_2^k - a_{03}x_3^k), \tag{46}$$

$$x_1^{k+1} = \frac{1}{a_{11}}(b_1 - a_{10}x_0^k - a_{12}x_2^k - a_{13}x_3^k), \tag{47}$$

$$x_2^{k+1} = \frac{1}{a_{22}}(b_2 - a_{20}x_0^k - a_{21}x_1^k - a_{23}x_3^k), \tag{48}$$

$$x_3^{k+1} = \frac{1}{a_{33}}(b_3 - a_{30}x_0^k - a_{31}x_1^k - a_{32}x_2^k). \tag{49}$$

Below is a Python implementation

```
def solve_jacobi(A,b,x=-1,w=1,max_iter=1000,EPS=1e-6):
    """
    Solves the linear system Ax=b using the Jacobian method, stops if
    solution is not found after max_iter or if solution changes less
    than EPS
    """
```

```
    if(x==-1): #default guess
        x=np.zeros(len(b))
    D=np.diag(A)
    R=A-np.diag(D)
    eps=1
    x_old=x
    iter=0
    w=0.1
    while(eps>EPS and iter<max_iter):
        iter+=1
        x=w*(b-np.dot(R,x_old))/D + (1-w)*x_old
        eps=np.sum(np.abs(x-x_old))
        x_old=x
    print('found solution after ' + str(iter) +' iterations')
    return x
```

A sufficient criteria for the Jacobian method to converge is if the matrix A is diagonally dominant. In the implementation above we have included a weight, which sometimes can help in the convergence even if the matrix is not diagonally dominant.

The iterative method can be appealing if we do not need a high accuracy, we can choose to stop whenever $|\mathbf{x}^{k+1} - \mathbf{x}^k|$ is small enough. For the direct method we have to follow through all the way.

Convergence.

The Jacobi method converges if the matrix $\mathbf{A}$ is strictly diagonally dominant. Strictly diagonally dominant means that the absolute value of each entry on the diagonal is greater than the sum of the absolute values of the other entries in the same row, i.e if $|a_{00}| > |a_{01} + a_{02} + \cdots|$. In general it can be shown that a iterative scheme $\mathbf{x}^{k+1} = \mathbf{P} \cdot \mathbf{x}^k + \mathbf{q}$ is convergent *if and only if* every eigenvalue, λ, of $\mathbf{P}$ satisfies $|\lambda| < 1$, i.e. the *spectral radius* $\rho(\mathbf{P}) < 1$.

4.3 The Gauss-Seidel method

It is tempting in equation (46) to use our estimate of x_0^{k+1} in the next equation, equation (47), instead of x_0^k. After all our estimate x_0^{k+1} is an *improved* estimate. This is actually the Gauss-Seidel method. This method also has the advantage that if there are memory issues, one can overwrite the old value of x_i^k. Usually the Gauss-Seidel method converges faster, but not always. A plus for the Jacobi method is that is can be parallelised, as the calculations is only dependent on the old values and do not require information about the new values as for the Gauss Seidel method. Below is a Python implementation of the Gauss-Seidel method

```
def solve_GS(A,b,x=-1,max_iter=1000,EPS=1e-6):
```

```
    """
    Solves the linear system Ax=b using the Gauss-Seidel method, stops if
    solution is not found after max_iter or if solution changes less
    than EPS
    """
    if(x==-1):
        x=np.zeros(len(b))
    D=np.diag(A)
    R=A-np.diag(D)
    eps=1
    iter=0
    while(eps>EPS and iter<max_iter):
        iter+=1
        eps=0.
        for i in range(len(x)):
            tmp=x[i]
            x[i]=(b[i]- np.dot(R[i,:],x))/D[i]
            eps+=np.abs(tmp-x[i])
    print('found solution after ' + str(iter) +' iterations')
    return x
```

5 Example: Linear regression

In the previous section, we considered a system of N equations and N unknown $(x_0, x_1, \ldots, x_N)$. In general we might have more equations than unknowns or more unknowns than equations. An example of the former is linear regression, we might have many data points and we would like to fit a line through the points. How do you fit a single lines to more than two points that does not line on the same line? One way to do it is to minimize the distance from the line to the points, as illustrated in figure 4.

Mathematically we can express the distance between a data point (x_i, y_i) and the line $f(x)$ as $y_i - f(x_i)$. Note that this difference can be negative or positive depending if the data point lies below or above the line. We can then take the absolute value of all the distances, and try to minimize them. When we minimize something we take the derivative of the expression and put it equal to zero. As you might remember from Calculus it is extremely hard to work with the derivative of the absolute value, because it is discontinuous. A much better approach is to square each distance and sum them:

$$S = \sum_{i=0}^{N-1} (y_i - f(x_i))^2 = \sum_{i=0}^{N-1} (y_i - a_0 - a_1 x_i)^2. \tag{50}$$

(For the example in figure 4, $N = 5$.) This is the idea behind *least square*, and linear regression. One thing you should be aware of is that points lying far from the line will contribute more to equation (50). The underlying assumption is that each data point provides equally precise information about the process, this

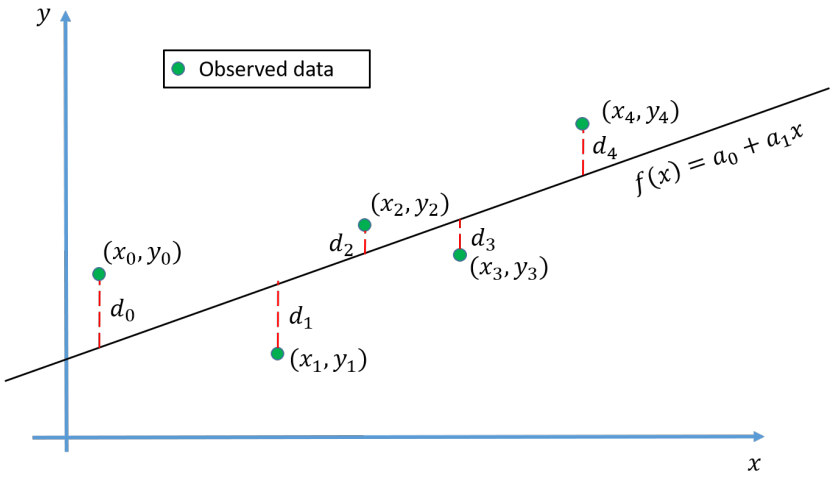

Figure 4: Linear regression by minimizing the total distance to all the points.

is often not the case. When analyzing experimental data, there may be points deviating from the expected behaviour, it is then important to investigate if these points are more affected by measurements errors than the others. If that is the case one should give them less weight in the least square estimate, by extending the formula above:

$$S = \sum_{i=0}^{N-1} \omega_i (y_i - f(x_i))^2 = \sum_{i=0}^{3} \omega_i (y_i - a_0 - a_1 x_i)^2, \tag{51}$$

ω_i is a weight factor.

5.1 Solving least square, using algebraic equations

Let us continue with equation (50), the algebraic solution is to simply find the value of a_0 and a_1 that minimizes S:

$$\frac{\partial S}{\partial a_0} = -2 \sum_{i=0}^{N-1} (y_i - a_0 - a_1 x_i) = 0, \tag{52}$$

$$\frac{\partial S}{\partial a_1} = -2 \sum_{i=0}^{N-1} (y_i - a_0 - a_1 x_i) x_i = 0. \tag{53}$$

Defining the mean value as $\overline{x} = \sum_i x_i/N$ and $\overline{y} = \sum_i y_i/N$, we can write equation (52) and (53) as:

$$\sum_{i=0}^{N-1}(y_i - a_0 - a_1 x_i) = N\overline{y} - a_0 N - a_1 N\overline{x} = 0, \tag{54}$$

$$\sum_{i=0}^{N-1}(y_i - a_0 - a_1 x_i)x_i = \sum_i y_i x_i - a_0 N\overline{x} - a_1 \sum_i x_i x_i = 0. \tag{55}$$

Solving equation (54) with respect to a_0, and inserting the expression into equation (55), we find:

$$a_0 = \overline{y} - a_1\overline{x}, \tag{56}$$

$$a_1 = \frac{\sum_i y_i x_i - N\overline{x}\,\overline{y}}{\sum_i x_i^2 - N\overline{x}^2} = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{\sum_i (x_i - \overline{x})^2}. \tag{57}$$

We leave it as an exercise to show the last expression for a_1. Clearly the equation (57) above will in most cases have a solution. But in addition to a solution, it would be good to have an idea of the goodness of the fit. Intuitively it make sense to add all the distances (residuals) d_i in figure 4. This is basically what is done when calculating R^2 (R-squared). However, we would also like to compare the R^2 between different datasets. Therefor we need to normalize the sum of residuals, and therefore the following form of the R^2 is used:

$$R^2 = 1 - \frac{\sum_{i=0}^{N-1}(y_i - f(x_i))^2}{\sum_{i=0}^{N-1}(y_i - \overline{y})^2}. \tag{58}$$

In python we can implement equation (56), (57) and (58) as:

```
def OLS(x, y):
    # returns regression coefficients
    # in ordinary least square
    # x: observations
    # y: response
    # R^2: R-squared
    n = np.size(x) # number of data points

    # mean of x and y vector
    m_x, m_y = np.mean(x), np.mean(y)

    # calculating cross-deviation and deviation about x
    SS_xy = np.sum(y*x) - n*m_y*m_x
    SS_xx = np.sum(x*x) - n*m_x*m_x

    # calculating regression coefficients
    b_1 = SS_xy / SS_xx
    b_0 = m_y - b_1*m_x
```

```
    #R^2
    y_pred = b_0 + b_1*x
    S_yy   = np.sum(y*y) - n*m_y*m_y
    y_res  = y-y_pred
    S_res  = np.sum(y_res*y_res)

    return(b_0, b_1,1-S_res/S_yy)
```

5.2 Least square as a linear algebra problem

It turns out that the least square problem can be formulated as a matrix problem. (Two great explanations see linear regression by matrices[3], and R^2-squared[4].) If we define a matrix $\mathbf{X}$ containing the observations x_i as:

$$\mathbf{X} = \begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ \vdots & \vdots \\ 1 & x_{N-1} \end{pmatrix}. \tag{59}$$

We introduce a vector containing all the response $\mathbf{y}$, and the regression coefficients $\mathbf{a} = (a_0, a_1)$. Then we can write equation (51) as a matrix equation:

$$S = (\mathbf{y} - \mathbf{X} \cdot \mathbf{a})^T (\mathbf{y} - \mathbf{X} \cdot \mathbf{a}). \tag{60}$$

Note that this equation can easily be extended to more than one observation variable x_i. By simply differentiating equation (60) with respect to $\mathbf{a}$, we can show that the derivative has a minimum when (see proof below):

$$\mathbf{X}^T \mathbf{X} \mathbf{a} = \mathbf{X}^T \mathbf{y} \tag{61}$$

Below is a python implementation of equation (61).

```
def OLSM(x, y):
    # returns regression coefficients
    # in ordinary least square using solve function
    # x: observations
    # y: response

    XT = np.array([np.ones(len(x)),x],float)
    X  = np.transpose(XT)
    B = np.dot(XT,X)
    C = np.dot(XT,y)
    return solve(B,C)
```

[3] https://medium.com/@andrew.chamberlain/the-linear-algebra-view-of-least-squares-regression-f67044b7f39b

[4] https://medium.com/@andrew.chamberlain/a-more-elegant-view-of-r-squared-a0a14c177dc3

5.3 Working with matrices on component form

Whenever you want to do some manipulation with matrices, it is very useful to simply write them on component form. If we multiply two matrices $\mathbf{A}$ and $\mathbf{B}$ to form a new matrix $\mathbf{C}$, the components of the new matrix is simply $\mathbf{C}_{ij} = \sum_k \mathbf{A}_{ik}\mathbf{B}_{kj}$. The strength of doing this is that the elements of a matrix, e.g. $\mathbf{A}_{ik}$ are *numbers*, and we can move them around. Proving that e.g. $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$ is straight forward using the component form. The transpose of a matrix is simply to exchange columns and rows, hence $\mathbf{C}^T_{ij} = \mathbf{C}_{ji}$

$$\mathbf{C}^T_{ij} = \mathbf{C}_{ji} = \sum_k \mathbf{A}_{jk}\mathbf{B}_{ki} = \sum_k \mathbf{B}^T_{ik}\mathbf{A}^T_{kj} = (\mathbf{B}^T\mathbf{A}^T)_{ij}, \tag{62}$$

thus $\mathbf{C}^T = \mathbf{B}^T\mathbf{A}^T$. To derive equation (61), we need to take the derivative of equation (61) with respect to $\mathbf{a}$. What we mean by this is that we want to evaluate $\partial S/\partial a_k$ for all the components of $\mathbf{a}$. A useful rule is $\partial a_i/\partial a_k = \delta_{ik}$, where δ_{ik} is the Kronecker delta, it takes the value of one if $i = k$ and zero otherwise. We can write $S = \mathbf{y}^T\mathbf{y} - \mathbf{y}\mathbf{X}\cdot\mathbf{a} - (\mathbf{X}\cdot\mathbf{a})^T\mathbf{y} - (\mathbf{X}\cdot\mathbf{a})^T\mathbf{X}\cdot\mathbf{a}$. All terms that do not contain $\mathbf{a}$ are zero, thus we only need to evaluate the following terms

$$\begin{aligned}
\frac{\partial}{a_k}(\mathbf{X}\cdot\mathbf{a})^T\mathbf{y} &= \frac{\partial}{a_k}(\mathbf{a}^T\cdot\mathbf{X}^T\mathbf{y}) = \frac{\partial}{a_k}\sum_{ij}\mathbf{a}^T_i\mathbf{X}^T_{ij}\mathbf{y}_j = \sum_{ij}\delta_{ik}\mathbf{X}^T_{ij}\mathbf{y}_j \\
&= \sum_j \mathbf{X}^T_{kj}\mathbf{y}_j = \mathbf{X}^T\mathbf{y} \qquad (63)\\
\frac{\partial}{a_k}\mathbf{y}^T\mathbf{X}\cdot\mathbf{a} &= \frac{\partial}{a_k}\sum_{ij}\mathbf{y}^T_i\mathbf{X}_{ij}\mathbf{a}_j = \sum_{ij}\mathbf{y}^T_i\mathbf{X}_{ij}\delta_{jk} = \sum_j \mathbf{y}^T_i\mathbf{X}_{ik} \\
&= \sum_j \mathbf{y}^T_i\mathbf{X}^T_{ki} = \mathbf{X}^T\mathbf{y} \qquad (64)\\
\frac{\partial}{a_k}(\mathbf{X}\cdot\mathbf{a})^T\mathbf{X}\cdot\mathbf{a} &= \frac{\partial}{a_k}\sum_{ijl}\mathbf{a}^T_i\mathbf{X}^T_{ij}\mathbf{X}_{jl}\mathbf{a}_l = \sum_{ijl}(\delta_{ik}\mathbf{X}^T_{ij}\mathbf{X}_{jl}\mathbf{a}_l + \mathbf{a}^T_i\mathbf{X}^T_{ij}\mathbf{X}_{jl}\delta_{lk}) \\
&= \sum_{jl}\mathbf{X}^T_{kj}\mathbf{X}_{jl}\mathbf{a}_l + \sum_{ij}\mathbf{a}^T_i\mathbf{X}^T_{ij}\mathbf{X}_{jk} \\
&= \mathbf{X}^T\mathbf{X}\mathbf{a} + \sum_{ij}\mathbf{X}^T_{kj}\mathbf{X}_{ji}\mathbf{a}_i = 2\mathbf{X}^T\mathbf{X}\mathbf{a}. \qquad (65)
\end{aligned}$$

It then follows that $\partial S/\partial\mathbf{a} = 0$ when

$$\mathbf{X}^T\mathbf{X}\mathbf{a} = \mathbf{X}^T\mathbf{y}. \tag{66}$$

6 Sparse matrices and Thomas algorithm

In many practical examples, such as solving partial differential equations the matrices could be quite large and also contain a lot of zeros. A very important

class of such matrices are *banded matrices* this is a type of *sparse matrices* containing a lot of zero elements, and the non-zero elements are confined to diagonal bands. In the following we will focus on one important type of sparse matrix the tridiagonal. In the next section we will show how it enters naturally in solving the heat equation. It turns out that solving banded matrices is quite simple, and can be coded quite efficiently. As with the Gauss-Jordan example, lets consider a concrete example:

$$\left(\begin{array}{ccccc|c} b_0 & c_0 & 0 & 0 & 0 & r_0 \\ a_1 & b_1 & c_1 & 0 & 0 & r_1 \\ 0 & a_2 & b_2 & c_2 & 0 & r_2 \\ 0 & 0 & a_3 & b_3 & c_3 & r_3 \\ 0 & 0 & 0 & a_4 & b_4 & r_4 \end{array}\right) \tag{67}$$

The right hand side is represented with r_i. The first Gauss-Jordan step is simply to divide by b_0, then we multiply with $-a_1$ and add to second row:

$$\rightarrow \left(\begin{array}{ccccc|c} 1 & c'_0 & 0 & 0 & 0 & r'_0 \\ 0 & b_1 - a_1 c'_0 & c_1 & 0 & 0 & r_1 - a_0 r'_0 \\ 0 & a_2 & b_2 & c_2 & 0 & r_2 \\ 0 & 0 & a_3 & b_3 & c_3 & r_3 \\ 0 & 0 & 0 & a_4 & b_4 & r_4 \end{array}\right), \tag{68}$$

Note that we have introduced some new symbols to simplify the notation: $c'_0 = c_0/b_0$ and $r'_0 = r_0/b_0$. Then we divide by $b_1 - a_1 c'_0$:

$$\left(\begin{array}{ccccc|c} 1 & c'_0 & 0 & 0 & 0 & r'_0 \\ 0 & 1 & c'_1 & 0 & 0 & r'_1 \\ 0 & a_2 & b_2 & c_2 & 0 & r_2 \\ 0 & 0 & a_3 & b_3 & c_3 & r_3 \\ 0 & 0 & 0 & a_4 & b_4 & r_4 \end{array}\right), \tag{69}$$

where $c'_1 = c_1/(b_1 - a_1 c'_0)$ and $r'_1 = (r_1 - a_0 r'_0)/(b_1 - a_1 c'_0)$. If you continue in this manner, you can easily convince yourself that to transform a tridiagonal matrix to the following form:

$$\rightarrow \left(\begin{array}{ccccc|c} 1 & c'_0 & 0 & 0 & 0 & r'_0 \\ 0 & 1 & c'_1 & 0 & 0 & r'_1 \\ 0 & 0 & 1 & c'_2 & 0 & r'_2 \\ 0 & 0 & 0 & 1 & c'_3 & r'_3 \\ 0 & 0 & 0 & 0 & 1 & r'_4 \end{array}\right), \tag{70}$$

where:

$$c'_0 = \frac{c_0}{b_0} \qquad r'_0 = \frac{r_0}{b_0} \tag{71}$$

$$c'_i = \frac{c_i}{b_i - a_i c'_{i-1}} \qquad r'_i = \frac{r_i - a_i r'_{i-1}}{b_i - a_i c'_{i-1}} \quad \text{, for } i = 1, 2, \ldots, N-1 \tag{72}$$

Note that we where able to reduce the tridiagonal matrix to an *upper triangular* matrix in only *one* Gauss-Jordan step. This equation can readily be solved using back-substitution, which can also be simplified as there are a lot of zeros in the upper part. Let us denote the unknowns x_i as we did for the Gauss-Jordan case, now we can find the solution as follows:

$$x_{N-1} = r'_{N-1} \tag{73}$$

$$x_i = r'_i - x_{i+1}c'_i \quad \text{, for } i = N-2, N-3, \ldots, 0 \tag{74}$$

Equation (71), (72), (73) and (74) is known as the Thomas algorithm after Llewellyn Thomas.

Notice.

Clearly tridiagonal matrices can be solved much more efficiently with the Thomas algorithm than using a standard library, such as LU-decomposition. This is because the solution method takes advantages of the *symmetry* of the problem. We will not show it here, but it can be shown that the Thomas algorithm is stable whenever $|b_i| \geq |a_i| + |c_i|$. If the algorithm fails, an advice is first to use the standard `solve` function in python. If this gives a solution, then *pivoting* combined with the Thomas algorithm might do the trick.

7 Example: Solving the heat equation using linear algebra

Exercise 1: Conservation Equation or the Continuity Equation

In figure 5, the continuity equation is derived for heat flow.

Heat equation for solids. As derived in the beginning of this chapter the heat equation for a solid is

$$\frac{d^2T}{dx^2} + \frac{\dot{\sigma}}{k} = \frac{\rho c_p}{k}\frac{dT}{dt}, \tag{75}$$

where $\dot{\sigma}$ is the rate of heat generation in the solid. This equation can be used as a starting point for many interesting models. In this exercise we will investigate the *steady state* solution, *steady state* is just a fancy way of expressing that we want the solution that *does not change with time*. This is achieved by ignoring the derivative with respect to time in equation (75). We want to study a system with size L, and is it good practice to introduce a dimensionless variable: $y = x/L$.

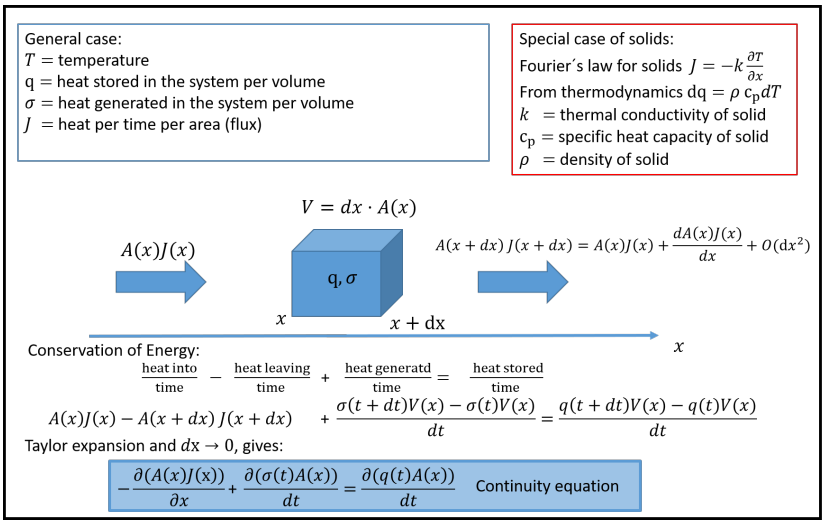

Figure 5: Conservation of energy and the continuity equation.

Part 1. Show that equation (75) now takes the following form:

$$\frac{d^2T}{dy^2} + \frac{\dot{\sigma}L^2}{k} = 0 \tag{76}$$

Exercise 2: Curing of Concrete and Matrix Formulation

Curing of concrete is one particular example that we can investigate with equation (76). When concrete is curing, there are a lot of chemical reactions happening, these reactions generate heat. This is a known issue, and if the temperature rises too much compared to the surroundings, the concrete may fracture. In the following we will, for simplicity, assume that the rate of heat generated during curing is constant, $\dot{\sigma}$ =100 W/m^3. The left end (at $x = 0$) is insulated, meaning that there is no flow of heat over that boundary, hence $dT/dx = 0$ at $x = 0$. On the right hand side the temperature is kept constant, $x(L) = y(1) = T_1$, assumed to be equal to the ambient temperature of $T_1 = 25$°C. The concrete thermal conductivity is assumed to be $k = 1.65$ W/m°C.

Part 1. Show that the solution to equation (76) in this case is:

$$T(y) = \frac{\dot{\sigma}L^2}{2k}(1 - y^2) + T_1. \tag{77}$$

Part 2. In order to solve equation (76) numerically, we need to discretize it. Show that equation (76) now takes the following form:

$$T_{i+1} + T_{i-1} - 2T_i = -h^2\beta, \tag{78}$$

where $\beta = \dot{\sigma} L^2/k$.

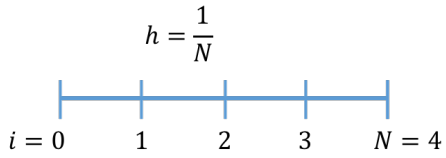

Figure 6: Finite difference grid for $N = 4$.

In figure 6, the finite difference grid is shown for $N = 4$.

Part 3. Show that equation (78) including the boundary conditions for $N = 4$ can be written as the following matrix equation

$$\begin{pmatrix} -\gamma & \gamma & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} T_0 \\ T_1 \\ T_2 \\ T_3 \end{pmatrix} = \begin{pmatrix} -h^2\beta \\ -h^2\beta \\ -h^2\beta \\ -h^2\beta - 25 \end{pmatrix}. \tag{79}$$

where $\gamma = 2$ for the central difference scheme and 1 for the forward difference scheme.

Part 4.

- Solve the set of equations in equation (2) using `numpy.linalg.solve`[5].
- Write the code so that you can easily switch between the central difference scheme and forward difference
- Evaluate the numerical error as you change h, how does it scale? Is it what you expect?

```
import numpy as np
import scipy as sc
import scipy.sparse.linalg
from numpy.linalg import solve
import matplotlib.pyplot as plt
```

```
central_difference=False
# set simulation parameters
h=0.25
L=1.0
n = int(round(L/h))
Tb=25 #rhs
```

[5] https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html

```
sigma=100
k=1.65
beta = sigma*L**2/k

y = np.arange(n+1)*h

def analytical(x):
    return beta*(1-x*x)/2+Tb
def tri_diag(a, b, c, k1=-1, k2=0, k3=1):
    """ a,b,c diagonal terms
        default k-values for 4x4 matrix:
        | b0 c0 0  0 |
        | a0 b1 c1 0 |
        | 0  a1 b2 c2|
        | 0  0  a2 b3|
    """
    return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)
# defina a, b and c vector
a=np.ones(n-1)
b=..
c=..

if central_difference:
    c[0]= ...
else:
    b[0]=...

A=tri_diag(a,b,c)
print(A) # view matrix - compare with N=4 to make sure no bugs
# define rhs vector
d=...
#rhs boundary condition
d[-1]=...

Tn=np.linalg.solve(A,d)
print(Tn)
```

The correct solution for $L = 1$ m, and $h = 1/4$, is: $[T_0, T_1.T_2, T_3]$=[55.3030303 , 53.40909091, 47.72727273, 38.25757576] (central difference) and $[T_0, T_1.T_2, T_3]$=[62.87878788, 59.09090909, 51.51515152, 40.15151515] (forward difference)

Exercise 3: Solve the full heat equation

Part 1. Replace the time derivative in equation (75) with

$$\frac{dT}{dt} \simeq \frac{T(t+\Delta t) - T(t)}{\Delta t} = \frac{T^{n+1} - T^n}{\Delta t}, \tag{80}$$

and show that by using an *implicit formulation* (i.e. that the second derivative with respect to x is to be evaluated at $T(t+\Delta t) \equiv T^{n+1}$) that equation (75)

can be written

$$T_{i+1}^{n+1} + T_{i-1}^{n+1} - (2 + \frac{\alpha h^2}{\Delta t})T_i^{n+1} = -h^2\beta - \frac{\alpha h^2}{\Delta t}T_i^n, \tag{81}$$

where $\alpha \equiv \rho c_p/k$.

Part 2. Use the central difference formulation for the boundary condition and show that for four nodes we can formulate equation (81) as the following matrix equation

$$\begin{pmatrix} -(2+\frac{\alpha h^2}{\Delta t}) & 2 & 0 & 0 \\ 1 & -(2+\frac{\alpha h^2}{\Delta t}) & 1 & 0 \\ 0 & 1 & -(2+\frac{\alpha h^2}{\Delta t}) & 1 \\ 0 & 0 & 1 & -(2+\frac{\alpha h^2}{\Delta t}) \end{pmatrix} \begin{pmatrix} T_0^{n+1} \\ T_1^{n+1} \\ T_2^{n+1} \\ T_3^{n+1} \end{pmatrix}$$
$$= \begin{pmatrix} -h^2\beta \\ -h^2\beta \\ -h^2\beta \\ -h^2\beta - 25 \end{pmatrix} - \frac{\alpha h^2}{\Delta t} \begin{pmatrix} T_0^n \\ T_1^n \\ T_2^n \\ T_3^n \end{pmatrix} \tag{82}$$

Part 3. Assume that the initial temperature in the concrete is 25°C, ρ=2400 kg/m^3, a specific heat capacity $c_p = 1000$ W/kg K, and a time step of $\Delta t = 86400$ s (1 day). Solve equation (3), plot the result each day and compare the result after 50 days with the steady state solution in equation (77).

Exercise 4: Using sparse matrices in python

In this part we are going to create a sparse matrix in python and use `scipy.sparse.linalg.spsolve` to solve it. The matrix is created using `scipy.sparse.spdiags`.

Part 1. Extend the code you developed in the last exercises to also be able to use sparse matrices, by e.g. a logical switch. Sparse matrices may be defined as follows

```
import scipy.sparse.linalg

#right hand side
# rhs vector
d=np.repeat(-h*h*beta,n)
#rhs - constant temperature
Tb=25
d[-1]=d[-1]-Tb
#Set up sparse matrix
diagonals=np.zeros((3,n))
diagonals[0,:]= 1
diagonals[1,:]= -2
```

```
diagonals[2,:]= 1
#No flux boundary condition
diagonals[2,1]= 2
A_sparse = sc.sparse.spdiags(diagonals, [-1,0,1], n, n,format='csc')
# to view matrix - do this and check that it is correct!
print(A_sparse.todense())
# solve matrix
Tb = sc.sparse.linalg.spsolve(A_sparse,d)

# if you like you can use timeit to check the efficiency
# %timeit sc.sparse.linalg.spsolve( ... )
```

- Compare the sparse solver with the standard Numpy solver using `%timeit`, how large must the linear system be before an improvement in speed is seen?

8 CO_2 diffusion into aquifers

The transport of CO_2 into aquifers can be described according to the diffusion equation

$$\frac{\partial C(z,t)}{\partial t} = \frac{\partial}{\partial z}\left(K(z)\frac{\partial C(z,t)}{\partial z}\right), \tag{83}$$

where $C(z,t)$ is the concentration of CO_2 as a function of depth (z) and time t, and $K(z)$ is the diffusion constant of CO_2 as a function of depth. This equation can be discretized using standard techniques, to help in that respect consider figure 7.

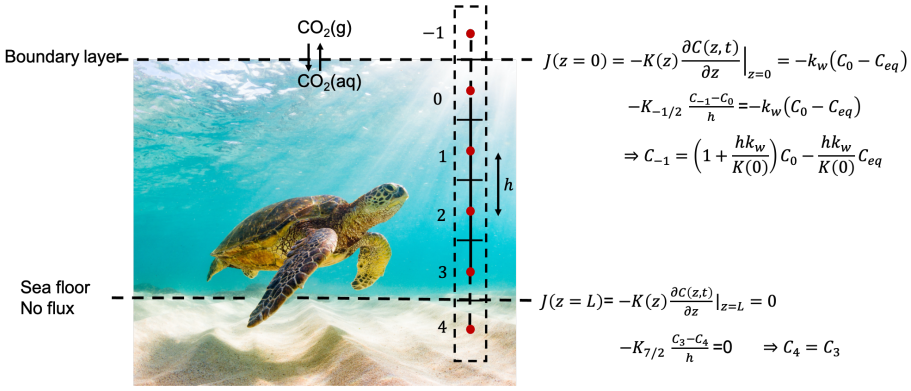

Figure 7: Discretization for diffusion of CO_2 into an aquifer, including boundary conditions.

In the following we will assume that there are only four nodes ($i = 0 \ldots 3$) in the physical domain, and two ghost nodes $i = -1$, and $i = 4$. There are many ways to attack this problem, but in the following we will borrow ideas from Finite Volume. Finite volume methods is a way of discretizing equations such that we *conserve mass*. The diffusion equation as it is derived in figure 5, express that the flux of something (heat, particles, etc) leaving the box surface minus the flux entering the surface of the box is equal to the rate of change of something inside the box. We can formulate this mathematically as:

$$\frac{\partial C(z,t)}{\partial t} \simeq \frac{1}{h}\left[K(z)\frac{\partial C(z,t)}{\partial z}\bigg|_{i+1/2} - K(z)\frac{\partial C(z,t)}{\partial z}\bigg|_{i-1/2}\right] \tag{84}$$

The notation $i \pm 1/2$, means that the flux is to be evaluated *at the surface* of the box (i.e. halfway between the red dots in figure 7). $K(z)$ is the diffusion constant, and it is known everywhere, so this is simple to evaluate at the surface. The concentrations are only known at the center of each box, the red dots in figure 7. The derivative of the concentration can be evaluated using the central difference formula (remember that the distance between the red dot and edge of the box is $h/2$), hence

$$\frac{C_i^{n+1} - C_i^n}{\Delta t} = \frac{1}{h}\left[K_{i+1/2}\frac{C_{i+1} - C_i}{h} - K_{i-1/2}\frac{C_i - C_{i-1}}{h}\right], \tag{85}$$

notice that we have discretized the time derivative, and that we have introduced n to indicate the time step. On the right hand side there are is no time indicated, it turns out that we have a choice to put time step n or $n+1$ on the concentrations on the right hand side. If we put n the scheme is said to be explicit, if we put $n+1$, the scheme is implicit. Implicit schemes are stable compared to explicit schemes, whereas explicit schemes has slightly higher numerical accuracy . In general we can write

TO DO 1: show this!

$$\frac{C_i^{n+1} - C_i^n}{\Delta t} = \frac{\theta}{h}\left[K_{i+1/2}\frac{C_{i+1}^n - C_i^n}{h} - K_{i-1/2}\frac{C_i^n - C_{i-1}^n}{h}\right] \tag{86}$$

$$+ \frac{1-\theta}{h}\left[K_{i+1/2}\frac{C_{i+1}^{n+1} - C_i^{n+1}}{h} - K_{i-1/2}\frac{C_i^{n+1} - C_{i-1}^{n+1}}{h}\right], \tag{87}$$

hence if $\theta = 1$ the scheme is explicit, if $\theta = 0$ the scheme is implicit, and if $\theta = 1/2$, the scheme is called the Crank-Nicolson method. The first and last boundary are special, let us first consider the $i = 0$, this is where the sea is in contact with the CO_2 in the atmosphere, and the flux is $k_w(C_0 - C_{eq})$, hence

$$\frac{C_0^{n+1} - C_0^n}{\Delta t} = \frac{\theta}{h}\left[K_{i+1/2}\frac{C_1^n - C_0^n}{h} - k_w(C_0^n - C_{eq}^n)\right] \tag{88}$$

$$+ \frac{1-\theta}{h}\left[K_{i+1/2}\frac{C_1^{n+1} - C_0^{n+1}}{h} - k_w(C_0^{n+1} - C_{eq}^{n+1})\right]. \tag{89}$$

For the last block the flux is zero towards the seafloor, and equation (87) can be written

$$\frac{C_3^{n+1} - C_3^n}{\Delta t} = \frac{\theta}{h}\left[-K_{5/2}\frac{C_3^n - C_2^n}{h}\right] \tag{90}$$

$$+ \frac{1-\theta}{h}\left[-K_{5/2}\frac{C_3^{n+1} - C_2^{n+1}}{h}\right]. \tag{91}$$

For the blocks $i = 1 \ldots 2$, we can collect all terms with $n+1$ on one side and terms with n on the other side and rewrite equation (87)

$$\begin{aligned}
&\left[1 + (1-\theta)\ \alpha(K_{i+1/2} + K_{i-1/2})\right] C_i^{n+1} \\
&\qquad - (1-\theta)\alpha K_{i+1/2} C_{i+1}^{n+1} - (1-\theta)\alpha K_{i-1/2} C_{i-1}^{n+1} \\
&= \left[1 - \theta\ \alpha(K_{i+1/2} + K_{i-1/2})\right] C_i^n \\
&\qquad + \theta\alpha K_{i+1/2} C_{i+1}^n + \theta\alpha K_{i-1/2} C_{i-1}^n,
\end{aligned} \tag{92}$$

where $\alpha \equiv \Delta t/h^2$. Next, we want to write down the corresponding matrix equations for four grid nodes as indicated in figure 7. Notice that we need to use the equations in figure 7, for C_{-1}, and C_4. The left and right hand coefficient matrix $\mathbf{L}$, and $\mathbf{R}$ are given as

$$\begin{pmatrix}
1 + (1-\theta)\alpha(K_{1/2} + hk_w) & -(1-\theta)\alpha K_{1/2} & 0 & 0 \\
-(1-\theta)\alpha K_{1/2} & 1 + (1-\theta)\alpha(K_{3/2} + K_{1/2}) & -(1-\theta)\alpha K_{3/2} & 0 \\
0 & -(1-\theta)\alpha K_{3/2} & 1 + (1-\theta)\alpha(K_{5/2} + K_{3/2}) & -(1-\theta)\alpha K_{5/2} \\
0 & 0 & -(1-\theta)\alpha K_{5/2} & 1 + (1-\theta)\alpha K_{5/2}
\end{pmatrix},$$

$$\begin{pmatrix}
1 - \theta\alpha(K_{1/2} + hk_w) & +\theta\alpha K_{1/2} & 0 & 0 \\
\theta\alpha K_{1/2} & 1 - \theta\alpha(K_{3/2} + K_{1/2}) & \theta\alpha K_{3/2} & 0 \\
0 & \theta\alpha K_{3/2} & 1 - \theta\alpha(K_{5/2} + K_{3/2}) & \theta\alpha K_{5/2} \\
0 & 0 & \theta\alpha K_{5/2} & 1 - \theta\alpha K_{5/2}
\end{pmatrix},$$

respectively. Introducing $\mathbf{S} = [k_w C_{eq} \Delta t/h, 0, 0, 0]^T$, we can finally write the diffusion equation (83) as

$$\mathbf{L}\mathbf{C}^{n+1} = \mathbf{R}\mathbf{C}^n + \theta\mathbf{S}^n + (1-\theta)\mathbf{S}^{n+1} \tag{93}$$

More stuff to do:

1. Assume zero flux over the air water interface (k_w=0), show from the equations above that if we start with a uniform concentration in the sea ($\mathbf{C}^n$=constant) that $\mathbf{C}^{n+1}$ does not change (as it should).

2. Assume that if the concentration at a specific time n in the sea is equal to $\mathbf{C}_{eq}$ then the concentration stays constant at all later times

3. Add chemical reactions

References

[1] Randall J. LeVeque et al. *Finite Volume Methods for Hyperbolic Problems*, volume 31. Cambridge University Press, 2002.

[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: the Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[3] William H. Press, William T. Vetterling, Saul A. Teukolsky, and Brian P. Flannery. *Numerical Recipes in C++: the Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 2nd edition, 2002.

[4] Josef Stoer and Roland Bulirsch. *Introduction to Numerical Analysis*, volume 12. Springer Science & Business Media, 2013.

[5] Gilbert Strang. *Linear Algebra and Learning From Data*. Wellesley-Cambridge Press, 2019.

[6] Lloyd N. Trefethen and David Bau III. *Numerical Linear Algebra*, volume 50. SIAM, 1997.

Index