Solving linear systems

Aksel Hiorth, the National IOR Centre & Institute for Energy Resources,

University of Stavanger

Sep 23, 2020

Contents

Solving systems of equations are one of the most common tasks that we use computers for within modeling. A typical task is that we have a model that contains a set of unknown parameters which we want to determine. To determine these parameters we need to solve a set of equations. In many cases

these equations are nonlinear, but often a nonlinear problem is solved *by linearize* the nonlinear equations, and thereby reducing it to a sequence of linear algebra problems. Thus 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 library. In this course we will only cover the theory behind numerical linear algebra superficially, and the main purpose is to shed some light on some of the challenges one might encounter solving linear systems. In particular it is important for you to understand when it is stated in the NumPy documentation that the standard linear solver: `solve` function uses *LU-decomposition* and *partial pivoting*.

1 Solving linear equations

There are a number of excellent books covering this topic, see e.g. [1, 5, 3, 4]. 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:

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

These equations can be written in matrix form as:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}, \tag{5}$$

where:

$$\mathbf{A} \equiv \begin{pmatrix} 2 & 1 & 1 & 3 \\ 1 & 1 & 3 & 1 \\ 1 & 4 & 1 & 1 \\ 1 & 1 & 2 & 2 \end{pmatrix} \qquad \mathbf{b} \equiv \begin{pmatrix} 1 \\ -3 \\ 2 \\ 1 \end{pmatrix} \qquad \mathbf{x} \equiv \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}. \tag{6}$$

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 (4) 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 [1].

1.1 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 [1] 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 (5). 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 (6) as:

$$\left(\begin{array}{cccc|c} 2 & 1 & 1 & 3 & 1 \\ 1 & 1 & 3 & 1 & -3 \\ 1 & 4 & 1 & 1 & 2 \\ 1 & 1 & 2 & 2 & 1 \end{array}\right). \tag{7}$$

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:

$$\left(\begin{array}{cccc|c} 2 & 1 & 1 & 3 & 1 \\ 1 & 1 & 3 & 1 & -3 \\ 1 & 4 & 1 & 1 & 2 \\ 1 & 1 & 2 & 2 & 1 \end{array}\right) \rightarrow \left(\begin{array}{cccc|c} 2 & 1 & 1 & 3 & 1 \\ 0 & 1/2 & 5/2 & -1/2 & -7/2 \\ 0 & 7/2 & 1/2 & -1/2 & 3/2 \\ 0 & 1/2 & 3/2 & 1/2 & 1/2 \end{array}\right) \rightarrow \tag{8}$$

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

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 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
```

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:

$$\begin{pmatrix} a'_{0,0} & a'_{0,1} & a'_{0,2} & a'_{0,3} \\ 0 & a'_{1,1} & a'_{1,2} & a'_{1,3} \\ 0 & 0 & a'_{2,2} & a'_{2,3} \\ 0 & 0 & 0 & a'_{3,3} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \end{pmatrix} \tag{9}$$

The backsubstitution can then be written formally as:

$$x_i = \frac{1}{a'_{ii}} \left[b'_i - \sum_{j=i+1}^{N-1} a'_{ij} x_j \right], \quad i = N-1, N-2, \ldots, 0 \tag{10}$$

The backsubstitution can now easily be implemented in Python as:

```
# Backsubstitution
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:

```
# Backsubstitution - 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]
```

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 (8).

- 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 (2) 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.

1.2 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 [1]. 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*.

1.3 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 backsubstitution. Thus if it was possible to decompose the matrix $\mathbf{A}$ as follows:

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U} \tag{11}$$

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

The solution procedure would then be to rewrite equation (5) as:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{L} \cdot \mathbf{U} \cdot \mathbf{x} = \mathbf{b}, \tag{12}$$

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

$$\mathbf{y} \equiv \mathbf{U} \cdot \mathbf{x}, \tag{13}$$

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

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b}, \tag{14}$$

and then for $\mathbf{x}$:

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y}. \tag{15}$$

Note that the solution to equation (14) would be done by *forward substitution*:

$$y_i = \frac{1}{l_{ii}} \left[b_i - \sum_{j=0}^{i-1} l_{ij} x_j \right], \quad i = 1, 2, \ldots N-1. \tag{16}$$

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 (12). 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`, 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)
```

2 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.

2.1 Iterative improvement

The first method [2] assumes that we already have solved the matrix equation (5), 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

$$\mathbf{A} \cdot \hat{\mathbf{x}} = \mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}, \tag{17}$$

subtracting equation (5) we get

$$\mathbf{A} \cdot \delta\mathbf{x} = \delta\mathbf{b}. \tag{18}$$

Solving equation (17) for $\delta\mathbf{b}$ an inserting in the equation above, we get

$$\mathbf{A} \cdot \delta\mathbf{x} = \mathbf{A} \cdot \hat{\mathbf{x}} - \mathbf{b}. \tag{19}$$

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 $\hat{\mathbf{x}}$.

2.2 The Jacobi method

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

$$\mathbf{A} = \mathbf{D} + \mathbf{R} \tag{20}$$

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

We can then write equation (5) as

$$\mathbf{D}\mathbf{x} = \mathbf{b} - \mathbf{R} \cdot \mathbf{x}. \tag{21}$$

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 $a_{ii} \to 1/a_{ii}$. 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 (21) to calculate the next solution $\mathbf{x}^{k+1}$, and so on

$$\mathbf{x}^{k+1} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R} \cdot \mathbf{x}^k). \tag{22}$$

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

$$x_0 = \frac{1}{a_{00}}(b_0 - a_{01}x_1^k - a_{02}x_2^k - a_{03}x_3^k), \tag{23}$$

$$x_1 = \frac{1}{a_{11}}(b_0 - a_{00}x_0^k - a_{02}x_2^k - a_{03}x_3^k), \tag{24}$$

$$x_2 = \frac{1}{a_{22}}(b_0 - a_{00}x_0^k - a_{01}x_1^k - a_{03}x_3^k), \tag{25}$$

$$x_3 = \frac{1}{a_{33}}(b_0 - a_{00}x_0^k - a_{01}x_1^k - a_{02}x_2^k). \tag{26}$$

Below is a Python implementation

```
def solve_jacobi(A,b,x=np.zeros(len(b)),max_iter=1000,EPS=1e-6):
    """
    Solves the linear system Ax=b using the jacobi method, stops if
    solution is not found after max_iter or if solution changes less
    than EPS
    """
    D=np.diag(A)
    R=A-np.diag(D)
    eps=1
    x_old=x
    iter=0
    while(eps>EPS and iter<max_iter):
        iter+=1
        x=(b-np.dot(R,x_old))/D
        eps=np.abs(np.sum(x-x_old))
        x_old=x
    print('found solution after ' + str(iter) +' iterations')
    return x
```

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* $\rho(\mathbf{P}) < 1$.

2.3 The Gauss-Seidel method

It is tempting in equation (23) to use our estimate of x_0^{k+1} in the next equation, equation (24), 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=np.zeros(len(b)),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
    """
    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]=w*(b[i]- np.dot(R[i,:],x))/D[i]
            eps+=np.abs(tmp-x[i])
    print('found solution after ' + str(iter) +' iterations')
    return x
```

3 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 1.

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:

$$S = \sum_{i=0}^{N-1} (y_i - f(x_i))^2 = \sum_{i=0}^{N-1} (y_i - a_0 - a_1 x_i)^2. \tag{27}$$

(For the example in figure 1, $N = 5$.) This is the idea behind *least square*, and linear regression. One thing you should be aware of is that points lying far from

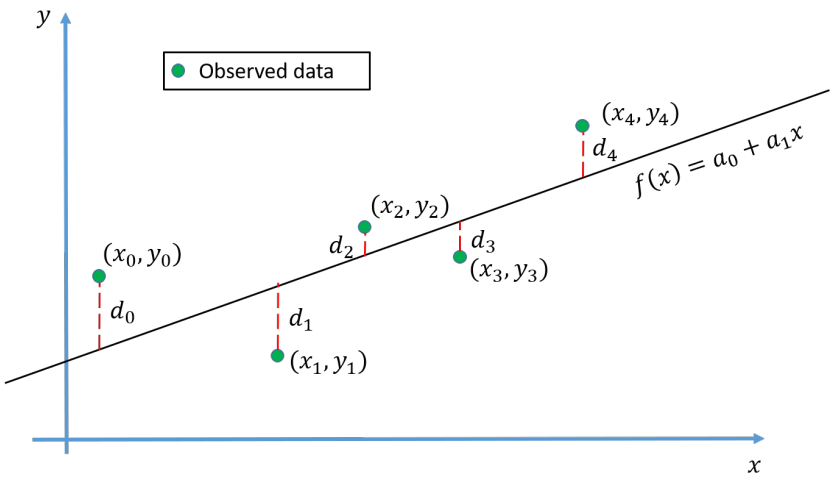

Figure 1: Linear regression by minimizing the total distance to all the points.

the line will contribute more to equation (27). The underlying assumption is that each data point provides equally precise information about the process, this 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:

$$S = \sum_{i=0}^{N-1} \omega_i (y_i - f(x_i))^2 = \sum_{i=0}^{3} \omega_i (y_i - a_0 - a_1 x_i)^2, \tag{28}$$

ω_i is a weight factor.

3.1 Solving least square, using algebraic equations

Let us continue with equation (27), the algebraic solution is to simply find the value of a_0 and a_1 that minimizes S:

$$\frac{\partial S}{\partial a_0} = -2 \sum_{i=0}^{N-1} (y_i - a_0 - a_1 x_i) = 0, \tag{29}$$

$$\frac{\partial S}{\partial a_1} = -2 \sum_{i=0}^{N-1} (y_i - a_0 - a_1 x_i) x_i = 0. \tag{30}$$

Defining the mean value as $\overline{x} = \sum_i x_i/N$ and $\overline{y} = \sum_i y_i/N$, we can write equation (29) and (30) as:

$$\sum_{i=0}^{N-1}(y_i - a_0 - a_1 x_i) = N\overline{y} - a_0 N - a_1 N\overline{x} = 0, \tag{31}$$

$$\sum_{i=0}^{N-1}(y_i - a_0 - a_1 x_i)x_i = \sum_i y_i x_i - a_0 N\overline{x} - a_1 \sum_i x_i x_i = 0. \tag{32}$$

Solving equation (31) with respect to a_0, and inserting the expression into equation (32), we find:

$$a_0 = \overline{y} - a_1\overline{x}, \tag{33}$$

$$a_1 = \frac{\sum_i y_i x_i - N\overline{x}\overline{y}}{\sum_i x_i^2 - N\overline{x}^2} = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{\sum_i (x_i - \overline{x})^2}. \tag{34}$$

We leave it as an exercise to show the last expression for a_1. Clearly the equation (34) 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 1. 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:

$$R^2 = 1 - \frac{\sum_{i=0}^{N-1}(y_i - f(x_i))^2}{\sum_{i=0}^{N-1}(y_i - \overline{y})^2}. \tag{35}$$

In python we can implement equation (33), (34) and (35) 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)
```

3.2 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, and R^2-squared.) If we define a matrix $\mathbf{X}$ containing the observations x_i as:

$$\mathbf{X} = \begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ \vdots & \vdots \\ 1 & x_{N-1} \end{pmatrix}. \tag{36}$$

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 (28) as a matrix equation:

$$S = (\mathbf{y} - \mathbf{X} \cdot \mathbf{a})^T (\mathbf{y} - \mathbf{X} \cdot \mathbf{a}). \tag{37}$$

Note that this equation can easily be extended to more than one observation variable x_i. By simply differentiating equation (37) with respect to $\mathbf{a}$, we can show that the derivative has a minimum when (see proof below):

$$\mathbf{X}^T \mathbf{X} \mathbf{a} = \mathbf{X}^T \mathbf{y} \tag{38}$$

Below is a python implementation of equation (38).

```
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)
```

3.3 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 $\mathbf{C}^T_{ij} = \mathbf{C}_{ji}$

$$\mathbf{C}^T_{ij} = \mathbf{C}_{ji} = \sum_k \mathbf{A}_{jk}\mathbf{B}_{ki} = \sum_k \mathbf{B}^T_{ik}\mathbf{A}^T_{kj} = (\mathbf{B}^T\mathbf{A}^T)_{ij}, \tag{39}$$

thus $\mathbf{C}^T = \mathbf{B}^T\mathbf{A}^T$. To derive equation (38), we need to take the derivative of equation (38) 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

$$\begin{aligned}\frac{\partial}{a_k}(\mathbf{X}\cdot\mathbf{a})^T\mathbf{y} &= \frac{\partial}{a_k}(\mathbf{a}^T\cdot\mathbf{X}^T\mathbf{y}) = \frac{\partial}{a_k}\sum_{ij}\mathbf{a}_i^T\mathbf{X}_{ij}^T\mathbf{y}_j = \sum_{ij}\delta_{ik}\mathbf{X}_{ij}^T\mathbf{y}_j \\ &= \sum_j \mathbf{X}_{kj}^T\mathbf{y}_j = \mathbf{X}^T\mathbf{y} \qquad (40)\\ \frac{\partial}{a_k}\mathbf{y}^T\mathbf{X}\cdot\mathbf{a} &= \frac{\partial}{a_k}\sum_{ij}\mathbf{y}_i^T\mathbf{X}_{ij}\mathbf{a}_j = \sum_{ij}\mathbf{y}_i^T\mathbf{X}_{ij}\delta_{jk} = \sum_j \mathbf{y}_i^T\mathbf{X}_{ik} \\ &= \sum_j \mathbf{y}_i^T\mathbf{X}_{ki}^T = \mathbf{X}^T\mathbf{y} \qquad (41)\\ \frac{\partial}{a_k}(\mathbf{X}\cdot\mathbf{a})^T\mathbf{X}\cdot\mathbf{a} &= \frac{\partial}{a_k}\sum_{ijl}\mathbf{a}_i^T\mathbf{X}_{ij}^T\mathbf{X}_{jl}\mathbf{a}_l = \sum_{ijl}(\delta_{ik}\mathbf{X}_{ij}^T\mathbf{X}_{jl}\mathbf{a}_l + \mathbf{a}_i^T\mathbf{X}_{ij}^T\mathbf{X}_{jl}\delta_{lk}) \\ &= \sum_{jl}\mathbf{X}_{kj}^T\mathbf{X}_{jl}\mathbf{a}_l + \sum_{ij}\mathbf{a}_i^T\mathbf{X}_{ij}^T\mathbf{X}_{jk} \\ &= \mathbf{X}^T\mathbf{X}\mathbf{a} + \sum_{ij}\mathbf{X}_{kj}^T\mathbf{X}_{ji}\mathbf{a}_i = 2\mathbf{X}^T\mathbf{X}\mathbf{a}. \qquad (42)\end{aligned}$$

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

$$\mathbf{X}^T\mathbf{X}\mathbf{a} = \mathbf{X}^T\mathbf{y}. \qquad (43)$$

4 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:

$$\left(\begin{array}{ccccc|c} b_0 & c_0 & 0 & 0 & 0 & r_0 \\ a_1 & b_1 & c_1 & 0 & 0 & r_1 \\ 0 & a_2 & b_2 & c_2 & 0 & r_2 \\ 0 & 0 & a_3 & b_3 & c_3 & r_3 \\ 0 & 0 & 0 & a_4 & b_4 & r_4 \end{array}\right) \qquad (44)$$

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:

$$\rightarrow \left(\begin{array}{ccccc|c} 1 & c'_0 & 0 & 0 & 0 & r'_0 \\ 0 & b_1 - a_1 c'_0 & c_1 & 0 & 0 & r_1 - a_0 r'_0 \\ 0 & a_2 & b_2 & c_2 & 0 & r_2 \\ 0 & 0 & a_3 & b_3 & c_3 & r_3 \\ 0 & 0 & 0 & a_4 & b_4 & r_4 \end{array}\right), \tag{45}$$

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 $b_1 - a_1 c'_0$:

$$\left(\begin{array}{ccccc|c} 1 & c'_0 & 0 & 0 & 0 & r'_0 \\ 0 & 1 & c'_1 & 0 & 0 & r'_1 \\ 0 & a_2 & b_2 & c_2 & 0 & r_2 \\ 0 & 0 & a_3 & b_3 & c_3 & r_3 \\ 0 & 0 & 0 & a_4 & b_4 & r_4 \end{array}\right), \tag{46}$$

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:

$$\rightarrow \left(\begin{array}{ccccc|c} 1 & c'_0 & 0 & 0 & 0 & r'_0 \\ 0 & 1 & c'_1 & 0 & 0 & r'_1 \\ 0 & 0 & 1 & c'_2 & 0 & r'_2 \\ 0 & 0 & 0 & 1 & c'_3 & r'_3 \\ 0 & 0 & 0 & 0 & 1 & r'_4 \end{array}\right), \tag{47}$$

where:

$$c'_0 = \frac{c_0}{b_0} \qquad r'_0 = r_0 b_0 \tag{48}$$

$$c'_i = \frac{c_i}{b_i - a_i c'_{i-1}} \qquad r'_i = \frac{r_i - a_i r'_{i-1}}{b_i - a_i c'_{i-1}} \quad \text{, for } i = 1, 2, \ldots, N-1 \tag{49}$$

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:

$$x_{N-1} = r'_{N-1} \tag{50}$$

$$x_i = r'_i - x_{i+1} c'_i \quad \text{, for } i = N-2, N-3, \ldots, 0 \tag{51}$$

Equation (48), (49), (50) and (51) 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.

5 Example: Solving the heat equation using linear algebra

Exercise 1: Conservation Equation or the Continuity Equation

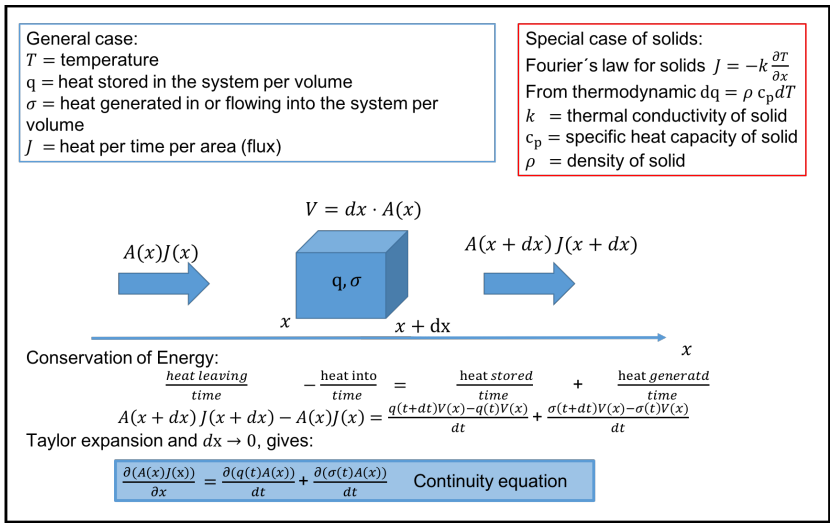

Figure 2: Conservation of energy and the continuity equation.

In figure 2, the continuity equation is derived for heat flow. In the case of heat exchange for a solid, we can show that it can be written:

$$\frac{d^2T}{dx^2} + \frac{\dot{\sigma}}{k} = \frac{\rho c_p}{k}\frac{dT}{dt}, \tag{52}$$

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 (52). We want to study a system with size L, and is it good practice to introduce a dimensionless variable: $y = x/L$. Equation (52) can now be written:

$$\frac{d^2T}{dy^2} + \frac{\dot{\sigma}L^2}{k} = 0 \tag{53}$$

Exercise 2: Curing of Concrete and Matrix Formulation

Curing of concrete is one particular example that we can investigate with equation (53). 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.

We leave it as an exercise to show that the analytical solution to equation (53) in this case is:

$$T(y) = \frac{\dot{\sigma}L^2}{2k}(1 - y^2) + T_1. \tag{54}$$

In order to solve equation (53) numerically, we need to discretize it. We replace the second derivative with $dT/dy^2 = (T(y + dy) + T(y - dy) - 2T(y))/dy^2$. Equation (52) can now be written:

$$T_{i+1} + T_{i-1} - 2T_i = -h^2\beta, \tag{55}$$

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

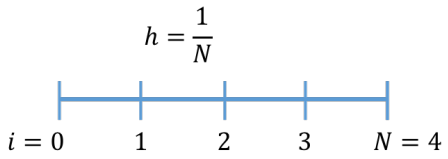

Figure 3: Finite difference grid for $N = 4$.

In figure 3, the finite difference grid is shown for $N = 4$. Let us write down equation (55) for each grid node to see how the implementation is done in

practice:

$$\begin{aligned} T_{-1} + T_1 - 2T_0 &= -h^2\beta, \\ T_0 + T_2 - 2T_1 &= -h^2\beta, \\ T_1 + T_3 - 2T_2 &= -h^2\beta, \\ T_2 + T_4 - 2T_3 &= -h^2\beta. \end{aligned} \tag{56}$$

The tricky part is now to introduce the boundary conditions. The right hand side is easy, because here the temperature is $T_4 = 25$. However, we see that T_{-1} enters and we have no value for this node. The boundary condition on the left hand side is $dT/dy = 0$, by using the central difference for the derivative allows us to write:

$$\left.\frac{dT}{dy}\right|_{y=0} = \frac{T_{-1} - T_1}{2h} = 0, \tag{57}$$

hence $T_{-1} = T_1$. Thus the final set of equations are:

$$\begin{aligned} 2T_1 - 2T_0 &= -h^2\beta, \\ T_0 + T_2 - 2T_1 &= -h^2\beta, \\ T_1 + T_3 - 2T_2 &= -h^2\beta, \\ T_2 + 25 - 2T_3 &= -h^2\beta, \end{aligned} \tag{58}$$

or in matrix form:

$$\begin{pmatrix} -2 & 2 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} T_0 \\ T_1 \\ T_2 \\ T_3 \end{pmatrix} = \begin{pmatrix} -h^2\beta \\ -h^2\beta \\ -h^2\beta \\ -h^2\beta - 25 \end{pmatrix}. \tag{59}$$

Note that it is now easy to increase N as it is only the boundaries that requires special attention.

- Solve the set of equations using `numpy.linalg.solve`.

The correct solution is $L = 1$ m, and $h = 1/4$, is: $[T_0, T_1.T_2, T_3] = [55.3030303, 53.40909091, 47.72727273, 38.25757$

Solution.

Notice.

The solution below implements equation (2) using sparse matrices, and the standard Numpy `solve` function. You can use the `%timeit` magic command in Ipython and Jupyter notebooks to test the efficiency.

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

# Set simulation parameters
h = 0.25             # element size
L = 1.0              # length of domain
n = int(round(L/h))  # number of unknowns
x=np.arange(n+1)*h   # includes right bc
T1=25
sigma = 100*L**2/1.65

def tri_diag(a, b, c, k1=-1, k2=0, k3=1):
    """ a,b,c diagonal terms """
    return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)

def analytical(sigma,x):
    return sigma*(1-x*x)/2+T1

#Create matrix for linalg solver
a=np.ones(n-1)
b=-np.ones(n)*2
c=np.ones(n-1)
#Create matrix for sparse solver
diagonals=np.zeros((3,n))
diagonals[0,:]= 1
diagonals[1,:]= -2
diagonals[2,:]= 1

# rhs vector
d=np.repeat(-h*h*sigma,n)

#----boundary conditions ------
#lhs - no flux of heat
diagonals[2,1]= 2
c[0]=2
#rhs - constant temperature
d[n-1]=d[n-1]-T1
#------------------------------

A=tri_diag(a,b,c)
A_sparse = sc.sparse.spdiags(diagonals, [-1,0,1], n, n,format='csc')
# to view matrix
print(A_sparse.todense())
#Solve linear problems
Ta = solve(A,d,)
Tb = sc.sparse.linalg.spsolve(A_sparse,d)
#Add right boundary node
Ta=np.append(Ta,T1)
Tb=np.append(Tb,T1)
#uncomment to test efficiency
#%timeit sc.sparse.linalg.spsolve(A_sparse,d)
#%timeit solve(A,d,)

# Plot solutions
plt.plot(x,Ta,x,Tb,'-.',x,analytical(sigma,x),':', lw=3)
plt.xlabel("Dimensionless length")
plt.ylabel(r"Temperature [$^\circ$C]")
plt.xlim(0,1)
plt.ylim(T1-1)
```

```
plt.legend(['sparse','linalg','analytical'])
plt.grid()
plt.show()
```

References

[1] 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.

[2] 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.

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

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

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