# Solving linear and nonlinear equations

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

University of Stavanger

Aug 27, 2019

## 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 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*.

After covering some basics of numerical linear algebra, we will shift focus to nonlinear equations. Contrary to linear equations, you will most likely find that the functions available in various Python library will *not* cover your needs and in many cases fail to give you the correct solution. The reason for this is that the solution of a nonlinear equation is greatly dependent on the starting point, and a combination of various techniques must be used.

# 1 Solving linear equations

There are a number of excellent books covering this topic, see e.g. [1, 4, 2, 3]. 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 solution. 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:

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

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

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 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

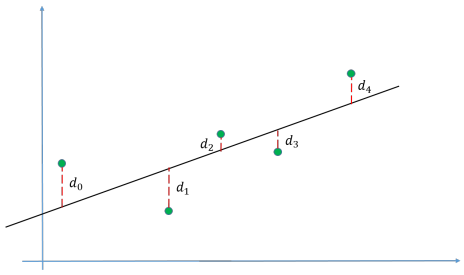


Figure 1: Linear regression by minimizing the total distance to all the points.

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:

$$S = \sum_{i=0}^{3} (y_i - f(x_i))^2 = \sum_{i=0}^{3} (y_i - a_0 - a_1 x_i)^2. \tag{17}$$

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 (17). 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:

$$S = \sum_{i=0}^{3} \omega_i (y_i - f(x_i))^2 = \sum_{i=0}^{3} \omega_i (y_i - a_0 - a_1 x_i)^2, \tag{18}$$

$\omega_i$ is a weight factor.

Let us continue with equation (17)

Two great explanations linear regression by matrices, and $R^2$-squared

# 3 Example: Solving the heat equation regression

# 4 Singular Value Decomposition

# 5 QR factorization

# 6 Solving nonlinear equations

The purpose of this section is to introduce a handful of techniques for solving a nonlinear equation. In many cases a combination of methods must be used, and the algorithm must be adopted to your specific problem.

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

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

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