Solve a System of Equations in Python (3 Examples)

Authored by

Mohammed looti

November 2, 2025

RECOMMENDED CITATION

Mohammed looti (2025). *Solve a System of Equations in Python (3 Examples)*. PSYCHOLOGICAL STATISTICS. Retrieved from https://statistics.arabpsychology.com/?p=8383

Solving a system of equations is a foundational task in mathematics, engineering, and data science. In Python, the most efficient and robust way to tackle this problem is by utilizing the powerful NumPy library, which provides optimized functions for numerical and matrix operations.

A system of linear equations can be conveniently represented in matrix form, $AX=B$, where A is the coefficient matrix, X is the vector of variables we wish to solve for, and B is the constant vector. NumPy allows us to solve for X by calculating the matrix inverse of A and multiplying it by B, such that $X = A^{-1}B$.

The following detailed examples demonstrate how to harness NumPy's linear algebra capabilities to solve systems ranging from two to four variables.

The Linear Algebra Foundation: Using Matrices

Before diving into the code, it is essential to understand the mathematical principle. Every system of linear equations can be translated into a matrix equation. For instance, the system:

$a_{1}x + b_{1}y = c_{1}$
$a_{2}x + b_{2}y = c_{2}$

is represented by the matrix equation:

$$
begin{bmatrix}
a_{1} & b_{1} \
a_{2} & b_{2}
end{bmatrix}
begin{bmatrix}
x \
y
end{bmatrix}
=
begin{bmatrix}
c_{1} \
c_{2}
end{bmatrix}
$$

By defining the coefficient matrix (A) and the constant vector (B) using NumPy arrays, we can leverage the library's built-in routines to find the solution vector (X). We will primarily use the combination of the inverse function (`np.linalg.inv`) and the dot product function (`.dot`) to achieve

this.

Example 1: Solving a System of Equations with Two Variables

Let us begin with a straightforward system involving two unknown variables, x and y. This scenario is often used to introduce the concept of solving linear systems graphically or substitutionally, but the matrix method proves far more scalable.

Consider the following system of equations:

5x + 4y = 35

2x + 6y = 36

To solve this using NumPy, we first define the coefficient matrix A (the left-hand side) and the constant vector B (the right-hand side) as NumPy arrays. The matrix A contains the coefficients and , while the vector B contains the constants .

The following code block demonstrates how to apply NumPy to efficiently solve for the values of x and y using the matrix inversion technique.

import numpy as np

#define left-hand side of equation (Coefficient Matrix A)
left_side = np.array(,])

#define right-hand side of equation (Constant Vector B)
right_side = np.array()

#solve for x and y using X = A^-1 * B
np.linalg.inv(left_side).dot(right_side)

array()

The resulting array provides the solution vector X. This output indicates that the value for x is **3** and the value for y is **5**. This method confirms the power of viewing linear systems of equations through the lens of matrix algebra.

Example 2: Solving a System of Equations with Three Variables

As the number of variables increases, manual substitution or elimination methods quickly become cumbersome and prone to error. This is where the matrix approach truly shines, as the procedural

steps remain identical regardless of the system size. We simply extend the dimensions of our matrices.

Suppose we have the following system of equations, and we need to determine the values of x, y, and z:

4x + 2y + 1z = 34

3x + 5y - 2z = 41

2x + 2y + 4z = 30

In this case, the coefficient matrix A will be a $3 times 3$ array, and the constant vector B will be a $3 times 1$ array. The process requires us to define these arrays and then apply the same inversion and multiplication steps as before.

The following code outlines how to use NumPy to solve for the three variables. Notice the simplicity of the solution structure, which is identical to the two-variable case.

import numpy as np

#define left-hand side of equation (3x3 Matrix A)
left_side = np.array(, ,])

#define right-hand side of equation (3x1 Vector B)
right_side = np.array()

#solve for x, y, and z
np.linalg.inv(left_side).dot(right_side)

array()

The resulting array indicates the solutions in order: the value for x is **5**, the value for y is **6**, and the value for z is **2**. This demonstrates the seamless scalability of the matrix method for solving larger sets of linear equations.

Example 3: Solving a System of Equations with Four Variables

For systems involving four or more variables, efficient computation becomes paramount. In real-world data science and physics applications, systems can easily reach hundreds or thousands of variables, making computational tools essential. Our method remains robust for these dimensions.

Let us consider a system with four variables: w, x, y, and z:

6w + 2x + 2y + 1z = 37

2w + 1x + 1y + 0z = 14

3w + 2x + 2y + 4z = 28

2w + 0x + 5y + 5z = 28

The definition of the coefficient matrix A must now accommodate a $4 times 4$ structure, and the constant vector B must be a $4 times 1$ structure. Pay close attention to including zeros explicitly in the matrix where a variable is missing (e.g., $0z$ in the second equation and $0x$ in the fourth equation).

We use the exact same NumPy methodology, highlighting how the general formula $X = A^{-1}B$ applies universally, regardless of the size of the coefficient matrix.

import numpy as np

#define left-hand side of equation (4x4 Matrix A)
left_side = np.array(, , ,])

#define right-hand side of equation (4x1 Vector B)
right_side = np.array()

#solve for w, x, y, and z
np.linalg.inv(left_side).dot(right_side)

array()

The resulting array provides the solution vector X. This tells us that the value for w is **4**, x is **3**, y is **3**, and z is **1**. The consistency of the NumPy approach makes it an invaluable tool for solving complex linear systems of equations.

Understanding the Role of Matrix Inversion and Dot Products

The core of the NumPy solutions presented relies on two specific linear algebra operations: matrix inversion and the dot product (matrix multiplication).

The Inverse Matrix (A^-1)

The function `np.linalg.inv(A)` calculates the inverse of the coefficient matrix A. The inverse matrix, when multiplied by the original matrix, yields the identity matrix. If a matrix is non-singular (meaning a solution exists and is unique), its inverse can be found. Calculating the inverse is

mathematically equivalent to dividing by the matrix, thus isolating the variable vector X.

The Dot Product (.dot or @)

Once the inverse A^{-1} is found, we must multiply it by the constant vector B. In NumPy, this matrix multiplication is achieved using the `.dot()` method (or the `@` operator in recent Python versions). The operation $A^{-1} cdot B$ performs the necessary calculations to generate the final solution vector X.

Alternative and Preferred Method: Using np.linalg.solve()

While calculating the matrix inversion explicitly using `np.linalg.inv()` and then multiplying by the vector B works perfectly for simple textbook examples, it is generally considered less numerically stable and computationally inefficient for very large or complex systems.

For high-performance or production environments, the preferred NumPy function is `np.linalg.solve(A, B)`. This function utilizes algorithms optimized for solving linear systems (such as LU decomposition) that avoid explicit inversion. It is faster and minimizes floating-point errors, especially when dealing with ill-conditioned matrices.

If we were to rewrite Example 1 using the preferred method, the code would be much simpler and more robust:

import numpy as np

# Define matrices A and B
A = np.array(,])
B = np.array()

# Use the specialized solver function
np.linalg.solve(A, B)

array()

For anyone routinely solving systems of equations in Python, adopting `np.linalg.solve()` is highly recommended due to its superior numerical properties.

Additional Resources

Understanding linear algebra concepts is crucial for mastering data manipulation in Python. For those interested in expanding their knowledge beyond basic matrix inversion, the following topics are recommended:

Exploring other functions within the NumPy `linalg` module, such as determinants and eigenvalues.
Studying Gaussian elimination and LU decomposition, which are the fundamental algorithms behind efficient system solvers.
Investigating the use of the `scipy.linalg` module, which offers even more advanced and specialized linear algebra routines.

The following tutorials explain how to solve a system of equations using other statistical software: