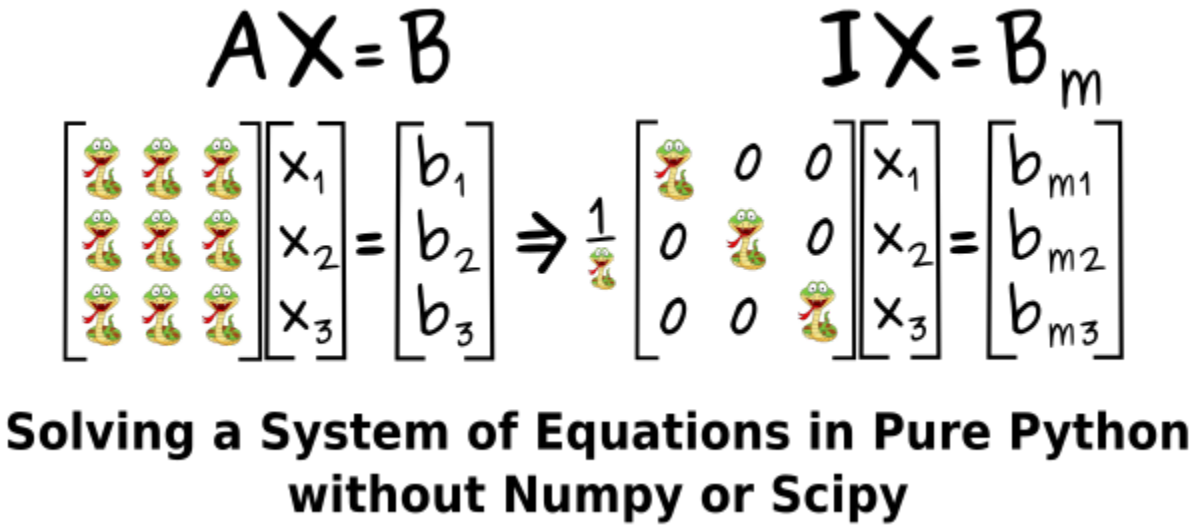
Solving a System of Equations in Pure Python without Numpy or Scipy

Originally Published by **Thom Ives** on December 3, 2018



Find the complimentary System Of Equations project on **GitHub**

Introduction

This post covers solving a system of equations from math to complete code, and it's VERY closely related to the **matrix inversion post**. There are times that we'd want an inverse matrix of a system for repeated uses of solving for **X**, but most of the time we simply need a single solution of **X** for a

system of equations, and there is a method that allows us to solve directly for **X** where we don't need to know the inverse of the system matrix.

We'll use python again, and even though the code is similar, it is a bit different. So there's a separate **GitHub repository** for this project. Also, we know that numpy or scipy or sklearn modules could be used, but we want to see how to solve for **X** in a system of equations without using any of them, because this post, like most posts on this site, is about understanding the principles from math to complete code. However, near the end of the post, there is a section that shows how to solve for **X** in a system of equations using numpy / scipy. Remember too, try to develop the code on your own with as little help from the post as possible, and use the post to compare to your math and approach. However, just working through the post and making sure you understand the steps thoroughly is also a great thing to do.

Linear Algebra Background

First, let's review the linear algebra that illustrates a system of equations. Consider $AX = B$, where we need to solve for X .

Consider a typical system of equations, such as:

$$AX = B, \qquad \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{11} & a_{12} & a_{13} \\ a_{11} & a_{12} & a_{13} \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \end{bmatrix} = \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix}$$

We want to solve for X, so we perform row operations on A that drive it to an identity matrix. As we perform those same steps on B, B will become the values of X.

$$IX = B_M, \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \end{bmatrix} = \begin{bmatrix} bm_{11} \\ bm_{21} \\ bm_{31} \end{bmatrix}$$

B has been renamed to B_M, and the elements of B have been renamed to b_m, and the M and m stand for morphed, because with each step, we are changing (morphing) the values of B.

Doing row operations on A to drive it to an identity matrix, and performing those same row operations on B, will drive the elements of B to become the elements of X.

The matrix below is simply used to illustrate the steps to accomplish this procedure for any size "system of equations" when A has dimensions $n\,x\,n$. Please note that these steps focus on the element used for scaling within the current row operations. Every step involves two rows: *one of these rows* is being used to act on the *other row* of these two rows.

$$S = \begin{bmatrix} S_{11} & \dots & \dots & S_{k2} & \dots & \dots & S_{n2} \\ S_{12} & \dots & \dots & S_{k3} & \dots & \dots & S_{n3} \\ \vdots & & & \vdots & & & \vdots \\ S_{1k} & \dots & \dots & S_{k1} & \dots & \dots & S_{nk} \\ \vdots & & & \vdots & & & \vdots \\ S_{1n-1} & \dots & \dots & S_{kn-1} & \dots & \dots & S_{nn-1} \\ S_{1n} & \dots & \dots & S_{kn} & \dots & \dots & S_{n1} \end{bmatrix}$$

Looking at the above, think of the solution method as a set of steps, S, *for each column*, and each column has one diagonal element in it. We work with columns from left to right, and work to change each element of each column to a **1** if it's on the diagonal, and to **0** if it's not on the diagonal. We'll call the current diagonal element the focus diagonal element or **fd** for short.

The first step for each column is to scale the row that has the **fd** in it by **1/fd**. We then operate on the remaining rows, the ones without **fd** in them, as follows:

1. Use the element that's in the same column as **fd** and make it a scaler;
2. Replace the row with the result of … [current row] – scaler * [row that has **fd**];
3. This will leave a zero in the column shared by **fd**.

We do this for columns from left to right in both the A and B matrices. When this is complete, A is an identity matrix, and B has become the solution for X.

These steps are essentially identical to the steps presented in the **matrix inversion post**. This is a conceptual overview. A detailed overview with numbers will be performed soon. The solution method is a set of steps, S, focusing on one column at a time. Each column has a diagonal element in it, of course, and these are shown as the S_{kj} diagonal elements. Start from the left column and moving right, we name the current diagonal element the focus diagonal (**fd**) element. We scale the row with **fd** in it to **1/fd**. Then, for each row without **fd** in them, we:

1. make the element in column-line with **fd** a *scaler*;
2. update that row with … [current row] – *scaler* * [row with **fd**];
3. a zero will now be in the **fd** column-location for that row.

We do those steps for each row that does not have the focus diagonal in it to drive all the elements in the current column to **0** that are NOT in the row with the focus diagonal in it. Once a diagonal element becomes **1** and all other elements in-column with it are **0's**, that diagonal element is a pivot-position, and that column is a pivot-column. These operations continue from left to right on

matrices A and B. At the end of the procedure, A equals an identity matrix, and B has become the solution for B.

Now let's perform those steps on a 3 x 3 matrix using numbers. Our starting matrices, A and B, are copied, code wise, to A_M and B_M to preserve A and B for later use.

$$A = \begin{bmatrix} 5 & 3 & 1 \\ 3 & 9 & 4 \\ 1 & 3 & 5 \end{bmatrix}, \qquad B = \begin{bmatrix} 9 \\ 16 \\ 9 \end{bmatrix}$$

Our starting matrices are:

$$A_M = \begin{bmatrix} 5 & 3 & 1 \\ 3 & 9 & 4 \\ 1 & 3 & 5 \end{bmatrix}, \qquad B_M = \begin{bmatrix} 9 \\ 16 \\ 9 \end{bmatrix}$$

Using the steps illustrated in the S matrix above, let's start moving through the steps to solve for X.

1. 1/5.0 * (row 1 of A_M) and 1/5.0 * (row 1 of B_M)

$$A_M = \begin{bmatrix} 1 & 0.6 & 0.2 \\ 3 & 9 & 4 \\ 1 & 3 & 5 \end{bmatrix}, \qquad B_M = \begin{bmatrix} 1.8 \\ 16 \\ 9 \end{bmatrix}$$

2. (row 2 of A_M) − 3.0 * (row 1 of A_M)
   (row 2 of B_M) − 3.0 * (row 1 of B_M)

$$A_M = \begin{bmatrix} 1 & 0.6 & 0.2 \\ 0 & 7.2 & 3.4 \\ 1 & 3 & 5 \end{bmatrix}, \qquad B_M = \begin{bmatrix} 1.8 \\ 10.6 \\ 9 \end{bmatrix}$$

3. (row 3 of A_M) − 1.0 * (row 1 of A_M)
   (row 3 of B_M) − 1.0 * (row 1 of B_M)

$$A_M = \begin{bmatrix} 1 & 0.6 & 0.2 \\ 0 & 7.2 & 3.4 \\ 0 & 2.4 & 4.8 \end{bmatrix}, \qquad B_M = \begin{bmatrix} 1.8 \\ 10.6 \\ 7.2 \end{bmatrix}$$

4. 1/7.2 * (row 2 of A_M) and 1/7.2 * (row 2 of B_M)

$$A_M = \begin{bmatrix} 1 & 0.6 & 0.2 \\ 0 & 1 & 0.472 \\ 0 & 2.4 & 4.8 \end{bmatrix}, \qquad B_M = \begin{bmatrix} 1.8 \\ 1.472 \\ 7.2 \end{bmatrix}$$

5. (row 1 of A_M) − 0.6 * (row 2 of A_M)
   (row 1 of BM) − 0.6 * (row 2 of B_M)

$$A_M = \begin{bmatrix} 1 & 0 & -0.083 \\ 0 & 1 & 0.472 \\ 0 & 2.4 & 4.8 \end{bmatrix}, \qquad B_M = \begin{bmatrix} 0.917 \\ 1.472 \\ 7.2 \end{bmatrix}$$

6. (row 3 of A_M) − 2.4 * (row 2 of A_M)
   (row 3 of B_M) − 2.4 * (row 2 of B_M)

$$A_M = \begin{bmatrix} 1 & 0 & -0.083 \\ 0 & 1 & 0.472 \\ 0 & 0 & 3.667 \end{bmatrix}, \qquad B_M = \begin{bmatrix} 0.917 \\ 1.472 \\ 3.667 \end{bmatrix}$$

7. 1/3.667 * (row 3 of A_M) and 1/3.667 * (row 3 of B_M)

$$A_M = \begin{bmatrix} 1 & 0 & -0.083 \\ 0 & 1 & 0.472 \\ 0 & 0 & 1 \end{bmatrix}, \qquad B_M = \begin{bmatrix} 0.917 \\ 1.472 \\ 1 \end{bmatrix}$$

8. (row 1 of A_M) − -0.083 * (row 3 of A_M)
   (row 1 of B_M) − -0.083 * (row 3 of B_M)

$$A_M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0.472 \\ 0 & 0 & 1 \end{bmatrix}, \qquad B_M = \begin{bmatrix} 1 \\ 1.472 \\ 1 \end{bmatrix}$$

9. (row 2 of A_M) − 0.472 * (row 3 of A_M)
   (row 2 of B_M) − 0.472 * (row 3 of B_M)

$$A_M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad B_M = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Looks correct. Let's check.

$$A \cdot B_M = A \cdot X = B = \begin{bmatrix} 9 \\ 16 \\ 9 \end{bmatrix}, \qquad YES!$$

$A \cdot B_M$ should be B and it is! Therefore, B_M morphed into X. Please appreciate that I completely contrived the numbers, so that we'd come up with an X of all **1's**.

The code in python employing these methods is shown in a Jupyter notebook called **SystemOfEquationsStepByStep.ipynb** in the repo. There are other Jupyter notebooks in the GitHub repository that I believe you will want to look at and try out. One creates the text for the mathematical layouts shown above using LibreOffice math coding. There are complementary .py files of each notebook if you don't use Jupyter.

System of Equations Solution Step-by-Step Programming

Helper Functions

In [1]:

```
def print_matrix(Title, M):
    print(Title)
    for row in M:
        print([round(x,3)+0 for x in row])

def print_matrices(Action, Title1, M1, Title2, M2):
    print(Action)
    print(Title1, '\t'*int(len(M1)/2)+"\t"*len(M1), Title2)
    for i in range(len(M1)):
        row1 = ['{0:+7.3f}'.format(x) for x in M1[i]]
        row2 = ['{0:+7.3f}'.format(x) for x in M2[i]]
        print(row1,'\t', row2)

def zeros_matrix(rows, cols):
    A = []
    for i in range(rows):
        A.append([])
        for j in range(cols):
```

SystemOfEquationsStepByStep.ipynb hosted with ❤ by **GitHub** **view raw**

The programming (*extra lines outputting documentation of steps have been deleted)* is in the block below.

```
01
02  AM = copy_matrix(A)
03  n = len(A)
04  BM = copy_matrix(B)
05
06  indices = list(range(n)) # allow flexible row referencing ***
07  for fd in range(n): # fd stands for focus diagonal
        fdScaler = 1.0 / AM[fd][fd]
```

```
        # FIRST: scale fd row with fd inverse.
        for j in range(n): # Use j to indicate column looping.
            AM[fd][j] *= fdScaler
        BM[fd][0] *= fdScaler

        # SECOND: operate on all rows except fd row.
        for i in indices[0:fd] + indices[fd+1:]: # skip fd row.
            crScaler = AM[i][fd] # cr stands for current row
            for j in range(n): # cr - crScaler * fdRow.
                AM[i][j] = AM[i][j] - crScaler * AM[fd][j]
            BM[i][0] = BM[i][0] - crScaler * BM[fd][0]
```

At the top portion of the code, copies of **A** and **B** are saved for later use, and we save **A**'s square dimension for later use. Then we save a list of the **fd** indices for reasons explained later. Next we enter the for loop for the **fd**'s. At the top of this loop, we scale **fd** rows using **1/fd**. The first nested for loop works on all the rows of **A** besides the one holding **fd**. The next nested for loop calculates (current row) – (row with **fd**) * (element in current row and column of **fd**) for matrices **A** and **B** .

Please clone the **code in the repository** and experiment with it and rewrite it in your own style. A file named **LinearAlgebraPurePython.py** contains everything needed to do all of this in pure python. **LinearAlgebraPurePython.py** is imported by **LinearAlgebraPractice.py**.

This work could be accomplished in as few as 10 – 12 lines of python. One such version is shown in **ShortImplementation.py**. I wouldn't use it. The fewest lines of code are rarely good code. However, it's a testimony to python that solving a system of equations could be done with so little code.

Solving a System of Equations *WITH* Numpy / Scipy

With one simple line of Python code, following lines to import numpy and define our matrices, we can get a solution for **X**. See documentation for numpy.linalg.solve (*that's the linear algebra solver of numpy*). The code below is stored in the repo as **System_of_Eqns_WITH_Numpy-Scipy.py**

.

```
import numpy as np

a = np.array([[5, 3, 1],[3, 9, 4],[1, 3, 5]])
b = np.array([[9],[16],[9]])

x = np.linalg.solve(a, b)

print(x)
```

I hope you'll run the code for practice and check that you got the same output as me, which is

elements of **X** being all **1's**.

Closing

It's my hope that you found this post insightful and helpful. If you did all the work on your own after reading the high level description of the math steps, congratulations! If not, don't feel bad. If you learned and understood, you are well on your way to being able to do such things from scratch once you've learned the math for future algorithms.

In the future, we'll sometimes use the material from this as a launching point for other machine learning posts.

Thom Ives

Data Scientist, PhD multi-physics engineer, and python loving geek living in the United States.