Journal of High School Science

A python code algorithm for balancing chemical equations as a system of simultaneous linear equations using matrix algebra.

Chacon D[1], Apte S[2]

Submitted: May 21, 2023
Accepted: May 21, 2023

Abstract
The trial-and-error method of balancing most chemical equations usually works well if certain heuristic rules are followed. These rules are; to balance individual reactant or product element(s) last and to preferably obtain an even number of atoms by doubling the moles of select reactant(s) or product(s). All chemical equations can be represented as systems of simultaneous linear equations; one equation for each element taking part in the reaction. Using matrix algebra hence provides a universal method to solve any chemical equation. The advantage of such an approach is that it is amenable to algorithmic compression, such that the teaching of the 'relatively non-value added' content of 'how to balance chemical equations', can instead be replaced or superseded by knowledge creating chemical concepts, such as, predicting the products of a chemical reaction, stoichiometric calculations, chemical equilibrium and reaction mechanisms. We present such a universal method to solve any chemical equation in this manuscript, and apply it to several examples of various types and complexities of chemical reactions to demonstrate its universality. The accompanying algorithm, written in python, is presented. The algorithm is also posted on Github.

Keywords
Balancing chemical equations, Matrix algebra, Simultaneous linear equations, Universal method, Coefficient, Element, Algorithm, Python, Chemical equations

---

[1]Diego Chacon, Harmony Science Academy, Euless, TX 76040, DiCh5055@student.harmonytx.org
[2]Corresponding author: Shireesh Apte, Harmony Science Academy, Euless, TX 76040, shireeshpapte@gmail.com

Introduction

The internet is replete with websites that balance input chemical equations (1-5). Many of these websites do not explain the algorithm or method used to balance the input equation. The Github website (https://github.com/) contains several algorithms that balance chemical equations. However, some of these do not list or explicitly state the underlying logic (6, 7), do not support some of the reaction types (8), or only support very specific types of reactions (9). A rudimentary manuscript describing the process of matrix algebra to balance simple chemical equations dates back nearly a decade (10).

It therefore appears that there is a significant lack of information on the methodology used to balance chemical equations. Furthermore, there is a need for a universal algorithm which can be used to balance *any* type of input chemical equation; without being limited to certain categories of chemical reactions. There is also a need to complement the Github hosted equation solver with its peer reviewed counterpart in the public domain so that users are able to completely understand the logic behind the algorithm without being relegated to deciphering cryptic instructions at the README page of the Github hosted algorithm.

Methods

Before starting to code, a library called "re" (regular expression), was imported in line 1 of the code from Python's library of modules, which checked for matches in different sets of letter combinations. This was necessary for the code to know what elements or compounds are input. A function called "findElements' ' was then created, whose purpose was to find and save the different combinations of uppercase and lowercase letters (or just uppercase letters) from the user's input. Another function called "elementsAndNumbers" was created, which was used to check if any of the uppercase and lowercase letters, (which are the input compounds), had parentheses. If the function detected any parenthesis, it saved the string inside of the parenthesis. Once all these functions executed, the code inserted the data into a matrix. The matrix was then imported from the matrix Python module that was added to the code in line 2, and - thanks to its built-in function – it could store the data provided in rows and columns. This module then allowed for the creation of several matrices to store the input information. A function called "lcm" (Least Common Multiple), was then imported which calculated the least common multiples of the previously created matrices . Once this is done, two different lists were created in the program lines 4-5; one termed "elementList" and the other termed "elementMatrix". These two lists stored the information input by the user for use in the algorithm.

The code was then instructed to print statements that provided instructions on how to use the program (program lines 7-11). Once this was done, two different input statements were created, allowing the user to enter the reactants of the formula and its products respectively. These input statements saved the user input information in two variables called "reactants" and "products". After these variables recorded the user's responses, a built-in Python function called "replace" was used to remove any empty spaces the user may have left in their input (program lines 13-14). Once the program removed the empty spaces, it saved the new version of the input in the variables called “reactants” and “inputs”. Thereafter, the first function was created and named as “addToMatrix”. This function worked in tandem with 4 other variables called “element”,“index”, count, and “side”, which were variables that held and managed the information that was input to the function (program lines 16-26).

Subsequently, the code for the function itself was written. At the start of the function, an ‘if’ statement in the code checked to determine whether all the variables were present. If not, an ‘else’ statement substituted them as zeros if determined to be empty. This function then saved the current information provided into the list “elementMatrix" and populated it with the same number of zeroes as the compounds provided. In other words, the function created a type of skeleton for the compounds provided wherein the function could update the chemical formula’s values. Subsequently, another statement in the code checked for elements that were not encountered before, and, upon finding one, it created a new row for the lists created by the function “addToMatrix”, mentioned previously, and populated the skeleton that the program created with zeroes, to symbolize the emptiness of the skeleton. A predefined function called “index” was then used to locate the empty column(s) from the skeleton that the functions created whose value needed to be changed/updated according to the balancing calculations that the program was in the process of calculating. Thereafter, a second function called “FindElements” was created, which used the information saved from the previous functions and input it into the function “FindElements” so that it could use this information and start checking it (program lines 28-38). This function executed a ‘while’ loop that continuously checked each element. Inside this loop, the program checked for two conditions: 1] if the length of the segment provided was greater than zero and 2] if it was greater than zero, it checked if the subsequent segment was non-zero. If these two conditions were met, the loop called the previous function to add that element to a matrix. Another function called “compound decipher” was then created (program lines 41-50). This function separated the parenthesis and brackets from the input equation using the “import re" library imported earlier. After this operation was completed, a “for” loop was created that continuously looped through each segment for however many segments there were and removed any extra parenthesis or brackets. Finally, this function saved the “segment”, “index”, “multiplier” and side values, which were the names of the variables that the program saved the calculated values in. After all these functions were defined and calculated, the program had all the data it required for solving the matrices generated by the previous functions.

The Python library “sympy import matrix, lcm” that was imported to the program earlier was used to calculate the result of the matrices. All the previous data was transferred to a specific matrix that Sympy could understand. A function from Sympy called “transpose” was then used to enable the program to save each column as an element. The program then calculated the nullspace of each column and generated the coefficients for the formula. The program then determined the lowest common multiple of the coefficients using “lcm” from the sympy library. The final solution was saved in the variable termed “solution”. Lastly, the code displayed the numbers from this solution as the corresponding coefficients of the input formula and displayed the final balanced equation on the screen.

The code is included as an appendix. It is also available on Github at https://github.com/diegoAchacong/python_chemical_equation_balancer/blob/main/PythonChemEuqationBalancer.py

Results and discussion

The following represent examples of equations that were balanced using the algorithm. A variety of reaction types are included such as combustion, decomposition, disproportionation replacement, ReDox, acid-base, complexation and synthesis.

1. Unbalanced equation: $C_4H_{10} + O_2 \rightarrow CO_2 + H_2O$

Balanced equation: $2\ C_4H_{10} + 13\ O_2 \rightarrow 8\ CO_2 + 10\ H_2O$

2. Unbalanced equation: $(NH_4)_2Cr_2O_7 \rightarrow N_2 + Cr_2O_3 + H_2O$

Balanced equation: $(NH_4)_2Cr_2O_7 \rightarrow N_2 + Cr_2O_3 + 4\ H_2O$

3. Unbalanced equation: $C_{57}H_{110}O_6 + O_2 \rightarrow CO_2 + H_2O$

Balanced equation: $2\ C_{57}H_{110}O_6 + 163\ O_2 \rightarrow 114\ CO_2 + 110\ H_2O$

4. Unbalanced equation: $KNO_3 + C_{12}H_{22}O_{11} \rightarrow N_2 + CO_2 + H_2O + K_2CO_3$

Balanced equation: $48\ KNO_3 + 5\ C_{12}H_{22}O_{11} \rightarrow 24\ N_2 + 36\ CO_2 + 55\ H_2O + 24\ K_2CO_3$

5. Unbalanced equation: $Cu_2S + HNO_3 \rightarrow Cu(NO_3)_2 + CuSO_4 + NO_2 + H_2O$

Balanced equation: $1\ Cu_2S + 12\ HNO_3 \rightarrow 1\ Cu(NO_3)_2 + 1\ CuSO_4 + 10\ NO_2 + 6\ H_2O$

6. Unbalanced equation: $K_4[Fe(SCN)_6] + K_2Cr_2O_7 + H_2SO_4 \rightarrow Fe_2(SO_4)_3 + Cr_2(SO_4)_3 + CO_2 + H_2O + K_2SO_4 + KNO_3$

Balanced equation: $6\ K_4[Fe(SCN)_6] + 97\ K_2Cr_2O_7 + 355\ H_2SO_4$ -> $3\ Fe_2(SO4)_3 + 97\ Cr_2(SO4)_3 + 36\ CO_2 + 355\ H_2O + 91\ K_2SO_4 + 36\ KNO_3$

7. Unbalanced equation: $Na_2S_2O_4 + NaOH \rightarrow Na_2SO_3 + Na_2S + H_2O$

Balanced equation: $3\ Na_2S_2O_4 + 6\ NaOH \rightarrow 5\ Na_2SO_3 + 1\ Na_2S + 3\ H_2O$

8. Unbalanced equation: $C_6H_8O_7 + NaHCO_3 \rightarrow Na_3C_6H_6O_7 + CO_2 + H_2O$

Balanced equation: $19\ C_6H_8O_7 + 54\ NaHCO_3 \rightarrow 18\ Na_3C_6H_6O_7 + 60\ CO_2 + 49\ H_2O$

9. Unbalanced equation: $P_4O_{10} + H_2O \rightarrow H_3PO_4$
Balanced equation: $1\ P_4O_{10} + 6\ H_2O \rightarrow 4\ H_3PO_4$

10. Unbalanced equation: $Ag^{+} + S_2O_3^{-2} \rightarrow [Ag(S_2O_3)_2]^{-3}$
Balanced equation: $1\ Ag^{+} + 2\ S_2O_3^{-2} \rightarrow 1\ [Ag(S_2O_3)_2]^{-3}$

Figures 1, 2 and the supplementary file illustrate the process used to balance equation 6 above, as an example. The supplementary .pdf file contains the step-by-step matrix procedure used to solve for the coefficients in the equation. The matrix calculator website, http://www.matrixcalc.org was used to solve the matrix obtained from the system of simultaneous linear equations generated for equation 6. Table 1 presents the element specific equations. The python code that is presented in this manuscript does this calculation as part of the code itself; in turn imported from the matrix library.

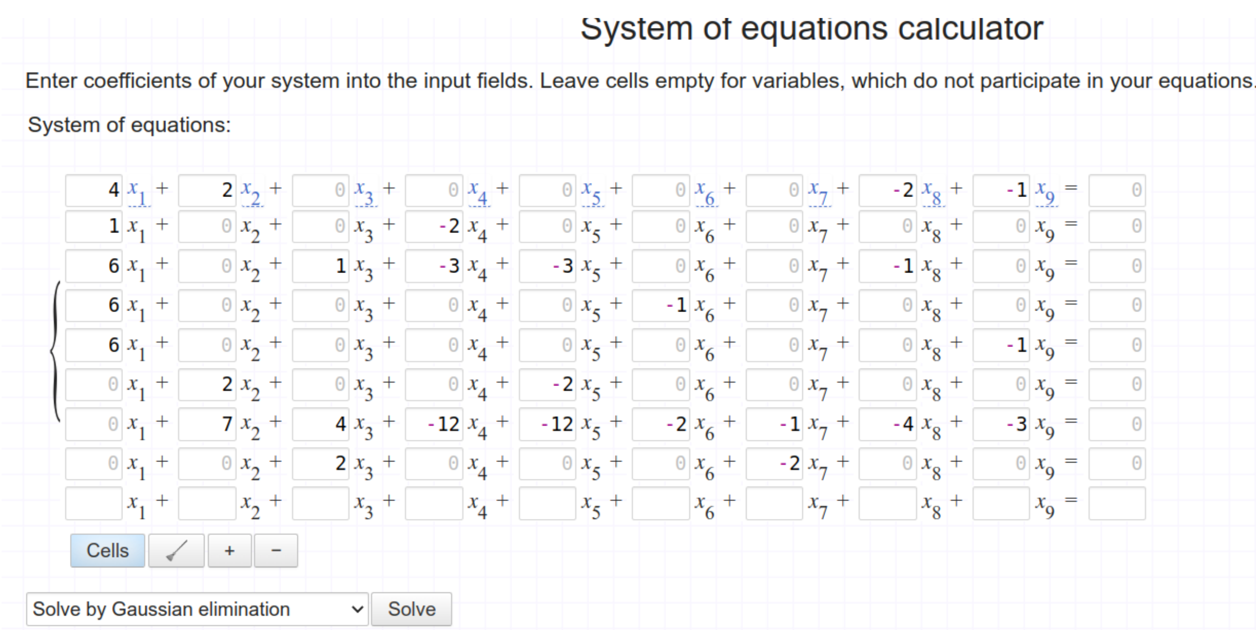

Figure 1: Input of simultaneous linear equations for equation 6 into the matrixcalc.org website. The coefficient variables X_1 through X_9 originate from those assigned to the unbalanced chemical equation: x_1 $K_4[Fe(SCN)_6]$ + x_2 $K_2Cr_2O_7$ + x_3 H_2SO_4 ⟶ x_4 $Fe_2(SO_4)_3$ + x_5 $Cr_2(SO_4)_3$ + x_6 CO_2 + x_7 H_2O + x_8 K_2SO_4 + x_9 KNO_3

Answer:

$$x_1 = \frac{1}{6} \cdot x_9$$
$$x_2 = \frac{97}{36} \cdot x_9$$
$$x_3 = \frac{355}{36} \cdot x_9$$
$$x_4 = \frac{1}{12} \cdot x_9$$
$$x_5 = \frac{97}{36} \cdot x_9$$
$$x_6 = x_9$$
$$x_7 = \frac{355}{36} \cdot x_9$$
$$x_8 = \frac{91}{36} \cdot x_9$$
$$x_9 = x_9$$

Figure 2: Answers generated from solving the matrix for equation 6 from the matrixcalc.org website.

Table 1: Element specific equations for the unbalanced chemical equation from the coefficients in Figure 1

Element	Equation
K	$4x_1 + 2x_2 = 2x_8 + 1x_9$
Fe	$1x_1 = 2x_4$
S	$6x_1 + 1x_3 = 3x_4 + 3x_5 + 1x_8$
C	$6x_1 = 1x_6$
N	$6x_1 = 1x_9$
Cr	$2x_2 = 2x_5$
O	$7x_2 + 4x_3 = 12x_4 + 12x_5 + 2x_6 + 1x_7 + 4x_8 + 3x_9$
H	$2x_3 = 2x_7$

Conclusion

Chemical reactions can be balanced when the coefficients of all the elements constituting that reaction are treated as variables in a set of simultaneous linear equations. A program was written in Python to solve the matrix created from these simultaneous equations. The code could balance any type of reaction and therefore is unique in its universal application.

References

1. https://www.webqc.org/balance.php

2. https://planetcalc.com/6145/

3. https://byjus.com/chemical-equation-calculator/

4. https://calculator-online.net/chemical-equation-balancer-calculator/

5. https://www.chemicalaid.com/tools/equationbalancer.php?hl=en

6. https://gist.github.com/CoolOppo/c52d0cef39fcc051c7c7

7. https://github.com/vano-maisuradze/chemical-equation-balancer

8. https://github.com/h-hg/ChemicalEquationBalancer

9. https://github.com/djinnome/rxneqn

10. Gabriel CI, Onwuka GI, Balancing of chemical equations using matrix algebra, J. Nat. Sci. Res., 5(5), 2015.

Appendix: Python code used to balance chemical equations

```
import re
from sympy import Matrix, lcm

elementList=[]
elementMatrix=[]

print("please input your reactants, this is case sensitive")
print("your input should look like: H2O+Ag3(Fe3O)4")
reactants=input("Reactants: ")
print("please input your products, this is case sensitive")
products=input("Products: ")

reactants=reactants.replace(' ', '').split("+")
products=products.replace(' ', '').split("+")

def addToMatrix(element, index, count, side):
    if(index == len(elementMatrix)):
       elementMatrix.append([])
       for x in elementList:
           elementMatrix[index].append(0)
```

```
    if(element not in elementList):
        elementList.append(element)
        for i in range(len(elementMatrix)):
            elementMatrix[i].append(0)
    column=elementList.index(element)
    elementMatrix[index][column]+=count*side

def findElements(segment,index, multiplier, side):
    elementsAndNumbers=re.split('([A-Z][a-z]?)',segment)
    i=0
    while(i<len(elementsAndNumbers)-1):#last element always blank
            i+=1
            if(len(elementsAndNumbers[i])>0):
              if(elementsAndNumbers[i+1].isdigit()):
                  count=int(elementsAndNumbers[i+1])*multiplier
                  addToMatrix(elementsAndNumbers[i], index, count, side)
                  i+=1
              else:
                  addToMatrix(elementsAndNumbers[i], index, multiplier, side)

def compoundDecipher(compound, index, side):
    segments=re.split('(\([A-Za-z0-9]*\)[0-9]*)',compound)
    for segment in segments:
        if segment.startswith("("):
            segment=re.split('\)([0-9]*)',segment)
            multiplier=int(segment[1])
            segment=segment[0][1:]
        else:
            multiplier=1
        findElements(re.sub('(\[|\])', '', segment), index, multiplier, side)

for i in range(len(reactants)):
    compoundDecipher(reactants[i],i,1)

for i in range(len(products)):
    compoundDecipher(products[i],i+len(reactants),-1)

elementMatrix = Matrix(elementMatrix)
elementMatrix = elementMatrix.transpose()
solution=elementMatrix.nullspace()[0]
multiple = lcm([val.q for val in solution])
solution = multiple*solution
coEffi=solution.tolist()
output=""
```

```
for i in range(len(reactants)):
    output+=str(coEffi[i][0])+reactants[i]
    if i<len(reactants)-1:
        output+=" + "
output+=" -> "

for i in range(len(products)):
   output+=str(coEffi[i+len(reactants)][0])+products[i]
   if i<len(products)-1:
        output+=" + "
print(output)
```