National Autonomous University of Mexico

Bachelor's Degree in Information Technologies in Sciences

Linear Algebra Project: Automation with Python of the Chemical Equation Balancing Process Using Linear Algebra Concepts

Linear Algebra

Author: Arnoldo Fernando Chue Sánchez

June 2023

Sections

1 Introduction

Based on [4], we can outline the concepts we will need for this project.
In general, a simple definition of a chemical reaction is that it is a process in which one or more substances interact to form new substances. The substances with which the process begins are called reactants, and the substances obtained at the end are known as products.
There are several rules for a chemical reaction to occur. However, for the purposes of this work, we will only consider two: the elements that make up the reactants must be exactly the same as those that make up the products (nuclear reactions are an exception to this rule, but we will not consider them for this work). The second rule is that the same number of atoms of an element must be present in both the reactants and the products.
This latter rule is known as the law of Lavoisier, and it is the reason why we balance chemical reactions. By balancing, we mean the process of adding the same number of atoms of all elements in both the reactants and the products. This is achieved by multiplying the molecules on both sides of the reaction by certain coefficients that we need to determine. Discovering those coefficients is exactly the balancing process.
Additionally, when doing so, we must take into account the subscripts already present in the reaction, as they represent the initial quantity of each element. When subscripts are not written, they are considered to be equal to 1. In the case where there are elements within parentheses and the parentheses have a subscript, the outer subscript multiplies all the inner subscripts.
For example, if we have a molecule of sulfuric acid:

$$H_2SO_4 \tag{1}$$

We can count that there are 2 hydrogen atoms, 1 sulfur atom (implicitly), and 4 oxygen atoms.
Now, let's consider the case of parentheses:

$$Al_2(SO_4)_3 \tag{2}$$

We count that there are 2 aluminum atoms, 3 sulfur atoms, and 12 oxygen atoms. In order to input the equations into the Python code

we will create, molecules and elements in the form of 2 should be transformed into the form of 1.
In this case, it would be:

$$Al_2S_3O_{12} \tag{3}$$

Finally, when we have coefficients for an element or molecule, we count them in a similar way to 2. It is as if the coefficient were a large subscript for the entire substance.
Example:

$$2\,Al_2(SO_4)_3 \tag{4}$$

4 is equivalent to

$$(Al_2(SO_4)_3)_2 \tag{5}$$

If we count it in the same way we counted 3, we have:

$$Al_4S_6O_{24} \tag{6}$$

Now that we have a clear understanding of how to count the number of atoms in a reaction, let's provide an example of an unbalanced equation and how it would look when balanced.

$$H_2 + O_2 \longrightarrow H_2O \tag{7}$$

In 7, we can count that there are 2 hydrogen atoms and 2 oxygen atoms in the reactants, while in the products, there are 2 hydrogen atoms and 1 oxygen atom. Therefore, it is unbalanced.
If we go through the balancing process, it becomes:

$$2\,H_2 + O_2 \longrightarrow 2\,H_2O \tag{8}$$

Although 8 represents the same reaction as 7, 8 is balanced because there are 4 hydrogen atoms and 2 oxygen atoms in the reactants, and the same number of hydrogen and oxygen atoms in the products.

2 Objectives

- Utilize the concepts of linear algebra learned during the current course to implement an algorithm in Python that takes a chemical reaction as input and returns it balanced.
- Program a class in Python that allows working with a chemical equation as an object with its corresponding methods.

3 Problem Statement

The concepts that will be presented below were covered in class. The book where the information can be consulted is [2]. The original idea for this problem was taken from [1]. In that book, it is proposed that any chemical equation can be balanced using linear algebra.
The first step is to assign a variable to each molecule in the equation. For example:

$$\mathrm{HCl} + \mathrm{Na_3PO_4} \longrightarrow \mathrm{H_3PO_4} + \mathrm{NaCl} \tag{9}$$

Let:

$$\begin{aligned} x_1 &= \mathrm{HCl} \\ x_2 &= \mathrm{Na_3PO_4} \\ x_3 &= \mathrm{H_3PO_4} \\ x_4 &= \mathrm{NaCl} \end{aligned}$$

The idea is to find the coefficients of these variables by solving a linear system and substitute them into the original equation to balance it.
The second step is to create a matrix to solve this linear system. This matrix will have n rows and m columns, where n is determined by the number of elements involved in the reaction and m is the number of variables (molecules) we have. Initially, we can create a matrix filled with zeros.
Therefore, the filling of the matrix is done as follows:

1. Assign an element to each row.
2. Iterate through all the rows and their entries.

3. Place in those entries the number of atoms of the element that appear in that molecule. If it does not appear, keep it as zero. If it is a variable corresponding to a reactant, the number remains positive; if it corresponds to a product, the number is negated.

This way, we would have our matrix ready to work with. If we want the augmented matrix of the system, the right-hand side would be filled with zeros (null vector).
Example:
Using equation 9 with its respective variables, the matrix (without the right-hand side) would be:

$$\begin{bmatrix} 1 & 0 & -3 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 3 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & 4 & -4 & 0 \end{bmatrix}$$

Where row 1 corresponds to hydrogen, row 2 to chlorine, row 3 to sodium, row 4 to phosphorus, and row 5 to oxygen.
Once the matrix is formed, the next step is to bring it to its reduced row-echelon form. Continuing with the example, it would become:

$$\begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1/3 \\ 0 & 0 & 1 & -1/3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

We know that the right-hand side of our matrix is equal to the null vector. This is extremely important because finding the solution to the linear system using the Gauss-Jordan method gives the same result as taking what we already have and obtaining the basis of the null space of the matrix.
In the next section, we will translate all this explanation into Python code. Therefore, we will choose to find the basis of the null space because computationally it is easier than automating the solution of a linear system using Gauss-Jordan.
Furthermore, in terms of interpretation for this chemistry problem, finding the null space makes more sense. What we are looking for

are the coefficients of the molecules that give us the same number of atoms per element on both sides of the equation. These coefficients must meet two requirements: they must be nonzero and non-fractional.
The first requirement is precisely why we are looking for the null space: a vector that, when multiplied by our matrix, gives us the null vector without being the null vector itself. As for the second requirement, it is the reason why we want the basis of the null space: so that if we end up with fractional components, we can scale the vector to the smallest possible positive integer values.
Continuing with our example, we obtain the basis of the null space of our matrix in reduced row-echelon form.

$$\begin{bmatrix} 1 & 0 & 0 & -1 & |0 \\ 0 & 1 & 0 & -1/3 & |0 \\ 0 & 0 & 1 & -1/3 & |0 \\ 0 & 0 & 0 & 0 & |0 \\ 0 & 0 & 0 & 0 & |0 \end{bmatrix}$$

The basis of the null space would be:

$$\left\{ \begin{pmatrix} 1 \\ 1/3 \\ 1/3 \\ 1 \end{pmatrix} \right\}$$

Next, what needs to be done is to scale the basis of the null space to eliminate fractional values. The way to do this and obtain the smallest possible integers is to multiply the vector by the least common denominator of its components.
In the case of our example, the vector would be:

$$\begin{pmatrix} 3 \\ 1 \\ 1 \\ 3 \end{pmatrix}$$

Having our vector with all its integer and positive components, the only thing left is to assign those values to the initial variables of the problem and substitute them in the original reaction.

Concluding with our example:

$$3\,\mathrm{HCl} + 1\,\mathrm{Na_3PO_4} \longrightarrow 1\,\mathrm{H_3PO_4} + 3\,\mathrm{NaCl} \tag{10}$$

Note that equation 10 is already balanced: 3 hydrogen atoms on each side, 3 chlorine atoms on both sides, 3 sodium atoms, 1 phosphorus atom, and 4 oxygen atoms.
It will be this procedure that we will translate into code to have an algorithm that balances any equation.

4 Development of the Code

To begin with, Python was chosen for this program due to the available libraries and being the main programming language used in the field.
Regarding the libraries that will be used:

- **NumPy.** [3] We will use NumPy to create the matrix that we will work with when entering values into the matrix.
- **SymPy.** [5] This is the library we will use the most because of its useful functions. It allows us to follow the algorithm we developed more easily (without having to worry as much about computational issues).

Now, the code development was approached in two parts: the first part was to create a class that would allow us to work with the reaction as an object itself. In other words, the class allows us to receive the equation as a text string, and the methods of this new object prepare us to solve this problem. The second part is the actual implementation of the problem. It involves everything from forming the matrix to obtaining the vector with the coefficients.

4.1 Creation of the Reaction class

The first thing we need to do is define the format in which the program will receive the equations:

- First, to denote the arrow of the reaction, the symbols => should be used.
- Second, for the symbols of addition, both in the reactants and in the products, the symbol + should be used.

- Third, the molecules should be written in the same way as described in 1, 3, or 6.
- Finally, there should be only one space between each molecule and its sign, but the last character should not be a space.

Example of how equations should be entered:
HCl + Na3PO4 => H3PO4 + NaCl
Once the input rules have been defined, we can begin constructing the class., we define the constructor:

```
class Reaction:
    def __init__(self, reaction: str):
        self.reaction = reaction
        self.reactants = []
        self.products = []
        self.elements = set()
        self.amounts = {}
```

In the self.reaction attribute, we will store the reaction as a string. The self.reactants and self.products attributes will be lists where we will store the molecules of the reactants and products, respectively. For the self.elements attribute, a set data structure was chosen to store the elements without repetition and to access them quickly. As for the self.amounts attribute, it will be a dictionary where the keys are the elements and their values are a list of numbers. This list of numbers will be exactly the same as their respective row in the matrix.
Following this logic, the first functions we need to implement inside the class are the ones for reactants and products:

```
    def get_reactants(self):
        self.products = self.reaction.strip().split("=>")
        self.products = self.products[0].split("+")
        self.products = [x.strip() for x in self.products]
        return self.products

    def get_products(self):
        self.reactants = self.reaction.strip().split("=>")
        self.reactants = self.reactants[1].split("+")
        self.reactants = [x.strip() for x in self.reactants]
        return self.reactants
```

Both functions work and do exactly the same thing, except that one stores the reactants and the other stores the products.

Next, we continued with the function that will store the elements in the set described in the constructor:

```
    def get_elements(self):
        x = ""
        for e in self.reaction:
            if e == "=" or e == ">": break
            elif e == "+": pass
            elif e == " " or e.isnumeric():
                self.elements.add(x)
                x = ""
            elif e.isupper():
                self.elements.add(x)
                x = e
            else: x += e
        self.elements.add(x)
        self.elements.discard("")
        return self.elements
```

Now, before we continue with the function that builds the dictionary with the element quantities per molecule, we need to process the reaction. Although we have already given specific instructions to the user regarding the input format of a reaction, we are going to process it further to explicitly show the subscripts that are equal to 1:

```
    def add_explicit_ones(self):
        count = 0
        for i, e in enumerate(self.reaction):
            i += count
            if i == 0 or e == "+" or e == ">": continue
            if e.isupper() and (self.reaction[i-1].isupper()
                or self.reaction[i-1].islower()):
                self.reaction = self.reaction[:i] + "1" +
                    self.reaction[i:]
                count += 1
                continue
            if e == " " and (self.reaction[i-1].isupper() or
                self.reaction[i-1].islower()):
                self.reaction = self.reaction[:i] + "1" +
                    self.reaction[i:]
                count += 1
                continue
        if not(self.reaction[-1].isnumeric()): self.reaction
            += "1"
        return self.reaction
```

To better understand, here is an example of how this function works:
Input:

HCl + Na3PO4 => H3PO4 + NaCl
Output:
H1Cl1 + Na3P1O4 => H3P1O4 + Na1Cl1
Once we have processed the reaction, we can proceed to count the number of times each element appears in each molecule and store it in the dictionary we described. This function is one of the most crucial parts of the entire code because if it fails, we won't be able to construct the matrix we need to work towards balancing the equation.

```
    def get_element_amounts_per_molecule(self,
        explicit_reaction: str):
        e, num, left_side, molecules = "", "", True, 1
        for x in explicit_reaction:
            if x.isnumeric(): num += x
            elif x.isalpha():
                if num != "":
                    if left_side:
                        try: self.amounts[e].append(int(num)
                            )
                        except: self.amounts[e] = [int(num)]
                    else:
                        try: self.amounts[e].append(-int(num
                            ))
                        except: self.amounts[e] = [-int(num)
                            ]
                    num, e = "", x
                else: e += x
            elif x == ">": left_side = False
            elif x == "+" or x == "=":
                if left_side:
                    try: self.amounts[e].append(int(num))
                    except: self.amounts[e] = [int(num)]
                else:
                    try: self.amounts[e].append(-int(num))
                    except: self.amounts[e] = [-int(num)]
                num, e = "", ""
                for y in self.elements:
                    try:
                        if len(self.amounts[y]) < molecules:
                            try: self.amounts[y].append(0)
                            except: self.amounts[y] = [0]
                    except:self.amounts[y] = [0]
                molecules += 1
        try: self.amounts[e].append(-int(num))
        except: self.amounts[e] = [-int(num)]
        for y in self.amounts:
            if len(self.amounts[y]) < molecules:
```

```
                try: self.amounts[y].append(0)
                except: self.amounts[y] = [0]
        return self.amounts
```

As we can see, it is the most extensive part of the entire program, as it handles many exceptions and is very meticulous to ensure that the counting is done correctly.
With all of the above, we could start working with our problem. However, it is worth creating an additional function for our class that is responsible for validating if the equation provided is already balanced from the beginning. This is done in order to save ourselves the entire process of the method we described that we will follow and to avoid errors due to certain exceptions.

```
    def is_initially_balanced(self):
        for x in self.amounts:
            if sum(self.amounts[x]) != 0: return False
        return True
```

With this, we can conclude our Reaction class. It is true that we could add additional functions that would be very useful for chemistry work. However, for the purposes of this project, this class is more than sufficient.

4.2 Problem Resolution. Applied Linear Algebra

The first thing we need to do to work on the main part of our program is to have a message with the input instructions for users stored in a variable. This variable can be printed just before requesting the input.

```
if __name__ == '__main__':
    clarificationForInputs = '''\nInstructions for inputs:
        Subscripts for elements come after the element
            itself and must be strictly less than 10.
        Leave a space between each compound and the reaction
             symbol.
        Use the symbol => to indicate a reaction.
    Example:
        HCl + Na3PO4 => H3PO4 + NaCl\n'''
    print(clarificationForInputs)
    input_reaction = Reaction(input("Enter the reaction:\n")
        )
```

Now what follows is to execute all the functions we created for the Reaction class:

```
    reactants = input_reaction.get_reactants()
    products = input_reaction.get_products()
    elements = input_reaction.get_elements()
    explicit_reaction = input_reaction.add_explicit_ones()
    amounts = input_reaction.
        get_element_amounts_per_molecule(explicit_reaction)
```

Having this, the first step is to validate if the reaction is already balanced from the beginning. If it is balanced, it would not be necessary to execute the rest of the program. But if it is not balanced, the first step is to construct our matrix.

```
    if input_reaction.is_initially_balanced(): print("\nThe
        reaction is already balanced.")
    else:
        M = np.matrix(len(elements) * [([0] * (len(products)
            + len(reactants)))])
        for i, c in enumerate(amounts): M[i] = amounts[c].
            copy()
```

Note that first we create a matrix of the required size but filled with zeros. Only then do we populate it with the data obtained from the counting.
The next step is to bring the matrix to its reduced row echelon form. SymPy has a method that does this automatically and also gives us the pivot columns. Therefore, we first need to convert our NumPy matrix to SymPy to be able to apply the method.

```
        f_e_r = sympy.Matrix(M)
        f_e_r, pivots = f_e_r.rref()
```

Having the above, the next step is to compute the null space, which we will store as a SymPy matrix.

```
        null_space = sympy.Matrix(f_e_r.nullspace())
```

It is important to note that at this point, the null space obtained using this method may have fractional values. This could be a computational problem because scaling it to the nearest positive integer would be very complicated. However, SymPy leaves it as fractions, which means we can access the denominators:

```
        denominators = null_space.applyfunc(lambda x: x.
            as_numer_denom()[1])
        denominators = set(denominators)
```

Note how we store the denominators in a set to eliminate duplicates, which makes it easier to find the common denominator.
Now, there is a case that could cause problems when searching for a common denominator: when all the denominators are the same (meaning our set only contains one value). To address this, we will filter that case and solve it by incrementing the same value once. We will also convert the set into a list to store the two values. This may not be a problem for humans when performing this exercise, but it can be problematic for computers and the lcm() function in SymPy.

```
        if len(denominators) == 1:
            denominators = list(denominators)
            denominators.append(denominators[0])
            common_denominator = sympy.lcm(denominators)
        else: common_denominator = sympy.lcm(*set(
            denominators))
```

Having the common denominator, it's just a matter of scaling our fractional null space and converting our data into integers.

```
        integer_coefficients = (common_denominator *
            null_space).applyfunc(lambda x: int(x))
```

Finally, with the vector of coefficients of the molecules, we only need to construct the output string of the algorithm with our data, and we will successfully conclude our program.

```
        balanced_reaction = ""
        molecules = reactants + products
        for i, m in enumerate(molecules):
            balanced_reaction += str(integer_coefficients[i
                ]) + m + " "
            if i == len(reactants) - 1: balanced_reaction +=
                 "=> "
            else: balanced_reaction += "+ "
        print("\nThe balanced equation is:")
        print(balanced_reaction[:-2])
```

5 Testing

Below are some tests that were conducted on the program in this project. Each of these tests consists of a chemical reaction, and the results provided by the program are also presented.
Inputs:

$$\mathrm{H_2 + O_2 \longrightarrow H_2O} \tag{11}$$

$$\mathrm{HCl + Na_3PO_4 \longrightarrow H_3PO_4 + NaCl} \tag{12}$$

$$\mathrm{Na + H_2O \longrightarrow NaOH + H_2} \tag{13}$$

$$\mathrm{CH_4 + O_2 \longrightarrow CO_2 + H_2O} \tag{14}$$

$$\mathrm{C_6H_{12}O_6 + O_2 \longrightarrow CO_2 + H_2O} \tag{15}$$

$$\mathrm{AgNO_3 + NaCl \longrightarrow AgCl + NaNO_3} \tag{16}$$

$$\mathrm{HCl + NaHCO_3 \longrightarrow H_2O + CO_2 + NaCl} \tag{17}$$

$$\mathrm{H_2S + FeCl_3 \longrightarrow HCl + S + FeCl_2} \tag{18}$$

$$\mathrm{HNO_3 + Cu \longrightarrow H_2O + NO_2 + CuN_2O_6} \tag{19}$$

$$\mathrm{Cl_2 + Mg \longrightarrow MgCl_2} \tag{20}$$

Outputs:

$$\mathrm{2\,H_2 + 1\,O_2 \longrightarrow 2\,H_2O} \tag{21}$$

$$\mathrm{3\,HCl + 1\,Na_3PO_4 \longrightarrow 1\,H_3PO_4 + 3\,NaCl} \tag{22}$$

$$\mathrm{2\,Na + 2\,H_2O \longrightarrow 2\,NaOH + 1\,H_2} \tag{23}$$

$$\mathrm{1\,CH_4 + 2\,O_2 \longrightarrow 1\,CO_2 + 2\,H_2O} \tag{24}$$

$$\mathrm{1\,C_6H_{12}O_6 + 6\,O_2 \longrightarrow 6\,CO_2 + 6\,H_2O} \tag{25}$$

$$\mathrm{1\,AgNO_3 + 1\,NaCl \longrightarrow 1\,AgCl + 1\,NaNO_3} \tag{26}$$

$$\mathrm{1\,HCl + 1\,NaHCO_3 \longrightarrow 1\,H_2O + 1\,CO_2 + 1\,NaCl} \tag{27}$$

$$\mathrm{1\,H_2S + 2\,FeCl_3 \longrightarrow 2\,HCl + 1\,S + 2\,FeCl_2} \tag{28}$$

$$\mathrm{4\,HNO_3 + 1\,Cu \longrightarrow 2\,H_2O + 2\,NO_2 + 1\,CuN_2O_6} \tag{29}$$

$$\mathrm{1\,Cl_2 + 1\,Mg \longrightarrow 1\,MgCl_2} \tag{30}$$

As seen from the results, all the balances were done correctly. The program works perfectly because the approach using linear algebra to balance the equations is correct. In the cases of equations 26, 27, and 30, the initial equation was already balanced, and the program identified it.

6 Conclusions

As discussed throughout this course, the applications of linear algebra in various fields of knowledge are extensive. In this project, the focus was on a repetitive chemistry task, and our results demonstrated the successful automation of this process. It is also important to acknowledge and appreciate the significant advancements made in terms of functions and libraries available today for performing complex mathematical tasks. Without the tools provided by the NumPy and SymPy libraries, the computational aspect of this project would have been infinitely more challenging.
For future work, it would be worth conducting more tests to obtain a meaningful sample of the algorithm's effectiveness rate. Additionally, there are two clear paths for the continued development of this project at a computational level. The first is to work on expanding the program's acceptance of equations in different formats, improving the input cleaning process to be less restrictive. Exploring the validation of equations using regular expressions would be an interesting topic to enhance the program.
The other path is to further expand the existing reaction class. There is more valuable information and properties about reactions that would greatly benefit specialists in this field. By automating more functions with this program, we would be saving them time and effort both inside and outside the laboratories.

7 Bibliography

References

[1] Chris Rorres Howard Anton. *Elementary Linear Algebra*. 2014. WILEY.

[2] David C. Lay. *Álgebra lineal y sus aplicaciones*. 2012. PEARSON EDUCACIÓN.

[3] Travis Oliphant. Numpy, 2005. `https://numpy.org/`.

[4] Bruce E. Bursten Julia R. Burdge Theodore L. Brown, H. Eugene LeMay. *Química. La ciencia central*. 2004. PEARSON EDUCACIÓN.

[5] Ondřej Čertík. Sympy, 2006. https://www.sympy.org/en/index.html.

[1] [2] [3] [4] [5]