2022 年臺灣國際科學展覽會
優勝作品專輯

作品編號 **190041**

參展科別 電腦科學與資訊工程

作品名稱 **Solving Mathematical and Chemical Equations using Python**

得獎獎項

國　　家 **Luxembourg**

就讀學校 **St. George's International School a.s.b.l.**

指導教師

作者姓名 **Max Gold**

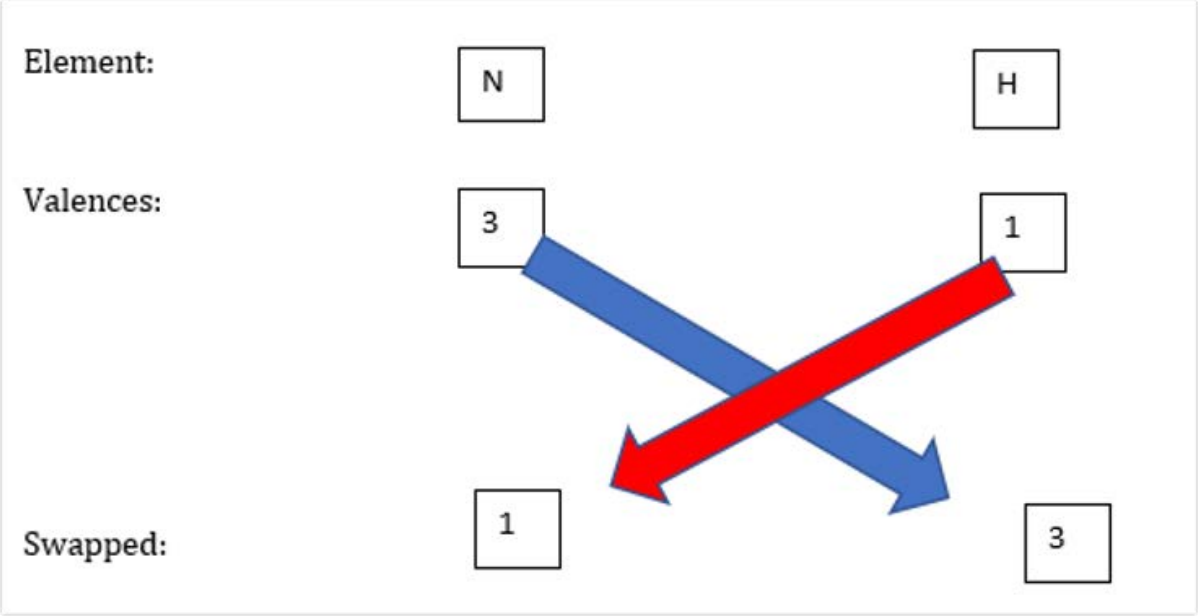
關鍵詞

作者照片

This is a continuation of my original application for the Jonk Fuerscher competition in Luxembourg (see “Original Report for the Jonk Fuerscher competition”), since which I have added and continued to work on my project and overall aim to expand my programming knowledge across multiple fields of chemistry and extend that to principles of maths as well. I have also added these chemistry and maths programmes to a website to allow other people to use, using the Flask python library as well as countless others.

Summary of Programmes mentioned in the Original

I began my project with the creation of a valency manipulator in order to obtain the smallest possible combination between any permutation of elements or compounds. The valency of an element or a compound is its willingness to react with other compounds or elements, essentially its combining capability. The valency value reflects how many electrons an element or compound needs to lose or gain in order to have a full outer shell for example oxygen has a valency of 2 since it needs to gain 2 electrons in order to have a full outer shell. In order to find the smallest possible combination, we were taught the “Swap and Drop” method at GCSE, which involves swapping the two valences of the compounds or elements involved, dividing them by a common factor and “dropping” them for use, as seen in an example here:



This yields a result of NH_3, as there are no common factors by which to divide. More complex forms, such as the combination of manganese and phosphate are mentioned in the first document.

In order to store all the information for each element and relevant compound for GCSE, I created a dictionary database in python storing each element’s symbol, valency/valences, molar mass, diatomic Boolean, electron structure, and number of electrons. This is because storing the valences locally would make run times for the programme much quicker and it would be useful to store a diatomic Boolean value for future use.

The subsequent programme I created was one which balances chemical equations. Chemical equations such as $N_2 + H_2 \rightarrow NH_3$ follows the swap and drop method, and produces the correct compound, it does not follow the law of conservation of mass or the law of conservation of energy, since the number of nitrogen and hydrogen atoms are different in the reactants to the products. In order to balance an equation, a coefficient must be added to each compound to make the number of atoms per element the same for both the reactants and products, in the above example $N_2 + 3H_2 \rightarrow 2NH_3$. In order to do this algebraically, meaning it will work for any inputted equation, the programme uses Gaussian-Jordan matrix elimination in order to obtain those coefficients. Firstly, the equation needs to be converted into a matrix, where the columns are the compounds and the rows reflect each element in the equation, e.g.

$$\begin{bmatrix} 2 & 0 & 1 \\ 0 & 2 & 3 \end{bmatrix}$$

This should then be converted through reduced row echelon form to form the identity matrix with 1 column spare to keep the coefficients, using 3 principles: row multiplication – multiplying by a constant; row addition and row multiplication, defined as the formula;

$$R_2 + (-xR_1) \mapsto R_2$$

and row switching – switching rows in order to get a non-zero value in a position where a 1 must be created. Using these rules, the matrix should be converted to:

$$\begin{bmatrix} 1 & 0 & 0 & * \\ 0 & 1 & 0 & * \\ 0 & 0 & 1 & * \end{bmatrix}$$

The stars represent the coefficients of the balanced equation, which can be then placed in the equation to make it balanced. If these numbers are fractions, the programme will find the greatest common denominator between them all and multiply them in order to get all integer values. If the matrix is square, the above matrix is desired. However, if the number of rows is greater than the number of columns, the last row is discarded as it is not necessary. If the number of columns is greater than the number of rows, an extra row is added all with 0's except for the last value which is a 1.

(See original document for more information)

Changes to the Balancing Equations Function

The first thing I changed after the Jonk Fuerscher competition was the matrix builder for the balancing function, which would turn a string equation into the matrix in order for reduced row echelon form. Originally, it was many If and Elif statements checking the conditions of each character and the next character next to it, sometimes checking 3 characters ahead

and becoming incredibly inefficient. It dealt with brackets very poorly, as equations can contain brackets and the number after the bracket is what all elements should be multiplied by, such as $(SO_4)_2$, in this case everything inside the bracket is multiplied by two. The function also didn't take into account multiple brackets and double brackets, [()], as well as multiple occurrences of the same element in a single compound, which made many errors when compounds and equations in A level chemistry are much more complex. Instead of many If statements, I created an algebraic and recursive function in order to deal with an infinite number of nested brackets and multiple occurrences of the same element in the same compound, an example of this is: TiCl2[As2[Ag2(CH3)2]2]2[As(CH4)2]3; many nested brackets and the use of multiple carbon, hydrogen, and astatine atoms.

The function checks whether there are any brackets, and will then create a list with the outermost brackets and the remaining elements outside of any brackets, in the above case ["TiCl2", "[As2[Ag2(CH3)2]2]2", "[As(CH4)2]3"]. It will then create a dictionary which will house all of the brackets and replace them in the list as (n, 0) which is a tuple. It will do the same for each of the elements in the dictionary, taking each layer of lists, putting them in the dictionary, and will replace them in their originating list with a tuple of the index they are positioned at in the dictionary, as shown in the example: ["TiCl2", ("0",), ("3",)], {"0":[["As2", ("1",)],2], "1":[["Ag2", ("2",)], 2], "2": [["CH3"], 2], "3":[["As", ("4",)], 3], "4":[["CH4"], 2]}. The tuples act as a placeholder to signify to the other elements in the dictionary that the number in the tuple is the corresponding index in the dictionary. By design, the function will take out the bracket coefficient and make that its own value in each index of the dictionary. This will only stop once every single index in the dictionary is of type string or tuple, in order to eliminate all lists. For each string, the function will take that string and split every capital letter, as every element begins with a capital letter, so the number of atoms will always correspond to the correct element, as seen in the example ["TiCl2"] -> [["Ti", 1], ["Cl", 2]. The function then works from right to left of the dictionary, filling in the placeholders with split values, until the list is purely 1 dimensional, meaning there are no lists and no placeholders, purely elements with the number of atoms of that specific element, as shown through: [['As', 4], ['Ag', 8], ['C', 8], ['H', 24], ['As', 3], ['C', 6], ['H', 24], ['Ti', 1], ['Cl', 2]]. This also solves the problem of multiple occurrences of the same element, seen through this programming snippet:

```
for j in elems:

    temp=0
    for x in lst:
        if j==x[0]:
            temp+=x[1]
        tlist.append(temp)
    matrix.append(tlist)
```

This searches through each element in the equation, and if the x[0] value (the element symbol) is the same as the element being searched for, the total adds the x[1] value (the number of atoms).

Overall, this tackles the 2 problems from the previous parsing function: permitting multiple occurrences of the same element and allows for multiple brackets and multiple nested brackets in any number of list dimensions to be parsed. This parsing function is also useful

for displaying extra information about the compounds being used in the equation, as the number of atoms for each element can be used – along with the relative atomic mass – to calculate the molar mass of each compound present.

New Programmes Created to Solve Mathematical Problems

After the creation of the 2 chemistry programmes, I decided to create some programmes to help me in my further maths iGCSE studies, the first of which is a programme to solve algebraic divisions. This can be used to divide a polynomial of degree n by a polynomial of degree 1, for example:

$$\frac{2x^3 - x^2 - 13x - 6}{x + 2}$$

where $2x^3 - x^2 - 13x - 6$ is the dividend and $x + 2$ is the divisor.

The process taught at iGCSE is to use algebraic long division, which can be seen here:

$$\begin{array}{r|l} & 2x^2 - 5x - 3 \\ x + 2 & 2x^3 - x^2 - 13x - 6 \\ & -(2x^3 + 4x^2) \\ \hline & -5x^2 - 13x \\ & -(-5x^2 - 10x) \\ \hline & -3x - 6 \\ & -(-3x - 6) \\ \hline & 0 \end{array}$$

As shown in the demonstration above, the first step is to divide the first term of the dividend $2x^3$ by the first term of the divisor x, which yields $2x^2$. This is then multiplied by the constant in the divisor, 2, to make $4x^2$, and the section of the dividend, $2x^3 - x^2$, subtract $2x^3 + 4x^2$ in order to cancel out the x^3 terms and leave the x^2 term to be used and the process is repeated again until the quotient is a constant; if this constant is non-

zero, it is the remainder, but if the constant is zero, then the divisor divides the dividend perfectly and there is no remainder.

The quotient of algebraic division can be represented by an initial term and then a sum for the remaining terms of the polynomial, which I defined as:

$$a_0 = \frac{m_0}{C_s} x^{E-1}$$

where a_0 is the initial term, E is the length of the dividend, m_0 is the coefficient of the first term in the dividend, and C_s is the coefficient of the x term in the divisor. The sum can then be represented as:

$$\sum_{\mathrm{n=1}}^{\mathrm{E}} \frac{\mathrm{m_n} - (\mathrm{m_0} * \mathrm{a_{n-1}})}{\mathrm{C_s}} x^{E-n-1}$$

where m_n is the nth coefficient of the dividend, and a_{n-1} is the previous term in the series. The final term in this series is the remainder, whose x index is -1, but this can be disregarded and considered the remainder constant.

In order to solve these equations, a parser also had to be made in order to split each x^n along with their corresponding coefficient. This also involved combining any terms where the indices were the same in order to get a total coefficient for each x^n.

The aim for the future of this programme is to allow for n degree polynomials as the divisor, so it can be an overall n degree polynomial divided by n degree polynomial calculator. Also, the aim is to be able to completely factorise up to 4^{th} degree polynomials with this calculator and display all roots, since there is a quadratic and cubic formula, so in order to display the roots for higher degree polynomials would be an interesting challenge.

Binomial Theorem Calculator

After completing the algebraic long division programme, I began to create a programme to solve binomial expansions. The binomial theorem expansion is commonly known as the following sum:

$$\sum_{k=1}^{n} \binom{n}{k} x^{(n-k)}\, y^{k} = (x+y)^{n}$$

This equation is true for all values where $n \geq 0, n \in \mathbb{Z}$, but the aim of a programmer and mathematician is to be able to solve every possible input, in this case where n can be a fraction or negative number. This requires a new formula in order to accommodate for these extra conditions:

$$a_0 = C$$

where C is the constant before the binomial, such as $2(1+x)^{(1/2)}$. The sum can then be represented as:

$$C \sum_{\mathrm{n=1}}^{\mathrm{M}} \frac{\mathrm{a_{n-1}(s-n-1)}x^{n}}{\mathrm{n}} \mathrm{C_0^n} = C(C_0 + x)^{s}$$

where C_0^n is the constant C_0 raised to the power of n, and M either the limit chosen by the user of how many coefficients they desire if s is a fraction or negative; otherwise, it is $s+1$ for $s \geq 0, s \in \mathbb{Z}$.

In order to harvest the values inside of the binomial, the parser from the algebraic division can be reused to create a list of the powers of x involved and the coefficients as well.

The area for improvement in the future which I intend to fix is the ability to remove surds and other roots from the equation if the power s is a fraction, in order to multiply the terms of the sum by that surd instead of a fraction approximation made by python. Also, if the value $C_0 < 0$ and $s \in \mathbb{Q}, s \notin \mathbb{Z}$, this will lead to complex $a + bi$ values for C_0, which will cause issues in the programme.

Website

In order to hold all of these programmes in one place, as well as making them readily available for the public to use, I have created a website using the Flask python library. This involves using routes in order to navigate between web locations and rendering HTML templates at each location. Using python as a backend language is also very useful, since all

the programmes are written in python, so it is very easy to integrate the programmes to the routes. Data can be passed through web pages to the backend python script, where it is passed through the corresponding programme's function, and the answers and other relevant information is sent back at its own unique URL. My website has 1 unique factor compared to other calculator website in the fact that whenever an input is registered by the backend server, it stores that input in a database called "Archive", which acts as a personal archive of calculations for each user, and this is spread over all current programmes, meaning that a user can switch back and forth from multiple calculations over multiple programmes with ease. This is facilitated by the "SQLAlchemy" section of the Flask library, which is an integrated SQL database manager in python, making it very easy to manage functions. Also, these archives are stored on each user's profile, meaning they are not deleted after logging out. The process of building a website has also improved my HTML, CSS, Javascript and JQuery skills, improving my overall skillset as a programmer. Each of the HTML templates used for each of the programmes uses "forms" to submit information to the backend, and these templates are also written in a combination of HTML and Jinja, which is a Flask tool to write python code in HTML. The current link to the website is govac.eu, but the site is currently down for maintenance and management.

Conclusion

In conclusion, there are 4 main programmes I have created: one to determine the smallest possible combination of two compounds or elements; one to balance a chemical equation using a parser programme I wrote myself, in conjunction with Gaussian-Jordan matrix elimination and reduced row echelon form in order to obtain coefficients for chemical equations; a programme to solve algebraic division using sums I derived and a parser for mathematical equations; finally, a programme to solve binomial equations of power $s \in \mathbb{Q}$. It has been a two year process that originated as a way to improve my programming skills for my computer science GCSE, but my passion for programming, as well as chemistry and maths, inspired me to take all of those skills to a new level and create a full website to house all the programmes I made so that others can appreciate them as well, and use them to help them with their maths and chemistry problems which is what I intended my programmes to do: help myself and others understand the underlying principles of all these interesting topics. The aims I have set myself for the binomial programme, along with the aims for the algebraic division and balancing equations as well, are a testament that this project is always ongoing and never finished, as there is always so much more to delve into and expand one's knowledge and add to a programme in order to refine it and generalise it as much as possible.

Original Document in case it is not available on the TISF website:

Chemistry Programme

by Max Gold

This programme is a combination of two particularly important chemistry topics we must study for iGCSE: balancing chemical equations and valency operations. Firstly, balancing chemical equations involves calculating the correct number of products is the same as the number of reactants. For example, $N_2 + H_2$ -> NH_3 is correct in the fact that the compounds are correct, but the number of reactants is larger to the number of products, meaning it breaks the rule of conservation of mass. This can be rectified by adding coefficients multiplying the compounds to get an equal number of elements, $N_2 + 3H_2$ -> $2NH_3$.

Valences are the combining capabilities of an element or a compound – how many electrons it has on its outer shell. For example, the valency of carbon is 4, since it has 4 electrons on its outer shell, and is coincidentally in group 4. The purpose of coding this is to allow for the smallest possible combination of 2 elements or compounds together, such as $N_2 + H_2$ -> NH_3, which takes the 2 elements Nitrogen and Hydrogen, and uses their valences to produce NH_3, the simplest possible compound.

In order to facilitate the needs of the valency calculator, I have also developed a database to locally store all the data pertinent to each element: its full name, its symbol, the number of electrons, its electron structure, whether it is diatomic (paired with another atom of the same element as a molecule), and its molar mass. Each element has the same "keys", but the value in each "key" is different depending on the element. For valences, I have also created a database to store equivalent categories for certain compounds as well, such as ammonium, sulfate, and ethanoate, saving their symbols, valency, and molar mass.

Splitting an equation for balancing

In order to begin balancing an equation, python cannot deal with an entire equation, or string, and therefore needs to split up the equation in a way that can easily be converted into a matrix. Let's suppose we are working with the equation $Mn + PO_4 = Mn(PO_4)_2$. Firstly, the equation must be split along the = sign into reactants and products, to make it easier to simplify: [$Mn + PO_4$, $Mn(PO_4)_2$] is an array with the reactants in one value, and the product in another. Secondly, the products must be split into individual compounds to trace for certain elements, causing a 2 dimensional array: {[Mn, PO_4], [$Mn(PO_4)_2$]. This, however, is not simplified enough for python, so it needs to obtain an array of just elements to look through the final equation, so it checks capitalisations and changes between uppercase and lowercase letters, making sure there are no duplicate elements: [Mn, P, O]. The array then cycles through the equation, assigning a number to the element per compound depending on the number shown, for example Mn would have a 1 in the first column, since there is 1 manganese, and Oxygen, for example, would have a 0, since there is no oxygen. The one other thing that the programme is trained to do is to check if there are any brackets in the compound, such as in $Mn(PO_4)_2$, since the subscript number multiplies everything by 2, so it cycles through and takes the values, then for the ones with brackets it multiplies those numbers by 2.

This successfully returns the matrix

Balancing an Equation [illegible] educed Row Echelon Form:

Convert an equation int [illegible] : $CO_2 + H_2O$

Mn	$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & 4 & 8 \end{bmatrix}$
P	
O	

C	$\begin{bmatrix} 3 & 0 & 1 & 0 \\ 8 & 0 & 0 & 2 \\ 0 & 2 & 2 & 1 \end{bmatrix}$
H	
O	

The matrix M can be written as (m·n). When $m < n$:

Convert to Row Echelon Form, REF goes Downwards and Rightwards, in that order

Aim = $\begin{bmatrix} 1 & * & * & * \\ 0 & 1 & * & * \\ 0 & 0 & 1 & * \end{bmatrix}$ where * indicates any number

To get 1 in the first section, one divides the **row** by the inverse, in this case $\frac{\mathbf{1}}{\boldsymbol{x}}$ where **x** is equal to the number at that position (this is known as **row multiplication**):

$$\begin{bmatrix} 1 & 0 & \frac{1}{3} & 0 \\ 8 & 0 & 0 & 2 \\ 0 & 2 & 2 & 1 \end{bmatrix}$$

Then we move downwards to 8, where we need a 0 as determined by the objective above. Therefore, we multiply the row with the column position directly above with a 1 by **-x**, where **x** is equal to number, then add it to the existing row (R). This is a combination of **row addition** and **row multiplication**.

So, this can be visualised in row 2 in the matrix as:

$$R_2 + (-xR_1) \mapsto R_2$$

$$xR_1 = [-8(1), -8(0), -8\left(\frac{1}{3}\right), -8(0)]$$

$$xR_1 = [-8, 0, -\frac{8}{3}, 0]$$

$$R_2 + xR_1 = [(8-8), (0+0), \left(0 - \frac{8}{3}\right), (2+0)]$$

$$R_2 = [0, 0, -\frac{8}{3}, 2\ ]$$

The matrix can then be updated to:

$$\begin{bmatrix} 1 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & -\frac{8}{3} & 2 \\ 0 & 2 & 2 & 1 \end{bmatrix}$$

We then continue downwards, checking that the number is correct – a 0 in this case. It is correct, and after seeing that all numbers are complete for this row, we move on to the next row, starting from position number **y**, where **y** is equal to the value in the position where the row number is equal to the column number.

$$\begin{bmatrix} 1 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & -\frac{8}{3} & 2 \\ 0 & 2 & 2 & 1 \end{bmatrix}$$

In this scenario, **y** is equal to 0, which is not the intended answer. We can use a matrix function to bypass this issue: we can swap rows 2 and 3 to get a positive number in the intended position (this is known as **row switching**).

$$\begin{bmatrix} 1 & 0 & \frac{1}{3} & 0 \\ 0 & 2 & 2 & 1 \\ 0 & 0 & -\frac{8}{3} & 2 \end{bmatrix}$$

This then allows for the entire row to be multiplied by $\frac{1}{2}$ in order to get 1 in position **y.**

$$\begin{bmatrix} 1 & 0 & 1/3 & 0 \\ 0 & 1 & 1 & \frac{1}{2} \\ 0 & 0 & -\frac{8}{3} & 2 \end{bmatrix}$$

Column 2 is now complete since the value in position 2,3 is already 0. The final value to be manipulated is 3,3 as it is the only other value needed to be changed in order to obtain a desired Row Echelon Form. The row is multiplied by the inverse, which is $-\frac{3}{8}$.

$$\begin{bmatrix} 1 & 0 & 1/3 & 0 \\ 0 & 1 & 1 & \frac{1}{2} \\ 0 & 0 & 1 & -\frac{3}{4} \end{bmatrix}$$

Row Echelon Form has now been achieved, and the next step is to create Reduced Row Echelon Form, which involves:

$$\begin{bmatrix} 1 & * & * & * \\ 0 & 1 & * & * \\ 0 & 0 & 1 & * \end{bmatrix} \text{ -> } \begin{bmatrix} 1 & 0 & 0 & * \\ 0 & 1 & 0 & * \\ 0 & 0 & 1 & * \end{bmatrix}$$

Converting the * numbers to 0 for all * except for the last column. RREF works in the opposite way to REF, going from right to left, down to up. The first element to be changed is (2,3), whose row can be **multiplied and added (**row 3 can be multiplied by -1 and added to row 2**).**

$$\begin{bmatrix} 1 & 0 & 1/3 & 0 \\ 0 & 1 & 0 & \frac{5}{4} \\ 0 & 0 & 1 & -\frac{3}{4} \end{bmatrix}$$

The 2nd row is now finished, and now the final number to be manipulated in this scenario is (1,3), since (1,2) is already 0. The same process can be done to (1,3) as (2,3), which involves multiplying the last row by -1/3 and adding it to row 1. The general rule is multiplying the row where the column 1 is located to the negative current value, then adding.

$$\begin{bmatrix} 1 & 0 & 0 & 1/4 \\ 0 & 1 & 0 & \frac{5}{4} \\ 0 & 0 & 1 & -\frac{3}{4} \end{bmatrix}$$

This matrix has now been fully converted into reduced row echelon form. In the case where m<n, an extra row must be added in order to retrieve 4 values to represent each compound. However, this extra row must also conform to the rules of reduced row echelon form, meaning the first values must be 0, and the last value must be 1.

The last row can now be used to obtain the balanced values for each compound. The absolute value of each component in the last row is placed into an array.

$$1/4 \quad 5/4 \quad 3/4 \quad 1$$

In order to obtain the balanced coefficients, each value needs to be an integer, so the least

$$\begin{bmatrix} 1 & 0 & 0 & 1/4 \\ 0 & 1 & 0 & \frac{5}{4} \\ 0 & 0 & 1 & -\frac{3}{4} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

common denominator must be obtained, which in this case is 4. Each value is then multiplied by the least common denominator, which returns the result:

$$1 \quad 5 \quad 3 \quad 4$$

The final equation is: $C_3H_8 + \mathbf{5}\ O_2 = \mathbf{3}\ CO_2 + \mathbf{4}\ H_2O$

In the cases of *m=n,* such as $S + HNO_3 \rightarrow H_2SO_4 + NO_2 + H_2O$, there is no need to add an extra row since it conforms to reduced row echelon form. In the cases of *m>n*, such as $NH_4 + SO_4 \rightarrow (NH_4)_2SO_4$, the last row is discarded, since there is no use for an extra value.

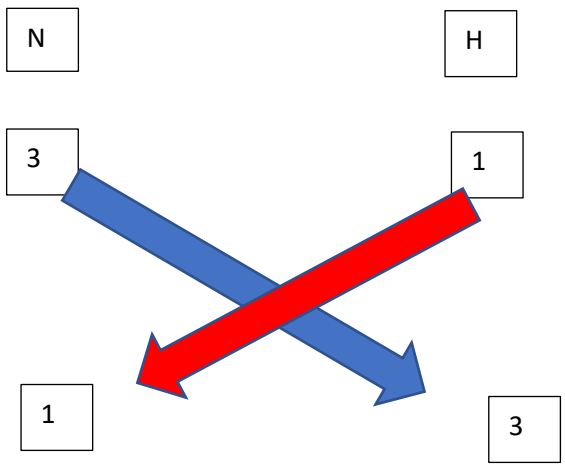
Valency Method

The method taught at GCSE chemistry and the one I have decided to implement is called the "Swap and Drop method". It involves obtaining the valences of any combination of 2 elements (either its oxidation state or its group number) or 2 compounds, for example combining nitrogen and hydrogen. The valence for nitrogen is 3, since it is group 5, and the valence for hydrogen is 1. The valences are then swapped:

Element:

Valences:

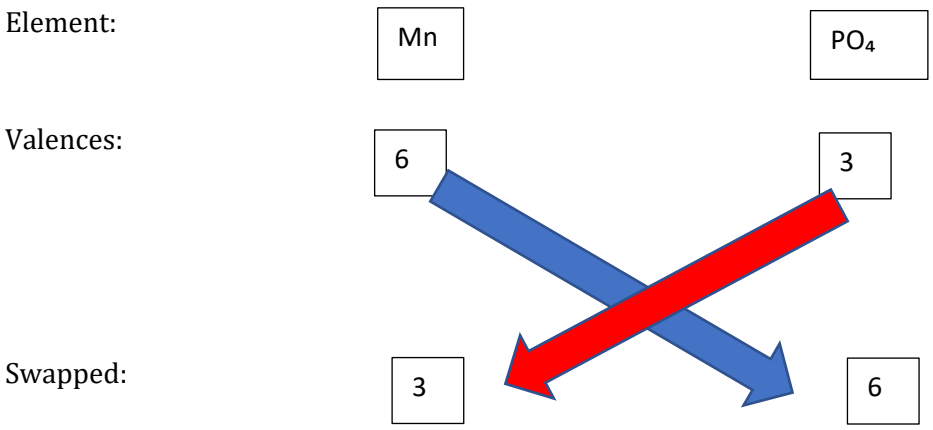
Swapped:



After swapping the valences, the programme checks whether the numbers can be simplified, since it aims to find the smallest possible combination. In this case, it cannot be simplified, so the result is the element with its new corresponding amount – since these 2 elements are also diatomic, the full equation must show them as such:

$$N_2 + H_2 \text{ -> } NH_3$$

Another, more complex equation is manganese (VI) and phosphate. Manganese is a transition element, and therefore has many oxidation states, but for this equation we will use 6+, or Mn (VI). The same swap and drop method is used – phosphate has a valency of 3-, as stored in the database:



This is an example of where the values can be simplified – both divided by 3 – in order to make the product the smallest possible combination.

Since the second value is a compound, we put the compound in brackets: (PO_4) and then add the subscripted amount next to it:

$$Mn + PO_4 = Mn(PO_4)_2$$

References:

MyWhyU's informative video on Gaussian Elimination:
https://www.youtube.com/watch?v=2GKESu5atVQ

Desmond Stephen's mention of gaussian elimination:
https://youtu.be/yCxDAj87W8M?t=227

【評語】190041

This project tries to solve mathematical and chemical equations using Python. The topic is very interesting, and the proposed work is technically sound and solid. Overall, this is good work, and it would be great if the authors could provide additional experiments and performance discussions to further strengthen the depth of this research.