# Balancing Chemical Equations

Brett Webb
Red Rocks Community College
MAT 225-002 Linear Algebra
Professor: Adam Forland

October $4^{th}$, 2019

## The Problem

Our software development team has been asked to write a piece of software for a local highschool chemistry class. The intention is to allow the students to enter in a chemical equation and the program will output the balanced equation (if possible). In addition, the software should include code comments to express how the code is working as well as mark-down/latex comments to express how the math (and chemistry) is working.

## 1 Introduction

As descrbed above, our goal was to develop a Python program that utilizes Linear Algebra to balance a given chemical equation. Chemical equations describe the quantities of substances consumed and produced by chemcial reactions. The Law of Conservation of Mass states that no atoms can be created nor destroyed in a chmeical reaction. This means that the number of atoms of the reactants (the left side of the equation) must be equal to the number of atoms of the products(the right side). So in order to balance an equation, coefficients must be added to the molecules of the equation. In the example and code to follow, you will see the process of how this is done by hand using matrix operations, and then by the program that was developed.

## 2 Example

For this example, we will use the following chemical equation,

$$(x_1)AgNO_3 + (x_2)MgCl_2 \rightarrow (x_3)AgCl + (x_4)Mg(NO_3)_2$$

Notice the coefficients that are in front of the molecules, these are placeholders just to represent the variables that we will be solving for in the following calculations. Keep in mind, these will always be integer numbers.

First, we must convert the reactants and products into vectors,

$$AgNO_3 = \begin{bmatrix} 1 \\ 1 \\ 3 \\ 0 \\ 0 \end{bmatrix}, MgCl_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 2 \end{bmatrix}, AgCl = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, Mg(NO_3)_2 = \begin{bmatrix} 0 \\ 2 \\ 6 \\ 1 \\ 0 \end{bmatrix} \qquad \begin{bmatrix} Ag \\ N \\ O \\ Mg \\ Cl \end{bmatrix}$$

These vectors each describe the numbers of atoms per molecule for each type of element present in the reaction. On the right you can see a visual representation of what element each row represents. Since the first reactant has three different atoms ($Ag$, $N$, and $O$) the corresponding number of atoms in that molecule in placed in the vector in their corresponding positions. Notice the last vector, the outside subscript of 2 represents that there are two molecules of $NO_3$ which means there are a total of 2 $N$ atoms and 6 $O$ atoms.

Next, we replace the molecules in our equation with their given vectors and replace the $\rightarrow$ arrow with an equals sign because we are trying to solve for the coefficiants that must satisy the equation.

$$x_1 \begin{bmatrix} 1 \\ 1 \\ 3 \\ 0 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 2 \end{bmatrix} = x_3 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} + x_4 \begin{bmatrix} 0 \\ 2 \\ 6 \\ 1 \\ 0 \end{bmatrix}$$

We then move all of the terms to left side of the equation to set the equation equal to 0. In this case, since we are using vectors, we use the zero vector, $\vec{0}$. This will change the signs of the vectors from the right side of the equation.

$$x_1 \begin{bmatrix} 1 \\ 1 \\ 3 \\ 0 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 2 \end{bmatrix} + x_3 \begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \\ -1 \end{bmatrix} + x_4 \begin{bmatrix} 0 \\ -2 \\ -6 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

From here, we can create the augmented matrix of the system in which our vectors become the columns of the matrix and the solution to our matrix, $\vec{0}$, is separated by

a vertical line.

$$\left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & -2 & 0 \\ 3 & 0 & 0 & -6 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 2 & -1 & 0 & 0 \end{array}\right]$$

We then apply row reduce the matrix using row operations.

$$\Longrightarrow \left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 2 & -1 & 0 & 0 \end{array}\right]$$

$$\Longrightarrow \left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 2 & 0 \end{array}\right]$$

$$\Longrightarrow \left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

After row reduction we are left with our matrix in "Row Reduced Echelon Form" (RREF).

$$\left[\begin{array}{cccc|c} 1 & 0 & 0 & -2 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

The solution to our matrix is then shown as,

$$x = \begin{cases} x_1 = 2x_4 \\ x_2 = x_4 \\ x_3 = 2x_4 \\ x_4 \ is \ free \end{cases}$$

Since $x_4$ is free, we can let it be equal to any number and it will still be a solution to our matrix. When balancing equations, it is best to keep coefficients as simplified as possible. So to do that, we let $x_4 = 1$.

$$x = \begin{cases} x_1 = 2 \\ x_2 = 1 \\ x_3 = 2 \\ x_4 = 1 \end{cases}$$

These are the the coefficants to our chemcial equation. So, the balanced equation is

$$2AgNO_3 + MgCl_2 \rightarrow 2AgCl + Mg(NO_3)_2$$

# 3 The Code

This code was developed through a collabaration effort between Jaden Adams, Connor Denney, Kevin Evers, Holly Hammons, Alex Langfield, Everett Oklar, Brett Webb, and Sophia Wyss with the supervision of our professor Adam Forland.
This code will successfully take an unbalanced chemcial equation, convert it into a matrix, row reduce the matrix, solve for free variables, and find the solution, giving us the coefficients for the balanced chemical equation.

```
from fractions import Fraction
import math
from sympy import *


# Takes a string and a delimiter character is an input and splits the string at the specified
    delimiter. Outputs the string as a tuple split at the delimiter.
def split(string, delimiter):
  output = [[]]       # Creates a tuple that will be the output of the function
  counter = 0         # Used to count how many times we have split our string

  for i in range(len(string)):      # Loop ''the number of characters in string'' times
    if string[i] == delimiter:      # Tests if the current character is the delimiter
      counter += 1                  # Increment the counter
      output.append([])             # Add a new string to the tuple
    else:                           # If the current character is not the delimiter
      output[counter] += string[i]  # Add character to current string
  return(output)                    # Output the list of strings


# Determines if an item is in a list and outputs the location of the item.
def existsin(thelist, item):
  for i in range(len(thelist)):     # Loop ''the number of entries in thelist'' times
    if thelist[i] == item:          # Tests if the current item is the item we're looking for
      return [True, i]              # Return True and location of the item.
  return [False, 0]                 # Returns False if the item is not in the current list.

# Turns a compund string into a vector. inp is the compund string input; thiscompound is an
    empty vector of the same length in which we will store our resulting vector; elements is an
     empty list to keep track of what elements are in the equation. Outputs an updated elements
```

```
    list and the compound vector, thiscompound.
def vectorizeCompound(inp, thiscompound, elements):
  for i in range(len(inp)):       # Loop ''the number of characters in the string inp'' times
    if inp[i].isnumeric() == False:       # Tests if the current character is not a number
      if i == 0:        # Tests if this is the first character in the element
        elements.append(inp[i])       # Add character to elements list
        continue          # Skip the rest of this iteration and go to the next one
      if inp[i-1].isnumeric() == False:   # Tests if the last character was also not a number
        elements[len(elements)-1] += inp[i]       # Add this character to the existing
    character, rather than making a new item for that character
      else:
        elements.append(inp[i])   # Else if it's just a lone letter so far, add it to the list

    else:   # Else if the current character is a number
      if len(elements) == 0:    # If there are no elements yet in the list
        continue        # Something went wrong, so just keep going
      if inp[i-1].isnumeric():        # If the last character was also a number
        thiscompound[Existence[1]] = inp[i-1] + inp[i]        # Add this character to the
    existing number ** (concatination of strings, not addition)
        continue        # Continue to next iteration to avoid redefining the entry
      NewElement = elements.pop(len(elements)-1)    # Remove the last element from the list
      Existence = existsin(elements, NewElement)    # Check to see if this element has already
     been counted
      if Existence[0] == False:   # If element has not been counted
        elements.append(NewElement)     # Add this element to the elements list
        thiscompound[len(elements)-1] = inp[i]    # Add the current number to the vector
        Existence[1] = len(elements)-1  # For the case where you have a multi-digit number **
      else:     # If the element has already been counted
        thiscompound[Existence[1]] = inp[i]   # Put this number in the appropriate place on
    the list.
    return [elements, thiscompound]   # Return the updated elements list and the compounds
    vector

################################################################################

equation = input("Please enter the chemical equation: \nUse no spaces. Use '=' for the arrow.
    Do not exclude 1 as a subscript. Simplify all compounds.")      # Asks user to enter
    equation
numElements = int(input("Count the total number of elements in the equation \n For example CHO
    +O2=CO2+H20 has 3 elements, C H and O."))     # Asks user to enter number of elements in
    equation
globalElements = [] # A list to keep track of which elements are which entry in each vector

sides = split(equation, '=')      # Splits the equation into the reactants and the products
if len(sides) != 2:   # Test if there is more than one product string and one reactant string
  print("!!! Product/reactant split error !!!")       # Print error message is this happens

reactants = split(sides[0],'+')   # Takes first entry of sides[] and splits it, these are the
    reactants
vector = []     # An empty vector to use in the following loop
reactantVectors = [] # This will be a list of lists to keep track of all the vectors for the
    reactants

# Creates an empty vector for each reactant and each vector is the appropriate size for the
    number of elements.
for i in reactants:       # For each entry in list reactants
  vector = []           # Zero out the vector
  for i in range(numElements):   # Loop ''the number of elements, numElements'' times
    vector.append(0)              # Add a zero to the vector
  reactantVectors.append(vector)    # Add that vector to our reactant Vectors

for i in range(len(reactants)): # Loop ''for how many items in list reactant'' times
  # Vectorize the reactant. Inputs the reactant string, the corresponding empty vector, and
    the list of elements
  vectorization = vectorizeCompound(reactants[i], reactantVectors[i], globalElements)
  globalElements = vectorization[0]   # Update element list
  reactantVectors[i] = vectorization[1]   # Update the compound vectors.

# The following section is the exact same is above, just for the products of the equation
products = split(sides[1],'+')    # Takes second entry of sides[] and splits it, these are the
```

```
    products
vector = []
productVectors = []
for i in products:
  vector = []
  for i in range(numElements):
    vector.append(0)
  productVectors.append(vector)

for i in range(len(products)):
  vectorization = vectorizeCompound(products[i], productVectors[i], globalElements)
  globalElements = vectorization[0]
  productVectors[i] = vectorization[1]

 # Multiplies all entries of the product vectors by -1 to simulate when we moved them across the equals sign.
for i in range(len(productVectors)):      # Loop through only the product vectors
  for c in range(len(productVectors[i])):   # Loop through all the entries in each vector
    productVectors[i][c] = -1 * int(productVectors[i][c])     # Multiply them all by -1

equationMatrix = reactantVectors # Create a new matrix (list of lists) that will store both reactant vectors and product vectors. This starts the list by copying the reactantVectors
for i in productVectors:     # For all the rows in the product matrix, add them to the matrix
  equationMatrix.append(i)

E = Matrix(equationMatrix)   # Converts the list of lists into an sympy matrix
E = E.transpose()        # Transpose the matrix so that the now row vectors become column vectors
A = E.rref()          # Row reduce this matrix and call it A

##########################################################################

FreeVariables = []      # Create a list for the free variables
for i in equationMatrix:   # This is the list of lists that became the matrix E. Since E got transposed, the number of items in this variable is the number of columns in the matrix E
  FreeVariables.append(1)  # Create an entry in the free variables list for every column of the matrix

for i in range(len(E.rref()[1])): # Loops ''for number of columns of E with pivots in them'' times
  FreeVariables[E.rref()[1][i]] = 0 # 'Turn off' each of our columns that is not associated with a free variable

Entries = []    # Save the value for what we should choose our free variables to be, so every output is a whole number
for c in range(len(FreeVariables)):   # Iterate through our free variables list
  if FreeVariables[c] == 1:     # If this column represents a free variable
    for r in range(E.shape[0]):   # Iterate through the entries of that column
      if A[0][r,c] != 0:      # As long as the entry is not zero
        Entries.append(Fraction(str(A[0][r,c])).denominator)      # Then save the denominator of the fraction
# The str(-sympy rational-) is neccesary because the float makes the denominator innacurate -- it will evaluate 1/3 as 3333333/10000000

# Finds the least common multiple of two numbers
def lcm(a,b):
  return (a*b) / math.gcd(a,b)    # The product of the two numbers, divided by the greatest common divisor of those two.


Current = Entries[0]  # This is just an initial condition to avoid out of range index errors **
for i in range(len(Entries)): # Iterate through our denominator list
  if i == 0:    # Forget the first entry, we already set that up above **
    continue
  Current = lcm(int(Current), int(Entries[i])) # Find the least common multiple of the current least common multiple and the current entry in the list.

V = []    # A vector to multiply our rows by to get to the solution.
for i in FreeVariables:     # Iterate through our free variables list
```

```
  if i == 1:                  # If the column is associated with a free variable
    V.append(Current)         # Multiply it by the largest multiple
  else:                       # If the column is a pivot column
    V.append(0)               # Zero the column out
    # This way, when we add the rows together we will get what the other variables equal.


F = A[0] * Matrix(V)    # Do this multiplication to get our answer as F

Coefficients = []           # An output medium for the solution
for i in range(len(FreeVariables)):   # Iterate through our columns
  if FreeVariables[i] == 1:       # If it represents our free variable
    Coefficients.append(int(Current))   # Make this free variable our chosen amount that makes
     all the numbers whole
  else:       # If the column is not a free variable
    Coefficients.append(-1 * int(F[i])) # Just take that entry from our solution vector, and
    invert it.
print(equation)         # Print the equation we set out to solve
print(Coefficients)     # and the coefficients we have solved for.
print("These are the coefficients in order that the compounds occur.")  # Hopefully it works.
```

# 4 Conclusion

It is amazing to see how a process such as balancing a chemical equation can seem so simple to do by hand but then look so complicated when being coded for a program. This is because we need to write very specific instructions for the computer to follow in order to do such a task. In doing this, we have created a much faster and more efficient way of balancing chemical equations than by doing them by hand. This would then save people time and money. However, this code can be improved in many ways and made more efficient. For example, it could be altered so that the user does not have to enter the equation without spaces or so that they would not have to enter a 1 for atoms without subscripts. Future implications of this program could be adding in some sort of GUI for the user to interact with or adding graphics so they can visually see the process of balancing the chemical equation. In reading this, I hope you have gained a better understanding of the process of balancing a chemical equation using linear algebra and the use of programming and technology to improve it.