MEng Group Project Final Report

Cristian Badoi, Aaron Butterworth, Daniel Gardam

1 Intended System Design

The initial system design is presented here as it was in our initial specification and design, we will highlight any differences or deviation from this in the actual system design section.

1.1 Expected Components

1.1.1 Web Server

The main interface to the system will be supplied by a web server to allow easy and portable access by members of the MIF. The web facing part of the system will be comprised of several smaller packages all working together under NodeJS:

NodeJS — A JavaScript runtime that allows execution of JS outside the context of a web browser.

Express — An open source web application framework for NodeJS, it is responsible for the high level web server logic and allows a high level interface to web traffic as it handles response codes, etc. itself.

Pug — An open source high-performance template engine, it allows us to alter the contents of a web page just before serving it to the user, allowing for finer levels of customisation while keeping the code base manageable.

Express-Session — An add-on for express that gives us easy control of user sessions. This allows us to keep track of user specific data, such as preferred precursors or recently searched compounds.

bCrypt2 — A widely trusted package for providing user password encryption.

1.1.2 Computation

The actual computation will be handled by C++ programs with some small python helper scripts for unit conversion and other pre/post processing.

This is done for several reasons, firstly to allow a significant performance increase over JS; secondly to allow us access to widely known mathematics libraries; and thirdly to allow easier modification of the source code by others in the MIF as these are their preferred languages as opposed to JS, this is also the start of an ongoing project which will be maintained and expanded upon after us so maintainability is an important factor.

The first step of computation is the stoichiometry calculator. This takes a selection of user defined elements, and then calculate all possible balanced compounds that may be produced by a chemical reaction of these elements. This process also takes into account user limits such as the maximum amount of atoms in the resulting combination to refine the search and prevent the creation of a technically correct but practically unreasonable result.

The programs data sources are the web interface which will invoke the users query and the database for relevant information about each element, its output is piped back to NodeJS where it may be cached in the database if it is a common search, the result is then displayed to the user via the web interface.

The second stage is the precursor calculator. Its input is a desired ratio of elements, and a set of available chemical precursors. It calculates possible combinations and quantities of the precursors that when mixed together create the desired target ratio. User bounds may also be given to keep the results practically usable.

1.1.3 Database

With the theme of future expansion the database will be hosted on a separate server instance running independently of the NodeJS process, this allows for easier expansion of the system if it cannot handle the user load placed on it as both the web server and database can be scaled completely separately from each other without modification to the program code.

The database will not only store the information needed to run calculations, but will also store the solutions to commonly executed combinations so as to reduce the computational demand by allowing these solutions to be quickly retrieved rather than repeatedly calculated.

2 Actual System Design

2.1 Computation

2.1.1 Stoichiometry Calculator

The first stage of the stoichiometry calculator is to form the **A** and **B** matrix. As we are always searching for charge balanced compounds, and the result is purely a combination of the inputs with no external factors (e.g. The reaction taking place in air vs under vacuum) we can fix the **B** matrix. In this example the user has queried for Aluminium, and Oxygen:

$$\mathbf{A} = \begin{matrix} \text{Al} & \text{O} \\ \begin{bmatrix} 1 & 1 \\ 3 & -2 \end{bmatrix} & \begin{matrix} \text{Initial Quantity} \\ \text{Charge Imbalance} \end{matrix} \end{matrix}$$

$$\mathbf{B} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{matrix} \text{Resulting Proportion} \\ \text{Desired Charge Imbalance} \end{matrix}$$

$$\begin{bmatrix} 1 & 1 \\ 3 & -2 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This gives us a matrix representation of the constraint equations. However, by our initial design we must consider all possible combinations of oxidation (charge) states in order to find all charge balanced compounds, in this example O has a total of 4 states, and Al a total of 3 giving a total of 132 permutations. While this is true computationally, some of the oxidation states are extremely uncommon and would not be desirable in a physical setting.

With this in mind our implementation differs from the design at this point, if we consider all possible charges for every element the calculator produces technically feasible but practically impossible results. As this tool is designed to be used by chemists in a real world application we decided to limit the

space by removing the rarer charges from the search. However, this data is dynamically loaded we can offer support for rarer charges to be included in the future, however at the current version this is not included.

This space reduction leaves only one charge each for Al and O, significantly reducing the search space from 132 variants of $\mathbf{A}$ to only one, while still returning all common compounds.

After having formed the $\mathbf{A}$ matrices, we solve each for a single solution via LU decomposition. Which in it general form takes a matrix $\mathbf{X}$ and finds a lower ($\mathbf{L}$) and upper ($\mathbf{U}$) triangular matrix where

$$\mathbf{X} = \mathbf{LU}$$

As these are triangular matrices we know the first row of $\mathbf{L}$ will take the form:

$$\begin{bmatrix} x & 0 & 0 & \dots & 0 \end{bmatrix}$$

With each subsequent row adding another value, left to right, in place of the zeros. By forwards substitution we can obtain a solution in the form:

$$\mathbf{L} \times \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \mathbf{B}$$

Calling this solution vector $\hat{\mathbf{y}}$ we can then solve $\mathbf{U}$ with back substitution for our actual solution $\hat{\mathbf{x}}$:

$$\mathbf{U}\hat{\mathbf{x}} = \hat{\mathbf{y}}$$

This process is not especially computationally intensive, therefore we can solve all required iterations of $\mathbf{A}$ in a relatively short space of time. The result of each iteration is checked for validity (no negative quantities) and given a 'score' based on the simplicity of the resulting compound (fewer atoms $\rightarrow$ better score, the actual complexity of the chemistry is ignored as we are computer scientists). After all iterations complete the results are written to a CSV file for the web server to import into the database.

2.1.2 Precursor Calculator

Most of the major deviations from the design are within the precursor calculator. We believe this is due to our initial unfamiliarity with the subject field, therefore as our familiarity grew we changed our methods to be both faster, and more effective.

The precursor and stoichiometry methods are initially much the same, we must form an $\mathbf{A}$ and $\mathbf{B}$ matrix that properly encode the linear equations we are attempting to solve. In this case we are presuming the input precursor compounds are charged balanced, therefore we can focus solely on the ratios of elements in the desired ratio and the precursor compounds.

$$\begin{array}{c} \begin{array}{ccccc} Li_2S & Al_2S_3 & Al_2O_3 & LiAlO_2 & Li_2O \end{array} \\ \begin{bmatrix} 2 & 0 & 0 & 1 & 2 \\ 0 & 2 & 2 & 1 & 0 \\ 1 & 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 2 & 1 \end{bmatrix} \end{array} \begin{array}{l} Li \\ Al \\ S \\ O \end{array}$$

The $\mathbf{A}$ matrix encodes the quantity of each element in the precursors, each column represents a precursor and each row a specific element. The $\mathbf{B}$ matrix is simply the desired output stoichiometry in a column vector, with the cosponsoring rows between the two matrices representing the same element.

We then solve the system of equations via LU decomposition as shown previously, however in this case we are not looking for a single solution as we have a potentially infinite solution space.

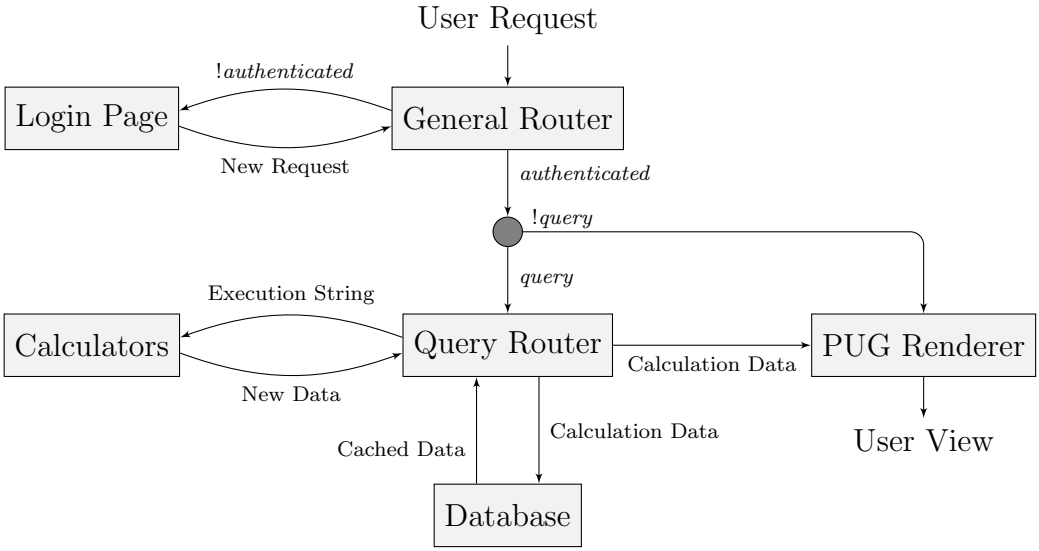
Therefore we take additional steps to generate more solutions from our first, this is done by first finding the null space of the **A** matrix, then diving the null space matrix into a set of vectors which when multiplied by arbitrary real scalars and added to our initial solution produce a new point within the solution space.

In earlier versions and in our initial design these arbitrary real numbers were generated systematically, then iterated over to produce all solutions. This system was very inefficient and contributed the bulk of the computation time. Whilst trying to optimise the speed of the computation we decided to drop this system in favour of a new dynamic system which will be described in detail in the realisation section.

To continue our stoichiometry example of Li, Al, S, O, and taking the **A** matrix shown above:

2.2 Web Server

The web server is written in NodeJS using Express to provide robust routing, and PUG to provide dynamic HTML generation. Given the highly asynchronous nature of NodeJS and our design specifically, we believe it is best to explain this section diagrammatically:



Here the italic text represents internal data (i.e. *authenticated* checks if the user has logged in, and *query* checks if the incoming request contains a query), and normal print represents an abstract piece of information. All boxes run asynchronously of each other, generally with callbacks to handle data transfer, however in some cases special functions are called to handle the async code (e.g. Calculators are called within a 'Promise' block).

2.3 Data Structures

The main source of data is the MongoDB database, and a pair of text files parsed by the calculators. We use the database to store all calculator results allowing us to lighten the computational load of the system and deliver results faster by first querying for stored data and only invoking the calculators if pre-existing results are not found.
The calculators share several classes which are used to store, process, and present data:

`Element`, is used to store all relevant data about each element we are considering in a specific query.

`Reagent`, is a fixed vector of elements representing a precursor in the query, it also contains a string field for the human readable chemical name.

`Solution`, a dynamic vector of doubles, used to store the raw unprocessed output from the algorithm.

Each of these classes then has a manager class which helps us access and use them in the wider context of the program:

`SolutionDB`, this maps instances of `Solution` to unique and predictable strings which act as keys. This allows us to quickly check the uniqueness of a new solution before we accept it into the output list, this is necessary as the generation of new solutions from the initial solution and null space is not guaranteed to be unique.

`ElementDB`, unordered map of element instances and strings which contain the chemical symbol of that particular element, ensures we only load each elements information once, and allows cross referencing between chemical symbols from the user input and in-memory data for the computation itself.

`ReagentDB`, mostly used for processes outside the main bulk of the computation.

`StateDB`, which has no corresponding child class, is a multimap (type of map which allows multiple values for one key) which stores the possible charge state(s) for each element. Keys take the form of the chemical symbol of the element, which is mapped to a set of integers representing possible charges.

2.4 Database

Collections in the Precursor and Stoichs database are generated on the fly when new calculator results arrive, their structure is not fixed but does follow a general pattern:

<Input Elements>			
<Element 1>	...	Mass	Score
<Ratio of E1>	...	<Compound Mass>	<Compound Score>
⋮	⋮	⋮	⋮

Table 1: Stoichiometry Calculator

<Desired Stoichiometry>			
<Precursor 1>		...	Score
<Ratio>	<Mass>	...	<Solution Score>
⋮	⋮	⋮	⋮

Table 2: Precursor Calculator

This dynamic structuring is required as we don't know the exact precursors that will be involved until a query arrives, the alternative would be a column for every element, and every allowable precursor which would massively inflate the tables with millions of 0 entries.

We chose to use a NoSQL database system which stores collections (tables) as JSON data as the only component which interacts with it directly is written in JavaScript. Therefore any data we read or write is already in a format JS implicitly understands and can be immediately used with little processing.

There is also an ElementData database which stores all of the information used by the calculators, although currently unused we decided to include it as the web server may find it useful in future iterations as it would allow us to display more information than Symbol and Z to the user, and it would also begin to allow the user to select which charge states they are interested in instead of the currently fixed version.

2.5 System Memory Management

On the computer which runs the main web server instance availability of RAM may become an issue, this would mainly be caused by multiple large queries executing simultaneously and such a situation is hard to avoid, however the memory footprint of the application is kept to a minimum by invoking the calculators as child processes instead of keeping them with the main NodeJS thread.

This allows the memory heavy segment of the system to only take up space in RAM when it is required, and also only occupy the exact amount it requires during computation. If resources become scarce then the child processes will wait for the OS to assign them memory, in future setting the OS priority for jobs may be necessary to ensure resources are still available for the web server when a queue of jobs has formed.

2.6 Interface Design

2.6.1 Web Server

The user interface is provided via the web server, we have used a single page style design where going further down the page gives you more in-depth data.

When a user first connects to the website they are asked to login with a simple username and password box. Account creation is not offered as the intended user base is small enough to have this done manually by the web admin, however automating this process would also be a simple task.

After successfully authenticating we present the user with a periodic table, an accompanying search bar, and a block of options. If the user selects elements on the periodic table they are auto-filled in the search bar and highlighted, the opposite is also true where if a user types elements into the search bar they are automatically selected on the periodic table.

The options presented allow the user to chose between displaying proportions, masses, or both in their results; filtering the number of samples displayed down from 1 to 100; and finally entering a desired mass of end product.

After pressing search the stoichiometry results will be displayed, and a new block of precursors will be shown. The list is pre-populated with common precursors used by chemists in the Material Innovation Factory, however there is a simple box to add more.

The user can then select a set of precursors, and then use the periodic table, search box, or stoichiometry results to generate precursor results. Our final design differs from the intended here as we decided to use a tabbed interface for different types of result as once the user has used the stoichiometry result they are likely not interested in it, so displaying it and making them scroll past it to get to the new precursor result was unintuitive.

Finally the precursor results will display in their tab, if the user has selected exactly 3 precursors a diagram will also be generated to display the points in a visual manner. However due to other technical challenges with the project we could not implement higher dimension visualisation in

time.

All of the webpages shown to the user are dynamically generated by PUG, this allowed us to write simple HTML-like templates which are then dynamically rendered into actual html and sent to the user.

2.6.2 Calculators

The interaction with the calculators is entirely command line based. Both calculators are combined into a single executable file which is controlled by command line arguments, once the calculator is invoked it requires no additional input from the user. This design is required by the web server as while it can send data via std::in the design is greatly simplified by passing command line arguments and simply waiting for the subprocess to return.

The available parameters are:

help — Displays a list of all possible parameters and their general function

stoichs — The input list of elements either as the desired final ratio, or to be turned into valid compounds

precursors — The precursors to be considered when trying to achieve the desired stoichiometry

samples — How many sample points should be generated in the solution space

mode — Toggle between Stoichiometry and Precursor calculations

dmass — The desired mass of final product (only relevant to precursor calculations)

drange — Maximum denominator to search for when calculating scores

debug — Debug flags toggle, additional text output and some safety features