ERYA – Bulk Technical Reference

Section 1. ERYA Bulk Description

ERYA Bulk is a program designed to evaluate the nuclear reaction yields using the theoretical framework derived first in [1], and implemented with two different software versions: a MS-DOS version, and later a Windows version that uses the LabView Runtime.

The current ERYA Bulk program was implemented on C++11 language, and uses the cross-platform framework wxWidgets to implement the GUI, and related handlers and events.
The major advantage of wxWidgets was the possibility to ease the port to several operating systems, since it acts as a wrapper library to make the code compile to several computer architectures. All current code, once linked with each wxWidgets libraries for each operating system, compile to several software architectures without change a single line of code.

Creating several native GUI implementations for the major operating systems (Linux, Windows, Mac OS X) would be too costly to maintain, and divert the focus on the main real problem, which is implementing a working scientific software with a GUI.
Also wxWidgets provides additional frameworks to uniform data structure handling, making the code more portable across several operating systems.

The ERYA-Bulk executable can be analyzed as a merge of three main modules:
1. The GUI handles and events for the several tools available to the user, using wxWidgets model framework. (That uses a chain of C++ classes to handle the windows, events, and the communication between all program code.)
2. A series of auxiliary routines (actually several C++ classes) dedicated to handle the native ERYA databases operations (create, delete, copy elements), and the some file formats (to load and save the ERYA database structures to files). Some additional file formats was also implemented as conversions tools, taking the necessary workarounds to parse the file into the Database Memory Structures.
3. The physical numerical evaluation routine is a module by his own right. It gather the user inputs, and the Databases loaded into memory since the data quantity are small, ERYA load the entire native Databases files into memory, in order to minimize the number of hard disc operations, that could slow down in some processes.

Section 2.1 The GUI Module

The GUI module of ERYA-Bulk are implemented according to the wxWidgets coding Standards. Each major dialog and their contents are defined by each own C++ class, and contains the function handlers to call to another C++ class that correspond to another window.
Apart of the standard wxWidgets 3.0.4 framework branch code, which is fully open-sourced, ERYA-Bulk only need a third-party plugin (a C++ class that uses the wxWidgets) to plot Cartesian graphs (wxMathPlot), avoiding the task to implement a similar one from scratch. [2]
If the author of ERYA Bulk [3] don't use a cross-platform framework like wxWidgets, it would require a different branch for the GUI code, including windows, events, handlers, and even native operating system syscalls (Also called Win32 API on Windows, Cocoa API on Mac OS X, and on Linux it could be X11 or Wayland on lower API side, and Qt or GTK on high API level.).
Since for each different operating system, would require a separate branch for each native operating system toolkit. Such doubling of tripling troublesome effort would increase the complexity of the new program, and probably the efforts would be redirected to a single platform, discarding the other ones.

Section 2.2 The Library Module

The next module, where all C++ files had the "Library" suffix", uses the wxWidgets framework libraries to create a series of specific routines to ERYA handles special kinds of data, offloading from the GUI components code.
The major two library modules are related to the Databases and the File Formats, that depends on other components that we detail later.

The **Databases classes** contains all methods and functions to handle their contents.
Taking the Elements Database, as an example, it is implemented as an overridden vector C++ class, where each individual register contains the raw contents of each physical characteristic of an element. Each Element register are identified by a name and gamma peak, and should store four physical quantities (What's their abundance, isotopic or atomic mass, and the atomic number).
An entire table of four columns (related to the experimental cross-section for each energy value, and the relative measurement errors) are also linked to the element register.

The vector class that handles the entire Element Database, contains a series of custom functions to ease several operations, such to find a element by a name or gamma peak, add new elements, delete some of them, and also to catch eventual errors (such as duplicates) in order to display to the user as error dialogs.

A similar vector class happens to the SRIM tables (actually the converted tables to the ERYA units), where each register contains a two columns table (for an experimental value for the stopping-power for each energy), and some additional data (such as the atomic number that corresponds to the element). A global overridden vector class takes the control of the entire array of SRIM tables, and takes the operations to find a table by his atomic number, add a new table, or check for duplicates, as prime examples.
The SRIM tables vector class is actually a private member of a single C++ class that handles the Ziegler's Parameters, where açso contains a matrix of values (for a table), and some additional information (such as the Ziegler's Version, or a custom macro code.)

The last singleton class are dedicated to store the contents of a Detector Efficiency, where contains a two column table for the Efficiency in terms of energy, and a custom macro for an algebraic expression.

The next important (and more complex in terms of code) library module is the **File Format Library** that contains the C++ classes and function to handle the actual loading and saving of Databases classes on memory to a storage device.
Actually it spans along several C++ files with special auxiliary functions linked to the main File Format Library, in order to make the implementation and debugging more efficient.

Since wxWidgets contains support to create XML documents (as their own C++ classes and methods), and XML files are an excellent choice to store text files with a formatted tree structure, the author of ERYA [3] decide to implement all native Database files as XML files.
The configuration file was also implemented as a XML file, but the functions to load and save a configuration file was placed on the GUI class that manage the main screen, in order to simplify the code.

Section 2.2.1 Compatibility with non-native File Formats

However, ERYA should support other file formats, in order to decrease the import time, and avoid manual conversion scenarios that could take too much time without a direct automation.

The File Format Library also contains functions to handle other formats, and was placed as an importation filter. Other files formats can be saved on that format, but the options depends on the current context.
ERYA don’t accept any Database that was not native (a recognizable XML document), as a primary database when the program starts, and requires a previous conversion using the tools provided by the GUI menus.

The files with plain ASCII format are the ones that ERYA can load and save without issues, as long are simple columns of numerical data. An auxiliary parsing routine derived from wxWidgets string functions makes the actual parsing of text files, line by line, and try to extract the relevant words. With the wxWidget’s string function classes, was trivial to check if a word is a number or a name, and select the correct ones by the context.
The next level of compatibility is ASCII files with special contents, such as the IBANDL files (with r33 extension). ERYA can parse an IBANDL file, once the ASCII file parser detects the signature of a IBANDL file, and extract not only the cross-section data, but some informations such as the target atomic number of mass, or the cross-section measurement units.
Due to the simplicity of the IBANDL file format, ERYA can save a single Database Element data, as a IBANDL file, but may require manual edition of some elements, since ERYA itself don’t store information such as the actual nuclear reaction, or the displacement angle of the detector that create the original file.
ERYA can also load and save the entire Element Database as a LabView Source file (a pure text file), where the load support is more advanced (to split elements with several gamma peaks to separate registers), than on save (it store elements with different gamma peaks into separate items).
The LabView Source file was a backup ASCII file created by the LabView Software of the original Database on binary form. The actual LabView Runtime ERYA version only operates on the binary form, that can load, update and save once compiled from the Source File.

The next ASCII special files are supported only as loading files, and cannot save on that format. The prime example is the SRIM Stopping-Power output files, that ERYA recognize when loads and tries to parse the necessary data, such as the atomic number of the target, the conversion factor, and the table itself. Since ERYA makes actual calculations to get the native ERYA units (eV*10^{15} atm/cm^2), the process cannot be reversed, and ERYA cannot store individual converted SRIM tables as SRIM output files.

Finally the last chain of compatibility, even on loading side only, is the support to load binary LabView files. Using third-party documentation, and some reverse-engineering, it was possible to implement (using the wxWidgets functions to generic binary files) two custom filters to load binary LabViews for the Elements Database, and the Ziegler’s Parameters matrix.
(The LabView ERYA don’t support the SRIM tables as an option.)
It is the ASCII file loading function that triggers the LabView Import once detects the magic “DTLG” header, and passes to the import C++ class to read the binary file registers (to find the file offset), and dumps the contents (as strings, integers or double float numbers on big endian hexadecimal format, that will be converted later) to the Database classes.
Later the same Database classes with the converted LabView data, or any ASCII sourced files, will call the File functions to store on other file formats.

More on the final development stage of ERYA, a new file format with loading and saving support was the **Microsoft Excel file** on xlsx file format (introduced on Office 2007).
It was a small challenge to the ERYA author [3], and requires some third-party documentation from [4] , and luckily wxWidgets provided the necessary requirements: support for XML documents, and ZIP file streams. An Excel file on xlsx is in reality a zip file with a strict structure of folders, and

XML files. Additional attachments such as images are also placed inside the zip file, but such advanced functionalities are not implemented on the ERYA Xlsx Library and then ignored.
The Xlsx class can read Excel files, but it will only read the first spreadsheet, and takes the contents (the Excel file delimits the actual storage data from the upper left cell to the lower right cell) to a matrix of strings.
Only cells declared as numbers, or strings (that will load another XML document inside the ZIP file with the strings), or binary values (That render the TRUE() and FALSE() functions on Excel itself) are recognized by ERYA. The transference matrix are later processed, line by line, and exported to the native ERYA Database functions.
ERYA can also store data as Excel files, but with a simple global format (much hardcoded than everything else), and with a single sheet. However, it can store numeric and text data.
The main advantage of a direct Excel file support is the easiness to fill tabular data, and the resulting Excel files created by ERYA can be edited on Excel to further work.

Section 2.2.2 ERYA Macro Language

An auxiliary library that deserve a special status is the **Parser and Macro Library**, where actually implements a small programming language for some advanced options on ERYA, like the Detector and Ziegler's Databases, or to support some user inputs.

Like some previous researchers on ITN [2] that implement some features as challenges, the ERYA macro language was designed initial to avoid constant recompilation of ERYA when the Detector's Efficiency function was changed.
This piece of software was starting developing when wxWidgets was an alpha software, and changing in terms of structure during the 1.x and 2.x betas, until reaches a final syntax format on 3.x betas. The macro language was finally workable on releasing 4.x versions.

On ERYA the macro feature resembles an runtime compiler, and internal stack interpreter. The syntax was inspired by the cheaper programmable calculators (CASIO and Texas Instruments), acting as a derivative of the Basic language, while the variable names follows the C standards.

When the ERYA program reads a string that should acts as a macro, such as the Detector's Efficiency custom function, it parses into a chain of words (tokens), and then convert all code into a postfix chain of commands (using the Shunting-Yard algorithm [5]). Once completed, the postfix chain passes into the stack interpreter (similar to the FORTH language logic), and runs the actual code on it, giving the results once ends.

Using special keywords as commands that assigns variables to special meanings (The assign "=" operator will add a variable with a numerical value to the variable stack of the interpreter, or the command stack will gather the variable with the special keyword), a macro can store a function on memory. Once the program code (specially the physical module) calls a macro function, and sends a variable as argument, the macro C++ class simply runs the code on the compiled postfix stack, along the command and variable stacks, to delivers the result as output (a double floating number).

Section 2.3 The Physics Module

The module responsible to evaluate the numerical physical simulation are handled by this module, and use several functions from the Library Module, and a basic Linear Algebra library to handle the Levenberg-Marquardt algorithm.

The simulation procedure C++ class handle the inputs and Databases, and coordinates all the processes until it gets a results, or delivers an error message.

When the whole simulation tarts, it check the validity of several inputs, and will abort with an error, if detects non-physical conditions, such a negative energy, a maximum energy lesser than the minimum or an non-numeric value on something else.

The next step is to build the sample, using several C++ classes, including the ones which are overridden vector classes. It starts when the original Elements, Detector and Ziegler's database are converted into numerical vectors and matrices.
The basic numerical element are the numerical vector that handles all user selected elements on the main screen, that combine the value of their Element Database, along the their Ziegler's Database, and the Detector's Efficiency for that element.
Also, any macro functions are compiled into stack arrays, in order to return quickly the values once the main program code calls it.
Once all numerical elements are compiled into several numeric elements, a vector of Elements are created.

Since the user can define Compounds, and set the fit attribute to some of the selected Elements, ERYA create two overridden vector classes to handle the additional constrains.
The Compounds classes handle a chain of Elements that share an unique Compound ID defined by the user on main screen. This will assign a fixed stoichiometry along the elements, and if this elements are assigned with the fitting procedure, their relative stoichiometry between them will never change.
An additional overridden vector class is responsible to handle the correct constrains between compound stoichiometry. Also it maps the correct element position in to the numerical matrices during the Levenberg-Marquardt algorithm.

At this step, ERYA already knew how many elements exists on sample, and how many elements are selected to be fit. If the number are actually zero, ERYA will evaluate the yields by a numerical integration method, using the variable step Simpson's Rule.
Taking the initial and final energy value, the sample are divided by equal portions, when each portion of the numerical integral are the energy step. (In many situations, 1 keV is enough.)
For simplicity, the stopping-power are evaluated at the beginning of each individual step.
Later the cross-section are evaluated by taking an interval between the current step and the next step boundary, and sum the partial cross-section values inside this interval range.
Once made the necessary conversions, and summed along the interval, a numerical approximation of the integral will give the theoretical yield.
Finally ERYA return those values to the screen, and ends the numerical procedure.

When a fitting procedure is activated, it starts the Levenberg-Marquardt algorithm, starting from the initial theoretical yield from the initial stoichiometry values, and taking the experimental yields in order to evaluate their difference.
Once started, it uses the Elements, Compound and Fitting Parameters vectorial classes to build numerical matrices needed to the fitting algorithm work. The ERYA implements the Levenberg-Marquardt algorithm using [6] as reference.
Once reached a suitable solution, it return the fitted stoichiometry and yields to ERYA display on the screen.

An optional profiling procedure that occurs once the fit ends, or when evaluate the yield if there's no fit, will evaluate the yields from the initial energy value to the final by the profiling step value.
The results are displayed on an additional table, accessible from the tabbed interface.

The user can store any results to a file, using the XML document format, or export the results to an Excel file, making use of the extensive Library routines implemented on ERYA.

Section 3. Main Algorithm Description

All essential routines will be described by a pseudo-code standard, making extensive use of partial routines to simplify the description.

Main Physics Routine (Main)
- Activated by the "Run" button event on GUI interface.
- It grabs the user inputs and databases loaded on memory to build the runtime data structures.

1. Checks the validity of several inputs and conditions
- Several physical conditions should be not negative values (Electric Charge, Energy Step, …), and the Energy range should be consistent ($E_{min} < E_{max}$).
- A failure to meet all requisites will abort the routine with an error.

2. Compile the Detector's Efficiency
- Should return the option flag.
2.1 – Convert the numerical table from Detector's Efficiency File into numerical vectors.
2.2 – Compile the algebraic function from Detector's Efficiency File, using the Macro Language feature.
2.3 – If the algebraic function is defined, then set the algebraic function option;
 Option = Algebraic.
 Else if the algebraic is not defined, then set the interpolation table option;
 Option = Table.
 Otherwise, abort with an error: "Undefined Detector's Efficiency".
 Stop.

3. Compile the Ziegler's Parameters
- It's a routine check to catch errors on the Ziegler's Efficiency File loaded on memory
3.1 – Get the Ziegler Option from source file
 - Get Option Flag
3.2 – Select case form Ziegler's Option Flag:
3.2.1 – Option Flag = Ziegler 1977
 If the Ziegler's Parameters are correct, then it's done, or get an error.
3.2.2 – Option Flag = Ziegler 1991
 If the Ziegler's Parameters are correct, then it's done, or get an error.
3.2.3 – Option Flag = Algebraic Function
3.2.3.1 - Compile the algebraic function using the Macro Language Feature.
3.2.3.2 – If the compiler don't find any error, then it's done;
 Otherwise, abort the main routine with an error. (Stop)
3.2.4 – Option Flag = SRIM
 If the SRIM tables are correct, then it's done, or get an error.
3.2.5 - Unknown flag: abort with an error. (Stop)

4. Compile the All User Defined Elements from the Sample
- It create an array of elements where the Elements Database File, and the user inputs from the GUI handled will be blended into a chain of numerical data.
4.1 – For Element ID = 0 to Total Number of Elements from the Sample (From GUI)
Note: ID = 0, 1, …, N – 1; where N is the size of the Element Sample
4.1.1 – Find the Database Position (Element Name and Gamma Peak) from the current Element ID.
4.1.2 – Create Sample Element object from Elements Databases File, and several user inputs like the Compounds ID, if the Elements had the fitting flag, the Ziegler's Parameters File and options, and Calibration factors.

4.1.3 – If any of internal numeric and data structures of the Numerical Sample Element class had any error:
- Will abort all main routine with an error, returned from the auxiliary routines that catch the first error.
4.1.4 – Prosed to the next Element ID

5. Create the auxiliary Compounds Array
5.1 – Copy the Sample Elements Array to the Compounds Array constructor to create this auxiliary memory object.
5.2 – If occurs any error, abort the main routine with an error.

6. Apply the Global Renormalization to all Elements and Compounds Stoichiometry
6.1 – Use the renormalization functions within the Sample Elements array and Compounds array in order to ensure that the total stoichiometry are equal to one.
6.2 – Once the stoichiometry are renormalized, create a third auxiliary array (Fitting Array) to lock the stoichiometry ratios between the compounds members, and also which elements had fitting flags actives. This will store the Elements ID as arguments to the Fitting Array, in order to map correctly during the Fitting Routine.

7. Apply the Simulation Routine
7.1 – Run the Yield and Fitting routines, described separately.
7.2 – Any error during all yield and fitting routines will return the initial yielding values, and prosed onwards.

8. Return the Yields and Fitted Values to the GUI handlers
8.1 – Copy all relevant data from the internal data structures from Sample Elements, Compounds and Fittings Arrays to the main GUI spreadsheet that represents the output values.

9. Create a Profiling Table
9.1 – If the Profiling Step is a positive number, then:
9.1.1 – For Energy = E_{min} until Energy = E_{max}, by Profiling Step values
9.1.1.1 – Evaluate all Yields with the fitted stoichiometry at Energy on the Current Value.
9.1.1.2 – Prosed to the next Energy value.
9.1.2 – Otherwise, skip this step.

10. Main Routine Completed!

Notes: Once some data objects such as the Detector’s Efficiency or the Ziegler’s Function (that aggregates both the Ziegler’s equations, or the interpolation from the SRIM tables) are compiled on the initial steps, the program will call them by a function call from the respective C++ class.
On this case, the program will call the public function by the energy (in keV units) as argument, and the C++ class will handle internally in order to give the function value.
Hard-coded functions, or the piecewise linear interpolation of numerical data are made within their function class methods.
Algebraic functions requires the help of another class that handles the Macro Language features, which requires a separate chapter to detail.

Section 4. Yield Evaluation Routine

The Yields are evaluated by a numeric integral that combines the Simpson's Method with some optimizations since the cross-section of any elements are an approximation by linear interpolation between two experimental values.
The basic equation are derived from the following integral:

$$Y(E_0)=\epsilon_{\text{abs}}(E_{\gamma})N_e f_m f_i A^{-1}\int_0^{E_0}\frac{\sigma(E)}{\epsilon(E)}dE \qquad (1)$$

Where the ε_{abs} is te Detector efficiency, where depends from the Element gamma emission.
N_e is the number of protons (electric charges), f_m are the mass fraction, f_i are the isotope abundance, and "A" are the element atomic mass.
Inside the integral (which are evaluated by numerical integration, using the user-defined Riemann step, where it is defaulted by 1 keV), it depends from the elements cross-section (σ), and stopping power (ε).
Since ERYA works on samples with multiple elements, it will obtain a chain of the same number of equations defined on (1).

Since the essential databases required to ERYA work are built around different units, and around several elements, it is necessary to adapt the equations to handle the unit conversion, otherwise it will get a meaningless result.

Let's suppose that exists **n** elements ordered by **j=1,2,…,n** and stoichiometries ranging from **s_1, s_2, …,** s_n where it would get the extended integral:

$$Y_j(E_0)=\epsilon_{\text{abs}}(E_{\gamma})_j N_e f_{mj} f_{ij}\phi M A_j^{-1}\int_0^{E_0}\frac{\sigma_j(E)}{\epsilon(E)}dE \qquad (2)$$

Where we introduce the φ factor, that is related to the unit conversion of all terms from (2):

$$\phi=\frac{N_A Q(\mu C)S(mb)m(\mu g)}{N_A F(b)e(C)}\approx 6241.5091258833 \qquad (3)$$

In this equation (3), N_A is the Avogadro's Number, e is the elemental electric change, and the other values are conversions from different units:

$Q=10^{-6}$ C, $S=10^{-27}$ cm^2 = 1 mb, $F=10^{-24}$ cm^2 = 1b, $m=10^{-6}$ g.

On ERYA code, the fundamental constants are coded by the following values:

$e = 1.6021766208*10^{-19}$ C

$N_A = 6.022140857*10^{23}$

And the actual (3) value are evaluated directly from this formula, by the program.

The program evaluate the mass fraction of the current element from the following expression:

$$f_{mj}=\frac{s_j m_j}{s_1 m_1+\ldots+s_j m_j+\ldots+s_n m_n} \qquad (4)$$

Where each "m" are the atomic mass of each individual element, where it should be already defined on the ERYA's Element Database. And "s" are the user-defined stoichiometry for each element.

The stopping-power are evaluated from the Bragg's rule, which also depends from the stoichiometry, as it follows:

$$\epsilon(E)=s_1\epsilon_1(E)+s_2\epsilon_2(E)+\ldots+s_n\epsilon_n(E) \qquad (5)$$

And each individual stopping-power are evaluate from the Ziegler functions publish by [2], or from linear interpolation from SRIM tables.

The last item is the molar mass “M”, which are evaluated from the usual formula:

$$M = s_1 m_1 + ... + s_n m_n \qquad (6)$$

In order to evaluate the integral, ERYA takes the integral intervals in terms of energy, from the user selected minimum to maximum energy value, and split the whole interval by equal spaced intervals defined by the user, which is the energy step.

Since the stopping-power varies too little between each step, it is taken out of the integral, and the whole (2) becomes a sum:

$$Y_j(E_0) = \frac{\epsilon_{\text{abs}}(E_\gamma)_j N_e f_{mj} f_{ij} \phi M}{A_j} \sum_{x=E_0}^{E} \frac{\sigma_j(x)}{\epsilon(x)} \Delta x \qquad (7)$$

Where the cross-section at each smaller interval are evaluated by the following expression:

$$\sigma_j(x) = \sum_{k=0}^{T} \frac{(x_{k+1} - x_k) * (\sigma(x_k + 1) + \sigma(x_k))}{2 \Delta x} ; k = 0,1,..., T : x \in [x_k, x_{k+1}] \qquad (8)$$

Where each intermediate point x_k belongs to an experimental value inside each subinterval between x_k and x_{k+1}.

This approach will combine the Simpson’s integral and the adjustable integration step, by selecting the experimental values from database.

The stopping-power are evaluated from the middle value between the interval boundaries.

The actual stopping-power and cross-section values are evaluated by the following expression, by taking the value between two experimental values, or got zero if the inputed variable are beyond the table data.

$$y = y_1 + \frac{(x - x_1) * (y_2 - y_1)}{x_2 - x_1} : x_1 < x < x_2 \qquad (9)$$

The same logic applies to the Detector’s efficiency, if the algebraic function are not defined.

As a final remark, if the sample had a finite depth (measured on $\mu g/cm^2$), the yield calculation simplifies to:

$$Y_j(E_0) = \epsilon_{\text{abs}}(E_\gamma)_j N_e f_{mj} f_{ij} \phi N_A F(b) A_j^{-1} \sigma(E_0) \Gamma \qquad (10)$$

And the conversion factor are slight different, apart of the factor there’s no actual integration, and the Γ are the sample depth density measured on $\mu g/cm^2$.

$$\phi N_A F(b) = \frac{N_A Q(\mu C) S(mb) m(\mu g)}{e(C)} \approx 3758.7247116320 \qquad (11)$$

Gathering all formulas together the algorithm to evaluate a theoretical yield are plain simple:

1. Evaluate all conversion factors derived on equation (3).

2. Evaluate the molar mass (6), and fraction mass (4) using the data from the Numerical Elements Array object.

3. Evaluate the terms outside the integral on (1), including the Detector’s Efficiency from the adequate C++ function class call (y=f(x)).

4. If the Sample Depth are a positive number (SD > 0), then:

- Define Y as the total yield, beginning with Y=0

4.1 – Evaluate the yield using the conversion (11) factor within the equation (10)

Else:
4.2 – For (E = Minimum Energy) until (E = Maximum Energy), by Energy Step (ΔE).
4.2.1 – Evaluate the cross-section using (8), by taking all experimental values between [E, E+ΔE].
- Get CS, from (8).
4.2.2- Evaluate the stopping-power using (5), where may depend from Ziegler's equations of interpolation of the SRIM tables, taking (9) formula.
- Get SP, from (5).
4.2.3 – Take the ratio of the two (4.2.1 and 4.2.2) formulas, and add to the sum defined on (7).
SY = CS / SP
Y = Y + SY
End If
4.3 - Return Y

Section 5. ERYA Macro Language Routines

This feature was implemented as an auxiliary library, and makes possible to the user code a simple program (even lacks several features, such as control flow, or code branches), as ERYA will compile to a intermediate state that simplifies the user-defined code.

When ERYA pass the initial program as a string, it will transform the initial string as a chain of words (tokens), then convert to a more optimized state (using the Shunting-Yard algorithm), making more easily to run the actual code.
Once completed, ERYA can grab the numerical result, or store the compiled chain of tokens on the correspondent algebraic Function object class, in order to the program numerical routines when call the compiled function, it could insert a input, and get an output by running the compiled code.
Basically this is a Run-Time Compiler type of Interpreter, where the entire code are converted prior to the actual execution. Due to several restrains, this feature was only made to implement simple functions, and not intended to made complex programs.

Step 1: Tokenization
- Get the initial String, that had a total of N characters.
- Define a temporary string, called: Cache.
1. For I=0 to String Size (N)
1.1 – Define the following auxiliary strings:
Character = String(I)
TestChar = Character
TestCache = Cache + Character
TestNumber = Cache + Character + "0"
- Notice that the sums signs are actually string concatenation.

- Now start a chain of If and Else If, to test the actual character, taking attention of the caches:
1.2 – If Character = " ", or "\t", or "\n" (Empty strings, or blank spaces)
1.2.1 – If Cache Size > 0, then store Cache as new Token, and Clear Cache.
- Blank characters are interpreted to store the previous cached strings to a new Token, and then clear any temporary string caches.

1.3 – If Character = "*", "/", "^", ":", "<", ">", ",", "(", ")" [Any single-character symbol]
1.3.1 – If Cache Size > 0, then store Cache as new Token, and Clear Cache.
1.3.2 – Store Character as new Token.
Note: Single character tokens are the easy part when writing an interpreter or parser, and this are implemented on first place. The real challenge happens with the tokens had several characters, or some characters will have several meanings.

1.4 – If Character = “-” or “+” [This can be an arithmetic operator, or part of a number]
- Define the auxiliary string:
 NextChar = String(I+1), if I is not the last character.
1.4.1 – If Cache Size > 0, Then:
1.4.1.1 – Define the following auxiliary string:
 TestSci = Cache + Character + NextChar [A number of scientific notation]
1.4.1.2- If is a valid number on Scientific Notation, then:
 Cache = Cache + Character (Concatenate the current character to the cache string).
 Else:
 Store Cache as new Token, and Clear Cache.
1.4.2 – Else If for Cache Size = 0 [It can be a genuine arithmetic operator, or a number sign]
1.4.2.1- Define the following auxiliary string:
 Test Number = Character + Next Number
1.4.2.2 – If Test Number is a Number Token , Then
1.4.2.2.1 – If the Token Stack Size > 0, Then:
1.4.2.2.1.1- If the Last Token Stack had Priority > 0, or is a “(“”, then:
 Cache = Cache + Character (Concatenate the current character to the cache string).
1.4.2.2.1.2- If the Last Token Stack is a “)”, or when is a valid number of symbol (see 1.3),Then:
- Clear Cache, since it’s an arithmetic operator.
- Add the current Character as a new Token
Else:
Cache = Cache + Character (Concatenate the current character to the cache string).
1.4.2.2.2- Else from Token Stack Size > 0
Cache = Cache + Character (Concatenate the current character to the cache string).
1.4.2.3 – Else from Test Number, then:
- Clear Cache, since it’s an arithmetic operator.
- Add the current Character as a new Token.
Note: This delicate routine was intended to parse negative numbers as a single token, since the macro language was designed to avoid unary operators.

1.5 – If Character is a Valid Variable Name or Number
Cache = Cache + Character (Concatenate the current character to the cache string).

1.6 – If all previous cases fail, then return an Error.
-The process abort with an unexpected error, such an invalid character type.

2. Once completed the cycle, and remains Cache Size > 0.
- Add the Cache as a new Token.

3. Classify all the Tokens on their Stack. If any of them are an invalid token, stop the process, and return the adequate error code.

4. End of Tokenization.
Note: The classification process will determine if any token is a number, a valid variable name (using the C standard, where the first character is a letter, and the other can be numbers and letters), or reserved words for commands and functions.
- The reserved words for commands and attributes are three: fxvar, fyvar, fnvar.
- On ERYA-Profiling, more two attributes are added: fxmin, fxmax.
-There’s 16 reserved words for the functions: asin, asinh, sin, sinh, acos, acosh, cos, cosh, atan, atanh, tan, tanh, exp, log, ln, sqrt.
- The valid operator symbols are: “+”, “-”, “*”, “/”, “^”, “(“, “)”, “<”, “>”, “:”, “,”

The Precedence Attributes of Tokens

Token Type	**Priority Order**	**Associativity**
Any Variable, Number, Symbol	0	Left
Assignment Operator “=”	1	Left
Less “<”, More “>”	2	Right
Sum “+”, Minus “-”	3	Left
Product “*”, Division “/”	4	Left
Power “^”	5	Right
Functions	6	Left

Step 2: Shunting-Yard Algorithm

The next phase of the interpreter is to convert the chain of tokens to the postfix notation using the Shunting-Yard algorithm [7].
The whole process grabs the input stack of tokens, and copy the contents to the output stack, while use an auxiliary stack of tokens called the operator stack, where the last member is the first to go away (LIFO buffer).

1. For I=0 to Input Stack Size (N)

1.1 – If Input(I) is a Number, Symbol or Attribute, Then:

- Copy the Token to the Output.

1.2 – If Input(I) is the “=” operator or function, Then:

- Copy the Token to the Operator Stack.

1.3 – If Input(I) is the comma “,”

- Pop the entire Operator Stack to the Output.

1.3.1 – If exists a parenthesis on the Operator Stack, abort with a mismatched parenthesis error.

1.4 – If Input(I) is the comma “:”

- Pop the entire Operator Stack to the Output.

1.4.1 – If exists a parenthesis on the Operator Stack, abort with a mismatched parenthesis error.

1.4.2 - Copy “:” Token to Output.

1.5 – If Input(I) is an Operator, and the Operator Stack is Empty, then:

-Copy the Token to the Operator Stack.

1.6 – If Input(I) is an Operator, and the Operator Stack is Not Empty, then:

1.6.1- While the Operator Stack is Not Empty

1.6.1.1-If the Token is left associative, and the Last Operator Stack Priority are equal or greater than the Token Priority, Then:

- Pop the Last Operator Stack Token to the Output.

1.6.1.2- Else If the Token is right associative, and the Last Operator Stack Priority are greater than the Token Priority, Then:

- Pop the Last Operator Stack Token to the Output.

1.6.1.3-If the previous conditions not meet, stop the cycle.

1.6.2 – Copy the Token to the Operator stack.

1.7 – If Input(I) is the “(” symbol, Then:

- Copy the Token to the Operator Stack.

1.8 – If Input(I) is the “)” symbol, Then:

1.8.1- While the Operator Stack is Not Empty.

1.8.1.1-If the Last Operator Stack Token is a “(“ parenthesis, Then:

- Delete the “(“ from Operator Stack, and stop the cycle.

1.8.1.1.1-If the Next Operator Stack is a Function, then:
- Pop the Function Token to the Output.
1.8.1.2 – If the Last Operator Stack is “=”, or “,”, or “:”, Then:
- Abort the procedure due to a syntax error.
1.8.1.3- Elsewhere, pop the Last Operator Stack Token to the Output.
1.8.2 – If the entire Operator Stack don’t contains a parenthesis “(“, abort the procedure due to mismatched parenthesis.
1.9 – If Token(I) is not any of previous options, Then:
- Abort Procedure due to unknown token type.

2 – Pop any remaining tokens on Operator Stack to the Output.
- Notice that presence of “(“, “:”, or “,” symbols will trigger a syntax error, and the cancellation of this procedure.

3. End of Shunting-Yard Algorithm

Step 3: Postfix Compiler

The final major step of the compilation stage is to run the postfix code, actually a kind of compiled program, in order to check if generates a correct result. ERYA implements two similar routines: this one to check the code and run it in order to assign all numeric values on variables or any attributes on them, and store them on the Variables and Attributes arrays.
The second version is a trimmed down postfix calculator that only update any variables at all, not create ones.
Since the code was intended to follow a standard pattern: declaration of variables, any constants, and the algebraic function itself; the last part are cached to a Function Stack.
That Function Stack are used by the second postfix algorithm to evaluate function values, when the code insert a numeric value for the independent variable, and calls the postfix algorithm to deliver the dependent variable of the coded function.
On both versions, the postfix uses a temporary Register Stack (made with the tokens objects) where numerical values are placed. Normally the temporary tokens are placed by appending them in direct sequence, but always take the last ones when an operator token are applied from the postfix stack.

1. For I=0 to Postfix Stack Size (N)
- Save the Postfix(I) to the Function Stack
1.1 – If Postfix(I) is a Number, Then:
Create a Numeric Token with their numeric value on Register Stack.
1.2 – If Postfix(I) is a Variable, Then:
Create a Variable Token on Register Stack.
1.3 – If Postfix(I) is an Attribute, Then:
Create an Attribute Token on Register Stack.
1.4 – If Postfix(I) is the “=” symbol, Then:
1.4.1- If Register Stack Size > 1, Then:
- Set Register Stack Last as α Token, the the Next Last Token as β Token.
1.4.1.1- If β Token Exists on Variable Stack, Then:
1.4.1.1.1- If the β Token is a Variable linked to an Attribute, Then:
- Is a Variable, and a new Variable Stack can be created/updated, with the numerical value from α Token.
- The α and β tokens are deleted from Register Stack.
Else: Stop, due to a Redefinition Constant Error.
1.4.1.1.2- If the β Token an Attribute, Then:
- Stop, due to a Redefinition Attribute Error.

1.4.1.1.3- If the β Token a Variable, Then:
- A new Variable Stack can be updated, with the numerical value from α Token.
- The α and β tokens are deleted from Register Stack.
1.4.1.1.4- If the β Token fail to classify:
- Stop, due to an incorrect syntax format.
1.4.1.2 – Else if the β don't exists on Variable Stack, Then:
- Set Register Stack Last as α Token, the the Next Last Token as β Token.
1.4.1.2.1 – If the β Token is an Attribute Token, Then:
1.4.1.2.1.1- If β = "fxvar " or "fyvar", and α is not a Numerical Token, Then:
- Check the Token α name's exists on the Variable and Attributes Stacks.
- If not exists, then adds the α Variable Name on Variables Stack, and the β Attribute name on the Attributes Stack.
- The α and β tokens are deleted from Register Stack.
† A Numeric Token on α will stop everything with an Invalid Argument error.
- Else, stop due to an attempt to overwrite the previous defined attribute.
1.4.1.2.1.2- If β = "fnvar", and α is not a Numerical Token, Then:
- Check the Token α name's exists on the Variable and Attributes Stacks.
- If not exists, then adds the α Variable Name on Variables Stack, and the β Attribute name on the Attributes Stack. Creates additional 15 variables tokens on the Variables Stack, where their names is the original name defined by α Variable Name, appended by a number from "1" until "15".
- The α and β tokens are deleted from Register Stack.
† A Numeric Token on α will stop everything with an Invalid Argument error.
- Else, stop due to an attempt to overwrite the previous defined attribute.
1.4.1.2.1.3 – Otherwise:
- Get the numerical value from the α Token Variable Name, or from a Numerical Token.
† If the variable name declared on α Token don't exists, then: Stop with an Undeclared Variable error.
- Add the Variables Stack with β Token Name, with the numerical value delivered from α.
- The α and β tokens are deleted from Register Stack.
1.4.1.2.2- If the β Token is a Variable Token, Then:
- Get the numerical value from the α Token Variable Name, or from a Numerical Token.
† If the variable name declared on α Token don't exists, then: Stop with an Undeclared Variable error.
- Add the Variables Stack with β Token Name, with the numerical value delivered from α.
- The α and β tokens are deleted from Register Stack.
1.4.1.2.3 – If the β Token don't match the previous cases, Then:
- Stop, due to an invalid argument error.
1.4.2 – If the Register Stack Size < 2, this means that:
- Stop, due to an incorrect number of arguments.

1.5 – If Postfix(I) is the ":" symbol, Then:
1.5.1 – Clear the Function Stack.
1.5.2 – Check the Register Stack Size:
1.5.2.1 – If the Register Stack Size = 0, Then:
- Set the Output Value = 0.
1.5.2.2 – If the Register Stack Size = 1, Then:
1.5.2.2.1 – If the Last Register Stack Token is Numeric, Then:
- Set the numerical value of that Token to the Output Value.
- Clear Register Stack.
1.5.2.2.2- If the Last Register Stack Token is a Variable, Then:
1.5.2.2.2.1 - If the Variable Exists on Variable Stack, Then:
- Set the numerical value of that Variable Token to the Output Value.

- Clear Register Stack.
1.5.2.2.2.2 – Otherwise:
- Stop, due to an Undefined Variable Error.
1.5.2.2.3 – If the Last Register Token are an Attribute:
- Stop, due to a misplaced attribute operator.
1.5.2.2.4 – If the Last Register Token are other kinds:
- Stop, due an invalid expression.
1.5.2.3 – If the Register Stack Size > 1, Then:
- Stop, due to an invalid expression.

1.6 – If Postfix(I) is a Function Token, Then:
1.6.1 – Register Stack Size >= 1, Then:
- Define the Last Register Token as a α Token.
1.6.1.1 – If the α Token is a Variable Token, Then:
- Find and replace the α Token by is numerical value from the Variable Stack.
† In case of failure, stop will an Undefined Variable error.
- Evaluate the Function Token, and get it's value.
- Replace the α Token by the new Numerical Token that was the function evaluation.
1.6.1.2 – If the α Token is a Numerical Token, Then:
- Evaluate the Function Token, and get it's value.
- Replace the α Token by the new Numerical Token that was the function evaluation.
1.6.2 – Register Stack Size < 1, Then:
- Stop, due to a wrong number of arguments to a function.

1.7 – If Postfix(I) is an Operation Token, Then:
1.7.1 – Register Stack Size >= 2, Then:
- Define the Last Register Token as a α Token.
- Define the Next Last Register Token as a β Token.
1.7.1.1 – If the α Token is a Variable Token, Then:
- Find and replace the α Token by is numerical value from the Variable Stack.
† In case of failure, stop will an Undefined Variable error.
1.7.1.2 – If the β Token is a Variable Token, Then:
- Find and replace the β Token by is numerical value from the Variable Stack.
† In case of failure, stop will an Undefined Variable error.
1.7.1.3 – Once rectified the α and β Tokens, Then:
- Evaluate the Operation Token, and get it's value.
- Delete the α and β Tokens from Register Stack.
- Add a new Numerical Token on Register Stack, that was the numerical result from the arithmetic operation.
1.7.2 – Register Stack Size < 2, Then:
- Stop, due to a wrong number of arguments to a function.

1.8 – If Postfix(I) is an unknown Token, Then:
- Stop, since found an illegal symbol.

2. Once completed the cycle, check if the Register Stack Size > 0
2.1 – If the Register Stack Size = 0, Then:
- Set the Output Value = 0.
2.2 – If the Register Stack Size = 1, Then:
2.2.1 – If the Last Register Stack Token is Numeric, Then:
- Set the numerical value of that Token to the Output Value.
- Clear Register Stack.

2.2.2- If the Last Register Stack Token is a Variable, Then:
2.2.2.1 - If the Variable Exists on Variable Stack, Then:
- Set the numerical value of that Variable Token to the Output Value.
- Clear Register Stack.

2.2.2.2 – Otherwise:
- Stop, due to an Undefined Variable Error.

2.2.3 – If the Last Register Token are an Attribute:
- Stop, due to a misplaced attribute operator.

2.2.4 – If the Last Register Token are other kinds:
- Stop, due an invalid expression.

2.3 – If the Register Stack Size > 1, Then:
- Stop, due to an invalid expression.

3. Once reached without errors, the initial string are converted by an array of operators and commands, forming a kind of computer code. END

Once completed all major three steps, it should maintain on memory:
- The initial string, as a reference.
- A Function Stack, that was a copy of the initial postfix array, trimmed after the last ":" operator.
- An Attribute Stack
- A Variable Stack.
- A numerical Output Value.

The numerical value of Output Value, is accessible from the public function AlgebraicFunction::GetValue(), and usually corresponds to expressions that was intended to use a pure arithmetic expressions, that is used on some features of ERYA interface.

Special Step: Postfix Calculator
- When the code calls the function double y = AlgebraicFunction::GetFyxValue(double x)

1. Find the variable name defined by "fxvar" and "fyvar" attributes.
1.1 – If fail to find any of those variables, Then:
- Delivers an error code related to Undefined Function, that causes any ERYA calculations to stop.

2. Pass the **x** numerical value to the variable defined by "fxvar".

Runs the Postfix algorithm, where the Function Stack are stored as a Program
3. For I=0 to Function Stack Size (N)
3.1 – If Function(I) is a Number, Variable, Attribute or Symbol, Then:
Copy Token to the Numeric Stack.

3.2. – If Function(I) is the "=" symbol, Then:
3.2.1- If Register Stack Size >= 2, Then:
- Set Register Stack Last as α Token, the the Next Last Token as β Token.

3.2.1.1 - If the α Token is a Variable Token, Then:
- Find and replace the α Token by is numerical value from the Variable Stack.
† In case of failure, stop will an Undefined Variable error.
3.2.1.2 – Copy the α Token numerical value to the β Token Variable Name.
† In case of failure, stop will an Undefined Variable error.
3.2.2 – If the Register Stack Size < 2, Then:
- Stop, due to a wrong number of arguments.

3.3 – If Function(I) is a Function Token, Then:

3.3.1 – Register Stack Size >= 1, Then:
- Define the Last Register Token as a α Token.

3.3.1.1 – If the α Token is a Variable Token, Then:
- Find and replace the α Token by is numerical value from the Variable Stack.
† In case of failure, stop will an Undefined Variable error.
- Evaluate the Function Token, and get it’s value.
- Replace the α Token by the new Numerical Token that was the function evaluation.

3.3.1.2 – If the α Token is a Numerical Token, Then:
- Evaluate the Function Token, and get it’s value.
- Replace the α Token by the new Numerical Token that was the function evaluation.

3.3.2 – Register Stack Size < 1, Then:
- Stop, due to a wrong number of arguments to a function.

3.4 – If Postfix(I) is an Operation Token, Then:
3.4.1 – Register Stack Size >= 2, Then:
- Define the Last Register Token as a α Token.
- Define the Next Last Register Token as a β Token.

3.4.1.1 – If the α Token is a Variable Token, Then:
- Find and replace the α Token by is numerical value from the Variable Stack.
† In case of failure, stop will an Undefined Variable error.

3.4.1.2 – If the β Token is a Variable Token, Then:
- Find and replace the β Token by is numerical value from the Variable Stack.
† In case of failure, stop will an Undefined Variable error.

3.4.1.3 – Once rectified the α and β Tokens, Then:
- Evaluate the Operation Token, and get it’s value.
- Delete the α and β Tokens from Register Stack.
- Add a new Numerical Token on Register Stack, that was the numerical result from the arithmetic operation.

3.4.2 – Register Stack Size < 2, Then:
- Stop, due to a wrong number of arguments to a function.

3.5 – If Postfix(I) is an unknown Token, Then:
- Stop, since found an illegal symbol.

4. Once completed the main cycle:
4.1 – If the Register Stack Size = 0, Then:
4.1.1 – Find the variable name related to the “fyvar” attribute.
† In case of failure, stop will an Undefined Function Variable error.
4.1.2 – Find the numerical value of the Variable Name associated to “fyvar” attribute.
† In case of failure, stop will an Undefined Function Variable error.
4.1.3 – Define the variable **y** as the numerical value found on last step, and return:
return y (END)
4.2 – Stop: Syntax Error on the function itself.

The Big Picture on Macro Language.
1. Compile the initial macro code, using the Tokenization, Shunting-Yard and Postfix Compiler algorithms.
2. If the main code calls a numeric expression (AlgebraicFunction::GetValue())
2.1 – Return the numeric value. Ans = AlgebraicFunction::GetValue()
3. If the main code calls a function expression (AlgebraicFunction::GetFyxValue(x))
3.1 – Runs the Postfix Calculator algorithm, and returns y = AlgebraicFunction::GetFyxValue(x)

Section 6. ERYA Fitting Algorithms

ERYA should answer the following basic question: Taking several experimental yields from the experimental gamma rays spectrum, once associate correctly to a series of elements, what’s their real chemical composition ?

Since any element and their gamma emission from sample can be measured from their nuclear reaction yield, the only variable parameters of the chain of yields are their stoichiometry.

Taking an initial stoichiometric (based from the theoretical chemical formula, or by a simple guess), where by (2) will get the initial yield, then by vary the stoichiometric values, the gap between the evaluated yields by the changing stoichiometric variables and the experimental values will get a minimal value. Finding the minimum of the gap is the prime objective of every minimal square problem, where in your case, the function to be minimized (actually the following residue function) is:

$$f(x)=\frac{1}{2}\epsilon^T\epsilon=\frac{1}{2}\|y_{gs}(E_0)-y_{th}(s_1,s_2,\ldots,s_n,E_0)\| \qquad (12)$$

The current ERYA-Bulk program also handle the possibility to assign fixed stoichiometric ratios to a group of particular elements on the sample. This happens when the sample had a well known chemical composition, and then a well known stoichiometry. Fitting the whole compound and another elements should take it on account.
The previous LabView version don’t have such feature, and fit all elements as independent ones.

This restraint means that any compound with fixed stoichiometric ratios are a single fitting variable, since changing the stoichiometry of an element that belongs to a compound should vary by same proportion the other elements. Isolated elements are also independent fitting parameters.
Implementing such functionality was done on actual ERYA, using a special C++ class to force additional restrictions on the fitting procedure.

Suppose the sample have a cluster of elements belonging to a particular well-defined chemical compound, then their stoichiometry will be:

$$e.g.\,A_xB_yC_z\ldots \qquad (13)$$

Where their stoichiometry ratios will be fixed:

$$\frac{y}{x}=\lambda_1;\frac{z}{x}=\lambda_2;\ldots \qquad (14)$$

The program will assign the stoichiometry as:

$$\ldots,s_x,s_y,s_z,\ldots=s_x,\lambda_1 s_x,\lambda_2 s_x,\ldots \qquad (15)$$

This means if on an experiment, we detected 10 elements, where 5 of them belongs to a compound, 3 belongs to another compound, and the remaining 2 are isolated, then we have 10 yields and 10 stoichiometries, but we only have 4 independent fitting parameters. Essentially, in this particular case, ERYA only needs to manage 4 independent variables, even it needs to fit 10 yields.

To describe the fitting algorithm with some legibility, the pseudo-code list adds some auxiliary routines, where on main routine are named by his alias.

†S1: Evaluate the Residue Function
Taking the vector **x** as the independent fitting stoichiometry parameters, and vector $\tilde{\mathbf{x}}$ for all fitting stoichiometry parameters, using (14) and (15) formulas:
It’s possible to evaluate the following vector **ε** function, defined by:

$$\boldsymbol{\epsilon}=\|Y_{gs}(\tilde{\mathbf{x}})-Y_{\exp}\| \qquad (16)$$

The experimental yields are values inputed by the user on the “Experimental Yield” column, while the guess yield, or computed yield, depends from the current stoichiometric values from $\tilde{x}$ evaluated by the integral defined on (2).
The user only inputs the initial stoichiometric values, and the $\tilde{x}$ values will change during the fitting process.

†S2: Evaluate the Jacobi Matrix
It should evaluate the numerical Jacobi matrix **J**, which size can be defined by the initial conditions. At any iteration step, the matrix size are constant, and so all numerical matrices are static allocated by a C++ class using array pointers, due to be more fast to allocate dynamic memory using dynamic memory pointers. This is the only exception of this program that use the Standard Template library and wxWidgets objects to dynamic arrays, since the main reason was the speed constraints on memory allocation was very important.
The number of columns of the Jacobi Matrix is the number of independent fitting parameters, while the number of rows are the number of yields and also the number of total elements select by the user on the main spreadsheet tab that define the whole sample.
Only when the number of fitting parameters are equal to the number of elements, the matrix is square. Fortunately, the whole process work with a rectangular matrix.

Each matrix cell are evaluated by a symmetric difference algorithm to approximate the derivative around the current function point (actually the current stoichiometry at a particular **x** value), using the following expression:

$$J_{ij}=\frac{\partial Y_i(x_j)}{\partial x_j}e_i\otimes e_j\approx\frac{Y_i(x_j+h)-Y_i(x_j-h)}{2h}e_i\otimes e_j : h\approx 10^{-9} \qquad (17)$$

Each yield are evaluated from the integral defined on (2), and the mixture of a numerical integral and a numerical derivative, makes this routine more time to complete that another ones.

†S3: Evaluate the Hesse and Gradient Matrix
Since the Jacobi matrix defined on S2 routine is rectangular in general, as the number of independent fitting parameters is equal or less than the number of experimental yields, it is necessary to take the second-order derivative to form a square matrix. Luckily, it is a simple matrix multiplication.
The Hesse matrix is matrix product of the self-adjoint matrix with the original matrix:

$$H=J^T J \qquad (18)$$

While the Gradient matrix (actually a vector), is the matrix product of the self-adjoin Jacobi matrix, with the residue vector **ε** defined on S1 routine:

$$g=J^T \epsilon \qquad (19)$$

†S4: Fitting Renormalization Step
This process is done automatically by the C++ function **ElementVector::Renormalize()**, and the alias on **CompoundVector::Renormalize()**, and hides a sightly complex algorithm to ensure a correct renormalization at each iteration step.
Since the sum of all stoichiometries are equal to one, it contains “n” non-fitting elements (the ones without “Fit” mark), and “m” fitting elements. The **x** vector already contains the independent fitting parameters, and using the **FittingParametersVector** methods, it is possible to compute all $\tilde{x}$ fitting parameters, leaving a vector **y** for the non-fitting parameters.

To renormalize only the fitting parameters, it requires the following steps:
1: Define a non-fitting constant “T”, evaluated by the sum of non-fitting elements of vector **y**:

$$T=1-\left(\overline{s_1}+\ldots+\overline{s_n}\right)=1-\sum \boldsymbol{y} \qquad (20)$$

2: Define a fitting constant “S”, evaluated from the fitting elements of vector $\tilde{x}$, using the ratios defined on **FittingParametersVector** methods, when necessary:

$$S=s_1+...+s_m=\sum\tilde{x}=\underbrace{s_1+s_1\lambda_1+s_1\lambda_2+...+s_2+...}_{\#m} \qquad (21)$$

3: Each new independent fitting parameters from vector **x,** and obviously from all fitting parameters from vector $\tilde{x}$, can now be redefined by the expression:

$$\overline{s_j}:=T\,s_j/S,\ j=1,...,m \qquad (22)$$

4: Once reevaluated the stoichiometric values on vector **x**, that only affects the “S” constant, when apply (21) again, it should give now the total sum S+T=1, since the renormalization is done.

†S5: Solving a System of Linear Equations by the Gauss Elimination with Partial Pivoting

This is a classic algorithm designed to direct solve a system of linear equations with n equations with n unknowns, returning the vector with the solutions.

To turn the algorithm description more simple to read, two critical steps was implement as subroutines to their own on Matrix C++ class.

T1 – Swap Rows of a Matrix.

```
- Change the contents of the row “i” to the row “j”, and vice-versa.
1. Copy the contents of Row “i” to a temporary row “temp”.
2. Copy the contents of Row “j” to the Row “i” (overwrite).
3. Copy the contents of row “temp” to Row “j”
END
```

T2 – Reduce the Rows by a Pivot.

```
- From the initial Row”i”, reduce the Row “k”, by the pivot value.
1. If i<k , Then:
1.1 – Define the pivot constant μ, by the formula: $\mu=-1*A_{ki}/A_{ii}$
1.2 – For Column j = 1 until Total Number of Columns N
1.2.1 – Evaluate: $A_{kj}:=A_{kj}+\mu*A_{ij}$
- Rarely used, but implemented.
2. If i>k , Then:
2.1 – Define the pivot constant μ, by the formula: $\mu=-1*A_{ii}/A_{ik}$
2.2 – For Column j = 1 until Total Number of Columns N
2.2.1 – Evaluate: $A_{ij}:=A_{ij}+\mu*A_{kj}$
- Once completed...
3. END
```

Main Gauss Routine: Solve Ax = b matrix equation.

```
1. Join the original matrix A, and the vector b in a single matrix Ab (Add an additional column)

2. For the Column k = 1 until Total Number of Rows of Ab “R”
2.1: For the Column l = k until Total Number of Rows of Ab “R”
2.1.1: If $|Ab_{lk}|>|Ab_{kk}|$ or $Ab_{kk}=0$ , Then:
- Apply algorithm T1 to swap Rows “l” to “k”.
End For 2.1
2.2: If $|Ab_{kk}|>0$ , Then:
```

2.2.1: For the Column m = k until Total Number of Rows of Ab "R"
- Apply algorithm T2 to reduce Rows "m" to "k".
2.2.2: Else, abort the algorithm with an error: "Singular Matrix". (**FAIL**)
End For 2

3. Starts backward substitution
3.1: For Column u = R, which is the Total Number of Columns of Ab, until decreases to u=1.
- u=R, R-1, …, 2,…, 1
- Define the temporary variable $\mu=0$.
3.1.1: For Column v = u, until v = R, which is the Total Number of Columns of Ab
- Sum the following: $\mu:=\mu+x_v * Ab_{uv}$
End For 3.1.1
3.1.2: Evaluate: $x_u=\frac{Ab_{uR}-\mu}{Ab_{uu}}$
End For 3.1

4: Return the solutions (vector **x**)

†The Main Levenberg-Marquardt algorithm

1. [Initial Setup]
1.1: Define the convergence parameters: e1,e2,τ,kmax from the user-interface or default values, signaled on bold.

$$e_1=10^{-\alpha}; e_2=10^{-\beta}; \tau=10^{-\gamma} \quad (23)$$
$$\alpha,\beta\in 0,\dots,\mathbf{3},\dots,6; \gamma\in 0,1,2,\mathbf{3} \quad (24)$$
$$kmax=0,1,\dots,\mathbf{100} \quad (25)$$

1.2: Get the initial residue function using **S1**, and the independent fitting parameters vector **x** (stoichiometry). Also create an additional vector of integers required as a constructor of the **FittingParametersVector** methods, to map correctly the Element position ID from the main spreadsheet screen, defined by the user, to all user-defined constraints.
This auxiliary map is needed to place correctly from all numerical matrices from which row or column belongs to a certain element.

1.3: Evaluate the Jacobi **J**, Hesse **H** and Gradient Matrix **ε**, using the **S2** and **S3** routines.

1.4: The initial damping parameter are evaluated from the expression:

$$\mu=\tau\, max|A_{ii}| \quad (26)$$

Where it is the maximum value along the module of each matrix cell from the main diagonal, multiplied by an additional user-defined damping factor τ, from (23).

1.5: Verify if meet the zeroth step gradient converge criteria:
if($||g||_\infty<e_1$ **) then return true**
And also return the results.

2. [Iteration Main Cycle]
Begins the iteration cycle, up to 100 iterations, or when reaches the user defined kmax.

2.1: While (k < kmax)

2.1.1: Solve $(A+\mu I)h=-g$, using the Gauss Elimination Algorithm with partial pivoting (**S5**)
Note: Any numerical error, or singular matrix during the Gauss Elimination Algorithm will trigger a **return false**, along an additional error code in form a pop-up message to the user.

2.1.2: If $\|h\|<e_2(\|x\|+e_2)$, where it is the stoichiometry convergence criteria.
2.1.2.1: Then **return true,** and renormalizes **x** using the **S4** routine.
2.1.3: Else from 2.1.2 $\|h\|<e_2(\|x\|+e_2)$ **condition.**
2.1.3.1: Update the fitting parameters **y = x + h,** and then apply the **S4** routine to renormalizes the new vector **y,** where also recalculate a new residue function, using **S1** routine.
21.3.2 Evaluate the gain ratio, defined by the following formula:

$$\rho=\frac{\|Y(x)\|-\|Y(y)\|}{h^T(\mu h-g)} \tag{27}$$

2.1.3.3: If $\rho>0$, then the updated fitting parameters **y** replaced the old **x** ones.
2.1.3.3.1: With $y:=x$, All matrices and vectors **J, g, ε, x** are updated using the routines **S1,S2,S3**

2.1.3.3.2: Updates the following steeping-descent parameters:

$$\mu=\mu*\max(1/3,1-(2\rho-1)^3);v=10 \tag{28}$$

2.1.3.3.3: If $\|g\|_\infty<e_1$, then a solution is achieved, it **return true,** and renormalizes **x** using the **S4** routine.

2.1.3.4: Else from 2.1.3.3 $\rho>0$ **condition.**

2.1.3.4.1: Update the steeping-descent parameters, while discard the testing stoichiometry vector **y** from memory:

$$\mu=\mu\nu;\nu=10\nu \tag{29}$$

2.1.3.4: End If $\rho>0$ **condition.**

2.1.4. End If $\|h\|<e_2(\|x\|+e_2)$ **condition.**

2.2: End While (k < k$_{max}$): increment k=k+1
Note: If the number of possible iterations are exceeded, it will **return false**, along the message of failure to find a solution.
3: [End]

Final Notes:
Once the algorithm got a solution in case of a positive solution (return true), the ERYA will renormalizes the stoichiometry, and reevaluate the fitted reactions yields from it, and the results will pass to the main spreadsheet tab.
It also delivers some additional information, such as graphic plots, and the number of required iterations on the bottom status bar window.
A false return value means an error, and it will be displayed by an error pop-up message, such as singular matrices during numerical evaluation, numerical overflows, but more common errors will be the failure to find a solution once the maximum number of iterations expired.