CLSQ2 Documentation

Contents

CLSQ2 and documentation written by Hunter Ratliff

Department of Nuclear Engineering at The University of Tennessee, Knoxville

CLSQ2 is based on CLSQ by J. B. Cumming at Brookhaven National Laboratory in 1962.

Documentation last updated on: 28 July 2015

Introduction

The original CLSQ code was written in 1962 in an old version of FORTRAN, likely FORTRAN IV, by J. B. Cumming at Brookhaven National Laboratory. See his paper *CLSQ, the Brookhaven Decay Curve Analysis Program* for a detailed discussion on the mathematics behind the code and how it is structured. The code remains useful, but its age shows. Aside from its most common form being a 16-bit executable which cannot be run on modern 64-bit machines, the input format was particularly unfriendly to inexperienced users, requiring values being inputted in precise columns of the input text file.

This modernized version of CLSQ is written in Python 3.4 and retains most of the functionality of the original code. For the most part, CLSQ2 is simply a translation of the old FORTRAN code into Python 3. However, much of the logic in the code was redone to make it more readable and modern. Additionally, the input file syntax has been changed to be far more user friendly.

To use the code, a single input file must be prepared. All customization and problem definition is done in this text file. Then when the code is ran, the user will be prompted to provide an input file. The code will then output another text file with results.

This document serves to help the user understand the different sections of the CLSQ2 input and output decks and how to construct an input deck.

Quick Guide

CLSQ2 should be distributed as a single ZIP file. Extract the files to a single folder. When unpacked, the file structure should appear as shown below.

```
CLSQ2
     README.txt
     Documentation
          CLSQ2 Documentation
          J. B. Cumming Paper
     Executable_CLSQ2
          CLSQ2.exe
          *.dll (4 files)
          tcl (folder)
     Source_Code_CLSQ2
          CLSQ2.py
          setup.py
     Examples
     Original_CLSQ
          CLSQ.exe
          CLSQ.FOR
          Input_guide.pdf
          Examples (folder)
```

It is advised that all of these files be kept together. The only true requirement though is that all of the ".dll" files and the "tcl" folder within the "Executable_CLSQ2" folder **must** stay together with the ".exe" file for the executable to work. Feel free to make a shortcut to the executable and place it somewhere more convenient. After creating an input file (see next section for detailed input guide), the code is run simply by running the CLSQ2.exe file (double-click, right-click → Run, using the command window, etc.). When the program is run, a file dialogue window should pop up; select your input file from this window. By default the folder window will open in the same directory as the executable, so it is advised that input files be placed in a nearby directory for the sake of convenience. The program will then run, and the output file should appear in the same directory as the input file with the same filename except with a ".out" extension. This output file is just a text file, so it can be opened in any text editor. See the output section for a detailed overview of the meaning of items in the output file.

If you have a version of Python 3 installed along with the NumPy library, you may just run the Python script CLSQ2.py located in the Source_Code_CLSQ2 folder if desired.

The original CLSQ code is included just for bookkeeping purposes. Feel free to look through the old input files and source code, but note that the CLSQ.exe file will not run on 64-bit computers.

Input File

A problem is defined entirely in its input file. On the following page is a sample input file. When discussing the input and output files, any text directly from the file will be in a `green Courier New font`. The input file may have any or no extension; the output file will have the same filename as the input file but will have the extension ".out" instead. The input file just needs to contain the text describing the problem.

With CLSQ2, it is suggested the user use the "Sample_CLSQ2_file.txt" file as a template. This is because the Python code is reading specific line numbers, so mistakes are less likely when using the template. For instance, line number 3 will always be read in as the input for number of unknown half-lives (variable name NV). Later, the input deck will be discussed line by line.

In general, the input deck can be described as having four sections: a descriptive comment line, a problem control/settings section, half-life data, and time data. The comment line (line #1) and control section (line #2 through #14) will always have the same line numbers. Following those sections are two text lines (each starting with a "`#`") just used for providing instructions. The text can be replaced, but the lines must remain. Next, the half-life section will occupy the number of lines equal to the number of components (NC) selected for the problem; it always begins on line #16. The next line is where the user specifies the end of bombardment time. Following this will be three more text lines (each starting with a "`#`"). Following the three text lines is the measured time and count data. The Python code will read as many lines as are provided until it reaches the end of the file.

In the control/settings section, the Python code for CLSQ2 just searches for the rightmost string that is separated by white space. The text to the left of each entry is there just for the sake of being descriptive. The number of spaces following the description before the supplied number does not matter as long as it is at least one; they are just arranged to be easier to read in the sample input file.

The half-lives and end of bombardment time must be a number followed by a single letter (indicating units of time) with no space in between. These cannot be in mixed units; for instance, 1 day and 12 hours must be inputted as `1.5D` or `36H` but not `1D 12H`.

The time and counts data can have any common delimiters (such as spaces, tabs, and commas). Here, they are tab delimited for the sake of easy copy/pasting to and from Microsoft Excel. All inputs are read in as character strings and then converted to the appropriate type (integer or floating point number). All numbers that are not explicitly intended to be integers are read into the Python CLSQ2 code as floating point numbers. When converting these strings to floating point numbers, Python can handle decimals and scientific notation, meaning values can easily be copy/pasted from Excel without worrying about formatting.

Below is a sample input deck. On the left in black are the line numbers and to the right in green is the contents of the input file. It should be noted that in the actual text files, the lines starting with the "#" character only occupy one line.

```
1   Sample CLSQ input deck (TH-232, (p,alpha), cont count, I-134 (847 keV))
2   Number of components or isotopes (NC):     1
3   Number of unknown half-lives (NV):         1
4   NCNV (leave set to 0, legacy code):        0
5   Iteration stop criteria (CNV):             0.05
6   Background counts (BGD):                   0
7   Background standard deviation (SBGD):      0
8   Input format (IN), (IN=2 suggested):       2
9   Matrix inversion verification (IT):        0
10  Counter dead time in microseconds (BLOCK): 5
11  Standard deviation cutoff percent (SCOFF): 0.5
12  Bad point rejection criteria (RJT):        6.0
13  Known component subtraction (set 0)(KCS):  0
14  YIELD (leave set to 0, legacy code):       0
15  # Below, input half-lives (in S, M, H, D, or Y with no
    space) for each component (NC), data for 1 component per
    line.
16  52.50M
17  End of bombardment reference time (EOB): 0.0M
18  # Now input measured time and count data below.  As long as
    each time is on its own line and values are separated by at
    least one space, tab, or comma, spacing does not matter.
19  # The first 3 columns are decay time from EOB in D, H, and
    M. The fourth column is counts and the fifth column is
    measurement time for each sample in minutes.
20  # Days Hrs Min    Counts      Measurement time (minutes)
21  0    1    0.9  6.11E+03  10
22  0    1    11.3 5.42E+03  10
23  0    1    21.6 4.53E+03  10
24  0    1    42.3 3.88E+03  10
25  0    1    52.6 3.56E+03  10
26  0    2    13.1 2.78E+03  10
27  0    2    23.3 2.16E+03  10
28  0    2    36.6 5.38E+03  30
29  0    3    7.2  4.00E+03  30
30  0    3    37.7 2.77E+03  30
31  0    4    8.2  2.10E+03  30
32  0    4    38.6 1.21E+03  30
33  0    5    9    8.52E+02  30
34  0    5    39.3 6.37E+02  30
35  0    6    9.7  4.61E+02  30
```

Now, the input deck will be explained line by line. Part of the explanation of each variable will be taken from J. B. Cumming's paper mentioned earlier.

In the original code, the "control card" (what has been referred to here as the controls/settings section) was a single line. The original FORTRAN code would read in values from exact columns on a single line of the input file for the control card. CLSQ2 seeks to preserve all of the input possibilities but to provide a clearer way of inputting these control elements. Some "legacy" features of the original code no longer are necessary or do not do the exact same things they used to, but their inputs are still here.

Line # 1

```
Sample CLSQ input deck (TH-232, (p,alpha), cont count, I-134 (847 keV))
```

This first line is for a descriptive comment. All of the text on this line will be copied to the top of the output file too.

Line # 2

```
Number of components or isotopes (NC):      1
```

NC is the number of components in the data. This is the total number of half-lives that will be given to the code to analyze.

Line # 3

```
Number of unknown half-lives (NV):          1
```

NV is the number of unknown half-lives. The first NV values of the NC (provided half-lives) will be treated as variable.

Line # 4

```
NCNV (leave set to 0, legacy code):         0
```

This is an input present in the original code that is not discussed in the original documentation. It seems that this variable has a small amount of control on the iterative solving process. In the input decks used with the original code, the column where this variable would be inputted in the input file was usually left blank. This value should be kept equal to zero.

Line # 5

```
Iteration stop criteria (CNV):              0.05
```

CNV is a criteria used to determine when iteration can end. The code will continue to iterate until CNV is less than the change in the decay constant divided by the standard deviation of decay constant for all NV variable half-lives. In mathematical form:

Iterate until: $CNV \leq \frac{\Delta\lambda}{\sigma_\lambda}$

The user may wish to change this value for more precise results. If CNV is set to zero, the code has a prebuilt number of iterations it will perform. In the original code, this default was set to nine iterations. In CLSQ2, the default is 20 iterations, but this can be easily changed in the Python code. If given a nonzero value for CNV, such as 0.05, the code will usually not hit the iteration limit.

Line # 6

```
Background counts (BGD):                    0
```

BGD is used for background subtraction for count data. CLSQ2 can do a universal background subtraction or a background subtraction at each point of time/count data.

If BGD = 0, no background subtraction will be done at all.

If BGD > 0, the value of BGD will be subtracted from every point in the calculated counts per minute array. Thus, if a positive BGD is used, BGD should be in units of counts per minute.

If BGD < 0, background subtraction at each point is turned on. The actual value of BGD does not matter in this case, just the fact that it is negative is used to trigger individual background subtraction. How the background data is inputted will be discussed later when covering the time data input.

Line # 7

```
Background standard deviation (SBGD):       0
```

SBGD is the standard deviation of the value of BGD, the universal background subtraction in counts per minute. SBGD is only used if BGD > 0. A background standard deviation (SBGD) may be provided here for root-mean-squared addition to the standard deviation of each provided data point.

Line # 8

```
Input format (IN), (IN=2 suggested):        2
```

This is another legacy setting. The original code possessed several different ways of inputting data. These other means of inputting data are no longer necessary. This IN value previously determined which input method for the time data would be used. However, in CLSQ2 this variable no longer does anything. If it is not set to equal to 2, the code will set it to 2 on its own and print a statement saying that IN has been set to equal 2.

Line # 9

```
Matrix inversion verification (IT):          0
```

This is another legacy feature. In short, this variable does nothing in CLSQ2. In the original CLSQ code, a large (102 lines) subroutine was used to invert the B matrix. In the original code, this was normally left blank but could be provided a number for troubleshooting and to verify that matrix inversion was executed correctly. Setting IT to a nonzero value would cause the code to perform a check to see if $B^{-1}B = U$, where U is the unity matrix. It would also cause the code to print indeterminate matrices. In modern codes, matrix inversion can be done with a single command. If the matrix inversion fails, the code itself should fail rather than successfully running while producing an erroneous result.

Line # 10

```
Counter dead time in microseconds (BLOCK): 5
```

BLOCK is the counter's dead time in microseconds. The code does perform a dead time correction.

Line # 11

```
Standard deviation cutoff percent (SCOFF): 0.5
```

SCOFF sets a cutoff (percent) on the smallest value at which the program will use standard deviation from statistics alone. For instance, if SCOFF is set to 0.5, the standard deviation of any point will never be less than 0.5% of the count rate, regardless of how many counts are recorded.

Line # 12

```
Bad point rejection criteria (RJT):        6.0
```

If not set to zero, RJT will cause the program to analyze its output and reject any bad points. Any point falling further than RJT times the standard deviation from the curve will be discarded and the fit will be redone without the bad point(s).

Line # 13

```
Known component subtraction (set 0)(KCS):  0
```

If KCS is any value other than zero, known component subtraction will be performed. KCS is the number of components that known component subtraction will be performed with. This causes the last KCS of the half-lives to be treated as having known intercepts. If using this feature, one should note that the code handles reading in this line slightly differently than the others. If using the template, everything should work fine. The code searches for the colon (:) and ignores it and everything to the left of it. Following the colon, the first string should be the value of KCS (which should be a positive integer). The code will then be expecting at least KCS and up to 2*KCS additional entries. The first KCS entries beyond the actual value of KCS should be the values of the intercepts. After that, the user may input standard deviations for each intercept. The code should be able to handle any common delimiter (comma, semicolon, space, tab, or any other whitespace) separating the values. If an expected entry is not provided, it will default to zero. KCS functionality needs additional testing.

Line # 14

```
YIELD (leave set to 0, legacy code):       0
```

This feature is not discussed in the original documentation but is present in the original FORTRAN code, thus its correct performance is not guaranteed. However, its functionality is still present in CLSQ2. When set to a nonzero value, all calculated counts per minute data are divided by YIELD. This was likely used for count rates per unit mass, but this is not certain since it was not documented properly.

Line # 15 is ignored by CLSQ2, but the line must be present for spacing.

Line # 16 through 16 + NC - 1

```
52.50M
```

On lines 16 through 16+NC-1, the user must input NC half-life values. Each line corresponds to each component NC. Half-lives should be given in a single unit, not mixed units. Accepted units are seconds (`S` or `s`), minutes (`M` or `m`), hours (`H` or `h`), days (`D` or `d`), and years (`Y` or `y`). The first NV of these are considered first guesses.

While only one input per line is required and is discussed in the original documentation, the FORTRAN code allowed for up to four inputs per line. This functionality remains in CLSQ2; it is just not extremely well understood since it was not documented. The first input is stored in a half-life array called H. The second input is another half-life which is stored in a separate half-life array called HL (using the same input syntax as the first half-life). This half-life is believed to be of the daughter to the isotope whose half-life was given earlier in the same line. CLSQ2 will not vary the daughter half-life, even if the parent is set to be variable and solved for. This means that the user's guess for the daughter half-life needs to be accurate.

The third entry in a line is stored in an array called CEF. It is reprinted in the output file as being equal to "C1/C2." This along with some reading of old literature that used CLSQ lead me to believe that this third entry CEF is the initial ratio of the activity of the daughter to the activity of the parent. In addition to listing a daughter on the same line as its parent, the daughter should also be listed on another line as its own component (NC).

The fourth entry on a line is stored in an array called TBOMT. The TBOM variable in the code may stand for "time at beginning of measurement" or "time beam is turned off in minutes." This is a time value which should be entered in units of minutes.

The entries can be separated with any common delimiter. CEF and TBOMT are read in as floating point numbers and will default to zero if no values are provided. Note that these values have no bearing on the problem if no value for HL is provided. This daughter isotope feature needs more testing. The CEF and TBOMT are still not entirely understood.

Line # 16 + NC

```
End of bombardment reference time (EOB): 0.0M
```

EOB should be thought of as a reference time, t = 0. This time value will end up being subtracted from all other time values. While this value is usually set to zero, it allows for absolute times to be used if desired. For instance, if a target is done being bombarded at 3:00 am on day 1 of an experiment (or, say, June 1) and the first count data was not collected until 10:30 am on day 3 of the experiment (or June 3), EOB could be set to `27H` or `1.125D` and all time data could be inputted in absolute terms, with the first time starting at 3 days, 10 hours, and 30 minutes. This feature is simply here to make inputting time data more convenient depending on how time was recorded.

Line # 16 + NC + 1 through 16 + NC + 3 are ignored by CLSQ2 but must be present for spacing.

Line # 16 + NC + 4 and beyond

```
0     1     0.9   6.11E+03   10
```

This is where the time, count, and count time data are inputted. Each line is interpreted as a single data point. Each line must have a minimum of five entries.

The first three entries are time data. The first entry is days. The second entry is hours. The third entry is minutes. The time data will end up being converted to minutes (which is how it will appear in the output file) using the formula:

$$\text{Time (minutes)} = (24*60*T_{days}) + (60*T_{hours}) + T_{minutes}$$

This means that the user is welcome to input time values in whatever units are desired, but three values for time must be provided. For example, 36 hours could be represented several ways:

```
1.5   0     0
0     36    0
0     0     2160
1     12    0
1     11    60
```

Following the time data, the fourth entry on each line must be counts. As stated earlier, Python is capable of reading in scientific notation, so copying from Excel or another program can be done without concern for formatting as long as no spaces are present in the number.

The fifth and final mandatory entry in this section is count collection time in minutes. This is used to convert the counts into a count rate in units of counts per minute.

The sixth (optional) entry is background count rate in counts per minute (BGND). As discussed earlier, this individual point background subtraction is turned on by setting BGD < 0.

The seventh (optional) entry is background count rate standard deviation / uncertainty (SBGND).

The last four entries (8 through 11) were not documented, thus are not very well understood. Regardless, their functionality remains present in CLSQ2.

The eight (optional) entry, called SIGPCT, appears to be a standard deviation / uncertainty percentage that can be set.

The ninth (optional) entry, called TYPE, is a character string and has no impact on the code. If text is input here, it will appear in the output file in the "TYPE-FWHM" column.

The tenth (optional) entry, called EP, is a floating point number and has no impact on the code. In the output, the EP value is printed to the energy column. This is likely just useful for bookkeeping. If left blank, EP = 0.0.

The eleventh (optional) entry, called IDENT, is a character string and has no impact on the code. This likely stands for "identity" as is used as a label for bookkeeping purposes. If text is input here, it will appear in the output file in the "ID1 ID2" column.

Output File

After running CLSQ2, an output file will be generated in the same directory as the input file with the same filename except with a ".out" extension. For instance, running a file called Sample_file.txt through CLSQ2 would generate an output file called Sample_file.out. The output is largely the same as it was in the original code. The primary difference is that the tabular output is tab delimited, allowing for easier copying and pasting to Excel.

The top line of the output file will contain the same descriptive comment as on the first line of the input file. The creation time of the output file is also printed within the output file.

The first section of the output file is restating the inputted time and count data, with the counts per minute calculation already done.

The remaining portion of the output file contains the results of the code. It is mainly a repeating block of output that is reprinted for each major iteration of the code. This means that when looking at an output file **the final results will be located at the bottom of the output file**. The first line in the calculated output section states the values of a number of variables. Most of these should be familiar from the input file section. The NP variable is the number of time points used in that iteration's calculation. Note that this number would be smaller than the number of points provided if any data points were rejected prior to that iteration. Additionally, due to how the code works, NV starts at zero and will work its way up to the provided NV value as it iterates. Some information is not calculated or printed for the first iteration.

Below this first line in every iteration where NV is greater than zero is a line stating the number of iterations performed and whether the results were convergent. Following this line is a table listing values for "COMP" 1 through 5. The number of filled columns should be equal to the value of NV listed above the table. The D row contains decay constants (in min^{-1}). The DELTA rows (one for each iteration) contain values which are used in calculating SIGMA and were used in determining when to stop iterating.

Next is a small table listing properties of each component (isotope). In the first iteration, the half-lives should be the same as what was listed in the input file. The table lists each component's half-life with its uncertainty SIGMA H, count rate contribution at the "EOB" time t_0 with its uncertainty SIGMA EOB CPM, and decay factor (which is equal to $e^{\lambda * t\_0}$ where λ is the decay constant in min^{-1}). If half-lives were provided for daughter isotopes, the relevant information for the daughter would then be printed beneath the line of the component with that daughter. Additionally, if known component subtraction (KCS) was used, relevant information to the KCS would be printed here too.

Then, the actual fit data is printed. First is a line containing two values that show the quality of the results. The "FIT" value was present in the original code. In the original documentation, it is said that FIT can thought to be similar to χ^2 (chi-squared). For more information on the FIT value, see the original paper by J. B. Cumming. For the sake of making

the quality of the fit easier for the modern user to understand quickly, a calculation for R^2 has been added in CLSQ2. This compares the provided data F to the calculated data FCALC to determine R^2.

Following the quality of fit information will be a table containing the actual data. The T(I) column contains all of the time data used (in minutes). F(I) is the counts per minute data calculated from the provided count and count time data. FCALC(I) is the count rate calculated by CLSQ2 using the half-lives listed for each component in that iteration. V(I) is simply equal to F(I) – FCALC(I). SIGMAF(I) is the calculated uncertainty of FCALC(I). RATIO(I) is equal to V(I) / SIGMAF(I) and is used in determining how long to iterate for (see the CVN variable in the input section, line #5).

If all data points were satisfactory, "ALL DATA POINTS OKAY." will be printed to the output file. If any data points meet the rejection criteria (set using RJT in the input file), the bad data points will be listed and the calculations will be repeated without them.

In some cases, the code will not run entirely as intended. If a problem is input incorrectly, it may lead to some strange things happening in the code that would usually cause it to break and then only write the output up to the point it failed. One such situation that seems to be most common is the code reaching a point where it needs to take a square root only to discover that the argument is negative. Here, rather than having the code fail, I have opted to allow the code to take the absolute value of the argument before taking the square root, but an obvious message saying that this action was performed will be printed to the output file warning the user not to trust the results.

If the code hits an error and fails to run to completion, the "CLSQ2_error_log.txt" file should be updated. The code is written so that output will be printed to the output file as it is generated, so even if the code fails, output up to the point of failure will be printed. A sign that the code did not run to completion is if the last value of NV printed in the output file does not equal what was inputted in the input file. This error log file will be located in the same directory as the input file. It should specify where in the Python code that failure occurred. Usually, the error message is descriptive enough to discern if an input mistake was made. Tinkering with the input file (like rearranging the order of provided NC half-lives) is likely a superior option to delving into the CLSQ2 source code.

Developer Notes

This section is intended for any users who plan on looking into or modifying the Python source code. It also contains some of my general notes about the code. If running the Python source code for CLSQ2, a version of Python 3 and the NumPy library will need to be installed. NumPy is the only Python library used in the CLSQ2 code that does not ship with Python by default. All of the files included in this CLSQ2 distribution can also be found online at:

https://github.com/Lindt8/CLSQ2

Notes on the code itself

As stated at the beginning, CLSQ2 is written in Python 3.4. It was translated from an old version of FORTRAN (likely FORTRAN IV). Due to the documentation on CLSQ2 being rather scarce (the J. B. Cumming paper being the most detailed information), the code was essentially translated to Python line by line rather than being rebuilt symbolically. Some of the code looks rather clunky, and that is why. That being said, the input and output processes were completely redone for CLSQ2. The old CLSQ would read in information from the file while performing calculations. The way it was done seems to be an attempt to minimize the amount of memory used. Since memory is not even remotely an issue for modern machines running this code, the input is read all at once. The input file is opened, all information is extracted and stored in variables, and the input file is then closed. Within the source code for CLSQ2, all of the output is written to the output file as the output is generated. In the event that the code fails to run to completion, this allows the user to have an idea of how far the code made it before reaching an error. In combination with the traceback error message found in the CLSQ2_error_log.txt file, troubleshooting should be made easier with these features.

In the original FORTRAN code, the user had to place inputs for the control cards and time data in the exactly correct columns. While sifting through the source code, this led to some interesting discoveries. The NCNV variable in the input deck was found to be a variable read in by the original code. However, in every example input deck I could find, this variable was never used. Additionally, this value was never discussed in the Cumming paper.

Perhaps the largest feature found by sifting through the read in inputs was in the half-life input lines. Nowhere in the original Cumming paper are daughter isotopes mentioned. The half-life lines were capable of receiving four inputs as opposed to the just one input that was used in every example input deck I have seen. After further studying the source code, it seemed that the second half-life inputted on a line could be that of a daughter isotope. There was even a line for outputting information about the daughter available for the output file. The meaning behind the other two inputs seems to be related to the daughter isotope but remain somewhat of a mystery to me. This seems like a rather important feature to not be included in the original paper by J. B. Cumming. I do not know if this lack of documentation means that the feature is nonfunctional,

produces incorrect results, or was simply implemented after the paper was written. Regardless, this lack of documentation leads me to being skeptical about this feature. It probably works, but I would not trust it unless it were tested thoroughly.

The new code alleviates much of the pain present in the original code in regards to input deck creation. The control cards are spread out over numerous lines and descriptive text has been added to help the user understand the meaning of each input parameter. Additionally, one of the largest benefits of using Python is the ability to easily parse the input text. Python can easily break a line of text into sections from which input values can be extracted. Values can be separated by commas, semicolons, or any form of whitespace such as tabs and spaces.

As for the code itself, evidence of its roots in FORTRAN remain present. Namely, large sections where variables are declared and initialized remain in the Python code. When coding this, I set the arrays to default to the same size as was in the original code. However, I have also coded in the ability for the code to automatically recognize if more data points have been entered than what the code would had originally allowed; in that case, the arrays will be made larger to fit the data.

One of the largest challenges in translating the code was preserving the logic and flow of it. Modern coding practices encourage structured programming, meaning the code uses loops and functions as opposed to GO TO statements and jumps. In Python, GO TO statements do not even exist. However, in the old FORTRAN code, flow was controlled exclusively with IF statements, DO loops (similar to for loops in other languages), GO TO statements, and a handful of old FORTRAN features (such as arithmetic IF statements, which are obsolescent in modern Fortran). These are used in combination to create structures equivalent to "if/else" and "while" in modern codes; however, the ordering of the code using these old methods is considerably different and a complete mess to read. The primary offender is the GO TO statement. Since CLSQ is attempting to solve a problem iteratively, it makes sense that the code would need to be structured in the form of a while loop which runs until some convergence criteria is met. Since while loops (DO WHILE in modern Fortran) did not exist in this old version of Fortran, GO TO statements were used to pass control of the code to an earlier line. The code would continue running and being redirected back up to an earlier line until a certain criteria was met so that another GO TO statement would pass control of the code to a line past the first GO TO statement or until the criteria of an IF statement containing the first GO TO statement was no longer met. One can easily see how this would get quite messy quickly. Additionally, GO TO statements were used in the CLSQ code to skip a few lines if a certain criteria was met, the ancient version of an IF ELSE statement.

So, CLSQ2 preserves the operation of the old code but should at least be easier to read and follow since the code is now written in a structured fashion. That being said, due to how the original code was written, CLSQ2 contains many IF statements and some nested WHILE loops. Especially lines 317 through 359 in the original code were tricky to translate. This section contains the bad point rejection and final checks of the code, and it is littered with GO TO statements. As can be seen in the CLSQ2 Python code, I have used variables named "gotoXXXX" where XXXX was the label number corresponding to the original code in an

attempt to make sense of all of the GO TO statements. I used these gotoXXXX variables as on/off switches for other sections of the code that would be skipped or executed depending on if the GO TO statement had been triggered. While this arrangement is still not ideal, it correctly mimics the logic of the original CLSQ code.

Creating the executable version

As long as Python 3 remains a relevant coding language (with no exceedingly major changes to syntax), the code itself should remain relevant for a long time in the future. However, the distributed executable may not. The original problem was that CLSQ was distributed as a 16-bit executable file which could not run on modern 64-bit machines. The executable for CLSQ2 is a 32-bit program. This was done because all 64-bit machines are capable of running 32-bit programs, and 32-bit machines are not *completely* obsolete just yet. Additionally, it will likely be very many years before the norm becomes 128-bit or some other architecture. Thus, this 32-bit executable should remain functional as long as 64-bit machines continue to support 32-bit programs. If this executable ever becomes obsolete, the Python code will hopefully still work and could be used to create a new executable. When looking back at the original CLSQ code, the need for an executable is apparent when the other option is to ask users to install a FORTRAN IV compiler and go through all of the trouble that may bring with it. However, the code also used a very outdated and user-unfriendly input system, so modernizing the code became a more appealing option. Python happened to be the most appealing option since it is free and easy to use. Still though, asking users to install a version of Python 3 and the NumPy library is more than the average user should be expected to do. So, converting the new Python code into an executable file was desired. Three different tools were available that seemed promising: py2exe, cx_Freeze, and PyInstaller.

PyInstaller seemed to be the most promising in regards to creating a single file. Unfortunately, as of writing this, Python 3 support is still experimental and in development. If you wish to look into PyInstaller, please refer to https://github.com/pyinstaller/pyinstaller/wiki . The development version for Python 3 support available as of writing this did not function.

The most promising tool ended up being py2exe. In the same directory as the source code is a file called "setup.py" which is used to create the executable. In the command window, simply navigate to the directory with the source code and setup.py and execute the command "`python setup.py`". This will generate within that directory a folder called "dist". The executable file in the /dist directory is what the user will end up running to use CLSQ2. This "dist" folder's contents must be kept together. Of note, setup.py is set to bundle files partially ('bundle_files': 2); it is unable to bundle everything into one file ('bundle_files': 1) due to the folder dialogue used with the tkinter library. Even if the file input mechanism is reworked so that the user just types the file name into the command window (allowing the tkinter library to be excluded), py2exe will still create an extra file that must be kept with the executable file since the NumPy library is still being used. So, since requiring the user to keep supplementary files with the executable is inevitable, the option yielding the easiest user experience was selected,

allowing the user to just double-click the application and select a file from the pop-up folder dialogue window to run the code. More information on py2exe can be found at https://pypi.python.org/pypi/py2exe/ . Note that the executable created with py2exe only works on Windows machines. To create an executable that works on Mac OS X, a similar tool called py2app (https://pypi.python.org/pypi/py2app/) must be used in a similar fashion while on a Mac computer.

The cx_Freeze tool was tried as well. It worked, but the total file size was about twice as large as with py2exe. Additionally, cx_Freeze ended up producing more files the user would need to keep track of than py2exe. Thus, py2exe was chosen. More information about cx_Freeze can be found at http://cx-freeze.sourceforge.net/ . One upside of cx_Freeze is its compatibility with numerous operating systems. While an executable created with cx_Freeze on a Windows machine will only run on Windows, cx_Freeze can be used to create executables for OS X and Linux when run on those operating systems.