DEVELOPING A MODEL TO ACCOUNT UNCERTAINTIES IN MEASURED AND DERIVED PHYSICOCHEMICAL PROPERTY: MOLALITY

A Research Project Report

Submitted to

Department of Statistics, Shivaji University, Kolhapur

As a Partial Fulfilment for the Degree of

Master of Science in

Applied Statistics and Informatics

By

Mr. Banke Akhilesh Shivaji

Under the Guidance of

Prof. Dr. S. B. Mahadik

Department of Statistics

Shivaji University, Kolhapur

April, 2025

Developing A Model to Account Uncertainties in Measured and Derived Physicochemical Property: Molality

Abstract

Precise measurement of solution concentration is important in physical chemistry, especially in the investigation of solute-solvent interactions and physicochemical properties. Molality, being independent of temperature, is an ideal measure of concentration. Experimental errors due to weight measurements influence the accuracy of molality calculation. Classical error calculation using intricate formulae of the method of propagation of errors makes it laborious and susceptible to errors by hand. An organized method of estimating and reducing these errors is required to enhance experimental accuracy.

In this study, molality has been taken as the independent variable and error in molality as the dependent variable. A polynomial regression model of degree 2 was constructed to study the interaction between these variables, taking care of heteroscedasticity and autocorrelation. The model was experimentally validated and tested against literature data, demonstrating its reliability. A *Graphical User Interface* (GUI) was also constructed, incorporating the method of propagation of errors to enable automation of error calculations.

The GUI tool and regression model developed minimize the complexity of error calculation to a great extent, avoiding manual calculations while ensuring greater precision and efficiency in molality calculation. This method minimizes experimental runs, optimizes solution preparation, and increases the reliability of physicochemical investigations.

Contents

1. Introduction

1.1 Background

In physical chemistry, solute-solvent interaction understanding is significant to investigate the physicochemical characteristics of solutions like density, viscosity, conductivity, and thermodynamic behaviour. All these characteristics tend to be concentration-dependent; therefore, accurate solution concentration measurement is important to achieve precise experimental investigation. Out of all concentration units, molality is mostly used because it is independent of temperature, resulting in a more stable and dependable measure than molarity, whose value varies with temperature because of volume dependence.

Nevertheless, in laboratory procedures, solution preparation requires several steps of measurements with errors introduced in concentrations. The primary source of the errors is from instrumental imperfections, weighing balance precision errors, and human manipulation differences. Small differences in weight measurements can have a significant influence on molality calculations, eventually leading to physicochemical property estimation inaccuracies. Therefore, it is required to systematically detect, measure, and minimize such errors to enhance experimental data consistency and reliability.

To provide for this need, error propagation methods are universally employed in experimental chemistry. These methods enable one to calculate the total uncertainty in molality considering individual measurement errors at every level of solution preparation. This makes solute, solvent, and weighing equipment errors systematically available in final concentration calculations.

1.2 Research problem

Precise determination of solution concentration is critical in physical chemistry, particularly for analysing solute-solvent interactions and physicochemical properties. While molality is a preferred concentration measure due to its temperature-independent nature, its accurate determination is affected by instrumental and human-induced errors during solution preparation. Weighing inaccuracies, equipment precision limits, and handling variations contribute to deviations in molality calculations, leading to errors in physicochemical property estimations and affecting the reliability of experimental findings.

Existing methods for error estimation in molality rely on complex mathematical formulae from the method of propagation of errors. These manual computations are time-consuming, prone to human mistakes, and inconsistent across different experiments, making precise error quantification a challenge. Additionally, experimental chemists often require repeated experimental runs to validate their results, further increasing the workload and resource consumption.

To address these challenges, there is a need for a systematic and automated approach to analyse, quantify, and minimize errors in molality determination. A solution that integrates statistical modelling and computational tools can help improve accuracy, efficiency, and reproducibility in experimental studies. Developing such an approach will enhance error estimation, reduce manual workload, and optimize solution preparation, ultimately contributing to more reliable and precise concentration-dependent research.

1.3 Objectives

1.3.1 Exploring the relationship between molality and its errors.

1.3.2 Modelling the explored relationship for general applicability for predicting errors in chosen molality.

1.3.3 Design and development of *Graphical User Interface* (GUI) tool for error analysis.

1.4 Scope

The work has wide-ranging scientific and industrial use, especially in areas where accurate concentration measurement is of paramount importance. In research chemical laboratories, the work presents a methodological approach to enhance solution preparation quality, which is necessary for experiments with thermodynamic studies, reaction kinetics, and material synthesis. In the field of pharmacy, the results can assist in making accurate formulation of drug solutions precise, where slight miscalculation in concentration can influence drug efficacy and stability. In medical and biochemical research, precise calculations of molality are important to prepare buffer solutions, enzyme assays, and diagnostic reagents, minimizing experimental variation and enhancing reproducibility.

In addition, this study can support industrial quality control procedures, especially in food and beverage manufacturing, cosmetics, and chemical production, where solution consistency and concentration precision are crucial for product standards. Creating a GUI provides these error calculations readily available, enabling researchers and industry experts to make accurate calculations without human error. This automation increases efficiency and minimizes experimental workload and is therefore an important tool in both academic and industrial applications.

2. Key Terms

2.1 Uncertainty/Error

Uncertainty or **Error** in an instrument while conducting an experiment refers to the **degree of doubt** in a measurement due to limitations in the measuring device or external factors affecting accuracy. It indicates how much the measured value may differ from the true value.

2.2 Method of propagation of errors

When a measurement involves multiple quantities, each with its own uncertainty, the total uncertainty in the final result is determined by **error propagation** rules. The method depends on the mathematical operation being performed.

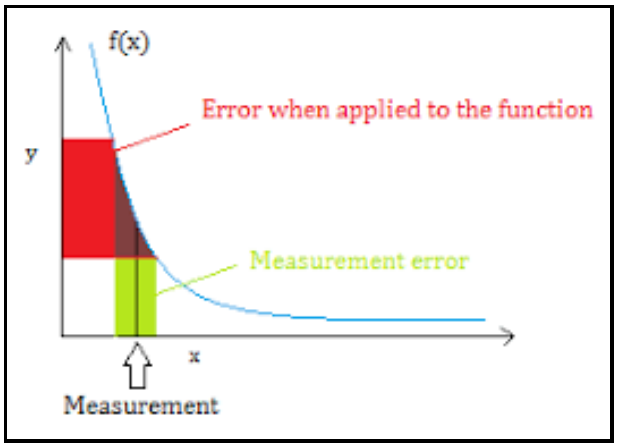


Figure 1: Method of propagation of errors

Figure 1, visually represents **error propagation in a function** f(x), illustrating how a small measurement error in x leads to a larger error in f(x).

The horizontal axis represents the measured quantity x. The **black arrow** at the bottom shows the measured value of x. The **green shaded region** represents the **measurement error** (uncertainty in x).

The vertical axis represents the function value f(x). Any uncertainty in x affects the calculated value of f(x). The **red shaded area** represents the propagated error in f(x), caused by the uncertainty in x. Since the function is nonlinear, the error in f(x) is larger than the original measurement error in x. This demonstrates that even a small error in x can lead to a **significant** error in f(x).

Type of calculation	Example	Standard Deviation of y
Addition or subtraction	y = a + b - c	$S_y = \sqrt{S_a^2 + S_b^2 + S_c^2}$
Multiplication or Division	y = a * b / c	$\frac{S_y}{y} = \sqrt{\left(\frac{S_a}{a}\right)^2 + \left(\frac{S_b}{b}\right)^2 + \left(\frac{S_c}{c}\right)^2}$

Table 1: Formulae for Method of propagation of errors

*a, b and c are experimental variables whose standard deviations are S_a, S_b and S_c respectively.

2.3 Solute

A **solute** is a substance that is dissolved in a **solvent** to form a **solution**. The solute is usually present in a smaller amount compared to the solvent. For example, in a **saltwater solution**, **salt** is the solute, and **water** is the solvent.

Solutes can be **solid, liquid, or gas**:

- **Solid solute**: Sugar in water
- **Liquid solute**: Alcohol in water
- **Gas solute**: Carbon dioxide in soda

2.4 Solvent

A **solvent** is a substance that dissolves a **solute** to form a **solution**. The solvent is usually present in a larger amount than the solute.

For example:

- In **saltwater**, **water** is the solvent, and **salt** is the solute.
- In **sugar water**, **water** is the solvent, and **sugar** is the solute.
- In **air**, **nitrogen** is the solvent, and **oxygen** and other gases are solutes.

Water is known as the **"universal solvent"** because it can dissolve many substances.

2.5 Solution

A **solution** is a **homogeneous mixture** of two or more substances, where a **solute** is dissolved in a **solvent**. The particles of the solute are evenly distributed within the solvent, making the mixture uniform throughout.

For example:

- **Saltwater** (salt as the solute, water as the solvent)
- **Sugar water** (sugar as the solute, water as the solvent)
- **Air** (oxygen and other gases dissolved in nitrogen)

2.6 Molecular weight of solute:

The molecular weight (molar mass) of a solute is the sum of the atomic masses of all the atoms in a molecule, expressed in grams per mole (g/mol).

Example: Molecular Weight of Glucose ($C_6H_{12}O_6$)

- Carbon (C) = 12.01 g/mol, and there are 6 atoms → $6 \times 12.01 = 72.066$
- Hydrogen (H) = 1.008 g/mol, and there are 12 atoms → $12 \times 1.008 = 12.096$
- Oxygen (O) = 16.00 g/mol, and there are 6 atoms → $6 \times 16.00 = 96.006$

  Total Molecular Weight = 72.06 + 12.096 + 96.00 = **180.16 g/mol**

2.7 Physicochemical Properties

2.7.1 Molality

Molality **(**m**)** is a measure of the concentration of a solution, defined as the number of moles of solute per kilogram of solvent**.** It is expressed in moles per kilogram (mol/kg)**.**

Formula:

$$\textbf{Molality} = \frac{\textbf{Weight of solute} \times \textbf{1000}}{\textbf{Weight of solvent} \times \textbf{Molecular weight of solute}}$$

2.7.2 Apparent Molar Volume

Apparent molar volume is the partial volume occupied by one mole of solute in a solution, taking into account its interactions with the solvent. It helps describe how the solute affects the total volume of the solution.

Formula:

$$\emptyset_v = \left(\frac{M_2}{\rho}\right) + \left(\frac{(\rho_0 - \rho)}{m\,\rho\,\rho_0}\right)$$

3. Literature Review

3.1 D. H. Dagade, Sandeep P. Shinde et al. (2014)

Aqueous solutions of amino acid ionic liquids with molalities of 0.05 to 0.5 mol kg^{-1} were prepared in this work. The influences of different molality on the volumetric and acoustic properties of the solutions were investigated systematically. In this study, apparent molar volumes were determined from density measurements and displayed concentration-dependent behaviour, reflecting the influences of hydration and structural changes in the solution. These values facilitate understanding of the influences of the presence of solute on the overall solution dynamics.

Herein, the coefficient was found through density measurements over a temperature interval, offering a measure of AAIL solution volumetric response as temperature varies. Low α signals relative volume stability with temperature variation, critical in numerous applications. These parameters—molality, apparent molar volume, and coefficient of thermal expansion—are very important in characterizing the physicochemical properties and behaviours of amino acid ionic liquids in aqueous solution.

3.2 Shrikant P. Musale, Dilip H. Dagade et al. (2018)

This work aims to determine the density and sound speed of dilute binary mixtures of diethylammonium-based protic ionic liquids and water.

- Four diethylammonium-based protic ionic liquids were prepared and their physicochemical properties examined.
- Experimental information was obtained at different temperatures, namely density measurements and sound speed at 298.15 K.
- The research investigates how ion association and hydration in these solutions vary with concentration and temperature.

4. Experiment

4.1 Preparation of Original Solution

The first step requires a precision weight balance and a clean, dry beaker. The manufacturer provides the instrument's error margin; for instance, in this study the weight balance at the Molecular Thermodynamics and Modeling Laboratory, Department of Chemistry, Shivaji University Kolhapur, had an error of **±0.001 kg.**

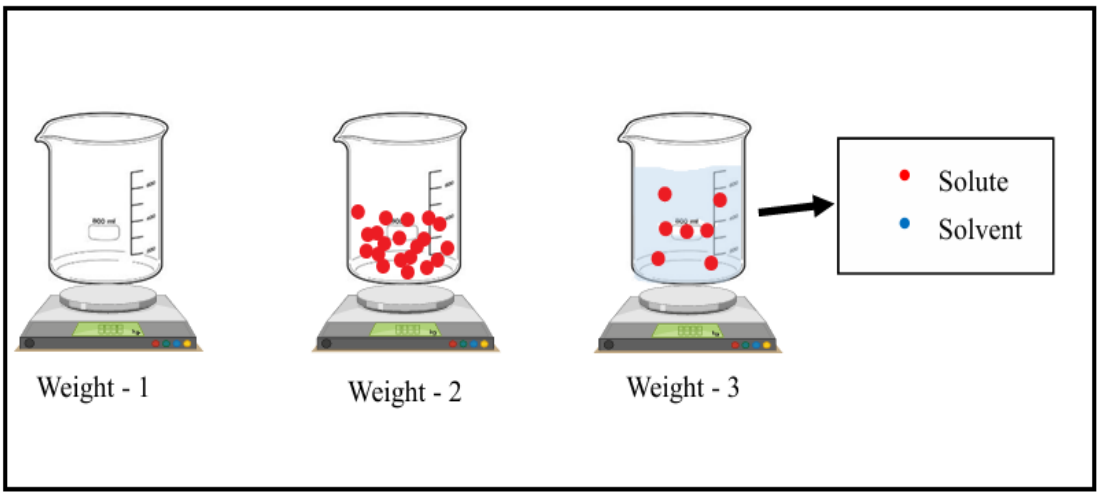


Figure 2: Preparation of original solution

Figure 2 shows an illustration of the step-by-step procedure of solution-making by dissolving a solute in a solvent. In the first step ($Weight_1$), an empty beaker is weighed on a weighing balance, and its weight is recorded. Here, nothing is in the beaker. In the second step ($Weight_2$), a quantity of solute, indicated by red dots, is poured into the beaker. The balance now reads the total weight of the beaker and the solute. The solute has not dissolved at this point. At the last step ($Weight_3$), a solvent, in this case blue dots, is added to the beaker, and the solute dissolves and creates a homogeneous solution. The combined weight read by the balance is now the aggregate mass of the beaker, the solute, and the solvent, demonstrating the law of conservation of mass. This value clearly illustrates how a solution is created and how mass is conserved during the process.

Calculations for Determining Solute Mass and its error:

To determine the actual weight of the **Solute** added to the solution, we can use the following calculations:

a) Solute doesn't contain water in nature:

The weight of the solute is simply the difference between the second and first weight measurements

$$\mathbf{Weight\ of\ solute_0 = Weight_2 - Weight_1} \quad \text{..... (1)}$$

Where,

$Weight_1$ is the weight of the empty beaker

$Weight_2$ is the weight of the beaker with the added solute.

b) Solute contain water in nature (hydrated solute):
In this case, the actual weight of the solute is calculated by accounting for the percentage of the solute in its hydrated form.

$$\textbf{Weight of solute} = (\textbf{Weight}_2 - \textbf{Weight}_1) \times \frac{\textbf{Percentage of pure solute}}{\textbf{100}} \quad \textbf{..... (2)}$$

In this research solute doesn't contain water in nature thus to calculate weight of solute the role of weight balance and its error is two times. By method of propagation of errors, the error in weight of Solute is given by,

$$\textbf{Error in Weight of Solute}_0 = \sqrt{(\textbf{Instrumetal Error})^2 + (\textbf{Instrumetal Error})^2}$$

Calculations for Determining Solvent Mass and its error:
To determine the actual weight of the **Solvent** added to the solution, we can use the following calculations:

a) Solute doesn't contain water in nature:

$$\textbf{Weight of Solvent}_0 = \textbf{Weight}_3 - \textbf{Weight}_2$$

Where,
$Weight_3$ is the weight of the empty beaker with solution.
$Weight_2$ is the weight of the beaker with the added solute.

b) Solute contain water in nature (hydrated solute):
In this case, the actual weight of the solute must be considered from Equation (2):

$$\textbf{Weight of Solvent} = \textbf{Weight}_3 - \textbf{Weight}_2 - \textbf{Weight of solute (in equation 2)}$$

By method of propagation of errors, the error in weight of Solvent is given by,

$$\textbf{Error in Weight of Solvent}_0 = \sqrt{(\textbf{Instrumetal Error})^2 + (\textbf{Instrumetal Error})^2}$$

Calculations for Molality and its error:

Now the molality is calculated by the below formula,

$$\textbf{Molality} = \frac{\textbf{Weight of Solute}_0 \times \textbf{1000}}{\textbf{Molecular weight of solute} \times \textbf{Weight of Solvent}_0}$$

By method of propagation of errors, the error in Molality calculated as,

$$\textbf{Error in Molality} = \textbf{Molality} \times \sqrt{\left(\frac{\textbf{Error in Weight of Solute}_0}{\textbf{Weight of solute}_0}\right)^2 + \left(\frac{\textbf{Error in Weight of Solvent}_0}{\textbf{Weight of Solvent}_0}\right)^2}$$

4.2 Preparation of Diluted Solution

Once the original solution is prepared, **dilution** can be performed to obtain solutions of lower concentrations.

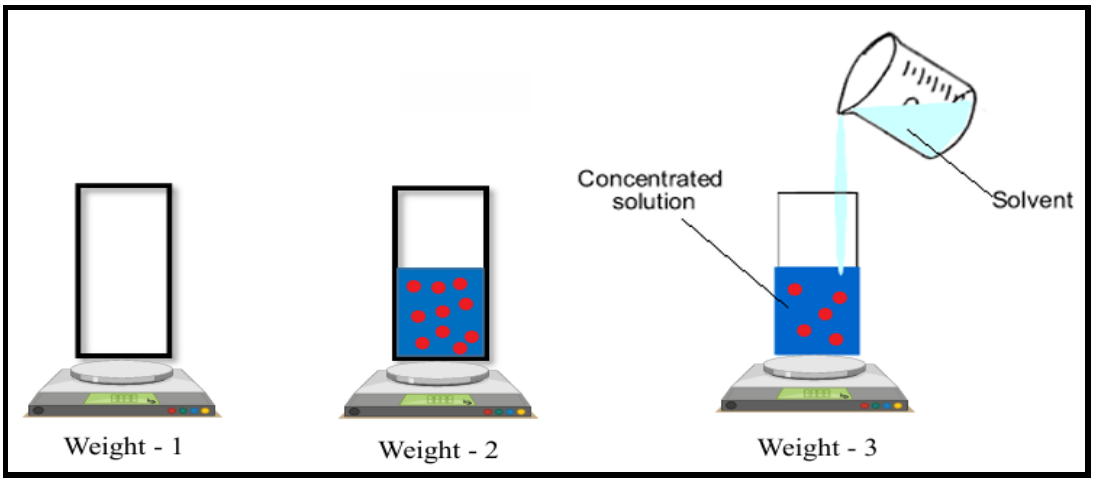


Figure 3: Preparation of Diluted solution

In order to make a diluted solution from the original concentrated solution, a dry and clean beaker is initially weighed on a precision balance, and its weight is noted ($Weight_1$). Then, a measured amount of the concentrated solution, which contains both solute and solvent, is poured into the beaker. The new weight shown on the balance is equivalent to the total mass of the beaker and the concentrated solution ($Weight_2$). Last, a predetermined amount of extra solvent is slowly added to the beaker in order to get the desired dilution. This decreases the concentration of the solute but maintains the total mass conserved. The last weight recorded on the balance ($Weight_3$) is the sum of the beaker, the original concentrated solution, and the solvent added.

Calculations for Determining Solute Mass and its error:

Since a portion of the original solution is taken for dilution, the true weight of the solute in the diluted solution has to be calculated from the fraction that reflects the ratio of solute to the total weight of the original solution. This fraction takes into consideration the proportion of solute to the total weight of the original solution. Based on this proportional relationship, the mass of the solute in the diluted solution can be precisely calculated using the formula below:

$$\mathbf{Weight\ of\ solute_1} = (\mathbf{Weight_2} - \mathbf{Weight_1}) \times \frac{\mathbf{Weight\ of\ solute_0}}{\mathbf{Weight\ of\ original\ solution}}$$

Where,

$Weight\ of\ solute_0$ is the weight of solute in original solution.

$Weight\ of\ solute_1$ is the weight of solute in diluted solution.

Additionally, the associated error in the solute mass must be evaluated using error propagation principles, ensuring precision in the final measurements. By method of propagation of errors, the error in Weight of $Solute_1$ is given by,

$$\text{Error in Weight of solute}_1 = \text{Weight of Solute}_1 \times \sqrt{\left(\frac{2 * \text{Instrumental Error}}{\text{Weight}_2 - \text{Weight}_1}\right)^2 + \left(\frac{\text{Error in Weight of solute}_0}{\text{Weight of solute}_0}\right)^2 + \left(\frac{\text{Error in Weight of original solution}}{\text{Weight of original solution}}\right)^2}$$

Calculations for Determining Solvent Mass and its error:

In the dilution process, the weight of the solvent in the diluted solution can be determined by subtracting the weight of the solute$_1$. The solvent mass in the diluted solution is calculated using the following formula:

$$\text{Weight of solvent}_1 = \text{Weight}_3 - \text{Weight}_2 - \text{Weight of solute}_1$$

Where,

Weight$_2$= Weight of the beaker with the taken portion of the original solution

Weight$_3$= Weight of the beaker after adding additional solvent.

Weight of solute$_1$ = Weight of Solute in diluted Solution.

$$\text{Error in Weight of Solvent}_1 = \sqrt{2 \times (\text{Instrumental Error})^2 + (\text{ Error in Weight of solute}_1)^2}$$

Calculations for Molality and its error:

The molality of the diluted solution is calculated in the same manner as for the original solution.

$$\text{Molality} = \frac{\text{Weight of Solute}_1 \times 1000}{\text{Molecular weight of solute} \times \text{Weight of Solvent}_1}$$

By method of propagation of errors, the error in Molality calculated as,

$$\text{Error in Molality} = \text{Molality} \times \sqrt{\left(\frac{\text{Error in Weight of Solute}_1}{\text{Weight of solute}_1}\right)^2 + \left(\frac{\text{Error in Weight of Solvent}_1}{\text{Weight of Solvent}_1}\right)^2}$$

5. Methodology

5.1 Study Design

This study is carried out in physical chemistry, with the major concentration being on the interactions between solute and solvent of different compounds. As most physicochemical properties are concentration dependent, accurate determination of solution concentration becomes a necessity. In this paper, molality has been selected as the main measure of concentration because it is unaffected by temperature fluctuations, thus being superior to molarity.

To guarantee accuracy while preparing solutions, there is a need to quantify errors resulting from various experimental steps. These errors can greatly affect the determination of physicochemical properties, and as such, a systematic process must be used to quantify them. The study thus seeks to create a predictive model for error in molality based on statistical and computational methods. The research also includes a software solution in the form of a Graphical User Interface (GUI), which performs error calculations automatically from experimental inputs.

5.2 Variables

In this study, two variables are taken into account:

Independent Variable: Molality of the solution, which is the input of primary interest.

Dependent Variable: Error in molality, which is calculated using the propagation of errors method and interpreted using statistical modeling.

As error in molality is caused by uncertainties in weight measurements and other experimental conditions, it is important to accurately capture and measure these errors in order to enhance the accuracy of physicochemical calculations.

5.3 Data collection

Experimental data was collected from chemistry research students at the Molecular Thermodynamics and Modeling Laboratory, Department of Chemistry, Shivaji University, Kolhapur. The data was derived from solutions prepared with care to ensure that differences in solute and solvent measurements were maintained systematically.

In order to verify the model developed, an outside dataset from a published research study was employed. This secondary dataset served to test the robustness and reliability of the regression model so as to ensure that it could generalize across the original experimental conditions well.

5.4 Data Analysis Techniques

5.4.1 Bivariate Analysis

Scatter plot: A scatterplot displays the relationship between two continuous variables, allowing for the visualization of trends, patterns, or correlations between them. It helps in identifying linear or non-linear associations and outliers in the data.

Correlation test: This test is used to measure the strength and direction of the linear relationship between two continuous variables. The Pearson correlation coefficient (r) is commonly used, with values ranging from -1 to 1, indicating the degree of correlation.

Linear Regression: Linear regression is a statistical method used to model the relationship between a dependent variable and one or more independent variables. It helps estimate the effect of independent variables on the dependent variable and predict outcomes based on given data. In this research, linear regression is employed primarily to explore the relationship between the dependent and independent variables, providing an initial understanding of their correlation and trend.

5.4.2 Statistical Model

Polynomial Regression Model:

Polynomial regression is an extension of linear regression that models the relationship between the dependent and independent variables by fitting a polynomial equation to the data. This approach allows for capturing non-linear patterns and more complex relationships that a simple linear model may not accurately represent. In this research, polynomial regression is used to improve the accuracy of predictions and better represent the underlying trends in the data.

5.4.3 Model Adequacy Checking

Heteroscedasticity: Heteroscedasticity occurs when the variance of residuals is not constant across all levels of the independent variable(s). This violates a key assumption of regression models and can lead to inefficient and biased parameter estimates. In this research, reciprocal transformation was applied to stabilize the variance and mitigate the impact of heteroscedasticity, ensuring a more reliable model fit.

Autocorrelation: Autocorrelation arises when residuals are correlated across observations, indicating that the model does not fully capture the underlying data patterns. This can lead to misleading statistical inferences and inefficient estimates. To address this issue, the Cochrane-Orcutt estimation method was employed, which iteratively adjusts the model to correct for serial correlation, improving the accuracy and reliability of parameter estimation.

5.4.4 Model Validation

Literature data validation of the model is a critical process to confirm the stability of the model outside the data on which the model was developed. Through independent, already published data, the model can be tested and assessed under varying conditions and data distributions. Cross-validation not only reinforces the validity of the model but also proves its applicability and relevance in similar research settings.

5.4.5 *Graphical User Interface*(GUI)

A *Graphical User Interface* (GUI) is a user-friendly visual platform that allows users to interact with software through graphical elements such as buttons, input fields, and displays, rather than relying on command-line operations. In this research, a custom-built GUI was developed to assist researchers in calculating errors in physicochemical properties. Specifically, the error in molality of

solutions. The GUI allows users to enter experimental measurements such as the weights of the beaker, solute, and solvent, and then automatically computes the error in molality using the standard formulae from the method of propagation of errors implemented in the backend. This eliminates the need for manual calculations, reduces the risk of human error, and ensures consistency and accuracy in results. The GUI not only streamlines the workflow for researchers but also serves as a practical tool for improving the precision and reliability of experimental data analysis in physical chemistry.

6. Analysis and Results

6.1 Bivariate Analysis

To understand the relationship between Molality and its error. I have plotted scatter plot Molality Vs Error in Molality. Pearson method is used to check correlation. The linear regression technique used to derive relationship between Molality and Error in Molality. Molality is regressed with Error in molality.

6.1.1 Scatter plot

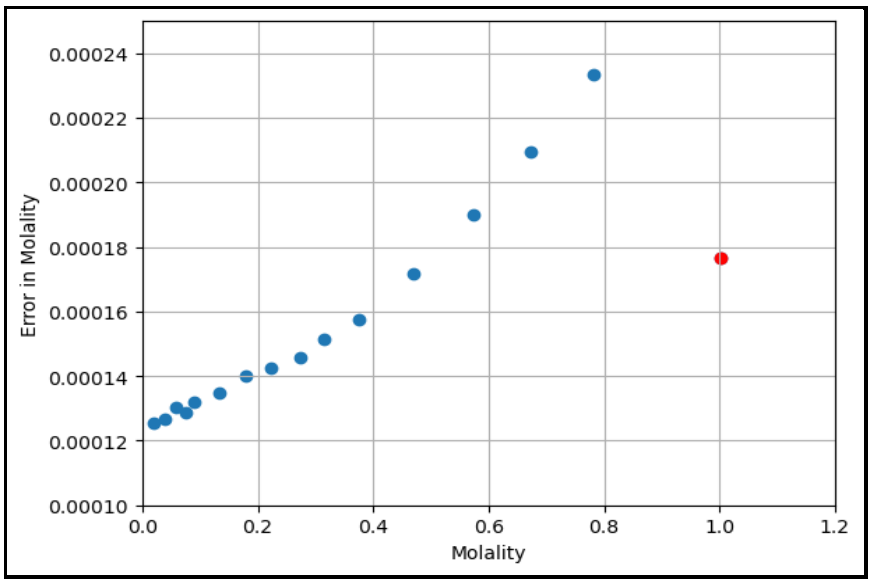


Figure 4: Scatter plot of Molality Vs Error in Molality

Figure 4 is a scatter plot illustrating the relationship of Molality (x-axis) and Error in Molality (y-axis). The red dot represents the original solution, and the blue dots correspond to the diluted solutions that are derived from it. For the original solution, molality as well as the error in molality are simply obtained using standard formulae. But for diluted solutions, the calculation is more involved since the mass of the solute is found by multiplying the proportion of the original solute present in the diluted solution. This change in the method of calculation leads to a different pattern of error propagation for diluted solutions than for the original solution. As a result, the original solution's error behavior is distinct from that of the diluted solutions from the scatter plot.

6.1.2 Correlation test

To determine the strength and direction of the linear relationship between molality and its error, a Pearson correlation test was used. The findings are as follows:

	Pearson Correlation Coefficient
With original solution	0.64
Without original solution	0.98

Table 2: Correlation between molality and error in molality

When the original solution point is taken into account, the Pearson correlation coefficient is 0.64, which reflects a moderate positive correlation between molality and its error. But when the original solution point is not taken into account—only the diluted solutions are considered—the correlation coefficient becomes much higher at 0.98, reflecting a strong positive linear correlation. This means that for diluted solutions, as molality goes up, the corresponding error also goes up in a very consistent and predictable way.

6.1.3 Linear Regression

In order to model the relationship between molality and error associated with it, a linear regression analysis was performed. The technique is effective for the quantification of strength and direction of an association between two variables based on measures such as the regression coefficients, p-values, and the coefficient of determination (R^2).

The linear regression equation obtained is:

$$\text{Error in Molality} = 0.00009427 \times \text{Molality} + 0.00012481$$

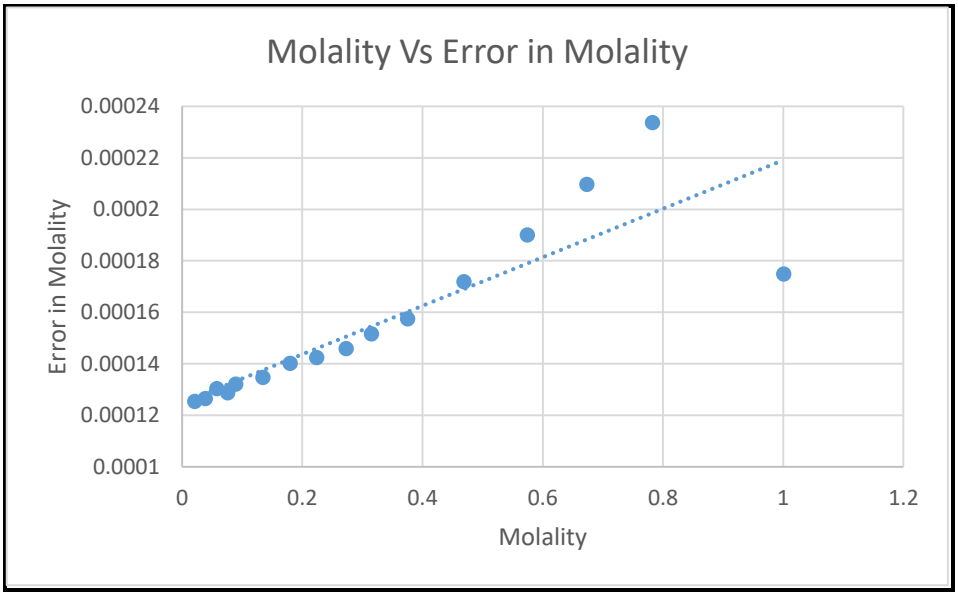


Figure 5: Fitting of linear trend line to molality vs error in molality

	Coefficients	p-value	R^2
Intercept	0.00012481	<0.001	0.7506
Molality	0.00009427	<0.001	

Table 3: Summary of model parameters for original solution

This R^2 of approximately 0.75 shows that molality explains roughly 75% of error variation and represents a moderately strong linear relationship. The extremely low p-values (< 0.001) for the coefficients tell us that the model terms are significant, as they validate the reliability of the model. Note that the addition of the original solution point can also cause minor lack of linearity since the original solution is worked out differently than the diluted solutions. If we remove the original solution and take only the diluted solutions, then the model clearly shows an even better linear fit, enabling enhanced generalization. This implies that the behavior of error in diluted solutions exhibits a systematic linear trend, thereby making the linear model more valid and consistent when extrapolated to analogous experimental conditions.

Once the initial solution point was removed, a new linear regression model was calibrated with only the diluted solution data, and it provided a much better model performance. The revised regression equation is:

$$\text{Error in Molality} = 0.0001326 \times \text{Molality} + 0.0001168$$

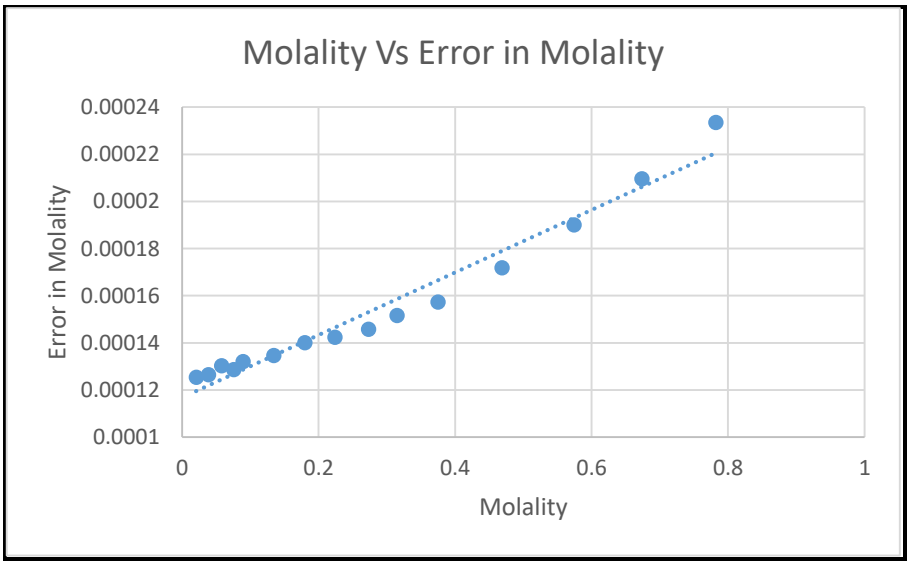


Figure 6: Linear trend fitting to diluted solutions

	Coefficients	p-value	R^2
Intercept	0.0001168	<0.001	0.9638
Molality	0.0001326	<0.001	

Table 4: Summary of model parameters for diluted solutions

This high R^2 value (96.38%) indicates a very strong linear relationship between molality and error in molality across the diluted solution data, suggesting that the model explains nearly all of the variation observed.

However, despite this strong fit, it is important to recognize that the magnitude of error values lies in the third or fourth decimal place, making the data extremely sensitive to even small prediction deviations. In such cases, a simple linear model, though statistically sound, may still result in noticeable inaccuracies in predictions due to its limited capacity to capture subtle nonlinear variations present in error behavior.

Hence, to further improve the precision and reliability of error modeling—particularly in precision-critical chemical experiments—it is worthwhile to seek polynomial regression models of higher order. These are able to capture nonlinear trends in the data more accurately and possibly decrease residual errors so that the reliability of error estimation in molality for different solution concentrations improves.

6.2 Statistical Model

6.2.1 Polynomial Regression Model

To improve the precision and reliability of error estimation in molality, a second-degree polynomial regression model was employed. The model equation is:

$$\text{Error in Molality} = 0.000126 + 0.000043 \times \text{Molality} + 0.000121 \times \text{Molality}^2$$

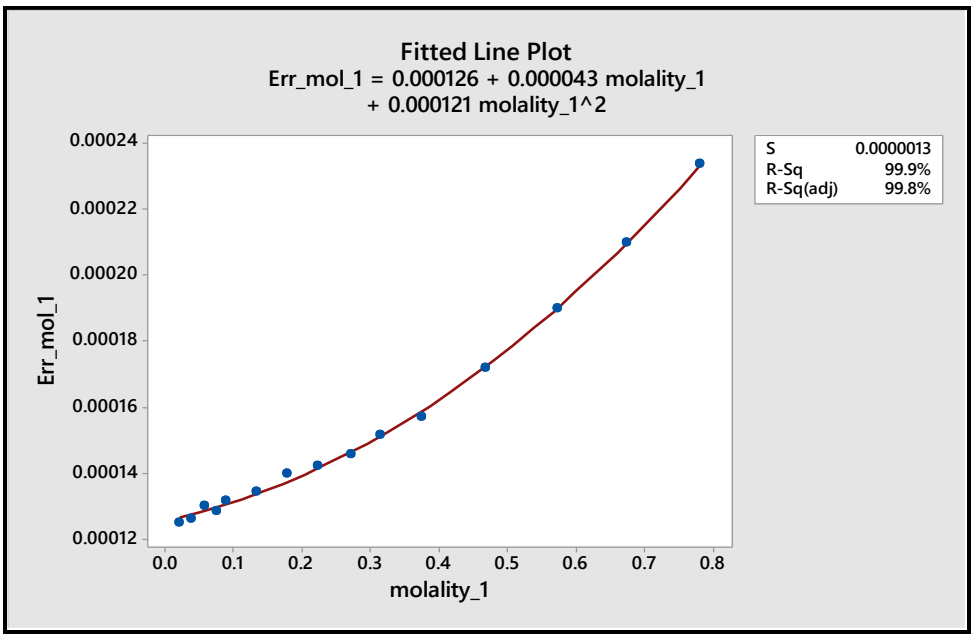


Figure 7: Polynomial fitting between molality and error in molality for diluted solutions

The model is a better fit than the linear model according to the large coefficient of determination (R^2 = 99.9%) and adjusted R^2 (99.8%), which implies the model accounts for nearly all variability in the measured errors. Further, the standard error (S = 0.0000013) is extremely low, implying that deviation of the model-predicted values from data points is close to zero. In the analogous plot, the red curve is the fitted quadratic regression line and the blue dots are the observed data. The curve closely mimics the trend of the observed data and visually ensures that the model is a superb fit. The improved model accurately describes the non-linear relationship between molality and its corresponding error and hence is better equipped for prediction and minimizing experimental uncertainties, especially for sensitive physicochemical measurements.

Residual Diagnosis for fitted model:

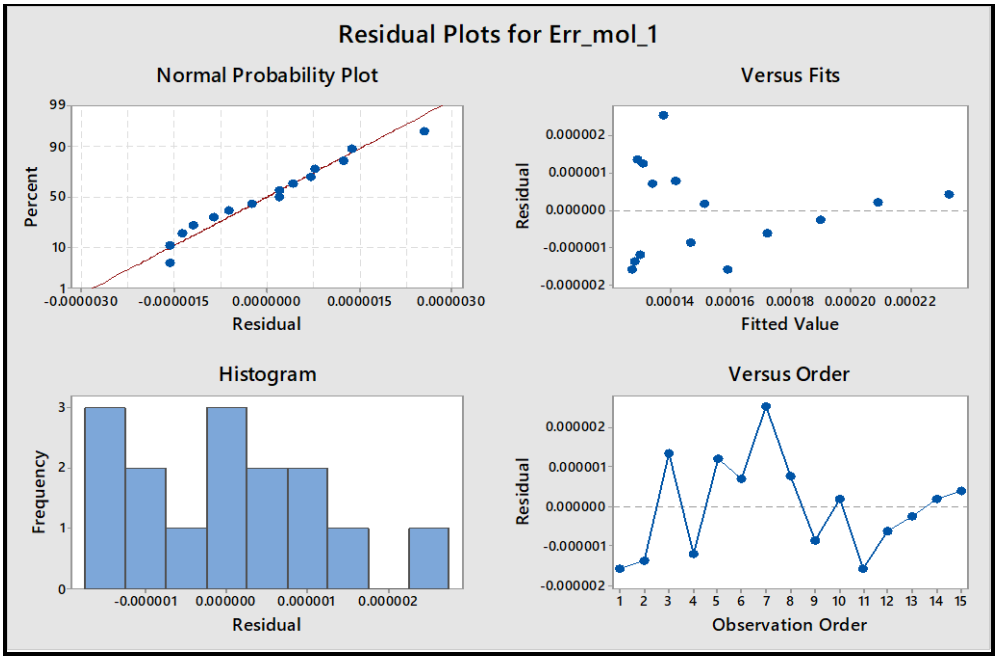


Figure 8: Residual diagnosis plots

Normal Probability Plot (Top Left):

The residuals mostly fall along the straight line. This suggests that the residuals are approximately normally distributed, satisfying one of the key assumptions of regression.

Residuals vs Fitted Values (Top Right):

Ideally, this plot should show residuals randomly scattered around zero, without any pattern. However, in this case, a pattern is evident, suggesting the presence of non-constant variance (heteroscedasticity). This violates another key assumption of linear regression and indicates the need for an appropriate transformation or alternative modeling approach to stabilize the variance.

Histogram of Residuals (Bottom Left):
The distribution is slightly skewed but overall looks reasonably symmetric. It supports the assumption of normality, though with a slight deviation on the right tail.

Residuals vs Observation Order (Bottom Right):
Residuals alternate up and down but no systematic trend is observed. Further analysis is required.

6.3 Model Adequacy Checking

6.3.1 Heteroscedasticity

To address the issue of non-constant variance observed in the residuals of the previous model, a reciprocal transformation was applied to the error values. The resulting quadratic regression model is expressed as:

$$\frac{1}{y} = -520.41\,x^2 - 4441.98\,x + 8033.66$$

or

$$\frac{1}{Error\ in\ Molality} = -520.41\,Molality^2 - 4441.98\,Molality + 8033.66$$

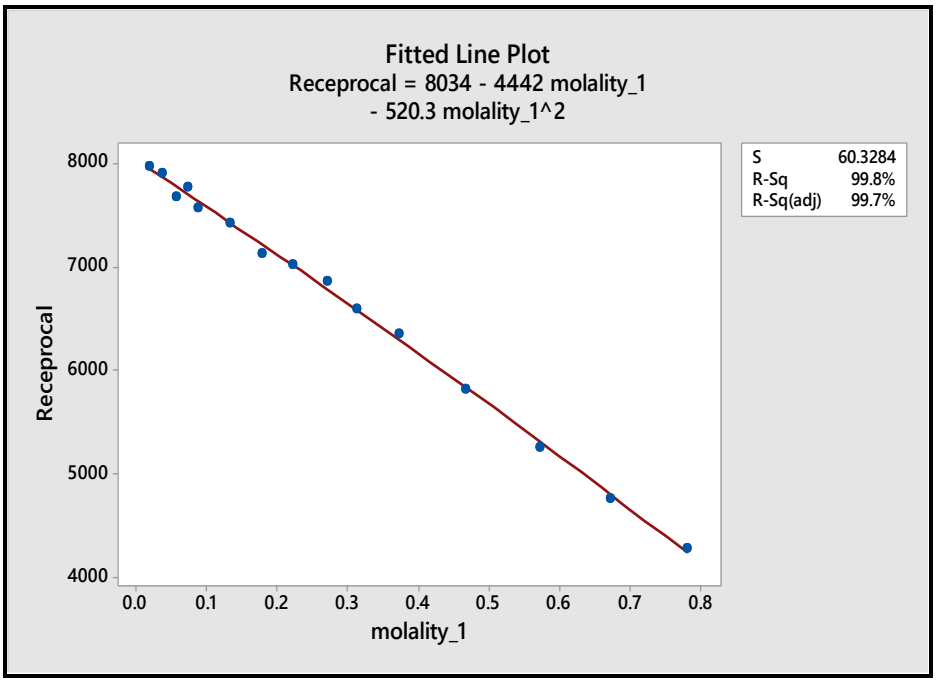


Figure 9: After Reciprocal Transformation fitted line plot

This transformation significantly improved the model fit, with an R^2 value of 99.8% and adjusted R^2 of 99.7%, indicating that nearly all variability in the reciprocal of error is explained by the model. Additionally, the standard error (S) is reasonably low at 60.3284, supporting the model's accuracy.

Residual Diagnosis for fitted transformed model:

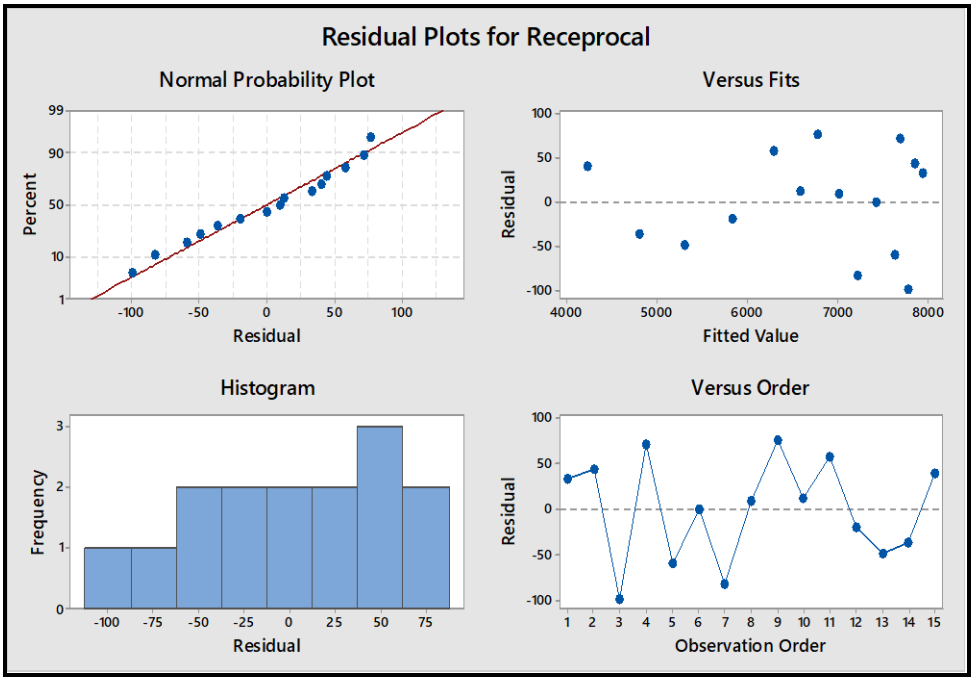


Figure 10: Residual diagnosis for fitted transformed model

Normal Probability Plot (Top Left):

This shows points closely aligning with the straight line, confirming normality; satisfying one of the key assumptions of regression.

Residuals vs Fitted Values (Top Right):

This plot shows randomness, indicating that the assumption of homoscedasticity (constant variance) is now satisfied.

Histogram of Residuals (Bottom Left):

The histogram of residuals suggests an approximately symmetric distribution, it supports the assumption of normality, though with a slight deviation on the right tail.

Residuals vs Observation Order (Bottom Right):

while the residuals vs observation order plot exhibits a **zig-zag pattern**, raising a slight **concern of negative autocorrelation**. Although this warrants further investigation, the reciprocal transformation overall appears effective in enhancing the model's reliability and in meeting the regression assumptions.

To confirm the presence of autocorrelation in the residuals of the reciprocal regression model, the Durbin-Watson (DW) test was applied. This test statistically measures the extent of correlation between adjacent residuals in a regression analysis.

After Receprocal Transformation, the residuals

```
[4]: import numpy as np
     residuals = np.array([
         33.19036275, 44.49418967, -99.32980431, 72.09703213, -59.04627108,
         -0.017462955, -82.72435721, 9.589931917, 76.27992488, 12.60768226,
         58.2094876, -19.58067809, -49.3363036, -36.40664667, 39.97291271])
     print(residuals)
```

```
[ 3.31903627e+01  4.44941897e+01 -9.93298043e+01  7.20970321e+01
 -5.90462711e+01 -1.74629550e-02 -8.27243572e+01  9.58993192e+00
  7.62799249e+01  1.26076823e+01  5.82094876e+01 -1.95806781e+01
 -4.93363036e+01 -3.64066467e+01  3.99729127e+01]
```

```
[6]: from statsmodels.stats.stattools import durbin_watson
     dw_statistic = durbin_watson(residuals)
     print(f"Durbin-Watson statistic: {dw_statistic:.4f}")
```

```
Durbin-Watson statistic: 2.5133
```

Figure 11: Durbin Watson test in Python

After computing the residuals from the fitted model, the **Durbin-Watson statistic** was found to be **2.5133**. In the DW test, a value near 2 indicates no autocorrelation, values below 2 suggest positive autocorrelation, and values above 2 suggest negative autocorrelation. Since the observed value is slightly above 2, it indicates the **presence of negative autocorrelation** in the residuals. This result aligns with the earlier visual observation from the residual vs observation order plot, which displayed a zig-zag pattern—a common sign of negative autocorrelation. Identifying and addressing such patterns is crucial to ensure the robustness and reliability of the regression model used for predicting error in molality.

6.3.2 Autocorrelation

To overcome the problem of autocorrelation identified in the residuals of the polynomial regression model, the Cochrane-Orcutt iterative procedure was employed. This method is particularly effective for addressing first-order autocorrelation in time-series or sequential data, where residuals are not independent. The original quadratic model is defined as:

$$y_t = \beta_0 + \beta_1 x_t + \beta_2 x_t^2 + \mu_t$$

with the assumption that the error term follows an AR (1) process:

$$\mu_t = \rho\, \mu_{t-1} + e_t, \quad |\rho|<1$$

The process begins by estimating the model parameters using **Ordinary Least Squares (OLS)** and computing residuals. Upon confirming autocorrelation, the next step is to **estimate the autocorrelation coefficient ρ** by regressing current residuals on their lagged values. With the estimated ρ, the dependent and independent variables are transformed accordingly:

- Regress μ_t on μ_{t-1}: $\hat{\mu}_t = \rho\, \widehat{\mu_{t-1}} + e_t$

- The estimated **ρ** is: $\hat{\rho} = \frac{\sum_2^T \hat{\mu}_t \widehat{\mu_{t-1}}}{\sum_2^T \widehat{\mu_t^2}}$
- Transform the dependent variable: $y_t^* = y_t - \hat{\rho}\, y_{t-1}$
- Transform the independent variables:

$$x_t^* = x_t - \hat{\rho}\, x_{t-1}$$

$$x_t^{2^*} = x_t^2 - \hat{\rho}\, x_{t-1}^2$$

- Transform the intercept: $\beta_0^* = (1 - \hat{\rho}\,)\beta_0$

The transformed variables are then used to re-fit the model via OLS. This process is iterated—re-estimating residuals and updating ρ, until the value of ρ converges. The result is a revised polynomial model that corrects for autocorrelation, thus enhancing the reliability and accuracy of parameter estimates.

After applying the Cochrane-Orcutt procedure to correct for autocorrelation in the polynomial regression model, the estimated autocorrelation coefficient was found to be $\hat{\boldsymbol{\rho}} = \mathbf{-0.2985}$, indicating **negative autocorrelation** in the original residuals. Following the transformation, the refined quadratic regression model was re-estimated using the transformed variables.

Refined Model:

$$\frac{\mathbf{1}}{\boldsymbol{y}} = \mathbf{10414.59} - \mathbf{4350.2}\,\boldsymbol{x} - \mathbf{640.573}\,\boldsymbol{x^2}$$

$$\frac{\mathbf{1}}{\boldsymbol{Error\_Molality}} = \mathbf{10414.59} - \mathbf{4350.2}\,\boldsymbol{Molality} - \mathbf{640.573}\,\boldsymbol{Molality^2}$$

SUMMARY OUTPUT	
Regression Statistics	
R Square	0.9986
Adjusted R Square	0.9984
Standard Error	58.7607
Observations	14

Table 5: Model summary 1

	Coefficients	***Standard Error***	***t Stat***	***P-value***	***Lower 95%***	***Upper 95%***
Intercept	**10414.59**	38.2238	272.46	<0.001	10330.46	10498.72
X	**-4350.2**	207.1308	-21.01	<0.001	-4806.08	-3894.30
X^2	**-640.573**	268.6991	-2.38	0.036	-1231.97	-49.16

Table 6: Model summary 2

The regression results indicate a very strong model fit **R^2 = 0.9986** and **Adjusted R^2 = 0.9984**, suggesting that the model explains over 99% of the variability in the response variable. The **standard error** of the estimate is relatively low at **58.76**, indicating that residuals are closely clustered around the fitted line.

All coefficients are statistically significant, with p-values less than 0.05, and These values are accompanied by narrow 95% confidence intervals, suggesting precise estimates.
Overall, the Cochrane-Orcutt corrected model not only maintains a high degree of fit but also addresses the violation of the autocorrelation assumption in the original model, thereby improving the reliability and interpretability of the regression results.

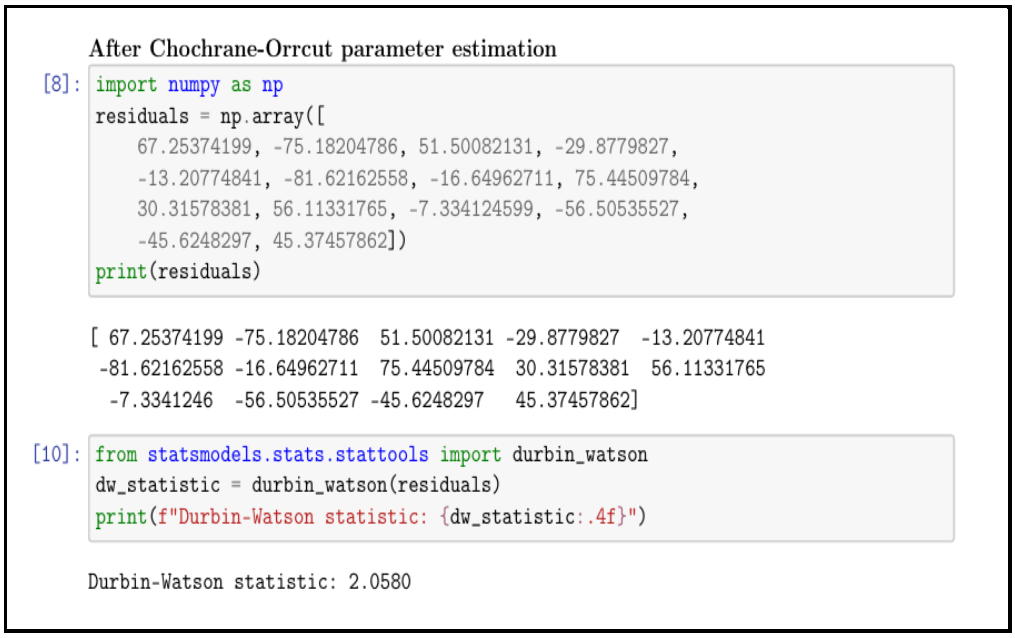


Figure 12: Durbin Watson test for refined model in Python

After estimating the regression parameters using the Cochrane-Orcutt method to correct for autocorrelation, the model residuals were re-evaluated using the Durbin-Watson test. The resulting Durbin-Watson statistic is 2.0580, which is very close to the ideal value of 2, indicating that autocorrelation is no longer present in the residuals.

6.4 Model Validation

Testing efficiency of model using molality and its error in JCED_2018 paper:

Experiment No.	Molality	Error	Experiment No.	Molality	Error
1	0.4918	0.000308	9	0.1409	0.000164
2	0.4103	0.000312	10	0.131	0.000161
3	0.3734	0.000292	11	0.118	0.000145
4	0.3189	0.000256	12	0.098	0.000076
5	0.2686	0.000227	13	0.0783	0.00007
6	0.2254	0.000203	14	0.0589	0.000055
7	0.1903	0.000188	15	0.0391	0.000043
8	0.1627	0.000172	16	0.0188	0.00003

Table 7: Molality and Error in molality in JCED_2018 research paper.

To validate the effectiveness of the refined regression model, an external dataset from the JCED_2018 paper was used. The dataset includes two categories:
The original solution, highlighted in red, has a concentration of 0.4918, from which all other solutions were prepared through dilution. Additionally, at a concentration of 0.118, a further dilution was performed to obtain double-diluted solutions. When examining the behavior of errors, the diluted solutions (blue points) exhibit a clear polynomial trend, which aligns well with the behavior captured by the constructed regression model. Therefore, the model can be effectively applied in two distinct parts—one for diluted solutions and another for double-diluted solutions—ensuring better prediction accuracy across different dilution levels.

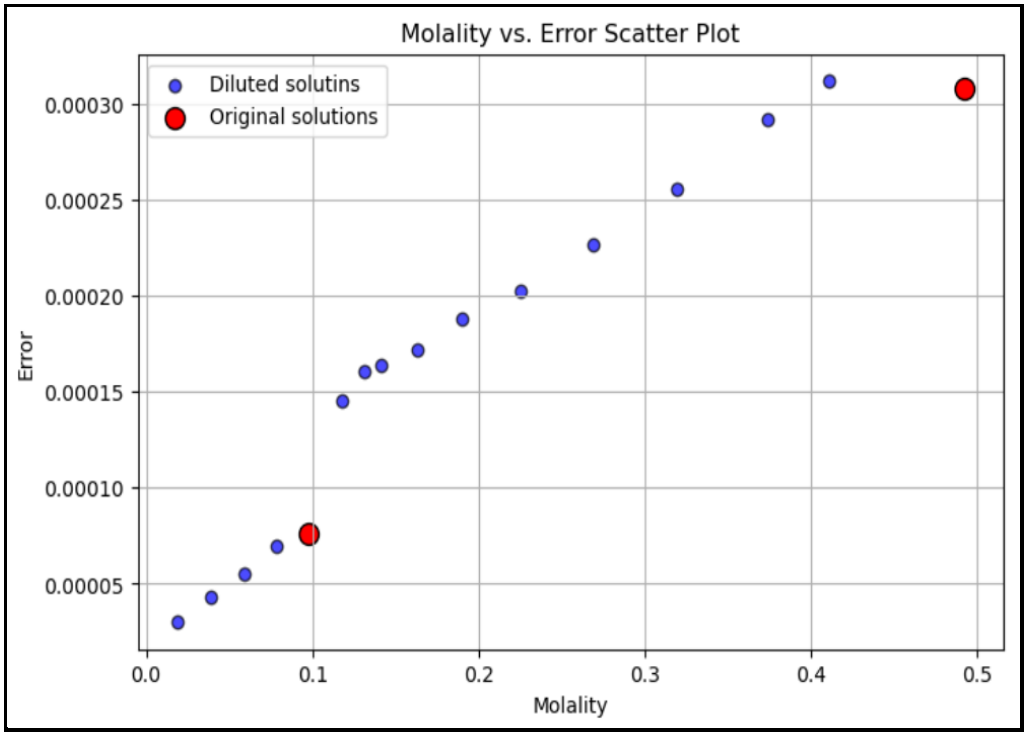


Figure 13: Scatter plot of Molality Vs Error in JCED_2018 paper

Experiment No.	Molality	Error	Using_model	Error_Hat	Residual
16	0.0188	0.00003	10332.58455	0.000097	-0.000067
15	0.0391	0.000043	10243.52262	0.000098	-0.000055
14	0.0589	0.000055	10156.14573	0.000098	-0.000043
13	0.0783	0.00007	10070.04688	0.000099	-0.000029

Table 8: Implementation of model on double diluted solutions

Experiment No.	Molality	Error	Using_model	Error_Hat	Residual
11	0.118	0.000145	9892.351963	0.000101088	0.0000439
10	0.131	0.000161	9833.725853	0.000101691	0.0000593
9	0.1409	0.000164	9788.934591	0.000102156	0.0000618
8	0.1627	0.000172	9689.860653	0.000103201	0.0000688
7	0.1903	0.000188	9563.554212	0.000104564	0.0000834
6	0.2254	0.000203	9401.515615	0.000106366	0.0000966
5	0.2686	0.000227	9199.916721	0.000108697	0.0001183
4	0.3189	0.000256	8962.172029	0.00011158	0.0001444
3	0.3734	0.000292	8700.917197	0.00011493	0.0001771
2	0.4103	0.000312	8521.870464	0.000117345	0.0001947

Table 9: Implementation of model on single diluted solutions

The table below shows the prediction results of the refined regression model when applied to molality-error data from external literature (JCED_2018). The "Error_Hat" column represents the error predicted using the model, while "Residual" is the difference between observed error and model-predicted error.

For low molality values (0.01–0.08), the model tends to overestimate the error (negative residuals). As molality increases beyond 0.1, the model underestimates the error, but the residuals remain small and consistent. This suggests the model captures the increasing trend of error with molality well, even when tested with external data.

The refined model demonstrates strong predictive power for diluted solutions, especially in the mid to high molality range. Minor discrepancies at very low molality may arise due to measurement sensitivities or transformation assumptions. However, the residuals remain within acceptable limits, affirming that the model is robust and applicable to real-world, literature-based datasets.

6.5 *Graphical User Interface*(GUI)

This Graphical User Interface (GUI) has been created using the Python programming language with the Tkinter library for the Department of Chemistry, Shivaji University, Kolhapur. The main aim of the GUI is to make and ease complex calculations by giving a user-friendly, interactive interface. It helps users to accomplish specific tasks efficiently without having thorough knowledge of programming or manual handling of data.

Designed with a focus on functionality and usability, the interface incorporates essential features such as data entry, result processing, report generation, and interactive controls tailored to the department's academic workflows. The GUI promotes ease of access, accuracy, and speed, ultimately enhancing the productivity of faculty, staff, and students.

Below is the formal step by step guide description for created GUI:

Step 1: Launching the GUI

Upon executing the code snippet, the graphical user interface (GUI) opens with a layout shown in figure 14 . The interface header prominently displays the Shivaji University, Kolhapur insignia and name, along with NAAC accreditation and institutional recognition details, establishing the academic credibility of the platform.

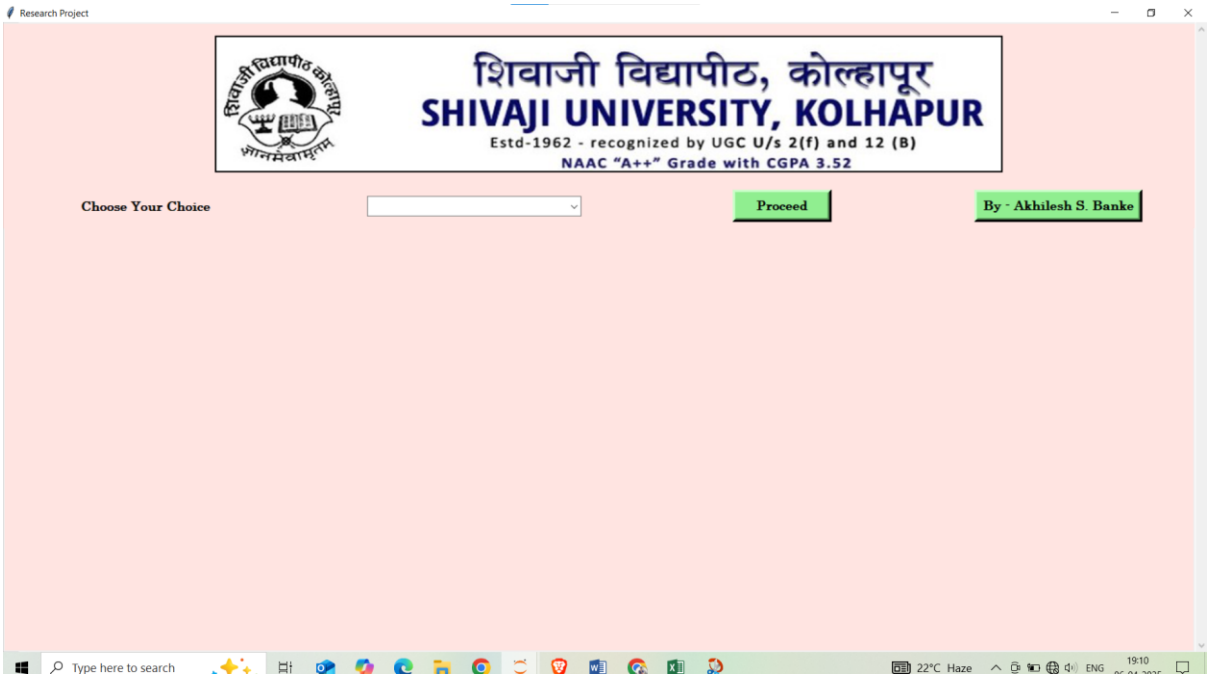


Figure 14: Welcome layout of GUI

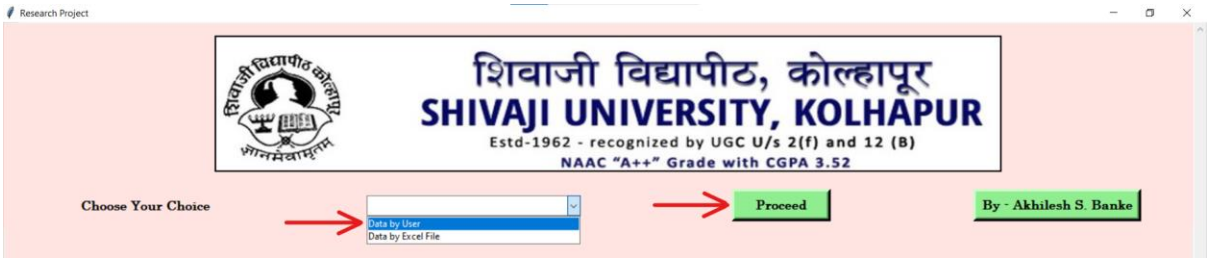


Figure 15: User Selection Panel

Step 2: User Selection Panel

After the application is launched, the user is prompted with the **"Choose Your Choice"** label, accompanied by a dropdown menu offering **two distinct input methods**:

1. **Data by User** – Allows manual data entry within the interface.
2. **Data by Excel File** – Facilitates data upload from an external Excel file.

To the right of the dropdown menu, a green "Proceed" button is positioned. Once a selection is made, clicking the "**Proceed**" button triggers the next module or data input panel based on the user's chosen method. This step ensures a structured workflow and simplifies user navigation by categorizing input methods at the very beginning.

6.5.1 When user chooses "Data by user" option:

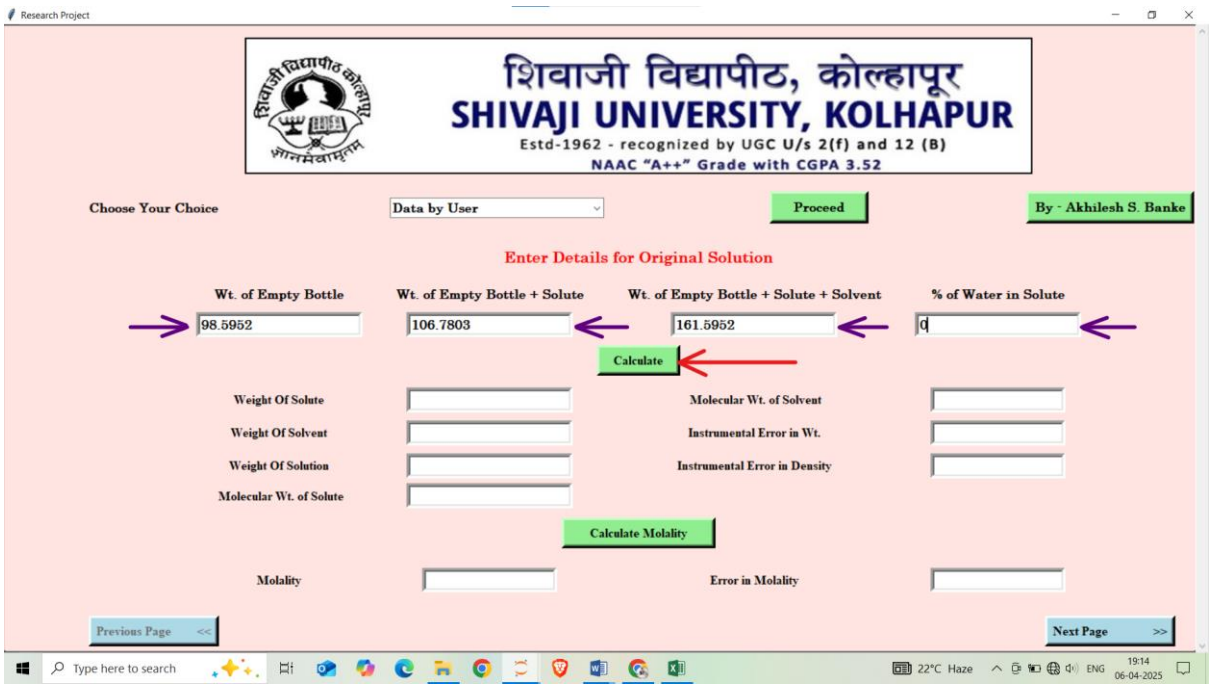


Figure 16: Manual Data Entry Panel

Step 3: Manual Data Entry – Original Solution Details

Upon selecting “Data by User” and clicking the “Proceed” button, the interface transitions to a structured input form for entering original solution data. The user is prompted to manually input the following values, as indicated by the violet arrows:

- Weight of Empty Bottle
- Weight of Empty Bottle + Solute
- Weight of Empty Bottle + Solute + Solvent
- Percentage of Water in Solute

These input fields ensure that all key parameters necessary for subsequent solution property calculations are accurately provided by the user. Once the relevant values are entered, the user can initiate the computation process by clicking the centrally placed green "**Calculate**" button, marked with a red arrow. This triggers internal calculations to determine various derived metrics such as:

- Weight of solute, solvent, and solution

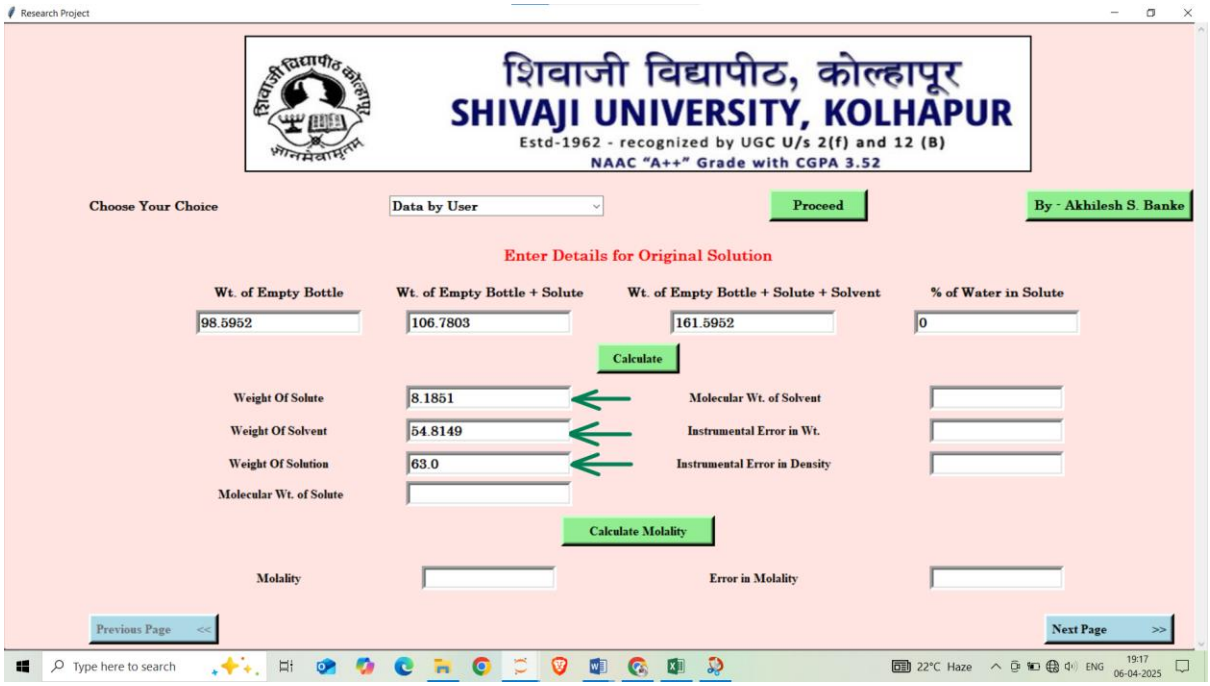


Figure 17: First Output of GUI (Marked by green arrows)

Step 4: Input Additional Data for Molality Calculation

In this step, the user is required to input additional parameters necessary for the precise calculation of molality. These inputs are marked by the violet arrows in the interface and include:

- Molecular Weight of Solute, Molecular Weight of Solvent
- Instrumental Error in Weight, Instrumental Error in Density

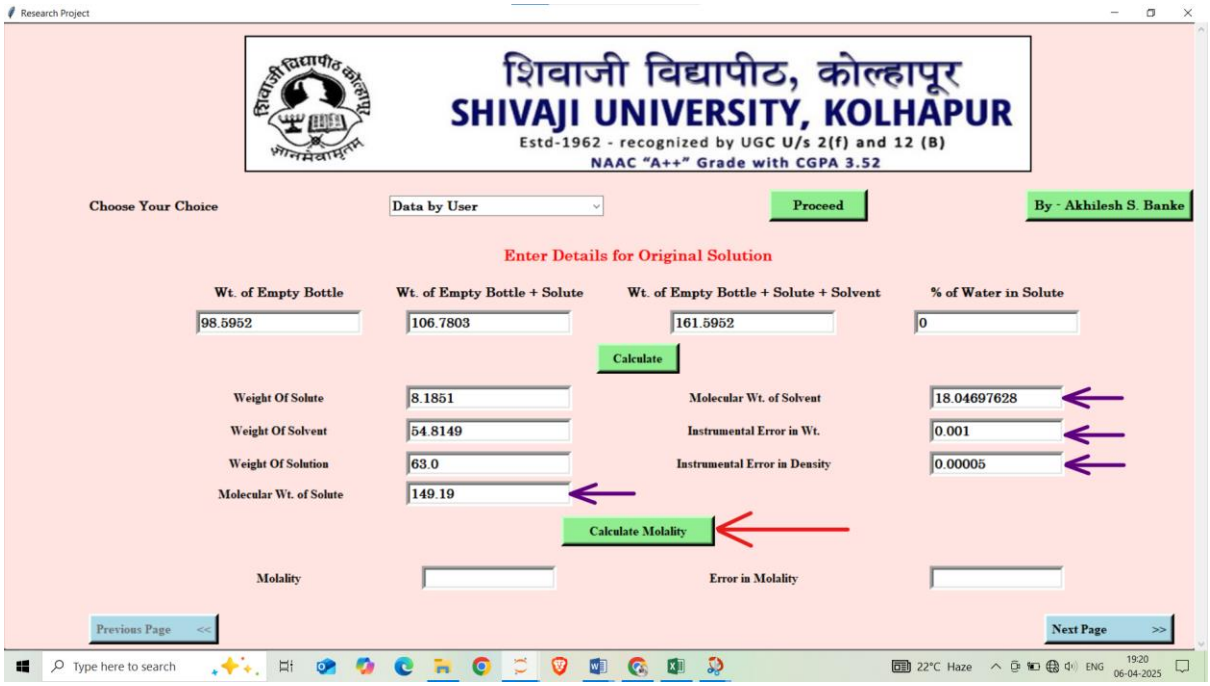


Figure 18: Input Additional Data for Molality Calculation

After entering the necessary data, the user must click the "**Calculate Molality**" button, highlighted by the red arrow, to generate Molality and Error in Molality for original solution.

These computed results will then be displayed in the corresponding fields located beneath the button. This functionality provides the user with critical information regarding the concentration of the solution and the margin of error associated with it.

Step 5: View Output and Proceed to Diluted Solution Calculations

Once all the required inputs have been provided, and the "Calculate Molality" button is pressed, the interface displays the calculated results:

- Molality
- Error in Molality

These outputs are shown in the fields marked by the green arrows. The error is computed using the method of propagation of errors, which is handled in the backend of the program to ensure scientifically accurate estimations.

To continue with the analysis, particularly for diluted solutions, the user should click the "**Next Page**" button, indicated by the red arrow. This will navigate to the subsequent interface designed specifically for handling and calculating parameters of diluted solutions. See figure 19.

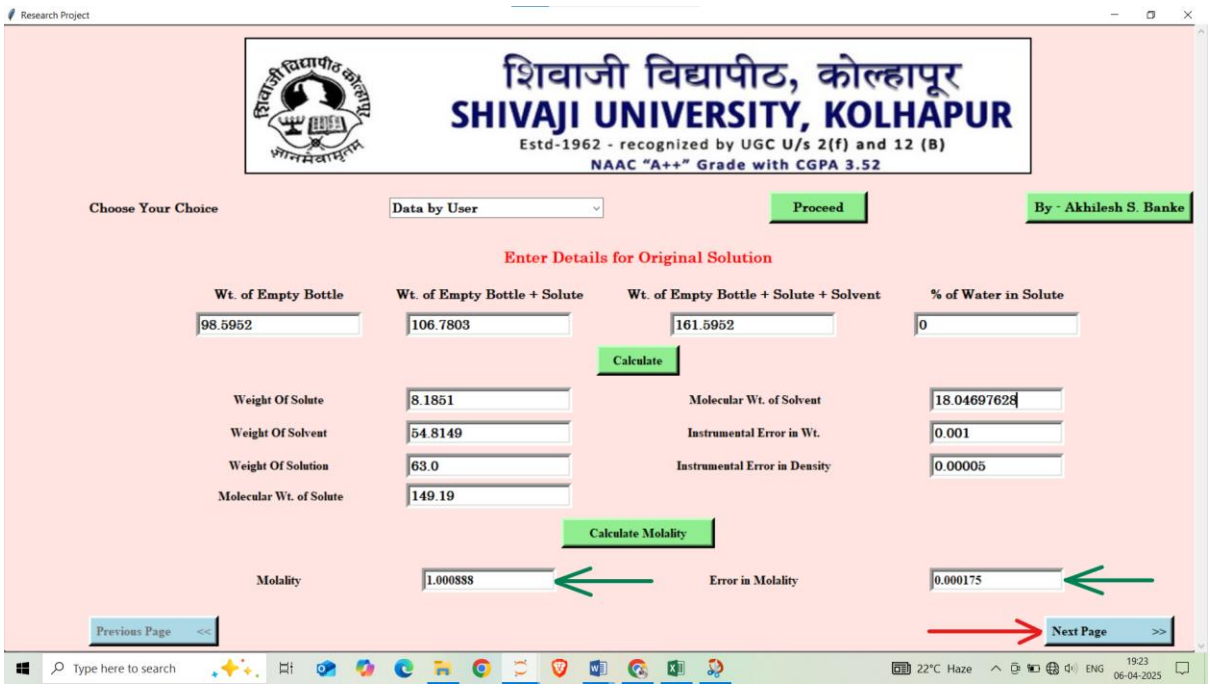


Figure 19: Molality and error in molality calculated.

Step 6: Enter Details for Diluted (Sub-) Solutions

After computing the molality and its associated error for the original solution, the user is directed to the next page by clicking the "**Next Page**" button. This interface is specifically designed to handle diluted or sub-solutions.

- The user is prompted to enter the number of sub-solutions to be prepared from the original solution in the input field indicated by the violet arrow.
- After specifying the number of sub-solutions (e.g., 3 in the figure 20), the user must click the "**Click Here**" button (indicated by the red arrow) to proceed.

This step initializes the fields for entering the respective weights and parameters for each diluted solution, which will be dynamically generated based on the number provided. The process ensures a flexible and interactive interface for handling multiple sub-solutions derived from the original solution.

Figure 20: Details for diluted solution

Step 7: Input Data for Sub-Solutions/Diluted Solutions

Once the number of sub-solutions is entered and the user clicks "**Click Here**", the interface dynamically generates an input grid for entering weight data corresponding to each sub-solution. Each sub-solution is assigned a label (e.g., Solution_1_Wt_1, Solution_1_Wt_2, Solution_1_Wt_3, etc.). User should enter value fields for diluted solutions as follows:

- Weight of empty bottle
- Weight of empty bottle + part of original solution
- Weight of empty bottle + part of original solution + added solvent.

The "**Calculate Molality & Errors**" button (indicated by the red arrow) must be clicked once all data is filled.

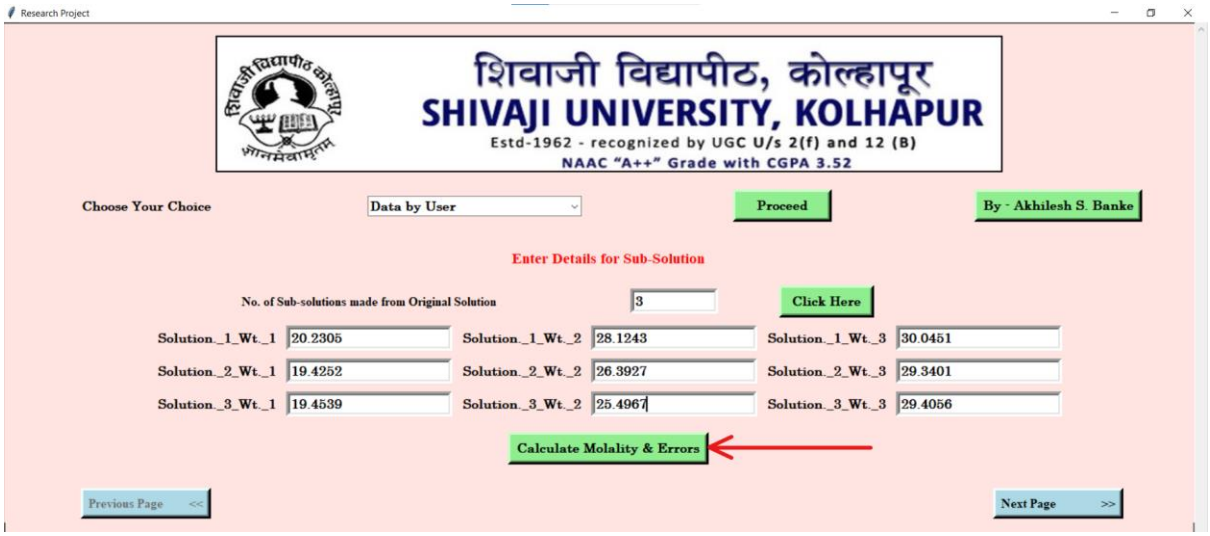


Figure 21: Input data for Sub-Solutions/Diluted Solutions

Step 8: Display and Export of Results

After clicking the "**Calculate Molality & Errors**" button, the interface displays the computed values of:

- Molality for each sub-solution
- Error in Molality, calculated using the method of propagation of errors

These results are shown just below the weight entry grid (indicated by green arrows).
To save or further analyze this data:

- The user can click the "Export Output in Excel file" button (indicated by the red arrow).
- This action will export the displayed molality and error values into an Excel spreadsheet format for documentation, sharing, or advanced analysis.

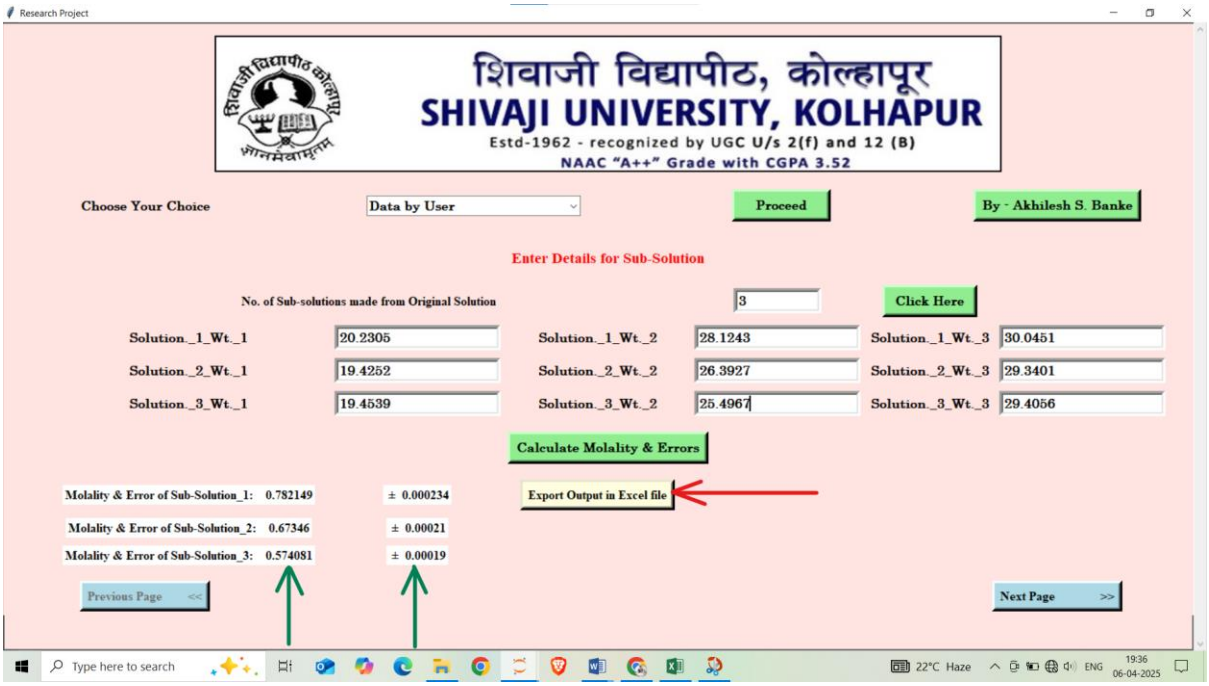


Figure 22: Display and Export of Results

Step 9: Saving Output to Excel File

When the user clicks on the **"Export Output in Excel file"** button, a **file dialog box** appears, as shown in the figure 23.

- The dialog allows the user to choose the **location** and **name of the Excel file** where the results (molality and error in molality for sub-solutions) will be saved.
- The user should:
    - Enter a suitable **file name** (e.g., Output_Molality_and_Error) — indicated by the **upper red arrow**.
    - Click the **"Save"** button — indicated by the **lower red arrow** — to complete the export.

This saves all calculated values into an Excel spreadsheet, making it easier for users to store, analyze, and share the experimental data.

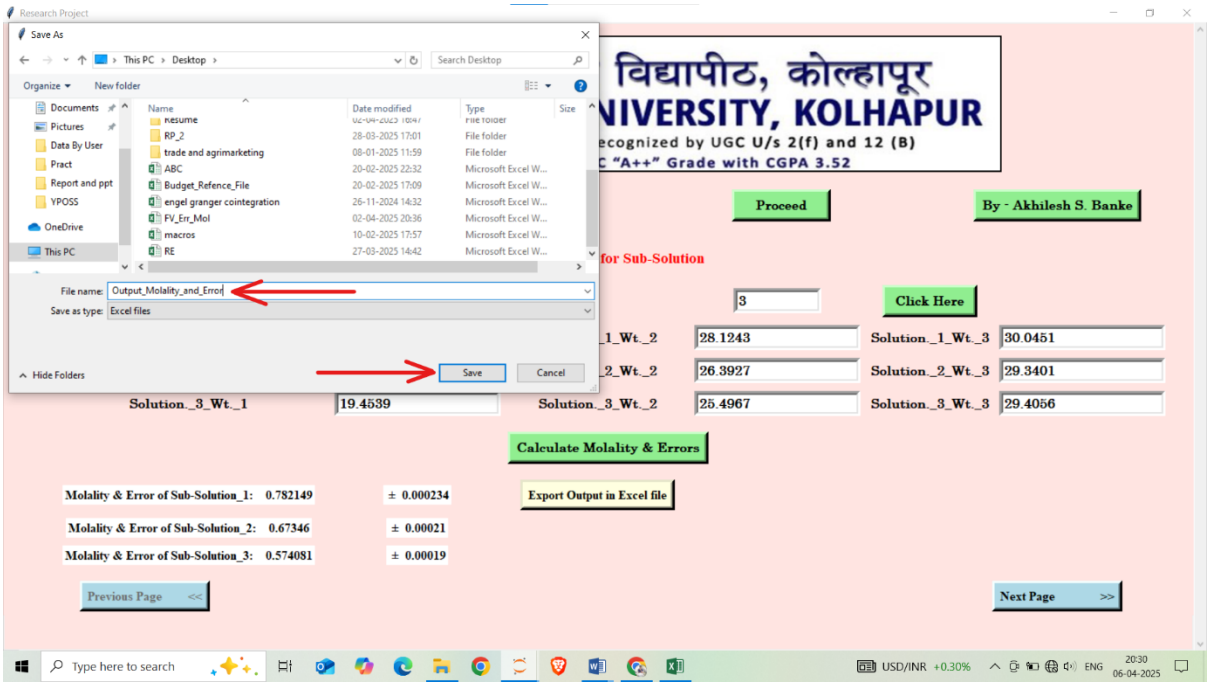


Figure 23: Saving Output to Excel File

Step 10: Export Confirmation and Proceeding Further

Once the user clicks the **"Save"** button in the dialog box, a **confirmation message** pops up stating: **"Export successful!"**

This message confirms that the molality and error data has been successfully saved to the selected Excel file at the chosen location.

- The confirmation is indicated by the **green arrow** in the figure 24.
- The user can now click **"OK"** to close the message box (shown with red arrow).
- After saving, the user can continue working by clicking the **"Next Page"** button (indicated by the **right red arrow**) to proceed with the next part of the analysis.

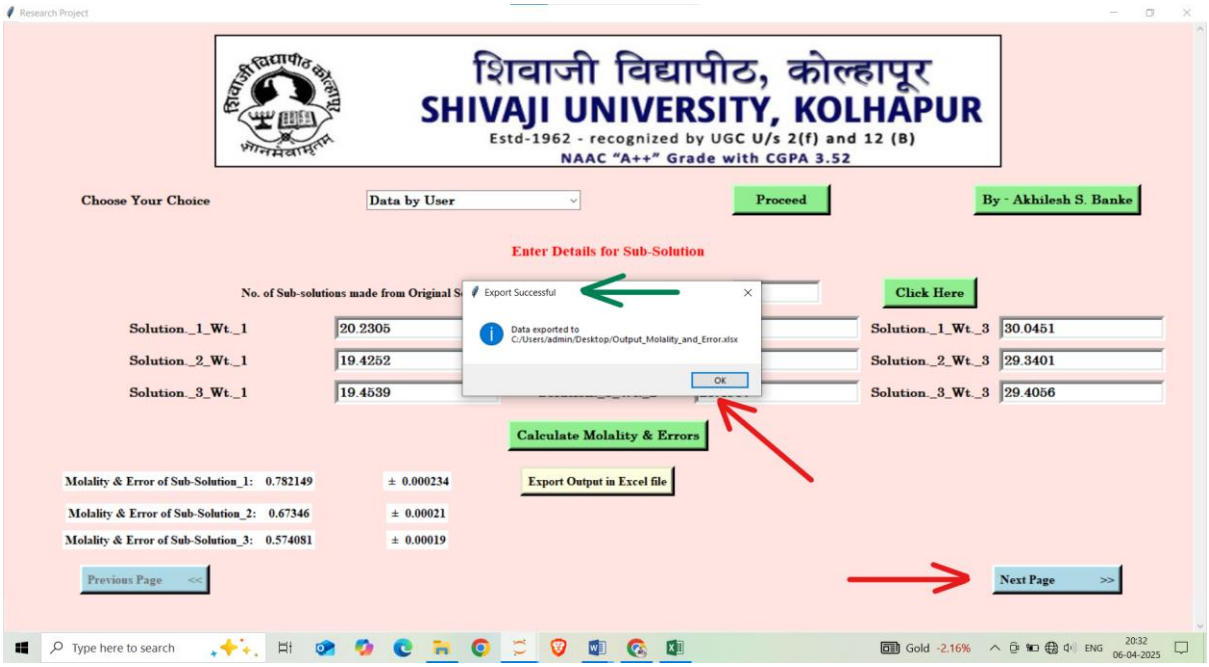
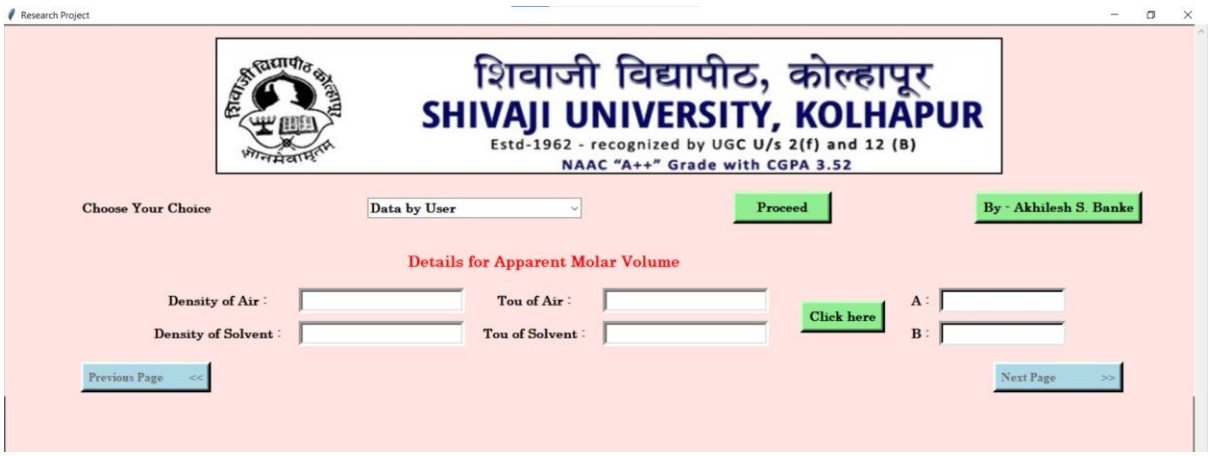


Figure 24: Export Confirmation and Proceeding Further



Step 11: Input Data for Apparent Molar Volume – Constants A and B

In this step, the user is required to enter the **basic experimental data** needed for the calculation of Apparent Molar Volume. Fields to Fill:

- **Density of Air**, **Density of Solvent**
- **Temperature of Air**, **Temperature of Solvent**

Once all fields are filled with the correct values (highlighted by purple arrows), click the **"Click here"** button (marked with a red arrow) to calculate and **generate the constants A and B**, which appear in the corresponding fields on the right.

These constants are likely determined through regression or fitting formulas used in the backend of the program.

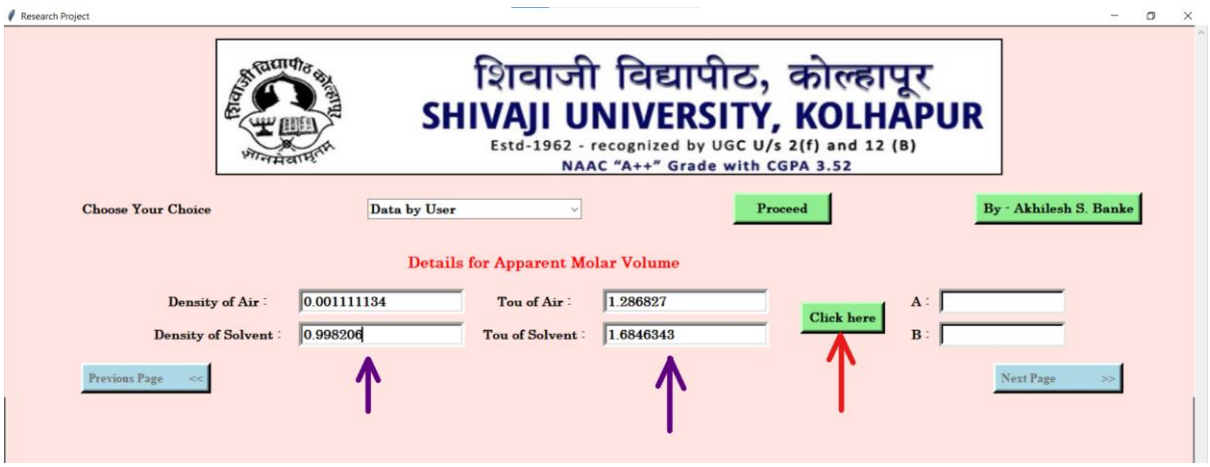


Figure 25: Input Data for Apparent Molar Volume

Step 12: Enter TOU for Solutions and Display Densities

Once constants **A** and **B** are generated:

- The user must enter the **TOU (Time of Ultrasonic)** values for each solution in the respective fields labelled as "TOU for Solution_1", "TOU for Solution_2", and "TOU for Solution_3".
- After entering these values, click the **"Show Densities"** button.
- This will calculate and display the **densities** of the entered solutions in the output box provided. See figure 26 and 27.

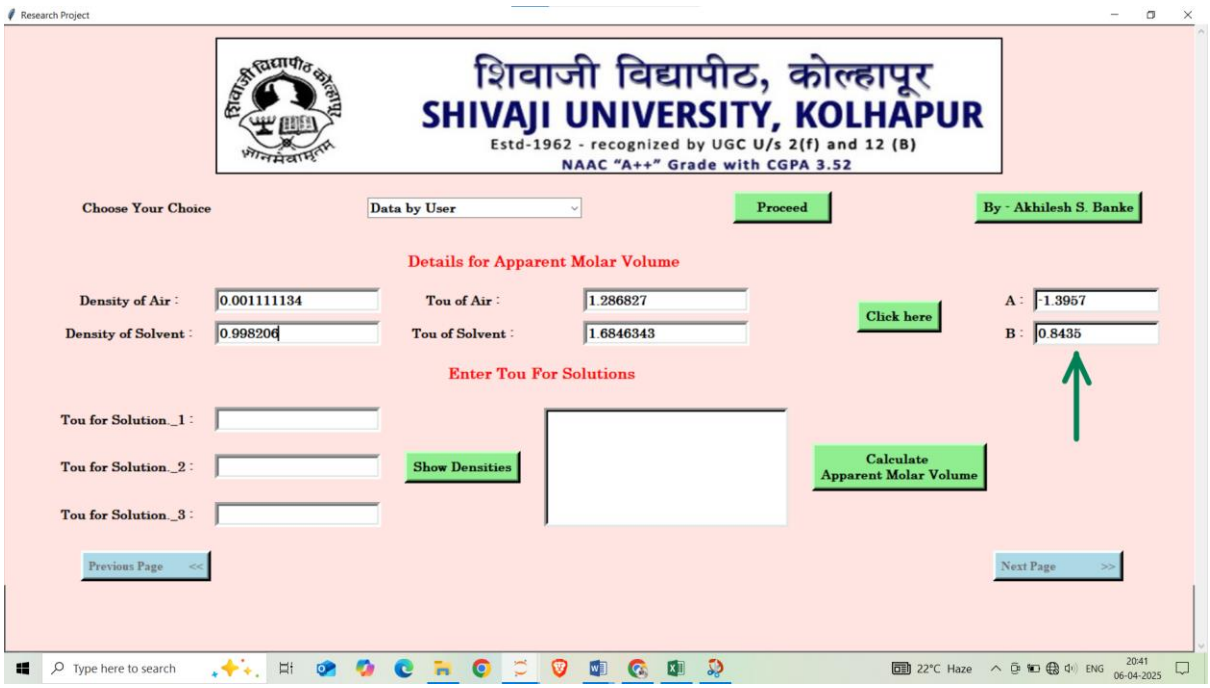


Figure 26: Constants A and B calculated.

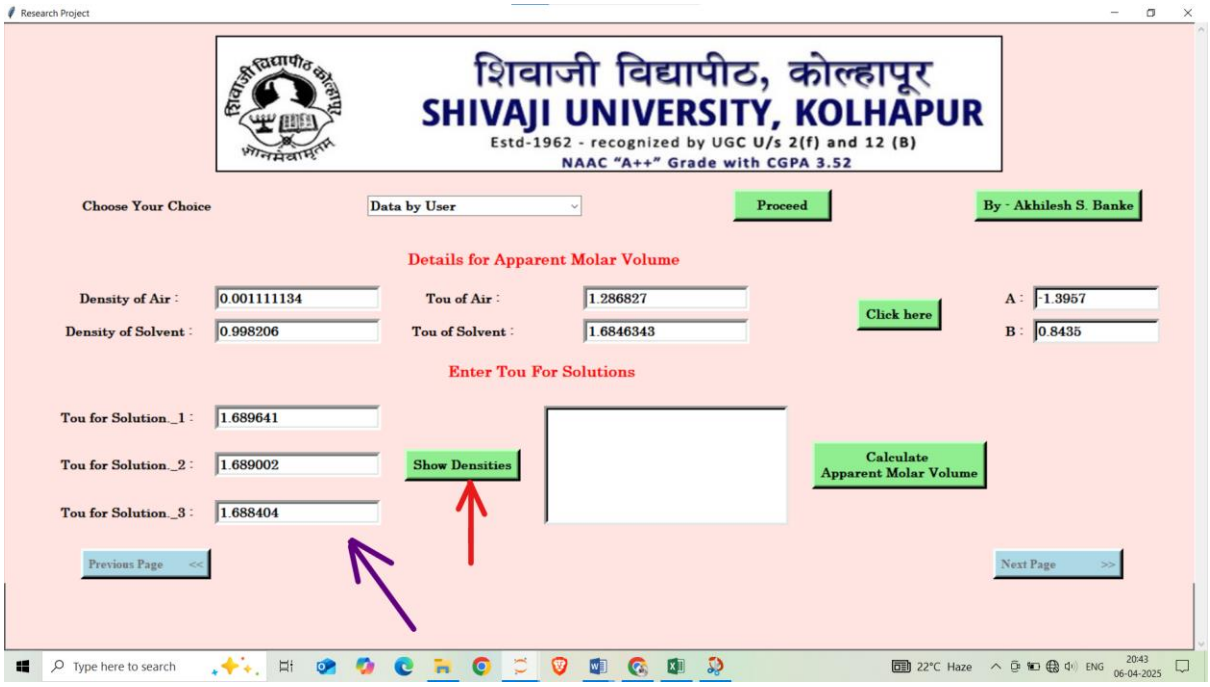


Figure 27: Enter data for density calculation.

Step 13: Display of Densities and Proceed to Apparent Molar Volume

After clicking the **"Show Densities"** button, the **densities** of all entered solutions will appear in the text box next to it. Once the densities are displayed, click on the **"Calculate Apparent Molar Volume"** button. This action will perform the necessary calculations using the input data and display the **Apparent Molar Volume** values for each solution.

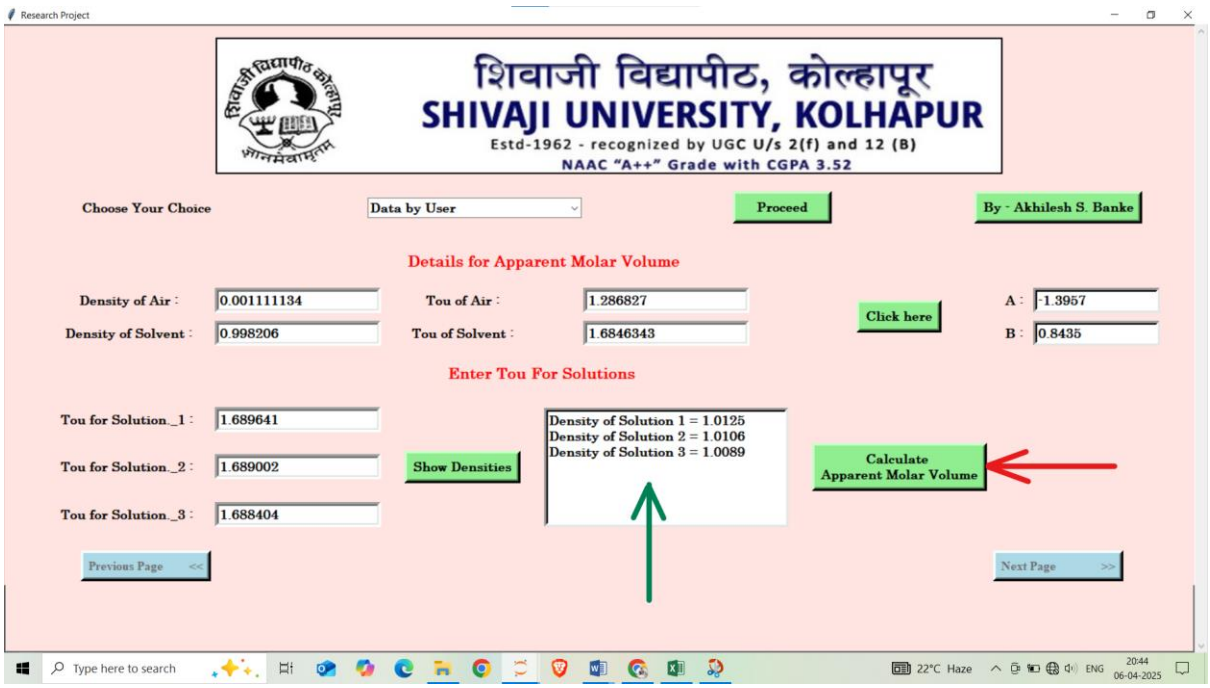


Figure 28: Display of Densities and Proceed to Apparent Molar Volume

Step 14: Apparent Molar Volume Results Displayed

After clicking **"Calculate Apparent Molar Volume"**, the values for all entered solutions are shown on the screen. Each result is displayed in the format:

Apparent Molar Volume of solution X: [value] ± [error]

These results represent the final calculated apparent molar volumes along with their corresponding **errors or uncertainties**, helping assess precision. See figure 29.

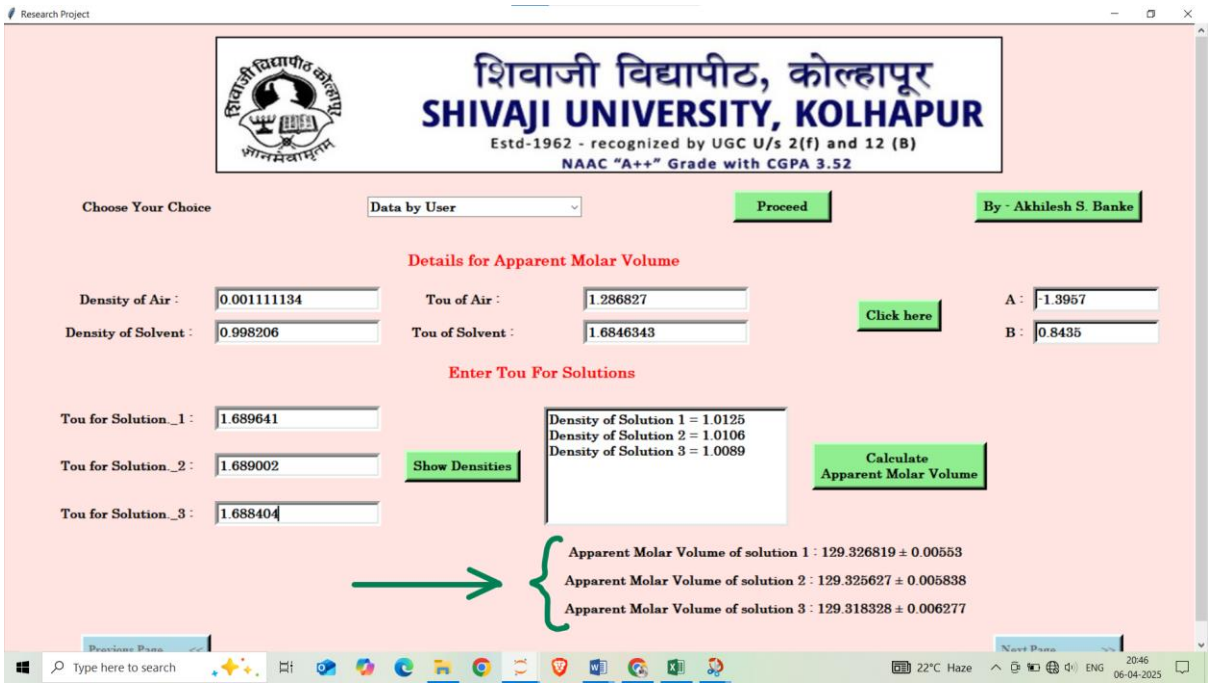


Figure 29: Apparent Molar Volume Results

6.5.2 When user chooses “Data by Excel file” option:

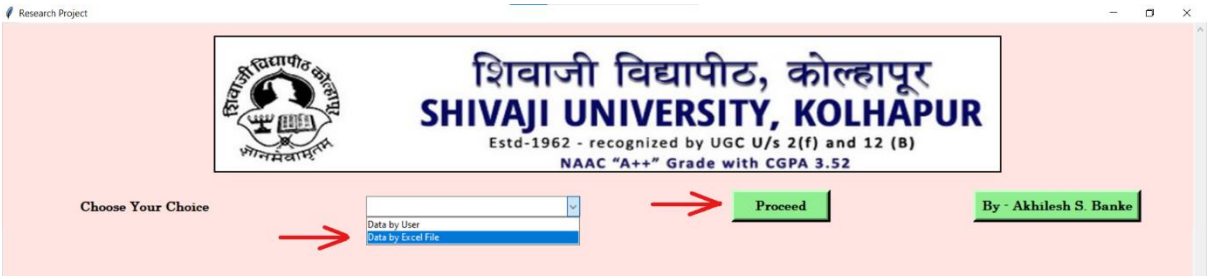


Figure 30: User Selection Panel

Step 3: Data by Excel file

After completing or opting out of the manual entry process, if the user chooses “**Data by Excel File**” from the dropdown menu and clicks “**Proceed**”, a new interface appears. This interface includes options to Import an Excel file or Download a Template to ensure proper formatting before importing. See figure 31.

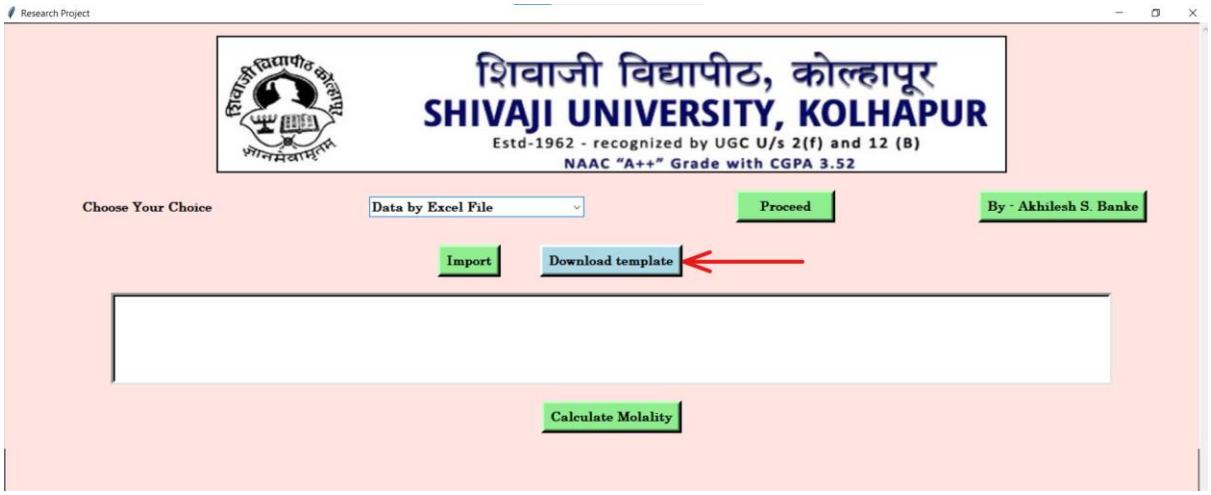


Figure 31: Interface after choice of data by excel file

Step 4: Download Excel Template

When the user selects the **"Download template"** button, a **Save As** dialog box will appear. In this window:

- A **purple arrow** indicates where the user should **enter the file name** (e.g., Template_For_Data). See figure 32.
- A **red arrow** shows the **"Save" button**, which the user must click to confirm the file name and location.

After saving, the template Excel file will be available at the chosen location. This template can then be filled with data and later imported for further calculations.

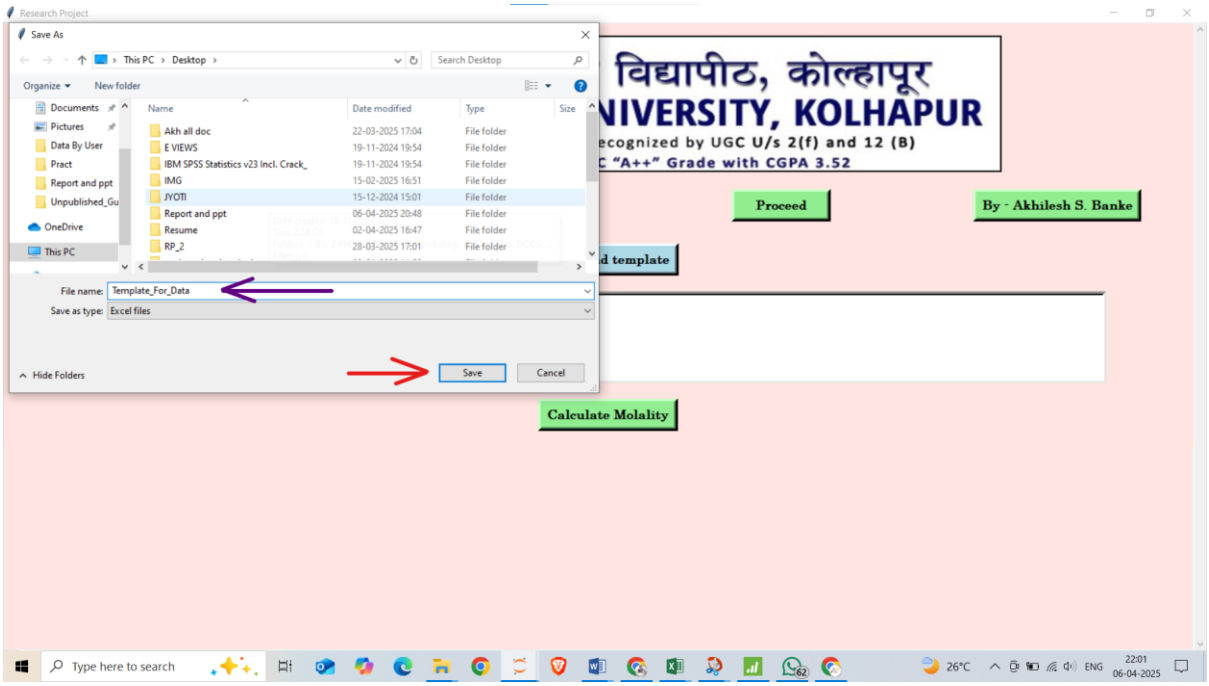


Figure 32: Download Excel Template as.

Step 5: Template Download Confirmation

Once the user clicks **"Save"**, a **success message box** will pop up confirming that the template has been downloaded successfully.

- The **green arrow** highlights the **location** where the template is saved on the device (e.g., C:\Users\Admin\Desktop\Template_For_Data.xlsx).
- The **red arrow** points to the **"OK"** button, which the user must click to close the message box and proceed.

This ensures the user knows where to find the file before importing it for analysis.

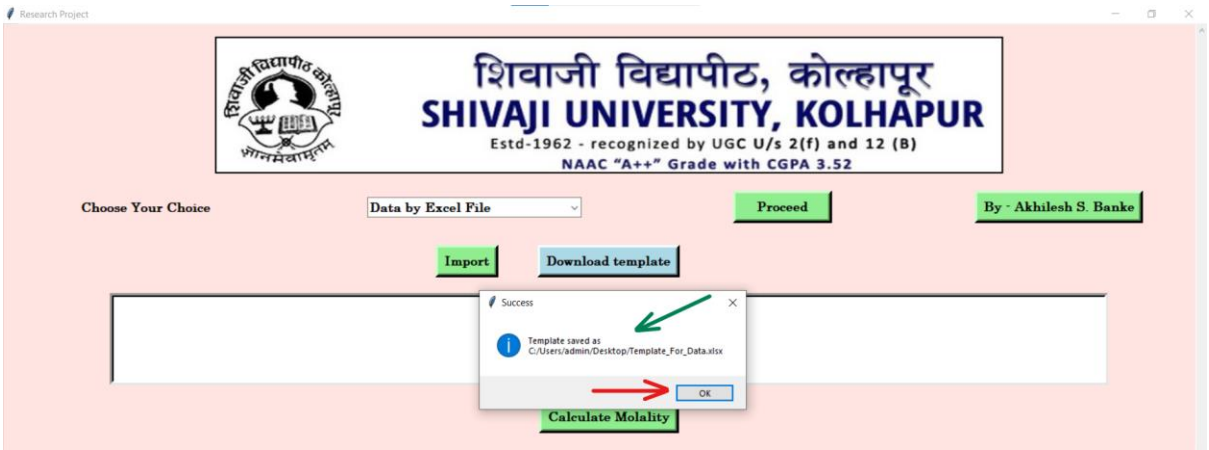
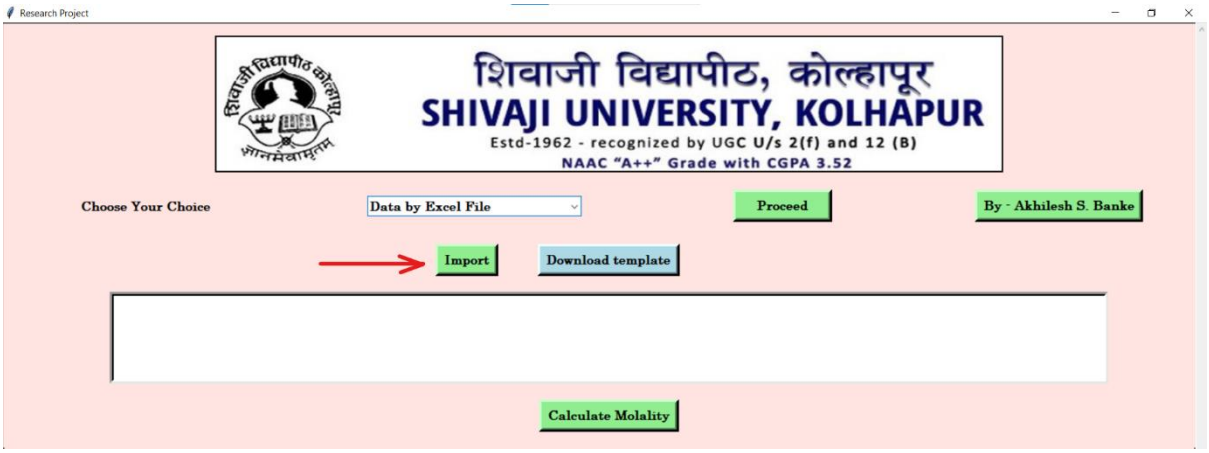


Figure 33: Template Download Confirmation



Step 6: Importing Excel Data File

Once the user has filled in the downloaded template with the required data, they should:

1. Click the **"Import"** button.
2. A dialog box will appear allowing the user to select the filled **Excel file**.
    - The **purple arrow** points to the **file name** (Template_For_Data) that the user needs to select.
    - The **red arrow** indicates the **"Open" button**, which the user must click to import the file into the application. See figure 34.

This action will load the data from the Excel sheet into the program for further analysis.

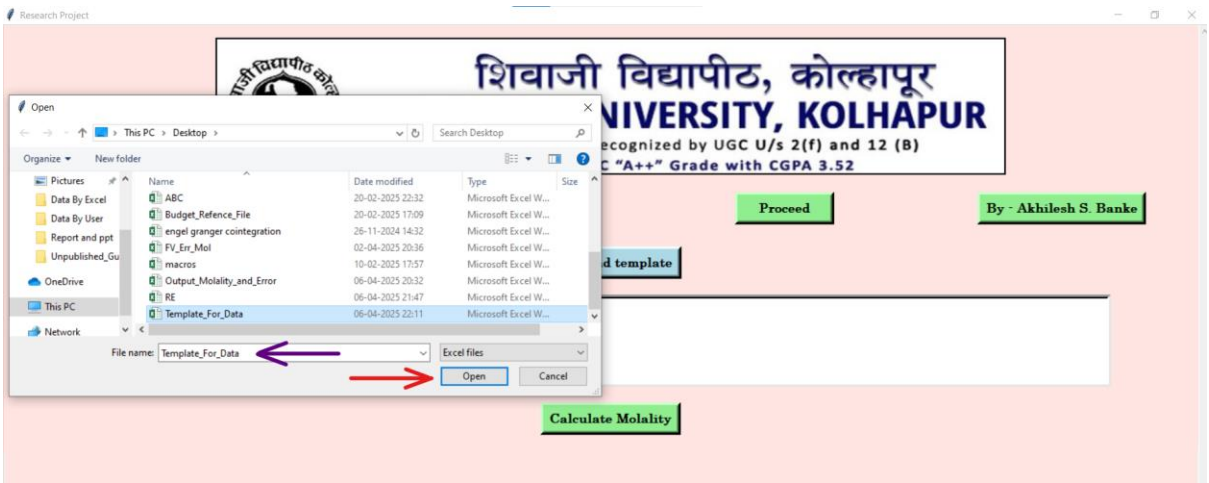


Figure 34: Importing Excel Data File

Step 7: View Imported Data and Proceed to Calculation

After importing the Excel file, a **preview of the data** will be displayed in tabular form on the interface.

- The **green arrow** highlights the **imported table preview**, showing the column headers and data structure from the Excel sheet. See figure 35.
- The user should now click on the **"Calculate Molality"** button (pointed by the **red arrow**) to begin processing the data and compute the molality values.

This step marks the transition from data input to calculation.

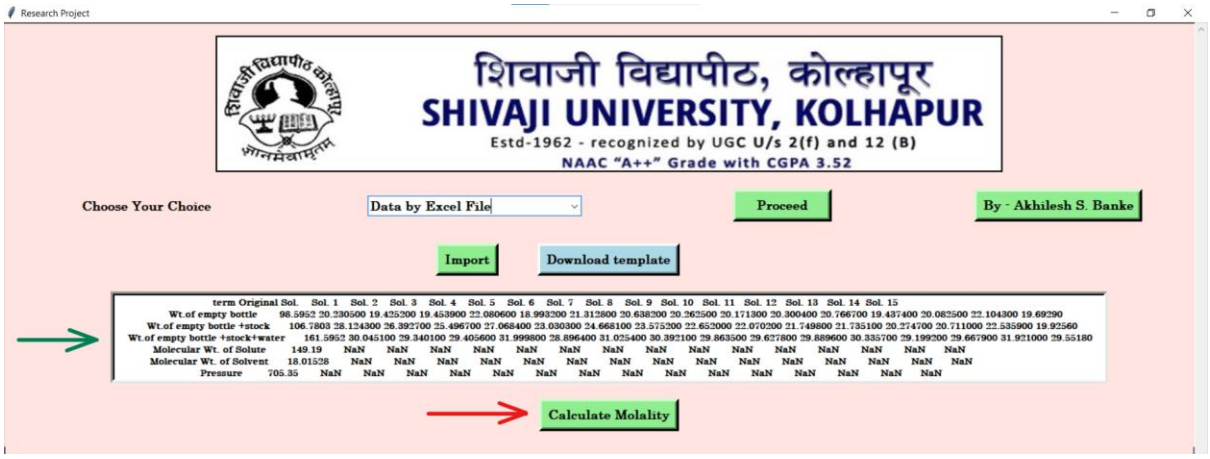


Figure 35: Preview of imported data file

Step 8: View Calculated Molality and Proceed Further

After clicking **"Calculate Molality"**, the system displays the molality values of the original solution and sub-solutions along with their respective errors.

- The **green arrow on the left** shows the **molality values and errors** of the sub-solutions.
- The **top green arrow** indicates the **molality of the original solution**.
- The **red arrow in the canter** points to the **"Export Output in Excel file"** button. Clicking this allows the user to **save the results** for further reference.
- To continue with the analysis, the user should click on the **"Proceed for Apparent Molar Volume"** button (pointed by the **red arrow**).

This step completes the molality calculation phase and moves toward apparent molar volume evaluation.

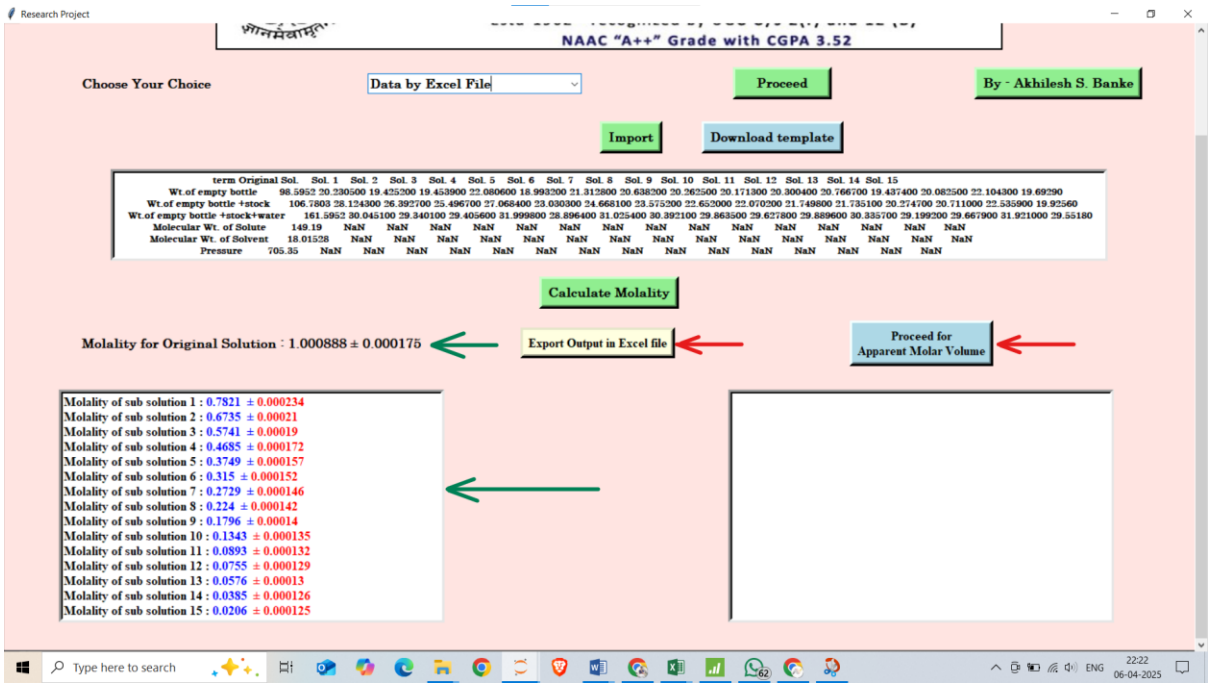


Figure 36: Output of GUI

Step 9: Apparent Molar Volume Calculation

After clicking the **"Proceed for Apparent Molar Volume"** button, the apparent molar volumes of all sub-solutions are displayed along with their associated errors.

- The **green arrow** highlights the calculated **Apparent Molar Volume values** and their corresponding **errors**. See figure 37.
- These values are generated using backend formulae based on previously calculated molalities.

This allows the user to observe how the apparent molar volume varies across different sub-solutions, which is useful for further analysis.

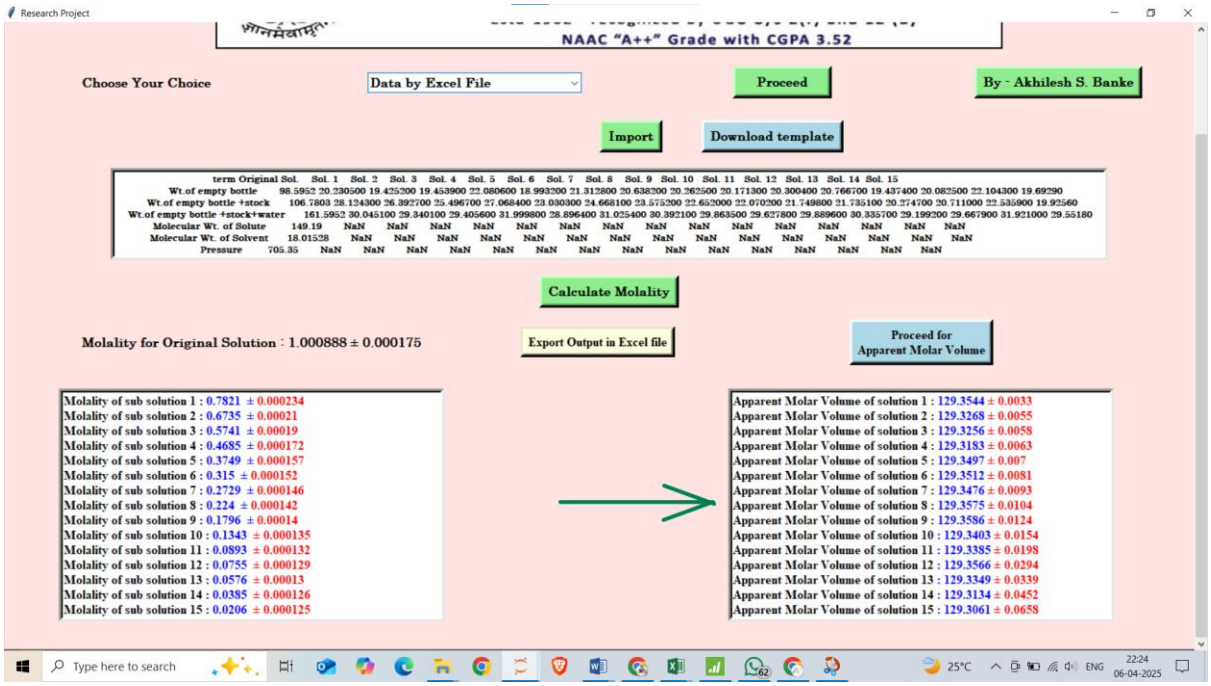


Figure 37: Output of GUI for apparent molar volume

End Note

This GUI-based application developed under Shivaji University, Kolhapur, provides a streamlined and user-friendly platform for calculating **Molality** and **Apparent Molar Volume** from either manually entered data or an Excel template. The step-by-step flow ensures smooth navigation—from data input, whether by user or Excel file, to advanced computations using built-in formulae. The system guides the user through selecting the input mode, downloading a template, importing correctly formatted data, calculating molality with errors, exporting results, and computing apparent molar volumes accurately. Interactive prompts, dialog boxes, and labelled arrows throughout the interface enhance the user experience, reducing the risk of error and ensuring clarity at each stage. The visual indicators like success messages and color-coded results support quick interpretation and validation of scientific data. This GUI stands out as a practical and efficient tool for academic and research purposes, particularly in physical chemistry and solution analysis. The well-integrated backend computations and easy export features make it suitable for both classroom teaching and advanced laboratory work.

7. Conclusion

- The final refined model, developed using **reciprocal transformation** and **Cochrane-Orcutt estimation**, has shown excellent accuracy and reliability in predicting errors associated with molality calculations.

- The resulting model:

$$\frac{1}{Error\_Molality} = 10414.59 - 4350.2\ Molality - 640.573\ Molality^2$$

- All the assumptions of core regression—normality, homoscedasticity, and independence of residuals—are met, so the model is statistically valid.

- Validation on external data (JCED 2018) proved that the model is capable of predicting errors both for diluted and for double-diluted solutions, revealing its high generalizability and potential for real-world application.

- A Graphical User Interface (GUI) was developed to allow chemists to easily input experimental data values and instantly obtain molality and its error as well as apparent molar volume and its error values—without any manual computation.

- The model minimizes the necessity of complex backend mathematical computation (such as error propagation formulae) and eliminates the likelihood of human error in experimental data processing.

- Furthermore, by providing accurate estimates through modelling, the approach can reduce the number of experimental runs, saving time, resources, and effort in the lab.

- In general, this advanced model is statistically sound, experimentally confirmed, computationally fast, and easy to use, and thus a valuable tool in physical chemistry for error modelling and concentration-based analysis.

8. Limitations

- **Volume Constraint:** The model is specifically developed based on experimental data from solutions prepared with a fixed volume of 10 mL. As a result, its predictive accuracy is optimized for this volume. If the solution volume is significantly increased or decreased, the error pattern in molality may not remain consistent, potentially affecting the model's validity.
- **Container Dependency:** The type of beaker used during solution preparation and measurement can introduce additional variability in the error. This study was performed using a specific type of container, and error behaviour can differ when using polyethylene beakers versus glass beakers due to factors like static charge, adhesion, and residue retention. These material-based differences were not explicitly modelled, which may influence results in varying lab conditions.

9. References

- Douglas A. Skoog, Donald M. West & F. James Holler. *Fundamentals of Analytical Chemistry*. HARCOURT ASIA PTE. LTD
- Montgomery, D. C., Peck, E. A., & Vining, G. G. (2021). *Introduction to linear regression analysis.* 5th Ed. John Wiley & Sons.
- Musale, S. P., Patil, K. R., Gavhane, R. J., & Dagade, D. H. (2018). Density and Speed-of-Sound Measurements for Dilute Binary Mixtures of Diethylammonium-Based Protic Ionic Liquids with Water. *Journal of Chemical & Engineering Data*, 63(6), 1859–1876. https://doi.org/10.1021/acs.jced.7b00909
- Dagade, D. H., Shinde, S. P., Madkar, K. R., & Barge, S. S. (2014). Density and sound speed study of hydration of 1-butyl-3-methylimidazolium based amino acid ionic liquids in aqueous solutions. *The Journal of Chemical Thermodynamics*, 79, 192–204. https://doi.org/10.1016/j.jct.2014.07.026
- Lundh, F. (1999). *An Introduction to Tkinter*. PythonWare. http://www.pythonware.com/library/tkinter/introduction/

.

Appendix

In [ ]:

```
import tkinter as tk
from tkinter import ttk, messagebox, filedialog
from PIL import Image, ImageTk
import openpyxl
from openpyxl import Workbook
import numpy as np
import pandas as pd
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure

# Initialize the main window
root = tk.Tk()
root.title("Research Project")
root.geometry("1500x950")
root.configure(bg="MistyRose")

#_____________________________________________________________
def on_frame_configure(canvas):
    """Reset the scroll region to encompass the inner frame"""
    canvas.configure(scrollregion=canvas.bbox("all"))

def on_mouse_wheel(event, canvas):
    canvas.yview_scroll(int(-1*(event.delta/120)), "units")

# Create a canvas object and a vertical scrollbar for scrolling it
canvas = tk.Canvas(root, bg="MistyRose")
scrollbar = tk.Scrollbar(root, orient="vertical", command=canvas.yview)
canvas.configure(yscrollcommand=scrollbar.set)

# Pack the scrollbar and canvas
scrollbar.grid(row=0, column=1, sticky="ns")
canvas.grid(row=0, column=0, sticky="nsew")

# Configure grid to allow resizing
root.grid_rowconfigure(0, weight=1)
root.grid_columnconfigure(0, weight=1)

# Create a frame inside the canvas which will contain the widgets
scrollable_frame = tk.Frame(canvas, bg="MistyRose")
scrollable_frame.bind("<Configure>", lambda event: on_frame_configure(canvas))

# Create a window in the canvas to contain the inner frame
canvas.create_window((0, 0), window=scrollable_frame, anchor="nw")

# Add mouse wheel scrolling support
canvas.bind_all("<MouseWheel>", lambda event: on_mouse_wheel(event, canvas))
#_____________________________________________________________

# Open and resize the image
image_path = "C:/Users/Admin/Desktop/RP_2/GUI/Uni_logo.jpg"
img = Image.open(image_path)
resized_img = img.resize((1000, 170), Image.LANCZOS)

# Create a PhotoImage object and store it as an attribute of the root window
root.new_img = ImageTk.PhotoImage(resized_img)

# Create a Label widget to display the image
dis = tk.Label(scrollable_frame, image=root.new_img, bg="black")
dis.grid(row=0, column=0, columnspan=4, padx=270, pady=15)

a = tk.Label(scrollable_frame, text = "Choose Your Choice",  font = ("Century", 12, 'bold'), bg = "MistyRose")
a.grid(row =1 , column = 0, padx = 10, pady =15)

x = tk.StringVar()

b = ttk.Combobox(scrollable_frame, textvariable = x, values =["Data by User", "Data by Excel File"], width = 25,
font = ("Century", 12, 'bold'))
b.grid(row =1 , column = 1, padx = 15, pady =15)

results = []

def proceed():
    option = x.get()

    # Forget grids only if they exist
    if hasattr(proceed, 'frame'):
        proceed.frame.grid_forget()

    if option == "Data by Excel File":
        proceed.frame = tk.Frame(scrollable_frame, bg = "mistyrose")
        proceed.frame.grid(row=2, column=0, columnspan=6, padx=10, pady=10)
```

```
        def import_excel():

            global chem
            file_path = filedialog.askopenfilename(filetypes=[("Excel files", "*.xlsx;*.xls")])

            if file_path:
                try:
                    chem = pd.read_excel(file_path)
                    data_text.delete(1.0, tk.END)
                    data_text.insert(tk.END, chem.to_string(index=False))
                except Exception as e:
                    messagebox.showerror("Error", f"Failed to import Excel file:\n{str(e)}")

        Button_1 = tk.Button(proceed.frame, text = "Import", command = import_excel, borderwidth=5, font=("Centur
y", 12, 'bold'), bg = "Lightgreen")
        Button_1.grid(row=2, column=0, columnspan = 3, padx=50, pady=10, sticky='e')

        data_text = tk.Text(proceed.frame, borderwidth=5, height=7, width=180, bg = 'white', font=("Century", 8,
"bold"))
        data_text.grid(row=3, column=0, columnspan = 6, padx=100, pady=10)

        def Download_template():

            # Create a workbook and add a worksheet
            wb = Workbook()
            ws = wb.active
            ws.title = "Template"

            # Add some headers and sample data
            headers = ["Term", "Original Solution", "Sub Sol. 1", "Sub Sol. 2", "Sub Sol. 3", "Sub Sol. 4", "Sub
Sol. 5.... so on"]
            sample_data = [
                ["Wt.of empty bottle", "0", "0", "0", "0", "0", "0"],
                ["Wt.of empty bottle +stock", "0", "0", "0", "0", "0", "0"],
                ["Wt.of empty bottle +stock+water", "0", "0", "0", "0", "0", "0"],
                ["Molecular Wt. of Solute", "0"],
                ["Molecular Wt. of Solvent", "0"],
                ["Pressure", "0"],
                ["Humidity", "0"],
                ["Temperature", "0"],
                ["Density of Air", "0"],
                ["Density of Water", "0"],
                ["Tou of Air", "0"],
                ["Tou of Water", "0"],
                ["Instrumental Error for wt.", "0"],
                ["Instrumental Error for Density", "0"],
                ["Instrumental Error for Pressure", "0"],
                ["Instrumental Error for Temperature", "0"],
                ["% of Water in Solute", "0"],
                ["Tou for Solution", "0", "0", "0", "0", "0", "0"]
            ]

            # Write headers
            for col_num, header in enumerate(headers, 1):
                ws.cell(row=1, column=col_num, value=header)

            # Write sample data
            for row_num, row_data in enumerate(sample_data, 2):
                for col_num, cell_value in enumerate(row_data, 1):
                    ws.cell(row=row_num, column=col_num, value=cell_value)

            # Save the workbook to a file
            file_path = filedialog.asksaveasfilename(defaultextension=".xlsx", filetypes=[("Excel files", "*.xlsx
")])
            if file_path:
                wb.save(file_path)
                messagebox.showinfo("Success", f"Template saved as {file_path}")
            else:
                messagebox.showwarning("Save Canceled", "Template not saved. Please try again.")


        template_button = tk.Button(proceed.frame, text = "Download template", command = Download_template, font=
("Century", 12, 'bold'), bg = "Lightblue", borderwidth = 5)
        template_button.grid(row = 2, column = 3, pady=10, sticky='w')


        def molality():
            global molality, er_molality, D_Sol, Mol_Wt_Solt, Density_Water, D_IE

            Mol_Wt_Solt = chem.iloc[3,1]
            Mol_Wt_Solv = chem.iloc[4,1]
            IE = chem.iloc[12,1]
            IE_2 = np.sqrt(pow(IE,2) + pow(IE,2))
            D_IE = chem.iloc[13,1]
            Density_Water = chem.iloc[9,1]
            Water = chem.iloc[16,1]

            E = chem.iloc[0,1]
            E_Solt = chem.iloc[1,1]
            E_Solv = chem.iloc[2,1]
```

```
            solute_with_water = E_Solt - E
            Wt_of_water_in_solute = solute_with_water * Water / 100

            Wt_Solt = solute_with_water - Wt_of_water_in_solute
            Wt_Solv = E_Solv - E_Solt + Wt_of_water_in_solute
            Wt_Soln = Wt_Solt + Wt_Solv

            er_Wt_Solt = er_Wt_Solv = np.sqrt(2 * pow(IE,2) )
            er_Wt_Soln = er_Wt_Solt + er_Wt_Solv

            Molality = (Wt_Solt*1000)/(Wt_Solv*Mol_Wt_Solt)
            er_Molality = Molality*np.sqrt(pow((er_Wt_Solt/Wt_Solt),2) + pow((er_Wt_Solv/Wt_Solv),2))

            Label_1 = tk.Label(proceed.frame, text = f"Molality for Original Solution : {round(Molality, 6)} ± {r
ound(er_Molality, 6)}", font=("Century", 12, 'bold'), bg ="MistyRose")
            Label_1.grid(row = 5, column = 0, pady = 10)

            e = np.empty(chem.shape[1]-2)
            e_Solv = np.empty(chem.shape[1]-2)
            e_Solt = np.empty(chem.shape[1]-2)
            wt_Solt = np.empty(chem.shape[1]-2)
            wt_Solv = np.empty(chem.shape[1]-2)
            wt_Soln = np.empty(chem.shape[1]-2)
            act_wt_Solt = np.empty(chem.shape[1]-2)
            act_wt_Solv = np.empty(chem.shape[1]-2)

            er_wt_Solt = np.empty(chem.shape[1]-1)
            er_wt_Solv = np.empty(chem.shape[1]-1)
            molality = np.empty(chem.shape[1]-1)
            er_molality = np.empty(chem.shape[1]-1)
            D_Sol = np.empty(chem.shape[1]-1)
            Fv = np.empty(chem.shape[1]-1)
            Er_Fv = np.empty(chem.shape[1]-1)

            er_wt_Solt[0]=er_Wt_Solt
            er_wt_Solv[0]=er_Wt_Solv
            molality[0]=Molality
            er_molality[0]=er_Molality

            for i in range(2,chem.shape[1]):

                e[i-2] = chem.iloc[0,i]
                e_Solt[i-2] = chem.iloc[1,i]
                e_Solv[i-2] = chem.iloc[2,i]

            for i in range(chem.shape[1]-2):
                wt_Solt[i] = e_Solt[i]-e[i]
                wt_Solv[i] = e_Solv[i]-e_Solt[i]
                wt_Soln[i] = wt_Solt[i] + wt_Solv[i]
                act_wt_Solt[i] = (wt_Solt[i])*(Wt_Solt/Wt_Soln)
                act_wt_Solv[i] = wt_Soln[i]-act_wt_Solt[i]
                molality[i+1] = (act_wt_Solt[i]*1000)/(act_wt_Solv[i]*Mol_Wt_Solt)

            for i in range(chem.shape[1]-2):
                er_wt_Solt[i+1] = act_wt_Solt[i]*np.sqrt(pow((IE_2/wt_Solt[i]),2) + pow((IE_2/Wt_Solt),2) + pow((
IE_2/Wt_Soln),2))
                er_wt_Solv[i+1] = np.sqrt(2*(IE**2) + pow(er_wt_Solt[i+1],2))
                er_molality[i+1] = molality[i+1]*np.sqrt(pow((er_wt_Solt[i+1]/act_wt_Solt[i]),2) + pow((er_wt_Sol
v[i+1]/act_wt_Solv[i]),2))

            text_mol = tk.Text(proceed.frame, height = 15, width = 60, font=("times", 12, 'bold'), borderwidth =
5, bg = 'White')
            text_mol.grid(row = 6, column = 0, rowspan = 8, padx = 20, pady=25)

            for i in range(chem.shape[1]-2):

                text_mol.insert(tk.END, f"Molality of sub solution {i+1} : {round(molality[i+1], 4)}\t ± {round(e
r_molality[i+1], 6)}\n")
                start_index = f"{i+1}.29"
                end_index = f"{i+1}.39"
                text_mol.tag_add("blue_word", start_index, end_index)
                text_mol.tag_config("blue_word", foreground="blue")

                start_index = f"{i+1}.38"
                end_index = f"{i+1}.49"
                text_mol.tag_add("red_word", start_index, end_index)
                text_mol.tag_config("red_word", foreground="red")


            def Export_excel():

                df = pd.DataFrame({"Molality" : molality, "Error in Molality": er_molality})
                file_path = filedialog.asksaveasfilename(defaultextension=".xlsx", filetypes=[("Excel files", "*.
xlsx"), ("All files", "*.*")])

                if file_path:
                    df.to_excel(file_path, index=False)
                    tk.messagebox.showinfo("Export Successful", f"Data exported to {file_path}")

            Button_Export_Excel = tk.Button(proceed.frame, text = "Export Output in Excel file", command = Export
_excel, font=("times", 12, 'bold'), borderwidth = 5, bg = 'LightYellow')
            Button_Export_Excel.grid(row = 5, column = 2, padx = 20, pady=5)
```

```
            def App_Molar_Volume():
                global A_1 ,B_1, Mol_Wt_Solt, molality, D_Sol, er_molality, D_IE, Density_Water

                B_1 = (chem.iloc[9,1] - chem.iloc[8,1]) / (pow(chem.iloc[11,1],2) - pow(chem.iloc[10,1],2))
                A_1 = chem.iloc[9,1]-( B_1 * pow(chem.iloc[11,1],2))

                text_data.delete(1.0, tk.END)

                for i in range(chem.shape[1]-2):
                    D_Sol[i] = A_1 + (B_1 * pow(chem.iloc[17,i+1],2))
                    Fv[i] = (Mol_Wt_Solt/D_Sol[i]) + (1000*(chem.iloc[9,1]-D_Sol[i])) / (molality[i] * chem.iloc[
9,1] * D_Sol[i])

                    Er_Fv[i] = np.sqrt( pow( ((1000 * (Density_Water - D_Sol[i]) / (molality[i] * Density_Water *
D_Sol[i])) * np.sqrt( 2 * pow(D_IE/D_Sol[i], 2) + pow(er_molality[i] / molality[i], 2) ) ), 2) + pow(D_IE / D_Sol
[i], 2))

                    text_data.insert(tk.END, f"Apparent Molar Volume of solution {i+1} : {round(Fv[i],4)} ± {roun
d(Er_Fv[i],4)}\n")
                    start_index = f"{i+1}.37"
                    end_index = f"{i+1}.49"
                    text_data.tag_add("blue_word", start_index, end_index)
                    text_data.tag_config("blue_word", foreground="blue")

                    start_index = f"{i+1}.47"
                    end_index = f"{i+1}.58"
                    text_data.tag_add("red_word", start_index, end_index)
                    text_data.tag_config("red_word", foreground="red")

            text_data = tk.Text(proceed.frame, height = 15, width = 60, font=("times", 12, 'bold'), borderwidth =
5, bg = 'White')
            text_data.grid(row = 6, column = 3, rowspan = 8, padx = 20, pady=25)


            B_Molar_Volume = tk.Button(proceed.frame, command = App_Molar_Volume, text = "Proceed for\nApparent M
olar Volume", font=("times", 12, 'bold'), borderwidth = 5, bg = 'Lightblue')
            B_Molar_Volume.grid(row = 5, column = 3, padx = 20, pady=5)

        Button_2 = tk.Button(proceed.frame, text = "Calculate Molality",command = molality, borderwidth = 5, font
=("Century", 12, 'bold'), bg = "Lightgreen" )
        Button_2.grid(row=4, column=0, columnspan = 6, pady = 10)


    elif option == "Data by User":

        proceed.frame = tk.Frame(scrollable_frame, bg = 'MistyRose', padx = 220)
        proceed.frame.grid(row=2, column=0, columnspan=4, padx=10, pady=10)

        label = tk.Label(proceed.frame, text="Enter Details for Original Solution", font=("century", 14, "bold"),
bg="MistyRose", fg="red")
        label.grid(row=2, columnspan=4, pady=10)

        # Create input labels and entry fields
        label1 = tk.Label(proceed.frame, text="Wt. of Empty Bottle", font=("Century", 12, 'bold'), bg="MistyRose"
)
        label1.grid(row=3, column=0, padx=25, pady=10)

        e_1 = tk.Entry(proceed.frame, font=("Century", 12, 'bold'), borderwidth=5)
        e_1.grid(row=4, column=0, padx=15)

        label2 = tk.Label(proceed.frame, text="Wt. of Empty Bottle + Solute", font=("Century", 12, 'bold'), bg="M
istyRose")
        label2.grid(row=3, column=1, padx=25, pady=10)

        e_2 = tk.Entry(proceed.frame, font=("Century", 12, 'bold'), borderwidth=5)
        e_2.grid(row=4, column=1, padx=15)

        label3 = tk.Label(proceed.frame, text="Wt. of Empty Bottle + Solute + Solvent", font=("Century", 12, 'bol
d'), bg="MistyRose")
        label3.grid(row=3, column=2, padx=25, pady=10)

        e_3 = tk.Entry(proceed.frame, font=("Century", 12, 'bold'), borderwidth=5)
        e_3.grid(row=4, column=2, padx=15)

        label4 = tk.Label(proceed.frame, text="% of Water in Solute", font=("Century", 12, 'bold'), bg="MistyRose
")
        label4.grid(row=3, column=3, padx=25, pady=10)

        Water = tk.Entry(proceed.frame, font=("Century", 12, 'bold'), borderwidth=5)
        Water.grid(row=4, column=3, padx=15)

        def wt():
            Wt_1 = e_1.get()
            Wt_2 = e_2.get()
            Wt_3 = e_3.get()
            Wt_4 = Water.get()

            solute_with_water = float(Wt_2) - float(Wt_1)
            Wt_of_water_in_solute = (solute_with_water * float(Wt_4)) / 100
```

```
            global Wt_solute, Wt_solvent, Wt_solution
            Wt_solute = solute_with_water - Wt_of_water_in_solute
            Wt_solvent = float(Wt_3) - float(Wt_2) + Wt_of_water_in_solute
            Wt_solution = Wt_solute + Wt_solvent

            text_wd.delete(1.0, tk.END)
            text_wd2.delete(1.0, tk.END)
            text_wd3.delete(1.0, tk.END)

            text_wd.insert(tk.END, round(Wt_solute, 4))
            text_wd2.insert(tk.END, round(Wt_solvent, 4))
            text_wd3.insert(tk.END, round(Wt_solution, 4))

        # Create Calculate button
        button = tk.Button(proceed.frame, text="Calculate", font=("times", 12, "bold"), borderwidth=5, command=wt
, width=10, height=1, bg="lightgreen")
        button.grid(row=5, column=0, columnspan=4, padx=20, pady=10)

        # Create labels and text widgets for displaying results
        L1 = tk.Label(proceed.frame, text="Weight Of Solute", font=('times', 12, 'bold'), bg="MistyRose")
        L1.grid(row=6, column=0, pady=5)

        text_wd = tk.Text(proceed.frame, width=20, height=1, borderwidth=5, font=('Century', 12, 'bold'))
        text_wd.grid(row=6, column=1, pady=5)

        L2 = tk.Label(proceed.frame, text="Weight Of Solvent", font=('times', 12, 'bold'), bg="MistyRose")
        L2.grid(row=7, column=0, pady=5)

        text_wd2 = tk.Text(proceed.frame, width=20, height=1, borderwidth=5, font=('Century', 12, 'bold'))
        text_wd2.grid(row=7, column=1, pady=5)

        L3 = tk.Label(proceed.frame, text="Weight Of Solution", font=('times', 12, 'bold'), bg="MistyRose")
        L3.grid(row=8, column=0, pady=5)

        text_wd3 = tk.Text(proceed.frame, width=20, height=1, borderwidth=5, font=('Century', 12, 'bold'))
        text_wd3.grid(row=8, column=1, pady=5)

        L4 = tk.Label(proceed.frame, text=" Molecular Wt. of Solute", font=('times', 12, 'bold'), bg="MistyRose")
        L4.grid(row=9, column=0, pady=5)

        e_4 = tk.Entry(proceed.frame, font=("Century", 12, "bold"), width=20, borderwidth=5)
        e_4.grid(row=9, column=1)

        L5 = tk.Label(proceed.frame, text=" Molecular Wt. of Solvent", font=('times', 12, 'bold'), bg="MistyRose"
)
        L5.grid(row=6, column=2, pady=5)

        e_5 = tk.Entry(proceed.frame, font=("Century", 12, "bold"), width=16, borderwidth=5)
        e_5.grid(row=6, column=3, pady=5)

        L6 = tk.Label(proceed.frame, text="Instrumental Error in Wt.", font=('times', 12, 'bold'), bg="MistyRose"
)
        L6.grid(row=7, column=2, pady=5)

        e_6 = tk.Entry(proceed.frame, font=("Century", 12, "bold"), width=16, borderwidth=5)
        e_6.grid(row=7, column=3, pady=5)

        L7 = tk.Label(proceed.frame, text="Instrumental Error in Density", font=('times', 12, 'bold'), bg="MistyR
ose")
        L7.grid(row=8, column=2, pady=5)

        e_7 = tk.Entry(proceed.frame, font=("Century", 12, "bold"), width=16, borderwidth=5)
        e_7.grid(row=8, column=3, pady=5)

        # Define the molality calculation function
        def Molality():
            global Wt_solute, Wt_solvent, Wt_solution, Mol_Wt_Solute, IE, D_IE
            Mol_Wt_Solute = float(e_4.get())
            # Formula for Molality
            Molality = (Wt_solute * 1000) / (Mol_Wt_Solute * Wt_solvent)

            text_wd4.delete(1.0, tk.END)
            text_wd4.insert(tk.END, round(Molality, 6))

            IE = float(e_6.get())
            D_IE = float(e_7.get())

            global Er_Wt_solute, Er_Wt_solvent, Er_Wt_solution
            # By method of propagation of errors
            Er_Wt_solute = Er_Wt_solvent = np.sqrt(2* pow(IE,2))
            Er_Wt_solution = Er_Wt_solute + Er_Wt_solvent

            Er_Molality = Molality * np.sqrt(pow(Er_Wt_solute/Wt_solute, 2) + pow(Er_Wt_solvent/Wt_solvent, 2))

            text_wd5.delete(1.0, tk.END)
            text_wd5.insert(tk.END, round(Er_Molality, 6))

        # Create Calculate Molality button
        button_2 = tk.Button(proceed.frame, text="Calculate Molality", font=("times", 12, "bold"), borderwidth=5,
command=Molality, width=20, height=1, bg="lightgreen")
        button_2.grid(row=10, column=0, columnspan=4, padx=20, pady=10)
```

```
        # Create text widget to display molality
        L7 = tk.Label(proceed.frame, text="Molality", font=('times', 12, 'bold'), bg="MistyRose")
        L7.grid(row=11, column=0, pady=15)

        text_wd4 = tk.Text(proceed.frame, width=20, height=1, borderwidth=5, font=('times', 12, 'bold'))
        text_wd4.grid(row=11, column=1, pady=15)

        L8 = tk.Label(proceed.frame, text="Error in Molality", font=('times', 12, 'bold'), bg="MistyRose")
        L8.grid(row=11, column=2, pady=15)

        text_wd5 = tk.Text(proceed.frame, width=20, height=1, borderwidth=5, font=('times', 12, 'bold'))
        text_wd5.grid(row=11, column=3, pady=15)

        def show_widgets():

            button_back.grid()
            button_nxt.grid()

        def hide_widgets(page_number):

            global button_back,button_nxt
            if hasattr(proceed, 'frame'):
                proceed.frame.grid_forget()

            proceed.frame = tk.Frame(scrollable_frame, bg = 'MistyRose')
            proceed.frame.grid(row=2, column=0, columnspan=6, padx=10, pady=10)

            if page_number == 1:
                global L_1, L_2, entry_1, sub_entries
                sub_entries = []

                L_1 = tk.Label(proceed.frame, text="Enter Details for Sub-Solution", font=("times", 14, "bold"),
bg="MistyRose", fg="red")
                L_1.grid(row=2, columnspan=6, padx=50, pady=15)

                L_2 = tk.Label(proceed.frame, text="No. of Sub-solutions made from Original Solution", font=("tim
es", 12, "bold"), bg="MistyRose")
                L_2.grid(row=3, columnspan=3, padx=20, pady=5)

                entry_1 = tk.Entry(proceed.frame, borderwidth=5, font=("Century", 12, "bold"), width = 10)
                entry_1.grid(row=3, column=3, padx=20, pady=5)


                def sub_solution():

                    global rows

                    try:
                        rows = int(entry_1.get())
                        cols = 3
                        if rows <= 0:
                            raise ValueError("Number of Sub Solution must be positive integers.")

                        for i in range(rows):
                            sub_entries.append([])
                            for j in range(cols):
                                Sub_Label = tk.Label(proceed.frame, text=f"Solution._{i+1}_Wt._{j+1}", bg='MistyR
ose', font=("century", 12, 'bold'))
                                Sub_Label.grid(row=i+4, column=2*j, padx=5, pady=5)

                                Sub_Entry = tk.Entry(proceed.frame, borderwidth=5, font=("century", 12, 'bold'))
                                Sub_Entry.grid(row=i+4, column=2*j+1, padx=5, pady=5)
                                sub_entries[-1].append(Sub_Entry)

                        Sub_Button = tk.Button(proceed.frame, text="Calculate Molality & Errors", borderwidth=5,
font=("century", 12, 'bold'), bg='Lightgreen', command=calculate_sub_solutions)
                        Sub_Button.grid(row=rows+4, columnspan=6, padx=50, pady=15)
                    except ValueError as e:
                        messagebox.showerror("Invalid Input", str(e))

                B_1 = tk.Button(proceed.frame, text=" Click Here ", borderwidth=5, font=("century", 12, 'bold'),
command=sub_solution, bg='Lightgreen')
                B_1.grid(row=3, column=4, padx=20, pady=5)

                button_nxt = tk.Button(scrollable_frame, text="Next Page \t>>", font=("times", 12, 'bold'), comma
nd= lambda : hide_widgets(page_number+1), borderwidth=5, bg="lightblue")
                button_nxt.grid(row=12, column=3, pady=5)


            if page_number == 2:

                Label_1 = tk.Label(proceed.frame, text = " Details for Apparent Molar Volume", bg = "MistyRose",
fg = "red", font=("century", 14, 'bold'))
                Label_1.grid(row = 2, column = 0, columnspan = 6, pady = 15)

                Label_2 = tk.Label(proceed.frame, text = "Density of Air :", font=("century", 12, 'bold'), bg = "
MistyRose")
                Label_2.grid(row = 3, column = 0, padx = 5, pady = 5)

                E_2 = tk.Entry(proceed.frame, font=("century", 12, 'bold'), borderwidth = 5)
                E_2.grid(row = 3, column = 1, pady = 5)
```

```
                Label_3 = tk.Label(proceed.frame, text = "Density of Solvent :", font=("century", 12, 'bold'), bg
= "MistyRose")
                Label_3.grid(row = 4, column = 0, padx = 5, pady = 5)

                E_3 = tk.Entry(proceed.frame, font=("century", 12, 'bold'), borderwidth = 5)
                E_3.grid(row = 4, column = 1, padx = 15, pady = 5)

                Label_4 = tk.Label(proceed.frame, text = "Tou of Air :", font=("century", 12, 'bold'), bg = "Mist
yRose")
                Label_4.grid(row = 3, column = 2, padx = 5, pady = 5)

                E_4 = tk.Entry(proceed.frame, font=("century", 12, 'bold'), borderwidth = 5)
                E_4.grid(row = 3, column = 3, padx = 15, pady = 5)

                Label_5 = tk.Label(proceed.frame, text = "Tou of Solvent :", font=("century", 12, 'bold'), bg = "
MistyRose")
                Label_5.grid(row = 4, column = 2, padx = 5, pady = 5)

                E_5 = tk.Entry(proceed.frame, font=("century", 12, 'bold'), borderwidth = 5)
                E_5.grid(row = 4, column = 3, padx = 5, pady = 5)

                def Constants_Tou():
                    global Density_Water, Mol_Wt_Solute

                    Density_Air = float(E_2.get())
                    Density_Water = float(E_3.get())
                    Tou_Air = float(E_4.get())
                    Tou_Water = float(E_5.get())

                    global A,B
                    B = (Density_Water - Density_Air)/(pow(Tou_Water,2) - pow(Tou_Air,2))
                    A = Density_Water - (B * pow(Tou_Water,2))

                    Text_1.delete(1.0,tk.END)
                    Text_1.insert(tk.END, round(A,4))

                    Text_2.delete(1.0,tk.END)
                    Text_2.insert(tk.END, round(B,4))

                    L = tk.Label(proceed.frame, text = "Enter Tou For Solutions", font=("century", 14, 'bold'), b
g = 'Mistyrose', fg = 'red')
                    L.grid(row = 5, column = 0, columnspan = 6, pady = 15)

                    global rows, sub_entries_tou
                    sub_entries_tou = []

                    for i in range(rows):
                            Sub_Label_tou = tk.Label(proceed.frame, text = f"Tou for Solution._{i+1} : ", bg='Mis
tyRose', font=("century", 12, 'bold'))
                            Sub_Label_tou.grid(row = i+6, column = 0, padx = 5, pady = 5)

                            Sub_Entry_tou = tk.Entry(proceed.frame, borderwidth=5, font=("century", 12, 'bold'))
                            Sub_Entry_tou.grid(row = i+6, column = 1, padx = 5, pady = 5)
                            sub_entries_tou.append(Sub_Entry_tou)

                    def Density():

                        global sub_entries_tou, A, B, densities, rows
                        densities = []

                        T_1.delete(1.0, tk.END)

                        try:
                            for entry in sub_entries_tou:
                                Tou = float(entry.get())
                                D = A + B * pow(Tou, 2)
                                densities.append(D)

                                T_1.insert(tk.END, f"Density of Solution {sub_entries_tou.index(entry) + 1} = {ro
und(D, 4)}\n")
                        except ValueError:
                                messagebox.showerror("Invalid Input", "Please enter valid numeric values for τ.")


                    B_1 = tk.Button(proceed.frame, text = "Show Densities", command = Density, font=("century", 1
2, 'bold'), borderwidth = 5, bg = "lightgreen")
                    B_1.grid(row = 6, column = 2, rowspan = rows, pady = 15, padx = 15 )

                    T_1 = tk.Text(proceed.frame, height = 7, width = 30, font = ("century", 12, 'bold'), borderwi
dth = 5, bg = "White")
                    T_1.grid(row = 6, column = 3, rowspan = rows, pady = 15, padx = 15)


                    def Apparent_Molar_Volume():
                        global App_Mol_Volume,Mol_Wt_Solute, Er_App_Mol_Volume, densities, rows, Mol_Wt_Solute, D
ensity_Water, Molalities, Er_Molalities, D_IE


                        for i in range(rows):
                            App_Mol_Volume = (Mol_Wt_Solute/densities[i]) + (1000 * (Density_Water - densities[i]
) / (Molalities[i] * Density_Water * densities[i]) )
```

```
                            Er_App_Mol_Volume = np.sqrt( pow( ((1000 * (Density_Water - densities[i]) / (Molaliti
es[i] * Density_Water * densities[i])) * np.sqrt( 2 * pow(D_IE/densities[i], 2) + pow(Er_Molalities[i] / Molaliti
es[i], 2) ) ), 2) + pow(D_IE / densities[i], 2))

                            L = tk.Label(proceed.frame, text = f"Apparent Molar Volume of solution {i+1} : {round
(App_Mol_Volume, 6)} ± {round(Er_App_Mol_Volume, 6)}", font=("century", 12, 'bold'), bg = "MistyRose")
                            L.grid(row  = i+rows+7, column  = 3, columnspan = 2, pady = 5, padx = 15)

                    B_2 = tk.Button(proceed.frame, text = "Calculate \nApparent Molar Volume", command = Apparent
_Molar_Volume, font=("century", 12, 'bold'), borderwidth = 5, bg = "lightgreen")
                    B_2.grid(row = 6, column = 4, rowspan = rows, pady = 15, padx = 15)

                B_1 = tk.Button(proceed.frame, text = "Click here", command = Constants_Tou, font=("century", 12,
'bold'), borderwidth = 5, bg = "lightgreen")
                B_1.grid(row = 3, column = 4, rowspan = 2, padx = 25, pady = 5)

                Label_6 = tk.Label(proceed.frame, text = "A :", font=("century", 12, 'bold'), bg = "MistyRose")
                Label_6.grid(row = 3, column = 5, padx = 5, pady = 5)

                Text_1 = tk.Text(proceed.frame, bg = "White", height = 1, width = 15, borderwidth = 5, font=("cen
tury", 12, 'bold'))
                Text_1.grid(row = 3, column = 6, padx = 5, pady = 5)

                Label_7 = tk.Label(proceed.frame, text = "B :", font=("century", 12, 'bold'), bg = "MistyRose")
                Label_7.grid(row = 4, column = 5, padx = 5, pady = 5)

                Text_2 = tk.Text(proceed.frame, bg = "White", height = 1, width = 15, borderwidth = 5, font=("cen
tury", 12, 'bold'))
                Text_2.grid(row = 4, column = 6, padx = 5, pady = 5)

                button_nxt.config(state = tk.DISABLED)

        # Create navigation buttons
        button_back = tk.Button(scrollable_frame, text="Previous Page \t<<", font=("times", 12, 'bold'), command=
show_widgets, borderwidth=5, bg="lightblue")
        button_back.grid(row=12, column=0, pady=5)
        button_back.config(state = tk.DISABLED)

        button_nxt = tk.Button(scrollable_frame, text="Next Page \t>>", font=("times", 12, 'bold'), command= lamb
da : hide_widgets(1), borderwidth=5, bg="lightblue")
        button_nxt.grid(row=12, column=3, pady=5)

        def calculate_sub_solutions():
            global Mol_Wt_Solute, Er_Wt_solute, Er_Wt_solvent,Wt_solute, Wt_solvent, Wt_solution, IE, results, Mo
lalities, Er_Molalities

            Mol_Wt_Solute = float(e_4.get())
            IE = float(e_6.get())
            IE_2 = np.sqrt( pow(IE,2) + pow(IE,2))

            Molalities = []
            Er_Molalities = []

            for entries in sub_entries:
                wt_1 = float(entries[0].get())
                wt_2 = float(entries[1].get())
                wt_3 = float(entries[2].get())

                wt_solute = (wt_2 - wt_1) * (Wt_solute/Wt_solution)
                wt_solvent = wt_3 - wt_1 - wt_solute
                wt_solution = wt_solute + wt_solvent

                global molality, Er_molality, results
                molality = (wt_solute * 1000) / (Mol_Wt_Solute * wt_solvent)

                Molalities.append(molality)

                #For Subsolutions :
                Er_wt_solute = wt_solute * np.sqrt( pow(IE_2/(wt_2 - wt_1), 2) + pow(IE_2/Wt_solute, 2) + pow(IE_
2/Wt_solution, 2))
                Er_wt_solvent = np.sqrt(2*IE*IE + pow(Er_wt_solute, 2) )
                Er_wt_solution = Er_wt_solute + Er_wt_solvent

                Er_molality = molality * np.sqrt( pow(Er_wt_solute/wt_solute, 2) + pow(Er_wt_solvent/wt_solvent,
2) )

                Er_Molalities.append(Er_molality)

                results.append((molality, Er_molality))

            for i, (molality, Er_molality) in enumerate(results):

                text_1 = tk.Label(proceed.frame, text=f"Molality & Error of Sub-Solution_{i+1}:  \t{round(molalit
y, 6)}", font=('times', 12, 'bold'), bg="white")
                text_1.grid(row=len(sub_entries)+5+i, column=0, padx = 20, pady=5)

                text_2 = tk.Label(proceed.frame, text=f" ±  {round(Er_molality, 6)}", font=('times', 12, 'bold'),
bg="white")
                text_2.grid(row=len(sub_entries)+5+i, column=1, padx = 20, pady=5)

            def Export_excel():
                global molality, Er_molality, results
```

```
                df = pd.DataFrame(results, columns=["Molality", "Error in Molality"])
                file_path = filedialog.asksaveasfilename(defaultextension=".xlsx", filetypes=[("Excel files", "*.
xlsx"), ("All files", "*.*")])

                if file_path:
                    df.to_excel(file_path, index=False)
                    tk.messagebox.showinfo("Export Successful", f"Data exported to {file_path}")

            Button_Export = tk.Button(proceed.frame, text = "Export Output in Excel file", command = Export_excel
, font=("times", 12, 'bold'), borderwidth = 5, bg = 'LightYellow')
            Button_Export.grid(row = len(sub_entries)+5, column = 2, padx = 20, pady=5)

    else:
        messagebox.showerror("Input Error", "Select Valid Chooice !")

button_proceed = tk.Button(scrollable_frame, text="Proceed", width=10, font=("Century", 12, 'bold'), command=proc
eed, bg = "lightgreen", borderwidth = 5)
button_proceed.grid(row=1, column=2, padx=5)

Creator = tk.Button(scrollable_frame, text="By - Akhilesh S. Banke", width=18, font=("Century", 12, 'bold'), bg =
"lightgreen",borderwidth = 5)
Creator.grid(row=1, column=3, padx=5)

root.mainloop()
```