# Design

# Data Modelling

## Flow chart

## Variables

## Algorithms

### Operational

When a teacher wishes to add a new class, it must be simple and easy for them to do so. To achieve this, I plan on implementing a binary search for teachers to syphon through a list of all their students as quickly as possible.

#### Implementation for Binary Search

public int BinarySearchRecursive(List<string> inputList, string key, int min, int max)

{

if (min > max) { return -1; }

else

{

int length = key.Length;

int mid = (int)Math.Ceiling((double)(min + max) / 2);

string item = inputList[mid].ToString().Substring(0, length);

if (key == item) { return mid; }

else if (string.Compare(key, item) > 0)

{ return BinarySearchRecursive(inputList, key, mid + 1, max, length); }

else { return BinarySearchRecursive(inputList, key, min, mid - 1, length); }

}

}

This function takes the list that was passed as an argument, the object that needs to be found and the two indexes of the beginning and ending of a ‘sub list’ (min/max respectively).

inputList -> A list containing all the names of students linked to the current teacher.

Key -> The current entry in the search textbox.

Note: The search is called on the ‘text changed’ event. Hence, the key can be a substring of any of the names in the list.

Min -> A marker for the index of the inputList which is the beginning of the new ‘sub list’

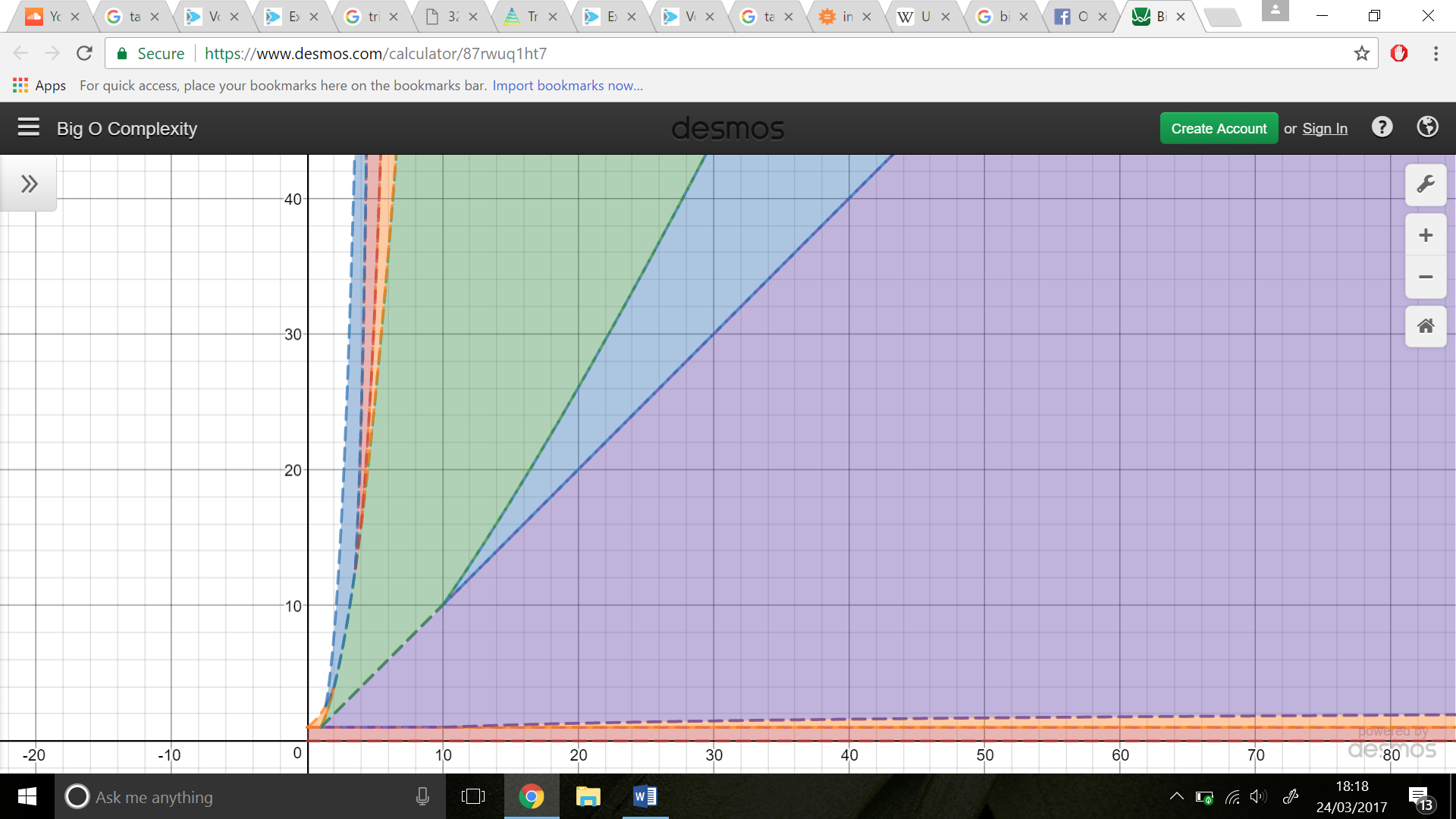
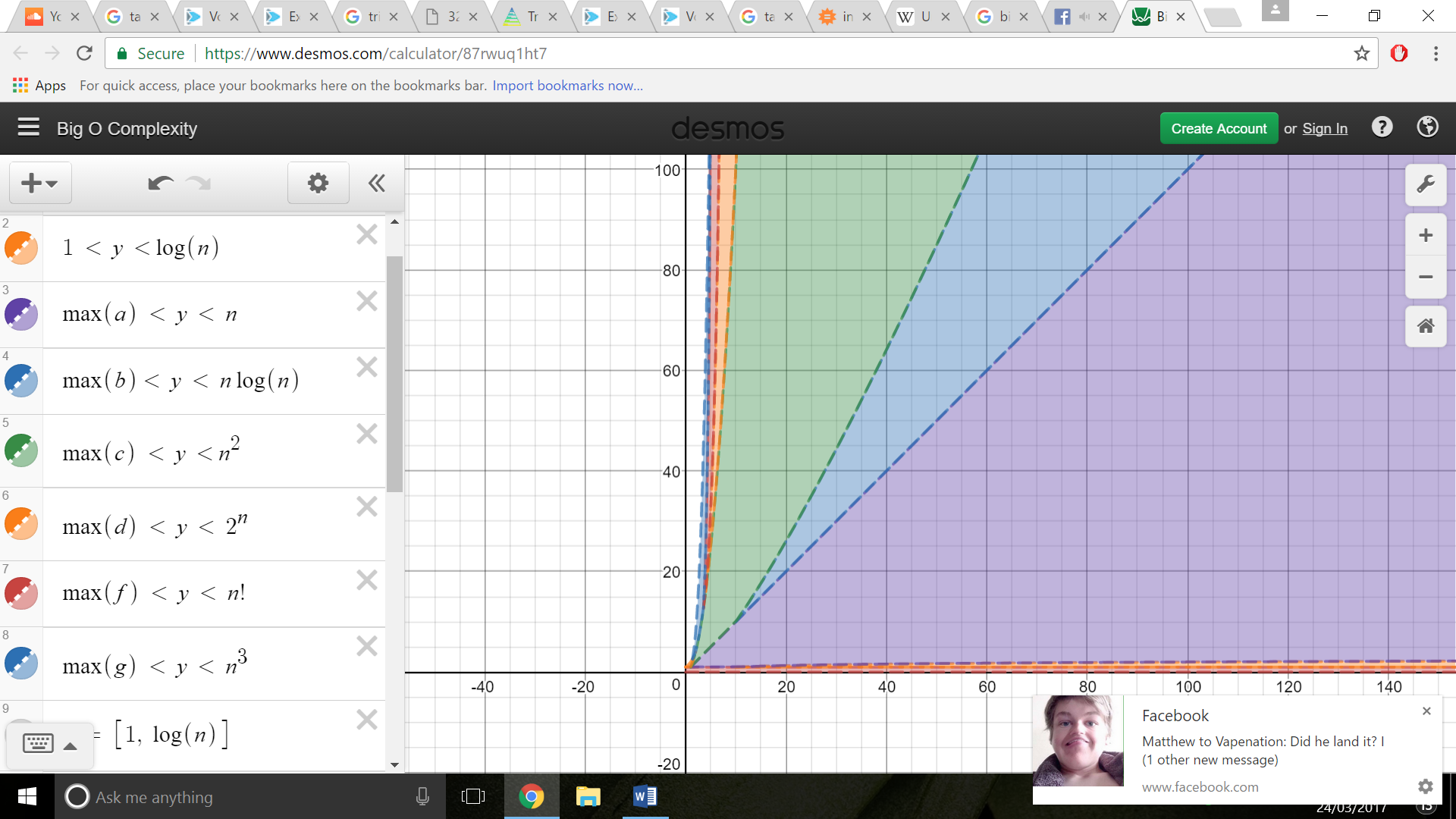
Max -> A marker for the index of the inputList which is the end of the new ‘sub list’

Note: This is not an actual sub list, but rather a section of the inputList which is currently being searched as the rest of the list has been ruled out.

Length -> The length of the input

The function will continue to call itself until the minimum index value is more than the maximum index value, meaning the item has not been found. If this is false, the middle of the list/sub list is found. Using the ‘mid’ variable, a substring of the middle of the current list/sub list is checked against the key. The length of the substring is equal to the length of the key, using the ‘length’ variable. If they are equal, then this index is returned and the item has been found. If not, the ‘String.Compare’ method is run on the key and this item. This returns a value greater than zero if the key precedes the substring (item) alphabetically, or a value less than zero if it follows the substring alphabetically. As such, if the value is greater than zero, then the index of the item to the right of the current item in the list becomes the minimum value. Thus, the new ‘sub list’ is all elements following the current item. If the value is less than zero, then the index of the item to the left of the current item becomes the maximum value. Thus, the new ‘sub list’ is all elements preceding the current item.

Complexity:

The binary search algorithm has a complexity of O(logn). Here is a comparison of the O(logn) in comparison to other orders of complexity (created via desmos.com):

From this graph, it’s easy to that the Binary Search algorithm has a much greater efficiency than other search algorithms, such as Linear Search (which has an order of complexity of O(n)). This is important as each teacher may have several classes, each with many students. Hence, an efficient algorithm for searching for each student is required as to ensure there is a smooth user interface for the teacher when creating/editing classes, as the names will need to be displayed using the DataGridView control.

### Stationary Waves

Stationary waves form on a wire when two waves of identical frequencies and opposing directions interfere with each other. In this experiment, a wire is attached between a signal generator and a fixed position. The student then alters the frequency of the wire until one single loop is formed on the wire, this is the 1st harmonic. From this, the student must carry out two separate investigations – the effect of altering the tension in the wire, and the effect of altering the length of the wire used.

Investigation 1

After finding the first harmonic, the frequency is to be recorded in a table, along with the length of wire used (control variable), and the current tension in the wire. Weights are then incrementally added to the wire to increase the tension. Each time a weight is added, the student must alter the frequency again to find 1st harmonic. The recordings are then repeated.

A graph of the frequency squared against the tension in the wire can then be plotted, which should be a straight-line graph.

Investigation 2

In this investigation, the student will be altering the length of wire used to discover the effects it has on the fundamental frequency. After finding the 1st harmonic, the student will need to incrementally move the second bridge away from the frequency generator. After doing so each time, the student must again find the 1st harmonic of the stationary waves, thus finding the fundamental frequency. This is recorded, along with the length of the wire also.

A graph of inverse frequency against length can then be plotted, which again should be a straight-line graph.

#### Code for the Stationary Waves experiment

private void pnlWave\_Paint(object sender, PaintEventArgs e)

{

Graphics g = e.Graphics;

e.Graphics.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias;

int harmonic = cbHarmonics.SelectedIndex;

int coefficient = getCoefficient(harmonic, 715);

g.DrawCurve(Pens.White, getPoints(coefficient, 50));

g.DrawCurve(Pens.White, getPoints(coefficient + 2, 50));

}

This first subroutine is the event handler for the Paint Event of the ‘pnlWave’ panel. The ‘harmonic’ variable is fetched from the ‘cbHarmonics’ combo box and is equal to the current selected index. The coefficient is then fetched from the ‘getCoefficient’ function. The coordinate pairs are then fetched and used to draw the curves using the Graphics.DrawCurve method.

* Harmonic -> The selected harmonic the student wished to view. Can be any integer greater than zero.
* Coefficient -> The coefficient used to transform the Sine graph according to the selected harmonic. This is part of the arithmetic sequence Nth term = h \* (N-1)th term) / (h + 1). Where h is one less than the harmonic, is an integer and is equal to the selected index.

private int getCoefficient(double h, int value)

{

if (h == 0) { if (value % 2 == 0) { value += 1; } return value; }

value = (int)((h \* value) / (h + 1));

return getCoefficient(h - 1, value);

}

This function returns the coefficient needed to transform the sine graph according to the selected index. If the selected index (h) is equal to zero and odd, then the value for the coefficient is returned. If not, then the value is worked out using the arithmetic sequence Nth term = h \* (N-1)th term) / (h + 1). This recursive process continues until the value of h reaches zero.

Use of even numbers:

private PointF[] getPoints(int coX, int coY)

{

double xStep = 3.0 / 12;

double[] xValues = new double[13];

double[] yValues = new double[13];

for (int i = 0; i < 13; i++)

{

xValues[i] = coX \* (i \* xStep);

yValues[i] = (coY \* (Math.Sin(xValues[i] \* 2 \* Math.PI))) + 100;

}

PointF[] pointArray = xValues.Zip(yValues, (xCoord, yCoord) => (new

PointF((float)xCoord, (float)yCoord))).ToArray();

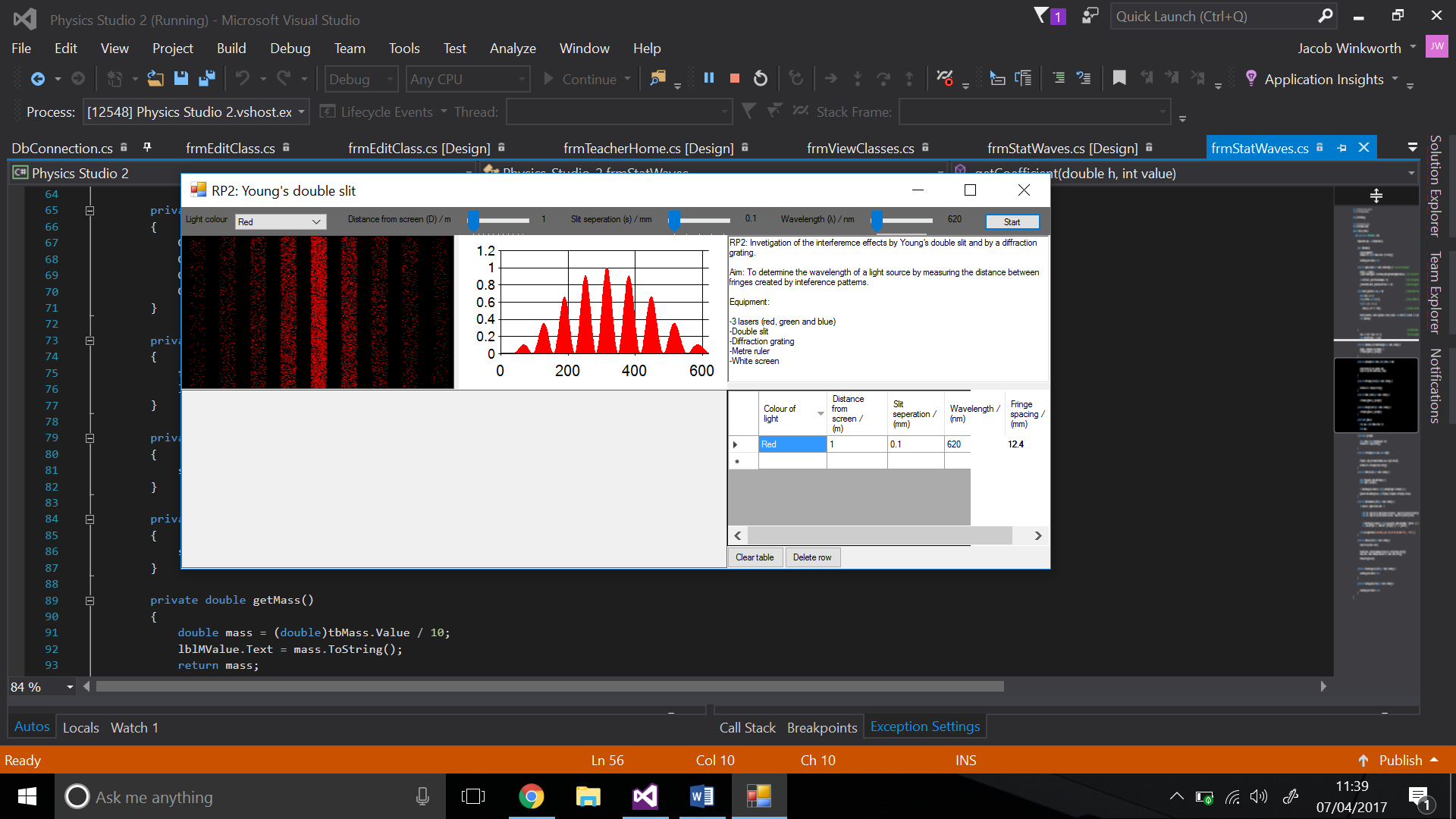
return pointArray;

}

This second function returns an array of coordinate pairs after having two coefficients passed as arguments. Two arrays of doubles are created which store the x and y coordinates generated in the For loop. As this For loop iterates, it uses the coefficients passed to calculate the values for both x and y. After completing this, the ‘Linq’ extension for Visual Studio is used to combine the arrays together. As such, the new array contains values of coordinate pairs that will be returned by the function.

* xStep = A double that is used to calculate the spacing between x values on the Sine graph.
* xValues = An array of doubles that represent the x Coordinates generated by the For loop.
* yValues = An array of doubles that represent the y Coordinates generated by the For loop.
* pointArray – An array of type PointF that are the coordinates returned by the function and used to draw the sine curve in the Paint Event handler. This is generated by the combination of both the xValues array and the yValues array.

### Young’s double slit

The Young’s Double slit experiment is an experiment which creates an interference pattern when a laser is fired between two slits of miniscule separation and width. This experiment is an example of the wave-particle duality of light, as it is seen that light acts as both a wave and a particle. Its particle property is displayed as the photons travel via a ‘wave of probability’, where the most intense areas of light on the screen show the areas where a photon is most likely to fall. Its wave property is shown via the diffraction of the light as it travels through the double slit.

This creates a projection onto the screen behind, with columns of light of varying intensities. The areas where no light is shown is due to the superposition of a peak and a trough in the wave, which cancels out the amplitude of the wave all together, hence the probability becomes zero. Whereas, the central maximum is the superposition of either two peaks or two troughs to form a more intense area of light. The image displayed below shows a rough visualisation of how the photons fall onto each column. Please note, this example is only a simulation, and follows no truth to the Quantum Mechanical laws by which these events are handled in nature, as that would not only exceed my objectives, but also venture way beyond the A level syllabus.

In summary, the algorithm used to achieve this should be based upon probability, as to simulate the underlying ‘randomness’ of Quantum Mechanics.

##### Code for Double Slit simulation

private void pnlScreen\_Paint(object sender, PaintEventArgs e)

{

Graphics g = e.Graphics;

Brush brush = new SolidBrush(Color.FromArgb(100, rVal, gVal, bVal));

Random rndNum = new Random();

int xStart = 0; int xValue; int yValue;

for (int i = 0; i < 10000; i++)

{

xStart = 170;

yValue = rndNum.Next(0, 200);

if (i % 3 == 0) { if (i % 2 == 0) { xStart = 130; } else xStart = 210; }

else if (i % 5 == 0) { if (i % 2 == 0) { xStart = 90; } else xStart = 250; }

else if (i % 7 == 0) { if (i % 2 == 0) { xStart = 50; } else xStart = 290; }

else if (i % 13 == 0) { if (i % 2 == 0) { xStart = 10; } else xStart = 330; }

xValue = xStart + rndNum.Next(0, 20);

g.FillEllipse(brush, xValue, yValue, 2, 2);

}

}

Please also note, the values added to or deducted from the ‘xStart’ variable depends on the size of the control used.

In short, this algorithm uses the properties of prime numbers to select which column each ‘photon’ should be drawn in. For example, every number between 0 and 10000 divisible by 3 (10000/3) will be placed in the column next to the central maximum, then every number divisible by 5 (10000/5) in the next column etc.

However, to make the diffraction symmetrical, each column either side of the maximum should contain the same amount of ‘particles’ as its corresponding column. To do this, a modulus function is also applied to each number as it is accepted by an if statement. If the number is divisible by two, then it is placed to the left of the centre. If not, then it is placed to the right of the centre.

Example:

For an input of 225:

**Input** 225

**Divisible by 3?** No

**Divisible by 5?** Yes 🡺 **Divisible by 2?** No

**Output** Placed in second column from the centre on the right.

Problems encountered:

Each number used with the modular function should not be divisible by the previous, as to help prevent collisions with numbers passed. For example, if ‘i mod 7’ was replaced with ‘i mod 10’, then every other number that is divisible by 10 will also be divisible by 5, hence will be placed in that column instead. This will create an uneven pattern. As such, primes are used to help prevent these collisions as best as possible.

This algorithm will also ensure the central maximum has a much greater intensity than the others, as all numbers divisible by any number other than the four presented above will be placed in this one. This, of course, includes all even numbers. Hence, this ensures over half of the dots are placed in this maximum.

### Snell’s law

The Snell’s law experiment is used to identify the refractive index of a medium. When light passes from one medium to another of different density, the light is refracted towards the normal (if the second medium is denser) or away from the normal (if the second medium is less dense). The angle of incidence to the medium and the angle of refraction after entering the medium can be measured to work out the **refractive index** of the medium (essentially how much slower light travels in that medium in comparison to a vacuum). The Snell’s law formula is as follows:

**N1Sin(i) = N2Sin(r)**

**Where N1 is the refractive index of the first medium, i is the angle of incidence, N2 is the refractive index of the second medium and r is the angle of refraction.**

After carrying out the experiment, recording a range of values of **r** for a range of values of **i**, a graph can be plotted for **sin(i) against sin(r).** This is because the refractive index of air is 1, hence:

Taking the gradient of this graph will therefore give the refractive index of **N2**.

#### Code for Snell’s law simulation

private List<PointF> getPoints(double angle, double index1, double index2, List<PointF> pointList, PointF startPoint, double height)

{

float yStart = startPoint.Y;

double angRefraction = getAngleR(index1, index2, angle);

if (yStart >= 0)

{

lblRefractedAngle.Text = Math.Abs(Math.Round((angle / (Math.PI / 180)),

2)).ToString();

float xStart = startPoint.X;

double length = Math.Abs(getLength(angle, height));

float xIncident = (float)(xStart + (Math.Sin(angle)) \* length);

float yIncident = (float)(yStart - (Math.Cos(angle)) \* length);

double critAngle = getCritAngle(index1, index2);

if (xIncident > pnlExperiment.Width && yIncident > pnlExperiment.Height / 3)

{

height = (xIncident - pnlExperiment.Width - 5) / (float)Math.Tan(angle);

yIncident += (int)height;

xIncident = pnlExperiment.Width; angRefraction = -angle;

}

else height = pnlExperiment.Height / 3;

PointF pointofIncidence = new PointF(xIncident, yIncident);

pointList.Add(pointofIncidence);

return getPoints(angRefraction, index2, index1, pointList, pointofIncidence,

height);

}

return pointList;

}

private double getAngleR(double ni, double nr, double I)

{

double angleR;

double theta = ni \* Math.Sin(I) / nr;

angleR = Math.Asin(theta);

return angleR;

}

private double getLength(double angle, double height)

{

double length = height / Math.Cos(angle);

return length;

}

The above code uses a recursive method to gather sets of points to draw lines between to simulate the refraction of light between two mediums. In short, the getPointsRecursive function adds points recursively to a list until the y value reaches 0, in which case meaning it has reached the top of the panel control.

The getAngleR function returns the value of the refracted angle after the ‘light’ has entered the next medium. This uses the formula explained earlier, but rearranged to give the equation:

Where *N1* is the refractive index of the first medium, *i* is the angle of incidence, and *N2* is the refractive index of the second medium.

The getLength function uses the trigonometric formula Cosθ = adjacent/hypotenuse to find the length of the line to draw. The angle θ is measured from the normal to the horizontal.

θ

Height

**float** xIncident = xStart + Cos(angfrmNorm) \* length**)**

**float** yIncident = yStart - Sin(angfrmNorm) \* length)

Each of these lines also use trigonometry to get both the x and y coordinates of the point in which the ‘light’ would be incident to the next medium. As seen, the ‘length’ variable is used here to transform the coordinates accordingly.

(xIncident, yIncident)

(xIncident, yStart)

(XStart, YStart)

#### The graph for Snell’s law

As shown earlier, plotting the sin of the angle of incidence against the sine of the angle of refraction should give the refractive index of the medium used, providing the medium the light is travelling from is air.

As these two values are proportional to each other, the graph should be a straight line passing through the point of origin. It should look not too dissimilar to this:

## Database

### Entity relationship diagram

∞

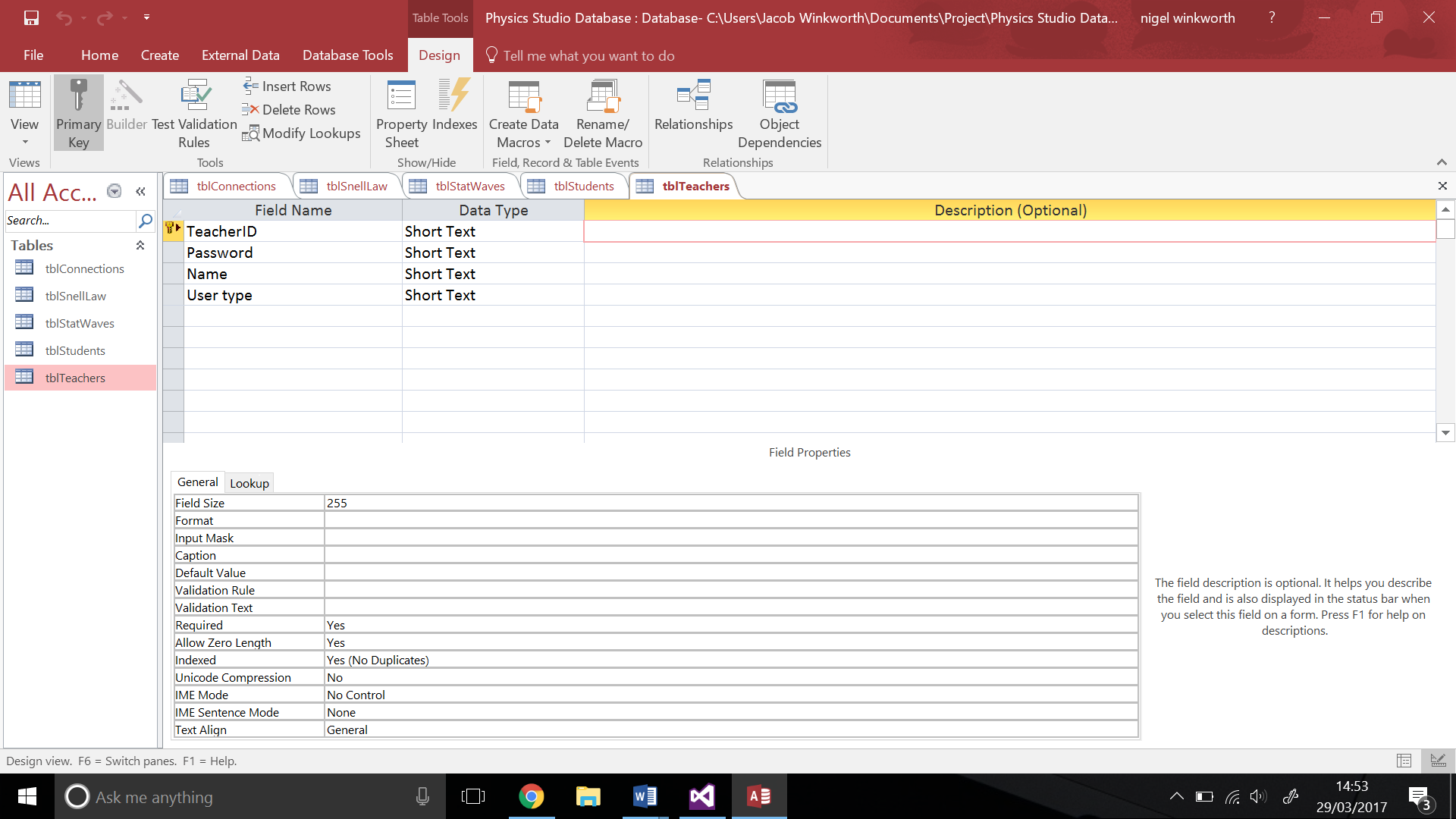
1

1

∞

### Table designs

Teacher table



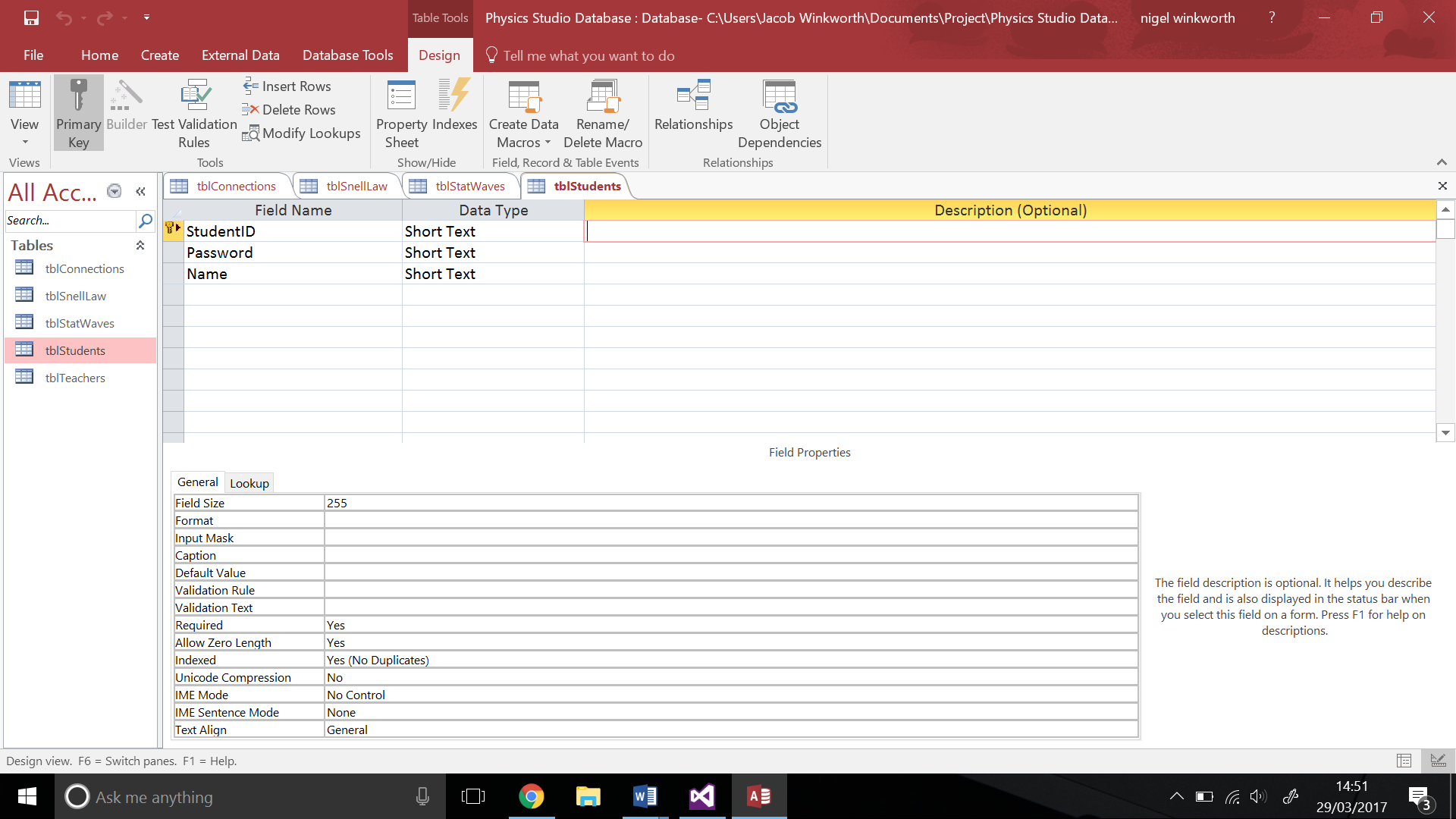
TeacherID: An alphanumeric, unique identifer given to a teacher and stored as Short Text. [PRIMARY KEY]

Password: An alphanumeric hash of a string entered by the admin upon creating the account. Stored as Short Text.

Name: The name of the teacher, entered upon the admin creating the account and stored as Short Text.

User type: Can be either A (Adminstrator) B (Teacher), as teachers and administrators are both stored in the same table. Used to know which home page to load upon login.

Student table

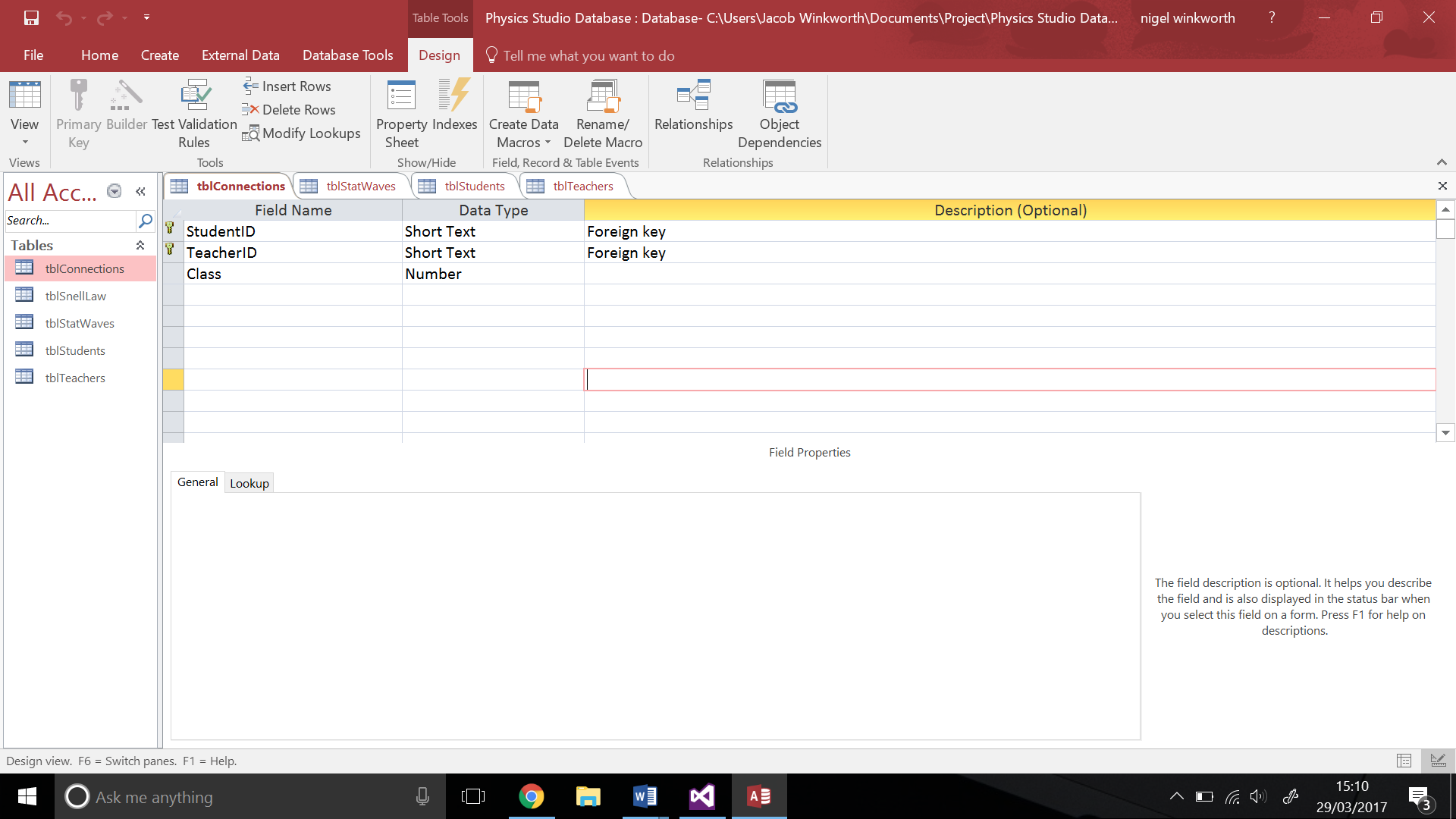


StudentID: An alphanumeric, unique identifer created upon student sign-up and stored as Short Text. [PRIMARY KEY]

Password: An alphanumeric hash of a string entered by the student upon sign-up. Stored as Short Text.

Name: The name of the student, entered by the student upon sign-up and stored as Short Text.

Connections table



StudentID: An alphanumeric, unique identifer created upon student sign-up and stored as Short Text. A foreign key that is taken from the Student table, and is one half of a composite key that is used as the overall Key for the table.

TeacherID: An alphanumeric, unique identifer given to a teacher and stored as Short Text. A foreign key that is taken from the Teacher table, and is one half of a composite key that is used as the overall Key for the table.

Class: A number assigned to a student when a teacher creates a class. Stored as a Number, and is used to identify specifically the class that a student is in.

### SQL Examples

#### Sign up & login

Student insertion

INSERT INTO tblStudents

VALUES(StudentID, PasswordHash, Name)

This will insert a student’s StudentID into the ‘Students’ table, along with a hash of their password and their name.

Student login SELECT statement

SELECT tblStudents.StudentID, Name, TeacherId, Class from tblStudents

INNER JOIN tblConnections ON tblStudents.StudentID = tblConnections.StudentID

WHERE tblStudents.StudentID = UserID and tblStudents.Password = Password

This returns all elements of the tblStudents table, along with their class number and the teacher the student is linked with via the tblConnections class. The data will be returned as a DataTable.

Error checking

#### Returning data

Selecting students linked to a specific teacher

SELECT tblStudents.StudentID, tblStudents.Name, tblConnections.Class FROM tblStudents

INNER JOIN tblConnections

ON tblStudents.StudentID = tblConnections.StudentID

WHERE tblConnections.TeacherID = @ID

ORDER BY tblStudents.StudentID ASC"

This returns all students linked to a specific teacher in alphabetical order.

### Link to objectives

## Object Oriented Programming

To accommodate for my use of a sign-up/login system, I used a three-tier architectural design. This includes three namespaces – DAL, BEL and BAL.

DAL – This namespace consists of the dbConnections class, which is used to either create or terminate a connection with the database, and pass SQL commands to the database.

BEL – This namespace consists of the Information class, which holds all variable necessary for the sign-up or login of a student. These variables are: UserID, Name, Password, and userType. This information is fetched from the SQL database via the dbConnections and SQLOperations classes.

BAL – This namespace consists of the SQLOperations class, which holds all the necessary operations used to access the database. Such operations include inserting information for a student, updating information for a student and fetching information about a student.

## Application links

### Saving results to a text file

In order for a teacher to have the ability to view the results of a particular student, the student is able to save there DataGridView control which holds the data gathered from the experiment directly to a texfile. To do this, I used the JSON.Net extension for Visual Studio C# to convert the DataGridView directly into a JSON format. This JSON string is then written to and saved as a textfile. Here is the code used which allows a student to save their results as a JSON string in a textfile.

private void btnUpload\_Click(object sender, EventArgs e)

{

string path = @"C:\Users\Jacob Winkworth\Documents\Computer Science\**Experiment**

**Name**\" + info.userID + ".text";

string json = ConvertDataTabletoJSON();

System.IO.File.WriteAllText(path, json);

}

public string ConvertDataTabletoJSON()

{

JavaScriptSerializer serializer = new JavaScriptSerializer();

List<Dictionary<string, object>> listRows = new List<Dictionary<string, object>>();

Dictionary<string, object> row;

foreach (DataRow dr in dt.Rows)

{

row = new Dictionary<string, object>();

foreach (DataColumn col in dt.Columns)

{

row.Add(col.ColumnName, dr[col]);

}

listRows.Add(row);

}

return serializer.Serialize(listRows);

}

This function returns a JSON string. A new instance of the JavaScriptSerializer is first created, along with a new List of Dictionaries. A new dictionary for any single row is also then created. A For loop iterates through each row of the DataTable. For each of the rows, the column name and contents of a cell is added to the ‘row’ dictionary. After this row has been exhausted, the Foreach loop will be complete and will fall back to the previous Foreach loop. From here, this row is then added to the list of rows, and the same process will occur for the next row in the datatable.

After this process has been carried out, the listRows List is serialized and returned by the function.

Their file will be saved into a folder which is named accordingly. I.E. for the experiment Young’s Double Slit, then the **Experiment Name** in the file path will be “Young’s Double Slit”.

To view this file, the code shown below is used:

private void dgvStudents\_CellContentDoubleClick(object sender, DataGridViewCellEventArgs e)

{

string folder = cbFilterExperiments.SelectedItem.ToString();

string username = dgvStudents.Rows[e.RowIndex].Cells[0].Value.ToString();

try

{

string jsonString = System.IO.File.ReadAllText(@"C:\Users\Jacob

Winkworth\Documents\Computer Science\" + folder + @"\" + username + ".text");

DataTable dtResults = JsonConvert.DeserializeObject<DataTable>(jsonString);

dgvLoadResults.DataSource = dtResults;

}

catch { MessageBox.Show("This user has not completed the chosen experiment"); } }

The ‘try-catch’ statement here is used to prevent the program attempting to find a file which does not exist as the selected user has not yet saved their results. In which case, the program would throw an exception.

# User interface

## Main interface

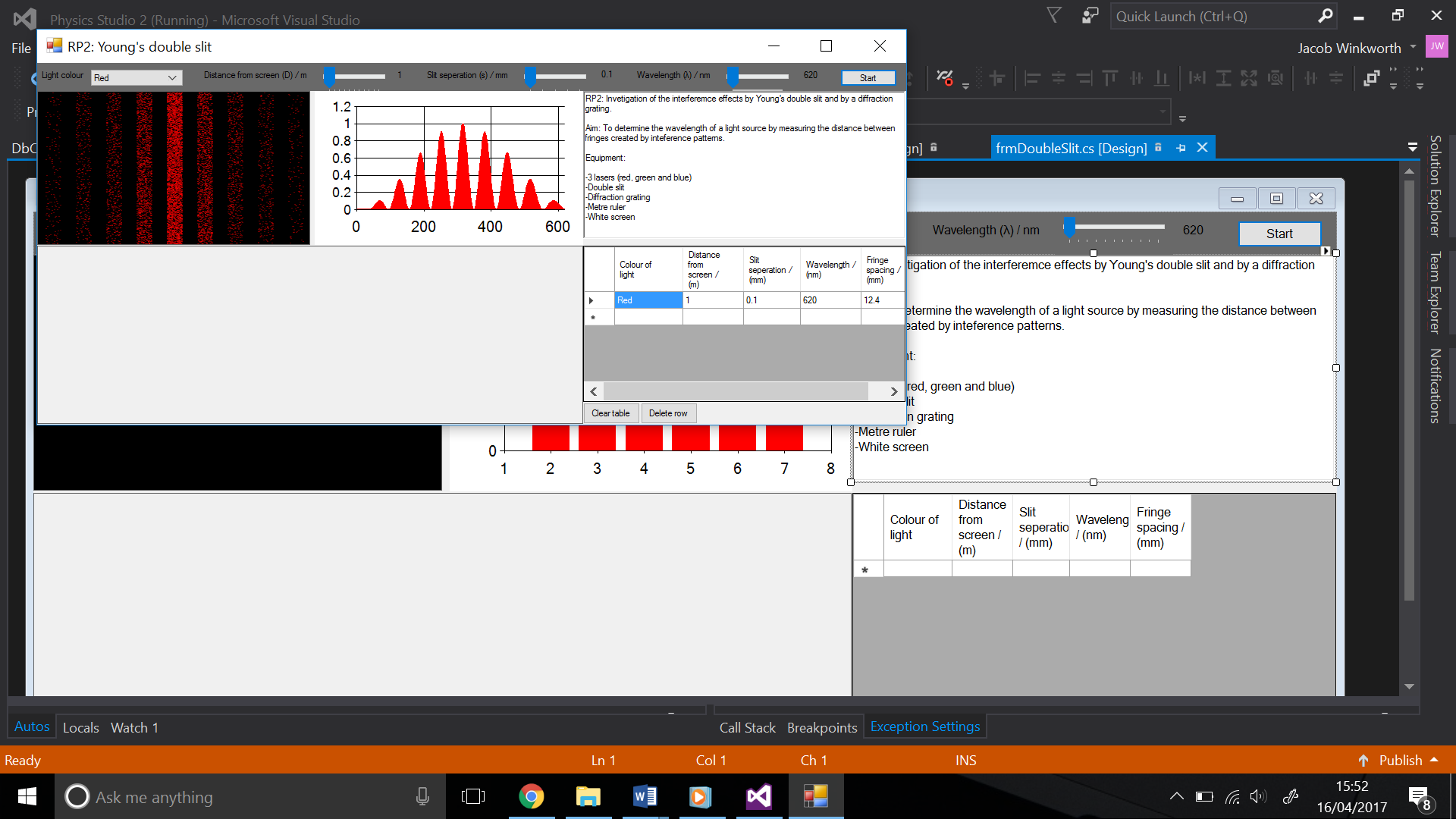
My program interface will consist of three main home pages, depending on what type of user has been logged in. The three user types that will be used are:

### Admin

### Student

### Teacher

## Experiment interface

The second type of interface that I will need to use is a standardised, consistent template for each experiment. Here is an example of this interface below, using the Young’s Double Slit experiment:

The upper control bar will be adapted in each experiment with different trackbars/buttons/combo boxes etc. to accommodate each one. This consistency allows the program to have a more professional, scientific finish, also remaining sleek between form transitions.

The top right panel shows a brief description of the experiment the user is carrying out (object 5.1).

# Security and backup

### Database security

To prevent SQL injections, all SQL used to access the database will be parameterised. Here is an example of this:

public DataTable studentLogin(Information info)

{

OleDbCommand cmd = new OleDbCommand();

cmd.CommandType = CommandType.Text;

cmd.CommandText = "SELECT tblStudents.StudentID, Name, TeacherId, Class from

tblStudents INNER JOIN tblConnections ON tblStudents.StudentID =

tblConnections.StudentID " +

"WHERE tblStudents.StudentID = @User and tblStudents.Password = @Password";

cmd.Parameters.AddWithValue("@User", info.userID);

cmd.Parameters.AddWithValue("@Password", info.Password);

return db.ExeReader(cmd);

}

All password entries in both the students table and the teachers table will also first be hashed to boost security. They will not be unhashed until the SELECT statement to get the student/teacher details is run.

Here is the code that will convert the password string into a hashed string:

public string getHash(string hashValue)

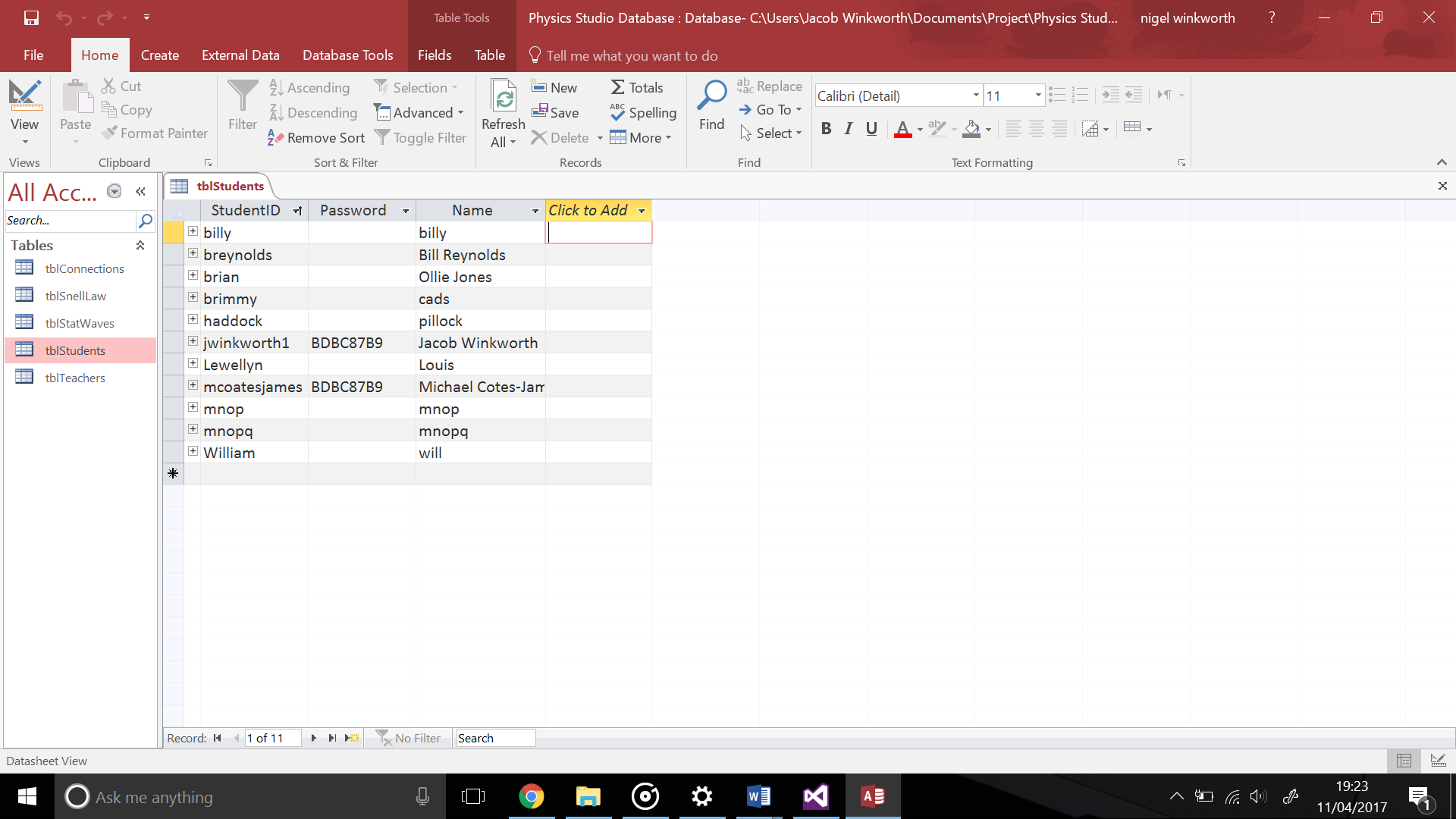
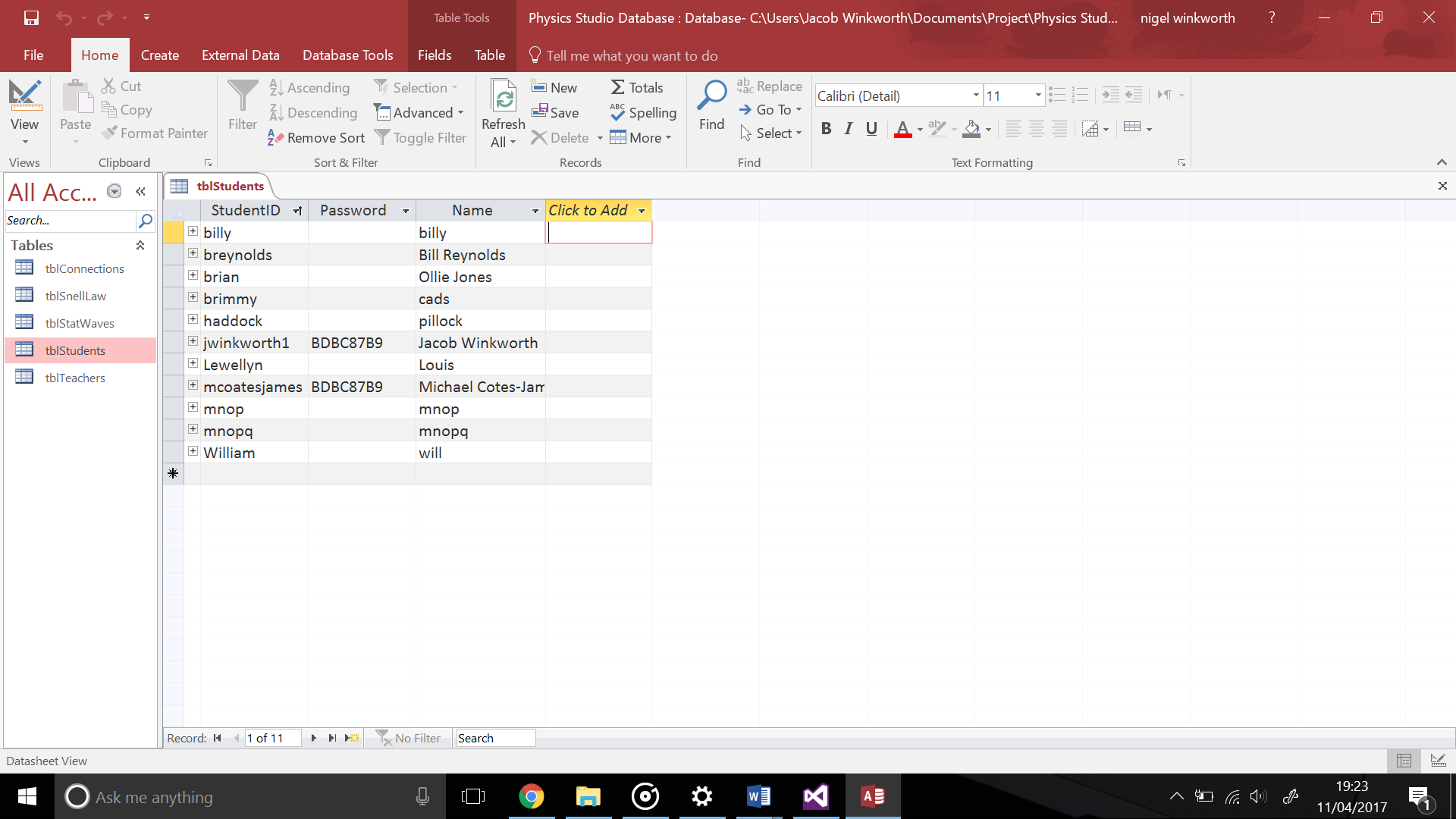
{

string hashCode = String.Format("{0:X}", hashValue.GetHashCode());

return hashCode;

}

To boost security further, at no point throughout the program will the password be converted into its original string. Instead, the password that was entered by the user when logging in will simply be converted into the hashed string and compared against the stored hash in the database.

Here is an example of a database entry where the password has been hashed using this method:

Where the original, unhashed password string was ‘Pass123’.

### Backup

As explained previously, the student has the ability to save their results to a file, which can also be read by the teacher. This also works as a backup mechanism for the students, as they are also able to load their results for the experiment back into the program.