ICCS240: Database Management System

Population Estimation in Thailand

Group member:

Sirin Chankao (5980753)

Chakeera Wansoh (6080566)

Natthakan Euaumpon (6081213)

Overview:

This report will act as both our demo and report.
This project can be referred in the Github repo:
https://github.com/chakeera/PopulationEstimation

Theory:

For the sake of explanation, let's assume that A is a matrix. This will work only if A is a symmetric matrix. We need to find all positive eigenvalues which are required for our numerical approach. We are going to implement this by combining Power Method with Rayleigh quotient iteration. This iteration has the following formula:

$$b_{i+1} = \frac{(A-\mu_i I)^{-1} b_i}{\|(A-\mu_i I)^{-1} b_i\|}$$

Where b_{i+1} is the next approximation of the eigenvector and I is the identity matrix. After that we can set the next approximation of an eigenvalue to the Rayleigh quotient:

$$\mu_{i+1} = \frac{b^*_{i+1} A b_{i+1}}{b^*_{i+1} b_{i+1}}$$

However, we need to compute all eigenvalues so we need to combine this technique with a deflation technique. Now we will get the largest possible eigenvalue. But we need to consider all of them. There are many deflation methods, but for this project, we are going to use Hotelling's deflation. This is one of the most popular techniques.

$$(A - \lambda_1 u_1 u_1{}^T)$$

Where u are the elements in a set of N vectors, as eigenvectors are the set of N vectors of A. From the equation above, we will get the matrix that has the same eigenvectors and eigenvalues as matrix A. But the largest eigenvalue is replaced with 0. Thus we can combine this with the Power method and Rayleigh quotient to find the next biggest and so on until we find all the eigenvalues. Then we substitute the eigenvalues into a matrix to get our eigenvectors where it will be multiplied with an $n \; x \; n$ matrix that is the matrix of the offspring and number of survivors and its result will give us the estimated population value.

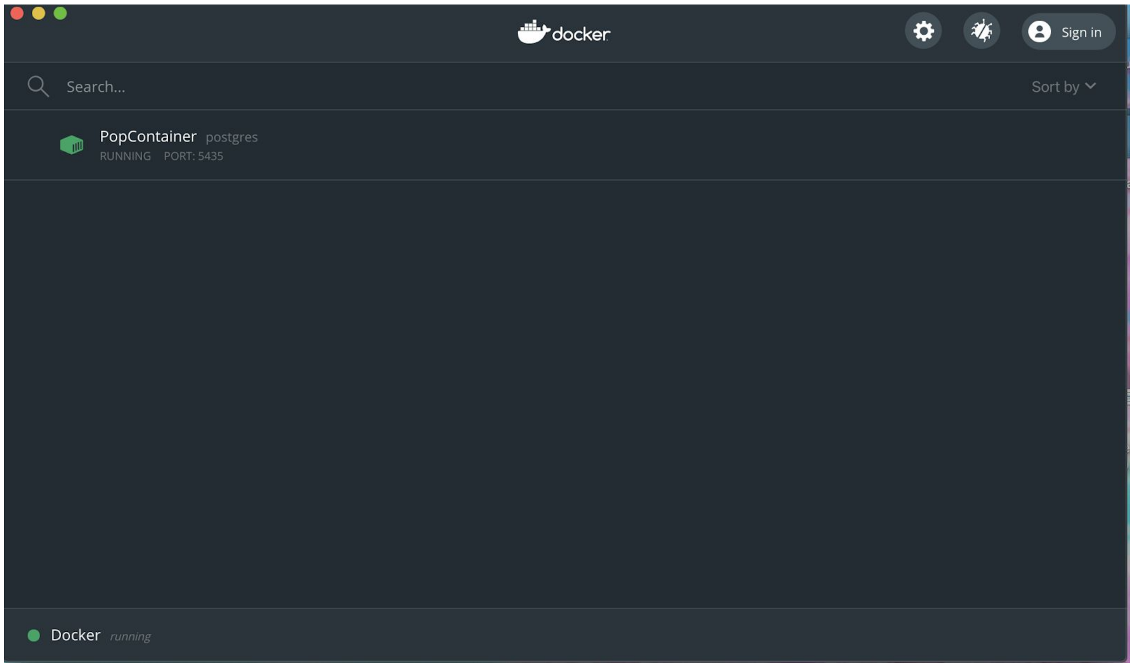
Let x be a fraction of the change in population in each year and y be the total number of offspring produced each year. Let x_1 and y_1 be the value for the first year and so on. Then we would get a matrix that looks like this:

$$\begin{bmatrix} 0 & y_2 & y_3 & y_4 & y_5 & y_6 \\ x_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & x_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & x_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_5 & 0 \end{bmatrix}$$

We will put this in the form of a nested 2D array, where each sub-array will contain values in each row. After that we compute the eigenvalues. For this project, we only want positive eigenvalues. This is done by using the Power method, Rayleigh quotient, LU decomposition and deflation techniques. We then convert a matrix from the above function to matrix form and compute. We will continue doing until we get satisfied values. Then we compute the eigenvector, which will then be multiplied with the original given matrix that we retrieved from the database to get our final results. Our final results would be an array where each value is the approximation of the total population in each age group.

DEMO:

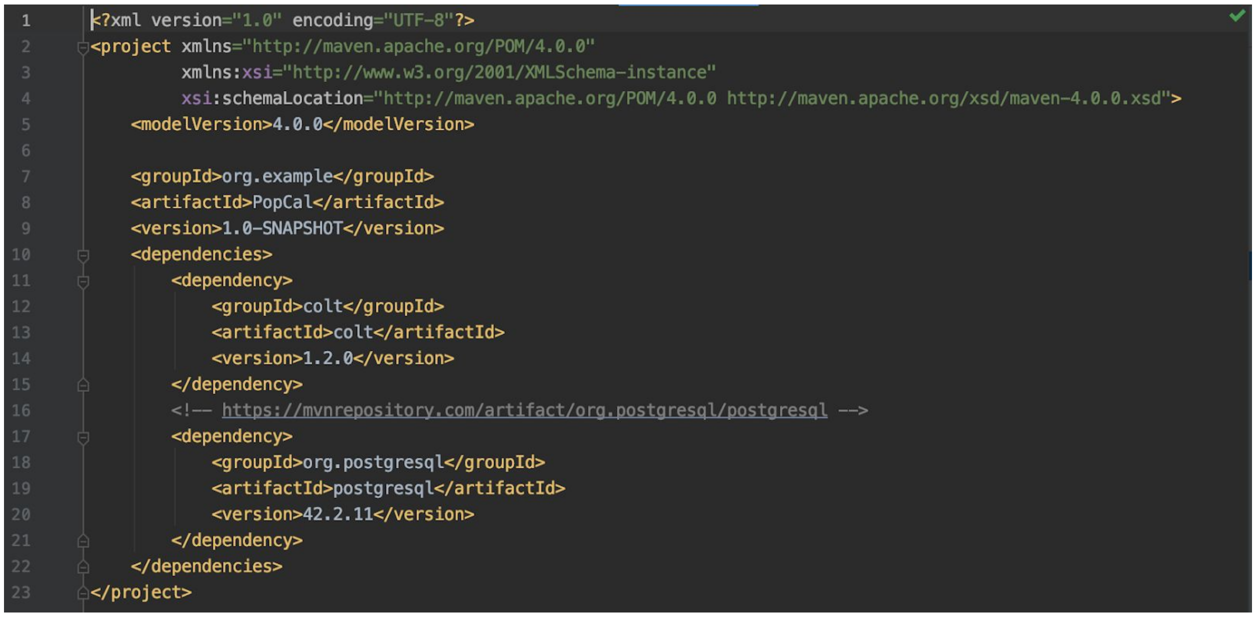
We used Java in **IntelliJ**, **Docker**, **Maven**, **Postgres** for this project.
Firstly, let us show the connection for databases in Docker and IntelliJ. In the below image is our container in Docker to run Postgresql.



PopContainer postgres
RUNNING PORT: 5435

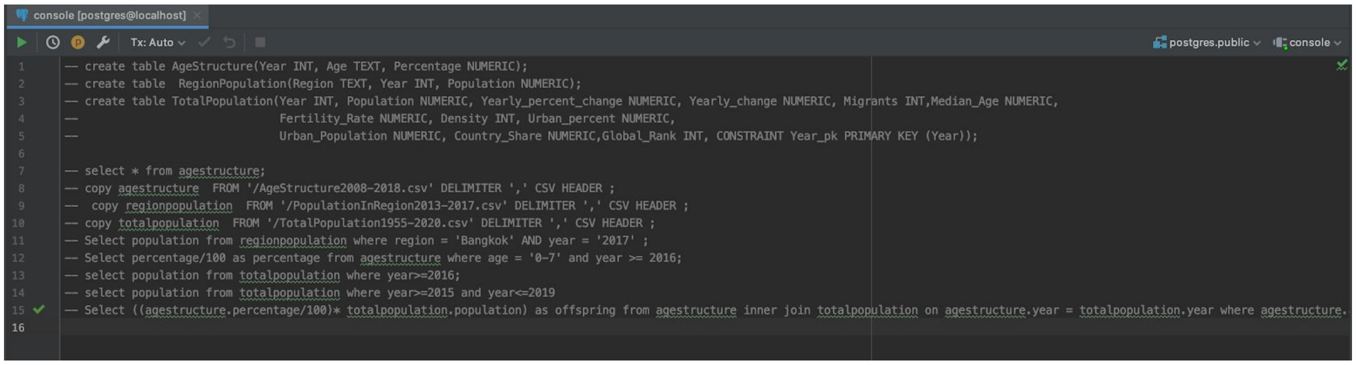
Docker running

Next is the dependencies that we use in the project. We used colt to solve the matrix during Rayleigh quotient iteration, and postgresql to connect to our Postgresql database.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>PopCal</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>colt</groupId>
            <artifactId>colt</artifactId>
            <version>1.2.0</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>42.2.11</version>
        </dependency>
    </dependencies>
</project>
```

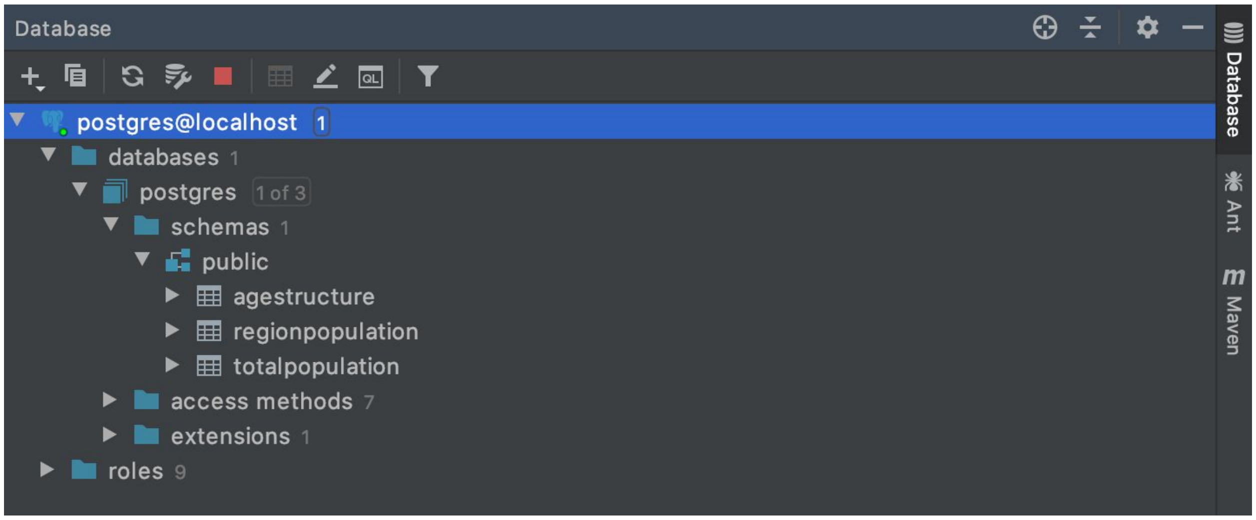
After all the connection we use the postgres console to try and test our sql as seen below,



```sql
-- create table AgeStructure(Year INT, Age TEXT, Percentage NUMERIC);
-- create table  RegionPopulation(Region TEXT, Year INT, Population NUMERIC);
-- create table TotalPopulation(Year INT, Population NUMERIC, Yearly_percent_change NUMERIC, Yearly_change NUMERIC, Migrants INT,Median_Age NUMERIC,
--                              Fertility_Rate NUMERIC, Density INT, Urban_percent NUMERIC,
--                              Urban_Population NUMERIC, Country_Share NUMERIC,Global_Rank INT, CONSTRAINT Year_pk PRIMARY KEY (Year));

-- select * from agestructure;
-- copy agestructure  FROM '/AgeStructure2008-2018.csv' DELIMITER ',' CSV HEADER ;
--  copy regionpopulation  FROM '/PopulationInRegion2013-2017.csv' DELIMITER ',' CSV HEADER ;
-- copy totalpopulation  FROM '/TotalPopulation1955-2020.csv' DELIMITER ',' CSV HEADER ;
-- Select population from regionpopulation where region = 'Bangkok' AND year = '2017' ;
-- Select percentage/100 as percentage from agestructure where age = '0-7' and year >= 2016;
-- select population from totalpopulation where year>=2016;
-- select population from totalpopulation where year>=2015 and year<=2019
-- Select ((agestructure.percentage/100)* totalpopulation.population) as offspring from agestructure inner join totalpopulation on agestructure.year = totalpopulation.year where agestructure.
```

Now, we will introduce our database schemas and its data types. We have three tables in total.

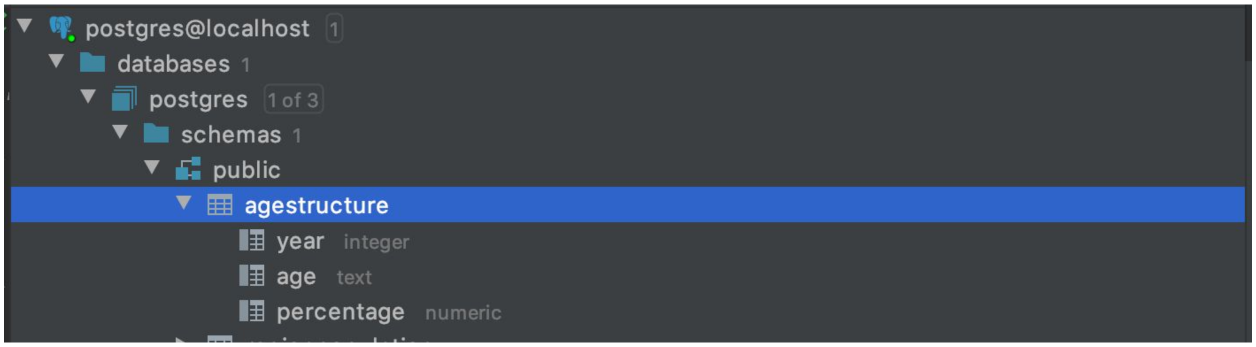
1) Agestructure
2) Regionpopulation
3) Totalpopulation



```
Database
▼ postgres@localhost 1
  ▼ databases 1
    ▼ postgres 1 of 3
      ▼ schemas 1
        ▼ public
          ▶ agestructure
          ▶ regionpopulation
          ▶ totalpopulation
      ▶ access methods 7
      ▶ extensions 1
  ▶ roles 9
```

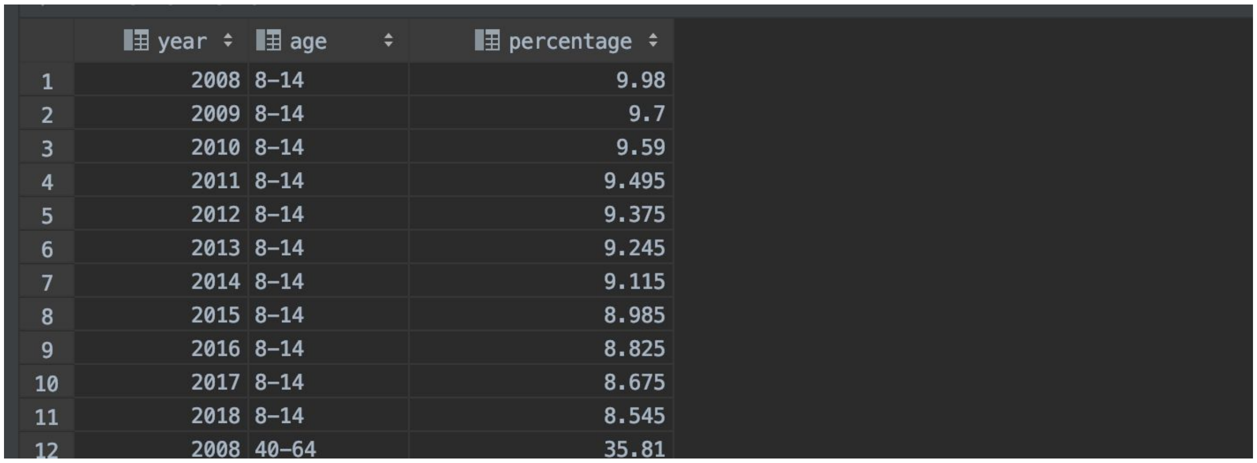
The table **Agestructure** contains the year, age, and the percentage of the people with that range of age. The schema is

agestructure(Year INT, Age TEXT, Percentage NUMERIC)



```
▼ postgres@localhost 1
  ▼ databases 1
    ▼ postgres 1 of 3
      ▼ schemas 1
        ▼ public
          ▼ agestructure
              year  integer
              age  text
              percentage  numeric
```

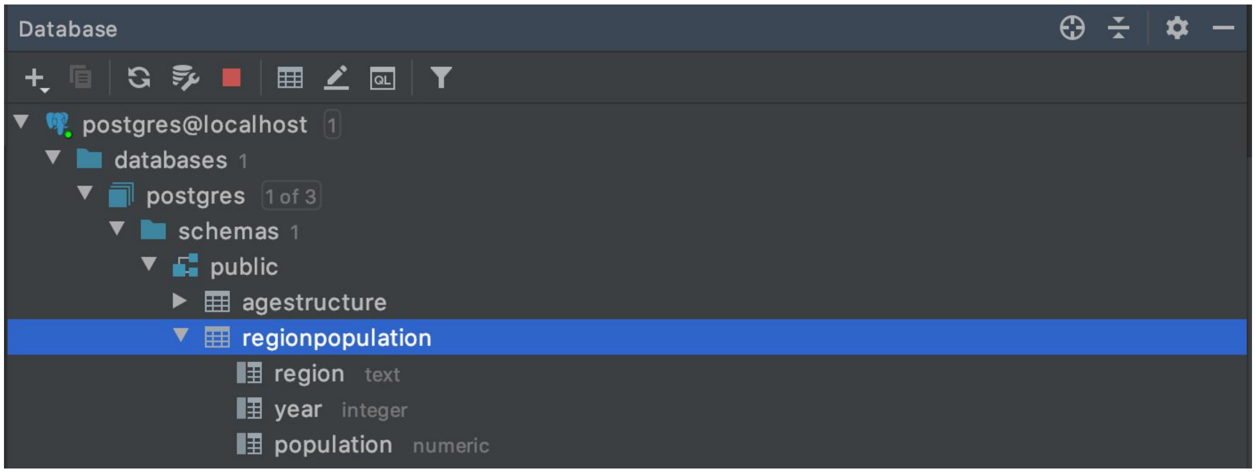
Here is some part of the data in agestructure. The whole data can be seen in the folder dataset in the github repo.



	year	age	percentage
1	2008	8–14	9.98
2	2009	8–14	9.7
3	2010	8–14	9.59
4	2011	8–14	9.495
5	2012	8–14	9.375
6	2013	8–14	9.245
7	2014	8–14	9.115
8	2015	8–14	8.985
9	2016	8–14	8.825
10	2017	8–14	8.675
11	2018	8–14	8.545
12	2008	40–64	35.81

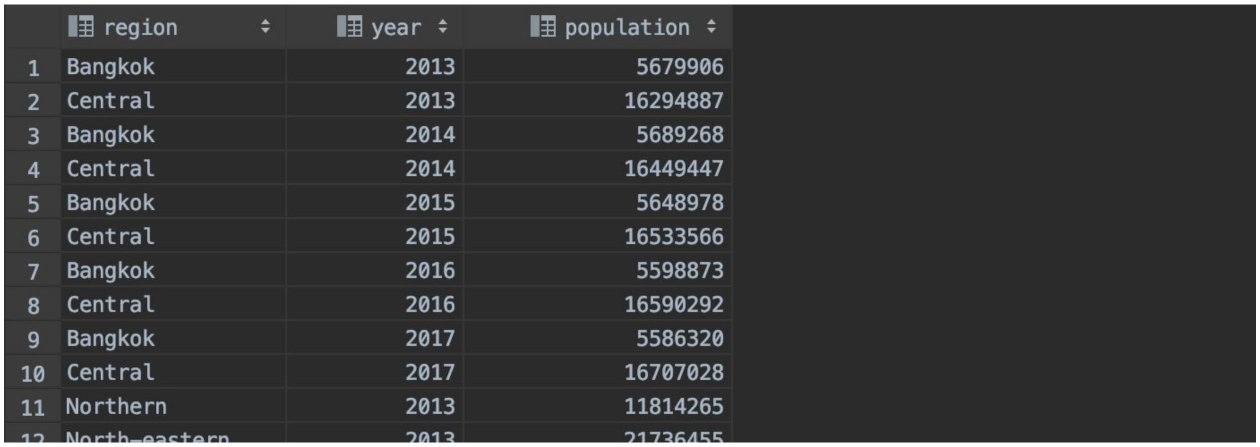
The table **Regionpopulation** contains region, year, and population. We have 5 regions: central, northern, north-eastern, eastern, and southern. The population of each region corresponds to each year. The schema is

RegionPopulation(Region TEXT, Year INT, Population NUMERIC)



```
postgres@localhost 1
  databases 1
    postgres 1 of 3
      schemas 1
        public
          agestructure
          regionpopulation
            region text
            year integer
            population numeric
```

Here is some part of the data in regionpopulation. The whole data can be seen in the folder dataset in the github repo.

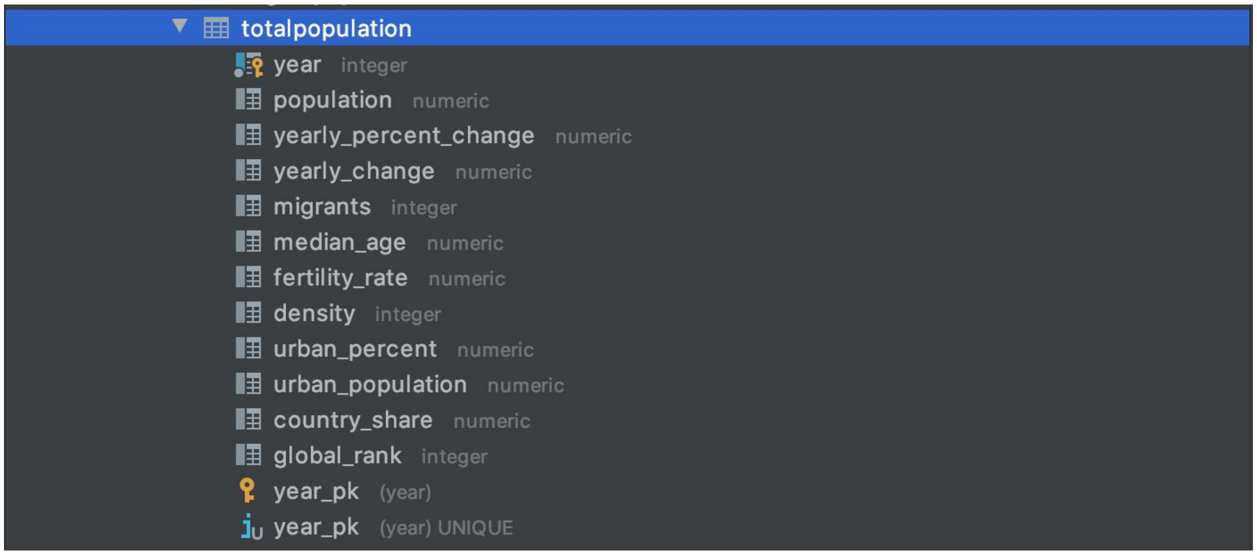


	region	year	population
1	Bangkok	2013	5679906
2	Central	2013	16294887
3	Bangkok	2014	5689268
4	Central	2014	16449447
5	Bangkok	2015	5648978
6	Central	2015	16533566
7	Bangkok	2016	5598873
8	Central	2016	16590292
9	Bangkok	2017	5586320
10	Central	2017	16707028
11	Northern	2013	11814265
12	North-eastern	2013	21736455

The table totalpopulation contains year, population, yearly percentage change, yearly change, migrants, median age, fertility rate, density, urban percent, urban population, country share, and global rank. The year range of this table is 1955-2020. The schemas are

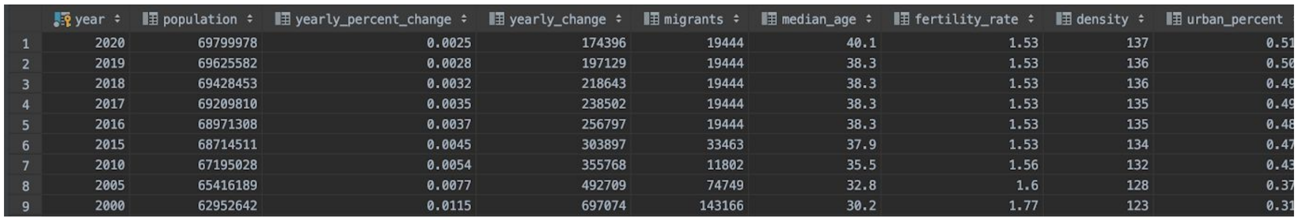
TotalPopulation(Year INT, Population NUMERIC, Yearly_percent_change NUMERIC, Yearly_change NUMERIC, Migrants INT,Median_Age NUMERIC, Fertility_Rate NUMERIC, Density INT, Urban_percent NUMERIC, Urban_Population NUMERIC, Country_Share NUMERIC,Global_Rank INT)

We set year as the primary key so that we can be sure that they are unique and can use them later in other functions.



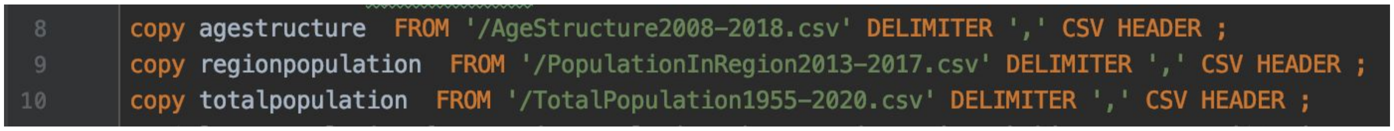
```
totalpopulation
    year integer
    population numeric
    yearly_percent_change numeric
    yearly_change numeric
    migrants integer
    median_age numeric
    fertility_rate numeric
    density integer
    urban_percent numeric
    urban_population numeric
    country_share numeric
    global_rank integer
    year_pk (year)
    year_pk (year) UNIQUE
```

Here is some part of the data in regionpopulation. The whole data can be seen in the folder dataset in the github repo.



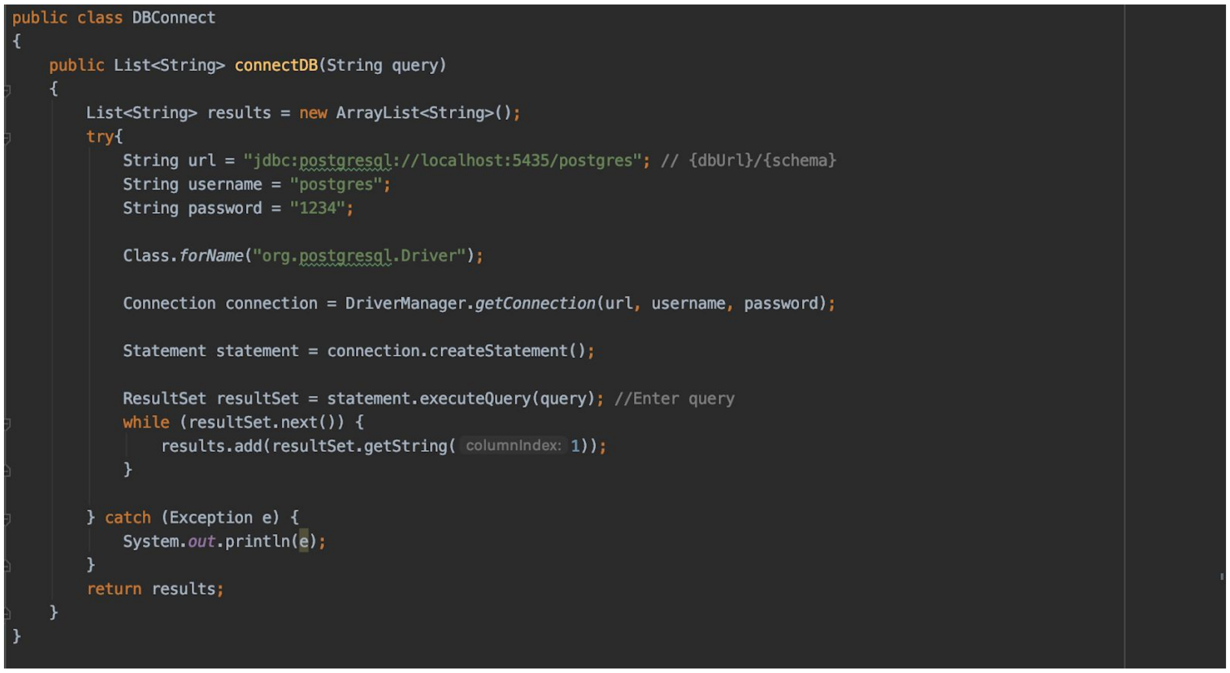
	year	population	yearly_percent_change	yearly_change	migrants	median_age	fertility_rate	density	urban_percent
1	2020	69799978	0.0025	174396	19444	40.1	1.53	137	0.51
2	2019	69625582	0.0028	197129	19444	38.3	1.53	136	0.50
3	2018	69428453	0.0032	218643	19444	38.3	1.53	136	0.49
4	2017	69209810	0.0035	238502	19444	38.3	1.53	135	0.49
5	2016	68971308	0.0037	256797	19444	38.3	1.53	135	0.48
6	2015	68714511	0.0045	303897	33463	37.9	1.53	134	0.47
7	2010	67195028	0.0054	355768	11802	35.5	1.56	132	0.43
8	2005	65416189	0.0077	492709	74749	32.8	1.6	128	0.37
9	2000	62952642	0.0115	697074	143166	30.2	1.77	123	0.31

All the dataset are in the form .csv file and we imported in using \COPY as in the console



```
copy agestructure  FROM '/AgeStructure2008-2018.csv' DELIMITER ',' CSV HEADER ;
copy regionpopulation  FROM '/PopulationInRegion2013-2017.csv' DELIMITER ',' CSV HEADER ;
copy totalpopulation  FROM '/TotalPopulation1955-2020.csv' DELIMITER ',' CSV HEADER ;
```

We have a seperate class for database connection called DBConnect. In this class, all the JDBC are ready with its connection, statement, and resultset. (image below)

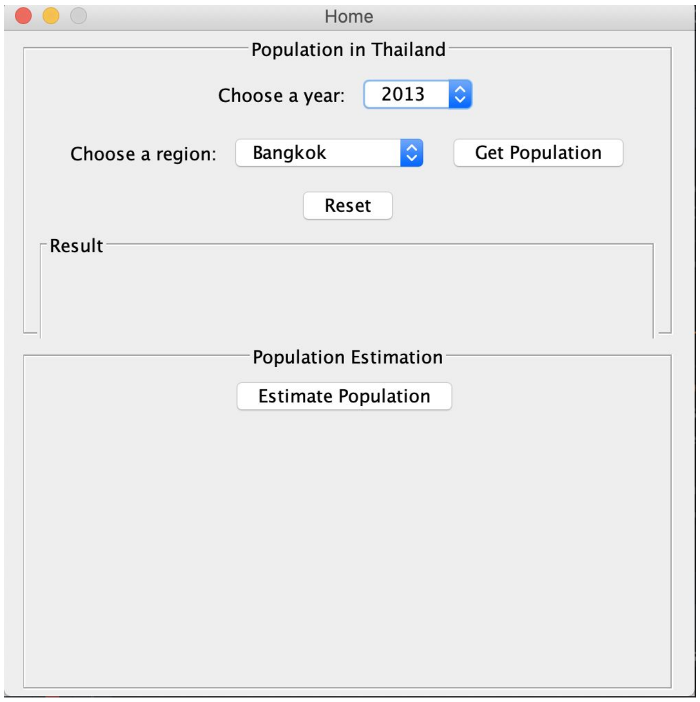


```java
public class DBConnect
{
    public List<String> connectDB(String query)
    {
        List<String> results = new ArrayList<String>();
        try{
            String url = "jdbc:postgresql://localhost:5435/postgres"; // {dbUrl}/{schema}
            String username = "postgres";
            String password = "1234";

            Class.forName("org.postgresql.Driver");

            Connection connection = DriverManager.getConnection(url, username, password);

            Statement statement = connection.createStatement();

            ResultSet resultSet = statement.executeQuery(query); //Enter query
            while (resultSet.next()) {
                results.add(resultSet.getString( columnIndex: 1));
            }

        } catch (Exception e) {
            System.out.println(e);
        }
        return results;
    }
}
```

Frontend:

For the frontend, we used Java Swings library to implement the graphic-user interface of the project. We have one window which is divided into 2 parts as in Fig 1. One part of the window is responsible for the data about population in Thailand and the other part deals with the estimation of population.

Fig 1



It contains 2 combo boxes, one for year (Fig 2.1) and the other for region (Fig 2.2). This allows the users to select the year and region they desire to see from our database.

Fig 2.1

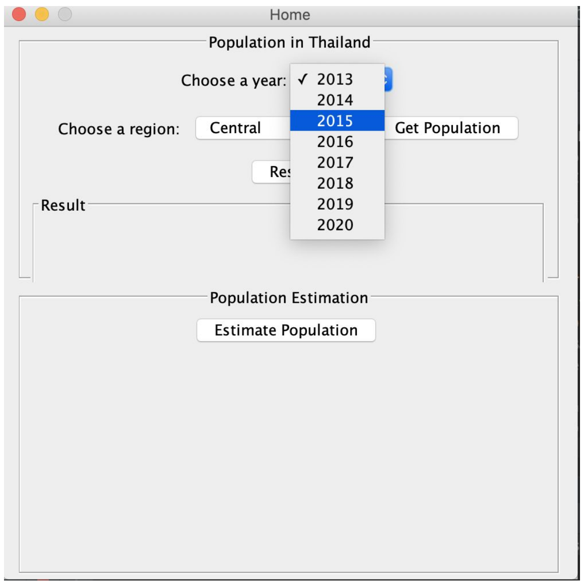
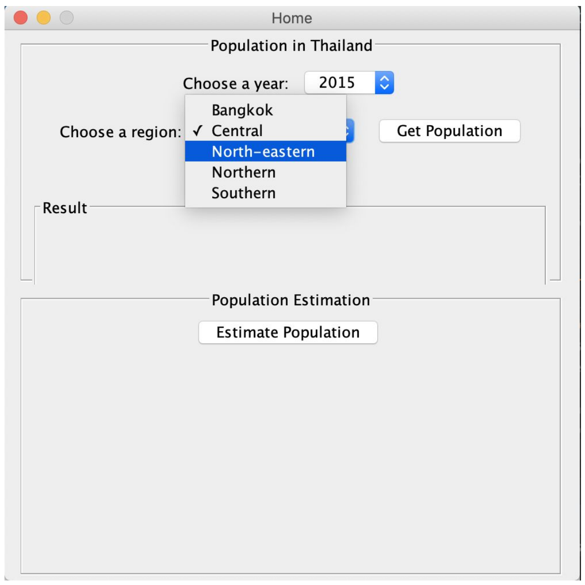
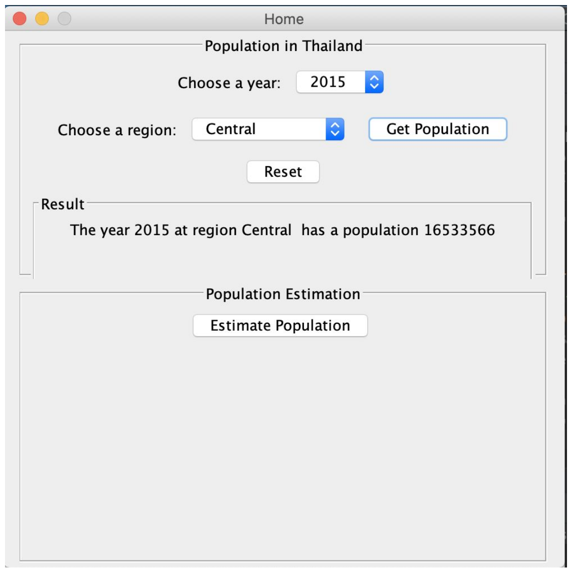


Fig 2.2



Then there are 2 buttons, one for fetching the data and the other for resetting this part of the window back to its original state.The get population button will record the data chosen from the combo boxes, add those data into the query, get the data from the database using the query (that we will explain in the query section below) and display onto a label below the buttons(as in Fig 3). The other button, reset button, does what it is labelled as, reset the combo boxes and labels back to the state it was when the window first opened.

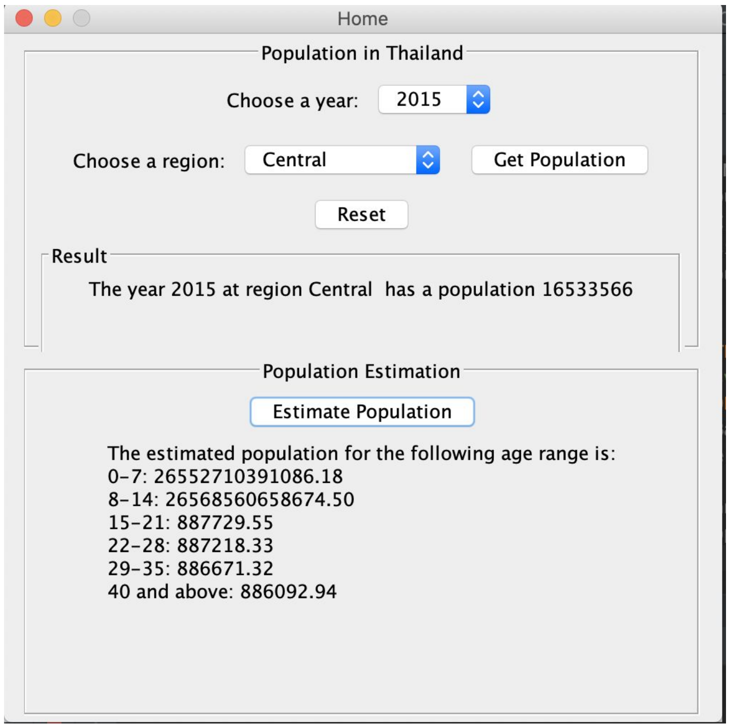
Fig 3



The year 2015 at region Central has a population 16533566

The bottom part of the window is responsible for the data about population estimation. Unlike the previous part, there is a minimum interaction between the user and the interface. The user is able to only press the button to estimate the population and the results will be printed onto the label below the button. What the button does is it calls a function to access the database and execute our premade queries to get the necessary data to estimate. The data is then thrown into the function for calculating and estimating the population. Once the estimation is done, the results are given back to the frontend side where the results are iterated and printed out for users to see.

Fig 4



Choose a year: 2015

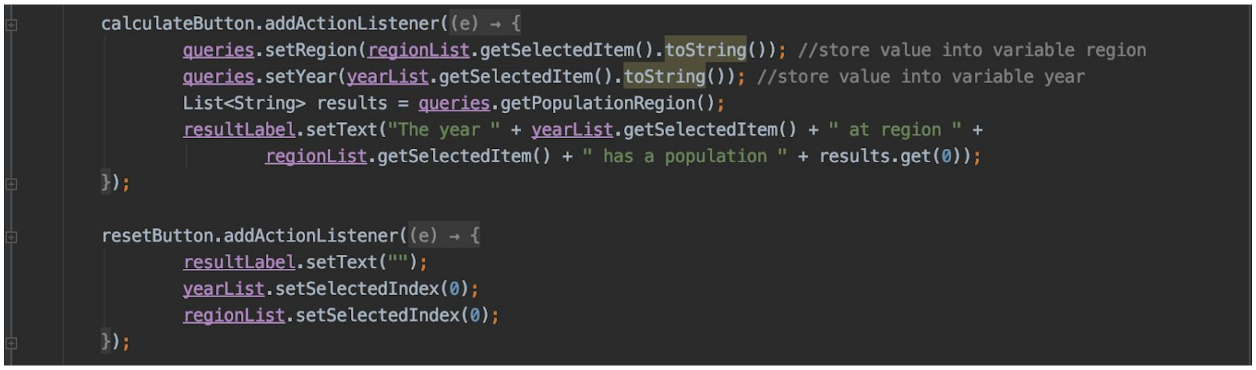
Choose a region: Central

```
The year 2015 at region Central  has a population 16533566
```

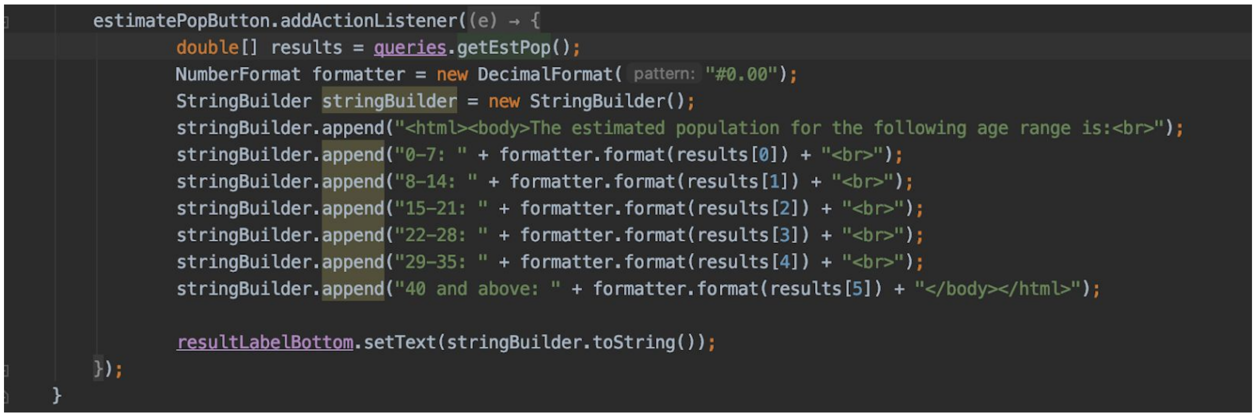
```
The estimated population for the following age range is:
0-7: 26552710391086.18
8-14: 26568560658674.50
15-21: 887729.55
22-28: 887218.33
29-35: 886671.32
40 and above: 886092.94
```

The fig 5.1 and fig 5.2 shows the Java GUI code that calls our sql queries to get values from the database and print them out in the frontend.

Fig 5.1



```java
    calculateButton.addActionListener((e) → {
            queries.setRegion(regionList.getSelectedItem().toString()); //store value into variable region
            queries.setYear(yearList.getSelectedItem().toString()); //store value into variable year
            List<String> results = queries.getPopulationRegion();
            resultLabel.setText("The year " + yearList.getSelectedItem() + " at region " +
                    regionList.getSelectedItem() + " has a population " + results.get(0));
    });

    resetButton.addActionListener((e) → {
            resultLabel.setText("");
            yearList.setSelectedIndex(0);
            regionList.setSelectedIndex(0);
    });
```



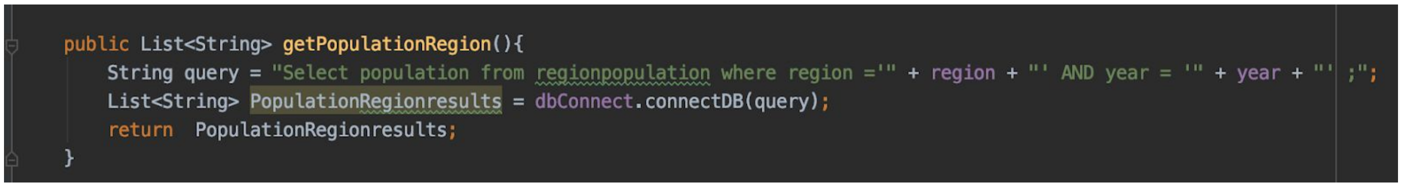
```java
        estimatePopButton.addActionListener((e) → {
                double[] results = queries.getEstPop();
                NumberFormat formatter = new DecimalFormat( pattern: "#0.00");
                StringBuilder stringBuilder = new StringBuilder();
                stringBuilder.append("<html><body>The estimated population for the following age range is:<br>");
                stringBuilder.append("0-7: " + formatter.format(results[0]) + "<br>");
                stringBuilder.append("8-14: " + formatter.format(results[1]) + "<br>");
                stringBuilder.append("15-21: " + formatter.format(results[2]) + "<br>");
                stringBuilder.append("22-28: " + formatter.format(results[3]) + "<br>");
                stringBuilder.append("29-35: " + formatter.format(results[4]) + "<br>");
                stringBuilder.append("40 and above: " + formatter.format(results[5]) + "</body></html>");

                resultLabelBottom.setText(stringBuilder.toString());
        });
    }
```

Queries:

More details can be found in PopulationEstimation/PopCal/src/main/java/GUI/queries.java
In the github repo.
For the queries, we have mainly three queries in total.

1) getPopulationRegion: current population of chosen year and region
2) getEstPop: offspring and people survive

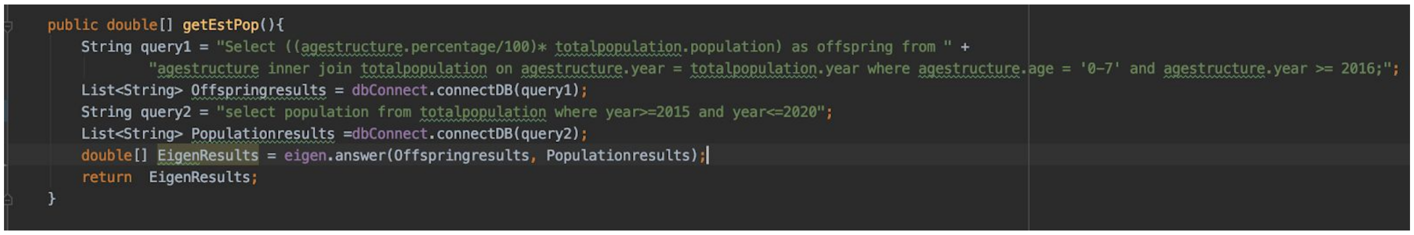
In the getPopulation(), we get the year and region from the user and use them in the WHERE condition to retrieve the data from the database and display them back to the frontend.



```java
    public List<String> getPopulationRegion(){
        String query = "Select population from regionpopulation where region ='" + region + "' AND year = '" + year + "' ;";
        List<String> PopulationRegionresults = dbConnect.connectDB(query);
        return  PopulationRegionresults;
    }
```

getEstPop() is the most important part of our project as it correlates to our Eigenvector algorithm. In this function, there are two queries, the first query or query1 will get the number of offspring from year 2016-2020. However, our table agestruction contains the percentage only so we have to divide by 100 and multiply with the population in the table totalpopulation. For that

exact reason, we use INNER JOIN to join the two tables and connect them with the column YEAR with the WHERE condition below to get our offspring list.
For query2, we will get the population from year 2015 to 2020. The reason we take an extra year here is because in our estimation matrix we need to get the population in the previous year subtract with the current year and use the result in our matrix or double[] to manipulate the estimation later. Since we need 5 values to put in the matrix, we have chosen a year range of 6 to fulfill that condition.

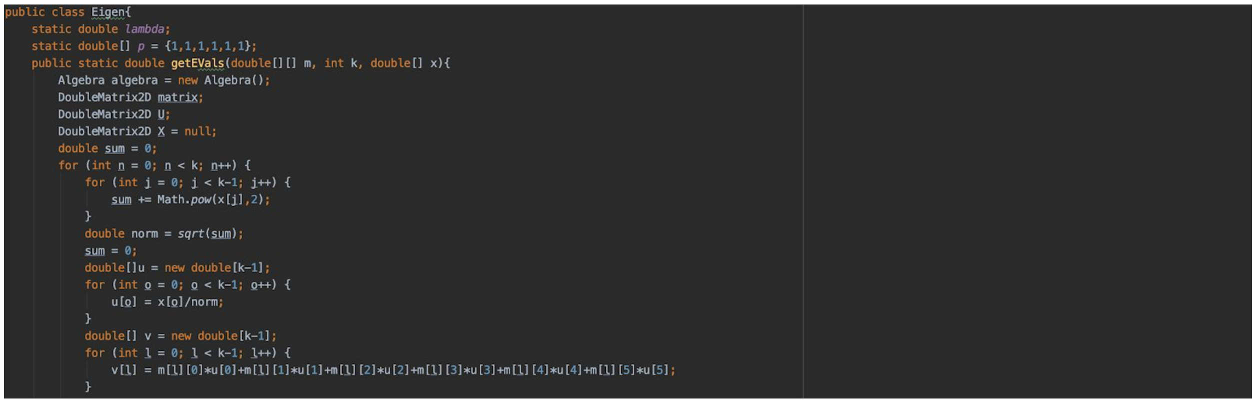


```java
public double[] getEstPop(){
    String query1 = "Select ((agestructure.percentage/100)* totalpopulation.population) as offspring from " +
            "agestructure inner join totalpopulation on agestructure.year = totalpopulation.year where agestructure.age = '0-7' and agestructure.year >= 2016;";
    List<String> Offspringresults = dbConnect.connectDB(query1);
    String query2 = "select population from totalpopulation where year>=2015 and year<=2020";
    List<String> Populationresults =dbConnect.connectDB(query2);
    double[] EigenResults = eigen.answer(Offspringresults, Populationresults);
    return  EigenResults;
}
```

Backend:

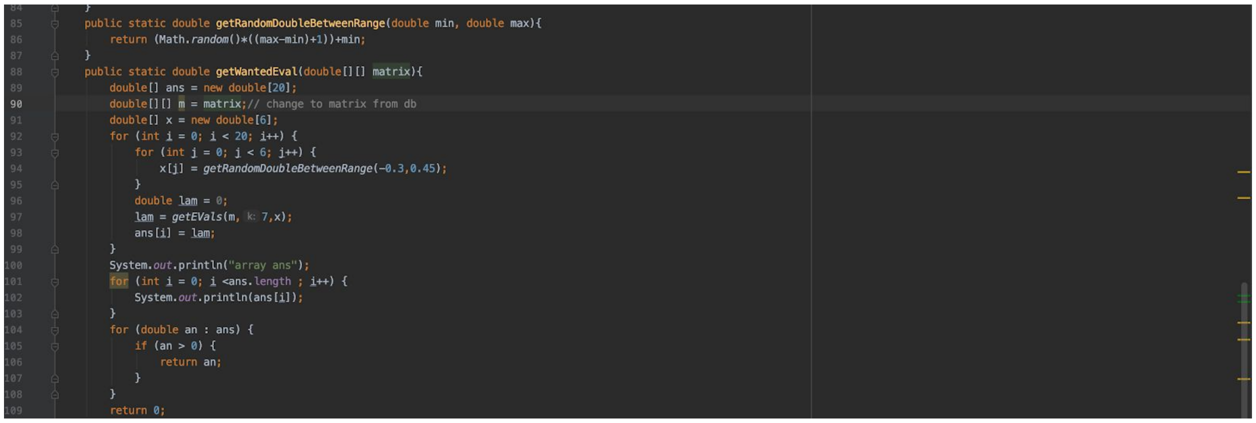
More details can be found on PopulationEstimation/PopCal/src/main/java/Math/Eigen.java in the github repo.

For the computation we split each part into various functions. The first function is the main function that is used to compute all possible eigenvalues, and it is called getEVals. This function involves using Rayleigh quotient and inverse power iteration. The input of this function are a matrix from our database, an integer k for looping, and a matrix x which is a 6*1 matrix. This contains random generated numbers given in the next function. First we normalize a matrix, then we start Rayleigh quotient iteration, where we dot product the original matrix with our normalized matrix. Then we dot product this matrix with x. After that we do inverse power iteration. We continue doing this until we get maximum eigenvalue.



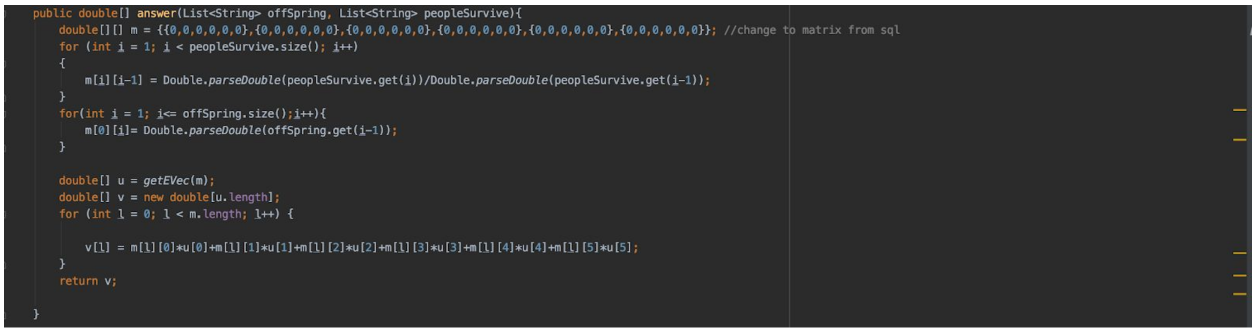
```java
public class Eigen{
    static double lambda;
    static double[] p = {1,1,1,1,1,1};
    public static double getEVals(double[][] m, int k, double[] x){
        Algebra algebra = new Algebra();
        DoubleMatrix2D matrix;
        DoubleMatrix2D U;
        DoubleMatrix2D X = null;
        double sum = 0;
        for (int n = 0; n < k; n++) {
            for (int j = 0; j < k-1; j++) {
                sum += Math.pow(x[j],2);
            }
            double norm = sqrt(sum);
            sum = 0;
            double[]u = new double[k-1];
            for (int o = 0; o < k-1; o++) {
                u[o] = x[o]/norm;
            }
            double[] v = new double[k-1];
            for (int l = 0; l < k-1; l++) {
                v[l] = m[l][0]*u[0]+m[l][1]*u[1]+m[l][2]*u[2]+m[l][3]*u[3]+m[l][4]*u[4]+m[l][5]*u[5];
            }
```

The second function is getRandomDoubleBetweenRange, this is a simple function. It is a random number generator.
The next function is used to find all possible eigenvalues. This function is called getWantedEval. For this project we only need one positive eigenvalue. So we loop getEVal 20 times. Each time we generate a new matrix x and store the eigenvalues in an array. This is to make sure that we would get at least one positive eigenvalue. Then we return the first positive one.



```java
    }
    public static double getRandomDoubleBetweenRange(double min, double max){
        return (Math.random()*((max-min)+1))+min;
    }
    public static double getWantedEval(double[][] matrix){
        double[] ans = new double[20];
        double[][] m = matrix;// change to matrix from db
        double[] x = new double[6];
        for (int i = 0; i < 20; i++) {
            for (int j = 0; j < 6; j++) {
                x[j] = getRandomDoubleBetweenRange(-0.3,0.45);
            }
            double lam = 0;
            lam = getEVals(m, k: 7,x);
            ans[i] = lam;
        }
        System.out.println("array ans");
        for (int i = 0; i <ans.length ; i++) {
            System.out.println(ans[i]);
        }
        for (double an : ans) {
            if (an > 0) {
                return an;
            }
        }
        return 0;
```

The population matrix here is formed from the list we got from our queries execution and put in the function as an argument. From the list, we change it into the Double[] type so that we can multiply our population matrix with our eigenvector to get the result. This result is then shown in the frontend.



```java
public double[] answer(List<String> offSpring, List<String> peopleSurvive){
    double[][] m = {{0,0,0,0,0,0},{0,0,0,0,0,0},{0,0,0,0,0,0},{0,0,0,0,0,0},{0,0,0,0,0,0},{0,0,0,0,0,0}}; //change to matrix from sql
    for (int i = 1; i < peopleSurvive.size(); i++)
    {
        m[i][i-1] = Double.parseDouble(peopleSurvive.get(i))/Double.parseDouble(peopleSurvive.get(i-1));
    }
    for(int i = 1; i<= offSpring.size();i++){
        m[0][i]= Double.parseDouble(offSpring.get(i-1));
    }

    double[] u = getEVec(m);
    double[] v = new double[u.length];
    for (int l = 0; l < m.length; l++) {

        v[l] = m[l][0]*u[0]+m[l][1]*u[1]+m[l][2]*u[2]+m[l][3]*u[3]+m[l][4]*u[4]+m[l][5]*u[5];
    }
    return v;

}
```

Resources:

Population Characteristics 2005 - 2006. (n.d.). Retrieved from http://web.nso.go.th/

Child and Youth. (n.d.). Retrieved from http://web.nso.go.th/

Population Changes 2005 - 2006. (n.d.). Retrieved from http://web.nso.go.th/

Population in Thailand. (n.d).

Retrieved from https://www.boi.go.th/index.php?page=demographic

-------------------------------------End of DEMO and Report---

DELETE FROM world WHERE virus_name = ‘COVID-19’;

TAKE CARE and STAY SAFE ka Ajarn :)