Assignment Report

$\boldsymbol{y' = \cos(x) - y}$; $\begin{cases} y(1) = 1 \\ x \in [1;\ 9.5] \end{cases}$ given first order, linear, nonhomogeneous DE

$\boldsymbol{y' + y = \cos(x)}$ – rewrite equation in this form

Now let's solve the complementary equation:

$\boldsymbol{y' + y = 0}$, $\boldsymbol{\frac{y'}{y} = -1}$, $\boldsymbol{\ln|y| = -x}$

$\boldsymbol{y = C * e^{-x} \Rightarrow y' = C' * e^{-x} - C * e^{-x}}$

$\boldsymbol{C' * e^{-x} - C * e^{-x} + C * e^{-x} = \cos(x)}$

$\boldsymbol{C' * e^{-x} = \cos(x) \Rightarrow C' = e^{x} * \cos(x)}$; $\boldsymbol{C = \int e^{x} * \cos(x)\, dx}$

$\boldsymbol{\int e^{x} * \cos(x)\, dx}$ We will integrate by parts twice in a row: $\int udv = uv - \int vdu$

$\boldsymbol{1)\ v = \cos(x)\ , du = e^{x} \quad dv = -\sin(x)\, dx, \quad u = e^{x}}$

$First\ step$: $\boldsymbol{\int e^{x} * \cos(x)\, dx = e^{x} * \cos(x) - \int -e^{x} * \sin(x)\, dx}$

$\boldsymbol{2)\ v = -\sin(x)\ , du = e^{x} \quad dv = -\cos(x)\, dx, \quad u = e^{x}}$

$Second\ step$: $\boldsymbol{\int e^{x} * \cos(x)\, dx}$

$$= \boldsymbol{e^{x} * \cos(x) - (-e^{x} * \sin(x) - \int -e^{x} * \cos(x) dx)}$$

The integral $\boldsymbol{\int e^{x} * \cos(x)\, dx}$ appears again on the right side of the equation, we can solve for it:

$$\boldsymbol{\int e^{x} * \cos(x)\, dx = \frac{e^{x}(\sin(x) + \cos(x)}{2} + C}$$

Thus: $\boldsymbol{y = \frac{(\sin(x) + \cos(x))}{2} + e^{-x} * C}$

Calculation method is base(main) class for calculation methods (Exact, Euler, Improved Eules, Runge-Kutta). It has methods: *getError()* and *getMaxError()* which are the same for all calculation methods.

Also it has abstract method *getCalculation()*, is the method that every calculation method should realize by itself.

CalculationMethod.java

```java
package sample;

import javafx.scene.chart.XYChart.Data;
import javafx.scene.chart.XYChart.Series;


public abstract class CalculationMethod {

    protected Function function;
    protected Variables vars;

    CalculationMethod(Variables variables){
        this.vars = variables;
        this.function = new MyFunction();
    }

    public abstract Series getCalculation();

    public Series getError(CalculationMethod otherMethod){
        Series series = new Series();
        Series currentMethodSeries = this.getCalculation();
        Series otherMethodSeries = otherMethod.getCalculation();
        series.setName(currentMethodSeries.getName());

        for(int i = 0; i < currentMethodSeries.getData().size(); i++){
            Data<Double, Double> currentData = (Data<Double, Double>) currentMethodSeries.getData().get(i);
            Data<Double, Double> otherData = (Data<Double, Double>) otherMethodSeries.getData().get(i);
            double difference = Math.abs(currentData.getYValue() - otherData.getYValue());
            series.getData().add(new Data<>(currentData.getXValue(), difference));
        }

        return series;
    }

    public Series getMaxError(CalculationMethod otherMethod){
        Series series = new Series();
        series.setName(this.getCalculation().getName());

        double oldValueOfN = vars.getN();
        for(double i = vars.getN0(); i <= vars.getnN(); i++){
            vars.setN(i);
            Series error = getError(otherMethod);
            double max_error = Double.MIN_VALUE;
            for(int j = 0; j < error.getData().size(); j++){
                Data<Double, Double> currentData = (Data<Double, Double>) error.getData().get(j);
                max_error = Math.max(max_error, currentData.getYValue());
            }
            series.getData().add(new Data<>(i, max_error));
        }

        vars.setN(oldValueOfN);
        return series;
    }
}
```

This class is **Euler method** which extends(Inherits) from Calculation Method(as it was mentioned above). Inside the class there is method *getCalculation(),* which is calculations manipulation for each methods(Exact, Euler, Improved Eules, Runge-Kutta) uniquely. In this case is calculations for **Euler** method.

Euler.java

```java
package sample;

import javafx.scene.chart.XYChart.Series;
import javafx.scene.chart.XYChart.Data;

public class Euler extends CalculationMethod {
    Euler(Variables variables) {
        super(variables);
    }

    @Override
    public Series getCalculation() {
        Series series = new Series();
        series.setName("Euler");

        double y = vars.getY0();
        double lastValue;
        double step = (vars.getX() - vars.getX0()) / vars.getN();

        for(int i = 0; i <= vars.getN(); i++){
            double xi = vars.getX0() + i * step;
            series.getData().add(new Data<>(xi, y));
            lastValue = function.getFunctionValue(xi, y);
            y = y + step * lastValue;
        }

        return series;
    }
}
```

This class is **Improved Euler method** which extends(Inherits) from Calculation Method(as it was mentioned above). Inside the class there is method *getCalculation(),* which is calculations manipulation for each methods(Exact, Euler, Improved Eules, Runge-Kutta) uniquely. In this case is calculations for **Improved Euler** method.

ImprovedEuler.java

```java
package sample;

import javafx.scene.chart.XYChart.Series;
import javafx.scene.chart.XYChart.Data;

public class ImprovedEuler extends CalculationMethod {
    ImprovedEuler(Variables variables) {
        super(variables);
    }

    @Override
    public Series getCalculation() {
        Series series = new Series();
        series.setName("Improved Euler");

        double y = vars.getY0();
        double lastValue = function.getFunctionValue(vars.getX0(), vars.getY0());
        double xInCalc, yInCalc;
        double step = (vars.getX() - vars.getX0()) / vars.getN();

        for(int i = 0; i <= vars.getN(); i++){
            double xi = vars.getX0() + i * step;
            series.getData().add(new Data<>(xi, y));
            xInCalc = xi + step / 2.0;
            yInCalc = y + (step / 2.0) * lastValue;
            y = y + step * function.getFunctionValue(xInCalc, yInCalc);
            lastValue = function.getFunctionValue(xi + step, y);
        }

        return series;
    }
}
```

This class is **Runge-Kutta method** which extends(Inherits) from Calculation Method (as it was mentioned above). Inside the class there is method *getCalculation(),* which is calculations manipulation for each methods(Exact, Euler, Improved Eules, Runge-Kutta) uniquely. In this case is calculations for **Runge-Kutta** method.

RungeKutta.java

```java
package sample;

import javafx.scene.chart.XYChart.Series;
import javafx.scene.chart.XYChart.Data;

public class RungeKutta extends CalculationMethod {
    RungeKutta(Variables variables) {
        super(variables);
    }

    @Override
    public Series getCalculation() {
        Series series = new Series();
        series.setName("Runge-Kutta");

        double y = vars.getY0();
        double lastValue = function.getFunctionValue(vars.getX0(), vars.getY0());
        double step = (vars.getX() - vars.getX0()) / vars.getN();

        for(int i = 0; i <= vars.getN(); i++){
            double xi = vars.getX0() + i * step;
            series.getData().add(new Data<>(xi, y));
            double k1 = lastValue;
            double k2 = function.getFunctionValue(xi + step / 2.0, y + (step * k1) / 2.0);
            double k3 = function.getFunctionValue(xi + step / 2.0, y + (step * k2) / 2.0);
            double k4 = function.getFunctionValue(xi + step, y + step * k3);
            y = y + step / 6.0 * (k1 + 2.0 * k2 + 2.0 * k3 + k4);
            lastValue = function.getFunctionValue(xi + step, y);
        }

        return series;
    }
}
```

This class is **Exact method** which extends(Inherits) from Calculation Method(as it was mentioned above). Inside the class there is method *getCalculation(),* which is calculations manipulation for each methods(Exact, Euler, Improved Eules, Runge-Kutta) uniquely. In this case is calculations for **Exact** method.

Exact.java

```java
package sample;

import javafx.scene.chart.XYChart.Series;
import javafx.scene.chart.XYChart.Data;

public class Exact extends CalculationMethod {
    Exact(Variables vars) {
        super(vars);
    }

    @Override
    public Series getCalculation() {
        Series series = new Series();
        series.setName("Exact");
        double y = vars.getY0();
        double step = (vars.getX() - vars.getX0()) / vars.getN();
        double c = function.getCoefficient(vars.getX0(), vars.getY0());

        for(int i = 0; i <= vars.getN(); i++){
            double xi = vars.getX0() + i * step;
            series.getData().add(new Data<>(xi, y));
            y = function.getSolvedFunctionValue(c, xi + step, y);
        }

        return series;
    }
}
```

This class is for storing, changing and getting value of variables. Where

x_0, y_0 - initial conditions. $x \in [x_0, X]$ – domain of function. N - number of grid cells. $[n_0, n_N]$ - given range for Maximum error.

Note: As you can see, initially I already set those values to given numbers. So, in the first run of the program you don't have to write them. But if you want to check correctness you can change them.

Variables.java

```
1  package sample;
2
3  public class Variables {
4
5      private double N;
6      private double x0;
7      private double y0;
8      private double X;
9      private double n0;
10     private double nN;
11
12     Variables(){
13         setN(99);
14         setX0(1);
15         setY0(1);
16         setX(9.5);
17         setN0(10);
18         setnN(29);
19     }
20     public double getX0() {
21         return x0;
22     }
23     public void setX0(double x0) {
24         this.x0 = x0;
25     }
26     public double getY0() {
27         return y0;
28     }
29     public void setY0(double y0) {
30         this.y0 = y0;
31     }
32     public double getX() {
33         return X;
34     }
35     public void setX(double x) {
36         X = x;
37     }
38     public double getN0() {
39         return n0;
40     }
```

This is **interface** for function and if I want to add new function, I just need to change 2 lines of code and realize its method.

Function.java

```
1  package sample;
2
3  public interface Function {
4
5      public double getFunctionValue(double x, double y);
6      public double getSolvedFunctionValue(double c, double x, double y);
7      public double getCoefficient(double x0, double y0);
8  }
```

This is the class that realize function interface. Line 6 – shows given function (first order DE), Line 10 – solved function in exact form. Line 15 – constant C expressed in terms of y and x.

MyFunction.java

```
1  package sample;
2
3  public class MyFunction implements Function{
4      @Override
5      public double getFunctionValue(double x, double y) {
6          return Math.cos(x) - y;
7      }
8      @Override
9      public double getSolvedFunctionValue(double c, double x, double y) {
10         double expression = c * Math.pow(Math.E, -x) + (Math.sin(x)+Math.cos(x))/2;
11         return expression;
12     }
13     @Override
14     public double getCoefficient(double x0, double y0){
15         return (y0 - (Math.sin(x0)+Math.cos(x0))/2)*Math.pow(Math.E, x0) ;
16     }
17 }
```

This method for handling errors. In my function y should be more than 0, also fields cannot be empty, they should contain only numbers, number of steps should be more than 0.

n_0 cannot be more than n_N. Also I add limitation to x_0, because otherwise computer can not compute it values and program will crash.

Limitations: $x \in [1; 9.5]$; $y \in [1; 10^5]$; X < 10; $N \in [0; 100]$;

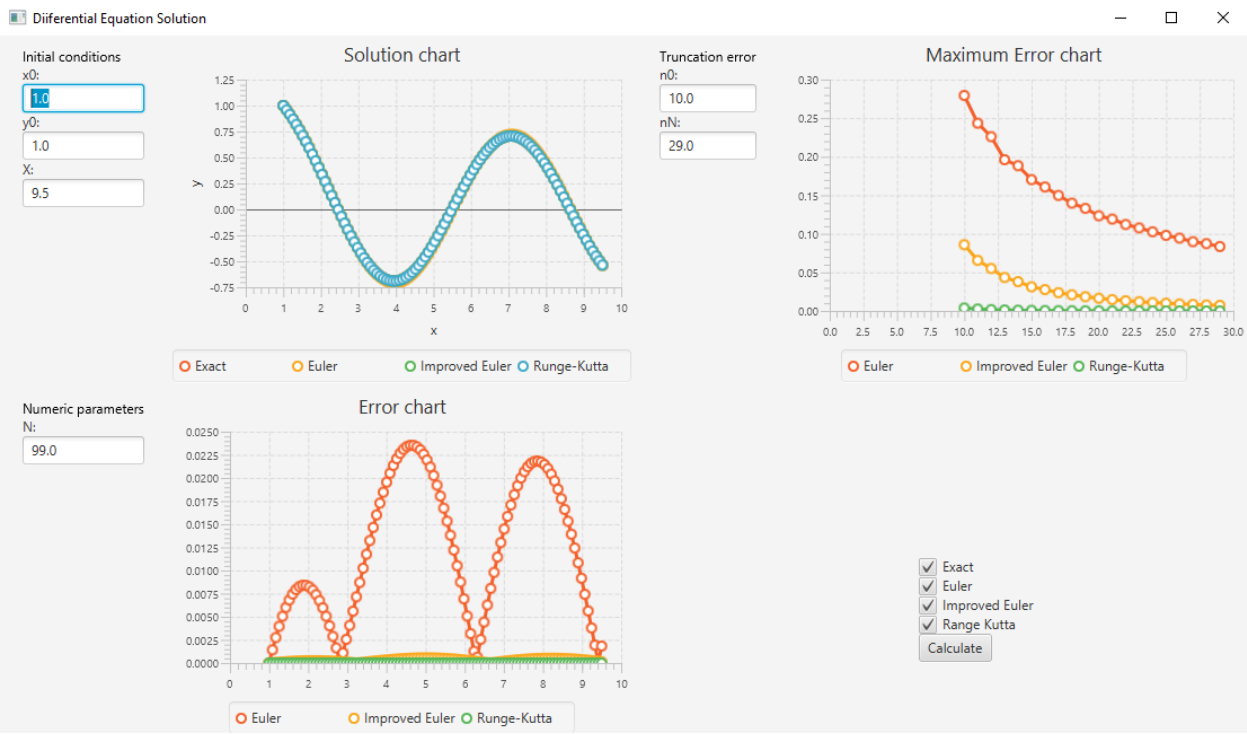
for n_0 and n_N: (n_N - n_0) < 100.

```
private boolean handleErrors() {
    Alert alert = new Alert(Alert.AlertType.ERROR);
    alert.setTitle("Error");
    String s = "";
    String textFields[] = new String[] {x0.getText(), y0.getText(), x.getText(), n.getText(), n0.getText(), nN.getText()};

    if(isEmpty(textFields)){
        s = "Fields should not be empty";
    } else if(!isNumbers(textFields)){
        s = "Fields should doubles";
    } else if(Double.valueOf(x0.getText()) > Double.valueOf(x.getText())){
        s = "x0 can not be bigger than X";
    } else if(Double.valueOf(x.getText()) > 10){
        s = "X can not be bigger than 15";
    } else if(Double.valueOf(y0.getText()) < 0){
        s = "y0 can not be lower than 0";
    } else if(Double.valueOf(y0.getText()) > 100000){
        s = "y0 can not be bigger than 100000";
    } else if(Double.valueOf(x0.getText()) > 10){
        s = "x0 can not be bigger than 8";
    } else if(Double.valueOf(x0.getText()) < 0){
        s = "x0 can not be lower than -12";
    } else if(Double.valueOf(n.getText()) < 0){
        s = "N (number of steps) can not be lower than 0";
    } else if(Double.valueOf(n0.getText()) > Double.valueOf(nN.getText())){
        s = "n0 can not be more than nN";
    } else if(Double.valueOf(nN.getText()) - Double.valueOf(n0.getText()) > 100){
        s = "Difference between n0 and nN can not be more than 100";
    } else if(Double.valueOf(n.getText()) > 1000){
        s = "N can not be more than 1000";
    }

    if(s.length() > 0){
        alert.setContentText(s);
        alert.showAndWait();
        return true;
    } else
        return false;
}
```

To finalize whole done work, below I will provide some screenshots of working interface(GUI).



Conclusion: All requirements of computational practicum (which was given in moodle) was done in the right way.