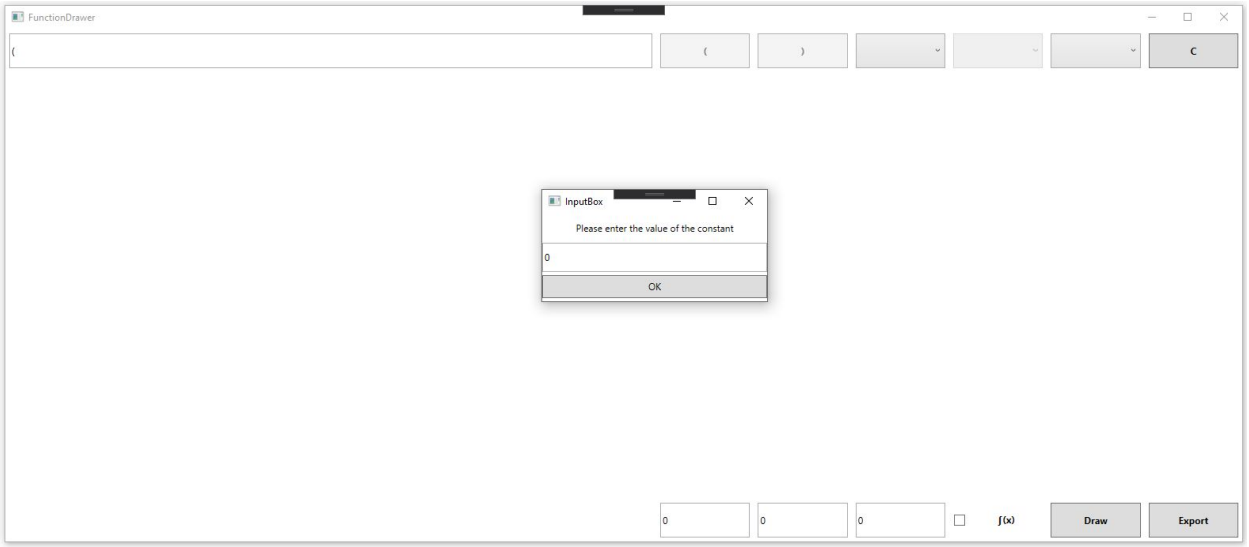
FunctionPlotter

I. Functional Documentation

This application is made for the purpose of visualizing the graph of a function in real time. The function can be inputted, through a series of dropboxes. The images below show how you can input a certain composite function by selecting the appropriate entries in the dropdowns.

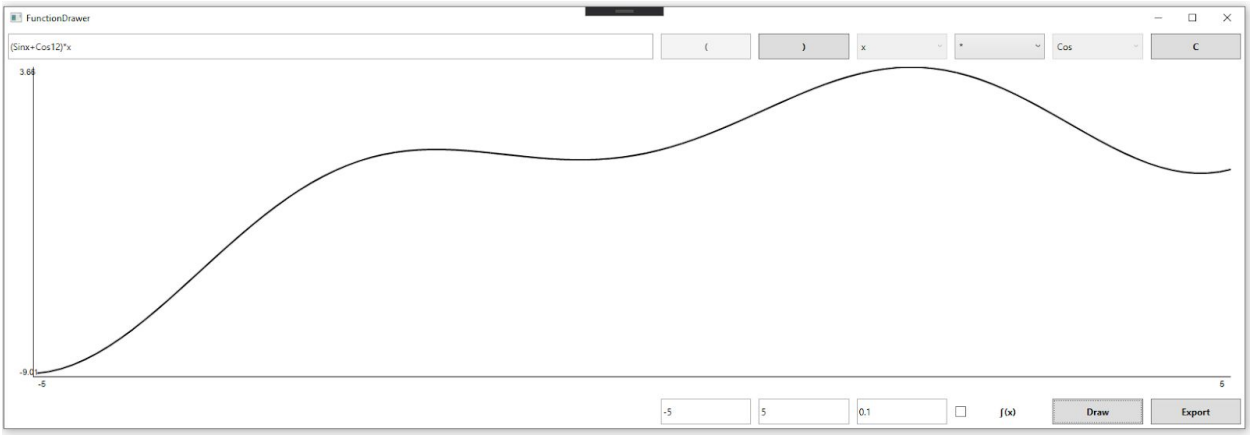
The dropdown on the far right contains the available simple functions.

In this pop-up you can select and insert a constant.

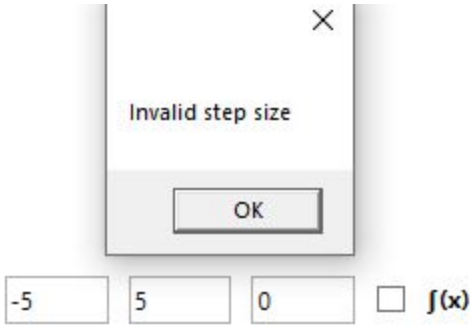


Here you can select one of the five operators.

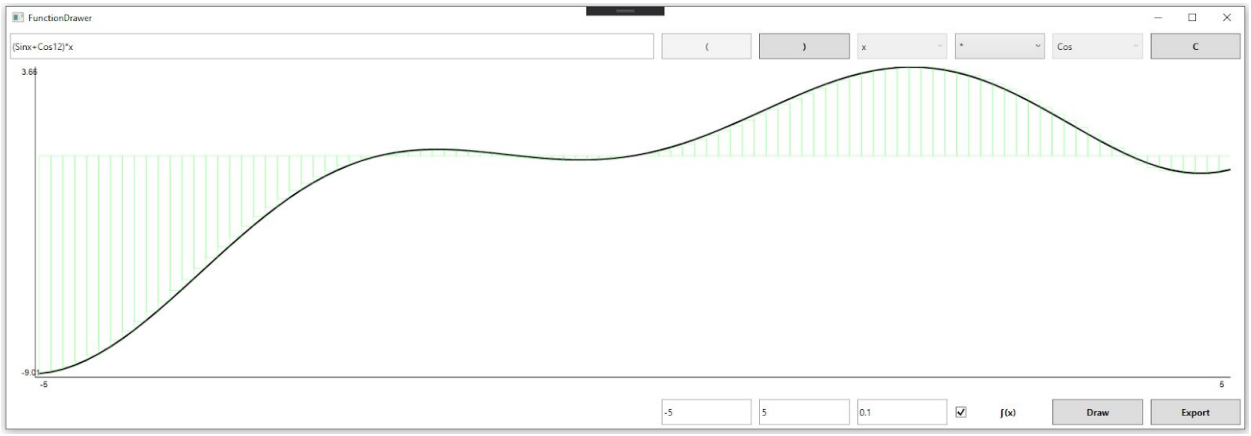
In order to draw the function you need to specify a range of definition and a step size. In this case you can see the function is drawn for the [-5,5] and the step size of 0.1



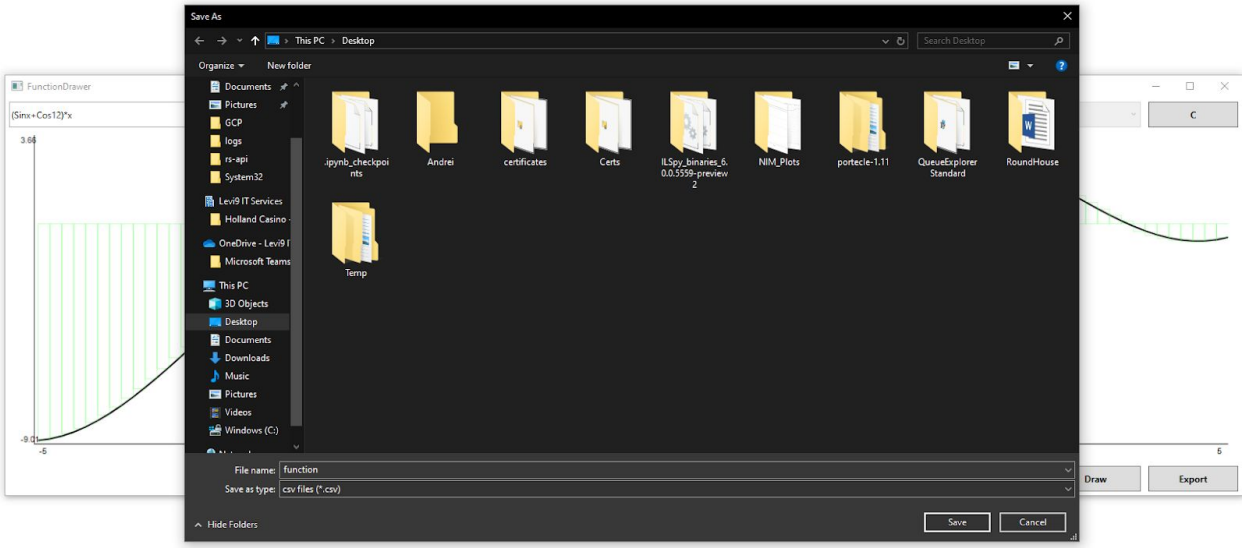
Be aware that we have some validations on the text input fields that do not allow for a negative or zero step size. Also it won't allow an invalid function range. That means that the right most end of the interval must be strictly greater than the lower right.

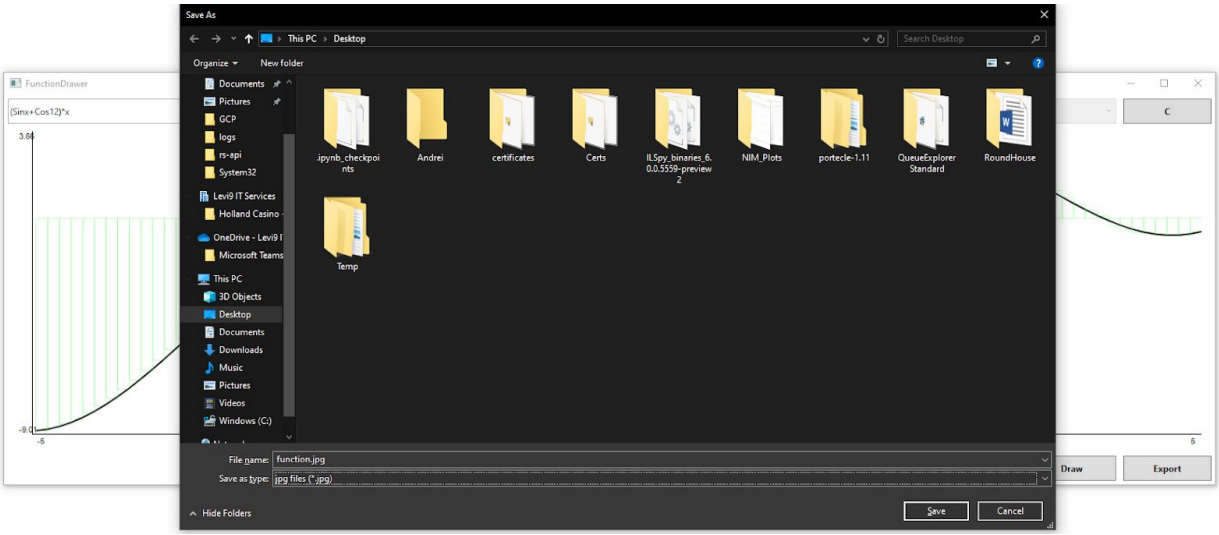


If you want to draw the integral of a function you can check the box near the draw button.



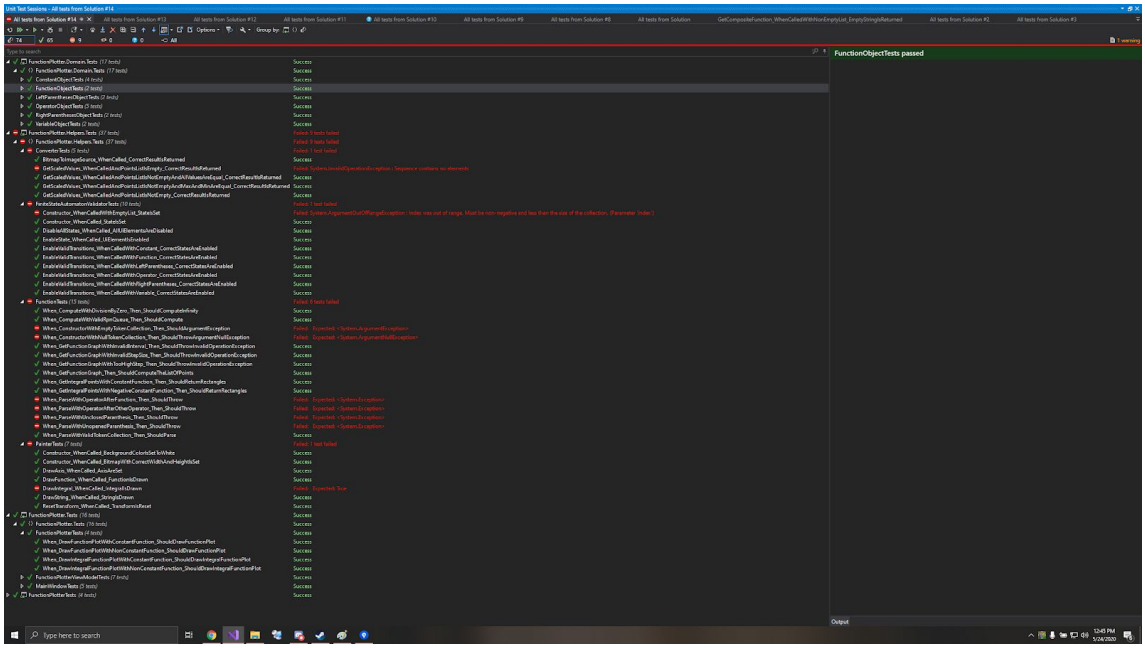
The function graph can be saved by pressing the ‘Export’ button. The user will be prompted with a File Chooser Dialog. He must provide a filename for the export. He can select to export as a csv file, in which case pairs of X and Y’s will be written in the output file, or as a jpg|png|jpeg file, in which case the image that is currently shown in the main window is saved. Below you can see the two cases.



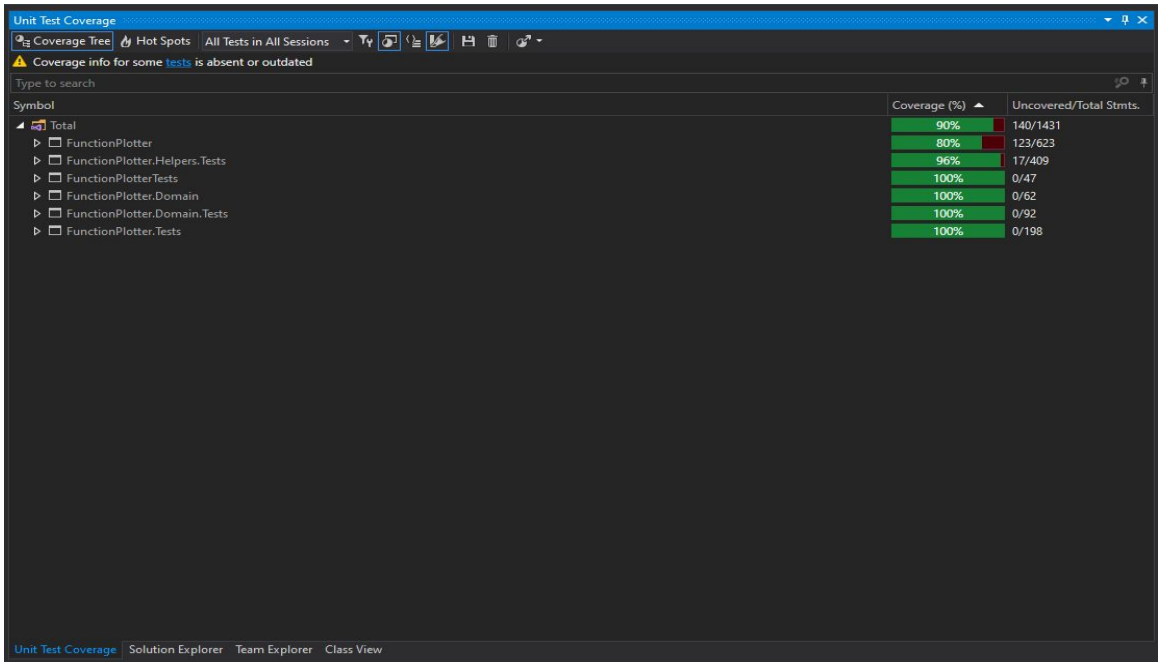


II. Unit Testing

The image below showcases the units done for the whole application. As you can see we have a fair amount of tests. As you can see we have some tests that fail in certain conditions.



We have an accurate 90% line coverage with some missing lines of code because of how the WPF technology works.



All the tests were designed using the AAA (Arange, Act, Assert) pattern, common for writing unit tests for a method under test. Below we can see two examples of such tests.

```
[Test]
0 references | Loghin Alexandru, 13 days ago | 1 author, 1 change
public void When_DrawFunctionPlotWithConstantFunction_ShouldDrawFunctionPlot()
{
    //Arrange
    const int width = 800;
    const int height = 600;

    var yValue:double = Math.Round(1d, 2);
    var maxY = (yValue + _functionPlotterViewModel.MaxY).ToString(CultureInfo.InvariantCulture);
    var minY = (yValue + _functionPlotterViewModel.MinY).ToString(CultureInfo.InvariantCulture);

    var functionPoints = new List<PointF>
    {
        new PointF(x:-1, y:1),
        new PointF(x:0, y:1),
        new PointF(x:1, y:1)
    };
    var integralRectangles = new List<(PointF, PointF)>
    {
        (new PointF(x:-1, y:1), new PointF(x:0, y:0)),
        (new PointF(x:0, y:1), new PointF(x:1, y:0))
    };

    _painterMock.Setup(expression:x:IPainter => x.DrawAxis(originOffset:It.IsAny<int>()));
    _painterMock.Setup(expression:x:IPainter => x.DrawFunction(points:It.IsAny<List<PointF>>()));
    _painterMock.Setup(expression:x:IPainter => x.ResetTransform());
    _painterMock.Setup(expression:x:IPainter => x.DrawString(point:It.IsAny<PointF>(), input:It.IsAny<string>(), size:It.IsAny<int>()));
    _
        .Setup(expression:x:IFunction => x.GetFunctionGraph(low:It.IsAny<double>(), high:It.IsAny<double>(), stepSize:It.IsAny<double>()))// ISetup<IFunction,List<...>>
        .Returns(functionPoints);

    //Act
    _functionPlotter.DrawFunctionPlot(width, height);

    //Assert
    const int twice = 2;

    _painterMock.Verify(expression:x:IPainter => x.DrawAxis(originOffset:It.IsAny<int>()), Times.Once);
    _painterMock.Verify(expression:x:IPainter => x.DrawFunction(points:It.IsAny<List<PointF>>()), Times.Once);
    _painterMock.Verify(expression:x:IPainter => x.ResetTransform(), Times.Exactly(twice));
    _painterMock.Verify(expression:x:IPainter => x.DrawString(point:It.IsAny<PointF>(), minY, size:It.IsAny<int>()), Times.Once);
    _painterMock.Verify(expression:x:IPainter => x.DrawString(point:It.IsAny<PointF>(), maxY, size:It.IsAny<int>()), Times.Once);
}
```

(field) Mock<IPainter> FunctionPlotterTests._painterMock

In the above example we test the method that draws the plot for the function. In the *Arrange* section we set the desired behaviour for the dependent instances that are used by the tested method, via mocking approach. The *Act* contains the actual call of the method. The *Assert*

section assures that the dependencies methods are used correctly by the method from the system under test.

```
[Test]
0 references | Strugari, Stefan, 14 days ago | 1 author, 1 change
public void When_ComputeWithValidRpnQueue_Then_ShouldCompute()
{
    // Arrange
    var tokens = new List<GraphObject> { exp, left, sin, x, add, cos, pi, right, sub, pi, pow, zero };
    var xValue = 5;

    var function = new Function(tokens);

    //  Act
    var yValue:double = function.Compute(xValue);

    // Assert
    Assert.AreEqual(expected:Math.Exp(Math.Sin(xValue) + Math.Cos(Math.PI)) - Math.Pow(Math.PI, 0), actual:yValue);
}
```

Here we test if the function computes the corresponding Y value for a fixed X value. We arrange the tokens and instantiate the function representation, we compute the actual value and finally we verify that the actual value matches with the expected.

III. Assertions

For this case of assertion I try to make sure that when doing a transition, the state is not null since this would break the application.

```
12 references | Loghin Alexandru, 6 days ago | 1 author, 6 changes
public sealed class FiniteStateAutomatonValidator : IFiniteStateAutomatonValidator
{
    // Predefined Order = (),var,op,func
    private readonly List<Control> _states;
    private GraphObject _currentState;

    11 references | Loghin Alexandru, 6 days ago | 1 author, 2 changes
    public FiniteStateAutomatonValidator(List<Control> states)
    {
        Ensure.Arg(states).IsNotNull();

        _states = states;
        EnableValidTransitions();
    }

    12 references | Loghin Alexandru, 6 days ago | 1 author, 2 changes
    public void DoTransition(GraphObject nextState)
    {
        Ensure.Arg(nextState).IsNotNull();

        _currentState = nextState;

        Ensure.Arg(_currentState == nextState);

        EnableValidTransitions();
    }
```

```
16 references | Strugari, Stefan, 27 days ago | 2 authors, 3 changes
public List<PointF> GetFunctionGraph(double low, double high, double stepSize)
{
    if (stepSize <= 0)
        throw new InvalidOperationException( message: "Invalid step size");

    if (low >= high)
        throw new InvalidOperationException( message: "Invalid interval");

    if (high - low < stepSize)
        throw new InvalidOperationException( message: "Too high step size");

    var points = new List<PointF>();
    for (var x :double = low; x <= high; x += stepSize)
    {
        points.Add( item: new PointF((float) x, y: (float) Compute(x)));
    }

    return points;
}
```

In the *GetFunctionGraph* method we have some preconditions for the input parameters:

- the step size should be a positive number;
- the low end of the interval should always have a lower value that the high end of the interval;
- the step size should be lower than the size of the interval.

```
8 references | Stefan Strugari, 5 days ago | 3 authors, 3 changes
public Queue<GraphObject> Parse(ICollection<GraphObject> graphObjects)
{
    Ensure.Arg(graphObjects).IsNotNullOrEmpty();

    Queue<GraphObject> outputQueue = new Queue<GraphObject>();
    Stack<GraphObject> operatorStack = new Stack<GraphObject>();

    foreach (var graphObject in graphObjects)
    {
        if (graphObject.GraphObjectType == GraphObjectType.Constant ||
            graphObject.GraphObjectType == GraphObjectType.Variable)
        {
            outputQueue.Enqueue(graphObject);
        }

        if (graphObject.GraphObjectType == GraphObjectType.Function)
        {
            operatorStack.Push(graphObject as FunctionObject);
        }

        if (graphObject.GraphObjectType == GraphObjectType.Operator)
        {
            while (operatorStack.Count != 0)
            {
                var stackTop:GraphObject = operatorStack.Peek();

                if (stackTop.GraphObjectType != GraphObjectType.Function &&
                    (stackTop.GraphObjectType != GraphObjectType.Operator ||
                     ((OperatorObject) stackTop).GetPrecedence() <
                     ((OperatorObject) graphObject).GetPrecedence()) ||
                    stackTop.GraphObjectType == GraphObjectType.LeftParentheses)
                    break;
                outputQueue.Enqueue(item:operatorStack.Pop());
            }

            operatorStack.Push(graphObject);
        }

        if (graphObject.GraphObjectType == GraphObjectType.LeftParentheses)
        {
            operatorStack.Push(graphObject);
        }

        if (graphObject.GraphObjectType == GraphObjectType.RightParentheses)
        {
            while (operatorStack.Count != 0)
            {
                var stackTop:GraphObject = operatorStack.Pop();
                if (stackTop.GraphObjectType == GraphObjectType.LeftParentheses)
                    break;
                outputQueue.Enqueue(stackTop);
            }
        }
    }

    while (operatorStack.Count != 0)
    {
        outputQueue.Enqueue(item:operatorStack.Pop());
    }

    Ensure.Arg(operatorStack.Count == 0);
    Ensure.Arg(outputQueue.All(predicate:op:GraphObject => !(op is RightParenthesesObject) && !(op is LeftParenthesesObject)));

    return outputQueue;
}
```

The above method transforms the collection of tokens from the infix order into reverse polish notation. RPN keeps track of the precedence of the operators and functions from the expression without the need for tokens of type left parenthesis or right parenthesis, so at the end of the transformation we expect to not have any of these delimiter operators in the output. Another postcondition for this method is represented by the condition that the operator stack should be empty.

```
8 references | Strugari, Stefan, 26 days ago | 2 authors, 4 changes
public List<(PointF, PointF)> GetIntegralPoints(List<PointF> functionPoints)
{
    Ensure.Arg(functionPoints).IsNotNullOrEmpty();
    Ensure.Arg(functionPoints.Count > 1);

    var points = new List<(PointF, PointF)>();

    var pointsY:List<float> = functionPoints.Select(entry:PointF => entry.Y).ToList();
    var minY:float = pointsY.Min();
    var maxY:float = pointsY.Max();

    if (Math.Abs(minY - maxY) < 0.001)
    {
        if (minY > 0)
            minY = 0;

        if (maxY < 0)
            maxY = 0;
    }

    for (var i = 0; i < functionPoints.Count - 1; i += 1)
    {
        PointF upperLeftPoint;
        PointF lowerRightPoint;

        if (functionPoints[i].Y > 0)
        {
            upperLeftPoint = functionPoints[i];
            lowerRightPoint = new PointF(functionPoints[i + 1].X, y:Math.Max(0, minY));
        }
        else
        {
            upperLeftPoint = new PointF(functionPoints[i].X, y:Math.Min(0, maxY));
            lowerRightPoint = functionPoints[i + 1];
        }

        Ensure.Arg(upperLeftPoint.X < lowerRightPoint.X);
        Ensure.Arg(upperLeftPoint.Y > lowerRightPoint.Y);

        points.Add((upperLeftPoint, lowerRightPoint));
    }

    Ensure.Arg(points.Count == functionPoints.Count - 1);

    return points;
}
```

The above method determines the rectangles that are used for the integral approximation. The input should be the function graph like points in the cartesian space, computed in a previous step, and the output are the rectangles, each of them represented by a pair of points upper left corner of the rectangle and lower right corner of the rectangle.
We expect that the client of this method would provide a collection of points that have at least two points in order to return at least one rectangle for the integral.

We can all see a loop variant. At each iteration we determine the current rectangle and always know that the upper left corner point will always have a lower X coordinate and higher Y coordinate than the lower left corner point.
At the end we guarantee to return n - 1 rectangles where n is the number of points from the input.

Contribution:

- UI/UX - Loghin Alexandru
- Graphics/Painter - Loghin Alexandru
- Project Structure - Loghin Alexandru
- Token Parser - Stefan Strugari
- Behaviour for computing the function values - Stefan Strugari
- Behaviour for determining the integral rectangles - Stefan Strugari

each for all phases of the project.