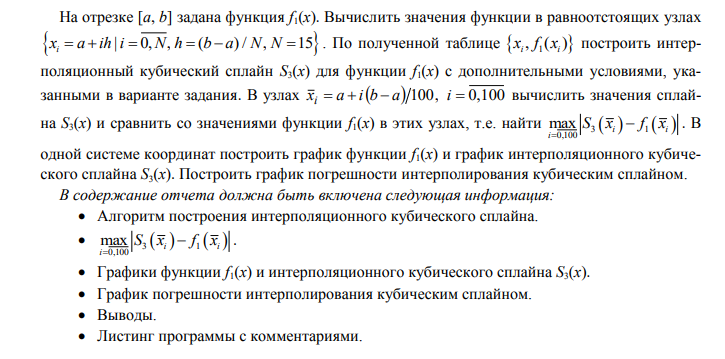
Отчет по дисциплине: «Численные методы»

Лабораторная работа №2

## «Интерполяционный кубический сплайн»

Подготовил студент 3 курса 4 группы

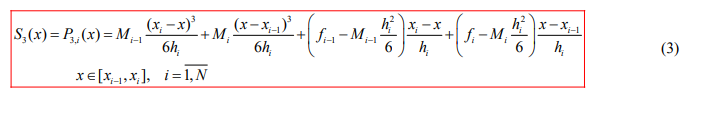
Кондратович Артём

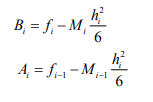


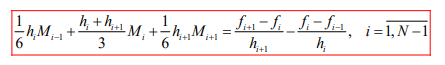






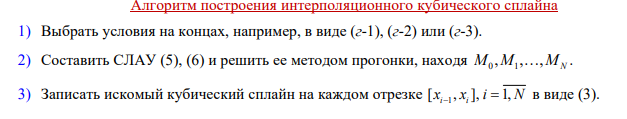






домноженный на -1





using lab\_2\_2.HelpClasses;

using OxyPlot.Series;

using OxyPlot;

using OxyPlot.Legends;

namespace lab\_2\_2

{

public partial class Form1 : Form

{

private readonly Func<double, double> \_f1\_0;

private readonly Func<double, double> \_f1\_1;

private readonly Func<double, double> \_f1\_2;

private readonly Segment \_segment;

public Form1()

{

InitializeComponent();

\_f1\_0 = (x) => Math.Cos(x) \* Math.Sin(x);

\_f1\_1 = (x) => Math.Cos(2 \* x);

\_f1\_2 = (x) => -2 \* Math.Sin(2 \* x);

\_segment = new(-3, 3);

}

private void BuildCubicSpline(int n)

{

var x = \_segment.GetEquidistantNodes(n);

var nodes = \_segment.GetEquidistantNodes(100);

var data\_1 = new Data(x, \_f1\_0);

var h = x[1] - x[0];

var f = new List<double>()

{

(\_f1\_0(x[1]) - \_f1\_0(x[0])) / h - \_f1\_1(x[0])

};

for (var i = 1; i < n; i++)

{

f.Add((data\_1.Values[i + 1] - data\_1.Values[i]) / h - (data\_1.Values[i] - data\_1.Values[i - 1]) / h);

}

f.Add(\_f1\_2(x[n]));

var m = SweepMethod(h, f, n);

var spline = GetSpline(m, data\_1, h, n);

var partitions = x.GetPartitions();

var splineValues = nodes

.Select(x => spline[GetIndexOfSpline(partitions, x)](x))

.ToList();

var functionValues = nodes

.Select(x => \_f1\_0(x))

.ToList();

var errorValues = new List<double>();

for (var i = 0; i < 101; i++)

{

errorValues.Add(Math.Abs(splineValues[i] - functionValues[i]));

}

var maxError = errorValues.Max();

label1.Text = $"max|f(x)-s(x)| = {maxError}";

var functionSeries = GetLineSeries(1);

var splineSeries = GetLineSeries(2);

var errorSeries = GetLineSeries(3);

for (var i = 0; i < 101; i++)

{

functionSeries.Points.Add(new DataPoint(nodes[i], functionValues[i]));

splineSeries.Points.Add(new DataPoint(nodes[i], splineValues[i]));

errorSeries.Points.Add(new DataPoint(nodes[i], errorValues[i]));

}

var firstModel = new PlotModel { Title = "" };

firstModel.Legends.Add(new Legend()

{

LegendTitle = "",

LegendPosition = LegendPosition.RightTop

});

firstModel.Series.Add(functionSeries);

firstModel.Series.Add(splineSeries);

plotView1.Model = firstModel;

var secondModel = new PlotModel { Title = "" };

secondModel.Legends.Add(new Legend()

{

LegendTitle = "",

LegendPosition = LegendPosition.RightTop

});

secondModel.Series.Add(errorSeries);

plotView2.Model = secondModel;

}

private static List<double> SweepMethod(double h, List<double> f, int n)

{

var a = new List<double>()

{ 0.0 };

for (var i = 1; i < n; i++)

{

a.Add(h / 6);

}

a.Add(0);

var b = new List<double>();

for (var i = 0; i < n; i++)

{

b.Add(h / 6);

}

b.Add(0);

var c = new List<double>

{

h / 3

};

for (var i = 0; i < n - 1; i++)

{

c.Add(2 \* h / 3);

}

c.Add(1);

return GetMoments(a, b, c, f, n);

}

private static List<double> GetAlpha(List<double> a, List<double> b, List<double> c, int n)

{

var alpha = new List<double>

{

0,

b[0] / c[0]

};

for (var i = 1; i < n; i++)

{

alpha.Add(b[i] / (c[i] - a[i] \* alpha[i]));

}

return alpha;

}

private static List<double> GetBeta(List<double> a, List<double> c, List<double> f, List<double> alpha, int n)

{

var beta = new List<double>

{

0,

f[0] / c[0]

};

for (var i = 1; i < n; i++)

{

beta.Add((f[i] + a[i] \* beta[i - 1]) / (c[i] - a[i] \* alpha[i]));

}

beta.Add((f[n] + a[n] \* beta[n]) / (c[n] - a[n] \* alpha[n]));

return beta;

}

private static List<double> GetMoments(List<double> a, List<double> b, List<double> c, List<double> f, int n)

{

var alpha = GetAlpha(a, b, c, n);

var beta = GetBeta(a, c, f, alpha, n);

var m = new List<double>();

for (var i = 0; i < n + 1; i++)

{

m.Add(0);

}

m[n] = beta[n + 1];

for (int i = n - 1; i >= 0; i--)

{

m[i] = alpha[i + 1] \* m[i + 1] + beta[i + 1];

}

return m;

}

public static List<Func<double, double>> GetSpline(List<double> m, Data data, double h, int n)

{

var x = data.Points;

var f = data.Values;

var a = GetA(m, f, h, n);

var b = GetB(m, f, h, n);

var spline = new List<Func<double, double>>()

{

(x) => 0

};

for (var i = 1; i <= n; i++)

{

var index = i;

spline.Add((y) => m[index - 1] \* Math.Pow(x[index] - y, 3) / (6 \* h)

+ m[index] \* Math.Pow(y - x[index - 1], 3) / (6 \* h)

+ a[index] \* (x[index] - y) / h

+ b[index] \* (y - x[index - 1]) / h);

}

return spline;

}

public static List<double> GetA(List<double> m, List<double> f, double h, int n)

{

var a = new List<double>()

{

0

};

for (var i = 1; i < n + 1; i++)

{

a.Add(f[i - 1] - m[i - 1] \* (Math.Pow(h, 2) / 6));

}

return a;

}

public static List<double> GetB(List<double> m, List<double> f, double h, int n)

{

var b = new List<double>()

{

0

};

for (var i = 1; i < n + 1; i++)

{

b.Add(f[i] - m[i] \* (Math.Pow(h, 2) / 6));

}

return b;

}

public static int GetIndexOfSpline(List<Segment> segments, double point)

{

for (var i = 0; i < segments.Count; i++)

{

if (segments[i].Belongs(point))

{

return i;

}

}

return -1;

}

private void Button1\_Click(object sender, EventArgs e)

{

BuildCubicSpline(15);

}

private static LineSeries GetLineSeries(int type)

{

var name = type switch

{

1 => $"f(x)",

2 => $"S(x)",

3 => $"Error",

\_ => throw new NotSupportedException()

};

var color = type switch

{

1 => OxyColors.Blue,

2 => OxyColors.Red,

3 => OxyColors.Green,

\_ => throw new NotImplementedException(),

};

var series = new LineSeries()

{

Title = name,

Color = color,

StrokeThickness = 1,

MarkerSize = 2,

MarkerType = MarkerType.Circle

};

return series;

}

private void label1\_Click(object sender, EventArgs e)

{

}

}

}

Data.cs

public class Data

{

public List<double> Points { get; set; }

public List<double> Values { get; set; }

public Data(List<double> points, Func<double, double> func)

{

Points = points;

Values = points

.Select(x => func(x))

.ToList();

}

}

Extensions.cs

public static class Extensions

{

public static List<double> GetEquidistantNodes(this Segment segment, int n)

{

var h = segment.Length / n;

var nodes = new List<double>();

for (var i = 0; i <= n; i++)

{

nodes.Add(segment.A + i \* h);

}

return nodes;

}

public static List<double> GetLengthOfThePartitionSegements(this List<double> partition)

{

var lengths = new List<double>();

for(var i = 1;i < partition.Count;i++)

{

lengths.Add(partition[i] - partition[i - 1]);

}

return lengths;

}

public static List<Segment> GetPartitions(this List<double> x)

{

var partitions = new List<Segment>()

{

new(-10,-9)

};

for (var i = 1; i < x.Count; i++)

{

partitions.Add(new(x[i - 1], x[i]));

}

return partitions;

}

}

Segment.cs

public class Segment

{

public double A { get; set; }

public double B { get; set; }

public double Length

{

get => B - A;

}

public Func<double, bool> Belongs

{

get => (x) => x >= A && x <= B;

}

public Segment(double a, double b) {

if (a >= b)

{

throw new InvalidOperationException();

}

A = a;

B = b;

}

}

