

```

1  package hyperDap.generator.main;
2
3  import java.util.ArrayList;
4  import java.util.Random;
5  import java.util.function.DoubleFunction;
6  import java.util.function.Function;
7  import hyperDap.base.helpers.Comparator;
8  import hyperDap.base.types.dataSet.ValueDataSet;
9
10 /**
11  * This class generates a section of data specific to one function.
12  * <p>
13  * The intended use is to initialise a {@code GenSegment} with the intended values
14  * and then retrieve
15  * lists of data points from {@link #generateValues(double, int)} as needed, before
16  * leaving the
17  * Object to be garbage-collected. calling data generation methods repeatedly with
18  * the same
19  * parameters will produce the same data points within limits given to any
20  * randomness (this will be
21  * added later to simulate noise).
22  * <p>
23  * Each Object represents a function of the format {@code a * Func(x + b) + c},
24  * where {@code Func}
25  * is a mathematical function specified by {@code functionEncoding} at construction
26  * time. The
27  * function is shifted such that it passes through the {@code intercept} at {@code
28  * x= -step}, which
29  * is expected to be the last value before this function begins. This aligns it with
30  * the previous
31  * values to transition smoothly.
32  * <p>
33  * The {@code functionEncoding} may specify: <br>
34  * {@code constant} : {@code y = c} <br>
35  * {@code linear} : {@code y = a * (x + b) + c} <br>
36  * {@code square} : {@code y = a * (x + b)^2 + c} <br>
37  * {@code cubic} : {@code y = a * (x + b)^3 + c} <br>
38  * {@code exp} : {@code y = a * Math.E^(x + b) + c} <br>
39  * {@code sine} : {@code y = a * sin(x + b) + c} <br>
40  * <p>
41  * Here {@code a} translates to the {@code scale} specified at construction, {@code
42  * b} to
43  * {@code shiftX}, while {@code c} is defined at construction such that the function
44  * returns {@code intercept} for {@code x=-step}.
45  *
46  * @author soenk
47  *
48  */
49 public class GenSegment {
50
51     private double step;
52     private double a;
53     private double b;
54     private double c;
55     private Function<Double, Double> func;
56     private Random rand = new Random();
57
58     /**
59     * The default constructor.
60     *
61     * @param functionEncoding A {@link String} encoding of the function to be
62     * generated.
63     * @param scale Used to scale and make the function more or less 'steep' {@code =>
64     * a}
65     * @param shiftX Shifts the function right or left. Use to fit split up functions
66     * together (e.g.
67     * bias) {@code => b}
68     * @param intercept Used to ensure the first value is in line with previous values
69     * and is assumed
70     * to be the last value before this segment. {@code = f(-step) + c}
71     *
72     */
73     public GenSegment(String functionEncoding, double scale, double shiftX, double

```

```

intercept,
61     double step) throws IllegalArgumentException {
62     this.step = step;
63     a = scale;
64     b = shiftX;
65     c = 0;
66     this.defineFunction(functionEncoding);
67     c = intercept - f(-step);
68     System.out.println(String.format("%s Generating Segment of %s with a= %s, b= %s
        c= %s",
69         GenSegment.class, functionEncoding, this.a, this.b, this.c));
70 }
71
72 /**
73  * Classifies the function represented by this Object based on {@code encoding.}
74  *
75  * @param encoding A {@link String} encoding of the function that is to be modelled.
76  */
77 private void defineFunction(String encoding) throws IllegalArgumentException {
78     encoding = encoding.toLowerCase();
79     switch (encoding) {
80         case "constant":
81             func = x -> 0.0;
82             break;
83         case "linear":
84             func = x -> x;
85             break;
86         case "square":
87             func = x -> Math.pow(x, 2);
88             break;
89         case "cubic":
90             func = x -> Math.pow(x, 3);
91             break;
92         case "exp":
93             func = x -> Math.pow(Math.E, x);
94             a = a / 1000;
95             break;
96         case "sine":
97             func = x -> Math.sin(x);
98             break;
99         default:
100             throw new IllegalArgumentException(
101                 String.format("'%'s' is not a valid function encoding!", encoding));
102     }
103 }
104
105 /**
106  * Returns a single value of the function specified in this Object.
107  * <p>
108  * This is specified as {@code a* Function(x + b) +c}.
109  *
110  * @param x The {@code xValue} to be fed into the function.
111  * @return The {@code yValue} corresponding to {@code x}.
112  */
113 private double f(double x) {
114     return a * this.func.apply(x + b) + c;
115 }
116
117 /**
118  * Returns a single value of the function specified by this Object with added
119  * noise. Noise here is
120  * a value added to {@link #f(double)}, that is randomly taken from a normal
121  * distribution around
122  * zero and of standard deviation {@code noise}.
123  *
124  * @param x The {@code xValue} that is passed to {@link #f(double)}
125  * @param noise The {@code standard deviation} of the value added (or subtracted).
126  * @return {@link #f(double) f(x)} {@code + noise *} {@link Random#nextGaussian()}.
127  */
128 private double noisyF(double x, double noise) {
129     return f(x) + noise * rand.nextGaussian();
130 }

```

```

130  /**
131  * Provides a means to reseed the internal instance of {@link Random}.
132  * <p>
133  * This may not mean that generated data can be used for machine learning, unless
134  * it is generated
135  * in small enough segments.
136  *
137  * @param seed Passed to {@link Random#setSeed(long)}
138  */
139  public void seedRandom(long seed) {
140      this.rand.setSeed(seed);
141  }
142
143  /**
144  * Encapsulation of {@link #generateValues(int, double)} with {@code noise=0}.
145  *
146  * @param N The number of data points to be generated.
147  * @return An {@link ArrayList} of the generated data points.
148  */
149  public ArrayList<Double> generateValues(int N) {
150      return this.generateValues(N, 0.0);
151  }
152
153  /**
154  * Generate a list of data points of length {@code N}, according to pre-set
155  * specifications and
156  * with the set amount of noise.
157  *
158  * @param N The number of data points to be generated.
159  * @param noise The noise factor passed to {@link #noisyF(double, double)}
160  * @return An {@link ArrayList} of the generated data points.
161  */
162  public ArrayList<Double> generateValues(int N, double noise) {
163      ArrayList<Double> list = new ArrayList<Double>();
164      for (Integer i = 0; i < N; i++) {
165          list.add(noisyF(i.doubleValue() * step, noise));
166      }
167      return list;
168  }
169
170  /**
171  * Generate the specified data points and add them to the end of {@code set}.
172  * <p>
173  * Calls {@link ValueDataSet#ensureCapacity(int)} before generating data.
174  * <p>
175  * Encapsulates {@link #addToDoubleDataSet(ValueDataSet, int, double)} with {@code
176  * noise=0}.
177  *
178  * @param set The {@link CalueDataSet} that the data points should be added to.
179  * @param step The distance between data points on the x-axis.
180  * @param N The number of data points that should be added.
181  * @throws IllegalArgumentException If {@link ValueDataSet#getStep()} is not equal
182  * to the pre-set
183  * step.
184  */
185  public void addToDoubleDataSet(ValueDataSet<Double> set, int N) throws
186  IllegalArgumentException {
187      this.addToDoubleDataSet(set, N, 0.0);
188  }
189
190  /**
191  * Generate the specified data points with noise and add them to the end of {@code
192  * set}.
193  * <p>
194  * Calls {@link ValueDataSet#ensureCapacity(int)} before generating data.
195  * <p>
196  * Noisy values are created using {@link #noisyF(double, double)}.
197  *
198  * @param set The {@link CalueDataSet} that the data points should be added to.
199  * @param N The number of data points that should be added.
200  * @param noise The noise factor passed to {@link #noisyF(double, double)}.
201  * @throws IllegalArgumentException If {@link ValueDataSet#getStep()} is not equal
202  * to the pre-set

```

```

196     *         step.
197     */
198     public void addToDoubleDataSet(ValueDataSet<Double> set, int N, double noise)
199         throws IllegalArgumentException {
200         int size = set.size();
201         if (this.step != set.getStep()) {
202             throw new IllegalArgumentException(
203                 String.format("%s. addToDoubleDataSet() does not match preset step! %s!=%s",
204                     GenSegment.class, this.step, set.getStep()));
205         }
206         set.ensureCapacity(N + set.size());
207         double val;
208         for (Integer i = 0; i < N; i++) {
209             val = Double.valueOf(noisyF(i.doubleValue() * step, noise));
210             set.add(val);
211             // if the value is too large or small, consider it invalid
212             // if (Comparator.equalApprox(0.0, val, 10000.0) == false) {
213             //     set.editValidityByIndex((i + size), false);
214             // }
215         }
216     }
217
218     /**
219     * Generate the specified data points and add them to the end of {@code set}.
220     * <p>
221     * This method requires that {@code set} has an assigned {@code convertFromDouble}
222     * {@link DoubleFunction Function} assigned.
223     * <p>
224     * {@link ValueDataSet#ensureCapacity(int)} is called before adding data points.
225     *
226     * @param set The {@link CalueDataSet} that the data points should be added to.
227     * @param N The number of data points that should be added.
228     * @throws IllegalArgumentException If {@link
229     ValueDataSet#hasConversionFunction()} returns
230     *     {@code false}.<br>
231     *     If {@link ValueDataSet#getStep()} is not equal to the pre-set step.
232     */
233     public void addToDataSet(ValueDataSet<? extends Number> set, int N)
234         throws IllegalArgumentException {
235         this.addToDataSet(set, N, 0.0);
236     }
237
238     /**
239     * Generate the specified data points and add them to the end of {@code set}.
240     * <p>
241     * This method requires that {@code set} has an assigned {@code convertFromDouble}
242     * {@link DoubleFunction Function} assigned.
243     * <p>
244     * {@link ValueDataSet#ensureCapacity(int)} is called before adding data points.
245     * Noisy values are
246     * created using {@link #noisyF(double, double)}.
247     *
248     * @param set The {@link CalueDataSet} that the data points should be added to.
249     * @param N The number of data points that should be added.
250     * @param noise The noise factor passed to {@link #noisyF(double, double)}.
251     * @throws IllegalArgumentException If {@link
252     ValueDataSet#hasConversionFunction()} returns
253     *     {@code false}.<br>
254     *     If {@link ValueDataSet#getStep()} is not equal to the pre-set step.
255     */
256     public void addToDataSet(ValueDataSet<? extends Number> set, int N, double noise)
257         throws IllegalArgumentException {
258         int size = set.size();
259         if (set.hasConversionFunction() == false) {
260             throw new IllegalArgumentException(
261                 "ValueDataSet must have a convertFromDouble function defined!");
262         }
263         if (this.step != set.getStep()) {
264             throw new IllegalArgumentException(
265                 String.format("%s. addToDoubleDataSet() does not match preset step! %s!=%s",
266                     GenSegment.class, this.step, set.getStep()));
267         }
268         set.ensureCapacity(N + set.size());

```

```

266     double val;
267     for (Integer i = 0; i < N; i++) {
268         val = noisyF(i.doubleValue() * step, noise);
269         set.add(val);
270         if (Comparator.equalApprox(0.0, val, 10000) == false) {
271             set.editValidityByIndex(i + size, false);
272         }
273     }
274 }
275
276 public void addRandomToDoubleDataSet(ValueDataSet<Double> set, int N)
277     throws IllegalArgumentException {
278     if (this.step != set.getStep()) {
279         throw new IllegalArgumentException(
280             String.format("%s. addToDoubleDataSet() does not match preset step! %s!=%s",
281                 GenSegment.class, this.step, set.getStep()));
282     }
283     set.ensureCapacity(set.size() + N);
284     Integer temp;
285     for (int i = 0; i < N; i++) {
286         temp = (rand.nextInt(10000) - 5000);
287         set.add(temp.doubleValue() / 100);
288     }
289 }
290
291 public void addRandomToDataSet(ValueDataSet<? extends Number> set, int N)
292     throws IllegalArgumentException {
293     if (set.hasConversionFunction() == false) {
294         throw new IllegalArgumentException(
295             "ValueDataSet must have a convertFromDouble function defined!");
296     }
297     if (this.step != set.getStep()) {
298         throw new IllegalArgumentException(
299             String.format("%s. addToDoubleDataSet() does not match preset step! %s!=%s",
300                 GenSegment.class, this.step, set.getStep()));
301     }
302     set.ensureCapacity(set.size() + N);
303     Integer temp;
304     for (int i = 0; i < N; i++) {
305         temp = (rand.nextInt(10000) - 5000);
306         set.add(temp.doubleValue() / 100);
307     }
308 }
309 }
310
311 }
312

```