```java
package hyperDap.base.helpers;

import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;
import java.util.ArrayList;
import hyperDap.base.types.dataSet.ValueDataSet;
import hyperDap.base.types.value.ValuePair;

/**
 * A helper class that allows finding tangents on data points.
 *
 * @author soenk
 *
 */

public final class Tangenter {

  private static double precision = 0.001;
  private static int bigDecimalPrecision = 10;
  private static MathContext standardContext =
      new MathContext(bigDecimalPrecision, RoundingMode.HALF_UP);

  /**
   * Private constructor to prevent implementing this class.
   */
  private Tangenter() {
    throw new AssertionError("No helper class instances for anyone!");
  }

  /**
   * Used to adjust the precision of {@link #tangentApprox(double, double, double)},
   which by
   * default is set to 0.001.
   *
   * @param precision The new precision used.
   */
  public static void setPrecision(double precision) {
    Tangenter.precision = precision;
  }

  /**
   * Gives the value of the precision currently used in
   * {@link #tangentApprox(double, double, double)}.
   *
   * @return The current value of precision.
   */
  public static double getPrecision() {
    return precision;
  }

  public static double tangentProp(double step, double y1, double y2) {
    if (Comparator.equalProportionate(y1, y2, precision)) {
      return 0.0;
    }
    return tangentSimple(step, y1, y2);
  }

  /**
   * An encapsulation of {@link #tangentApprox(double, double, double, double)},
   using the static
   * precision set by {@link #setPrecision(double)}.
   *
   * @param step The difference in {@code xValue} between the values.
   * @param y1 The first of the values with a lower {@code xValue}.
   * @param y2 The second of the values with the higher {@code xValue}
   * @return Zero if {@code y1} and {@code y2} are equal within the set precision,
   the slope of the
   *         tangent between them otherwise.
   */
  public static double tangentApprox(double step, double y1, double y2) {
    return tangentApprox(step, y1, y2, precision);
  }
```

```
71
72      /**
73       * Calculates the slope of the tangent between two points with {@code yValues}
         {@code y1} and
74       * {@code y2} that are a distance {@code step} apart in their {@code xValues}. If
         the two values
75       * are too close, based on {@link Comparator#equalApprox(double, double, double)},
         the slope will
76       * be approximated to zero.
77       *
78       * @param step The difference in {@code xValue} between the values.
79       * @param y1 The first of the values with a lower {@code xValue}.
80       * @param y2 The second of the values with the higher {@code xValue}
81       * @param precision The precision argument passed to
82       *        {@link Comparator#equalApprox(double, double, double)}
83       * @return Zero if {@code y1} and {@code y2} are equal within the set precision,
         the slope of the
84       *         tangent between them otherwise.
85       */
86
87      public static double tangentApprox(double step, double y1, double y2, double
        precision) {
88        if (Comparator.equalApprox(y1, y2, precision)) {
89          return 0.0;
90        }
91        return tangentSimple(step, y1, y2);
92      }
93
94      /**
95       * An exact implementation of {@link #tangentSimple(double, double, double)} to
         calculate the
96       * tangent between two points, making use of {@link BigDecimal}.
97       *
98       * @param step The difference in {@code xValue} between the values.
99       * @param y1 The first of the values with a lower {@code xValue}.
100      * @param y2 The second of the values with the higher {@code xValue}
101      * @return The slope of the tangent between the points.
102      */
103     public static double tangentExact(double step, double y1, double y2) {
104       BigDecimal val1 = new BigDecimal(y1, standardContext);
105       BigDecimal val2 = new BigDecimal(y2, standardContext);
106       val1 = val2.subtract(val1);
107       val2 = new BigDecimal(step, standardContext);
108       val1 = val1.divide(val2, standardContext);
109       return val1.doubleValue();
110     }
111
112     /**
113      * Calculate the tangent between two values, given the distance between them.
114      *
115      * @param step The distance between the two xValues
116      * @param y1 The first value (with a lower xValue)
117      * @param y2 The second value (with the higher xValue)
118      * @return The value of the tangent in this point.
119      */
120     public static double tangentSimple(double step, double y1, double y2) {
121       return (y2 - y1) / step;
122     }
123
124     /**
125      * Calculate the tangent between two x-y data points.
126      *
127      * @param v1
128      * @param v2
129      * @return The value of the tangent between the two points.
130      */
131     public static double tangentSinmple(ValuePair<? extends Number> v1,
132         ValuePair<? extends Number> v2) {
133       double x1 = v1.getX().doubleValue();
134       double x2 = v2.getX().doubleValue();
135       double y1;
136       double y2;
137
```

```
138       if (x1 < x2) {
139         y1 = v1.getY().doubleValue();
140         y2 = v2.getY().doubleValue();
141       } else {
142         y1 = x1;
143         x1 = x2;
144         x2 = y1;
145         y1 = v2.getY().doubleValue();
146         y2 = v1.getY().doubleValue();
147       }
148
149       return Tangenter.tangentSimple(x2 - x1, y1, y2);
150     }
151
152     /**
153      * Calculate the tangent between two x-y data points. The order of points does not
         matter.
154      * <p>
155      * This implementation calculates the derivative data point, that is its return is
         an x-y data
156      * point of the derivative data set of the original data set that {@code v1} and
         {@code v2} belong
157      * to.
158      *
159      * @param v1
160      * @param v2
161      * @return A {@link ValuePair} of type {@link Double} representing the derivative
         data point.
162      */
163     public static ValuePair<Double> tangent(ValuePair<? extends Number> v1,
164         ValuePair<? extends Number> v2) {
165       return new ValuePair<Double>(Double.valueOf(v1.getX().doubleValue()),
166           Tangenter.tangentSinmple(v1, v2));
167     }
168
169     /**
170      * Calls {@link #calcDerivDepth(ValueDataSet, int) calcDerivDepth(ValueDataSet,
         10)}.
171      *
172      * @param dataset
173      * @return
174      */
175     public static ArrayList<Integer> calcDerivDepth(ValueDataSet<? extends Number>
       dataset) {
176       return Tangenter.calcDerivDepth(dataset, 10);
177     }
178
179     /**
180      * Calls {@link #calcDerivDepth(ValueDataSet, int, boolean)
         calcDerivDepth(ValueDataSet, int,
181      * true)}.
182      *
183      * @param dataset
184      * @param maxDepth
185      * @return
186      */
187     public static ArrayList<Integer> calcDerivDepth(ValueDataSet<? extends Number>
       dataset,
188         int maxDepth) {
189       return calcDerivDepth(dataset, maxDepth, true);
190     }
191
192     /**
193      * Calculates and returns the depth of derivative ({@code derivDepth}) for {@code
         dataset}.
194      * <p>
195      * The {@code derivDepth} is the number of times a trace derivative (the tangent
         between two
196      * points, see {@link #tangentSimple(double, double, double)}) is NOT zero. For
         each point this
197      * indicates the degree of the polynomial the data is representing, if it is
         polynomial. If not
198      * the {@code derivDepth} will be assigned {@link Integer#MAX_VALUE} until further
```

```
199       analysis, to
          * represent infinity. This will also be assigned if the derivDepth would be
          larger than
200       * {@code maxDepth - 1}.
201       * <p>
202       * If {@code doInfiniteDepths} is {@code true} any {@link Integer#MAX_VALUE}
          {@code derivDepth}
203       * values will be further analysed and assigned {@code -2} if exponential, {@code
          -3} for
204       * trigonometric and {@code -5} otherwise, with the correct change values of
          {@code -1} also
205       * assigned.
206       *
207       * @param dataset The {@link ValueDataSSet} that is to be analysed.
208       * @param maxDepth The maximum depth to which the derivative should be calculated.
209       *        {@code derivDepth} larger than this will be assigned {@link
          Integer#MAX_VALUE},
210       *        representing infinity.
211       * @param doInfiniteDepths Whether infinite derivDepths should be further analysed
          to exponential,
212       *        trigonometric etc. (={@code true}) or not (={@code false}).
213       * @return An {@link ArrayList ArrayList<Integer>} of the {@code derivDepth} for
          each value of
214       *         {@code dataset}. Note that the last {@code maxDepth} values may be
          inaccurate.
215       */
216      public static ArrayList<Integer> calcDerivDepth(ValueDataSet<? extends Number>
          dataset,
217          int maxDepth, boolean doInfiniteDepths) {
218        int size = dataset.size();
219        ArrayList<Integer> depths = new ArrayList<Integer>(size);
220        double[][] derivs = new double[size][maxDepth];
221        // calculate trace by trace derivatives
222        calcDerivs(derivs, dataset);
223        // count derivDepth
224        countDerivDepths(derivs, depths);
225        // detect and mark points of change
226        detectDepthChanges(derivs, depths);
227        // further analysis
228        if (doInfiniteDepths == true) {
229          Tangenter.checkInfs(dataset, depths, maxDepth);
230        }
231        smoothEndOfDepths(depths, maxDepth);
232        // finished
233        return depths;
234      }
235
236      /**
237       * This method populates the derivative matrix used in {@link
          #calcDerivDepth(ValueDataSet, int)}.
238       *
239       * @category helper
240       *
241       * @param derivs A reference to the initialised derivative matrix.
242       * @param set The {@link ValueDataSet} that is to be analysed.
243       *
244       * @see ValueDataSet#calcDerivDepths()
245       */
246      private static void calcDerivs(double[][] derivs, ValueDataSet<? extends Number>
          set) {
247        int size = derivs.length;
248        int maxDepth = derivs[0].length;
249        double step = set.getStep();
250        double precision = set.getPrecision();
251        int X; // this variable helps ensure that tangents are calculated left to right
          on the x-axis
252        if (step > 0) {
253          X = 0;
254        } else {
255          X = size - 1;
256        }
257        derivs[0][0] = set.getByIndex(X).doubleValue();
258        for (int k = 1; k < size - 1; k++) {
```

```
259         derivs[k][0] = set.getByIndex(Math.abs(X - k)).doubleValue();
260         for (int i = k - 1, j = 1; i >= 0 && j < maxDepth; i--, j++) {
261           derivs[i][j] = tangentApprox(step, derivs[i][j - 1], derivs[i + 1][j - 1],
              precision);
262         }
263       }
264     }
265
266     /**
267      * This method counts the depth of the derivatives in the derivative matrix, used in
268      * {@link #calcDerivDepth(ValueDataSet, int)}.
269      *
270      * @category helper
271      *
272      * @param derivs A matrix of derivatives.
273      * @param depths The recorded {@link ArrayList} of {@code derivDepth} values.
274      *
275      * @see ValueDataSet#calcDerivDepths()
276      */
277     private static void countDerivDepths(double[][] derivs, ArrayList<Integer> depths) {
278       int maxDepth = derivs[0].length;
279       int size = derivs.length;
280       int depth;
281       for (int i = 0; i < size; i++) {
282         depth = Integer.MAX_VALUE;
283         for (int j = 1; j < maxDepth; j++) {
284           if (derivs[i][j] == 0) {
285             depth = j - 1;
286             break;
287           }
288         }
289         depths.add(depth);
290       }
291     }
292
293     /**
294      * This method uses the derivative depths over the derivative matrix, used in
295      * {@link #calcDerivDepth(ValueDataSet, int)}, to detect changes iin the {@link
       ValueDataSet} that
296      * is being analysed.
297      *
298      * @category helper
299      *
300      * @param derivs A matrix of derivatives.
301      * @param depths The recordedd {@link ArrayList} of {@code derivDepth} values.
302      *
303      * @see ValueDataSet#calcDerivDepths()
304      */
305     private static void detectDepthChanges(double[][] derivs, ArrayList<Integer>
       depths) {
306       int maxDepth = derivs[0].length;
307       int size = derivs.length;
308       boolean tracking = false;
309       int depth;
310       int depthTemp = 0;
311       for (int i = 0; i < size; i++) {
312         depth = depths.get(i);
313         if (tracking == true) {
314           if (depth != -1) {
315             depths.set(i, depthTemp);
316           }
317         } else {
318           if (derivs[i][maxDepth - 1] != 0 && depth < maxDepth - 1) {
319             depthTemp = depth;
320             depths.set(i + maxDepth - 2, -1);
321             tracking = true;
322           } else {
323             // tracking = false;
324           }
325         }
326         if (depth == -1) {
327           tracking = false;
328         }
```

```
329       }
330     }
331
332     /**
333      * Used within {@link #calcDerivDepth(ValueDataSet, int, boolean)} to ensure the
         last few values
334      * of {@code derivDepth} are consistent with the remaining ones. This does not
         mean that these are
335      * the true derivDepth values, but the true one cannot be calculated close to the
         end.
336      *
337      * @param depths The recordedd {@link ArrayList} of {@code derivDepth} values.
338      * @param maxDepth The {@code defrivDepth} to which the analysis extends.
339      */
340     private static void smoothEndOfDepths(ArrayList<Integer> depths, int maxDepth) {
341       int firstIndex = depths.size() - maxDepth;
342       if (firstIndex < 0) {
343         firstIndex = 1;
344       }
345       int depth = depths.get(firstIndex - 1);
346       if (depth == -1) {
347         depth = -5;
348       }
349       for (int i = firstIndex; i < depths.size(); i++) {
350         depths.set(i, depth);
351       }
352     }
353
354     /**
355      * Used within {@link #calcDerivDepth(ValueDataSet, int, boolean)} to check for
         {@code dericDepth}
356      * values of {@link Integer#MAX_VALUE} and initiate further analysis on these
         elements.
357      *
358      * @param set The original {@link ValueDataSet}.
359      * @param depths The recordedd {@link ArrayList} of {@code derivDepth} values.
360      * @param maxDepth The {@code defrivDepth} to which the analysis extends.
361      */
362     private static void checkInfs(ValueDataSet<? extends Number> set,
         ArrayList<Integer> depths,
363         int maxDepth) {
364       int size = depths.size();
365       boolean checking = false;
366       int startI = 0;
367       int endI = 0;
368       // check all derivDepths for yet undefined values
369       for (int i = 0; i < size; i++) {
370         if (depths.get(i) == Integer.MAX_VALUE) {
371           depths.set(i, -5); // depth is undefined until we know otherwise
372           if (checking == false) {
373             // begin tracking a this segment for further analysis.
374             startI = i;
375             checking = true;
376           }
377         } else if (checking == true) {
378           // end of segment, stop tracking
379           endI = i;
380           checking = false;
381           if (endI - startI < maxDepth) {
382             // if segment too small no point
383             continue;
384           }
385           // initate further analysis
386           checkForExp(set, depths, startI, endI, maxDepth);
387         }
388       }
389     }
390
391     /**
392      * Used by {@link #checkInfs(ValueDataSet, ArrayList, int)} to check for
         exponential functions, as
393      * the first step in further analysis.
394      *
```

```
395      * @param set The original {@link ValueDataSet}.
396      * @param depths The recordedd {@link ArrayList} of {@code derivDepth} values.
397      * @param startI The {@code index} within {@code set} at which the analysis should
         begin,
398      *        inclusively.
399      * @param endI The {@code index} within {@code set} at which analysis ends,
         exclusively.
400      * @param maxDepth The {@code defrivDepth} to which the analysis extends.
401      */
402     private static void checkForExp(ValueDataSet<? extends Number> set,
         ArrayList<Integer> depths,
403         int startI, int endI, int maxDepth) {
404       double val;
405       double smallest = Double.MIN_VALUE;
406       ArrayList<Double> values = new ArrayList<>();
407       for (int i = startI; i < endI; i++) {
408         val = set.getByIndex(i).doubleValue();
409         if (val < smallest) {
410           smallest = val;
411         }
412         values.add(val);
413       }
414       // if there are negative values, move all values up such that they are all
          positive
415       // this is required to prevent NaN or infinity values when taking the logarithm
416       // it does not affect the derivative values beyond possible floating point errors
417       if (smallest <= 0) {
418         smallest = Math.abs(smallest);
419         for (int i = 0; i < values.size(); i++) {
420           values.set(i, values.get(i) + smallest + Double.MIN_VALUE);
421         }
422       }
423       // prepare a DataSet to recalculate derivDepth of the logarithmic values
424       double step = set.getStep();
425       ValueDataSet<Double> otherSet = new ValueDataSet<>(set.getBase() + startI *
          step, step,
426           set.getPrecision(), d -> Double.valueOf(d));
427       for (Double element : values) {
428         otherSet.add(Math.log(element));
429       }
430       ArrayList<Integer> list = calcDerivDepth(otherSet, maxDepth, false); // prevent
          infinite
431                                                                            // recursion
432       // recheck if there are Integer.Max_Value derivDepths
433       int depth;
434       Integer otherStartI = null;
435       for (int i = 0; i < list.size(); i++) {
436         depth = list.get(i);
437         if (depth == Integer.MAX_VALUE) {
438           // track if not exponential for further analysis
439           if (otherStartI == null) {
440             otherStartI = i;
441           }
442         } else if (depth == 1) {
443           // mark as exponential
444           depths.set(i + startI, -2);
445           if (otherStartI != null) {
446             // if was tracking then mark change and stop
447             depths.set(i + startI - 1, -1);
448             // TODO trig
449             otherStartI = null;
450           }
451         } else {
452           // else transfer value over (e.g. change within exponential or bias)
453           depths.set(i + startI, depth);
454           if (otherStartI != null) {
455             // if was tracking then mark change
456             depths.set(i + startI - 1, -1);
457             // TODO trig
458             otherStartI = null;
459           }
460
461         }
```

```
462       }
463       // mark the change
464       depths.set(endI - 1, -1);
465     }
466
467   }
468
```