```java
/**
 * Erzeugt ein Feld, das für einen übergebenen Paramter {@code 0 < n <= 8} die
 * Nullstellen des {@code n}-ten Legendre-Polynoms bereitstellt und für
 * {@code n > 8} äquidistante Punkte im Intervall {@code (-1, 1).}
 */
public class KollokationsPunkte {

   /**
    * Das Feld {@code double[] rho} enthält die {@code n} Punkte.
    */
    final double[] rho;

    /**
     * Erzeugt eine Instanz für ein {@code n > 0}.
     * @param n
     */
    public KollokationsPunkte(int n) {
        rho = setzePunkte(n);
    }

//    private double[] setzePunkte(int n) {
//        double[] tempRho = new double[(int) (n+1)/2];
//        double temp1, temp2;
//        PolynomialFunction nthLegendre = PolynomialsUtils.createLegendrePolynomial(n);
//        tempRho[0] = findeNullstelle(nthLegendre, -1, 0);
//        tempRho[0] = findeNullstelle(nthLegendre, -1, tempRho[0]);
//        for (int i = 1; i < Math.ceil(n/2); i++) {
//            Arrays.sort(tempRho);
//            temp1 = findeNullstelle(nthLegendre, -1, tempRho[i-1]);
//            temp2 = findeNullstelle(nthLegendre, tempRho[i-1], 0);
//            if ((int) tempRho[i-1]*100 != (int) temp1*100)
//                tempRho[i] = temp1;
//            else if ((int) tempRho[i-1]*100 != (int) temp2*100)
//                tempRho[i] = temp2;
//        }
//                return tempRho;
//    }
//
//    private double findeNullstelle (PolynomialFunction func, double min, double max) {
//        return new NewtonRaphsonSolver().solve(200, func, min, max);
//    }

    /**
     * Erstellt ein {@code double[]}-Array, dass die
     * {@code \rho_i, i = 1, ..., k \subset (-1, 1)} enthält, aus denen dann
     * die {@code \tau_i} berechnet werden können.
     * @param n Anzahl der Kollokationspunkte.
     * @return das {@code double[]}-Array mit den {@code \rho_i}.
     */
    private double[] setzePunkte(int n) {
        double[] tempRho = new double[n];
        switch (n) {
        case 1:
            tempRho[0] = 0;
            break;
        case 2:
            tempRho[1] = .577350269189626;
            tempRho[0] = - tempRho[1];
            break;
        case 3:
            tempRho[2] = .774596669241483;
            tempRho[1] = 0;
```

```
63              tempRho[0] = - tempRho[2];
64              break;
65          case 4:
66              tempRho[3] = .861136311594053;
67              tempRho[2] = .339981043584856;
68              tempRho[1] = - tempRho[2];
69              tempRho[0] = - tempRho[3];
70              break;
71          case 5:
72              tempRho[4] = .906179845938664;
73              tempRho[3] = .538469310105683;
74              tempRho[2] = 0;
75              tempRho[1] = - tempRho[3];
76              tempRho[0] = - tempRho[4];
77              break;
78          case 6:
79              tempRho[5] = .932469514203152;
80              tempRho[4] = .661209386466265;
81              tempRho[3] = .238619186083197;
82              tempRho[2] = - tempRho[3];
83              tempRho[1] = - tempRho[4];
84              tempRho[0] = - tempRho[5];
85              break;
86          case 7:
87              tempRho[6] = .949107912342759;
88              tempRho[5] = .741531185599394;
89              tempRho[4] = .405845151377397;
90              tempRho[3] = 0;
91              tempRho[2] = - tempRho[4];
92              tempRho[1] = - tempRho[5];
93              tempRho[0] = - tempRho[6];
94              break;
95          case 8:
96              tempRho[7] = .960289856497536;
97              tempRho[6] = .796666477413627;
98              tempRho[5] = .525532409916329;
99              tempRho[4] = .183434642495650;
100             tempRho[3] = - tempRho[4];;
101             tempRho[2] = - tempRho[5];
102             tempRho[1] = - tempRho[6];
103             tempRho[0] = - tempRho[7];
104             break;
105         default:
106             double temp = (n + 1) / 2d;
107             for (int i = 1; i <= tempRho.length; i++) {
108                 tempRho[i-1] = i / temp - 1;
109             }
110         }
111         return tempRho;
112     }
113
114     /**
115      * Gibt den {@code i}-ten Kollokationspunkt zurück.
116      * @return {@code rho[i]}
117      */
118     public double getRho(int i) {
119         return rho[i];
120     }
121
122     /**
123      * Gibt eine Kopie des Feldes {@code double[] rho} der Kollokationspunkte
124      * im Intervall {@code [-1, 1]} zurück.
```

```
125      * @return eine Kopie von {@code double[] rho}
126      */
127     public double[] getRho() {
128         double out[] = new double[rho.length];
129         System.arraycopy(rho, 0, out, 0, rho.length);
130         return out;
131     }
132 
133     /**
134      * Gibt die Anzahl {@code n} der erzeugten Kollokationspunkte zurück.
135      * @return {@code n} Anzahl der Kollokationspunkte.
136      */
137     public int getN() {
138         return rho.length;
139     }
140 }
141 
```