```java
import org.apache.commons.math3.analysis.UnivariateFunction;
import org.apache.commons.math3.analysis.polynomials.PolynomialFunction;
import org.apache.commons.math3.linear.*;

/**
 * Implementierung der in der Ausarbeitung beschriebenen Kollokationsmethode
 * unter Verwendung der Bernsteinbasis.
 */
public class BezierKollokation {
    /**
     * Die Intervallgrenzen {@code [s, t]} und die Randwerte {@code y(s) = eta1}
     * und {@code y(t) = eta2}.
     */
    private final double s, t, eta1, eta2;
    /**
     * Die Anzahl {@code k} von Kollokationspunkten.
     */
    private final int k;
    /**
     * Die Kollokationspunkte {@code \tau_1, ... , \tau_k}.
     */
    private final double[] tau;
    /**
     * Die häufig auftretenden Faktoren {@code \mu_j^i := ((\tau_j - s) /
     * (t - s))^i, j = 1, ..., k, i = 1, ..., k+1}. Es ist
     * {@code mu[j-1][i-1] = \mu_j^i}.
     */
    private final double[][] mu;
    /**
     * Koeffizientenmatrix {@code A} des linearen Gleichungssystems.
     */
    private final OpenMapRealMatrix A;
    /**
     * rechte Seite {@code b} des linearen Gleichungssystems.
     */
    private final OpenMapRealVector v;
    /**
     * Die an der Differentialgleichung {@code y'' + ay' + by = f} beteiligten
     * Funktionen.
     */
    private UnivariateFunction a, b, f;
    /**
     * Die Näherungslösung g von {@code y'' + ay' + by = f}.
     */
    private BezierDarstellung g;

    /**
     * Erzeugt eine Instanz des Näherungsverfahrens und stößt die Berechnung
     * der Näherungslösung an.
     * @param k Anzahl der zu verwendenden Kollokationspunkte.
     * @param s linkes Intervallende.
     * @param t rechtes Intervallende.
     * @param eta1 linker Randwert.
     * @param eta2 rechter Randwert.
     * @param a Koeffizientenfunktion in {@code y'' + ay' + by = f}.
     * @param b Koeffizientenfunktion in {@code y'' + ay' + by = f}.
     * @param f rechte Seite der Differentialgleichung {@code y'' + ay' + by = f}.
     */
    public BezierKollokation(int k, double s, double t, double eta1, double eta2,
            UnivariateFunction a, UnivariateFunction b, UnivariateFunction f) {
        this.k = k;
        this.s = s;
```

```
63          this.t = t;
64          this.eta1 = eta1;
65          this.eta2 = eta2;
66          this.a = a;
67          this.b = b;
68          this.f = f;
69          tau = initialisiereTau();
70          mu = initialisiereMus();
71          A = initialisiereA();
72          v = initialisiereV();
73          RealVector loesung = new LUDecomposition(A).getSolver().solve(v);
74          g = new BezierDarstellung(loesung.toArray(), s, t);
75      }
76
77      /**
78       * Berechnet alle {@code \mu_j^i := \mu(\tau_j)^i} und
79       * {@code (1 - \mu_j)^i := (1 - \mu(\tau_j))^i} und speichert diese. Vor dem
80       * Aufruf dieser Prozedur muss das Feld {@code double[] tau} mit den
81       * Kollokationspunkten initialisiert sein. Die Berechnung der Funktionswerte
82       * ist dabei auf ein Minimum beschränkt unter Ausnutzung von<p>
83       * {@code \mu_j = 1 - \mu_(k-j-1)},</p>
84       * und äquivalent <p>
85       * {@code 1 - \mu_j = \mu_(k-j-1)}.</p>
86       */
87      private double[][] initialisiereMus () {
88          double[][] tempmu = new double[k][k+1];
89          for (int j = 0; j < k; j++) {
90              if (j > k - j)
91                  tempmu[j][0] = 1 - tempmu[k - j - 1][0];
92              else
93                  tempmu[j][0] = BezierDarstellung.mu(getTau(j+1), s, t);
94              for (int i = 1; i <= k; i++) {
95                  tempmu[j][i] = tempmu[j][i-1] * tempmu[j][0];
96              }
97          }
98          return tempmu;
99      }
100
101     /**
102      * Berechnet die streng monoton steigende Folge {@code \tau_j, j = 1, ..., k}
103      * im Intervall {@code [s, t]}.
104      */
105     private double[] initialisiereTau () {
106         double[] temptau = new double[k];
107         KollokationsPunkte rhos = new KollokationsPunkte(k);
108         for (int j = 0; j < k; j++) {
109             temptau[j] = (s + t + rhos.getRho(j) * (t - s)) / 2;
110         }
111         return temptau;
112     }
113
114     /**
115      * Berechnet die Einträge der Koeffizientenmatrix {@code A} des zu lösenden
116      * Gleichungssystems.
117      * @return A.
118      */
119     private OpenMapRealMatrix initialisiereA () {
120         /**
121          * Die Binomialkoeffizienten, deren Werte häufig gebraucht werden.
122          */
123         Binomialkoeffizient KMinusEins = new Binomialkoeffizient(k - 1);
124         Binomialkoeffizient K = new Binomialkoeffizient(k);
```

```
125         Binomialkoeffizient KPlusEins = new Binomialkoeffizient(k + 1);
126         /**
127          * Häufig verwendete Berechnungsschritte und Funktionswerte.
128          */
129         double TMinusS = t - s,
130                 kPlusDivTMinusS = (k+1) / TMinusS,
131                 kPlusKDivTMinusSSqr = kPlusDivTMinusS * k / TMinusS,
132                 TMinusSSqr = Math.pow(TMinusS, 2), aJ, bJ;
133         OpenMapRealMatrix tempA = new OpenMapRealMatrix(k + 2, k + 2);
134         /**
135          * Befüllt die erste und letzte Zeile der Matrix {@code A} mit den
136          * Randbedingungen.
137          */
138         OpenMapRealVector tempvektor = new OpenMapRealVector(k+2);
139         tempvektor.setEntry(0, 1d);
140         tempA.setRowVector(0, tempvektor);
141         tempvektor.setEntry(0, 0d);
142         tempvektor.setEntry(k+1, 1d);
143         tempA.setRowVector(k+1, tempvektor);
144         /**
145          * Befüllt die {@code k} Zeilen der Matrix {@code A}, welche aus den
146          * Kollokationsbedingungen hervorgehen.
147          */
148         for (int j = 1; j <= k; j++) {
149             /**
150              * Berechnung der Funktionswerte von {@code a, b} für
151              * {@code x = tau_j}.
152              */
153             aJ = a.value(getTau(j));
154             bJ = b.value(getTau(j));
155             /**
156              * Berechnung des Summanden bezüglich {@code b_0} der j-ten
157              * Kollokationsbedingung.
158              */
159             tempA.setEntry(j, 0, getMu(j, k - 1, true) * ( kPlusKDivTMinusSSqr +
160                     getMu(j, 1, true) * (bJ * getMu(j, 1, true) -
161                             aJ * kPlusDivTMinusS)));
162             /**
163              * Berechnung des Summanden bezüglich {@code b_1} der j-ten
164              * Kollokationsbedingung.
165              */
166             tempA.setEntry(j, 1, (k+1) * getMu(j, k-2, true) * (k / TMinusSSqr *
167                     ((k-1) * getMu(j, 1, false) - 2 * getMu(j, 1, true)) +
168                     aJ / TMinusS * (1 - (k+1) * getMu(j, 1, false)) *
169                     getMu(j, 1, true) + bJ * getMu(j, 2, true) *
170                     getMu(j, 1, false)));
171             /**
172              * Berechnung des Summanden bezüglich {@code b_i, i = 2, ..., k-1}
173              * der j-ten Kollokationsbedingung.
174              */
175             for (int i = 2; i < k ; i++) {
176                 tempA.setEntry(j, i, kPlusKDivTMinusSSqr *(KMinusEins.getUeber(i-2)
177                         * getMu(j, 2, true) - 2 * KMinusEins.getUeber(i-1) *
178                         getMu(j, 1, true) * getMu(j, 1, false) +
179                         KMinusEins.getUeber(i) * getMu(j, 2, false)) *
180                         getMu(j, k - 1 - i, true) * getMu(j, i - 2, false)
181                         + aJ * kPlusDivTMinusS *
182                         (K.getUeber(i-1) - KPlusEins.getUeber(i) *
183                                 getMu(j, 1, false)) * getMu(j, k-i, true) *
184                         getMu(j, i-1, false) + bJ * KPlusEins.getUeber(i) *
185                         getMu(j, k+1-i, true) * getMu(j, i, false));
186             }
```

```
187             /**
188              * Berechnung des Summanden bezüglich {@code b_k} der j-ten
189              * Kollokationsbedingung.
190              */
191             tempA.setEntry(j, k, (k+1) * getMu(j, k-2, false) * (k / TMinusSSqr *
192                     ((k-1) * getMu(j, 1, true) - 2 * getMu(j, 1, false)) +
193                     aJ / TMinusS * (k - (k+1) * getMu(j, 1, false)) *
194                     getMu(j, 1, false) + bJ * getMu(j, 2, false) *
195                     getMu(j, 1, true)));
196             /**
197              * Berechnung des Summanden bezüglich {@code b_(k+1)} der j-ten
198              * Kollokationsbedingung.
199              */
200             tempA.setEntry(j, k+1, getMu(j, k - 1, false) * (kPlusKDivTMinusSSqr+
201                     getMu(j, 1, false) * (bJ * getMu(j, 1, false)
202                             - aJ * kPlusDivTMinusS)));
203         }
204 //        /**
205 //         * Gibt die Matrix {@code A} zu Testzwecken aus.
206 //         */
207 //        for (int j = 0; j < k+2; j++) {
208 //            for (int i = 0; i < k+2; i++) {
209 //                System.out.println("A[" + j + "," + i + "] = " +
210 //                tempA.getEntry(j, i));
211 //            }
212 //        }
213         return tempA;
214     }
215 
216     /**
217      * Berechnet die Einträge der rechten Seite {@code v} des zu lösenden
218      * Gleichungssystems.
219      * @return v.
220      */
221     private OpenMapRealVector initialisiereV() {
222         OpenMapRealVector tempV = new OpenMapRealVector(k+2);
223         tempV.setEntry(0, eta1);
224         for (int i = 1; i <= k; i++) {
225             tempV.setEntry(i, f.value(getTau(i)));
226         }
227         tempV.setEntry(k+1, eta2);
228         return tempV;
229     }
230 
231     /**
232      * Gibt {@code tau_j, j = 1, ..., k} zurück.
233      * @param j für das {@code tau_j, j = 1, ..., k} zurückgegeben werden soll.
234      * @return tau_j
235      */
236     public double getTau (int j) {
237         return tau[j-1];
238     }
239 
240     /**
241      * Gibt {@code mu_j^i} oder {@code (1 - mu_j)^i} zurück.
242      * @param j aus {@code 1, ..., k} zur Auswahl von {@code mu_j} oder
243      * {@code (1 - mu_j)}.
244      * @param exponent {@code i \in -1, 0, ..., k+1}, für den {@code mu_j^i} oder
245      * {@code (1 - mu_j)^i}.
246      * @param invers {@code true}, falls {@code (1 - mu_j)^i} und {@code false},
247      * falls {@code mu_j^i} zurückgegeben werden soll
248      * @return {@code mu_j^i} oder {@code (1 - mu_j)^i}
```

```
249      * @throws ArrayIndexOutOfBoundsException falls {@code i < -1} oder
250      * {@code i > k+1}.
251      */
252     public double getMu (int j, int exponent, boolean invers)
253             throws ArrayIndexOutOfBoundsException {
254         if (exponent < -1 || exponent > k+1)
255             throw new ArrayIndexOutOfBoundsException();
256         if (exponent == -1)
257             if (invers)
258                 return 1 / mu[k - j][0];
259             else
260                 return 1 / mu[j - 1][0];
261         else if (exponent == 0)
262             return 1;
263         else if (invers)
264                 return mu[k - j][exponent-1];
265             else
266                 return mu[j - 1][exponent-1];
267         }
268 
269     /**
270      * Gibt die bestimmte Näherungslösung zurück.
271      * @return g
272      */
273     public BezierDarstellung getG () {
274         return g;
275     }
276 
277     /**
278      * Auszug aus den möglichen Farbnamen in Maple.
279      */
280     enum Linienfarbe {Brown, Crimson, Chocolate, Orange, SkyBlue, Magenta, Gold};
281 
282     /**
283      * Erzeugt für ein übergebenes {@code k} eine Instanz des
284      * Kollokationsverfahrens zu Beispiel3 aus {G. Müllenheim, 1986}.
285      * @param k die Anzahl der Kollokationspunkte.
286      * @param mode Gibt an, welche Testdaten auf der Konsole ausgegeben werden
287      * sollen: <p>
288      * {@code mode = 1}: Gibt die Differenzen der Ableitungswerte an einigen
289      * Stellen des Intervalls {@code [0, 1]} aus.
290      * </p><p>
291      * {@code mode = 2}: Gibt die Maplebefehle zum Plotten der Näherungslösung
292      * aus. </p><p>
293      * {@code mode = 3}: Gibt die Abweichungen von den Kollokations- und
294      * Randbedingungen aus.
295      * </p><p>
296      * {@code mode = 4}: Gibt die {@code mu_j^i} für alle zulässigen
297      * Parameterwerte aus. </p><p>
298      * {@code mode = 5}: Gibt alle {@code tau_j} aus.
299      * </p><p>
300      * {@code mode = 6}: Gibt alle Abweichungen der Ableitungen i-ter Ordnung
301      * ({@code i = 0, 1, 2}) der korrespondierenden exakten Lösung und der
302      * Näherungslösung aus.
303      * </p>
304      * @return Ausgabe gemäß {@code mode}.
305      */
306     public static String teste3 (int k, int mode) {
307         String ausgabe = "k = " + k + "\n";
308         /**
309          * Der linke Rand {@code s} des Kollokationsintervalls {@code [s, t]}.
310          */
```

```
311         double s = -1;
312         /**
313          * Der rechte Rand {@code t} des Kollokationsintervalls {@code [s, t]}.
314          */
315         double t = 1;
316         /**
317          * Der Randwert {@code y(s) = \eta_1}.
318          */
319         double eta1 = 0;
320         /**
321          * Der Randwert {@code y(t) = \eta_2}.
322          */
323         double eta2 = 0;
324         /**
325          * Die Koeffizientenfunktion {@code a} in {@code y'' + a y' + b y = f}.
326          */
327         PolynomialFunction a = new PolynomialFunction(new double[] {0});
328         /**
329          * Die Koeffizientenfunktion {@code b} in {@code y'' + a y' + b y = f}.
330          */
331         PolynomialFunction b = new PolynomialFunction(new double[] {-2});
332         /**
333          * Die Funktion {@code f} in {@code y'' + a y' + b y = f}.
334          */
335         UnivariateFunction f = new UnivariateFunction() {
336
337             public double value(double x) {
338                 return 4 * Math.pow(x, 2) * Math.exp(Math.pow(x, 2));
339             }
340         };
341         BezierKollokation bkol = new BezierKollokation
342                 (k, s, t, eta1, eta2, a, b, f);
343         BezierDarstellung g = bkol.getG();
344         Beispiel3 u;
345         double x;
346         int n = 50;
347         switch (mode) {
348         case 1:
349             u = new Beispiel3();
350             for (int i = 0; i <= n; i++) {
351                 x = s + i * (t-s)/n;
352                 for (int j = 0; j <= 2; j++) {
353                     ausgabe += "u^(" + j + ")(" + x + ") - g^(" + j + ")(" +
354                             x + ") = " + (u.getAbleitung(x, j) -
355                                     g.derivative(x, j));
356                     if (j < 2) ausgabe += "\n";
357                 }
358                 if (i < n) ausgabe += "\n";
359             }
360             break;
361         case 2:
362             String werte = "", stellen = "";
363             for (int i = 0; i <= n; i++) {
364                 x = s + i * (t-s)/n;
365                 werte += g.value(x);
366                 stellen += x;
367                 if (i != n) {
368                     werte += ",";
369                     stellen += ",";
370                 }
371             }
372             ausgabe = "g" + k + ":=pointplot([" + stellen + " ], ["+ werte +
```

```
373                     "], legend = g^" + k + ", color = \"" +
374                     Linienfarbe.values()[(k-1)%Linienfarbe.values().length]
375                             + "\", connect = true):";
376             break;
377         case 3:
378             ausgabe += "g(" + s + ") = " + g.value(s) + "\n";
379             for (int j = 1; j <= k; j++) {
380                 ausgabe += "tau" + j + " = " + bkol.getTau(j) +
381                         ": g'' + a * g' + b * g - f = " +
382                         (g.derivative(bkol.getTau(j), 2) +
383                                 a.value(bkol.getTau(j)) *
384                                 g.derivative(bkol.getTau(j), 1) +
385                                 b.value(bkol.getTau(j)) *
386                                 g.derivative(bkol.getTau(j), 0) -
387                                 f.value(bkol.getTau(j)))
388                         + "\n";
389             }
390             ausgabe += "g(" + t + ")= " + g.value(t);
391             break;
392         case 4:
393             for (int j = 1; j <= k; j++) {
394                 for (int i = -1; i <= k+1; i++) {
395                     ausgabe += "mu_" + j + "^" + i + " = " +
396                 bkol.getMu(j, i, false) + "\n";
397                 }
398             }
399             break;
400         case 5:
401             for (int j = 1; j <= k; j++) {
402                 ausgabe += "tau_" + j + " = " + bkol.getTau(j) + "\n";
403             }
404             break;
405         case 6:
406             double max;
407             u = new Beispiel3();
408             for (int i = 0; i <= 2; i++) {
409                 max = 0;
410                 for (int j = 1; j <= k; j++) {
411                     max = Math.max(max, Math.abs(g.derivative(bkol.getTau(j), i)
412                             - u.getAbleitung(bkol.getTau(j), i)));
413                 }
414                 ausgabe += "E_" + k + "^" + i + " = " + max + "\n";
415             }
416             break;
417         }
418         return ausgabe;
419     }
420
421     /**
422      * Erzeugt für ein übergebenes {@code k} eine Instanz des
423      * Kollokationsverfahrens zu Beispiel4 aus {G. Müllenheim, 1986}.
424      * @param k die Anzahl der Kollokationspunkte.
425      * @param mode Gibt an, welche Testdaten auf der Konsole ausgegeben werden
426      * sollen: <p>
427      * {@code mode = 1}: Gibt die Differenzen der Ableitungswerte an einigen
428      * Stellen des Intervalls {@code [0, 1]} aus.
429      * </p><p>
430      * {@code mode = 2}: Gibt die Maplebefehle zum Plotten der Näherungslösung
431      * aus.</p><p>
432      * {@code mode = 3}: Gibt die Abweichungen von den Kollokations- und
433      * Randbedingungen aus.
434      * </p><p>
```

```
435      * {@code mode = 4}: Gibt die {@code mu_j^i} für alle zulässigen
436      * Parameterwerte aus. </p><p>
437      * {@code mode = 5}: Gibt alle {@code tau_j} aus.
438      * </p><p>
439      * {@code mode = 6}: Gibt alle Abweichungen der Ableitungen i-ter Ordnung
440      * ({@code i = 0, 1, 2}) der korrespondierenden exakten Lösung und der
441      * Näherungslösung aus.
442      * </p>
443      * @return Ausgabe gemäß {@code mode}.
444      */
445     public static String teste4 (int k, int mode) {
446         String ausgabe = "k = " + k + "\n";
447         /**
448          * Der linke Rand {@code s} des Kollokationsintervalls {@code [s, t]}.
449          */
450         double s = 0;
451         /**
452          * Der rechte Rand {@code t} des Kollokationsintervalls {@code [s, t]}.
453          */
454         double t = 1;
455         /**
456          * Der Randwert {@code y(s) = \eta_1}.
457          */
458         double eta1 = 0;
459         /**
460          * Der Randwert {@code y(t) = \eta_2}.
461          */
462         double eta2 = 0;
463         /**
464          * Die Koeffizientenfunktion {@code a} in {@code y'' + a y' + b y = f}.
465          */
466         PolynomialFunction a = new PolynomialFunction(new double[] {0});
467         /**
468          * Die Koeffizientenfunktion {@code b} in {@code y'' + a y' + b y = f}.
469          */
470         PolynomialFunction b = new PolynomialFunction(new double[] {-4});
471         /**
472          * Die Funktion {@code f} in {@code y'' + a y' + b y = f}.
473          */
474         UnivariateFunction f = new UnivariateFunction() {
475 
476             private final double value = 4 * Math.cosh(1);
477 
478             public double value(double x) {
479                 return value;
480             }
481         };
482         BezierKollokation bkol = new BezierKollokation
483                 (k, s, t, eta1, eta2, a, b, f);
484         BezierDarstellung g = bkol.getG();
485         Beispiel4 u;
486         double x;
487         int n = 50;
488         switch (mode) {
489         case 1:
490             u = new Beispiel4();
491             for (int i = 0; i <= n; i++) {
492                 x = s + i * (t-s)/n;
493                 for (int j = 0; j <= 2; j++) {
494                     ausgabe += "u^(" + j + ")(" + x + ") - g^(" + j + ")(" + x
495                             + ") = " + (u.getAbleitung(x, j)- g.derivative(x, j));
496                     if (j < 2) ausgabe += "\n";
```

```
497                 }
498                 if (i < n) ausgabe += "\n";
499             }
500             break;
501         case 2:
502             String werte = "", stellen = "";
503             for (int i = 0; i <= n; i++) {
504                 x = s + i * (t-s)/n;
505                 werte += g.value(x);
506                 stellen += x;
507                 if (i != n) {
508                     werte += ",";
509                     stellen += ",";
510                 }
511             }
512             ausgabe = "u" + k + ":=pointplot([" + stellen + " ], ["+ werte +
513                     "], legend = g^" + k + ", color = \"" + Linienfarbe.values()
514                     [(k-1)%Linienfarbe.values().length] + "\", connect = true):";
515             break;
516         case 3:
517             ausgabe += "g(" + s + ") = " + g.value(s) + "\n";
518             for (int j = 1; j <= k; j++) {
519                 ausgabe += "tau" + j + " = " + bkol.getTau(j) +
520                         ": g'' + a * g' + b * g - f = " +
521                         (g.derivative(bkol.getTau(j), 2) +
522                                 a.value(bkol.getTau(j)) *
523                                 g.derivative(bkol.getTau(j), 1) +
524                                 b.value(bkol.getTau(j)) *
525                                 g.derivative(bkol.getTau(j), 0) -
526                                 f.value(bkol.getTau(j)))
527                         + "\n";
528             }
529             ausgabe += "g(" + t + ")= " + g.value(t);
530             break;
531         case 4:
532             for (int j = 1; j <= k; j++) {
533                 for (int i = -1; i <= k+1; i++) {
534                     ausgabe += "mu_" + j + "^" + i + " = " +
535                 bkol.getMu(j, i, false) + "\n";
536                 }
537             }
538             break;
539         case 5:
540             for (int j = 1; j <= k; j++) {
541                 ausgabe += "tau_" + j + " = " + bkol.getTau(j) + "\n";
542             }
543             break;
544         case 6:
545             double max;
546             u = new Beispiel4();
547             for (int i = 0; i <= 2; i++) {
548                 max = 0;
549                 for (int j = 1; j <= k; j++) {
550                     max = Math.max(max, Math.abs(g.derivative(bkol.getTau(j), i)
551                             - u.getAbleitung(bkol.getTau(j), i)));
552                 }
553                 ausgabe += "E_" + k + "^" + i + " = " + max + "\n";
554             }
555             break;
556         }
557         return ausgabe;
558     }
```

```
559
560     public static void main (String[] args) {
561         /**
562          * Testen zweier Beispiele in verschiedenen Modi durch Ausgabe
563          * interessierender Daten auf der Konsole.
564          */
565         for (int k = 1; k <= 9; k++) {
566             System.out.println(teste3(k, 6));
567 //            System.out.println(teste4(k, 6));
568         }
569     }
570 }
571
```