```cpp
// SimplexAlgorithm.cpp (c) Thursday, October 24, 2024
// by James Pate Williams, Jr., BA, BS, MSWE, PhD
// See "Elementary Numerical Analysis : An Algorithmic
// Approach" by S.D. Conte and Carl de Boor (c) 1980
// Third Edition

#include <stdio.h>
#include <iostream>
#include <limits>
#include <vector>
using namespace std;

class Simplex {
public:
    Simplex(
        const vector<vector<double>>& A,
        const vector<double>& b,
        const vector<double>& c)
        : A(A), b(b), c(c), m(b.size()), n(c.size()),
        tableau(m + 1, vector<double>(n + m + 1)) {
        initializeTableau();
    }

    void solve() {
        while (true) {
            int pivotCol = findPivotColumn();
            if (pivotCol == -1) break; // Optimal solution found

            int pivotRow = findPivotRow(pivotCol);
            if (pivotRow == -1) {
                cout << "Unbounded solution" << endl;
                return;
            }

            pivot(pivotRow, pivotCol);
        }
        printSolution();
    }

private:
    vector<vector<double>> A;
    vector<double> b;
    vector<double> c;
    int m, n;
    vector<vector<double>> tableau;

    void initializeTableau() {
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                tableau[i][j] = A[i][j];
            }
            tableau[i][n + i] = 1.0;
```

```
53              tableau[i][n + m] = b[i];
54          }
55          for (int j = 0; j < n; ++j) {
56              tableau[m][j] = -c[j];
57          }
58      }
59
60      int findPivotColumn() {
61          int pivotCol = -1;
62          double minValue = 0.0;
63          for (int j = 0; j < n + m; ++j) {
64              if (tableau[m][j] < minValue) {
65                  minValue = tableau[m][j];
66                  pivotCol = j;
67              }
68          }
69          return pivotCol;
70      }
71
72      int findPivotRow(int pivotCol) {
73          int pivotRow = -1;
74          double minRatio = numeric_limits<double>::infinity();
75          for (int i = 0; i < m; ++i) {
76              if (tableau[i][pivotCol] > 0) {
77                  double ratio = tableau[i][n + m] / tableau[i][pivotCol];
78                  if (ratio < minRatio) {
79                      minRatio = ratio;
80                      pivotRow = i;
81                  }
82              }
83          }
84          return pivotRow;
85      }
86
87      void pivot(int pivotRow, int pivotCol) {
88          double pivotValue = tableau[pivotRow][pivotCol];
89          for (int j = 0; j <= n + m; ++j) {
90              tableau[pivotRow][j] /= pivotValue;
91          }
92          for (int i = 0; i <= m; ++i) {
93              if (i != pivotRow) {
94                  double factor = tableau[i][pivotCol];
95                  for (int j = 0; j <= n + m; ++j) {
96                      tableau[i][j] -= factor * tableau[pivotRow][j];
97                  }
98              }
99          }
100     }
101
102     void printSolution() {
103         vector<double> solution(n, 0.0);
104         for (int i = 0; i < m; ++i) {
```

```
105             bool isBasic = true;
106             int basicVar = -1;
107             for (int j = 0; j < n; ++j) {
108                 if (tableau[i][j] == 1.0) {
109                     if (basicVar == -1) {
110                         basicVar = j;
111                     }
112                     else {
113                         isBasic = false;
114                         break;
115                     }
116                 }
117                 else if (tableau[i][j] != 0.0) {
118                     isBasic = false;
119                     break;
120                 }
121             }
122             if (isBasic && basicVar != -1) {
123                 solution[basicVar] = tableau[i][n + m];
124             }
125         }
126         cout << "Optimal solution: ";
127         for (double x : solution) {
128             cout << x << " ";
129         }
130         cout << endl;
131     }
132 };
133
134 class SteepestDescent
135 {
136 public:
137     SteepestDescent(
138         const vector<double> x0) :
139         x0(x0), n(x0.size())
140     { }
141     double F(vector<double> x);
142     vector<double> gradient(vector<double> x);
143     double HillClimber(
144         double tolerance,
145         int m, int p, int seed);
146     vector<double> getxm() {
147         return xm;
148     };
149 private:
150     int n;
151     vector<double> uv, x0, xm;
152 };
153
154 double SteepestDescent::HillClimber(
155     double tolerance,
156     int m, int p, int seed) {
```

```
157     bool satisfied = false;
158     double ts = 0;
159     vector<double> tv(p, 0.0), uv;
160     vector<double> gv(p, 0.0);
161     vector<vector<double>> xv;
162
163     srand(seed);
164     uv = gradient(x0);
165
166     for (int i = 0; i < p; i++) {
167         tv[i] = (double)rand() / RAND_MAX;
168         xm.clear();
169
170         for (int j = 0; j < n; j++)
171             xm.push_back(x0[j] - tv[i] * uv[j]);
172
173         gv[i] = F(xm);
174         xv.push_back(xm);
175     }
176
177     for (int i = 1; i <= m; i++)
178     {
179         int parent1 = rand() % p;
180         int parent2 = rand() % p;
181         int parent0 = tv[parent1] > tv[parent2] ?
182             parent1 : parent2;
183         double gt = 0, ts = (double)rand() / RAND_MAX;
184
185         ts = 0.5 * (double)rand() / RAND_MAX;
186         uv = gradient(x0);
187
188         for (int j = 0; j < n; j++)
189             xm[j] = x0[j] - ts * uv[j];
190
191         gt = F(xm);
192
193         satisfied = true;
194
195         for (int j = 0; satisfied && j < n; j++)
196             satisfied = fabs(uv[j]) < tolerance;
197
198         if (satisfied && ts > 0)
199             break;
200
201         for (int j = 0; j < n; j++)
202             x0[j] = xm[j];
203
204         double maxtv = DBL_MIN;
205
206         for (int j = 0; j < p; j++)
207             if (tv[j] > maxtv)
208                 maxtv = tv[j];
```

```
209
210         vector<int> index;
211
212         for (int j = 0; j < p; j++)
213             if (maxtv == tv[j])
214                 index.push_back(j);
215
216         int replace = index[rand() % index.size()];
217
218         tv[replace] = ts;
219     }
220
221     if (!satisfied) {
222         ts = DBL_MAX;
223
224         for (int i = 0; i < p; i++)
225         {
226             if (tv[i] < ts)
227                 ts = tv[i];
228         }
229
230         for (int i = 0; i < n; i++)
231             xm[i] = x0[i] - ts * uv[i];
232     }
233
234     return ts;
235 }
236
237 double SteepestDescent::F(vector<double> x) {
238     double x1 = x[0];
239     double x2 = x[1];
240
241     return x1 * x1 * x1 + x2 * x2 * x2 -
242         2.0 * x1 * x1 + 3.0 * x2 * x2 - 8.0;
243 }
244
245 vector<double> SteepestDescent::gradient(vector<double> x) {
246     double x1 = x[0];
247     double x2 = x[1];
248     vector<double> grad(x.size(), 0.0);
249     grad[0] = 3.0 * x1 * x1 - 4.0 * x1;
250     grad[1] = 3.0 * x2 * x2 + 6.0 * x2;
251     return grad;
252 }
253
254 class Jacobian {
255 public:
256     Jacobian(
257         vector<double> x, vector<double> f,
258         double(*di)(int),
259         vector<double>(*func)(int, vector<double>)) :
260         x(x), f(f), n(x.size())
```

```
261     {
262         // x is the n-dimensional point vector
263         // f is the n-dimensional function vector
264         // func is an n-dimensional function
265         this->di = di;
266         this->func = func;
267     }
268     void CalculateJacobianMatrix(double** jac);
269     double(*di)(int);
270     vector<double>(*func)(int, vector<double>);
271 private:
272     int n;
273     vector<double> f, x;
274 };
275 
276 void Jacobian::CalculateJacobianMatrix(double** jac)
277 {
278     vector<double> f1(n, 0);
279 
280     for (int i = 0; i < n; i++)
281     {
282         double step = di(i);
283         double aid = x[i];
284 
285         x[i] = aid + step;
286         step = 1.0 / step;
287         f1 = func(n, x);
288 
289         for (int j = 0; j < n; j++)
290             jac[j][i] = (f1[j] - f[j]) * step;
291 
292         x[i] = aid;
293     }
294 }
295 
296 class NewtonsMethod
297 {
298 public:
299     NewtonsMethod(
300         const vector<double> x0,
301         double(*di)(int),
302         vector<double>(*func)(int, vector<double>)) :
303         x0(x0), n(x0.size())
304     {
305         this->di = di;
306         this->func = func;
307     };
308     vector<double> NewtonFunction(
309         int n, vector<double> x);
310     long Solve(
311         double epsilon,
312         vector<double> f1,
```

```
313         vector<double> x1,
314         int jMax, long maxIt);
315 private:
316     int n;
317     vector<double> f, x0, x1;
318     double(*di)(int);
319     vector<double>(*func)(int, vector<double>);
320     void DupColVec(
321         int l, int u, int j, double** a, double b[]);
322     void ICCol(int l, int u, int i, int j, double** a);
323     void ICRow(int l, int u, int i, int j, double** a);
324     double MatMat(
325         int l, int u, int i, int j, double** a, double** b);
326     double MatTam(
327         int l, int u, int i, int j, double** a, double** b);
328     void Dec(double** a, int n, double aux[], int p[]);
329     void Inv(double** a, int n, int p[]);
330     void DecInv(double** a, int n);
331     double VecNorm2(int n, double v[]);
332     void MatVecMul(
333         int n, double a[], double** b, double v[]);
334 };
335
336 void NewtonsMethod::DupColVec(
337     int l, int u, int j, double** a, double b[]) {
338     for (; l < u; l++)
339         a[l][j] = b[l];
340 }
341
342 void NewtonsMethod::ICCol(
343     int l, int u, int i, int j, double** a) {
344     for (; l < u; l++)
345     {
346         double r = a[l][i];
347
348         a[l][i] = a[l][j];
349         a[l][j] = r;
350     }
351 }
352
353 void NewtonsMethod::ICRow(int l, int u, int i, int j, double** a) {
354     for (; l < u; l++)
355     {
356         double r = a[i][l];
357
358         a[i][l] = a[j][l];
359         a[j][l] = r;
360     }
361 }
362
363 double NewtonsMethod::MatMat(
364     int l, int u, int i, int j, double** a, double** b)
```

```
365 {
366     double s = 0.0;
367 
368     for (int k = l; k < u; k++)
369         s += a[i][k] * b[k][j];
370 
371     return s;
372 }
373 
374 double NewtonsMethod::MatTam(
375     int l, int u, int i, int j, double** a, double** b)
376 {
377     double s = 0.0;
378 
379     for (int k = l; k < u; k++)
380         s += a[i][k] * b[j][k];
381 
382     return s;
383 }
384 
385 void NewtonsMethod::Dec(
386     double** a, int n, double aux[], int p[])
387 {
388     double eps, r, s;
389     int k, pk = 0;
390     vector<double> v(n, 0);
391 
392     r = -1.0;
393 
394     for (int i = 0; i < n; i++)
395     {
396         s = sqrt(MatTam(0, n, i, i, a, a));
397 
398         if (s > r)
399             r = s;
400 
401         v[i] = 1.0 / s;
402     }
403 
404     eps = aux[2] * r;
405 
406     int d = 1;
407 
408     for (k = 0; k < n; k++)
409     {
410         int k1;
411 
412         r = -1.0;
413         k1 = k - 1;
414 
415         for (int i = k; i < n; i++)
416         {
```

```
417             a[i][k] -= MatMat(0, k1, i, k, a, a);
418             s = fabs(a[i][k]) * v[i];
419
420             if (s > r)
421             {
422                 r = s;
423                 pk = i;
424             }
425         }
426
427         p[k] = pk;
428         v[pk] = v[k];
429         s = a[pk][k];
430
431         if (fabs(s) < eps)
432             break;
433
434         if (s < 0.0)
435             d = -d;
436
437         if (pk != k)
438         {
439             d = -d;
440             ICRow(0, n, k, pk, a);
441         }
442
443         for (int i = k + 1; i < n; i++)
444             a[k][i] = (a[k][i] - MatMat(0, k1, k, i, a, a)) / s;
445     }
446
447     aux[0] = d;
448     aux[2] = k - 1.0;
449 }
450
451 void NewtonsMethod::Inv(double** a, int n, int p[])
452 {
453     double r;
454     int j, k, k1;
455     vector<double> v(n, 0);
456
457     for (k = n - 1; k >= 0; k--)
458     {
459         k1 = k + 1;
460
461         for (j = n - 1; j >= k1; j--)
462         {
463             a[j][k1] = v[j];
464             v[j] = -MatMat(k1, n, k, j, a, a);
465         }
466
467         r = a[k][k];
468
```

```
469         for (j = n - 1; j >= k1; j--)
470         {
471             a[k][j] = v[j];
472             v[j] = -MatMat(k1, n, j, k, a, a) / r;
473         }
474
475         v[k] = (1.0 - MatMat(k1, n, k, k, a, a)) / r;
476     }
477
478     DupColVec(0, n, 0, a, v.data());
479
480     for (k = n - 2; k >= 0; k--)
481     {
482         k1 = p[k];
483
484         if (k1 != k)
485             ICCol(0, n, k, k1, a);
486     }
487 }
488
489 void NewtonsMethod::DecInv(double** a, int n)
490 {
491     double aux[3] = { };
492     vector<int> p(n, 0);
493
494     aux[0] = 1.0e-10;
495
496     Dec(a, n, aux, p.data());
497
498     if (aux[2] == n)
499         Inv(a, n, p.data());
500 }
501
502 double NewtonsMethod::VecNorm2(int n, double v[])
503 {
504     double s = 0.0;
505
506     for (int i = 0; i < n; i++)
507         s += v[i] * v[i];
508
509     return sqrt(s);
510 }
511
512 void NewtonsMethod::MatVecMul(
513     int n, double a[], double** b, double v[])
514 {
515     for (int i = 0; i < n; i++)
516     {
517         double s = 0;
518
519         for (int j = 0; j < n; j++)
520             s += b[i][j] * v[j];
```

```
521
522         a[i] = s;
523     }
524 }
525
526 long NewtonsMethod::Solve(
527     double epsilon,
528     vector<double> f1,
529     vector<double> x1,
530     int jMax, long maxIt)
531 {
532     vector<double> f0 = func(n, x0);
533     vector<double> h(n, 0), g(n, 0);
534     double** jac = new double* [n];
535
536     for (int i = 0; i < n; i++)
537         jac[i] = new double[n];
538
539     int i, j;
540     long iterations = 0;
541
542     do
543     {
544         Jacobian jacobian(x0, f0, di, func);
545         jacobian.CalculateJacobianMatrix(jac);
546         DecInv(jac, n);
547         MatVecMul(n, h.data(), jac, f0.data());
548
549         for (int k = 0; k < n; k++)
550             h[k] = -h[k];
551
552         double n0 = VecNorm2(n, f0.data()), n1;
553
554         for (j = 0; j < jMax; j++)
555         {
556             for (int k = 0; k < n; k++)
557             {
558                 g[k] = h[k] / pow(2.0, (double)j);
559                 x1[k] = x0[k] + g[k];
560             }
561
562             f1 = func(n, x1);
563
564             n1 = VecNorm2(n, f1.data());
565
566             if (n1 < n0)
567             {
568                 i = j;
569                 break;
570             }
571         }
572
```

```
573         if (j == jMax)
574             return maxIt;
575 
576         char buffer[1024] = { }, line[128] = { };
577         double norm2 = VecNorm2(n, f0.data());
578         double norm1 = 0;
579 
580         for (int i = 0; i < n; i++)
581             norm1 += fabs(f0[i]);
582 
583         buffer[0] = '\0';
584 
585         sprintf_s(line, 127, "%3ld\t", iterations + 1);
586         strcat_s(buffer, line);
587 
588         for (int i = 0; i < n; i++) {
589             sprintf_s(line, 127, "%12.10lf\t", x0[i]);
590             strcat_s(buffer, line);
591         }
592 
593         sprintf_s(line, 127, "%17.10e\t%17.10e", norm1, norm2);
594         strcat_s(buffer, line);
595 
596         cout << buffer << endl;
597 
598         if (norm2 < epsilon)
599             return iterations;
600 
601         iterations++;
602 
603         for (int k = 0; k < n; k++)
604         {
605             f0[k] = f1[k];
606             x0[k] = x1[k];
607         }
608     } while (iterations < maxIt);
609 
610     return iterations;
611 }
612 
613 double di(int i)
614 {
615     return 0.00000001;
616 }
617 
618 vector<double> NewtonFunction(
619     int n, vector<double> x)
620 {
621     vector<double> f(n, 0);
622 
623     for (int i = 0; i < n; i++)
624     {
```

```
625         if (i == 0)
626             f[i] = x[i + 1] * x[i + 1] - 3.0 * x[i] * x[i];
627         else if (i < n - 1)
628             f[i] = x[i - 1] * x[i - 1] +
629             x[i - 1] * x[i + 1] + x[i + 1] * x[i + 1]
630             - 3.0 * x[i] * x[i];
631         else
632             f[i] = x[i - 1] * x[i - 1] + x[i - 1] +
633             1.0 - 3.0 * x[i] * x[i];
634     }
635
636     return f;
637 }
638
639 int main() {
640     vector<vector<double>> A = {
641         {2, 1},
642         {1, 1}
643     };
644     vector<double> b = { 20, 16 };
645     vector<double> c = { 3, 2 };
646
647     Simplex simplex(A, b, c);
648     cout << "Simplex Solution" << endl << endl;
649     simplex.solve();
650     cout << endl;
651
652     double tolerance = 1.0e-15;
653     int m = 100000, n = 2;
654     vector<double> x0(2, 0), xm;
655
656     x0[0] = +1;
657     x0[1] = -1;
658
659     SteepestDescent sd(x0);
660
661     double ts = sd.HillClimber(
662         tolerance, m, 25, 1);
663
664     cout << "Steepest Descent Solution" << endl;
665     cout << endl;
666
667     for (int i = 0; i < n; i++)
668     {
669         if (fabs(sd.getxm()[i]) < tolerance)
670             sd.getxm()[i] = 0;
671
672         cout << "x[" << i << "] = " << sd.getxm()[i] << endl;
673     }
674
675     cout << endl;
676
```

```
677     n = 3;
678     vector<double> f1(n, 0), y0(n, 0.0), y1(n, 0);
679
680     for (int i = 0; i < n; i++)
681         y0[i] = (i + 1.0) / (n + 1.0);
682
683     NewtonsMethod nm(y0, di, NewtonFunction);
684
685     int jMax = 100;
686     long maxIt = 1000;
687
688     cout << "Newton's Method for a System" << endl;
689     cout << endl;
690     cout << "Its\t";
691     cout << "x0\t\tx1\t\tx2\t\t Norm-1\t\t\t Norm-2" << endl;
692     cout << endl;
693
694     long iterations = nm.Solve(
695         1.0e-15, f1, y1, jMax, maxIt);
696
697     return 0;
698 }
```