

TP1 Pesquisa Operacional

João Francisco Barreto da Silva Martins

May 8, 2017

simplex.c

```
1  /* Simplex Solver
2   * Developed by Joao Francisco B. S. Martins <joaofbsm@dcc.ufmg.br>
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <math.h>
9  #include <ctype.h>
10
11 #include "lalgebra.h"
12
13 int main(int argc, char* argv[]) {
14     // Linear Programming represented as a matrix similar to the tableau
15     double** lp;
16
17     // Auxiliar LP built upon the original LP
18     double** auxiliar_lp;
19
20     // Bases are column numbers ordered by rows. If a column contains the base for the first
21     // restriction(first row of A), it is going to be on the first index of base and so forth
22     int* base;
23
24     // m and n are the dimensions of the LP. auxiliar_n is the columns dimension for the auxiliar_lp.
25     // mode is the mode chosen by the user. simplex_result is the return value of the simplex algorithms
26     int m, n, auxiliar_n, mode, simplex_result;
27
28     // User choice for primal or dual simplex in mode 2
29     char simplex_type;
30
31     // Receive and discard the string "modo". We make this to facilitate the parsing
32     char string_mod0[4];
33
34     // Input file
35     FILE* input;
36     if(argc >= 2) { // Input file name has been given
37         input = fopen(argv[1], "r+");
38     }
39     else { // Default input file name
40         input = fopen("input.txt", "r+");
41     }
42
43     // Gets the modus operandi
44     fscanf(input, "%s %d", string_mod0, &mode);
45
46     // Primal or dual simplex
47     if(mode == 2) {
48         fscanf(input, " %c", &simplex_type);
49     }
```

```

50
51 fscanf(input, "%d %d ", &m, &n); // Reads LP dimensions
52 m += 1; // Adjust m so it corresponds to the tableau dimensions
53 n += 1; // Adjust n so it corresponds to the tableau dimensions
54 lp = allocate_matrix(m, n); // Allocate memory for matrix of dimensions m x n
55 parse_input(input, lp, m, n); // Fill the allocated matrix with the input
56
57 lp = format_sef(lp, m, n); // Adds the slack variables for the problem by formatting it to the standard equalities form
58 n += m - 1; // (m - 1) columns were added to the matrix
59
60 format_tableau(lp, m, n); // Negates the first row for the tableau
61
62 lp = add_operations_register(lp, m, n); // Adds the operation register matrix to the left of the LP
63 n += m - 1;
64
65 auxiliar_lp = NULL;
66 // Base will always be a vector with (m - 1) columns because this is the rank of the matrix
67 base = malloc((m - 1) * sizeof(int));
68
69 switch(mode) {
70     case 1:
71         auxiliar_lp = create_auxiliar_lp(lp, m, n);
72         auxiliar_n = n + m - 1; // Auxiliar LP creates (m - 1) new columns in A
73
74         set_initial_base(auxiliar_lp, m, auxiliar_n, base); // Set the initial base for the auxiliar LP
75
76         primal_simplex(auxiliar_lp, m, auxiliar_n, base, 0); // Runs simplex for Auxiliar LP but doesn't print the output
77
78         if(auxiliar_lp[0][auxiliar_n - 1] < 0) { // LP is infeasible
79             printf("PL inviável, aqui está um certificado ");
80             // The optimal solution for the dual of the auxiliar LP is a certificate of infeasibility for the original LP
81             print_output_vector(get_dual_optimal_solution(auxiliar_lp, m), m - 1);
82             printf("\n");
83         }
84         else {
85             // The base now is the final base of the auxiliar LP, which is a good one to begin the simplex with
86             simplex_result = primal_simplex(lp, m, n, base, 0);
87
88             if(simplex_result > 0) { // LP is unbounded
89                 printf("PL ilimitada, aqui está um certificado ");
90                 print_output_vector(generate_unboundedness_certificate(lp, m, n, simplex_result, base), (n - 1 - (m - 1) - (m - 1)));
91                 printf("\n");
92             }
93             else { // LP is optimal
94                 printf("Solução ótima x = ");
95                 print_output_vector(get_primal_optimal_solution(lp, m, n, base), (n - 1 - (m - 1) - (m - 1)));
96                 printf(", com valor objetivo %g, e solução dual y = ", round(lp[0][n - 1] * 100000) / 100000);
97                 print_output_vector(get_dual_optimal_solution(lp, m), m - 1);
98                 printf("\n");
99             }
100         }
101     break;
102
103     case 2:
104
105         switch(simplex_type) {
106             case 'P':
107                 // If b has some negative entry, use auxiliar LP to find a good base of columns to start the simplex with
108                 if(is_b_negative(lp, m, n)) {
109                     auxiliar_lp = create_auxiliar_lp(lp, m, n);
110                     auxiliar_n = n + m - 1;
111                     set_initial_base(auxiliar_lp, m, auxiliar_n, base);
112                     primal_simplex(auxiliar_lp, m, auxiliar_n, base, 0);
113                 }
114                 else {

```

```

115         // Set base columns to the slack variables
116         set_initial_base(lp, m, n, base);
117     }
118
119     simplex_result = primal_simplex(lp, m, n, base, 1);
120     break;
121
122     case 'D':
123         // We can only run the dual simplex if we have a positive c vector in the tableau.
124         // That is not entirely true, but for the scope of this assignment it will
125         if(is_c_positive(lp, m, n)) {
126             set_initial_base(lp, m, n, base);
127             simplex_result = dual_simplex(lp, m, n, base, 1);
128         }
129         else {
130             printf("Não foi possível rodar o simplex dual com a PL dada.\n");
131         }
132         break;
133
134     default:
135         printf("Erro: Opção Inválida.\n");
136     }
137     break;
138
139     default:
140         printf("Erro: Opção Inválida.\n");
141     }
142
143     fclose(input);
144
145     free(lp);
146     free(base);
147     if(auxiliar_lp != NULL) { // If it was used to solve the LP, we need to free it
148         free(auxiliar_lp);
149     }
150
151     return 0;
152 }

```

lalgebra.h

```

1  /* Linear Algebra and Simplex Algorithms Library
2  * Developed by Joao Francisco B. S. Martins <joaofbsm@dcc.ufmg.br>
3  */
4
5  #ifndef __LALGEBRA_HEADER__
6  #define __LALGEBRA_HEADER__
7
8  /***** INPUT AND OUTPUT *****/
9  void parse_input(FILE* input, double** matrix, int m, int n);
10 void print_output_vector(double* vector, int n);
11 void print_output_matrix(double** matrix, int m, int n);
12
13 /***** MATRIX OPERATIONS *****/
14 double** allocate_matrix(int m, int n);
15 double** identity(int m);
16 void print_matrix(double** matrix, int m, int n);
17 void copy_matrix(double** original, double** copy, int m, int n);
18 void insert_matrix(double** source, double** target, int from_row, int to_row, int from_column, int to_column);
19
20 void operate_on_rows(double** matrix, int row, int n, double multiply_by, int sum_to);
21 void operate_on_columns(double** matrix, int m, int column, double multiply_by, int sum_to);
22
23 double** format_sef(double** matrix, int m, int n);

```

```

24 void format_tableau(double** matrix, int m, int n);
25 double** add_operations_register(double** matrix, int m, int n);
26 double** create_auxiliar_lp(double** matrix, int m, int n);
27
28 int is_b_negative(double** matrix, int m, int n);
29 void make_b_non_negative(double** matrix, int m, int n);
30 int is_c_positive(double** matrix, int m, int n);
31
32 void set_initial_base(double** matrix, int m, int n, int* base);
33
34 int find_non_zero_element(double** matrix, int m, int column);
35 void format_canonical(double** matrix, int m, int n, int* base);
36 int primal_next_base(double** matrix, int m, int n, int* base_row, int* base_column);
37 int primal_simplex(double** matrix, int m, int n, int* base, int print_output);
38 int dual_next_base(double** matrix, int m, int n, int* base_row, int* base_column);
39 int dual_simplex(double** matrix, int m, int n, int* base, int print_output);
40
41 double* get_primal_optimal_solution(double** matrix, int m, int n, int* base);
42 double* get_dual_optimal_solution(double** matrix, int m);
43 double* generate_unboundedness_certificate(double** matrix, int m, int n, int column, int* base);
44
45 #endif

```

lalgebra.c

```

1  /* Linear Algebra and Simplex Algorithms Library
2   * Developed by Joao Francisco B. S. Martins <joaofbsm@dcc.ufmg.br>
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <math.h>
9  #include <ctype.h>
10
11  #include "lalgebra.h"
12
13  // Constant to solve floating point comparisons
14  #define EPSILON 0.000001
15
16  /***** INPUT AND OUTPUT *****/
17
18  // Parses through the input file and format the given linear programming
19  void parse_input(FILE* input, double** matrix, int m, int n) {
20      int i, j, input_size;
21
22      i = 0;
23
24      input_size = (2 + (4 * m) + (m * (9 * n))); // Approximated size of input
25      char* input_matrix = malloc(input_size * sizeof(char)); // Unformatted LP
26      fgets(input_matrix, input_size, input);
27
28      char* row = strtok(input_matrix, "{"); // Parsed first row
29
30      // Parsing of rows to get elements and fill matrix
31      while(row != NULL) {
32          char* element = row;
33          while (*element) {
34              if ((element[0] == '-' && isdigit(element[1])) || isdigit(element[0])) { // Check if char is a number
35                  double val = strtol(element, &element, 10); // Convert char to number in base 10
36                  matrix[i][j] = val; // Add element to the corresponding position in the matrix
37                  j++;
38              }
39              else {

```

```

40         element++;
41     }
42 }
43
44     row = strtok(NULL, "{");
45     i++;
46     j = 0;
47 }
48 }
49
50
51 // Prints the vector received in the format specified by the problem
52 void print_output_vector(double* vector, int n) {
53     int i;
54
55     printf("{");
56     for(i = 0; i < n; i++) {
57         printf("%g", round(vector[i] * 100000) / 100000);
58         if(i != (n - 1)) {
59             printf(", ");
60         }
61     }
62     printf("}");
63 }
64
65 // Prints the subvectors of the matrix and wrap everything up to the specified format
66 void print_output_matrix(double** matrix, int m, int n) {
67     int i;
68
69     printf("{");
70     for(i = 0; i < m; i++) {
71         print_output_vector(matrix[i], n);
72         if(i != (m - 1)) {
73             printf(", ");
74         }
75     }
76     printf("}\n\n");
77 }
78
79
80 /***** MATRIX OPERATIONS *****/
81
82 // Allocate matrix of dimensions m x n in the pointer of pointers **matrix.
83 double** allocate_matrix(int m, int n) {
84     double** matrix;
85     int i;
86
87     matrix = malloc(m * sizeof(double*));
88     for(i = 0; i < m; i++) {
89         matrix[i] = malloc(n * sizeof(double));
90     }
91
92     return matrix;
93 }
94
95 // Creates identity matrix of dimensions m x m.
96 double** identity(int m) {
97     double** identity_matrix;
98     int i, j;
99
100     identity_matrix = allocate_matrix(m, m);
101     for(i = 0; i < m; i++) {
102         for(j = 0; j < m; j++) {
103             if(i == j) {
104                 identity_matrix[i][j] = 1;

```

```

105     }
106     else {
107         identity_matrix[i][j] = 0;
108     }
109 }
110 }
111
112 return identity_matrix;
113 }
114
115 // Print matrix on the screen. This is not used by the program in its final version
116 // but was very useful during the development of it
117 void print_matrix(double** matrix, int m, int n) {
118     int i, j;
119
120     for(i = 0; i < m; i++) {
121         for(j = 0; j < n; j++) {
122             if(matrix[i][j] < 0 || matrix[i][j] > 9) {
123                 printf("%g ", round(matrix[i][j] * 100000) / 100000);
124             }
125             else {
126                 printf(" %g ", round(matrix[i][j] * 100000) / 100000);
127             }
128         }
129         printf("\n");
130     }
131     printf("\n");
132 }
133
134 // Copy the value of every element in the original matrix to the new one.
135 void copy_matrix(double** original, double** copy, int m, int n) {
136     int i, j;
137
138     for(i = 0; i < m; i++) {
139         for(j = 0; j < n; j++) {
140             copy[i][j] = original[i][j];
141         }
142     }
143 }
144
145 // Insert source matrix inside target matrix in the submatrix comprehended by the integer offsets(from_row, to_row, etc).
146 // The dimensions must match. The received indexes start at 1 so we convert them.
147 void insert_matrix(double** source, double** target, int from_row, int to_row, int from_column, int to_column) {
148     int i, j, m, n;
149
150     m = 0;
151     n = 0;
152
153     for(i = (from_row - 1); i < to_row; i++) {
154         for(j = (from_column - 1); j < to_column; j++) {
155             target[i][j] = source[m][n];
156             n++;
157         }
158         m++;
159         n = 0;
160     }
161 }
162
163 // Offers a way to make linear operations. If sum_to is -1, replaces the actual line. row stands for the row
164 // to operate on and n stands for the dimension of columns. The index for sum_to begins at 0
165 void operate_on_rows(double** matrix, int row, int n, double multiply_by, int sum_to) {
166     double* new_row;
167     int j;
168
169     new_row = malloc(n * sizeof(double));

```

```

170
171 // Solves problem for really small negative numbers causing -0 to be printed and for really big ratios to appear
172 if(fabs(multiply_by) < EPSILON) {
173     multiply_by = 0;
174 }
175
176 for(j = 0; j < n; j++) { // Create new row multiplied by multiply_by
177     if(matrix[row][j] != 0) {
178         new_row[j] = matrix[row][j] * multiply_by;
179     }
180 }
181
182 if(sum_to == -1) { // Operations should stay in the same row
183     for(j = 0; j < n; j++) {
184         matrix[row][j] = new_row[j];
185         if(fabs(matrix[row][j]) < EPSILON) {
186             matrix[row][j] = 0;
187         }
188     }
189 }
190 else { // Add created row to the specified one
191     for(j = 0; j < n; j++) {
192         matrix[sum_to][j] += new_row[j];
193         if(fabs(matrix[sum_to][j]) < EPSILON) {
194             matrix[sum_to][j] = 0;
195         }
196     }
197 }
198 }
199
200 // Offers a way to make linear operations. If sum_to is -1, replaces the actual line. m stands for the
201 // dimension of rows and column stands for the column to operate on. The index for sum_to begins at 0
202 void operate_on_columns(double** matrix, int m, int column, double multiply_by, int sum_to) {
203     double* new_column;
204     int i;
205
206     new_column = malloc(m * sizeof(double));
207
208     // Solves problem for really small negative numbers causing -0 to be printed and for really big ratios to appear
209     if(fabs(multiply_by) < EPSILON) {
210         multiply_by = 0;
211     }
212
213     for(i = 0; i < m; i++) { // Create new column multiplied by multiply_by
214         if(matrix[i][column] != 0) {
215             new_column[i] = matrix[i][column] * multiply_by;
216         }
217     }
218
219     if(sum_to == -1) { // Operations should stay in the same column
220         for(i = 0; i < m; i++) {
221             matrix[i][column] = new_column[i];
222             if(fabs(matrix[i][column]) < EPSILON) {
223                 matrix[i][column] = 0;
224             }
225         }
226     }
227     else { // Add created column to the specified one
228         for(i = 0; i < m; i++) {
229             matrix[i][sum_to] += new_column[i];
230             if(fabs(matrix[i][sum_to]) < EPSILON) {
231                 matrix[i][sum_to] = 0;
232             }
233         }
234     }

```

```

235 }
236
237 // Format the LP to the Standard Equality Form adding the slack variables
238 double** format_sef(double** original_matrix, int m, int n) {
239     double** new_matrix;
240     int i, j, new_columns, new_n;
241
242     new_columns = m - 1;
243     new_n = n + new_columns;
244
245     new_matrix = allocate_matrix(m, (n + new_columns));
246
247     // Set first row to 0
248     for(j = 0; j < new_columns; j++) {
249         new_matrix[0][j] = 0;
250     }
251
252     // Insert the original matrix in the beginning of the new one without the last column
253     insert_matrix(original_matrix, new_matrix, 1, m, 1, (n - 1));
254
255     // Adds the identity matrix in the correct position
256     insert_matrix(identity(new_columns), new_matrix, 2, m, n, (n + new_columns));
257
258     // Adds the last column of the original LP to the new one
259     for(i = 0; i < m; i++) {
260         new_matrix[i][new_n - 1] = original_matrix[i][n - 1];
261     }
262
263     // This pointer will not be used anymore
264     free(original_matrix);
265
266     return new_matrix;
267 }
268
269 // Negates the entries in the first row for the tableau
270 void format_tableau(double** matrix, int m, int n) {
271     int i;
272     for(i = 0; i < n; i++) {
273         if(matrix[0][i] != 0) {
274             matrix[0][i] = matrix[0][i] * -1;
275         }
276     }
277 }
278
279 // Create a new matrix that consists of the operation register submatrix added to the left of the original one.
280 double** add_operations_register(double** original_matrix, int m, int n) {
281     double** new_matrix;
282     int j, new_columns;
283
284     new_columns = m - 1; // Number of rows added
285
286     new_matrix = allocate_matrix(m, (n + new_columns));
287
288     // Set first row to 0
289     for(j = 0; j < new_columns; j++) {
290         new_matrix[0][j] = 0;
291     }
292
293     // Adds the identity matrix in the correct position
294     insert_matrix(identity(new_columns), new_matrix, 2, m, 1, new_columns);
295
296     // Insert the original matrix as a submatrix of the new one
297     insert_matrix(original_matrix, new_matrix, 1, m, m, (n + new_columns));
298
299     free(original_matrix);

```



```

300
301     return new_matrix;
302 }
303
304 // Creates the auxiliar LP in the correct format
305 double** create_auxiliar_lp(double** matrix, int m, int n) {
306     double** auxiliar_lp;
307     double** copied_matrix;
308     int i, j, auxiliar_n;
309
310     auxiliar_n = n + m - 1;
311     auxiliar_lp = allocate_matrix(m, auxiliar_n);
312
313     copied_matrix = allocate_matrix(m, n);
314     // Holds the values of the original matrix but doesn't mess with the original data in any sense
315     copy_matrix(matrix, copied_matrix, m, n);
316
317     make_b_non_negative(copied_matrix, m, n); // Makes b non negative before adding the new columns of the auxiliar LP
318
319     // Fulfill the auxiliar lp without the last column(thats why to_column equals n - 1)
320     insert_matrix(copied_matrix, auxiliar_lp, 1, m, 1, (n - 1));
321
322     // Adds the last column to the auxiliar lp
323     for(i = 0; i < m; i++) {
324         auxiliar_lp[i][auxiliar_n - 1] = copied_matrix[i][n - 1];
325     }
326
327     // Creates the first row of the auxiliar LP with -1(1 in the tableau) above the new columns
328     for(j = 0; j < auxiliar_n; j++) {
329         if(j >= (auxiliar_n - m) && j < (auxiliar_n - 1)) {
330             auxiliar_lp[0][j] = 1;
331         }
332         else {
333             auxiliar_lp[0][j] = 0;
334         }
335     }
336
337     // Adds the identity matrix below the 1's in the first row
338     insert_matrix(identity(m - 1), auxiliar_lp, 2, m, (auxiliar_n - m + 1), (auxiliar_n - 1));
339
340     return auxiliar_lp;
341 }
342
343 // Check if b has any negative element
344 int is_b_negative(double** matrix, int m, int n) {
345     int i;
346
347     for(i = 1; i < m; i++) {
348         if(matrix[i][n - 1] < 0) {
349             return 1;
350         }
351     }
352
353     return 0;
354 }
355
356 // For rows where b is negative multiply the entire row by -1
357 void make_b_non_negative(double** matrix, int m, int n) {
358     int i;
359
360     for(i = 1; i < m; i++) {
361         if(matrix[i][n - 1] < 0) {
362             operate_on_rows(matrix, i, n, -1, -1);
363         }
364     }

```

```

365 }
366
367 // Check if c is entirely positive(0 is positive)
368 int is_c_positive(double** matrix, int m, int n) {
369     int j;
370
371     for(j = 0; j < (n - 1); j++) {
372         if(matrix[0][j] < 0) {
373             return 0;
374         }
375     }
376
377     return 1;
378 }
379
380
381 // Set the base to the default: Last (m - 1) columns of the A matrix
382 void set_initial_base(double** matrix, int m, int n, int* base) {
383     int b, i;
384     b = (n - 1 - (m - 1));
385     for(i = 0; i < (m - 1); i++) {
386         base[i] = b;
387         b++;
388     }
389 }
390
391 // Finds first non zero element on received column and returns it's index
392 int find_non_zero_element(double** matrix, int m, int column) {
393     int i;
394
395     for(i = 1; i < m; i++) {
396         if(matrix[i][column] != 0) {
397             return i;
398         }
399     }
400
401     return 0;
402 }
403
404 // Format the LP to the Canonical Form
405 void format_canonical(double** matrix, int m, int n, int* base) {
406     int i, j;
407
408     for(i = 0; i < (m - 1); i++) { // Goes through all the basic columns
409         // If the row for the base is 0, find other row on the same column that can make the first one != 0
410         if(matrix[i + 1][base[i]] == 0) {
411             operate_on_rows(matrix, (find_non_zero_element(matrix, m, base[i])), n, 1, (i + 1));
412         }
413         if(matrix[i + 1][base[i]] != 1) { // Make element equals 1
414             operate_on_rows(matrix, (i + 1), n, (1 / matrix[i + 1][base[i]]), -1);
415         }
416         for(j = 0; j < m; j++) { // Make all the other elements in that column equals 0
417             if(j != (i + 1)) {
418                 if(matrix[j][base[i]] != 0) {
419                     operate_on_rows(matrix, (i + 1), n, (-1 * (matrix[j][base[i]] / matrix[i + 1][base[i]])), j);
420                 }
421             }
422         }
423     }
424 }
425
426 // Returns if LP is unbounded(column number), optimal(-1) or if we need one more round of simplex(0).
427 // Uses Bland's Rule to prevent loops. Pass the row and column of the base by reference
428 int primal_next_base(double** matrix, int m, int n, int* base_row, int* base_column) {
429     int i, j;

```

```

430     double min_ratio, row_ratio;
431
432     min_ratio = 999999;
433
434     for(j = (m - 1); j < (n - 1); j++) { // Skips the operation register columns
435         if(matrix[0][j] < 0) { // Chooses the first negative element in the first row
436             *base_column = j;
437             for(i = 1; i < m; i++) {
438                 // b will never be negative after primal simplex starts to run, so,
439                 // for a valid ratio, we need a positive number that is not zero
440                 if(matrix[i][j] > 0) {
441                     row_ratio = matrix[i][n - 1] / matrix[i][j];
442                     if(row_ratio <= min_ratio + EPSILON) {
443                         min_ratio = row_ratio;
444                         *base_row = i; // Chooses row with minimum ratio in that column
445                     }
446                 }
447             }
448             if(min_ratio == 999999) {
449                 return j; // LP is unbounded
450             }
451             else {
452                 return 0; // Goes to the next round of simplex
453             }
454         }
455     }
456
457     return -1; // LP is optimal
458 }
459
460 // If return == -1 the LP is optimal and if return > 0 it's unbounded. In the last case,
461 // the return value equals the column where we can get the certificate of unboundedness
462 int primal_simplex(double** matrix, int m, int n, int* base, int print_output) {
463     int result, new_base_row, new_base_column;
464
465     make_b_non_negative(matrix, m, n);
466
467     while(1) {
468         // Reset variables to prevent garbage
469         new_base_row = 0;
470         new_base_column = 0;
471
472         // First we need to present the LP in the canonical form
473         format_canonical(matrix, m, n, base);
474
475         if(print_output) { // Print the current tableau if flag is set
476             print_output_matrix(matrix, m, n);
477         }
478
479         // Find the next base for the primal simplex
480         result = primal_next_base(matrix, m, n, &new_base_row, &new_base_column);
481
482         if(result != 0) { // If LP is optimal or unbounded
483             return result;
484         }
485
486         base[new_base_row - 1] = new_base_column; // Adds chosen column to the base
487     }
488 }
489
490
491 // Returns if LP is unbounded(column number), optimal(-1) or if we need one more round of simplex(0).
492 // Uses Bland's Rule to prevent loops. Pass the row and column of the base by reference
493 int dual_next_base(double** matrix, int m, int n, int* base_row, int* base_column) {

```

```

495     int i, j;
496     double min_ratio, row_ratio;
497
498     min_ratio = 999999;
499
500     for(i = 1; i < m; i++) {
501         if(matrix[i][n - 1] < 0) {
502             *base_row = i;
503             for(j = (m - 1); j < (n - 1); j++) {
504                 if(matrix[i][j] < 0) {
505                     row_ratio = matrix[0][j] / (-1 * matrix[i][j]);
506                     if(row_ratio >= 0 && row_ratio < min_ratio) {
507                         min_ratio = row_ratio;
508                         *base_column = j;
509                     }
510                 }
511             }
512             if(min_ratio == 999999) {
513                 return j; // LP is unbounded
514             }
515             else {
516                 return 0; // Goes to the next round of simplex
517             }
518         }
519     }
520
521     return -1; // LP is optimal
522 }
523
524 // If return == -1 the LP is optimal and if return > 0 it's unbounded. In the last case,
525 // the return value equals the column where we can get the certificate of unboundedness
526 int dual_simplex(double** matrix, int m, int n, int* base, int print_output) {
527     int result, new_base_row, new_base_column;
528
529     while(1) {
530
531         // Reset variables to prevent garbage
532         new_base_row = 0;
533         new_base_column = 0;
534
535         // First we need to present the LP in the canonical form
536         format_canonical(matrix, m, n, base);
537
538         if(print_output) { // Print the current tableau if flag is set
539             print_output_matrix(matrix, m, n);
540         }
541
542         // Find the next base for the primal simplex
543         result = dual_next_base(matrix, m, n, &new_base_row, &new_base_column);
544
545         if(result != 0) { // If LP is optimal or unbounded
546             return result;
547         }
548
549         base[new_base_row - 1] = new_base_column; // Adds chosen column to the base
550     }
551 }
552
553 // Extract solution from optimal LP based in the base of columns
554 double* get_primal_optimal_solution(double** matrix, int m, int n, int* base) {
555     double* vector;
556     int i;
557
558     vector = malloc((n - 1 - (m - 1)) * sizeof(double));
559

```

```

560     for(i = 0; i < (n - 1 - (m - 1)); i++) { // Solution is zero in columns that are not in the base
561         vector[i] = 0;
562     }
563
564     // Assigns the value of b to the columns in the solution that correspond to the columns in the base
565     for(i = 0; i < (m - 1); i++) {
566         vector[base[i] - (m - 1)] = matrix[i + 1][n - 1];
567     }
568
569     return vector;
570 }
571
572 // Extract dual optimal solution which can be used as infeasibility or optimality certificates
573 double* get_dual_optimal_solution(double** matrix, int m) {
574     double* vector;
575     int i;
576
577     vector = malloc((m - 1) * sizeof(double));
578
579     for(i = 0; i < (m - 1); i++) { // Group the elements of the operations register that are in the first row
580         vector[i] = matrix[0][i];
581     }
582
583     return vector;
584 }
585
586 double* generate_unboundedness_certificate(double** matrix, int m, int n, int column, int* base) {
587     double* vector;
588     int i;
589
590     vector = malloc((n - 1 - (m - 1)) * sizeof(double));
591
592     for(i = 0; i < (n - 1 - (m - 1)); i++) {
593         vector[i] = 0;
594     }
595
596     // Column that shows unboundedness will be 1 to make it easier to create the rest of the certificate
597     vector[column - (m - 1)] = 1;
598
599     // Assigns the value of -1 * element in the "unbounded column" to the columns
600     // in the certificate that correspond to the columns in the base
601     for(i = 0; i < (m - 1); i++) {
602         if(matrix[i + 1][column] != 0) {
603             vector[base[i] - (m - 1)] = -1 * matrix[i + 1][column];
604         }
605     }
606
607     return vector;
608 }

```
