TP1 Pesquisa Operacional

João Francisco Barreto da Silva Martins

May 8, 2017

simplex.c

```c
/* Simplex Solver
 * Developed by Joao Francisco B. S. Martins <joaofbsm@dcc.ufmg.br>
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>

#include "lalgebra.h"

int main(int argc, char* argv[]) {
  // Linear Programming represented as a matrix similar to the tableau
  double** lp;

  // Auxiliar LP built upon the original LP
  double** auxiliar_lp;

  // Bases are column numbers ordered by rows. If a column contains the base for the first
  // restriction(first row of A), it is going to be on the first index of base and so forth
  int* base;

  // m and n are the dimensions of the LP. auxiliar_n is the columns dimension for the auxiliar_lp.
  // mode is the mode chosen by the user. simplex_result is the return value of the simplex algorithms
  int m, n, auxiliar_n, mode, simplex_result;

  // User choice for primal or dual simplex in mode 2
  char simplex_type;

  // Receive and discard the string "modo". We make this to facilitate the parsing
  char string_modo[4];

  // Input file
  FILE* input;
  if(argc >= 2) { // Input file name has been given
     input = fopen(argv[1], "r+");
  }
  else { // Default input file name
    input = fopen("input.txt", "r+");
  }

  // Gets the modus operandi
  fscanf(input, "%s %d", string_modo, &mode);

  // Primal or dual simplex
  if(mode == 2) {
    fscanf(input, " %c", &simplex_type);
  }
```

```c
  fscanf(input, "%d %d ", &m, &n); // Reads LP dimensions
  m += 1; // Adjust m so it corresponds to the tableau dimensions
  n += 1; // Adjust n so it corresponds to the tableau dimensions
  lp = allocate_matrix(m, n); // Allocate memory for matrix of dimensions m x n
  parse_input(input, lp, m, n); // Fill the allocated matrix with the input

  lp = format_sef(lp, m, n); // Adds the slack variables for the problem by formating it to the standard equalities form
  n += m - 1; // (m - 1) columns were added to the matrix

  format_tableau(lp, m, n); // Negates the first row for the tableau

  lp = add_operations_register(lp, m, n); // Adds the operation register matrix to the left of the LP
  n += m - 1;

  auxiliar_lp = NULL;
  // Base will always be a vector with (m - 1) columns because this is the rank of the matrix
  base = malloc((m - 1) * sizeof(int));

  switch(mode) {
    case 1:
      auxiliar_lp = create_auxiliar_lp(lp, m, n);
      auxiliar_n = n + m - 1; // Auxiliar LP creates (m - 1) new columns in A

      set_initial_base(auxiliar_lp, m, auxiliar_n, base); // Set the initial base for the auxiliar LP

      primal_simplex(auxiliar_lp, m, auxiliar_n, base, 0); // Runs simplex for Auxiliar LP but doesn't print the output

      if(auxiliar_lp[0][auxiliar_n - 1] < 0) { // LP is infeasible
        printf("PL inviável, aqui está um certificado ");
        // The optimal solution for the dual of the auxiliar LP is a certificate of infeasibility for the original LP
        print_output_vector(get_dual_optimal_solution(auxiliar_lp, m), m - 1);
        printf("\n");
      }
      else {
        // The base now is the final base of the auxiliar LP, which is a good one to begin the simplex with
        simplex_result = primal_simplex(lp, m, n, base, 0);

        if(simplex_result > 0) { // LP is unbounded
          printf("PL ilimitada, aqui está um certificado ");
          print_output_vector(generate_unboundedness_certificate(lp, m, n, simplex_result, base), (n - 1 - (m - 1) - (m - 1)));
          printf("\n");
        }
        else { // LP is optimal
          printf("Solução ótima x = ");
          print_output_vector(get_primal_optimal_solution(lp, m, n, base), (n - 1 - (m - 1) - (m - 1)));
          printf(", com valor objetivo %g, e solução dual y = ", round(lp[0][n - 1] * 100000) / 100000);
          print_output_vector(get_dual_optimal_solution(lp, m), m - 1);
          printf("\n");
        }
      }
    break;

    case 2:

      switch(simplex_type) {
        case 'P':
          // If b has some negative entry, use auxiliar LP to find a good base of columns to start the simplex with
          if(is_b_negative(lp, m, n)) {
            auxiliar_lp = create_auxiliar_lp(lp, m, n);
            auxiliar_n = n + m - 1;
            set_initial_base(auxiliar_lp, m, auxiliar_n, base);
            primal_simplex(auxiliar_lp, m, auxiliar_n, base, 0);
          }
          else {
```

```
            // Set base columns to the slack variables
            set_initial_base(lp, m, n, base);
          }

          simplex_result = primal_simplex(lp, m, n, base, 1);
        break;

        case 'D':
          // We can only run the dual simplex if we have a positive c vector in the tableau.
          // That is not entirely true, but for the scope of this assignment it will
          if(is_c_positive(lp, m, n)) {
            set_initial_base(lp, m, n, base);
            simplex_result = dual_simplex(lp, m, n, base, 1);
          }
          else {
            printf("Não foi possível rodar o simplex dual com a PL dada.\n");
          }
        break;

        default:
        printf("Erro: Opção Inválida.\n");
      }
    break;

    default:
    printf("Erro: Opção Inválida.\n");
  }

  fclose(input);

  free(lp);
  free(base);
  if(auxiliar_lp != NULL) { // If it was used to solve the LP, we need to free it
    free(auxiliar_lp);
  }

  return 0;
}
```

lalgebra.h

```
/* Linear Algebra and Simplex Algorithms Library
 * Developed by Joao Francisco B. S. Martins <joaofbsm@dcc.ufmg.br>
 */

#ifndef __LALGEBRA_HEADER__
#define __LALGEBRA_HEADER__

/***** INPUT AND OUTPUT ******/
void parse_input(FILE* input, double** matrix, int m, int n);
void print_output_vector(double* vector, int n);
void print_output_matrix(double** matrix, int m, int n);

/***** MATRIX OPERATIONS ******/
double** allocate_matrix(int m, int n);
double** identity(int m);
void print_matrix(double** matrix, int m, int n);
void copy_matrix(double** original, double** copy, int m, int n);
void insert_matrix(double** source, double** target, int from_row, int to_row, int from_column, int to_column);

void operate_on_rows(double** matrix, int row, int n, double multiply_by, int sum_to);
void operate_on_columns(double** matrix, int m, int column, double multiply_by, int sum_to);

double** format_sef(double** matrix, int m, int n);
```

```c
void format_tableau(double** matrix, int m, int n);
double** add_operations_register(double** matrix, int m, int n);
double** create_auxiliar_lp(double** matrix, int m, int n);

int is_b_negative(double** matrix, int m, int n);
void make_b_non_negative(double** matrix, int m, int n);
int is_c_positive(double** matrix, int m, int n);

void set_initial_base(double** matrix, int m, int n, int* base);

int find_non_zero_element(double** matrix, int m, int column);
void format_canonical(double** matrix, int m, int n, int* base);
int primal_next_base(double** matrix, int m, int n, int* base_row, int* base_column);
int primal_simplex(double** matrix, int m, int n, int* base, int print_output);
int dual_next_base(double** matrix, int m, int n, int* base_row, int* base_column);
int dual_simplex(double** matrix, int m, int n, int* base, int print_output);

double* get_primal_optimal_solution(double** matrix, int m, int n, int* base);
double* get_dual_optimal_solution(double** matrix, int m);
double* generate_unboundedness_certificate(double** matrix, int m, int n, int column, int* base);

#endif
```

lalgebra.c

```c
/* Linear Algebra and Simplex Algorithms Library
 * Developed by Joao Francisco B. S. Martins <joaofbsm@dcc.ufmg.br>
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>

#include "lalgebra.h"

// Constant to solve floating point comparisons
#define EPSILON 0.000001

/***** INPUT AND OUTPUT ******/

// Parses through the input file and format the given linear programming
void parse_input(FILE* input, double** matrix, int m, int n) {
  int i, j, input_size;

  i = 0;

  input_size = (2 + (4 * m) + (m * (9 * n))); // Approximated size of input
  char* input_matrix = malloc(input_size * sizeof(char)); // Unformated LP
  fgets(input_matrix, input_size, input);

  char* row = strtok(input_matrix, "{"); // Parsed first row

  // Parsing of rows to get elements and fill matrix
  while(row != NULL) {
    char* element = row;
    while (*element) {
      if ((element[0] == '-' && isdigit(element[1])) || isdigit(element[0])) { // Check if char is a number
        double val = strtol(element, &element, 10); // Convert char to number in base 10
        matrix[i][j] = val; // Add element to the corresponding position in the matrix
        j++;
      }
      else {
```

```
        element++;
      }
    }

    row = strtok(NULL, "{");
    i++;
    j = 0;
  }
}


// Prints the vector received in the format specified by the problem
void print_output_vector(double* vector, int n) {
  int i;

  printf("{");
  for(i = 0; i < n; i++) {
    printf("%g", round(vector[i] * 100000) / 100000);
    if(i != (n - 1)) {
      printf(", ");
    }
  }
  printf("}");
}

// Prints the subvectors of the matrix and wrap everything up to the specified format
void print_output_matrix(double** matrix, int m, int n) {
  int i;

  printf("{");
  for(i = 0; i < m; i++) {
    print_output_vector(matrix[i], n);
    if(i != (m - 1)) {
      printf(", ");
    }
  }
  printf("}\n\n");
}


/***** MATRIX OPERATIONS ******/

// Allocate matrix of dimensions m x n in the pointer of pointers **matrix.
double** allocate_matrix(int m, int n) {
  double** matrix;
  int i;

  matrix = malloc(m * sizeof(double*));
  for(i = 0; i < m; i++) {
    matrix[i] = malloc(n * sizeof(double));
  }

  return matrix;
}

// Creates identity matrix of dimensions m x m.
double** identity(int m) {
  double** identity_matrix;
  int i, j;

  identity_matrix = allocate_matrix(m, m);
  for(i = 0; i < m; i++) {
    for(j = 0; j < m; j++) {
      if(i == j) {
        identity_matrix[i][j] = 1;
```

```c
      }
      else {
        identity_matrix[i][j] = 0;
      }
    }
  }

  return identity_matrix;
}

// Print matrix on the screen. This is not used by the program in its final version
// but was very useful during the development of it
void print_matrix(double** matrix, int m, int n) {
  int i, j;

  for(i = 0; i < m; i++) {
    for(j = 0; j < n; j++) {
      if(matrix[i][j] < 0 || matrix[i][j] > 9) {
        printf("%g ", round(matrix[i][j] * 100000) / 100000);
      }
      else {
        printf(" %g ", round(matrix[i][j] * 100000) / 100000);
      }
    }
    printf("\n");
  }
  printf("\n");
}

// Copy the value of every element in the original matrix to the new one.
void copy_matrix(double** original, double** copy, int m, int n) {
  int i, j;

  for(i = 0; i < m; i++) {
    for(j = 0; j < n; j++) {
      copy[i][j] = original[i][j];
    }
  }
}

// Insert source matrix inside target matrix in the submatrix comprehended by the integer offsets(from_row, to_row, etc).
// The dimensions must match. The received indexes start at 1 so we convert them.
void insert_matrix(double** source, double** target, int from_row, int to_row, int from_column, int to_column) {
  int i, j, m, n;

  m = 0;
  n = 0;

  for(i = (from_row - 1); i < to_row; i++) {
    for(j = (from_column - 1); j < to_column; j++) {
      target[i][j] = source[m][n];
      n++;
    }
    m++;
    n = 0;
  }
}

// Offers a way to make linear operations. If sum_to is -1, replaces the actual line. row stands for the row
// to operate on and n stands for the dimension of columns. The index for sum_to begins at 0
void operate_on_rows(double** matrix, int row, int n, double multiply_by, int sum_to) {
  double* new_row;
  int j;

  new_row = malloc(n * sizeof(double));
```

```c

  // Solves problem for really small negative numbers causing -0 to be printed and for really big ratios to appear
  if(fabs(multiply_by) < EPSILON) {
    multiply_by = 0;
  }

  for(j = 0; j < n; j++) { // Create new row multiplied by multiply_by
    if(matrix[row][j] != 0) {
      new_row[j] = matrix[row][j] * multiply_by;
    }
  }

  if(sum_to == -1) { // Operations should stay in the same row
    for(j = 0; j < n; j++) {
      matrix[row][j] = new_row[j];
      if(fabs(matrix[row][j]) < EPSILON) {
        matrix[row][j] = 0;
      }
    }
  }
  else { // Add created row to the specified one
    for(j = 0; j < n; j++) {
      matrix[sum_to][j] += new_row[j];
      if(fabs(matrix[sum_to][j]) < EPSILON) {
        matrix[sum_to][j] = 0;
      }
    }
  }
}

// Offers a way to make linear operations. If sum_to is -1, replaces the actual line. m stands for the
// dimension of rows and column stands for the column to operate on. The index for sum_to begins at 0
void operate_on_columns(double** matrix, int m, int column, double multiply_by, int sum_to) {
  double* new_column;
  int i;

  new_column = malloc(m * sizeof(double));

  // Solves problem for really small negative numbers causing -0 to be printed and for really big ratios to appear
  if(fabs(multiply_by) < EPSILON) {
    multiply_by = 0;
  }

  for(i = 0; i < m; i++) { // Create new column multiplied by multiply_by
    if(matrix[i][column] != 0) {
      new_column[i] = matrix[i][column] * multiply_by;
    }
  }

  if(sum_to == -1) { // Operations should stay in the same column
    for(i = 0; i < m; i++) {
      matrix[i][column] = new_column[i];
      if(fabs(matrix[i][column]) < EPSILON) {
        matrix[i][column] = 0;
      }
    }
  }
  else { // Add created column to the specified one
    for(i = 0; i < m; i++) {
      matrix[i][sum_to] += new_column[i];
      if(fabs(matrix[i][sum_to]) < EPSILON) {
        matrix[i][sum_to] = 0;
      }
    }
  }
```

```c
}

// Format the LP to the Standard Equality Form adding the slack variables
double** format_sef(double** original_matrix, int m, int n) {
  double** new_matrix;
  int i, j, new_columns, new_n;

  new_columns = m - 1;
  new_n = n + new_columns;

  new_matrix = allocate_matrix(m, (n + new_columns));

  // Set first row to 0
  for(j = 0; j < new_columns; j++) {
    new_matrix[0][j] = 0;
  }

  // Insert the original matrix in the beginning of the new one without the last column
  insert_matrix(original_matrix, new_matrix, 1, m, 1, (n - 1));

  // Adds the identity matrix in the correct position
  insert_matrix(identity(new_columns), new_matrix, 2, m, n, (n + new_columns));

  // Adds the last column of the original LP to the new one
  for(i = 0; i < m; i++) {
    new_matrix[i][new_n - 1] = original_matrix[i][n - 1];
  }

  // This pointer will not be used anymore
  free(original_matrix);

  return new_matrix;
}

// Negates the entries in the first row for the tableau
void format_tableau(double** matrix, int m, int n) {
  int i;
  for(i = 0; i < n; i++) {
    if(matrix[0][i] != 0) {
      matrix[0][i] = matrix[0][i] * -1;
    }
  }
}

// Create a new matrix that consists of the operation register submatrix added to the left of the original one.
double** add_operations_register(double** original_matrix, int m, int n) {
  double** new_matrix;
  int j, new_columns;

  new_columns = m - 1; // Number of rows added

  new_matrix = allocate_matrix(m, (n + new_columns));

  // Set first row to 0
  for(j = 0; j < new_columns; j++) {
    new_matrix[0][j] = 0;
  }

  // Adds the identity matrix in the correct position
  insert_matrix(identity(new_columns), new_matrix, 2, m, 1, new_columns);

  // Insert the original matrix as a submatrix of the new one
  insert_matrix(original_matrix, new_matrix, 1, m, m, (n + new_columns));

  free(original_matrix);
```

```c

  return new_matrix;
}

// Creates the auxiliar LP in the correct format
double** create_auxiliar_lp(double** matrix, int m, int n) {
  double** auxiliar_lp;
  double** copied_matrix;
  int i, j, auxiliar_n;

  auxiliar_n = n + m - 1;
  auxiliar_lp = allocate_matrix(m, auxiliar_n);

  copied_matrix = allocate_matrix(m, n);
  // Holds the values of the original matrix but doesn't mess with the original data in any sense
  copy_matrix(matrix, copied_matrix, m, n);

  make_b_non_negative(copied_matrix, m, n); // Makes b non negative before adding the new columns of the auxiliar LP

  // Fulfill the auxiliar lp without the last column(thats why to_column equals n - 1)
  insert_matrix(copied_matrix, auxiliar_lp, 1, m, 1, (n - 1));

  // Adds the last column to the auxiliar lp
  for(i = 0; i < m; i++) {
    auxiliar_lp[i][auxiliar_n - 1] = copied_matrix[i][n - 1];
  }

  // Creates the first row of the auxiliar LP with -1(1 in the tableau) above the new columns
  for(j = 0; j < auxiliar_n; j++) {
    if(j >= (auxiliar_n - m) && j < (auxiliar_n - 1)) {
      auxiliar_lp[0][j] = 1;
    }
    else {
      auxiliar_lp[0][j] = 0;
    }
  }

  // Adds the identity matrix below the 1's in the first row
  insert_matrix(identity(m - 1), auxiliar_lp, 2, m, (auxiliar_n - m + 1), (auxiliar_n - 1));

  return auxiliar_lp;
}

// Check if b has any negative element
int is_b_negative(double** matrix, int m, int n) {
  int i;

  for(i = 1; i < m; i++) {
    if(matrix[i][n - 1] < 0) {
      return 1;
    }
  }

  return 0;
}

// For rows where b is negative multiply the entire row by -1
void make_b_non_negative(double** matrix, int m, int n) {
  int i;

  for(i = 1; i < m; i++) {
    if(matrix[i][n - 1] < 0) {
      operate_on_rows(matrix, i, n, -1, -1);
    }
  }
```

```
}

// Check if c is entirely positive(0 is positive)
int is_c_positive(double** matrix, int m, int n) {
  int j;

  for(j = 0; j < (n - 1); j++) {
    if(matrix[0][j] < 0) {
      return 0;
    }
  }

  return 1;
}


// Set the base to the default: Last (m - 1) columns of the A matrix
void set_initial_base(double** matrix, int m, int n, int* base) {
  int b, i;
  b = (n - 1 - (m - 1));
  for(i = 0; i < (m - 1); i++) {
    base[i] = b;
    b++;
  }
}

// Finds first non zero element on received column and returns it's index
int find_non_zero_element(double** matrix, int m, int column) {
  int i;

  for(i = 1; i < m; i++) {
    if(matrix[i][column] != 0) {
      return i;
    }
  }

  return 0;
}

// Format the LP to the Canonical Form
void format_canonical(double** matrix, int m, int n, int* base) {
  int i, j;

  for(i = 0; i < (m - 1); i++) { // Goes through all the basic columns
    // If the row for the base is 0, find other row on the same column that can make the first one != 0
    if(matrix[i + 1][base[i]] == 0) {
      operate_on_rows(matrix, (find_non_zero_element(matrix, m, base[i])), n, 1, (i + 1));
    }
    if(matrix[i + 1][base[i]] != 1) { // Make element equals 1
      operate_on_rows(matrix, (i + 1), n, (1 / matrix[i + 1][base[i]]), -1);
    }
    for(j = 0; j < m; j++) { // Make all the other elements in that column equals 0
      if(j != (i + 1)) {
        if(matrix[j][base[i]] != 0) {
          operate_on_rows(matrix, (i + 1), n, (-1 * (matrix[j][base[i]] / matrix[i + 1][base[i]])), j);
        }
      }
    }
  }
}

// Returns if LP is unbounded(column number), optimal(-1) or if we need one more round of simplex(0).
// Uses Bland's Rule to prevent loops. Pass the row and column of the base by reference
int primal_next_base(double** matrix, int m, int n, int* base_row, int* base_column) {
  int i, j;
```

```
  double min_ratio, row_ratio;

  min_ratio = 999999;

  for(j = (m - 1); j < (n - 1); j++) { // Skips the operation register columns
    if(matrix[0][j] < 0) { // Chooses the first negative element in the first row
      *base_column = j;
      for(i = 1; i < m; i++) {
        // b will never be negative after primal simplex starts to run, so,
        // for a valid ratio, we need a positive number that is not zero
        if(matrix[i][j] > 0) {
          row_ratio = matrix[i][n - 1] / matrix[i][j];
          if(row_ratio <= min_ratio + EPSILON) {
            min_ratio = row_ratio;
            *base_row = i; // Chooses row with minimum ratio in that column
          }
        }
      }
      if(min_ratio == 999999) {
        return j; // LP is unbounded
      }
      else {
        return 0; // Goes to the next round of simplex
      }
    }
  }

  return -1; // LP is optimal
}

// If return == -1 the LP is optimal and if return > 0 it's unbounded. In the last case,
// the return value equals the column where we can get the certificate of unboundedness
int primal_simplex(double** matrix, int m, int n, int* base, int print_output) {
  int result, new_base_row, new_base_column;

  make_b_non_negative(matrix, m, n);

  while(1) {

    // Reset variables to prevent garbage
    new_base_row = 0;
    new_base_column = 0;

    // First we need to present the LP in the canonical form
    format_canonical(matrix, m, n, base);

    if(print_output) { // Print the current tableau if flag is set
      print_output_matrix(matrix, m, n);
    }

    // Find the next base for the primal simplex
    result = primal_next_base(matrix, m, n, &new_base_row, &new_base_column);

    if(result != 0) { // If LP is optimal or unbounded
      return result;
    }

    base[new_base_row - 1] = new_base_column; // Adds chosen column to the base
  }
}


// Returns if LP is unbounded(column number), optimal(-1) or if we need one more round of simplex(0).
// Uses Bland's Rule to prevent loops. Pass the row and column of the base by reference
int dual_next_base(double** matrix, int m, int n, int* base_row, int* base_column) {
```

```c
  int i, j;
  double min_ratio, row_ratio;

  min_ratio = 999999;

  for(i = 1; i < m; i++) {
    if(matrix[i][n - 1] < 0) {
      *base_row = i;
      for(j = (m - 1); j < (n - 1); j++) {
        if(matrix[i][j] < 0) {
          row_ratio = matrix[0][j] / (-1 * matrix[i][j]);
          if(row_ratio >= 0 && row_ratio < min_ratio) {
            min_ratio = row_ratio;
            *base_column = j;
          }
        }
      }
      if(min_ratio == 999999) {
        return j; // LP is unbounded
      }
      else {
        return 0; // Goes to the next round of simplex
      }
    }
  }

  return -1; // LP is optimal
}

// If return == -1 the LP is optimal and if return > 0 it's unbounded. In the last case,
// the return value equals the column where we can get the certificate of unboundedness
int dual_simplex(double** matrix, int m, int n, int* base, int print_output) {
  int result, new_base_row, new_base_column;

  while(1) {

    // Reset variables to prevent garbage
    new_base_row = 0;
    new_base_column = 0;

    // First we need to present the LP in the canonical form
    format_canonical(matrix, m, n, base);

    if(print_output) { // Print the current tableau if flag is set
      print_output_matrix(matrix, m, n);
    }

    // Find the next base for the primal simplex
    result = dual_next_base(matrix, m, n, &new_base_row, &new_base_column);

    if(result != 0) { // If LP is optimal or unbounded
      return result;
    }

    base[new_base_row - 1] = new_base_column; // Adds chosen column to the base
  }
}

// Extract solution from optimal LP based in the base of columns
double* get_primal_optimal_solution(double** matrix, int m, int n, int* base) {
  double* vector;
  int i;

  vector = malloc((n - 1 - (m - 1)) * sizeof(double));

```

```c
  for(i = 0; i < (n - 1 - (m - 1)); i++) { // Solution is zero in columns that are not in the base
    vector[i] = 0;
  }

  // Assigns the value of b to the columns in the solution that correspond to the columns in the base
  for(i = 0; i < (m - 1); i++) {
    vector[base[i] - (m - 1)]  = matrix[i + 1][n - 1];
  }

  return vector;
}

// Extract dual optimal solution which can be used as infeasibility or optimality certificates
double* get_dual_optimal_solution(double** matrix, int m) {
  double* vector;
  int i;

  vector = malloc((m - 1) * sizeof(double));

  for(i = 0; i < (m - 1); i++) { // Group the elements of the operations register that are in the first row
    vector[i] = matrix[0][i];
  }

  return vector;
}

double* generate_unboundedness_certificate(double** matrix, int m, int n, int column, int* base) {
  double* vector;
  int i;

  vector = malloc((n - 1 - (m - 1)) * sizeof(double));

  for(i = 0; i < (n - 1 - (m - 1)); i++) {
    vector[i] = 0;
  }

  // Column that shows unboundedness will be 1 to make it easier to create the rest of the certificate
  vector[column - (m - 1)] = 1;

  // Assigns the value of -1 * element in the "unbounded column" to the columns
  // in the certificate that correspond to the columns in the base
  for(i = 0; i < (m - 1); i++) {
    if(matrix[i + 1][column] != 0) {
      vector[base[i] - (m - 1)] = -1 * matrix[i + 1][column];
    }
  }

  return vector;
}
```