Code for solving Tetravex using Douglas Rachford algorithm

Pulkit Bansal [1]

July 21, 2010

[1]Supervised by Prof. Heinz Bauschke and Prof. Shawn Wang

Contents

Chapter 1

Documentation

The aim of this Documentation is not to explain the modeling behind the puzzle but just to explain how the code works. This documentation contains everything you need to know about how to use this program in the best way. The version of code(see Appendix) provided here will work for both Matlab and Octave. The program works much faster in Matlab as compared to Octave.

1.1 Brief Modeling and Terminology

We have assumed that we are given an $n \times n$ Tetravex puzzle. We will call n as order of the puzzle. The code works pretty well for 3 x 3 and 4 x 4 Tetravex and on an average they are solved in 40 and 200 iterations repectively. The modeling of Tetravex is done in terms of an $n^2 \times n^2$ matrix which we will call *Representation-Matrix*. Each tile is given a number between 1 to n^2 in a row-major order on the $n \times n$ grid. Each row of the Representation-Matrix denotes a position on the $n \times n$ grid. The placement of 1 in each row tells us which tile is to be placed at the position corresponding to that row number. Two tiles shall be called *vertically-compatible* if they can be placed vertically over each other and similarly *horizontally-compatible* if they can be placed horizontally beside each other.

The program works using three matrices M1, M2 and M3 and the current value of M1, M2 and M3 together

determines the current value of the iterate. M1 and M2 represents the 1st two components corresponding to the two basic constraints. The basic constraints force each tile to be used exactly once and each position to be filled by exactly one tile. There are additional constraints due to the rules of the game. These are called horizontal and vertical constraints. These are due to the rules of the Tetravex that only tiles with matching common edge can be placed adjacent to each other. There are $2n(n-1)$ such constraints in total. M3 represents a matrix which consists of all these $2n(n-1)$ basic matrices. So M3 alone maintains all these horizontal and vertical constraints.

The program starts from the M-file Solve_Tetravex_Puzzle.m and it allows us to choose the number of tiles(9, 16, 25.....) and then let us input the puzzle we want the program to solve. After completion the program will display the number of iterations and the solution of the puzzle by producing an $n \times n$ matrix containing a unique number between 1 and n^2 at each cell on the grid. The number depicts the tile number(from input configuration) to be placed at the cell under consideration.

1.2 Different Files and Functions

1. Solve_Tetravex_Puzzle.m (M-file containing several functions)
2. Take_Tiles.m (Single function M-file)
3. StrToMatrix.m (Single function M-file)
4. Horizontal.m (Single function M-file)
5. Vertical.m (Single function M-file)
6. Diag_Ref.m (Single function M-file)
7. rows.m (Single function M-file)
8. columns.m (Single function M-file)
9. Unit_Reflection.m (Single function M-file)

10. Complete_Reflection.m (Single function M-file)

11. Allowed_Vertical.m (Single function M-file)

12. Allowed_Horizontal.m (Single function M-file)

13. Dist.m (Single function M-file)

14. Round_Average.m (Single function M-file)

1.3 Solve_Tetravex_Puzzle()

This file contains the main function Solve_Tetravex_Puzzle from where the program starts and it contain several other subfunctions that help the main function to calculate the desired arrangement of the tiles which solves the puzzle.

1.3.1 The main function: Solve_Tetravex_Puzzle

Arguments: No arguments, Output: Solution to the puzzle

The main function uses Take_Tiles function(see 1.4) to allow users to input tiles and select a grid size. Then it calculates "hor" and "ver" matrices(see 1.6) which stores the tiles that are compatible horizontally and vertically. The function uses several other function to find a solution of the puzzle using Douglas Rachford algorithm.

The current iterate is initialized as the given set of tiles in the order they are entered, hence M1, M2 are taken as Identity($n^2 \times n^2$) matrices and for M3 each of the constituent $2n(n-1)$ matrices are taken to be Identity matrices($n^2 \times n^2$). The iterate(M1, M2 and M3) keeps on changing as the iterations proceed. The variable *check* is defined which stores the value of the current iterate as a 0-1 matrix which is produced by rounding the diagonal set projection of the current iterate to a binary matrix. The function Satisfy_Tetravex(see 1.3.2) checks if the current value of variable check satisfy the constraints or not. Depending upon that the algorithm proceeds or produces a solution.

During each iteration the Douglas Rachford Algorithm is applied on the current iterate and new values of M1, M2, M3 are calculated. These new values are calculated using M11, M22, M33. First R_D is computed using the function Diag_Ref. Then we calculate R_C using Unit_Reflection(for M1), Complete_Reflection(for M2) and Reflect_Big(for M3). Hence we calculate $R_C R_D(M1, M2, M3))$. After this using Identity operator and averaging these values we calculate the next iterate. The variable check is updated using the Round_Average function(see 1.11.2) and the process continues unless the function Satisfy_Tetravex returns 1 or the iterations exceed 1000.

Once the while loop ends(a solution is found) the program displays the check variable and uses it to represent the solution in grid-form using the Represent_Good function(**??**)

1.3.2 Satisfy_Tetravex(T, M)

Arguments: Two matrices T($n^2 \times n^2$) and M($4 \times n$), Return Value: 1 or 0

This function checks if T is a solution to the given puzzle M. The function checks four conditions(constraints). The first two loops check if each row and each column is a unit vector.

The other two loops checks if the given matrix T satisfy the horizontal and the vertical constraints using the Aloowed_Horizontal(sec. 1.9.1) and Allowed_Vertical(sec. 1.9.2) functions.

If all the constraints are met then 1 is returned otherwise 0 is returned as soon as any constraint is found to be violated.

1.3.3 Horizontal_Constratints(a, b, M, hor)

Arguments: Integers(a and b) and Matrices(M and hor), Return Value: A matrix

This function computes the projection of a given $n^2 \times n^2$ matrix on a particular horizontal constraint that enforces positions a and b to have horizontally compatible tiles. The function uses Dist(1.10) to find the nearest compatible tiles relative to the current value of M. The matrix 'hor' is taken as an input because it is not accessible directly.

1.3.4 Vertical_Constraints(a, b, M, ver)

Arguments: Integers(a and b) and Matrices(M and ver), Return Value: A matrix

This function computes the projection of a given $n^2 \times n^2$ matrix on a particular vertical constraint that enforces positions a and b to have vertical compatible tiles. The function works in the same way as the function Horizontal_Constraint.

1.3.5 Reflect_Big(M, hor, ver)

Arguments: Three matrices, Return Value: A matrix

This function computes the reflection of M3 on the set with vertical and horizontal constraints in one step. The resultant matrix contains the reflection of each of the constituent $2n(n-1)$ matrices on the corresponding horizontal or vertical constraints. The function uses the functions Vertical_Constraints(sec. 1.3.3) and Horizontal_Constraints(sec. 1.3.4) functions to compute the reflection of constituent matrices.

1.3.6 Distance(M1, M2)

Arguments: Two row vectors , Return Value: A float value

This function computes the euclidian distance between two input row vectors M1 and M2.

1.3.7 Represent_Good(M)

Arguments: A matrix , Return Value: A matrix

This function is used by the main function to ultimately transform the solved puzzle from an $n^2 \times n^2$ matrix to an $n \times n$ matrix. The output matrix contain numbers from 1 to n^2 at each cell. The number on each cell indicates which tile is to be placed at that cell from the original input configuration.

1.4 Take_Tiles()

Arguments: No arguments , Return value: A matrix

This function is called by the Solve_Tetravex_Puzzle function and it allows the user to choose the size of the tetravex puzzle board and lets him to enter the puzzle pieces as a string in the format - 'a' 'b' 'c' 'd' (four numbers seperated by spaces) which denotes the digits on North(N), East(E), South(S) and West(W) edges of each tile respectively. In other words it lets the user to enter the digits on each tile in clockwise direction. The resultant matrix is of size $n^2 \times 4$ and is used throughout the program. The function uses StrToMatrix function to break each string(a tile) into a row vector of length 4.

1.5 StrToMatrix(str)

Arguments: A string , Return value: A row vector

This function is called by by Take_Tiles function only once during the start of the program to facilitate the user to input tiles of his/her choice. It takes a string denoting a sequence of integers each separated by a white space and convert the string into a row vector. The row vector now contains the input sequence.

1.6 Horizontal and Vertical

1.6.1 Horizontal(M)

Arguments: A matrix , Return value: A matrix

This function is called by Solve_Tetravex_Puzzle file and it's purpose is to create a list of those tiles which can be placed horizontally adjacent to each other. It produces a matrix which has 2 columns and hence each row contain two numbers. In every row these two numbers suggest that the 1st tile can be placed adjacent to the 2nd tile on the left side. The list of these tiles is calculated merely by going through all possible combinations of 2 tiles and then checking if they can be placed beside each other.

1.6.2 Vertical(M)

Arguments: A matrix , Return value: A matrix

This function is called by Solve_Tetravex_Puzzle file and it creates a list of those tiles which can be placed vertically adjacent to each other. It produces a matrix which has 2 columns similar to the function above. The only difference is that here the 1st tile can be placed vertically above the 2nd one.

1.7 rows and columns

1.7.1 rows(matrix)

Arguments: A matrix , Return value: An integer

This function calculates the number of rows in the given matrix using the inbuilt function size.

1.7.2 columns(matrix)

Argument: A matrix , Return value: An Integer

This function calculates the number of columns in the given matrix using the inbuilt function size.

1.8 Unit_Reflection and Complete_Reflection

1.8.1 Unit_Reflection(M)

Arguments: A matrix , Return value: A binary(0-1) matrix

This function calculates a reflection of the given matrix on a set which enforces every row to be a unit vector. Firstly it computes projection on the 1st constraint(Every position contains exactly one tile). In every row maximum entry is converted to 1 and all others to zero. In case there are more than 1 maximum entries the leftmost is converted to 1. Now reflection is directly calculated using projection.

1.8.2 Complete_Reflection(M)

Arguments: A matrix , Return value: A binary(0-1) matrix

This function calculates a reflection of the given matrix on a set that enforces every column to be a unit vector. This represents the projection on the 2nd constraint(Every tile is used exactly once). It rotates the given matrix anticlockwise then computes it's row projection and then again rotate it clockwise. This gives us the column projection using which reflection is directly calculated.

1.9 Allowed_Horizontal and Allowed_Vertical

1.9.1 Allowed_Horizontal(v1, v2)

Arguments: Two row vectors , Return value: 0 or 1

This function takes two tiles as an input. Each of the two tiles is represented by a row vector with 4 entries denoting digits on the N, E, S, W directions respectively. The function returns 0 or 1 depending upon whether the 1st tile can be placed adjacent to the 2nd tile on the left side. It is equivalent to checking if the 2nd entry of the 1st vector is same as the 4th entry of the 2nd.

1.9.2 Allowed_Vertical(v1, v2)

Arguments: Two row vectors , Return value: 0 or 1

This function works in the same manner as Allowed_Horizontal. It basically checks if the first of the two tiles can be placed above the second tile. It is basically equivalent to checking if the 3rd entry of the first vector is the same as the 1st entry of the 2nd.

1.10 Dist(T1, num)

Arguments: One row vector(T1) and one integer(num) , Return value: a float value

This function computes the square of the distance between two vectors. The first vector is 'T1'. The second vector is a unit vector with length same as that of T1 with 1 placed at the num'th position.

1.11 Diag_Ref and Round_Average

1.11.1 Diag_Ref(M1, M2, M3)

Arguments: Three matrices , Return value: A matrix

This function finds the reflection of the current iterate on the Diagonal set. The matrices M1($n^2 \times n^2$) and M2($n^2 \times n^2$) represents the components of the iterate along the first two constraints. M3($n^2 \times 2n^3(n-1)$) represents the component along the horizontal and vertical constraints. The function splits M3 into $2n(n-1)$ different matrices and then takes the average of M1, M2 and these $2n(n-1)$ matrices. This gives us the projection on diagonal set(D). Using this projection, reflection on this set(Diagonal set) is calculated.

1.11.2 Round_Average(M1, M2, M3)

Arguments: Three matrices , Return value: A matrix

This function is used to create a matrix on which the stopping condition is tested. It uses the same idea as Diag_Ref and first calculates the projection on set D(Diagonal set) by splitting M3 into individual matrices of size($n^2 \times n^2$). Then it rounds the resultant matrix into a 0-1 matrix and returns it.

Chapter 2

Using the Program

In this chapter we will show an example of how to enter the input tiles and how to run the code. We will also give you some links where you can get 3 x 3, 4 x 4 and 5 x 5 puzzles.

2.1 Starting the program

Firstly you are required to save all the M-files shown in the program in a directory. Then you can start your Matlab/Octave window and go into the directory in which all the files are saved. Now enter the following command on your command window. We are assuming Octave language here, but the procedure with Matlab is exactly the same.

```
octave-3.2.4.exe:4> Solve_Tetravex_Puzzle
```

This will result into the following:

```
enter the number of tiles in your tetravex
```

You can enter the number of tiles on your tetravex depending upon it's size (e.g 16 for a 4 x 4 Tetravex). After this the following window will appear

```
Enter the numbers on each tile in the form
in the form  a b c d  where a, b, c, d represents
```

```
the N,E,S,W direction numbers on each tile \n

enter the numbers a b c d
```

Now the program will allow you to enter the tiles one by one by asking each of the four numbers every tile. After entering all the n^2 tiles the program will produce output as an nxn matrix containing numbers from 1 to n^2. This will become clear when we will show you example of a 3 x 3 tetravex puzzle(sec 2.2).

2.2 Example

In this section we will show how we can solve a Tetravex puzzle using this code. We will take the following 3 x 3 puzzle as an example.

Figure 2.1: Sample 3 x 3 Tetravex Puzzle

Our next task is to solve this puzzle.

2.2.1 Solving Sample Puzzle

To solve this puzzle we will run the program by the following command

```
octave-3.2.4.exe:4> Solve_Tetravex_Puzzle
```

Now we will enter the following:

```
enter the number of tiles in your tetravex  9
```

After this the program will allow us to enter each tile and we will enter a string in the form a_b_c_d here '_' represent whitespaces. Make sure not to put any white space after 'd' or don't try to input in any other format.

```
Enter the numbers on each tile in the form
in the form  a b c d  where a, b, c, d represents
the N,E,S,W direction numbers on each tile \n
enter the numbers a b c d 3 4 8 1
enter the numbers a b c d 9 4 4 5
enter the numbers a b c d 3 1 2 4
enter the numbers a b c d 7 5 6 7
enter the numbers a b c d 8 3 9 9
enter the numbers a b c d 2 3 8 3
enter the numbers a b c d 2 9 7 5
enter the numbers a b c d 8 9 2 4
enter the numbers a b c d 4 1 2 6
```

The tiles will be entered from top to bottom and from left to right in each row of the 3 x 3 puzzle(Row major order). In a sense tiles will be indexed 1 to 9 in the order they are entered. Each tile is entered in clockwise direction(N, E, S and then W). For instance Fig. 2.2 shows tile to be entered on 5th time is the string '8 3 9 9'.

Figure 2.2: A tile from the Sample Tetravex

1	2	3
4	5	6
7	8	9

The above figure shows the Row major order of the 3 x 3 grid and it indicates the numbering of the corresponding tiles.(E.g. in this case tile 1 is '3 4 8 1' and tile 6 is '2 3 8 3'(see Fig 2.1)). So each tile can be represented by a number between 1 to 9.

After entering this puzzle as an input the following output will be generated:

```
the tiles that you have chosen for your tetravex are as follows:

    3   4   8   1

    9   4   4   5

    3   1   2   4

    7   5   6   7

    8   3   9   9

    2   3   8   3

    2   9   7   5

    8   9   2   4

    4   1   2   6

Matrix =


    3   4   8   1
```

```
     9     4     4     5

     3     1     2     4

     7     5     6     7

     8     3     9     9

     2     3     8     3

     2     9     7     5

     8     9     2     4

     4     1     2     6


hor =


     1     3

     1     8

     2     3

     2     8

     3     1

     4     2

     4     7

     5     6

     7     5

     8     5

     9     1


ver =


     1     5

     1     8

     2     9
```

```
    3   6

    3   7

    5   2

    6   5

    6   8

    7   4

    8   6

    8   7

    9   6

    9   7


ans = we are inside the Solving now

iterations =  1

iterations =  2

iterations =  3

iterations =  4

iterations =  5

iterations =  6

iterations =  7

iterations =  8

iterations =  9

iterations =  10

iterations =  11

iterations =  12

iterations =  13

iterations =  14

iterations =  15

iterations =  16
```

```
iterations =  17

iterations =  18

iterations =  19

iterations =  20

iterations =  21

iterations =  22

iterations =  23

iterations =  24

iterations =  25

iterations =  26

solution is:

9   1   3

7   5   6

4   2   8
```

2.2.2 Interpreting the solution

As you can see the program has given the solution as the 3 x 3 matrix:

```
9   1   3

7   5   6

4   2   8
```

Numbers in the above matrix represent tiles using numbering by the way the tiles are entered as we discussed earlier. This matrix represents the following arrangement of tiles(see fig. 2.1):

Congratulations! You have solved your first Tetravex puzzle using this program.

2.3 Useful Websites

These puzzle can be found from the following websites:

Figure 2.3: Solution to the Sample Puzzle

`http://bezumie.com/tetravex/index.php`

`http://gamegix.com/tetravex/game.php`

Appendix A

Tetravex Solver Code

In this appendix we have provided the code for the tetravex solving program described in Chapter 1. We have written the codes file by file.

A.1 Solve_Tetravex_Puzzle.m

```
% This program is an attempt to solve a Tetravex puzzle.
function y=Solve_Tetravex_Puzzle()
%This is the main function from where the program will start
    Matrix = Take_Tiles()
    hor = Horizontal(Matrix)
    ver = Vertical(Matrix)
    'we are inside the Solving now'
    row=rows(Matrix);
    M1 = eye(row);
    M2 = eye(row);
    order=sqrt(row);
```

```
        for i = [1:2*order*(order-1)]
            M3(1:row,row*i-row+1:row*i) = eye(row);
        end


        check=Round_Average(M1, M2, M3);
        iterations = 0;
        while(~Satisfy_Tetravex(check, Matrix) & iterations<1000)
            [M11, M22, M33]=Diag_Ref(M1, M2, M3);
            M11 = Unit_Reflection(M11);
            M22 = Complete_Reflection(M22);
            M33 = Reflect_Big(M33, hor, ver);
            M1 = (M1 + M11)/2;
            M2 = (M2 + M22)/2;
            M3 = (M3 + M33)/2;
            check=Round_Average(M1, M2, M3);
            iterations = iterations+1
        end
        disp('solution is:');
        disp(Represent_Good(check));
    end


function y=Vertical_Constraint(a,b,M, ver)
% This function computes the projection of the given matrix on the set which has Matri
    y=M;
    siz=rows(M);
    min_index=1;
```

```
    min=Dist(M(a,:),ver(1,1)) + Dist(M(b,:),ver(1,2));
        for i=[1:rows(ver)]
            if((Dist(M(a,:),ver(i,1)) + Dist(M(b,:),ver(i,2)))<min)
            min=Dist(M(a,:),ver(i,1)) + Dist(M(b,:),ver(i,2));
            min_index=i;
        end
    end
    y(a,:) = zeros(1,siz);
    y(a, ver(min_index,1)) =1;
    y(b,:) = zeros(1,siz);
    y(b, ver(min_index,2)) =1;
end



function y=Horizontal_Constraint(a, b, M, hor)
% This function computes the projection of the given matrix on the Set which has Matri
    y=M;
    siz=rows(M);
    min_index=1;
    min=Dist(M(a,:), hor(1,1)) + Dist(M(b,:), hor(1,2));
    for i=[1:rows(hor)]
        if((Dist(M(a,:),hor(i,1)) + Dist(M(b,:),hor(i,2)))<min)
            min=Dist(M(a,:),hor(i,1)) + Dist(M(b,:),hor(i,2));
            min_index=i;
        end
    end
```

```
    y(a,:) = zeros(1,siz);
    y(a, hor(min_index,1)) =1;
    y(b,:) = zeros(1,siz);
    y(b, hor(min_index,2)) =1;
end


function y=Reflect_Big(M, hor, ver)
% This function reflects a (n*n) X (n*n*n*(n-1)*2(in our case (9X108))) {Assuming given
% "M" matrix on the set which has adjacent tiles(both vertical and horizontal) with the
    %disp("inside Reflect_Big now")
    y=M;
    siz=rows(M);  %size = total number of pieces in the tetravex = n*n
    order=sqrt(siz);     % order is n
    ctr=0;
    M1=[];
    for i=[1:order^2]
        if(~(mod(i,order)==0))
            ctr=ctr+1;
            M1(ctr)=i;
        end
    end
    ctr=0;
    for i= M1
        y(1:siz,siz*ctr+1:siz*ctr+siz) = 2*Horizontal_Constraint(i, i+1, y(1:siz,siz*c
        ctr=ctr+1;
    end
```

```
    ctr=0;
    for i= [1:order^2]
        if(i<=order*(order-1))
            ctr=ctr+1;
            M1(ctr)=i;
        end
    end
    order;
    siz;
    ctr=order*(order-1);
    for i=M1
        y(1:siz, siz*ctr+1:siz*ctr+siz) = 2*Vertical_Constraint(i, i+order, y(1:siz, si
        ctr=ctr+1;
    end
end



function y = Satisfy_Tetravex(T, Matrix)
    %the function basically checks if T is a solution to the puzzle represented by Mat
    y=1;
% checking that each position has exactly one tile
    R_T=[]; % this variable will represent each row of T by a number which is the colun
    for i=[1:rows(T)]
    ctr=0;
        for j=[1:columns(T)]
            if(T(i,j)==1)
```

```
                ctr=ctr+1;
                R_T(i)=j;
            end
        end
        if(~(ctr==1))
            y=0;
            return;
        end
    end
% checking that each tile is used exactly once
    I=ones(1,rows(T));
    for i=1:rows(T)
        if(~(I(R_T(i))==1))
            y=0;
            return;
        end
        I(R_T(i))=0;
    end
% checking the constraint of adjacent tiles in horizontal direction
    order=sqrt(rows(T));
    M1=[];
    ctr=0;
    for j=[1:order^2]
        if(~(mod(j,order)==0))
            ctr=ctr+1;
            M1(ctr)=j;
```

```
        end
    end


    for i=M1
        if(~Allowed_Horizontal(Matrix(R_T(i), :) , Matrix(R_T(i+1), :)))
            y=0;
            return;
        end
    end


% checking for the constraint of adjacent tiles in vertical direction
    for i=[1:order*(order-1)]
        if(~Allowed_Vertical(Matrix(R_T(i), :) , Matrix(R_T(i+order), :)))
            y=0;
            return;
        end
    end
end


function y= Distance(M1,M2)
%This function calculates the Eucledian distance between 2 vectors.


    Rows=rows(M1);
    Columns=columns(M1);
    diff=0;
    for i=[1:Rows]
```

```
        for j=[1:Columns]
            diff = diff + (M1(i,j)-M2(i,j))^2;
        end
    end
    y=sqrt(diff);
end


function y= Represent_Good(M)
    siz=sqrt(rows(M));
    for i=[1:siz]
        for j=[1:siz]
            [max_val,max_index]= max(M(i*siz+j-siz,:));
            y(i,j)=max_index;
        end
    end
end
```

A.2 Take_Tiles.m

```
        function M=Take_Tiles()
            %this function allows user to input tiles of his choice and also choose the
            tiles = input('enter the number of tiles in your tetravex ', 's');
                tiles = str2num(tiles);
            M=[];
            disp('Enter the numbers on each tile in the form');
            disp('in the form  a b c d  where a, b, c, d represents');
```

```
    disp('the N,E,S,W direction numbers on each tile \n');


    for i=[1:tiles]
        str=input('enter the numbers a b c d ', 's' );
        M(i,:)=StrToMatrix(str);
    end
    disp('the tiles that you have chosen for your tetravex are as follows:');
    disp(M);
end
```

A.3 StrToMatrix.m

```
function y=StrToMatrix(str)
    ctr=1;
    temp='';
    for i=1:length(str)
        if(str(i)==' ')
            y(ctr)=str2num(temp);
            ctr=ctr+1;
            temp='';
        else
            temp=strcat(temp, str(i));
        end
    end
    y(ctr)=str2num(temp);
end
```

A.4 Vertical.m

```
function y= Vertical(M)
%This function creates a list of those tiles in which 1st tile can be placed above
    counter=1;
    row=rows(M);
    for i=[1:row]
        for j=[1:row]
            if(Allowed_Vertical(M(i,:),M(j,:)) & ~(i==j))
                y(counter,:)=[i,j];
                counter= counter+1;
            end
        end
    end
end
```

A.5 Horizontal.m

```
function y= Horizontal(M)
%This function creates a list of those tiles in which 1st tile can be placed a
    counter=1;
    row=rows(M);
    for i=[1:row]
        for j=[1:row]
            if(Allowed_Horizontal(M(i,:),M(j,:)) & ~(i==j))
                y(counter,:)=[i,j];
```

```
                counter=counter+1;
            end
        end
    end
end
```

A.6 Diag_Ref.m

```
function [M11, M22, M33] = Diag_Ref(M1, M2, M3)
% This function computes the reflecton of the sets M1, M2, M3 on the Diagonal
    rows_M1=size(M1);
    rows_M1=rows_M1(1);
    x=size(M3);
    x=x(2);    % variable x is the number of columns in M3
    siz = x / rows_M1;
    num=siz+2;
    Total=M1+M2;
    for i=[1:siz]
        Total = Total + M3(1:rows_M1,rows_M1*i-rows_M1+1:rows_M1*i);
    end
    y=Total/num;
    M11 = 2*y;
    M22 = 2*y;
    for i=[1:siz]
        M33(1:rows_M1,rows_M1*i-rows_M1+1:rows_M1*i) = 2*y;
        end
```

```
    M11=M11-M1;
    M22=M22-M2;
    M33=M33-M3;
end
```

A.7 rows.m

```
function y= rows(matrix)
%this function computes the number of rows of the input matrix.
    siz = size(matrix);
    y=siz(1);
end
```

A.8 columns.m

```
function y= columns(matrix)
%this function computes the number of columns of a matrix
    siz=size(matrix);
    y= siz(2);
end
```

A.9 Unit_Reflection.m

```
function y = Unit_Reflection(M)
%this function gives projection on the set which has every row a unit vector(i
%Tetravex contains exactly one tile)
    siz=size(M);
    for i = [1:siz(1)]
```

```
        [maxim, max_index] = max(M(i, :));
        y(i,:) = zeros(1, siz(2));
        y(i, max_index) = 1;
    end
    y=2*y-M;
end
```

A.10 Complete_Reflection.m

```
function y = Complete_Reflection(M)
%This function gives reflection on the constraint that every tile is used exac
%is a unit vector)
    y = rot90(M);
    y = rot90(Unit_Reflection(y),-1);
end
```

A.11 Allowed_Vertical.m

```
function y=Allowed_Vertical(v1, v2)
%This function checks whether the tile v1 can be placed above v2 or not.
    y=0;
    if(v1(3)==v2(1))
        y=1;
    end
end
```

A.12 Allowed_Horizontal.m

```
        function y=Allowed_Horizontal(M1,M2)
        %This function checks whether the 1st tile can be placed on the left of the 2n
            y=0;
            if(M1(2)==M2(4))
                y=1;
            end
        end
    \end{lstliting}
\section{Dist.m}
    \begin{lstlisting}[language=Matlab]
        function d=Dist(T1,num)
        %This function computes square of Euclidean distance between a n-tuple and a g
            T2=zeros(columns(T1));
            T2(num)=1;
            d=0;
            for i=[1:columns(T1)]
                d = d + (T1(i)-T2(i))^2;
            end
        end
```

A.13 Round_Average.m

```
        function y= Round_Average(M1, M2, M3)
        %This function is used for calculating the average of the 14 different sets an
            col=columns(M1);
            siz = columns(M3)/col;
```

```
    num=siz+2;
    Total=M1+M2;
    for i=[1:siz]
        Total = Total + M3(1:rows(M1) , col*i-col+1:col*i);
    end
    y=Total/num;
    for i=[1:rows(y)]
        for j=[1:columns(y)]
            if(y(i,j)>=0.5)
                y(i,j)=1;
            else
                y(i,j)=0;
            end
        end
    end
end
```