A Project Report

On

Point generation using quad-tree data structure

BY

Kartik Srivastava

2014B4A7755H

Under the supervision of

Dr. N. Anil

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF

MATH F376

DESIGN ORIENTED PROJECT

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI (RAJASTHAN)

HYDERABAD CAMPUS

(MARCH 2017)

ACKNOWLEDGMENT

I would like to thank my teacher and mentor Dr. N. Anil under whose supervision I am pursuing this Design Oriented Project. He has been a great source of support and inspiration and has helped me with his knowledge and insight. I would also like to thank my friends and seniors who have helped by reviewing and providing their valuable suggestions in various parts of the project.

Birla Institute of Technology and Science-Pilani,

Hyderabad Campus

CERTIFICATE

This is to certify that the project report entitled “**Point generation using quad-tree data structure**” submitted by **Mr**. **Kartik Srivastava** (ID No. **2014B4A7755H**) in partial fulfillment of the requirements of the course MATH F376, Design Oriented Project Course, embodies the work done by him under my supervision and guidance.

Date: 20th March 2017

(Dr. N. Anil)
BITS- Pilani, Hyderabad Campus

ABSTRACT

In this project I have tried to generate the point distribution for meshless solvers by implementing the method outlined by U. Mohan Varma[1]. The method uses the quadtree data structure to generate the distribution in two-dimensions which can be further extended to three-dimensional bodies by using octree data structures. Every leaf node of the balanced quadtree supplies a point to the final distribution. The quadtree thus formed can also be used to get neighbour properties which can help in further boosting of points at specific parts of the strucure and obtain a fine distribution of points. This distribution can be fed to a meshless solver to solve flow over the given geometry.

CONTENTS

1. Introduction

After several strides of progress in mesh generation techniques for geometries, few complex geometries still need a lot of manual intervention and thus are highly time consuming. To overcome these problems, meshless solvers are being developed these days which work on a point distribution instead of the cartesian mesh structure. One major problem in this is the accurate generation of point distribution which is fed to the solver. It should be dense at the boundary and sparse away from the boundaries. Also it should be empty inside the boundary. This paper aims at solving this problem of point generation for 2-D geometries using quadtree data structure.

2. Quadtree Data Structure

It is a hierarchical data structure similar to the binary tree but it has exactly 4 children at each node instead of two. These children nodes are - Nort-West node, North-East node, South-West node and South-East node. A quadtree is the most suitable data structure to store information about two dimensional images as every image can be split into four quadrants which can be further subdivided recursively and at each step the 4 quadrants correspond to each of the children nodes.

The qudtree data structure to be used in the program is as follows:

```
typedef struct node
{
        struct node *nw;//pointer to North-West neighbour
        struct node *ne;//pointer to North-East neighbour
        struct node *sw;//pointer to South-West neighbour
        struct node *se;//pointer to South-East neighbour

        struct node *par;//pointer to the parent node

        double lx,ly;//Coordinates of lower left corner of the area
        double hx,hy;//Coordinates of upper right corner of the area
        //double x,y;//Coordinates of boundary/solid point

        int level;//Level of node in tree
        int point;

}Node;//Structure of node
```

3. Generating the point distribution

Generation of the point distribution is done in three steps:

a. Quadtree Generation

The bottom left(P_1) and upper right coordinates(P_2) along with the coordinates of the outer boundary are given as inputs. Now the total area formed by P_1P_2 is divided into four quadrants of equal sizes. Then we count the number of points in each of these quadrants. If any of the quadrants contain more than one point, the quadrants is further subdivided. This is a recursive procedure and it goes on until each of the quadrants contains maximum one point.

```
Node * generateQuadTree(double l1,double l2,double h1,double h2,int n,double arr[][2],Node *parent)
{
        Node * temp=createNode(l1,l2,h1,h2,parent);

        int c=0,i;

        for(i=0;i<n;i++)
        if(arr[i][0]>=l1 && arr[i][0]<=h1 && arr[i][1]>=l2 && arr[i][1]<=h2)
        c++;

        if(c>1)
        {
                temp->nw=generateQuadTree(l1,(l2+h2)*0.5,(l1+h1)*0.5,h2,n,arr,temp);
                temp->ne=generateQuadTree((l1+h1)*0.5,(l2+h2)*0.5,h1,h2,n,arr,temp);
                temp->sw=generateQuadTree(l1,l2,(l1+h1)*0.5,(l2+h2)*0.5,n,arr,temp);
                temp->se=generateQuadTree((l1+h1)*0.5,l2,h1,(l2+h2)*0.5,n,arr,temp);
        }
        else
        return temp;

}// Function to generate Quadtree
```

b. Quadtree Balancing

The quadtree thus generated is now balanced. This is achieved by further subdividing the leaf nodes which are courser than any of its neighbouring nodes by more than one level.

Further all the quadtrants which have no points inside them contribute their centroid to the final distribution of points. This gives a symmetric cartesian point distribution.

```
int balanceQuadTree(Node * temp)
{
        if(isLeafNode(temp)==1)
        {
                if((height-(temp->level)) > 1)
                {
                        int Ln,Le,Lw,Ls,l1,l2,h1,h2;
                        int h=temp->level;
                        Ln=getNorthNeighbourLevel(temp);
                        Le=getEastNeighbourLevel(temp);
                        Lw=getWestNeighbourLevel(temp);
                        Ls=getSouthNeighbourLevel(temp);

                        if((Ln-h>1) || (Le-h>1) || (Lw-h>1) || (Ls-h>1))
                        {
                                l1=temp->lx;
                                l2=temp->ly;
                                h1=temp->hx;
                                h2=temp->hy;

                                temp->nw=createNode(l1,(l2+h2)*0.5,(l1+h1)*0.5,h2,temp);
                                temp->ne=createNode((l1+h1)*0.5,(l2+h2)*0.5,h1,h2,temp);
                                temp->sw=createNode(l1,l2,(l1+h1)*0.5,(l2+h2)*0.5,temp);
                                temp->se=createNode((l1+h1)*0.5,l2,h1,(l2+h2)*0.5,temp);
                        }

                }
        }

        if(isLeafNode(temp)==0)
        {
                balanceQuadTree(temp->nw);
                balanceQuadTree(temp->ne);
                balanceQuadTree(temp->sw);
                balanceQuadTree(temp->se);
        }

        return 0;
}
```

c. Blanking the interior points

The points falling inside the boundary of the geometry are now removed from the final point distribution by using the ray tracing algorithm. In the ray tracing algorithm we draw a line parallel to X axis from a point to infinity and if the point makes odd number of intersections, it falls inside the boundary else it falls outside the boundary. Thus all the points falling inside the given boundary points are removed and the resultant distribution is our final distribution.

The following code checks if two lines intersect:

```
int onSegment(double px,double py,double qx,double qy,double rx,double ry)
{
        if(qx <= max(px,rx) && qx>=min(px,rx) && qy <= max(py,ry) && qy>=min(py,ry))
        return 1;
        else
        return 0;
}

int orientation(double px,double py,double qx,double qy,double rx,double ry)
{
        double val=(qy-py)*(rx-qx)-(qx-px)*(ry-qy);

        if(val==0)
        return 0;
        else if(val>0)
        return 1;
        else return 2;
}

int doIntersect(double p1x,double p1y,double q1x,double q1y,double p2x,double p2y,double q2x,double q2y)
{
        int o1=orientation(p1x,p1y,q1x,q1y,p2x,p2y);
        int o2=orientation(p1x,p1y,q1x,q1y,q2x,q2y);
        int o3=orientation(p2x,p2y,q2x,q2y,p1x,p1y);
        int o4=orientation(p2x,p2y,q2x,q2y,q1x,q1y);

        if(o1!=o2 && o3!=o4)
        return 1;

        if(o1==0 && onSegment(p1x,p1y,p2x,p2y,q1x,q1y)==1)
        return 1;
        if(o2==0 && onSegment(p1x,p1y,q2x,q2y,q1x,q1y)==1)
        return 1;
        if(o3==0 && onSegment(p2x,p2y,p1x,p1y,q2x,q2y)==1)
        return 1;
        if(o4==0 && onSegment(p2x,p2y,q1x,q1y,q2x,q2y)==1)
        return 1;

        return 0;
}
```

The next part of the code checks if a point falls inside a given set of sides of a polygon. Here we have to make sure that the points supplied are given in clockwise or counter-clockwise sequence.

The following code checks if the point passed as parameter is inside the given boundary:

```
int isInside(double arr[][2],int n,double px,double py)
{
        if(n<3)
        return 0;

        double exX=INF,exY=py;

        int count=0,i=0;

        do
        {
                int next=(i+1)%n;
                //printf("%lf %lf - %lf %lf\n",arr[i][0],arr[i][1],arr[next][0],arr[next][1]);

                if(doIntersect(arr[i][0],arr[i][1],arr[next][0],arr[next][1],px,py,exX,exY)==1)
                {
                        //printf("%d %d\n",i,next);
                        if(orientation(arr[i][0],arr[i][1],px,py,arr[next][0],arr[next][1])==0)
                        return onSegment(arr[i][0],arr[i][1],px,py,arr[next][0],arr[next][1]);

                        count++;
                }

                i=next;

        }while (i!=0);

        printf("%lf %lf count=%d\n",px,py,count);

        if(count%2==1)
        return 1;
        else
        return 0;
}
```

The next part finally finds all the leaf nodes and checks if the points supplied by them are inside or outside the polygon:

```
void removeInteriorPoints(Node *temp,double arr[][2],int n)
{
        if(isLeafNode(temp)==1)
        {
                if(temp->point==-1)
                {
                        //printf("success\n");
                        if(isInside(arr,n,(temp->lx+temp->hx)*0.5,(temp->ly+temp->hy)*0.5)==1)
                        {
                                temp->point=-2;
                                printf("success\n");
                        }
                }
        }
        else
        {
                removeInteriorPoints(temp->nw,arr,n);
                removeInteriorPoints(temp->ne,arr,n);
                removeInteriorPoints(temp->sw,arr,n);
                removeInteriorPoints(temp->se,arr,n);
        }
        return;
}
```

4. Results

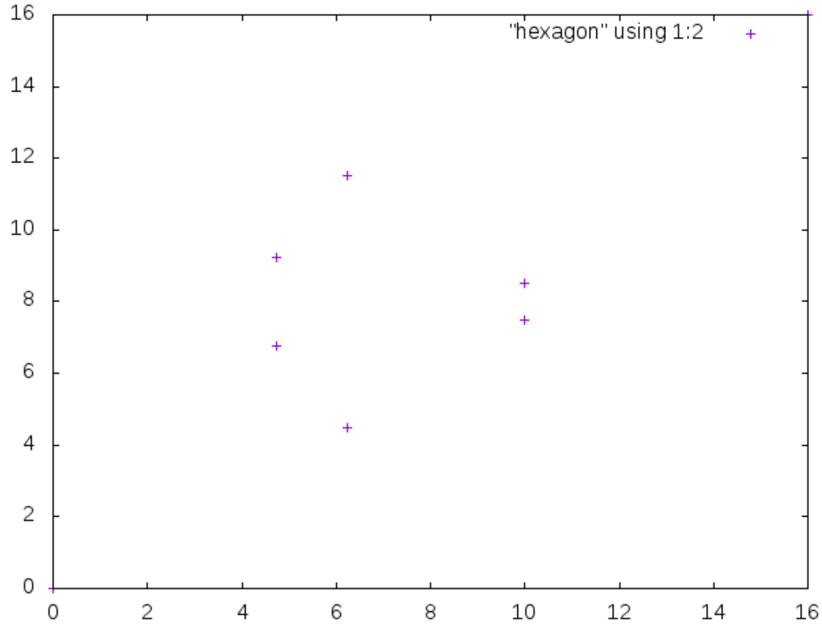
The point generation program created is successfully generating the quadtree structure by dividing the qudrants as expected as shown in the following images:

As an example we havetaken an irregular hexagon for our tests. It has the following points:

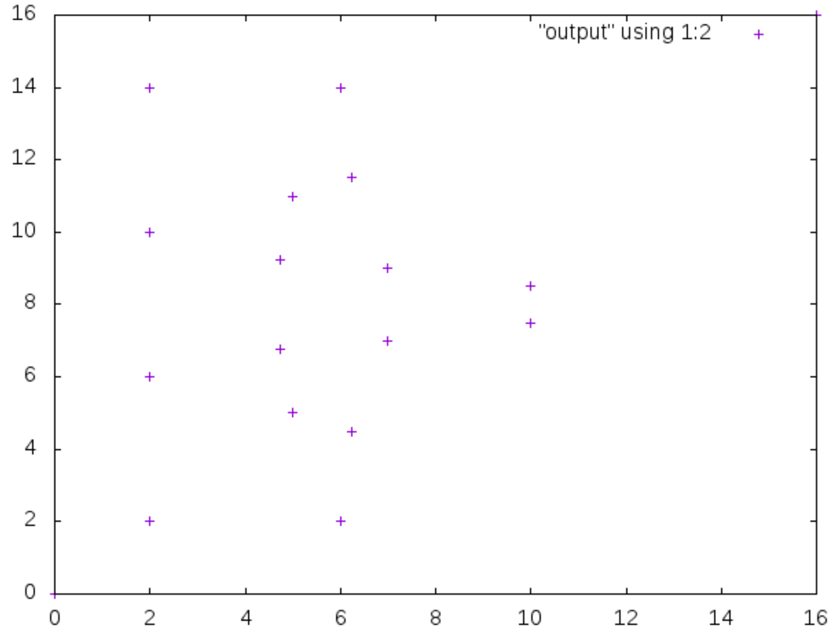
These are the extreme points in the quadrant: (0.0 ,0.0) (16.0 ,16.00)

(6.25 ,11.5)	(4.75 ,9.25)	(4.75 ,6.75)
(6.25, 4.5)	(10.0 ,7.5)	(10.0 ,8.5)

This is the image of the given geometry



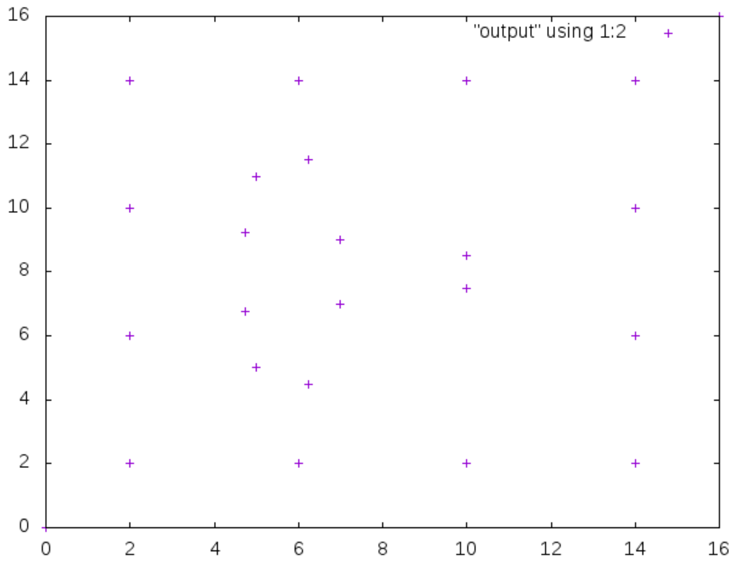
Alongside is the image after quadtree generation. The extra points are due to the newly generated quadtrants.



Following is the quadrants formed after balancing:

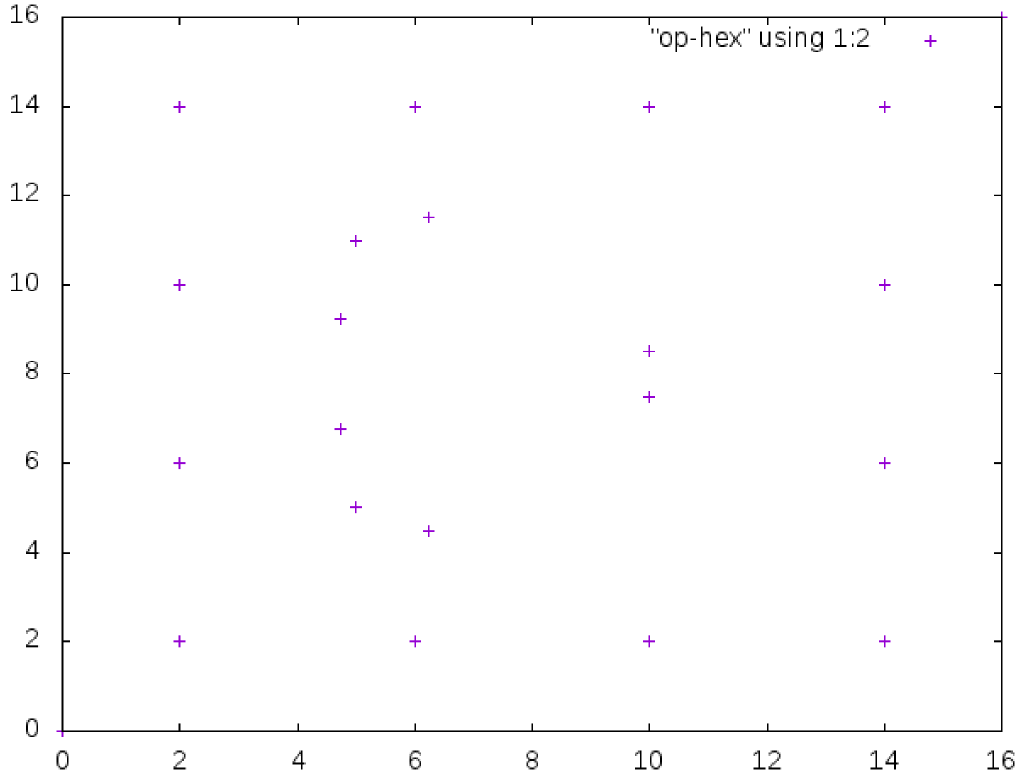
(0.000000,0.000000) - (16.000000,16.000000)
(0.000000,8.000000) - (8.000000,16.000000)
(0.000000,12.000000) - (4.000000,16.000000)
(4.000000,12.000000) - (8.000000,16.000000)
(0.000000,8.000000) - (4.000000,12.000000)
(4.000000,8.000000) - (8.000000,12.000000)
(4.000000,10.000000) - (6.000000,12.000000)
(6.000000,10.000000) - (8.000000,12.000000)
(4.000000,8.000000) - (6.000000,10.000000)
(6.000000,8.000000) - (8.000000,10.000000)
(8.000000,8.000000) - (16.000000,16.000000)
(8.000000,12.000000) - (12.000000,16.000000)
(12.000000,12.000000) - (16.000000,16.000000)
(8.000000,8.000000) - (12.000000,12.000000)
(12.000000,8.000000) - (16.000000,12.000000)
(0.000000,0.000000) - (8.000000,8.000000)
(0.000000,4.000000) - (4.000000,8.000000)
(4.000000,4.000000) - (8.000000,8.000000)
(4.000000,6.000000) - (6.000000,8.000000)
(6.000000,6.000000) - (8.000000,8.000000)
(4.000000,4.000000) - (6.000000,6.000000)
(6.000000,4.000000) - (8.000000,6.000000)
(0.000000,0.000000) - (4.000000,4.000000)
(4.000000,0.000000) - (8.000000,4.000000)
(8.000000,0.000000) - (16.000000,8.000000)
(8.000000,4.000000) - (12.000000,8.000000)
(12.000000,4.000000) - (16.000000,8.000000)
(8.000000,0.000000) - (12.000000,4.000000)
(12.000000,0.000000) – (16.000000,4.000000)

Following is the final distribution after balancing the quadtree:



Here we can see that the point distribution generated is as required.

Following is the image generated after blanking the interior points:



The point which were given as output are:

X	Y
2	14
6	14
2	10
5	11
6.25	11.5
4.75	9.25
10	14
14	14
10	8.5
14	10
2	6
4.75	6.75
5	5
6.25	4.5
2	2
6	2
10	7.5
14	6
10	2
14	2

References

1.Point Distribution generation using heirarchical Data Structures
(U. Mohan Varma, S.V. Raghurama Rao and S.M. Deshpande)

2. Neighbour Finding Techniques for Images Represented by Quadtrees
(Hanan Samet)

3. Efficient Neighbour Finding Algorithms in Quadtree and Octree
(Parthajit Bhattacharya)