Approximating Center Point of Planar Point Set with Iterated Radon Points

方启明 2004310428　李若皓 2004211044　程永席 2004310454

fangqiming@tsinghua.org.cn　mouseleeky@sina.com　cyx@mails.tsinghua.edu.cn

In our project, we implement a practical and provably good Monte Carlo algorithm that finds approximating center point on a plane.

1 Introduction

A **center point** of a set P of n points in R^d is a point c of R^d such that every hyperplane passing through c partitions P into two subsets each of size at most $nd/(d+1)$ (notice that there is no need that $c \in P$). This balanced separation property makes the center point useful for efficient divide-and-conquer algorithms in geometric computing and large-scale scientific computing.

The existence of a center point of any point set follows from the classical Helly theorem. However, finding an *exact* center seems a rather difficult task. It is possible to compute center point by solving a set of $\Theta(n^d)$ linear inequalities, using linear programming. The only improved results are that a center point in two dimensions can be computed in $c \in P$ time, and in three dimensions in $O(n^2 \log^7 n)$ time; the two-dimensional result had been improved to linear time later.

For most applications, it suffices to have an *approximate* center point, i.e. a point that every hyperplane through it partition P into subsets of size at least $n(\frac{1}{d+1} - \varepsilon)$. What is more, for constant β, a point $c \in R^d$ is called a β-center if every closed halfspace containing c contains at least βn points of P. Thus a center point is a $\frac{1}{d+1}$-center. For most situations, it suffices to have a β-center with reasonable value β. In [1] an efficient randomized algorithm finds a $\frac{1}{(d+1)(d+2)}$-center is presented, it runs in $O(d^2 \cdot [1+(d+1)\log_2 n + \log_2(1/\beta_d)]^{\log_2(d+2)})$ time, with probability of error at most 1/n. Where n=|P| is the number of points of P.

Our work is to implement this algorithm for the case d=2, i.e. finds a β-center with $\beta \geq 1/12$ of a set of points on a plane in time $O(\log^2 n)$, with small probability of error.

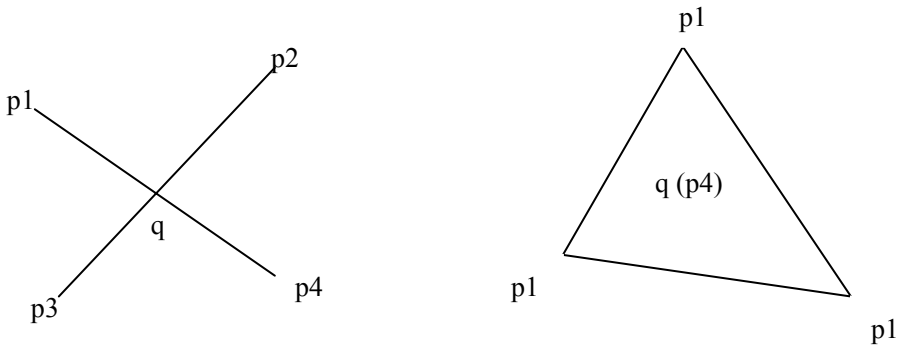
2 Radon Points

There is a fact important for the algorithm, Radon's Theorem (see [1] for proof).

Theorem 2.1 (Radon Theorem) If $P \subset R^d$ with $|P| \geq d+2$, then there is a partition (P1,P2) of P such that the convex hull of P1 has a point common with the convex hull of P2.

We will call the point common to the convex hulls of P1 and P2 a **Radon Point** of P. These kind of points are basis of the algorithm.

Definition 2.2 (Radon Points) Let P be a set of points in R^d, A point q is a Radon point of P if P can be partitioned into 2 disjoint subsets P1 and P2 such that q is a common point of the convex hull of P1 and the convex hull of P2.



The Radon point q of four points {p1, p2, p3, p4} in R^2. When no point is in the convex hull of the other three (the left figure), then the Radon point is the unique cross of two linear segments. Otherwise (the right figure), the point that is in the convex hull of the other three is a Radon point.

Figure 1

For degenerate case, i.e. if there are three points out of the four in a plane that are con-linear, then the median point of the three is a Radon point.

Why are Radon points useful in computing centers? A Radon point of a set of d+2

points is a 2/(d+2)-center of that set: any closed halfspace containing a Radon point r must contain a point of P1 and a point of P2. Hence the splitting ratio of a hyperplane containing r is at most d/(d+2)

The Radon point of a set P of more than d+1 points can be computed in $O(d^3)$ time [1]. *Especially for the case d=2, a Radon point can be computed in constant time.*

3 The Algorithm

Now we present the main algorithm that computes an approximating center point (an $\frac{1}{(d+1)(d+2)}$-center). W.l.g., we present the algorithm for the general case when the dimension is d.

The algorithm iteratively reduces the point set by replacing groups of (d+2) points by their Radon points. Such a reduction is guided by a complete (d+2)-way tree. It can be shown that the final point of this reduction process is an approximate center point with high probability.

Algorithm (Iterated Radon Points):
Input: a set of points $P \subset R^d$

1. Construct a complete balanced (d+2)-way tree T of L leaves.
2. For each leaf of T, choose a point from P uniformly at random, independent of other leaves.
3. Evaluate tree T in a bottom-up fashion to assign a point in R^d to each internal note of T such that the point of each internal node is a Radon point of the points with its (d+2) children.
4. Output the point associated with the root of T.

A complete (d+2)-way tree of L leaves has at most $L \cdot [1/(d+2)+1/(d+2)^2+\cdots] \le L/(d+1)$ internal nodes. The above algorithm takes $O(d^3 \cdot L/(d+1)) = O(d^2 L)$ time. It can be proved [1] $L = [(d+1)\log_2 n]^{\log_2(d+2)}$ is large enough to guarantee the probability that the output point is not an $\frac{1}{(d+1)(d+2)}$-center is at most 1/n.Hence the time complexity of the algorithm can be up

bounded to be $O(d^2 \cdot [(d+1)\log_2 n]^{\lfloor \log_2 (d+2) \rfloor})$. For the case d=2, it is $O(\log^2 n)$. It is mentioned in [1] the experimental results suggest that, independent of the size of original point set, L=800 is sufficient for the case d=3 and L=1000 for d=4. Thus for the case d=2, L=800 should be sufficient.

4 Computing Radon Point in A Plane

For the case d=2, the algorithm needs to compute Radon point of a set of four points $P = \{p_1, p_2, p_3, p_4\}$ in a plane. For degenerate case, i.e. there are three con-linear points, from definition the median of the three points is a Radon point of P. Otherwise, as shown in Fig.1 there are two nondegenerate cases. Thus there are three cases in all:

1. Degenerate case. i.e. there are three points of P con-linear.
2. Nondegenerate case. The convex hull of P is a triangle.
3. Nondegenerate case. The convex hull of P is a quadrilateral.

To distinguish these cases above, the following fact will be useful:

Lemma 5.1 Let $P = \{p_1, p_2, p_3\} \subset R^2$, denotes $D(p_1, p_2, p_3)$ to be the determinant

$$D(p_1, p_2, p_3) = \begin{vmatrix} p_{1x} & p_{1y} & 1 \\ p_{2x} & p_{2y} & 1 \\ p_{3x} & p_{3y} & 1 \end{vmatrix}$$

Where p_{ix}, p_{iy} denote the x,y coordinates of p_i, for $i = 1, 2, 3$. Then

1. $D(p_1, p_2, p_3) = 0$ if and only if p_1, p_2, p_3 are con-linear.
2. $D(p_1, p_2, p_3) > 0$ if and only if p_1, p_2, p_3 form a left turn (i.e. counterclockwise)
3. $D(p_1, p_2, p_3) < 0$ if and only if p_1, p_2, p_3 form a right turn (i.e. clockwise)

From lemma 5.1 above, it easy to check whether there are three points con-linear. Hence the task remained is to distinguish the two nondegenerate cases.

Let $P = \{p_1, p_2, p_3, p_4\} \subset R^2$ be a set of four affine independent points. It is easy to see: if the convex hull of P is a triangle, there are $4 \times 2 = 8$ different combinatorial

configurations; if the convex hull of P is a quadrilateral, there are $4!/4=6$ different combinatorial configurations. Totally 14(Fig. 2,3,4).

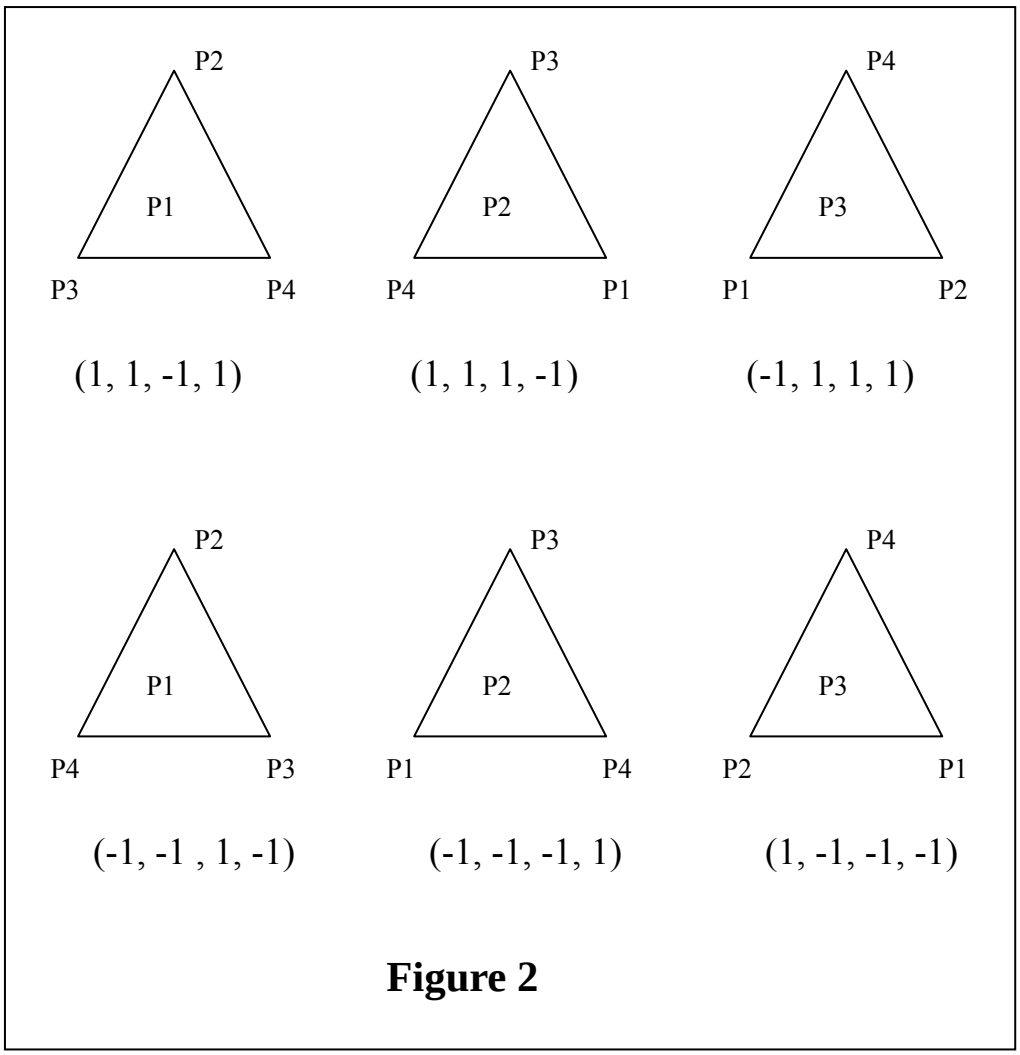


Figure 2

These different configurations can be characterized by sign of the four value $D(p_2, p_3, p_4), D(p_3, p_4, p_1), D(p_4, p_1, p_2)$
$D(p_1, p_2, p_3)$. Thus we can define the **characterized 4-tuple** of P, $c(P)$ to be:

Definition 5.2 Suppose $P=\{p_1, p_2, p_3, p_4\}\subset R^2$ to be a set of four affine independent points, define the *characterized 4-tuple* of P, $c(P)$ to be:
$c(P)=\left(\text{sgn}(D(p_2, p_3, p_4)),\ \text{sgn}(D(p_3, p_4, p_1)),\ \text{sgn}(D(p_4, p_1, p_2)),\ \text{sgn}(D(p_1, p_2, p_3))\right)$
where $\text{sgn}(x)$ denotes the sign of x, i.e.

$$\text{sgn}(x)=\begin{cases}1 & x>0\\ -1 & x<0\\ 0 & x=0\end{cases}$$

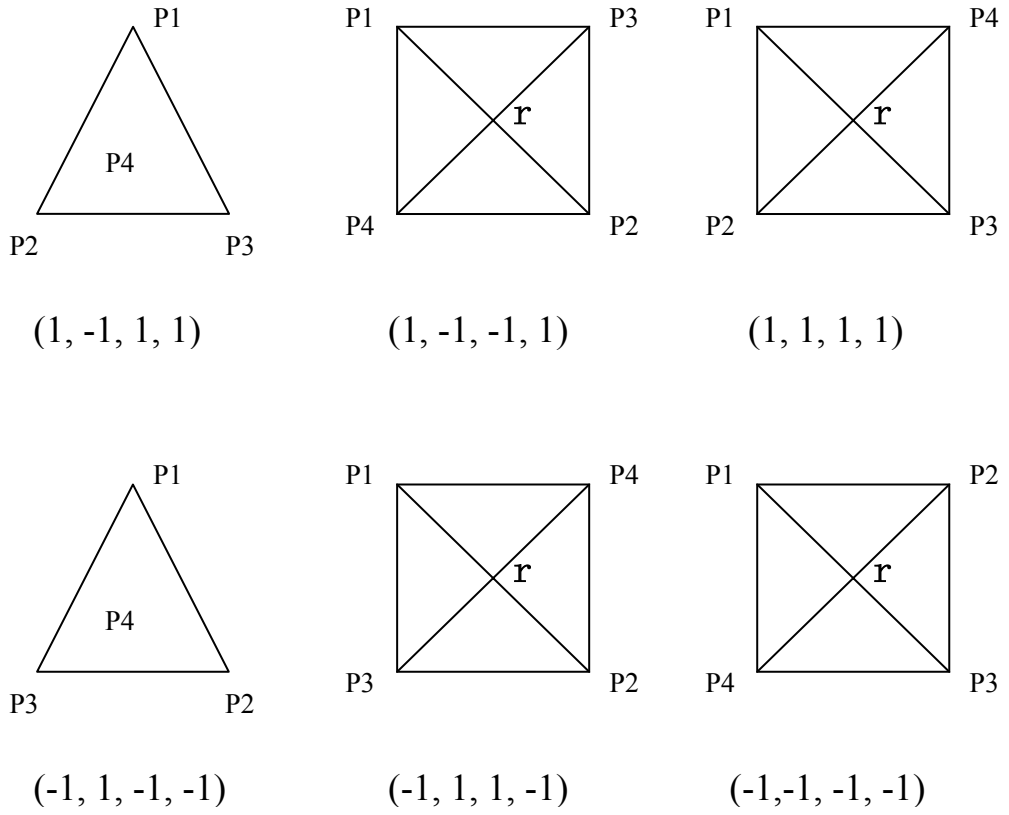


Figure 3

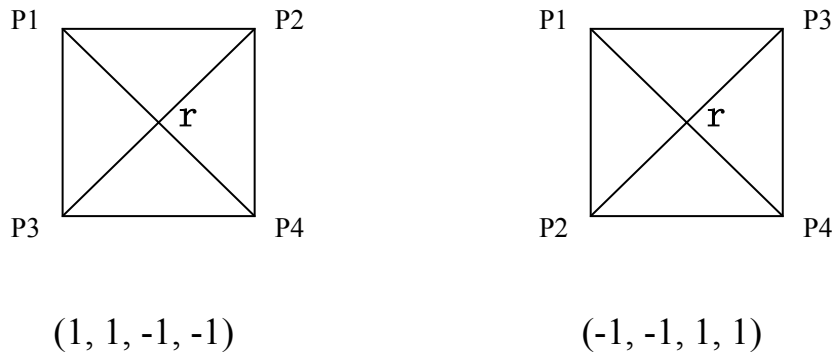


Figure 4

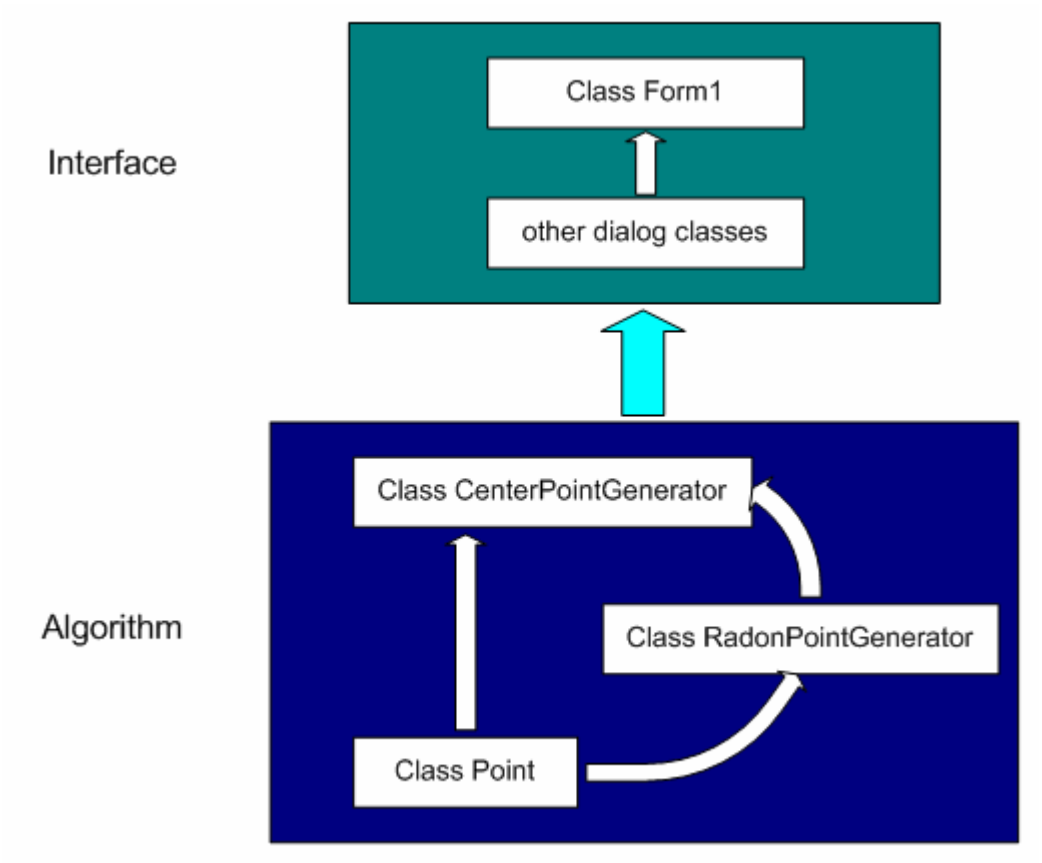
For the 14 different configurations, the 14 different characterized 4-tuples in accordance are shown in the figures. Also it is easy to check that the remained two

4-tuples (1,-1,1,-1) and (-1,1,-1,1) of the $2^4 = 16$ can not correspond to any actual

configuration. Hence to compute Radon point of four planar affine independent points, we can use characterized tuples to recognize their configurations, then choose according point (for the first 8 configurations) or compute the intersection of the two according line segment (for the last 6 configurations).

5 System Design and Data Structure

We experiment on Windows XP operation system, in Microsoft Visual Studio .Net 2003 environment, using VC++ language as development tool. Our experiment system can be divided into two parts: **algorithm** and **display**.



5.1 Three classes in the algorithm

There are three important classes in the algorithm part:

Class *Point***,** a class describe of a single planar point

Class *RadonPointGenerator*, a class to generate radon point of four planar points.

Class *CenterPointGenerator*, a class to generate center point of a set of planar points.

```
class Point
{
public:
	inline Point(int = 0, int = 0);
public:
	inline int GetX() const;
	inline int GetY() const;
	inline void SetX(int);
	inline void SetY(int);
private:
	int m_nX;
	int m_nY;
};
```

```
class RadonPointGenerator
{
public:
	inline RadonPointGenerator(Point* = NULL);
public:
	void Generate(int, int, int, int, Point&);
private:
	int GetTurningDirection(int, int, int);
	int GetMedianPoint(int, int, int);
	void GetIntersectionPoint(int, int, int, int, Point&);
private:
	Point* m_arrPoints;
};
```

```
class CenterPointGenerator
{
public:
	inline CenterPointGenerator();
	inline ~CenterPointGenerator();
```

```
public:
        void GeneratePoints(int, int, int);
        void SelectPoints(int);
        void Generate();
        double Validate(int, int&, int&);
        void Destroy();
public:
        Point* m_arrPoints;
        int m_nPoints;
        Point* m_arrSelectedPoints;
        int m_nSelectedPoints;
        int m_nCurrentIndex;
        int m_nCurrentBound;
        int m_nRemainedPoints;
        Point m_ptFirst;
        RadonPointGenerator* m_pRadonPointGenerator;
};
```

5.2 important interfaces and operations

Next we describe in detail the operations of the two classes RadonPointGenerator and CenterPointGenerator.

A. Operations of RadonPointGenerator

- void RadonPointGenerator::Generate(int nIndex1, int nIndex2, int nIndex3, int nIndex4, Point& radonPoint)

  Function： generate Radon Point of the 4 planar points denoted by their indices

  Parameters：

  nIndex1，nIndex2，nIndex3，nIndex4： the subscriptions of the 4 points in the sampling set.

  radonPoint： Radon Point of the 4 planar points denoted by the indices，return by reference

  Return type： void

- int RadonPointGenerator::GetTurningDirection(int nIndex1, int nIndex2, int nIndex3)

  Function： identify the turn direction of three planar points, i.e. left turn, right turn or con-linear. Subroutine of RadonPointGenerator::Generate.

  Parameters：

  nIndex1，nIndex2，nIndex3：the subscriptions of the three points in the sampling set.

  Return： integer, 1 denotes left return, -1 denotes right turn, 0 denotes con-linear.

- int RadonPointGenerator::GetMedianPoint(int nIndex1, int nIndex2, int nIndex3)

  Function：get the median point of three con-linear planar points. Subroutine of RadonPointGenerator::Generate.

  Parameters：

  nIndex1，nIndex2，nIndex3：the subscriptions of the three points in the sampling set.

  Return：integer, subscription of the median point.

- void RadonPointGenerator::GetIntersectionPoint(int nIndex1, int nIndex2, int nIndex3, int nIndex4, Point& intersectionPoint)

  Function：get intersection point of the two lines, which are determined by nIndex1 and nIndex2, nIndex3 and nIndex4 respectively. Subroutine of RadonPointGenerator:: Generate.

  Parameters：

  nIndex1，nIndex2，nIndex3，nIndex4：the subscriptions of the 4 points in the sampling set.

  intersectionPoint：the intersection point, return by reference

  Return：void

B. Operations of CenterPointGenerator

- void CenterPointGenerator::GeneratePoints(int nPoints, int nWidth, int nHeight)

  Function： generate a set of planar points randomly and uniformly

  Parameters：

  nPoints：number of points generated.

  nWidth：width of display window.

  nHeight：Height of display window.

  Return：void

- void CenterPointGenerator::SelectPoints(int nSelectedPoints)

  Function：Sampling from input points set uniformly.

  Parameters：

  nSelectedPoints：size of sampling set (power of four)

  Return：void

- void CenterPointGenerator::Generate()

  Function：generate Center Point

  Parameters：None

  Return：void

  Implementation：By the Iterated Radon Point algorithm presented in section 3.

- void CenterPointGenerator::Validate(int nAngle, int& nPositivePtNum, int& nNegativePtNum)

  Function：check quality of the Center Point found by the algorithm

  Parameters：

  nAngle：the slope angle of the rotation line that pass through the Center Point.

nPositivePtNum：at one moment, the number of original input points that lie in the positive side to (on) the rotation line.

nNegativePtNum：at one moment, the number of original input points that lie in the negetive side to (on) the rotation line.

Return： void

Above all, we have described the important interfaces and operations of the algorithm part.

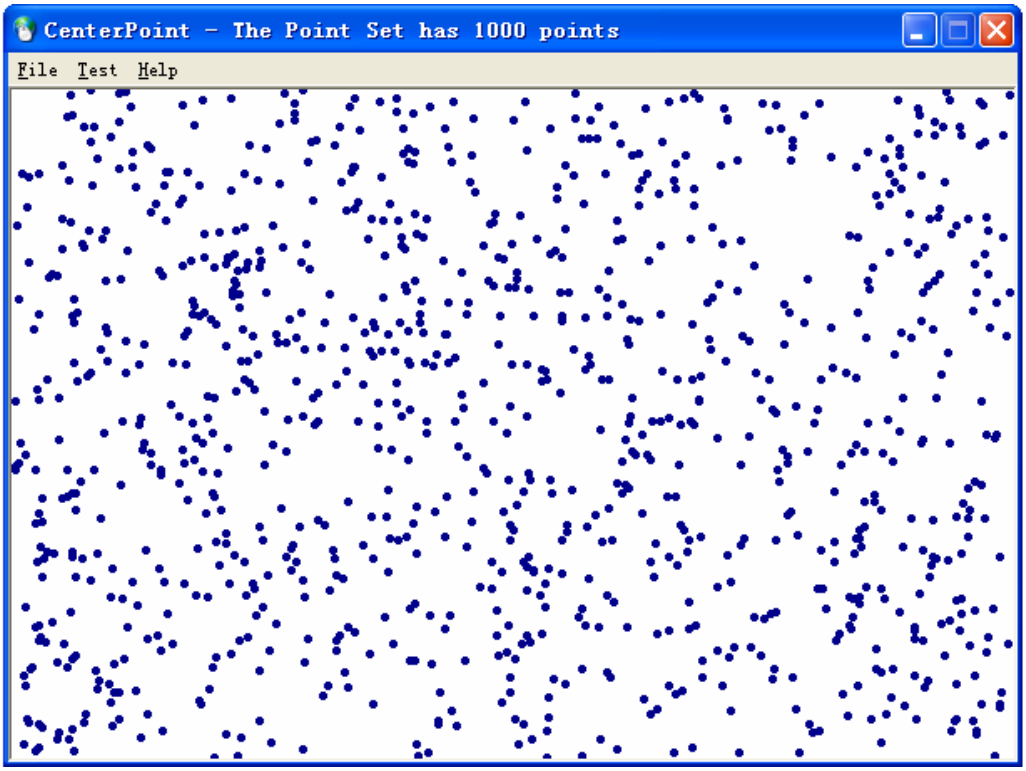
5.3 Data Structure

We haven't use any complicated data structure, only the most ordinary data structure—array are used in our implementation. Specifically, the two places we use array are for the member variables *Point* m_arrPoints* and *Point* m_arrSelectedPoints* of class *CenterPointGenerator*. They are used to store the originally input point set and the sampling set.

In section 3, the iterated algorithm is presented using a complete balanced (d+2)-way tree T. Actually the tree structure is only for clear presentation of the iterated algorithm and give an intuitive idea, it does no help for the algorithm's performance. So it is simplest (also reasonable) to use array when we implement the algorithm, moreover this avoid paying extra time for constructing the tree and paying extra space to store the tree's internal nodes.

5.4 Interface

We adopt menu and dialog pattern for the interface design, follows the principles that clear and easy to operate. Also shortcut keys are available for quick operation. Our main interface is as below :



There are three main menus: File、Test、Help. Their submenus and functions are described respectively below :

1） Submenu of 'File'

Submenu	Function	Shortcut
Export	Export the original input point set to a file.	Ctrl+E
Exit	Exit from the program	Alt+F4

2） Submenu of 'Test'

Submenu	Function	Shortcut
Radon Point	Test the Radon Point generation algorithm	Ctrl+R
Input Point Set	Input original point set （three input manner）	Ctrl+I
Random Sample	Randomly sampling from input point set	Ctrl+S
Run	Run Center Point generation algorithm，single step/continuous patterns are available	Single: F10 Continous: F5
Validate	Test the quality of Center Point	Ctrl+V
Clear	Clear screen and last input	F4

3） Submenu of ‘Help’

Submenu	Function	Shortcut
How to use	Display shortcut key list	F1
About CenterPoint	Brief description of the function, version and developers of this software	Ctrl+A

6 Experimental Results

Our aim: Since L is power of 4, i.e. L=4,16,64,256…. Our aim is for all the value n (the size of input point set) that lie in [10, 500,000], find suitable value L for the iterated algorithm, for both efficiency and high probability of correctness.

The main experimental work are done in our lab, on the computer with Intel Pentium 4 CPU 2.80GHz and 512MB of RAM. Which can compute and validate a center point of a set of 100 thousand input points in 1/20 minute. Although in the process a virus attack caused a serious interruption, we are glad that we obtain the experimental data needed at last.

Interpretation of the Tables: Table i (i=1,2,…) show our experiment results. For instance in **Table 1.** For each input size n=35,40,… 95, we run the algorithm 5000 times, each time for different input point set (which are generated by system randomly and uniformly) of n points, in each implementation of the algorithm we randomly select L=256 points from the input set to build the iteration tree. Then we count the times that c, the center point found by the algorithm, failed to be a β-center (recall that a point $c \in R^d$ is called a *β-center* if every closed halfspace containing c contains at least βn points of P), for β=1/3 and β=1/3-1/30 respectively. We calculate the failure ratio (which approximates failure probability) as follows:

Failure Ratio = (Failure times)/(Total Run Times)

From the experiment results, we can obtain the following conclusions:

- **Conclusion I: Selecting L=256 points is recommended when the size of the point set $|P| \in [60, 500{,}000]$. An (1/3 – 1/30) -center can be found with small error probability (<0.01).**

The theoretically recommended value for L is implied in [1] to be $9\log^2_2 n$, which is 1024 for $n \in (40, 1625]$ and 4096 for $n \in (1625, 2{,}642{,}245]$. Moreover the theoretical result in [1] only ensure an 1/12-center. Thus the algorithm actually has much better performance than that can be seen by theoretical analysis.

Table 1 - 4 below show performance of the algorithm, for the input size lie in different intervals and select L=256 points. From the results we can see, although the existence of a 1/3-center is theoretically proved, a little tradeoff between the partition ratio and success probability seems a good choice. I.e. if we require a (1/3-1/30)-center rather than a 1/3-center (which suffices for most of the applications), the failure probability will be much reduced.

Table 1.	Failure Ratio for n<100, L=256 Points Selected, Run 5000 times.												
Number of points n:	35	40	45	50	55	60	65	70	75	80	85	90	95
Failure times (β<1/3)	484	619	289	317	364	217	181	305	148	209	212	154	168
Failure ratio (%) (β<1/3)	9.68	12.38	5.78	6.34	7.28	4.34	3.62	6.1	2.96	4.18	4.24	3.08	3.36
Failure times (β<1/3 – 1/30)	163	76	105	54	68	40	39	19	49	28	29	27	30
Failure ratio (%) (β<1/3 – 1/30)	3.26	1.52	2.1	1.08	1.36	0.8	0.78	0.38	0.98	0.56	0.58	0.54	0.6

Table 2.	Failure Ratio for 100 ≤ n<1000, L=256 Points Selected, Run 5000 times.								
Number of points n:	100	200	300	400	500	600	700	800	900
Failure times (β<1/3)	212	92	60	62	64	57	53	68	36

Failure ratio (%) ($\beta<1/3$)	4.24	1.84	1.2	1.24	1.28	1.14	1.06	1.36	0.72
Failure times ($\beta<1/3-1/30$)	31	9	8	8	7	4	13	12	5
Failure ratio (%) ($\beta<1/3-1/30$)	0.62	0.18	0.16	0.16	0.14	0.08	0.26	0.24	0.1

Table 3.	Failure Ratio for $1000 \le n \le 10{,}000$. L=256 Points Selected, Run 5000 times.									
Number of points n:	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Failure times ($\beta<1/3$)	45	23	43	44	34	45	44	32	40	46
Failure ratio (%) ($\beta<1/3$)	0.9	0.46	0.86	0.88	0.68	0.9	0.88	0.64	0.8	0.92
Failure times ($\beta<1/3-1/30$)	8	0	5	9	5	4	5	6	3	5
Failure ratio (%) ($\beta<1/3-1/30$)	0.16	0	0.1	0.18	0.1	0.08	0.1	0.12	0.06	0.1

Table 4.	Failure Ratio for $20{,}000 \le n < 500{,}000$ L=256 Points Selected												
	Run 2000 times.								Run 500 times.				
n (thousands):	20	30	40	50	60	70	80	90	100	200	300	400	500
Failure times ($\beta<1/3$)	12	12	19	9	12	12	15	14	4	7	6	4	3
Failure ratio (%) ($\beta<1/3$)	0.6	0.6	0.95	0.45	0.6	0.6	0.75	0.7	0.8	1.4	1.2	0.8	0.6
Failure times ($\beta<1/3-1/30$)	3	1	4	2	3	2	3	3	0	2	0	1	1
Failure ratio (%) ($\beta<1/3-1/30$)	0.15	0.05	0.2	0.1	0.15	0.1	0.15	0.15	0	0.4	0	0.2	0.2

- **Conclusion II: Selecting L=1024 (or more) points give (1/3 – 1/30) -center with small error probability (<0.01). for any input size $|P| \in [10, 500{,}000]$.**

For $|P| \in [1,10]$, the error probability fluctuates drastically, this randomized algorithm doesn't work well for $n \in [1,10]$ even L=1024 points selected.

Table 5. below show performance of our algorithm, for input size lie in [10, 300] and select L=1024 points. From the experimental results we can see that selecting L=1024 points is enough for n in this interval.

For input size larger than 300, since selecting 256 points already gives good solution, selecting 1024 points will has at least the same correct probability. But the running time to calculate p (which is proportional to L) will be long.

For small input size, from **table 1** say less than 60, classical deterministic algorithm such as linear programming will also be good choice.

Table 5.	Failure Ratio for $10 \le n \le 300$, L=1024 Points Selected, Run 5000 times.													
Number of points n:	10	20	30	40	50	60	70	80	90	100	150	200	250	300
Fail times (β<1/3)	1696	212	24	57	13	0	26	6	9	9	0	0	0	0
Fail ratio (%) (β<1/3)	33.92	4.24	0.48	1.14	0.26	0	0.52	0.12	0.18	0.18	0	0	0	0
Fail times (β<1/3 − 1/30)	6	6	1	5	0	0	0	0	0	2	0	0	0	0
Fail ratio (%) (β<1/3 − 1/30)	0.12	0.12	0.02	0.1	0	0	0	0	0	0.04	0	0	0	0

- **Conclusion III: L=64 (or less) is not recommended. Selecting L=64 points give (1/3 − 1/30) -center with relatively large error probability (>0.05).**

Table 6.	Failure Ratio for $50 \le n \le 900$, L=64 Points Selected, Run 5000 times.													
Number of points n:	50	60	70	80	90	100	200	300	400	500	600	700	800	900
Fail times ($\beta<1/3$)	1788	1820	1374	1289	802	2746	916	923	1133	1034	755	900	850	1002
Fail ratio (%) ($\beta<1/3$)	35.76	36.4	27.48	25.78	16.04	54.92	18.32	18.46	2266	20.68	15.1	18	17	20.04
Fail times ($\beta<1/3-1/30$)	822	538	467	656	613	999	415	533	378	532	314	320	470	359
Fail ratio (%) ($\beta<1/3-1/30$)	16.44	10.76	9.36	13.12	12.26	19.98	8.3	10.66	7.56	10.64	6.28	6.4	9.4	7.18

Table 7.	Failure Ratio for $1000 \le n \le 9000$, L=64 Points Selected, Run 5000 times.								
Number of points n:	1000	2000	3000	4000	5000	6000	7000	8000	9000
Failure times ($\beta<1/3$)	772	809	821	795	821	826	754	800	841
Failure ratio (%) ($\beta<1/3$)	15.44	16.18	16.42	15.9	16.42	16.52	15.08	16	16.82
Failure times ($\beta<1/3-1/30$)	284	340	347	357	342	361	362	343	358
Failure ratio (%) ($\beta<1/3-1/30$)	5.68	6.8	6.94	7.14	6.84	7.22	7.24	6.86	7.16

Table 8.	Failure Ratio for $10{,}000 \le n \le 500{,}000$, L=64 Points Selected, Run 500 times.									
n (thousands):	10	20	40	60	80	100	200	300	400	500
Failure times ($\beta<1/3$)	77	84	68	68	71	72	68	82	82	86
Failure ratio (%) ($\beta<1/3$)	15.4	16.8	13.6	13.6	14.2	14.4	13.6	16.4	16.4	17.2
Failure times ($\beta<1/3-1/30$)	18	38	25	24	28	34	31	34	35	38
Failure ratio (%) ($\beta<1/3-1/30$)	3.6	7.6	5	4.8	5.6	6.8	6.2	6.8	7	7.6

Above all, for reasonable value n (lie in [60, 500,000]) L=256 is a good choice. An (1/3-1/30)-center can be found with error probability less than 0.01. The time need to compute and validate a (1/3-1/30)-center due to different input size is shown below:

Running Time Using Intel Pentium 4 CPU 2.80GHz and 512MB of RAM		
Input size n:	$10 \le n \le 100{,}000$	$100{,}000 \le n \le 500{,}000$
Time to compute and validate:	≤ 3 **s**	$\le 3n/100{,}000$ **s** ≤ 15 **s**

We can see that this algorithm works quite well in practice, and has much better performance than the theoretical proved result in [1].

7 Unsolved Issues and Further Improvement

1. For the display restriction, we set the coordinates of the input points to be integer. So there are 1024×768 different positions for any input point. When input size n is small, the influence of this restriction can be omitted; but when n is large, say $n=\Theta(1024 \times 768)$, the integer restriction will lose some generalities, so the analysis of our experimental results can not extend to the case $n \ge \Theta(1024 \times 768)$.

2. Our validate algorithm is simply for each rotating line pass through p, with degree $d \in \{1^\circ, 2^\circ, 3^\circ, ..., 360^\circ\}$, check each input point which side it lies to the line. It is almost surely can be improved with more careful method, say by incremental algorithm.

3. We only test the algorithm for uniform distribution point set in the plane. Although the analysis [1] does not depend on the distribution, it only ensure an 1/12-center. The algorithm may perform differently for different distributions. This needs more analysis or experimental test about the algorithm.

8 References

[1]Kenneth L.Clarkson, David Eppstein, Gary L. Miller, Carl Sturtivant, and Shang-Hua Teng. ***Approximating center points with iterated Radon points.*** Int. J.

Computational Geometry & Applications 6(3):357-377, Sep 1996. Proc. 9th Symp. Computational Geometry, ACM, May 1993, pp. 91-98.