软件工程 实验报告

姓名	班级	学号
聂路	软工1506	U201517146
李泽康	数媒1501	U201517164
齐达	软工1506	U201517128
明煦智	软工1503	U201517066

如何运行（Python 2.7）

```
git clone https://github.com/MartinNey/se-draw-balls.git
cd se-draw-balls
pyvenv .venv
. .venv/bin/activate
pip install -r requirements.txt
python main.py
```

算法

- 将所有的面，点，球抽象为‘限制’，而很容易得知，在满足题设条件的球，必定与四个限制‘相切’（面相切、点在球面上、两个球相切）。
- 所以这个问题就转换成了，求出，所有四个限制的组合，求出每一个组合中满足条件的球，找到其中半径最大的那个球。
- 那么开始具体过程

    1. 传进初始的限制 `list` （即四条的限制，或者需要添加的障碍点）、你初始添加的圆 `list` （默认为空）、需求的圆的个数 `number`
    2. 如果圆的列表长度达到了需要的个数，将结果返回，反之继续。
    3. 求目前最大的圆将圆添加进限制列表以及圆列表，将这两个量连同需要圆个数记录下来返回步骤2。

        > 关于如何求最大的圆：
        >
        > 1. 得到所有的四个限制的组合。
        > 2. 求出每个组合中满足条件的圆。
        > 3. 比较得出最大的圆。

    4. 得出结果，进行数据展示以及可视化。

- 关于优化空间：

    - 求四个限制组合以及最大圆的过程，可以使用动态规划，可以大幅度降低计算量。
    - 目前解方程这个过程是通过库计算的，理论上可以通过手算获取最后的结果表达式，可以大幅降低计算量。
    - 可以将限制的组合，优化为区域，区域间不重复，可以大幅降低计算量，但需要额外判断。

测试用例

```
    # 带了障碍点的情况
    restrictions_3d = [
        Point_3D({ 'x': 40, 'y': 100, 'z': 40 }),
        Coordinate('x', border=0, is_max=False),
        Coordinate('x', border=200, is_max=True),
        Coordinate('y', border=0, is_max=False),
        Coordinate('y', border=200, is_max=True),
        Coordinate('z', border=0, is_max=False),
        Coordinate('z', border=200, is_max=True),
    ]
    # n 为10, 24
    balls, restrictions = calculate_3d(n, balls, restrictions_3d, real_time_callback=  lambda ball: print(ball.dictify()))
    # 没带障碍点的情况
        restrictions_2d = [
        Point_2D({ 'x': 40, 'y': 100 }),
        Coordinate('x', border=0, is_max=False),
        Coordinate('x', border=200, is_max=True),
        Coordinate('y', border=0, is_max=False),
        Coordinate('y', border=200, is_max=True),
    ]
    # n 为10, 24, 50
    circles, restrictions = calculate_2d(n, circles, restrictions_2d, real_time_callback=  lambda circle: print(circle.dictify()))
```

代码

该有注释的地方应该都有注释

main.py

导入其他部分用于计算的库，计算结果，并且显示，可以在这里面做可视化，也可以将数据导出在其他地方显示。

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

from common import Point_2D, Point_3D, Circle, Ball, Coordinate, combination_traversal
from calculate_2d import calculate_2d
from calculate_3d import calculate_3d

def main():
    """
    example on how all these work
    """

    balls = []
    restrictions_3d = [
        Point_3D({ 'x': 40, 'y': 100, 'z': 40 }),
        Coordinate( 'x', border=0, is_max=False),
        Coordinate( 'x', border=200, is_max=True),
        Coordinate( 'y', border=0, is_max=False),
        Coordinate( 'y', border=200, is_max=True),
        Coordinate( 'z', border=0, is_max=False),
        Coordinate( 'z', border=200, is_max=True),
    ]
    balls, restrictions = calculate_3d( 12, balls, restrictions_3d, real_time_callback= lambd
a ball: print(ball.dictify()))

    circles = []
    restrictions_2d = [
        Point_2D({ 'x': 40, 'y': 100 }),
        Coordinate( 'x', border=0, is_max=False),
        Coordinate( 'x', border=200, is_max=True),
        Coordinate( 'y', border=0, is_max=False),
        Coordinate( 'y', border=200, is_max=True),
    ]
    circles, restrictions = calculate_2d( 12, circles, restrictions_2d, real_time_callback= l
ambda circle: print(circle.dictify()))
if __name__ == "__main__":
    main()
```

common.py

包含一些公用部分的实现，多个限制的类，从 m 中取 n 个组合的实现。

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

NEARLY_ZERO = 0.001

class Restriction(object):
    """
    Restriction.
    Actually this is unnecessary.
    There is no common things but method name.
    """
    def __init__(self):
        super(Restriction, self).__init__()

    def is_valid(self, point):
        return True

    def is_tangent_to(self, point):
        return 0

    def dictify(self):
        return {}

class Point(Restriction):
    """
    Point
    """
    def __init__(self, point):
        super(Point, self).__init__()
        self.x, self.y = point['x'], point['y']

    def is_valid(self, ball):
        center, radius = ball.center, ball.radius
        return self.distant_to_pow(center.dictify()) - radius** 2 > -NEARLY_ZERO

    def distant_to_pow(self, point):
        x, y = point['x'], point['y']
        return (self.x - x) ** 2 + (self.y - y) ** 2

    def is_tangent_to(self, circle):
        center, radius = circle['center'], circle['radius']
        return self.distant_to_pow(center.dictify()) - radius** 2

    def dictify(self):
        return { 'x': self.x, 'y': self.y }


class Point_2D(Point):
    """
    Point_2D
    """
    def __init__(self, point):
        super(Point_2D, self).__init__(point)
        self.x, self.y = point['x'], point['y']

    def distant_to_pow(self, point):
        x, y = point['x'], point['y']
        return (self.x - x) ** 2 + (self.y - y) ** 2

    def dictify(self):
        return { 'x': self.x, 'y': self.y }
```

```
class Point_3D(Point):
    """
    Point_3D
    """
    def __init__(self, point):
        super(Point_3D, self).__init__(point)
        self.x, self.y, self.z = point['x'], point['y'], point['z']

    def distant_to_pow(self, point):
        x, y, z = point['x'], point['y'], point['z']
        return (self.x - x) ** 2 + (self.y - y) ** 2 + (self.z - z) ** 2

    def dictify(self):
        return { 'x': self.x, 'y': self.y, 'z': self.z }

class Coordinate(Restriction):
    """
    Coordinate.
    Limit x, y, z, by (max||min) borders.
    Can be optimized by passing function as argument.
    """
    def __init__(self, attr, border, is_max=True):
        super(Coordinate, self).__init__()
        self.attr = attr
        self.border = border
        self.is_max = is_max

    def in_range(self, value):
        if self.is_max and value <= self.border:
            return True
        if not self.is_max and value >= self.border:
            return True
        return False

    def is_valid(self, ball):
        center, radius = ball.center, ball.radius
        in_range = self.in_range(center.dictify()[self.attr])
        in_distant = self.distant_to_pow(center.dictify()) - radius **   2 > -NEARLY_ZERO
        return in_range and in_distant

    def distant_to_pow(self, point_3D):
        req_coordinate = point_3D[self.attr]
        distant = abs(req_coordinate - self.border)
        return distant ** 2

    def is_tangent_to(self, ball):
        center, radius = ball['center'], ball['radius']
        return self.distant_to_pow(center.dictify()) - radius** 2

    def dictify(self):
        return { 'attr': self.attr, 'border': self.border, 'is_max': self.is_max }

class Round_Like(Restriction):
    def __init__(self, center, radius):
        super(Round_Like, self).__init__()
        self.center = Point(center)
        self.radius = radius

    def is_valid(self, circle):
        center, radius = circle.center, circle.radius
        return self.center.distant_to_pow(center.dictify()) - (radius + self.radius)**  2 > -
```

```
NEARLY_ZERO

    def is_tangent_to(self, ball):
        center, radius = ball['center'], ball['radius']
        return self.center.distant_to_pow(center.dictify()) - (self.radius + radius)** 2

    def dictify(self):
        return { 'center': self.center.dictify(), 'radius': self.radius }

class Circle(Round_Like):
    """
    Circle
    """
    def __init__(self, center, radius):
        super(Circle, self).__init__(center, radius)
        self.center = Point_2D(center)

class Ball(Round_Like):
    """
    Ball
    """
    def __init__(self, center, radius):
        super(Ball, self).__init__(center, radius)
        self.center = Point_3D(center)

# pick all n-combination from all collections
def combination_traversal(combine_list, combine_num):
    # traversal end
    if len(combine_list) <= combine_num:
        return [combine_list]
    if combine_num == 0:
        return [[]]

    combinations = []

    comb_with_first = combination_traversal(combine_list[1:], combine_num - 1)

    for comb in comb_with_first:
        comb.insert(0, combine_list[0])

    combinations.extend(comb_with_first)

    comb_without_first = combination_traversal(combine_list[1:], combine_num)
    combinations.extend(comb_without_first)

    return combinations
```

calculate_2d

平面的实验一实验二计算部分的实现

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from common import Point_2D, Circle, combination_traversal
from scipy.optimize  import fsolve

LOG_LEVEL = 0

def calculate_2d(length, circles, restrictions, real_time_callback=lambda x: x)  :
    def max_circle(restriction):
        # TODO:this should be saved in each restriction for less calculation
        # but I don't want to do this
        '''get max circle in each part'''
        def get_formulas(array):
            x, y, r = array
            circle = {
                'center': Point_2D({
                    'x': x,
                    'y': y,
                }),
                'radius':r
            }
            return list(map(lambda re: re.is_tangent_to(circle), restriction))

        if len(restriction) > 3:
            # for comparing
            the_max_circle = Circle({  'x': 100, 'y': 100}, 0)

            all_four_combinations = combination_traversal(restriction,   3)
            for combination  in all_four_combinations:
                circle = max_circle(combination)
                all_valid =  True
                for single_restriction  in restriction:
                    valid = single_restriction.is_valid(circle)
                    if not valid:
                        if LOG_LEVEL > 0:
                            print(single_restriction.dictify(), valid, circle.dictify())
                        all_valid =  False
                        break
                if circle.radius >= the_max_circle.radius  and all_valid:
                    the_max_circle = circle
            return the_max_circle

        # if we have the previous circle, use its center as root point.
        # if not, use the default point.
        prev_circle = restriction[ 0]
        start_point_2D = [ 100, 100, 100]
        if hasattr(prev_circle,  'center'):
            start_point_2D = [prev_circle.center.x, prev_circle.center.y, prev_circle.radiu
s]

        x, y, r = fsolve(get_formulas, start_point_2D)

        return Circle({  'x': x, 'y': y}, r)

    # loop end
    if len(circles) >= length:
        return circles, restrictions

    the_max_circle = max_circle(restrictions)
    real_time_callback(the_max_circle)
```

```
        circles.append(the_max_circle)

        # put the largest circle at [0]
        restrictions.insert( 0, the_max_circle)

        return calculate_2d(length, circles, restrictions, real_time_callback)
```

calculate_3d

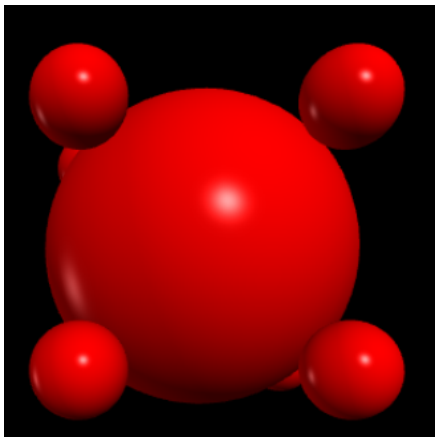
实验三，计算的具体实现

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from common import Point_3D, Ball, combination_traversal
from scipy.optimize  import fsolve

LOG_LEVEL = 0

def calculate_3d(length, balls, restrictions, real_time_callback=lambda x: x)  :
    def max_ball(restriction):
        # TODO:this should be saved in each restriction for less calculation
        # but I don't want to do this
        '''get max ball in each part'''
        def get_formulas(array):
            x, y, z, r = array
            ball = {
                'center': Point_3D({
                    'x': x,
                    'y': y,
                    'z': z,
                }),
                'radius':r
            }
            return list(map(lambda re: re.is_tangent_to(ball), restriction))

        if len(restriction) > 4:
            # for comparing
            the_max_ball = Ball({  'x': 100, 'y': 100, 'z': 100}, 0)

            all_four_combinations = combination_traversal(restriction,   4)
            for combination  in all_four_combinations:
                ball = max_ball(combination)
                all_valid =  True
                for single_restriction  in restriction:
                    valid = single_restriction.is_valid(ball)
                    if not valid:
                        if LOG_LEVEL > 0:
                            print(single_restriction.dictify(), valid, ball.dictify())
                        all_valid =  False
                        break
                if ball.radius >= the_max_ball.radius  and all_valid:
                    the_max_ball = ball
            return the_max_ball

        # if we have the previous ball, use its center as root point.
        # if not, use the default point.
        prev_ball = restriction[ 0]
        start_point_3D = [ 100, 100, 100, 100]
        if hasattr(prev_ball,  'center'):
            start_point_3D = [prev_ball.center.x, prev_ball.center.y, prev_ball.center.z, p
rev_ball.radius]

        x, y, z, r = fsolve(get_formulas, start_point_3D)

        return Ball({ 'x': x, 'y': y , 'z': z}, r)

    # loop end
    if len(balls) >= length:
        return balls, restrictions

    the_max_ball = max_ball(restrictions)
```

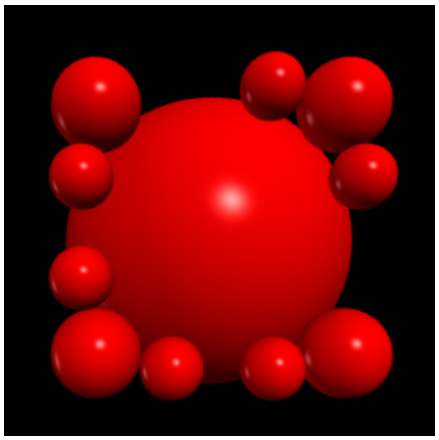
```
        real_time_callback(the_max_ball)
        balls.append(the_max_ball)

        # put the largest ball at [0]
        restrictions.insert( 0, the_max_ball)

        return calculate_3d(length, balls, restrictions, real_time_callback)
```

结果展示

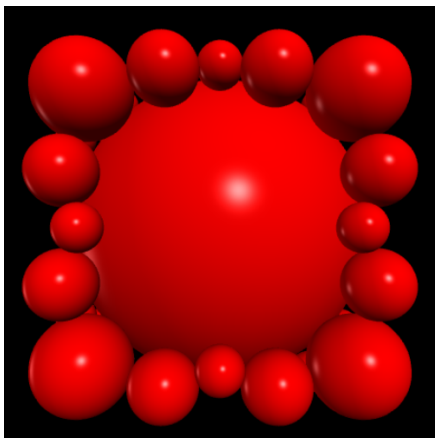
- 实验三，个数10



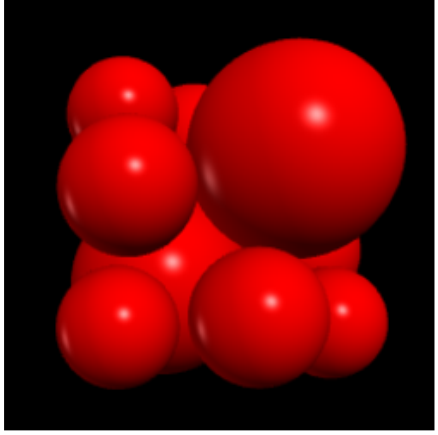
- 实验三，个数24



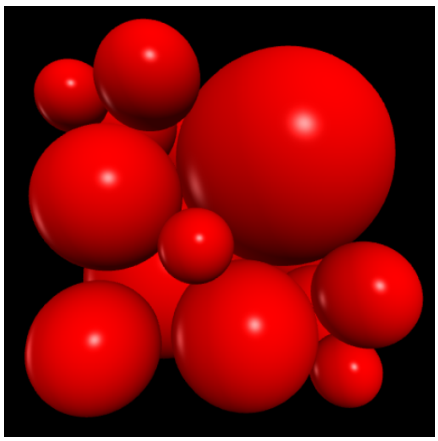
- 实验三，个数50



- 实验三（有障碍），个数10



- 实验三（有障碍），个数24



附录

```
   * a957c51 - (HEAD -> master, origin/master) update README (8 minutes ago) <MartinNey>
*   8ba6c13 - merge (24 minutes ago) <MartinNey>
|\
| * 5de5950 - add plot for 2d (51 minutes ago) <lizekang>
| * 632cffc - update readme.md (2 hours ago) <ZekangLi>
| * 3b145e5 - update readme (2 hours ago) <lizekang>
| * f4a8e99 - add screenshots (2 hours ago) <lizekang>
* | 6947e72 - add cb (59 minutes ago) <MartinNey>
|/
* 45cf72d - optimize structure (2 hours ago) <MartinNey>
* 51d9712 - update readme (4 hours ago) <MartinNey>
* 42cb912 - update readme (4 hours ago) <MartinNey>
* 5294fde - add 'log leve'l option (4 hours ago) <MartinNey>
* df8fed8 - first blood, 3d (4 hours ago) <MartinNey>
(END)
```