# 软件工程 实验报告

| 姓名 | 班级 | 学号 |

齐达 1506 U20157128

## 如何运行（Python 2.7）

```bash

git clone https://github.com/MartinNey/se-draw-balls.git

cd se-draw-balls

virtualenv .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

```python

   # 仅为说明测试用例，实际代码稍有不同

   # 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),# is\_max 意思是，该边界是否代表上限，假则说明这个代表下限

       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\_3d = [

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, 50

balls, restrictions = calculate\_3d(n, balls, restrictions\_3d, real\_time\_callback=lambda ball: print(ball.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。

```python

#!/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=lambda 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=lambda 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。

```python

#!/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

```python

#!/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.radius]

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、四，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, prev\_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

<img src="./images/3\_10.png" width=250 height=250>

- 实验三，个数24

<img src="./images/3\_24.png" width=250 height=250>

- 实验三，个数50

<img src="./images/3\_50.png" width=250 height=250>

- 实验四，个数10

<img src="./images/4\_10.png" width=250 height=250>

- 实验四，个数24

<img src="./images/4\_24.png" width=250 height=250>