Examen

Junio

16 de Junio 2015

Informática
Año 2014/2015
Facultad de CC.
Matemáticas

1. Ajuste de imagen [4 puntos]

En este problema se pretende representar la temperatura que hay en diversas partes de una placa rectangular de un determinado material. Para ello, tras aplicar diversas fuentes de calor de distinta intensidad en algunas de las partes del objeto, se realiza la medición de la temperatura de cada una de ellas y su resultado se transcribe en una matriz.

Cada elemento de la matriz almacenará la temperatura de una parte de la placa. Se ha valorado que la temperatura puede oscilar entre 30 y 99 grados centígrados (se redondea al entero más próximo).

Sin embargo, los investigadores creen necesario corregir posibles errores en la medición. **Consideran que se ha producido un error en la medición si, dado un elemento de la matriz, su valor difiere en más de 10 unidades de los valores de cada uno de sus vecinos** (8 vecinos, 5 ó 3 según la posición del elemento en la matriz).

La corrección consistirá en asignar a dicho elemento el valor resultante de calcular la media de los valores de sus vecinos redondeándola al entero más próximo. Ten en cuenta que el resultado de la corrección se debe registrar en otra matriz. En la figura **??** puedes ver un ejemplo de una matriz con temperaturas y la correspondiente corrección.

72	63	57	68	73	78	80	85	81	76
75	65	56	69	75	85	89	90	86	75
70	59	55	64	73	79	70	94	85	78
64	63	50	65	76	85	88	94	90	87
45	53	49	63	75	83	84	89	88	86
39	49	49	59	70	76	74	78	74	80
37	43	48	59	69	71	69	**40**	70	75
70	42	45	55	58	60	68	75	78	80
35	39	43	52	62	64	65	67	86	85
32	36	40	50	60	69	74	85	88	90

Tras la corrección:

72	63	57	68	73	78	80	85	81	76
75	65	56	69	75	85	89	90	86	75
70	59	55	64	73	79	70	94	85	78
64	63	50	65	76	85	88	94	90	87
45	53	49	63	75	83	84	89	88	86
39	49	49	59	70	76	74	78	74	80
37	43	48	59	69	71	69	**73**	70	75
39	42	45	55	58	60	68	75	78	80
35	39	43	52	62	64	65	67	86	85
32	36	40	50	60	69	74	85	88	90

Figura 1: Matriz de temperaturas con el resultado de su corrección

1. Generar temperaturas **[0.5 puntos]**

   Escribe una función que genere de forma aleatoria una matriz de 15×15 elementos con valores enteros en el conjunto $\{30, \ldots, 99\}$, equiprobablemente.

2. Corrección de las temperaturas **[2 puntos]**

   Escribe una función que obtenga una nueva matriz como resultado de realizar la corrección de las temperaturas a una matriz dada. La matriz original debe quedar sin modificar.

3. Punto fijo de la corrección **[1.5 puntos]**

   Fíjate que no basta una sola corrección para obtener una matriz libre de errores. Escribe una función que aplique la corrección de errores repetidamente hasta que el resultado no tenga ningún error.

2. Figuras geométricas: Polígonos [6 puntos]

Dadas las definiciones de las clases `Point`,`Vector`,`Segment` y `Line` (ver implementación al final).

1. **[1 punto]** Escribe un método en la clase `Vector` para calcular el ángulo (con signo) entre dos vectores. Para ello puedes notar que el producto vectorial de dos vectores $u = (u_x, u_y)$ y $v = (v_x, v_y)$ viene dado como: $u \times v = (u_x v_y - u_y v_x)$. Además $u \times v = |u| \cdot |v| \cdot sen(\alpha)$

2. **[1 punto]** Escribe el método `intersects_int(self,other)` en la clase Segment que indique si los interiores de los segmentos `self` y `other` intersecan.

3. **[4 puntos]** Escribe la clase `Polygon` que representa un polígono de cualquier número de lados. Un polígono queda determinado por una lista ordenada de puntos. Sus lados son los segmentos que unen cada punto con el siguiente, además de otro que une el último con el primero. Para que el polígono no sea *degenerado* cada segmento debe tener intersección vacía con todos (salvo sus dos contiguos, a los que debe intersecar sólo en un punto).

   - El constructor de la clase debe comprobar la corrección de los datos de entrada.
   - Se debe implementar un método `contains(self, other)` que indica si other, que puede ser un punto o un segmento, está totalmente contenido en el polígono.
   - La clase `Polygon` debe implementar el método `area` que calcule el área del polígono.

Pistas:

- Dado un polígono cualquiera de vértices $P_0, P_1, \ldots, P_n$ existe al menos una diagonal (`d = Segment(`P_i`, `P_j`)`) totalmente contenida en el polígono. El área del polígono original es la suma de las áreas de los dos polígonos obtenidos al *cortar* por `d`.

- Para definir `contains(self,other)` en el caso `Polygon-Point`:

  Sea P un punto y A el polígono determinado por la lista de puntos (ordenados) `l = [p0, p1, ..., pn]` consideramos α_i el ángulo (con signo) entre los vectores `P.vector_to(l[i])` y `P.vector_to(l[i+1])`. El valor $index = \sum \alpha_i$ puede ser: 0, $2 \cdot \pi$ o $-2 \cdot \pi$. Si $index = 0$ el punto está fuera del polígono, si no está dentro.

Listing 1: Código de las clases `Vector`, `Point`, `Line`, `Segment`

```
from sympy import *
def is_number_type(n):
    return isinstance(n,int) or isinstance(n,float) or isinstance(n,Expr)

class Vector(object):
    def __init__(self,vx,vy):
        if is_number_type(vx) and is_number_type(vy):
            self.x=vx
            self.y=vy
        else:
            raise Exception('Bad data for Vector construction')

    def __repr__(self):
        return 'Vector('+str(self.x)+','+str(self.y)+')'

    def dot(self,otro):
        return self.x*otro.x+self.y*otro.y

    def __add__(self,other):
        return Vector(self.x+other.x,self.y+other.y)

    def norm(self):
        return sqrt(self.dot(self))

    def unit(self):
        l=self.norm()
        return Vector(self.x/l,self.y/l)

    def ortogonal(self):
        return Vector(-self.y,self.x)

    def is_parallel(self,other):
        return self.ortogonal().dot(other)==0

class Point(object):
    def __init__(self,px,py):
        if is_number_type(px) and is_number_type(py):
            self.x=px
            self.y=py
        else:
            raise Exception('Bad data for Point construction')

    def __repr__(self):
        return 'Point('+str(self.x)+','+str(self.y)+')'

    def intersects(self,other):
        if isinstance(other,Point):
            return self.x==other.x and self.y==other.y
        else:
            return other.intersects(self)

    def distance(self,other):
        if isinstance(other,Point):
            return self.vector_to(other).norm()
        else:
            return other.distance(self)

    def __add__(self,v):
        return Point(self.x+v.x,self.y+v.y)

    def vector_to(self,other):
        return Vector(other.x-self.x,other.y-self.y)
```

```
class Line(object):
    def __init__(self,point,point1):
        if isinstance(point,Point) and isinstance(point1,Point)\
          and point.distance(point1)>0:
            self.p=point
            self.p1=point1
            self.v=point.vector_to(point1)
            self.normal=self.v.ortogonal().unit()
        else:
            raise Exception('Bad data for Line construction')

    def __repr__(self):
        return 'Line('+str(self.p)+','+str(self.p1)+')'

    def intersects(self,other):
        if isinstance(other,Point):
            v=self.p.vector_to(other)
            return v.is_parallel(self.v)
        elif isinstance(other,Line):
            if self.v.is_parallel(other.v):
                return self.intersects(other.p)
            else:
                return True
        else:
            return other.intersects(self)

    def distance(self,other):
        if isinstance(other,Point):
            v_orto=self.v.ortogonal().unit()
            return abs(v_orto.dot(self.p.vector_to(other)))
        elif isinstance(other,Line):
            if self.v.is_parallel(other.v):
                return self.distance(other.p)
            else:
                return 0
        else:
            return other.distance(self)
```

```
class Segment(object):
    def __init__(self,p1,p2):
        if isinstance(p1,Point) and isinstance(p2,Point) and p1.distance(p2)>0:
            self.end1=p1
            self.end2=p2
            self.v=p1.vector_to(p2)
            self.length=self.v.norm()
            self.unit=self.v.unit()
            self.l=Line(self.end1,self.end2)
        else:
            raise Exception('Bad data for Segment construction')

    def __repr__(self):
        return 'Segment('+str(self.end1)+','+str(self.end2)+')'

    def support(self):
        return self.l

    def intersects(self,other):
        if isinstance(other,Point):
            on_line=self.l.intersects(other)
            proy=self.unit.dot(self.end1.vector_to(other))
            return on_line and (0<=proy and proy<=self.v.norm())
        elif isinstance(other,Line):
            v_ort=other.v.ortogonal()
            proy1=other.p.vector_to(self.end1).dot(v_ort)
            proy2=other.p.vector_to(self.end2).dot(v_ort)
            return proy1*proy2<=0
        elif isinstance(other,Segment):
            if self.v.is_parallel(other.v):
                return self.intersects(other.end1) or self.intersects(other.end2) or\
                       other.intersects(self.end1)
            else:
                return other.intersects(self.l) and self.intersects(other.l)
        else:
            return other.intersects(self)

    def distance(self,other):
        if isinstance(other,Point):
            proyection=self.unit.dot(self.end1.vector_to(other))
            if proyection>self.length:
                return self.end2.distance(other)
            elif proyection>=0:
                return Line(self.end1,self.end2).distance(other)
            else:
                return self.end1.distance(other)
        elif isinstance(other,Line):
            if self.intersects(other):
                return 0
            else:
                return min(self.end1.distance(other),self.end2.distance(other))
        elif isinstance(other,Segment):
            if self.intersects(other):
                return 0
            else:
                return min(self.end1.distance(other),self.end2.distance(other),\
                           other.end1.distance(self),other.end2.distance(self))
        else:
            return other.distance(self)
```