Problemas de Geometría

1. Defina una estructura `Circulo` que guarde las coordenadas del centro de un disco y su radio. Después, escriba una función que reciba dos círculos y devuelva 1 si ambos se superponen o 0 si no lo hacen.

   Solución:

```
struct Circulo {
    float x,y;
    float r;
};

int circulo_super(struct Circulo c1, struct Circulo c2) {
    return (c1.x-c2.x)*(c1.x-c2.x)+(c1.y-c2.y)*(c1.y-c2.y) < (c1.r+c2.
        r)*(c1.r+c2.r);
}
```

2. Defina una estructura `Treslados` que guarde las longitudes de tres segmentos. Para que con estos tres segmentos se pueda construir un triángulo, sus longitudes deben verificar la desigualdad triangular: cada lado no puede ser mayor que la suma de los otros dos. Escriba una función que devuelva 1 si la estructura es válida para construir un triángulo o 0 si no lo es.

   Solución:

```
struct Treslados {
    float l1,l2,l3;
};

int es_triangulo(struct Treslados a) {
    return (a.l1 < a.l2+a.l3) && (a.l2 < a.l1+a.l3) && (a.l3 < a.l1+a.l2);
}
```

3. Defina una estructura `Rectangulo` que almacene las coordenadas de un rectángulo genérico, es decir, las coordenadas X,Y de dos de sus vértices opuestos: el inferior izquierdo y el superior derecho (asuma que siempre se guardan en el mismo orden). Después, escriba una función que reciba dos estructuras Rectangulo y un puntero a una tercera y que retorne 0 en caso de que no se intersequen los dos primeros, o 1 en caso de que sí lo hagan. En este último caso, la función rellenará además la tercera estructura con las coordenadas correspondientes a la intersección (que será también un rectángulo).

   Solución:

```
struct Rectangulo {
    float x1,y1;
    float x2,y2;
};

#define min(a,b) (((a)<(b))?(a):(b))
```

```
#define max(a,b) (((a)>(b))?(a):(b))

int rectangulo_interseca(struct Rectangulo r1, struct Rectangulo r2,
                         struct Rectangulo *r3)  {
    if( r1.x2 < r2.x1 || r1.x1 > r2.x2
     || r1.y2 < r2.y1 || r1.y1 > r2.y2 ) {
        return 0;
    } else {
        r3->x1 = max(r2.x1,r1.x1);
        r3->y1 = max(r2.y1,r1.y1);
        r3->x2 = min(r1.x2,r2.x2);
        r3->y2 = min(r1.y2,r2.y2);
    }
    return 1;
}
```

4. Defina una estructura `Punto` que almacene las coordenadas de un punto del plano y otra estructura `Triangulo` que almacene los puntos que definen los vértices de un triángulo del plano. Después, escriba una función que reciba un `Triangulo` y un `Punto` y determine si el punto se halla dentro del área del `Triangulo` (y retorne, entonces, 1) o fuera de él (y retorne 0).

   [**Nota**: un punto P se halla en el interior de un triángulo si los productos vectoriales de los vectores que unen P con cada par de vértices consecutivos tienen el mismo signo]

   Solución:

```
struct Punto {
    float x,y;
};

struct Triangulo {
    struct Punto a, b, c;
}

float area(struct Punto o, struct Punto a, struct Punto b) {
    return (a.x-o.x)*(b.y-o.y)-(a.y-o.y)*(b.x-o.x);
}

int dentro_triangulo(struct Triangulo t, struct Punto p) {
    float a1, a2, a3;
    a1=area(p, t.a, t.b);
    a2=area(p, t.b, t.c);
    a3=area(p, t.c, t.a);
    if( (a1>0 && a2>0 && a3>0)
     || (a1<0 && a2<0 && a3<0) )
        return 1;
    return 0;
}
```

5. Se quiere buscar el mejor recubrimiento de un rectángulo de lados W y H por círculos de radio R. Para ello, se van colocando los centros de éstos en puntos del interior del rectángulo, tomados aleatoriamente (calcúlense usando la función `uniforme()` que devuelve un número pseudoaleatorio entre 0 y 1). Cada vez que se escoge un punto, se comprueba que el círculo no colisionará con los anteriores; si lo hace,

se escoge otro centro y se vuelve a probar. Si después de un número K de intentos fallidos no se ha podido añadir un nuevo círculo, se considera que se ha completado el recubrimiento. Escríbase una función que haga esto:

```
int recubre(float W, float H, float R, float Xc[], float Yc[])
```

W, H: dimensiones del rectángulo y dos arrays de números flotantes

R: radio de los círculos del recubrimiento

Xc, Yc: arrays en los que se guardarán las coordenadas de los centros de los círculos del recubrimiento

La función devolverá el número de círculos que ha logrado colocar.

Solución:

```
#define K 100
int recubre(float W, float H, float R, float Xc[], float Yc[]) {
    int n, nc, ni;
    float x, y;
    nc=0;
    ni=0;
    while( ni < K ) {
        x=W*uniforme();
        y=H*uniforme();
        for(n=0; n < nc; nc++) {
            if( hypot(x-Xc[n], y-Yc[n]) < 2*R ) {
                ni++;
                break;
            }
        }
        if( ni == 0 ) {
            Xc[nc]=x;
            Yc[nc]=y;
            nc++;
            ni=0;
        }
    }
    return nc;
}
```

6. Declare una estructura que represente un poliedro: que guarde el número de vértices de éste (menor o igual que 20) y sus coordenadas XYZ. Defina una función que calcule la distancia entre dos cualesquiera de esos vértices, con la salvaguarda de que ambos vértices pertenezcan al poliedro (en caso contrario deberá devolver el valor NAN, *not a number* o indeterminado).

Solución:

```
#define NVERTICES_MAX 20
struct Poliedro {
    int nVertices;
    float x[NVERTICES_MAX],
          y[NVERTICES_MAX],
          z[NVERTICES_MAX];
};
float dVertices(struct Poliedro* p, int na, int nb) {
```

```
    float dx, dy, dz, d=NAN;
    if( 0<=na && na<p->nVertices && 0<=nb && nb<p->nVertices ) {
        dx=p->x[na]-p->x[nb];
        dy=p->y[na]-p->y[nb];
        dz=p->z[na]-p->z[nb];
        d=sqrt(dx*dx+dy*dy+dz*dz);
    }
    return d;
}
```

7. **Problema (6 puntos).** Supongamos que tenemos un array de tamaño $N \times N$ relleno con unos y ceros de modo aleatorio. El objetivo de este ejercicio es diseñar una función externa a `main`, que reciba ese array y que calcule la masa M contenida en un círculo de radio r centrado en el centro del array, para diferentes valores de r. Se define la masa $M(r)$ como el número de celdas con valor 1 dentro del círculo de radio r. La función debe exportar a un archivo de texto (cuyo nombre es pasado también como argumento de la función) dos columnas de datos: la primera columna debe ser r y la segunda el valor correspondiente de $M(r)$. Asumiremos que N es impar, de modo que el centro de la matriz está bien definido. El radio variará como $r = n \cdot r_0$ con $n = 1, 2, 3, \ldots$ hasta $n_{\max} = (N-1)/(2r_0)$, donde r_0 es otro parámetro que debemos pasar en forma de argumento a la función. De este modo, el radio máximo será $r_{\max} = n_{\max} \cdot r_0 = (N-1)/2$ que representa la circunferencia inscrita en la matriz.

   La declaración de la función deberá tener la forma:

   ```
   void calcula_masa (int N, int matriz[][N], int r0, char* nombre_archivo)
   ```

   Nota importante: Para que el algoritmo sea eficiente, la matriz con los datos deberá ser recorrida **una única vez**. Para ello debemos hacer uso de la propiedad de que cada elemento contenido en el interior de un circulo de radio r también lo estará en todos los círculos de radio mayor que r.

   Solución: Ver código `calcula_masa_array.c`

8. Para algunos problemas se requiere disponer de vectores que apunten a puntos aleatorios distribuidos uniformemente sobre la superficie de la esfera unitaria (de radio 1), esto es, vectores que unan el centro de ésta con puntos aleatorios de su superficie. Una forma de generar estos vectores aleatorios es la siguiente: generar ternas de números aleatorios (x, y, z) en el cubo $[-1, +1] \times [-1, +1] \times [-1, +1]$, descartar aquellas ternas cuyo módulo ($l = \sqrt{x^2 + y^2 + z^2}$) sea menor que 0.1 o mayor que 1 y, por último, dividir las coordenadas por el módulo para obtener $(x/l, y/l, z/l)$, que es un vector de norma unidad. **Escriba una función** que reciba un array de números reales (declarado como **double** u[3]) y lo rellene con las componentes de un vector unitario aleatorio usando el procedimiento antes descrito.

   Nota: Use la función `rand()` para generar los números aleatorios en el cubo. Recuerde que para generar un número aleatorio entre 0 y 1, puede usar:

   ```
   r=rand()/(double)RAND_MAX;
   ```

   Solución:

   ```
   void uVector(double u[3]) {
     int n;
     double l;
     do {
       l=0.0;
       for(n=0; n<3; n++) {
   ```

```
      u[n]=-1.0 + 2*rand()/(double)RAND_MAX;
      l+=u[n]*u[n];
    }
    l=sqrt(l);
    if( 0.1 <= l && l <= 1.0 ) {
      for(n=0; n<3; n++) {
        u[n]=u[n]/l;
    }
  } while(l < 0.1 || 1.0 < l);
  return;
}
```