CS 100: Programming Assignment P7

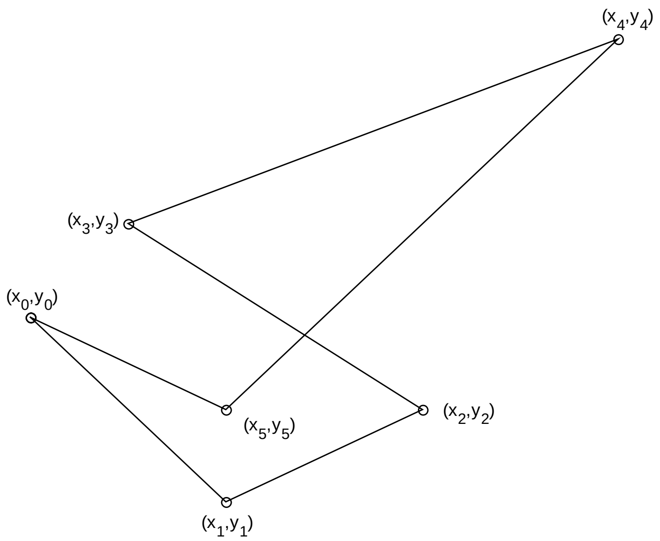
Due: Friday, May 7, 5pm, Carpenter Lab (or in lecture)

You may work in pairs. Do not submit your assignment for grading unless you have read and understand the CS100 webpage on Academic Integrity. Follow the course rules for the submission of assignments or lose points.

Part A. Polygon Computations

Background

Suppose we are given *n* points $(x_0 , y_0),\ldots,(x_{n-1} , y_{n-1})$ in the plane and connect them in order with "wraparound" so that the point (x_{n-1} , y_{n-1}) is connected to the point (x_0 , y_0), e.g.,



We call this a *closed polyline*. The following class can be used to represent such a geometric entity:

```
public class closedPolyline {
   protected point alpha;          // The reference point.
   protected int n;                // Number of vertices.
   protected point[] relVertex;    // The array of vertices.

   // The Constructors:
   public closedPolyline(){}

   public closedPolyline(point alpha_point, point[] v){
       alpha = alpha_point;
       n = v.length;
       relVertex = new point[n];
       for(int k=0;k<n;k++)
          relVertex[k] = new point(v[k]);
    }

    // Yields an array whose k-th component is the k-th absolute vertex.
    public point[] get_absVertex(){
       point[] q = new point[n];
       for (int k=0;k<n;k++)
          q[k] = new point(alpha.get_x() + relVertex[k].get_x(),alpha.get_y() + relVertex[k].get_y());
       return q;
    }

    // Translates each absolute vertex a units in the x-direction and b units in the y-direction.
    public void translate(double a, double b){
       alpha.translate(a,b);
    }
    // Yields the length of the closed polygonal line
    public double perimeter() {
       double s=0;
       for(int k=0;k<n;k++)
          s = s + point.dist(relVertex[k],relVertex[(k+1)%n]);
       return s;
    }
}
```

Notice the reliance on the class `point`. (Specifications of its members are given at the end of the handout.) In order to understand the `closedPolyline` representation you need to understand the role of the *reference point* and the difference between the *relative vertices* and the *absolute vertices*. This is most easily done with an example. If the relative vertices are

(3,4)   (5,-7)   (9,0)   (3,2)   (-8,-9)   (0,0)

and the reference point is (10,20), then the absolute vertices are given by

(13,24)   (15,13)   (19,20)   (13,22)   (2,11)   (10,20)

The absolute vertices prescribe the location of the polyline. With this representation, translation of the polyline is simply a matter of translating the reference point. In particular, to translate the polyline a units in the x-direction and b units in the y-direction, we simply add those amounts to the x and y components of the reference point.

Note that the class `closedPolyline` has three fields: an `int` field for the number of vertices, a `point` field for the reference point, and a `point[]` field for the relative vertices. Study the methods in this class and how they rely upon the members of the class `point`.

We'll define a *polygon* to be a closed polyline with no edge crossovers. Note that operations like translation and perimeter apply without modification to polygons. So as a piece of software, `closedPolyline` is relevant should we be interested in moving polygons around and computing perimeters. However, if we want to compute polgon centroids then we certainly have to write some code. In this problem you develop a subclass of the class `closedPolyline` called `polygon` that can be used to represent and manipulate polygons as polylines, but which can also perform the necessary centroid calculations.

Start by copying the files `P7A.java` and `Geometry.java` from the website. The latter contains the `point` class and the `closedPolyline` class. You can place your `polygon` class in `Geometry.java` as well. That way you need only work with two files in the project. Set the Java main class target to `P7A` and proceed to develop

```
public class polygon extends closedPolylines {. . .}
```

as follows:

The Constructors

No new fields are required for polygons—they are just polylines with no crossovers. The mission of a `polygon` constructor is simply to build prescribed polylines with no edge crossovers. An easy way to start out is to have a constructor that builds *regular* polygons. (A regular polygon has equal sides and equal interior angles.) Implement a regular polygon constructor

```
public polygon(point alpha_point, int nSides, double r)
```

It should assign `alpha_point` to `alpha`, `nSides` to `n`, and set up the relative vertex array `relVertex` so that its k-th component represents the point ($r\cos(2\pi\, k/n)$, $r\sin(2\pi\, k/n)$). In effect the reference point is the polygon's center.

For more general polygons write a constructor that works with a polar coordinate specification of the relative vertices:

```
public polygon(point alpha_point, double[] theta, double[] r)
```

Once again, we assign `alpha_point` to `alpha`. But now we set `n=theta.length=r.length` and define the k-th relative vertex to be (r_k cos(θ_k) , r_k sin(θ_k)). We assume that the incoming polar coordinate values prescribe distinct relative vertices and that they satisfy three criteria:

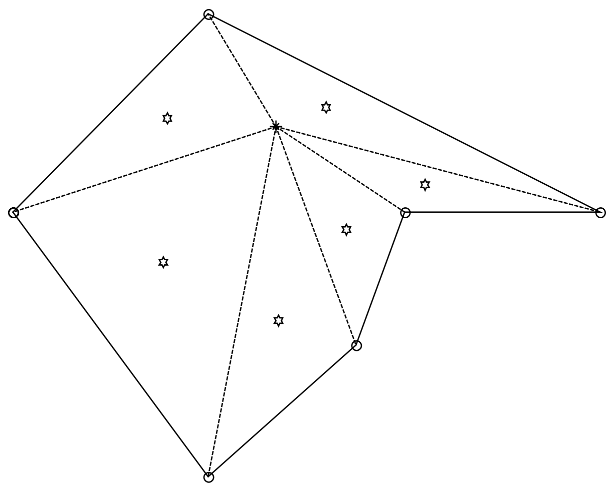
(a) $0 <= \theta_0 <= \theta_1 <= ... <= \theta_{n-1} <= 2\pi$
(b) $r_0 > 0 ,\ldots, r_{n-1} > 0$.
(c) $\theta_0 + \pi <= \theta_{n-1}$.

Conditions (a) and (b) ensure no crossovers and condition (c) simplifies some centroid computations that follow. These stipulations do not have to be checked inside the constructor. Just make sure to write clear comments so that users of this constructor "play by the rules."

Centroids

The centroid of a polygon is simply its center of mass. (It is where we would attach a thread so as to make a horizontally floating "polygon mobile.") For triangles there is a simple formula for the centroid. If the vertices are (a_0,b_0), (a_1,b_1), and (a_2,b_2), then the centroid is ((a_0+a_1+a_2)/3,(b_0,b_1+b_2)/3).

For polygons of the type that either constructor builds we can compute the centroid by partitioning the *relative* polygon into n-triangles defined by (0,0) and the n relative vertices:



(Condition (c) above ensures that (0,0) is inside the relative polygon.) We have displayed the centroids of the triangles because they can be used to compute the centroid of the polygon. The centroid (x_c,y_c) of the relative polygon is given by

$$x_c = (x_0A_0 + \ldots + x_{n-1}A_{n-1})/A \qquad y_c = (y_0A_0 + \ldots + y_{n-1}A_{n-1})/A$$

where A_k is the area of the k-th triangle, (x_k,y_k) is the centroid of the k-th triangle, and $A = A_0 + \ldots + A_{n-1}$. Add an instance method

```
public point centroid()
```

that returns the centroid of the (absolute) polygon. Note that this requires adding the x and y coordinates of the reference point to x_c and y_c respectively. Methods for computing triangle areas and triangle centroids can be found in the given `point` class.

Normalization

Add an instance method

```
public void normalize()
```

to the class `polygon`. It should translate this polygon so that its centroid is at (0,0).

Intersection

Add an instance method

`public boolean intersect(polygon P)`

to the class polygon. It should yield true if the boundary of this polygon intersects the boundary of `P`. Make effective use of the method `intersect` that is available through the given class `point.`

What to Submit

The file `P7A.java` has code that graphically displays various polygons and centroids as they are produced by your `polygon` methods. You can use this code to debug merely by commenting out those lines that are irrelevant to your current debugging. Resize the window so that it is large enough to show everything. Submit a copy of the graphic produced by `P7A.java` and a listing of your finished `polygon` class. For grading purposes it will be important that we be able to discern the various polygons and centroids in the display. If you are using a black-and-white printer, then you are allowed to modify the color choices in `P7A.java` accordingly. Other than that, leave `P7A.java` alone! And you are not allowed to modify `point` and `closedPolyline` classes. Style points will be deducted if you do not make effective use of the methods in those two classes.

Description of the `point` Class

```
// An instance of this class is a point in the xy-plane.
public class point
{
   private double x;   // The x-coordinate.
   private double y;   // The y-coordinate.

   // The Constructors:

   public point(double xVal, double yVal)

   public point(point P)

   // Moves the point a units in the x-direction and b units in the y-direction.
   public void translate(double a, double b)

   // Yields the x-coordinate of this point.
   public double get_x()

   // Yields the y-coordinate of this point.
   public double get_y()

   // Yields the distance from P0 to P1.
   public static double dist(point P0, point P1)

   // Yields the centroid of the triangle defined by P0, P1, and P2.
   public static point triangleCentroid(point P0, point P1, point P2)

   // Yields the area of the triangle defined by P0, P1, and P2.
   public static double triangleArea(point P0, point P1, point P2)

   // Yields true if the line segment that connects P0 and P1 intersects the line segment
   // that connects Q0 and Q1. Assume P0 and P1 are distinct and Q0 and Q1 are distinct.
   public static boolean intersect(point P0, point P1, point Q0, point Q1)

   // Yields a string representation of this point.
   public String pointString()

}
```

Note. Part B of this assignment will be distributed next week.