Learning Plane Geometry

Alvaro Briz Redon, Angel Serrano Aroca

2018-01-05

Introduction

Several authors defend that learning to program provides powerful strategies for thinking, designing and solving problems. They talk about a two-phase process: first, the solution of the problem must be found, but then this solution has to be rewritten in an alternative and precise way in a language that can be understood by the computer.

In addition, programming provides certain opportunities to the user that few mental activities are in a position to do. According to Seymour Papert (1928-2016), programming not only allows you to communicate with a computer, but also to explore the way you think, to improve your logical reasoning and to increase your capacity to correct your own mistakes.

There are multiple studies that confirm the benefits of programming on young students. For example, in the early 1970s, it was concluded in a survey conducted by Wallace Feurzeig (1927-2013) that the use of LOGO programming language could enhance the reading ability of some students, as well as increasing the interest in learning and the level of self-confidence in many of them. Moreover, beyond all these aspects, learning through the use of a programming language is usually more interesting and exciting for students. For this reason, programming languages have been used as a tool to study Mathematics during the last decades, with overall successful results.

Of course, there are some difficulties when teaching young students to program, but there are numerous strategies to reduce them. Code simplification, problem subdivision or the creation by the tutors of structures more simple than the ones existing in the programming language are some of them.

The package *LearnGeom* has been created with the hope of becoming useful to young students (and their teachers) to learn (and teach) plane geometry while programming in R.

How to start? Basic functions of the package

The package provides several functions to work plane geometry, so it is expected that the user works on a coordinate plane in order to manipulate different geometric objects and constructions. The function *CoordinatePlane* allows the user to plot an empty coordinate plane with customizable limits for the X and Y axis. Once a coordinate plane is started, it is time to create and plot on it different geometric objects.

The basic geometric objects that can be used in this version of the package are five: points, segments, arcs, lines and polygons. All of them can be plotted in the coordinate plane with function *Draw* of the package, and the appropriate use of the possibilities R offers could help to learn different mathematical concepts or solve many kind of geometric problems.

Each of this five basic objects can be defined in a concrete way, but most of them also admit different definitions. In the next lines it is mentioned the way these objects can be built, and what are the functions of the package that allow their creation.

Point: A point can be created by simply defining a two-dimensional vector in R, e.g, c(0,0) for the usual origin of coordinates. It should be remarked that this definition is also the one used for the creation of a geometric vector to determine a direction in the plane. The coincidence of both definitions also exists if one follows classical mathematical notations, so it shouldn't be a source of confusion for the user.

Segment: There are two possibilities to define a segment in the plane. The first one, and the most basic one, consists in the choice of two points of the plane, P and Q. The shortest path (in euclidean distance)

to connect these two points is, by definition, the segment that joins P with Q. There is another common method to define a segment in the plane: from an starting point, choosing an angle and a length for the segment. Both possibilities can be achieved with functions *CreateSegmentPoints* and *CreateSegmentAngle* of the package.

Arc: An arc is simply a part of a circumference, or even the circumference itself. The function *CreateArcAngle* allows the user to make an arc from a circumference with three parameters to choose: the center of the circumference, the radius of the circumference and the two angles, from 0 to 360 degrees that determine the part of the circumference to be plotted.

There is another possibility to create an arc in the plane: by connecting two points. The function *CreateArcPointsDist* makes easy to create an arc that joins two points in the plane. Of course, there are many (infinite) arcs that pass through every two points in a plane. The parameter *radius* fixes a radius for the arc to be built. Depending on the distance between the points, some values for this parameter will produce no arc.

Line: A line, as a segment, can be defined in two ways: from two points, or from a point and an angle. Moreover, there is an standard combination of parameters to characterize every line in the plane: the slope and the intercept. For this reason, the functions *CreateLinePoints* and *CreateLineAngle* return a two-dimensional vector that contains the slope and intercept of the line, regardless of the way it is defined.

The use of the pair slope-intercept has a problem with a particular type of line: vertical lines, which are parallel to Y axis. In the case the user defines a line of this kind, with any of the two available functions, the returning object will be a string two-dimensional vector. It will include the word *"Inf"* for the first position (infinite slope) and the constant X-value for the line in the second (as a character).

Polygon: A polygon is a closed figure made of a finite number of points (there must be 3 points at least). The function *CreatePolygon* admits any finite number of points to produce a polygon. It is important to introduce the points in a certain way to get the desired output, as the same combination of points can lead to different figures. In order to make a polygon without self-intersections, the points must be passed to the function following a clockwise (or counterclockwise) direction. If the user is interested in building a regular polygon, function *CreateRegularPolygon* does all the job.

It is quite simple to create these objects with basic R functions, which belong to the *graphics* package, but the intention was to create an homogeneous group of functions with a minimal number of parameters. Although the pre-existing functions are simple to use for a programmer, they may contain too many parameters for a novice user. Moreover, it is vital to define the functions with a little number of parameters in order to highlight the different existing methods to define the same geometric object.

The following code contains examples of use of some of the functions included in the package.

```
library(LearnGeom)
x_min <- -5
x_max <- 5
y_min <- -5
y_max <- 5
CoordinatePlane(x_min, x_max, y_min, y_max)
```

```
## NULL
```

```
P1 <- c(0,0)
P2 <- c(1,1)
P3 <- c(2,0)
Poly <- CreatePolygon(P1, P2, P3)
```

```
## [1] "Some of the inserted points are collinear. This could lead to a defective polygon."
```

```
Draw(Poly, "blue")
```

```
## NULL
```

```
Hepta <- CreateRegularPolygon(7, c(-3,0), 1)
Draw(Hepta, "orange")
```

```
## NULL
```

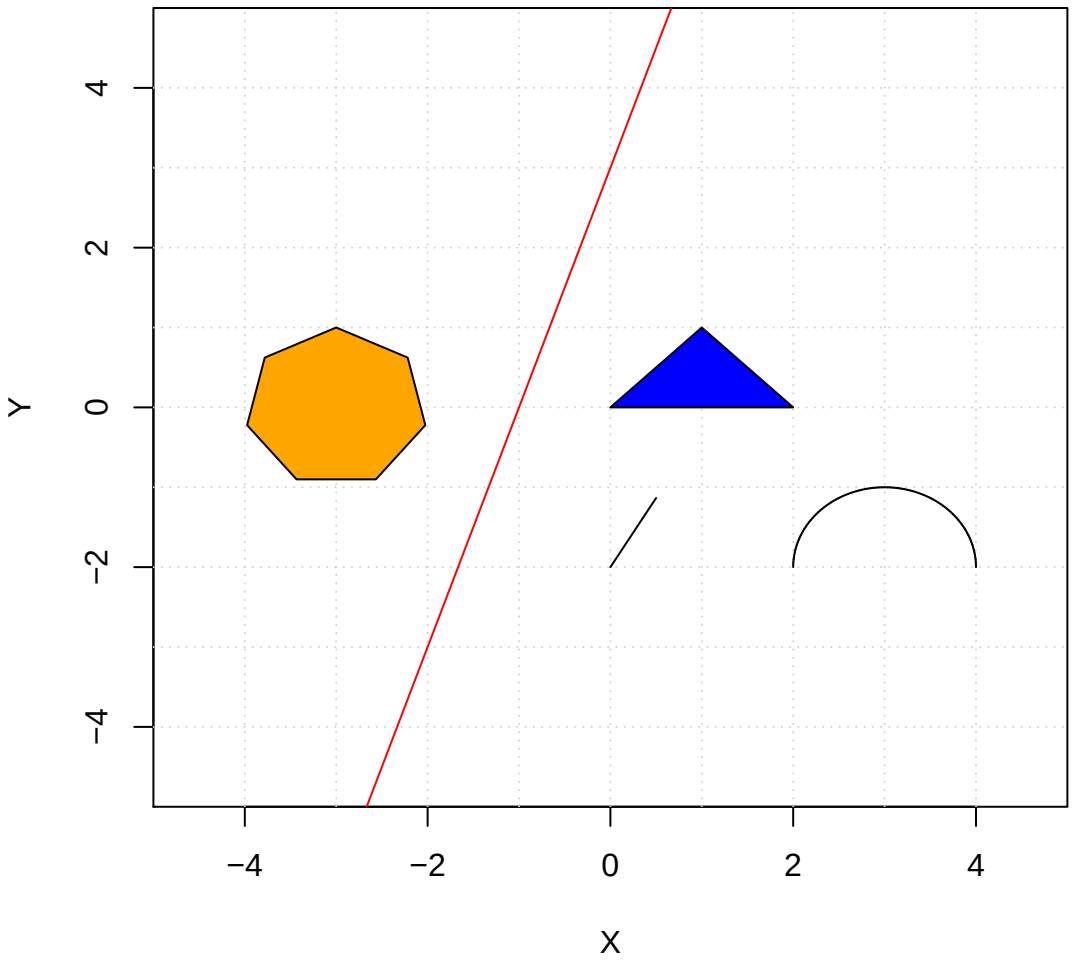
```
L <- CreateLinePoints(c(-1,0), c(0,3))
Draw(L, "red")
```

```
## NULL
```

```
S <- CreateSegmentAngle(c(0,-2), 60, 1)
Draw(S, "black")
```

```
## NULL
```

```
A <- CreateArcAngles(c(3,-2), 1, 0, 180, "anticlock")
Draw(A, "black")
```




```
## NULL
```

Affine transformations

Affine functions are geometric transformations that preserve collinearity and ratios of distances. LearnGeom contains functions to apply six different affine transformations: **homothety**, **reflection**, **rotation**, **shear**, **similarity** and **translation**. Each of these transformations is associated with a 2 x 2 matrix, depending on one or several specific parameters.

For example, *Rotate* function can be used to rotate a line, a polygon or a segment. The following code can be used to rotate the triangle (in blue) that connects the points $P1 = (0,0)$, $P2 = (1,1)$ and $P3 = (2,0)$. There are shown two options for the *fixed* parameter: (-1,-1) and (2,0). These are the points that are left fixed by the rotation in each of the executions, producing different triangles in the plane.

```
x_min <- -5
x_max <- 5
y_min <- -5
y_max <- 5
CoordinatePlane(x_min, x_max, y_min, y_max)
```

```
## NULL
```

```
P1 <- c(0,0)
P2 <- c(1,1)
P3 <- c(2,0)
Poly <- CreatePolygon(P1, P2, P3)
```

```
## [1] "Some of the inserted points are collinear. This could lead to a defective polygon."
```

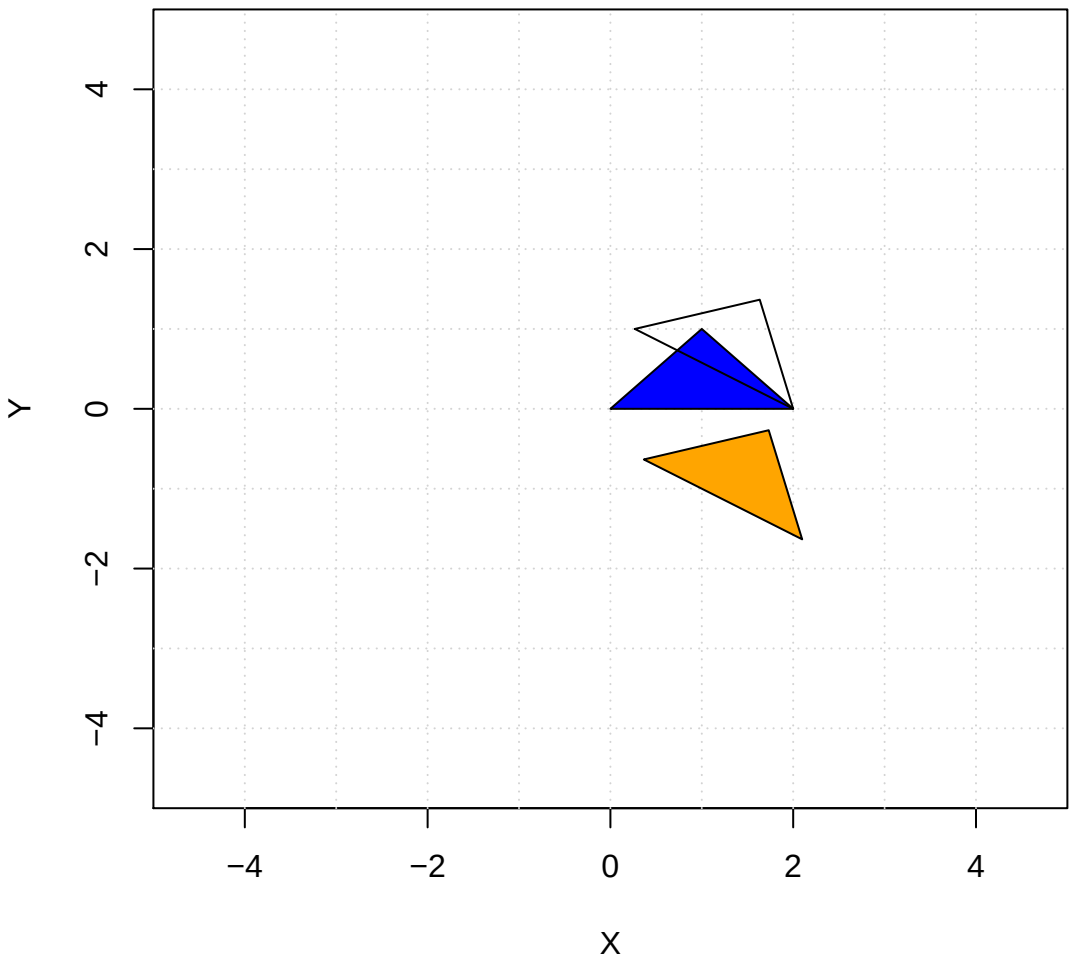
```
Draw(Poly, "blue")
```

```
## NULL
```

```
fixed <- c(-1,-1)
angle <- 30
Poly_rotated <- Rotate(Poly, fixed, angle)
Draw(Poly_rotated, "orange")
```

```
## NULL
```

```
fixed <- c(2,0)
Poly_rotated <- Rotate(Poly, fixed, angle)
Draw(Poly_rotated, "transparent")
```




```
## NULL
```

Points of the triangle

As an example of combined use of the package functions and the R language capabilities, let's see how we can obtain some notable points of a triangle as the circumcenter or the incenter. The package also includes functions *Circumcenter* and *Incenter* to find these points with only one instruction.

For example, the following blocks of code give as a result the circumcenter of a triangle, that is, the intersection of the three perpendicular bisectors from each of the sides. First of all, it must be created the triangle.

```
P1 <- c(0,0)
P2 <- c(1,1)
P3 <- c(2,0)
Tri <- CreatePolygon(P1, P2, P3)
```

```
## [1] "Some of the inserted points are collinear. This could lead to a defective polygon."
```

Then, the vectors that connect each pair of points are computed, and also their associated orthogonal vectors.

```
A <- Tri[1,]
B <- Tri[2,]
C <- Tri[3,]
AB <- B-A
AB_orto=c(-AB[2],AB[1])
AC <- C-A
AC_orto=c(-AC[2],AC[1])
BC <- C-B
BC_orto=c(-BC[2],BC[1])
```

The middle point of each of the sides of the triangle is obtained with function *MidPoint*.

```
AB_mid <- MidPoint(A,B)
AC_mid <- MidPoint(A,C)
BC_mid <- MidPoint(B,C)
```

To create the bisector lines three auxiliar points are computed.

```
AB1 <- AB_mid+AB_orto
AC1 <- AC_mid+AC_orto
BC1 <- BC_mid+BC_orto
```

This leads to the creation of the three bisector lines, which intersect in a point which is called the circumcenter.

```
L1 <- CreateLinePoints(AB_mid,AB1)
L2 <- CreateLinePoints(AC_mid,AC1)
L3 <- CreateLinePoints(BC_mid,BC1)
I <- IntersectLines(L1,L2)
```

This final block of code draws all the important geometric objects involved in the obtention of the circumcenter (and the circumcenter itself).

```
x_min <- -6
x_max <- 6
y_min <- -6
y_max <- 6
CoordinatePlane(x_min, x_max, y_min, y_max)
```

```
## NULL
```

```
P1 <- c(0,0)
P2 <- c(1,1)
P3 <- c(2,0)
Tri <- CreatePolygon(P1, P2, P3)
```

```
## [1] "Some of the inserted points are collinear. This could lead to a defective polygon."
```

```
Draw(Tri, "blue")
```

```
## NULL
```

```
Draw(L1, "red")
```

```
## NULL
```

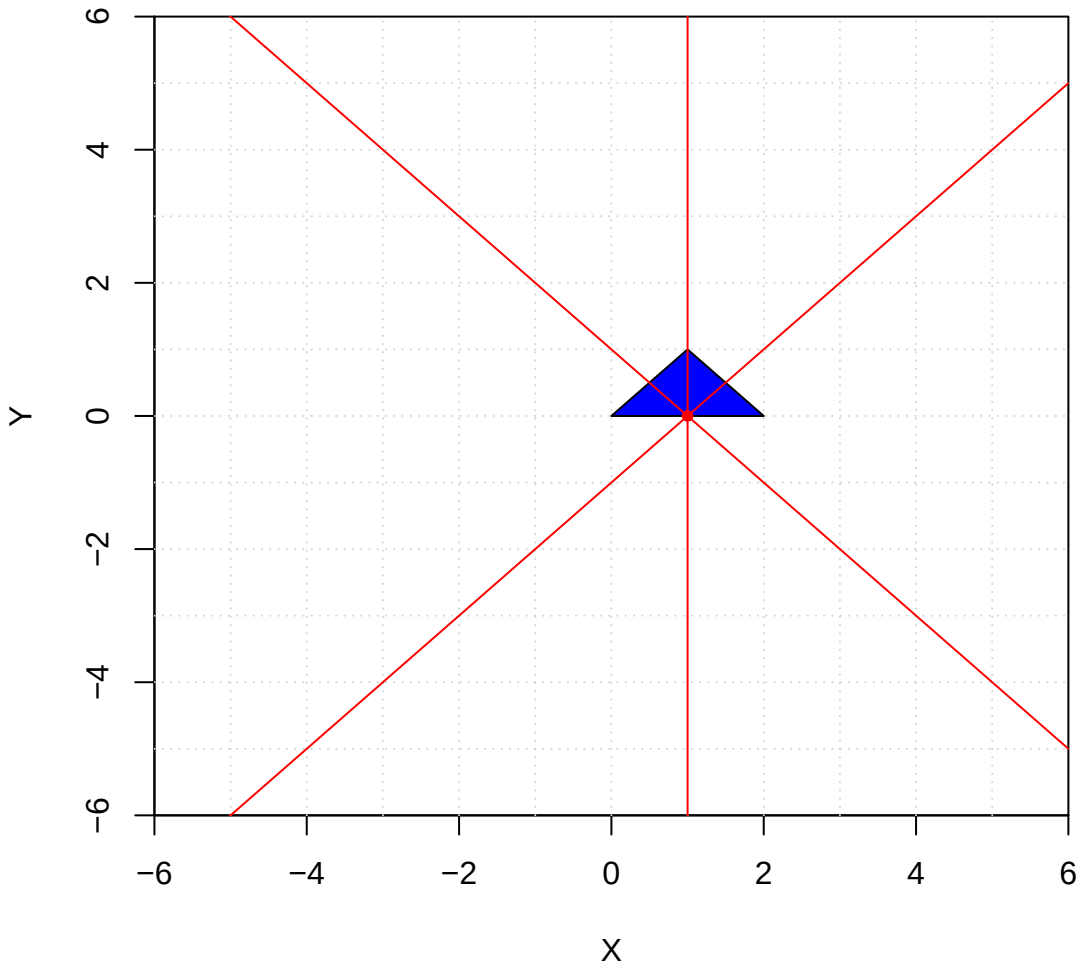
```
Draw(L2, "red")
```

```
## NULL
```

```
Draw(L3, "red")
```

```
## NULL
```

```
Draw(I, "red")
```




```
## NULL
```

As it can be observed in the code, several functions of the package make it easy to obtain some of the partial results, but it is also necessary to create several vectors and to know about orthogonality and operations between vectors and points to get the correct point at the end.

Tessellations

A tessellation, also known as a tiling, is a pattern that is made by repeating a basic geometric figure, or combination of figures, along a plane. Tessellations appear sometimes in nature and are often present in architectural constructions.

For example, beehives, the perfect structures created by bees to produce their honey, can be considered as a tessellation made of regular hexagons. With the help of the function *Tessellation* is quite simple to generate a pattern that reminds a beehive, like the one in the next figure. The following code could be an starting point. The creation of a loop would allow the extension of the pattern to a wider region of the plane.

```
## Creation of the starting hexagon, Hexa0
x_min <- -5; x_max <- 5; y_min <- -5; y_max <- 5
CoordinatePlane(x_min, x_max, y_min, y_max)
```

```
## NULL
```

```
Hexa0 <- CreateRegularPolygon(6, c(-3,0), 1)
C=CenterPolygon(Hexa0)
Draw(Hexa0, "gold", label = T)
```

```
## NULL
```

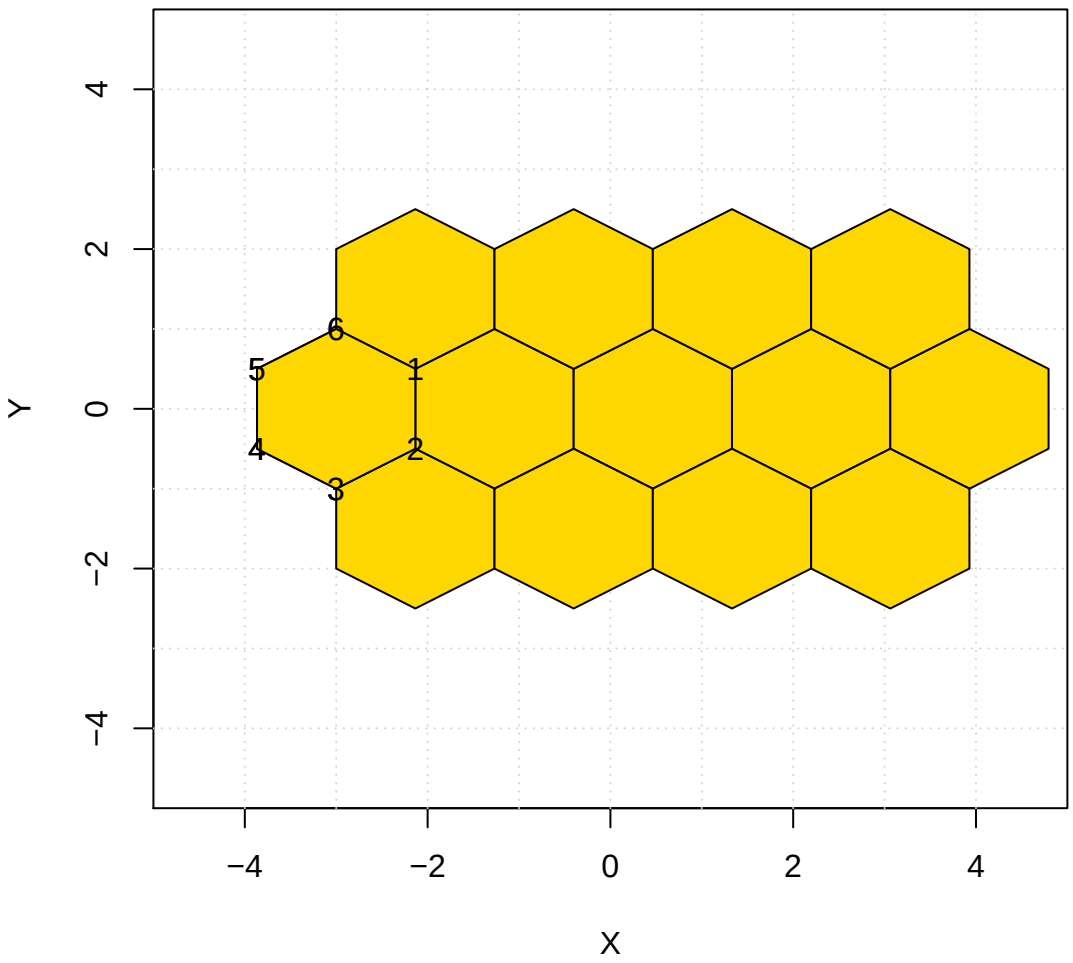
```
## Computation of the distance from the right extreme of the hexagon
## to its central axis
d <- DistancePoints(Hexa0[1,], c(-3,Hexa0[1,2]))
separation <- 2*d
Tessellation(list(Hexa0), "gold", c(1,0), separation, 4)
mid1 <- MidPoint(Hexa0[1,],Hexa0[6,])
mid2 <- MidPoint(Hexa0[2,],Hexa0[3,])
v1 <- mid1-CenterPolygon(Hexa0)
v2 <- mid2-CenterPolygon(Hexa0)
Hexa1 <- Translate(Hexa0,2*v1)
Hexa2 <- Translate(Hexa0,2*v2)
Draw(Hexa1,"gold")
```

```
## NULL
```

```
Tessellation(list(Hexa1), "gold", c(1,0), separation, 3)
Draw(Hexa2,"gold")
```

```
## NULL
```

```
Tessellation(list(Hexa2), "gold", c(1,0), separation, 3)
## Drawing again the starting polygon to visualize the labels at its
## vertex
Draw(Hexa0, "gold", label = T)
```




```
## NULL
```

To achieve the desired pattern it is very important to set the *separation* parameter of the function correctly. Otherwise, the resulting pattern could contain some overlaps, which are uncommon in the creation of tessellations.

However, one could minimize the code to get the hexagons *Hexa1* and *Hexa2* by using the function *ReflectedPolygon*. It is easy to notice that these two hexagons must be symmetric to *Hexa0* about the two lines that connect the points 1 and 6, and 2 and 3. This would be the code in the case of taking advantage of this circumstance.

```
L1 <- CreateLinePoints(Hexa1[1,],Hexa1[6,])
L2 <- CreateLinePoints(Hexa1[2,],Hexa1[3,])
Hexa1 <- ReflectedPolygon(Hexa1,L1)
Hexa2 <- ReflectedPolygon(Hexa1,L2)
```

The two options to create *Hexa1* and *Hexa2* seem enough to show that the use of this learning approach could be beneficial to improve, simultaneously, geometric thinking and programming skills.

Recursive programming

Recursive programming is one of the most efficient strategies to find the solution of some problems. However, it is also a difficult task for novice programmers. For quite advanced students, recursive programming could be treated to build a well-known mathematical structure: a fractal, a concept introduced by Benoit Mandelbrot (1924-2010). Fractals are geometric objects which satisfy the property of self-similarity, which basically means that each of their parts satisfy the same properties and internal relationships as the complete object does.

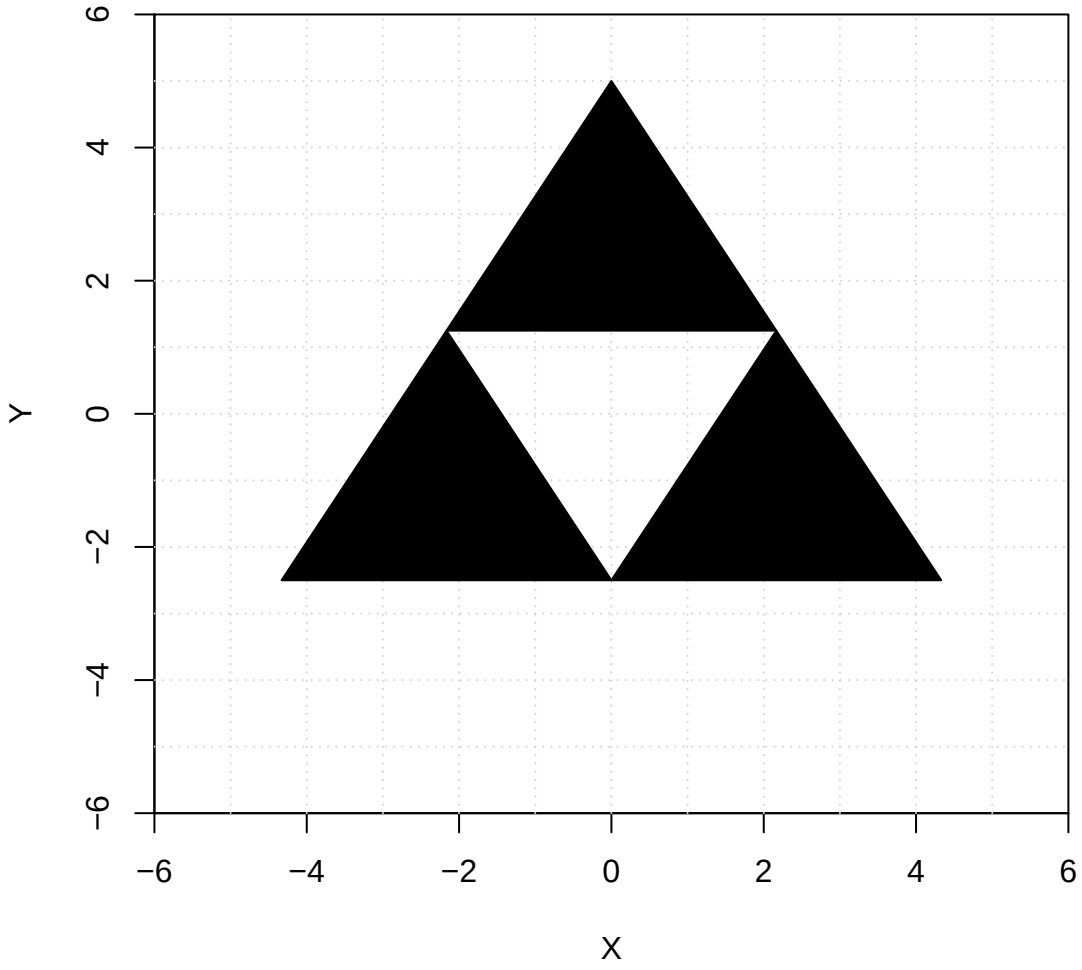
As an illustration, this definition can be easily intuited with the help of one of the most famous fractals: the Sierpinski triangle.

The following code uses function *Sierpinski* of the package to build the first iteration of the triangle and the three first iterations. It can be appreciated how the second figure contains nine little copies of the object represented at the first iteration.

```
x_min <- -6
x_max <- 6
y_min <- -6
y_max <- 6
CoordinatePlane(x_min, x_max, y_min, y_max)
```

```
## NULL
```

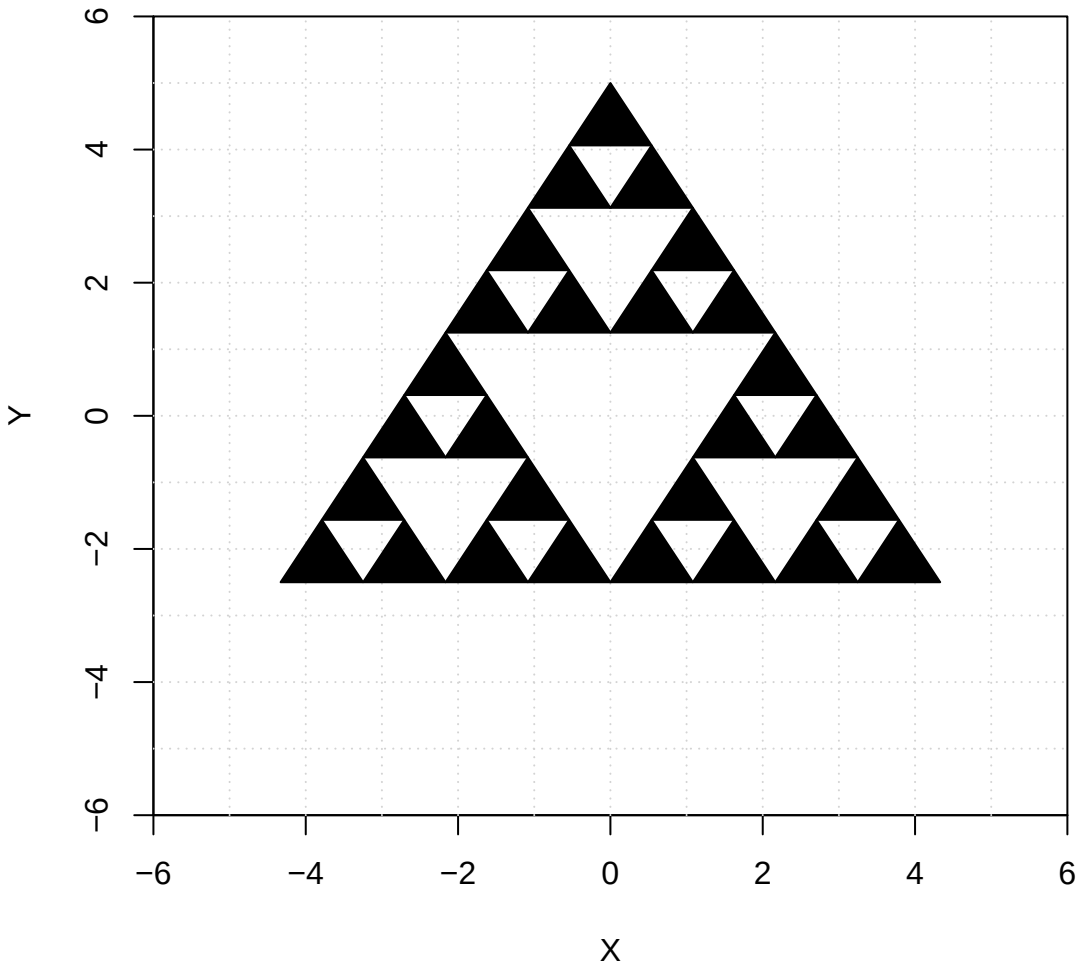
```
n <- 3; C <- c(0,0); l <- 5
Tri <- CreateRegularPolygon(n, C, l)
it <- 1
Sierpinski(Tri, it)
```




```
CoordinatePlane(x_min, x_max, y_min, y_max)
```

```
## NULL
```

```
it <- 3
Sierpinski(Tri, it)
```



Of course, one can attempt to construct the Sierpinski triangle by programming recursively with the aid of some of the function of the package. Here it is simply shown the output that produces function *Sierpinski*.

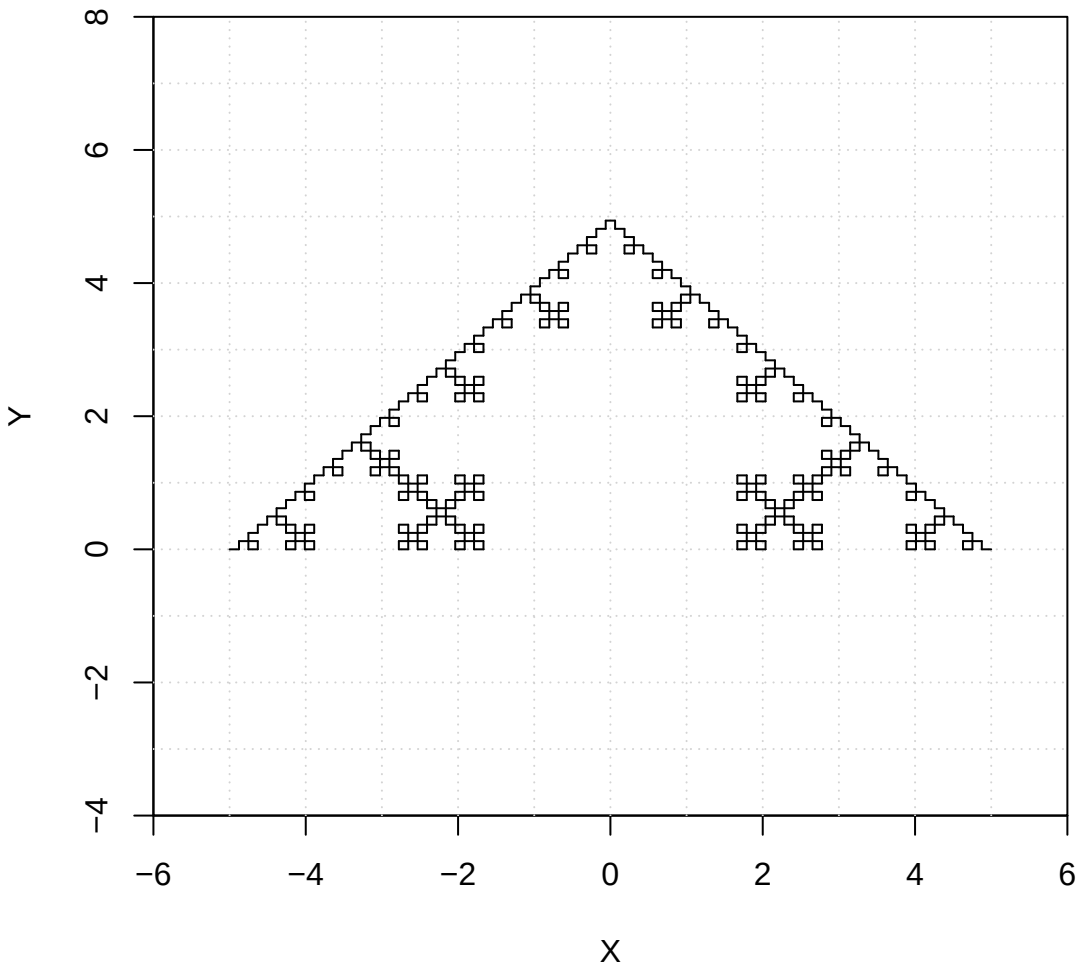
Another very important fractal is the one known as Koch's snowflake. The package contains a function called *FractalSegment* that is able to produce this fractal and much more of the same nature. The following code presents to possibilities of use of this function. There is only one difference in the parameters setting, but the outputs are very different. The curve which is obtained with $angle = 60$ is the aforementioned Koch's snowflake.

```
x_min <- -6
x_max <- 6
y_min <- -4
y_max <- 8
CoordinatePlane(x_min, x_max, y_min, y_max)
```

```
## NULL
```

```
P1 <- c(-5,0)
P2 <- c(5,0)
```

```
angle <- 90
cut1 <- 1/3
cut2 <- 2/3
f <- 1
it <- 4
FractalSegment(P1, P2, angle, cut1, cut2, f, it)
```

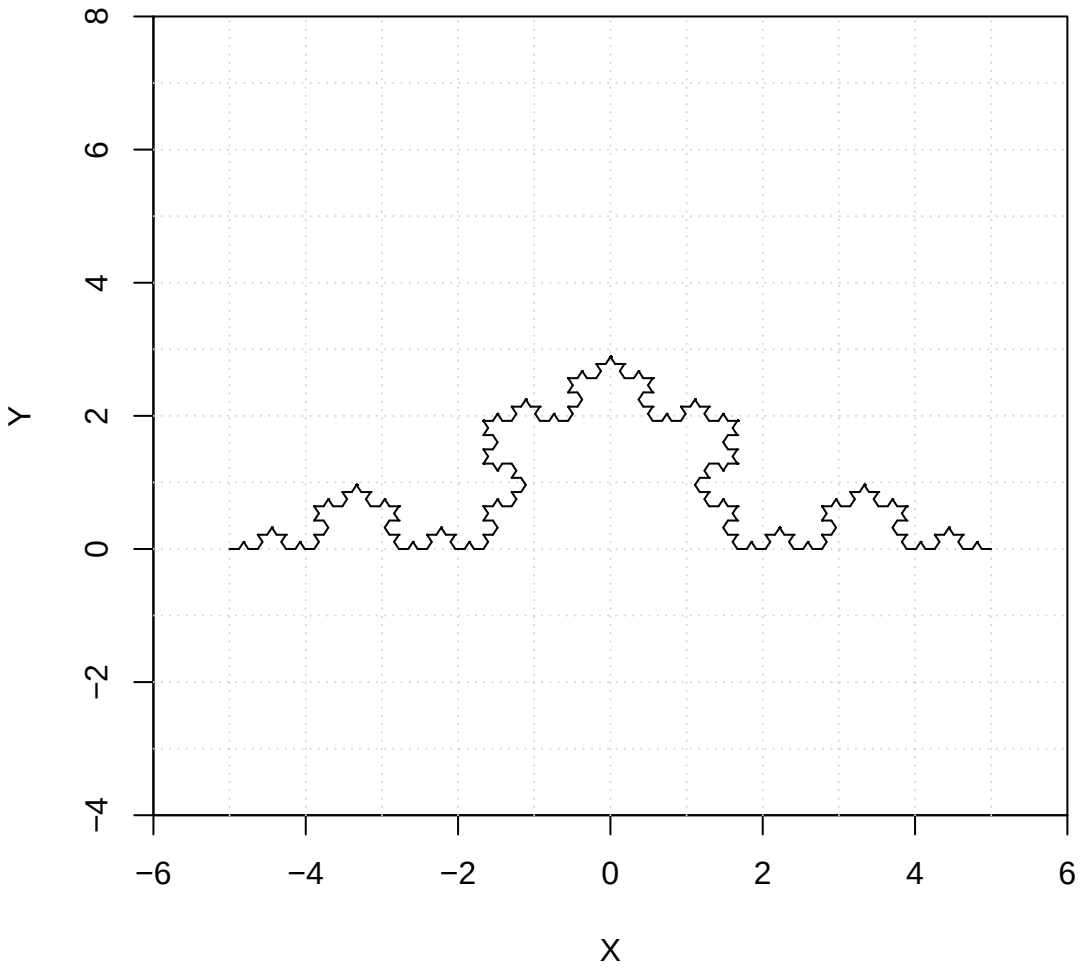



```
x_min <- -6
x_max <- 6
y_min <- -4
y_max <- 8
CoordinatePlane(x_min, x_max, y_min, y_max)
```

```
## NULL
```

```
P1 <- c(-5,0)
P2 <- c(5,0)
angle <- 60
```

```
cut1 <- 1/3
cut2 <- 2/3
f <- 1
it <- 4
FractalSegment(P1, P2, angle, cut1, cut2, f, it)
```



Generation of trochoids

A trochoid is a closed curve that can be obtained by the conjunction of three geometric figures: two circles, one fixed and the other moving, and a moving point which is connected to the moving circle. There are three parameters to characterize each trochoid: the radius of the fixed circle, the radius of the moving circle and the distance from the moving point to the center of the moving circle.

The presence of these parameters makes possible the definition of every trochoid by a set of parametrical equations, involving trigonometry functions. However, as it was first proposed in *Turtle Geometry The Computer as a Medium for Exploring Mathematics* (Abelson and diSessa) with the aid of the LOGO turtle,

these curves can be approximated by iteratively drawing segments of certain lengths and angles. The LOGO turtle, which originally was a real robot developed at MIT at the end of the 60s, refers to a on-screen cursor implemented in the LOGO language that responded to easy instructions from the user (basically direction setting of the turtle and rectilinear movements). This turtle is also available in R with package *TurtleGraphics*.

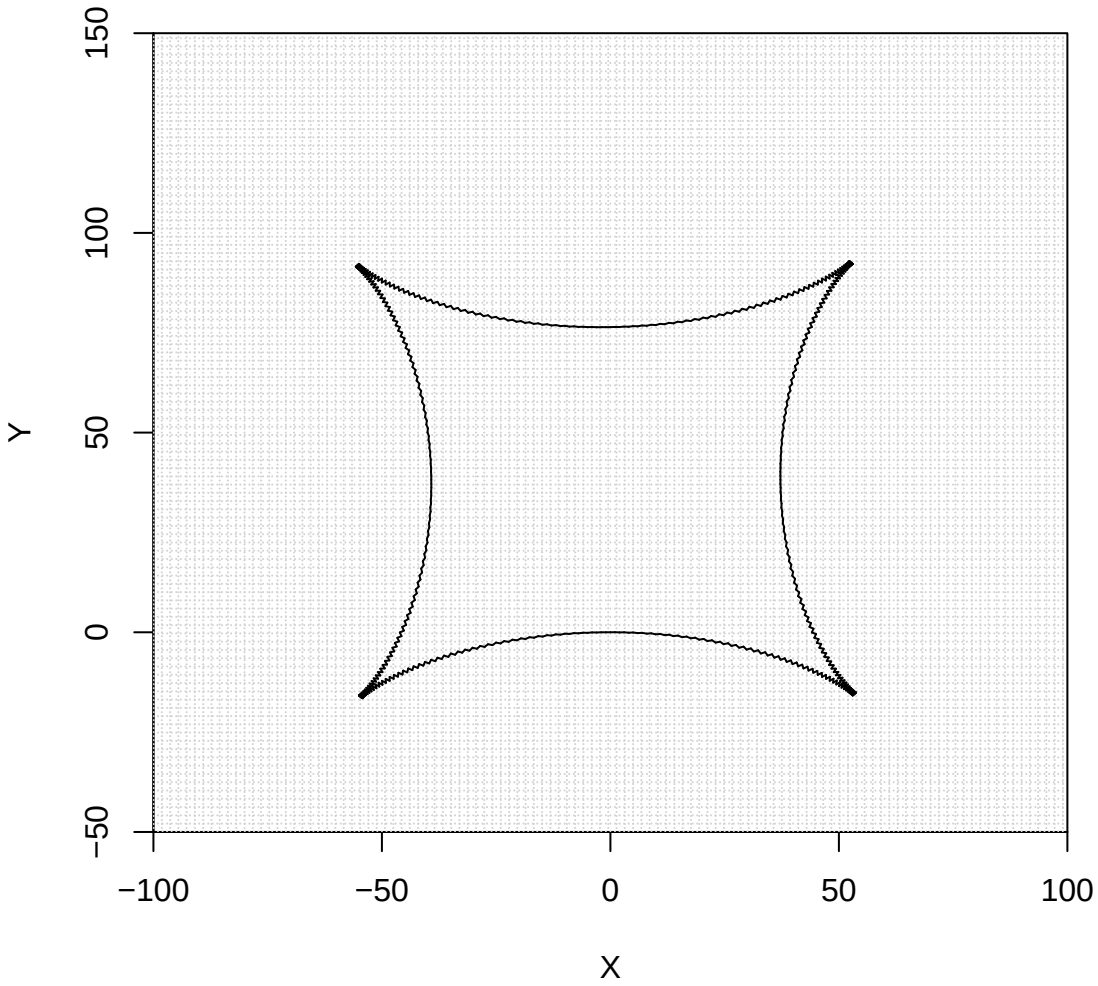
As it can be seen in the work of Abelson and DiSessa, the procedure to obtain any trochoid needs four parameters: a pair of angles and a pair of lengths. In each iteration of the process, a couple of segments are drawn according to these angles and lenghts.

The following code includes three examples of use of the function *Duopoly* implemented in the package. Parameters *color* and *time* also allow the user to appreciate the points that are drawn in the generation of the trochoid and the order of them when building it.

```
P=c(0,0)
CoordinatePlane(-100,100,-50,150)
```

```
## NULL
```

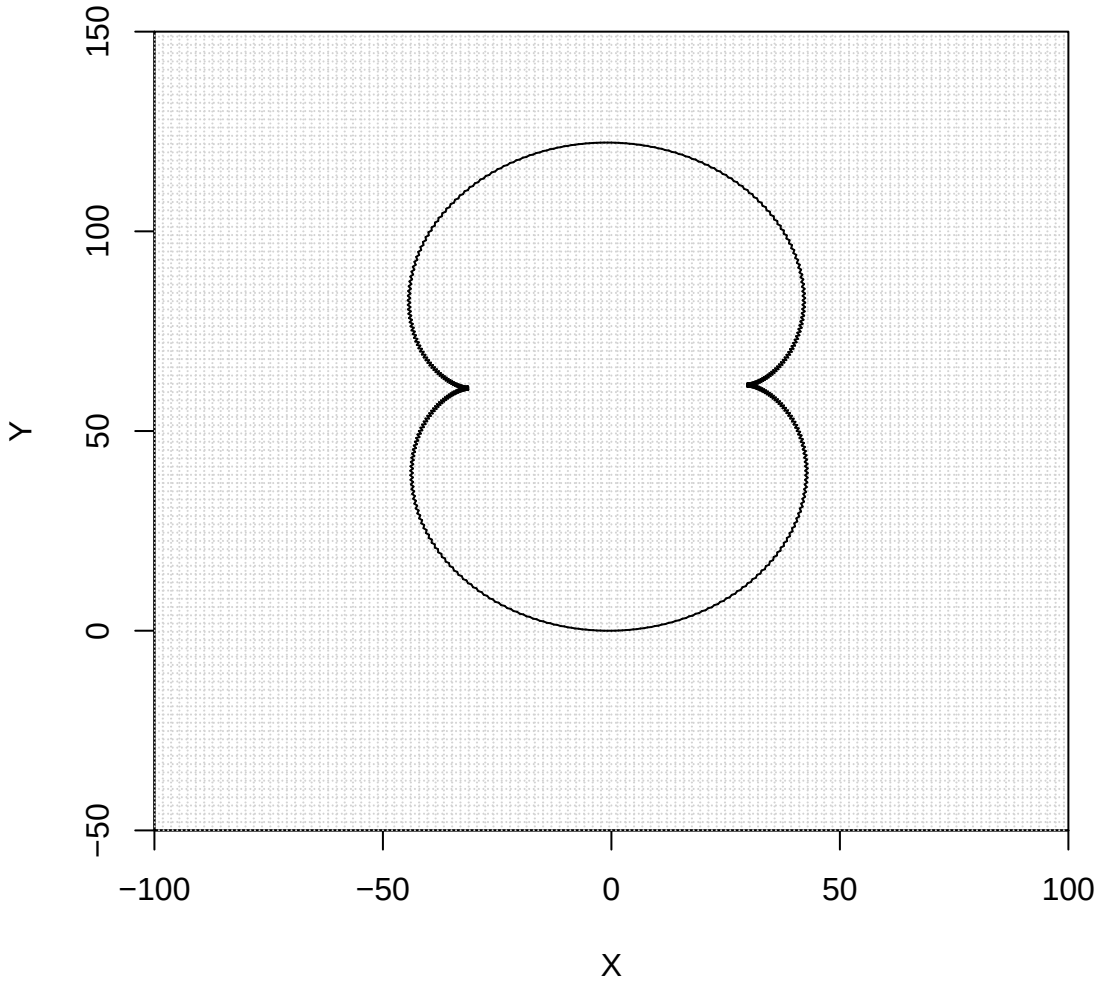
```
Duopoly(P,1,1,1,-3)
```



```
CoordinatePlane(-100,100,-50,150)
```

```
## NULL
```

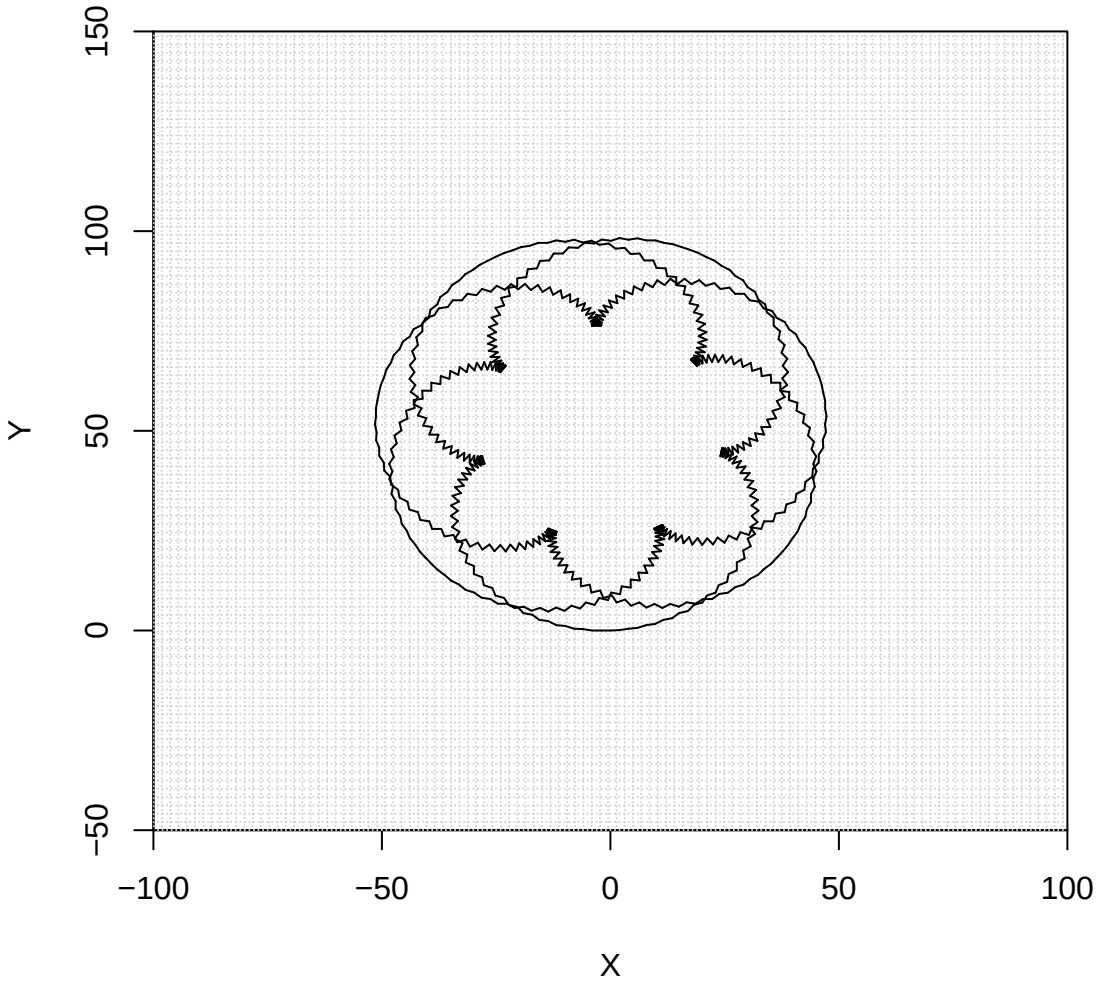
```
Duopoly(P,0.8,1,0.8,3)
```




```
CoordinatePlane(-100,100,-50,150)
```

```
## NULL
```

```
Duopoly(P,2,3,2,10)
```



Working on the real world

Function *GetMap* of the package *RgoogleMaps* gives us the possibility of treating geometric objects over a plane that represents a piece of the world we live in. The *zoom* parameter ranges from 0 to 21, depending on the location, allowing the user to visualize big pictures of the world but also little details of some buildings.

The function *CoordinateImage* of this package works as *CoordinatePlane*, but in this case it is capable of setting the axis and the grid over an image obtained from Google Maps. Moreover, *CoordinateImage* is not only expected to be used with Google Maps images, as it can work with any bitmap image that is passed to the function.