Drawing Polygons in R: A Tutorial Using the polygon() Function

Authored by

Mohammed looti

November 13, 2025

When generating sophisticated R visualizations, analysts frequently need to overlay custom geometric shapes directly onto an existing plot. These shapes are fundamental for communicating complex ideas, whether they are used to highlight specific regions of interest, delineate confidence intervals, or accurately map geographical boundaries. The process of defining and rendering these multi-sided forms, known formally as polygons, is critical for producing professional and highly informative data visualizations. To achieve this, we rely on a precise method for defining the shape based on the exact coordinates of its corners, or **vertices**.

The core utility within R's base graphics package tailored for this task is the **polygon()** function. This function efficiently processes a defined sequence of X and Y coordinates, connecting them sequentially to automatically close the loop and form a solid, filled shape. Mastering the coordinate sequence is essential; the order in which the **vertices** are provided dictates the resulting geometry, ensuring the polygon is drawn correctly without crossing lines or producing unexpected visual artifacts.

Defining Polygons: The Syntax of polygon()

The **polygon()** function serves as a foundational component of R's graphical toolkit, specifically designed to add custom geometric elements to an already initiated plot area. Its strength lies in its simplicity, requiring only the basic geometric inputs to establish the boundary of the desired shape. However, to create visualizations suitable for publication or detailed analysis, we must effectively utilize the optional parameters that control the aesthetic properties, such as fill color, border color, and line thickness.

The fundamental syntax for the polygon() function is straightforward, relying on named arguments to clearly define both the shape's geometry and its appearance. The use of ellipses (`...`) signifies that various additional graphical parameters can be passed to the function, enabling fine-grained control over aspects like line type (lty) or line thickness (lwd).

polygon(x, y, col=NA, border=NULL, ...)

The two required arguments, **x** and **y**, must be supplied as numerical vectors. These vectors contain the corresponding coordinates that define the shape's **vertices**. The remaining arguments are optional but provide vital customization capabilities:

x: This vector lists the X-coordinates for the sequence of **vertices** that define the boundary. It is mandatory that these points are listed in consecutive order around the perimeter.
y: This vector contains the Y-coordinates, which must be paired sequentially with the X-coordinates to complete the definition of each vertex pair.
col: Specifies the fill color for the interior of the polygon. By default, this is set to **NA**, rendering the interior transparent and leaving only the border visible.

border: Defines the color of the outline or boundary line surrounding the polygon. If this parameter is set to **NULL** (the default behavior), the border will be rendered using the current foreground color established by the existing plotting environment.

The following practical examples illustrate how to successfully leverage the **polygon()** function to draw standard geometric shapes, demonstrating the necessary steps to define the coordinate system and render the resulting figure within R's graphical device.

Practical Application 1: Drawing a Basic Geometric Shape

Before any custom shape can be drawn using **polygon()**, a suitable coordinate space must first be established. This is typically achieved using the **plot()** function, which defines the visual boundaries and scaling within which the polygon will reside. Since **polygon()** is an additive function--designed to overlay graphics onto an existing structure--it will fail if no base plot is present. For our initial demonstration, we will construct a simple square centered near the coordinates (10, 10), which requires specifying four distinct **vertices**.

We will construct the square using the following set of Cartesian coordinates. It is crucial that these points are listed in sequential order, tracing the perimeter either clockwise or counter-clockwise. R connects the points exactly as they are supplied, and any deviation from sequential order will result in a complex, possibly self-intersecting, shape:

(8, 8) (Lower-left starting point)
(8, 10) (Upper-left vertex)
(10, 10) (Upper-right vertex)
(10, 8) (Lower-right vertex, which connects back to (8, 8) to close the loop)

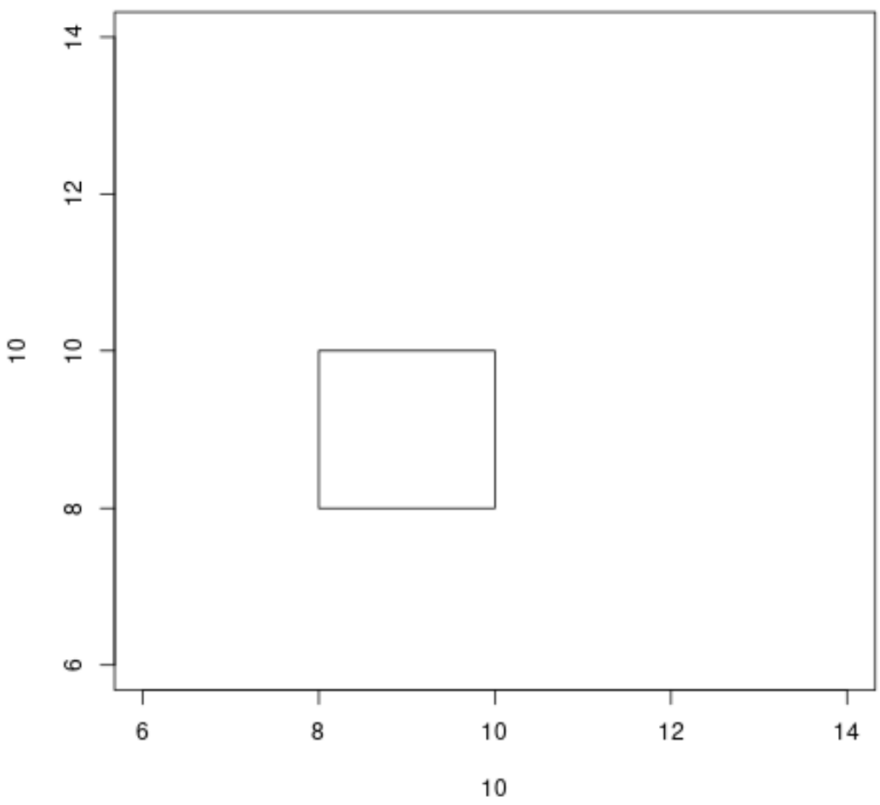
To implement this, we pass the X-coordinates (8, 8, 10, 10) and the corresponding Y-coordinates (8, 10, 10, 8) as separate vectors to the function. The initial **plot()** command simply sets up a blank, white-background plot spanning the range necessary to contain our square.

```
#create plot centered at (x, y) coordinates of (10, 10)
plot(10, 10, col='white')

#draw a polygon with specific vertices
polygon(x = c(8, 8, 10, 10),
y = c(8, 10, 10, 8))
```

Executing the code above successfully renders the resulting square. Because we omitted the **col** argument, the default rendering uses the system's default line color (usually black) for the border, leaving the interior transparent. This illustrates the minimum required implementation for defining

and drawing a simple polygon based purely on its geometric definition.



Practical Application 2: Adding Fill Color and Clarity

While an outlined shape is perfectly functional for certain tasks, most advanced graphical applications necessitate the ability to fill the shape with a specific color. This is essential for visual clarity, differentiation, and emphasizing the area of interest. As noted previously, the default behavior of the **polygon()** function is to leave the interior unfilled (transparent, or **col=NA**). To introduce a solid fill color, we utilize the **col** argument, which readily accepts standard R color names (such as 'blue', 'red', or 'gray') or precise hexadecimal color codes.

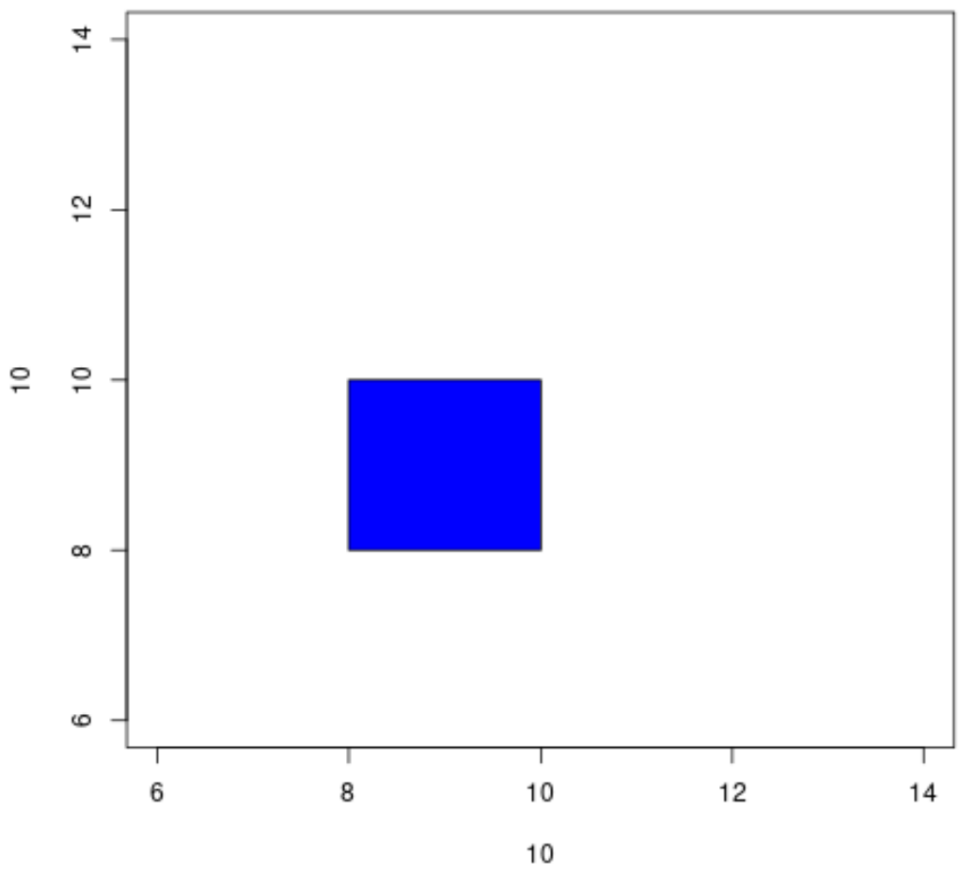
By using the identical coordinate definition from Example 1, we can easily modify the function call to fill the square with a solid blue color. This transformation significantly enhances the visibility of the shape, allowing it to stand out against the background of the plot. This coloring capability is paramount when constructing graphics such as confidence bands, histograms, or density plots, where the filled area conveys crucial information like uncertainty or probability distribution.

The following code snippet demonstrates the straightforward inclusion of the **col** parameter. Notice that the definition of the **x** and **y** coordinates remains constant, ensuring we draw the exact same square, but we are now applying a visual enhancement to its interior.

```
#create plot centered at (x, y) coordinates of (10, 10)
plot(10, 10, col='white')

#draw a polygon with specific vertices, now filled blue
polygon(x = c(8, 8, 10, 10),
y = c(8, 10, 10, 8),
col = 'blue')
```

Upon execution, the visualization shows the identical square shape, now completely saturated with the specified blue color. This simple addition drastically alters the visual weight of the element, establishing it as a solid, distinct feature within the overall graph. This methodology for filling the area can be applied universally to any polygon, regardless of the complexity of its geometry, provided the coordinate vectors are correctly structured and ordered.



Practical Application 3: Customizing Boundary Aesthetics (Border and Thickness)

In addition to controlling the interior fill, it is often necessary to precisely control the appearance of

the boundary line itself. This customization is vital for distinguishing between overlapping shapes or ensuring that the polygon is clearly visible against a busy or complex background. The **polygon()** function addresses this need through two primary parameters: **border** and **lwd** (line width).

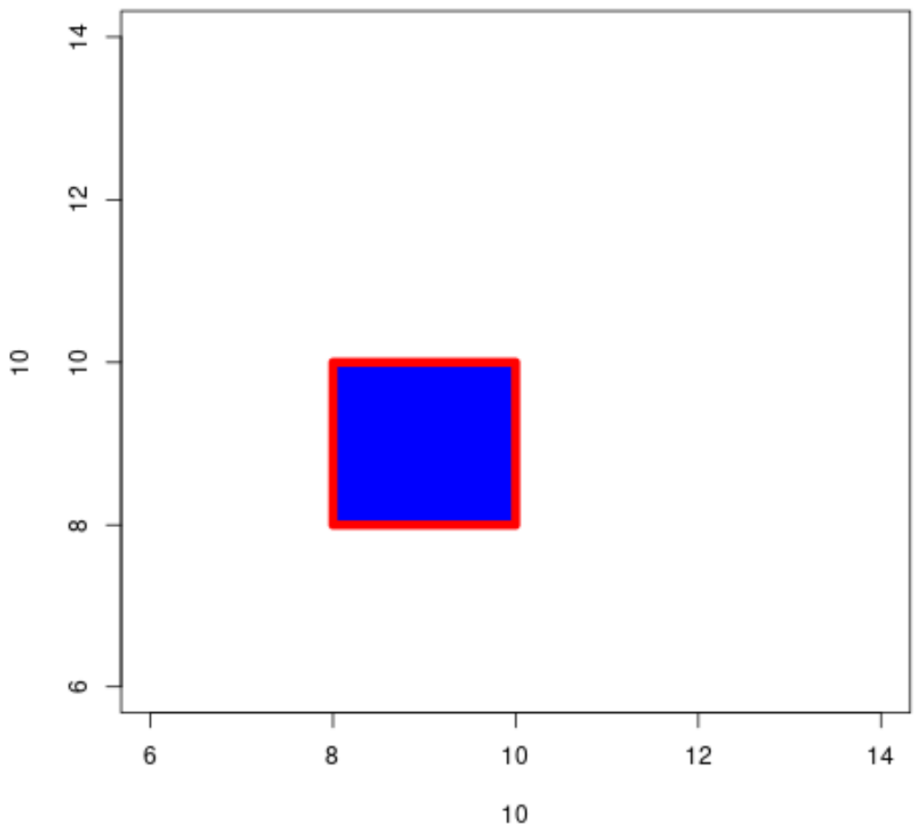
The **border** argument accepts a color specification, functioning identically to the **col** argument, defining the color of the perimeter. Conversely, **lwd** controls the thickness of the line drawn along the perimeter. By strategically utilizing both **col** and **border**, we can create high-contrast shapes. For instance, we can retain the solid blue fill introduced in the previous example while simultaneously adding a highly visible red border. Increasing the **lwd** value, which expects a numerical input (where larger numbers mean thicker lines), further emphasizes the boundary.

In this final example, we reuse the same square coordinates. We define the fill as blue, the border as red, and set the line width significantly higher than the default value (typically 1) by choosing **lwd = 6**. This modification clearly accentuates the boundary of the polygon, showcasing the flexibility and power inherent in R's graphics parameters.

```
#create plot centered at (x, y) coordinates of (10, 10)
plot(10, 10, col='white')

#draw a polygon with specific vertices
polygon(x = c(8, 8, 10, 10),
y = c(8, 10, 10, 8),
col = 'blue',
border = 'red',
lwd = 6)
```

The resulting visualization confirms that the square maintains its blue interior but is now sharply delineated by a thick, bright red border. This level of aesthetic customization allows users to tailor the appearance of polygons precisely to meet specific visualization requirements, whether the goal is to prepare graphics for publication or conduct detailed internal data analysis.



It is important to recognize that **lwd** is a global graphical parameter in R, and its application here specifically modifies the line width of the polygon's border. Analysts are strongly encouraged to experiment with various values for **col**, **border**, and **lwd** to discover the optimal aesthetic combination for any specific dataset or graphical context.

Conclusion and Further Applications

The **polygon()** function is an indispensable component within R's graphical toolkit. It offers a robust and highly flexible method for defining, drawing, and styling custom multi-sided shapes on any existing plot. Its fundamental reliance on ordered coordinate **vectors** for defining **vertices** guarantees geometric precision, while the optional aesthetic arguments--namely **col**, **border**, and **lwd**--provide the necessary control for creating effective and compelling data visualizations. Mastery of this function is a key step toward advancing beyond simple plots and building complex, highly informative graphics.

While our focus here was on a simple four-sided square, the principles demonstrated apply universally. Polygons can be used to represent complex non-convex shapes, accurately define areas under probability curves (such as confidence intervals), or delineate specific regions in spatial data analysis. The primary limitation rests solely on the quality and sequential order of the

input coordinate vectors. Always ensure that the X and Y coordinate lists are of equal length and that the points are listed sequentially around the perimeter of the desired shape to ensure correct rendering.

To further enhance your R programming capabilities, consider exploring how the **polygon()** function integrates with other base R graphics functions, such as **lines()** or **points()**, and how it can be incorporated into loops to programmatically draw multiple shapes based on the output of statistical models. Continuous practice with these fundamental graphical elements will significantly improve the quality and interpretability of your data visualizations in R.

Additional Resources

The following tutorials explain how to perform other common tasks in R, building upon the foundational knowledge of graphical manipulation:

<!--

Featured Posts

-->