YOUR SCIENCE

CREATING OPPORTUNITY

Topological Data Analysis
Persistent Homology

Your Science Luxembourg
Mathematical Consulting

2024

Contents

1 What is Topological Data Analysis? 2

2 Intuitive Introduction to Homology 2

3 Persistent Homology: A Primer 5

4 Persistent Homology: An Example 12

1 What is Topological Data Analysis?

Topological Data Analysis (TDA) is a mathematical framework that applies topological tools to analyze the structure and shape of complex datasets. It identifies key topological features of a data point cloud, such as connected components, loops, and voids, unveiling hidden patterns and offering valuable insights into the geometric properties inherent in the data.

One of the main techniques in TDA is Persistent Homology. It adapts the concept of homology from algebraic topology to identify features in a data cloud that persist as we zoom out of the cloud. The general assumption is that features persisting over prolonged zooming iterations are genuine, while those vanishing swiftly are considered noise.

Persistent homology is the core of these notes and will be elaborated upon in the next sections.

2 Intuitive Introduction to Homology

Imagine a network of pipes used for transporting fluids. Corrosion can create holes and potentially even separate previously connected pipe segments.

In this situation, the specific geometry of the network is largely inconsequential. What truly matters are the leaks in the network – its connected components and the holes within the pipes.

To mathematically explore connected components, holes, and voids, we start approximating the pipe network through a triangulation, as depicted in the toroidal pipe shown in the following figure.

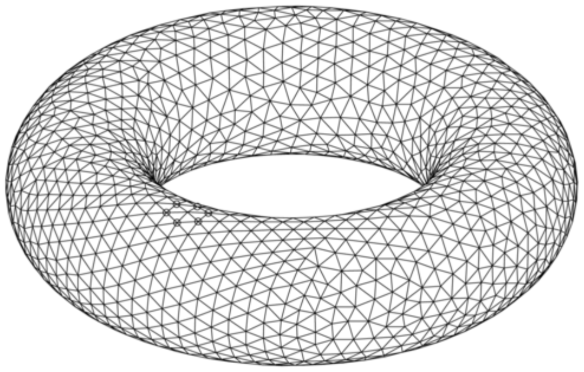

Figure 1: A triangulation of a torus

The next, more complex triangulation is not limited to triangles, also known as 2-dimensional simplices, but also includes a tetrahedron, referred to as 3-simplex, line segments or 1-simplices, and points or 0-simplices.

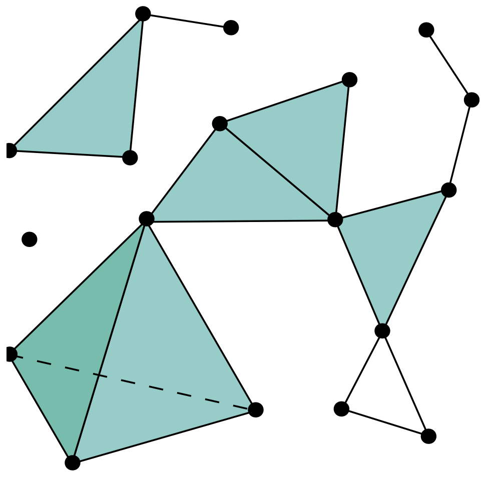

Figure 2: A simplicial complex

In the present context, it's important to note that 2-simplices (triangles), 3-simplices (tetrahedra), et cetera, are always considered solid shapes. Otherwise, we talk about an empty triangle or tetrahedron, or about the edges of a 2-simplex and the faces of a 3-simplex. Finally, the more complex triangulation depicted in Figure 2, composed of simplices of various dimensions, is referred to as a simplicial complex.

The mathematical toolkit we apply to this simplicial complex begins considering all 1-chains, i.e., all possible chains (in the common sense of the word) of 1-simplices (line segments). Within this set, it computes those 1-chains that are cycles, these 1-cycles being of course essentially made of the three edges of a triangle (see Figure 2). Subsequently, **the tool eliminates the 1-cycles that are** boundaries: in other words, it views 1-cycles that are boundaries of a triangle (a solid triangle, a green triangle in Figure 2) as **homologs of zero**. Hence, the 1-homology of the simplicial complex in Figure 2 corresponds to the sole 1-cycle that does NOT qualify as a boundary, i.e., to the loop surrounding the hole situated in the bottom-right corner of the complex.

Similarly, the 2-homology detects the 2-holes, such as the void or cavity that would manifest if the 2-cycle represented by the four faces of the tetrahedron contained no interior.

Additional Insights

The reader may already have observed that the cycles are the chains without boundary (do not confuse: a cycle *can* be a boundary, yet it does not possess a boundary itself). For example, if a 2D creature resides within the four faces of a tetrahedron, it can move freely without encountering any boundary: therefore, the 2-chain of these four faces is a 2-cycle, as previously asserted and intuitively evident. Similarly, a 1D being dwelling solely on the three edges of a triangle also experiences a boundless environment: hence, the 1-chain of these edges is a 1-cycle, as already well-known. Furthermore, since a 0-simplex or point lacks a boundary (otherwise, it would resemble a small disc), all points are considered 0-cycles. In 0-homology, akin to higher homologies, two 0-cycles, essentially points, are deemed homologous and are identified, if they form the boundary of a chain of line segments. Consequently, in 0-homology, all points within the same connected component of the simplicial complex are identified. Thus, for the simplicial complex depicted in Figure 2, the 0-homology comprises two elements, corresponding to the two connected components of the considered simplicial complex. ■

Finally, homology is a mathematical tool, which identifies hole-like structures in a simplicial complex or a topological space that the complex approximates. In summary, the 0-homology group H_0 determines the connected components, the 1-homology group H_1 identifies the loops around holes, the 2-homology group H_2 recognizes the cavities, and so forth in higher dimensions.

3 Persistent Homology: A Primer

Imagine a dataset representing the distribution of stars in a region of the night sky, with known distances between them. Initially, the stars may seem randomly scattered, but there's a suspicion of underlying patterns, such as clusters or voids.

If we start connecting stars that are close together (say, within a certain distance threshold ε_1), the resulting line segments begin to form simple shapes like triangles or tetrahedrons, building a simplicial complex. Now, if we slowly increase the distance threshold (let's call the new value ε_2, with $\varepsilon_2 > \varepsilon_1$), the complex changes. For example, if there were six connected components at ε_1, there might be only two at ε_2. However, if the initial components were widely separated, forming 'truly distinct' entities, a significantly higher threshold ($\varepsilon_3 \gg \varepsilon_2$) may be needed for them to connect. In such cases, the initial connected components persist over a broader range of thresholds. This persistence across thresholds helps us distinguish between true features of the data, which persist over a broad range, and noise-like features that vanish quickly.

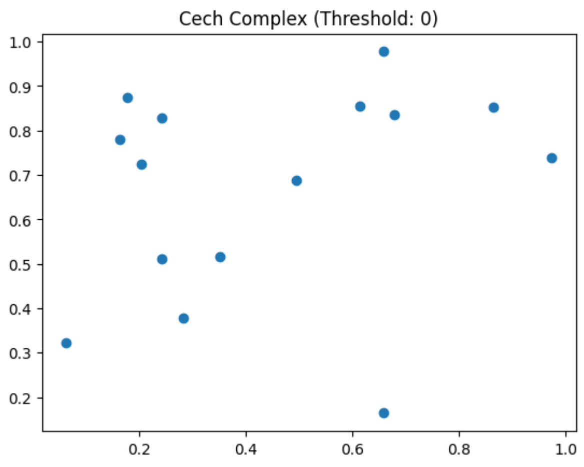

Figure 3

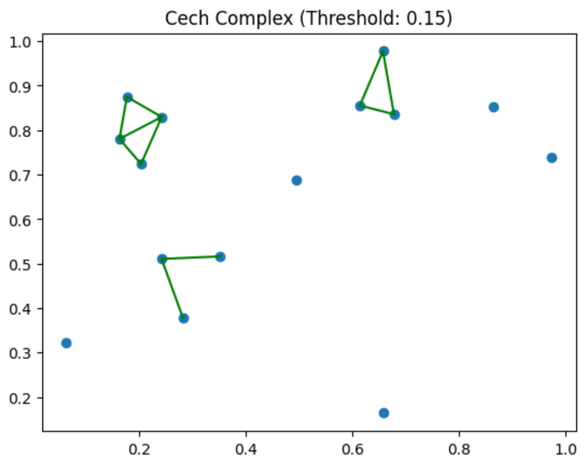

Figure 4

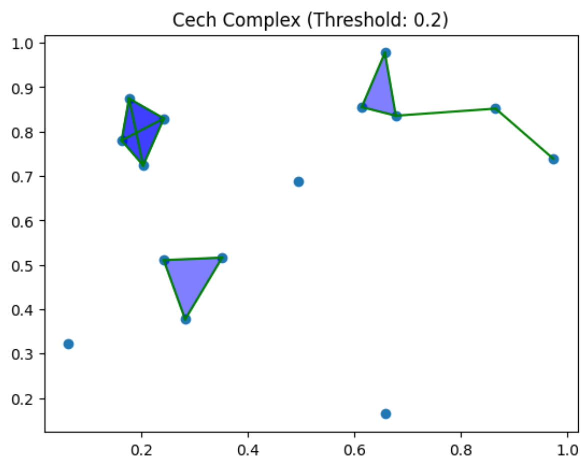

Figure 5

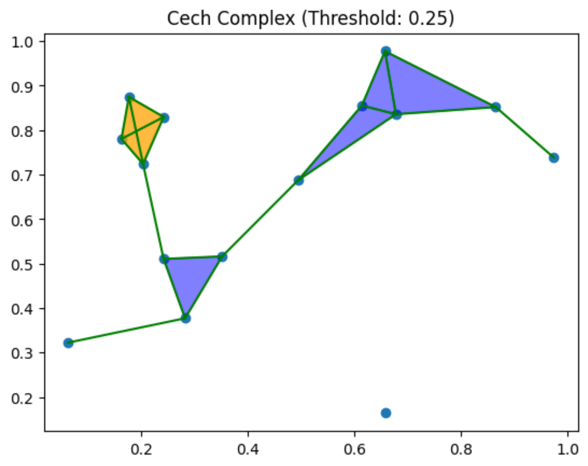

Figure 6

While the previous figures depict the transformation of the simplicial complex as the threshold increases, the question arises as to what set of rules we have applied to construct the successive simplicial complexes?

Let's imagine a scenario where we have 3D data points that together form a certain shape, such as a torus, but without us knowing that. One approach to revealing this unknown shape is to thicken the data points into **balls**, as depicted in the next figure, represented in a 2D scenario for simplicity. If our 3D data set contains a sufficient number of points distributed over the torus, the **union** of these small balls would be very similar to the torus. Computing the **homology** of this union would then reveal the topological features of a torus and thus **decipher the toroidal shape of the data cloud** considered. However, we can only compute the homology of a simplicial complex. The following figure illustrates the construction of such a complex in the 2D setting from the considered balls, which we interpret as closed balls with a diameter of ε_1 centered at the data points.

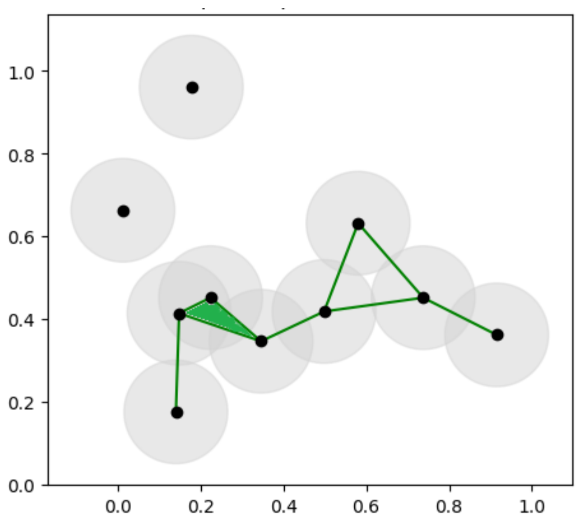

Figure 7: Čech Complex

The procedure for constructing the green simplicial complex in Figure 7 is applicable for every point cloud in a space where distance can be measured between points. Indeed, the availability of a notion of distance means that we can define the concept of a 'ball', which is the set of points whose distance to the center is at most the radius, in our case $\frac{1}{2}\varepsilon_1$. In the following precise description of the procedure, the text between square brackets pertains to concrete spaces such as the standard ambient space.

Every point of the cloud is, of course, a 0-simplex. Two (different) points form a 1-simplex [i.e., are connected by a line segment] if the intersection of the two associated balls is not empty [i.e., if the distance between the two points is at most ε_1]. Three points form a 2-simplex [i.e., are the vertices of a (solid) triangle] if the intersection of the three associated balls is not empty. Four points form a 3-simplex [i.e., are the vertices of a (solid) tetrahedron] if the intersection of the four associated balls is not empty. More generally, k points form a $(k-1)$-simplex if the intersection of the k associated balls is not empty.

Note that for three points to be connected by a (solid) triangle, the requirement is that the three balls share at least one common element, as illustrated in the left-hand side (LHS) scenario in Figure 7. In the right-hand side (RHS) scenario, this condition is not met. However, the balls associated with every pair of the three points intersect, establishing each pair as a 1-simplex. In the LHS case, the three faces of the triangle constitute a 1-cycle and this 1-cycle is the boundary of a 2-simplex. In the RHS case, we encounter a 1-cycle that is not the boundary of a 2-simplex.

The simplicial complex we have just constructed is referred to as the Čech Complex of the data point cloud at threshold ε_1 . As we increase the threshold, the Čech complex undergoes changes (see Figures 3, 4, 5, 6), consequently affecting its homology. A glance at Figure 7 reveals that the **homology** (connected components, holes, et cetera) of the **Čech complex** mirrors that of the union of balls we must compute (as mentioned above). However, determining the Čech homology is computationally intensive, necessitating checks for intersections of intersections of balls (for example, in the RHS case, the intersection of the three pairwise intersections of balls is empty, while in the LHS case, it is not). Hence, it is advantageous to substitute the Čech complex of the cloud with another, simpler complex, which is defined similarly, except that we replace the intersection of the k associated balls with the pairwise intersections:

A subset of k points from a point cloud forms a $(k-1)$-simplex if the pairwise intersections of the associated balls for these k points are non-empty [i.e., if the pairwise distances between the points are at most ε_1].

This simplicial complex is known as the Vietoris-Rips Complex (VR Complex) of the data cloud. The distinction from the Čech complex becomes apparent in the RHS scenario depicted in Figure 7: while the intersection of the three balls is empty, their pairwise intersections are not. Consequently, in the VR complex, the three points form a 2-simplex and are connected by a (solid) triangle, just as in the LHS scenario.

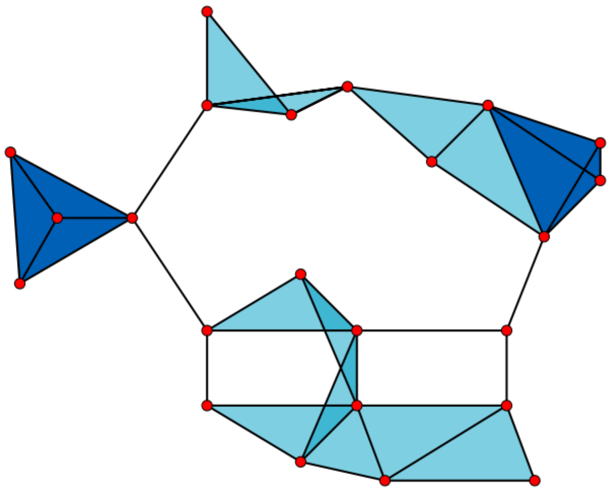

Figure 8: A Vietoris-Rips complex

Hence, in the Čech complex, the faces of this RHS triangle constitute a 1-cycle that is not a boundary, so that it survives in the Čech homology. Conversely, in the VR complex, this 1-cycle is a boundary and is thus disregarded in the VR homology. While the Čech homology, as mentioned earlier, is ideal for capturing the desired topological features, it is computationally intractable. In contrast, VR homology offers a computable approach, but it may not perfectly capture the exact homology we seek. Despite this, the **VR homology** serves effectively the purpose of capturing the primary topological characteristics of the data. Read on to find out why!

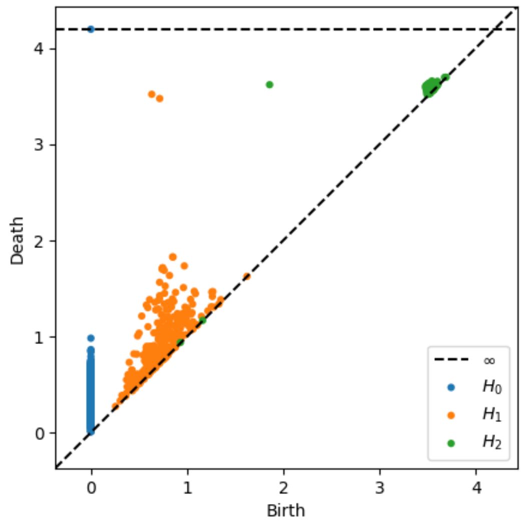

Figure 9: Persistence Diagram

In Figure 9, the (colored) points encode the appearance threshold (horizontal coordinate of the point considered) and disappearance threshold (vertical coordinate) of a topological feature (blue points: connected components; orange points: holes; green points: cavities) within the underlying 3D data point cloud (see, for instance, Figures 3, 4, 5, and 6). Points located on the diagonal exhibit identical birth and death thresholds, indicating the corresponding feature vanishes immediately after formation. Those situated close to the diagonal perish soon after their inception, lacking persistence across a broader range of thresholds, thus failing to represent genuine topological features, as previously noted. However, there exist one blue, two orange, and one green point distanced from the diagonal, thus enduring across a wide spectrum of thresholds, and consequently, characterizing authentic topological features. Specifically, the data cloud underlying the persistence diagram in Figure 9 contains 1 connected component, 2 holes, and 1 cavity.

As already mentioned, in our persistence diagram, the blue (resp., orange, green) points represent connected components (resp., holes, cavities), i.e., 0-cycles (resp., 1-cycles, 2-cycles) that are not boundaries. More precisely, they represent cycles in the 0-homology (resp., 1-homology, 2-homology) of the VR complex of the underlying cloud. As previously said, our primary interest should lie in the homology of the Čech complex of the data cloud. However, the salient aspect pertains to points far removed from the diagonal within the persistence diagram of the complex. It can be shown that the persistence diagrams of both complexes are quite similar, rendering it sufficient to utilize the homology of the computationally less demanding VR approximation of the Čech complex.

4 Persistent Homology: An Example

A leading tire manufacturer specializes in producing tires for heavy-duty machinery such as construction equipment, mining vehicles, and agricultural machinery. These tires are notably larger than standard automotive tires, necessitating meticulous inspection to guarantee safety, optimal performance, and extended durability.

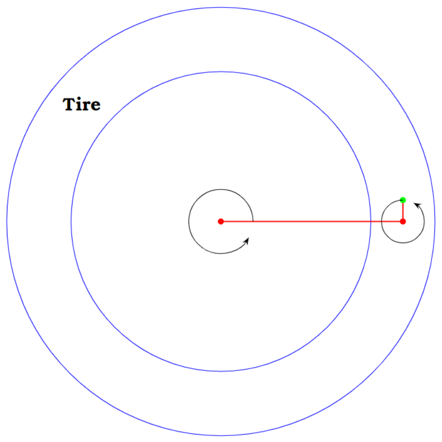

Figure 10: Two-Joint Robotic Arm

Figure 10 illustrates the two-joint robotic arm utilized for comprehensive tire quality assessment. The inner joint on the elongated arm facilitates horizontal rotation, while the outer joint on the shorter arm enables vertical rotation. This configuration empowers sensors at the short arm's end to scan the entire interior surface of the tire, monitoring various quality parameters.

The following figure depicts the locations where the robot conducted inspection measurements. It is evident that the resulting data point cloud exhibits a toroidal topology.

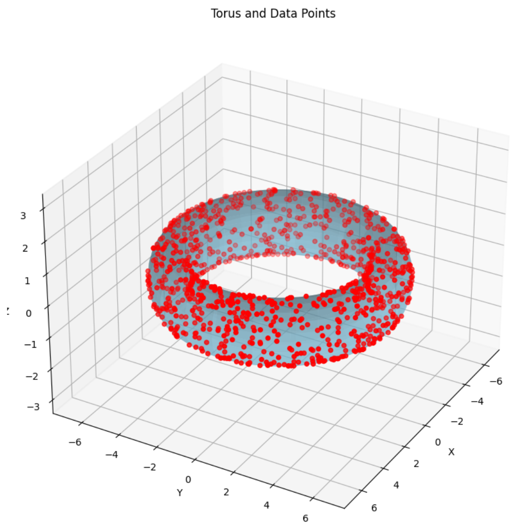

Figure 11: Data Points on the Torus

Consequently, computing its persistence diagram through the homology computation of its VR complex at various thresholds should reveal 1 connected component, 2 holes, and 1 cavity that persist across a broader range of thresholds. This persistence diagram is actually the one depicted in Figure 9. We include it again below for the reader's convenience. It clearly demonstrates the requisite features, thus confirming the effectiveness of the VR simplicial complex, persistent homology, and the persistence diagram in fulfilling their intended purpose.

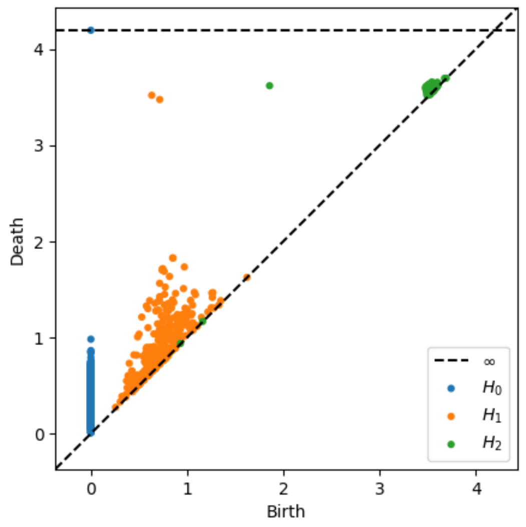

Figure 12: Persistence Diagram

Below is the Python code responsible for generating the 1000 data points distributed on the torus of Figure 11. It also computes the associated VR complexes and the persistence diagram in Figure 12. We have included comprehensive explanations, particularly aimed at readers who may be new to Python.

```
# Import the numpy library as np (numpy is the library for numerical
    computing in Python)
import numpy as np

# Import the pyplot module from the matplotlib library as plt (
    matplotlib is a plotting library whose name recalls that it was
    initially developed to replicate MATLAB's plotting capabilities;
    pyplot is a module within matplotlib that provides a simplified
    interface for creating 2D plots using Python)
import matplotlib.pyplot as plt

# Import the Axes3D class from the mpl_toolkits.mplot3d module (
    mpl_toolkits is an extension of matplotlib; mplot3d is a module
    within this extension for creating 3D plots; Axes3D is a class from
     this module used to create 3D axes)
```

```
from mpl_toolkits.mplot3d import Axes3D

# Import the Circle class from the matplotlib.patches module (
    matplotlib.patches is a module within matplotlib that provides
    various shapes and patches; Circle is a class from this module used
     to create circles and circular arcs in matplotlib plots)
from matplotlib.patches import Circle

# Import the ripser function from the ripser library (ripser is a
    Python library used for computing persistent homology of point
    clouds, and the ripser function is one of its crucial components;
    the name ripser reflects the use of the Vietoris-Rips complex in
    persistent homology computations)
from ripser import ripser

# Import the plot_diagrams function from the persim library (persim
    provides functions for visualizing persistence diagrams;
    plot_diagrams is such a function)
from persim import plot_diagrams

# Disable LaTeX rendering in matplotlib (plt.rcParams is an interface
    within the matplotlib.pyplot (plt) module that provides access to
    matplotlib's default settings for various plot parameters, such as
    line widths, colors, ... ; rc means that the parameters can be
    configured at runtime; the following code line disables the use of
    LaTeX for rendering text in plots)
plt.rcParams['text.usetex'] = False

# Define a function that generates a dataset on a torus (the following
     function generates random parameter pairs (phi, theta) that define
     points on a torus; using the toroidal formula, it computes the
    Cartesian coordinates (x, y, z) of these points, storing them in a
    numpy (np) array; np.random.uniform() is a function call within the
     np.random module in np; 'uniform' means that every value within a
    defined range has an equal probability of being generated; when
    evaluated at (x, y, z) the np function np.column_stack takes the 1
    x num_samples arrays x, y, and z and stacks them together as
    columns to form a new num_samples x 3 array)
def generate_torus_dataset(num_samples, tube_radius,
    distance_center_to_hole):
    theta = np.random.uniform(0, 2*np.pi, num_samples)
    phi = np.random.uniform(0, 2*np.pi, num_samples)
    x = (distance_center_to_hole + tube_radius * np.cos(theta)) * np.
    cos(phi)
    y = (distance_center_to_hole + tube_radius * np.cos(theta)) * np.
    sin(phi)
    z = tube_radius * np.sin(theta)
    return np.column_stack((x, y, z))

# Define a function that creates a figure containing two subplots, one
    showing a torus and the other displaying data points on the torus
    (the value fig of the function plt.figure in the module plt is a
    figure window of width 18 units and height 8 units)
def plot_torus_standard_view(torus_data, tube_radius,
    distance_center_to_hole):
    fig = plt.figure(figsize=(18, 8))

    # Create a subplot showing a torus surface (the subplot code adds
    to the figure window a grid layout with 1 row and 2 columns, the
    subplot being addressed is in position 1, and the subplot uses a 3D
    projection; the np function np.linspace generates arrays of evenly
    spaced numbers; the function ax.plot_surface plots a surface
    within the subplot ax; alpha is a transparency parameter, the last
    two parameters fix the row and column stride in the plot)
    ax = fig.add_subplot(1, 2, 1, projection='3d')
    num_points = 100
    theta, phi = np.meshgrid(np.linspace(0, 2 * np.pi, num_points), np
    .linspace(0, 2 * np.pi, num_points))
    x = (distance_center_to_hole + tube_radius * np.cos(theta)) * np.
    cos(phi)
    y = (distance_center_to_hole + tube_radius * np.cos(theta)) * np.
    sin(phi)
    z = tube_radius * np.sin(theta)
    ax.plot_surface(x, y, z, color='skyblue', alpha=0.5, rstride=5,
    cstride=5)

    # Scatter data points on the torus (the torus_data will be
    generated later as a num_samples x 3 array using the
```

```
   generate_torus_dataset function above; s is the size of the points;
    the function ax.view_init allows us to set the two viewing angles)
    ax.scatter(torus_data[:, 0], torus_data[:, 1], torus_data[:, 2],
   color='red', s=20)
    ax.view_init(azim=30, elev=30)
    ax.set_title('Torus and Data Points')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    # Set equal scaling for x and y axes and adjust z axis limits (in
   the getattr function the f-string contains the name of the
   attribute to get retrieved from the subplot ax for dim = x and for
   dim = y; the parentheses () at the end of getattr(...) is a method
   call, i.e., it executes the method's code and retrieves the values
   of the attribute; the square brackets [] stores the limits of x and
    y in a list, say [[0,1], [1,2]], and np.array transforms this list
    into the obvious 2x2 numpy array; the unusual operator *...*2
   replicates the list [0,2] with the minimal and maximal values of
   the numpy array scaling twice and makes sure we get equal scaling
   for the x and y axes; the interval [-3,3] is the scaling for the z
   axis; the function call plt.tight_layout() ensures that the plot
   fits into the figure area)
    scaling = np.array([getattr(ax, f'get_{dim}lim')() for dim in 'xy'
   ])
    ax.auto_scale_xyz(*[[np.min(scaling), np.max(scaling)]]*2, [-3,
   3])

    plt.tight_layout()
    plt.show()

# Set the parameters required to call the two previous functions
num_samples = 1000
tube_radius = 1
distance_center_to_hole = 5

# Call the first function defined above to generate the torus data
torus_data = generate_torus_dataset(num_samples, tube_radius,
   distance_center_to_hole)
```

```

# Call the second function defined above to plot a torus and data
    points on it, using the previous parameters, equal x and y axes
    scaling, and adjusted z axis range
plot_torus_standard_view(torus_data, tube_radius,
    distance_center_to_hole)

# Compute the Vietoris-Rips complex of the torus data and its homology
     at different thresholds, as well as the corresponding persistence
    diagram, including H_2 (ripser(...) is a dictionary containing
    various features including persistence diagrams)
diagrams = ripser(torus_data, maxdim=2)['dgms']

# Plot the persistence diagram (the code creates a figure of 10x5
    units and axes inside it; show=True means that the diagram will be
    displayed immediately after calling the function; ax is the
    parameter that specifies the axes where the diagram will be plotted
    )
fig, ax = plt.subplots(figsize=(10, 5))
plot_diagrams(diagrams, show=True, ax=ax)
ax.set_title('Persistence Diagram (including H_0, H_1, and H_2)')

# Explicitly call the function plt.show as a fail-safe mechanism to
    ensure the plot is displayed even in case of inconsistencies in
    your Python environment's automatic display settings.
plt.show()
```

You can now execute the previous code in Google Colab or your local Python environment to generate figures similar to Figures 11 and 12. Due to the random nature of the data, the generated figures may differ slightly from those presented in this document. The code may take several minutes to complete its execution.