Exploring Roulettes: A Comprehensive Study of Cycloids, Hypotrochoids, and Epitrochoids

Content

1. Introduction
2. Importing Dependencies
3. Cycloids
   - Common Cycloid
   - Curtate Cycloids
   - Prolate Cycloids
4. Trochoids class
5. Trochoids Comparison
6. Epicycloids
7. Hypocycloids
8. Hypotrochoids
9. Epitrochoids
10. Cycloidal Motion of a Circular Disk in 3D
11. Cycloidal Pendulum
12. Tautochrone problem
13. References

Introduction

This notebook explores roulettes, which are curves generated by the motion of one curve rolling on another. Specifically, it focuses on three main types of roulettes:

1. **Trochoids**: Generated when a circle rolls along a straight line.
2. **Hypotrochoids**: Generated when a circle rolls inside another circle.
3. **Epitrochoids**: Generated when a circle rolls outside another circle.

Key Topics Covered:

Cycloids:

- Discussed the three types of cycloids: **Common Cycloids**, **Curtate Cycloids**, and **Prolate Cycloids.**
- Plotted and animated the motion of each type.
- Introduced a generalized class `Trochoids` to handle different types of cycloids.

Hypotrochoids:

- Explored curves generated by a circle rolling inside another circle.
- Discussed **Hypocycloids** as a special case of hypotrochoids.
- Covered special cases like **Rose Curves** and **Ellipses.**
- Analyzed the effect of the ratio $n = \frac{a}{b}$ (where a is the radius of the fixed circle and b is the radius of the rolling circle) on the shape and revolution of the curve for integer, rational, and irrational values of n.

Epitrochoids:

- Examined curves generated by a circle rolling outside another circle.
- Discussed **Epicycloids** as a special case of epitrochoids.
- Covered other types like **Rose Curves**, **Circles**, and **Limacons.**
- Analyzed the dependence of the curve's shape and revolution on the ratio n.

Tautochronic Property of Cycloids:

- Demonstrated the **tautochronic property** of cycloids, where a particle sliding under gravity along an inverted cycloidal path takes the same time to reach the bottom, regardless of its starting point.
- Discussed the **Cycloidal Pendulum**, its theory, equations of motion, and animated its motion for different amplitudes.
- Illustrated the tautochronic problem using an animation where multiple balls, starting from different positions on an inverted cycloidal path, reach the bottom simultaneously.

This notebook combines theoretical insights, visualizations, and animations to provide a comprehensive understanding of roulettes and their fascinating properties.

Importing Dependencies

In [ ]:
```
# Core libraries
import os
import math
import itertools
import warnings
from pathlib import Path
from fractions import Fraction

# Numerical and scientific computing
import numpy as np

# Visualization
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation, FFMpegWriter
from matplotlib import patches
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

# Interactive widgets and display
import ipywidgets as widgets
from IPython.display import display, clear_output

# Progress bar
from tqdm import tqdm

# Matplotlib interactive mode
%matplotlib ipympl
%matplotlib widget
```

Cycloids

Common Cycloid

Overview

A **cycloid** is a curve generated by a point on the circumference of a circle as it rolls along a straight line without slipping. It is a fascinating curve with applications in physics, engineering, and mathematics, particularly in the study of motion, pendulums, and gear design.

Parametric Equations of a Cycloid

The parametric equations for a cycloid are derived based on the geometry of the rolling circle. Let:

- `r` be the radius of the rolling circle.
- `θ` (theta) be the angle of rotation of the circle (in radians).

The parametric equations for the cycloid are:

1. **X-coordinate**:

$$x(\theta) = r(\theta - \sin(\theta))$$

   This represents the horizontal displacement of the point.

2. **Y-coordinate**:

$$y(\theta) = r(1 - \cos(\theta))$$

   This represents the vertical displacement of the point.

Here:

- `sin(θ)` and `cos(θ)` account for the circular motion of the point.
- `θ` accounts for the linear motion of the circle along the straight line.

How the Motion is Generated

The motion of a cycloid is generated by the combination of:

1. **Linear motion**: The circle rolls along a straight line without slipping. The distance traveled by the circle along the line is proportional to the angle of rotation (`θ`).
2. **Rotational motion**: The point on the circle moves in a circular path relative to the center of the circle.

The combination of these two motions results in the cycloid curve.

Types of Cycloids

Cycloids can be classified into three main types based on the position of the tracing point relative to the rolling circle:

1. **Normal Cycloid**:
   - The tracing point lies on the circumference of the rolling circle.
   - Parametric equations:

$$x(\theta) = r(\theta - \sin(\theta))$$
$$y(\theta) = r(1 - \cos(\theta))$$

2. **Curtate Cycloid**:
   - The tracing point lies **inside** the rolling circle (closer to the center).
   - Parametric equations:

$$x(\theta) = r(\theta - k\sin(\theta))$$
$$y(\theta) = r(1 - k\cos(\theta))$$

   Here, `k < 1` is a scaling factor representing the distance of the point from the center of the circle.

3. **Prolate Cycloid**:
   - The tracing point lies **outside** the rolling circle (beyond the circumference).
   - Parametric equations:

$$x(\theta) = r(\theta - k\sin(\theta))$$
$$y(\theta) = r(1 - k\cos(\theta))$$

   Here, `k > 1` is a scaling factor representing the distance of the point from the center of the circle.

Key Properties of Cycloids

1. **Arc Length**: The arc length of one complete revolution of a cycloid is `8r`.
2. **Area Under One Arch**: The area under one arch of a cycloid is `3πr²`.
3. **Tautochrone Property**: A particle sliding along a cycloid under gravity will take the same time to reach the bottom, regardless of its starting point.
4. **Brachistochrone Property**: The cycloid is the curve of fastest descent under gravity between two points.

Applications of Cycloids

1. **Pendulum Design**: Cycloidal paths are used in pendulum clocks to ensure isochronous motion.
2. **Gear Design**: Cycloidal gears are used in mechanical systems for smooth and efficient motion.
3. **Roller Coasters**: Cycloidal curves are used in designing roller coaster loops for optimal speed and safety.
4. **Physics**: The tautochrone and brachistochrone properties make cycloids important in classical mechanics.

In [ ]:
```
def plot_cycloid(r=1.0, num_revolutions=2, save_fig=False, filename=None):
    """
    Plot a cycloid with its companion curve.

    Parameters:
    -----------
    r : float
        Radius of the rolling circle (default: 1.0)
    num_revolutions : int
        Number of complete revolutions (default: 2)
    save_fig : bool
        Whether to save the figure (default: False)
    filename : str, optional
        Name of the file to save the figure (default: None)
    """
    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=(10, 4))
```

```python
    ax.set_title(f'Cycloid with Companion Curve\nRadius = {r}, Revolutions = {num_revolutions}')
    fig.subplots_adjust(left=0.1, right=0.95, top=0.9, bottom=0.1)

    theta_max = num_revolutions * 2 * np.pi
    theta_vals = np.linspace(0, theta_max, 200)

    # Set x-axis ticks in multiples of pi*r
    xticks = np.linspace(0, theta_max * r, int(2 * num_revolutions + 1))
    xtick_labels = []
    for x in xticks:
        val = x / (np.pi * r)
        if np.isclose(val, 0):
            xtick_labels.append('$0$')
        elif np.isclose(val, 1):
                xtick_labels.append(r"$\pi r$") # Show r explicitly for clarity
        elif np.isclose(val % 1, 0):
                xtick_labels.append(f"${int(val)}\\pi r$")
        else:
                xtick_labels.append(f"${val:.1f}\\pi r$") # Fallback for non-integers

    ax.set_xticks(xticks)
    ax.set_xticklabels(xtick_labels)

    y_padding = 0.5
    ax.set_xlim(-r, r * theta_max + r)
    ax.set_ylim(-y_padding, 2 * r + y_padding + 0.5)
    ax.set_yticks(np.arange(-y_padding, 2 * r + y_padding + 0.5))
    ax.set_yticklabels([f'{val:.1f}' for val in np.arange(-y_padding, 2 * r + y_padding + 0.5)])
    ax.set_xlabel("Distance rolled (proportional to $\\theta$)")
    ax.set_ylabel("Y(in units of r)")
    ax.grid(False)
    ax.set_aspect('equal')

    # Parametric Equation of the Cycloid
    def cycloid_equation(theta):
        x = r * (theta - np.sin(theta))
        y = r * (1 - np.cos(theta))
        return x, y

    # Parametric Equation of the Companion curve
    x_com = r * theta_vals
    y_com = r * (1 - np.cos(theta_vals))

    # Plotting
    # Base line
    ax.axhline(0, color='gray', linewidth=2)

    # Cycloid path
    x_cycloid, y_cycloid = cycloid_equation(theta_vals)
    ax.plot(x_cycloid, y_cycloid, 'r-', linewidth=1.5, label='Cycloid Path')

    # Companion Curve
    ax.plot(x_com, y_com, color='orange', lw=1.5, ls='--', label='Companion Curve')

    # Legend
    ax.legend(loc='upper right', fontsize=8)

    # Circle
    for i, angle in enumerate([0, np.pi/2, np.pi, 3*np.pi/2, 2*np.pi]):
        opacity = 0.2 * (i + 1)  # Increasing opacity
        X, Y = cycloid_equation(angle)
        ax.plot([r * angle, X], [r, Y], color='blue', lw=1, marker='o', ms=10,
                mfc='white', mec='black', zorder=4, alpha=opacity)
        circle = patches.Circle((r * angle, r), r, fill=False, color='green', lw=1.5, alpha=opacity, zorder=3)
        ax.add_patch(circle)

    # Save figure
    if save_fig and filename is not None:
        save_dir = "IMAGES/CYCLOIDS"
        os.makedirs(save_dir, exist_ok=True)
        filename = os.path.join(save_dir, filename)
        print(f"Saving figure to {os.path.abspath(filename)}...")
        plt.savefig(filename, bbox_inches='tight', dpi=300)
        print(f"Figure saved successfully!")
        plt.close(fig)
    else:
        plt.show()


plot_cycloid(r=1, num_revolutions=2, save_fig=False, filename="cycloid.png")
```

Curtate Cycloid

An Overview

A **Curtate Cycloid** is a type of cycloidal curve generated by a point that is located inside a rolling circle. This is in contrast to a **Prolate Cycloid**, where the point is outside the circle, and a **Simple Cycloid**, where the point is on the circumference of the circle.

The Curtate Cycloid is a fascinating curve because it arises in various physical and mathematical contexts, such as the motion of gears, pendulums, and even in the design of certain mechanical systems.

How the Motion is Generated

The motion of a Curtate Cycloid is generated as follows:

1. Imagine a circle of radius `r` rolling along a straight line without slipping.
2. A point is fixed inside the circle at a distance `a` from the center of the circle, where `a < r`.
3. As the circle rolls, the point traces a curve in space. This curve is the Curtate Cycloid.

The key difference between a Curtate Cycloid and other types of cycloids is the position of the tracing point:

- **Curtate Cycloid**: The point is inside the circle (`a < r`).
- **Simple Cycloid**: The point is on the circumference of the circle (`a = r`).
- **Prolate Cycloid**: The point is outside the circle (`a > r`).

Parametric Equations of a Curtate Cycloid

The parametric equations for a Curtate Cycloid are derived based on the geometry of the rolling circle and the position of the tracing point. Let:

- `r` be the radius of the rolling circle,
- `a` be the distance of the tracing point from the center of the circle,
- `θ` (theta) be the angle of rotation of the circle.

The parametric equations are:

1. **X-coordinate**:

$$x(\theta) = r(\theta - a\sin(\theta))$$

   This represents the horizontal displacement of the tracing point.

2. **Y-coordinate**:

$$y(\theta) = r(1 - a\cos(\theta))$$

   This represents the vertical displacement of the tracing point.

Here:

- `θ` is the angular position of the rolling circle (in radians),
- `sin(θ)` and `cos(θ)` account for the projection of the tracing point's motion inside the circle.

Key Features of the Curtate Cycloid

1. **Oscillatory Nature**:
   The curve oscillates above and below the baseline (the line along which the circle rolls). The amplitude of oscillation depends on the distance `a`.

2. **Periodicity**:
   The curve is periodic, repeating itself after one complete revolution of the circle. The period is proportional to the circumference of the circle, i.e., `2πr`.

3. **No Cusps**:
   Unlike a Simple Cycloid, which has sharp cusps where the tracing point touches the baseline, a Curtate Cycloid does not have cusps because the tracing point never reaches the baseline.

4. **Applications**:
   Curtate Cycloids are used in the design of gears, cams, and other mechanical systems where smooth, oscillatory motion is

required.

Visualization of the Motion

To visualize the motion:

1. Imagine the circle rolling along a straight line.
2. The point inside the circle moves in a smooth, oscillatory path, tracing the Curtate Cycloid.
3. The horizontal motion (`x`) is influenced by the rolling of the circle, while the vertical motion (`y`) is influenced by the position of the point inside the circle.

Example in Python

The Python function `plot_curtate_cycloid` below visualizes this motion. It uses the parametric equations of the Curtate Cycloid to compute the `x` and `y` coordinates for a range of angles `θ` and plots the resulting curve. The function also compares the Curtate Cycloid with its companion curve (the path traced by a point on the circle's circumference).

In [ ]:
```
def plot_curtate_cycloid(r=1.0, a=0.8, num_revolutions=2, save_fig=False, filename=None):
    """
    Plot a curtate cycloid with its companion curve.
    """
    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=(10, 4))
    ax.set_title(f'Curtate Cycloid with Companion Curve\nRadius = {r}, a = {a}, Revolutions = {num_revolutions}')
    fig.subplots_adjust(left=0.1, right=0.95, top=0.9, bottom=0.1)

    theta_max = num_revolutions * 2 * np.pi
    theta_vals = np.linspace(0, theta_max, 200)

    # Set x-axis ticks in multiples of pi*r
    xticks = np.linspace(0, theta_max * r, int(2 * num_revolutions + 1))
    xtick_labels = []
    for x in xticks:
        val = x / (np.pi * r)
        if np.isclose(val, 0):
            xtick_labels.append('$0$')
        elif np.isclose(val, 1):
                xtick_labels.append(r"$\pi r$") # Show r explicitly for clarity
        elif np.isclose(val % 1, 0):
                xtick_labels.append(f"${int(val)}\\pi r$")
        else:
                xtick_labels.append(f"${val:.1f}\\pi r$") # Fallback for non-integers

    ax.set_xticks(xticks)
    ax.set_xticklabels(xtick_labels)


    y_padding = 0.5
    ax.set_xlim(-r, r * theta_max + r)
    ax.set_ylim(-y_padding, 2 * r + y_padding + 0.5)
    ax.set_yticks(np.arange(-y_padding, 2 * r + y_padding + 0.5))
    ax.set_yticklabels([f'{val:.1f}' for val in np.arange(-y_padding, 2 * r + y_padding + 0.5)])
    ax.set_xlabel("Distance rolled (proportional to $\\theta$)")
    ax.set_ylabel("Y (in units of r)")
    ax.grid(False)
    ax.set_aspect('equal', adjustable='box')

    # Parametric Equation of the Prolate Cycloid
    def curtate_cycloid_equation(theta):
        x = r * (theta - a*np.sin(theta))
        y = r * (1 - a*np.cos(theta))
        return x, y

    # Plotting
    # base line
    ax.axhline(0, color='gray', linewidth=2)
    ax.axvline(0, color='gray', linewidth=2)


    # cycloid path
    x_curtate, y_curtate = curtate_cycloid_equation(theta_vals)
    ax.plot(x_curtate, y_curtate, 'r-', lw=1.5, label='Cycloid Path')

    # Companion Curve
    x_com = r * theta_vals
    y_com = r * (1 - np.cos(theta_vals))
    ax.plot(x_com, y_com, color='orange', lw=1.5, ls='--', label='Companion Curve')
```

```python
    # Legend
    ax.legend(loc='upper right', fontsize=8)

    # Circle
    for i, angle in enumerate([0, np.pi/2, np.pi, 3*np.pi/2, 2*np.pi]):
        opacity = 0.2 * (i + 1)  # Increasing opacity
        X, Y = curtate_cycloid_equation(angle)
        ax.plot([r*angle, X], [r, Y],color='blue', lw=1, marker='o', ms=10,
                mfc='white', mec='black', zorder=4, alpha=opacity)
        circle = patches.Circle((r*angle, r), r, fill=False, color='green', lw=1.5, alpha=opacity, zorder=3)
        ax.add_patch(circle)

    # Save figure
    if save_fig and filename is not None:
        save_dir = "IMAGES/CYCLOIDS "
        os.makedirs(save_dir, exist_ok=True)
        filename = os.path.join(save_dir, filename)
        print(f"Saving figure to {os.path.abspath(filename)}...")
        plt.savefig(filename, bbox_inches='tight', dpi=300)
        print(f"Figure saved successfully!")
        plt.close(fig)
    else:
        plt.show()

plot_curtate_cycloid(r=1.0, a=0.5, num_revolutions=2, save_fig=True, filename="curtate_cycloid.png")
```

Prolate Cycloid

An Overview

A **prolate cycloid** is a type of cycloid, which is a curve generated by a point on the circumference of a circle as it rolls along a straight line. Specifically, in a prolate cycloid, the point generating the curve is located **outside the rolling circle**. This is in contrast to a regular cycloid (where the point is on the circle) or a curtate cycloid (where the point is inside the circle).

The prolate cycloid exhibits a wavelike motion with larger amplitudes due to the point being outside the circle. This makes it useful in various applications, such as modeling certain mechanical motions or wave-like phenomena.

Parametric Equations of a Prolate Cycloid

The parametric equations for a prolate cycloid are derived based on the geometry of the rolling circle and the position of the tracing point. Let:

- **r**: Radius of the rolling circle.
- **a**: Distance of the tracing point from the center of the circle (where **a > r** for a prolate cycloid).
- **θ**: Angle of rotation of the circle (in radians).

The parametric equations are:

1. **X-coordinate**:

$$x(\theta) = r(\theta - a\sin(\theta))$$

   This represents the horizontal displacement of the tracing point.

2. **Y-coordinate**:

$$y(\theta) = r(1 - acos(\theta))$$

   This represents the vertical displacement of the tracing point.

Here:

- The term `r * θ` accounts for the horizontal motion of the circle as it rolls.
- The terms involving `sin(θ)` and `cos(θ)` account for the oscillatory motion of the tracing point relative to the circle.

How the Motion is Generated

The motion of a prolate cycloid is generated by the following process:

1. **Rolling Circle**: A circle of radius **r** rolls along a straight line without slipping. The no-slip condition ensures that the distance traveled by the circle along the line is equal to the arc length of the circle.

2. **Tracing Point**: A point located at a distance **a** (where **a > r**) from the center of the circle traces the curve. Since the point is outside the circle, it moves in a more exaggerated, wave-like path compared to the center of the circle.

3. **Combination of Motions**:

   - The **horizontal motion** of the tracing point is influenced by the rolling of the circle and the oscillation caused by the point's position outside the circle.
   - The **vertical motion** is determined by the oscillation of the point as the circle rotates.

4. **Resulting Path**: The combination of these motions produces the characteristic wavelike shape of the prolate cycloid, with peaks and troughs that are larger than those of a regular cycloid.

Key Characteristics of Prolate Cycloids

1. **Amplitude**: The amplitude of the prolate cycloid is proportional to the distance **a**. Larger values of **a** result in taller peaks and deeper troughs.

2. **Periodicity**: The curve is periodic, repeating itself after every full revolution of the circle (i.e., after an angle of **2π** radians).

3. **Applications**: Prolate cycloids are used in physics and engineering to model wave-like motions, pendulum paths, and certain gear mechanisms.

Visualization

In the Python code below, the prolate cycloid is visualized using the parametric equations. The function `prolate_cycloid_equation` computes the **x** and **y** coordinates for a range of angles **θ**, and the resulting curve is plotted. The rolling circle and the tracing point are also shown to illustrate how the motion is generated.

In [ ]:
```
def plot_prolate_cycloid(r=1.0, a=1.5, num_revolutions=2, save_fig=False, filename=None):
    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=(10, 4))
    ax.set_title(f'Prolate Cycloid\nRadius = {r}, a = {a}, Revolutions = {num_revolutions}')
    fig.subplots_adjust(left=0.1, right=0.95, top=0.9, bottom=0.1)

    theta_max = num_revolutions * 2 * np.pi
    theta_vals = np.linspace(0, theta_max, 200)

    # Set x-axis ticks in multiples of pi*r
    xticks = np.linspace(0, theta_max * r, int(2 * num_revolutions + 1))
    xtick_labels = []
    for x in xticks:
        val = x / (np.pi * r)
        if np.isclose(val, 0):
            xtick_labels.append('$0$')
        elif np.isclose(val, 1):
                xtick_labels.append(r"$\pi r$") # Show r explicitly for clarity
        elif np.isclose(val % 1, 0):
                xtick_labels.append(f"${int(val)}\\pi r$")
        else:
                xtick_labels.append(f"${val:.1f}\\pi r$") # Fallback for non-integers

    ax.set_xticks(xticks)
    ax.set_xticklabels(xtick_labels)

    # Adjust y_padding based on 'a' parameter since prolate cycloid (a>r) has larger amplitude
    y_padding = 0.15
    ax.set_xlim(-r, r * theta_max + r)
    # Increase y-axis limits to accommodate the larger amplitude of prolate cycloid
    y_lim_down = a-r + 0.5
    y_lim_up = r + a + 1.2
    ax.set_ylim(-y_lim_down, y_lim_up)
    ax.set_yticks(np.arange(-y_lim_down, y_lim_up))
    ax.set_yticklabels([f'{val:.1f}' for val in np.arange(-y_lim_down, y_lim_up)])
    ax.set_xlabel("Distance rolled (proportional to $\\theta$)")
    ax.set_ylabel("Y (in units of r)")
    ax.grid(False)
    ax.set_aspect('equal', adjustable='box')

    # Parametric Equation of the Prolate Cycloid
    def prolate_cycloid_equation(theta):
        x = r * (theta - a*np.sin(theta))
        y = r * (1 - a*np.cos(theta))
        return x, y

    # Plotting
    # base line
```

```python
    ax.axhline(0, color='gray', linewidth=2)
    ax.axvline(0, color='gray', linewidth=2)

    # cycloid path
    x_prolate, y_prolate = prolate_cycloid_equation(theta_vals)
    ax.plot(x_prolate, y_prolate, 'r-', lw=1.5, label='Cycloid Path')


    # Circle and tracing point
    for i, angle in enumerate([0, np.pi/2, np.pi, 3*np.pi/2, 2*np.pi]):
        opacity = 0.2 * (i + 1)  # Increasing opacity
        X, Y = prolate_cycloid_equation(angle)
        ax.plot([r*angle, X], [r, Y], color='blue', lw=1, marker='o', ms=10,
                mfc='white', mec='black', zorder=4, alpha=opacity)
        circle = patches.Circle((r*angle, r), r, fill=False, color='green', lw=1.5, alpha=opacity, zorder=3)
        ax.add_patch(circle)

    ax.legend(loc='upper right', fontsize=8)

    # Save figure
    if save_fig and filename is not None:
        save_dir = "IMAGES/CYCLOIDS"
        os.makedirs(save_dir, exist_ok=True)
        filename = os.path.join(save_dir, filename)
        print(f"Saving figure to {os.path.abspath(filename)}...")
        plt.savefig(filename, bbox_inches='tight', dpi=300)
        print(f"Figure saved successfully!")
        plt.close(fig)
    else:
        plt.show()

plot_prolate_cycloid(r=1.0, a=1.5, num_revolutions=2, save_fig=True, filename="prolate_cycloid.png")
```

Trochoids

Overview

A **trochoid** is a type of curve generated by a point attached to a circle as it rolls along a straight line. The motion of the point depends on its position relative to the rolling circle, which determines the type of trochoid.

Types of Trochoids

Trochoids can be classified into three main types based on the position of the tracing point relative to the rolling circle:

1. **Common Trochoid (Cycloid)**:
   - The tracing point lies on the circumference of the rolling circle.
   - The curve is called a **cycloid**.
   - Example: The path traced by a point on the rim of a rolling wheel.
2. **Curtate Trochoid**:
   - The tracing point lies **inside** the rolling circle (closer to the center).
   - The curve oscillates above and below the baseline but does not form sharp cusps.
   - Example: The motion of a point inside a rolling gear.
3. **Prolate Trochoid**:
   - The tracing point lies **outside** the rolling circle (beyond the circumference).
   - The curve has larger amplitudes and may form loops.
   - Example: The path traced by a point on a spoke of a rolling wheel.

Parametric Equations of Trochoids

The parametric equations for a trochoid are derived based on the geometry of the rolling circle and the position of the tracing point. Let:

- `r` be the radius of the rolling circle.
- `d` be the distance of the tracing point from the center of the rolling circle.
- `θ` (theta) be the angle of rotation of the circle (in radians).

The parametric equations are:

1. **X-coordinate**:

$$x(\theta) = r\theta - d\sin(\theta)$$

   This represents the horizontal displacement of the tracing point.

2. **Y-coordinate**:

$$y(\theta) = r - d\cos(\theta)$$

   This represents the vertical displacement of the tracing point.

Here:

- `d = r` for a common trochoid (cycloid).
- `d < r` for a curtate trochoid.
- `d > r` for a prolate trochoid.

Key Parameters of Trochoids

1. **Radius (`r`)**:
   - The radius of the rolling circle.
   - Determines the size of the curve.
2. **Distance (`d`)**:
   - The distance of the tracing point from the center of the rolling circle.
   - Determines the type of trochoid:
     - `d = r` : Common trochoid (cycloid).
     - `d < r` : Curtate trochoid.
     - `d > r` : Prolate trochoid.
3. **Number of Revolutions**:
   - The number of complete revolutions of the rolling circle.
   - Determines the length of the curve.
4. **Angle (`θ`)**:
   - The angle of rotation of the rolling circle.
   - Used to calculate the position of the tracing point.

Applications of Trochoids

1. **Engineering**:
   - Trochoidal paths are used in the design of gears, cams, and other mechanical systems.
   - Cycloidal gears are known for their smooth and efficient motion.
2. **Physics**:
   - Trochoids describe the motion of particles in certain physical systems, such as pendulums and wave patterns.
3. **Mathematics**:
   - Trochoids are studied for their geometric and algebraic properties.
4. **Animation and Visualization**:
   - Trochoids are used to create visually appealing animations and patterns.

Visualization in Python

In the code shell below, the `Trochoids` class is used to generate and animate trochoid curves. The class allows customization of parameters such as the radius (`r`), distance (`d`), and number of revolutions. The animations provide an intuitive understanding of how trochoids are formed and how their shapes vary with different parameters.

Key Components:

1. **Attributes**:
   - `r` , `a` , `b` : Radius of the circle and distances of the tracing point for curtate/prolate cycloids.
   - `num_revolutions` : Number of revolutions for the animation.
   - `save_anim` , `filename` : Whether to save the animation and the file name.

2. **Methods**:

   - `_trochoid_coordinates` : Computes the `(x, y)` coordinates of the trochoid using the parametric equations:

$$x(\theta) = r\theta - d\sin(\theta)$$
$$y(\theta) = r - d\cos(\theta)$$

   - `animate` : Sets up the animation for the specified trochoid type (`common` , `curtate` , or `prolate`), validates parameters, and either displays or saves the animation.
3. **Animation**:

   - Uses `matplotlib` to animate the rolling circle and the traced curve.
   - Dynamically updates the circle's position, the tracing point, and the trochoid path for each frame.
4. **Validation**:

   - Ensures `a < r` for curtate and `b > r` for prolate cycloids.
   - Raises errors for invalid parameters or unsupported trochoid types.
5. **Example Usage**:

   - Demonstrates creating and animating different trochoid types, with options to save the animation as a `.gif` or `.mp4` .

This code is useful for visualizing mathematical curves and understanding the geometry of rolling circles. - `animate` : Sets up the animation for the specified trochoid type (`common` , `curtate` , or `prolate`), validates parameters, and either displays or saves the animation.

3. **Animation**:

   - Uses `matplotlib` to animate the rolling circle and the traced curve.
   - Dynamically updates the circle's position, the tracing point, and the trochoid path for each frame.
4. **Validation**:

   - Ensures `a < r` for curtate and `b > r` for prolate cycloids.
   - Raises errors for invalid parameters or unsupported trochoid types.
5. **Example Usage**:

   - Demonstrates creating and animating different trochoid types, with options to save the animation as a `.gif` or `.mp4` .

In [ ]:
```
class Trochoids:
    """
    A class to generate and animate trochoids: common, curtate, and prolate.

    A trochoid is the curve traced by a point attached to a circle as it rolls
    along a straight line.
    - Common Cycloid: The point is on the circumference (distance d = r).
    - Curtate Cycloid: The point is inside the circle (distance a < r).
    - Prolate Cycloid: The point is outside the circle (distance b > r).

    Attributes:
    -----------
    r : float
        Radius of the rolling circle.
    a : float
        Distance of the tracing point from the center for a curtate cycloid (a < r).
    b : float
        Distance of the tracing point from the center for a prolate cycloid (b > r).
    num_revolutions : int
        Number of complete revolutions for the animation.
    save_anim : bool
        Flag to indicate whether to save the animation to a file.
    filename : str or None
        The name of the file to save the animation (if save_anim is True).
    fig : matplotlib.figure.Figure
        The figure object for the plot.
    ax : matplotlib.axes.Axes
        The axes object for the plot.
    """

    def __init__(self, r=1.0, a=0.5, b=1.5, num_revolutions=2, save_anim=False, filename=None):
        """
        Initializes the Trochoids class with parameters.
```

```python
        Parameters:
        -----------
        r : float, optional
            Radius of the rolling circle (default: 1.0).
        a : float, optional
            Distance parameter for curtate cycloid (default: 0.5). Must be < r.
        b : float, optional
            Distance parameter for prolate cycloid (default: 1.5). Must be > r.
        num_revolutions : int, optional
            Number of complete revolutions (default: 2).
        save_anim : bool, optional
            Whether to save the animation (default: False).
        filename : str, optional
            Name of the file to save the animation (default: None). If saving,
            a default name will be generated based on the trochoid type if None.
        """
        if not isinstance(r, (int, float)) or r <= 0:
            raise ValueError("Radius 'r' must be a positive number.")
        if not isinstance(a, (int, float)) or a <= 0:
            raise ValueError("Parameter 'a' must be a positive number.")
        if not isinstance(b, (int, float)) or b <= 0:
            raise ValueError("Parameter 'b' must be a positive number.")
        # Note: Validation a < r and b > r is done in the animate method
        #       as it depends on the type being animated.

        self.r = float(r)
        self.a = float(a)
        self.b = float(b)
        self.num_revolutions = int(num_revolutions)
        self.save_anim = bool(save_anim)
        self.filename = filename

        # Placeholders for animation elements initialized in animate()
        self.fig = None
        self.ax = None
        self.path_line = None
        self.circle_patch = None
        self.point = None
        self.radius_line = None
        self.center_point = None
        self.anim = None
        self._trochoid_type = None
        self._distance_param = None # Will be r, a, or b depending on type
        self._color = None
        self._theta_vals = None
        self._total_frames = None
        self._interval_ms = 25

    def _trochoid_coordinates(self, theta, d):
        """
        Calculates the x and y coordinates of the trochoid point.
        Uses the standard parameterization:
        x = r*theta - d*sin(theta)
        y = r - d*cos(theta)

        Parameters:
        -----------
        theta : float or np.ndarray
            The angle(s) of rotation in radians.
        d : float
            The distance of the tracing point from the circle's center.

        Returns:
        --------
        tuple (np.ndarray, np.ndarray)
            x and y coordinates.
        """
        x = self.r * theta - d * np.sin(theta)
        y = self.r - d * np.cos(theta)
        return x, y

    def _init_animation(self):
        """Initializes the plot elements for the animation."""
        self.path_line.set_data([], [])
        self.circle_patch.center = (0, self.r)
        self.point.set_data([], [])
        self.radius_line.set_data([], [])
        self.center_point.set_data([], [])
        return [artist for artist in (self.path_line, self.circle_patch, self.point,
                                      self.radius_line, self.center_point) if artist is not None]
```

```python
    def _update_frame(self, frame):
        """Updates the plot elements for each animation frame."""
        # Calculate points up to the current frame
        theta_current = self._theta_vals[:frame + 1]
        x_trochoid, y_trochoid = self._trochoid_coordinates(theta_current, self._distance_param)

        # Current position of the tracing point
        current_x = x_trochoid[-1]
        current_y = y_trochoid[-1]

        # Current position of the circle center
        center_x = self.r * self._theta_vals[frame]
        center_y = self.r

        # Update plot elements
        self.path_line.set_data(x_trochoid, y_trochoid)
        self.circle_patch.center = (center_x, center_y)
        self.point.set_data([current_x], [current_y])
        self.radius_line.set_data([center_x, current_x], [center_y, current_y])
        self.center_point.set_data([center_x], [center_y])

        return [artist for artist in (self.path_line, self.circle_patch, self.point,
                                      self.radius_line, self.center_point) if artist is not None]

    def animate(self, trochoid_type='common'):
        """
        Generates and displays or saves the animation for a specified trochoid type.

        Parameters:
        -----------
        trochoid_type : str, optional
            The type of trochoid to animate. Must be one of 'common',
            'curtate', or 'prolate' (case-insensitive, default: 'common').

        Returns:
        --------
        matplotlib.animation.FuncAnimation
            The animation object.

        Raises:
        -------
        ValueError
            If trochoid_type is invalid or if parameters a/b do not meet the
            requirements for the selected type (a < r for curtate, b > r for prolate).
        """
        self._trochoid_type = trochoid_type.lower()
        valid_types = ['common', 'curtate', 'prolate']
        if self._trochoid_type not in valid_types:
            raise ValueError(f"trochoid_type must be one of {valid_types}")

        # --- Parameter Setup based on type ---
        if self._trochoid_type == 'common':
            self._distance_param = self.r
            self._color = 'lime'
            param_label = 'd=r'
        elif self._trochoid_type == 'curtate':
            if not self.a < self.r:
                raise ValueError(f"For curtate trochoid, 'a' ({self.a}) must be less than 'r' ({self.r}).")
            self._distance_param = self.a
            self._color = 'yellow'
            param_label = f'a={self.a:.2f}'
        else: # 'prolate'
            if not self.b > self.r:
                 raise ValueError(f"For prolate trochoid, 'b' ({self.b}) must be greater than 'r' ({self.r}).")
            self._distance_param = self.b
            self._color = 'cyan'
            param_label = f'b={self.b:.2f}'

        # --- Plot Setup ---
        plt.style.use('dark_background')
        self.fig, self.ax = plt.subplots(figsize=(10, 4))
        plt.subplots_adjust(left=0.08, right=0.97, top=0.92, bottom=0.1)

        # Set title
        title = (f"{self._trochoid_type.capitalize()} Cycloid ({param_label})\n"
                 f"Radius r={self.r:.2f}, Revolutions={self.num_revolutions}")
        self.ax.set_title(title, fontsize=14, pad=15)

        # --- Animation Parameters ---
```

```python
        frames_per_revolution = 100
        self._total_frames = int(self.num_revolutions * frames_per_revolution)

        theta_max = self.num_revolutions * 2 * np.pi
        self._theta_vals = np.linspace(0, theta_max, self._total_frames)

        # --- Calculate Plot Limits ---
        # y coordinates range from r - d to r + d
        y_min_coord = abs(self.r - self._distance_param)  # Distance from baseline to lowest point
        y_max_coord = self.r + self._distance_param       # Distance from baseline to highest point
        y_padding_up = 1.5 * max(self.r, self._distance_param)
        y_padding_down = 0.25 * self.r
        x_max_coord = self.r * theta_max + max(self.r, self._distance_param)
        x_min_coord = max(self.r, self._distance_param) # Start from -r or -d

        self.ax.set_xlim(-x_min_coord, x_max_coord)
        self.ax.set_ylim(-y_min_coord - y_padding_down, y_max_coord + y_padding_up)
        self.ax.set_aspect('equal', adjustable='box')

        # --- Axis Properties ---
        xticks = np.linspace(0, theta_max * self.r, int(2 * self.num_revolutions + 1))
        xtick_labels = []
        for x in xticks:
            val = x / (np.pi * self.r)
            if np.isclose(val, 0):
                xtick_labels.append('$0$')
            elif np.isclose(val, 1):
                 xtick_labels.append(r"$\pi r$")
            elif np.isclose(val % 1, 0):
                 xtick_labels.append(f"${int(val)}\\pi r$")
            else:
                 xtick_labels.append(f"${val:.1f}\\pi r$") # Fallback for non-integers

        self.ax.set_xticks(xticks)
        self.ax.set_xticklabels(xtick_labels)
        self.ax.set_xlabel("Distance Rolled (proportional to $\\theta$)")
        self.ax.set_ylabel("Y (in units of r)")
        self.ax.grid(False)

        # Base Line
        self.ax.axhline(0, color='gray', lw=1.5, ls='--', alpha=0.6)
        # Vertical  Line
        self.ax.axvline(0, color='gray', lw=1.5, ls='--', alpha=0.6)

        # --- Initialize Plot Elements ---
        self.path_line, = self.ax.plot([], [], color=self._color, lw=2, label='Trochoid Path')
        self.circle_patch = patches.Circle((0, self.r), self.r, fill=False, color='white', lw=1.5, alpha=0.7)
        self.ax.add_patch(self.circle_patch)

        if self._trochoid_type != 'common':
             distance_marker = patches.Circle((0, self.r), self._distance_param, fill=False, color=self._color, l
             self.ax.add_patch(distance_marker)

        self.point, = self.ax.plot([], [], 'o', color=self._color, ms=8, label='Tracing Point')
        self.radius_line, = self.ax.plot([], [], '--', color=self._color, lw=1, alpha=0.8)
        self.center_point, = self.ax.plot([], [], 'o', color='white', ms=5, mec='black')

        self.ax.legend(loc='upper right', fontsize=8)

        # --- Create Animation ---
        self.anim = FuncAnimation(self.fig, self._update_frame, frames=self._total_frames,
                                  init_func=self._init_animation, interval=self._interval_ms,
                                  repeat=False, blit=False)

        # --- Save or Show Animation ---
        if self.save_anim:
            # Determine filename
            fname = self.filename
            if fname is None:
                 # Generate default filename if none provided
                 fname = f"{self._trochoid_type}_trochoid_r{self.r}_d{self._distance_param:.2f}".replace('.', '_
            elif not fname.lower().endswith(('.gif', '.mp4', '.mov')):
                 fname += ".gif" # Default to gif if extension missing


            save_dir = "ANIMATIONS/TROCHOIDS"
            os.makedirs(save_dir, exist_ok=True)
            full_path = os.path.join(save_dir, fname)

            print(f"Saving animation to {os.path.abspath(full_path)}...")
```

```python
            try:
                # Use pillow for GIF, ffmpeg needed for mp4/mov (ensure installed)
                writer = 'pillow' if full_path.lower().endswith('.gif') else 'ffmpeg'
                self.anim.save(full_path, writer=writer, fps=int(1000 / self._interval_ms), dpi=150)
                print("Animation saved successfully!")
            except Exception as e:
                print(f"Error saving animation: {e}")
                print("If saving as MP4/MOV, ensure ffmpeg is installed and in your system's PATH.")
                print("Showing animation instead...")
                plt.show() # Fallback to showing if save fails
            finally:
                 plt.close(self.fig) # Close the figure after saving or error

        else:
            plt.show() # Display the animation

        return self.anim



if __name__ == "__main__":
    # Example 1: Common Cycloid (d=r)
    try:
        trochoid_common = Trochoids(r=1.0, num_revolutions=2,
                                    save_anim=False
                                    )
        anim1 = trochoid_common.animate(trochoid_type='common') # Show
    except ValueError as e:
        print(f"Error creating common trochoid: {e}")

    # Example 2: Curtate Cycloid (a=0.5 < r=1.0), Save as GIF
    try:
        trochoid_curtate = Trochoids(r=1.0, a=0.5, num_revolutions=2,
                                     save_anim=True,
                                     filename="curtate_cycloid_animation.gif"
                                     )
        anim2 = trochoid_curtate.animate(trochoid_type='curtate') # Save
    except ValueError as e:
        print(f"Error creating curtate trochoid: {e}")

    # Example 3: Prolate Cycloid (b=1.5 > r=1.0), Save using default name
    try:
        trochoid_prolate = Trochoids(r=1.0, b=1.5, num_revolutions=3,
                                     save_anim=True               # No filename provided
                                     )
        anim3 = trochoid_prolate.animate(trochoid_type='prolate') # Save with default name
    except ValueError as e:
        print(f"Error creating prolate trochoid: {e}")

    # Example 4: Invalid parameters (a > r for curtate)
    try:
        trochoid_invalid = Trochoids(r=1.0, a=1.2)
        # anim4 = trochoid_invalid.animate(trochoid_type='curtate')
    except ValueError as e:
        print(f"\nSuccessfully caught expected error: {e}")

    # Example 5: Invalid type
    try:
        trochoid_invalid_type = Trochoids(r=1.0)
        # anim5 = trochoid_invalid_type.animate(trochoid_type='hyper')
    except ValueError as e:
        print(f"Successfully caught expected error: {e}")
```

This code below creates an interactive widget interface for visualizing cycloid curves using Python's `ipywidgets` library.

Main Components

1. **Interactive Controls**:
   - A dropdown menu to select cycloid type (normal, curtate, or prolate)
   - Sliders for adjusting parameters:
     - Radius (r)
     - Factor (a) for curtate cycloids
     - Factor (b) for prolate cycloids
     - Revolutions
   - Animation control buttons (Animate, Pause, Play)
2. **Key Functions**:

- `update_sliders()` : Enables/disables parameter sliders based on cycloid type
- `animate_cycloid()` : Handles animation creation and display
- `pause_animation()` : Stops the animation
- `play_animation()` : Resumes a paused animation

3. **Widget Layout**:

- Controls are organized vertically (parameters) and horizontally (buttons)
- Output area for displaying the animation

Usage Flow

1. User selects cycloid type from dropdown
2. Relevant sliders become active/inactive based on selection
3. User adjusts parameters using sliders
4. Animation can be started, paused, and resumed using control buttons

In [ ]:
```
def create_trochoid_widget():
    """Create interactive widgets for trochoid visualization using the Trochoids class."""

    state = {'trochoid_instance': None, 'animation': None, 'paused': False}

    # --- Create Widgets with Layout ---
    trochoid_type = widgets.Dropdown(
        options=['common', 'curtate', 'prolate'],
        value='common',
        description='Type:',
        style={'description_width': 'initial'},
        layout=widgets.Layout(width='200px')
    )

    r_slider = widgets.FloatSlider(
        value=1.0, min=0.1, max=3.0, step=0.1,
        description='Radius (r):',
        readout_format='.1f',
        continuous_update=False,
        style={'description_width': '90px'},
        layout=widgets.Layout(width='400px')
    )

    a_slider = widgets.FloatSlider(
        value=0.5, min=0.1, max=0.9, step=0.1,
        description='Distance (a<r):',
        readout_format='.1f',
        disabled=True,
        continuous_update=False,
        style={'description_width': '90px'},
        layout=widgets.Layout(width='400px')
    )

    b_slider = widgets.FloatSlider(
        value=1.5, min=1.1, max=4.0, step=0.1,
        description='Distance (b>r):',
        readout_format='.1f',
        disabled=True,
        continuous_update=False,
        style={'description_width': '90px'},
        layout=widgets.Layout(width='400px')
    )

    num_rev_slider = widgets.FloatSlider(
        value=2.0, min=0.5, max=5.0, step=0.1,
        description='Revolutions:',
        readout_format='.1f',
        continuous_update=False,
        style={'description_width': '90px'},
        layout=widgets.Layout(width='400px')
    )

    # Animation control buttons
    animate_button = widgets.Button(
        description='Create/Update Animation',
        button_style='success',
        layout=widgets.Layout(width='200px')
    )
```

```python
pause_button = widgets.Button(
    description='Pause',
    button_style='warning',
    disabled=True,
    layout=widgets.Layout(width='100px')
)

play_button = widgets.Button(
    description='Play',
    button_style='info',
    disabled=True,
    layout=widgets.Layout(width='100px')
)

# Output widget and message area
output = widgets.Output()
message_area = widgets.HTML(value="", layout=widgets.Layout(height='30px'))


# --- Callback Functions ---
def update_parameter_sliders(change):
    """Enable/disable a/b sliders and adjust limits based on type and r."""
    r_val = r_slider.value
    typ = trochoid_type.value

    # Enable/disable based on type
    a_slider.disabled = (typ != 'curtate')
    b_slider.disabled = (typ != 'prolate')

    # Adjust limits and values dynamically based on r
    # For 'a' (curtate): max must be < r
    a_slider.max = max(a_slider.min, r_val - a_slider.step) # Ensure max >= min
    if a_slider.value >= r_val:
        a_slider.value = a_slider.max # Clamp value if it becomes invalid

    # For 'b' (prolate): min must be > r
    b_slider.min = min(b_slider.max, r_val + b_slider.step) # Ensure min <= max
    if b_slider.value <= r_val:
        b_slider.value = b_slider.min # Clamp value if it becomes invalid

def handle_animation_button(b):
    """Create or update the animation based on current widget values."""
    with output:
        clear_output(wait=True)
    message_area.value = ""

    # Close existing plot figure if present
    if state['trochoid_instance'] and state['trochoid_instance'].fig:
        if plt.fignum_exists(state['trochoid_instance'].fig.number):
            plt.close(state['trochoid_instance'].fig)

    r = r_slider.value
    a = a_slider.value
    b = b_slider.value
    num_rev = num_rev_slider.value
    typ = trochoid_type.value

    try:
        state['trochoid_instance'] = Trochoids(r=r, a=a, b=b, num_revolutions=num_rev, save_anim=False)
        # Only store the animation object, don't display it again
        state['animation'] = state['trochoid_instance'].animate(trochoid_type=typ)
        state['paused'] = False
        pause_button.disabled = False
        play_button.disabled = True
        message_area.value = "<i style='color: green;'>Animation created/updated.</i>"
    except ValueError as e:
        message_area.value = f"<b style='color: red;'>Error:</b> {e}"
        state['animation'] = None
        pause_button.disabled = True
        play_button.disabled = True
    except Exception as e:
        message_area.value = f"<b style='color: red;'>Unexpected Error:</b> {e}"
        state['animation'] = None
        pause_button.disabled = True
        play_button.disabled = True


def pause_animation(b):
    """Handle pause button click."""
    if state['animation'] is not None and hasattr(state['animation'], 'event_source') and state['animation']
```

```
            state['animation'].event_source.stop()
            state['paused'] = True
            pause_button.disabled = True
            play_button.disabled = False
            message_area.value = "<i style='color: orange;'>Animation paused.</i>"


    def play_animation(b):
        """Handle play button click."""
        if state['animation'] is not None and hasattr(state['animation'], 'event_source') and state['animation']
            state['animation'].event_source.start()
            state['paused'] = False
            pause_button.disabled = False
            play_button.disabled = True
            message_area.value = "<i style='color: blue;'>Animation playing.</i>"

    # --- Connect Callbacks ---
    trochoid_type.observe(update_parameter_sliders, names='value')
    r_slider.observe(update_parameter_sliders, names='value')
    # Also trigger update when a/b change, although they are constrained by r
    a_slider.observe(update_parameter_sliders, names='value')
    b_slider.observe(update_parameter_sliders, names='value')


    animate_button.on_click(handle_animation_button)
    pause_button.on_click(pause_animation)
    play_button.on_click(play_animation)

    # --- Initialize Slider States ---
    # Call initially to set correct enabled/disabled states and limits
    update_parameter_sliders({'new': trochoid_type.value})

    # --- Layout Widgets ---
    parameter_box = widgets.VBox([
        trochoid_type,
        r_slider,
        a_slider,
        b_slider,
        num_rev_slider
    ], layout=widgets.Layout(margin='10px 0px'))

    button_box = widgets.HBox([animate_button, pause_button, play_button],
                              layout=widgets.Layout(margin='10px 0px'))

    controls = widgets.VBox([
        parameter_box,
        button_box,
        message_area
    ], layout=widgets.Layout(margin='10px'))

    # Display controls and output vertically
    display(widgets.VBox([controls, output]))

create_trochoid_widget()
```

Derivations

1. Arc Length of a Trochoid

The arc length of a trochoid, a curve traced by a point fixed to a circle rolling along a straight line, can be derived using parametric equations and the arc length formula. A trochoid is generically defined by the parametric equations:

$$x(\theta) = r\theta - d\sin(\theta)$$
$$y(\theta) = r - d\cos(\theta)$$

Here, r represents the radius of the rolling circle, d is the distance from the center of the circle to the tracing point, and θ is the angle of rotation of the circle. The shape of the trochoid depends on the relationship between r and d:

- If $d = r$, the curve is a common cycloid.
- If $d < r$, the curve is a curtate cycloid.
- If $d > r$, the curve is a prolate cycloid.

The arc length L of a parametric curve $x(\theta)$, $y(\theta)$ for θ ranging from θ_1 to θ_2 is given by the formula:

$$\boxed{L = \int_{\theta_1}^{\theta_2} \sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2}\, d\theta}$$

First, we find the derivatives of the parametric equations with respect to θ:

$$\frac{dx}{d\theta} = \frac{d}{d\theta}(r\theta - d\sin(\theta)) = r - d\cos(\theta)$$
$$\frac{dy}{d\theta} = \frac{d}{d\theta}(r - d\cos(\theta)) = d\sin(\theta)$$

Now, substitute these derivatives into the arc length formula:

$$\begin{aligned} L &= \int_{\theta_1}^{\theta_2} \sqrt{(r - d\cos(\theta))^2 + (d\sin(\theta))^2} d\theta \\ &= \int_0^{2\pi} \sqrt{r^2 - 2rd\cos(\theta) + d^2} d\theta \end{aligned}$$

For the general cases of prolate and curtate cycloids ($d \neq r$), this integral does not have a simple closed form and is typically expressed in terms of elliptic integrals.

However, for the special case of a **common cycloid** where $d = r$, the integral simplifies:

$$L = \int_0^{2\pi} \sqrt{r^2 - 2r^2\cos(\theta) + r^2} d\theta = \int_0^{2\pi} \sqrt{2r^2(1 - \cos(\theta))} d\theta$$

Using the half-angle identity $1 - \cos(\theta) = 2\sin^2\left(\frac{\theta}{2}\right)$:

$$\begin{aligned} L &= \int_0^{2\pi} \sqrt{2r^2\left(2\sin^2\left(\frac{\theta}{2}\right)\right)} d\theta \\ &= \int_0^{2\pi} \sqrt{4r^2\sin^2\left(\frac{\theta}{2}\right)} d\theta \\ &= \int_0^{2\pi} \left|2r\sin\left(\frac{\theta}{2}\right)\right| d\theta \end{aligned}$$

For $0 \leq \theta \leq 2\pi$, $0 \leq \frac{\theta}{2} \leq \pi$, so $\sin\left(\frac{\theta}{2}\right) \geq 0$. Thus, $|2r\sin(\frac{\theta}{2})| = 2r\sin(\frac{\theta}{2})$ (assuming $r > 0$).

$$L = \int_0^{2\pi} 2r\sin\left(\frac{\theta}{2}\right) d\theta$$

Now, we can evaluate this integral:

$$L = 2r\int_0^{2\pi} \sin\left(\frac{\theta}{2}\right) d\theta$$

Let $u = \frac{\theta}{2}$, so $du = \frac{1}{2}d\theta$, which means $d\theta = 2du$. When $\theta = 0$, $u = 0$. When $\theta = 2\pi$, $u = \pi$.

$$\begin{aligned} L &= 2r\int_0^{\pi} \sin(u)(2du) \\ &= 4r\int_0^{\pi} \sin(u)du \\ &= 4r[-\cos(u)]_0^{\pi} \\ &= 8r \end{aligned}$$

Thus, the arc length of one arch of a common cycloid is $8r$.

Hence, the arc length of a general trochoid for one arch is given by the integral $L = \int_0^{2\pi} \sqrt{r^2 - 2rd\cos(\theta) + d^2} d\theta$. For the specific case of a common cycloid ($d = r$), this integral evaluates to $8r$. For prolate and curtate cycloids, the arc length is expressed through the integral form, which is related to elliptic integrals.

2. Area Enclosed by a Trochoid

The area A under a parametric curve defined by $x(\theta)$ and $y(\theta)$ from $\theta = \theta_1$ to $\theta = \theta_2$ is given by the formula:

$$\boxed{A = \int_{\theta_1}^{\theta_2} y(\theta)\frac{dx}{d\theta} d\theta}$$

assuming the curve is traversed from left to right as θ increases. For one arch of the trochoid, we integrate from $\theta_1 = 0$ to $\theta_2 = 2\pi$.

First, we need to find the derivative of x with respect to θ:

$$\frac{dx}{d\theta} = \frac{d}{d\theta}(r\theta - d\sin(\theta)) = r - d\cos(\theta)$$

Now, substitute $y(\theta)$ and $\frac{dx}{d\theta}$ into the area formula:

$$\begin{aligned}
A &= \int_0^{2\pi} (r - d\cos(\theta))(r - d\cos(\theta))d\theta \\
&= \int_0^{2\pi} (r - d\cos(\theta))^2 d\theta \\
&= \int_0^{2\pi} (r^2 - 2rd\cos(\theta) + d^2\cos^2(\theta))d\theta \\
&= \int_0^{2\pi} r^2 d\theta - 2rd\int_0^{2\pi} \cos\theta d\theta + d^2\int_0^{2\pi} \cos^2(\theta)d\theta \\
&= 2\pi r^2 - 0 + d^2 \cdot \frac{2\pi}{2} \\
&= 2\pi r^2 + \pi d^2
\end{aligned}$$

Thus, the area enclosed by one arch of the trochoid and the x-axis is

$$\boxed{A = 2\pi r^2 + \pi d^2}$$

This formula gives the signed area. For a common cycloid ($d = r$), the area is $2\pi r^2 + \pi r^2 = 3\pi r^2$, which is a well-known result. For curtate cycloids ($d < r$), the curve lies entirely above the x-axis, and the area is positive. For prolate cycloids ($d > r$), the curve dips below the x-axis, and the integral calculates the net area (area above the x-axis minus the area below the x-axis). If the total unsigned area for a prolate cycloid is required, further calculation considering the loops below the x-axis would be necessary. However, the standard derivation provides the signed area as $2\pi r^2 + \pi d^2$.

Trochoids Comparison

Plotting the Trajectories

In [ ]:
```
def compare_cycloids(r=1.0, a=0.5, b=1.5, num_revolutions=2, save_fig=False, filename=None):
    """
    Plot three types of cycloids in a single figure for comparison:
    - Normal cycloid (a = r)
    - Curtate cycloid (a < r) using parameter 'a'
    - Prolate cycloid (b > r) using parameter 'b'

    Parameters:
    -----------
    r : float
        Radius of the rolling circle (default: 1.0)
    a : float
        Parameter for curtate cycloid, should be < r (default: 0.5)
    b : float
        Parameter for prolate cycloid, should be > r (default: 1.5)
    num_revolutions : int
        Number of complete revolutions (default: 2)
    save_fig : bool
        Whether to save the figure (default: False)
    filename : str, optional
        Name of the file to save the figure (default: None)
    """
    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=(10, 5))
    ax.set_title(f"Trochoid types\nRadius (r) = {r}, Revolutions = {num_revolutions}",
                 fontsize=16, pad=5)
    fig.subplots_adjust(left=0.1, right=0.95, top=0.9, bottom=0.1)

    # Setup parameters
    theta_max = num_revolutions * 2 * np.pi
    theta_vals = np.linspace(0, theta_max, 200)

    # Set x-axis ticks in multiples of pi*r
    xticks = np.linspace(0, theta_max * r, int(2 * num_revolutions + 1))
    xtick_labels = []
    for x in xticks:
        val = x / (np.pi * r)
        if np.isclose(val, 0):
            xtick_labels.append('$0$')
        elif np.isclose(val, 1):
                xtick_labels.append(r"$\pi r$")
```

```python
        elif np.isclose(val % 1, 0):
                xtick_labels.append(f"${int(val)}\\pi r$")
        else:
                xtick_labels.append(f"${val:.1f}\\pi r$") # Fallback for non-integers

    ax.set_xticks(xticks)
    ax.set_xticklabels(xtick_labels)

    def cycloid_equation(theta, factor):
        """Calculate cycloid points for given parameter"""
        x_cycloid = r * (theta - factor*np.sin(theta))
        y_cycloid = r * (1 - factor*np.cos(theta))
        return x_cycloid, y_cycloid

    # Define cycloid types with their parameters
    cycloids = [
        {'factor': r, 'color': 'red', 'label': 'Normal Cycloid ', 'zorder': 2},
        {'factor': a, 'color': 'yellow', 'label': f'Curtate Cycloid (a = {a:.2f})', 'zorder': 3},
        {'factor': b, 'color': 'cyan', 'label': f'Prolate Cycloid (b = {b:.2f})', 'zorder': 1}
    ]

    # Calculate plot limits
    y_max = max([r * (1 + abs(c['factor'])) for c in cycloids])
    y_padding = 1.5 * r
    ax.set_xlim(-r, r * theta_max + r)
    ax.set_ylim(-y_padding, y_max + y_padding)
    ax.set_yticks(np.arange(-y_padding, y_max + y_padding))
    ax.set_yticklabels([f'{val:.1f}' for val in np.arange(-y_padding, y_max + y_padding)])
    ax.set_aspect('equal', adjustable='box')

    # Add labels and grid
    ax.set_xlabel("Distance Rolled (proportional to $\\theta$)")
    ax.set_ylabel("Y (in units of r)")
    ax.grid(False)

    # Base lines
    ax.axhline(0, color='gray', linewidth=2, alpha=0.5)
    ax.axvline(0, color='gray', linewidth=2, alpha=0.5)

    # Plot each cycloid
    for cycloid in cycloids:
        factor = cycloid['factor']
        x_cycloid, y_cycloid = cycloid_equation(theta_vals, factor)

        # Plot cycloid path
        ax.plot(x_cycloid, y_cycloid,
                color=cycloid['color'],
                linewidth=2,
                label=cycloid['label'],
                zorder=cycloid['zorder'])

        # Show circle and point at π position
        angle = np.pi
        x_center = r * angle
        y_center = r

        # Calculate point position at π
        x_point, y_point = cycloid_equation(angle, factor)

        # Plot circle
        circle = patches.Circle((x_center, r), r,
                                fill=False,
                                color='green',
                                lw=1.5,
                                alpha=0.5)
        ax.add_patch(circle)

        # Plot connecting line and point
        ax.plot([x_center, x_point], [y_center, y_point],
                color=cycloid['color'],
                lw=1,
                zorder=cycloid['zorder'],)
        ax.plot(x_point, y_point, 'o',
                color='white',
                ms=8,
                mec=cycloid['color'],
                zorder=5)
        ax.plot(x_center, y_center, 'o',
                color='white',
                ms=8,
```

```
                mec='black',
                zorder=5)

    # Add legend
    ax.legend(loc='upper right', fontsize=8)

    if save_fig and filename is not None:
        # Save the figure
        save_dir = "IMAGES/CYCLOIDS"
        os.makedirs(save_dir, exist_ok=True)
        filename = os.path.join(save_dir, filename)
        print(f"Saving figure to {os.path.abspath(filename)}...")
        plt.savefig(filename, bbox_inches='tight', dpi=300)
        print(f"Figure saved successfully!")
        plt.close(fig)
    else:
        plt.show()


compare_cycloids(r=1.0, a=0.5, b=1.5, num_revolutions=2, save_fig=True, filename="cycloid_comparison.png")
```

Animating the Motion

In [ ]:
```
def animate_cycloids(r=1.0, a=0.5, b=1.5, num_revolutions=2, save_anim=False, filename=None):
    """
    Animate three types of cycloids being traced simultaneously:
    - Normal cycloid (a = r)
    - Curtate cycloid (a < r)
    - Prolate cycloid (b > r)
    """
    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=(10, 5))
    ax.set_title(f"Trochoid Types Animation\nRadius (r) = {r}, Revolutions = {num_revolutions}",
                 fontsize=16, pad=10)
    fig.subplots_adjust(left=0.1, right=0.95, top=0.9, bottom=0.1)

    # Animation parameters
    frames_per_revolution = 100
    total_frames = int(num_revolutions * frames_per_revolution)
    interval_ms = 25

    # Setup parameters
    theta_max = num_revolutions * 2 * np.pi
    theta_vals = np.linspace(0, theta_max, total_frames)

    # Set x-axis ticks in multiples of pi
    xticks = np.linspace(0, theta_max * r, int(2 * num_revolutions + 1))
    xtick_labels = []
    for x in xticks:
        val = x / (np.pi * r)
        if np.isclose(val, 0):
            xtick_labels.append('$0$')
        elif np.isclose(val, 1):
                xtick_labels.append(r"$\pi r$")
        elif np.isclose(val % 1, 0):
                xtick_labels.append(f"${int(val)}\\pi r$")
        else:
                xtick_labels.append(f"${val:.1f}\\pi r$") # Fallback for non-integers

    ax.set_xticks(xticks)
    ax.set_xticklabels(xtick_labels)

    def cycloid_equation(theta, factor):
        """Calculate cycloid points for given parameter"""
        x_cycloid = r * (theta - factor*np.sin(theta))
        y_cycloid = r * (1 - factor*np.cos(theta))
        return x_cycloid, y_cycloid

    # Define cycloid types with their parameters
    cycloids = [
        {'factor': r, 'color': 'red', 'label': 'Normal Cycloid', 'zorder': 2},
        {'factor': a, 'color': 'yellow', 'label': f'Curtate Cycloid (a={a:.2f})', 'zorder': 3},
        {'factor': b, 'color': 'cyan', 'label': f'Prolate Cycloid (b={b:.2f})', 'zorder': 1}
    ]

    # Calculate plot limits
    y_max = max([r * (1 + abs(c['factor'])) for c in cycloids])
    y_padding = 1.5 * r
    ax.set_xlim(-r, r * theta_max + r)
```

```
    ax.set_ylim(-y_padding, y_max + y_padding)
    ax.set_yticks(np.arange(-y_padding, y_max + y_padding))
    ax.set_yticklabels([f'{val:.1f}' for val in np.arange(-y_padding, y_max + y_padding)])
    ax.set_aspect('equal', adjustable='box')

    # Add labels and grid
    ax.set_xlabel("Distance Rolled (proportional to $\\theta$)")
    ax.set_ylabel("Y (in units of r)")
    ax.grid(False)

    # Base line
    ax.axhline(0, color='gray', linewidth=2, alpha=0.5)
    ax.axvline(0, color='gray', linewidth=2, alpha=0.5)

    # Initialize plot elements
    paths = []
    circles = []
    points = []
    lines = []
    circle_centers = []

    # Create initial plot elements for each cycloid
    for cycloid in cycloids:
        # Path line
        path, = ax.plot([], [], color=cycloid['color'], lw=2, label=cycloid['label'])
        paths.append(path)

        # Rolling circle
        circle = patches.Circle((0, r), r, fill=False, color='white', lw=2, alpha=0.5)
        ax.add_patch(circle)
        circles.append(circle)

        # Circle Center
        circle_center, = ax.plot([], [], 'o', color='white', ms=8, mec='black', zorder=3)
        circle_centers.append(circle_center)

        # Tracing point
        point, = ax.plot([], [], 'o', color=cycloid['color'], ms=8)
        points.append(point)

        # Radius line
        line, = ax.plot([], [], '--', color=cycloid['color'], lw=1, zorder=cycloid['zorder'])
        lines.append(line)

    ax.legend(loc='upper right', fontsize=8)
    def init():
        """Initialize animation"""
        for path, circle, point, line, circle_center in zip(paths, circles, points, lines, circle_centers):
            path.set_data([], [])
            circle.center = (0, r)
            circle_center.set_data([], [])
            point.set_data([], [])
            line.set_data([], [])
        return paths + circles + points + lines + circle_centers

    def animate(frame):
        """Update animation for each frame"""
        for cycloid, path, circle, point, line, circle_center in zip(cycloids, paths, circles, points, lines, ci
            # Calculate current cycloid points
            factor = cycloid['factor']
            theta = theta_vals[:frame+1]
            x_cycloid, y_cycloid = cycloid_equation(theta, factor)

            # Current position
            current_x = x_cycloid[-1]
            current_y = y_cycloid[-1]

            # Circle center
            center_x = r * theta_vals[frame]
            center_y = r

            # Update plot elements
            path.set_data(x_cycloid, y_cycloid)
            circle.center = (center_x, center_y)
            circle_center.set_data([center_x], [center_y])
            point.set_data([current_x], [current_y])
            line.set_data([center_x, current_x], [center_y, current_y])

        return paths + circles + points + lines + circle_centers
```

```python
    anim = FuncAnimation(fig, animate, frames=total_frames,
                                 init_func=init, interval=interval_ms,
                                 repeat=False,
                                 blit=False)

    if save_anim and filename is not None:
        # Save the animation
        save_dir = "ANIMATIONS/TROCHOIDS"
        os.makedirs(save_dir, exist_ok=True)
        filename = os.path.join(save_dir, filename)
        # Ensure the filename already exist
        if os.path.exists(filename):
            print(f"File {filename} already exists, overwriting...")
        # Save the animation using Pillow writer
        print("Saving animation... (this may take a while)")
        print(f"Animation will be saved to {os.path.abspath(filename)}")
        anim.save(filename, writer='pillow', fps=int(1000/interval_ms), dpi=200)
        print(f"Animation saved successfully!")
        plt.close(fig)
    else:
        plt.show()

    return anim


animation = animate_cycloids(r=1.0, a=0.5, b=1.5, num_revolutions=2,
                save_anim=True, filename="cycloids_animation.gif")
```

Epicycloids

Overview

An **epicycloid** is a type of plane curve generated by the motion of a point on the circumference of a circle (called the *rolling circle*) as it rolls without slipping around the *outside* of a fixed circle (called the *base circle*). Epicycloids are a special case of epitrochoids, where the tracing point lies exactly on the circumference of the rolling circle.

Mathematical Representation

Parametric Equations

The parametric equations for an epicycloid are given by:

$$x(\phi) = (a+b)\cos(\phi) - b\cos\left(\frac{a+b}{b}\phi\right)$$

$$y(\phi) = (a+b)\sin(\phi) - b\sin\left(\frac{a+b}{b}\phi\right)$$

Where:

- a is the radius of the fixed circle (base circle),
- b is the radius of the rolling circle,
- ϕ is the angle parameter.

Polar Form

The polar equation of an epicycloid can also be derived, but it is less commonly used compared to the parametric form. The polar equations can ve derived by computing

$$x^2 = (a+b)^2\cos^2\phi - 2b(a+b)\cos\phi\cos\left(\frac{a+b}{b}\phi\right) + b^2\cos^2\left(\frac{a+b}{b}\phi\right)$$

$$y^2 = (a+b)^2\sin^2\phi - 2b(a+b)\sin\phi\sin\left(\frac{a+b}{b}\phi\right) + b^2\sin^2\left(\frac{a+b}{b}\phi\right)$$

so

$$r^2 = x^2 + y^2 = (a+b)^2 + b^2 - 2b(a+b)\left\{\cos\left(\frac{a+b}{b}\phi\right)\cos\phi + \sin\left(\frac{a+b}{b}\phi\right)\sin\phi\right\}$$

But

$$\cos\alpha\,\cos\beta + \sin\alpha\,\sin\beta = \cos(\alpha-\beta)$$

so

$$r^2 = (a+b)^2 + b^2 - 2b(a+b)\cos\left(\frac{a+b}{b}\phi - \phi\right)$$
$$= (a+b)^2 + b^2 - 2b(a+b)\cos\left(\frac{a}{b}\phi\right)$$

Note that ϕ is the parameter here not the polar angle. The polar angle from the center is

$$\tan\theta = \frac{y}{x} = \frac{(a+b)\sin\phi - b\sin\left(\frac{a+b}{b}\phi\right)}{(a+b)\cos\phi - b\cos\left(\frac{a+b}{b}\phi\right)}$$

Special Cases

1. **Cardioid**: When `a = b`, the epicycloid becomes a cardioid, a heart-shaped curve.
2. **Nephroid**: When `a = 2b`, the epicycloid becomes a nephroid, a kidney-shaped curve.
3. **Ranunculoid**: When `a = 5b`, the epicycloid is called a ranunculoid.

Number of Cusps

The number of cusps (sharp points) in an epicycloid is determined by the ratio of the radii:

- If `a/b = n` (a rational number), the epicycloid will have `n` cusps.
- If `a/b` is irrational, the curve never closes and forms a dense pattern.

Arc Length and Area

1. **Arc Length**: The total length of one complete epicycloid curve is:

   $$L = 8b(a/b + 1)$$

   To know more about the Arc Length check out the derivation section
2. **Area Enclosed**: The area enclosed by the epicycloid is:

   $$A = \pi q b^2 (k+1)(k+2)$$

   where k = a/b is a rational number in the simplest form of p/q. For more details check out the derivation section.

Construction and Visualization

Geometric Construction

To construct an epicycloid:

1. Start with a fixed circle of radius `a`.
2. Roll a smaller circle of radius `b` around the outside of the fixed circle.
3. Track the path of a point on the circumference of the rolling circle.

Symmetry

Epicycloids exhibit rotational symmetry. The number of symmetry axes corresponds to the number of cusps.

Related Curves

1. **Hypocycloids**: Generated when the rolling circle rolls *inside* the fixed circle.
2. **Epitrochoids**: A generalization of epicycloids where the tracing point is not restricted to the circumference of the rolling circle.
3. **Cycloids**: Generated when a circle rolls along a straight line.

References

For more information, visit:

- MathWorld: Epicycloid
- Wikipedia: Epicycloid

Implementation in Python

The code below defines a Python function `plot_epicycloids` that visualizes an **epicycloid**, a type of curve traced by a point on the circumference of a circle rolling around another fixed circle. Here's a brief breakdown:

Function Purpose

The function generates a plot of an epicycloid with the following features:

1. **Stationary Circle**: A fixed circle with radius `a`.
2. **Rolling Circle**: A smaller circle with radius `b = a/n` that rolls around the stationary circle.
3. **Tracing Point**: A point on the rolling circle's circumference that traces the epicycloid path.
4. **Epicycloid Path**: The curve traced by the point as the rolling circle moves.

Key Parameters

- `a`: Radius of the stationary circle.
- `n`: Number of cusps (sharp points) in the epicycloid.
- `save_fig`: Whether to save the plot as an image.
- `filename`: Name of the file to save the plot.

Steps in the Code

1. **Setup**:
   - Calculate the radius of the rolling circle (`b = a/n`).
   - Define the range of angles (`theta_vals`) for plotting the curve.
2. **Plot Configuration**:
   - Use a dark background style.
   - Set up a square plot with equal aspect ratio and no axis ticks or grid.
3. **Epicycloid Equation**:
   - Define the parametric equations for the epicycloid:
     ```
     x = (a + b) * cos(theta) - b * cos((a/b + 1) * theta)
     y = (a + b) * sin(theta) - b * sin((a/b + 1) * theta)
     ```
4. **Plot Elements**:
   - **Stationary Circle**: Drawn using `patches.Circle`.
   - **Rolling Circle**: Positioned at an angle of 45° and drawn similarly.
   - **Tracing Point**: Highlighted in red, with an annotation (`P`) and a connecting line to the rolling circle's center.
   - **Epicycloid Path**: The full curve traced by the point, plotted in red.
5. **Save and Display**:
   - If `save_fig` is `True`, save the plot to the specified `filename` in an `IMAGES` directory.
   - Display the plot using `plt.show()`.

Example Output

For `a=1.0` and `n=4`, the function will generate an epicycloid with 4 cusps and save it as `epicycloid.png` if `save_fig=True`.

This function is useful for visualizing mathematical curves and understanding the geometry of epicycloids.

In [ ]:
```
def plot_epicycloids(a, n, save_fig=False, filename=None):
    """
    Plot epicycloids with given parameters.

    Parameters:
    -----------
    a : float
        Radius of the fixed circle
    n : int
        Number of cusps
    save_fig : bool
        Whether to save the figure (default: False)
    filename : str, optional
        Name of the file to save the figure (default: None)
    """
    # Set up parameters
```

```python
    padding = 0.5  # Padding for the plot limits
    theta_max = 2 * np.pi
    theta_vals = np.linspace(0, theta_max, 500)
    b = a/n

    # Set up the figure
    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.set_xlim(-(a + 2 * b + padding), a + 2 * b + padding)
    ax.set_ylim(-(a + 2 * b + padding), a + 2 * b + padding)
    ax.set_aspect('equal', adjustable='box')
    ax.set_xticks([])  # Hide x ticks
    ax.set_yticks([])  # Hide y ticks
    ax.grid(False)  # Hide grid lines
    ax.set_title(f"Epicycloid with {n} cusps", fontsize=16, pad=10)
    plt.subplots_adjust(left=0.08, right=0.92, top=0.98, bottom=0.02)

    # Epicycloid parametric equations
    def epicycloid_equation(theta, a, b):
        x = (a + b) * np.cos(theta) - b * np.cos((a/b + 1) * theta)
        y = (a + b) * np.sin(theta) - b * np.sin((a/b + 1) * theta)
        return x, y

    # Plot the stationary circle
    circle_fixed = patches.Circle((0, 0), a, fill=False, color='gray', lw=2,
                                  label=f'Stationary Circle(a={a})')
    ax.add_patch(circle_fixed)

    # Angle at which to plot the rolling circle
    angle = np.pi / 4  # 45 degrees

    # Calculate the center of the rolling circle
    center_x = (a + b) * np.cos(angle)
    center_y = (a + b) * np.sin(angle)

    # Calculate the tracing point
    tracing_x, tracing_y = epicycloid_equation(angle, a, b)

    # Plot the rolling circle
    circle_rolling = patches.Circle((center_x, center_y), b, fill=False, color='blue', lw=2,
                                    label=f'Rolling Circle(b={b:.2f})', zorder=3)
    ax.add_patch(circle_rolling)

    # Draw connecting line
    ax.plot([center_x, tracing_x], [center_y, tracing_y], color='white', linestyle='--', lw=1.5)

    # Plot tracing point
    ax.scatter(tracing_x, tracing_y, c='red', s=80, label='Tracing Point(P)', zorder=4)

    # Calculate label position to avoid overlap
    # Use angle to determine label position relative to tracing point
    label_angle = angle - np.pi/4 # Offset by 45 degrees
    label_radius = 0.3  # Distance from point to label
    label_x = tracing_x + label_radius * np.cos(label_angle)
    label_y = tracing_y + label_radius * np.sin(label_angle)

    # Add label with a line connecting to the point
    ax.annotate('P',
                xy=(tracing_x, tracing_y),
                xytext=(label_x, label_y),
                color='white',
                fontsize=10,
                ha='center',
                va='center',
                bbox=dict(facecolor='black', edgecolor='white', alpha=0.7),
                arrowprops=dict(arrowstyle='->', color='white', alpha=0.7))

    # Plot center of rolling circle
    ax.scatter(center_x, center_y, c='white', s=80)

    # Plot epicycloid path
    x_epicycloid, y_epicycloid = epicycloid_equation(theta_vals, a, b)
    ax.plot(x_epicycloid, y_epicycloid, color='red', lw=2, label='Epicycloid Path')

    # Add legend
    ax.legend(loc='upper right', fontsize=8)

    # Save figure if requested
    if save_fig and filename is not None:
        save_dir = "IMAGES"
```

```
        os.makedirs(save_dir, exist_ok=True)
        filename = os.path.join(save_dir, filename)
        print(f"Saving figure to {os.path.abspath(filename)}...")
        plt.savefig(filename, bbox_inches='tight', dpi=300)
        print(f"Figure saved successfully!")

    plt.show()

plot_epicycloids(a=1.0, n=4, save_fig=True, filename="epicycloid.png")
```

Animating the motion

This code below generates an animation of an **epicycloid**, a curve traced by a point on the circumference of a smaller circle rolling around a larger fixed circle.

1. **Function Purpose**:
   The `animate_epicycloid` function creates an animated visualization of the epicycloid. It allows customization of the radii of the circles (`a` for the fixed circle, `b` for the rolling circle), the number of revolutions (`n`), and whether to save the animation as a file.

2. **Key Parameters**:
   - `a` : Radius of the fixed circle.
   - `n` : Number of revolutions of the rolling circle.
   - `b` : Calculated as `a / n` , the radius of the rolling circle.
   - `theta_vals` : Angles used to compute the positions of the rolling circle and the tracing point.
3. **Animation Setup**:
   - A **matplotlib figure** is created with a dark background.
   - The fixed circle, rolling circle, and the epicycloid path are drawn using `matplotlib.patches` and `ax.plot` .
   - The `init` function initializes the animation elements (e.g., empty path, circles, and points).
4. **Epicycloid Calculation**:
   - The `epicycloid_points` function computes:
     - The center of the rolling circle (`center_x` , `center_y`).
     - The tracing point coordinates (`trace_x` , `trace_y`) using parametric equations of the epicycloid.
5. **Animation Logic**:
   - The `animate` function updates the positions of the rolling circle, tracing point, and epicycloid path for each frame based on the current angle (`theta`).
6. **Saving or Displaying**:
   - If `save_anim` is `True` , the animation is saved as a `.gif` using the `pillow` writer.
   - Otherwise, the animation is displayed interactively using `plt.show()` .

This code uses **matplotlib's** `FuncAnimation` to smoothly animate the rolling motion and the traced epicycloid path.

In [ ]:
```
def animate_epicycloid(a=2, n=2, save_anim=False, filename=None):
    """
    Animate an epicycloid formed by a small circle rolling around a larger circle.

    Parameters:
    -----------
    a : int
        Radius of the fixed circle (default: 2)
    n : int
        Number of revolutions of the rolling circle (default: 2)
    num_frames : int
        Number of frames in the animation (default: 200)
    save_anim : bool
        Whether to save the animation (default: False)
    filename : str
        Name of the file to save the animation (optional)
    """
    # Set up parameters
    b = a/n  # Radius of the rolling circle
    padding = 0.5  # Padding for the plot limits

    # Animation parameters
    frames_per_revolution = 100
    total_frames = int(n * frames_per_revolution)
    interval_ms = 20
```

```python
# Calculate total angle based on number of revolutions needed
theta_max = 2 * np.pi
theta_vals = np.linspace(0, theta_max, total_frames)

# Set up the figure
plt.style.use('dark_background')
fig, ax = plt.subplots(figsize=(10, 10))
ax.set_xlim(-(a + 2*b + padding), a + 2*b + padding)
ax.set_ylim(-(a + 2*b + padding), a + 2*b + padding)
ax.set_xticks([])
ax.set_yticks([])
ax.set_aspect('equal', adjustable='box')
ax.grid(False)
ax.set_title(f"Epicycloid with {n} " + ("cusp" if n==1 else "cusps"),
             fontsize=14, pad=10)

# Create plot elements
# Stationary circle
circle_fixed = patches.Circle((0, 0), a, fill=False, color='gray', lw=2,
                              label=f'Stationary Circle (a={a})')
ax.add_patch(circle_fixed)

# Rolling circle (will be updated)
circle_rolling = patches.Circle((0, 0), b, fill=False, color='blue', lw=2,
                                label=f'Rolling Circle (b={b:.2f})', zorder=3)
ax.add_patch(circle_rolling)

# Path line (will be updated)
path_line, = ax.plot([], [], 'r-', lw=2, label='Epicycloid Path')

# Radius line (will be updated)
radius_line, = ax.plot([], [], 'w--', lw=1.5)

# Points (will be updated)
tracing_point, = ax.plot([], [], 'ro', ms=8, label='Tracing Point')
center_point, = ax.plot([], [], 'wo', ms=8)

# Add legend
ax.legend(loc='upper right', fontsize=8)

def init():
    """Initialize animation"""
    path_line.set_data([], [])
    radius_line.set_data([], [])
    tracing_point.set_data([], [])
    center_point.set_data([], [])
    circle_rolling.center = (a + b, 0)
    return path_line, circle_rolling, radius_line, tracing_point, center_point

def epicycloid_points(theta):
    """Calculate epicycloid points"""
    # Center of rolling circle
    center_x = (a + b) * np.cos(theta)
    center_y = (a + b) * np.sin(theta)

    # Tracing point
    trace_x = (a + b) * np.cos(theta) - b * np.cos((a/b + 1) * theta)
    trace_y = (a + b) * np.sin(theta) - b * np.sin((a/b + 1) * theta)

    return center_x, center_y, trace_x, trace_y

def animate(frame):
    """Update animation for each frame"""
    # Get current angle
    theta = theta_vals[:frame+1]

    # Calculate all points up to current frame
    centers_x, centers_y, traces_x, traces_y = [], [], [], []
    for t in theta:
        cx, cy, tx, ty = epicycloid_points(t)
        centers_x.append(cx)
        centers_y.append(cy)
        traces_x.append(tx)
        traces_y.append(ty)

    # Update rolling circle position
    current_center_x, current_center_y = centers_x[-1], centers_y[-1]
    circle_rolling.center = (current_center_x, current_center_y)
```

```python
        # Update path line
        path_line.set_data(traces_x, traces_y)

        # Update points
        current_trace_x, current_trace_y = traces_x[-1], traces_y[-1]
        tracing_point.set_data([current_trace_x], [current_trace_y])
        center_point.set_data([current_center_x], [current_center_y])

        # Update radius line
        radius_line.set_data([current_center_x, current_trace_x],
                             [current_center_y, current_trace_y])

        return path_line, circle_rolling, radius_line, tracing_point, center_point

    # Create animation
    anim = FuncAnimation(fig, animate, frames=total_frames,
                         init_func=init, interval=interval_ms,
                         blit=False, repeat=False)

    # Save animation if requested
    if save_anim:
        save_dir = "ANIMATIONS/EPICYCLOIDS"
        os.makedirs(save_dir, exist_ok=True)
        # Default filename if not provided
        if filename is None:
            filename = f"epicycloid_{n}_cusps.gif"

        filepath = os.path.join(save_dir, filename)
        print("Saving animation... (this may take a while)")
        print(f"Saving animation to {os.path.abspath(filepath)}...")
        anim.save(filepath, writer='pillow', fps=int(1000/interval_ms), dpi=100)
        print("Animation saved successfully!")
        plt.close(fig)
    else:
        plt.show()

    return anim


anim = animate_epicycloid(a=2, n=2, save_anim=True,
                          filename="Nephroid.gif")
```

Derivations

1. Arc Length of Epicycloids

An epicycloid is a plane curve produced by tracing the path of a point on a circle (the epicycle) that rolls without slipping around the outside of a fixed circle. The parametric equations for an epicycloid generated by a fixed circle of radius R and a rolling circle of radius r are given by:

$$x(\theta) = (R+r)\cos\theta - r\cos\left(\frac{R+r}{r}\theta\right)$$

$$y(\theta) = (R+r)\sin\theta - r\sin\left(\frac{R+r}{r}\theta\right)$$

where θ is the angle the line segment from the center of the fixed circle to the center of the rolling circle makes with the positive x-axis.

The arc length L of a parametric curve $x = x(\theta)$ and $y = y(\theta)$ from $\theta = \alpha$ to $\theta = \beta$ is given by the formula:

$$L = \int_{\alpha}^{\beta} \sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2}\, d\theta$$

First, we find the derivatives of the parametric equations with respect to θ:

$$\frac{dx}{d\theta} = -(R+r)\sin\theta + r\left(\frac{R+r}{r}\right)\sin\left(\frac{R+r}{r}\theta\right) = -(R+r)\sin\theta + (R+r)\sin\left(\frac{R+r}{r}\theta\right)$$

$$\frac{dy}{d\theta} = (R+r)\cos\theta - r\left(\frac{R+r}{r}\right)\cos\left(\frac{R+r}{r}\theta\right) = (R+r)\cos\theta - (R+r)\cos\left(\frac{R+r}{r}\theta\right)$$

Now, we compute the square of these derivatives and add them:

$$\left(\frac{dx}{d\theta}\right)^2 = (R+r)^2\left(-\sin\theta + \sin\left(\frac{R+r}{r}\theta\right)\right)^2 = (R+r)^2\left(\sin^2\theta - 2\sin\theta\sin\left(\frac{R+r}{r}\theta\right) + \sin^2\left(\frac{R+r}{r}\theta\right)\right)$$

$$\left(\frac{dy}{d\theta}\right)^2 = (R+r)^2\left(\cos\theta - \cos\left(\frac{R+r}{r}\theta\right)\right)^2 = (R+r)^2\left(\cos^2\theta - 2\cos\theta\cos\left(\frac{R+r}{r}\theta\right) + \cos^2\left(\frac{R+r}{r}\theta\right)\right)$$

Summing these squares:

$$\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2$$
$$= (R+r)^2\left(\sin^2\theta + \cos^2\theta + \sin^2\left(\frac{R+r}{r}\theta\right) + \cos^2\left(\frac{R+r}{r}\theta\right) - 2\left(\sin\theta\sin\left(\frac{R+r}{r}\theta\right) + \cos\theta\cos\left(\frac{R+r}{r}\theta\right)\right)\right)$$

Using the identity $\sin^2 x + \cos^2 x = 1$ and the cosine subtraction formula $\cos(A-B) = \cos A\cos B + \sin A\sin B$:

$$\begin{aligned}\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2 &= (R+r)^2\left(1 + 1 - 2\cos\left(\frac{R+r}{r}\theta - \theta\right)\right)\\ &= (R+r)^2\left(2 - 2\cos\left(\frac{R}{r}\theta\right)\right)\\ &= 2(R+r)^2\left(1 - \cos\left(\frac{R}{r}\theta\right)\right)\end{aligned}$$

Using the half-angle identity $1 - \cos(2\alpha) = 2\sin^2(\alpha)$, with $2\alpha = \frac{R}{r}\theta$, so $\alpha = \frac{R}{2r}\theta$:

$$= 2(R+r)^2\left(2\sin^2\left(\frac{R}{2r}\theta\right)\right) = 4(R+r)^2\sin^2\left(\frac{R}{2r}\theta\right)$$

Now, take the square root:

$$\sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2} = \sqrt{4(R+r)^2\sin^2\left(\frac{R}{2r}\theta\right)} = 2|R+r|\left|\sin\left(\frac{R}{2r}\theta\right)\right|$$

Since R and r are radii, $R+r>0$, so $|R+r| = R+r$.

$$\sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2} = 2(R+r)\left|\sin\left(\frac{R}{2r}\theta\right)\right|$$

The epicycloid is a closed curve if and only if the ratio $\frac{R}{r}$ is a rational number. Let $\frac{R}{r} = \frac{p}{q}$, where p and q are coprime positive integers. The curve completes one full cycle when the parameter θ ranges from 0 to $2\pi q$.

To find the total arc length of the closed epicycloid, we integrate from 0 to $2\pi q$:

$$L = \int_0^{2\pi q} 2(R+r)\left|\sin\left(\frac{p}{2q}\theta\right)\right| d\theta$$

Let $u = \frac{p}{2q}\theta$. Then $du = \frac{p}{2q}d\theta$, so $d\theta = \frac{2q}{p}du$. When $\theta = 0$, $u = 0$. When $\theta = 2\pi q$, $u = \frac{p}{2q}(2\pi q) = p\pi$.

$$L = \int_0^{p\pi} 2(R+r)|\sin(u)|\frac{2q}{p}du = \frac{4q(R+r)}{p}\int_0^{p\pi}|\sin(u)|du$$

The integral $\int_0^{p\pi}|\sin(u)|du$ is the sum of the areas under the absolute value of the sine function over p periods of π. The area under $|\sin(u)|$ from 0 to π is $\int_0^{\pi}\sin(u)du = [-\cos(u)]_0^{\pi} = -\cos(\pi) - (-\cos(0)) = 1 - (-1) = 2$. So, $\int_0^{p\pi}|\sin(u)|du = p \times 2 = 2p$.

Substituting this back into the arc length formula:

$$L = \frac{4q(R+r)}{p}(2p) = 8q(R+r)$$

Therefore, the total arc length of an epicycloid where the ratio of the fixed circle radius R to the rolling circle radius r is a rational number $\frac{p}{q}$ in simplest form is $8q(R+r)$.

If $\frac{R}{r} = k$ is an integer, then $p = k$ and $q = 1$. The total arc length is $8(1)(R+r) = 8(R+r)$.

If the ratio $\frac{R}{r}$ is irrational, the epicycloid is not a closed curve and the concept of a total arc length for the entire curve as a single finite value does not apply.

Therefore, the total **Arc Length** of an Epicycloid is

$\boxed{8q(R+r) \text{ where } R/r = p/q \text{ in simplest form}}$.

2. Area Enclosed by an Epicycloid

The standard parametric equations for the epicycloid are:

$$x(\theta) = (R+r)\cos\theta - r\cos\left(\frac{R+r}{r}\theta\right)$$

$$y(\theta) = (R+r)\sin\theta - r\sin\left(\frac{R+r}{r}\theta\right)$$

Substituting $R = kr$:

$$\frac{R+r}{r} = \frac{kr+r}{r} = k+1$$

The equations become:

$$\begin{aligned} x(\theta) &= r(k+1)\cos\theta - r\cos((k+1)\theta) \\ y(\theta) &= r(k+1)\sin\theta - r\sin((k+1)\theta) \end{aligned}$$

The curve forms a closed loop (or set of loops) that closes when θ completes q rotations, i.e., θ goes from 0 to $2\pi q$, where $k = p/q$ with p, q being coprime integers. If k is an integer, $q = 1$, and the curve closes after $\theta = 2\pi$.

1. Area Formula using Green's Theorem

The area A enclosed by a simple closed curve C defined parametrically by $(x(\theta), y(\theta))$ for $\theta \in [a, b]$ is given by Green's Theorem:

$$\boxed{A = \frac{1}{2}\oint_C (x\,dy - y\,dx) = \frac{1}{2}\int_a^b \left(x(\theta)\frac{dy}{d\theta} - y(\theta)\frac{dx}{d\theta}\right) d\theta}$$

For the entire closed curve of the epicycloid, the integration interval for θ is from 0 to $2\pi q$.

2. Calculate Derivatives

First, find the derivatives $\frac{dx}{d\theta}$ and $\frac{dy}{d\theta}$:

$$\begin{aligned} \frac{dx}{d\theta} &= -r(k+1)\sin\theta - r(k+1)(-\sin((k+1)\theta)) \\ &= -r(k+1)\sin\theta + r(k+1)\sin((k+1)\theta) \\ &= r(k+1)[\sin((k+1)\theta) - \sin\theta] \end{aligned}$$

$$\begin{aligned} \frac{dy}{d\theta} &= r(k+1)\cos\theta - r(k+1)\cos((k+1)\theta) \\ &= r(k+1)[\cos\theta - \cos((k+1)\theta)] \end{aligned}$$

3. Calculate the Integrand: $x\frac{dy}{d\theta} - y\frac{dx}{d\theta}$

$$\begin{aligned} x\frac{dy}{d\theta} &= [r(k+1)\cos\theta - r\cos((k+1)\theta)] \cdot [r(k+1)(\cos\theta - \cos((k+1)\theta))] \\ &= r^2(k+1)[(k+1)\cos\theta - \cos((k+1)\theta)][\cos\theta - \cos((k+1)\theta)] \\ &= r^2(k+1)[(k+1)\cos^2\theta - (k+1)\cos\theta\cos((k+1)\theta) - \cos\theta\cos((k+1)\theta) + \cos^2((k+1)\theta)] \\ &= r^2(k+1)[(k+1)\cos^2\theta - (k+2)\cos\theta\cos((k+1)\theta) + \cos^2((k+1)\theta)] \end{aligned}$$

$$\begin{aligned} y\frac{dx}{d\theta} &= [r(k+1)\sin\theta - r\sin((k+1)\theta)] \cdot [r(k+1)(\sin((k+1)\theta) - \sin\theta)] \\ &= r^2(k+1)[(k+1)\sin\theta - \sin((k+1)\theta)][\sin((k+1)\theta) - \sin\theta] \\ &= r^2(k+1)[(k+1)\sin\theta\sin((k+1)\theta) - (k+1)\sin^2\theta - \sin^2((k+1)\theta) + \sin\theta\sin((k+1)\theta)] \\ &= r^2(k+1)[(k+2)\sin\theta\sin((k+1)\theta) - (k+1)\sin^2\theta - \sin^2((k+1)\theta)] \end{aligned}$$

Now subtract $y\frac{dx}{d\theta}$ from $x\frac{dy}{d\theta}$:

$$\begin{aligned} & x\frac{dy}{d\theta} - y\frac{dx}{d\theta} \\ &= r^2(k+1)[((k+1)\cos^2\theta - (k+2)\cos\theta\cos((k+1)\theta) + \cos^2((k+1)\theta)) - ((k+2)\sin\theta\sin((k+1)\theta) - (k+1)\sin^2\theta - \sin^2((k+1)\theta))] \\ &= r^2(k+1)[(k+1)(\cos^2\theta + \sin^2\theta) + (\cos^2((k+1)\theta) + \sin^2((k+1)\theta)) - (k+2)(\cos\theta\cos((k+1)\theta) + \sin\theta\sin((k+1)\theta))] \end{aligned}$$

Using identities $\cos^2 A + \sin^2 A = 1$ and $\cos(A - B) = \cos A\cos B + \sin A\sin B$:

$$
\begin{aligned}
&= r^2(k+1)[(k+1)(1)+(1)-(k+2)\cos(\theta-(k+1)\theta)] \\
&= r^2(k+1)[k+1+1-(k+2)\cos(-k\theta)] \\
&= r^2(k+1)[k+2-(k+2)\cos(k\theta)] \\
&= r^2(k+1)(k+2)[1-\cos(k\theta)]
\end{aligned}
$$

4. Integrate to Find the Area

The integration interval is $\theta \in [0, 2\pi q]$, where $k = p/q$ in lowest terms.

$$
\begin{aligned}
A &= \frac{1}{2}\int_0^{2\pi q} r^2(k+1)(k+2)[1-\cos(k\theta)]\,d\theta \\
&= \frac{r^2(k+1)(k+2)}{2}\int_0^{2\pi q}(1-\cos(k\theta))\,d\theta \\
&= \frac{r^2(k+1)(k+2)}{2}\left[\theta - \frac{1}{k}\sin(k\theta)\right]_0^{2\pi q}
\end{aligned}
$$

Since $k = p/q$, $k \cdot 2\pi q = (p/q)\cdot 2\pi q = 2\pi p$. Since p is an integer, $\sin(2\pi p) = 0$ and $\sin(0) = 0$.

$$
\begin{aligned}
A &= \frac{r^2(k+1)(k+2)}{2}[(2\pi q - 0) - (0-0)] \\
&= \pi q r^2(k+1)(k+2)
\end{aligned}
$$

5. Result and Interpretation

The total area enclosed by the epicycloid traced over $\theta \in [0, 2\pi q]$ (where $k = R/r = p/q$ in lowest terms) is:

$$\boxed{A = \pi q r^2(k+1)(k+2)}$$

We can express this in terms of R and r:

$$
\begin{aligned}
A &= \pi q r^2\left(\frac{R}{r}+1\right)\left(\frac{R}{r}+2\right) \\
&= \pi q r^2 \frac{(R+r)}{r}\frac{(R+2r)}{r} \\
&= \pi q(R+r)(R+2r)
\end{aligned}
$$

Special Case: Integer k If $k = R/r$ is an integer, then $q = 1$, and the formula simplifies to:

$$\boxed{A = \pi r^2(k+1)(k+2)}$$

or

$$A = \pi(R+r)(R+2r)$$

- **Example: Cardioid (k=1)** $R = r$, $k = 1$ (integer, so $q = 1$).

  $$A = \pi(1)r^2(1+1)(1+2) = \pi r^2(2)(3) = 6\pi r^2$$

  Using the R, r formula:

  $$A = \pi(1)(r+r)(r+2r) = \pi(2r)(3r) = 6\pi r^2$$

- **Example: Nephroid (k=2)** $R = 2r$, $k = 2$ (integer, so $q = 1$).

  $$A = \pi(1)r^2(2+1)(2+2) = \pi r^2(3)(4) = 12\pi r^2$$

  Using the R, r formula:

  $$A = \pi(1)(2r+r)(2r+2r) = \pi(3r)(4r) = 12\pi r^2$$

The derived formula $A = \pi q(R+r)(R+2r)$ gives the net area enclosed by the epicycloid when $k = R/r = p/q$. For integer values of $k \geq 1$, this corresponds to the geometric area of the region bounded by the curve.

Hypocycloids

Overview

A **hypocycloid** is a type of plane curve generated by tracing a fixed point on the circumference of a circle (called the *rolling circle*) as it rolls **without slipping inside** a larger fixed circle (called the *base circle*). Hypocycloids are a special case of

hypotrochoids, where the tracing point lies exactly on the circumference of the rolling circle.

Mathematical Representation

Parametric Equations

The parametric equations for a hypocycloid are given by:

$$x(\phi) = (a-b)\cos(\phi) + b\cos\left(\frac{a-b}{b}\phi\right)$$

$$y(\phi) = (a-b)\sin(\phi) - b\sin\left(\frac{a-b}{b}\phi\right)$$

Where:

- a is the radius of the fixed circle (base circle),
- b is the radius of the rolling circle,
- ϕ is the angle parameter (not the polar angle).

Polar Form

Although the polar form of the hypocycloid is not as commonly used, it can be derived from the parametric form. First, calculate:

$$x^2 = (a-b)^2\cos^2\phi + 2b(a-b)\cos\phi\cos\left(\frac{a-b}{b}\phi\right) + b^2\cos^2\left(\frac{a-b}{b}\phi\right)$$

$$y^2 = (a-b)^2\sin^2\phi - 2b(a-b)\sin\phi\sin\left(\frac{a-b}{b}\phi\right) + b^2\sin^2\left(\frac{a-b}{b}\phi\right)$$

So:

$$r^2 = x^2 + y^2 = (a-b)^2 + b^2 + 2b(a-b)\cos\left(\frac{a-b}{b}\phi + \phi\right) = (a-b)^2 + b^2 + 2b(a-b)\cos\left(\frac{a}{b}\phi\right)$$

Here, ϕ is the curve parameter, not the polar angle. The polar angle $theta$ is given by:

$$\tan\theta = \frac{y}{x} = \frac{(a-b)\sin\phi - b\sin\left(\frac{a-b}{b}\phi\right)}{(a-b)\cos\phi + b\cos\left(\frac{a-b}{b}\phi\right)}$$

Special Cases

1. **Astroid**: When $a = 4b$, the hypocycloid is an astroid — a star-shaped curve with four cusps.
2. **Deltoid**: When $a = 3b$, the hypocycloid is a deltoid, a curve with three cusps.
3. **Line Segment**: When $a = 2b$, the hypocycloid degenerates into a straight line segment traced back and forth.

Number of Cusps

The number of cusps (sharp points) in a hypocycloid is determined by the ratio of the radii:

- If $a/b = n$, a rational number, then the hypocycloid has n cusps.
- If a/b is irrational, the curve never closes and densely fills the region inside the base circle.

Arc Length and Area

1. **Arc Length**: The total arc length L of one complete hypocycloid is:

   $$L = 8b\left(\frac{a}{b} - 1\right)$$

   If you want to know how the Arc Length is calculated check out the derivation section.
2. **Area Enclosed**: The area A enclosed by one complete hypocycloid is:

   $$A = (k-1)(k-2)\pi q b^2$$

   where k = a/b is a rational in the simplest form of p/q. To know more check out the derivation section.

Construction and Visualization

Geometric Construction

To construct a hypocycloid:

1. Begin with a fixed circle of radius a.
2. Roll a smaller circle of radius b along the *inside* of the fixed circle, without slipping.
3. Track the motion of a point on the circumference of the rolling circle.

Symmetry

Hypocycloids have rotational symmetry. The number of lines of symmetry equals the number of cusps in the curve.

Related Curves

1. **Epicycloids**: Generated by rolling a circle around the *outside* of a fixed circle.
2. **Hypotrochoids**: A generalization of hypocycloids, where the tracing point lies *inside or outside* the circumference of the rolling circle.
3. **Cycloids**: Generated when a circle rolls along a straight line.

References

For more information, visit:

- MathWorld: Hypocycloid
- Wikipedia: Hypocycloid

Plotting the Hypocycloid

Hypocycloids and the Role of `n`

A **hypocycloid** is the curve traced by a point on the circumference of a circle (called the rolling circle) as it rolls inside a fixed circle. The parameter `n` determines the ratio of the radii of the fixed circle (`a`) to the rolling circle (`b`), i.e., `n = a / b`. The value of `n` has a significant impact on the shape and behavior of the hypocycloid:

1. Integer `n` (e.g., `n = 3, 4, 5, ...`)

- **Behavior**: When `n` is an integer, the rolling circle completes exactly one revolution inside the fixed circle, and the curve closes on itself. The resulting hypocycloid has `n` **cusps** (sharp corners).
- **Why One Revolution?**:
  - For integer `n`, the rolling circle's circumference is an exact multiple of the fixed circle's circumference. This means the rolling circle aligns perfectly with its starting position after one complete revolution.
  - Mathematically, the angle of rotation of the rolling circle relative to the fixed circle is a multiple of `2π`, ensuring closure.
- **Examples**:
  - `n = 3`: A **deltoid** (3 cusps).
  - `n = 4`: An **astroid** (4 cusps).
- **Code Behavior**:
  - The code identifies integer `n` by checking if `n` is an `int` or a `Fraction` with a denominator of 1.
  - It sets `theta_revolutions = 1`, meaning the curve is plotted for one complete revolution of the rolling circle.

2. Rational `n` (e.g., `n = 5/3, 7/4, ...`)

- **Behavior**: When `n` is a rational number (a fraction), the rolling circle completes multiple revolutions before the curve closes on itself. The number of cusps is equal to the numerator of the fraction (`p`), and the curve closes after `q` revolutions, where `n = p / q`.
- **Why More Than One Revolution?**:
  - For rational `n`, the rolling circle's circumference is not an exact multiple of the fixed circle's circumference. It takes `q` revolutions for the rolling circle to align with its starting position.
  - Mathematically, the angle of rotation of the rolling circle relative to the fixed circle is a rational multiple of `2π`, ensuring closure after `q` revolutions.
- **Examples**:
  - `n = 5/3`: The curve has 5 cusps and closes after 3 revolutions.
  - `n = 7/4`: The curve has 7 cusps and closes after 4 revolutions.

- **Code Behavior**:
  - The code converts `n` to a `Fraction` and extracts its numerator (`p`) and denominator (`q`).
  - It sets `theta_revolutions = q`, meaning the curve is plotted for `q` revolutions of the rolling circle.

3. Irrational `n` (e.g., `n = √2, π, e, ...`)

- **Behavior**: When `n` is an irrational number, the rolling circle never aligns with its starting position, and the curve never closes on itself. The cusps are infinite, and the curve appears as a dense, non-repeating pattern.
- **Why Infinite Revolutions?**:
  - For irrational `n`, the rolling circle's circumference is incommensurable with the fixed circle's circumference. This means the angle of rotation of the rolling circle relative to the fixed circle is an irrational multiple of `2π`, preventing closure.
  - The curve theoretically requires infinite revolutions to complete, but in practice, we limit the number of revolutions for visualization.
- **Examples**:
  - `n = √2`: The curve never closes and appears as a dense, intricate pattern.
  - `n = π`: Similar behavior, with infinite cusps.
- **Code Behavior**:
  - The code uses the `is_irrational` function to check if `n` is irrational.
  - It sets `theta_revolutions = max_revolutions_irrational`, limiting the number of revolutions to a finite value (default: 5 or user-specified).

Degenerate Cases

1. `n = 1`:
   - The rolling circle has the same radius as the fixed circle (`a = b`).
   - The tracing point remains stationary, resulting in a single point at `(a, 0)`.
   - The code handles this by returning a single point for the hypocycloid equation.
2. `n = 2`:
   - The rolling circle traces a straight line segment (a degenerate hypocycloid).
   - The code handles this as a special case of the hypocycloid equation.

Code Overview

The key parts of the code is discussed below:

- **`is_irrational(num)`:** This function tries to determine if a number is irrational. It's not a perfect test (due to the limitations of floating-point representation), but it's good enough for practical purposes in this context. It checks against known constants like `pi` and `e`, and also attempts to simplify the number to a fraction. If the denominator of the simplified fraction is large, it's treated as potentially irrational.
- **`plot_hypocycloid(...)`:** This is the main function that generates the hypocycloid plot.
  - It first validates the input parameters (`a`, `n`, etc.).
  - It then determines the type of `n` (integer, rational, or irrational) and sets the appropriate number of revolutions (`theta_revolutions`).
  - It calculates the `theta_vals` (angles) for plotting the curve. The number of points is adjusted based on the number of revolutions to maintain smoothness.
  - The `hypocycloid_equation` function calculates the x and y coordinates of the tracing point for a given angle `theta`.
  - The code then sets up the plot (axes, limits, title, etc.) using `matplotlib`.
  - It plots the fixed circle, the rolling circle (at a snapshot angle), the tracing point, and the hypocycloid path.
  - Finally, it saves the figure to a file (if `save_fig` is True) or displays it on the screen.
- **`hypocycloid_equation(theta, a_loc, b_loc, n_loc)`:**
  - This function calculates the x and y coordinates of a point on the hypocycloid for a given angle `theta`.
  - It takes the radii of the fixed circle (`a_loc`) and rolling circle (`b_loc`), and the ratio `n_loc` as input.
  - The equations used are the standard parametric equations for a hypocycloid.

Key Code Sections Explained

1. **Determining `n_type` and `theta_revolutions`:**

```
if isinstance(n, int) or (isinstance(n, Fraction) and n.denominator == 1):
    n_type = "integer"
    n = int(n) # Ensure integer type
    theta_revolutions = 1 # Closes in 1 revolution
    cusps = n
    print(f"Plotting Hypocycloid: Integer n={n} ({cusps} cusps)")
```

```
else: # Float or Fraction
    try:
        frac_n = Fraction(n).limit_denominator()
        p = frac_n.numerator
        q = frac_n.denominator

        if q == 1: # Simplified to integer
            n_type = "integer"
            n = p
            theta_revolutions = 1
            cusps = n
            print(f"Plotting Hypocycloid: Rational n simplified to integer n={n} ({cusps}
cusps)")
        elif is_irrational(n_float): # Irrational or complex rational
            n_type = "irrational"
            theta_revolutions = max_revolutions_irrational
            print(f"Plotting Hypocycloid: Irrational n ≈ {n_float:.4f} (plotting
{theta_revolutions} center revolutions)")
            cusps = 'Infinite'
        else: # Non-integer rational
            n_type = "rational"
            theta_revolutions = q # Closes after q revolutions
            cusps = p
            print(f"Plotting Hypocycloid: Rational n = {p}/{q} ({cusps} cusps, plotting {q}
center revolutions)")

    except (ValueError, OverflowError):
        n_type = "irrational" # Treat errors as irrational case
        theta_revolutions = max_revolutions_irrational
        print(f"Plotting Hypocycloid: Could not represent n={n_float} as simple fraction.
Treating as irrational (plotting {theta_revolutions} center revolutions)")
        cusps = 'Infinite'
```

This section is crucial. It checks the type of `n` and sets `n_type` and `theta_revolutions` accordingly. It handles integers, rationals (by simplifying to a fraction), and irrationals (using the `is_irrational` function). Error handling is included to treat cases where `n` cannot be represented as a simple fraction as irrational.

2. **Hypocycloid Equation:**

```
def hypocycloid_equation(theta, a_loc, b_loc, n_loc):
    """Calculate hypocycloid points (h=b case)."""
    if float(n_loc) == 1: # Degenerate case b=a
        # Return arrays matching theta shape if theta is array
        if isinstance(theta, np.ndarray):
            return np.full_like(theta, a_loc), np.zeros_like(theta)
        else:
            return np.array([a_loc]), np.array([0.0]) # Single point as array

    center_x = (a_loc - b_loc) * np.cos(theta)
    center_y = (a_loc - b_loc) * np.sin(theta)
    trace_angle = (n_loc - 1) * theta
    trace_x = center_x + b_loc * np.cos(trace_angle)
    trace_y = center_y - b_loc * np.sin(trace_angle) # Standard formula sign
    return np.asarray(trace_x), np.asarray(trace_y)
```

This function implements the parametric equations for the hypocycloid. It calculates the x and y coordinates of the tracing point based on the angle `theta` and the radii `a_loc` and `b_loc`.

In [ ]:

```
def is_irrational(num):
    """Approximate check for irrational numbers like pi, sqrt(2), etc."""
    # This is tricky. Standard float representation makes true irrationals rational.
    # We can check against known constants or if it's not rational representable easily.
    if isinstance(num, (int, Fraction)):
        return False
    if num == math.pi or num == math.e:
         return True
    # Check if it might be a simple square root
    if num > 0:
        sqrt_num = math.sqrt(num)
        if sqrt_num == int(sqrt_num): # Check if it's a perfect square (rational root)
            return False # num is a perfect square, sqrt is rational
        # If num itself is an integer, and not a perfect square, its sqrt is irrational
        # But here num = n = a/b. This check is complex.
        # Let's rely on Fraction simplification. If it doesn't simplify well, treat as potentially irrational/com
    try:
        # If it simplifies to a fraction with large denominator, treat as complex/irrational for animation purpos
        f = Fraction(num).limit_denominator(10000) # Limit complexity
```

```
        # Allow reasonably complex fractions
        if f.denominator > 100: # Threshold for "complex enough to animate longer"
             # print(f"Treating n={num} as complex rational or irrational.")
             return True
    except (OverflowError, ValueError):
        # Could happen for very large numbers or non-finite
        return True # Treat as effectively irrational for animation loop
    return False # Assume rational otherwise
```

In [ ]:

```
def plot_hypocycloid(a=4, n=3.0, num_points=1000, max_revolutions_irrational=5,
                     angle_snapshot=np.pi/4, save_fig=False, filename=None):
    """
    Plot a static hypocycloid curve with given parameters.

    Handles integer n (cusps), rational n (intersecting closed curve),
    and irrational n (non-closing curve up to max_revolutions_irrational).

    Parameters:
    -----------
    a : float or int
        Radius of the fixed circle (default: 4). Must be positive.
    n : float, int, or Fraction
        Ratio a/b, determining the shape. Must be >= 1.
        (default: 3.0)
    num_points : int
        Number of points to calculate for plotting the curve (default: 1000).
    max_revolutions_irrational : float or int
        Number of 2*pi revolutions of the center to plot for irrational n.
        (default: 5)
    angle_snapshot : float
        Angle (in radians) at which to display the rolling circle snapshot.
        (default: pi/4)
    save_fig : bool
        Whether to save the figure (default: False).
    filename : str, optional
        Name of the file to save the figure (required if save_fig is True).
        Default filenames are generated if None.
    """
    # --- Parameter Validation ---
    if not isinstance(a, (int, float)) or a <= 0:
        raise ValueError("Fixed circle radius 'a' must be a positive number.")
    # Allow n to be Fraction instance directly
    if not isinstance(n, (int, float, Fraction)) or float(n) < 1:
        raise ValueError("Ratio 'n = a/b' must be a number >= 1.")
    if float(n) == 1:
         warnings.warn("n=1 results in a degenerate hypocycloid (a point). Plot will show the point.")
         b = a # Set b=a for calculations
    else:
         b = a / float(n)  # Radius of the rolling circle

    padding = a * 0.1  # Padding for the plot limits

    # --- Determine Plotting Range (Theta) ---
    n_float = float(n) # Use float representation for calculations after type check
    n_type = "integer"
    theta_revolutions = 1 # Default for integer n
    cusps = 'N/A'
    p, q = None, None

    if isinstance(n, int) or (isinstance(n, Fraction) and n.denominator == 1):
        n_type = "integer"
        n = int(n) # Ensure integer type
        theta_revolutions = 1 # Closes in 1 revolution
        cusps = n
        print(f"Plotting Hypocycloid: Integer n={n} ({cusps} cusps)")
    else: # Float or Fraction
        try:
            frac_n = Fraction(n).limit_denominator()
            p = frac_n.numerator
            q = frac_n.denominator

            if q == 1: # Simplified to integer
                 n_type = "integer"
                 n = p
                 theta_revolutions = 1
                 cusps = n
                 print(f"Plotting Hypocycloid: Rational n simplified to integer n={n} ({cusps} cusps)")
            elif is_irrational(n_float): # Irrational or complex rational
                 n_type = "irrational"
                 theta_revolutions = max_revolutions_irrational
```

```
                print(f"Plotting Hypocycloid: Irrational n ≈ {n_float:.4f} (plotting {theta_revolutions} center
                cusps = 'Infinite'
            else: # Non-integer rational
                n_type = "rational"
                theta_revolutions = q # Closes after q revolutions
                cusps = p
                print(f"Plotting Hypocycloid: Rational n = {p}/{q} ({cusps} cusps, plotting {q} center revolutic

        except (ValueError, OverflowError):
            n_type = "irrational" # Treat errors as irrational case
            theta_revolutions = max_revolutions_irrational
            print(f"Plotting Hypocycloid: Could not represent n={n_float} as simple fraction. "
                  f"Treating as irrational (plotting {theta_revolutions} center revolutions)")
            cusps = 'Infinite'

    theta_max = 2 * np.pi * theta_revolutions
    # Increase num_points for longer plots to maintain smoothness
    adjusted_num_points = int(num_points * theta_revolutions)
    theta_vals = np.linspace(0, theta_max, adjusted_num_points)

    # --- Hypocycloid Equation ---
    def hypocycloid_equation(theta, a_loc, b_loc, n_loc):
        """Calculate hypocycloid points (h=b case)."""
        if float(n_loc) == 1: # Degenerate case b=a
            # Return arrays matching theta shape if theta is array
            if isinstance(theta, np.ndarray):
                return np.full_like(theta, a_loc), np.zeros_like(theta)
            else:
                return np.array([a_loc]), np.array([0.0]) # Single point as array

        center_x = (a_loc - b_loc) * np.cos(theta)
        center_y = (a_loc - b_loc) * np.sin(theta)
        trace_angle = (n_loc - 1) * theta
        trace_x = center_x + b_loc * np.cos(trace_angle)
        trace_y = center_y - b_loc * np.sin(trace_angle) # Standard formula sign
        return np.asarray(trace_x), np.asarray(trace_y)

    # --- Plot Setup ---
    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=(10, 10))
    limit = a + padding
    ax.set_xlim(-limit, limit)
    ax.set_ylim(-limit, limit)
    ax.set_aspect('equal', adjustable='box')
    ax.set_xticks([])
    ax.set_yticks([])
    ax.grid(False)
    if n_type == "integer":
        title = f"Hypocycloid with n={n} ({cusps} cusps)"
    elif n_type == "rational":
        title = f"Hypocycloid n={p}/{q} ({cusps} cusps)"
    else:
        title = f"Hypocycloid n≈{n_float:.4f} (Irrational)"
    ax.set_title(title, fontsize=16, pad=10)
    plt.subplots_adjust(left=0.08, right=0.92, top=0.98, bottom=0.02)


    # --- Plot Elements ---
    # Plot the stationary circle
    circle_fixed = patches.Circle((0, 0), a, fill=False, color='gray', lw=2,
                                  label=f'Fixed Circle (a={a})')
    ax.add_patch(circle_fixed)

    # Calculate snapshot positions
    if float(n) != 1:
        center_x_snap = (a - b) * np.cos(angle_snapshot)
        center_y_snap = (a - b) * np.sin(angle_snapshot)
        tracing_x_snap, tracing_y_snap = hypocycloid_equation(angle_snapshot, a, b, n_float)

        # Plot the rolling circle snapshot
        circle_rolling = patches.Circle((center_x_snap, center_y_snap), b, fill=False, color='cyan', lw=2,
                                        label=f'Rolling Circle (b={b:.2f})', zorder=3)
        ax.add_patch(circle_rolling)

        # Plot center of rolling circle snapshot
        ax.scatter(center_x_snap, center_y_snap, c='white', s=60, zorder=4)

        # Draw connecting line snapshot
        ax.plot([center_x_snap, tracing_x_snap], [center_y_snap, tracing_y_snap],
                color='white', linestyle='--', lw=1.5, alpha=0.7)
```

```python
    else:
        # For n=1, the "rolling" circle is centered at (0,0) with radius a=b
        # The tracing point is fixed at (a,0)
        tracing_x_snap, tracing_y_snap = a, 0
        center_x_snap, center_y_snap = 0, 0
        circle_rolling = patches.Circle((0, 0), b, fill=False, color='cyan', lw=2,
                                        label=f'Rolling Circle (b={b:.2f})', zorder=3)
        ax.add_patch(circle_rolling) # Draw the circle even if degenerate
        ax.scatter(0, 0, c='white', s=60, zorder=4, label='Rolling Center (Origin)')
        ax.plot([0, a], [0, 0], color='white', linestyle='--', lw=1.5, alpha=0.7) # Line to point

    # Plot tracing point snapshot
    ax.scatter(tracing_x_snap, tracing_y_snap, c='yellow', s=80, label='Tracing Point (P)', zorder=5)

    # Add 'P' annotation near the tracing point snapshot
    label_offset_angle = angle_snapshot + np.pi / 2 # Offset perpendicular to radius approx
    label_radius = max(a * 0.05, 0.4) # Distance from point to label, scaled slightly
    label_x = tracing_x_snap + label_radius * np.cos(label_offset_angle)
    label_y = tracing_y_snap + label_radius * np.sin(label_offset_angle)

    # Convert numpy arrays to float values if necessary
    x_snap = float(tracing_x_snap) if isinstance(tracing_x_snap, np.ndarray) else tracing_x_snap
    y_snap = float(tracing_y_snap) if isinstance(tracing_y_snap, np.ndarray) else tracing_y_snap
    x_label = float(label_x) if isinstance(label_x, np.ndarray) else label_x
    y_label = float(label_y) if isinstance(label_y, np.ndarray) else label_y

    ax.annotate('P',
                xy=(x_snap, y_snap),
                xytext=(x_label, y_label),
                color='white',
                fontsize=10,
                ha='center',
                va='center',
                bbox=dict(facecolor='black', edgecolor='white', alpha=0.7),
                arrowprops=dict(arrowstyle='->', connectionstyle="arc3", color='white', alpha=0.7))

    # Plot hypocycloid path using the full theta range
    x_hypo, y_hypo = hypocycloid_equation(theta_vals, a, b, n_float)
    ax.plot(x_hypo, y_hypo, color='yellow', lw=2, label='Hypocycloid Path')

    # Add legend
    ax.legend(loc='upper right', fontsize=9)

    # --- Save or Show ---
    if save_fig:
        if filename is None:
            # Create descriptive default filename
             n_str = (f"{n}" if isinstance(n, int) else (f"{p}d{q}" if n_type=='rational'
                      else f"{n_float:.3f}".replace('.','p')))
             filename = f"hypocycloid_static_a{a}_n{n_str}.png"

        save_dir = Path("IMAGES")
        save_dir.mkdir(parents=True, exist_ok=True)
        filepath = save_dir / filename

        print(f"Saving figure to {filepath.resolve()}...")
        try:
            plt.savefig(filepath, bbox_inches='tight', dpi=300)
            print(f"Figure saved successfully!")
        except Exception as e:
            print(f"Error saving figure: {e}")

    plt.show()
```

Integer `n`

In [ ]:
```python
# Integer n (Astroid)
print("\nPlotting Astroid (n=4)...")
# plot_hypocycloid(a=4, n=4, save_fig=True)

# Integer n (Deltoid)
print("\nPlotting Deltoid (n=3)...")
# plot_hypocycloid(a=3, n=3, save_fig=True)
```

Rational `n` (intersecting)

In [ ]:
```python
# Rational n (intersecting)
print("\nPlotting Rational Hypocycloid (n=5/3)...")
```

```
# plot_hypocycloid(a=5, n=5/3, num_points=2000, save_fig=True) # Needs 3 revs

# Rational n (more complex intersecting)
# print("\nPlotting Rational Hypocycloid (n=7/4)...")
# plot_hypocycloid(a=7, n=7/4, num_points=2500, save_fig=True) # Needs 4 revs
```

Irrational `n` (non-closing)

In [ ]:
```
# "Irrational" n (non-closing)
print("\nPlotting Irrational Hypocycloid (n=sqrt(5) ≈ 2.236)...")
sqrt5 = math.sqrt(5)
# plot_hypocycloid(a=4, n=sqrt5, max_revolutions_irrational=20, num_points=5000, save_fig=True)

print("\nPlotting Irrational Hypocycloid (n=pi ≈ 3.142)...")
# plot_hypocycloid(a=4, n=math.pi, max_revolutions_irrational=20, num_points=5000, save_fig=True)

print("\nPlotting Irrational Hypocycloid (n=e ≈ 2.718)...")
# plot_hypocycloid(a=4, n=math.e, max_revolutions_irrational=20, num_points=5000, save_fig=True)
```

Degenerate Cases

In [ ]:
```
# Degenerate case n=2 (straight line segment)
# print("\nPlotting Degenerate Hypocycloid (n=2)...")
# plot_hypocycloid(a=4, n=2, save_fig=True)

# Degenerate case n=1 (point)
# print("\nPlotting Degenerate Hypocycloid (n=1)...")
# plot_hypocycloid(a=4, n=1, save_fig=False) # Shows the point at (a,0)
```

Animating the motion of Hypocycloids

In [ ]:
```
def animate_hypocycloid_rational(a=4, n=3, save_anim=False, filename=None):
    """
    Animate multiple hypocycloids formed by small circles rolling inside a larger circle.

    Parameters:
    -----------
    a : int
        Radius of the fixed circle (default: 4)
    n : int or list
        Number of cusps or list of numbers of cusps for each hypocycloid (default: 3)
    save_anim : bool
        Whether to save the animation (default: False)
    filename : str
        Name of the file to save the animation (optional)
    """
    # Convert single n value to list if necessary
    n = [n] if isinstance(n, (int, float)) else n

    # Set up parameters
    b = [a/m for m in n]  # List of radii for rolling circles
    padding = 0.5  # Padding for the plot limits

    # Rest of the code remains the same
    frames_per_revolution = 100
    total_frames = int(max(n) * frames_per_revolution)
    interval_ms = 20

    theta_max = 2 * np.pi
    theta_vals = np.linspace(0, theta_max, total_frames)

    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.set_xlim(-(a + padding), a + padding)
    ax.set_ylim(-(a + padding), a + padding)
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_aspect('equal', adjustable='box')
    ax.grid(False)
    ax.set_title(f"Hypocycloids\na = {a}, n = {n}", fontsize=14, pad=10)

    circle_fixed = patches.Circle((0, 0), a, fill=False, color='gray', lw=2)
    ax.add_patch(circle_fixed)

    circles_rolling = []
    path_lines = []
    radius_lines = []
```

```python
    tracing_points = []
    center_points = []

    colors = plt.cm.rainbow(np.linspace(0, 1, len(b)))

    for i, radius in enumerate(b):
        circle = patches.Circle((0, 0), radius, fill=False, color=colors[i], lw=2)
        ax.add_patch(circle)
        circles_rolling.append(circle)

        path_line, = ax.plot([], [], '-', color=colors[i], lw=2, label=f'{int(a/radius)} cusps(b={radius:.2f})')
        path_lines.append(path_line)

        radius_line, = ax.plot([], [], '--', color='white', lw=1.5, alpha=0.8)
        radius_lines.append(radius_line)

        tracing_point, = ax.plot([], [], 'o', color=colors[i], ms=8)
        tracing_points.append(tracing_point)

        center_point, = ax.plot([], [], 'wo', ms=8, alpha=0.5)
        center_points.append(center_point)

    ax.legend(loc='upper right', fontsize=8)

    def init():
        """Initialize animation"""
        for i in range(len(b)):
            path_lines[i].set_data([], [])
            radius_lines[i].set_data([], [])
            tracing_points[i].set_data([], [])
            center_points[i].set_data([], [])
            circles_rolling[i].center = (a - b[i], 0)
        return path_lines + circles_rolling + radius_lines + tracing_points + center_points

    def hypocycloid_points(theta, a, b):
        """Calculate hypocycloid points"""
        center_x = (a - b) * np.cos(theta)
        center_y = (a - b) * np.sin(theta)
        trace_x = (a - b) * np.cos(theta) + b * np.cos((a/b - 1) * theta)
        trace_y = (a - b) * np.sin(theta) - b * np.sin((a/b - 1) * theta)
        return center_x, center_y, trace_x, trace_y

    def animate(frame):
        """Update animation for each frame"""
        theta = theta_vals[:frame+1]

        for i in range(len(b)):
            centers_x, centers_y, traces_x, traces_y = [], [], [], []
            for t in theta:
                cx, cy, tx, ty = hypocycloid_points(t, a, b[i])
                centers_x.append(cx)
                centers_y.append(cy)
                traces_x.append(tx)
                traces_y.append(ty)

            circles_rolling[i].center = (centers_x[-1], centers_y[-1])
            path_lines[i].set_data(traces_x, traces_y)
            tracing_points[i].set_data([traces_x[-1]], [traces_y[-1]])
            center_points[i].set_data([centers_x[-1]], [centers_y[-1]])
            radius_lines[i].set_data([centers_x[-1], traces_x[-1]],
                                     [centers_y[-1], traces_y[-1]])

        return path_lines + circles_rolling + radius_lines + tracing_points + center_points

    anim = FuncAnimation(fig, animate, frames=total_frames,
                         init_func=init, interval=interval_ms,
                         blit=False, repeat=False)

    if save_anim :
        save_dir = "ANIMATIONS/HYPOCYCLOIDS"
        os.makedirs(save_dir, exist_ok=True)

        if filename is None:
            # Create descriptive default filename
            n_str = "_".join([str(int(n_val)) if isinstance(n_val, int) else f"{n_val:.2f}" for n_val in n])
            filename = f"hypocycloid_a{a}_n{n_str}.gif"

        filepath = os.path.join(save_dir, filename)
        print(f"Saving animation to {os.path.abspath(filepath)}...")
        anim.save(filepath, writer='pillow', fps=int(1000/interval_ms), dpi=100)
```

```
        print("Animation saved successfully!")
        plt.close(fig)
    else:
        plt.show()

    return anim


anim = animate_hypocycloid_rational(a=4, n=3, save_anim=True, filename="deltoid.gif")
```

This code below animates a **hypocycloid**. The function `animate_hypocycloid` handles different cases of the ratio `n = a/b` (where `a` is the radius of the fixed circle and `b` is the radius of the rolling circle):

1. **Integer** `n`: The hypocycloid has `n` cusps and forms a closed curve.
2. **Rational** `n` **(e.g.,** `p/q`**)**: The curve self-intersects and closes after `q` revolutions of the rolling circle.
3. **Irrational** `n`: The curve never closes, and the animation runs for a fixed number of revolutions.

Key Steps:

1. **Parameter Validation**:
   - Ensures `a` is positive and `n` is at least 1.
   - Handles special cases like `n=1` (degenerate point).
2. **Animation Setup**:
   - Calculates the number of frames and the angle (`theta`) values for the animation.
   - Sets up the plot with a fixed circle, rolling circle, and the hypocycloid path.
3. **Animation Logic**:
   - The `hypocycloid_points` function computes:
     - The center of the rolling circle.
     - The tracing point on the rolling circle that forms the hypocycloid.
   - The `animate` function updates the rolling circle's position, the tracing point, and the path for each frame.
4. **Saving or Displaying**:
   - The animation can be saved as a GIF or MP4, or displayed interactively.

Example Behavior:

- For `n=3`, the rolling circle completes one revolution inside the fixed circle, forming a 3-cusp hypocycloid.
- For `n=3/2`, the curve closes after 2 revolutions, forming a self-intersecting shape.
- For irrational `n`, the curve never closes, creating a complex pattern.

This function uses **matplotlib** for plotting and animating, and optionally saves the animation to a file.

In [ ]:
```
def animate_hypocycloid(a=4, n=3.0, frames_per_rev=100, max_revolutions_irrational=5,
                        fps=25, save_anim=False, save_format="GIF", filename=None):
    """
    Animate a hypocycloid formed by a small circle rolling inside a larger circle.

    Handles integer n (cusps), rational n (intersecting closed curve),
    and irrational n (non-closing curve).

    Parameters:
    -----------
    a : float or int
        Radius of the fixed circle (default: 4). Must be positive.
    n : float, int, or Fraction
        Ratio a/b, determining the shape. Must be >= 1.
        If integer: n cusps, non-intersecting (except at center if n=2).
        If rational (p/q): p cusps, closes after q revolutions, self-intersects.
        If irrational: Does not close. Animation runs for max_revolutions_irrational.
        (default: 3.0)
    frames_per_rev : int
        Number of frames per 2*pi revolution of the rolling circle's center.
        Controls animation smoothness. (default: 100)
    max_revolutions_irrational : float or int
        Number of 2*pi revolutions of the center to simulate for irrational n.
        (default: 5)
    fps : int
        Frames per second for the animation. (default: 25)
    save_anim : bool
        Whether to save the animation (default: False).
    save_format : str
```

```
        Format to save the animation (default: "GIF"). Options: "GIF", "MP4".
    filename : str
        Name of the file to save the animation (required if save_anim is True).
        Default filenames are generated if None.
    """
    # --- Parameter Validation ---
    if not isinstance(a, (int, float)) or a <= 0:
        raise ValueError("Fixed circle radius 'a' must be a positive number.")
    if not isinstance(n, (int, float, Fraction)) or n < 1:
        raise ValueError("Ratio 'n = a/b' must be a number >= 1.")
    if n == 1:
         warnings.warn("n=1 results in a degenerate hypocycloid (a point). Animation will show a stationary point
         # Set b=a, calculations should result in center=0, point=a
         b = a
    else:
         b = a / n  # Radius of the rolling circle

    padding = a * 0.15  # Padding for the plot limits based on fixed radius

    # --- Determine Animation Length ---
    n_type = "integer"
    theta_revolutions = 1 # Default for integer n

    if isinstance(n, int):
        n_type = "integer"
        theta_revolutions = 1 # Closes in 1 revolution of the center
        cusps = n
        print(f"Animating Hypocycloid: Integer n={n} ({cusps} cusps)")
    else: # Float or Fraction
        try:
            # Use Fraction to get p/q form
            frac_n = Fraction(n).limit_denominator() # Simplify to p/q
            p = frac_n.numerator
            q = frac_n.denominator

            if q == 1: # It simplified to an integer
                 n_type = "integer"
                 n = p # Use the integer value
                 theta_revolutions = 1
                 cusps = n
                 print(f"Animating Hypocycloid: Rational n simplified to integer n={n} ({cusps} cusps)")
            elif is_irrational(n): # Check if it's irrational or complex rational
                 n_type = "irrational"
                 theta_revolutions = max_revolutions_irrational
                 print(f"Animating Hypocycloid: Irrational n ≈ {n:.4f} (running for {theta_revolutions} center re
                 cusps = 'Infinite'
            else: # It's a non-integer rational number
                 n_type = "rational"
                 theta_revolutions = q # Closes after q revolutions of the center
                 cusps = p
                 print(f"Animating Hypocycloid: Rational n = {p}/{q} ({cusps} cusps, closing in {q} center revolu

        except (ValueError, OverflowError):
             n_type = "irrational" # Treat errors as irrational case
             theta_revolutions = max_revolutions_irrational
             print(f"Animating Hypocycloid: Could not represent n={n} as simple fraction. Treating as irrational
             cusps = 'Infinite'


    # Animation parameters
    FPS = fps  # Frames per second
    theta_max = 2 * np.pi * theta_revolutions
    total_frames = int(theta_revolutions * frames_per_rev)
    if total_frames <= 0: total_frames = frames_per_rev # Ensure minimum frames
    theta_vals = np.linspace(0, theta_max, total_frames)

    # --- Plot Setup ---
    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=(10, 10))
    fig.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.1)
    limit = a + padding
    ax.set_xlim(-limit, limit)
    ax.set_ylim(-limit, limit)
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_aspect('equal', adjustable='box')
    ax.grid(False)
    if n_type == "integer":
        title = f"Hypocycloid with n={n} ({cusps} cusps)"
    elif n_type == "rational":
```

```
        title = f"Hypocycloid n={p}/{q} ({cusps} cusps)"
    else:
        title = f"Hypocycloid n≈{n:.4f} (Irrational)"
    ax.set_title(title, fontsize=14, pad=10)

    # --- Create Plot Elements ---
    # Stationary circle
    circle_fixed = patches.Circle((0, 0), a, fill=False, color='gray', lw=2,
                                  label=f'Fixed Circle (a={a})')
    ax.add_patch(circle_fixed)

    # Rolling circle (will be updated)
    circle_rolling = patches.Circle((a - b, 0), b, fill=False, color='cyan', lw=2,
                                    label=f'Rolling Circle (b={b:.2f})', zorder=3)
    ax.add_patch(circle_rolling)

    # Path line (will be updated) - Use lists for efficiency
    path_line, = ax.plot([], [], '-', color='yellow', lw=2, label='Hypocycloid Path')
    x_hist, y_hist = [], [] # Store history outside animate

    # Radius line (will be updated)
    radius_line, = ax.plot([], [], 'w--', lw=1.5, alpha=0.7)

    # Points (will be updated)
    tracing_point, = ax.plot([], [], 'o', color='yellow', ms=8, label='Tracing Point', zorder=4)
    center_point, = ax.plot([], [], 'o', color='white', ms=6, alpha=0.7)

    # Add legend
    ax.legend(loc='upper right', fontsize=9)

    # Add text annotation for revolution counter (only for non-integer n)
    rev_text = None
    if not isinstance(n, int):
        rev_text = ax.text(0.02, 0.98, '', transform=ax.transAxes,
                           color='white', fontsize=10, ha='left', va='top')

    # --- Animation Functions ---
    def init():
        """Initialize animation"""
        nonlocal x_hist, y_hist
        x_hist, y_hist = [], []
        path_line.set_data([], [])
        radius_line.set_data([], [])
        tracing_point.set_data([], [])
        center_point.set_data([], [])
        if n != 1:
            circle_rolling.center = (a - b, 0)
        else: # n=1 case
            circle_rolling.center = (0,0)
            _, _, tx_init, ty_init = hypocycloid_points(0)
            tracing_point.set_data([tx_init], [ty_init])
            center_point.set_data([0],[0])
            radius_line.set_data([0, tx_init], [0, ty_init])


        # Update the text annotation for revolutions
        if rev_text is not None:
            rev_text.set_text(f'Revolutions: 0/{theta_revolutions}')

        elements = [path_line, circle_rolling, radius_line, tracing_point, center_point]
        return [rev_text] + elements if rev_text else elements

    def hypocycloid_points(theta):
        """Calculate hypocycloid points (h=b case)."""
        if n == 1: # Degenerate case b=a
            return 0.0, 0.0, a, 0.0 # Center at origin, point fixed at (a,0)

        # Center of rolling circle
        center_x = (a - b) * np.cos(theta)
        center_y = (a - b) * np.sin(theta)
        # Angle for the tracing point calculation relative to the rolling circle's center rotation
        trace_angle = (n - 1) * theta

        # Tracing point (Hypocycloid: h=b)
        trace_x = center_x + b * np.cos(trace_angle)
        trace_y = center_y - b * np.sin(trace_angle)

        return center_x, center_y, trace_x, trace_y

    def animate(frame):
```

```python
        """Update animation for each frame (efficient version)."""
        t = theta_vals[frame]

        # Calculate points for the current frame
        current_center_x, current_center_y, current_trace_x, current_trace_y = hypocycloid_points(t)

        # Update rolling circle position
        circle_rolling.center = (current_center_x, current_center_y)

        # Update points
        tracing_point.set_data([current_trace_x], [current_trace_y])
        center_point.set_data([current_center_x], [current_center_y])

        # Update radius line
        radius_line.set_data([current_center_x, current_trace_x],
                             [current_center_y, current_trace_y])

        # Append to history and update path line
        x_hist.append(current_trace_x)
        y_hist.append(current_trace_y)
        path_line.set_data(x_hist, y_hist)

        # Update revolution counter for non-integer n
        if rev_text is not None:
            current_rev = frame // frames_per_rev
            rev_text.set_text(f'Revolutions: {current_rev}/{theta_revolutions}')

        elements = [path_line, circle_rolling, radius_line, tracing_point, center_point]
        return [rev_text] + elements if rev_text else elements

    # --- Create and Run Animation ---
    anim = FuncAnimation(fig, animate, frames=total_frames,
                         init_func=init, interval=int(1000 / FPS),
                         blit=False, repeat=False)

    # --- Save or Show ---
    if save_anim:
        if filename is None:
            # Create a more descriptive default filename
            n_str = f"{n}" if isinstance(n, int) else (f"{p}d{q}" if n_type=='rational' else f"{n:.3f}".replace(
            if save_format.lower() == "gif":
                filename = f"hypocycloid_a{a}_n{n_str}.gif"
            else:
                filename = f"hypocycloid_a{a}_n{n_str}.mp4"

        save_dir = Path("ANIMATIONS/HYPOCYCLOIDS")
        save_dir.mkdir(parents=True, exist_ok=True)
        filepath = save_dir / filename

        print(f"Saving animation to {filepath.resolve()}...")

        # Use tqdm progress bar
        try:
             with tqdm(total=total_frames, desc="Saving Animation", unit="frame") as pbar:
                def progress_callback(current_frame, total_frames):
                    pbar.update(1)

                if save_format.lower() == "gif":
                    anim.save(str(filepath), writer='pillow', fps=FPS,
                              dpi=100, progress_callback=progress_callback)
                else:
                    writer = FFMpegWriter(fps=FPS, metadata=dict(artist='Hypocycloid Animator'), bitrate=1800)
                    anim.save(str(filepath), writer=writer,
                              dpi=150, progress_callback=progress_callback)
             print("Animation saved successfully!")
        except Exception as e:
             print(f"Error saving animation: {e}")

        plt.close(fig) # Close plot window after saving
    else:
        plt.show() # Display interactively

    return anim
```

Integer n

In [ ]:
```python
# Integer n (Astroid)
print("\nGenerating Astroid (n=4)...")
# anim_astroid = animate_hypocycloid(a=4, n=4, save_anim=True, filename="astroid.gif", fps=20)
```

```
# Integer n (Deltoid)
print("\nGenerating Deltoid (n=3)...")
# anim_deltoid = animate_hypocycloid(a=3, n=3, save_anim=True, filename="deltoid.gif", fps=20)

# Pentoid (n=5)
print("\nGenerating Pentoid (n=5)...")
# anim_pentoid = animate_hypocycloid(a=5, n=5, save_anim=True, filename="pentoid.gif", fps=20)
```

Rational n

In [ ]:
```
# Rational n (intersecting)
print("\nGenerating Rational Hypocycloid (n=5/3)...")
anim_5d3 = animate_hypocycloid(a=5, n=5/3, frames_per_rev=100, save_anim=True) # Needs 3 revs

# Rational n (more complex intersecting)
print("\nGenerating Rational Hypocycloid (n=7/4)...")
# anim_7d4 = animate_hypocycloid(a=7, n=7/4, frames_per_rev=100, save_anim=True) # Needs 4 revs
```

Irrational n

In [ ]:
```
# "Irrational" n (non-closing) - using float approximation
print("\nGenerating Irrational Hypocycloid (n=sqrt(5) ≈ 2.236)...")
sqrt5 = math.sqrt(5)
# anim_root_5 = animate_hypocycloid(a=4, n=sqrt5, max_revolutions_irrational=20, frames_per_rev=100, save_anim=Tr

print("\nGenerating Irrational Hypocycloid (n=e ≈ 2.718)...")
# anim_e = animate_hypocycloid(a=4, n=math.e,
#                               max_revolutions_irrational=10,
#                               frames_per_rev=100, fps = 40,
#                               save_anim=False,
#                               filename="hypocycloid_e.gif")

# Another "Irrational" n (Pi)
print("\nGenerating Irrational Hypocycloid (n=pi ≈ 3.141)...")
anim_pi = animate_hypocycloid(a=4, n=np.pi,
                              max_revolutions_irrational=10,
                              frames_per_rev=100,
                              fps=40,
                              save_anim=False,
                              filename="hypocycloid_pi.gif")
```

Degenerate cases

In [ ]:
```
# Degenerate case n=2 (straight line segment)
# print("\nGenerating Degenerate Hypocycloid (n=2)...")
# animate_hypocycloid(a=4, n=2, save_anim=True)

# Degenerate case n=1 (point)
# print("\nGenerating Degenerate Hypocycloid (n=1)...")
# animate_hypocycloid(a=4, n=1, save_anim=False) # No point saving a static point
```

Derivation

1. Arc Length of Hypocycloids

A hypocycloid is a plane curve produced by tracing the path of a point on a circle (the hypocycle) that rolls without slipping around the *inside* of a fixed circle. The parametric equations for a hypocycloid generated by a fixed circle of radius R and a rolling circle of radius r are given by:

$$x(\theta) = (R-r)\cos\theta + r\cos\left(\frac{R-r}{r}\theta\right)$$

$$y(\theta) = (R-r)\sin\theta - r\sin\left(\frac{R-r}{r}\theta\right)$$

where θ is the angle the line segment from the center of the fixed circle to the center of the rolling circle makes with the positive x-axis. Note the sign changes compared to the epicycloid equations.

The arc length L of a parametric curve $x = x(\theta)$ and $y = y(\theta)$ from $\theta = \alpha$ to $\theta = \beta$ is given by the formula:

$$L = \int_\alpha^\beta \sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2}\, d\theta$$

First, we find the derivatives of the parametric equations with respect to θ:

$$\frac{dx}{d\theta} = -(R-r)\sin\theta + r\left(\frac{R-r}{r}\right)\left(-\sin\left(\frac{R-r}{r}\theta\right)\right) = -(R-r)\sin\theta - (R-r)\sin\left(\frac{R-r}{r}\theta\right)$$

$$\frac{dy}{d\theta} = (R-r)\cos\theta - r\left(\frac{R-r}{r}\right)\cos\left(\frac{R-r}{r}\theta\right) = (R-r)\cos\theta - (R-r)\cos\left(\frac{R-r}{r}\theta\right)$$

Now, we compute the square of these derivatives and add them:

$$\left(\frac{dx}{d\theta}\right)^2 = (R-r)^2\left(-\sin\theta - \sin\left(\frac{R-r}{r}\theta\right)\right)^2 = (R-r)^2\left(\sin^2\theta + 2\sin\theta\sin\left(\frac{R-r}{r}\theta\right) + \sin^2\left(\frac{R-r}{r}\theta\right)\right)$$

$$\left(\frac{dy}{d\theta}\right)^2 = (R-r)^2\left(\cos\theta - \cos\left(\frac{R-r}{r}\theta\right)\right)^2 = (R-r)^2\left(\cos^2\theta - 2\cos\theta\cos\left(\frac{R-r}{r}\theta\right) + \cos^2\left(\frac{R-r}{r}\theta\right)\right)$$

Summing these squares:

$$\begin{aligned}&\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2 \\ &= (R-r)^2\left(\sin^2\theta + \cos^2\theta + \sin^2\left(\frac{R-r}{r}\theta\right) + \cos^2\left(\frac{R-r}{r}\theta\right) + 2\sin\theta\sin\left(\frac{R-r}{r}\theta\right) - 2\cos\theta\cos\left(\frac{R-r}{r}\theta\right)\right)\end{aligned}$$

Using the identity $\sin^2 x + \cos^2 x = 1$ and the cosine addition formula $\cos(A+B) = \cos A\cos B - \sin A\sin B$:

$$\begin{aligned}\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2 &= (R-r)^2\left(1 + 1 - 2\left(\cos\theta\cos\left(\frac{R-r}{r}\theta\right) - \sin\theta\sin\left(\frac{R-r}{r}\theta\right)\right)\right) \\ &= (R-r)^2\left(2 - 2\cos\left(\theta + \frac{R-r}{r}\theta\right)\right) \\ &= (R-r)^2\left(2 - 2\cos\left(\frac{r+R-r}{r}\theta\right)\right) \\ &= (R-r)^2\left(2 - 2\cos\left(\frac{R}{r}\theta\right)\right) \\ &= 2(R-r)^2\left(1 - \cos\left(\frac{R}{r}\theta\right)\right)\end{aligned}$$

Using the half-angle identity $1 - \cos(2\alpha) = 2\sin^2(\alpha)$, with $2\alpha = \frac{R}{r}\theta$, so $\alpha = \frac{R}{2r}\theta$:

$$= 2(R-r)^2\left(2\sin^2\left(\frac{R}{2r}\theta\right)\right) = 4(R-r)^2\sin^2\left(\frac{R}{2r}\theta\right)$$

Now, take the square root:

$$\sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2} = \sqrt{4(R-r)^2\sin^2\left(\frac{R}{2r}\theta\right)} = 2|R-r|\left|\sin\left(\frac{R}{2r}\theta\right)\right|$$

We assume $R > r$ for a non-trivial hypocycloid, so $|R-r| = R-r$.

$$\sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2} = 2(R-r)\left|\sin\left(\frac{R}{2r}\theta\right)\right|$$

The hypocycloid is a closed curve if and only if the ratio $\frac{R}{r}$ is a rational number. Let $\frac{R}{r} = \frac{p}{q}$, where p and q are coprime positive integers. The curve completes one full cycle when the parameter θ ranges from 0 to $2\pi q$.

To find the total arc length of the closed hypocycloid, we integrate from 0 to $2\pi q$:

$$L = \int_0^{2\pi q} 2(R-r)\left|\sin\left(\frac{p}{2q}\theta\right)\right| d\theta$$

Let $u = \frac{p}{2q}\theta$. Then $du = \frac{p}{2q}d\theta$, so $d\theta = \frac{2q}{p}du$. When $\theta = 0$, $u = 0$. When $\theta = 2\pi q$, $u = \frac{p}{2q}(2\pi q) = p\pi$.

$$L = \int_0^{p\pi} 2(R-r)|\sin(u)|\frac{2q}{p}du = \frac{4q(R-r)}{p}\int_0^{p\pi}|\sin(u)|du$$

As derived before, the integral $\int_0^{p\pi}|\sin(u)|du = 2p$.

Substituting this back into the arc length formula:

$$L = \frac{4q(R-r)}{p}(2p) = 8q(R-r)$$

Therefore, the total arc length of a hypocycloid where the ratio of the fixed circle radius R to the rolling circle radius r is a rational number $\frac{p}{q}$ in simplest form is $8q(R-r)$.

If $\frac{R}{r} = k$ is an integer, then $p = k$ and $q = 1$. The total arc length is $8(1)(R-r) = 8(R-r)$. This is a common case, and for $k \geq 2$, the hypocycloid has k cusps. A special case is when $R = 2r$ ($k = 2$). The hypocycloid is a straight line segment traced back and forth along the diameter of the fixed circle. The formula gives $L = 8q(R-r) = 8(1)(2r-r) = 8r$. This matches the physical reality: the diameter of the fixed circle is $4r$, and as the point traces this diameter twice (once in each direction), the total arc length is indeed $8r$.

For $R = 2r$, the parametric equations show that:

- As θ goes from 0 to π, the point moves from $(2r, 0)$ to $(-2r, 0)$
- As θ goes from π to 2π, the point moves from $(-2r, 0)$ back to $(2r, 0)$

Thus, the total distance traveled is $4r + 4r = 8r$, confirming our derived formula.

The formula for the total arc length of a hypocycloid where the ratio $R/r = k$ is an integer is $8(R-r)$.

For the case where $R/r = p/q$ in simplest form, the curve closes after $\theta = 2\pi q$. The number of cusps is p. The number of arches is also p. The length of one arch, as derived by integrating from 0 to $2\pi/p$ (the range of θ for one arch based on the cusps), is $\frac{8(R-r)}{p}$. The total length would then be $p \times \frac{8(R-r)}{p} = 8(R-r)$. This still seems inconsistent with the $8q(R-r)$ result.

Let's carefully consider the period of the parametric equations. The terms $\cos(\theta)$ and $\sin(\theta)$ have a period of 2π. The terms $\cos\left(\frac{R-r}{r}\theta\right)$ and $\sin\left(\frac{R-r}{r}\theta\right)$ have a period of $\frac{2\pi}{\frac{R-r}{r}} = \frac{2\pi r}{R-r}$. For the curve to close, both parts must return to their initial values simultaneously. The period of the epicycloid is the least common multiple of the periods of the two components. Period of θ terms is 2π. Period of $\frac{R-r}{r}\theta$ terms is $\frac{2\pi r}{R-r}$. Let $\frac{R}{r} = \frac{p}{q}$ (coprime). Then $\frac{R-r}{r} = \frac{p/q-r-r}{r} = \frac{p/q-1}{1} = \frac{p-q}{q}$. The period of the second terms is $\frac{2\pi}{(p-q)/q} = \frac{2\pi q}{p-q}$. The periods are 2π and $\frac{2\pi q}{p-q}$. The least common multiple of their denominators (1 and $p-q$) is $p-q$. The least common multiple of their numerators (2π and $2\pi q$) is $2\pi q$ if $\gcd(1, p-q) = 1$, which is true. However, we need the LCM of the periods themselves. LCM of 2π and $\frac{2\pi q}{p-q}$ is $\frac{\text{LCM}(2\pi(p-q), 2\pi q)}{|p-q|}$. This approach seems complicated.

Let's go back to the arc length integral from 0 to $2\pi q$. $L = \frac{4q(R-r)}{p}\int_0^{p\pi} |\sin(u)| du$ The integral $\int_0^{p\pi} |\sin(u)| du = 2p$. $L = \frac{4q(R-r)}{p}(2p) = 8q(R-r)$.

This formula represents the total distance traced by the point as θ goes from 0 to $2\pi q$, which is the interval for the curve to close when $R/r = p/q$.

The number of cusps in a hypocycloid with $R/r = p/q$ (coprime) is p. The curve consists of p identical arches. The total length is p times the length of one arch. The range of θ for one arch goes from one cusp to the next. Cusps occur when $\frac{R}{r}\theta$ is a multiple of 2π. Since $\frac{R}{r} = \frac{p}{q}$, $\frac{p}{q}\theta = 2\pi n$, so $\theta = \frac{2\pi n q}{p}$. The cusps are at $\theta = 0, \frac{2\pi q}{p}, \frac{4\pi q}{p}, \ldots, 2\pi q$. One arch is from $\theta = 0$ to $\theta = \frac{2\pi q}{p}$. Integral for one arch: $L_{\text{arch}} = \int_0^{2\pi q/p} 2(R-r)\left|\sin\left(\frac{R}{2r}\theta\right)\right| d\theta = \int_0^{2\pi q/p} 2(R-r)\left|\sin\left(\frac{p}{2q}\theta\right)\right| d\theta$ Let $u = \frac{p}{2q}\theta$. When $\theta = 0$, $u = 0$. When $\theta = \frac{2\pi q}{p}$, $u = \frac{p}{2q}\frac{2\pi q}{p} = \pi$. $L_{\text{arch}} = \int_0^{\pi} 2(R-r)|\sin(u)|\frac{2q}{p} du = \frac{4q(R-r)}{p}\int_0^{\pi} \sin(u) du$ (since $\sin(u) \geq 0$ for $u \in [0, \pi]$) $L_{\text{arch}} = \frac{4q(R-r)}{p}[-\cos(u)]_0^{\pi} = \frac{4q(R-r)}{p}(1-(-1)) = \frac{8q(R-r)}{p}$.

Total length = Number of arches $\times$ Length of one arch = $p \times \frac{8q(R-r)}{p} = 8q(R-r)$.

The derivation seems consistent. The total arc length of a hypocycloid where $R/r = p/q$ (in simplest form) is $8q(R-r)$.

Let's re-check the $R = 2r$ case with $R/r = 2/1$, so $p = 2, q = 1$. Total length = $8(1)(2r-r) = 8r$. This is still the distance traced, which is twice the diameter.

The total **Arc Length** of a Hypocycloid is

$\boxed{8q(R-r) \text{ where } R/r = p/q \text{ in simplest form}}$.

2. Area enclosed by a Hypocycloid

The standard parametric equations for the hypocycloid are:

$$x(\theta) = (R-r)\cos\theta + r\cos\left(\frac{R-r}{r}\theta\right)$$

$$y(\theta) = (R-r)\sin\theta - r\sin\left(\frac{R-r}{r}\theta\right)$$

Substituting $R = kr$:

$$\frac{R-r}{r} = \frac{kr-r}{r} = k-1$$

The equations become:

$$\begin{aligned} x(\theta) &= r(k-1)\cos\theta + r\cos((k-1)\theta) \\ y(\theta) &= r(k-1)\sin\theta - r\sin((k-1)\theta) \end{aligned}$$

The curve forms a closed loop (or set of loops) that closes when θ completes q rotations, i.e., θ goes from 0 to $2\pi q$, where $k = p/q$ with p, q being coprime integers. If k is an integer, $q = 1$, and the curve closes after $\theta = 2\pi$.

1. Area Formula using Green's Theorem

The area A enclosed by a simple closed curve C defined parametrically by $(x(\theta), y(\theta))$ for $\theta \in [a, b]$ can be calculated using Green's Theorem:

$$\boxed{A = \frac{1}{2}\oint_C (x\,dy - y\,dx) = \frac{1}{2}\int_a^b \left(x(\theta)\frac{dy}{d\theta} - y(\theta)\frac{dx}{d\theta}\right) d\theta}$$

The sign of the result depends on the orientation of the parametrization. We expect a positive area, so we may need to take the absolute value or adjust the sign if the parametrization turns out to be clockwise.

2. Calculate Derivatives

First, find the derivatives $\frac{dx}{d\theta}$ and $\frac{dy}{d\theta}$:

$$\begin{aligned} \frac{dx}{d\theta} &= -r(k-1)\sin\theta - r(k-1)\sin((k-1)\theta) \\ &= -r(k-1)[\sin\theta + \sin((k-1)\theta)] \end{aligned}$$

$$\begin{aligned} \frac{dy}{d\theta} &= r(k-1)\cos\theta - r(k-1)\cos((k-1)\theta) \\ &= r(k-1)[\cos\theta - \cos((k-1)\theta)] \end{aligned}$$

3. Calculate the Integrand: $x\frac{dy}{d\theta} - y\frac{dx}{d\theta}$

$$\begin{aligned} x\frac{dy}{d\theta} &= [r(k-1)\cos\theta + r\cos((k-1)\theta)] \cdot [r(k-1)(\cos\theta - \cos((k-1)\theta))] \\ &= r^2(k-1)[(k-1)\cos\theta + \cos((k-1)\theta)][\cos\theta - \cos((k-1)\theta)] \\ &= r^2(k-1)[(k-1)\cos^2\theta - (k-1)\cos\theta\cos((k-1)\theta) + \cos\theta\cos((k-1)\theta) - \cos^2((k-1)\theta)] \\ &= r^2(k-1)[(k-1)\cos^2\theta - (k-2)\cos\theta\cos((k-1)\theta) - \cos^2((k-1)\theta)] \end{aligned}$$

$$\begin{aligned} y\frac{dx}{d\theta} &= [r(k-1)\sin\theta - r\sin((k-1)\theta)] \cdot [-r(k-1)(\sin\theta + \sin((k-1)\theta))] \\ &= -r^2(k-1)[(k-1)\sin\theta - \sin((k-1)\theta)][\sin\theta + \sin((k-1)\theta)] \\ &= -r^2(k-1)[(k-1)\sin^2\theta + (k-1)\sin\theta\sin((k-1)\theta) - \sin\theta\sin((k-1)\theta) - \sin^2((k-1)\theta)] \\ &= -r^2(k-1)[(k-1)\sin^2\theta + (k-2)\sin\theta\sin((k-1)\theta) - \sin^2((k-1)\theta)] \end{aligned}$$

Now subtract $y\frac{dx}{d\theta}$ from $x\frac{dy}{d\theta}$:

$$\begin{aligned} & x\frac{dy}{d\theta} - y\frac{dx}{d\theta} \\ &= r^2(k-1)[((k-1)\cos^2\theta - (k-2)\cos\theta\cos((k-1)\theta) - \cos^2((k-1)\theta)) - (-(k-1)\sin^2\theta - (k-2)\sin\theta\sin((k-1)\theta) + \sin^2((k-1)\theta))] \\ &= r^2(k-1)[(k-1)(\cos^2\theta + \sin^2\theta) - (\cos^2((k-1)\theta) + \sin^2((k-1)\theta)) - (k-2)(\cos\theta\cos((k-1)\theta) - \sin\theta\sin((k-1)\theta))] \end{aligned}$$

Using identities $\cos^2 A + \sin^2 A = 1$ and $\cos(A+B) = \cos A\cos B - \sin A\sin B$:

$$\begin{aligned} &= r^2(k-1)[(k-1)(1) - (1) - (k-2)\cos(\theta + (k-1)\theta)] \\ &= r^2(k-1)[k-1-1-(k-2)\cos(k\theta)] \\ &= r^2(k-1)[k-2-(k-2)\cos(k\theta)] \\ &= r^2(k-1)(k-2)[1-\cos(k\theta)] \end{aligned}$$

4. Integrate to Find the Area

The integration interval is $\theta \in [0, 2\pi q]$, where $k = p/q$ in lowest terms.

$$\begin{aligned}
A &= \frac{1}{2}\int_0^{2\pi q} r^2(k-1)(k-2)[1-\cos(k\theta)]\,d\theta \\
&= \frac{r^2(k-1)(k-2)}{2}\int_0^{2\pi q}(1-\cos(k\theta))\,d\theta \\
&= \frac{r^2(k-1)(k-2)}{2}\left[\int_0^{2\pi q} 1\,d\theta - \int_0^{2\pi q}\cos(k\theta)\,d\theta\right] \\
&= \frac{r^2(k-1)(k-2)}{2}[2\pi q - 0] \\
&= \pi q r^2(k-1)(k-2)
\end{aligned}$$

5. Result and Interpretation

The total area enclosed by the hypocycloid traced over $\theta \in [0, 2\pi q]$ (where $k = R/r = p/q$ in lowest terms) is:

$$\boxed{A = \pi q r^2(k-1)(k-2)}$$

We can express this in terms of R and r:

$$\begin{aligned}
A &= \pi q r^2\left(\frac{R}{r}-1\right)\left(\frac{R}{r}-2\right) \\
&= \pi q r^2\frac{(R-r)}{r}\frac{(R-2r)}{r} \\
&= \pi q(R-r)(R-2r)
\end{aligned}$$

Special Case: Integer k If $k = R/r$ is an integer, then $q = 1$, and the formula simplifies to the commonly cited result for hypocycloids with k cusps:

$$\boxed{A = \pi r^2(k-1)(k-2)}$$

or

$$A = \pi(R-r)(R-2r)$$

- **Example: Astroid (k=4)** $R = 4r$, $k = 4$ (integer, so $q = 1$).

  $$A = \pi(1)r^2(4-1)(4-2) = \pi r^2(3)(2) = 6\pi r^2$$

  Using the R, r formula:

  $$A = \pi(1)(4r-r)(4r-2r) = \pi(3r)(2r) = 6\pi r^2$$

- **Example: Deltoid (k=3)** $R = 3r$, $k = 3$ (integer, so $q = 1$).

  $$A = \pi(1)r^2(3-1)(3-2) = \pi r^2(2)(1) = 2\pi r^2$$

  Using the R, r formula:

  $$A = \pi(1)(3r-r)(3r-2r) = \pi(2r)(r) = 2\pi r^2$$

- **Case: k=2** $R = 2r$. The formula gives $A = \pi(1)(2r-r)(2r-2r) = 0$. This is correct, as a hypocycloid with $R = 2r$ degenerates into a straight line segment (a diameter of the fixed circle) traced back and forth, enclosing no area.

The derived formula $A = \pi q(R-r)(R-2r)$ gives the net area enclosed by the hypocycloid when $k = R/r = p/q$. For integer values of $k \geq 3$, this corresponds to the geometric area of the region bounded by the curve.

Hypotrochoids

Overview

A **hypotrochoid** is a type of roulette curve generated by a point attached to a circle (called the *rolling circle*) as it rolls **without slipping inside** a larger fixed circle (called the *base circle*). The tracing point can be located either inside, on, or outside the circumference of the rolling circle, which gives rise to different shapes of the curve.

Mathematical Representation

Parametric Equations

The parametric equations for a hypotrochoid are given by:

$$x(\phi) = (a-b)\cos(\phi) + h\cos\left(\frac{a-b}{b}\phi\right)$$

$$y(\phi) = (a-b)\sin(\phi) - h\sin\left(\frac{a-b}{b}\phi\right)$$

Where:

- a: Radius of the fixed circle (base circle),
- b: Radius of the rolling circle,
- h: Distance of the tracing point from the center of the rolling circle,
- ϕ: Angle parameter (not the polar angle).

Polar Form

The polar form of a hypotrochoid can be derived from the parametric equations. First, calculate $r^2 = x^2 + y^2$:

$$r^2 = (a-b)^2 + h^2 + 2h(a-b)\cos\left(\frac{a}{b}\phi\right)$$

Here, ϕ is the parameter, not the polar angle. The polar angle θ is given by:

$$\tan\theta = \frac{y}{x} = \frac{(a-b)\sin(\phi) - h\sin\left(\frac{a-b}{b}\phi\right)}{(a-b)\cos(\phi) + h\cos\left(\frac{a-b}{b}\phi\right)}$$

Special Cases of Hypotrochoids

1. **Hypocycloid**:
   - When $h = b$, the tracing point lies on the circumference of the rolling circle.
   - The resulting curve is a **hypocycloid**, which is a special case of a hypotrochoid.
   - Parametric equations:

$$x(\phi) = (a-b)\cos(\phi) + b\cos\left(\frac{a-b}{b}\phi\right)$$

$$y(\phi) = (a-b)\sin(\phi) - b\sin\left(\frac{a-b}{b}\phi\right)$$

2. **Ellipse**:
   - When $a = 2b$, the hypotrochoid becomes an **ellipse**.
   - Parametric equations:

$$x(\phi) = b\cos(\phi) + h\cos(\phi)$$

$$y(\phi) = b\sin(\phi) - h\sin(\phi)$$

3. **Rose Curve**:
   - When $h = a - b$, the hypotrochoid resembles a **rose curve**.
   - The polar equation of a rose curve takes the form $r = a\cos(n\phi)$. where, $n = a/b$.
   - The number of petals depends on the ratio $n = a/b$.
   - If n is odd, the rose is n-petalled. If n is even, the rose is $2n$-petalled.
4. **Line Segment**:
   - When $a = 2b$ and $h = b$, the hypotrochoid degenerates into a straight line segment.

Key Properties

1. **Number of Cusps**:
   - The number of cusps in a hypotrochoid depends on the ratio a/b:
     - If $a/b = n$ (a rational number), the hypotrochoid will have n cusps.
     - If a/b is irrational, the curve never closes and forms a dense pattern.
2. **Arc Length**:

- The total arc length of a closed hypotrochoid where $a/b = p/q$, in the interval $[0, 2\pi q]$ is

$$L_{\text{total}} = (a-b)\int_0^{2\pi q}\sqrt{1+\left(\frac{h}{b}\right)^2 - 2\frac{h}{b}\cos\left(\frac{p}{q}\theta\right)}\,d\theta$$

If you are interested how the arc length is calculated you can check out the derivation section.

3. **Area Enclosed**:

- The area A enclosed by one complete hypotrochoid is:

$$A = \pi q(k-1)\left[b^2(k-1) - h^2\right]$$

where $k = \frac{a}{b} = \frac{p}{q}$ is a rational in the simplest form. For further details check out the derivation section.

Visualization and Construction

Geometric Construction

1. Start with a fixed circle of radius a
2. Roll a smaller circle of radius b along the *inside* of the fixed circle without slipping.
3. Track the motion of a point located at a distance h from the center of the rolling circle.

Symmetry

- Hypotrochoids exhibit rotational symmetry. The number of symmetry axes corresponds to the number of cusps.

Related Curves

1. **Epicycloids**: Generated when the rolling circle rolls *outside* the fixed circle.
2. **Cycloids**: Generated when a circle rolls along a straight line.
3. **Epitrochoids**: A generalization of epicycloids where the tracing point is not restricted to the circumference of the rolling circle.

References

For more information, visit:

- MathWorld: Hypotrochoid

Animating Hypotrochoids

This code below defines a `Hypotrochoids` class to generate and animate mathematical curves called hypotrochoids, which are traced by a point on a circle rolling inside a larger stationary circle. The key functions and special case handling are discussed below.

Key Functions

1. `_is_irrational`:
   - Determines if a number is irrational or a complex rational (e.g., with a large denominator).
   - Uses `Fraction` to approximate the number and checks the denominator size.
   - Special cases like `π` and `e` are treated as irrational.
2. `_process_parameters`:
   - Processes the input parameters (`n_values`, `r_values`, and `d`) to calculate:
     - `b` (rolling circle radius).
     - `h` (distance of the tracing point).
     - `n` (ratio `R/r`).
     - `n_types` (integer, rational, or irrational).
   - Handles special cases for different curve types:
     - **Ellipse**: Enforces `n=2` (rolling circle radius = `R/2`).
     - **Rose curves**: Sets `h = |R - r|`.
     - **Hypocycloids**: Sets `h = r`.
   - Validates inputs and adjusts them for consistency.
3. `_calculate_animation_parameters`:

- Computes the animation parameters:
  - Plot limits based on the maximum reach of the tracing point.
  - Total frames and angle range (`theta`) based on the number of revolutions required for the curves to close.

4. `_hypotrochoid_points` :
   - Calculates the positions of the rolling circle's center and the tracing point for a given angle (`theta`).
   - Uses the parametric equations of a hypotrochoid.
5. `_setup_plot` :
   - Sets up the Matplotlib plot for the animation, including:
     - The stationary circle.
     - Rolling circles, tracing points, and paths.
     - Legends and revolution counters.
6. `generate_animation` :
   - Creates and displays or saves the animation using `FuncAnimation` .

Handling of `n` and `r` Types

- **`n` (Ratio `R/r`):**
  - Can be an integer, rational (e.g., `3/2`), or irrational (e.g., `π`).
  - **Integer `n`** : The curve closes in 1 revolution.
  - **Rational `n`** : The curve closes in `q` revolutions, where `q` is the denominator of the fraction.
  - **Irrational `n`** : The curve does not close, so a maximum number of revolutions (`max_revolutions_irrational`) is used.
- **`r` (Rolling Circle Radius):**
  - If provided, `n` is calculated as `R/r` .
  - Handles cases where `r > R` (internal rolling) or invalid values (e.g., `r <= 0`).

Special Cases

1. **Ellipse**:
   - Enforces `n=2` (rolling circle radius = `R/2`).
   - Default `d` values are set if not provided.
2. **Rose Curves**:
   - Sets `h = |R - r|` to create rose-like patterns.
3. **Hypocycloids**:
   - Sets `h = r` to create star-like patterns.
4. **Irrational `n`** :
   - Limits the animation to a fixed number of revolutions (`max_revolutions_irrational`).
5. **Invalid Inputs**:
   - Handles invalid or inconsistent inputs (e.g., negative radii, non-numeric values) with warnings or corrections.

In [ ]:
```
class Hypotrochoids:
    """
    Generates and animates multiple hypotrochoids formed by circles rolling
    inside a larger stationary circle.

    Handles integer, rational, and irrational ratios n = R/r, adjusting
    animation duration accordingly. Also handles special cases like
    hypocycloids, rose curves, and ellipses.
    """
    def __init__(self, R=4, d=None, n_values=None, r_values=None,
                 type='random', save_anim=False, filename=None,
                 frames_per_rev=100, fps=25,
                 max_revolutions_irrational=5):
        """
        Initializes the Hypotrochoids animation setup.

        Parameters:
        -----------
        R : float
            Radius of the stationary circle (default: 4). Also called 'a'.
        d : float or list[float], optional
```

```
        Distance(s) of the tracing point from the center of the rolling circle.
        Also called 'h'. If None, defaults are chosen based on type.
        If float, applied to all rolling circles.
        If list, must match the number of rolling circles.
    n_values : int, float, Fraction, or list, optional
        Determines rolling circle radii as r = R/n. Used if r_values is None.
        Accepts integers, floats, Fractions, or lists containing these.
        Affects animation duration:
         - Integer n: Closes in 1 revolution of the center.
         - Rational n=p/q: Closes in q revolutions of the center.
         - Irrational n: Animates for max_revolutions_irrational.
        (default: [3, 4, 5] for random type)
    r_values : int, float, or list, optional
        Explicit radii ('b' or 'r') for the rolling circles. Overrides n_values if provided.
        If int/float, converted to a list.
        Corresponding n values (R/r) will be calculated and used for animation duration.
    type : str, optional
        Type of curve to generate ('random', 'hypocycloid', 'rose', 'ellipse').
        Defaults to 'random'. Affects default parameters and calculations if d, n_values, r_values are None.
    save_anim : bool, optional
        Whether to save the animation to a file (default: False).
    filename : str, optional
        Name of the file to save the animation (required if save_anim is True).
        Default filenames are generated based on type if None.
    frames_per_rev : int, optional
        Number of frames per 2*pi revolution of the rolling circle's center (default: 100).
        Controls animation smoothness.
    fps : int, optional
        Frames per second for the animation (default: 25). Controls speed.
    max_revolutions_irrational : float or int, optional
        Number of 2*pi revolutions of the center to simulate for irrational n values.
        (default: 5)
    """
    self.R = float(R)
    self.a = self.R  # Use 'a' for radius of fixed circle in formulas
    if self.a <= 0:
         raise ValueError("Fixed circle radius 'R' must be positive.")
    self.type = type.lower()
    self.save_anim = save_anim
    self.filename = filename
    self.frames_per_revolution = int(frames_per_rev)
    self.FPS = fps
    self.max_revolutions_irrational = float(max_revolutions_irrational)

    # Lists to store calculated parameters for each curve
    self.n_values_input_type = [] # Store original input type for n (esp Fraction)
    self.n_values_numeric = [] # The numeric ratio a/b for each curve for calcs
    self.b_values = []  # Radius of rolling circle (r)
    self.h_values = []  # Distance of tracing point (d)
    self.n_types = []   # Type of n ('integer', 'rational', 'irrational')
    self.n_revs = []    # Revolutions needed for each curve to close/complete

    self._process_parameters(d, n_values, r_values)
    self._calculate_animation_parameters()

    # Plotting attributes to be initialized later
    self.fig = None
    self.ax = None
    self.circles_rolling = []
    self.path_lines = []
    self.radius_lines = []
    self.tracing_points = []
    self.center_points = []
    self.fixed_circle = None
    self.anim = None
    self.rev_text = None


@staticmethod
def _is_irrational(num, denom_threshold=100):
    """
    Approximate check for irrational numbers or complex rationals based on Fraction representation.
    Returns True if number seems irrational or is a rational with a large denominator.
    """
    # Handle obvious non-numeric types if they somehow sneak in
    if not isinstance(num, (int, float, Fraction)):
        return True # Treat unexpected types as irrational
    if isinstance(num, int): # Integers are rational
        return False
    if isinstance(num, Fraction): # Already a fraction
```

```
            # Check if denominator is large enough to be treated as complex/irrational for animation
            return num.denominator > denom_threshold

        # Handle Floats
        if not np.isfinite(num): # infinity or NaN
            return True
        if num == math.pi or num == math.e: # Known irrationals
            return True
        # Check via Fraction conversion
        try:
            f = Fraction(num).limit_denominator(10000) # Limit complexity
            # Check if denominator is large after limiting (suggests irrational or complex rational)
            if f.denominator > denom_threshold:
                return True
        except (OverflowError, ValueError):
            return True # Treat conversion errors as irrational
        return False # Assume rational otherwise

    def _process_parameters(self, d, n_values_in, r_values_in):
        """
        Determines rolling circle radii (b), tracing distances (h), n values,
        and n types based on inputs and curve type. Handles integer, rational,
        and irrational r_values similarly to n_values.
        """
        default_n = [3, 4, 5] # Default for random if nothing else specified
        processed_n_values = [] # Store numeric n for calculations
        processed_r_values = []
        n_values_input_type = [] # Store original input type for n (esp Fraction)
        r_was_irrational_flags = [] # Track if input r was likely irrational

        # --- Step 1: Determine the primary list of n or r values ---
        if r_values_in is not None:
            # Convert r_values to a list of floats
            if isinstance(r_values_in, (int, float)):
                input_r_list = [float(r_values_in)]
            else:
                input_r_list = [float(r) for r in r_values_in]

            # Calculate corresponding n_values and check r's nature
            temp_n_values = []
            valid_r_values = []
            for r in input_r_list:
                if r <= 0:
                    warnings.warn(f"Rolling radius r={r} must be positive. Skipping.")
                    continue
                if not np.isfinite(r):
                    warnings.warn(f"Rolling radius r={r} is not finite. Skipping.")
                    continue
                # Allow r > R for hypotrochoids (rolls internally, n < 1)
                if r > self.a and self.type != 'ellipse': # Allow r=a/2 for ellipse
                    warnings.warn(f"Rolling radius r={r:.3f} > R={self.a:.3f} (n<1). Rolling circle is larger tha
                if self.type == 'ellipse' and abs(self.a / r - 2.0) > 1e-6:
                    warnings.warn(f"Type is 'ellipse', requires R/r = 2. Adjusting r={r:.3f} to {self.a/2.0:.3f}
                    r = self.a / 2.0 # Correct r for ellipse type

                valid_r_values.append(r)
                n_numeric = self.a / r
                temp_n_values.append(n_numeric)
                # Check if the input r itself seems irrational/complex
                r_was_irrational_flags.append(self._is_irrational(r, denom_threshold=1000))

            processed_n_values = temp_n_values
            processed_r_values = valid_r_values
            if not processed_n_values:
                raise ValueError("No valid rolling circles could be determined from input r_values.")
            # Since r was input, we don't have an original 'n' input type list yet
            # We will create a placeholder or use numeric n later if needed for display
            n_values_input_type = processed_n_values # Use numeric n as stand-in for 'input type'

        else: # Use n_values_in
            input_n_list = []
            if n_values_in is None:
                # Use defaults based on type if n_values not given
                if self.type == 'ellipse':
                     # Special handling if 'd' is a list but n is not - generate multiple ellipses
                     if d is not None and isinstance(d, (list, tuple)):
                          input_n_list = [2.0] * len(d)
                     else:
                          input_n_list = [2.0] # Ellipse requires n=2 (a=2b)
                     if d is None: d = [self.R * 0.25, self.R * 0.5] # Example distances for ellipse
```

```
            elif self.type == 'rose':
                input_n_list = [3, 5/2, 7/3] # Example values for rose patterns
            elif self.type == 'hypocycloid':
                input_n_list = default_n
            else: # Random
                input_n_list = default_n
        elif isinstance(n_values_in, (int, float, Fraction)):
            input_n_list = [n_values_in]
        else: # It's a list
            input_n_list = list(n_values_in)

        # Validate n_values and calculate corresponding r (b) values
        temp_r_values = []
        valid_n_input_type = [] # Keep original type (like Fraction)
        valid_n_numeric = []

        for n_raw in input_n_list:
            n_numeric = None
            try:
                # Try converting to float first for numeric checks
                n_numeric = float(n_raw)
            except (TypeError, ValueError):
                warnings.warn(f"Invalid type for n value: {n_raw}. Skipping.")
                continue

            if n_numeric == 0:
                warnings.warn(f"n=0 detected, results in infinite radius 'b'. Skipping.")
                continue
            if not np.isfinite(n_numeric):
                 warnings.warn(f"n={n_numeric} is not finite. Skipping.")
                 continue
            # Allow n < 1 for hypotrochoids (internal rolling)
            if n_numeric < 1 and self.type != 'ellipse': # n=2 for ellipse is >= 1
                warnings.warn(f"n={n_numeric:.3f} < 1 detected (r > R). Rolling circle is larger than fixed c

            if self.type == 'ellipse' and abs(n_numeric - 2.0) > 1e-6:
                warnings.warn(f"Type is 'ellipse', requires n=R/r=2. Adjusting input n={n_numeric:.3f} to 2.0
                n_numeric = 2.0
                n_raw = 2.0 # Update raw value as well

            r_calc = self.a / n_numeric
            temp_r_values.append(r_calc)
            valid_n_input_type.append(n_raw) # Store original input form
            valid_n_numeric.append(n_numeric)# Store numeric form for calculations

        if not valid_n_numeric:
            raise ValueError("No valid rolling circles could be determined from input n_values.")

        processed_r_values = temp_r_values
        processed_n_values = valid_n_numeric # Store numeric values
        n_values_input_type = valid_n_input_type # Store input types separately

    num_circles = len(processed_r_values)
    self.b_values = processed_r_values
    # Ensure these attributes exist before assigning
    self.n_values_numeric = processed_n_values # Store numeric n for calcs/type checking
    self.n_values_input_type = n_values_input_type # Store input types separately


    # --- Step 2: Determine n_types and revolutions needed ---
    self.n_types = []
    self.n_revs = []
    # Use the numeric n values for determining type and revolutions
    for i, n_numeric in enumerate(self.n_values_numeric):
        n_type = "irrational" # Default assumption
        revolutions = self.max_revolutions_irrational
        is_forced_irrational = False
        # Get original input n if it exists (i.e., if n_values were input)
        n_input_val = self.n_values_input_type[i] if r_values_in is None else n_numeric

        # Check if r was flagged as irrational (only if r_values were the input source)
        if r_values_in is not None and i < len(r_was_irrational_flags) and r_was_irrational_flags[i]:
             print(f"Input r={self.b_values[i]:.4f} flagged as irrational/complex. "
                   f"Forcing n={n_numeric:.4f} to be treated as irrational."
                   )
             n_type = "irrational"
             revolutions = self.max_revolutions_irrational
             is_forced_irrational = True
        # Check if n was input directly as a float likely representing irrational
        elif r_values_in is None and isinstance(n_input_val, float) and self._is_irrational(n_input_val, den
```

```
                print(f"Input n={n_input_val:.4f} flagged as irrational/complex float. Forcing treatment as irra
                n_type = "irrational"
                revolutions = self.max_revolutions_irrational
                is_forced_irrational = True

        # Only proceed with int/rational checks if not forced to irrational
        if not is_forced_irrational:
            try:
                # Check if it's essentially an integer
                if np.isclose(n_numeric, round(n_numeric)):
                    n_int = round(n_numeric)
                    # For hypotrochoid, n=1 means b=a, rolling circle fills fixed circle - edge case.
                    # Let's treat n=1 as integer case needing 1 rev.
                    if n_int >= 1: # Valid integer n (or n=1 edge case)
                        n_type = "integer"
                        revolutions = 1
                    else:
                        # Handle cases like n=0.5 treated as integer 0 or 1 incorrectly
                        # Or n < 1 that are not close to 0 or 1. Treat as rational/irrational.
                        pass # Keep default irrational if rounded n < 1

                # ONLY check for rational IF it wasn't classified as integer
                elif not self._is_irrational(n_numeric, denom_threshold=1000):
                    is_rational_simple = False
                    # Check if input was explicitly a Fraction
                    if r_values_in is None and isinstance(n_input_val, Fraction):
                         # Check if it's a non-integer Fraction (denominator > 1)
                         if n_input_val.denominator > 1 and n_input_val.denominator <= 1000:
                             n_type = "rational"
                             revolutions = n_input_val.denominator
                             is_rational_simple = True
                         # If input was Fraction(X,1), treat as integer
                         elif n_input_val.denominator == 1:
                             # It was already classified as integer above, but confirm revs=1
                             n_type = "integer"
                             revolutions = 1
                             is_rational_simple = True # Mark as handled

                    # If not handled as an input Fraction, check the numeric value
                    if not is_rational_simple:
                        frac_n = Fraction(n_numeric).limit_denominator(1000)
                        p = frac_n.numerator
                        q = frac_n.denominator
                        # Check if it's a non-integer rational (denominator > 1)
                        if q > 1:
                             n_type = "rational"
                             revolutions = q
                        # If q == 1, it was already handled by the integer check above.
                        # If q <= 0, something is wrong, keep default irrational.
                # else: If neither integer nor simple rational, it remains irrational (the default)

            except (ValueError, OverflowError):
                # Error converting, treat as irrational
                n_type = "irrational"
                revolutions = self.max_revolutions_irrational

        self.n_types.append(n_type)
        self.n_revs.append(revolutions)

        # Print info of n interpretation
        n_display = n_input_val if r_values_in is None else n_numeric # Use input n if available else numeric

        if n_type == 'irrational':
            # Display original input if it was float, otherwise show numeric
            input_display_str = f"{float(n_display):.4f}" if isinstance(n_display, (float, np.floating)) else
            print(
                f"n={n_numeric:.4f} "
                f"(Input: ~{input_display_str}) "
                f"interpreted as {n_type}, "
                f"setting max revolutions to {revolutions:.2f}."
            )
        elif n_type=='rational':
            # For rational numbers, always display the fraction form clearly
            frac_n = Fraction(n_numeric).limit_denominator(1000)
            # Display original input if it was Fraction, otherwise show numeric
            input_display_str = str(n_input_val) if isinstance(n_input_val, Fraction) else f"{float(n_display
            print(
                f"n={float(n_numeric):.4f} "
                f"(Input: {input_display_str}, Approx Fraction: {frac_n.numerator}/{frac_n.denominator}) "
                f"interpreted as {n_type}, "
```

```
                f"setting max revolutions to {int(revolutions)}." # Revolutions for rational are integer
            )
        else: # integer
            # Display original input if it was int/Fraction(X,1), otherwise show numeric
            input_display_str = str(n_input_val) if isinstance(n_input_val, (int, Fraction)) else f"{float(n_
            print(
                f"n={int(n_numeric)} "
                f"(Input: {input_display_str}) "
                f"interpreted as {n_type}, "
                f"requires {revolutions} revolution(s)."
            )


    # --- Step 3: Determine d_values (tracing distances 'h') ---
    final_d_values = []
    if self.type == 'hypocycloid':
        final_d_values = list(self.b_values) # h = b
        if d is not None:
            warnings.warn(f"Type is 'hypocycloid', ignoring provided 'd'. Using d=r.")
    elif self.type == 'rose':
        # h = |a - b| creates specific rose-like patterns
        final_d_values = [abs(self.a - b) for b in self.b_values]
        if d is not None:
            warnings.warn(f"Type is 'rose', ignoring provided 'd'. Using d=|R - r|.")
    elif self.type == 'ellipse':
         # n=2 (a=2b) was enforced earlier. Use provided d.
        if d is None:
            # Default 'd' if none provided for ellipse
            default_ellipse_d = [b * 0.5 for b in self.b_values] # e.g., d = r/2 = R/4
            final_d_values = default_ellipse_d
            warnings.warn(f"Type is 'ellipse' but 'd' not provided. Using default d=r/2 (=R/4).")
        elif isinstance(d, (int, float)):
             final_d_values = [float(d)] * num_circles
        else: # d is a list
             d_float = [float(val) for val in d]
             if len(d_float) != num_circles:
                # If n_values wasn't specified, maybe adjust num_circles based on d
                if n_values_in is None and r_values_in is None:
                     num_circles = len(d)
                     # Need to resize n_values, b_values etc. assuming n=2
                     self.n_values_numeric = [2.0] * num_circles
                     self.n_values_input_type = [2.0] * num_circles
                     self.b_values = [self.a / 2.0] * num_circles
                     self.n_types = ["integer"] * num_circles
                     self.n_revs = [1] * num_circles
                    #  warnings.warn(f"Type is 'ellipse' and 'd' is a list. Neither n_values nor r_values pro
                                #    f"Adjusted number of curves to match length of 'd' ({num_circles}).")
                     final_d_values = [float(val) for val in d]
                else:
                    raise ValueError(f"Length of 'd' ({len(d_float)}) must match the number of rolling circle
                                     f"({num_circles}) for ellipse type when 'd' is a list and n/r values wer
                                     )
             else:
                 final_d_values = d_float
    else: # Random type
        if d is None:
            # Default d for random: use r/2 for variety
            final_d_values = [b * 0.5 for b in self.b_values]
            # warnings.warn(f"Parameter 'd' not provided for random type. Using default d=r/2.")
        elif isinstance(d, (int, float)):
            final_d_values = [float(d)] * num_circles
        else: # d is a list
            d_float = [float(val) for val in d]
            if len(d_float) != num_circles:
               # Adjust d list to match num_circles (repeat last or truncate)
               if len(d_float) < num_circles:
                   warnings.warn(f"Length of 'd' ({len(d_float)}) < number of circles ({num_circles}). Repeat
                   final_d_values = d_float + [d_float[-1]] * (num_circles - len(d_float))
               else: # len > num_circles
                   warnings.warn(f"Length of 'd' ({len(d_float)}) > number of circles ({num_circles}). Trunca
                   final_d_values = d_float[:num_circles]
            else:
               final_d_values = d_float

    # Ensure h (d) is not negative, store as self.h_values
    self.h_values = [abs(val) for val in final_d_values]

    # --- Step 4: Set default filename if needed ---
    if self.save_anim and self.filename is None:
        # Use numeric n for filename consistency
```

```
        n_info = "_".join([f"n{n:.3f}".replace('.','p') for n in self.n_values_numeric])
        h_info = "_".join([f"d{h:.2f}".replace('.', 'p') for h in self.h_values])
        # Maybe include r info as well for clarity if r was input
        r_info = "_".join([f"r{r:.3f}".replace('.','p') for r in self.b_values]) if r_values_in is not None e
        self.filename = f"{self.type}_R{self.R:.1f}_{n_info}_{h_info}{'_'+r_info if r_info else ''}.gif"

    def _calculate_animation_parameters(self):
        """
        Calculate padding, theta range, and frame count based on n_types.
        The animation runs long enough for the longest closing pattern or
        the max duration for any irrational patterns.
        """
        # Calculate padding for plot limits
        # Max distance from origin = center_dist + tracing_dist = |a-b| + h
        max_reach = max([abs(self.a - b) + h for b, h in zip(self.b_values, self.h_values)], default=self.a)
        # Add padding relative to the larger of fixed radius or max reach
        base_limit = max(self.a, max_reach)
        self.plot_padding = base_limit * 0.15 # Add 15% padding
        self.plot_limit = base_limit + self.plot_padding

        # Determine total angle needed based on the maximum revolutions required by any curve
        if not self.n_revs: # Handle case of no valid circles
             self.max_total_revolutions = 1
        else:
             self.max_total_revolutions = max(self.n_revs)

        print(f"Animation requires max {self.max_total_revolutions:.2f} revolutions of the center.")

        self.theta_max_pattern = 2 * np.pi * self.max_total_revolutions
        self.total_frames = int(self.max_total_revolutions * self.frames_per_revolution)
        if self.total_frames <= 0: self.total_frames = self.frames_per_revolution # Ensure minimum frames

        # Generate theta values for the animation
        self.theta_vals = np.linspace(0, self.theta_max_pattern, self.total_frames)

    @staticmethod
    def _hypotrochoid_points(theta, a, b, h):
        """
        Calculate hypotrochoid points for a given angle theta (or array of thetas).
        a = fixed radius, b = rolling radius, h = tracing distance.
        """
        if b == 0: # Avoid division by zero
             # Point traces circle of radius h around center of rolling circle,
             # which itself is fixed at distance 'a' if we interpret b=0 literally
             # Or point just stays at origin Let's assume it traces a circle around (a,0) - unlikely case.
             # More likely, n=inf -> b=0. Point on circumference (h=b=0) stays at (a,0).
             # If h != 0, it would rotate around (a,0)
             # Let's return a fixed point at origin for simplicity. Requires clearer definition for b=0.
             center_x = a # Center stays at R
             center_y = 0 * theta # Keep shape consistent with theta input
             trace_x = a + h * np.cos(0) # Fixed point relative to center
             trace_y = 0 * theta
             if isinstance(theta, (np.ndarray)): # Match shape for array input
                return np.full_like(theta, center_x), np.full_like(theta, center_y), \
                       np.full_like(theta, trace_x), np.full_like(theta, trace_y)
             else: # Scalar input
                return center_x, center_y, trace_x, trace_y

        # Center of rolling circle
        center_x = (a - b) * np.cos(theta)
        center_y = (a - b) * np.sin(theta)

        trace_angle_term = (a - b) / b * theta # Angle for the tracing point calculation
        # Equivalent to (n - 1) * theta, where n = a/b

        # Tracing point position relative to the fixed circle's center
        trace_x = center_x + h * np.cos(trace_angle_term)
        trace_y = center_y - h * np.sin(trace_angle_term) # Minus sign for internal rolling

        return center_x, center_y, trace_x, trace_y

    def _setup_plot(self):
        """Sets up the Matplotlib figure and axes."""
        plt.style.use('dark_background')
        self.fig, self.ax = plt.subplots(figsize=(10, 10))
        self.fig.subplots_adjust(left=0.05, right=0.95, top=0.9, bottom=0.05)
        # Use calculated limit
        self.ax.set_xlim(-self.plot_limit, self.plot_limit)
        self.ax.set_ylim(-self.plot_limit, self.plot_limit)
        self.ax.set_xticks([])
```

```
        self.ax.set_yticks([])
        self.ax.set_aspect('equal', adjustable='box')
        self.ax.grid(False)
        title = f"Hypotrochoids (R={self.R:.2f}, Type: {self.type.capitalize()})"
        self.ax.set_title(title, fontsize=14, pad=10)

        # Stationary circle
        self.fixed_circle = patches.Circle((0, 0), self.a, fill=False, color='gray', lw=1.5, ls='--')
        self.ax.add_patch(self.fixed_circle)

        # Use a colormap for distinct colors
        if len(self.b_values) == 1:
             colors = ['yellow'] # Single curve color
        else:
             colors = plt.get_cmap('rainbow')(np.linspace(0, 1, len(self.b_values)))

        # Create plot elements for each hypotrochoid
        self.circles_rolling = []
        self.path_lines = []
        self.radius_lines = []
        self.tracing_points = []
        self.center_points = []

        for i, (b, h, n_val, n_typ) in enumerate(zip(self.b_values, self.h_values, self.n_values_numeric, self.n_
            color = colors[i]

            # Rolling circle patch
            circle = patches.Circle((self.a - b, 0), b, fill=False, color=color, lw=1.5, alpha=0.8, zorder=3)
            self.ax.add_patch(circle)
            self.circles_rolling.append(circle)

            # Path line and label generation
            n_input = self.n_values_input_type[i] if i < len(self.n_values_input_type) else n_val
            if n_typ == "rational":
                 # Prefer original Fraction if input, else format the limited fraction
                frac_n = n_input if isinstance(n_input, Fraction) else Fraction(n_val).limit_denominator(1000)
                n_label = f"{frac_n.numerator}/{frac_n.denominator}"
            elif n_typ == "integer":
                 n_label = f"{int(round(n_val))}"
            else: # irrational
                 n_label = f"{n_val:.4f}"

            label = f'n={n_label}, r={b:.3f}, d={h:.3f}'
            path_line, = self.ax.plot([], [], '-', color=color, lw=2, label=label, zorder=2)
            self.path_lines.append(path_line)

            # Radius line (center to tracing point)
            radius_line, = self.ax.plot([], [], '--', color=color, lw=1.5, alpha=0.8, zorder=4)
            self.radius_lines.append(radius_line)

            # Tracing point marker
            tracing_point, = self.ax.plot([], [], 'o', color=color, ms=7, zorder=5)
            self.tracing_points.append(tracing_point)

            # Rolling circle center marker
            center_point, = self.ax.plot([], [], 'o', color='white', ms=5, alpha=0.6, zorder=4)
            self.center_points.append(center_point)

        if len(self.b_values) > 0: # Add legend only if there are curves
            self.ax.legend(loc='upper right', fontsize=9, facecolor='#1C1C1C', framealpha=0.7)

        # Add text annotation for revolution counter if animation > 1 rev
        if self.max_total_revolutions > 1.01: # Add threshold to avoid showing for exactly 1 rev
            self.rev_text = self.ax.text(0.02, 0.98, '', transform=self.ax.transAxes,
                                         bbox=dict(facecolor='#2e4053', alpha=0.7, boxstyle='round,pad=0.3'),
                                         color='white', fontsize=12, ha='left', va='top', zorder=10)


    def _init_animation(self):
        """Initializes animation elements."""
        all_elements = []
        for i in range(len(self.b_values)):
            # Reset lines to empty
            self.path_lines[i].set_data([], [])
            self.radius_lines[i].set_data([], [])
            self.tracing_points[i].set_data([], [])
            self.center_points[i].set_data([], [])
            # Reset circle position
            init_position = self._hypotrochoid_points(0, self.a, self.b_values[i], self.h_values[i])
            init_center_x, init_center_y, init_trace_x, init_trace_y = init_position
```

```python
            self.circles_rolling[i].center = (init_center_x, init_center_y)
            self.tracing_points[i].set_data([init_trace_x], [init_trace_y])
            self.center_points[i].set_data([init_center_x], [init_center_y])
            self.radius_lines[i].set_data([init_center_x, init_trace_x], [init_center_y, init_trace_y])

            all_elements.extend([
                self.path_lines[i], self.circles_rolling[i], self.radius_lines[i],
                self.tracing_points[i], self.center_points[i]
            ])

        # Update the text annotation for revolutions
        if self.rev_text is not None:
             # Display total as float if it's not an integer
            total_rev_display = (f"{int(self.max_total_revolutions)}"
                                 if np.isclose(self.max_total_revolutions, round(self.max_total_revolutions))
                                 else f"{self.max_total_revolutions:.2f}")
            self.rev_text.set_text(f'Revs: 0/{total_rev_display}')
            all_elements.append(self.rev_text)

        return all_elements

    def _animate_frame(self, frame):
        """Updates animation for a single frame."""
        # Current angle for this frame
        current_theta = self.theta_vals[frame]
        # History of angles up to this frame for drawing the path
        theta_history = self.theta_vals[:frame+1]

        plot_elements_to_update = []

        for i, (b, h) in enumerate(zip(self.b_values, self.h_values)):
            # Calculate positions for the entire history for the path line
            (centers_x_hist, centers_y_hist,
             traces_x_hist, traces_y_hist) = self._hypotrochoid_points(theta_history, self.a, b, h)

            # Get position for the current frame for markers and rolling circle
            # The last point in the history corresponds to the current frame
            if len(traces_x_hist) > 0:
                 current_center_x = centers_x_hist[-1]
                 current_center_y = centers_y_hist[-1]
                 current_trace_x = traces_x_hist[-1]
                 current_trace_y = traces_y_hist[-1]
            else: # Frame 0 case
                 (current_center_x, current_center_y,
                  current_trace_x, current_trace_y) = self._hypotrochoid_points(0, self.a, b, h)

            # Update plot elements
            self.circles_rolling[i].center = (current_center_x, current_center_y)
            self.path_lines[i].set_data(traces_x_hist, traces_y_hist)
            self.tracing_points[i].set_data([current_trace_x], [current_trace_y])
            self.center_points[i].set_data([current_center_x], [current_center_y])
            self.radius_lines[i].set_data([current_center_x, current_trace_x],
                                          [current_center_y, current_trace_y])

            plot_elements_to_update.extend([
                 self.path_lines[i], self.circles_rolling[i], self.radius_lines[i],
                 self.tracing_points[i], self.center_points[i]
            ])

        # Update revolution counter text
        if self.rev_text is not None:
            current_rev = current_theta // (2 * np.pi)
            # Format display based on whether total revs is integer or not
            total_rev_display = (f"{int(self.max_total_revolutions)}"
                                 if np.isclose(self.max_total_revolutions, round(self.max_total_revolutions))
                                 else f"{self.max_total_revolutions:.2f}")
            # Display current revs with precision
            self.rev_text.set_text(f'Revs: {int(current_rev)}/{total_rev_display}')
            plot_elements_to_update.append(self.rev_text)

        return plot_elements_to_update


    def generate_animation(self):
        """Creates and displays or saves the hypotrochoid animation."""
        if not self.b_values:
            print("No valid hypotrochoids to animate.")
            return None

        self._setup_plot()
```

```python
        if self.fig is None:
            raise ValueError("Figure object (self.fig) is not initialized.")


        # Create the animation object
        print(f"Generating animation with {self.total_frames} frames...")
        self.anim = FuncAnimation(self.fig, self._animate_frame, frames=self.total_frames,
                                  init_func=self._init_animation, interval=int(1000/self.FPS),
                                  blit=False,
                                  repeat=False)

        # Save or show the animation
        if self.save_anim:
            if not self.filename:
                print("Error: filename must be provided when save_anim is True.")
                plt.close(self.fig)
                return None

            # Ensure filename ends with .gif or .mp4 (default to gif)
            if not self.filename.lower().endswith(('.gif', '.mp4')):
                 self.filename += ".gif"

            save_dir = Path("ANIMATIONS/HYPOTROCHOIDS")
            save_dir.mkdir(parents=True, exist_ok=True)
            filepath = save_dir / self.filename

            print(f"Saving animation to {filepath.resolve()}...")
            writer_choice = 'pillow' if self.filename.lower().endswith('.gif') else 'ffmpeg'

            # Use tqdm for progress bar
            try:
                with tqdm(total=self.total_frames, desc="Saving Animation", unit="frame", ncols=100) as pbar:
                     def progress_update(current_frame, total_frames):
                          pbar.n = current_frame # Update tqdm progress
                          pbar.refresh() # Force redraw

                     self.anim.save(str(filepath), writer=writer_choice, fps=self.FPS,
                                    dpi=72, progress_callback=progress_update)
                # Ensure the progress bar finishes at 100%
                if pbar.n < self.total_frames:
                    pbar.update(self.total_frames - pbar.n)
                print("\nAnimation saved successfully!")
            except Exception as e:
                print(f"\nError saving animation: {e}")
                print("Ensure 'ffmpeg' (for MP4) or 'Pillow' (for GIF) is installed.")

            plt.close(self.fig) # Close the plot window after saving
        else:
            plt.show() # Display the animation interactively

        return self.anim # Return the animation object
```

Random Case

In [ ]:
```python
# --- Random Case Example (Default behavior if type not specified) ---
# d is fixed, radii of the rolling circles are varied
print("\nGenerating Random Hypotrochoids...")
random_anim = Hypotrochoids(R=5, d=1, n_values=[3, 5, 7],
                            save_anim=True, # filename automatically generated, if not specified
                            frames_per_rev=400, fps=50)

# random_anim.generate_animation() # Integer n values, 1 rev each


# --- Random Example with float n and custom r ---
print("\nGenerating Random Hypotrochoids with float n and specified r...")
custom_random = Hypotrochoids(R=5, r_values=[1.5, 1, math.pi], # n = 5/1.5=10/3, n=5/1=5, n=5/pi=1.5915..
                              d=[0.5, 1, 1.5], type='random',
                              save_anim=True,
                              filename="hypotrochoids_random_n.gif",
                              max_revolutions_irrational=5)
custom_random.generate_animation() # Should run for 5 revs as the max_revolutions_irrational is set to 5
```

Special Cases

Hypocycloids

In [ ]:
```
# --- Integer n ---
print("\nGenerating Hypocycloid (Integer n=3)...")
hypo_int = Hypotrochoids(R=5, n_values=3, type='hypocycloid', save_anim=False)
# hypo_int.generate_animation() # Should complete in 1 revolution

# Multiple curves with integer n
print("\nGenerating Hypocycloid (Integer n=3, 4, 5)...")
hypo_anim = Hypotrochoids(R=4, n_values=[3, 4, 5], type='hypocycloid', save_anim=False)
# hypo_anim.generate_animation()

# --- Rational n ---
print("\nGenerating Hypocycloid (Rational n=3.5 = 7/2)...")
hypo_rational = Hypotrochoids(R=4, n_values=Fraction(7, 2), type='hypocycloid',
                              save_anim=False)
# hypo_rational.generate_animation() # Should complete in 2 revolutions

print("\nGenerating Hypocycloid (Rational n=2.25 = 9/4)...")
hypo_rational_2 = Hypotrochoids(R=6, n_values=2.25, type='hypocycloid',
                              save_anim=True,
                              fps=40)
# hypo_rational_2.generate_animation() # Should complete in 4 revolutions

# Multiple curves with rational n
print("\nGenerating Mixed Hypocycloids with rational n (n=5/3, 7/4, 9/5)...")
hypo_rational_mixed = Hypotrochoids(R=4, n_values=[5/2, Fraction(11,4), 18/5],
                                    type='hypocycloid', save_anim=True, fps=50)
# hypo_rational_mixed.generate_animation() # Should complete in 2, 4 and 5 revolutions respectively

# --- Irrational n ---
print("\nGenerating Hypocycloids (Irrational n=pi)...")
hypo_irrational = Hypotrochoids(R=4, n_values=math.pi, type='hypocycloid',
                              max_revolutions_irrational=4, save_anim=False)
# hypo_irrational.generate_animation() # Should run for 4 revolutions

# --- Multiple Curves with Mixed n Types ---
print("\nGenerating Mixed Hypocycloids (n=3, 7/2, pi)...")
mixed_anim = Hypotrochoids(R=5, n_values=[3, Fraction(7,2), math.pi], type='hypocycloid',
                           max_revolutions_irrational=5, save_anim=False, fps=40)
mixed_anim.generate_animation() # Revolutions depend on the max_revolutions_irrational number (longest duration r
```

Ellipse

In [ ]:
```
# --- Ellipse Example (n must be 2) with Rational/Irrational d ---
print("\nGenerating Ellipses (n=2 forced), different d values...")
# Note: n=2 is integer, duration is 1 rev. 'd' doesn't affect duration.
ellipse_anim = Hypotrochoids(R=6, d=[0.5, 1.25, np.sqrt(3)], type='ellipse',
                              save_anim=True, filename="ellipses_example.gif")
ellipse_anim.generate_animation()

# Single ellipse, specifying n=2 explicitly
ellipse_anim_single = Hypotrochoids(R=4, d=1, n_values=2, type='ellipse', save_anim=False)
# ellipse_anim_single.generate_animation()
```

Rose

In [ ]:
```
 # --- Rose Example (Integer n) ---
print("\nGenerating Rose Curves (n=3, 4, 5)...")
# Note: n=3,4,5 are integers, duration is 1 rev. 'd' doesn't affect duration.
rose_anim = Hypotrochoids(R=4, n_values=[3, 7, 5], type='rose', frames_per_rev=400, fps=50,
                           save_anim=True, filename="rose_integer_2.gif")
# rose_anim.generate_animation() # Should run for 1 revolution

print("\nGenerating Rose Curves for n = 4/3, 5/4, 7/5...")
# Note: n=4/3, 5/4, 7/5 are rational, duration is 3 revs, 4 revs and 5 revs respectively.
rose_anim_rational = Hypotrochoids(R=4, n_values=[Fraction(4,3), Fraction(5,4), Fraction(7,5)],
                                    type='rose', frames_per_rev=100, fps=50,
                                    save_anim=True,
                                    filename="rose_rational.gif")
rose_anim_rational.generate_animation() # Should run for 3, 4 and 5 revolutions respectively
```

Observations

1. Derivation of the Rose Curve from Hypotrochoid

Starting from the standard hypotrochoid (rolling circle of radius b inside fixed circle of radius a, tracing point at distance h from the rolling-circle center),

$$\begin{cases} x(t) \;=\; (a-b)\,\cos t \;+\; h\,\cos\!\left(\frac{a-b}{b}\,t\right), \\ y(t) \;=\; (a-b)\,\sin t \;-\; h\,\sin\!\left(\frac{a-b}{b}\,t\right). \end{cases}$$

We now set

$$h \;=\; a-b.$$

Define

$$n \;=\; \frac{a-b}{b},$$

so that $h = bn$ and also $n = a/b - 1$. The curve becomes

$$\begin{aligned} x(t) &= (a-b)\big[\cos t + \cos(nt)\big], \\ y(t) &= (a-b)\big[\sin t - \sin(nt)\big]. \end{aligned}$$

We next use the sum-to-product identities

$$\begin{aligned} \cos t + \cos(nt) &= 2\,\cos\!\left(\tfrac{t+nt}{2}\right)\,\cos\!\left(\tfrac{t-nt}{2}\right) \;=\; 2\cos\!\left(\tfrac{n+1}{2}t\right)\,\cos\!\left(\tfrac{1-n}{2}t\right), \\ \sin t - \sin(nt) &= 2\,\cos\!\left(\tfrac{t+nt}{2}\right)\,\sin\!\left(\tfrac{t-nt}{2}\right) \;=\; 2\cos\!\left(\tfrac{n+1}{2}t\right)\,\sin\!\left(\tfrac{1-n}{2}t\right). \end{aligned}$$

So

$$\big(x(t),\,y(t)\big) \;=\; 2(a-b)\,\cos\!\left(\tfrac{n+1}{2}t\right)\left(\cos\tfrac{1-n}{2}t,\;\sin\tfrac{1-n}{2}t\right).$$

If we now introduce a new "angle" parameter

$$\varphi \;=\; \frac{1-n}{2}\,t,$$

then

$$t \;=\; \frac{2}{1-n}\,\varphi \quad\Longrightarrow\quad \frac{n+1}{2}\,t \;=\; \frac{n+1}{2}\cdot\frac{2\varphi}{1-n} \;=\; \frac{n+1}{1-n}\,\varphi.$$

Hence in polar form $(x,y) = \big(r(\varphi)\cos\varphi,\; r(\varphi)\sin\varphi\big)$ we get

$$r(\varphi) \;=\; 2\,(a-b)\,\cos\!\left(\tfrac{n+1}{1-n}\,\varphi\right).$$

But

$$\frac{n+1}{1-n} \;=\; \frac{\frac{a-b}{b}+1}{1-\frac{a-b}{b}} \;=\; \frac{\frac{a}{b}}{\frac{2b-a}{b}} \;=\; \frac{a}{2b-a} \;=:\; m.$$

Thus

$$\boxed{r(\varphi) \;=\; 2\,(a-b)\,\cos\!\big(m\,\varphi\big), \quad m \;=\; \frac{a}{2b-a}}$$

which is exactly the polar equation of a rose curve (with m "petals" if m is odd, or $2m$ petals if m is even).

Key steps in brief

1. **Set** $h = a - b$ in the hypotrochoid formulas.
2. **Factor** and apply sum-to-product to combine the two cosines (and two sines).
3. **Reparametrize** by $\varphi = (1-n)t/2$.
4. **Read off** $r(\varphi) = R\cos(m\varphi)$: the standard rose curve.

2. Number of Petals and Revolutions

Let's now break down the connection between the derivation above, the `Hypotrochoids` class, and the results we're seeing. There are two main points to clarify: the calculation of the rose curve parameter m and the meaning of the "revolutions" calculated by the class.

1. Calculating the Rose Parameter 'm' from Class 'n'

The derivation correctly establishes that for a hypotrochoid with the specific condition $h = a - b$, the resulting curve can be described in polar coordinates by $r(\varphi) = R\cos(m\varphi)$, where $R = 2(a-b)$.

The key is the relationship between the parameter m in the rose equation and the parameter n used in the `Hypotrochoids` class (where $n = a/b$, or `R/r` in the code).

From the derivation: $m = \frac{a}{2b-a}$

Now, substitute $b = a/n$ (since $n = a/b$ in the class): $m = \frac{a}{2(a/n)-a} = \frac{a}{a(2/n-1)} = \frac{1}{(2-n)/n} = \frac{n}{2-n}$

So, the correct formula to get the rose parameter m from the class parameter n is $m = \frac{n}{2-n}$.

Let's re-evaluate few examples using this correct relationship:

- **Class n = 4:**
  - $m = \frac{4}{2-4} = \frac{4}{-2} = -2$.
  - The rose equation is $r(\varphi) \propto \cos(-2\varphi) = \cos(2\varphi)$.
  - Here, the rose parameter k (from the standard $r = \cos(k\varphi)$) is $k = 2$. Since $k = 2$ is an *even integer*, the number of petals is $2k = 2 \times 2 = 4$.
  - **Our Observation:** We get 4 petals. This **matches** the calculation based on the derivation.
- **Class n = 3:**
  - $m = \frac{3}{2-3} = \frac{3}{-1} = -3$.
  - The rose equation is $r(\varphi) \propto \cos(-3\varphi) = \cos(3\varphi)$.
  - Here, $k = 3$. Since $k = 3$ is an *odd integer*, the number of petals is $k = 3$.
  - **Our Observation:** We get 3 petals. This **matches** the calculation based on the derivation.
- **Class n = 4/3:**
  - $m = \frac{4/3}{2-4/3} = \frac{4/3}{(6-4)/3} = \frac{4/3}{2/3} = 2$.
  - The rose equation is $r(\varphi) \propto \cos(2\varphi)$.
  - Here, $k = 2$. Since $k = 2$ is an *even integer*, the number of petals is $2k = 2 \times 2 = 4$.
  - **Our Observation:** We get 4 petals. This **matches** the calculation.
- **Class n = 7/5:**
  - $m = \frac{7/5}{2-7/5} = \frac{7/5}{(10-7)/5} = \frac{7/5}{3/5} = \frac{7}{3}$.
  - The rose equation is $r(\varphi) \propto \cos(7/3\varphi)$.
  - Here, $k = 7/3$. This is rational, $k = p/q$ with $p = 7, q = 3$. Since $p = 7$ is *odd*, the number of petals is $p = 7$.
  - **Our Observation:** We get 7 petals. This **matches** the calculation.

Conclusion on Petals: The number of petals generated by the class when `type='rose'` (which sets $h = |a - b|$) is fully consistent with the derivation when the relationship $m = n/(2-n)$ is used correctly.

2. Understanding the Number of Revolutions

The second part of the confusion is the number of revolutions.

- **Class Calculation:** The `Hypotrochoids` class calculates the animation duration based on the properties of the **hypotrochoid** curve itself, specifically how many revolutions the *center of the rolling circle* needs to make for the full path to close (or repeat). The rule implemented is: if $n = a/b = p/q$ (in simplest fractional form), the hypotrochoid path closes after q revolutions of the center around the origin. This is calculated in the `_process_parameters` method and stored in `self.n_revs`.
  - For $n = 4/3$, $q = 3$. The class animates for 3 revolutions of the center.
  - For $n = 7/5$, $q = 5$. The class animates for 5 revolutions of the center.
- **Rose Curve Tracing:** The rose curve $r \propto \cos(m\varphi)$ is traced in the polar angle φ. The number of revolutions in φ needed to trace the full rose depends on m.
  - For integer $m = k$, the curve $r = \cos(k\varphi)$ traces fully as φ goes from 0 to π (if k is even) or 0 to 2π (if k is odd).
  - For rational $m = p/q$, the curve $r = \cos(p/q\varphi)$ traces fully as φ goes from 0 to $q\pi$ (if p odd) or 0 to $2q\pi$ (if p even). (Assuming p/q is reduced).
- **Connecting the Revolutions:** The key is the relationship between the hypotrochoid parameter t and the rose angle φ: $\varphi = \frac{1-n_{deriv}}{2}t = \frac{2-n_{class}}{2}t$.
  - **n=4/3:** The class runs for $q = 3$ revolutions of the center, meaning t goes from 0 to $2\pi \times 3 = 6\pi$. In this time, $\varphi = \frac{2-4/3}{2}(6\pi) = \frac{2/3}{2}(6\pi) = \frac{1}{3}(6\pi) = 2\pi$. The rose curve is $\cos(2\varphi)$. As φ goes from 0 to 2π, the 4-petal pattern (which completes in $\varphi \in [0, \pi]$) is traced twice.

- **n=7/5:** The class runs for $q = 5$ revolutions of the center, meaning t goes from 0 to $2\pi \times 5 = 10\pi$. In this time, $\varphi = \frac{2-7/5}{2}(10\pi) = \frac{(10-7)/5}{2}(10\pi) = \frac{3/5}{2}(10\pi) = \frac{3}{10}(10\pi) = 3\pi$. The rose curve is $\cos(7/3\varphi)$. As φ goes from 0 to 3π, the 7-petal pattern (which requires a range of $q\pi = 3\pi$ since $p = 7$ is odd) is traced exactly once.

Summary:

- The number of **petals** we observe in the class for `type='rose'` correctly matches the theoretical rose curve $r \propto \cos(m\varphi)$ derived from the hypotrochoid under the condition $h = a - b$, where $m = n/(2-n)$.
- The number of **revolutions** reported by the class (e.g., 3 revs for n=4/3, 5 revs for n=7/5) refers to the revolutions of the *rolling circle's center* required for the *hypotrochoid path* to close, determined by the denominator q when $n = p/q$. This animation duration is sufficient to fully trace the derived rose curve pattern.

Derivations

The parametric equations for a hypotrochoid are:

$$x(\theta) = (R-r)\cos\theta + d\cos\left(\frac{R-r}{r}\theta\right)$$

$$y(\theta) = (R-r)\sin\theta - d\sin\left(\frac{R-r}{r}\theta\right)$$

where θ is the angle of the line connecting the centers of the two circles with the positive x-axis.

Let $k = R/r$. The equations can be written as:

$$x(\theta) = r(k-1)\cos\theta + d\cos((k-1)\theta)$$

$$y(\theta) = r(k-1)\sin\theta - d\sin((k-1)\theta)$$

The curve closes if $k = R/r$ is a rational number. Let $k = p/q$ where p and q are coprime positive integers. The curve closes after θ ranges from 0 to $2\pi q$.

1. Arc Length of a Hypotrochoid

1. **Find the derivatives:** We need $\frac{dx}{d\theta}$ and $\frac{dy}{d\theta}$.

$$\begin{aligned}\frac{dx}{d\theta} &= -(R-r)\sin\theta + d\left(-\frac{R-r}{r}\sin\left(\frac{R-r}{r}\theta\right)\right)\\ &= -(R-r)\left[\sin\theta + \frac{d}{r}\sin\left(\frac{R-r}{r}\theta\right)\right]\end{aligned}$$

$$\begin{aligned}\frac{dy}{d\theta} &= (R-r)\cos\theta - d\left(\frac{R-r}{r}\cos\left(\frac{R-r}{r}\theta\right)\right)\\ &= (R-r)\left[\cos\theta - \frac{d}{r}\cos\left(\frac{R-r}{r}\theta\right)\right]\end{aligned}$$

2. **Compute the sum of squares:**

$$\begin{aligned}&\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2\\ &= (R-r)^2\left[\sin\theta + \frac{d}{r}\sin\left(\frac{R-r}{r}\theta\right)\right]^2 + (R-r)^2\left[\cos\theta - \frac{d}{r}\cos\left(\frac{R-r}{r}\theta\right)\right]^2\\ &= (R-r)^2\left[\sin^2\theta + 2\frac{d}{r}\sin\theta\sin\left(\frac{R-r}{r}\theta\right) + \left(\frac{d}{r}\right)^2\sin^2\left(\frac{R-r}{r}\theta\right) + \cos^2\theta - 2\frac{d}{r}\cos\theta\cos\left(\frac{R-r}{r}\theta\right) + \left(\frac{d}{r}\right)^2\cos^2\left(\frac{R-r}{r}\theta\right)\right]\end{aligned}$$

Group terms using $\sin^2\alpha + \cos^2\alpha = 1$ and $\cos(\alpha+\beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$:

$$\begin{aligned}&= (R-r)^2\left[(\sin^2\theta + \cos^2\theta) + \left(\frac{d}{r}\right)^2\left(\sin^2\left(\frac{R-r}{r}\theta\right) + \cos^2\left(\frac{R-r}{r}\theta\right)\right) - 2\frac{d}{r}\left(\cos\theta\cos\left(\frac{R-r}{r}\theta\right) - \sin\theta\sin\left(\frac{R-r}{r}\theta\right)\right)\right]\\ &= (R-r)^2\left[1 + \left(\frac{d}{r}\right)^2 - 2\frac{d}{r}\cos\left(\theta + \frac{R-r}{r}\theta\right)\right]\\ &= (R-r)^2\left[1 + \left(\frac{d}{r}\right)^2 - 2\frac{d}{r}\cos\left(\frac{R}{r}\theta\right)\right]\end{aligned}$$

3. **Set up the arc length integral:** The arc length L is given by:

$$\boxed{L = \int_\alpha^\beta \sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2}\, d\theta}$$

Assuming $R > r$, we have $|R - r| = R - r$.

$$L = \int_\alpha^\beta (R-r)\sqrt{1 + \left(\frac{d}{r}\right)^2 - 2\frac{d}{r}\cos\left(\frac{R}{r}\theta\right)}\, d\theta$$

For the total length of a closed hypotrochoid where $R/r = p/q$, the interval is $[0, 2\pi q]$.

$$\boxed{L_{\text{total}} = (R-r)\int_0^{2\pi q} \sqrt{1 + \left(\frac{d}{r}\right)^2 - 2\frac{d}{r}\cos\left(\frac{p}{q}\theta\right)}\, d\theta}$$

4. **Evaluation:** This integral is generally **not expressible in terms of elementary functions**. It is a form of an elliptic integral. Therefore, unlike the hypocycloid (where $d = r$), there is no simple closed-form formula for the arc length of a general hypotrochoid.

   Special Case: Hypocycloid (d=r) If we set $d = r$, the expression under the square root becomes:

$$1 + \left(\frac{r}{r}\right)^2 - 2\frac{r}{r}\cos\left(\frac{R}{r}\theta\right) = 1 + 1 - 2\cos\left(\frac{R}{r}\theta\right) = 2\left(1 - \cos\left(\frac{R}{r}\theta\right)\right)$$

   Using the half-angle identity $1 - \cos(2\alpha) = 2\sin^2(\alpha)$, with $2\alpha = \frac{R}{r}\theta$:

$$= 2\left(2\sin^2\left(\frac{R}{2r}\theta\right)\right) = 4\sin^2\left(\frac{R}{2r}\theta\right)$$

   The square root is $\sqrt{4\sin^2\left(\frac{R}{2r}\theta\right)} = 2\left|\sin\left(\frac{R}{2r}\theta\right)\right|$. The integrand becomes $(R-r)\cdot 2\left|\sin\left(\frac{R}{2r}\theta\right)\right|$, which matches the integrand derived for the hypocycloid. Integrating this from 0 to $2\pi q$ yields the result $L = 8q(R-r)$, confirming consistency.

Conclusion for Arc Length: The arc length of a hypotrochoid over an interval $[\alpha, \beta]$ is given by the integral

$$L = (R-r)\int_\alpha^\beta \sqrt{1 + (d/r)^2 - 2(d/r)\cos(R\theta/r)}\, d\theta$$

. This integral generally requires numerical methods or elliptic functions to evaluate, except for the specific case $d = r$ (hypocycloid).

2. Area Enclosed by a Hypotrochoid

We use Green's Theorem in the form

$$\boxed{A = \frac{1}{2}\oint (x\,dy - y\,dx) = \frac{1}{2}\int_0^{2\pi q}\left(x(\theta)\frac{dy}{d\theta} - y(\theta)\frac{dx}{d\theta}\right) d\theta}$$

.

1. **Calculate the integrand $x\frac{dy}{d\theta} - y\frac{dx}{d\theta}$:**

   Let $A = R - r$ and $B = (R-r)/r$. Therefore,

$$\begin{aligned} x &= A\cos\theta + d\cos(B\theta) \\ y &= A\sin\theta - d\sin(B\theta) \\ \frac{dx}{d\theta} &= -A\sin\theta - dB\sin(B\theta) \\ \frac{dy}{d\theta} &= A\cos\theta - dB\cos(B\theta) \end{aligned}$$

$$\begin{aligned} y\frac{dx}{d\theta} &= (A\sin\theta - d\sin(B\theta))(-A\sin\theta - dB\sin(B\theta)) \\ &= -A^2\sin^2\theta - AdB\sin\theta\sin(B\theta) + Ad\sin(B\theta)\sin\theta + d^2B\sin^2(B\theta) \\ &= -A^2\sin^2\theta + Ad(1-B)\sin\theta\sin(B\theta) + d^2B\sin^2(B\theta) \end{aligned}$$

$$\begin{aligned} x\frac{dy}{d\theta} &= (A\cos\theta + d\cos(B\theta))(A\cos\theta - dB\cos(B\theta)) \\ &= A^2\cos^2\theta - AdB\cos\theta\cos(B\theta) + Ad\cos(B\theta)\cos\theta - d^2B\cos^2(B\theta) \\ &= A^2\cos^2\theta + Ad(1-B)\cos\theta\cos(B\theta) - d^2B\cos^2(B\theta) \end{aligned}$$

Now calculating the integrands

$$\begin{aligned}
x\frac{dy}{d\theta} - y\frac{dx}{d\theta} &= (A^2\cos^2\theta + Ad(1-B)\cos\theta\cos(B\theta) - d^2B\cos^2(B\theta)) \\
&\quad - (-A^2\sin^2\theta + Ad(1-B)\sin\theta\sin(B\theta) + d^2B\sin^2(B\theta)) \\
&= A^2(\cos^2\theta + \sin^2\theta) - d^2B(\cos^2(B\theta) + \sin^2(B\theta)) \\
&\quad + Ad(1-B)(\cos\theta\cos(B\theta) - \sin\theta\sin(B\theta)) \\
&= A^2 - d^2B + Ad(1-B)\cos(\theta + B\theta)
\end{aligned}$$

Substitute back $A = R - r$ and $B = (R-r)/r$, we have:

$$\begin{aligned}
\theta + B\theta &= \theta + \frac{R-r}{r}\theta = \frac{r+R-r}{r}\theta = \frac{R}{r}\theta \\
1 - B &= 1 - \frac{R-r}{r} = \frac{r-(R-r)}{r} = \frac{2r-R}{r}
\end{aligned}$$

and the integrand now becomes

$$\begin{aligned}
x\frac{dy}{d\theta} - y\frac{dx}{d\theta} &= (R-r)^2 - d^2\frac{R-r}{r} + (R-r)d\frac{2r-R}{r}\cos\left(\frac{R}{r}\theta\right) \\
&= (R-r)\left[(R-r) - \frac{d^2}{r} + \frac{d(2r-R)}{r}\cos\left(\frac{R}{r}\theta\right)\right]
\end{aligned}$$

Let $k = R/r$. $R - r = r(k-1)$, $R = kr$.

$$\begin{aligned}
&= r(k-1)\left[r(k-1) - \frac{d^2}{r} + \frac{d(2r-kr)}{r}\cos(k\theta)\right] \\
&= r^2(k-1)^2 - d^2(k-1) + rd(k-1)(2-k)\cos(k\theta)
\end{aligned}$$

2. **Integrate over $[0, 2\pi q]$:**

$$A = \frac{1}{2}\int_0^{2\pi q}\left[r^2(k-1)^2 - d^2(k-1) + rd(k-1)(2-k)\cos(k\theta)\right]d\theta$$

The integral of $\cos(k\theta) = \cos((p/q)\theta)$ over $[0, 2\pi q]$ is:

$$\int_0^{2\pi q}\cos\left(\frac{p}{q}\theta\right)d\theta = \left[\frac{q}{p}\sin\left(\frac{p}{q}\theta\right)\right]_0^{2\pi q} = \frac{q}{p}\sin(2\pi p) - \frac{q}{p}\sin(0) = 0 \quad (\text{if } p \neq 0)$$

So the cosine term integrates to zero.

$$\begin{aligned}
A &= \frac{1}{2}\int_0^{2\pi q}\left[r^2(k-1)^2 - d^2(k-1)\right]d\theta \\
&= \frac{1}{2}\left[r^2(k-1)^2 - d^2(k-1)\right][\theta]_0^{2\pi q} \\
&= \frac{1}{2}[r^2(k-1)^2 - d^2(k-1)](2\pi q) \\
&= \pi q(k-1)[r^2(k-1) - d^2]
\end{aligned}$$

3. **Final Formula and Interpretation:** The area enclosed by the hypotrochoid is:

$$\boxed{A = \pi q(k-1)[r^2(k-1) - d^2]}$$

where $k = R/r = p/q$ in simplest form.

We can rewrite this in terms of R and r: $k - 1 = (R-r)/r$.

$$\begin{aligned}
A &= \pi q\left(\frac{R-r}{r}\right)\left[r^2\left(\frac{R-r}{r}\right) - d^2\right] \\
&= \pi q\left(\frac{R-r}{r}\right)[r(R-r) - d^2] \\
&= \frac{\pi q(R-r)}{r}[rR - r^2 - d^2]
\end{aligned}$$

Orientation: The sign of the area depends on the parameters and the direction of parametrization. If $r^2(k-1) < d^2$, the area can be negative. The geometric area is the absolute value $|A|$. For $k > 1$ ($R > r$), $k - 1 > 0$. The sign depends on $r^2(k-1) - d^2$. If d is large enough ($d > r\sqrt{k-1} = \sqrt{r(R-r)}$), the orientation reverses.

Special Case: Hypocycloid (d=r) Substitute $d = r$ into the formula:

$$A = \pi q(k-1)[r^2(k-1) - r^2] = \pi q(k-1)r^2[k-1-1] = \pi q r^2(k-1)(k-2)$$

This matches the hypocycloid area formula derived before: $\pi q(R-r)(R-2r)$. If k is an integer, $q=1$, giving $A=\pi r^2(k-1)(k-2)=\pi(R-r)(R-2r)$.

3. Discussion of Special Cases

1. **Hypocycloid ($d=r$):**
   - As shown above, the hypotrochoid formulas for arc length (integrand) and area correctly reduce to the hypocycloid formulas when $d=r$.
   - Hypocycloids are characterized by cusps where the tracing point touches the fixed circle. The number of cusps is p when $R/r=p/q$.
2. **Ellipse ($R=2r$ or $k=2$):**
   - Let's examine the hypotrochoid equations when $k=2$: $R-r=2r-r=r$ and $\frac{R-r}{r}=\frac{r}{r}=1$.

$$x(\theta)=r\cos\theta+d\cos(\theta)=(r+d)\cos\theta$$
$$y(\theta)=r\sin\theta-d\sin(\theta)=(r-d)\sin\theta$$

   - These are the parametric equations of an **ellipse** centered at the origin with semi-axes $a=|r+d|$ and $b=|r-d|$. The area of this ellipse would be $\pi|r^2-d^2|$ which can also be derived from the area of a Hypotrochoid by setting $k=2$.
   - If $d=r$, then $y(\theta)=0$, and $x(\theta)=2r\cos\theta$. This is the degenerate hypocycloid (*Tusi couple*) tracing a line segment of length $4r$ (diameter of the fixed circle) back and forth. The formulas give:
     - Arc Length (total traced path): $L=8q(R-r)=8(1)(2r-r)=8r$. (Correct, $4r$ out and $4r$ back).
     - Area: $A=\pi qr^2(k-1)(k-2)=\pi(1)r^2(2-1)(2-2)=0$. (Correct, the line segment encloses no area).
   - If $d\neq r$, it's a non-degenerate ellipse. Let's check the area formula for $k=2$ ($q=1$): $A=\pi q(k-1)[r^2(k-1)-d^2]=\pi(1)(2-1)[r^2(2-1)-d^2]=\pi(r^2-d^2)$. The standard area of an ellipse is $\pi ab=\pi|r+d||r-d|=\pi|(r+d)(r-d)|=\pi|r^2-d^2|$. Our formula gives $\pi(r^2-d^2)$. The sign indicates the orientation (clockwise vs. counter-clockwise depending on whether $r>d$ or $r<d$). The magnitude matches the geometric area of the ellipse.
3. **Rose Curves:**
   - Rose curves are typically defined in polar coordinates by $\rho=a\cos(n\phi)$ or $\rho=a\sin(n\phi)$. They consist of petals centered around the origin.
   - Hypotrochoids are generally *not* centered at the origin (unless $R=r$, which is trivial, or the center is shifted).
   - While some hypotrochoids, particularly curtate ones ($d<r$) where loops form, might visually resemble the petals of a rose curve, their mathematical definition and properties are different. Hypotrochoids arise from the rolling motion (they are roulettes), whereas rose curves arise from a sinusoidal relationship between the radial distance and the angle in polar coordinates.
   - Therefore, a hypotrochoid is generally **not** a rose curve, although visual similarities may occur for specific parameter choices.

In summary:

- The **arc length** of a general hypotrochoid involves an elliptic integral without a simple closed form, but it reduces to $8q(R-r)$ for the hypocycloid case ($d=r$).
- The **area** enclosed by a hypotrochoid has a closed-form formula: $A=\pi q(k-1)[r^2(k-1)-d^2]$.
- A **hypocycloid** is a special case of a hypotrochoid ($d=r$).
- An **ellipse** is a special case of a hypotrochoid ($R=2r$, $d\neq r$).
- A **rose curve** is a distinct type of curve, not generally generated as a hypotrochoid.

Epitrochoids

Overview

An **epitrochoid** is a type of plane curve generated by a point attached to a circle (called the *rolling circle*) as it rolls **without slipping around the outside** of a larger fixed circle (called the *base circle*). The tracing point can be located either inside, on, or outside the circumference of the rolling circle, which gives rise to different shapes of the curve.

Epitrochoids are a generalization of **epicycloids**, where the tracing point is not restricted to the circumference of the rolling circle. These curves are widely studied in mathematics and have applications in engineering, physics, and art (e.g., spirographs).

Mathematical Representation

Parametric Equations

The parametric equations for an epitrochoid are given by:

$$x(\phi) = (a+b)\cos(\phi) - h\cos\left(\frac{a+b}{b}\phi\right)$$

$$y(\phi) = (a+b)\sin(\phi) - h\sin\left(\frac{a+b}{b}\phi\right)$$

Where:

- a: Radius of the fixed circle (base circle),
- b: Radius of the rolling circle,
- h: Distance of the tracing point from the center of the rolling circle,
- ϕ: Angle parameter (not the polar angle).

Polar Form

The polar form of an epitrochoid can be derived from the parametric equations. First, calculate $r^2 = x^2 + y^2$:

$$r^2 = (a+b)^2 + h^2 - 2h(a+b)\cos\left(\frac{a}{b}\phi\right)$$

Here, ϕ is the parameter, not the polar angle. The polar angle θ is given by:

$$\tan\theta = \frac{y}{x} = \frac{(a+b)\sin(\phi) - h\sin\left(\frac{a+b}{b}\phi\right)}{(a+b)\cos(\phi) - h\cos\left(\frac{a+b}{b}\phi\right)}$$

Special Cases of Epitrochoids

1. **Epicycloid**:
   - When $h = b$, the tracing point lies on the circumference of the rolling circle.
   - The resulting curve is an **epicycloid**, which is a special case of an epitrochoid.
   - Parametric equations:

$$x(\phi) = (a+b)\cos(\phi) - b\cos\left(\frac{a+b}{b}\phi\right)$$

$$y(\phi) = (a+b)\sin(\phi) - b\sin\left(\frac{a+b}{b}\phi\right)$$

2. **Limaçon**:
   - When $a = b$, the epitrochoid becomes a **limaçon.**
   - If $h = b$, the limaçon becomes a **cardioid** (heart-shaped curve).
3. **Circle**:
   - When $h = 0$, the tracing point is at the center of the rolling circle.
   - The resulting curve is a **circle** with radius $a + b$.
4. **Rose**:
   - **Rose curves** can also be generated from Epitrochoids.
   - when $h = a + b$ Rose like pattern is observed.
   - For more details on this check out the discussion section

Key Properties

1. **Number of Cusps**:
   - The number of cusps in an epitrochoid depends on the ratio a/b:
     - If $a/b = n$ (a rational number), the epitrochoid will have n cusps.
     - If a/b is irrational, the curve never closes and forms a dense pattern.
2. **Arc Length**: The total arc length of a closed epitrochoid where $a/b = p/q$, in the interval $[0, 2\pi q]$ is

$$L_{\text{total}} = (a+b)\int_0^{2\pi q}\sqrt{1+\left(\frac{h}{b}\right)^2 - 2\frac{h}{b}\cos\left(\frac{p}{q}\theta\right)}\,d\theta$$

If you want to know how the arc length is calculated check out the derivation section.

3. **Area Enclosed**:

   - The area A enclosed by one complete epitrochoid is:

     $$A = \pi q(k+1)\left[b^2(k+1)+h^2\right]$$

     Where $k = \frac{a}{b} = \frac{p}{q}$ is a rational in the simplest form. To know more check out the derivation section.

Visualization and Construction

Geometric Construction

1. Start with a fixed circle of radius a.
2. Roll a smaller circle of radius b along the *outside* of the fixed circle without slipping.
3. Track the motion of a point located at a distance h from the center of the rolling circle.

Symmetry

- Epitrochoids exhibit rotational symmetry. The number of symmetry axes corresponds to the number of cusps.

Related Curves

1. **Hypotrochoids**: Generated when the rolling circle rolls *inside* the fixed circle.
2. **Cycloids**: Generated when a circle rolls along a straight line.
3. **Epicycloids**: A special case of epitrochoids where the tracing point lies on the circumference of the rolling circle.

References

For more information, visit:

- MathWorld: Epitrochoid
- Wikipedia: Epitrochoid

Animating Epitrochoids

This code below is part of a Python project for generating and animating **epitrochoid curves** (a type of roulette curve traced by a point on a circle rolling around another circle). The key functions and special cases are discussed below

Key Functions

1. `_is_irrational`:
   - Determines if a number (e.g., `n` or `r`) is irrational or a complex rational.
   - Uses `Fraction` representation to check if the denominator exceeds a threshold (`denom_threshold=1000`).
   - Handles edge cases like `NaN`, infinity, or unexpected types.
2. `_process_parameters`:
   - Processes input parameters (`n_values` or `r_values`) to calculate:
     - `b`: Rolling circle radii.
     - `h`: Tracing distances.
     - `n_types`: Classifies `n` as **integer**, **rational**, or **irrational**.
   - Handles special cases:
     - Invalid or non-positive `r` values are skipped.
     - For **limaçon curves**, forces `n=1`.
     - Adjusts `d` values if mismatched with the number of circles.
3. `_calculate_animation_parameters`:
   - Computes:
     - Plot padding (`max_padding`) for visualization.
     - Total revolutions and frames required for animation based on `n_types`.
   - Ensures irrational patterns run for a maximum duration.
4. `_epitrochoid_points`:

- Calculates the coordinates of the rolling circle's center and the tracing point for given angles (`theta`).
- Handles edge cases like `b=0` (rolling circle is a point).

5. `_setup_plot` :

   - Sets up the Matplotlib plot for animation.
   - Creates visual elements like rolling circles, tracing points, and path lines.
   - Uses color maps for distinct curve visualization.
   - Adds a legend and optional revolution counter for multi-revolution animations.

Handling of `n` and `r` Types

- **`n` (Ratio of fixed to rolling circle radii):**
  - Classified as:
    - **Integer**: If `n` is close to a whole number (e.g., `n=2`).
    - **Rational**: If `n` can be represented as a fraction with a small denominator (e.g., `n=3/2`).
    - **Irrational**: If `n` cannot be represented as a simple fraction (e.g., `n=√2`).
  - Determines the number of revolutions required for the curve to close:
    - **Integer**: 1 revolution.
    - **Rational**: Denominator of the fraction.
    - **Irrational**: Runs for a maximum duration.
- **`r` (Rolling circle radius):**
  - Derived from `n` as `r = a / n` or directly input.
  - Checked for irrationality using `_is_irrational` .

Special Cases

1. **Invalid Inputs**:
   - Non-positive or non-finite `r` values are skipped with warnings.
   - Invalid `n` values (e.g., `n=0`) are ignored.
2. **Limaçon Curves**:
   - Forces `n=1` (rolling circle radius equals fixed circle radius).
3. **`b=0` (Rolling Circle is a Point)**:
   - Simplifies calculations for the tracing point.
4. **Animation Parameters**:
   - Ensures a minimum number of frames and padding for visualization.
5. **Dynamic `d` Values**:
   - Adjusts tracing distances (`h`) based on the curve type or input.

In [ ]:
```
class Epitrochoids:
    """
    Generates and animates multiple epitrochoids formed by circles rolling
    around the *outside* of a stationary circle.

    Handles integer, rational, and irrational ratios n = R/r, adjusting
    animation duration accordingly. Also handles special cases like epicycloids,
    limaçons, roses, and circles (degenerate case h=0).
    """
    def __init__(self, R=4, d=None, n_values=None, r_values=None,
                 type='random', save_anim=False, filename=None,
                 frames_per_rev=100, fps=25,
                 max_revolutions_irrational=5):
        """
        Initializes the Epitrochoids animation setup.

        Parameters:
        -----------
        R : float
            Radius of the stationary circle (default: 4). a = R.
        d : float or list[float], optional
            Distance(s) of the tracing point from the center of the rolling circle (h = d).
            If None, defaults are chosen based on type.
            If float, applied to all rolling circles.
            If list, must match the number of rolling circles.
        n_values : int, float, Fraction, or list, optional
```

```
        Determines rolling circle radii as r = R/n. Used if r_values is None.
        Accepts integers, floats, Fractions, or lists containing these.
        Affects animation duration:
         - Integer n: Closes in 1 revolution of the center.
         - Rational n=p/q: Closes in q revolutions of the center.
         - Irrational n: Animates for max_revolutions_irrational.
        (default: [3, 2, 1] for random type)
    r_values : int, float, or list, optional
        Explicit radii for the rolling circles (b = r). Overrides n_values if provided.
        If int/float, converted to a list.
        Corresponding n values (R/r) will be calculated and used for animation duration.
    type : str, optional
        Type of curve to generate ('random', 'epicycloid', 'limacon', 'circle', 'rose').
        Defaults to 'random'. Affects default parameters and calculations.
    save_anim : bool, optional
        Whether to save the animation to a file (default: False).
    filename : str, optional
        Name of the file to save the animation (required if save_anim is True).
        Default filenames are generated based on type if None.
    frames_per_rev : int, optional
        Number of frames per 2*pi revolution of the rolling circle's center (default: 100).
        Controls animation smoothness.
    fps : int, optional
        Frames per second for the animation (default: 25). Controls speed.
    max_revolutions_irrational : float or int, optional
        Number of 2*pi revolutions of the center to simulate for irrational n values.
        (default: 5)
    """
    self.R = float(R)
    self.a = self.R  # Use 'a' for radius of fixed circle in formulas
    if self.a <= 0:
        raise ValueError("Fixed circle radius 'R' must be positive.")
    self.type = type.lower()
    self.save_anim = save_anim
    self.filename = filename
    self.frames_per_revolution = int(frames_per_rev)
    self.FPS = fps
    self.max_revolutions_irrational = float(max_revolutions_irrational)

    # Lists to store calculated parameters for each curve
    self.n_values_input_type = [] # Store original input type for n (esp Fraction)
    self.n_values_numeric = [] # The numeric ratio a/b for each curve for calcs
    self.b_values = []  # Radius of rolling circle (r)
    self.h_values = []  # Distance of tracing point (d)
    self.n_types = []   # Type of n ('integer', 'rational', 'irrational')
    self.n_revs = []    # Revolutions needed for each curve to close/complete

    self._process_parameters(d, n_values, r_values)
    self._calculate_animation_parameters()

    # Plotting attributes to be initialized later
    self.fig = None
    self.ax = None
    self.circles_rolling = []
    self.path_lines = []
    self.radius_lines = []
    self.tracing_points = []
    self.center_points = []
    self.fixed_circle = None
    self.anim = None
    self.rev_text = None

@staticmethod
def _is_irrational(num, denom_threshold=1000):
    """
    Approximate check for irrational numbers or complex rationals based on Fraction representation.
    Returns True if number seems irrational or is a rational with a large denominator.
    """
    # Handle obvious non-numeric types if they somehow sneak in
    if not isinstance(num, (int, float, Fraction)):
        return True # Treat unexpected types as irrational
    if isinstance(num, int): # Integers are rational
        return False
    if isinstance(num, Fraction): # Already a fraction
        # Check if denominator is large enough to be treated as complex/irrational for animation
        return num.denominator > denom_threshold

    # Handle Floats
    if not np.isfinite(num): # infinity or NaN
        return True
```

```python
        # Check via Fraction conversion
        try:
            # Use a large limit_denominator to better represent the float
            f = Fraction(num).limit_denominator(100000) # Increased precision limit
            # Check if denominator is large after limiting (suggests irrational or complex rational)
            if f.denominator > denom_threshold:
                return True
        except (OverflowError, ValueError):
            return True # Treat conversion errors as irrational
        return False # Assume rational otherwise

    def _process_parameters(self, d, n_values_in, r_values_in):
        """
        Determines rolling circle radii (b), tracing distances (h), n values,
        and n types based on inputs and curve type.
        """
        default_n = [3, 2, 1] # Default for random if nothing else specified
        processed_n_values = [] # Store n as calculated or input
        processed_r_values = []
        r_was_irrational_flags = [] # Track if input r was likely irrational

        # --- Step 1: Determine the primary list of n or r values ---
        if r_values_in is not None:
            # Convert r_values to a list of floats
            if isinstance(r_values_in, (int, float)):
                input_r_list = [float(r_values_in)]
            else:
                input_r_list = [float(r) for r in r_values_in]

            # Calculate corresponding n_values and check r's nature
            temp_n_values = []
            valid_r_values = []
            for r in input_r_list:
                if r <= 0:
                    warnings.warn(f"Rolling radius r={r} must be positive. Skipping.")
                    continue
                if not np.isfinite(r):
                    warnings.warn(f"Rolling radius r={r} is not finite. Skipping.")
                    continue

                valid_r_values.append(r)
                n_numeric = self.a / r
                temp_n_values.append(n_numeric)
                # Check if the input r itself seems irrational/complex
                r_was_irrational_flags.append(self._is_irrational(r, denom_threshold=1000))

            processed_n_values = temp_n_values
            processed_r_values = valid_r_values
            if not processed_n_values:
                raise ValueError("No valid rolling circles could be determined from input r_values.")

        else: # Use n_values_in
            input_n_list = []
            if n_values_in is None:
                # Use defaults based on type if n_values not given
                if self.type == 'rose':
                    input_n_list = [2, 3/2, 5/3] # Example values for rose patterns
                elif self.type == 'limacon':
                    input_n_list = [1] # Limaçon requires n=1 (a=b)
                    if d is None: d = [self.R * 0.5, self.R * 1.0, self.R * 1.5] # Example distances for limaçon
                elif self.type == 'epicycloid':
                    input_n_list = default_n
                elif self.type == 'circle':
                    input_n_list = default_n
                    if d is not None: warnings.warn(f"Type is 'circle', ignoring provided 'd'. Using d=0.")
                    d = 0 # Force d=0 for circle type
                else: # Random
                    input_n_list = default_n
            elif isinstance(n_values_in, (int, float, Fraction)):
                input_n_list = [n_values_in]
            else: # It's a list
                input_n_list = list(n_values_in)

            # Validate n_values and calculate corresponding r (b) values
            temp_r_values = []
            valid_n_input_type = [] # Keep original type (like Fraction)
            valid_n_numeric = []

            for n_raw in input_n_list:
```

```
            n_numeric = None
            try:
                # Try converting to float first for numeric checks
                n_numeric = float(n_raw)
            except (TypeError, ValueError):
                warnings.warn(f"Invalid type for n value: {n_raw}. Skipping.")
                continue

            if n_numeric == 0:
                warnings.warn(f"n=0 detected, results in infinite radius 'b'. Skipping.")
                continue
            if not np.isfinite(n_numeric):
                warnings.warn(f"n={n_numeric} is not finite. Skipping.")
                continue

            if self.type == 'limacon' and abs(n_numeric - 1.0) > 1e-6:
                warnings.warn(f"Type is 'limacon', requires n=R/r=1. Adjusting input n={n_numeric} to 1.0.")
                n_numeric = 1.0
                n_raw = 1.0 # Update raw value as well

            r_calc = self.a / n_numeric
            temp_r_values.append(r_calc)
            valid_n_input_type.append(n_raw) # Store original input form
            valid_n_numeric.append(n_numeric)# Store numeric form for calculations

        if not valid_n_numeric:
            raise ValueError("No valid rolling circles could be determined from input n_values.")

        processed_r_values = temp_r_values
        processed_n_values = valid_n_numeric # Store numeric values
        self.n_values_input_type = valid_n_input_type # Store input types separately

    num_circles = len(processed_r_values)
    self.b_values = processed_r_values
    self.n_values_numeric = processed_n_values # Store numeric n for calcs/type checking


    # --- Step 2: Determine n_types and revolutions needed ---
    self.n_types = []
    self.n_revs = []
    for i, n_numeric in enumerate(self.n_values_numeric): # Iterate using numeric n
        determined_n_type = "irrational" # Default assumption
        determined_revolutions = self.max_revolutions_irrational
        is_forced_irrational = False
        n_input_val = self.n_values_input_type[i] if i < len(self.n_values_input_type) else n_numeric # Get 

        # Check if r was flagged as irrational (only if r_values were the input source)
        if r_values_in is not None and i < len(r_was_irrational_flags) and r_was_irrational_flags[i]:
             print(f"Input r={self.b_values[i]:.4f} flagged as irrational/complex. Forcing n={n_numeric:.4f}
             determined_n_type = "irrational"
             determined_revolutions = self.max_revolutions_irrational
             is_forced_irrational = True

        # Also check if n was input directly as a float likely representing irrational
        elif r_values_in is None and isinstance(n_input_val, float) and self._is_irrational(n_input_val):
             print(f"Input n={n_input_val:.4f} flagged as irrational/complex float. Forcing treatment as irra
             determined_n_type = "irrational"
             determined_revolutions = self.max_revolutions_irrational
             is_forced_irrational = True

        # Only proceed with int/rational checks if not forced to irrational
        if not is_forced_irrational:
            try:
                # Check if it's essentially an integer
                if np.isclose(n_numeric, round(n_numeric)):
                    n_int = round(n_numeric)
                    if n_int >= 1: # Valid integer n
                        determined_n_type = "integer"
                        determined_revolutions = 1
                    else:
                       # Handle cases like n=0.5 treated as integer 0 or 1 incorrectly
                       pass # Keep default irrational if rounded n < 1
                else:
                    # Check if it's truly rational (not too complex)
                    # Use the original input Fraction if available, otherwise check the float
                    is_rational_simple = False
                    if isinstance(n_input_val, Fraction):
                        # If input was Fraction, respect it unless denominator is huge
                        if n_input_val.denominator <= 1000: # Threshold for 'simple' rational
                            determined_n_type = "rational"
```

```
                        determined_revolutions = n_input_val.denominator
                        is_rational_simple = True
                # If not a simple input Fraction, check the float n_numeric
                if not is_rational_simple and not self._is_irrational(n_numeric, denom_threshold=1000):
                     # It's representable by a fraction with a reasonably small denominator
                    frac_n = Fraction(n_numeric).limit_denominator(1000) # Find that fraction
                    p = frac_n.numerator
                    q = frac_n.denominator
                    if q > 1: # Ensure it's not simplified back to integer
                        determined_n_type = "rational"
                        determined_revolutions = q
                # else: keep default of irrational / max_revolutions

            except (ValueError, OverflowError):
                # Error converting, treat as irrational
                pass

        self.n_types.append(determined_n_type)


        if self.type == 'circle':
            # Force revolutions to 1 for circle type
            self.n_revs.append(1)
            print(f"Type is 'circle', using n={n_numeric:.4f} (classified as {determined_n_type}) "
                  f"but forcing completion in 1 revolution.")
        else:
            # Not a circle, store the naturally determined revolutions
            self.n_revs.append(determined_revolutions)

            # Print info about n interpretation (only for non-circle types now)
            n_display = n_input_val if r_values_in is None else n_numeric
            if determined_n_type=='irrational':
                print(
                    f"n={n_numeric:.4f} "
                    f"(Input: ~{n_display:.4f}) "
                    f"interpreted as {determined_n_type}, "
                    f"setting max revolutions to {determined_revolutions:.2f}."
                )
            elif determined_n_type=='rational':
                # Display fraction from input if Fraction, else format calculated fraction
                frac_display = (n_input_val
                                if isinstance(n_input_val, Fraction)
                                else Fraction(n_numeric).limit_denominator(1000))
                print(
                    f"n={float(n_numeric):.4f} "
                    f"(Input: {float(n_display):.4f}, Approx Fraction: {frac_display}) "
                    f"interpreted as {determined_n_type}, "
                    f"setting max revolutions to {determined_revolutions:.2f}."
                )
            else: # integer
                print(
                    f"n={n_numeric:.1f} "
                    f"interpreted as {determined_n_type}, "
                    f"requires {determined_revolutions} revolution(s)."
                )

    # --- Step 3: Determine d_values (tracing distances 'h') ---
    final_d_values = []
    if self.type == 'epicycloid':
        final_d_values = list(self.b_values)
        if d is not None: warnings.warn(f"Type is 'epicycloid', ignoring provided 'd'. Using d=r.")
    elif self.type == 'rose':
        final_d_values = [self.a + b for b in self.b_values]
        if d is not None: warnings.warn(f"Type is 'rose', ignoring provided 'd'. Using d=R+r (a+b).")
    elif self.type == 'limacon':
        # Ensure b reflects n=1 if needed (already handled during n processing)
        # Use provided d for limacon, otherwise default
        if d is None:
            final_d_values = [self.a] * num_circles # Default to h=a (Cardioid)
            warnings.warn(f"Type is 'limacon' but 'd' not provided. Using default d=R (h=a) -> Cardioid.")
        elif isinstance(d, (int, float)):
            final_d_values = [float(d)] * num_circles
        else: # d is list
            if len(d) != num_circles:
                 # If n_values wasn't specified, maybe adjust num_circles based on d
                if n_values_in is None and r_values_in is None:
                    num_circles = len(d)
                    # Need to resize n_values, b_values etc. assuming n=1
                    self.n_values_numeric = [1.0] * num_circles
                    self.n_values_input_type = [1.0] * num_circles
```

```
                    self.b_values = [self.a] * num_circles
                    self.n_types = ["integer"] * num_circles
                    self.n_revs = [1] * num_circles
                    # warnings.warn(f"Type is 'limacon' and 'd' is a list. Neither n_values nor r_values prov
                    # f"Adjusted number of curves to match length of 'd' ({num_circles}).")
                    final_d_values = [float(val) for val in d]
                else:
                    raise ValueError(f"Length of 'd' ({len(d)}) must match the number of rolling circles "
                                     f"({num_circles}) for limacon type when 'd' is a list and n/r values wer
                                     )
            else:
                 final_d_values = [float(val) for val in d]

    elif self.type == 'circle':
        final_d_values = [0.0] * num_circles
    else: # Random type
        if d is None:
            final_d_values = [b * 0.5 for b in self.b_values]
            warnings.warn(f"Parameter 'd' not provided for random type. Using default d=r/2.")
        elif isinstance(d, (int, float)):
            final_d_values = [float(d)] * num_circles
        else: # d is a list
            d_float = [float(val) for val in d]
            if len(d_float) != num_circles:
               # Adjust d list to match num_circles (repeat last or truncate)
               if len(d_float) < num_circles:
                   warnings.warn(f"Length of 'd' ({len(d_float)}) < number of circles ({num_circles}). Repeat
                   final_d_values = d_float + [d_float[-1]] * (num_circles - len(d_float))
               else: # len > num_circles
                   warnings.warn(f"Length of 'd' ({len(d_float)}) > number of circles ({num_circles}). Trunca
                   final_d_values = d_float[:num_circles]
            else:
               final_d_values = d_float

    # Ensure h (d) is not negative, use absolute value.
    self.h_values = [abs(val) for val in final_d_values] # Use h in formulas

    # Set default filename if needed
    if self.save_anim and self.filename is None:
        # Use numeric n for filename consistency
        n_info = "_".join([f"n{n:.3f}".replace('.','p') for n in self.n_values_numeric])
        h_info = "_".join([f"d{h:.2f}".replace('.', 'p') for h in self.h_values])
        self.filename = f"{self.type}_R{self.R:.1f}_{n_info}_{h_info}.gif"


def _calculate_animation_parameters(self):
    """
    Calculate padding, theta range, and frame count based on n_types.
    The animation runs long enough for the longest closing pattern or
    the max duration for any irrational patterns.
    """
    # Calculate the maximum extent needed for the plot limits
    max_dist_needed = self.a # Start with the fixed circle radius

    if not self.b_values: # Handle case with no rolling circles
        self.max_padding = self.a * 1.1 if self.a > 0 else 1.0 # Default padding
    else:
        for b, h in zip(self.b_values, self.h_values):
            trace_max = self.a + b + h
            circle_edge_max = self.a + 2 * b
            current_max = max(trace_max, circle_edge_max)
            if current_max > max_dist_needed:
                max_dist_needed = current_max
        # Add a buffer for better visualization
        self.max_padding = max_dist_needed * 1.1

    # Ensure padding is at least a small positive value
    if self.max_padding <= 0:
        self.max_padding = 1.0 # Default minimum padding

    # Determine total angle needed based on the maximum revolutions required
    if not self.n_revs: # Handle case of no valid circles
        self.max_total_revolutions = 1
    else:
        self.max_total_revolutions = max(self.n_revs) # Use the calculated revs

    print(f"Animation requires max {self.max_total_revolutions:.2f} revolutions of the center.")

    self.theta_max_pattern = 2 * np.pi * self.max_total_revolutions
    # Ensure total_frames is integer
```

```python
        self.total_frames = int(np.ceil(self.max_total_revolutions * self.frames_per_revolution))
        if self.total_frames <= 0: self.total_frames = self.frames_per_revolution

        self.theta_vals = np.linspace(0, self.theta_max_pattern, self.total_frames)

    @staticmethod
    def _epitrochoid_points(theta, a, b, h):
        """Calculate epitrochoid points for a given angle theta (or array of thetas)."""
        # Handle potential division by zero if b is zero
        if np.isclose(b, 0): # Use isclose for float comparison
            # If b=0, the "rolling" circle is a point. Its center is on the fixed circle.
            # Path of center IS the fixed circle.
            center_x = a * np.cos(theta)
            center_y = a * np.sin(theta)
            # The tracing point 'h' is relative to this center point, rotating with theta.
            # The "trace angle" for h relative to the center's position vector is 0.
            # So, h is just added radially outward/inward.
            trace_x = (a + h) * np.cos(theta)
            trace_y = (a + h) * np.sin(theta)
            return center_x, center_y, trace_x, trace_y

        # Center of rolling circle (outside)
        center_x = (a + b) * np.cos(theta)
        center_y = (a + b) * np.sin(theta)

        # Angle for the tracing point calculation relative to the rolling circle's center rotation
        trace_angle = ((a + b) / b) * theta

        # Standard formula
        trace_x = (a + b) * np.cos(theta) - h * np.cos(trace_angle)
        trace_y = (a + b) * np.sin(theta) - h * np.sin(trace_angle)

        return center_x, center_y, trace_x, trace_y


    def _setup_plot(self):
        """Sets up the Matplotlib figure and axes."""
        plt.style.use('dark_background')
        self.fig, self.ax = plt.subplots(figsize=(10, 10))
        self.fig.subplots_adjust(left=0.05, right=0.95, top=0.9, bottom=0.05)
        # Use calculated limit
        limit = self.max_padding
        self.ax.set_xlim(-limit, limit)
        self.ax.set_ylim(-limit, limit)
        self.ax.set_xticks([])
        self.ax.set_yticks([])
        self.ax.set_aspect('equal', adjustable='box')
        self.ax.grid(False)
        title = f"Epitrochoids (R={self.R:.2f}, Type: {self.type.capitalize()})"
        self.ax.set_title(title, fontsize=14, pad=10)

        # Stationary circle
        self.fixed_circle = patches.Circle((0, 0), self.a, fill=False, color='gray', lw=1.5, ls='--')
        self.ax.add_patch(self.fixed_circle)

        # Use a colormap for distinct colors
        if len(self.b_values) == 1:
            colors = ['#FFD700'] # Single curve color (e.g., gold)
        elif len(self.b_values) <= 10: # Use perceptually uniform viridis for few colors
             colors = plt.get_cmap('viridis')(np.linspace(0.1, 0.9, len(self.b_values)))
        else: # Use rainbow for many colors, accepting potential perception issues
             colors = plt.get_cmap('rainbow')(np.linspace(0, 1, len(self.b_values)))


        # Create plot elements for each epitrochoid
        self.circles_rolling = []
        self.path_lines = []
        self.radius_lines = []
        self.tracing_points = []
        self.center_points = []

        for i, (b, h, n_num, n_typ) in enumerate(zip(self.b_values, self.h_values, self.n_values_numeric, self.n_
            color = colors[i]

            # Rolling circle patch
            circle = patches.Circle((self.a + b, 0), b, fill=False, color=color, lw=1.5, zorder=3)
            self.ax.add_patch(circle)
            self.circles_rolling.append(circle)

            # Path line label generation
```

```
            n_input = self.n_values_input_type[i] if i < len(self.n_values_input_type) else n_num # Get original
            if n_typ == "rational":
                 # Prefer original Fraction if input, else format the limited fraction
                frac_n = n_input if isinstance(n_input, Fraction) else Fraction(n_num).limit_denominator(1000)
                n_label = f"{frac_n.numerator}/{frac_n.denominator}"
            elif n_typ == "integer":
                 n_label = f"{int(round(n_num))}" # Display as clean integer
            else: # irrational
                 n_label = f"{n_num:.4f}" # Display float value

            label = f'n={n_label}, r={b:.3f}, d={h:.3f}'
            path_line, = self.ax.plot([], [], '-', color=color, lw=2, label=label, zorder=2)
            self.path_lines.append(path_line)

            # Radius line (center to tracing point)
            radius_line, = self.ax.plot([], [], '--', color=color, lw=1.5, alpha=0.8, zorder=4)
            self.radius_lines.append(radius_line)

            # Tracing point marker
            tracing_point, = self.ax.plot([], [], 'o', color=color, ms=7, zorder=5)
            self.tracing_points.append(tracing_point)

            # Rolling circle center marker
            center_point, = self.ax.plot([], [], 'o', color='white', ms=5, alpha=0.6, zorder=4)
            self.center_points.append(center_point)

        if len(self.b_values) > 0: # Add legend only if there are curves
            self.ax.legend(loc='upper right', fontsize=9, facecolor='#1C1C1C', framealpha=0.7)

        # Add text annotation for revolution counter if animation > 1 rev
        if self.max_total_revolutions > 1.01: # Add threshold to avoid showing for exactly 1 rev
            self.rev_text = self.ax.text(0.02, 0.98, '', transform=self.ax.transAxes,
                                         bbox=dict(facecolor='#2e4053', alpha=0.7, boxstyle='round,pad=0.3'),
                                         color='white', fontsize=12, ha='left', va='top', zorder=10)


    def _init_animation(self):
        """Initializes animation elements."""
        all_elements = []
        for i in range(len(self.b_values)):
            # Reset lines to empty
            self.path_lines[i].set_data([], [])
            self.radius_lines[i].set_data([], [])
            self.tracing_points[i].set_data([], [])
            self.center_points[i].set_data([], [])
            # Reset circle position and radius line at theta=0
            init_position = self._epitrochoid_points(0, self.a, self.b_values[i], self.h_values[i])
            init_center_x, init_center_y, init_trace_x, init_trace_y = init_position
            self.circles_rolling[i].center = (init_center_x, init_center_y)
            self.tracing_points[i].set_data([init_trace_x], [init_trace_y])
            self.center_points[i].set_data([init_center_x], [init_center_y])
            self.radius_lines[i].set_data([init_center_x, init_trace_x], [init_center_y, init_trace_y])

            all_elements.extend([
                self.path_lines[i], self.circles_rolling[i], self.radius_lines[i],
                self.tracing_points[i], self.center_points[i]
            ])

        # Update the text annotation for revolutions
        if self.rev_text is not None:
             # Display total as float if it's not an integer
            total_rev_display = (f"{int(self.max_total_revolutions)}"
                                 if np.isclose(self.max_total_revolutions, round(self.max_total_revolutions))
                                 else f"{self.max_total_revolutions:.2f}")
            self.rev_text.set_text(f'Revs: 0/{total_rev_display}')
            all_elements.append(self.rev_text)

        return all_elements

    def _animate_frame(self, frame):
        """Updates animation for a single frame."""
        # Current angle for this frame
        # Handle potential index out of bounds if total_frames=0 or 1
        if frame >= len(self.theta_vals):
             current_theta = self.theta_vals[-1] if len(self.theta_vals) > 0 else 0
        else:
             current_theta = self.theta_vals[frame]
        # History of angles up to this frame for drawing the path
        theta_history = self.theta_vals[:frame+1]
```

```python
        plot_elements_to_update = []

        for i, (b, h) in enumerate(zip(self.b_values, self.h_values)):
            # Calculate positions for the entire history for the path line
            (centers_x_hist, centers_y_hist,
             traces_x_hist, traces_y_hist) = self._epitrochoid_points(theta_history, self.a, b, h)

            # Get position for the current frame for markers and rolling circle
            # Use the last point in the history
            if len(traces_x_hist) > 0:
                current_center_x = centers_x_hist[-1]
                current_center_y = centers_y_hist[-1]
                current_trace_x = traces_x_hist[-1]
                current_trace_y = traces_y_hist[-1]
            else: # Frame 0 or empty history case
                (current_center_x, current_center_y,
                 current_trace_x, current_trace_y) = self._epitrochoid_points(0, self.a, b, h)

            # Update plot elements
            self.circles_rolling[i].center = (current_center_x, current_center_y)
            self.path_lines[i].set_data(traces_x_hist, traces_y_hist)
            self.tracing_points[i].set_data([current_trace_x], [current_trace_y])
            self.center_points[i].set_data([current_center_x], [current_center_y])
            self.radius_lines[i].set_data([current_center_x, current_trace_x],
                                          [current_center_y, current_trace_y])

            plot_elements_to_update.extend([
                self.path_lines[i], self.circles_rolling[i], self.radius_lines[i],
                self.tracing_points[i], self.center_points[i]
            ])

        # Update revolution counter text
        if self.rev_text is not None:
            current_rev = current_theta // (2 * np.pi)
            # Format display based on whether total revs is integer or not
            total_rev_display = (f"{int(self.max_total_revolutions)}"
                                 if np.isclose(self.max_total_revolutions, round(self.max_total_revolutions))
                                 else f"{self.max_total_revolutions:.2f}")
            # Display current revs with precision
            self.rev_text.set_text(f'Revs: {int(current_rev)}/{total_rev_display}')
            plot_elements_to_update.append(self.rev_text)

        return plot_elements_to_update


    def generate_animation(self):
        """Creates and displays or saves the epitrochoid animation."""
        if not self.b_values:
            print("No valid epitrochoids to animate.")
            return None

        self._setup_plot()
        if self.fig is None:
            raise ValueError("Figure object (self.fig) is not initialized.")


        # Create the animation object
        print(f"Generating animation with {self.total_frames} frames...")
        self.anim = FuncAnimation(self.fig, self._animate_frame, frames=self.total_frames,
                                  init_func=self._init_animation, interval=max(1, int(1000/self.FPS)),
                                  blit=False,
                                  repeat=False)

        # Save or show the animation
        if self.save_anim:
            if not self.filename:
                print("Error: filename must be provided when save_anim is True.")
                plt.close(self.fig)
                return None

            # Ensure filename ends with .gif or .mp4 (default to gif)
            if not self.filename.lower().endswith(('.gif', '.mp4')):
                self.filename += '.gif'

            save_dir = Path("ANIMATIONS/EPITROCHOIDS")
            save_dir.mkdir(parents=True, exist_ok=True)
            filepath = save_dir / self.filename

            print(f"Saving animation to {filepath.resolve()}...")
            writer_choice = 'pillow' if self.filename.lower().endswith('.gif') else 'ffmpeg'
```

```
            # Use tqdm for progress bar
            try:
                with tqdm(total=self.total_frames, desc="Saving Animation", unit="frame", ncols=100) as pbar:
                    def progress_update(current_frame, total_frames):
                        pbar.n = current_frame + 1 # Update tqdm progress (callback is 0-based)
                        pbar.refresh() # Force redraw

                    self.anim.save(str(filepath), writer=writer_choice, fps=self.FPS,
                                   dpi=72, progress_callback=progress_update)
                # Ensure the progress bar finishes at 100%
                if pbar.n < self.total_frames: pbar.update(self.total_frames - pbar.n)

                print("\nAnimation saved successfully!")
            except Exception as e:
                print(f"\nError saving animation: {e}")
                print("Ensure 'ffmpeg' (for MP4) or 'Pillow' (for GIF) is installed.")
                print("If using blit=True, try setting blit=False if errors occur during save.")


            plt.close(self.fig) # Close the plot window after saving
        else:
            plt.show() # Display the animation interactively

        return self.anim # Return the animation object
```

Random Case

In [ ]:
```
# --- Random Case Example ---
print("\nGenerating Random Epitrochoids...")
random_epi_anim = Epitrochoids(R=4, d=1, n_values=[3, 4.5, 1.5],
                               save_anim=True,
                               frames_per_rev=150,
                               fps=30)
random_epi_anim.generate_animation()

# --- Example specifying r_values directly ---
# We can insert integer, rational or irrational values for r_values
print("\nGenerating Epitrochoids with specified r_values...")
custom_r_epi_anim = Epitrochoids(R=4, r_values=[1.0, 5/3, math.pi], d=[0.5, 1.5, 2.5], type='random',
                                 save_anim=True,
                                 filename="custom_r_epitrochoids.gif",
                                 max_revolutions_irrational=5, fps=50) # Limit irrational to 5 revolutions
# custom_r_epi_anim.generate_animation()
```

Special Cases

Epicycloids

In [ ]:
```
# --- Epicycloid Example (h = b) ---
print("\nGenerating Epicycloids (h=b)...")
# n=1 -> Cardioid, n=2 -> Nephroid
epicycloid_anim = Epitrochoids(R=3, n_values=[3, 4, 5], type='epicycloid',
                               save_anim=True,
                               frames_per_rev=200)
# epicycloid_anim.generate_animation()

# Epicycloids with rational n values
print("\nGenerating Epicycloids with rational n values...")
epicycloid_rational_anim = Epitrochoids(R=3, n_values=[Fraction(2, 3), 3/2, 5/4],
                                        type='epicycloid',
                                        save_anim=True,
                                        fps=40)
# epicycloid_rational_anim.generate_animation()

# Epicycloids with irrational n values
# Displaying for a single value first
print("\nGenerating Epicycloid with with irrational n value...")
epi_irrational_anim = Epitrochoids(R=3, n_values=math.e, type='epicycloid',
                                   save_anim=True,
                                   filename='epicycloid_e.gif',
                                   frames_per_rev= 120,
                                   max_revolutions_irrational=5, fps=50)
epi_irrational_anim.generate_animation()

# Now with multiple irrational values
print("\nGenerating Epicycloids with irrational n values...")
```

```
epicycloid_irrational_anim = Epitrochoids(R=3, n_values=[math.sqrt(2), math.pi, math.e], type='epicycloid',
                                          save_anim=True,
                                          max_revolutions_irrational=10,
                                          frames_per_rev=100, fps=50)
# epicycloid_irrational_anim.generate_animation()

# Epicycloids with mixed rational and irrational n values
print("\nGenerating Epicycloids with mixed rational and irrational n values...")
mixed_epi_anim = Epitrochoids(R=3, n_values=[3, 5/2, math.pi], type='epicycloid',
                              save_anim=True,
                              filename='epicycloid_3_2p25_pi.mp4',
                              frames_per_rev=200,
                              max_revolutions_irrational=10, fps=100)
# mixed_epi_anim.generate_animation()
```

Limacons

In [ ]:
```
# --- Limaçon Example (a = b, n=1) ---
print("\nGenerating Limaçons (a=b)...")
# Vary d (h) relative to R (a). h=a=R gives Cardioid.
limacon_anim = Epitrochoids(R=2, d=[0.5, 1, 2, 3, 4], type='limacon',
                            save_anim=True,
                            frames_per_rev=200)
limacon_anim.generate_animation()
```

Rose

In [ ]:
```
# --- Rose Example (h = a + b) ---
print("\nGenerating Rose from Epitrochoids (h=a+b)...")
rose_anim = Epitrochoids(R=3, n_values=5, type='rose',
                         save_anim=True,
                         frames_per_rev=500,
                         fps=50)
# rose_anim.generate_animation()

# Rose with rational n values
print("\nGenerating Rose with rational n values...")
rose_rational_anim = Epitrochoids(R=3, n_values=[3/2, 5/3], type='rose',
                                  save_anim=True,
                                  frames_per_rev=170,
                                  fps=50)
rose_rational_anim.generate_animation()
```

Circle

In [ ]:
```
# --- Circle Example (h = 0)
print("\nGenerating Circle from Epitrochoid(h = 0)") # h=0, the tracing point lies at the center of the rolling c
circle_anim = Epitrochoids(R=4, n_values=[2, 4, math.e], type='circle',
                           save_anim=True
                           )
circle_anim.generate_animation()
```

Explanation: Why 'Circle' Type `Epitrochoids` Complete in 1 Revolution

In the `Epitrochoids` class, curves are typically traced by a point at distance `d` from the center of a rolling circle (radius `r`) moving around a fixed circle (radius `R`). The ratio `n = R/r` usually dictates how many revolutions are needed for this tracing point to complete the curve (e.g., `q` revolutions for rational `n = p/q`).

However, the 'circle' type is a special case where the tracing distance `d` is zero (d=0). This means:

1. **Tracing Point** = Center: The point traced is the rolling circle's center.
2. **Center's Path**: This center moves in a circle of radius `R + r` around the origin (the fixed circle's center).
3. **Completion in 1 Rev**: This circular path is fully traced in exactly one revolution (2π radians).
4. `n` is Irrelevant for Path: While `n` governs the rolling circle's rotation relative to its orbit, this rotation doesn't affect the path of the center itself when `d=0`. The center completes its circular orbit in one revolution regardless of `n`.

Conclusion: Since the 'circle' type (`d=0`) traces the rolling circle's center, the path is always a circle of radius `R + r`, completed in one revolution. Animating further is redundant as it just retraces this circle. Therefore, type='circle' animations are limited to a single revolution for efficiency and clarity, irrespective of the specific `n` or `r` values.

Degenerate cases

In [ ]:
```
# --- Example with R=a=0 (Should degenerate) ---
# print("\nGenerating Epitrochoid with R=a=0 (Degenerate Case)...")
# With a=0, formulas become x = b*cos(t)+h*cos(t), y = b*sin(t)-h*sin(t)
# Let's try type='random'
# zero_r_anim = Epitrochoids(R=0, r_values=[1], d=[0.5], type='random', save_anim=False, filename="zero_R_epitroc
# zero_r_anim.generate_animation()
# Let's try type='limacon' (a=b forced to 0=0)
# zero_r_limacon = Epitrochoids(R=0, d=[1], type='limacon', save_anim=True, filename="zero_R_limacon.gif")
# zero_r_limacon.generate_animation()
```

Deriving the Rose Curve from an **Epitrochoid**

We start with the general form of an **epitrochoid**:

$$\begin{cases} x(t) = (a+b)\cos t - h\cos\left(\frac{a+b}{b}t\right) \\ y(t) = (a+b)\sin t - h\sin\left(\frac{a+b}{b}t\right) \end{cases}$$

Step 1: Set $h = a + b$

We choose the tracing point to lie on the circumference of the rolling circle.

Substitute:

$$x(t) = (a+b)\left[\cos t - \cos\left(\frac{a+b}{b}t\right)\right]$$

$$y(t) = (a+b)\left[\sin t - \sin\left(\frac{a+b}{b}t\right)\right]$$

Let:

$$n = \frac{a+b}{b} = \frac{a}{b} + 1$$

Then:

$$x(t) = (a+b)[\cos t - \cos(nt)]$$
$$y(t) = (a+b)[\sin t - \sin(nt)]$$

Step 2: Apply Correct Sum-to-Product Identities

Using:

$$\cos A - \cos B = -2\sin\left(\frac{A+B}{2}\right)\sin\left(\frac{A-B}{2}\right)$$

$$\sin A - \sin B = 2\cos\left(\frac{A+B}{2}\right)\sin\left(\frac{A-B}{2}\right)$$

Apply to t and nt:

$$x(t) = 2(a+b)\,\sin\left(\frac{n+1}{2}t\right)\,\sin\left(\frac{n-1}{2}t\right)$$
$$y(t) = -2(a+b)\cos\left(\frac{n+1}{2}t\right)\,\sin\left(\frac{n-1}{2}t\right)$$

Step 3: Factor the Common Term

$$\boxed{\begin{aligned} x(t) &= 2(a+b)\,\sin\left(\frac{n-1}{2}t\right)\,\sin\left(\frac{n+1}{2}t\right) \\ y(t) &= -2(a+b)\,\sin\left(\frac{n-1}{2}t\right)\,\cos\left(\frac{n+1}{2}t\right) \end{aligned}}$$

This can be written as:

$$(x(t), y(t)) = 2(a+b)\,\sin\left(\frac{n-1}{2}t\right)\cdot\left(\sin\left(\frac{n+1}{2}t\right), -\cos\left(\frac{n+1}{2}t\right)\right)$$

Step 4: Introduce a New Angle Parameter

Let:

$$\varphi = \frac{n+1}{2}t \quad \Rightarrow \quad t = \frac{2\varphi}{n+1} \Rightarrow \frac{n-1}{2}t = \frac{n-1}{n+1}\varphi$$

Then:

$$(x(\varphi), y(\varphi)) = 2(a+b)\,\sin(m\varphi) \cdot (\sin\varphi, -\cos\varphi)$$

where:

$$\boxed{m = \frac{n-1}{n+1} = \frac{a}{2b+a}}$$

So, in polar form:

$$\boxed{r(\varphi) = 2(a+b)\,\sin(m\varphi)} \quad \text{with} \quad \boxed{m = \frac{a}{2b+a}}$$

Final Result

This is a rose curve expressed in polar coordinates:

$$\boxed{r(\varphi) = 2(a+b)\,\sin(m\varphi), \quad \text{with} \quad m = \frac{a}{2b+a}}$$

Derivations

The parametric equations for an epitrochoid are:

$$x(\theta) = (R+r)\cos\theta - d\cos\left(\frac{R+r}{r}\theta\right)$$

$$y(\theta) = (R+r)\sin\theta - d\sin\left(\frac{R+r}{r}\theta\right)$$

where θ is the angle of the line connecting the centers of the two circles with the positive x-axis.

Let $k = R/r$. The equations can be written as:

$$x(\theta) = r(k+1)\cos\theta - d\cos((k+1)\theta)$$

$$y(\theta) = r(k+1)\sin\theta - d\sin((k+1)\theta)$$

The curve closes if $k = R/r$ is a rational number. Let $k = p/q$ where p and q are coprime positive integers. The curve closes after θ ranges from 0 to $2\pi q$.

1. Arc Length of an Epitrochoid

1. **Find the derivatives:**

$$\frac{dx}{d\theta} = -(R+r)\sin\theta - d\left(-\frac{R+r}{r}\sin\left(\frac{R+r}{r}\theta\right)\right)$$

$$= -(R+r)\left[\sin\theta - \frac{d}{r}\sin\left(\frac{R+r}{r}\theta\right)\right]$$

$$\frac{dy}{d\theta} = (R+r)\cos\theta - d\left(\frac{R+r}{r}\cos\left(\frac{R+r}{r}\theta\right)\right)$$

$$= (R+r)\left[\cos\theta - \frac{d}{r}\cos\left(\frac{R+r}{r}\theta\right)\right]$$

2. **Compute the sum of squares:**

$$\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2$$

$$= (R+r)^2\left[\sin\theta - \frac{d}{r}\sin\left(\frac{R+r}{r}\theta\right)\right]^2 + (R+r)^2\left[\cos\theta - \frac{d}{r}\cos\left(\frac{R+r}{r}\theta\right)\right]^2$$

$$= (R+r)^2\left[\sin^2\theta - 2\frac{d}{r}\sin\theta\sin\left(\frac{R+r}{r}\theta\right) + \left(\frac{d}{r}\right)^2\sin^2\left(\frac{R+r}{r}\theta\right) + \cos^2\theta - 2\frac{d}{r}\cos\theta\cos\left(\frac{R+r}{r}\theta\right) + \left(\frac{d}{r}\right)^2\cos^2\left(\frac{R+r}{r}\theta\right)\right]$$

Group terms using $\sin^2\alpha + \cos^2\alpha = 1$ and $\cos(\alpha - \beta) = \cos\alpha\cos\beta + \sin\alpha\sin\beta$:

$$= (R+r)^2 \left[(\sin^2\theta + \cos^2\theta) + \left(\frac{d}{r}\right)^2 \left(\sin^2\left(\frac{R+r}{r}\theta\right) + \cos^2\left(\frac{R+r}{r}\theta\right)\right) - 2\frac{d}{r}\left(\cos\theta\cos\left(\frac{R+r}{r}\theta\right) + \sin\theta\sin\left(\frac{R+r}{r}\theta\right)\right)\right]$$

$$= (R+r)^2 \left[1 + \left(\frac{d}{r}\right)^2 - 2\frac{d}{r}\cos\left(\theta - \frac{R+r}{r}\theta\right)\right]$$

$$= (R+r)^2 \left[1 + \left(\frac{d}{r}\right)^2 - 2\frac{d}{r}\cos\left(-\frac{R}{r}\theta\right)\right]$$

$$= (R+r)^2 \left[1 + \left(\frac{d}{r}\right)^2 - 2\frac{d}{r}\cos\left(\frac{R}{r}\theta\right)\right] \quad (\text{since } \cos(-x) = \cos(x))$$

3. **Set up the arc length integral:** The arc length L is given by:

$$L = \int_\alpha^\beta \sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2}\, d\theta$$

Assuming $R, r > 0$, we have $|R+r| = R+r$.

$$L = \int_\alpha^\beta (R+r)\sqrt{1 + \left(\frac{d}{r}\right)^2 - 2\frac{d}{r}\cos\left(\frac{R}{r}\theta\right)}\, d\theta$$

For the total length of a closed epitrochoid where $R/r = p/q$, the interval is $[0, 2\pi q]$.

$$\boxed{L_{\text{total}} = (R+r)\int_0^{2\pi q} \sqrt{1 + \left(\frac{d}{r}\right)^2 - 2\frac{d}{r}\cos\left(\frac{p}{q}\theta\right)}\, d\theta}$$

4. **Evaluation:** Similar to the hypotrochoid, this integral is generally **not expressible in terms of elementary functions** and is a form of an elliptic integral. A simple closed-form formula exists only for the special case $d = r$.

Special Case: Epicycloid (d=r) If we set $d = r$, the expression under the square root becomes:

$$1 + \left(\frac{r}{r}\right)^2 - 2\frac{r}{r}\cos\left(\frac{R}{r}\theta\right) = 2\left(1 - \cos\left(\frac{R}{r}\theta\right)\right) = 4\sin^2\left(\frac{R}{2r}\theta\right)$$

The square root is $2\left|\sin\left(\frac{R}{2r}\theta\right)\right|$. The integrand becomes $(R+r)\cdot 2\left|\sin\left(\frac{R}{2r}\theta\right)\right|$. Integrating this from 0 to $2\pi q$:

$$L = \int_0^{2\pi q} 2(R+r)\left|\sin\left(\frac{R}{2r}\theta\right)\right| d\theta$$

Let $u = \frac{R}{2r}\theta = \frac{p}{2q}\theta$. $d\theta = \frac{2q}{p}du$. Limits 0 to $p\pi$.

$$L = \int_0^{p\pi} 2(R+r)|\sin(u)|\frac{2q}{p}du = \frac{4q(R+r)}{p}\int_0^{p\pi} |\sin(u)|du$$

Using $\int_0^{p\pi} |\sin(u)|du = 2p$:

$$L = \frac{4q(R+r)}{p}(2p) = \boxed{8q(R+r)}$$

This is the well-known formula for the total arc length of an epicycloid.

Conclusion for Arc Length: The arc length of an epitrochoid over an interval $[\alpha, \beta]$ is given by the integral

$$L = (R+r)\int_\alpha^\beta \sqrt{1 + (d/r)^2 - 2(d/r)\cos(R\theta/r)}\, d\theta$$

This integral generally requires numerical methods or elliptic functions, except for the epicycloid case ($d = r$), where the total length is $8q(R+r)$.

2. Area Enclosed by an Epitrochoid

We use Green's Theorem:

$$\boxed{A = \frac{1}{2}\oint (x\,dy - y\,dx) = \frac{1}{2}\int_0^{2\pi q}\left(x(\theta)\frac{dy}{d\theta} - y(\theta)\frac{dx}{d\theta}\right) d\theta}$$

.

1. **Calculate the integrand $x\frac{dy}{d\theta} - y\frac{dx}{d\theta}$:**

Let $A = R + r$ and $B = (R + r)/r$. Therefore,

$$\begin{aligned} x &= A\cos\theta - d\cos(B\theta) \\ y &= A\sin\theta - d\sin(B\theta) \\ \frac{dx}{d\theta} &= -A\sin\theta + dB\sin(B\theta) \\ \frac{dy}{d\theta} &= A\cos\theta - dB\cos(B\theta) \end{aligned}$$

$$\begin{aligned} x\frac{dy}{d\theta} &= (A\cos\theta - d\cos(B\theta))(A\cos\theta - dB\cos(B\theta)) \\ &= A^2\cos^2\theta - AdB\cos\theta\cos(B\theta) - Ad\cos(B\theta)\cos\theta + d^2B\cos^2(B\theta) \\ &= A^2\cos^2\theta - Ad(1+B)\cos\theta\cos(B\theta) + d^2B\cos^2(B\theta) \end{aligned}$$

$$\begin{aligned} y\frac{dx}{d\theta} &= (A\sin\theta - d\sin(B\theta))(-A\sin\theta + dB\sin(B\theta)) \\ &= -A^2\sin^2\theta + AdB\sin\theta\sin(B\theta) + Ad\sin(B\theta)\sin\theta - d^2B\sin^2(B\theta) \\ &= -A^2\sin^2\theta + Ad(1+B)\sin\theta\sin(B\theta) - d^2B\sin^2(B\theta) \end{aligned}$$

$$\begin{aligned} x\frac{dy}{d\theta} - y\frac{dx}{d\theta} &= (A^2\cos^2\theta - Ad(1+B)\cos\theta\cos(B\theta) + d^2B\cos^2(B\theta)) - (-A^2\sin^2\theta + Ad(1+B)\sin\theta\sin(B\theta) - d^2B\sin^2(B\theta)) \\ &= A^2(\cos^2\theta + \sin^2\theta) + d^2B(\cos^2(B\theta) + \sin^2(B\theta)) - Ad(1+B)(\cos\theta\cos(B\theta) + \sin\theta\sin(B\theta)) \\ &= A^2 + d^2B - Ad(1+B)\cos(\theta - B\theta) \end{aligned}$$

Substitute back $A = R + r$ and $B = (R + r)/r$:

$$\begin{aligned} \theta - B\theta &= \theta - \frac{R+r}{r}\theta = \frac{-R}{r}\theta, \quad \cos(\theta - B\theta) = \cos(R\theta/r) \\ 1 + B &= 1 + \frac{R+r}{r} = \frac{r+R+r}{r} = \frac{R+2r}{r} \end{aligned}$$

we get,

$$\begin{aligned} x\frac{dy}{d\theta} - y\frac{dx}{d\theta} &= (R+r)^2 + d^2\frac{R+r}{r} - (R+r)d\frac{R+2r}{r}\cos\left(\frac{R}{r}\theta\right) \\ &= (R+r)\left[(R+r) + \frac{d^2}{r} - \frac{d(R+2r)}{r}\cos\left(\frac{R}{r}\theta\right)\right] \end{aligned}$$

Let $k = R/r$. $R + r = r(k+1)$, $R = kr$.

$$\begin{aligned} &= r(k+1)\left[r(k+1) + \frac{d^2}{r} - \frac{d(kr+2r)}{r}\cos(k\theta)\right] \\ &= r^2(k+1)^2 + d^2(k+1) - rd(k+1)(k+2)\cos(k\theta) \end{aligned}$$

2. **Integrate over $[0, 2\pi q]$:**

$$A = \frac{1}{2}\int_0^{2\pi q}\left[r^2(k+1)^2 + d^2(k+1) - rd(k+1)(k+2)\cos(k\theta)\right]d\theta$$

Assuming $k = p/q \neq 0$, the integral of the $\cos(k\theta)$ term over the full period $[0, 2\pi q]$ is zero.

$$\begin{aligned} A &= \frac{1}{2}\int_0^{2\pi q}\left[r^2(k+1)^2 + d^2(k+1)\right]d\theta \\ &= \frac{1}{2}\left[r^2(k+1)^2 + d^2(k+1)\right]\left[\theta\right]_0^{2\pi q} \\ &= \frac{1}{2}[r^2(k+1)^2 + d^2(k+1)](2\pi q) \\ &= \pi q(k+1)[r^2(k+1) + d^2] \end{aligned}$$

> *Note: If $k = 0$ ($R = 0$), the integral of the cosine term is not necessarily zero. This case is handled separately under Special Cases.*

3. **Final Formula and Interpretation:** For $k = R/r = p/q \neq 0$, the area (swept by the vector from the origin) is:

$$\boxed{A = \pi q(k+1)[r^2(k+1) + d^2]}$$

where $k = R/r = p/q$ in simplest form.

Rewriting in terms of R and r: $k + 1 = (R + r)/r$.

$$
\begin{aligned}
A &= \pi q\left(\frac{R+r}{r}\right)\left[r^2\left(\frac{R+r}{r}\right)+d^2\right] \\
&= \pi q\left(\frac{R+r}{r}\right)[r(R+r)+d^2] \\
&= \frac{\pi q(R+r)}{r}[rR+r^2+d^2]
\end{aligned}
$$

Area Interpretation: This formula gives the "vectorial area" swept by the position vector from the origin. If the curve has loops (which occurs for epitrochoids when $d > r$), this area counts looped regions multiple times or with negative orientation. It may not correspond to the simple geometric area enclosed by the outermost loop.

Special Case: Epicycloid (d=r) Substitute $d = r$ into the formula (assuming $k \neq 0$):

$$A = \pi q(k+1)[r^2(k+1)+r^2] = \pi q(k+1)r^2[k+1+1] = \pi q r^2(k+1)(k+2)$$

Rewriting in terms of R and r:

$$A = \pi q r^2(\frac{R}{r}+1)(\frac{R}{r}+2) = \pi q r^2\frac{R+r}{r}\frac{R+2r}{r} = \pi q(R+r)(R+2r)$$

This matches the known area formula for an epicycloid (which doesn't have loops penetrating the origin, so the vectorial area matches the geometric area). If k is an integer, $q = 1$, giving $A = \pi(R+r)(R+2r)$.

3. Discussion of Special Cases

1. **Epicycloid ($d = r$):**
   - As shown, the epitrochoid formulas reduce to the epicycloid formulas: $L = 8q(R+r)$ and $A = \pi q(R+r)(R+2r)$.
   - Epicycloids have p cusps if $k = R/r = p/q$.
2. **Circle:**
   - **Case 1: $d = 0$ (Tracing point is the center of the rolling circle):**

     $$x(\theta) = (R+r)\cos\theta$$
     $$y(\theta) = (R+r)\sin\theta$$

     This is a circle of radius $R+r$ centered at the origin. The curve closes in $\theta = 2\pi$, so $q = 1$.

     - Arc Length: Formula integrand is $(R+r)\sqrt{1+0-0} = R+r$.

       $$L = \int_0^{2\pi}(R+r)d\theta = 2\pi(R+r).$$

       Matches circumference.
     - Area: Using $A = \pi q(k+1)[r^2(k+1)+d^2]$ with $q = 1, d = 0$:

       $$A = \pi(1)(k+1)[r^2(k+1)+0] = \pi r^2(k+1)^2 = \pi r^2(\frac{R+r}{r})^2 = \pi(R+r)^2$$

       Matches circle area $\pi \times \text{radius}^2$.
   - **Case 2: $R = 0$ (Fixed "circle" is a point):** $k = R/r = 0$. Here $p = 0, q = 1$.

     $$x(\theta) = (0+r)\cos\theta - d\cos\left(\frac{0+r}{r}\theta\right) = r\cos\theta - d\cos\theta = (r-d)\cos\theta$$
     $$y(\theta) = (0+r)\sin\theta - d\sin\left(\frac{0+r}{r}\theta\right) = r\sin\theta - d\sin\theta = (r-d)\sin\theta$$

     This is a circle of radius $|r-d|$ centered at the origin.

     - Arc Length: Formula integrand is $(0+r)\sqrt{1+(d/r)^2-2(d/r)\cos(0)} = r\sqrt{1+d^2/r^2-2d/r} = r\sqrt{(1-d/r)^2} = r|1-d/r| = |r-d|$.

       $$L = \int_0^{2\pi}|r-d|d\theta = 2\pi|r-d|.$$

       Matches circumference.
     - Area: We must re-integrate as the general formula assumed $k \neq 0$. The integrand was $xdy - ydx = (r-d)^2 d\theta$.

       $$A = \frac{1}{2}\int_0^{2\pi}(r-d)^2 d\theta = \frac{1}{2}(r-d)^2(2\pi) = \pi(r-d)^2$$

Matches circle area $\pi \times \text{radius}^2$.

3. **Limaçon ($R = r$ or $k = 1$):**

   - If $R = r$, then $k = 1$ ($p = 1, q = 1$). The equations become:

$$x(\theta) = (r + r)\cos\theta - d\cos\left(\frac{r + r}{r}\theta\right) = 2r\cos\theta - d\cos(2\theta)$$
$$y(\theta) = (r + r)\sin\theta - d\sin\left(\frac{r + r}{r}\theta\right) = 2r\sin\theta - d\sin(2\theta)$$

   - This curve is known as **Pascal's Limaçon**. It can be shown that its equation in polar coordinates (ρ, ϕ) centered appropriately is $\rho = 2r - d\cos(\phi')$ (or similar form $b + a\cos\phi'$ with $b = 2r, a = -d$).
   - If $d = r$, it's a special limaçon called the Cardioid (an epicycloid with $k = 1$).
   - If $d = 2r$, it's a Trisectrix.
   - Arc Length: The integral becomes $L = (r + r)\int_0^{2\pi}\sqrt{1 + (d/r)^2 - 2(d/r)\cos(\theta)}d\theta$. This is still generally an elliptic integral unless $d = r$. For the Cardioid ($d = r$), $L = 8q(R + r) = 8(1)(r + r) = 16r$.
   - Area: Using the general formula $A = \pi q(k + 1)[r^2(k + 1) + d^2]$ with $k = 1, q = 1$:

$$A = \pi(1)(1 + 1)[r^2(1 + 1) + d^2] = 2\pi[2r^2 + d^2] = \pi(4r^2 + 2d^2)$$

   - As noted before, this is the vectorial area. The geometric area of the limaçon $\rho = b + a\cos\phi$ is $\pi(b^2 + a^2/2)$. With $b = 2r, a = -d$, this is $\pi((2r)^2 + (-d)^2/2) = \pi(4r^2 + d^2/2)$. The vectorial area formula differs from the geometric area when $d \neq r$ due to loops. For the Cardioid ($d = r$), the vectorial area is $A = 2\pi(2r^2 + r^2) = 6\pi r^2$. The geometric area is $\pi(4r^2 + r^2/2) = 4.5\pi r^2$? No, the standard Cardioid area is $6\pi r^2$. So for $d = r$, the vectorial and geometric areas match. The standard limaçon area formula $\pi(b^2 + a^2/2)$ might assume a specific form or orientation, or apply only to the outer loop. The Green's theorem result $A = 2\pi(2r^2 + d^2)$ is consistent for the vectorial area of the $k = 1$ epitrochoid / limaçon.

In summary:

- The **arc length** of a general epitrochoid involves an elliptic integral, simplifying to $8q(R + r)$ for epicycloids ($d = r$).
- The **vectorial area** enclosed by an epitrochoid ($k \neq 0$) is $A = \pi q(k + 1)[r^2(k + 1) + d^2]$. This matches the geometric area for epicycloids ($d = r$).
- An **epicycloid** is a special case ($d = r$).
- A **circle** is obtained if $d = 0$ (radius $R + r$) or if $R = 0$ (radius $|r - d|$).
- A **limaçon** is obtained if $R = r$ ($k = 1$). The cardioid is the case $d = r$.

Cycloidal Motion of a circular disc

In [ ]:
```
def animate_rolling_disc(disc_radius=1.0, disc_thickness=0.3, num_revolutions=2, save_anim=False, filename=None)
    """
    Animate a disc rolling on top of a cuboid along the y-axis with improved visualization.

    Parameters:
    -----------
    disc_radius : float
        Radius of the rolling disc (default: 1.0)
    disc_thickness : float
        Thickness of the disc (default: 0.3)
    num_revolutions : int
        Number of complete disc revolutions (default: 2)
    save_anim : bool
        Whether to save the animation (default: False)
    filename : str, optional
        Name of the file to save the animation (default: None)
    """
    # Calculate cuboid dimensions based on cycloid path length
    cuboid_length = 6.0  # Fixed width for better visualization
    cuboid_width = 2 * np.pi * disc_radius * num_revolutions * 1.2  # Total path length plus margin
    cuboid_height = 2.5 # Fixed height

    # Animation parameters
    frames_per_revolution = 25
    total_frames = int(num_revolutions * frames_per_revolution)
    interval_ms = 10

    # Setup figure
```

```python
plt.style.use('dark_background')
fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(111, projection='3d')
plt.subplots_adjust(left=0.05, right=0.95, top=0.95, bottom=0.05)

# Setup parameters
theta_max = num_revolutions * 2 * np.pi
theta_vals = np.linspace(0, theta_max, total_frames)

# Set plot limits
x_limits = (0, cuboid_length)
y_limits = (0, cuboid_width)
z_limits = (0, cuboid_height + 2.5*disc_radius)

# Create the cuboid
vertices = np.array([
    [0, 0, 0],                              # vertex 0
    [cuboid_length, 0, 0],                  # vertex 1
    [cuboid_length, cuboid_width, 0],       # vertex 2
    [0, cuboid_width, 0],                   # vertex 3
    [0, 0, cuboid_height],                  # vertex 4
    [cuboid_length, 0, cuboid_height],      # vertex 5
    [cuboid_length, cuboid_width, cuboid_height],  # vertex 6
    [0, cuboid_width, cuboid_height]        # vertex 7
])

faces = [
    [vertices[0], vertices[1], vertices[2], vertices[3]],  # bottom
    [vertices[4], vertices[5], vertices[6], vertices[7]],  # top
    [vertices[0], vertices[1], vertices[5], vertices[4]],  # front
    [vertices[2], vertices[3], vertices[7], vertices[6]],  # back
    [vertices[1], vertices[2], vertices[6], vertices[5]],  # right
    [vertices[0], vertices[3], vertices[7], vertices[4]]   # left
]

# Create Poly3DCollection for cuboid
cuboid = Poly3DCollection(faces, alpha=0.2, linewidth=1, edgecolor='white')
cuboid.set_facecolor('gray')
cuboid.set_zorder(1)  # Set zorder to ensure cuboid is below the disc
ax.add_collection3d(cuboid)

def calculate_cycloid(theta):
    """Calculate the position of the disc center and the point tracing the cycloid"""
    center_x = cuboid_length/2
    center_y = disc_radius * theta
    center_z = cuboid_height + disc_radius

    y_cycloid = center_y - disc_radius * np.sin(theta)
    z_cycloid = center_z - disc_radius * np.cos(theta)

    return center_x, center_y, center_z, y_cycloid, z_cycloid

# Calculate full cycloid path for reference
full_path_y = []
full_path_z = []
full_path_x = []
for t in theta_vals:
    x, y, z, y_cyc, z_cyc = calculate_cycloid(t)
    full_path_x.append(x)
    full_path_y.append(y_cyc)
    full_path_z.append(z_cyc)

# Initialize empty path
cycloid_path, = ax.plot([], [], [], color='cyan', linestyle='-', linewidth=2,
                        alpha=0.8, zorder=30)

# Initialize trace point
trace_point, = ax.plot([], [], [], 'o', color='red', markersize=8, zorder=35)

# Function to create disc data with plot_surface compatible format
def create_disc_surfaces(center_x, center_y, center_z, theta):
    # Number of points for the disc surfaces
    n_radius = 20
    n_theta = 50

    # Create meshgrid for the disc
    r = np.linspace(0, disc_radius, n_radius)
    theta_disc = np.linspace(0, 2*np.pi, n_theta)
    R, T = np.meshgrid(r, theta_disc)
```

```python
    # Create initial disc coordinates (normal along x-axis)
    # Base position: disc lying in y-z plane
    Y = R * np.cos(T)
    Z = R * np.sin(T)
    X = np.zeros_like(Y)

    # Create rotation matrix for rotation around x-axis
    cos_t = np.cos(theta)
    sin_t = np.sin(theta)

    # Apply rotation to y and z coordinates only (rotating around x-axis)
    Y_rot = Y * cos_t - Z * sin_t
    Z_rot = Y * sin_t + Z * cos_t

    # Create top and bottom disc surfaces
    X_top = X + center_x
    Y_top = Y_rot + center_y
    Z_top = Z_rot + center_z + disc_thickness/2

    X_bottom = X + center_x
    Y_bottom = Y_rot + center_y
    Z_bottom = Z_rot + center_z - disc_thickness/2

    # For edge surface (adjust points for better depth sorting)
    theta_edge = np.linspace(0, 2*np.pi, n_theta)
    t_edge = np.linspace(-disc_thickness/2, disc_thickness/2, 8)
    T_edge, H_edge = np.meshgrid(theta_edge, t_edge)

    X_edge = np.zeros_like(T_edge) + center_x
    Y_edge = disc_radius * np.cos(T_edge) * cos_t - disc_radius * np.sin(T_edge) * sin_t + center_y
    Z_edge = disc_radius * np.cos(T_edge) * sin_t + disc_radius * np.sin(T_edge) * cos_t + center_z + H_edge

    return X_top, Y_top, Z_top, X_bottom, Y_bottom, Z_bottom, X_edge, Y_edge, Z_edge

# Initial surfaces for the disc
disc_top = None
disc_bottom = None
disc_edge = None

def animate(frame):
    nonlocal disc_top, disc_bottom, disc_edge

    # Current theta value
    theta = theta_vals[frame]

    # Calculate current positions
    center_x, center_y, center_z, y_cycloid, z_cycloid = calculate_cycloid(theta)

    # Remove previous disc surfaces
    if disc_top is not None:
        disc_top.remove()
    if disc_bottom is not None:
        disc_bottom.remove()
    if disc_edge is not None:
        disc_edge.remove()

    # Create new disc surfaces
    X_top, Y_top, Z_top, X_bottom, Y_bottom, Z_bottom, X_edge, Y_edge, Z_edge = create_disc_surfaces(
        center_x, center_y, center_z, theta)

    # Plot new surfaces with appropriate zorder
    disc_bottom = ax.plot_surface(X_bottom, Y_bottom, Z_bottom,
                                  color='skyblue', shade=True, antialiased=True,
                                  zorder=25)  # Bottom face

    disc_edge = ax.plot_surface(X_edge, Y_edge, Z_edge,
                                color='deepskyblue', shade=True, antialiased=True,
                                zorder=26)  # Edge surface

    disc_top = ax.plot_surface(X_top, Y_top, Z_top,
                               color='dodgerblue', shade=True, antialiased=True,
                               zorder=27)  # Top face gets highest zorder

    # Update the cycloid path (only show path up to current point)
    x_path = full_path_x[:frame+1]
    y_path = full_path_y[:frame+1]
    z_path = full_path_z[:frame+1]
    cycloid_path.set_data_3d(x_path, y_path, z_path)

    # Update trace point
```

```python
        trace_point.set_data_3d([center_x], [y_cycloid], [z_cycloid])

        return disc_top, disc_bottom, disc_edge, cycloid_path, trace_point

    # Set plot properties
    ax.set_xlim(x_limits)
    ax.set_ylim(y_limits)
    ax.set_zlim(z_limits)
    ax.set_box_aspect([cuboid_length, cuboid_width, z_limits[1]])
    ax.set_title(f"Disc Rolling on Cuboid - {num_revolutions} Revolutions", fontsize=12)

    # Remove axis panes and ticks
    ax.set_axis_off()

    ax.view_init(elev=15, azim=20)  # Set initial view angle

    # Create animation
    anim = FuncAnimation(fig, animate, frames=total_frames,
                         interval=interval_ms, blit=False, repeat=False)

    if save_anim and filename is not None:
        save_dir = "ANIMATIONS"
        os.makedirs(save_dir, exist_ok=True)
        filepath = os.path.join(save_dir, filename)
        print(f"Saving animation to {os.path.abspath(filepath)}...")
        anim.save(filepath, writer='pillow', fps=int(1000/interval_ms), dpi=100)
        print("Animation saved successfully!")
        plt.close(fig)
    else:
        plt.show()

    return anim

# Example usage
animation = animate_rolling_disc(
    disc_radius=1.0,
    disc_thickness=0.2,
    num_revolutions=2,
    save_anim=True,
    filename="disc_rolling_cycloid.gif"
)
```

Cycloidal Pendulum

Cycloidal Pendulums: Properties and Mathematical Analysis

Cycloidal pendulums represent a fascinating intersection of geometry, physics, and calculus. Unlike standard pendulums that follow an approximately circular path (for small angles), a cycloidal pendulum follows a precise cycloidal path due to the constraints imposed by two inverted cycloids flanking its swing path.

Key Properties of Cycloidal Pendulums

The most remarkable property of cycloidal pendulums is their **tautochrone** property. This means that regardless of where along the cycloidal path the pendulum bob starts its motion, it will take exactly the same time to reach the bottom. This is in stark contrast to ordinary pendulums, which have periods dependent on the amplitude of their swing.

This property made cycloidal pendulums historically significant for timekeeping. Christiaan Huygens, in the 17th century, realized that constraining a pendulum with two inverted cycloids would result in more accurate timekeeping since the period would remain constant regardless of the amplitude of the swing.

Mathematical Analysis of the Cycloidal Pendulum

Total Length Calculation

The total length of the pendulum cord is critical in this analysis. This length equals 4a, where a is the radius of the generating circle for the cycloid. This can be proven using the arc length formula for a parametric curve.

Let's calculate the arc length of half a cycloid. Given:

$$x(t) = r(t - \sin t), \quad y(t) = r(1 - \cos t)$$

Differentiate:

$$\frac{dx}{dt} = r(1-\cos t), \quad \frac{dy}{dt} = r\sin t$$

Using the arc length formula:

$$L = \int_0^\pi \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\, dt = \int_0^\pi r\sqrt{(1-\cos t)^2 + \sin^2 t}\, dt$$

Simplify the expression inside the root:

$$(1-\cos t)^2 + \sin^2 t = 2(1-\cos t)$$

So the integral becomes:

$$L = \int_0^\pi r\sqrt{2(1-\cos t)}\, dt = r\sqrt{2}\int_0^\pi \sqrt{1-\cos t}\, dt$$

Using the identity:

$$1-\cos t = 2\sin^2\left(\frac{t}{2}\right) \Rightarrow \sqrt{1-\cos t} = \sqrt{2}\left|\sin\left(\frac{t}{2}\right)\right|$$

So:

$$L = r\sqrt{2}\cdot\sqrt{2}\int_0^\pi \left|\sin\left(\frac{t}{2}\right)\right| dt = 2r\int_0^\pi \sin\left(\frac{t}{2}\right) dt$$

Now integrate:

$$= 2r\cdot\left[-2\cos\left(\frac{t}{2}\right)\right]_0^\pi = 4r$$

So the **arc length of a half cycloid** is $4r$. This is the required **length of the pendulum string** so the bob follows a cycloid. Therefore:

> **Total pendulum length = 4a**, where a is the radius of the generating circle.

Parametric Equation of the Pendulum Bob

When the pendulum swings, its cord wraps around the inverted cycloid constraints. For any point in the path, the cord can be divided into two sections:

- L_1: the part wrapped around the inverted cycloid
- L_2: the part that swings free

The point $P_1(x_1, y_1)$ where the pendulum leaves the inverted cycloid has parametric equations:

$$x_1 = a(\theta - \sin\theta)$$
$$y_1 = -a(1-\cos\theta)$$

The slope of L_2 is tangent to the inverted cycloid at P_1, given by:

$$m_{L_2} = \frac{dy}{dx} = \frac{dy/d\theta}{dx/d\theta} = -\frac{\sin\theta}{1-\cos\theta}$$

The length of L_1 is the arc length of the cycloid at θ, and the length of L_2 is found by subtracting L_1 from the total length $4a$:

$$L_2 = 4a - \text{arc length from } 0 \text{ to } \theta = 4a - \left(-4a\cos\frac{\theta}{2}\right)\Big|_0^\theta = 4a\cos\left(\frac{\theta}{2}\right)$$

Finding the Coordinates of the Pendulum Bob

To find the coordinates of the bob (x_2, y_2), we use the fact that we know:

- The coordinates of $P_1(x_1, y_1)$
- The length of segment L_2
- The slope of segment L_2

Using the slope and distance formulas, we can derive:

$$x_2 = x_1 \pm \frac{L_2}{\sqrt{m^2+1}}$$

$$y_2 = y_1 \pm \frac{m \cdot L_2}{\sqrt{m^2+1}}$$

Substituting the values we have:

$$\begin{aligned}
x_2 &= a(\theta - \sin\theta) \pm \frac{4a\cos\left(\frac{\theta}{2}\right)}{\sqrt{\frac{\sin^2\theta}{(1-\cos\theta)^2}+1}} \\
&= a(\theta - \sin\theta) \pm 4a\cos\left(\frac{\theta}{2}\right) \cdot \sqrt{\frac{1-\cos\theta}{2}} \\
&= a(\theta - \sin\theta) \pm 4a\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\theta}{2}\right) \\
&= a(\theta - \sin\theta) \pm 2a\sin\theta \\
&= a(\theta + \sin\theta) \quad \text{or} \quad a(\theta - 3\sin\theta)
\end{aligned}$$

and

$$\begin{aligned}
y_2 &= -a(1-\cos\theta) \pm \frac{-\frac{\sin\theta}{1-\cos\theta} \cdot 4a\cos\left(\frac{\theta}{2}\right)}{\sqrt{\frac{\sin^2\theta}{(1-\cos\theta)^2}+1}} \\
&= -a(1-\cos\theta) \pm \left(-\frac{\sin\theta}{1-\cos\theta}\right) 2a\sin\theta \\
&= -a(1-\cos\theta) \mp \frac{2a\sin^2\theta}{1-\cos\theta} \\
&= -a(1-\cos\theta) \mp 2a(1+\cos\theta) \\
&= -a(3+\cos\theta) \quad \text{or} \quad a(1+3\cos\theta)
\end{aligned}$$

There are two potential solutions due to the $\pm$ signs, corresponding to moving in opposite directions along the tangent line. The correct set corresponding to the pendulum bob's path is determined to be:

$$x_2 = a(\theta + \sin\theta) \quad , \quad y_2 = -a(3+\cos\theta)$$

Proving the Bob Follows a Shifted Cycloid

To show that these equations represent a shifted cycloid, we need to demonstrate that they match the parametric equations of a cycloid with appropriate shifts.

A standard cycloid has equations:

$$x = a(\theta - \sin\theta) \quad , \quad y = a(1-\cos\theta)$$

An inverted cycloid would have:

$$x = a(\theta - \sin\theta) \quad , \quad y = -a(1-\cos\theta)$$

For a shifted version, we replace θ with $\theta + \pi$ and apply horizontal and vertical shifts:

$$\begin{aligned}
x &= a((\theta+\pi) - \sin(\theta+\pi)) - \pi a \\
y &= -a(1-\cos(\theta+\pi)) - 2a
\end{aligned}$$

Using the identities $\sin(\theta+\pi) = -\sin\theta$ and $\cos(\theta+\pi) = -\cos\theta$:

$$\begin{aligned}
x &= a(\theta + \pi + \sin\theta) - \pi a = a(\theta + \sin\theta) \\
y &= -a(1+\cos\theta) - 2a = -3a - a\cos\theta
\end{aligned}$$

These match our derived equations for the pendulum bob, confirming that it follows a shifted cycloid path.

Physical Interpretation

This mathematical result has a beautiful physical interpretation: the pendulum bob, constrained by inverted cycloids, traces its own cycloid path. This is why the cycloidal pendulum exhibits the *tautochrone* property - all points on a cycloid take the same time to reach the bottom when acted upon by gravity alone.

The fact that the path remains a cycloid regardless of the amplitude of the swing made cycloidal pendulums theoretically perfect for timekeeping. However, practical limitations in physically implementing the precise cycloidal constraints meant that this theoretical advantage didn't fully translate to practical improvements in early clock design.

This elegant mathematical result shows how geometry can be applied to create physical systems with remarkable properties, demonstrating the deep connection between mathematical structures and physical phenomena.

Plotting the Cycloidal Pendulum

This code below creates a visualization of a cycloidal pendulum, which is a special type of pendulum that swings along a cycloid curve. Here's a brief breakdown:

Main Components

1. **Function Purpose**

```
def plot_cycloidal_pendulum(r=1, phi_max_deg=60, save_fig=False, filename=None):
```

Creates a plot showing a cycloidal pendulum with customizable parameters including radius and maximum swing angle.

2. **Key Elements Plotted**

- Inverted cycloid curves ("cheeks")
- Pendulum bob position
- Path of the pendulum bob
- Wrapped (L1) and free (L2) portions of the string
- Point P (unwrap point)

3. **Mathematical Core**

```
def inverted_cycloid_equation(theta):
    x = r * (theta - np.sin(theta))
    y = - r * (1 - np.cos(theta))
    return x, y
```

This nested function calculates the parametric equations for the inverted cycloid curve.

Visualization Features

- Uses dark background style
- Creates an equal-aspect plot
- Includes labels and legend
- Option to save the figure to a specified directory

In [ ]:

```
def plot_cycloidal_pendulum(r=1, phi_max_deg=60, save_fig=False, filename=None):
    """
    Plot a cycloidal pendulum showing the inverted cycloid cheeks, pendulum bob, and its path.

    Parameters:
    -----------
    r : float
        Radius of the rolling circle generating the cycloid (default: 1.0)
    phi_max_deg : float
        Maximum swing angle in degrees (default: 60)
    save_fig : bool
        Whether to save the figure (default: False)
    filename : str, optional
        Name of the file to save the figure (default: None)
    """
    # Convert angle from degrees to radians
    phi_max = np.radians(phi_max_deg)

    # Set style and create figure
    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.set_title(f"Cycloidal Pendulum (ϕ_max = {phi_max_deg}°)", fontsize=14, pad=10)

    # Generate theta values for plotting
    theta_vals = np.linspace(-np.pi, np.pi, 200)

    # Set plot limits with padding
    y_padding = 0.5
    x_max = abs(r * (phi_max + np.sin(phi_max)))  # Maximum x-coordinate of bob
    ax.set_xlim(-x_max - y_padding, x_max + y_padding)
    ax.set_ylim(-(4*r + 2.8*y_padding), y_padding)
    ax.set_xticks([])
    ax.set_yticks([])
    ax.grid(False)
    ax.set_aspect('equal')
```

```python
    # Rest of the function remains the same...
    def inverted_cycloid_equation(theta):
        x = r * (theta - np.sin(theta))
        y = - r * (1 - np.cos(theta))
        return x, y

    # Plot base line
    ax.axhline(0, color='gray', linewidth=2)

    # Plot cycloid cheeks
    x_cycloid, y_cycloid = inverted_cycloid_equation(theta_vals)
    ax.plot(x_cycloid, y_cycloid, color='gray', lw=2, label='Cycloid cheeks')
    ax.fill_between(x_cycloid, 0, y_cycloid, color='dimgray', alpha=0.5)

    # Calculate unwrap point P
    point_angle = phi_max  # Use the input angle
    x_point, y_point = inverted_cycloid_equation(point_angle)
    # Plot point P
    ax.plot(x_point, y_point, 'yo', ms=8, label='Point P (unwrap point)', zorder=5)

    # Plot L1 (wrapped portion of string)
    L1_curve_theta_vals = np.linspace(0, point_angle, 50)
    L1_x_vals, L1_y_vals = inverted_cycloid_equation(L1_curve_theta_vals)
    ax.plot(L1_x_vals, L1_y_vals, color='orange', ls='-', lw=2, label='L1 (wrapped portion)')

    # Calculate and plot pendulum bob position
    xB = r * (np.sin(point_angle) + point_angle)
    yB = - r * (3 + np.cos(point_angle))
    ax.plot(xB, yB, 'ro', ms=16, label=f'Pendulum bob (ϕ_max = {phi_max_deg}°)', zorder=5)

    # Plot L2 (free portion of string)
    ax.plot([x_point, xB], [y_point, yB], 'r-', lw=2, label='L2')

    # Plot pendulum bob path
    bob_angle = np.linspace(-phi_max, phi_max, 200)
    l2_x_vals = r * (np.sin(bob_angle) + bob_angle)
    l2_y_vals = - r * (3 + np.cos(bob_angle))
    ax.plot(l2_x_vals, l2_y_vals, color='gray', ls='--', lw=1.5, label='Pendulum path')

    # Add text labels
    ax.text(x_point+0.1, y_point+0.3, 'P', va='top', ha='right', fontsize=12, color='yellow')
    ax.text(xB-0.2, yB-0.4, 'Bob', fontsize=12, color='red')
    ax.text(x_cycloid[20], y_cycloid[20]-0.2, 'Cycloid cheek', fontsize=10, color='white')

    # Add legend
    ax.legend(loc='lower right', fontsize=8)
    # plt.tight_layout()
    # Save figure if requested
    if save_fig and filename is not None:
        save_dir = "IMAGES/CYCLOIDS"
        os.makedirs(save_dir, exist_ok=True)
        filename = os.path.join(save_dir, filename)
        print(f"Saving figure to {os.path.abspath(filename)}...")
        plt.savefig(filename, bbox_inches='tight', dpi=200)
        print(f"Figure saved successfully!")
        plt.close(fig)
    else:
        plt.show()

# Example usage with different angles
plot_cycloidal_pendulum(phi_max_deg=90, save_fig=True, filename="pendulum_90deg.png")
```

Animating the Pendulum Motion

In [ ]:

```python
class CycloidalPendulumAnimator:
    """
    Animates one or more cycloidal pendulums with complete oscillations.
    """
    def __init__(self,
                 phi_max_deg_list,
                 num_oscillations=5,
                 oscillation_duration=2.0, # Duration of one full oscillation (forth and back)
                 fps=30,
                 save_animation=False,
                 animation_filename=None,
                 r=1.0):
        """
        Initializes the Cycloidal Pendulum Animator.
```

```python
        Parameters:
        -----------
        phi_max_deg_list : float or list[float]
            Maximum swing angle(s) in degrees. If a float, animates one pendulum.
            If a list, animates multiple pendulums with the specified max angles.
        num_oscillations : int
            Number of complete oscillations to simulate (default: 5).
        oscillation_duration : float
            The time period (in seconds) for one complete oscillation (default: 2.0).
        fps : int
            Frames per second for the animation (default: 30).
        save_animation : bool
            Whether to save the animation to a file (default: False).
        animation_filename : str, optional
            Name of the file to save the animation. If None and save_animation is True,
            a default name is generated.
        r : float
            Radius of the generating circle for the cycloid (default: 1.0).
        """
        self.r = r
        self.num_oscillations = num_oscillations
        self.oscillation_duration = float(oscillation_duration) # Ensure it's float
        self.fps = fps
        self.save_animation = save_animation
        self.animation_filename = animation_filename

        # Handle single angle input
        if isinstance(phi_max_deg_list, (int, float)):
            self.phi_max_deg_list = [float(phi_max_deg_list)]
        elif isinstance(phi_max_deg_list, list):
            self.phi_max_deg_list = [float(angle) for angle in phi_max_deg_list]
        else:
            raise TypeError("phi_max_deg_list must be a number or a list of numbers.")

        self.num_pendulums = len(self.phi_max_deg_list)
        self.phi_max_list = [np.radians(phi) for phi in self.phi_max_deg_list]

        # Calculate total animation duration and frames
        self.total_duration = self.num_oscillations * self.oscillation_duration
        self.num_frames = int(self.total_duration * self.fps)
        self.total_length = 4 * self.r # Bob path hangs down by 4r

        # Animation elements (will be populated in setup_plot)
        self.fig = None
        self.ax = None
        self.lines_L1 = [] # Tangent segment on left cheek (only for single pendulum)
        self.lines_L2 = [] # Straight segment from P to Bob (only for single pendulum)
        self.points_bob = [] # List to hold bob plot objects
        self.point_P = None  # Unwrapping point (only for single pendulum)
        self.counter_text = None
        self.cheek_fill = None
        self.bob_colors = []

    # --- Parametric Equations ---
    def cheek_cycloid(self, theta):
        """Inverted cycloid for the cheeks (cusp at (0,0))."""
        x = self.r * (theta - np.sin(theta))
        y = -self.r * (1 - np.cos(theta)) # Negative y to go downwards
        return x, y

    def bob_cycloid(self, phi):
        """Inverted cycloid path for the bob (cusp at (0, -4r))."""
        # Derived from involute properties or Huygens' construction
        x = self.r * (phi + np.sin(phi))
        y = -self.r * (3 + np.cos(phi)) # Shifted down by 3r, cusp at y=-4r when phi=0
        return x, y

    def _setup_plot(self):
        """Sets up the matplotlib figure and axes."""
        plt.style.use('dark_background')
        self.fig, self.ax = plt.subplots(figsize=(10, 8))
        self.ax.set_title(f"Cycloidal Pendulum Animation\n{self.num_oscillations} oscillations", fontsize=14, pad
        self.fig.subplots_adjust(left=0.1, right=0.95, top=0.9, bottom=0.1)

        # Determine plot limits dynamically based on max phi
        max_phi_abs = max(abs(phi) for phi in self.phi_max_list) if self.phi_max_list else np.pi
        max_x_bob = self.r * (max_phi_abs + np.sin(max_phi_abs))
        padding = 0.5 * self.r
        self.ax.set_xlim(-max_x_bob - padding, max_x_bob + padding)
        self.ax.set_ylim(-self.total_length - 2*padding, padding) # y goes from 0 down to -4r
```

```
        self.ax.set_xticks([])
        self.ax.set_yticks([])
        self.ax.grid(False)
        self.ax.set_aspect('equal', adjustable='box')

        # --- Static Elements ---
        theta_cheek_vals = np.linspace(-np.pi, np.pi, 400)
        x_cheeks, y_cheeks = self.cheek_cycloid(theta_cheek_vals)
        self.ax.plot(x_cheeks, y_cheeks, color='gray', lw=2, label='Cycloidal Cheeks' if self.num_pendulums == 1

        # Fill area between x-axis (y=0) and cheeks
        self.cheek_fill = self.ax.fill_between(x_cheeks, 0, y_cheeks, color='dimgray', alpha=0.5)

        # Plot the path of the bob
        phi_bob_path = np.linspace(-max_phi_abs, max_phi_abs, 200)
        x_bob_path, y_bob_path = self.bob_cycloid(phi_bob_path)
        self.ax.plot(x_bob_path, y_bob_path, color='white', ls=':', lw=1, alpha=0.7,
                     label='Bob Path' if self.num_pendulums == 1 else None)

        self.ax.axhline(0, color='gray', lw=2) # Support line
        self.ax.plot([0, 0], [0, -self.total_length], color='gray', lw=1) # Vertical line from origin to -4r

        # --- Animated Elements Setup ---
        color_cycle = itertools.cycle(plt.cm.tab10.colors) # Use a colormap for distinct colors

        for i in range(self.num_pendulums):
            bob_color = next(color_cycle)
            self.bob_colors.append(bob_color)
            # Bob point
            # Different labels based on number of pendulums
            if self.num_pendulums == 1:
                bob_label = f"Bob ($\\phi_{{\\mathrm{{max}}}} = {self.phi_max_deg_list[i]}^\\circ$)"
            else:
                bob_label = f"$\\phi = {self.phi_max_deg_list[i]}^\\circ$"
            point_bob, = self.ax.plot([], [], 'o', ms=13, color=bob_color, label=bob_label)
            self.points_bob.append(point_bob)

            # Initialize lines L1 and L2
            line_L1, = self.ax.plot([], [], '-', lw=2, color=bob_color)
            line_L2, = self.ax.plot([], [], '-', lw=2, color=bob_color)
            self.lines_L1.append(line_L1)
            self.lines_L2.append(line_L2)

            # Unwrapping Point P - only for single pendulum
            if self.num_pendulums == 1:
                self.point_P, = self.ax.plot([], [], 'o', ms=8, color='yellow',
                                             label='P (Unwrap Point)')

        # Oscillation counter text
        self.counter_text = self.ax.text(0.02, 0.98, '', transform=self.ax.transAxes,
                                         color='white', fontsize=12, ha='left', va='top')

        # Show legend only for single pendulum
        # if self.num_pendulums == 1:
        self.ax.legend(loc='lower right', fontsize=10)


    def _update(self, frame):
        """Updates the animation elements for each frame."""
        # Calculate current time in the animation
        t_anim = frame / self.fps

        # Calculate the phase within a single oscillation (0 to 2*pi)
        # Use modulo to handle multiple oscillations correctly
        phase = (t_anim / self.oscillation_duration) * 2 * np.pi

        # One oscillation is completed every 'oscillation_duration' seconds
        current_oscillation_float = t_anim / self.oscillation_duration

        # Check if this is the last frame
        is_last_frame = frame == self.num_frames - 1

        if is_last_frame:
            # On the last frame, show total oscillations
            completed_oscillations = self.num_oscillations
        else:
            # Floor to get completed oscillations during animation
            completed_oscillations = int(np.floor(current_oscillation_float))
            # Prevent counter exceeding total number
```

```
            if completed_oscillations > self.num_oscillations:
                completed_oscillations = self.num_oscillations

        self.counter_text.set_text(f'Oscillations: {completed_oscillations}/{self.num_oscillations}')

        artists_to_update = [self.counter_text]

        for i in range(self.num_pendulums):
            phi_max = self.phi_max_list[i]
            current_phi = phi_max * np.cos(phase) # Simple harmonic motion for angle phi

            # Calculate Bob's position
            xB, yB = self.bob_cycloid(current_phi)
            self.points_bob[i].set_data([xB], [yB])
            artists_to_update.append(self.points_bob[i])

            # Angle theta corresponding to the unwrapping point P on the cheek
            current_theta_p = current_phi

            # Update L1 and L2 for all pendulums
            xP, yP = self.cheek_cycloid(current_theta_p)

            # Calculate points for L1 (wrapped part on the cheek)
            if np.isclose(current_theta_p, 0):
                xL1, yL1 = [], [] # No wrapped part if at the bottom
            else:
                # Generate points along the cheek from the cusp (0) to P (current_theta_p)
                num_L1_points = 50
                if current_theta_p > 0:
                    theta_L1_vals = np.linspace(0, current_theta_p, num_L1_points)
                else: # current_theta_p < 0
                    theta_L1_vals = np.linspace(current_theta_p, 0, num_L1_points)
                xL1, yL1 = self.cheek_cycloid(theta_L1_vals)

            # Update plot elements
            self.lines_L1[i].set_data(xL1, yL1)
            self.lines_L2[i].set_data([xP, xB], [yP, yB])
            artists_to_update.extend([self.lines_L1[i], self.lines_L2[i]])

            # Update point P only for single pendulum
            if self.num_pendulums == 1:
                self.point_P.set_data([xP], [yP])
                artists_to_update.append(self.point_P)


    def animate(self):
        """Creates and runs (or saves) the animation."""
        self._setup_plot() # Setup the plot elements

        if self.fig is None:
            raise ValueError("Figure was not properly initialized")

        ani = FuncAnimation(self.fig, self._update,
                            frames=self.num_frames,
                            interval=int(1000 / self.fps),
                            blit=False,
                            repeat=False)

        if self.save_animation:
            if not self.animation_filename:
                 # Generate a default filename if none provided
                 angles_str = "_".join(map(str, self.phi_max_deg_list))
                 self.animation_filename = f"cycloidal_pendulum_{self.num_pendulums}p_{angles_str}deg_{self.num_c

            save_dir = "ANIMATIONS/CYCLOIDS"
            os.makedirs(save_dir, exist_ok=True)
            filepath = os.path.join(save_dir, self.animation_filename)
            print(f"Saving animation to {os.path.abspath(filepath)}...")
            try:
                # Using Pillow writer for GIF
                ani.save(filepath, writer='pillow', fps=self.fps)
                print("Animation saved successfully!")
            except Exception as e:
                 print(f"Error saving animation: {e}")
                 print("Attempting to show animation instead.")
                 plt.show()
            finally:
                 plt.close(self.fig) # Close the figure after saving

        else:
```

```
            plt.show()

        return ani # Return the animation object



# Example 1: Single pendulum
print("Running single pendulum animation...")
animator1 = CycloidalPendulumAnimator(
    phi_max_deg_list=120,       # Single max angle
    num_oscillations=3,
    oscillation_duration=5, # Each oscillation takes 5 seconds
    fps=20,
    save_animation=True,       # Set to True to save
    animation_filename="single_pendulum_120deg.gif"
)
# anim1 = animator1.animate() # Uncomment to run


# Example 2: Multiple pendulums
print("\nRunning multiple pendulum animation...")
animator2 = CycloidalPendulumAnimator(
    phi_max_deg_list=[15, 45, 90, 135, 180], # List of max angles
    num_oscillations=5,
    oscillation_duration=6,    # Each oscillation takes 6 seconds
    fps=20,
    save_animation=True,        # Set to True to save
    # animation_filename="multi_pendulum_15_45_90_135_180deg.gif"
)
anim2 = animator2.animate()
```

Cycloidal Pendulum vs. Simple Harmonic Motion (SHM)

Aspect	Simple Harmonic Motion (SHM)	Cycloidal Pendulum Motion
Definition	Periodic motion where restoring force is proportional to displacement (e.g., springs, small-angle pendulum).	Motion of a pendulum constrained to follow an **inverted cycloid** path, rather than a circular arc.
Path	Typically **linear** or **circular** (arc) depending on system.	Follows an **inverted cycloid** — a special curve traced by a point on a rolling circle.
Restoring Force	$F = -kx$ (Hooke's Law) or $F = -mg\sin\theta$ (for a pendulum, approximated as $F \approx -mg\theta$ for small angles).	Restoring force leads to **exact isochronous motion** because the shape of the path compensates for non-linear restoring forces.
Period Dependence	For a simple pendulum, period $T = 2\pi\sqrt{\frac{l}{g}}$ — **valid only for small amplitudes**. At larger amplitudes, period increases.	Period remains **constant** regardless of amplitude due to the **tautochrone** property of the cycloid.
Tautochronism	SHM is only approximately tautochronous at small displacements.	**Perfectly tautochronous**: all starting points take the **same time** to reach the lowest point.
Equation of Motion	For ideal SHM: $x(t) = A\cos(\omega t + \phi)$	Complex parametric form: $x = r(\theta + \sin\theta)$ $y = -3r - r\cos\theta$
Origin of Isochronism	Approximation due to linear restoring force (valid for small oscillations).	Comes from the geometric property of the **cycloid**, not from a linear force.
Real-world Application	Useful approximation in physics; accurate only under limited conditions.	Used by **Huygens** to improve clocks — though difficult to implement mechanically.

Summary:

- **SHM** is a general approximation valid for many systems, but **only exact under idealized conditions**.
- A **cycloidal pendulum** solves the amplitude-dependence issue by ensuring **exact isochronous behavior** at **all amplitudes**, thanks to the unique shape of the **inverted cycloid**.

Tautochrone Problem

The Tautochrone Curve: Definition and History

The **tautochrone** (from the Greek *tauto* meaning "same" and *chronos* meaning "time") is a curve for which the **time taken by a particle sliding without friction under the influence of gravity to its lowest point is independent of its starting point**.

The solution to this problem is the **cycloid** — the curve traced by a point on the rim of a circle as it rolls along a straight line.

This problem was famously solved by **Christiaan Huygens** in the 17th century, who even designed a clock using the cycloidal path to ensure isochronous motion.

Parametric Equation of a Cycloid

A cycloid generated by a circle of radius a, with the lowest point at the origin, is given parametrically by:

$$x(\theta) = a(\theta - \sin\theta)$$
$$y(\theta) = a(1 - \cos\theta)$$

Here, $\theta \in [0, \pi]$ traces the arc from the highest point of the cycloid to the bottom.

Property of Tautochronism

Let's consider a frictionless particle sliding under gravity along a cycloidal path from any starting point. The remarkable result is:

> **The time taken by the particle to reach the lowest point is the same regardless of the initial point on the curve.**

This property makes the cycloid the solution to the **tautochrone problem**.

Derivation of Time to Reach the Bottom

Let's derive the time taken for the particle to reach the bottom from any point on the cycloid.

We use the **conservation of energy** and the **arc length** of the cycloid.

1. Arc Length of the Cycloid

The infinitesimal arc length ds is given by:

$$ds = \sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2}\, d\theta$$

We calculate the derivatives:

$$\frac{dx}{d\theta} = a(1 - \cos\theta), \quad \frac{dy}{d\theta} = a\sin\theta$$

$$ds = a\sqrt{(1-\cos\theta)^2 + \sin^2\theta}\, d\theta = a\sqrt{2(1-\cos\theta)}\, d\theta$$

Using the identity $1 - \cos\theta = 2\sin^2\left(\frac{\theta}{2}\right)$, we simplify:

$$ds = 2a\sin\left(\frac{\theta}{2}\right) d\theta$$

2. Speed from Energy Conservation

The total energy at the starting point (taking the bottom as zero potential energy) is potential:

$$E = mgy(\theta) = mga(1 - \cos\theta)$$

At any point, all potential energy is converted to kinetic energy:

$$\frac{1}{2}mv^2 = mga(1 - \cos\theta) \Rightarrow v = \sqrt{2ga(1 - \cos\theta)}$$

3. Time Integral

$$dt = \frac{ds}{v} = \frac{2a\sin\left(\frac{\theta}{2}\right)}{\sqrt{2ga(1-\cos\theta)}} d\theta$$

Using $1 - \cos\theta = 2\sin^2\left(\frac{\theta}{2}\right)$, we get:

$$v = \sqrt{4ga\sin^2\left(\frac{\theta}{2}\right)} = 2\sqrt{ga}\sin\left(\frac{\theta}{2}\right)$$

So,

$$dt = \frac{2a\sin\left(\frac{\theta}{2}\right)}{2\sqrt{ga}\sin\left(\frac{\theta}{2}\right)}d\theta = \frac{a}{\sqrt{ga}}d\theta = \sqrt{\frac{a}{g}}d\theta$$

Now integrate from 0 to θ (corresponding to the starting point):

$$t = \int_0^\theta \sqrt{\frac{a}{g}}d\theta = \sqrt{\frac{a}{g}}\theta$$

But the full descent corresponds to $\theta = \pi$, so the time to the bottom is:

$$T = \sqrt{\frac{a}{g}}\pi$$

✅ **Key Result: The time $T = \pi\sqrt{a/g}$ is constant and independent of the initial position.**

Implications and Applications

- The cycloid is not only tautochronous but also **brachistochrone** (the curve of quickest descent).
- Used by Huygens in designing accurate pendulum clocks.
- Demonstrates an early example of variational principles in physics.

In [ ]:
```
class TautochroneAnimator:
    """
    Animates the tautochrone problem, showing particles sliding down
    a cycloidal path from different starting points. Demonstrates
    that they reach the bottom simultaneously.
    """
    def __init__(self,
                 initial_angles_deg,
                 r=1.0,
                 g=9.81, # Acceleration due to gravity
                 fps=30,
                 save_anim=False,
                 filename=None,
                 marker_size=15,
                 ball_offset_factor=0.07): # Factor of r to offset ball center
        """
        Initializes the Tautochrone Animator.

        Parameters:
        -----------
        initial_angles_deg : float or list[float]
            Starting angle(s) in degrees along the cycloid (0 < angle < 180).
            Angle 0 is the cusp, Angle 180 is the lowest point.
            If a float, animates one ball. If a list, animates multiple balls.
        r : float
            Radius of the generating circle for the cycloid (default: 1.0).
        g : float
            Acceleration due to gravity (default: 9.81). Affects the time scale.
        fps : int
            Frames per second for the animation (default: 30).
        save_anim : bool
            Whether to save the animation to a file (default: False).
        filename : str, optional
            Name of the file to save the animation. If None and save_anim is True,
            a default name is generated.
        marker_size : float
            Size of the marker used for the balls (default: 15).
        ball_offset_factor : float
            Factor determining how much the ball's center is offset perpendicularly
            from the cycloid curve to make it look like it's resting on the curve.
            Expressed as a fraction of 'r'. (default: 0.07)
        """
        self.r = r
        self.g = g
        self.fps = fps
        self.save_anim = save_anim
        self.filename = filename
        self.marker_size = marker_size
        self.ball_offset = ball_offset_factor * self.r

        # Handle single angle input
        if isinstance(initial_angles_deg, (int, float)):
```

```python
            self.initial_angles_deg = [float(initial_angles_deg)]
        elif isinstance(initial_angles_deg, list):
            self.initial_angles_deg = [float(angle) for angle in initial_angles_deg]
        else:
            raise TypeError("initial_angles_deg must be a number or a list of numbers.")

        # Validate angles (must be less than 180 degrees)
        if not all( angle < 180 for angle in self.initial_angles_deg):
            raise ValueError("Initial angles must be less than 180 degrees.")

        self.theta0_list = [np.radians(angle) for angle in self.initial_angles_deg]
        self.num_balls = len(self.theta0_list)

        # Calculate physics constants and timing
        self.omega = np.sqrt(self.g / (4 * self.r))
        self.time_to_bottom = np.pi / (2 * self.omega) # Time = pi * sqrt(r/g)

        # Set animation duration slightly longer than time_to_bottom
        self.duration = self.time_to_bottom * 1.5
        self.num_frames = int(self.duration * self.fps)

        # Animation elements (will be populated in _setup_plot)
        self.fig = None
        self.ax = None
        self.balls = []      # List to hold ball plot objects
        self.timers = []     # List to hold timer text objects
        self.ball_colors = []

    def _cycloid_path(self, theta):
        """Parametric equation for the inverted cycloid path."""
        # Cusp at (0, 0), lowest point at (pi*r, -2*r) for theta=pi
        x = self.r * (theta - np.sin(theta))
        y = -self.r * (1 - np.cos(theta))
        return x, y

    def _get_theta_at_time(self, theta0, t):
        """Calculates the angular position theta(t) of a ball."""
        if t >= self.time_to_bottom:
            # Ball has reached or passed the bottom, keep it there
            return np.pi
        elif t <= 0:
            return theta0
        else:
            # Argument for arccos: cos(theta0/2) * cos(omega*t)
            # This value is guaranteed to be between -cos(theta0/2) and +cos(theta0/2)
            # Since 0 < theta0 < pi, 0 < theta0/2 < pi/2, so cos(theta0/2) is between 0 and 1.
            # The argument is thus always between -1 and 1.
            cos_val = np.cos(theta0 / 2.0) * np.cos(self.omega * t)
            # Ensure value is clipped due to potential floating point inaccuracies
            cos_val = np.clip(cos_val, -1.0, 1.0)
            current_theta_half = np.arccos(cos_val)
            return 2.0 * current_theta_half

    def _get_offset_position(self, theta: float) -> tuple[float, float]:
        """ Calculates the center position for the ball marker so it appears
            to rest *on* the curve rather than being centered *on* it. """
        x_curve, y_curve = self._cycloid_path(theta)

        # Handle endpoints where normal calculation can be problematic
        if np.isclose(theta, np.pi): # Bottom point
             x_center, y_center = self._cycloid_path(np.pi)
             # Offset directly upwards
             y_center += self.ball_offset
             return x_center, y_center
        elif np.isclose(theta, 0): # Top cusp
             x_center, y_center = self._cycloid_path(0)
             # Offset slightly right (along the outward normal at the cusp)
             x_center += self.ball_offset # Normal is (1, 0) here
             return x_center, y_center

        # Calculate the outward normal vector (normalized)
        # Tangent vector: (dx/dtheta, dy/dtheta) = (r(1-cos(theta)), -r*sin(theta))
        # Normal vector (outward): (r*sin(theta), r(1-cos(theta)))
        # Using half-angle identities:
        # Normal = (2r*sin(th/2)cos(th/2), 2r*sin^2(th/2))
        # Magnitude = 2r*sin(th/2)  (for 0 < theta < 2pi)
        # Normalized outward normal = (cos(theta/2), sin(theta/2))
        nx = np.cos(theta / 2.0)
        ny =  np.sin(theta / 2.0)
```

```python
        # Ensure normalization (prevents issues if sin(theta/2) is zero, handled above)
        norm = np.sqrt(nx**2 + ny**2)
        if norm > 1e-9: # Avoid division by zero
            nx /= norm
            ny /= norm

        # Offset the center position along the outward normal
        x_center = x_curve + self.ball_offset * nx
        y_center = y_curve + self.ball_offset * ny

        return x_center, y_center


    def _setup_plot(self):
        """Sets up the matplotlib figure and axes."""
        plt.style.use('dark_background')
        self.fig, self.ax = plt.subplots(figsize=(10, 8))
        formula = r"$T = \frac{\pi}{2\omega} = \pi \sqrt{\frac{r}{g}}$"
        self.ax.set_title(f"Tautochrone Problem (Cycloid)\nTime to bottom: {self.time_to_bottom:.3f}s\n{formula}'
                        fontsize=14, pad=10)
        self.fig.subplots_adjust(left=0.1, right=0.9, top=0.92, bottom=0.05)

        # Set plot limits
        x_padding = self.r * 0.2
        y_padding = self.r * 0.2
        self.ax.set_xlim(-x_padding, self.r * np.pi + x_padding)
        self.ax.set_ylim(-2 * self.r - y_padding, y_padding + self.ball_offset * 2)
        self.ax.set_aspect('equal', adjustable='box')
        self.ax.grid(False)
        self.ax.set_xticks([])
        self.ax.set_yticks([])

        # Plot the static cycloid path
        theta_path = np.linspace(0, np.pi, 400)
        x_path, y_path = self._cycloid_path(theta_path)
        self.ax.plot(x_path, y_path, color='gray', lw=2)
        self.ax.plot([-x_padding, 0], [0, 0], color='gray', lw=2)
        self.ax.fill_between(x_path, y_path, -2*self.r - y_padding, color='dimgray', alpha=0.3)
        self.ax.fill_between([-x_padding, 0], 0, -2*self.r - y_padding, color='dimgray', alpha=0.3)

        # Initialize animated elements
        color_cycle = itertools.cycle(plt.cm.tab10.colors)

        for i in range(self.num_balls):
            color = next(color_cycle)
            self.ball_colors.append(color)
            theta0 = self.theta0_list[i]

            # Calculate initial offset position
            x0, y0 = self._get_offset_position(theta0)

            # Create ball object with label for legend
            ball, = self.ax.plot([x0], [y0], 'o', ms=self.marker_size,
                                 color=color, zorder=5,
                                 label=f'Ball {i+1} ({self.initial_angles_deg[i]}°): 0.00s')
            self.balls.append(ball)

        # Create legend
        self.legend = self.ax.legend(loc='upper right', fontsize=10, title='Time', title_fontsize=12, markerscale

    def _update(self, frame):
        """Updates the animation elements for each frame."""
        t = frame / self.fps
        time_display = min(t, self.time_to_bottom)

        artists_to_update = []

        for i in range(self.num_balls):
            theta0 = self.theta0_list[i]
            current_theta = self._get_theta_at_time(theta0, t)
            x_center, y_center = self._get_offset_position(current_theta)

            # Update ball position
            self.balls[i].set_data([x_center], [y_center])
            # Update legend label
            self.balls[i].set_label(f'Ball {i+1} ({self.initial_angles_deg[i]}°): {time_display:.2f}s')
            artists_to_update.append(self.balls[i])

        # Update legend
        self.legend.get_texts()[0].set_text(self.balls[0].get_label())
```

```python
        for i in range(1, len(self.balls)):
            self.legend.get_texts()[i].set_text(self.balls[i].get_label())
        artists_to_update.append(self.legend)

        return artists_to_update


    def animate(self):
        """Creates and runs (or saves) the animation."""
        self._setup_plot()

        if self.fig is None:
            raise ValueError("Figure was not properly initialized")

        ani = FuncAnimation(self.fig, self._update,
                            frames=self.num_frames,
                            interval=int(1000 / self.fps),
                            blit=False,
                            repeat=True)

        if self.save_anim:
            if not self.filename:
                 angles_str = "_".join(map(str, self.initial_angles_deg))
                 self.filename = f"tautochrone_{self.num_balls}balls_{angles_str}deg.gif"

            save_dir = "ANIMATIONS/TAUTOCHRONE"
            os.makedirs(save_dir, exist_ok=True)
            filepath = os.path.join(save_dir, self.filename)
            print(f"Saving animation to {os.path.abspath(filepath)}...")
            try:
                # Using Pillow writer for GIF is often reliable
                ani.save(filepath, writer='pillow', fps=self.fps)
                print("Animation saved successfully!")
            except Exception as e:
                 print(f"Error saving animation: {e}")
                 print("Make sure you have necessary writers installed (e.g., Pillow).")
                 print("Attempting to show animation instead.")
                 plt.show()
            finally:
                 plt.close(self.fig) # Close the figure after saving/error

        else:
            plt.show()

        return ani # Return the animation object


# Example 1: Single ball starting at 45 degrees
print("Running single ball animation (45 deg)...")
animator1 = TautochroneAnimator(initial_angles_deg=45, r=1, g=9.81, fps=30, marker_size=18)
# anim1 = animator1.animate() # Uncomment to run

# Example 2: Four balls starting at different angles
print("\nRunning multiple ball animation (30, 90, 150 deg)...")
animator2 = TautochroneAnimator(
    initial_angles_deg=[0, 45, 90, 150],
    r=20,
    g=10,
    fps=10,
    marker_size=20,
    save_anim=True # Set to True to save
)
anim2 = animator2.animate()
```

References

- Cycloid - MathWorld
- Hypotrochoid - MathWorld
- Epitrochoid - MathWorld
- Cycloid - MathMonks
- Celebrating the Cycloid - Medium
- Cycloidal Pendulum - PSU
- Tautochrone Problem - MathWorld
- MathCurves