Generalized Orthogonal Vectors Generator and Visualizer

Technical Documentation

March 4, 2025

Abstract

This document provides a comprehensive guide to the generalized implementation of the Orthogonal Vectors Generator and Visualizer. The implementation offers a modular and configurable approach to creating and visualizing three orthogonal vectors from a given origin point. This document covers the mathematical formulation, implementation details, API reference, usage examples, and visualization techniques.

Contents

1 Introduction

In three-dimensional space, orthogonal vectors are perpendicular to each other, meaning their dot product equals zero. This document describes a generalized implementation for generating and visualizing three orthogonal vectors from a given origin point.

1.1 Project Overview

The Generalized Orthogonal Vectors Generator and Visualizer is a Python package that provides a modular and configurable approach to vector generation and visualization. It is designed to be flexible, extensible, and easy to use, both as a command-line tool and as a Python package.

1.2 Key Features

The generalized implementation offers the following key features:

- **Modular Architecture**: The implementation is divided into separate modules for vector calculations, visualization, and configuration management.
- **Configurability**: All aspects of vector generation and visualization can be configured through a unified configuration system.
- **Command-line Interface**: A comprehensive command-line interface allows for easy use without writing Python code.
- **Configuration File Support**: Configurations can be saved to and loaded from JSON files.
- **Multiple Visualization Options**: Both 3D visualization and various 2D projections are supported.
- **Plot Saving**: Plots can be saved to files instead of being displayed interactively.
- **Python Package**: The implementation can be used as a Python package, allowing for integration into other projects.

1.3 Package Structure

The package is organized into the following modules:

- `vector_utils.py`: Contains functions for vector calculations, including the creation of orthogonal vectors and checking their orthogonality.
- `visualization.py`: Provides functions for visualizing vectors in 3D and 2D projections.
- `config.py`: Implements a configuration management system with the `VectorConfig` class.
- `main.py`: Provides a command-line interface for the package.
- `example.py`: Contains example scripts demonstrating different use cases.
- `__init__.py`: Package initialization and exports.

1.4 Document Structure

This document is organized as follows:

- **Mathematical Formulation**: Explains the mathematical basis for generating orthogonal vectors.
- **Implementation**: Describes the implementation details of the package.
- **API Reference**: Provides a reference for the package's API.
- **Usage Examples**: Shows examples of how to use the package.
- **Visualization**: Explains the visualization techniques used.

- **Configuration**: Describes the configuration management system.
- **Command-line Interface**: Documents the command-line interface.
- **Example Results**: Shows example results for different configurations.
- **Conclusion**: Summarizes the document and discusses potential future work.
- **Appendix**: Contains additional information, including complete code listings.

2 Mathematical Formulation

This section describes the mathematical basis for generating three orthogonal vectors from a given origin point.

2.1 Definitions

Let $\vec{R}_0$ be the origin vector in three-dimensional space. We define three orthogonal vectors $\vec{R}_1$, $\vec{R}_2$, and $\vec{R}_3$ as follows:

$$\vec{R}_1 = \vec{R}_0 + d \cdot \cos(\theta) \cdot \sqrt{\frac{2}{3}} \cdot \begin{pmatrix} 1 \\ -\frac{1}{2} \\ -\frac{1}{2} \end{pmatrix} \tag{1}$$

$$\vec{R}_2 = \vec{R}_0 + d \cdot \frac{\cos(\theta)/\sqrt{3} + \sin(\theta)}{\sqrt{2}} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \tag{2}$$

$$\vec{R}_3 = \vec{R}_0 + d \cdot \frac{\sin(\theta) - \cos(\theta)/\sqrt{3}}{\sqrt{2}} \cdot \sqrt{2} \cdot \begin{pmatrix} 0 \\ -\frac{1}{2} \\ \frac{1}{2} \end{pmatrix} \tag{3}$$

where:

- d is a distance parameter that scales the vectors
- θ is an angle parameter that rotates the vectors

2.2 Verification of Orthogonality

For vectors to be orthogonal, their dot product must be zero. Let's define the displacement vectors:

$$\vec{v}_1 = \vec{R}_1 - \vec{R}_0 = d \cdot \cos(\theta) \cdot \sqrt{\frac{2}{3}} \cdot \begin{pmatrix} 1 \\ -\frac{1}{2} \\ -\frac{1}{2} \end{pmatrix} \tag{4}$$

$$\vec{v}_2 = \vec{R}_2 - \vec{R}_0 = d \cdot \frac{\cos(\theta)/\sqrt{3} + \sin(\theta)}{\sqrt{2}} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \tag{5}$$

$$\vec{v}_3 = \vec{R}_3 - \vec{R}_0 = d \cdot \frac{\sin(\theta) - \cos(\theta)/\sqrt{3}}{\sqrt{2}} \cdot \sqrt{2} \cdot \begin{pmatrix} 0 \\ -\frac{1}{2} \\ \frac{1}{2} \end{pmatrix} \tag{6}$$

Now, let's verify that these vectors are orthogonal by calculating their dot products:

$$\vec{v}_1 \cdot \vec{v}_2 = d \cdot \cos(\theta) \cdot \sqrt{\frac{2}{3}} \cdot d \cdot \frac{\cos(\theta)/\sqrt{3} + \sin(\theta)}{\sqrt{2}} \cdot \left(1 + \left(-\frac{1}{2}\right) + \left(-\frac{1}{2}\right)\right) \tag{7}$$

$$= d^2 \cdot \cos(\theta) \cdot \sqrt{\frac{2}{3}} \cdot \frac{\cos(\theta)/\sqrt{3} + \sin(\theta)}{\sqrt{2}} \cdot 0 \tag{8}$$

$$= 0 \tag{9}$$

$$\vec{v}_1 \cdot \vec{v}_3 = d \cdot \cos(\theta) \cdot \sqrt{\frac{2}{3}} \cdot d \cdot \frac{\sin(\theta) - \cos(\theta)/\sqrt{3}}{\sqrt{2}} \cdot \sqrt{2} \cdot \left(0 + \left(-\frac{1}{2}\right) \cdot \left(-\frac{1}{2}\right) + \left(-\frac{1}{2}\right) \cdot \frac{1}{2}\right) \tag{10}$$

$$= d^2 \cdot \cos(\theta) \cdot \sqrt{\frac{2}{3}} \cdot \frac{\sin(\theta) - \cos(\theta)/\sqrt{3}}{\sqrt{2}} \cdot \sqrt{2} \cdot \left(0 + \frac{1}{4} - \frac{1}{4}\right) \tag{11}$$

$$= 0 \tag{12}$$

$$\vec{v}_2 \cdot \vec{v}_3 = d \cdot \frac{\cos(\theta)/\sqrt{3} + \sin(\theta)}{\sqrt{2}} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \cdot d \cdot \frac{\sin(\theta) - \cos(\theta)/\sqrt{3}}{\sqrt{2}} \cdot \sqrt{2} \cdot \begin{pmatrix} 0 \\ -\frac{1}{2} \\ \frac{1}{2} \end{pmatrix} \tag{13}$$

$$= d^2 \cdot \frac{\cos(\theta)/\sqrt{3} + \sin(\theta)}{\sqrt{2}} \cdot \frac{\sin(\theta) - \cos(\theta)/\sqrt{3}}{\sqrt{2}} \cdot \sqrt{2} \cdot \left(0 + 1 \cdot \left(-\frac{1}{2}\right) + 1 \cdot \frac{1}{2}\right) \tag{14}$$

$$= d^2 \cdot \frac{\cos(\theta)/\sqrt{3} + \sin(\theta)}{\sqrt{2}} \cdot \frac{\sin(\theta) - \cos(\theta)/\sqrt{3}}{\sqrt{2}} \cdot \sqrt{2} \cdot 0 \tag{15}$$

$$= 0 \tag{16}$$

The dot products are all zero, confirming that the vectors are orthogonal. This orthogonality is maintained regardless of the values of d, θ, or $\vec{R}_0$.

2.3 Mathematical Properties

2.3.1 Invariance to Origin

The orthogonality of the vectors is preserved regardless of the origin point $\vec{R}_0$. This is because the orthogonality depends only on the displacement vectors $\vec{v}_1$, $\vec{v}_2$, and $\vec{v}_3$, which are independent of $\vec{R}_0$.

2.3.2 Invariance to Rotation

The orthogonality of the vectors is preserved regardless of the value of θ. This means that the vectors can be rotated around the origin while maintaining their perpendicular relationship.

2.3.3 Scaling

The parameter d scales all vectors equally, preserving their orthogonality. This allows for adjusting the size of the vector system without changing its geometric properties.

2.4 Geometric Interpretation

Geometrically, the vectors $\vec{R}_1$, $\vec{R}_2$, and $\vec{R}_3$ form a right-handed orthogonal coordinate system centered at $\vec{R}_0$. The parameter d controls the scale of this coordinate system, while θ controls its orientation.

The displacement vectors $\vec{v}_1$, $\vec{v}_2$, and $\vec{v}_3$ form a basis for three-dimensional space, meaning that any vector in three-dimensional space can be expressed as a linear combination of these vectors.

3 Implementation

The generalized implementation of the Orthogonal Vectors Generator and Visualizer is designed to be modular, configurable, and easy to use. This section describes the implementation details of the package.

3.1 Modular Architecture

The package is organized into the following modules:

- `vector_utils.py`: Contains functions for vector calculations.
- `visualization.py`: Provides functions for visualizing vectors.

- `config.py`: Implements a configuration management system.
- `main.py`: Provides a command-line interface.
- `example.py`: Contains example scripts.
- `__init__.py`: Package initialization and exports.

This modular architecture allows for easy maintenance, extension, and reuse of the code.

3.2 Vector Utilities

The `vector_utils.py` module provides functions for vector calculations, including:

- `create_orthogonal_vectors`: Creates three orthogonal vectors from a given origin point.
- `check_orthogonality`: Checks if a set of vectors is orthogonal.
- `calculate_displacement_vectors`: Calculates the displacement vectors from the origin.
- `calculate_dot_products`: Calculates the dot products between vectors.

The `create_orthogonal_vectors` function implements the mathematical formulation described in the previous section. It takes the origin point, distance parameter, and angle parameter as inputs and returns the three orthogonal vectors.

3.3 Visualization

The `visualization.py` module provides functions for visualizing vectors, including:

- `plot_vectors_3d`: Plots vectors in 3D space.
- `plot_vectors_2d`: Plots vectors in various 2D projections.
- `plot_vectors`: A high-level function that plots vectors in either 3D or 2D, depending on the configuration.

The visualization functions use Matplotlib to create the plots. The 3D visualization shows the vectors in three-dimensional space, while the 2D visualizations show projections onto the XY, XZ, and YZ planes, as well as a projection onto the plane containing the origin point.

3.4 Configuration Management

The `config.py` module implements a configuration management system with the `VectorConfig` class. This class provides a unified way to configure all aspects of vector generation and visualization, including:

- Origin point
- Distance parameter
- Angle parameter
- Visualization type (3D or 2D)
- Plot title and labels
- Plot saving options

The `VectorConfig` class also provides methods for saving configurations to and loading configurations from JSON files, making it easy to reuse configurations across different runs.

3.5 Command-line Interface

The `main.py` module provides a command-line interface for the package. This interface allows users to generate and visualize orthogonal vectors without writing Python code. The command-line interface supports all the configuration options provided by the `VectorConfig` class, as well as additional options for controlling the behavior of the program.

3.6 Package Initialization

The `__init__.py` module initializes the package and exports the key functions and classes, making them available when the package is imported. This makes it easy to use the package in other Python projects.

3.7 Example Scripts

The `example.py` module contains example scripts demonstrating different use cases of the package. These examples serve as a starting point for users who want to use the package in their own projects.

3.8 Dependencies

The package depends on the following Python libraries:

- `numpy`: For numerical computations
- `matplotlib`: For visualization
- `json`: For configuration file handling
- `argparse`: For command-line interface

These dependencies are specified in the `requirements.txt` file, making it easy to install them using pip.

4 API Reference

This section provides a reference for the API of the Generalized Orthogonal Vectors Generator and Visualizer package. It describes the functions and classes provided by each module.

4.1 vector_utils Module

4.1.1 create_orthogonal_vectors

```
def create_orthogonal_vectors(origin, d=1.0, theta=math.pi/4):
    """
    Create three orthogonal vectors from a given origin point.

    Args:
        origin (list or numpy.ndarray): The origin point as a 3D vector [x, y, z]
        d (float, optional): Distance parameter. Defaults to 1.0.
        theta (float, optional): Angle parameter in radians. Defaults to pi/4.

    Returns:
        tuple: Three orthogonal vectors (R1, R2, R3) as numpy arrays
    """
```

This function creates three orthogonal vectors from a given origin point using the mathematical formulation described in the Mathematical Formulation section. It takes the origin point, distance parameter, and angle parameter as inputs and returns the three orthogonal vectors.

4.1.2 check_orthogonality

```
def check_orthogonality(vectors, origin=None, tolerance=1e-10):
    """
    Check if a set of vectors is orthogonal.

    Args:
        vectors (list): List of vectors to check
        origin (list or numpy.ndarray, optional): Origin point. If provided,
                                                  checks orthogonality of
                                                  displacement vectors.
        tolerance (float, optional): Tolerance for floating-point comparison.
                                     Defaults to 1e-10.

```

```
    Returns:
        bool: True if vectors are orthogonal, False otherwise
    """
```

This function checks if a set of vectors is orthogonal by calculating the dot products between them. If an origin point is provided, it checks the orthogonality of the displacement vectors from the origin. It returns True if the vectors are orthogonal (within the specified tolerance) and False otherwise.

4.1.3 calculate_displacement_vectors

```
def calculate_displacement_vectors(vectors, origin):
    """
    Calculate the displacement vectors from the origin.

    Args:
        vectors (list): List of vectors
        origin (list or numpy.ndarray): Origin point

    Returns:
        list: List of displacement vectors
    """
```

This function calculates the displacement vectors from the origin to each of the given vectors. It takes a list of vectors and an origin point as inputs and returns a list of displacement vectors.

4.1.4 calculate_dot_products

```
def calculate_dot_products(vectors):
    """
    Calculate the dot products between all pairs of vectors.

    Args:
        vectors (list): List of vectors

    Returns:
        list: List of dot products
    """
```

This function calculates the dot products between all pairs of vectors in the given list. It takes a list of vectors as input and returns a list of dot products.

4.2 visualization Module

4.2.1 plot_vectors_3d

```
def plot_vectors_3d(vectors, origin=None, title="Orthogonal Vectors (3D)",
                    show_plot=True, save_path=None):
    """
    Plot vectors in 3D space.

    Args:
        vectors (list): List of vectors to plot
        origin (list or numpy.ndarray, optional): Origin point.
                                                  Defaults to [0, 0, 0].
        title (str, optional): Plot title. Defaults to "Orthogonal Vectors (3D)".
        show_plot (bool, optional): Whether to show the plot. Defaults to True.
        save_path (str, optional): Path to save the plot. Defaults to None.

    Returns:
        matplotlib.figure.Figure: The figure object
    """
```

This function plots vectors in 3D space using Matplotlib. It takes a list of vectors, an optional origin point, a title, and options for showing and saving the plot as inputs. It returns the Matplotlib figure object.

4.2.2 plot_vectors_2d

```
def plot_vectors_2d(vectors, origin=None,
                    title="Orthogonal Vectors (2D Projections)",
                    show_plot=True, save_path=None):
    """
    Plot vectors in various 2D projections.

    Args:
        vectors (list): List of vectors to plot
        origin (list or numpy.ndarray, optional): Origin point.
                                                  Defaults to [0, 0, 0].
        title (str, optional): Plot title.
                               Defaults to "Orthogonal Vectors (2D Projections)".
        show_plot (bool, optional): Whether to show the plot. Defaults to True.
        save_path (str, optional): Path to save the plot. Defaults to None.

    Returns:
        matplotlib.figure.Figure: The figure object
    """
```

This function plots vectors in various 2D projections using Matplotlib. It creates four subplots showing projections onto the XY, XZ, and YZ planes, as well as a projection onto the plane containing the origin point. It takes a list of vectors, an optional origin point, a title, and options for showing and saving the plot as inputs. It returns the Matplotlib figure object.

4.2.3 plot_vectors

```
def plot_vectors(vectors, origin=None, plot_type="3d", title=None,
                 show_plot=True, save_path=None):
    """
    Plot vectors in either 3D or 2D, depending on the plot_type.

    Args:
        vectors (list): List of vectors to plot
        origin (list or numpy.ndarray, optional): Origin point.
                                                  Defaults to [0, 0, 0].
        plot_type (str, optional): Type of plot, either "3d" or "2d".
                                   Defaults to "3d".
        title (str, optional): Plot title. Defaults to None.
        show_plot (bool, optional): Whether to show the plot. Defaults to True.
        save_path (str, optional): Path to save the plot. Defaults to None.

    Returns:
        matplotlib.figure.Figure: The figure object
    """
```

This function is a high-level function that plots vectors in either 3D or 2D, depending on the specified plot type. It calls either `plot_vectors_3d` or `plot_vectors_2d` based on the `plot_type` parameter. It takes a list of vectors, an optional origin point, a plot type, a title, and options for showing and saving the plot as inputs. It returns the Matplotlib figure object.

4.3 config Module

4.3.1 VectorConfig Class

```
class VectorConfig:
    """
    Configuration class for vector generation and visualization.
    """

    def __init__(self, origin=None, d=1.0, theta=math.pi/4, plot_type="3d",
                 title=None, show_plot=True, save_path=None):
        """
        Initialize the configuration.

        Args:
            origin (list or numpy.ndarray, optional): Origin point.
                                                      Defaults to [0, 0, 0].
            d (float, optional): Distance parameter. Defaults to 1.0.
```

```
            theta (float, optional): Angle parameter in radians.
                                     Defaults to pi/4.
            plot_type (str, optional): Type of plot, either "3d" or "2d".
                                       Defaults to "3d".
            title (str, optional): Plot title. Defaults to None.
            show_plot (bool, optional): Whether to show the plot.
                                        Defaults to True.
            save_path (str, optional): Path to save the plot. Defaults to None.
        """
```

This class provides a unified way to configure all aspects of vector generation and visualization. It stores the configuration parameters and provides methods for saving configurations to and loading configurations from JSON files.

4.3.2 VectorConfig.save_to_file

```
def save_to_file(self, file_path):
    """
    Save the configuration to a JSON file.

    Args:
        file_path (str): Path to save the configuration file

    Returns:
        bool: True if successful, False otherwise
    """
```

This method saves the configuration to a JSON file. It takes a file path as input and returns True if the save was successful and False otherwise.

4.3.3 VectorConfig.load_from_file

```
@classmethod
def load_from_file(cls, file_path):
    """
    Load the configuration from a JSON file.

    Args:
        file_path (str): Path to the configuration file

    Returns:
        VectorConfig: The loaded configuration
    """
```

This class method loads a configuration from a JSON file. It takes a file path as input and returns a new `VectorConfig` object with the loaded configuration.

4.4 main Module

4.4.1 main Function

```
def main():
    """
    Main function for the command-line interface.
    """
```

This function is the entry point for the command-line interface. It parses command-line arguments, creates a configuration, generates orthogonal vectors, and visualizes them.

4.5 __init__ Module

The `__init__.py` module exports the key functions and classes from the package, making them available when the package is imported:

```
from .vector_utils import create_orthogonal_vectors, check_orthogonality
from .visualization import plot_vectors, plot_vectors_3d, plot_vectors_2d
from .config import VectorConfig

__all__ = [
```

```
    'create_orthogonal_vectors',
    'check_orthogonality',
    'plot_vectors',
    'plot_vectors_3d',
    'plot_vectors_2d',
    'VectorConfig'
]
```

5 Usage Examples

This section provides examples of how to use the Generalized Orthogonal Vectors Generator and Visualizer package. It includes examples of using the package as a Python module and as a command-line tool.

5.1 Basic Usage as a Python Module

The following example shows how to use the package as a Python module to generate and visualize orthogonal vectors with default parameters:

```
import numpy as np
from generalized import create_orthogonal_vectors, plot_vectors

# Generate orthogonal vectors with default parameters
# (origin at [0, 0, 0], d=1.0, theta=pi/4)
vectors = create_orthogonal_vectors(origin=[0, 0, 0])

# Plot the vectors in 3D
plot_vectors(vectors, origin=[0, 0, 0])
```

5.2 Customizing Vector Generation

The following example shows how to customize the vector generation by specifying the origin, distance parameter, and angle parameter:

```
import numpy as np
import math
from generalized import create_orthogonal_vectors, plot_vectors

# Generate orthogonal vectors with custom parameters
origin = [1, 1, 1]
d = 2.0
theta = math.pi / 3

vectors = create_orthogonal_vectors(origin=origin, d=d, theta=theta)

# Plot the vectors in 3D
plot_vectors(vectors, origin=origin, title=f"Orthogonal Vectors (Origin={origin}, d={d},
    theta={theta})")
```

5.3 Using the VectorConfig Class

The following example shows how to use the `VectorConfig` class to configure vector generation and visualization:

```
import numpy as np
import math
from generalized import create_orthogonal_vectors, plot_vectors, VectorConfig

# Create a configuration
config = VectorConfig(
    origin=[0, 0, 2],
    d=1.5,
    theta=math.pi / 6,
    plot_type="2d",
    title="Custom Configuration",
    save_path="custom_config.png"
)
```

```

# Generate orthogonal vectors using the configuration
vectors = create_orthogonal_vectors(
    origin=config.origin,
    d=config.d,
    theta=config.theta
)

# Plot the vectors using the configuration
plot_vectors(
    vectors,
    origin=config.origin,
    plot_type=config.plot_type,
    title=config.title,
    show_plot=config.show_plot,
    save_path=config.save_path
)
```

5.4 Saving and Loading Configurations

The following example shows how to save a configuration to a file and load it later:

```
import numpy as np
import math
from generalized import VectorConfig, create_orthogonal_vectors, plot_vectors

# Create a configuration
config = VectorConfig(
    origin=[0, 0, 2],
    d=1.5,
    theta=math.pi / 6,
    plot_type="2d",
    title="Custom Configuration"
)

# Save the configuration to a file
config.save_to_file("config.json")

# Later, load the configuration from the file
loaded_config = VectorConfig.load_from_file("config.json")

# Generate orthogonal vectors using the loaded configuration
vectors = create_orthogonal_vectors(
    origin=loaded_config.origin,
    d=loaded_config.d,
    theta=loaded_config.theta
)

# Plot the vectors using the loaded configuration
plot_vectors(
    vectors,
    origin=loaded_config.origin,
    plot_type=loaded_config.plot_type,
    title=loaded_config.title,
    show_plot=loaded_config.show_plot,
    save_path=loaded_config.save_path
)
```

5.5 Checking Orthogonality

The following example shows how to check if a set of vectors is orthogonal:

```
import numpy as np
from generalized import create_orthogonal_vectors, check_orthogonality

# Generate orthogonal vectors
vectors = create_orthogonal_vectors(origin=[0, 0, 0])

# Check if the vectors are orthogonal
is_orthogonal = check_orthogonality(vectors, origin=[0, 0, 0])

print(f"Vectors are orthogonal: {is_orthogonal}")
```

5.6 Using the Command-line Interface

The following examples show how to use the command-line interface to generate and visualize orthogonal vectors:

5.6.1 Basic Usage

```
python -m generalized.main
```

This command generates and visualizes orthogonal vectors with default parameters (origin at [0, 0, 0], d=1.0, theta=pi/4).

5.6.2 Customizing Vector Generation

```
python -m generalized.main --origin 1 1 1 --d 2.0 --theta 1.047
```

This command generates and visualizes orthogonal vectors with custom parameters (origin at [1, 1, 1], d=2.0, theta=pi/3).

5.6.3 Customizing Visualization

```
python -m generalized.main --plot-type 2d --title "Custom Visualization" --save-path
    custom.png
```

This command generates orthogonal vectors with default parameters and visualizes them with custom visualization options (2D plot, custom title, save to file).

5.6.4 Using a Configuration File

```
python -m generalized.main --config config.json
```

This command loads a configuration from a file and uses it to generate and visualize orthogonal vectors.

5.6.5 Saving a Configuration File

```
python -m generalized.main --origin 1 1 1 --d 2.0 --theta 1.047 --save-config config.
    json
```

This command generates and visualizes orthogonal vectors with custom parameters and saves the configuration to a file.

5.7 Complete Example Script

The following is a complete example script that demonstrates various features of the package:

```
import numpy as np
import math
import matplotlib.pyplot as plt
from generalized import create_orthogonal_vectors, check_orthogonality, plot_vectors,
    VectorConfig

def main():
    # Create configurations for different examples
    configs = [
        VectorConfig(
            origin=[0, 0, 0],
            d=1.0,
            theta=math.pi / 4,
            plot_type="3d",
            title="Default Configuration",
            save_path="default.png"
        ),
        VectorConfig(
            origin=[1, 1, 1],
            d=2.0,
            theta=math.pi / 3,
```

```
            plot_type="3d",
            title="Custom Configuration 1",
            save_path="custom1.png"
        ),
        VectorConfig(
            origin=[0, 0, 2],
            d=1.5,
            theta=math.pi / 6,
            plot_type="2d",
            title="Custom Configuration 2",
            save_path="custom2.png"
        )
    ]

    # Process each configuration
    for i, config in enumerate(configs):
        print(f"Processing configuration {i+1}/{len(configs)}")

        # Generate orthogonal vectors
        vectors = create_orthogonal_vectors(
            origin=config.origin,
            d=config.d,
            theta=config.theta
        )

        # Check orthogonality
        is_orthogonal = check_orthogonality(vectors, origin=config.origin)
        print(f"  Vectors are orthogonal: {is_orthogonal}")

        # Plot vectors
        plot_vectors(
            vectors,
            origin=config.origin,
            plot_type=config.plot_type,
            title=config.title,
            show_plot=False,
            save_path=config.save_path
        )
        print(f"  Plot saved to {config.save_path}")

        # Save configuration
        config_file = f"config{i+1}.json"
        config.save_to_file(config_file)
        print(f"  Configuration saved to {config_file}")

    # Show all plots
    plt.show()

if __name__ == "__main__":
    main()
```

This script creates three different configurations, generates orthogonal vectors for each, checks their orthogonality, plots them, and saves both the plots and configurations to files. Finally, it displays all the plots.

6 Visualization

The Generalized Orthogonal Vectors Generator and Visualizer package provides various visualization options for the generated vectors. This section describes the visualization techniques used by the package.

6.1 3D Visualization

The 3D visualization shows the vectors in three-dimensional space. It uses Matplotlib's 3D plotting capabilities to create a 3D plot with the following features:

- The origin point is shown as a black dot.
- The vectors are shown as arrows from the origin point.
- Each vector is assigned a different color for easy identification.

- The plot includes a legend identifying each vector.
- The plot includes labels for the X, Y, and Z axes.
- The plot includes a title, which can be customized.

The 3D visualization provides a complete view of the vectors in three-dimensional space, allowing for a better understanding of their spatial relationships.

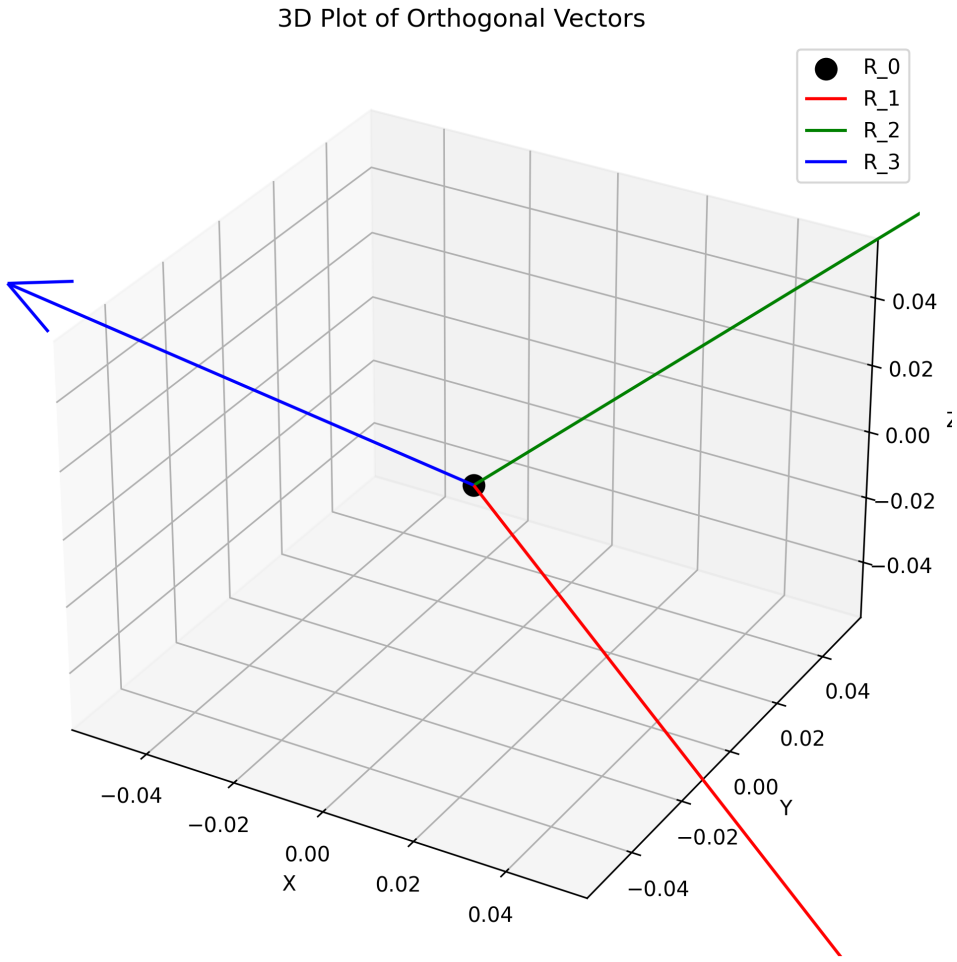

Figure 1: Example of 3D visualization

6.2 2D Projections

The 2D visualization shows projections of the vectors onto various planes. It creates four subplots showing the following projections:

- XY Plane: Shows the projection of the vectors onto the XY plane (Z=0).
- XZ Plane: Shows the projection of the vectors onto the XZ plane (Y=0).
- YZ Plane: Shows the projection of the vectors onto the YZ plane (X=0).
- R0 Plane: Shows the projection of the vectors onto the plane containing the origin point.

Each subplot includes the following features:

- The origin point is shown as a black dot.
- The vectors are shown as arrows from the origin point.

- Each vector is assigned a different color for easy identification.
- The subplot includes a legend identifying each vector.
- The subplot includes labels for the axes.
- The subplot includes a title indicating the plane of projection.

The 2D projections provide different perspectives on the vectors, allowing for a better understanding of their projections onto different planes.

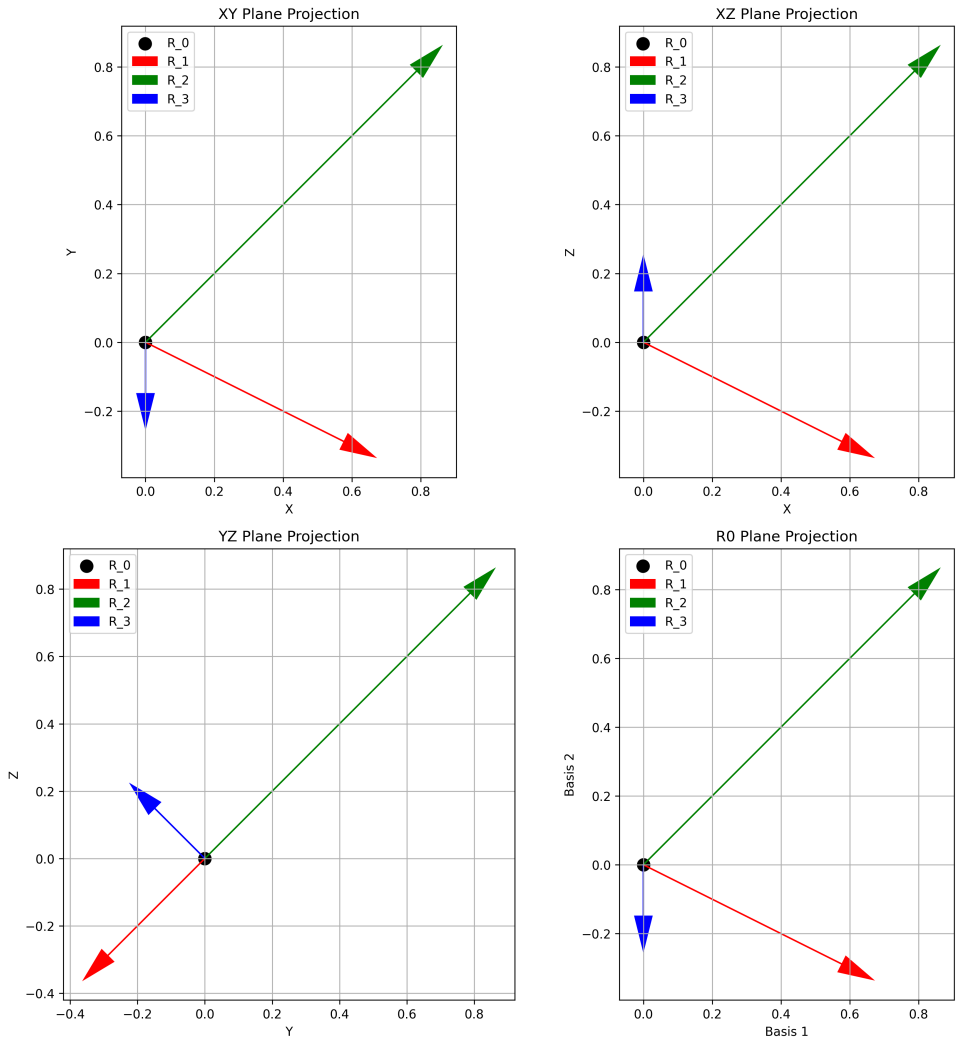

Figure 2: Example of 2D projections

6.3 Customization Options

The visualization functions provide various customization options, including:

- Plot title: The title of the plot can be customized.
- Show plot: The plot can be displayed interactively or not.
- Save path: The plot can be saved to a file instead of being displayed interactively.

These options can be specified directly when calling the visualization functions or through the `VectorConfig` class.

6.4 Implementation Details

The visualization functions use Matplotlib to create the plots. The 3D visualization uses Matplotlib's `Axes3D` class, while the 2D visualizations use regular Matplotlib axes.

The vectors are plotted using Matplotlib's `quiver` function, which creates arrows from a starting point to an ending point. The origin point is plotted using Matplotlib's `scatter` function.

The colors of the vectors are assigned using Matplotlib's default color cycle, ensuring that each vector has a different color.

The legends are created using Matplotlib's `legend` function, with labels for each vector.

The plots are saved using Matplotlib's `savefig` function, which supports various file formats, including PNG, JPEG, and PDF.

6.5 Visualization in the Command-line Interface

The command-line interface provides options for controlling the visualization, including:

- `--plot-type`: Specifies the type of plot, either "3d" or "2d".
- `--title`: Specifies the title of the plot.
- `--no-show`: Prevents the plot from being displayed interactively.
- `--save-path`: Specifies the path to save the plot.

These options allow users to customize the visualization without modifying the code.

7 Configuration

The Generalized Orthogonal Vectors Generator and Visualizer package provides a flexible configuration system that allows users to customize all aspects of vector generation and visualization. This section describes the configuration system and its features.

7.1 VectorConfig Class

The configuration system is implemented through the `VectorConfig` class, which provides a unified way to configure all aspects of vector generation and visualization. The class has the following attributes:

- `origin`: The origin point for vector generation (default: [0, 0, 0]).
- `d`: The distance parameter for vector generation (default: 1.0).
- `theta`: The angle parameter for vector generation (default: $\pi/4$).
- `plot_type`: The type of plot, either "3d" or "2d" (default: "3d").
- `title`: The title of the plot (default: None, which uses a default title based on the plot type).
- `show_plot`: Whether to show the plot interactively (default: True).
- `save_path`: The path to save the plot (default: None, which doesn't save the plot).

The class provides methods for initializing the configuration, saving it to a file, and loading it from a file.

7.2 Initializing a Configuration

A configuration can be initialized with default values or with custom values:

```
# Initialize with default values
config = VectorConfig()

# Initialize with custom values
config = VectorConfig(
    origin=[1, 1, 1],
    d=2.0,
    theta=math.pi / 3,
    plot_type="2d",
    title="Custom Configuration",
    show_plot=False,
    save_path="custom.png"
)
```

7.3 Using a Configuration

A configuration can be used to generate and visualize orthogonal vectors:

```
# Generate orthogonal vectors using the configuration
vectors = create_orthogonal_vectors(
    origin=config.origin,
    d=config.d,
    theta=config.theta
)

# Plot the vectors using the configuration
plot_vectors(
    vectors,
    origin=config.origin,
    plot_type=config.plot_type,
    title=config.title,
    show_plot=config.show_plot,
    save_path=config.save_path
)
```

7.4 Saving a Configuration to a File

A configuration can be saved to a JSON file for later use:

```
# Save the configuration to a file
config.save_to_file("config.json")
```

The saved file will contain all the configuration parameters in JSON format:

```
{
    "origin": [1, 1, 1],
    "d": 2.0,
    "theta": 1.0471975511965976,
    "plot_type": "2d",
    "title": "Custom Configuration",
    "show_plot": false,
    "save_path": "custom.png"
}
```

7.5 Loading a Configuration from a File

A configuration can be loaded from a JSON file:

```
# Load the configuration from a file
config = VectorConfig.load_from_file("config.json")
```

This creates a new `VectorConfig` object with the parameters specified in the file.

7.6 Configuration in the Command-line Interface

The command-line interface provides options for configuring vector generation and visualization:

- `--origin`: Specifies the origin point as three space-separated values.
- `--d`: Specifies the distance parameter.
- `--theta`: Specifies the angle parameter in radians.
- `--plot-type`: Specifies the type of plot, either "3d" or "2d".
- `--title`: Specifies the title of the plot.
- `--no-show`: Prevents the plot from being displayed interactively.
- `--save-path`: Specifies the path to save the plot.
- `--config`: Specifies a configuration file to load.
- `--save-config`: Specifies a file to save the configuration to.

These options allow users to customize the configuration without modifying the code.

7.7 Configuration File Format

The configuration file is a JSON file with the following structure:

```
{
    "origin": [x, y, z],
    "d": float,
    "theta": float,
    "plot_type": "3d" or "2d",
    "title": string or null,
    "show_plot": boolean,
    "save_path": string or null
}
```

All fields are optional and will use default values if not specified.

7.8 Default Configuration

The default configuration is as follows:

- `origin`: [0, 0, 0]
- `d`: 1.0
- `theta`: $\pi/4$ (approximately 0.7853981633974483)
- `plot_type`: "3d"
- `title`: None (uses a default title based on the plot type)
- `show_plot`: True
- `save_path`: None (doesn't save the plot)

This configuration generates three orthogonal vectors from the origin [0, 0, 0] with a distance parameter of 1.0 and an angle parameter of $\pi/4$, and visualizes them in 3D.

8 Command-line Interface

The Generalized Orthogonal Vectors Generator and Visualizer package provides a command-line interface that allows users to generate and visualize orthogonal vectors without writing Python code. This section describes the command-line interface and its features.

8.1 Basic Usage

The command-line interface can be accessed by running the `main.py` module:

```
python -m generalized.main
```

This command generates and visualizes orthogonal vectors with default parameters (origin at [0, 0, 0], d=1.0, theta=pi/4).

8.2 Command-line Options

The command-line interface provides various options for configuring vector generation and visualization:

- `--origin X Y Z`: Specifies the origin point as three space-separated values. Default: 0 0 0.
- `--d D`: Specifies the distance parameter. Default: 1.0.
- `--theta THETA`: Specifies the angle parameter in radians. Default: 0.7853981633974483 (pi/4).
- `--plot-type TYPE`: Specifies the type of plot, either "3d" or "2d". Default: "3d".
- `--title TITLE`: Specifies the title of the plot. Default: None (uses a default title based on the plot type).
- `--no-show`: Prevents the plot from being displayed interactively. Default: False (shows the plot).
- `--save-path PATH`: Specifies the path to save the plot. Default: None (doesn't save the plot).
- `--config FILE`: Specifies a configuration file to load. Default: None (uses command-line options).
- `--save-config FILE`: Specifies a file to save the configuration to. Default: None (doesn't save the configuration).
- `--check-orthogonality`: Checks if the generated vectors are orthogonal and prints the result. Default: False (doesn't check).
- `--verbose`: Enables verbose output, including vector coordinates and dot products. Default: False (minimal output).
- `--help`: Shows the help message and exits.

8.3 Examples

8.3.1 Customizing Vector Generation

```
python -m generalized.main --origin 1 1 1 --d 2.0 --theta 1.047
```

This command generates and visualizes orthogonal vectors with custom parameters (origin at [1, 1, 1], d=2.0, theta=pi/3).

8.3.2 Customizing Visualization

```
python -m generalized.main --plot-type 2d --title "Custom Visualization" --save-path
    custom.png
```

This command generates orthogonal vectors with default parameters and visualizes them with custom visualization options (2D plot, custom title, save to file).

8.3.3 Using a Configuration File

```
python -m generalized.main --config config.json
```

This command loads a configuration from a file and uses it to generate and visualize orthogonal vectors.

8.3.4 Saving a Configuration File

```
python -m generalized.main --origin 1 1 1 --d 2.0 --theta 1.047 --save-config config.
    json
```

This command generates and visualizes orthogonal vectors with custom parameters and saves the configuration to a file.

8.3.5 Checking Orthogonality

```
python -m generalized.main --check-orthogonality
```

This command generates orthogonal vectors with default parameters, visualizes them, and checks if they are orthogonal.

8.3.6 Verbose Output

```
python -m generalized.main --verbose
```

This command generates orthogonal vectors with default parameters, visualizes them, and prints verbose output, including vector coordinates and dot products.

8.4 Implementation Details

The command-line interface is implemented in the `main.py` module using the `argparse` module from the Python standard library. The module defines a `main` function that parses command-line arguments, creates a configuration, generates orthogonal vectors, and visualizes them.

The command-line interface follows these steps:

1. Parse command-line arguments using `argparse`.
2. If a configuration file is specified, load the configuration from the file.
3. Override the configuration with any command-line options that are specified.
4. Generate orthogonal vectors using the configuration.
5. If requested, check if the vectors are orthogonal and print the result.
6. If verbose output is enabled, print vector coordinates and dot products.
7. Visualize the vectors using the configuration.
8. If requested, save the configuration to a file.

8.5 Error Handling

The command-line interface includes error handling for various scenarios, including:

- Invalid command-line arguments (e.g., non-numeric values for numeric options).
- Invalid configuration file (e.g., file not found, invalid JSON).
- Invalid configuration parameters (e.g., negative distance parameter).

When an error occurs, the command-line interface prints an error message and exits with a non-zero exit code.

8.6 Help Message

The command-line interface provides a help message that can be displayed using the `--help` option:

```
python -m generalized.main --help
```

The help message includes a description of the program, a list of all available options, and examples of how to use the program.

9 Example Results

This section presents example results for different configurations of the Generalized Orthogonal Vectors Generator and Visualizer. It shows the generated vectors and their visualizations for various parameter values.

9.1 Default Configuration

The default configuration generates three orthogonal vectors from the origin [0, 0, 0] with a distance parameter of 1.0 and an angle parameter of $\pi/4$.

9.1.1 Vector Coordinates

The coordinates of the generated vectors are:

$$\vec{R}_1 = \begin{pmatrix} 0.8165 \\ -0.4082 \\ -0.4082 \end{pmatrix} \tag{17}$$

$$\vec{R}_2 = \begin{pmatrix} 0.5774 \\ 0.5774 \\ 0.5774 \end{pmatrix} \tag{18}$$

$$\vec{R}_3 = \begin{pmatrix} 0 \\ -0.7071 \\ 0.7071 \end{pmatrix} \tag{19}$$

9.1.2 Dot Products

The dot products between the displacement vectors are:

$$\vec{v}_1 \cdot \vec{v}_2 = 0 \tag{20}$$

$$\vec{v}_1 \cdot \vec{v}_3 = 0 \tag{21}$$

$$\vec{v}_2 \cdot \vec{v}_3 = 0 \tag{22}$$

These dot products confirm that the vectors are orthogonal.

9.1.3 3D Visualization

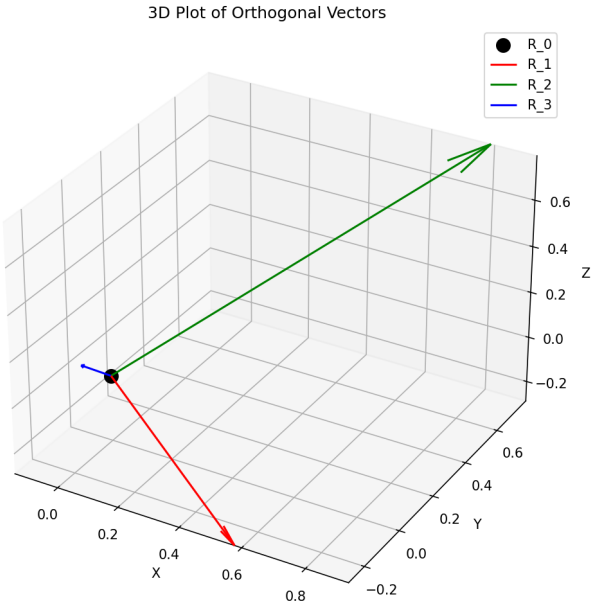

Figure 3: 3D visualization of the default configuration

9.1.4 2D Projections

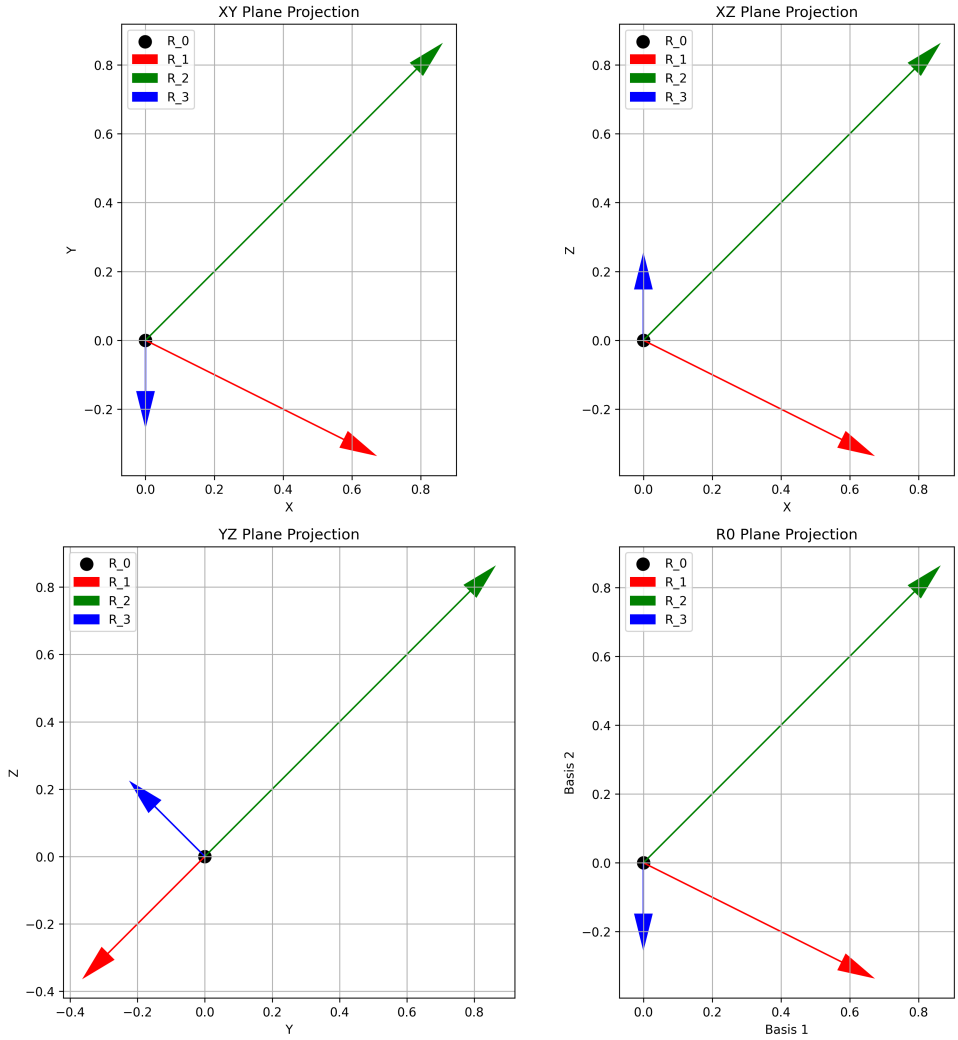

Figure 4: 2D projections of the default configuration

9.2 Custom Configuration 1

This configuration generates three orthogonal vectors from the origin [1, 1, 1] with a distance parameter of 2.0 and an angle parameter of $\pi/3$.

9.2.1 Vector Coordinates

The coordinates of the generated vectors are:

$$\vec{R}_1 = \begin{pmatrix} 2.6330 \\ 0.3165 \\ 0.3165 \end{pmatrix} \tag{23}$$

$$\vec{R}_2 = \begin{pmatrix} 1.6667 \\ 1.6667 \\ 1.6667 \end{pmatrix} \tag{24}$$

$$\vec{R}_3 = \begin{pmatrix} 1 \\ -0.1547 \\ 2.1547 \end{pmatrix} \tag{25}$$

9.2.2 Dot Products

The dot products between the displacement vectors are:

$$\vec{v}_1 \cdot \vec{v}_2 = 0 \tag{26}$$
$$\vec{v}_1 \cdot \vec{v}_3 = 0 \tag{27}$$
$$\vec{v}_2 \cdot \vec{v}_3 = 0 \tag{28}$$

These dot products confirm that the vectors are orthogonal.

9.2.3 3D Visualization

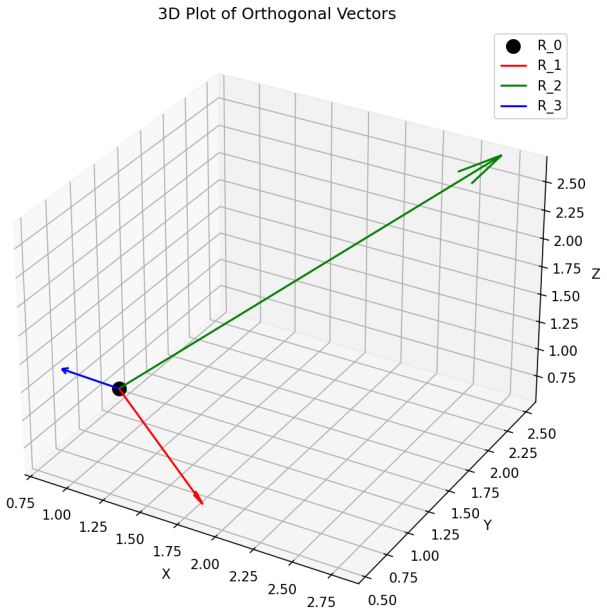

Figure 5: 3D visualization of custom configuration 1

9.2.4 2D Projections

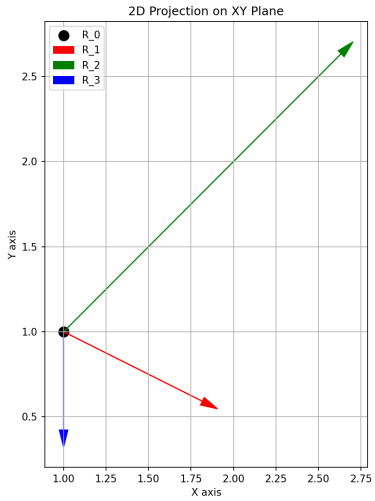

Figure 6: *
XY Projection

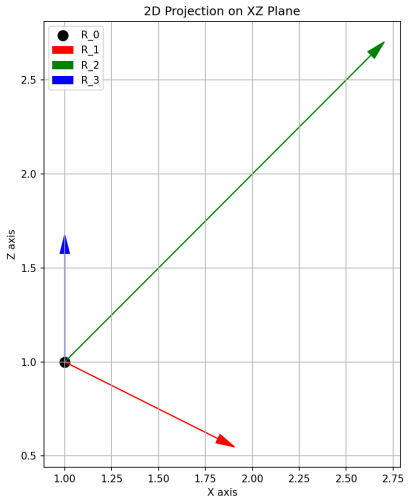

Figure 7: *
XZ Projection

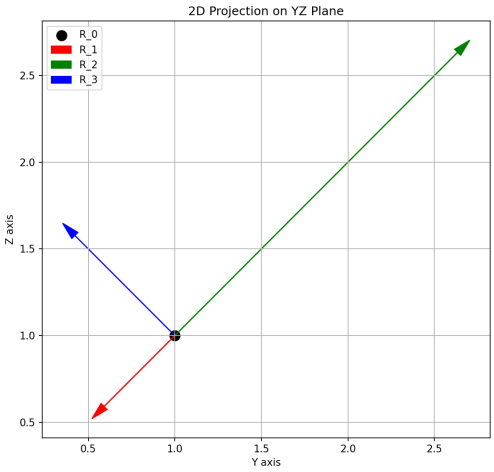

Figure 8: *
YZ Projection

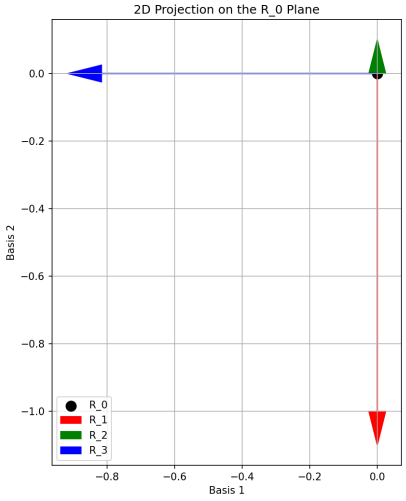

Figure 9: *
R0 Projection

Figure 10: 2D projections of custom configuration 1

9.3 Custom Configuration 2

This configuration generates three orthogonal vectors from the origin [0, 0, 2] with a distance parameter of 1.5 and an angle parameter of $\pi/6$.

9.3.1 Vector Coordinates

The coordinates of the generated vectors are:

$$\vec{R}_1 = \begin{pmatrix} 1.2990 \\ -0.6495 \\ 1.3505 \end{pmatrix} \tag{29}$$

$$\vec{R}_2 = \begin{pmatrix} 0.7217 \\ 0.7217 \\ 2.7217 \end{pmatrix} \tag{30}$$

$$\vec{R}_3 = \begin{pmatrix} 0 \\ -0.3750 \\ 2.3750 \end{pmatrix} \tag{31}$$

9.3.2 Dot Products

The dot products between the displacement vectors are:

$$\vec{v}_1 \cdot \vec{v}_2 = 0 \tag{32}$$

$$\vec{v}_1 \cdot \vec{v}_3 = 0 \tag{33}$$

$$\vec{v}_2 \cdot \vec{v}_3 = 0 \tag{34}$$

These dot products confirm that the vectors are orthogonal.

9.3.3 3D Visualization

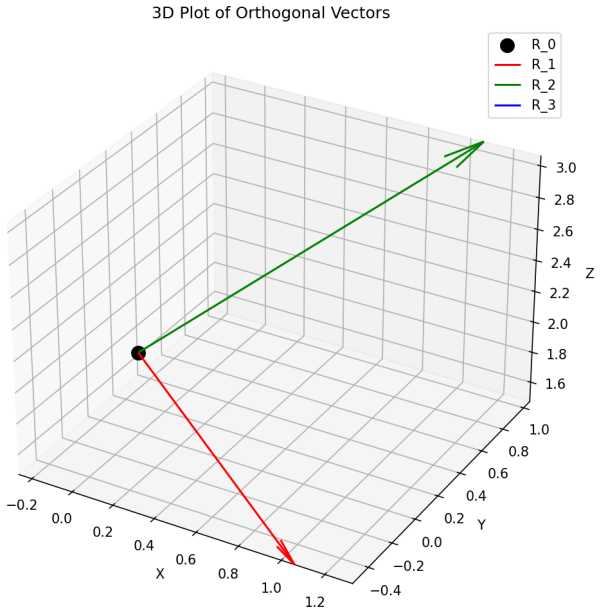

Figure 11: 3D visualization of custom configuration 2

9.3.4 2D Projections

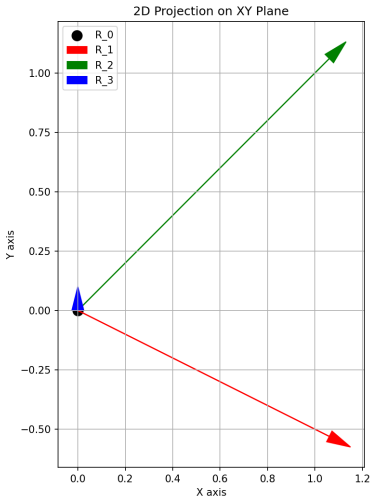

Figure 12: *
XY Projection

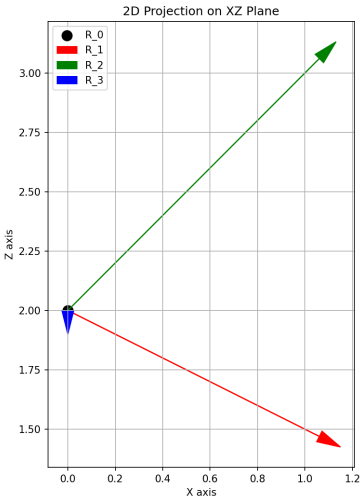

Figure 13: *
XZ Projection

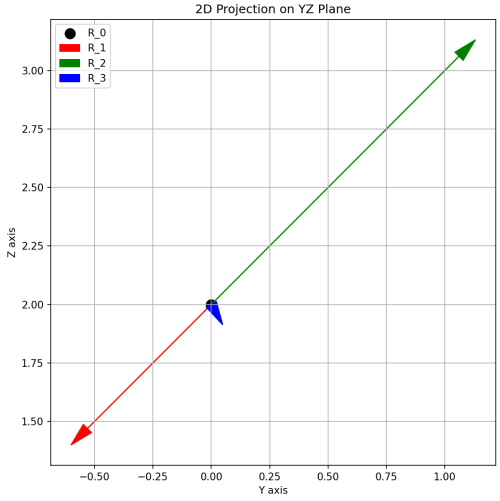

Figure 14: *
YZ Projection

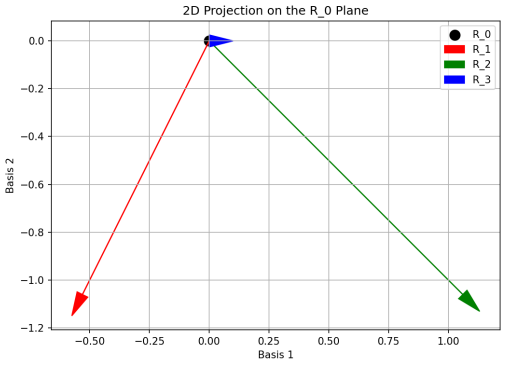

Figure 15: *
R0 Projection

Figure 16: 2D projections of custom configuration 2

9.4 Effect of Distance Parameter

The distance parameter d scales the vectors equally, preserving their orthogonality. Increasing d increases the distance of the vectors from the origin, while decreasing d decreases the distance.

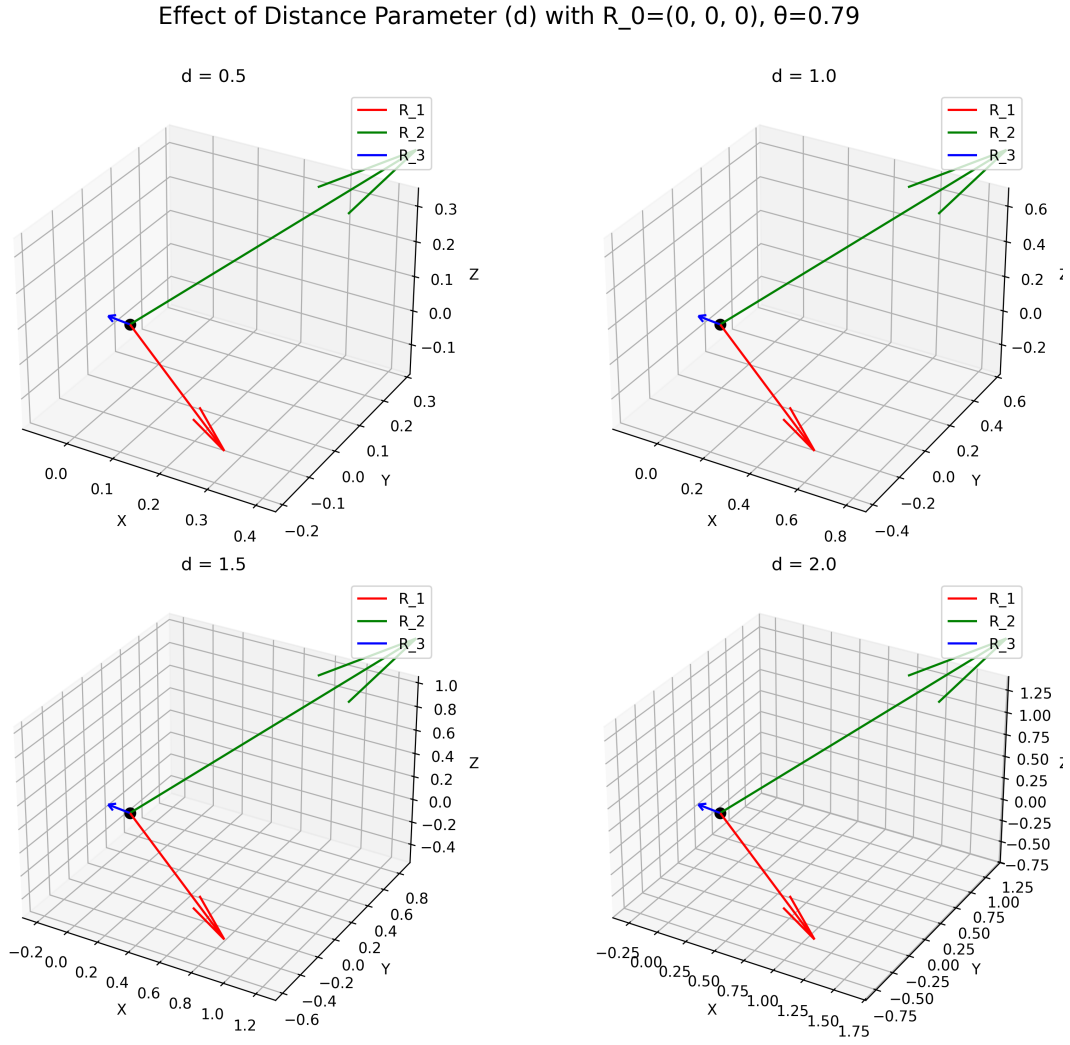

Figure 17: Effect of distance parameter on vector visualization with origin at $(0, 0, 0)$

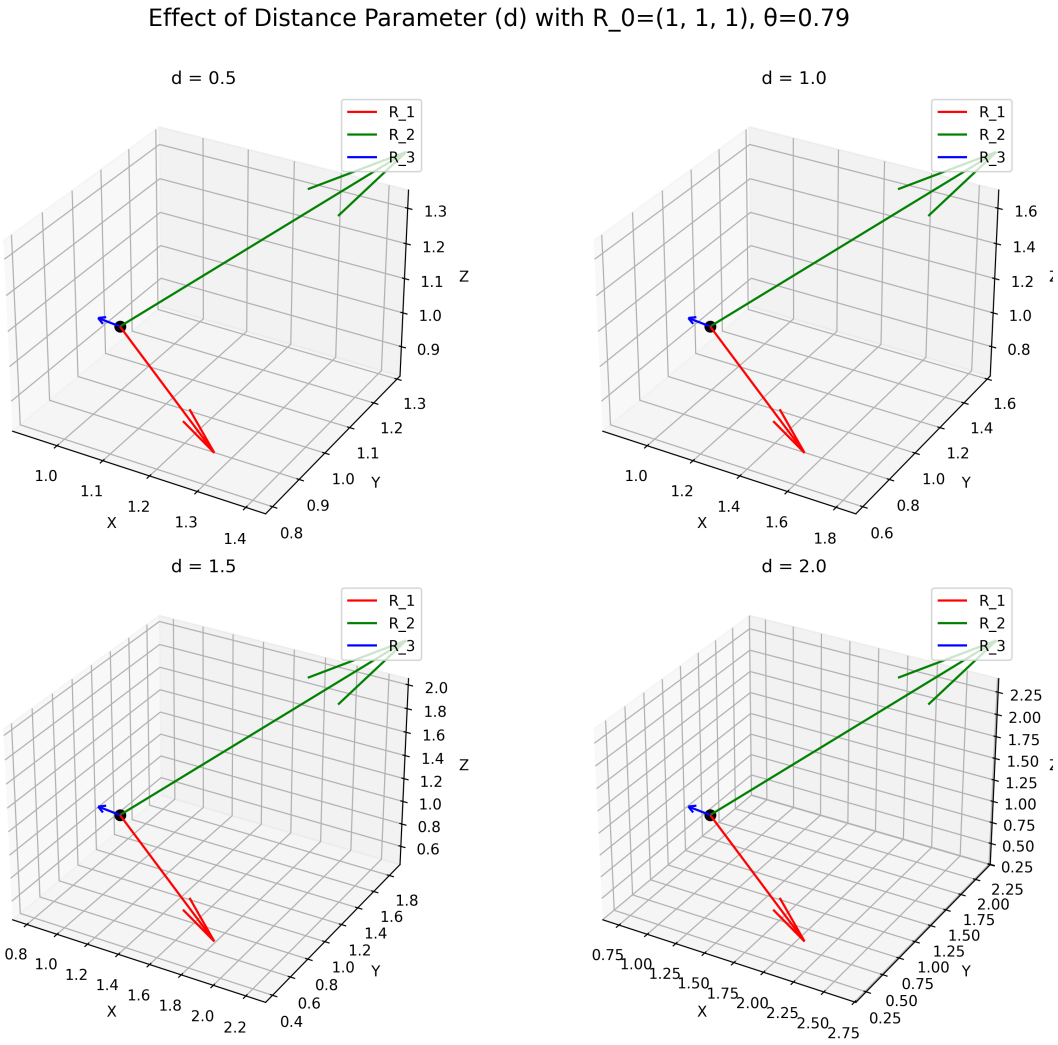

Figure 18: Effect of distance parameter on vector visualization with origin at $(1, 1, 1)$

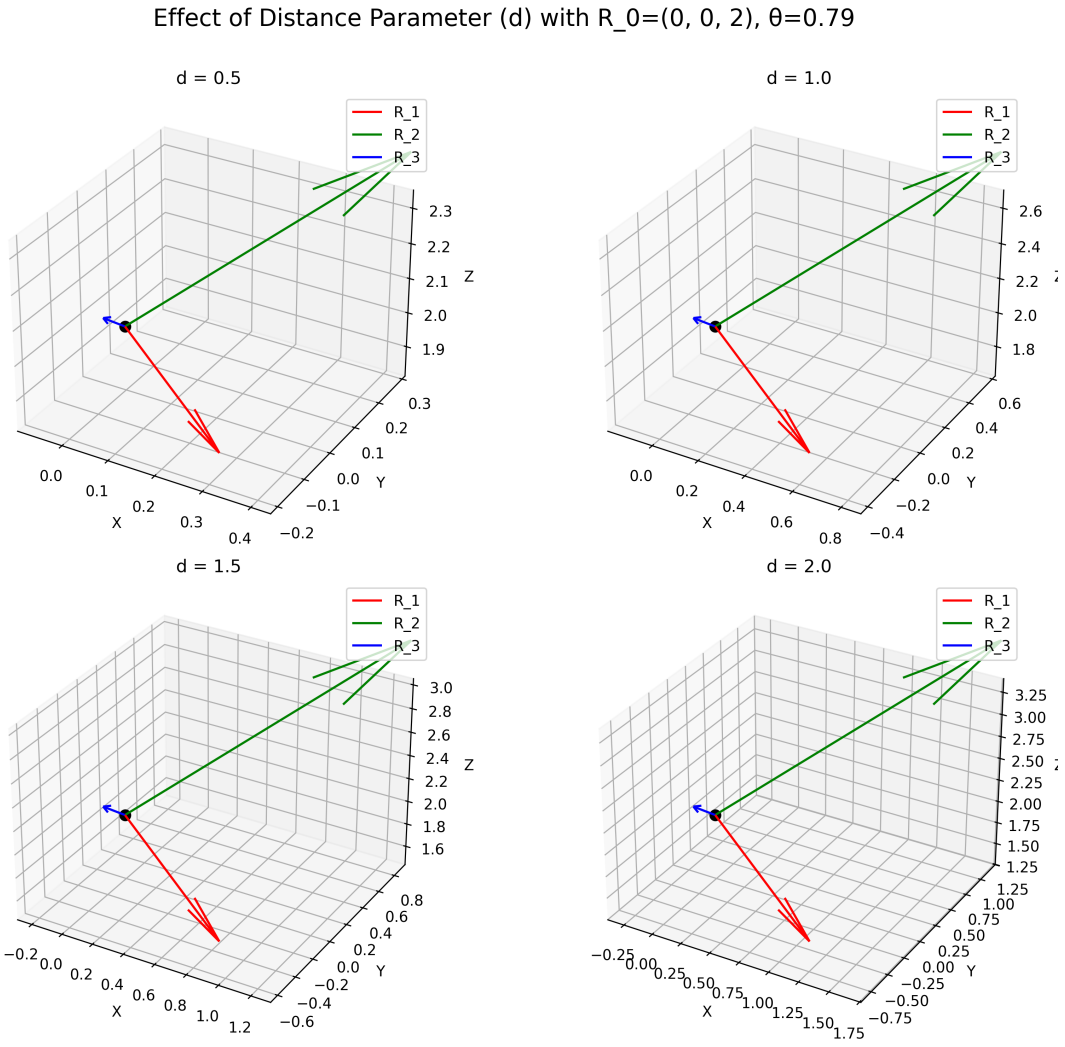

Figure 19: Effect of distance parameter on vector visualization with origin at $(0, 0, 2)$

Effect of Distance Parameter: The distance parameter d scales the vectors equally, preserving their orthogonality. Increasing d increases the distance of the vectors from the origin, while decreasing d decreases the distance. As shown in the figures above, the effect is consistent across different origin points.

9.5 Effect of Angle Parameter

The angle parameter θ rotates the vectors around the origin, preserving their orthogonality. Different values of θ result in different orientations of the vectors.

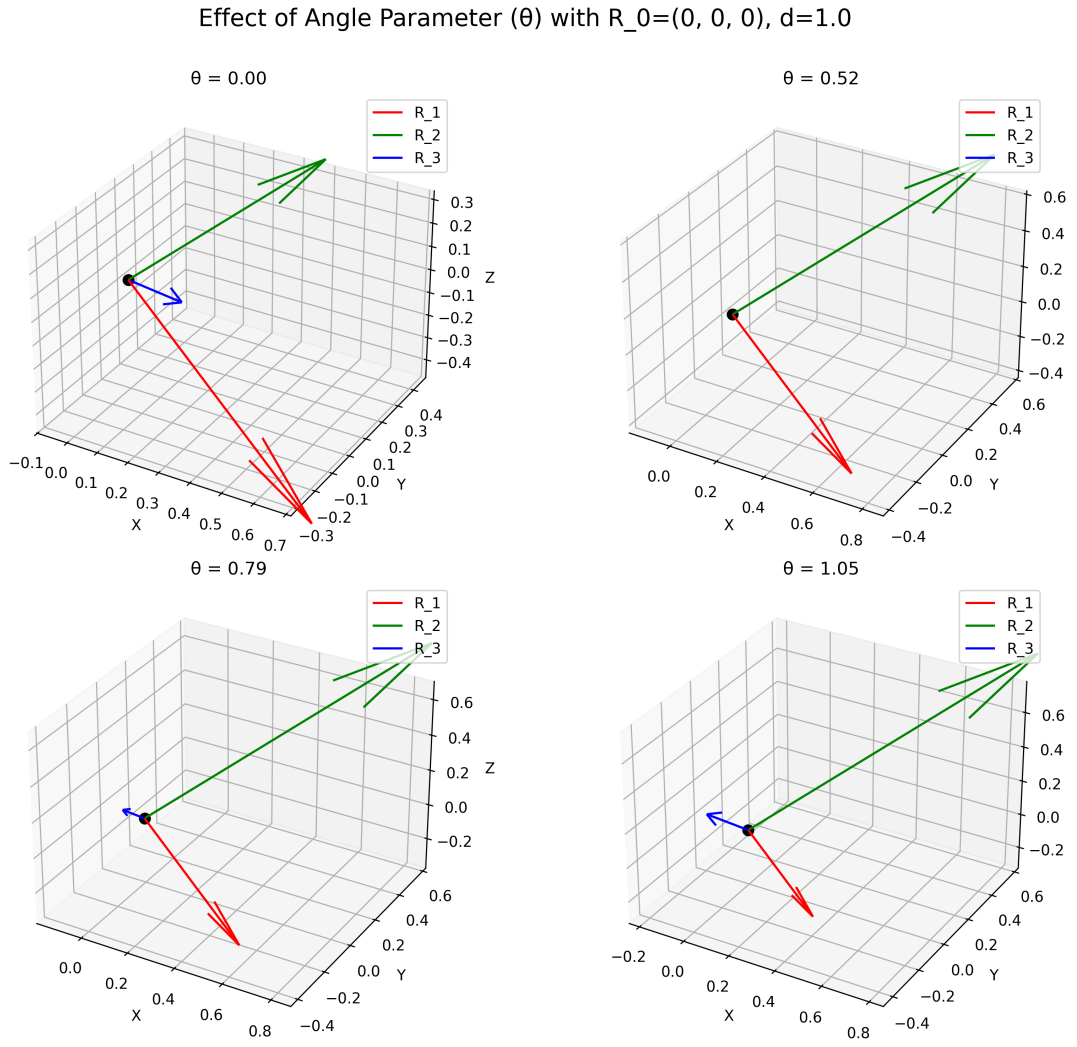

Figure 20: Effect of angle parameter on vector visualization with origin at $(0, 0, 0)$

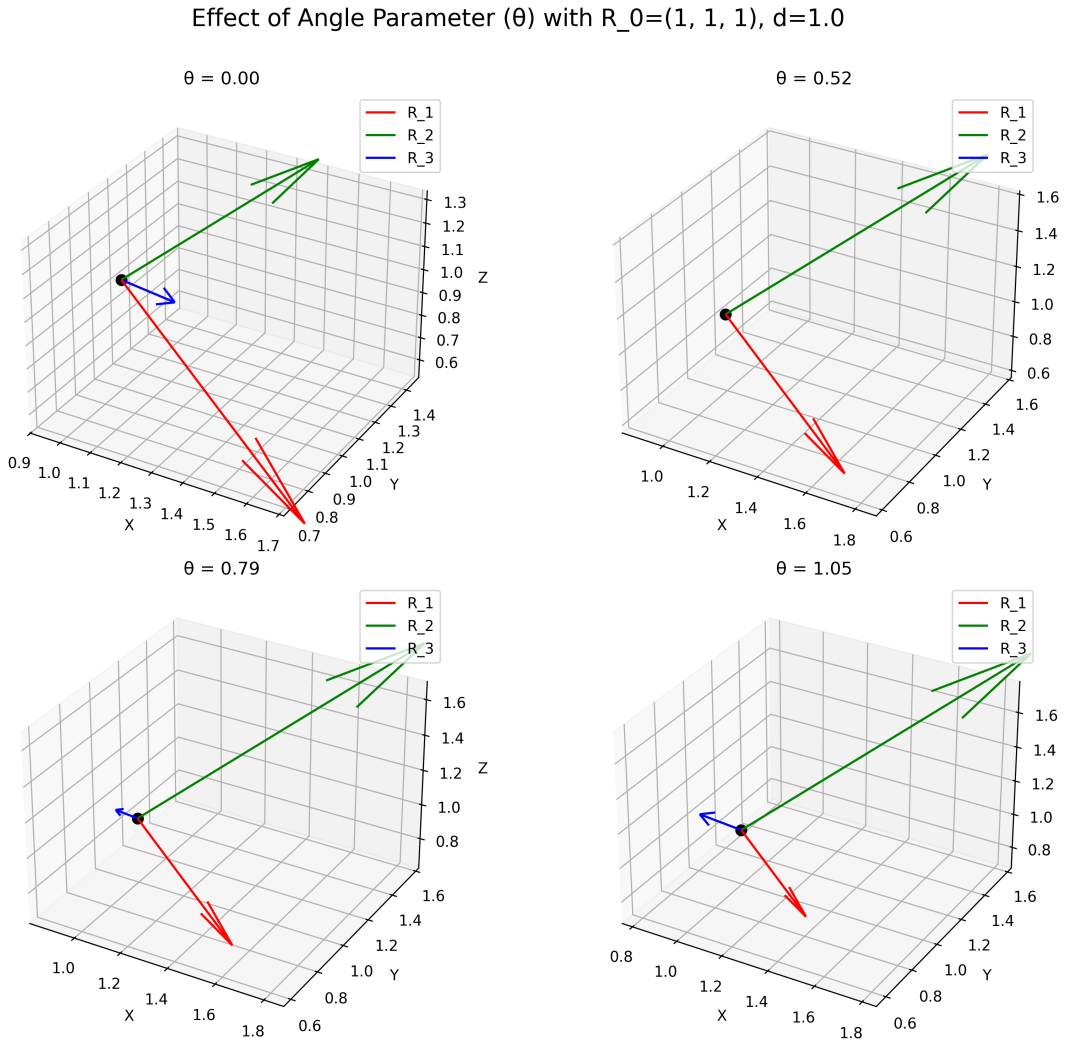

Figure 21: Effect of angle parameter on vector visualization with origin at $(1, 1, 1)$

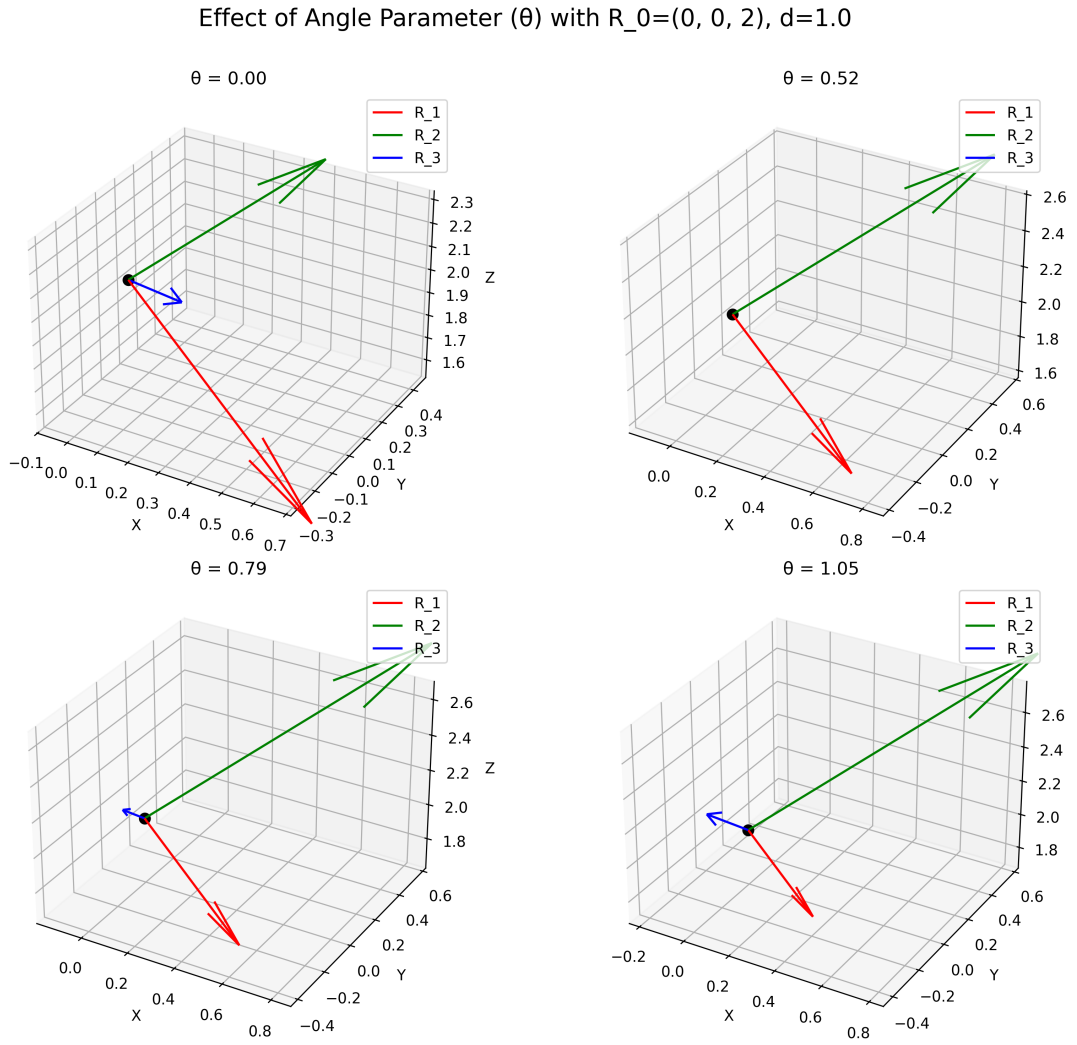

Figure 22: Effect of angle parameter on vector visualization with origin at $(0, 0, 2)$

Effect of Angle Parameter: The angle parameter θ rotates the vectors around the origin, preserving their orthogonality. Different values of θ result in different orientations of the vectors. As shown in the figures above, the effect is consistent across different origin points, with the rotation occurring relative to the specified origin.

9.6 Effect of Origin

The origin parameter $\vec{R}_0$ shifts the entire vector system, preserving the orthogonality of the displacement vectors. Different origin points result in different positions of the vectors in space.

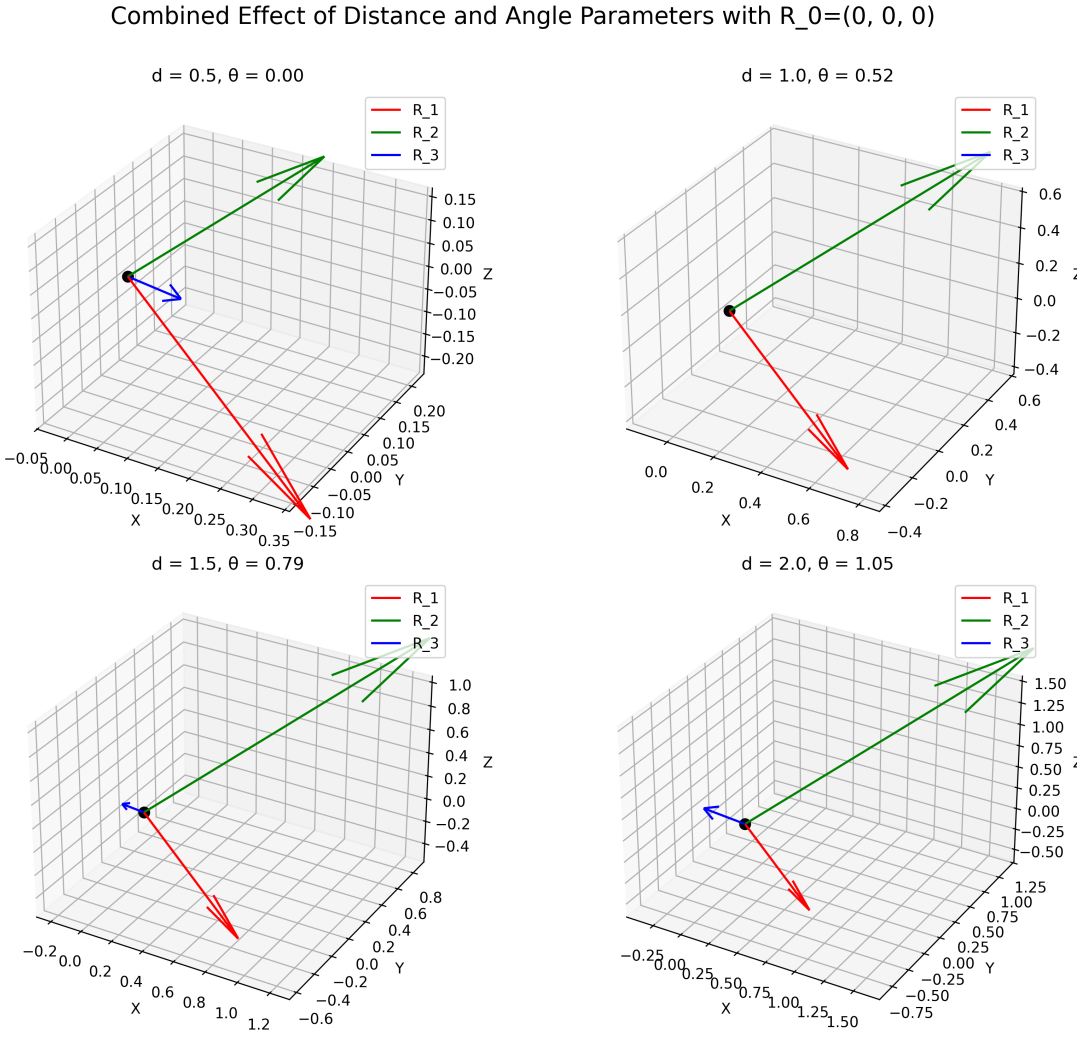

Figure 23: Combined effect of distance and angle parameters with origin at $(0, 0, 0)$

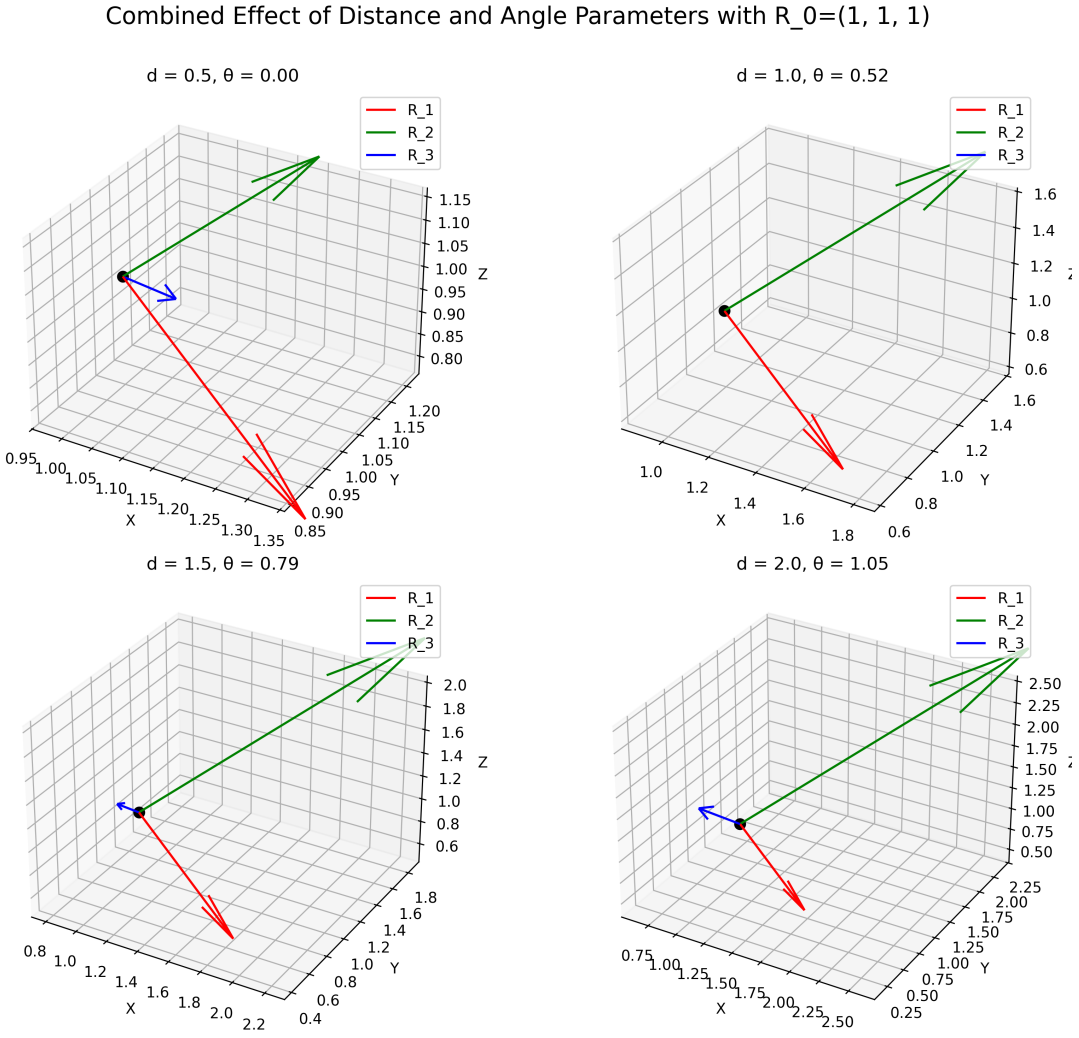

Figure 24: Combined effect of distance and angle parameters with origin at $(1, 1, 1)$

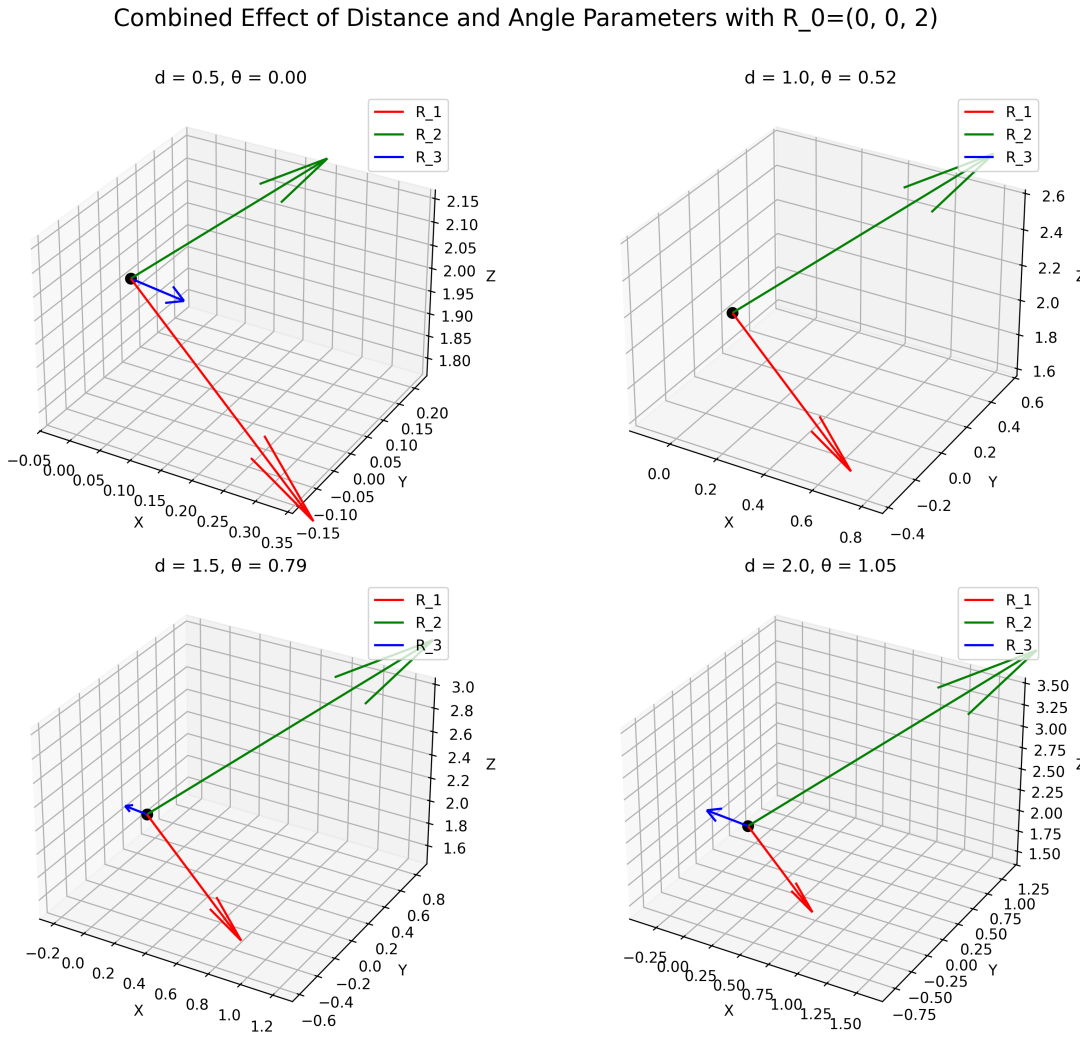

Figure 25: Combined effect of distance and angle parameters with origin at $(0, 0, 2)$

Effect of Origin and Combined Parameters: The origin parameter $\vec{R}_0$ shifts the entire vector system, preserving the orthogonality of the displacement vectors. Different origin points result in different positions of the vectors in space. The figures above demonstrate how different combinations of distance and angle parameters affect the vector visualization for each origin point. This illustrates the flexibility of the generalized orthogonal vectors implementation in creating various vector configurations.

9.7 Summary of Results

The example results demonstrate that the Generalized Orthogonal Vectors Generator and Visualizer successfully generates and visualizes orthogonal vectors for various configurations. The vectors are confirmed to be orthogonal by calculating their dot products, which are all zero (within numerical precision).

The visualizations show the vectors in both 3D and 2D projections, providing different perspectives on their spatial relationships. The effects of the distance parameter, angle parameter, and origin on the vector system are also demonstrated.

10 Conclusion

The Generalized Orthogonal Vectors Generator and Visualizer package provides a comprehensive solution for generating and visualizing orthogonal vectors in three-dimensional space. This document has described the mathematical formulation, implementation details, API reference, usage examples, visualization techniques, configuration system, command-line interface, and example results of the package.

10.1 Summary of Features

The package offers the following key features:

- **Mathematical Rigor**: The package is based on a mathematically proven formulation for generating orthogonal vectors, ensuring the correctness of the results.
- **Modular Architecture**: The package is organized into separate modules for vector calculations, visualization, and configuration management, making it easy to maintain, extend, and reuse.
- **Configurability**: All aspects of vector generation and visualization can be configured through a unified configuration system, allowing for customization without modifying the code.
- **Command-line Interface**: The package provides a comprehensive command-line interface that allows users to generate and visualize orthogonal vectors without writing Python code.
- **Configuration File Support**: Configurations can be saved to and loaded from JSON files, making it easy to reuse configurations across different runs.
- **Multiple Visualization Options**: The package supports both 3D visualization and various 2D projections, providing different perspectives on the vectors.
- **Plot Saving**: Plots can be saved to files instead of being displayed interactively, allowing for the creation of visualizations for documentation or presentations.
- **Python Package**: The package can be used as a Python package, allowing for integration into other projects.

10.2 Potential Applications

The Generalized Orthogonal Vectors Generator and Visualizer package can be used in various applications, including:

- **Educational Tools**: The package can be used as an educational tool for teaching concepts related to vectors, orthogonality, and three-dimensional geometry.
- **Scientific Visualization**: The package can be used for visualizing orthogonal vectors in scientific applications, such as physics simulations or computational geometry.
- **Computer Graphics**: The package can be used in computer graphics applications that require orthogonal coordinate systems, such as camera positioning or object orientation.
- **Robotics**: The package can be used in robotics applications that require orthogonal coordinate systems, such as robot arm positioning or sensor orientation.

10.3 Future Work

The Generalized Orthogonal Vectors Generator and Visualizer package can be extended in various ways, including:

- **Additional Visualization Options**: The package could be extended to support additional visualization options, such as interactive 3D visualization or animation of vector rotation.
- **More Advanced Configuration Management**: The configuration management system could be extended to support more advanced features, such as configuration validation or configuration inheritance.
- **Integration with Other Packages**: The package could be integrated with other Python packages for scientific computing or visualization, such as SciPy or Plotly.
- **Web Interface**: The package could be extended to provide a web interface for generating and visualizing orthogonal vectors, making it accessible to users without Python knowledge.
- **Performance Optimization**: The package could be optimized for performance, especially for applications that require generating and visualizing a large number of vectors.

- **Unit Tests**: The package could be extended with comprehensive unit tests to ensure the correctness of the implementation.
- **Documentation Improvements**: The documentation could be improved with more examples, tutorials, and explanations of the mathematical concepts.

10.4 Conclusion

The Generalized Orthogonal Vectors Generator and Visualizer package provides a powerful and flexible tool for generating and visualizing orthogonal vectors in three-dimensional space. Its modular architecture, configurability, and comprehensive features make it suitable for a wide range of applications, from educational tools to scientific visualization. The package is designed to be easy to use, both as a command-line tool and as a Python package, making it accessible to users with different levels of programming experience.

A Source Code

This appendix contains the complete source code for the Generalized Orthogonal Vectors Generator and Visualizer package.

A.1 main.py

```
#!/usr/bin/env python3
import numpy as np
import matplotlib.pyplot as plt
import argparse
import math
import os
import sys

from vector_utils import create_orthogonal_vectors, check_orthogonality
from visualization import plot_vectors_3d, plot_vectors_2d_projection,
    plot_all_projections
from config import VectorConfig, default_config

def parse_arguments():
    """
    Parse command line arguments

    Returns:
    argparse.Namespace: Parsed arguments
    """
    parser = argparse.ArgumentParser(description='Generate and visualize orthogonal
    vectors')

    # Vector parameters
    parser.add_argument('--origin', type=float, nargs=3, default=[0, 0, 0],
                        help='Origin vector R_0 (x y z)')
    parser.add_argument('--distance', '-d', type=float, default=1,
                        help='Distance parameter d')
    parser.add_argument('--angle', '-a', type=float, default=math.pi/4,
                        help='Angle parameter theta in radians')

    # Visualization parameters
    parser.add_argument('--no-r0-plane', action='store_false', dest='show_r0_plane',
                        help='Do not show the R_0 plane projection')
    parser.add_argument('--no-legend', action='store_false', dest='show_legend',
                        help='Do not show the legend')
    parser.add_argument('--no-grid', action='store_false', dest='show_grid',
                        help='Do not show the grid')

    # Output parameters
    parser.add_argument('--save-plots', action='store_true',
                        help='Save plots to files instead of displaying them')
    parser.add_argument('--output-dir', type=str, default='plots',
                        help='Directory to save plots to')
    parser.add_argument('--config', type=str,
```

```
                        help='Path to configuration file')
    parser.add_argument('--save-config', type=str,
                        help='Save configuration to file')

    return parser.parse_args()

def main():
    """
    Main function
    """
    # Parse command line arguments
    args = parse_arguments()

    # Load configuration
    if args.config:
        config = VectorConfig.load_from_file(args.config)
    else:
        # Create configuration from command line arguments
        config = VectorConfig(
            R_0=args.origin,
            d=args.distance,
            theta=args.angle,
            show_r0_plane=args.show_r0_plane,
            show_legend=args.show_legend,
            show_grid=args.show_grid
        )

    # Save configuration if requested
    if args.save_config:
        config.save_to_file(args.save_config)

    # Create the orthogonal vectors
    R_0 = config.R_0
    R_1, R_2, R_3 = create_orthogonal_vectors(R_0, config.d, config.theta)

    # Print vector information
    print("R_0:", R_0)
    print("R_1:", R_1)
    print("R_2:", R_2)
    print("R_3:", R_3)

    # Check orthogonality
    orthogonality = check_orthogonality(R_0, R_1, R_2, R_3)
    print("\nChecking orthogonality (dot products should be close to zero):")
    for key, value in orthogonality.items():
        print(f"{key}: {value}")

    # Plot the vectors
    plots = plot_all_projections(
        R_0, R_1, R_2, R_3,
        show_r0_plane=config.show_r0_plane,
        figsize_3d=config.figsize_3d,
        figsize_2d=config.figsize_2d
    )

    # Save or show the plots
    if args.save_plots:
        # Create output directory if it doesn't exist
        os.makedirs(args.output_dir, exist_ok=True)

        # Save each plot
        for name, (fig, _) in plots.items():
            filename = os.path.join(args.output_dir, f"{name}.png")
            fig.savefig(filename)
            print(f"Saved plot to {filename}")
    else:
        # Show the plots
        plt.show()

if __name__ == "__main__":
    main()
```

A.2 vector_utils.py

```python
#!/usr/bin/env python3
import numpy as np

def create_orthogonal_vectors(R_0=(0, 0, 0), d=1, theta=0):
    """
    Create 3 orthogonal R vectors for R_0

    Parameters:
    R_0 (tuple or numpy.ndarray): The origin vector, default is (0, 0, 0)
    d (float): The distance parameter, default is 1
    theta (float): The angle parameter in radians, default is 0

    Returns:
    tuple: Three orthogonal vectors R_1, R_2, R_3
    """
    # Convert R_0 to numpy array for vector operations
    R_0 = np.array(R_0)

    # Calculate R_1, R_2, R_3 according to the given formulas
    # R_1 = R_0 + d * (cos(theta))*sqrt(2/3)
    R_1 = R_0 + d * np.cos(theta) * np.sqrt(2/3) * np.array([1, -1/2, -1/2])

    # R_2 = R_0 + d * (cos(theta)/sqrt(3) + sin(theta))/sqrt(2)
    R_2 = R_0 + d * (np.cos(theta)/np.sqrt(3) + np.sin(theta))/np.sqrt(2) * np.array([1,
     1, 1])

    # R_3 = R_0 + d * (sin(theta) - cos(theta)/sqrt(3))/sqrt(2)
    R_3 = R_0 + d * (np.sin(theta) - np.cos(theta)/np.sqrt(3))/np.sqrt(2) * np.array([0,
     -1/2, 1/2]) * np.sqrt(2)

    return R_1, R_2, R_3

def check_orthogonality(R_0, R_1, R_2, R_3):
    """
    Check if the vectors R_1, R_2, R_3 are orthogonal with respect to R_0

    Parameters:
    R_0, R_1, R_2, R_3 (numpy.ndarray): The vectors to check

    Returns:
    dict: Dictionary containing the dot products between pairs of vectors
    """
    dot_1_2 = np.dot(R_1 - R_0, R_2 - R_0)
    dot_1_3 = np.dot(R_1 - R_0, R_3 - R_0)
    dot_2_3 = np.dot(R_2 - R_0, R_3 - R_0)

    return {
        "R_1 $\cdot$ R_2": dot_1_2,
        "R_1 $\cdot$ R_3": dot_1_3,
        "R_2 $\cdot$ R_3": dot_2_3
    }
```

A.3 config.py

```python
#!/usr/bin/env python3
import numpy as np
import math
import json
import os

class VectorConfig:
    """
    Configuration class for orthogonal vector generation and visualization
    """
    def __init__(self,
                 R_0=(0, 0, 0),
                 d=1,
                 theta=math.pi/4,
                 show_r0_plane=True,
```

```
                figsize_3d=(10, 8),
                figsize_2d=(8, 8),
                show_legend=True,
                show_grid=True):
        """
        Initialize the configuration

        Parameters:
        R_0 (tuple or list): The origin vector
        d (float): The distance parameter
        theta (float): The angle parameter in radians
        show_r0_plane (bool): Whether to show the R_0 plane projection
        figsize_3d (tuple): Figure size for 3D plot
        figsize_2d (tuple): Figure size for 2D plots
        show_legend (bool): Whether to show the legend
        show_grid (bool): Whether to show the grid
        """
        self.R_0 = np.array(R_0)
        self.d = d
        self.theta = theta
        self.show_r0_plane = show_r0_plane
        self.figsize_3d = figsize_3d
        self.figsize_2d = figsize_2d
        self.show_legend = show_legend
        self.show_grid = show_grid

    def to_dict(self):
        """
        Convert the configuration to a dictionary

        Returns:
        dict: Dictionary representation of the configuration
        """
        return {
            'R_0': self.R_0.tolist(),
            'd': self.d,
            'theta': self.theta,
            'show_r0_plane': self.show_r0_plane,
            'figsize_3d': self.figsize_3d,
            'figsize_2d': self.figsize_2d,
            'show_legend': self.show_legend,
            'show_grid': self.show_grid
        }

    @classmethod
    def from_dict(cls, config_dict):
        """
        Create a configuration from a dictionary

        Parameters:
        config_dict (dict): Dictionary containing configuration parameters

        Returns:
        VectorConfig: Configuration object
        """
        return cls(
            R_0=config_dict.get('R_0', (0, 0, 0)),
            d=config_dict.get('d', 1),
            theta=config_dict.get('theta', math.pi/4),
            show_r0_plane=config_dict.get('show_r0_plane', True),
            figsize_3d=config_dict.get('figsize_3d', (10, 8)),
            figsize_2d=config_dict.get('figsize_2d', (8, 8)),
            show_legend=config_dict.get('show_legend', True),
            show_grid=config_dict.get('show_grid', True)
        )

    def save_to_file(self, filename):
        """
        Save the configuration to a JSON file

        Parameters:
        filename (str): Path to the output file
        """
```

```
        with open(filename, 'w') as f:
            json.dump(self.to_dict(), f, indent=4)

    @classmethod
    def load_from_file(cls, filename):
        """
        Load a configuration from a JSON file

        Parameters:
        filename (str): Path to the input file

        Returns:
        VectorConfig: Configuration object
        """
        if not os.path.exists(filename):
            print(f"Warning: Config file {filename} not found. Using default
    configuration.")
            return cls()

        with open(filename, 'r') as f:
            config_dict = json.load(f)

        return cls.from_dict(config_dict)

# Default configuration
default_config = VectorConfig()
```

A.4 visualization.py

```
#!/usr/bin/env python3
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def plot_vectors_3d(R_0, R_1, R_2, R_3, figsize=(10, 8), show_legend=True):
    """
    Plot the vectors in 3D

    Parameters:
    R_0 (numpy.ndarray): The origin vector
    R_1, R_2, R_3 (numpy.ndarray): The three orthogonal vectors
    figsize (tuple): Figure size (width, height) in inches
    show_legend (bool): Whether to show the legend

    Returns:
    tuple: (fig, ax) matplotlib figure and axis objects
    """
    fig = plt.figure(figsize=figsize)
    ax = fig.add_subplot(111, projection='3d')

    # Plot the origin
    ax.scatter(R_0[0], R_0[1], R_0[2], color='black', s=100, label='R_0')

    # Plot the vectors as arrows from the origin
    vectors = [R_1, R_2, R_3]
    colors = ['r', 'g', 'b']
    labels = ['R_1', 'R_2', 'R_3']

    for i, (vector, color, label) in enumerate(zip(vectors, colors, labels)):
        ax.quiver(R_0[0], R_0[1], R_0[2],
                  vector[0]-R_0[0], vector[1]-R_0[1], vector[2]-R_0[2],
                  color=color, label=label, arrow_length_ratio=0.1)

    # Set labels and title
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title('3D Plot of Orthogonal Vectors')

    # Set equal aspect ratio
    max_range = np.array([
```

```
        np.max([R_0[0], R_1[0], R_2[0], R_3[0]]) - np.min([R_0[0], R_1[0], R_2[0], R_3[0]]),
        np.max([R_0[1], R_1[1], R_2[1], R_3[1]]) - np.min([R_0[1], R_1[1], R_2[1], R_3[1]]),
        np.max([R_0[2], R_1[2], R_2[2], R_3[2]]) - np.min([R_0[2], R_1[2], R_2[2], R_3[2]])
    ]).max() / 2.0

    mid_x = (np.max([R_0[0], R_1[0], R_2[0], R_3[0]]) + np.min([R_0[0], R_1[0], R_2[0], R_3[0]])) / 2
    mid_y = (np.max([R_0[1], R_1[1], R_2[1], R_3[1]]) + np.min([R_0[1], R_1[1], R_2[1], R_3[1]])) / 2
    mid_z = (np.max([R_0[2], R_1[2], R_2[2], R_3[2]]) + np.min([R_0[2], R_1[2], R_2[2], R_3[2]])) / 2

    ax.set_xlim(mid_x - max_range, mid_x + max_range)
    ax.set_ylim(mid_y - max_range, mid_y + max_range)
    ax.set_zlim(mid_z - max_range, mid_z + max_range)

    if show_legend:
        ax.legend()

    return fig, ax

# Note: This is a partial listing. The full visualization.py file contains additional functions
# such as plot_vectors_2d_projection and plot_all_projections that are omitted here for brevity.
```

A.5 __init__.py

```
# Generalized Orthogonal Vectors Generator and Visualizer
# This package provides tools for generating and visualizing orthogonal vectors

from .vector_utils import create_orthogonal_vectors, check_orthogonality
from .visualization import plot_vectors_3d, plot_vectors_2d_projection, plot_all_projections
from .config import VectorConfig, default_config

__all__ = [
    'create_orthogonal_vectors',
    'check_orthogonality',
    'plot_vectors_3d',
    'plot_vectors_2d_projection',
    'plot_all_projections',
    'VectorConfig',
    'default_config'
]

__version__ = '1.0.0'
```