Technical Analysis: Scatter Plot Visualization Script (sc_np_plt.py)

Executive Summary

This document provides a comprehensive technical analysis of the scatter plot visualization script. The script generates randomized scatter plots from CSV data containing hexadecimal values, offering both GUI and CLI interfaces. While functional, the code exhibits several areas requiring improvement in terms of error handling, code organization, mathematical operations, and overall robustness.

Code Structure Overview

Main Components

1. **File Selection** (`select_file`): GUI-based file dialog
2. **Data Processing** (`read_and_process_csv`): CSV parsing and hex extraction
3. **Visualization** (`create_scatter_plot`): Random scatter plot generation
4. **Output** (`save_plot`): SVG export functionality
5. **CLI Interface** (`main`): Command-line argument handling

Strengths

1. User Interface Flexibility

- Dual interface support (GUI dialogs and CLI)
- Interactive file selection with fallback to command-line arguments
- User-friendly error messages

2. Data Handling

- Supports multiple pandas versions with compatibility handling
- Robust CSV reading with error line skipping
- Hex value extraction from structured data

3. Visualization Features

- Wide variety of matplotlib markers (25 types)
- Random style selection from available themes
- Dynamic color, size, and transparency randomization
- SVG output for scalable graphics

4. Command-Line Interface

- Well-structured argparse implementation
- Sensible default values

- Batch processing capability with `--no-show` option

Weaknesses

1. Mathematical Operations

- **Problematic random number generation**:

```
def rand_num():
    number = np.abs(np.random.normal(0, 1))
    if number > 1:
        number = number - np.floor(number)
    return number
```

Issues:

  - Inefficient approach to generating [0,1] random numbers
  - Logic doesn't ensure uniform distribution
  - Should use `np.random.random()` directly

- **Unclear mathematical formulas**:

```
dfs["x"] = np.random.randn(len_df) * np.exp(np.pi**2)
dfs["y"] = np.random.rand(len_df) ** np.log(np.pi)
```

  - Arbitrary mathematical operations without clear purpose
  - May produce extreme values affecting plot scaling

2. Error Handling

- **Division by zero risk**:

```
denominator = np.abs(rand_divisor / np.random.normal(-1, 1))
```

  - `np.random.normal(-1, 1)` can be very close to zero
  - Protection `if denominator < 1` is insufficient

- **Generic exception handling**:

```
except Exception as e:
    print(f"Error reading/processing CSV: {e}")
```

  - Catches all exceptions indiscriminately
  - Makes debugging difficult

3. Input Validation

- No validation of CSV structure
- Assumes second column always contains hex values
- No handling for missing or malformed hex values
- Hard-coded column indices without flexibility

4. Code Organization

- Plotting logic scattered across multiple functions
- Inconsistent variable naming (dfs, len_df)
- Magic numbers without explanation (np.exp(np.pi**2))
- Style configuration mixed with plotting logic

5. Resource Management

- Multiple matplotlib figure instances without proper cleanup
- Tkinter roots created repeatedly
- No context managers for file operations

6. Documentation

- Minimal inline comments
- No explanation for mathematical transformations
- Missing parameter type hints
- No docstring examples

Areas for Improvement

1. Mathematical Operations

```
# Current problematic code
def rand_num():
    number = np.abs(np.random.normal(0, 1))
    if number > 1:
        number = number - np.floor(number)
    return number

# Improved version
def rand_num():
    """Generate a random number between 0 and 1"""
    return np.random.random()
```

2. Error Handling

```
# Improved CSV reading with specific exception handling
def read_and_process_csv(file_path):
    """Read CSV file and process hex values"""
    try:
        df = pd.read_csv(file_path, header=None)
        if df.shape[1] < 2:
            raise ValueError("CSV must have at least 2 columns")

        # Validate hex format
        hex_pattern = r'0x[0-9A-Fa-f]+'
```

```
        df['hex_values'] = df.iloc[:, 1].str.extract(f'({hex_pattern})')

        if df['hex_values'].isna().all():
            raise ValueError("No valid hex values found")

        df['value'] = df['hex_values'].apply(lambda x: int(x, 16))
        return df.dropna(subset=['value'])

    except FileNotFoundError:
        print(f"File not found: {file_path}")
        sys.exit(1)
    except pd.errors.ParserError as e:
        print(f"CSV parsing error: {e}")
        sys.exit(1)
    except ValueError as e:
        print(f"Data validation error: {e}")
        sys.exit(1)
```

3. Input Validation

```
def validate_input_file(file_path):
    """Validate input file exists and is readable"""
    if not os.path.exists(file_path):
        raise FileNotFoundError(f"File not found: {file_path}")
    if not os.access(file_path, os.R_OK):
        raise PermissionError(f"Cannot read file: {file_path}")
    if os.path.getsize(file_path) == 0:
        raise ValueError("File is empty")
```

4. Code Refactoring

```
class ScatterPlotGenerator:
    """Encapsulates scatter plot generation logic"""

    def __init__(self, data, config=None):
        self.data = data
        self.config = config or self.default_config()

    def default_config(self):
        return {
            'sample_size': 750,
            'figure_size': (15, 10),
            'markers': ['o', 's', '^', 'v', 'd', '*'],
            'styles': ['dark_background', 'ggplot', 'fivethirtyeight']
        }
```

```
    def generate_coordinates(self):
        """Generate random x,y coordinates with configurable distributions"""
        n = len(self.data)
        self.data['x'] = np.random.normal(0, 2, n)
        self.data['y'] = np.random.uniform(0, 1, n)

    def calculate_sizes(self):
        """Calculate point sizes based on data values"""
        values = self.data['value'].values
        normalized = (values - values.min()) / (values.max() - values.min())
        self.data['size'] = 10 + normalized * 1000  # Scale between 10-1010
```

5. Configuration Management

```
import configparser

def load_config(config_file='config.ini'):
    """Load configuration from file"""
    config = configparser.ConfigParser()
    config.read(config_file)

    return {
        'plot': {
            'sample_size': config.getint('plot', 'sample_size', fallback=750),
            'figure_size': eval(config.get('plot', 'figure_size', fallback='(15, 10)')),
            'output_format': config.get('plot', 'output_format', fallback='svg')
        },
        'data': {
            'hex_column': config.getint('data', 'hex_column', fallback=1),
            'skip_bad_lines': config.getboolean('data', 'skip_bad_lines', fallback=True)
        }
    }
```

6. Testing

```
import unittest

class TestScatterPlot(unittest.TestCase):
    def test_hex_extraction(self):
        """Test hex value extraction from strings"""
        test_data = pd.DataFrame({
            0: ['id1', 'id2'],
            1: ['data 0x1A2B3C', 'info 0x4D5E6F']
        })

        result = extract_hex_values(test_data)
```

```
        self.assertEqual(result[0], 0x1A2B3C)
        self.assertEqual(result[1], 0x4D5E6F)

    def test_coordinate_generation(self):
        """Test random coordinate generation"""
        data = pd.DataFrame({'value': [100, 200, 300]})
        generator = ScatterPlotGenerator(data)
        generator.generate_coordinates()

        self.assertTrue(all(data['x'].notna()))
        self.assertTrue(all(data['y'].notna()))
```

Recommended Improvements

1. Type Hints and Documentation

```
from typing import Optional, Tuple, Dict
import pandas as pd

def create_scatter_plot(
    df: pd.DataFrame,
    sample_size: int = 750,
    figure_size: Tuple[int, int] = (15, 10),
    config: Optional[Dict] = None
) -> plt.Figure:
    """
    Create a random scatter plot from dataframe.

    Args:
        df: DataFrame with 'value' column containing numeric data
        sample_size: Number of points to plot
        figure_size: Figure dimensions (width, height)
        config: Optional configuration dictionary

    Returns:
        matplotlib Figure object

    Example:
        >>> df = pd.DataFrame({'value': [100, 200, 300]})
        >>> fig = create_scatter_plot(df, sample_size=100)
    """
```

2. Logging Instead of Print

```
import logging

logging.basicConfig(level=logging.INFO)
```

```python
logger = logging.getLogger(__name__)

def read_and_process_csv(file_path: str) -> pd.DataFrame:
    logger.info(f"Reading CSV file: {file_path}")
    try:
        df = pd.read_csv(file_path)
        logger.info(f"Successfully loaded {len(df)} rows")
        return df
    except Exception as e:
        logger.error(f"Failed to read CSV: {e}")
        raise
```

3. Context Managers

```python
from contextlib import contextmanager

@contextmanager
def tk_window():
    """Context manager for tkinter windows"""
    root = tk.Tk()
    root.withdraw()
    root.attributes("-topmost", True)
    try:
        yield root
    finally:
        root.destroy()

def select_file(title: str = "Select CSV file") -> str:
    with tk_window() as root:
        file_path = fd.askopenfilename(
            title=title,
            filetypes=[("CSV files", "*.csv"), ("All files", "*.*")]
        )
    return file_path
```

4. Configuration File Support

```ini
[plot]
sample_size = 750
figure_size = (15, 10)
output_format = svg
default_style = ggplot

[data]
hex_column = 1
skip_bad_lines = true
```

```
encoding = utf-8

[display]
show_plot = true
save_dialog = true
```

5. Better Mathematical Functions

```python
def generate_plot_coordinates(df: pd.DataFrame) -> pd.DataFrame:
    """Generate x,y coordinates with configurable distributions"""
    n = len(df)

    # Use clear, documented distributions
    df['x'] = np.random.normal(loc=0, scale=2, size=n)  # Normal distribution
    df['y'] = np.random.uniform(low=0, high=1, size=n)  # Uniform distribution

    return df

def calculate_point_sizes(values: np.ndarray, min_size: float = 10,
                          max_size: float = 1000) -> np.ndarray:
    """Calculate point sizes with proper normalization"""
    if len(values) == 0:
        return np.array([])

    # Normalize to [0, 1]
    vmin, vmax = values.min(), values.max()
    if vmax == vmin:
        normalized = np.ones_like(values) * 0.5
    else:
        normalized = (values - vmin) / (vmax - vmin)

    # Scale to desired range
    return min_size + normalized * (max_size - min_size)
```

Testing Strategy

Unit Tests

- Test hex value extraction
- Test coordinate generation
- Test size calculations
- Test error handling

Integration Tests

- Test full pipeline with sample data
- Test GUI interactions

- Test CLI argument parsing

Performance Tests

- Test with large datasets (>100k rows)
- Memory usage profiling
- Rendering performance

Security Considerations

1. **File Path Injection**: Validate file paths to prevent directory traversal
2. **CSV Injection**: Sanitize CSV data to prevent formula injection
3. **Resource Limits**: Implement maximum file size and row limits
4. **GUI Security**: Validate file dialog responses

Performance Optimization

1. **Vectorized Operations**: Replace loops with numpy operations
2. **Lazy Loading**: Read CSV in chunks for large files
3. **Caching**: Cache calculated values for reuse
4. **Parallel Processing**: Use multiprocessing for large datasets

Conclusion

The scatter plot visualization script provides useful functionality but requires significant improvements in error handling, code organization, and mathematical operations. Implementing the suggested improvements would enhance reliability, maintainability, and performance while making the code more professional and production-ready.

Priority Improvements

1. Fix mathematical operations and random number generation
2. Implement proper error handling and validation
3. Add comprehensive logging and documentation
4. Refactor into object-oriented design
5. Add configuration file support
6. Implement comprehensive testing

Estimated Effort

- High Priority Fixes: 8-12 hours
- Code Refactoring: 16-24 hours
- Testing Implementation: 8-12 hours
- Documentation: 4-6 hours
- Total: 36-54 hours