PlotGenC++ Documentation

This documentation details the usage of the PlotGenC++ library for generating graphics in C++.

Table of Contents

1. Introduction
2. Installation
3. Basic Concepts
4. API Reference
5. Detailed Examples
6. Symbols and Styles
7. Advanced Features
8. Tips and Best Practices

Introduction

PlotGenC++ is a C++ data visualization library inspired by common plotting systems like MATLAB and matplotlib (Python). The library is built on SFML for graphical rendering and offers an intuitive interface for creating high-quality scientific charts.

Design Goals

- Simple and intuitive interface
- Flexibility and customization
- Publication-ready export quality
- Optimized performance for real-time display

Installation

Prerequisites

- C++ compiler with C++17 support
- CMake 3.10 or higher
- SFML 2.5 or higher

Installation Steps

1. Clone the repository or download the source
2. Configure with CMake:

```
mkdir build
cd build
cmake ..
```

3. Compile the project:

```
cmake --build .
```

4. Link your project with the PlotGenCpp library and SFML dependencies

Basic Concepts

Chart Structure

Each `PlotGen` instance can contain multiple subplots organized in a grid. Here are the essential concepts:

- **Figure**: Main container for all charts
- **Subplot**: An individual chart in the grid
- **Curve**: A series of data plotted on a chart
- **Style**: Defines the visual appearance of a curve

Typical Workflow

1. Create a `PlotGen` instance
2. Get a reference to a subplot
3. Configure subplot properties
4. Plot data on the subplot
5. Display or save the result

```
// Typical workflow example
PlotGen plt(800, 600);              // Window dimensions
auto& fig = plt.subplot(0, 0);      // Reference to subplot
plt.set_title(fig, "My Chart");     // Configuration
plt.plot(fig, x_data, y_data);      // Plotting
plt.show();                         // Display
```

API Reference

PlotGen Class

Constructor

```
PlotGen(unsigned int width = 1200, unsigned int height = 900, unsigned int
rows = 1, unsigned int cols = 1)
```

- **width**: Window width in pixels
- **height**: Window height in pixels
- **rows**: Number of rows in the subplot grid
- **cols**: Number of columns in the subplot grid

Style Structure

```
struct Style {
    sf::Color color;
    float thickness;
    std::string line_style;
    std::string legend;
    std::string symbol_type;
    float symbol_size;
}
```

- **color**: Curve color
- **thickness**: Line thickness
- **line_style**: Line style ("solid", "dashed", "points", "none")
- **legend**: Legend text
- **symbol_type**: Symbol type ("none", "circle", "square", "triangle", "diamond", "star")
- **symbol_size**: Symbol size in pixels

Main Methods

Subplot Management

```
Figure& subplot(unsigned int row, unsigned int col)
```

Gets a reference to the subplot at position (row, col).

Chart Configuration

```
void set_title(Figure& fig, const std::string& title)
void set_xlabel(Figure& fig, const std::string& label)
void set_ylabel(Figure& fig, const std::string& label)
void set_axis_limits(Figure& fig, float xmin, float xmax, float ymin, float
ymax)
void set_polar_axis_limits(Figure& fig, float max_radius)
void show_legend(Figure& fig, bool show)
void set_legend_position(Figure& fig, const std::string& position)
void grid(Figure& fig, bool major = true, bool minor = false)
void set_grid_color(Figure& fig, sf::Color major_color, sf::Color
minor_color)
void set_equal_axes(Figure& fig, bool equal = true)
```

Data Plotting

```
void plot(Figure& fig, const std::vector<float>& x, const
std::vector<float>& y, const Style& style = Style())
void hist(Figure& fig, const std::vector<float>& data, int bins = 10, const
Style& style = Style(), float bar_width_ratio = 0.9f)
void polar_plot(Figure& fig, const std::vector<float>& theta, const
std::vector<float>& r, const Style& style = Style())
```

Display and Export

```
void show()
void save(const std::string& filename)
```

Legend Positioning

```
void set_legend_position(Figure& fig, const std::string& position)
```

This function allows you to control where the legend appears on the chart. Available positions are:

- **"top-right"** (default): Positions the legend in the top-right area of the chart
- **"top-left"**: Positions the legend in the top-left area of the chart
- **"bottom-right"**: Positions the legend in the bottom-right area of the chart
- **"bottom-left"**: Positions the legend in the bottom-left area of the chart
- **"outside-right"**: Places the legend outside the chart on the right side

Example:

```
auto& fig = plt.subplot(0, 0);
// Configure and plot...
plt.set_legend_position(fig, "bottom-left");
```

Benefits of the new legend system:

- Automatic sizing based on content
- Visual representation of line styles and symbols within legends
- Flexible positioning for optimal chart layout
- Support for placing legends outside the chart area to avoid overlapping with data

Optimized Window Management

PlotGenC++ now features improved window management:

- No window is displayed when only using `save()` without `show()`
- Reduced font sizes for titles and legend text for better proportions

- Titles positioned outside the plotting area for cleaner data visualization

Image Export

```
void save(const std::string& filename)
```

This method allows you to export the chart in PNG or JPG format. Quality and compression options are automatically managed to ensure good image quality. The file format is determined by the extension of the filename:

- Use `.png` for lossless PNG format (best for diagrams and charts with sharp lines)
- Use `.jpg` for compressed JPG format (suitable for images with many color gradients)

Example:

```
plt.save("my_chart.png");  // Save as PNG
plt.save("my_chart.jpg");  // Save as JPG
```

Interactive Display

```
void show()
```

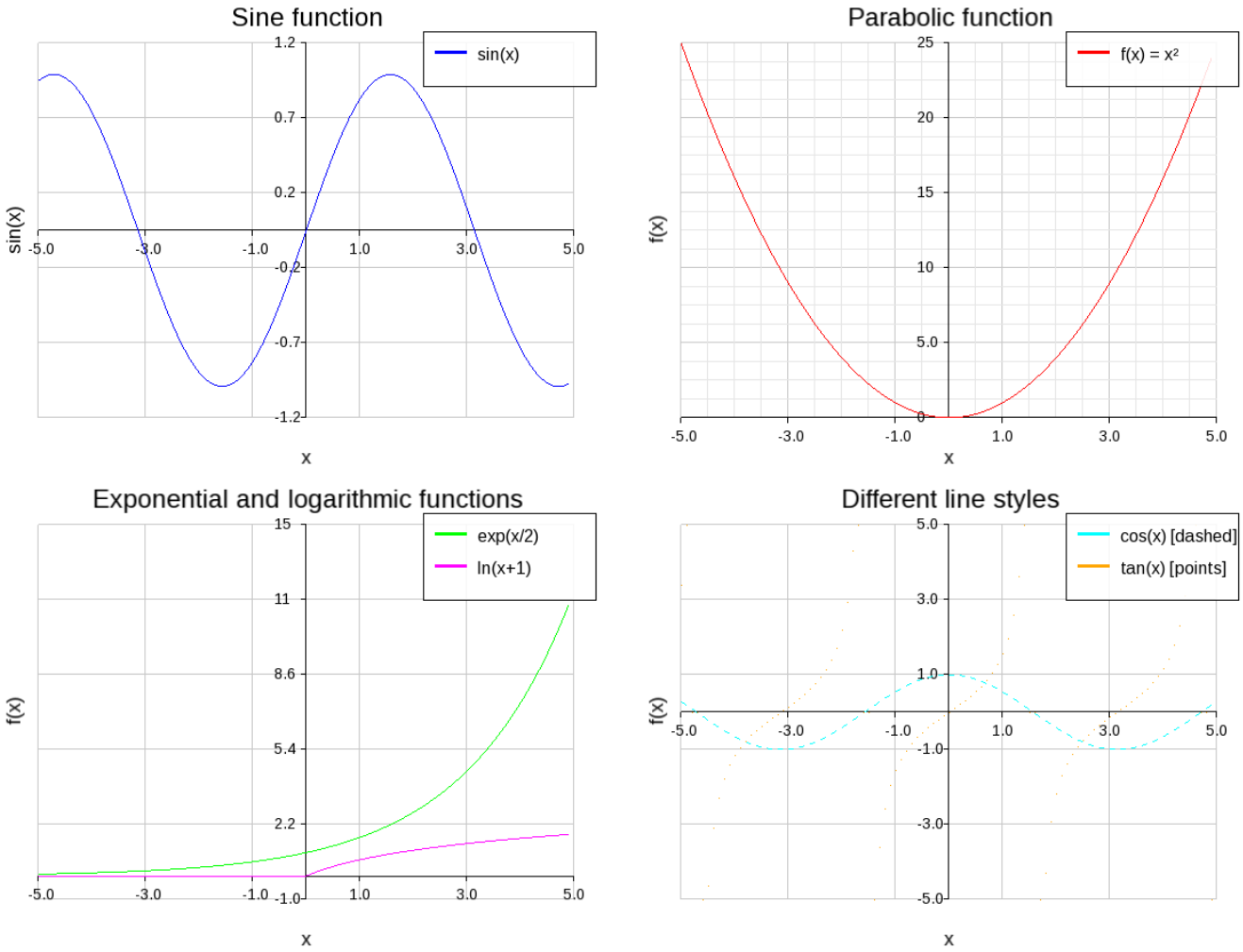
This method opens an interactive window to visualize the chart. It handles window events, allowing you to close the window when you're done viewing the chart. The window remains open until it is closed by the user, either by clicking the close button or pressing the Escape key.

Example:

```
plt.show();  // Display the chart in an interactive window
```

Detailed Examples

Example 1: Basic 2D Plots



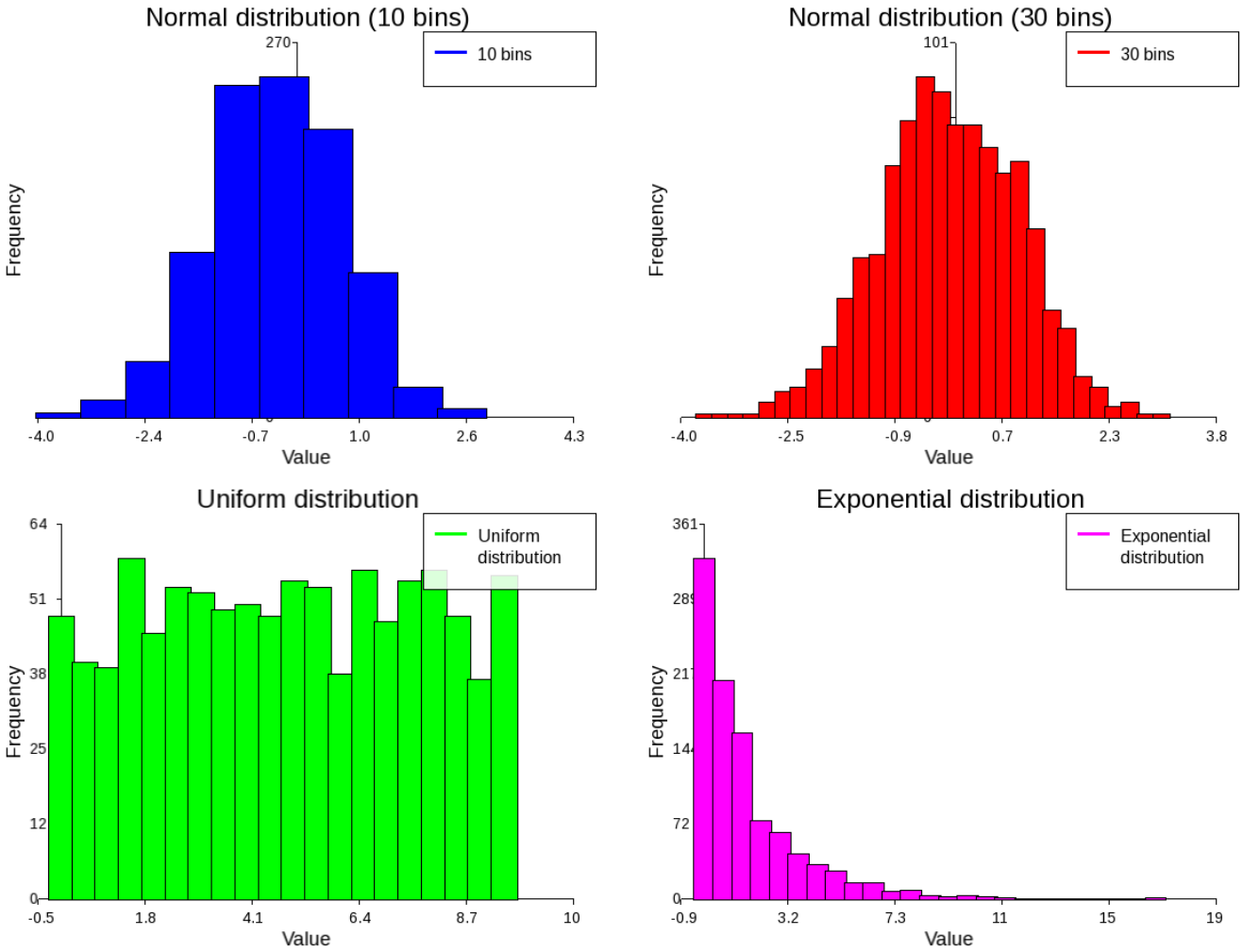
This example shows how to create simple 2D plots with different mathematical functions:

- Sine function
- Parabolic function
- Exponential and logarithmic functions
- Different line styles

Key points:

- 2x2 grid organization
- Axis limit customization
- Grid and legend usage
- Various line styles

Example 2: Histograms



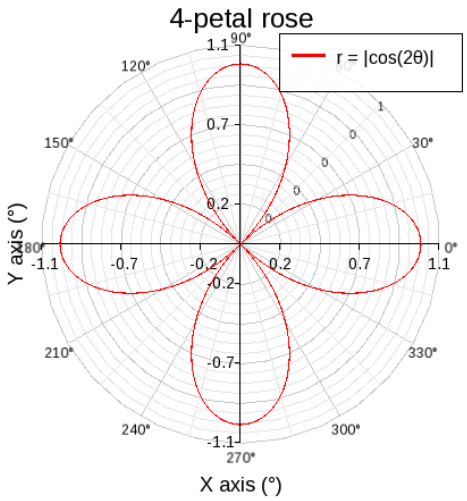
This example illustrates creating histograms with different distributions:

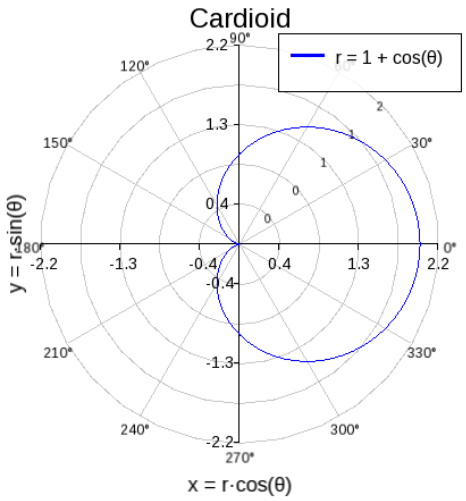
- Normal distribution with different bin counts
- Uniform distribution
- Exponential distribution

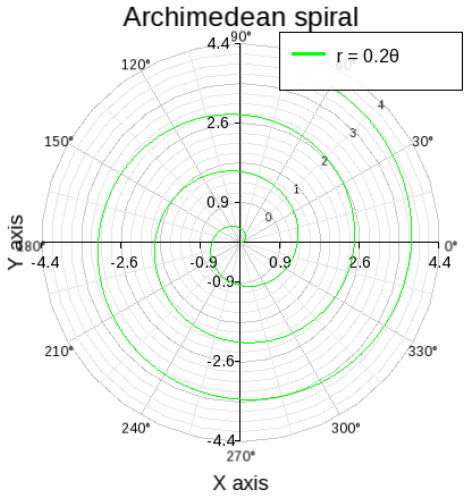
Key points:

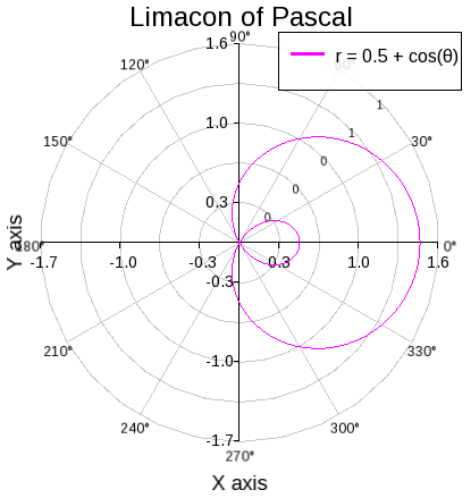
- Control of bin count
- Automatic axis limit detection
- Color customization

Example 3: Polar Plots









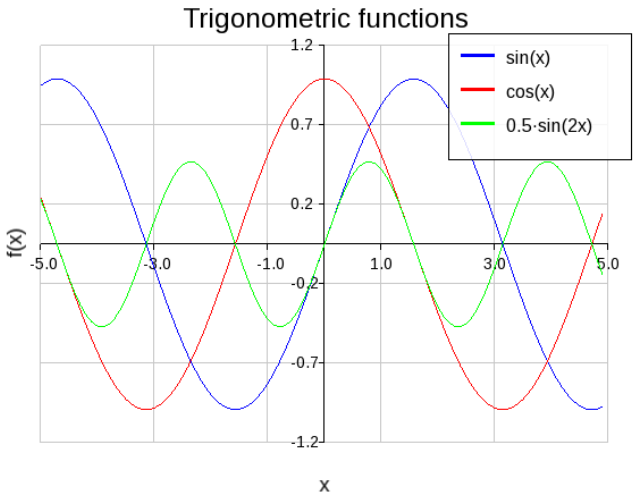
This example presents plots in polar coordinates:

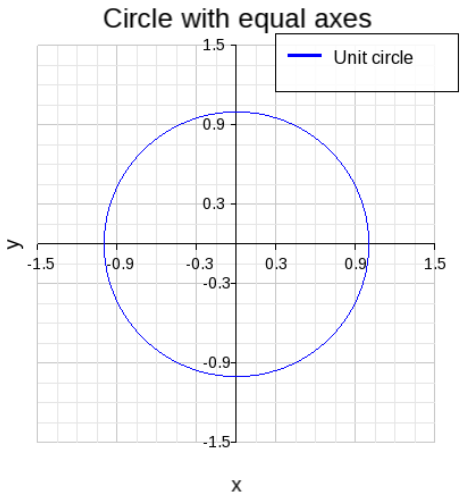
- 4-petal rose
- Cardioid
- Archimedes' spiral
- Pascal's Limaçon

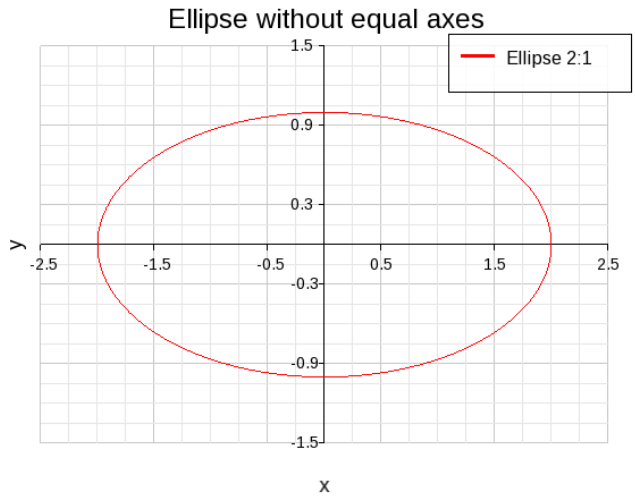
Key points:

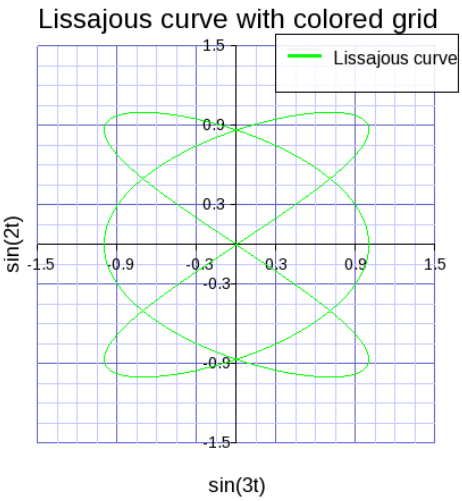
- Automatic conversion between polar and Cartesian coordinates
- Polar grids with annotations
- Equal scale axes to preserve shape

Example 4: Multiple Plots and Customization









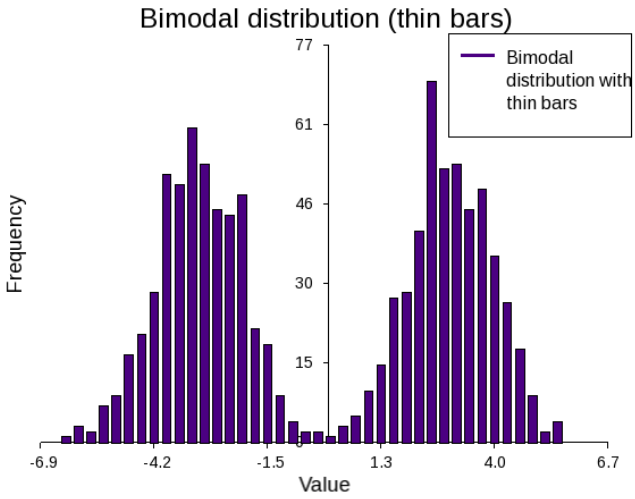
This example shows advanced customization features:

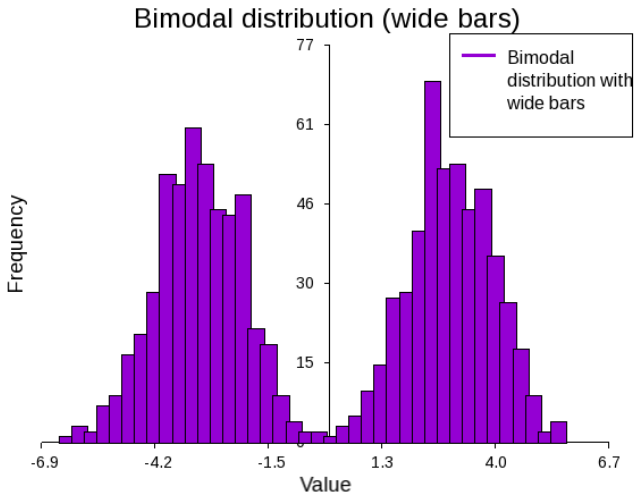
- Multiple curves on the same chart
- Equal scale axes for circles
- Grid color customization

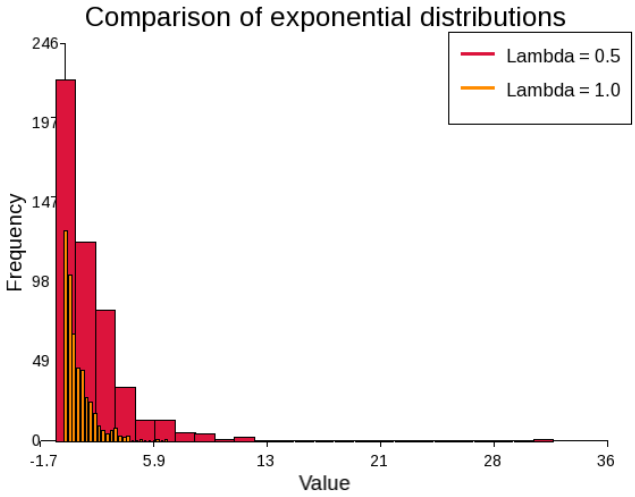
Key points:

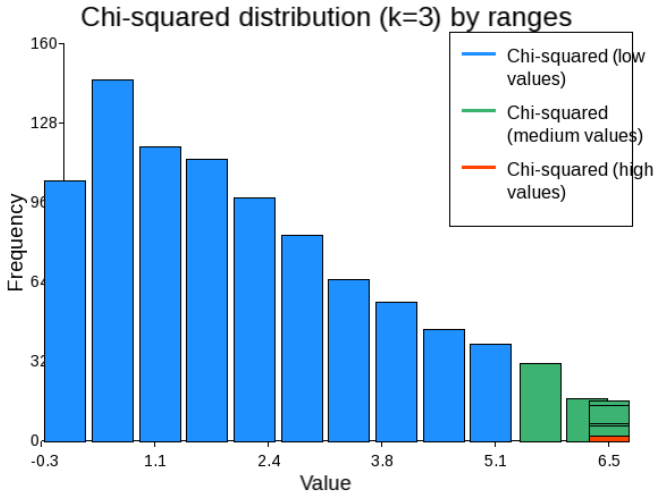
- Overlaying multiple curves
- Equal axis options
- Grid and color customization

Example 5: Advanced Histograms









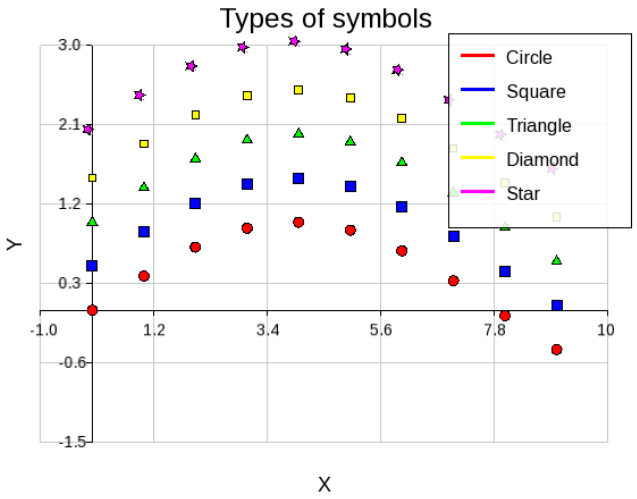
This example presents advanced techniques for histograms:

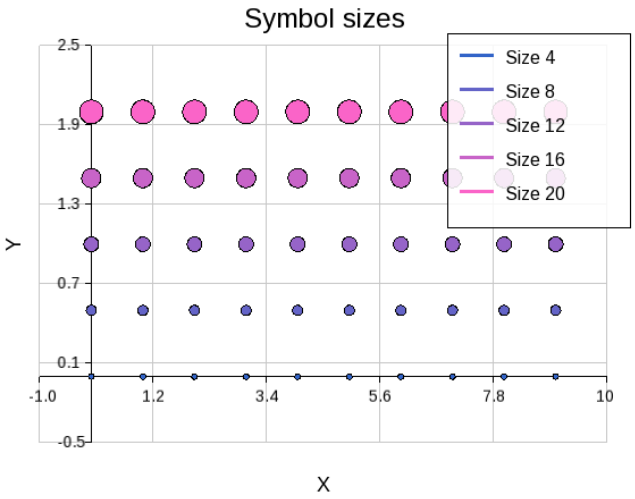
- Bimodal distribution with thin and wide bars
- Comparison of distributions on the same chart
- Histograms with color gradients

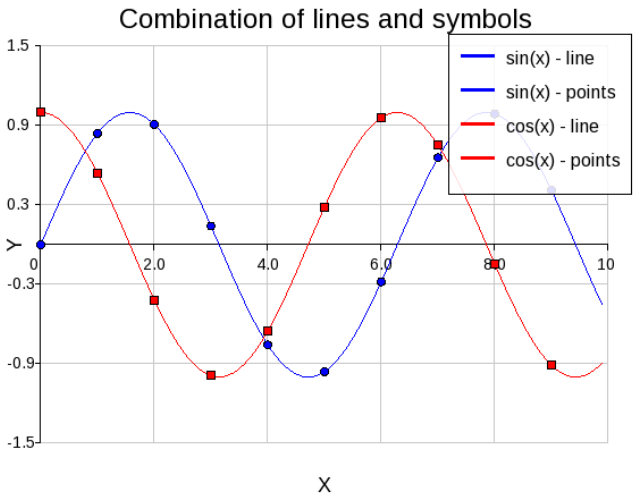
Key points:

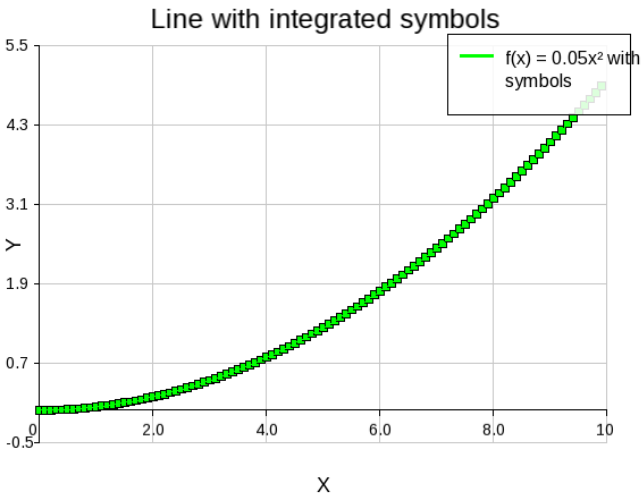
- Control of bar width
- Histogram overlay
- Data division by ranges

Example 6: Plots with Symbols









This example illustrates the use of symbols:

- Different symbol types
- Various symbol sizes
- Combination of lines and symbols

Key points:

- Control of symbol types
- Size customization
- Combination of visualization styles

Symbols and Styles

Available Symbol Types

- **circle**: Circle
- **square**: Square
- **triangle**: Triangle
- **diamond**: Diamond
- **star**: Star
- **none**: No symbol

Line Styles

- **solid**: Continuous line
- **dashed**: Dotted line
- **points**: Points only
- **none**: No line (useful for displaying only symbols)

Predefined Colors

- `sf::Color::Red`
- `sf::Color::Green`
- `sf::Color::Blue`
- `sf::Color::Yellow`
- `sf::Color::Magenta`
- `sf::Color::Cyan`
- `sf::Color::White`
- `sf::Color::Black`

Custom color: `sf::Color(r, g, b)` with r, g, b between 0 and 255.

Advanced Features

Custom Grids

```
void grid(Figure& fig, bool major, bool minor)
void set_grid_color(Figure& fig, sf::Color major_color, sf::Color
minor_color)
```

The library allows customization of grid display with:

- Major and minor grid lines
- Customizable colors
- Automatic adaptation to polar coordinates

Equal Axis Scale

```
void set_equal_axes(Figure& fig, bool equal)
```

This feature ensures that units on the X and Y axes are identical, which is essential for:

- Preserving the shape of circles
- Correctly visualizing geometric objects
- Producing accurate polar plots

Multiline Legends

Legends are automatically wrapped if they exceed a certain width, which allows:

- Displaying detailed descriptions

- Maintaining readability of long legends
- Including Unicode symbols in legends

Tips and Best Practices

Performance Optimization

- Limit the number of points for complex plots
- Use symbols judiciously (they are expensive to display)
- Prefer PNG export for best quality

Troubleshooting Common Problems

- If fonts don't load correctly, verify that the arial.ttf file is in the correct location
- For Linux systems, the library automatically searches for system fonts if arial.ttf is not found

Tips for Beautiful Charts

- Use contrasting colors for different curves
- Enable grids to improve readability
- Adapt axis limits to the data rather than using default values
- Prefer different line styles when colors are similar
- Use symbols for important points