Documentation PlotGenC++

Cette documentation détaille l'utilisation de la bibliothèque PlotGenC++ pour la génération de graphiques en C++.

Table des matières

Introduction

PlotGenC++ est une bibliothèque de visualisation de données en C++ qui s'inspire des systèmes de tracé de graphiques courants comme MATLAB et matplotlib (Python). La bibliothèque est construite sur SFML pour le rendu graphique et offre une interface intuitive pour créer des graphiques scientifiques de haute qualité.

Objectifs de conception

- Interface simple et intuitive
- Flexibilité et personnalisation
- Qualité d'exportation adaptée aux publications
- Performances optimisées pour l'affichage en temps réel

Installation

Prérequis

- Compilateur C++ avec support C++17
- CMake 3.10 ou supérieur
- SFML 2.5 ou supérieur

Étapes d'installation

1. Cloner le dépôt ou télécharger les sources
2. Configurer avec CMake :

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

3. Compiler le projet :

```
cmake --build .
```

4. Lier votre projet avec la bibliothèque PlotGenCpp et les dépendances SFML

Concepts de base

Structure d'un graphique

Chaque instance de `PlotGen` peut contenir plusieurs sous-graphiques (subplots) organisés en grille. Voici les concepts essentiels :

- **Figure** : Conteneur principal pour l'ensemble des graphiques
- **Subplot** : Un graphique individuel dans la grille
- **Courbe (Curve)** : Une série de données tracée sur un graphique
- **Style** : Définit l'apparence visuelle d'une courbe

Flux de travail typique

1. Création d'une instance `PlotGen`
2. Récupération d'une référence à un subplot
3. Configuration des propriétés du subplot
4. Tracé de données sur le subplot
5. Affichage ou sauvegarde du résultat

```
// Exemple de flux de travail typique
PlotGen plt(800, 600);              // Dimensions de la fenêtre
auto& fig = plt.subplot(0, 0);      // Référence au subplot
plt.set_title(fig, "Mon graphique"); // Configuration
plt.plot(fig, x_data, y_data);       // Tracé
plt.show();                          // Affichage
```

API de référence

Classe PlotGen

Constructeur

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

- **width** : Largeur de la fenêtre en pixels
- **height** : Hauteur de la fenêtre en pixels
- **rows** : Nombre de lignes dans la grille de subplots

- **cols** : Nombre de colonnes dans la grille de subplots

Structure Style

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

- **color** : Couleur de la courbe
- **thickness** : Épaisseur de la ligne
- **line_style** : Style de ligne ("solid", "dashed", "points", "none")
- **legend** : Texte de légende
- **symbol_type** : Type de symbole ("none", "circle", "square", "triangle", "diamond", "star")
- **symbol_size** : Taille du symbole en pixels

Méthodes principales

Gestion des subplots

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

Obtient une référence au subplot à la position (row, col).

Configuration des graphiques

```
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 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)
```

Tracé de données

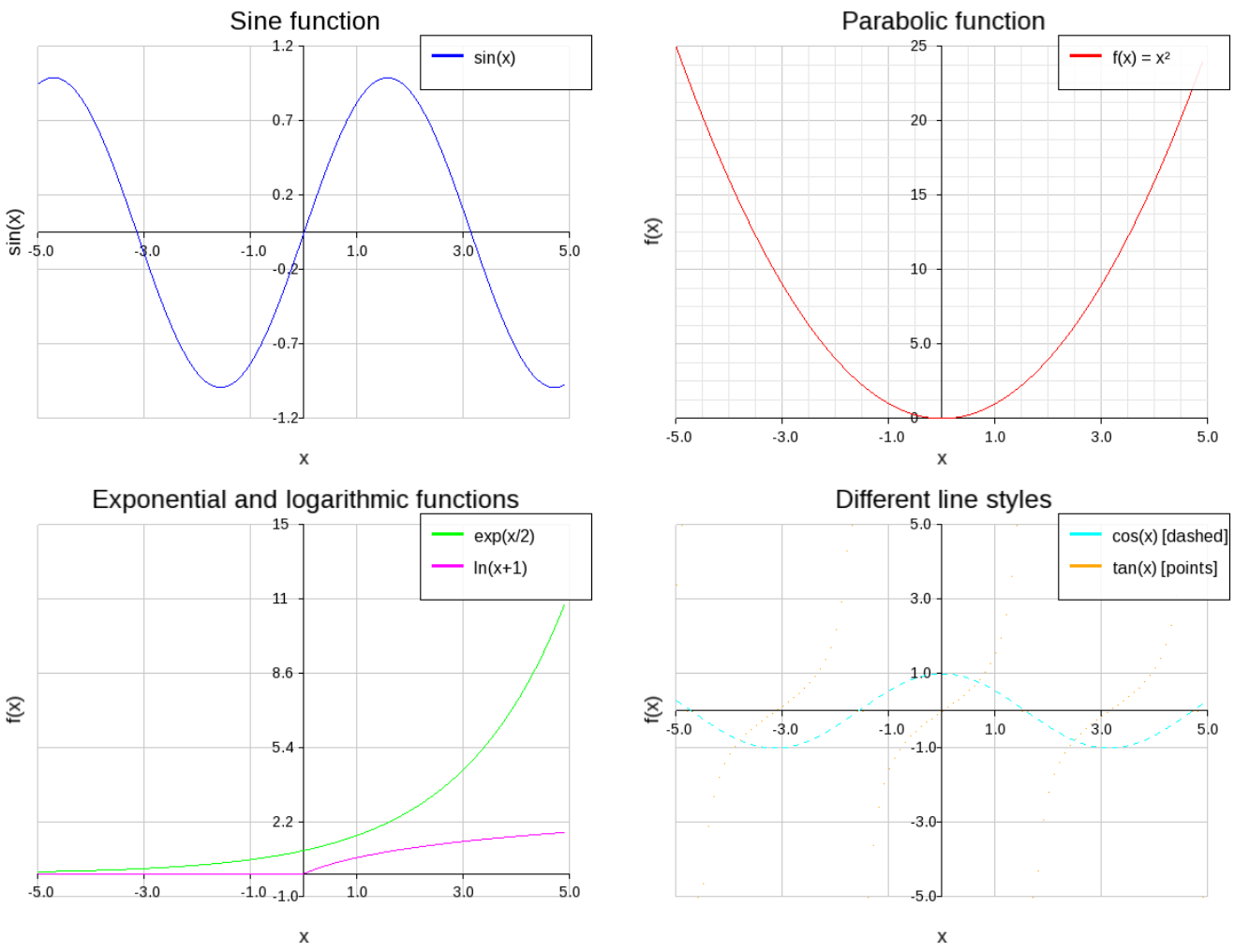
```
void plot(Figure& fig, const std::vector<float>& x, const
std::vector<float>& y, const Style& style = Style())
void hist(Figure& fig, const std::vector<float>& data, int bins = 10, const
Style& style = Style(), float bar_width_ratio = 0.9f)
void polar_plot(Figure& fig, const std::vector<float>& theta,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

Affichage et exportation

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

Exemples détaillés

Exemple 1 : Graphiques 2D basiques



Cet exemple montre comment créer des graphiques 2D simples avec différentes fonctions mathématiques :

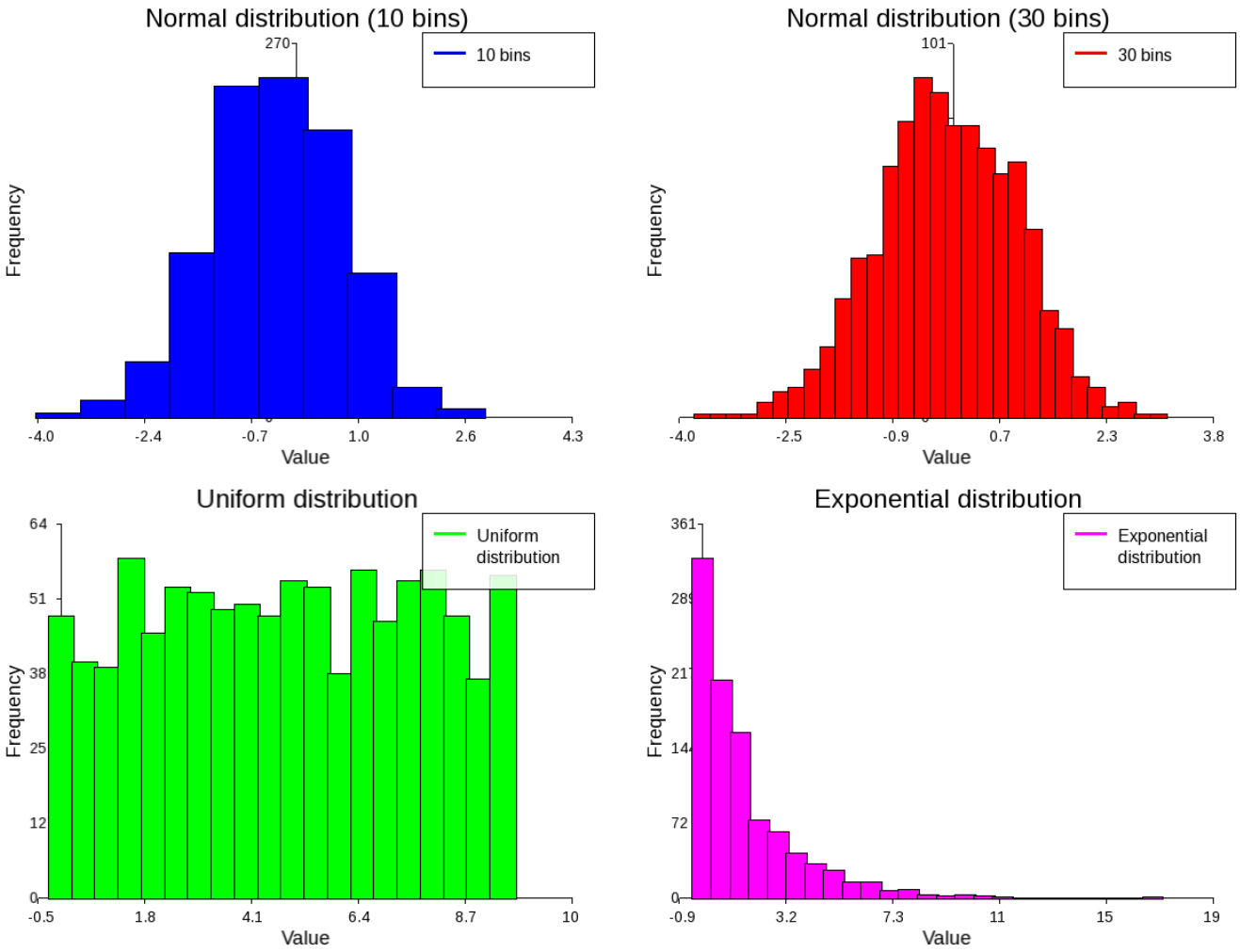
- Fonction sinus
- Fonction parabolique
- Fonctions exponentielle et logarithmique

- Différents styles de lignes

Points clés :

- Organisation en grille 2x2
- Personnalisation des limites d'axes
- Utilisation des grilles et légendes
- Styles de lignes variés

Exemple 2 : Histogrammes



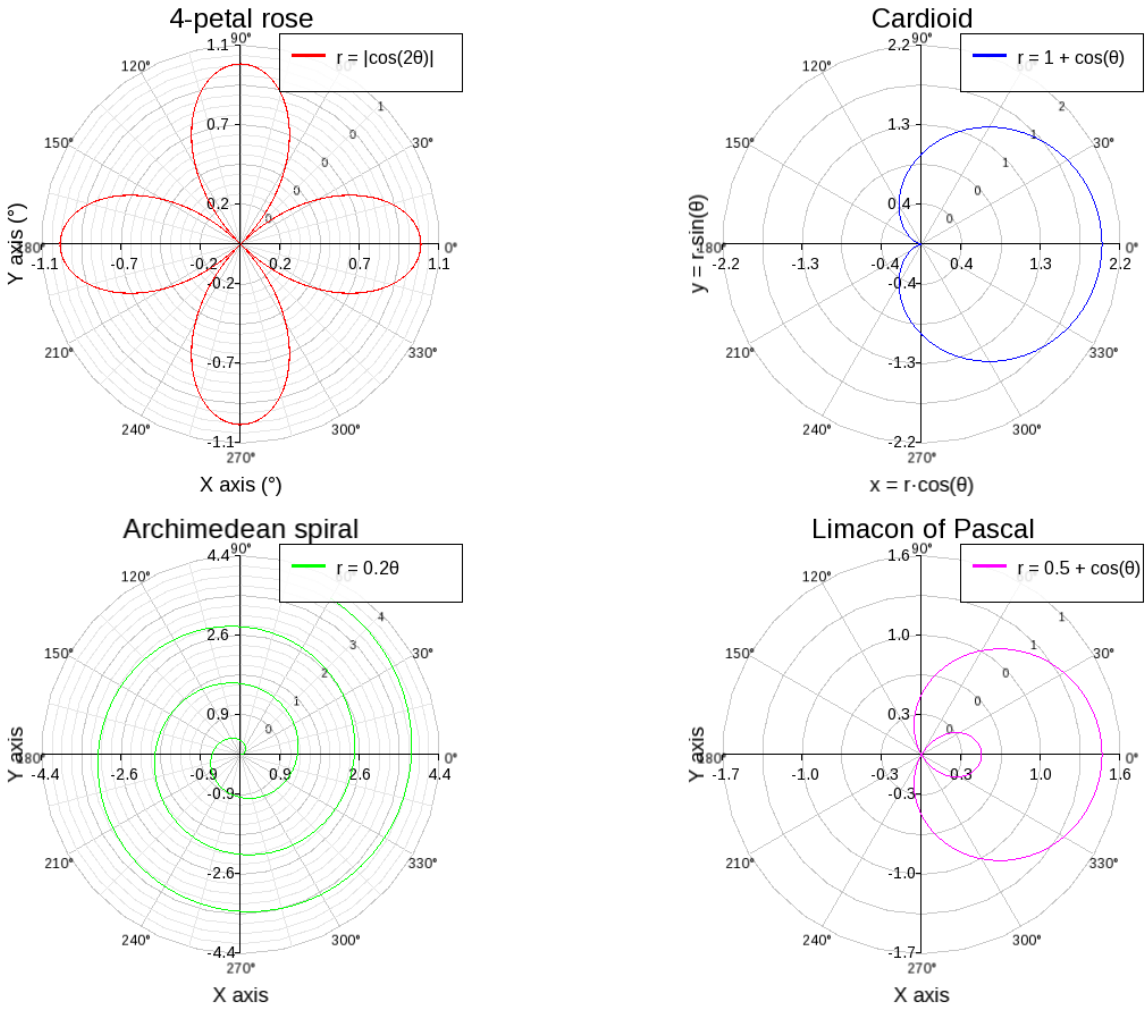
Cet exemple illustre la création d'histogrammes avec différentes distributions :

- Distribution normale avec différents nombres de bins
- Distribution uniforme
- Distribution exponentielle

Points clés :

- Contrôle du nombre de bins
- Détection automatique des limites d'axes
- Personnalisation des couleurs

Exemple 3 : Graphiques polaires



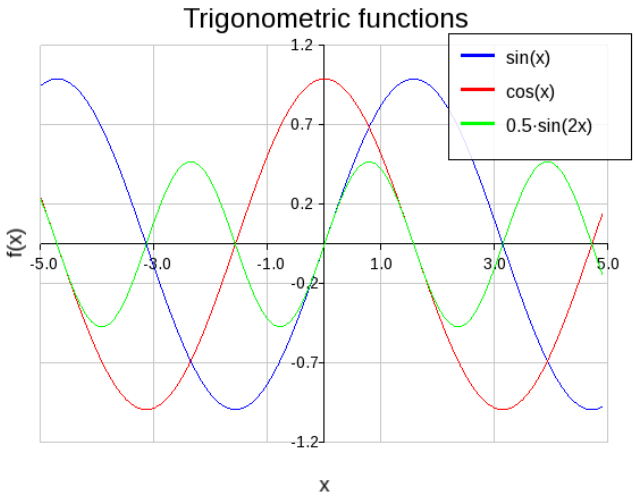
Cet exemple présente des graphiques en coordonnées polaires :

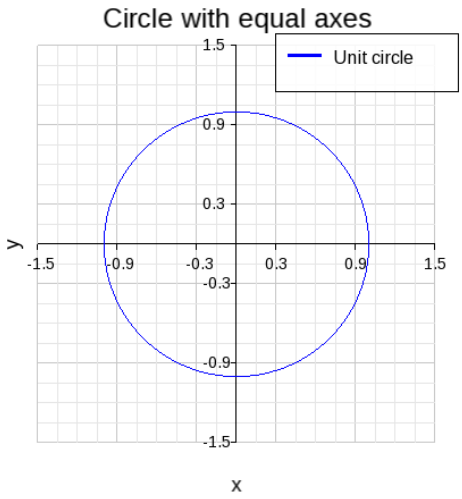
- Rose à 4 pétales
- Cardioïde
- Spirale d'Archimède
- Limaçon de Pascal

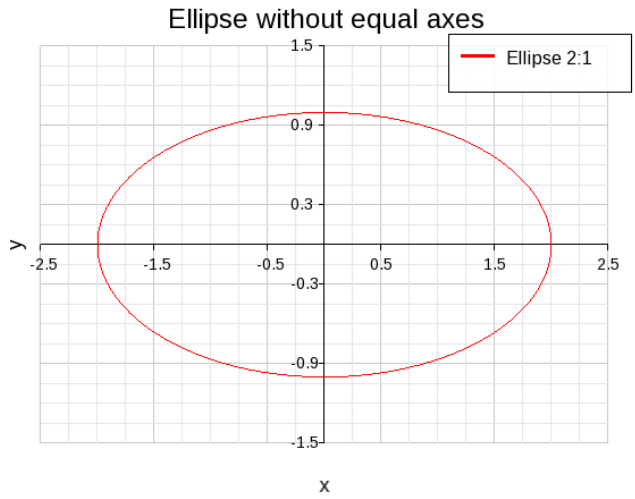
Points clés :

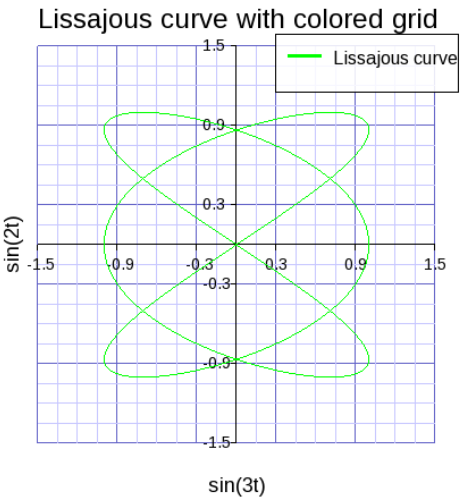
- Conversion automatique entre coordonnées polaires et cartésiennes
- Grilles polaires avec annotations
- Axes d'échelle égale pour préserver la forme

Exemple 4 : Graphiques multiples et personnalisation









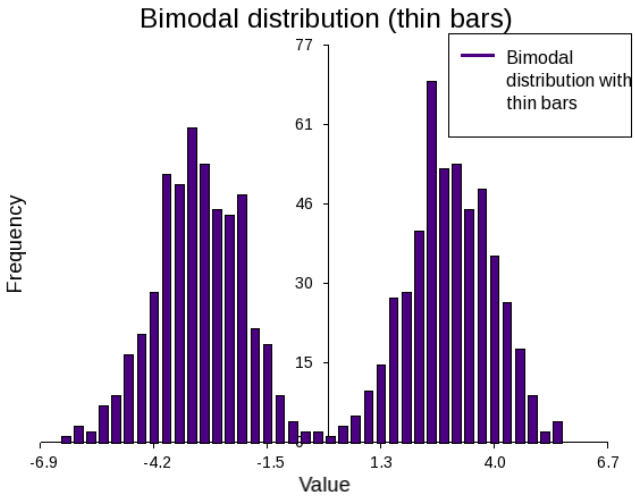
Cet exemple montre des fonctionnalités avancées de personnalisation :

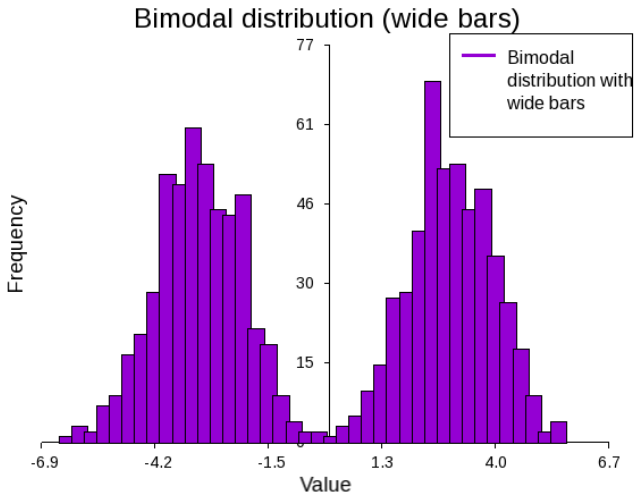
- Plusieurs courbes sur un même graphique
- Axes d'échelle égale pour les cercles
- Personnalisation des couleurs de grille

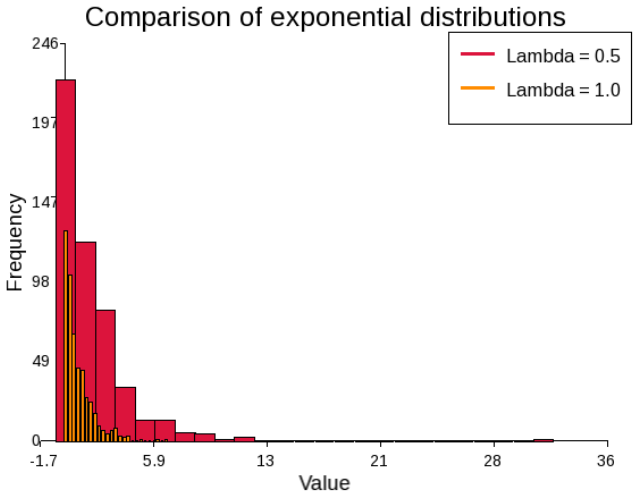
Points clés :

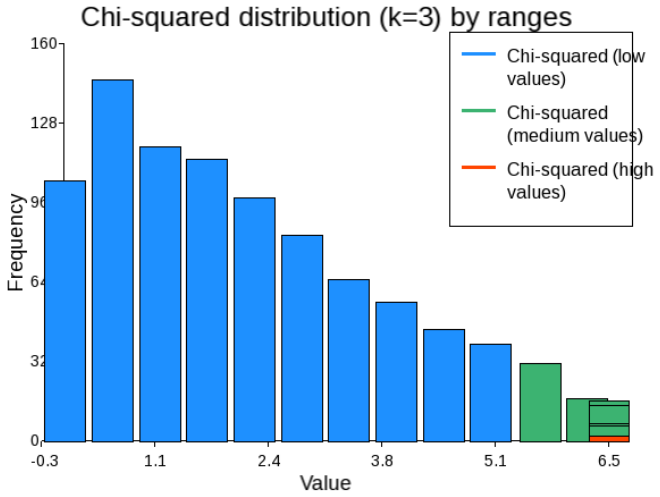
- Superposition de plusieurs courbes
- Options d'axes égaux
- Personnalisation des grilles et couleurs

Exemple 5 : Histogrammes avancés









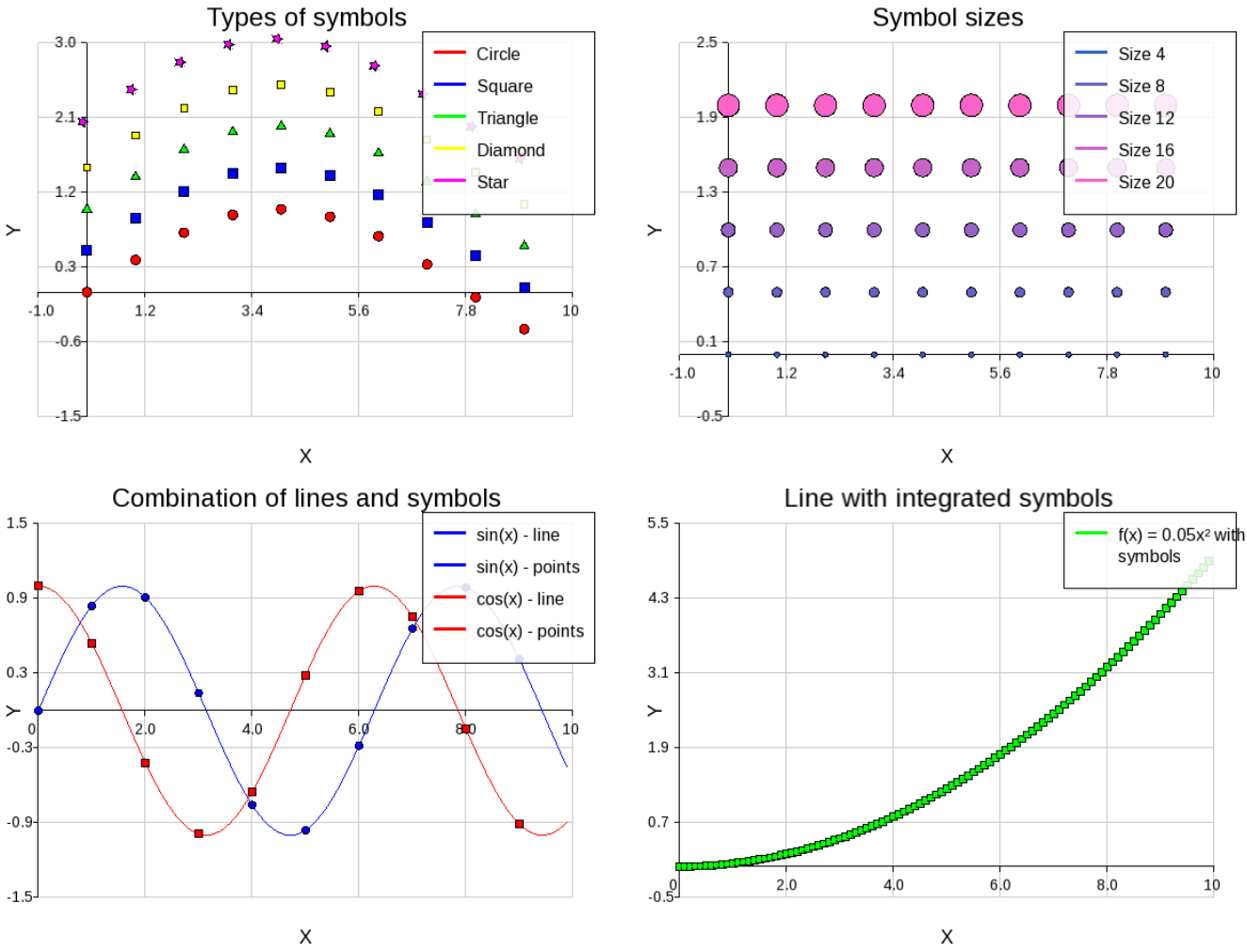
Cet exemple présente des techniques avancées pour les histogrammes :

- Distribution bimodale avec barres fines et larges
- Comparaison de distributions sur le même graphique
- Histogrammes avec dégradés de couleurs

Points clés :

- Contrôle de la largeur des barres
- Superposition d'histogrammes
- Division des données par plages

Exemple 6 : Courbes avec symboles



Cet exemple illustre l'utilisation de symboles :

- Différents types de symboles
- Tailles de symboles variées
- Combinaison de lignes et symboles

Points clés :

- Contrôle des types de symboles
- Personnalisation des tailles
- Combinaison de styles de visualisation

Symboles et styles

Types de symboles disponibles

- **circle** : Cercle
- **square** : Carré
- **triangle** : Triangle
- **diamond** : Losange
- **star** : Étoile
- **none** : Pas de symbole

Styles de ligne

- **solid** : Ligne continue
- **dashed** : Ligne pointillée
- **points** : Points uniquement
- **none** : Pas de ligne (utile pour n'afficher que les symboles)

Couleurs prédéfinies

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

Couleur personnalisée : `sf::Color(r, g, b)` avec r, g, b entre 0 et 255.

Fonctionnalités avancées

Grilles personnalisées

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

La bibliothèque permet de personnaliser l'affichage des grilles avec :

- Lignes de grille majeures et mineures
- Couleurs personnalisables
- Adaptation automatique aux coordonnées polaires

Échelle d'axes égale

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

Cette fonctionnalité garantit que les unités sur les axes X et Y sont identiques, ce qui est essentiel pour :

- Préserver la forme des cercles
- Visualiser correctement les objets géométriques
- Produire des graphiques polaires précis

Légendes multilignes

Les légendes sont automatiquement découpées si elles dépassent une certaine largeur, ce qui permet :

- D'afficher des descriptions détaillées

- De maintenir la lisibilité des légendes longues
- D'inclure des symboles Unicode dans les légendes

Astuces et bonnes pratiques

Optimisation des performances

- Limiter le nombre de points pour les tracés complexes
- Utiliser judicieusement les symboles (ils sont coûteux à afficher)
- Préférer l'exportation en PNG pour la meilleure qualité

Résolution des problèmes courants

- Si les fonts ne se chargent pas correctement, vérifier que le fichier arial.ttf est présent à l'emplacement adéquat
- Pour les systèmes Linux, la bibliothèque recherche automatiquement les polices système si arial.ttf n'est pas trouvé

Conseils pour de beaux graphiques

- Utiliser des couleurs contrastées pour les différentes courbes
- Activer les grilles pour améliorer la lisibilité
- Adapter les limites d'axes aux données plutôt que d'utiliser les valeurs par défaut
- Préférer des styles de lignes différents lorsque les couleurs sont similaires
- Utiliser des symboles pour les points importants