Documentation PlotGenC++

- Auteur : Sofiane KHELLADI
- Date : 21/04/2025
- Version: 1.0

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 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)
```

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())
void circle(Figure& fig, double x0, double y0, double r, const Style& style
= Style())
void text(Figure& fig, double x, double y, const std::string& text_content,
const Style& style = Style())
```

Affichage et exportation

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

Positionnement des légendes

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

Cette fonction permet de contrôler où la légende apparaît sur le graphique. Les positions disponibles sont :

- **"top-right"** (par défaut) : Positionne la légende dans la zone en haut à droite du graphique
- **"top-left"** : Positionne la légende dans la zone en haut à gauche du graphique
- **"bottom-right"** : Positionne la légende dans la zone en bas à droite du graphique
- **"bottom-left"** : Positionne la légende dans la zone en bas à gauche du graphique
- **"outside-right"** : Place la légende à l'extérieur du graphique sur le côté droit

Exemple :

```
auto& fig = plt.subplot(0, 0);
// Configuration et tracé...
plt.set_legend_position(fig, "bottom-left");
```

Avantages du nouveau système de légendes :

- Dimensionnement automatique basé sur le contenu
- Représentation visuelle des styles de lignes et des symboles dans les légendes
- Positionnement flexible pour une mise en page optimale des graphiques
- Prise en charge du placement des légendes en dehors de la zone de graphique pour éviter le chevauchement avec les données

Gestion optimisée des fenêtres

PlotGenC++ propose désormais une gestion améliorée des fenêtres :

- Aucune fenêtre n'est affichée lors de l'utilisation de `save()` sans `show()`
- Tailles de police réduites pour les titres et les textes de légende pour de meilleures proportions
- Titres positionnés en dehors de la zone de tracé pour une visualisation plus claire des données

Exportation vectorielle SVG

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

Cette méthode permet d'exporter le graphique au format SVG (Scalable Vector Graphics). Le format SVG présente plusieurs avantages par rapport aux formats bitmap comme PNG ou JPG :

- **Mise à l'échelle parfaite** : Les graphiques SVG peuvent être agrandis à n'importe quelle taille sans perte de qualité
- **Taille de fichier réduite** pour les graphiques composés principalement de lignes et de formes géométriques
- **Édition ultérieure** : Les fichiers SVG peuvent être modifiés avec des éditeurs vectoriels comme Inkscape ou Adobe Illustrator
- **Intégration dans des documents** : Idéal pour l'intégration dans des documents scientifiques et des présentations
- **Compatibilité web** : Les fichiers SVG peuvent être directement intégrés dans des pages web

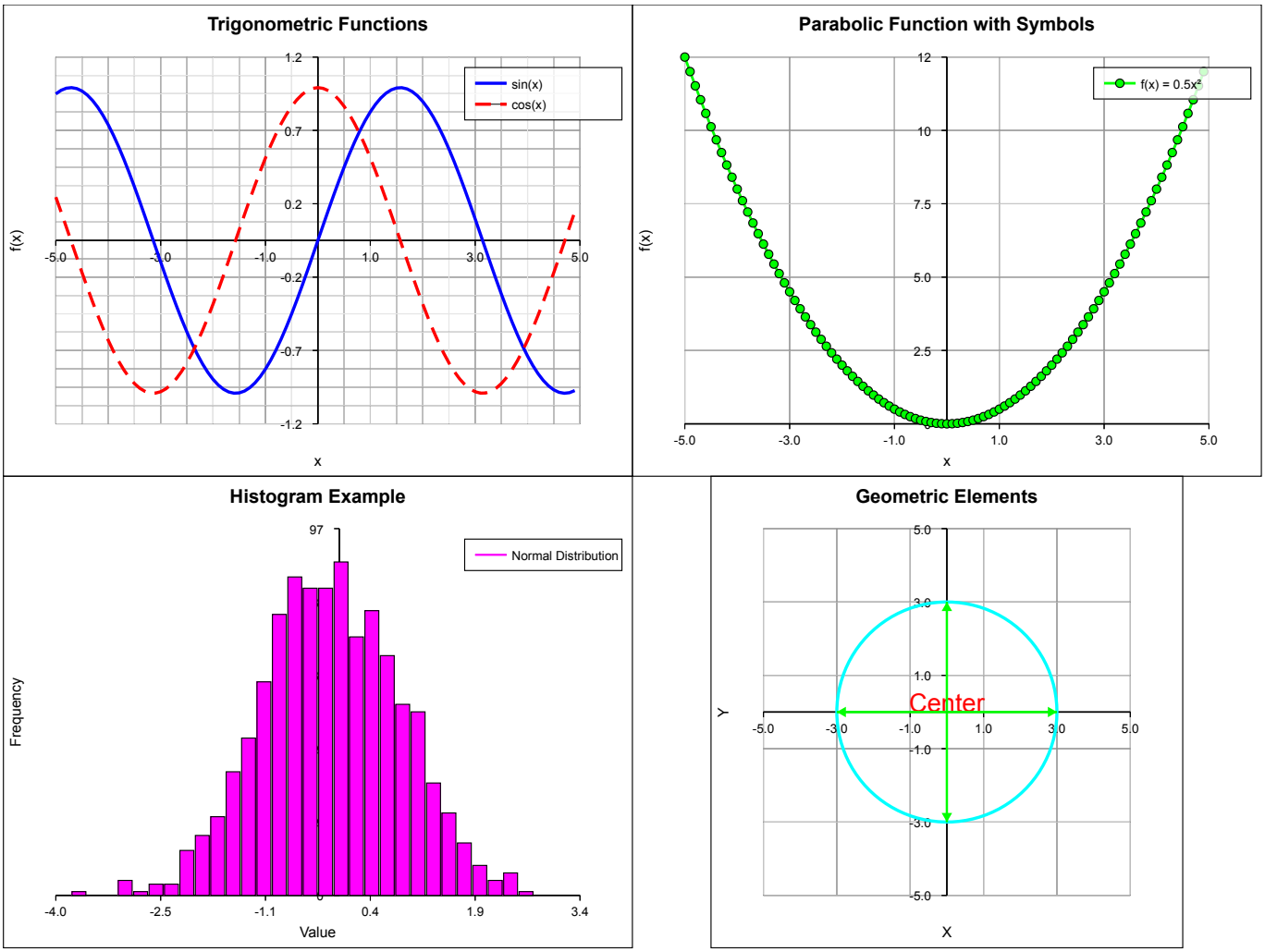
Les graphiques exportés en SVG incluent toutes les fonctionnalités visuelles de la version raster :

- Grilles polaires et cartésiennes avec annotations améliorées (valeurs décimales)
- Courbes, symboles et légendes
- Textes et annotations
- Formes géométriques et flèches

Exemple d'utilisation :

```
// Après avoir créé votre graphique
plt.save_svg("mon_graphique.svg");
```

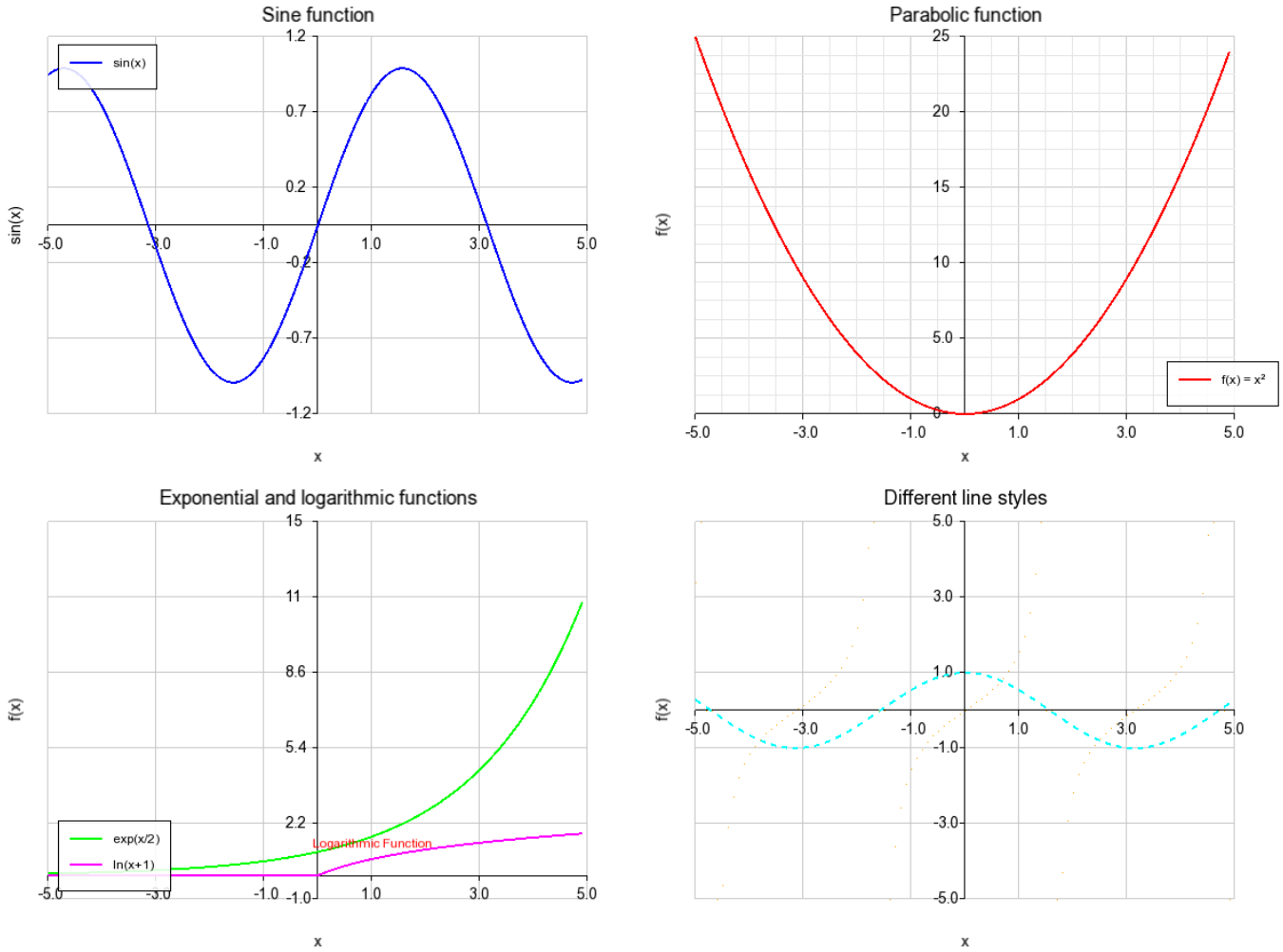
Exemple de sortie SVG



L'exemple ci-dessus montre un graphique exporté en SVG avec la fonction `save_svg()`. Notez la netteté des lignes et du texte, ainsi que la qualité préservée à n'importe quelle échelle d'affichage.

Exemples détaillés

Exemple 1 : Graphiques 2D basiques



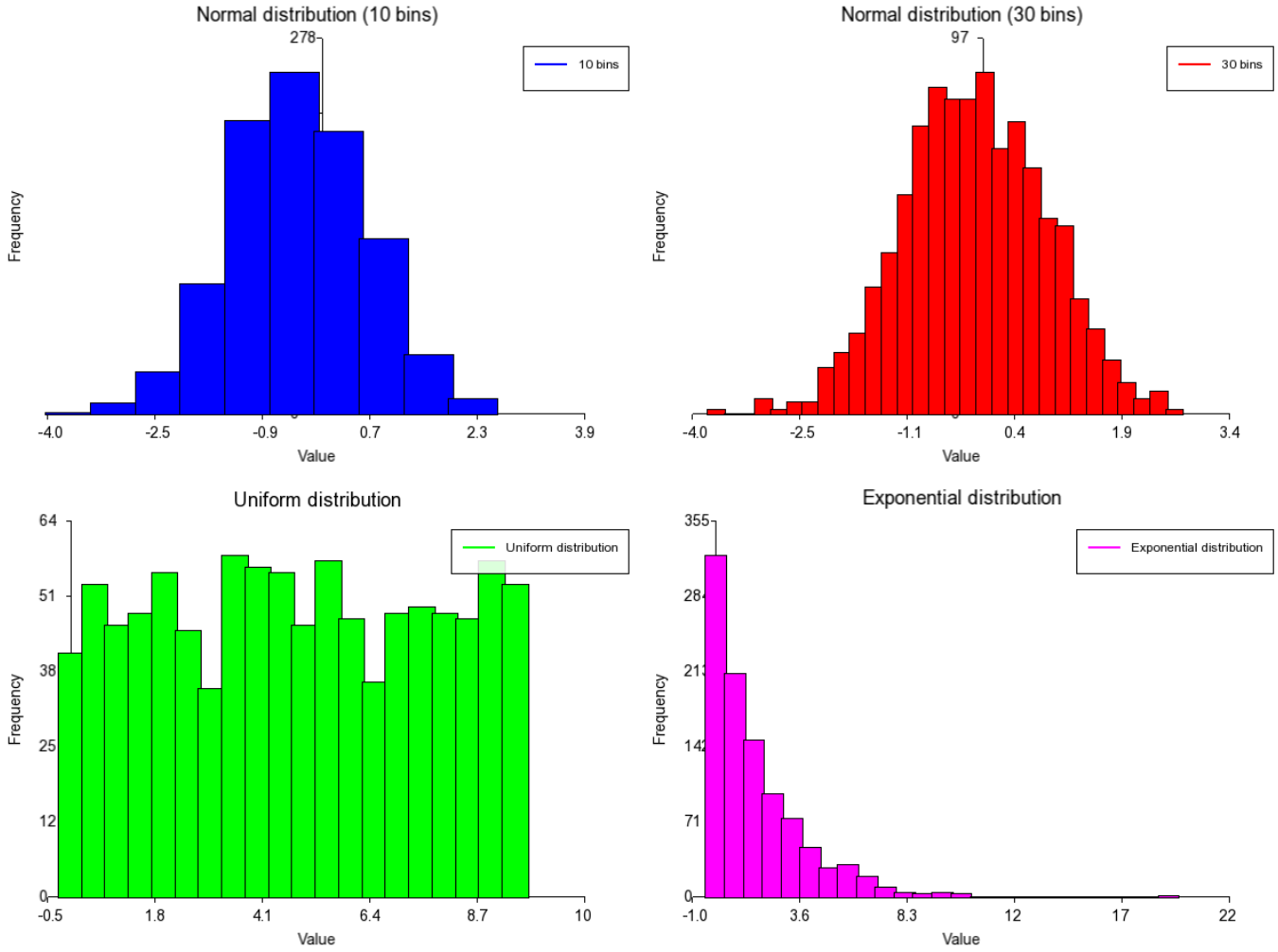
Cet exemple montre comment créer des graphiques 2D simples avec différentes fonctions mathématiques :

- 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



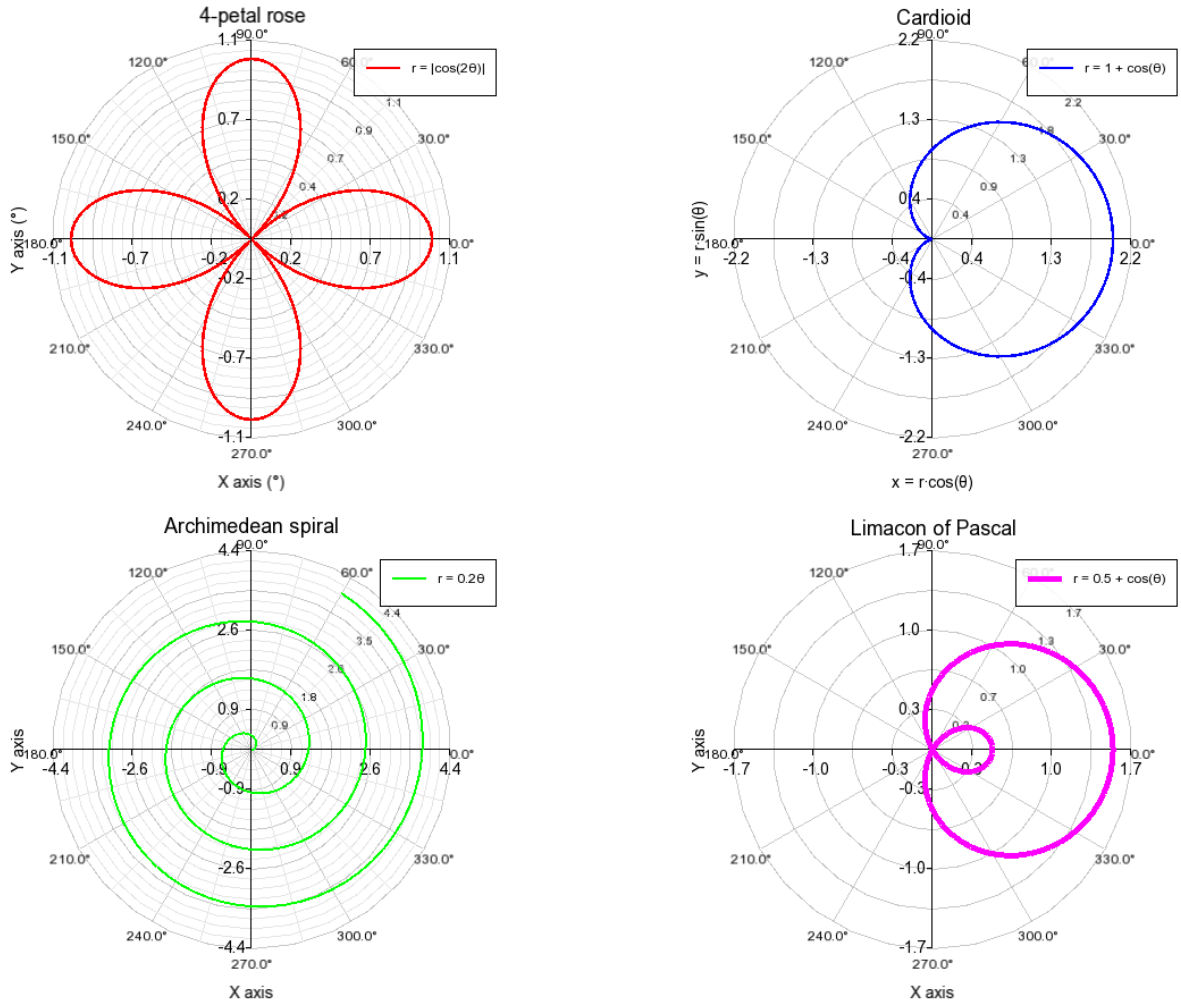
Cet exemple illustre la création d'histogrammes avec différentes distributions :

- 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



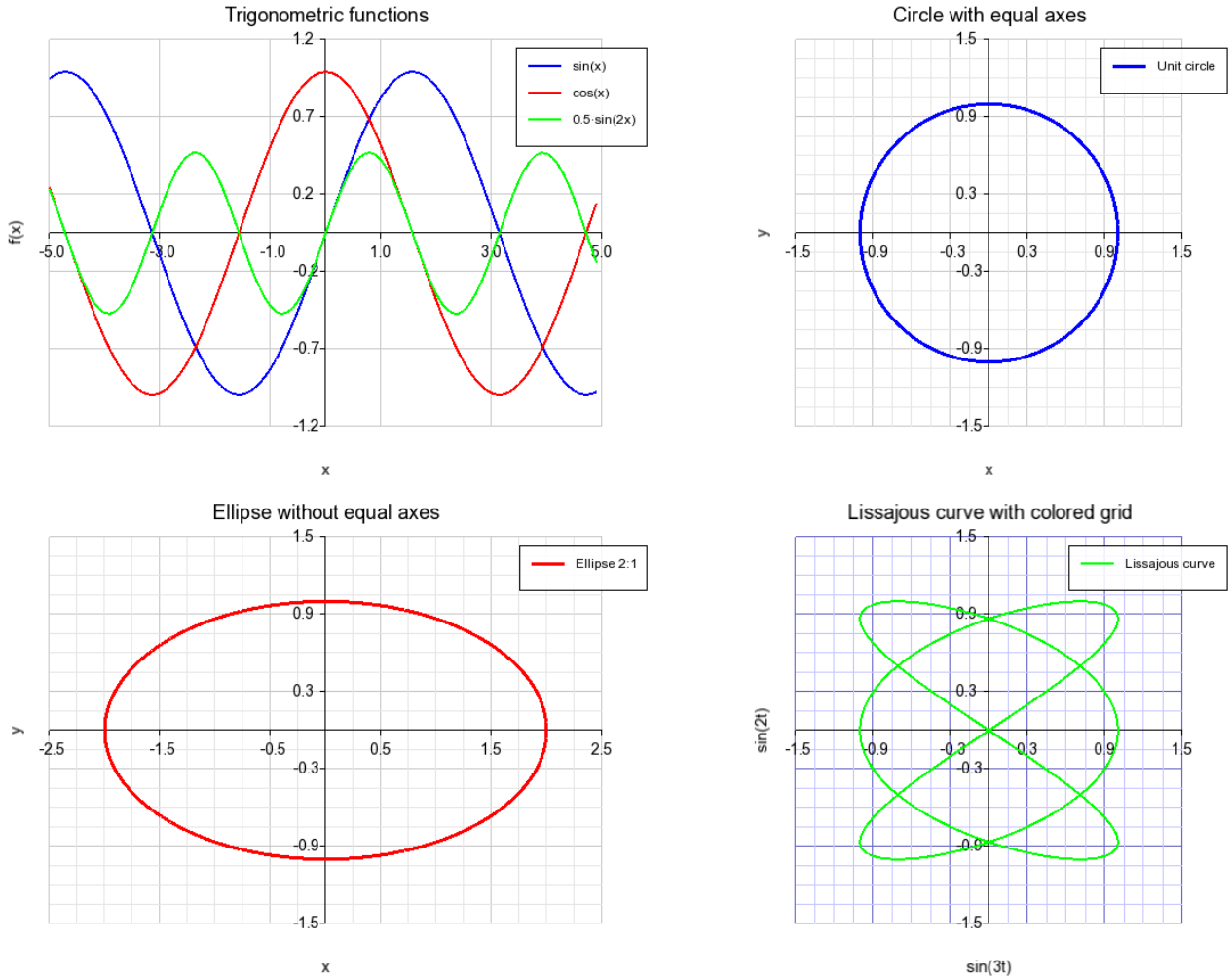
Cet exemple présente des graphiques en coordonnées polaires :

- 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



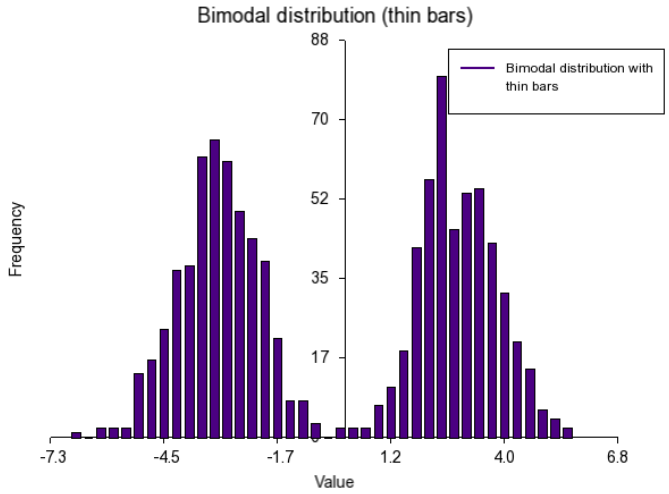
Cet exemple montre des fonctionnalités avancées de personnalisation :

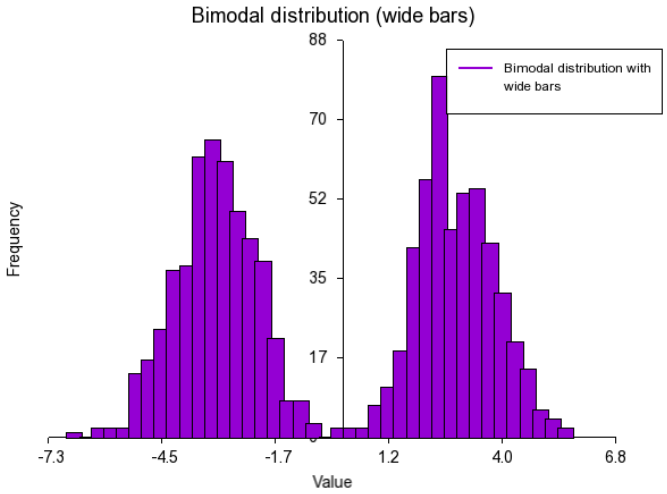
- Plusieurs courbes sur un même graphique
- Axes d'échelle égale pour les cercles
- Personnalisation des couleurs de grille

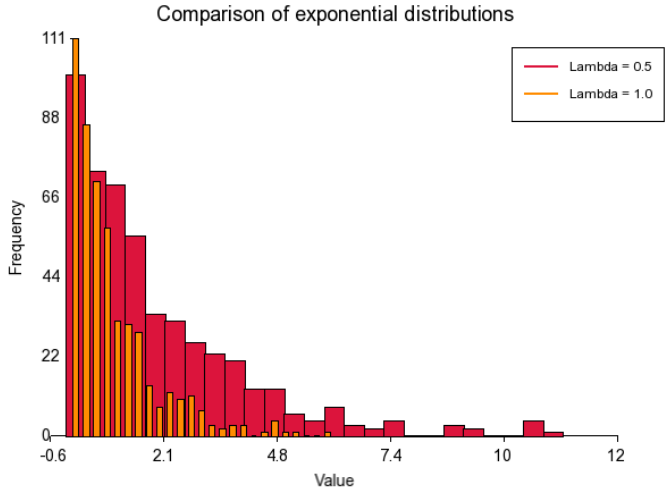
Points clés :

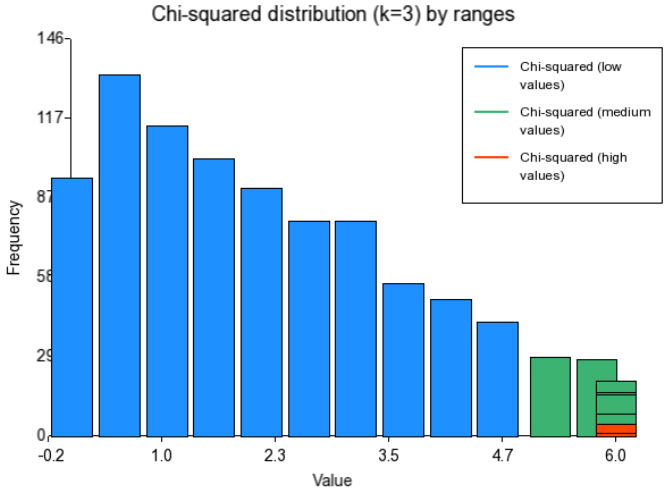
- Superposition de plusieurs courbes
- Options d'axes égaux
- Personnalisation des grilles et couleurs

Exemple 5 : Histogrammes avancés









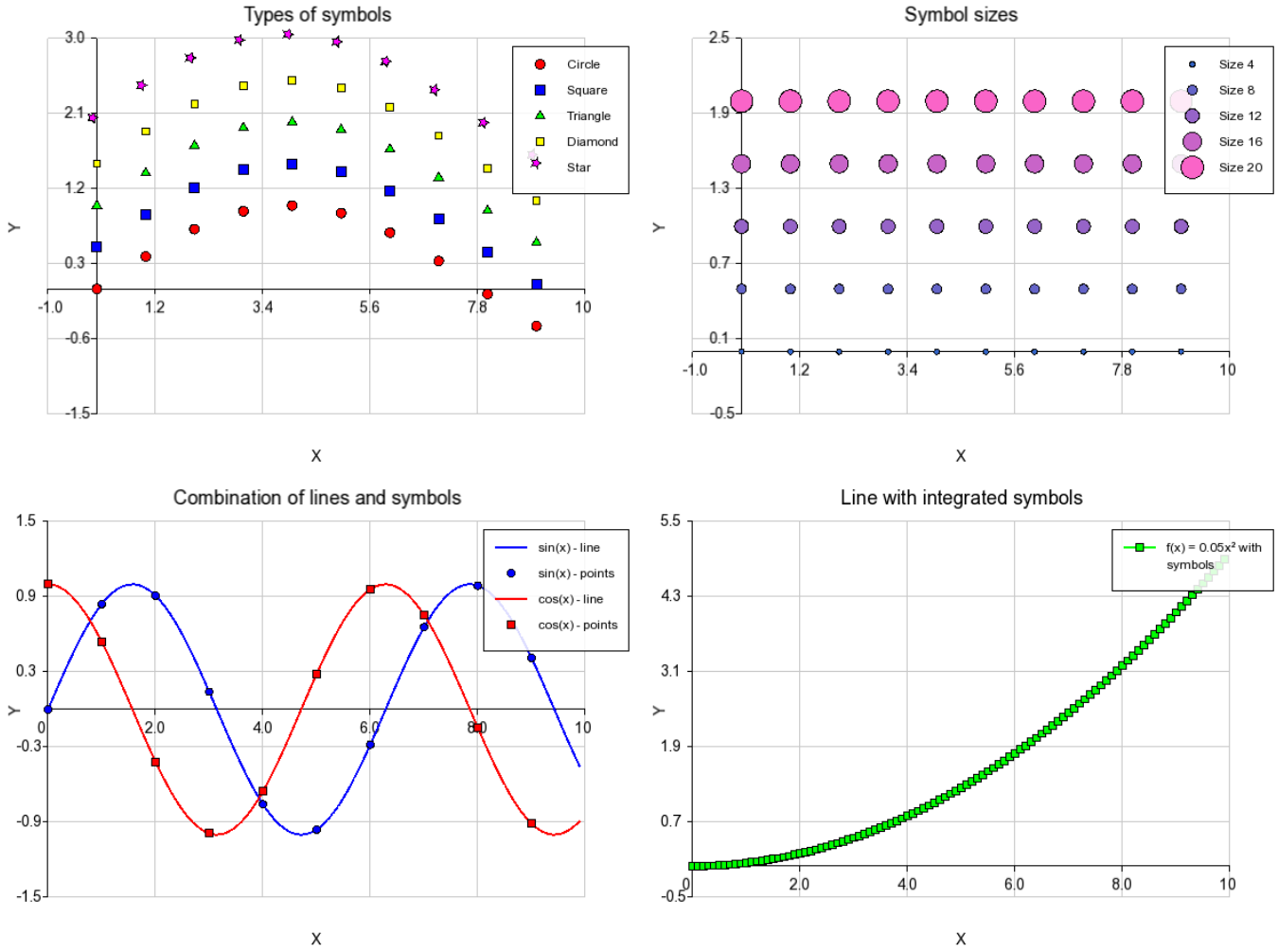
Cet exemple présente des techniques avancées pour les histogrammes :

- 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



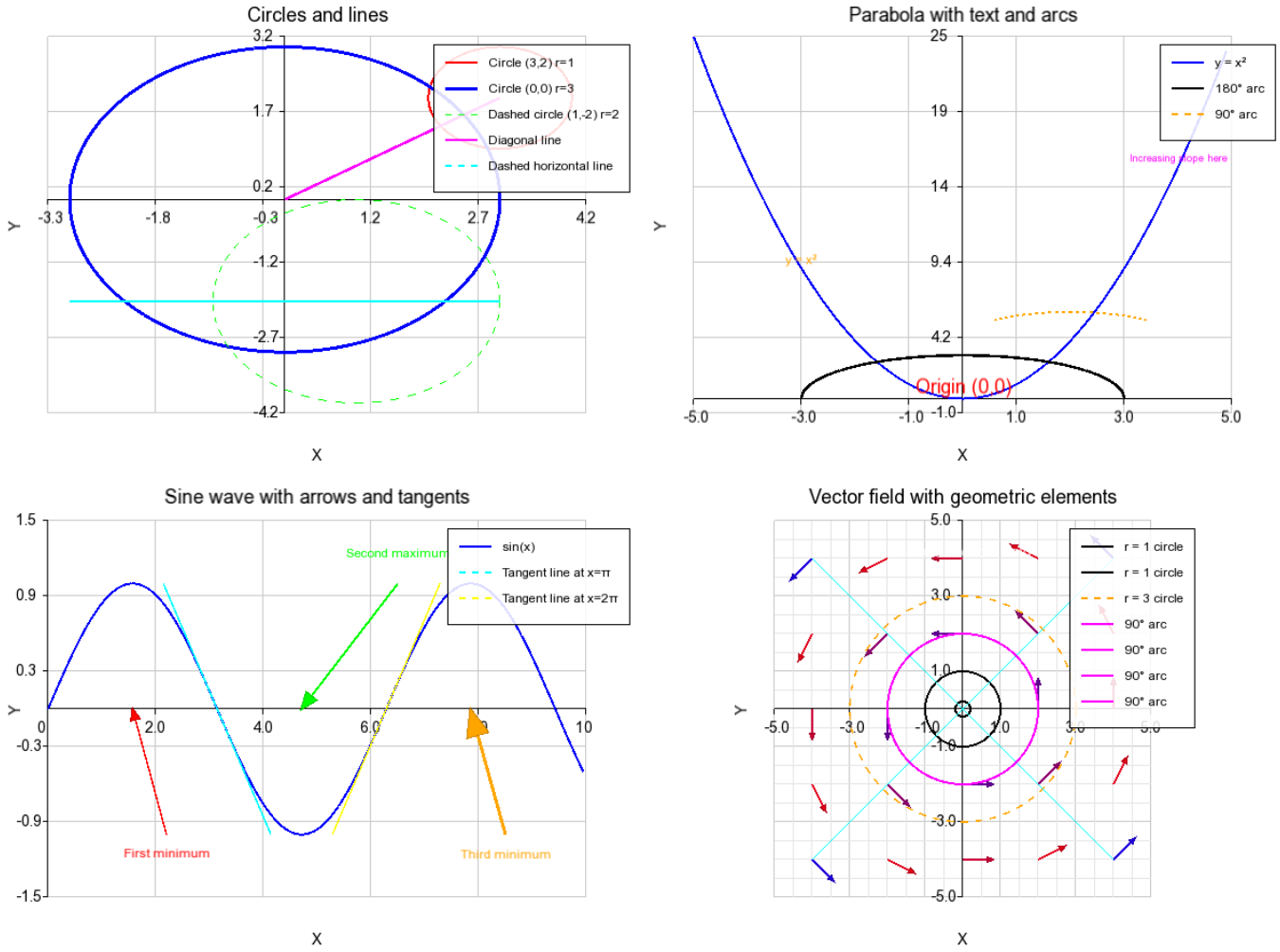
Cet exemple illustre l'utilisation de symboles :

- 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

Exemple 7 : Cercles, Texte et Flèches



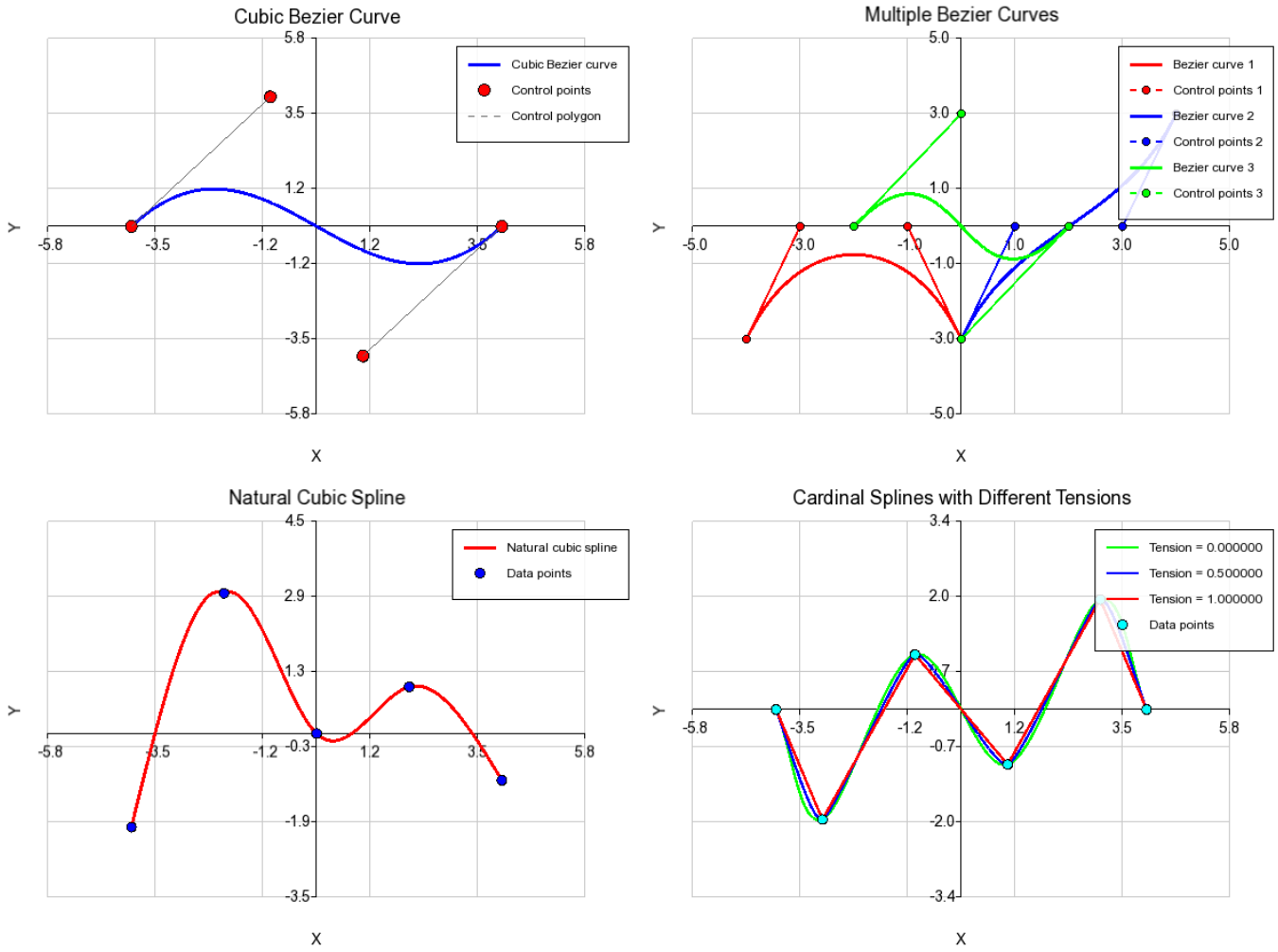
Cet exemple démontre l'utilisation des annotations et formes géométriques :

- Tracé de cercles parfaits avec différents styles
- Ajout de texte explicatif à des positions spécifiques
- Utilisation de flèches pour indiquer des caractéristiques importantes
- Combinaison de lignes et d'arcs pour des constructions géométriques

Points clés :

- Création d'objets géométriques précis
- Annotations claires et lisibles
- Intégration harmonieuse des différents éléments
- Utilisation efficace des styles visuels

Exemple 8 : Courbes de Bézier et Splines



Cet exemple illustre les différentes méthodes d'interpolation de courbes :

- Courbes de Bézier cubiques contrôlées par des points
- Splines naturelles passant par des points de données
- Splines cardinales avec différentes tensions
- Comparaison des différentes techniques d'interpolation

Points clés :

- Compréhension des différences entre les types de courbes
- Contrôle précis de la forme avec les courbes de Bézier
- Interpolation naturelle avec les splines cubiques
- Ajustement de la tension pour les splines cardinales
- Visualisation des points de contrôle et des courbes résultantes

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

Tracé de cercles

```
void circle(Figure& fig, double x0, double y0, double r, const Style& style
= Style())
```

Cette fonction permet de tracer un cercle parfait sur un graphique en spécifiant les coordonnées de son centre et son rayon :

- **x0, y0** : Les coordonnées du centre du cercle
- **r** : Le rayon du cercle
- **style** : Le style visuel du cercle (couleur, épaisseur, etc.)

La fonction automatiquement :

- Assure que le cercle apparaît parfaitement rond (en activant temporairement les axes égaux si nécessaire)
- Ajuste les limites des axes pour afficher correctement le cercle si nécessaire
- Applique le style spécifié au contour du cercle

Exemple :

```
auto& fig = plt.subplot(0, 0);
plt.set_title(fig, "Exemple de cercle");
plt.grid(fig, true, false);

// Tracer un cercle rouge avec centre en (0,0) et rayon 5
PlotGen::Style circle_style;
circle_style.color = sf::Color::Red;
circle_style.thickness = 2.0;
circle_style.legend = "Cercle (0,0) r=5";
plt.circle(fig, 0, 0, 5, circle_style);

plt.show();
```

Lignes, Arcs et Flèches

```
void line(Figure& fig, double x1, double y1, double x2, double y2, const
Style& style = Style())
void arc(Figure& fig, double x0, double y0, double r, double angle1, double
angle2, const Style& style = Style(), int num_points = 50)
```

```
void arrow(Figure& fig, double x1, double y1, double x2, double y2, const
Style& style = Style(), double head_size = 10.0)
```

Ces fonctions permettent d'ajouter des annotations géométriques à vos graphiques :

Ligne

```
void line(Figure& fig, double x1, double y1, double x2, double y2, const
Style& style = Style())
```

Trace une ligne droite entre deux points :

- **(x1,y1)** : Point de départ
- **(x2,y2)** : Point d'arrivée
- **style** : Style visuel de la ligne

Les lignes sont utiles pour :

- Créer des lignes de grille personnalisées
- Ajouter des lignes de référence pour mettre en évidence des seuils
- Dessiner des formes géométriques en combinaison avec d'autres primitives

Exemple :

```
auto& fig = plt.subplot(0, 0);
plt.set_title(fig, "Exemple de lignes");
plt.grid(fig, true, false);
plt.set_axis_limits(fig, -5, 5, -5, 5);

// Tracer des lignes diagonales avec différents styles
PlotGen::Style dash_style;
dash_style.color = sf::Color::Red;
dash_style.thickness = 2.0;
dash_style.line_style = "dashed";
dash_style.legend = "Ligne pointillée";
plt.line(fig, -4, -4, 4, 4, dash_style);

PlotGen::Style solid_style;
solid_style.color = sf::Color::Blue;
solid_style.thickness = 3.0;
solid_style.legend = "Ligne continue";
plt.line(fig, -4, 4, 4, -4, solid_style);

plt.show();
```

Arc

```
void arc(Figure& fig, double x0, double y0, double r, double angle1, double
angle2, const Style& style = Style(), int num_points = 50)
```

Trace un arc (cercle partiel) centré sur un point spécifique :

- **(x0,y0)** : Centre de l'arc
- **r** : Rayon de l'arc
- **angle1** : Angle de départ en degrés (0 = droite, 90 = haut)
- **angle2** : Angle de fin en degrés
- **style** : Style visuel de l'arc
- **num_points** : Nombre de points à générer pour l'arc (plus élevé = plus lisse)

Les arcs sont utiles pour :

- Créer des secteurs circulaires
- Dessiner des indicateurs d'angle
- Construire des formes complexes à partir de primitives géométriques

Exemple :

```
auto& fig = plt.subplot(0, 0);
plt.set_title(fig, "Exemple d'arcs");
plt.grid(fig, true, false);
plt.set_axis_limits(fig, -6, 6, -6, 6);
plt.set_equal_axes(fig, true);

// Tracer d'abord un cercle complet
PlotGen::Style circle_style;
circle_style.color = sf::Color(200, 200, 200);  // Gris clair
circle_style.thickness = 1.0;
plt.circle(fig, 0, 0, 5.0, circle_style);

// Tracer plusieurs arcs autour du cercle
PlotGen::Style arc1_style;
arc1_style.color = sf::Color::Red;
arc1_style.thickness = 3.0;
arc1_style.legend = "0° à 90°";
plt.arc(fig, 0, 0, 5.0, 0, 90, arc1_style);

PlotGen::Style arc2_style;
arc2_style.color = sf::Color::Blue;
arc2_style.thickness = 3.0;
arc2_style.legend = "90° à 180°";
plt.arc(fig, 0, 0, 5.0, 90, 180, arc2_style);

PlotGen::Style arc3_style;
arc3_style.color = sf::Color::Green;
arc3_style.thickness = 3.0;
arc3_style.legend = "180° à 270°";
plt.arc(fig, 0, 0, 5.0, 180, 270, arc3_style);
```

```
PlotGen::Style arc4_style;
arc4_style.color = sf::Color::Yellow;
arc4_style.thickness = 3.0;
arc4_style.legend = "270° à 360°";
plt.arc(fig, 0, 0, 5.0, 270, 360, arc4_style);

plt.show();
```

Flèche

```
void arrow(Figure& fig, double x1, double y1, double x2, double y2, const
Style& style = Style(), double head_size = 10.0)
```

Trace une flèche d'un point à un autre :

- **(x1,y1)** : Point de départ (queue de la flèche)
- **(x2,y2)** : Point d'arrivée (où pointe la tête de flèche)
- **style** : Style visuel de la flèche
- **head_size** : Taille de la tête de flèche en pixels

Les flèches sont utiles pour :

- Indiquer une direction ou un flux
- Mettre en évidence des caractéristiques spécifiques sur un graphique
- Créer des visualisations de champs vectoriels
- Ajouter des annotations avec une emphase directionnelle

Exemple :

```
auto& fig = plt.subplot(0, 0);
plt.set_title(fig, "Exemple de flèches");
plt.grid(fig, true, false);
plt.set_axis_limits(fig, -5, 5, -5, 5);

// Tracer des flèches avec différents styles et tailles de tête
PlotGen::Style arrow1_style;
arrow1_style.color = sf::Color::Red;
arrow1_style.thickness = 2.0;
arrow1_style.legend = "Flèche standard";
plt.arrow(fig, -3, -3, 3, 3, arrow1_style, 10.0);

PlotGen::Style arrow2_style;
arrow2_style.color = sf::Color::Blue;
arrow2_style.thickness = 3.0;
arrow2_style.legend = "Grande tête de flèche";
plt.arrow(fig, 3, -3, -3, 3, arrow2_style, 20.0);
```

```
// Ajouter des étiquettes textuelles
PlotGen::Style text_style;
text_style.color = sf::Color::Green;
plt.text(fig, 3.2, 3.2, "Destination", text_style);
plt.text(fig, -3.2, -3.2, "Origine", text_style);

plt.show();
```

Annotations textuelles

```
void text(Figure& fig, double x, double y, const std::string& text_content,
const Style& style = Style())
```

Cette fonction permet d'ajouter des étiquettes ou annotations textuelles à des coordonnées spécifiques sur votre graphique :

- **x, y** : La position où le texte doit être placé
- **text_content** : Le texte à afficher
- **style** : Le style visuel du texte (couleur, taille, etc.)

Les annotations textuelles sont utiles pour :

- Étiqueter des points de données spécifiques
- Ajouter des explications directement sur le graphique
- Marquer des caractéristiques ou régions importantes
- Inclure des formules mathématiques ou équations

La taille du texte peut être contrôlée en utilisant la propriété `thickness` dans le style :

- Des valeurs d'épaisseur plus élevées créent un texte plus grand
- La valeur par défaut produit un texte lisible qui correspond à l'échelle du graphique

Exemple :

```
auto& fig = plt.subplot(0, 0);
// Tracer des données...

// Ajouter une annotation textuelle à la position (2, 3)
PlotGen::Style text_style;
text_style.color = sf::Color::Blue;
text_style.thickness = 3.0;  // Texte plus grand
plt.text(fig, 2, 3, "Valeur maximale", text_style);

// Ajouter un autre texte avec taille par défaut
plt.text(fig, 4, 1, "Point d'inflexion", PlotGen::Style(sf::Color::Red));

plt.show();
```

Légendes personnalisées

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

Cette fonction permet de contrôler où la légende apparaît sur le graphique. Les positions disponibles sont :

- **"top-right"** (par défaut) : Positionne la légende dans la zone en haut à droite du graphique
- **"top-left"** : Positionne la légende dans la zone en haut à gauche du graphique
- **"bottom-right"** : Positionne la légende dans la zone en bas à droite du graphique
- **"bottom-left"** : Positionne la légende dans la zone en bas à gauche du graphique
- **"outside-right"** : Place la légende à l'extérieur du graphique sur le côté droit Exemple :

```
auto& fig = plt.subplot(0, 0);
// Configuration et tracé...
plt.set_legend_position(fig, "bottom-left");
```

Gestion optimisée des fenêtres

PlotGenC++ propose désormais une gestion améliorée des fenêtres :

- Aucune fenêtre n'est affichée lors de l'utilisation de `save()` sans `show()`
- Tailles de police réduites pour les titres et les textes de légende pour de meilleures proportions
- Titres positionnés en dehors de la zone de tracé pour une visualisation plus claire des données

Exportation d'images

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

Cette méthode permet d'exporter le graphique au format PNG. Les options de qualité et de compression sont gérées automatiquement pour garantir une bonne qualité d'image.

Affichage interactif

```
void show()
```

Cette méthode ouvre une fenêtre interactive pour visualiser le graphique. Elle gère les événements de la fenêtre, permettant de zoomer, déplacer et interagir avec le graphique.

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