Université Paris-Saclay & École 42

Mathématiques et informatique

Les images en mathématiques :
L'indice sous tous les angles

Frédéric Becerril

March 29, 2023

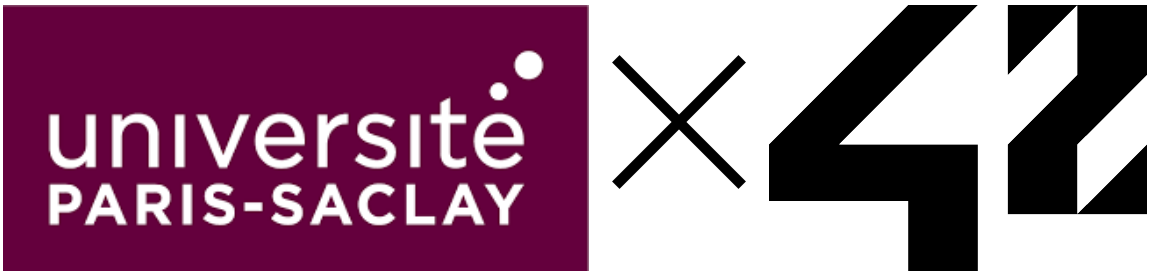


Table des matières

I Introduction

Ma compréhension mathématique, comme celle de beaucoup de monde, passe par les images : de la droite des réels à la continuité des fonctions et leur dérivabilité en passant par la connexité ou la convexité. Plus récemment, le développement de mes compétences en informatique m'a permis de construire des outils de visualisation de plus en plus complexes et perfectionnés. Ces derniers m'aident à intégrer de nouveaux concepts, me poussent à produire des algorithmes de plus en plus élaborés et, par-dessus tout, alimentent mon amour des mathématiques.

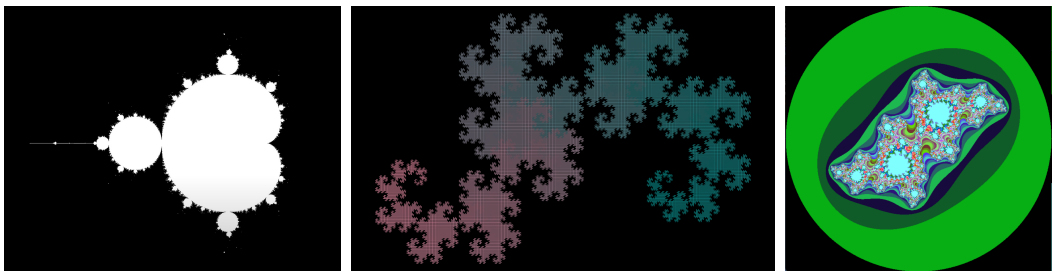


Figure 1: Ce qui m'a fait tomber dans la potion magique

I.1 L'indice et ses applications

Introduit comme une règle de combinatoire par August Ferdinand Möbius en 1865 puis, de nouveau, par James Waddell Alexander II en 1928, l'indice tient une place importante dans de nombreux domaines en mathématiques, mais aussi en physique et en informatique, nous avons essayé dans ce projet de l'illustrer à travers quelques exemples.

Nous les tirerons de l'informatique, où l'indice est notamment utilisé pour calculer l'aire de polygones. On le retrouve dans le code de l'application google maps où il permet aux utilisateurs de calculer l'aire de zones géographiques. Il était aussi, jusqu'à récemment, utilisé en informatique graphique pour déterminer l'appartenance d'un point à un polygone.

I.2 Objectifs du projet

Le principal objectif de ce travail est d'acquérir une intuition fidèle de ce que représente l'indice d'un point pour une courbe sur $\mathbb{R}^2$ (ou $\mathbb{C}$). Pour ce faire, nous ferons appel à différents outils informatiques et mathématiques. Nous produirons la preuve de certains théorèmes importants et détaillerons les algorithmes clés de notre code. Nous finirons par donner une idée visuelle de la preuve du théorème des deux couleurs.

II Méthodes et outils

Ayant peu de connaissances en fonctions holomorphes, nous avons choisi pour ce projet de considérer les lacets comme des fonctions continues de $\mathbb{R}$ dans $\mathbb{R}^2$ et non dans $\mathbb{C}$. Cette approche est strictement équivalente (sauf peut être d'un point de vue conceptuel), la notion de dérivation complexe n'étant pas indispensable aux démonstrations de ce rapport.

En ce qui concerne la programmation, nous avons réalisé le code en C++ à l'aide uniquement de la librairie SDL2. Ce choix est motivé par la vitesse de calcul de ce langage qui nous a permis de produire des animations interactives fluides.

II.1 Mathématiques

II.1.1 Définitions

Définition II.1 (Chemin)**.** On appelle chemin sur $\mathbb{R}^2$ toute application continue

$$\gamma : [0,1] \to \mathbb{R}^2.$$

Définition II.2 (Lacet)**.** On dit d'un chemin γ qu'il est un lacet si $\gamma(0) = \gamma(1)$, autrement dit si son point initial est égal à son point final.

On notera $\mathbb{R}_N[X]$ l'ensemble des polynômes à coefficients réels de degrés au plus N.

Théorème II.1 (Existence et unicité des polynômes interpolateurs)**.** *Soient $x_1, \ldots, x_N$ des réels deux à deux distincts, l'application :*

$$\begin{array}{rccc} \Phi & : & \mathbb{R}_{N-1}[X] & \to & \mathbb{R}^N \\ & & P & \mapsto & \begin{pmatrix} P(x_1) \\ ... \\ P(x_N) \end{pmatrix} \end{array}$$

est bijective.

Démonstration II.1. Nous montrerons la surjectivité plus bas, grâce aux polynômes interpolateurs de Lagrange. Montrons l'injectivité :

Soient $x_1, ..., x_n$ des réels deux à deux distincts et P et Q deux polynômes de degrés inférieurs ou égaux à $N-1$ qui vérifient $(P(x_1), \ldots, P(X_N)) = (Q(x_1), \ldots, Q(X_N))$.

$P - Q$ a N racines par construction, or $\deg(P-Q) = N-1$ donc $P - Q = 0$ par le théorème de d'Alembert-Gauss.

Définition II.3 (Polynômes de Lagrange)**.** Soient $x_1, \ldots, x_n$ des réels deux à deux distincts.

Le i-ème polynôme de Lagrange, noté $L_i \in \mathbb{R}_{N-1}[X]$, pour $1 \leq i \leq N$ est défini par :

$$L_i = \prod_{j \neq i} \frac{X - x_j}{x_i - x_j}$$

Définition II.4 (Polynômes interpolateurs de Lagrange)**.** Soient $x_1, \ldots, x_n$ N réels deux à deux distincts et $y_1, \ldots, y_N$ des réels, le polynôme :

$$P = \sum_{i=1}^{N} y_i L_i$$

est l'unique polynôme de $\mathbb{R}_{N-1}[X]$ qui vérifie $P(x_i) = y_i$, $\forall\ 1 \leq i \leq N$. On l'appelle le polynôme interpolateur de Lagrange.

Démonstration II.2. La démonstration est directe, en effet, $\forall j \neq i$, $L_i(x_j) = 0$ et $L_i(x_i) = 1$.

On note S^1 le cercle unité dans $\mathbb{R}^2$ pour la norme euclidienne.

Théorème II.2 (Théorème de relèvement)**.** *Soient γ un lacet à valeur dans S^1 et $\alpha \in \mathbb{R}$ tel que $\begin{pmatrix}\cos(\alpha)\\ \sin(\alpha)\end{pmatrix} = \gamma(0)$. Il existe un unique chemin θ sur $\mathbb{R}$ tel que $\gamma = \begin{pmatrix}\cos(\theta)\\ \sin(\theta)\end{pmatrix}$ et $\theta(0) = \alpha$.*

Démonstration II.3. Nous démontrerons le théorème du relèvement uniquement dans le cas d'un lacet C^1.

Existence : on notera respectivement a et b les fonctions qui décomposent γ sur la base canonique de $\mathbb{R}^2$. Montrons que :

$$\begin{array}{rccl} \theta & : & [0,1] & \to & \mathbb{R} \\ & & t & \mapsto & \alpha + \displaystyle\int_0^t (b'(u)a(u) - a'(u)b(u))\mathrm{d}u \end{array} \tag{1}$$

convient.

Pour cela montrons que pour tout t, $a(t) = \cos(\theta(t))$ et $b(t) = \sin(\theta(t))$.

On pose $A := (a - \cos(\theta))^2 + (b - \sin(\theta))^2$
$= 2 - 2(a\cos(\theta) + b\sin(\theta))$

Pour tout t :

$A' = 2\left[a\sin(\theta)\theta' - a'\cos(\theta) - b\cos(\theta)\theta' - b'\sin(\theta)\right]$

$= 2\left[\left[a\sin(\theta) - b\cos(\theta)\right]\theta' - a'\cos(\theta) - b'\sin(\theta)\right]$

$= 2\left[\left[a\sin(\theta) - b\cos(\theta)\right](b'a - a'b) - a'\cos(\theta) - b'\sin(\theta)\right]$

Car $\theta' = b'a - a'b$

$A' = 2\left[a^2b'\sin(\theta) - aa'b\sin(\theta) - abb'\cos(\theta) + a'b^2\cos(\theta) - a'\cos(\theta) - b'\sin(\theta)\right]$

$= 2\left[a^2b'\sin(\theta) + b'b^2\sin(\theta) + a^2a'\cos(\theta) + a'b^2\cos(\theta) - a'\cos(\theta) - b'\sin(\theta)\right]$

Car $a^2 + b^2 = 1$ et donc $aa' = -bb'$

$A' = 2\left[\left[a^2 + b^2\right]b'\sin(\theta) + \left[a^2 + b^2\right]a'\cos(\theta) - a'\cos(\theta) - b'\sin(\theta)\right]$

$= 2\left[b'\sin(\theta) + a'\cos(\theta) - a'\cos(\theta) - b'\sin(\theta)\right]$

$= 0.$

Ainsi A est constante et comme $\theta(0) = \alpha$, $\begin{pmatrix}\cos(\theta(0))\\ \sin(\theta(0))\end{pmatrix} = \begin{pmatrix}\cos(\alpha)\\ \sin(\alpha)\end{pmatrix} = \gamma(0) = \begin{pmatrix}a(0)\\ b(0)\end{pmatrix}$

$\cos(\theta(0)) = a(0)$ et $\sin(\theta(0)) = b(0)$. Donc $A = A(0) = \cos(\theta(0)) - a(0) + \sin(\theta(0)) - b(0) = 0$.

Et finalement on a bien $a = \cos(\theta)$ et $b = \sin(\theta)$.

Unicité : soit $\tilde{\theta}$ un autre chemin qui vérifie les hypothèses du théorème, on a $\cos(\theta) = \cos(\tilde{\theta})$.

Or comme θ et $\tilde{\theta}$ sont continues, il existe $k \in \mathbb{Z}$ tq $\theta = \tilde{\theta} + 2k\pi$.

De plus $\theta(0) = \alpha = \tilde{\theta}(0)$

Donc $k = 0 \Rightarrow \theta = \tilde{\theta}$

Définition II.5. Soit X un espace métrique, on dit que X est convexe si les seuls ouverts fermés de X sont X et l'ensemble vide.

Définition II.6. Soit X un espace métrique, un sous ensemble convexe M de X est dit maximal si le seul sous ensemble convexe de X qui contient M est X.

Théorème II.3 (Variante du théorème des deux couleurs)**.** *Soit γ un lacet C^1 sur $\mathbb{R}^2$, on peut colorier les parties connexes maximales de $\mathbb{R}^2 \setminus \gamma$ avec seulement deux couleurs, de façon à respecter la règle suivante :*

Deux parties connexes maximales X_1 et X_2 sont coloriées différemment si elles sont mitoyennes, c'est à dire, s'il existe un point x de γ et un $\epsilon > 0$ tel que pour tout $o < \epsilon' < \epsilon$, la boule de centre x et de rayon ϵ' intersecte X_1 et X_2 mais aucune autre partie convexe maximale de $\mathbb{R}^2 \setminus \gamma$.

II.1.2 Calcul de l'indice au point ω

Soit Γ un lacet C^1 à valeur dans $\mathbb{R}^2 \setminus \{\omega\}$.

On note γ la fonction $\Gamma - \omega$ et $\tilde{\gamma}$ la fonction $\dfrac{\gamma}{||\gamma||}$.

Soient a et b les fonctions réelles qui composent $\tilde{\gamma}$ dans la base canonique et soit $\alpha \in]-\pi, \pi]$ telle que $\tilde{\gamma}(0) = \begin{pmatrix}\cos(\alpha)\\ \sin(\alpha)\end{pmatrix}$

D'après le théorème du relèvement on peut écrire : $\tilde{\gamma} = \begin{pmatrix}\cos(\theta)\\ \sin(\theta)\end{pmatrix}$ avec le θ de la formule 1.

On a donc $\gamma = ||\gamma|| \begin{pmatrix}\cos(\theta)\\ \sin(\theta)\end{pmatrix}$ et finalement $\Gamma = \omega + ||\gamma|| \begin{pmatrix}\cos(\theta)\\ \sin(\theta)\end{pmatrix}$.

On définit l'indice de Γ au point ω par la formule $\dfrac{1}{2\pi}(\theta(0) - \theta(1))$. Cette valeur représente intuitivement le nombre de tours dans le sens trigonométrique qu'un point décrivant le lacet Γ parcourt autour du point ω.

On obtient la formule :

$$Ind_\omega(\Gamma) = \frac{1}{2\pi}(\theta(1) - \theta(0)) = \int_0^t (b'(u)a(u) - a'(u)b(u))du \tag{2}$$

On vérifie facilement que l'indice est toujours un entier : comme $\cos(\theta(0)) = \cos(\theta(1))$, il existe $k \in \mathbb{Z}$ tel que $\theta(0) = \theta(1) + 2k\pi$.

$Ind_\omega(\Gamma) = \frac{1}{2\pi}(\theta(1) - \theta(0)) = k$

II.1.3 Une autre définition de l'indice

Une autre définition de l'indice (que nous développerons moins) s'intéresse à l'orientation du vecteur dérivé de notre lacet.

Soit Γ un lacet C^2 sur $\mathbb{R}^2$, tel que sa dérivée ne s'annule pas et que $\Gamma'(0) = \Gamma'(1)$, alors on définit l'indice de la courbe Γ de la façon suivante : $Ind(\Gamma) := Ind_0(\Gamma')$

Cet indice a de nombreuses propriétés intéressantes, par exemple il est possible de montrer que si Γ_1 et Γ_2 sont deux lacets C^2, il existe une transformation continue entre ces deux courbes si et seulement si $Ind(\Gamma_1) = Ind(\Gamma_2)$.

II.2 Informatique

II.2.1 Fonctions mathématiques usuelles

Nous avons codé des fonctions mathématiques usuelles VII, comme la facorielle, les coefficients binomiaux, la dérivée d'une fonction en un point, la normalisation d'un vecteur, l'intégrale d'une fonction réelle sur un intervalle et la fonction qui, à un complexe, associe son argument.

Nous détaillerons ici seulement le code qui nous permet de calculer nos intégrales pour lequel nous nous sommes servis de la méthode de Simpson composée :

(3)

```
1 double  integrate(double a, double b, std::function<double(double)> f, int N)
2 {
3     double  pas = (b - a) / static_cast<double>(N);
4     double  sum = 0;
5     for (int i = 0 ; i < N ; i++) {
6         sum +=  (f(a + i * pas) + 4 * f(a + (i + 0.5) * pas) + f(a + (i + 1) *
      pas));
7     }
8     return (sum * pas / 6);
9 }
```

Cet algorithme revient à découper notre intervalle en N sous-intervalles avant d'utiliser la méthode de Simpson sur chacun de ces sous-intervalles.

La méthode de Simpson pour approximer l'intégrale d'une fonction sur un segment [a, b] consiste à utiliser l'interpolation de Lagrange sur l'ensemble des points $((a, f(a)), (\frac{a+b}{2}, f(\frac{a+b}{2}), (b, f(b))$ pour obtenir une approximation de notre fonction par un polynôme de degré 2. Enfin nous obtenons notre approximation de l'intégrale en intégrant ce polynôme.

II.2.2 Fonctions mathématiques propres à l'indice

Pour calculer numériquement l'indice d'un lacet dérivable en un point, VII nous avons utilisé la formule 2.

On utilise des lamdas fonctions afin de transformer la fonction γ en la fonction $I = ab' - a'b$, qu'on intègre grâce à la méthode de Simpson 3.

La fonction suivante :

```
1  std::tuple<std::function<double(double)>, std::function<double(double)>>
       shiftFunction(std::function<double(double)> xt, std::function<double(double)>
        yt, Point<double> p) {
2      /* Shift the function to the point p */
3      double  x = p.getX();
4      double  y = p.getY();
5
6      std::function<double(double)> xt_ = {[x, xt](double t) {return (xt(t) - x)
       ;}};
7      std::function<double(double)> yt_ = {[y, yt](double t) {return (yt(t) - y)
       ;}};
8      return (std::make_tuple(xt_, yt_));
9  }
```

décale notre fonction pour se ramener au cas où le lacet ne passe pas par 0.

La fonction suivante:

```
1  std::function<double(double)>   inIntegralFunction(std::function<double(double)>
        xt, std::function<double(double)> yt, Point<double> p) {
2      /* return the function used in the integeal */
3      std::function<double(double)> at;
4      std::function<double(double)> bt;
5      tie(at, bt) = normalizeFunction(xt, yt, p);
6      std::function<double(double)> I = {[at, bt](double t) {return (at(t) *
       derivate(bt, t) - bt(t) * derivate(at, t));}};
7      return (I);
8  }
```

renvoie la fonction I.

II.2.3 Renderer

Pour le renderer nous avons utilisé la librairie SDL2 ainsi que son extension SDL2_GFX.

Nous avons implémenté de nombreuses fonctions de rendu, que nous détaillerons davantage pendant notre soutenance orale, afin de pouvoir mieux les illustrer par leurs animations.

Nous avons également codé plusieurs fonctions de dessin qui n'étaient pas présentes dans cette librairie VII ainsi qu'une classe : "parametric plot" qui automatise l'affichage de fonctions paramétriques VII.

II.2.4 Animations

Les animations sont des boucles infinies avec un event listener que nous avons programmé, là encore, en utilisant la SDL2 VII.

Cela permet de créer des animations en temps réel et interactives. Nous calculons, par exemple, l'indice du point que survole la souris et affichons sa valeur sur l'écran.

Pour visualiser cet indice de façon plus concrète, l'utilisateur peut cliquer où il le désire pour que le programme lance une animation le représentant VII.

II.3 Courbes de Bézier

Une des difficultés pour représenter l'indice est d'arriver à générer des lacets C^1 aléatoires. Nous utilisons, pour ce faire, des courbes de Bézier.

Pour construire une courbe de Bézier, il suffit de choisir un point de départ, un point d'arrivée puis de générer autant de points aléatoires que l'on veut sur l'image.

Plus le nombre de points sera grand, plus la courbe sera complexe VII.

II.3.1 Algorithme

Soit $P = \{P_1, \ldots, P_N\}$ l'ensemble des points devant générer une courbe de Bézier. On commence par relier tous les points par un segment.

On obtient $S = \{S_1, \ldots, S_{N-1}\}$ avec $S_i = [P_i, P_{i+1}]$.

On représente ces segments par des fonctions paramétriques : $F = (F_1, \ldots, F_{N-1})$ où $F_i = P_i \times (1-t) + P_{i+1} \times t$.

Pour tout t, $F(t)$ produit $N-1$ points.

On réitère le procédé sur ce nouvel ensemble de points jusqu'à n'en obtenir plus qu'un, qui sera la valeur de notre courbe de Bézier au point t.

II.3.2 Formule paramétrique

On peut également obtenir une formule décrivant notre courbe de Bézier:

Soit $P = \{P_1, \ldots, P_N\}$ l'ensemble des points devant générer une courbe de Bézier.

$$B(t) = \sum_{i=0}^{N} (1-t)^{N-i-1} \times P_i \times t^i \times \binom{n}{i}$$

Afin d'implémenter cette formule dans notre code, nous avons créé une classe "polynôme" sur laquelle on a surchargé les opérations multiplication par un scalaire, multiplication entre deux polynômes, addition d'une constante et addition entre deux polynômes VII.

II.3.3 Animations

Pour mieux comprendre comment étaient construites les courbes de Bézier, nous avons implémenté une fonction qui anime en temps réel le procédé itératif de construction VII.

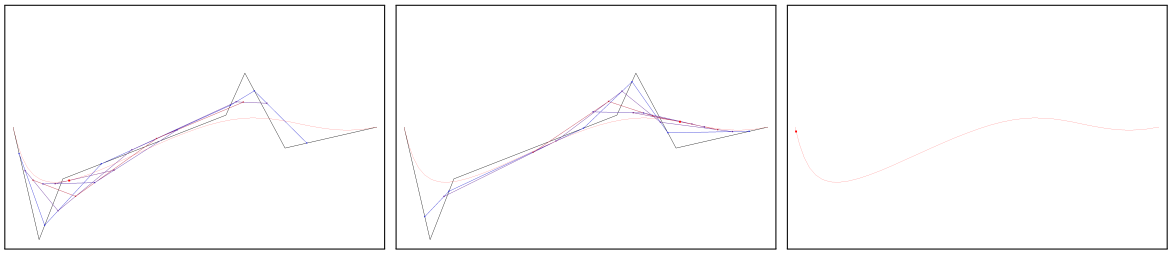


Figure 2: Animation de la construction d'une courbe de Bézier

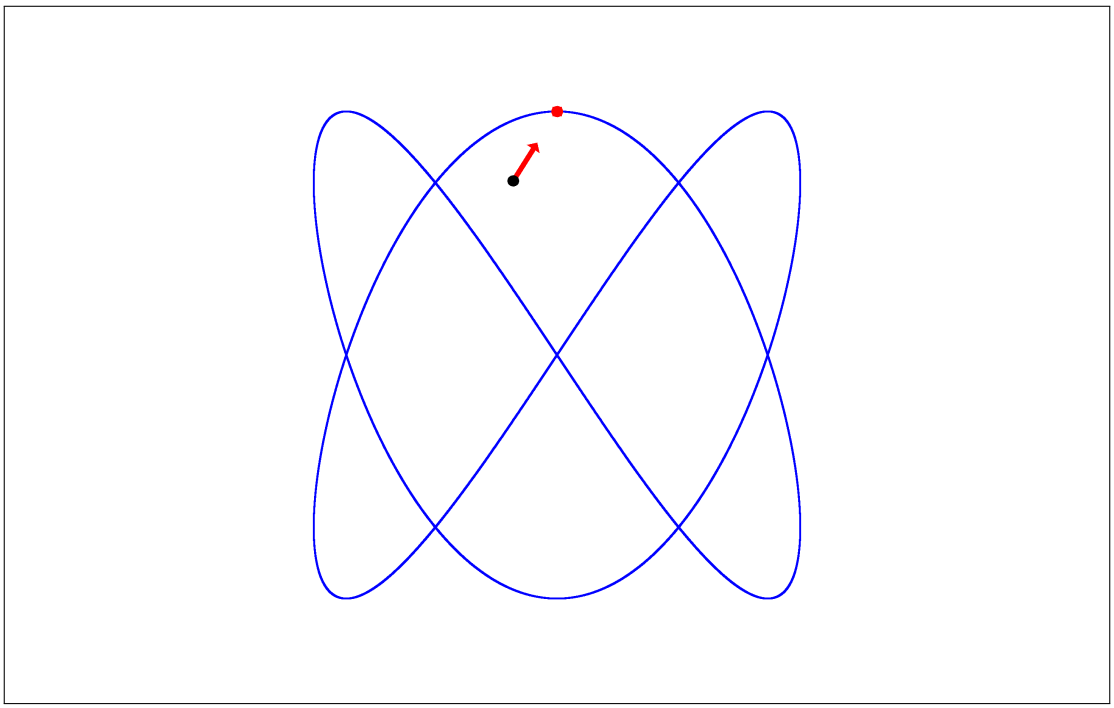
III Visualisation de l'indice

III.1 Courbe de Lissajou

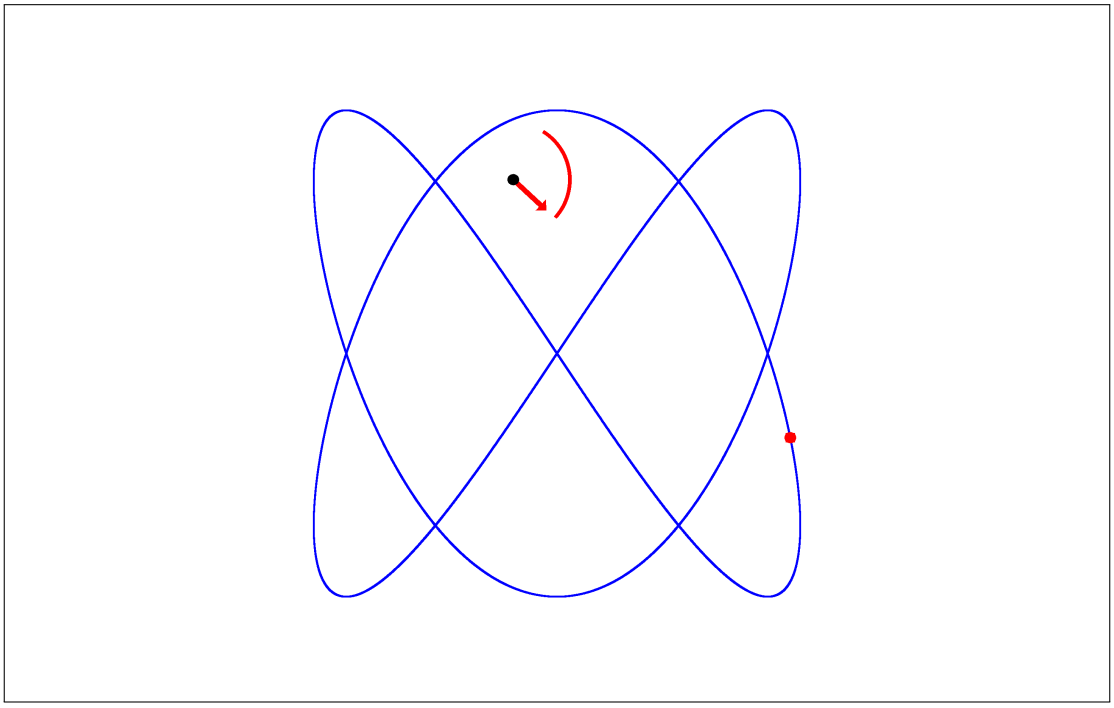
Soit $\gamma := \begin{pmatrix} \cos(2 \times \_) \\ \sin(3 \times \_) \end{pmatrix}$ une courbe de Lissajou de paramètres (2, 3) (cette fonction étant de période 2π, nous l'avons reparamétrée dans notre code).

On peut voir l'indice d'un lacet en ω comme le nombre de tours qu'un point parcourant ce lacet effectue autour d'ω. On compte positivement quand la courbe tourne autour du point dans le sens trigonométrique et négativement quand elle le fait dans le sens horaire.

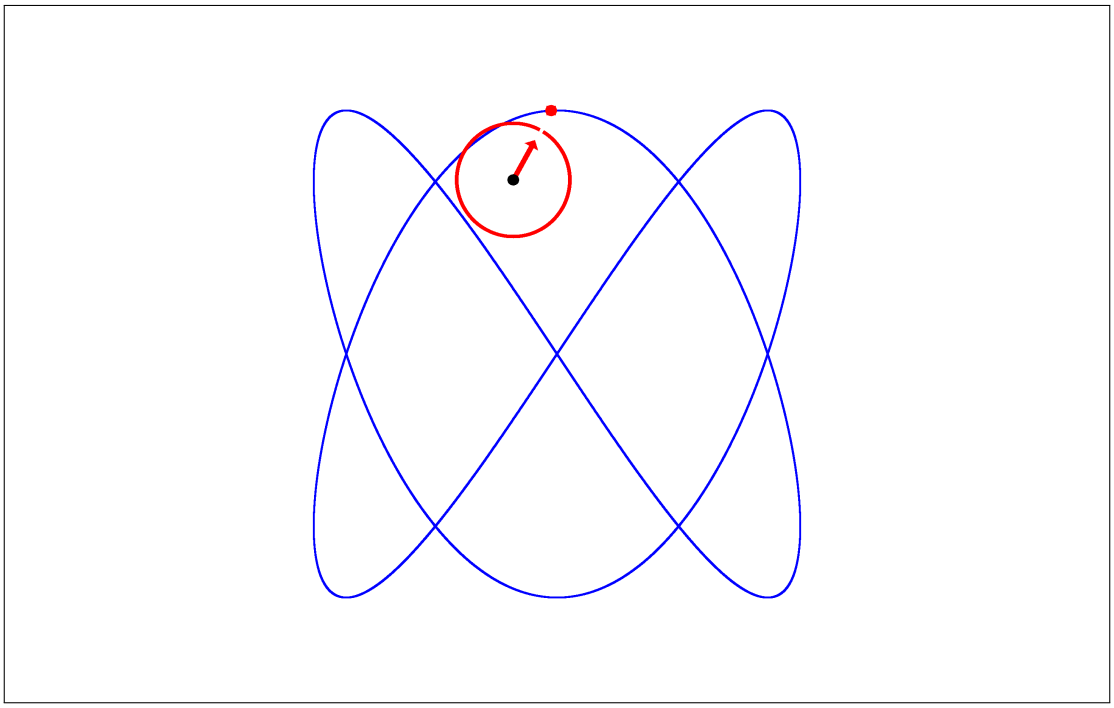
Dans le graphe ci-dessous, on peut voir une flèche partant du point dont on veut calculer l'indice vers le point $\gamma(0)$.



Puis le point se déplace sur la courbe et la flèche le suit faisant progressivement apparaître un cercle qui montre l'angle parcouru depuis $\gamma(0)$.

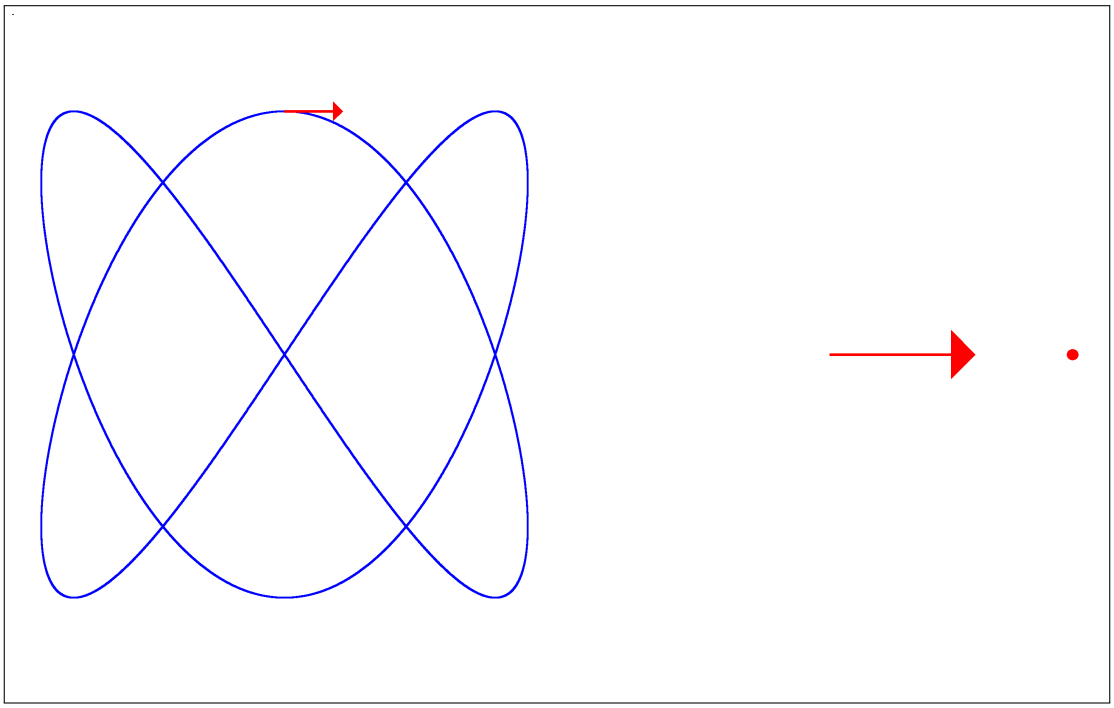


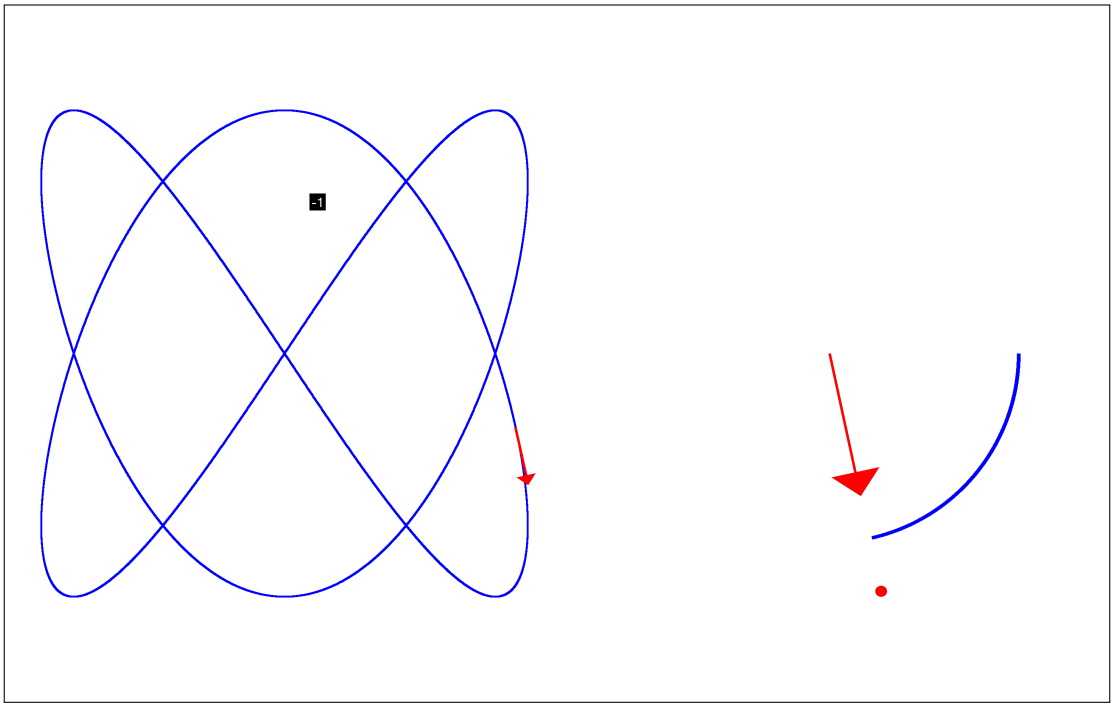
Comme on le voit sur cette dernière figure, quand le point arrive en $\gamma(1)$ (retourne en $\gamma(0)$) un cercle complet finit de se former dans le sens horaire, son indice en ce point est donc -1.

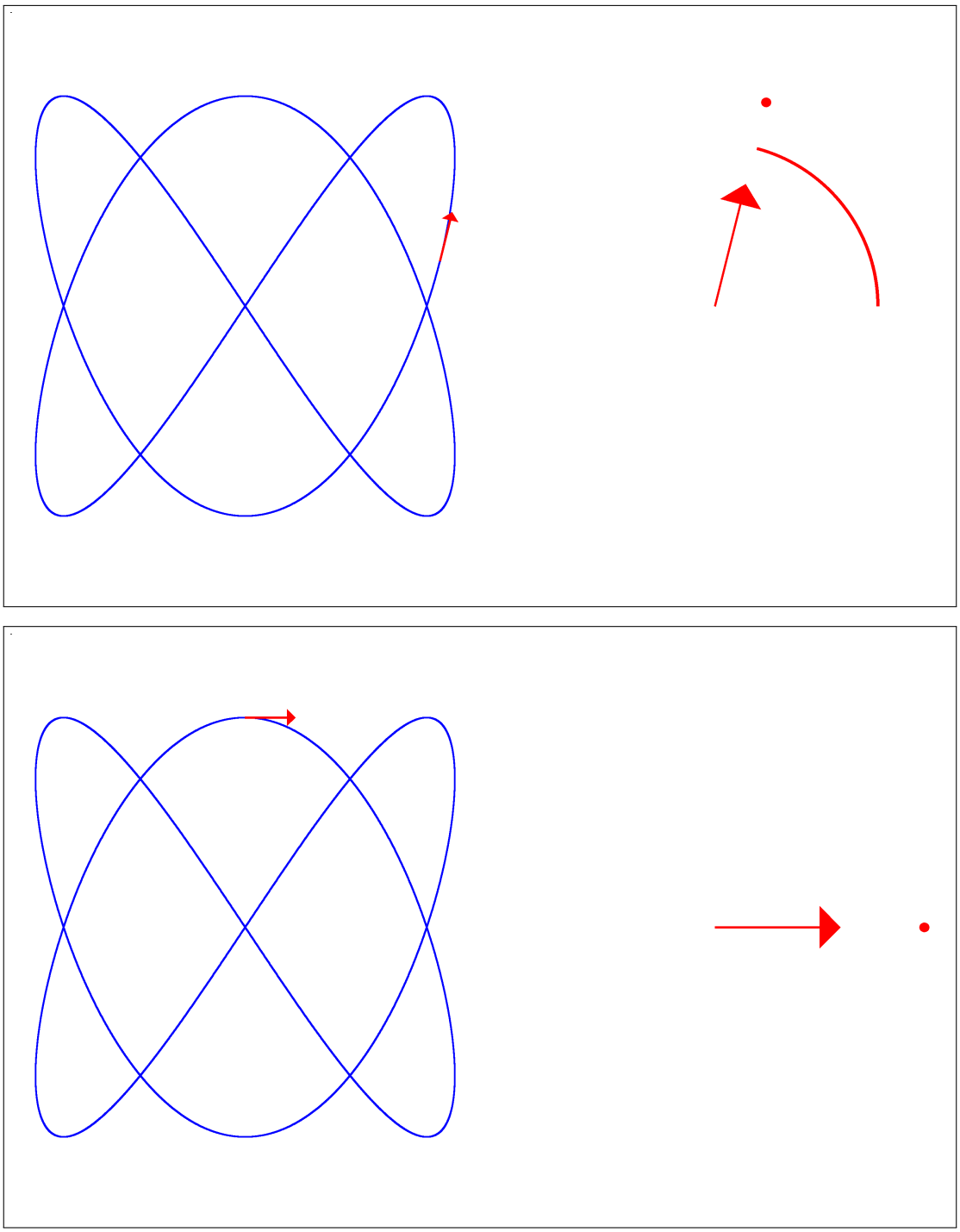


On peut également représenter l'indice de cette courbe:

À droite, on voit l'évolution de l'argument de la dérivée de la courbe, les cercles sont dessinés en bleu quand l'indice est négatif, en rouge sinon



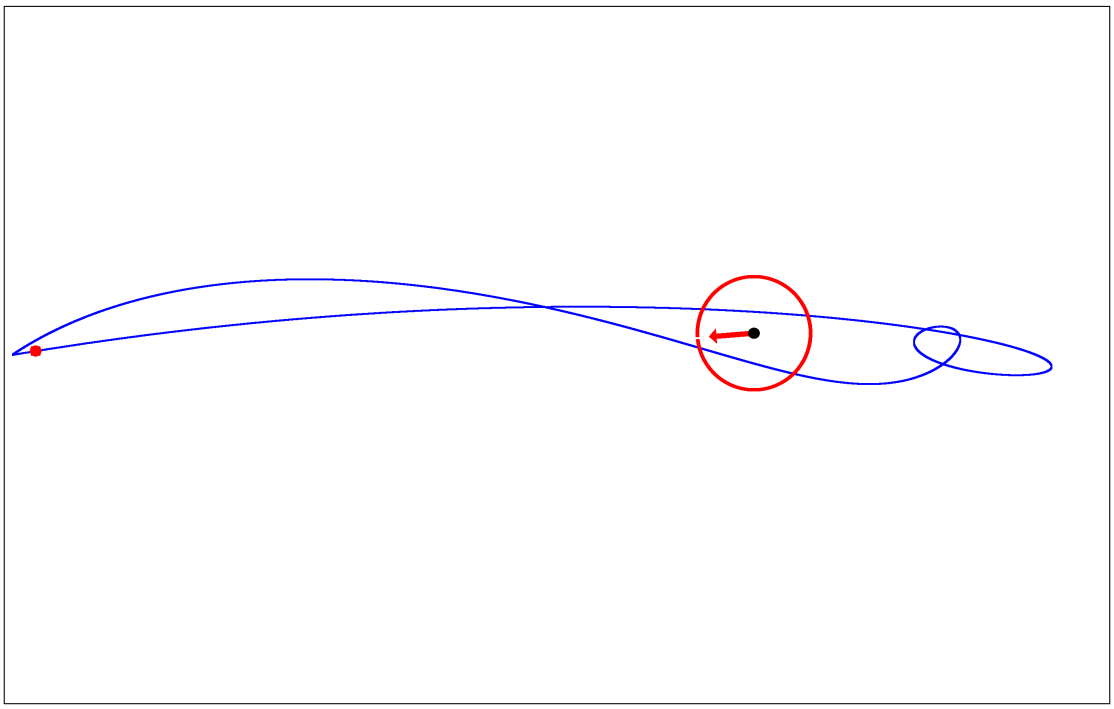




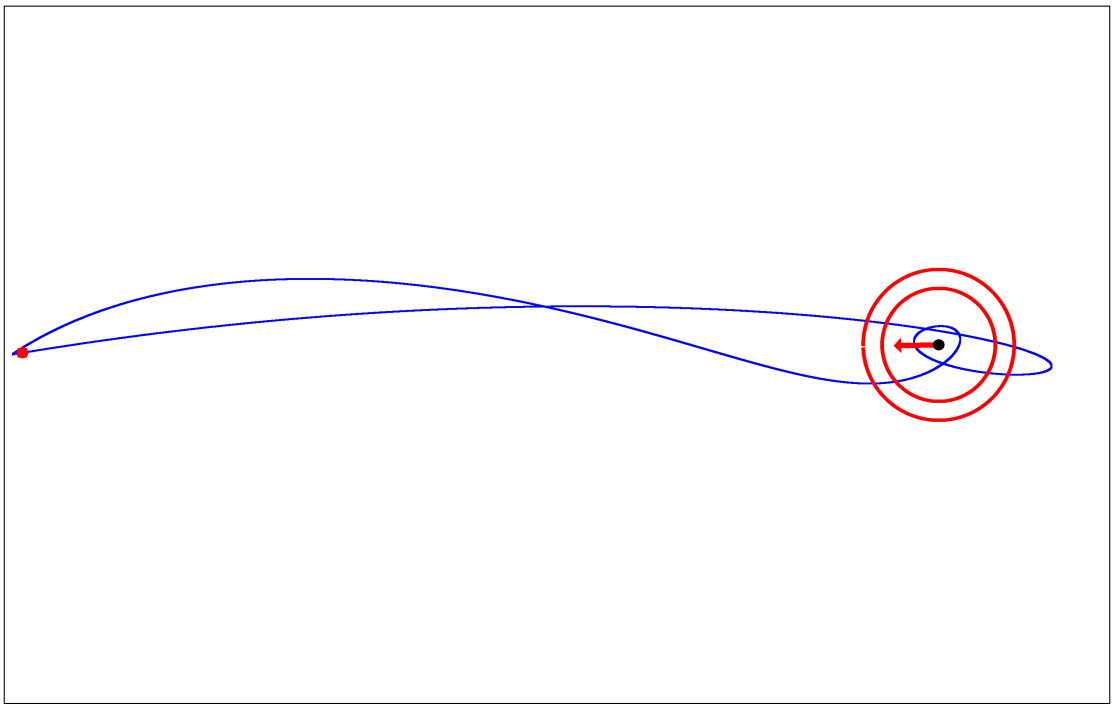
On observe que l'indice de la courbe de Lissajou de paramètre (2, 3) est 0.

III.2 Courbes de Bézier

On regarde maintenant l'indice d'une courbe de Bézier fermée en différents points du plan.



Si l'indice de ce point est supérieur à 1, une animation tracera un nouveau cercle autour des cercles précédents.



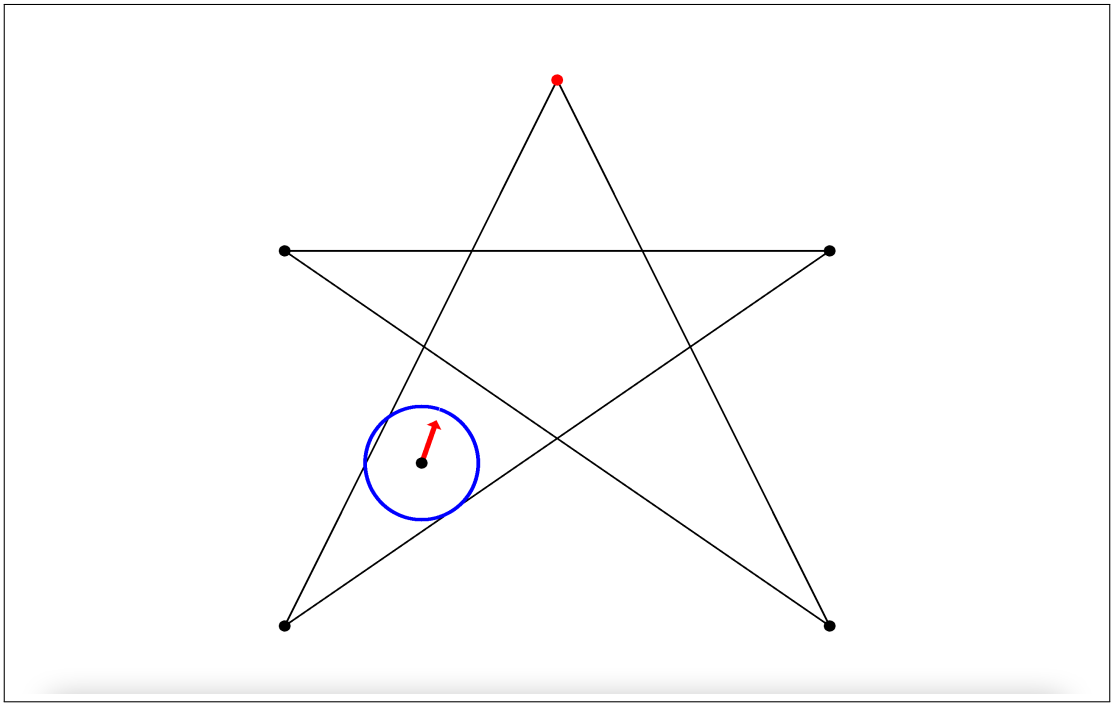
III.3 Cas des polygones

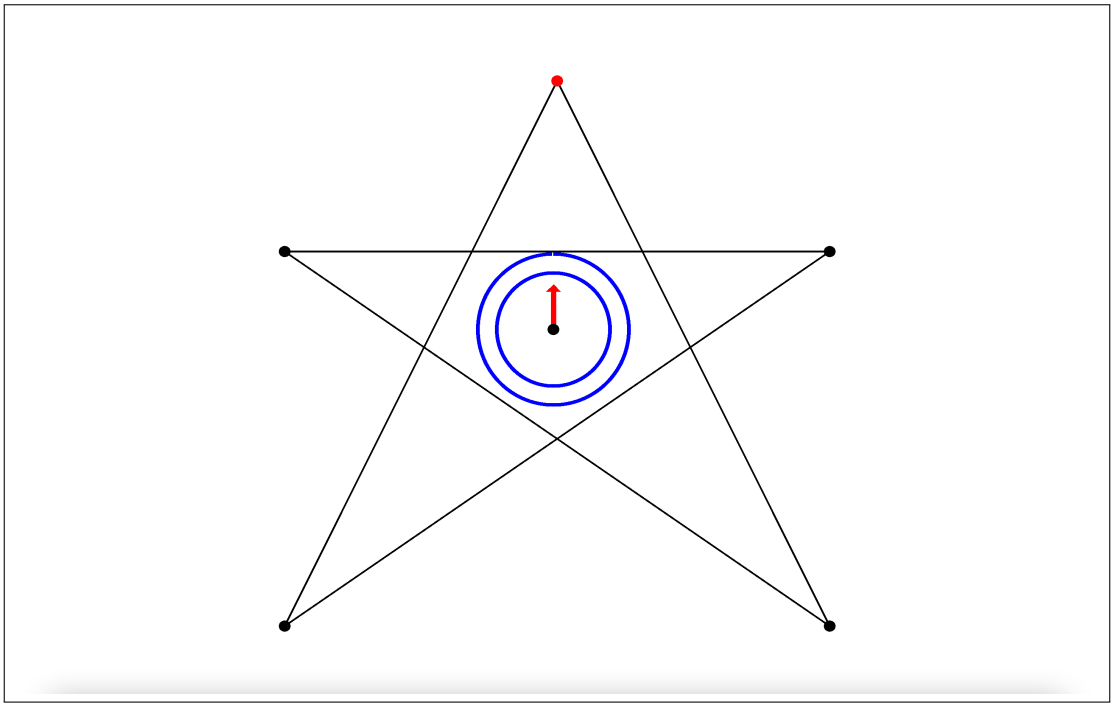
On regarde maintenant l'indice de polygones. Le tracé des polygones n'est pas C^1 mais seulement C^1 par morceaux. Pour calculer l'indice d'un point, on calcule l'indice "partiel" de ce point pour chacun des segments du polygone. L'indice en ce point est la somme de ces indices

partiels.

Pour représenter l'indice d'un polygone, nous avons écrit une fonction qui permet de construire des polygones en choisissant les sommets VII.

Pour cette représentation nous avons utilisé un polygone étoilé.





IV Théorème des deux couleurs

Si l'on remarque que l'indice augmente ou diminue d'1 quand on "traverse" la courbe, il est facile de prouver le théorème des trois couleurs. En effet, il nous suffit simplement de colorier chaque point de notre plan en fonction de la parité de son indice.

Nous avons donc écrit un algorithme permettant, grâce au calcul de l'indice, de colorier le plan avec deux couleurs.

La première version, "naïve", de cet algorithme était très gourmande en temps (30 minutes pour obtenir un coloriage). Elle consistait à simplement recalculer l'indice en tout point, puis à colorier par parité.

Le deuxième algorithme implémenté pour effectuer ce coloriage était dans un premier temps de colorier la courbe, puis de prendre un pixel pas encore colorié, calculer l'indice en ce point et le propager aux pixels adjacents n'étant pas déjà coloriés et de recommencer jusqu'à qu'il n'y est plus de pixel sans couleur.

Nous avons implémenté cet algorithme de deux manières différentes d'une façon récursive d'abord :

```
1  void     colorMapRecursive(std::vector<std::vector<Case>>& map, int x, int y, int
       indice) {
2      if (x < 0 || x >= map.size() || y < 0 || y >= map[x].size()) {
3          return;
4      }
5      if (map[x][y].colored == true) {
6          return;
7      }
8      map[x][y].colored = true;
9      map[x][y].indice = indice;
10     colorMapRecursive(map, x - 1, y, indice);
11     colorMapRecursive(map, x, y - 1, indice);
12     colorMapRecursive(map, x + 1, y, indice);
13     colorMapRecursive(map, x, y + 1, indice);
14 }
```

Mais cette méthode engendrait systématiquement des stack overflow, la récursion devenant vite extrêmement profonde.

Nous avons donc réécrit cet algorithme de façon itérative :

```
1  void     colorMapIterative(std::vector<std::vector<Case>>& map, int x, int y, int
       indice) {
2      std::vector<Point<int>> toColor;
3      toColor.push_back(Point<int>{x, y});
4      while (toColor.size() > 0) {
5          Point<int>  p = toColor.back();
6          toColor.pop_back();
7          if (p.getX() < 0 || p.getX() >= map.size() || p.getY() < 0 || p.getY()
       >= map[p.getX()].size()) {
8              continue;
```

```
9           }
10          if (map[p.getX()][p.getY()].colored == true || map[p.getX()][p.getY()].
        onCurve == true) {
11              continue;
12          }
13          map[p.getX()][p.getY()].colored = true;
14          map[p.getX()][p.getY()].indice = indice;
15          toColor.push_back(Point<int>{p.getX() - 1, p.getY()});
16          toColor.push_back(Point<int>{p.getX(), p.getY() - 1});
17          toColor.push_back(Point<int>{p.getX() + 1, p.getY()});
18          toColor.push_back(Point<int>{p.getX(), p.getY() + 1});
19      }
20  }
```

Cette dernière méthode est beaucoup plus efficace et met en moyenne moins de 5 secondes à render, soit 360 fois moins de temps que l'approche naïve.

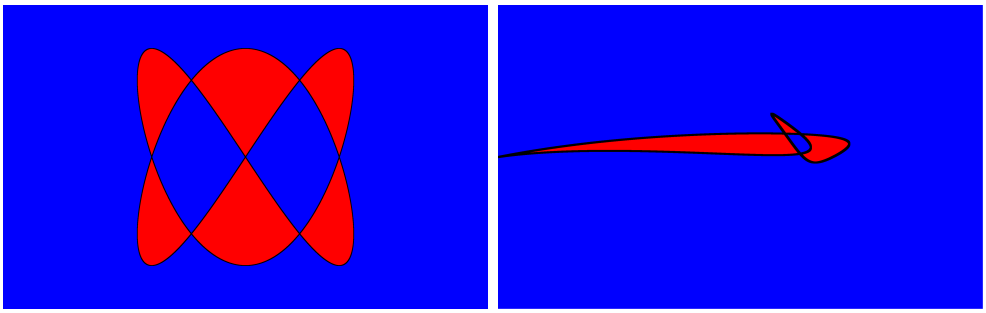
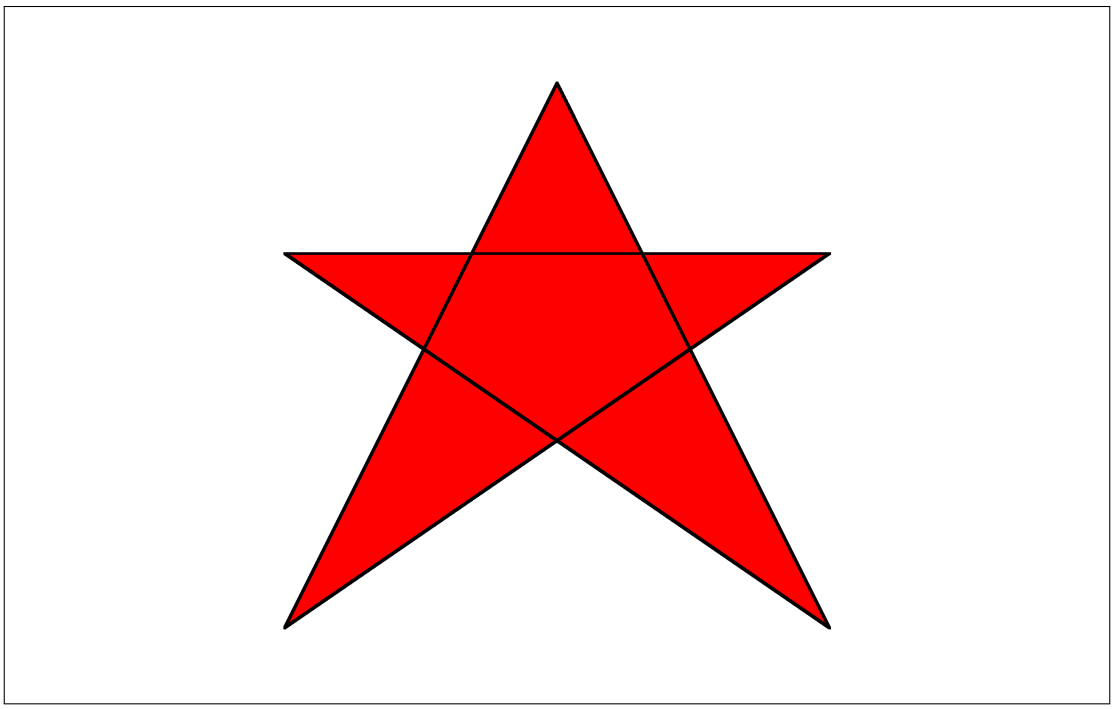


Figure 3: Différents coloriages

V Aire de polygone

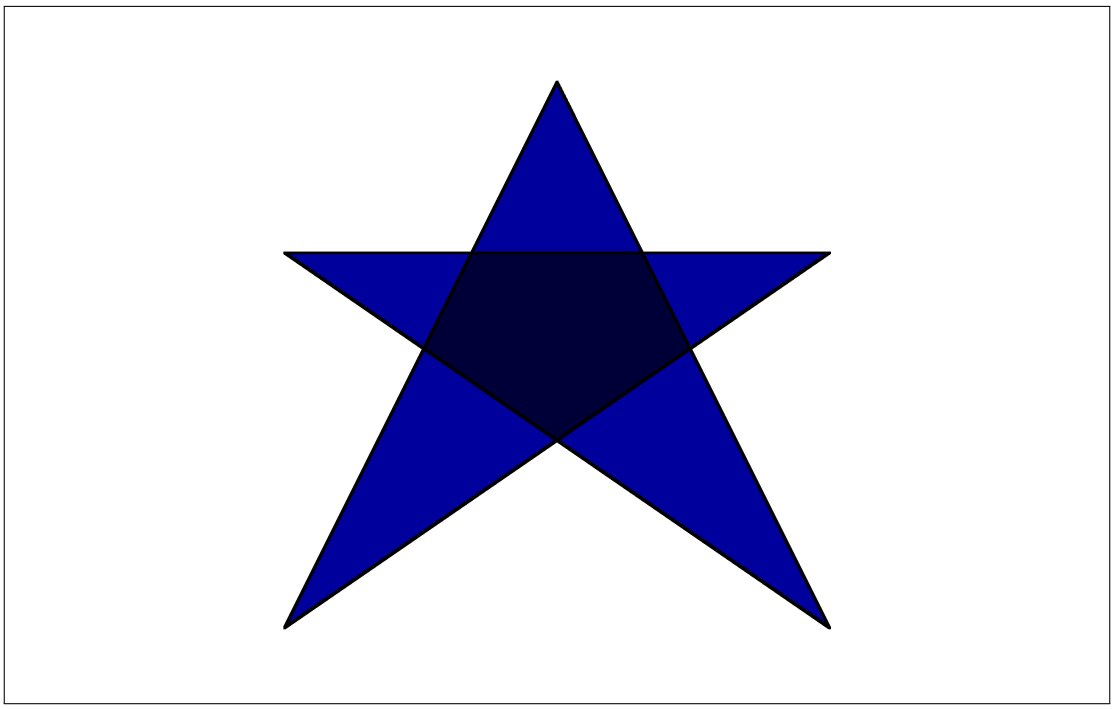
Pour illustrer comment calculer l'aire d'un polygone, nous avons codé un outil similaire à celui de google maps. Il y a plusieurs façons de définir l'aire d'un polygone, une d'elle est de considérer qu'un point est dans le polygone si son indice est non nul.



Ici, les points blancs représentent les points d'indice nul, et les points rouges sont dont l'indice est non nul.

Avec cette méthode, on obtient une aire de : 0.157597

Certains logiciels (comme google maps) vont plus loin. L'air du polygone est définit en comptant un point autant de fois que son indice. Pour expliciter : si un point a un indice de 2 on le comptera deux fois dans le calcul de l'aire, si son indice est -1, il comptera négativement. Selon cette perspective, les polygones peuvent avoir des aires nulles ou négatives.



Ici encore les points blancs représentent les points d'indice nul, et on les points bleu claire ont un indice de -1, et ceux en bleu foncé ont un indice de -2.

Avec cette méthode, on obtient une aire de : -0.201459

VI Conclusion

Aujourd'hui la notion d'indice a été supplantée par la notion de degré, qui la généralise, elle demeure cependant omniprésente en mathématiques ; on la retrouve notamment dans la preuve du retournement une sphère et en analyse complexe. L'indice est aussi très utile pour montrer que deux courbes ne sont pas homotopes (càd telles qu'il existe une application continue de l'une vers l'autre).

VII Annexes

```hpp
#ifndef DRAW_HPP

# define DRAW_HPP

# include <SDL2/SDL.h>
# include <SDL2/SDL_ttf.h>
# include "screen.hpp"
# include "point.hpp"
# include "polynome.hpp"
# include "polygone.hpp"
# include "mathBonus.hpp"
# include <vector>

extern TTF_Font *my_font;

void    drawLine(VirtualScreen& screen, Point<double> p1, Point<double> p2);
void    drawLine(VirtualScreen& screen, Point<double> p1, Point<double> p2, int size, int r, int g, int b);
void    drawLines(VirtualScreen& screen, const std::vector<Point<double>>& points);
void    drawLines(VirtualScreen& screen, const std::vector<Point<double>>& points, int size, int r, int g, int b);
void    drawPoint(VirtualScreen& screen, SDL_Point point, int size = 1);
void    drawPoints(VirtualScreen& screen, std::vector<Point<double>> points, int size);
void    drawArrow(VirtualScreen& screen, double x, double y, double dx, double dy);
void    drawArrowAngle(VirtualScreen& screen, Point<double> p, double angle, int size);
void    drawPointCoord(VirtualScreen& screen, Point<double> p, int x, int y);
void    drawCircleIndice(VirtualScreen& screen, Point<double> p, double angle0, double indice, int radius = 150);
void    drawPolygone(VirtualScreen& screen, Polygone polygone, int size = 1, int r = 0, int g = 0, int b = 0);
void    drawPointPolygone(VirtualScreen& screen, Polygone poly, int size, int r, int g, int b, double t);
void    drawPolygoneOnMap(VirtualScreen& screen, Polygone& poly, std::vector<std::vector<Case>> &map);

#endif
```

fichierHPP/draw.hpp

```hpp
#ifndef INDICE_HPP

# define INDICE_HPP

#include "projet.hpp"
#include "draw.hpp"

void     drawIndice(Info& info, int index);
void     drawIndice(Info& info, double index);
int      calcIndice(std::function<double(double)> xt, std::function<double(double)> yt, Point<double> p);
int      calcIndice(Polygone poly, Point<double> p);
double   calcIndicePart(std::function<double(double)> xt, std::function<double(double)> yt, Point<double> p, double t);
double   calcIndicePart(Polygone poly, Point<double> p, double t);
std::tuple<std::function<double(double)>, std::function<double(double)>> shiftFunction(std::function<double(double)> xt, std::function<double(double)> yt, Point<double> p);
int      calcIndice2(std::function<double(double)> xt, std::function<double(double)> yt);
double   calcIndicePart2(std::function<double(double)> xt, std::function<double(double)> yt, double t);

#endif
```

fichierHPP/indice.hpp

```hpp
#ifndef MATHBONUS_HPP

# define MATHBONUS_HPP

# include <cmath>
# include <functional>
# include <tuple>
# include "point.hpp"

struct Case  {
    int      indice = 0;
    bool     colored = false;
    bool     onCurve = false;
};

double                          fact(int n);
double                          comb(int k, int n);
double                          derivate(std::function<double(double)> f, double t);
std::tuple<double, double>      normalize(double x, double y);
double                          normalizeX(double x, double y);
double                          normalizeY(double x, double y);
double                          integrate(double a, double b, std::function<double(
    double)> f, int N);
double                          polaireAngle(Point<double> p1);
int                             radianToDegree(double radian);
std::tuple<std::function<double(double)>, std::function<double(double)>>
    lineFunction(Point<double> p1, Point<double> p2);
std::tuple<std::function<double(double)>, std::function<double(double)>>
    lineFunction(Point<double> p1, Point<double> p2);
std::function<double(double)>    derivateFunction(std::function<double(double)> f
    );
std::tuple<std::function<double(double)>, std::function<double(double)>>
    shiftFunction(std::function<double(double)> xt, std::function<double(double)>
     yt, Point<double> p);
std::tuple<std::function<double(double)>, std::function<double(double)>>
    normalizeFunction(std::function<double(double)> xt, std::function<double(
    double)> yt);

#endif
```

fichierHPP/mathBonus.hpp

```hpp
#ifndef PARAMETRICPLOT_HPP

# define PARAMETRICPLOT_HPP

# include <functional>
# include "screen.hpp"
# include "mathBonus.hpp"

class PPlot {
    public:
        PPlot(std::function<double(double)> xt, std::function<double(double)> yt);
        void    plot(VirtualScreen &screen, double tStart, double tEnd, int size = 1);
        void    plotOnMap(VirtualScreen &screen, double tStart, double tEnd, std::vector<std::vector<Case>> &map);
        void    showDerivate(VirtualScreen &screen, double t);
        std::function<double(double)>   getXt() const;
        std::function<double(double)>   getYt() const;
    private:
        std::function<double(double)>   xt_;
        std::function<double(double)>   yt_;
};

#endif
```

fichierHPP/parametricPlot.hpp

```hpp
#ifndef POINT_HPP

# define POINT_HPP

# include <vector>
# include <iostream>

template <class T>
class Point {
    public:
        Point();
        Point(T x, T y);
        ~Point();

        void    setPoint(T x, T y);
        friend std::ostream& operator<<(std::ostream &os, const Point<double> &p
    );
        friend std::ostream& operator<<(std::ostream &os, const Point<int> &p);
        bool    operator<(const Point<T>& p) const;
        bool    operator>(const Point<T>& p) const;
        T       getX() const { return (x_); } ;
        T       getY() const { return (y_); };

    private:
        T x_;
        T y_;
};

std::ostream& operator<<(std::ostream &os, const Point<double> &p);
std::ostream& operator<<(std::ostream &os, const Point<int> &p);

#include "point.tpp"

#endif
```

fichierHPP/point.hpp

```cpp
#include "point.hpp"
#include <iostream>

template <class T>
void    Point<T>::setPoint(T x, T y) {
    x_ = x;
    y_ = y;
}

template <class T>
Point<T>::Point(T x, T y) : x_(x), y_(y) {}

template <class T>
Point<T>::Point() : x_(), y_(0) {}


template <class T>
Point<T>::~Point() {}

template <class T>
bool    Point<T>::operator<(const Point<T>& p) const {
    return (this->x_ < p.x_);
}

template <class T>
bool    Point<T>::operator>(const Point<T>& p) const {
    return (this->x_ > p.x_);
}
```

fichierHPP/point.tpp

```hpp
#ifndef POLYGONE_HPP

# define POLYGONE_HPP

# include "point.hpp"
# include <vector>
# include <functional>

class Polygone {
    public:
        Polygone(std::vector<Point<double>> points);
        Polygone();
        ~Polygone();

        void     addPoint(Point<double> point);
        bool     isClosed();

        std::vector<Point<double>>  getPoints() const;
    private:
        std::vector<Point<double>>  points_;
        bool                        closed_;
};

#endif
```

fichierHPP/polygone.hpp

```hpp
#ifndef POLYNOME_HPP

# define POLYNOME_HPP

# include "point.hpp"
# include <vector>
# include <iostream>
# include <tuple>
# include <string>

class Poly {
    /*
    **  Classe polynome
    **  Coefficient stocke dans ordre croissant de degree coefs_[0] = a * x^0
    */
    public:
        Poly();
        Poly(std::string file, int c);
        Poly(size_t deg, double val);
        Poly(std::vector<double> coefs);
        ~Poly();
        Poly    operator*(const double& a) const;
        Poly    operator*(const Poly& p) const;
        Poly    operator+(const double& a) const;
        Poly    operator+(const Poly& p) const;
        double  operator()(double x) const;
        void    savePoly(std::string name);
        friend std::ostream& operator<<(std::ostream &os, const Poly &p);

    private:
        std::vector<double> coefs_;
};

Poly    puissance(const Poly& p, const int& a);
std::ostream& operator<<(std::ostream &os, const Poly &p);
std::tuple<Poly, Poly>  BezierPoly(std::vector<Point<double>> points);

#endif
```

fichierHPP/polynome.hpp

```cpp
#ifndef PROJET_HPP

# define PROJET_HPP

# include <SDL2/SDL.h>
# include <SDL2/SDL_timer.h>
# include <vector>
# include <functional>
# include "draw.hpp"
# include "timer.hpp"
# include "point.hpp"
# include "polynome.hpp"
# include "screen.hpp"
# include "parametricPlot.hpp"

class Info {
    public:
        Info(bool fullScreen = true);
        ~Info();
        void            addVirtualScreen(Flag f);
        VirtualScreen   *getCurrentScreen() const;
        void            selectScreen(int index);
        Timer           &getTimer();
        void            hideOrShowAlgo(bool show_);
        void            setMouseInfo(int x, int y, bool m);
        void            setClickInfo(int x, int y, int c);
        bool            showAlgo() const;
        int             getMouseX() const;
        int             getMouseY() const;
        int             getHighDPI() const;
        bool            getMouseMooved() const;
        int             getClicked() const;
        bool            getColorMap() const;
        void            setColorMap(bool b);

        std::vector<Point<double>>      getBezierPoints() const;
        void                            setBezierPoints(std::vector<Point<double
    >> points);
        std::function<double(double)>   getXT() const;
        std::function<double(double)>   getYT() const;
        void                            setXT(std::function<double(double)> f);
        void                            setYT(std::function<double(double)> f);

    private:
        Screen                          S;
        std::vector<VirtualScreen>      Vscreens;
        std::vector<Point<double>>      bezierPoints;
        VirtualScreen                   *screen;
        Timer                           timer;
        bool                            show;
        int                             mouse_x;
        int                             mouse_y;
        bool                            mouseMooved;
        int                             click;
```

```
54          bool                            colorMap;
55
56          std::function<double(double)>    xt;
57          std::function<double(double)>    yt;
58  };
59
60  void    bezierCurveLoop();
61  void    bezierCurveLoopGFX();
62  void    lissajousCurveLoop(int m, int n);
63  void    lissajousCurveLoop2(int m, int n);
64  void    bezierCurveCloseLoop();
65  void    polygoneBuilder();
66  void    indiceAnimationLoop(Info& info, Point<double> p);
67  void    indiceAnimationLoopPolygone(Info& info, Point<double> p, Polygone poly);
68  void    twoColorMapLoop(Info& info);
69  void    twoColorMapLoopImprove(Info& info);
70  void    representationAire(Info& info, Polygone &poly);
71
72  bool    handleEvent(std::function<void(Info& info)> f1, std::function<void(Info&
        info)> f2, Info& info);
73  void    noneFunction(Info& info);
74  void    hideOrShowAlgo(Info& info);
75  void    changeBezierPoint(Info& info);
76  void    launchColorMap(Info& info);
77
78  bool    findPointNotColored(std::vector<std::vector<Case>>& map, int&x, int &y);
79  void    colorMapIterative(std::vector<std::vector<Case>>& map, int x, int y, int
        indice);
80
81
82  std::vector<Point<double>>  generateBezierPoints(Point<double> p1, Point<double>
        p2, int n, double y_min = -1, double y_max = 1);
83  std::vector<Point<double>>  generateBezierPointsLoop(Point<double> p1, int n,
      double y_min = -1, double y_max = 1);
84  std::vector<Point<double>>  generateNextPoints(std::vector<Point<double>> p,
      double t);
85  std::tuple<std::function<double(double)>, std::function<double(double)>>
      bezierCurvePoly(std::vector<Point<double>> bezierPoints);
86
87  # if SDL_BYTEORDER == SDL_BIG_ENDIAN
88      #define rmask 0xff000000
89      #define gmask 0x00ff0000
90      #define bmask 0x0000ff00
91      #define amask 0x000000ff
92  # else
93      #define rmask 0x000000ff
94      #define gmask 0x0000ff00
95      #define bmask 0x00ff0000
96      #define amask 0xff000000
97  # endif
98
99  #endif
```

fichierHPP/projet.hpp

```hpp
#ifndef SCREEN_HPP

# define SCREEN_HPP

# include "point.hpp"
// # include "polynome.hpp"

# include <SDL2/SDL.h>
# include <SDL2/SDL_timer.h>

enum struct Flag {
    left,
    right,
    up,
    down,
    full,
};

class  Screen {
    friend class VirtualScreen;
    private:
        SDL_Window      *window;
        SDL_Renderer    *render;
        int             window_w;
        int             window_h;
        int             highDPI;
        double          ratio;

    public:
        Screen(bool fullScreen = true);
        ~Screen();
        double          getRatio() const;
        int             getWindowW() const;
        int             getWindowH() const;
        int             getHighDPI() const;
        SDL_Renderer    *getRenderer() const;
};


class VirtualScreen {
    public:
        VirtualScreen(const Screen &screen, Flag flag);
        ~VirtualScreen();
        void            createPlan(Point<double> planUL, Point<double> planDR);
        Point<int>      convPoint(Point<double> p);
        Point<double>   convPoint(Point<int> p);
        SDL_Point       convPointSDL(Point<double> p);

        void        startDraw();
        void        renderPresent();

        int         getVirtualW() const;
        int         getVirtualH() const;

```

```
55          SDL_Renderer    *getRenderer() const;
56          double          getRatio() const;
57      private:
58          const Screen    &S;
59          double          ratio;
60          SDL_Rect        virtualRect;
61          double          planW;
62          double          planH;
63          Point<double>   planUL;
64          Point<double>   planDR;
65          SDL_Texture     *texture;
66  };
67
68  #endif
```

fichierHPP/screen.hpp

```
#ifndef TIMER_HPP

# define TIMER_HPP

# include <SDL2/SDL.h>
# include <SDL2/SDL_timer.h>

class Timer
{
    private:
        int startTicks;
        int pausedTicks;

        bool paused;
        bool started;

    public:
        Timer();

        void start();
        void stop();
        void pause();
        void unpause();

        int get_ticks();

        bool is_started();
        bool is_paused();
};

#endif
```

fichierHPP/timer.hpp

```cpp
#include "draw.hpp"
#include "projet.hpp"
#include "point.hpp"
#include "polynome.hpp"
#include "mathBonus.hpp"
#include "screen.hpp"
#include "indice.hpp"
#include "parametricPlot.hpp"
#include <iostream>
#include <vector>
#include <boost/math/constants/constants.hpp>

void    lissajousCurveLoop(int m, int n) {
    /* draw the lissajou curve */
    Info            info;
    VirtualScreen   *screen;
    std::function<double(double)> xt = [m](double x) {return (sin(x * m * 2 * M_PI));};
    std::function<double(double)> yt = [n](double x) {return (cos(x * n * 2 * M_PI));};
    info.setXT(xt);
    info.setYT(yt);
    info.addVirtualScreen(Flag::full);
    screen = info.getCurrentScreen();
    screen->createPlan({-1.4 * screen->getRatio(), 1.4}, {1.4 * screen->getRatio(), -1.4});

    Timer   fps;
    PPlot   lissajousPlot(xt, yt);

    info.getTimer().start();
    while (handleEvent(noneFunction, launchColorMap, info))
    {
        fps.start();

        double t = info.getTimer().get_ticks() / 10000.0;
        screen->startDraw();
        SDL_SetRenderDrawColor(screen->getRenderer(), 255, 255, 255, 255);
        SDL_RenderClear(screen->getRenderer());
        SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 255, 255);
        lissajousPlot.plot(*screen, 0, 1, 5);
        // lissajousPlot.showDerivate(*screen, t);
        int indice = calcIndice(xt, yt, screen->convPoint(Point<int>{info.getMouseX(), info.getMouseY()}));
        if (info.getMouseMooved()) {
            drawIndice(info, indice);
        }
        if (info.getClicked()) {
            indiceAnimationLoop(info, screen->convPoint(Point<int>{info.getMouseX(), info.getMouseY()}));
        }
        screen->renderPresent();
        SDL_RenderPresent(screen->getRenderer());
        SDL_Delay(fmax(0, (1000 / 30) - fps.get_ticks()));
```

```
50          }
51      }
```

fichierCPP/LissajousCurve.cpp

```cpp
#include "draw.hpp"
#include "projet.hpp"
#include "point.hpp"
#include "polynome.hpp"
#include "mathBonus.hpp"
#include "screen.hpp"
#include "indice.hpp"
#include "parametricPlot.hpp"
#include <iostream>
#include <vector>
#include <boost/math/constants/constants.hpp>
#include "SDL2_gfxPrimitives.h"

void    lissajousCurveLoop2(int m, int n) {
    /* draw the lissajou curve */
    Info            info;
    VirtualScreen   *screen;
    std::function<double(double)> xt = [m](double x) {return (sin(x * m * 2 * M_PI));};
    std::function<double(double)> yt = [n](double x) {return (cos(x * n * 2 * M_PI));};
    // info.setXT(xt);
    // info.setYT(yt);
    info.addVirtualScreen(Flag::left);
    screen = info.getCurrentScreen();
    screen->createPlan({-1.4 * screen->getRatio(), 1.4}, {1.4 * screen->getRatio(), -1.4});
    info.addVirtualScreen(Flag::right);
    screen = info.getCurrentScreen();
    screen->createPlan({-1.4 * screen->getRatio(), 1.4}, {1.4 * screen->getRatio(), -1.4});
    // screen->createPlan({-1, 1 / screen->getRatio()}, {1, -1 / screen->getRatio()});
    // info.addVirtualScreen(Flag::down);
    // screen = info.getCurrentScreen();
    // screen->createPlan({-1.4 * screen->getRatio(), 1.4}, {1.4 * screen->getRatio(), -1.4});


    std::function<double(double)> dxt = derivateFunction(xt);
    std::function<double(double)> dyt = derivateFunction(yt);

    tie(dxt, dyt) = normalizeFunction(dxt, dyt);

    Timer   fps;
    PPlot   lissajousPlot(xt, yt);
    PPlot   lissajousPlotDerivate(dxt, dyt);

    info.getTimer().start();

    while (handleEvent(noneFunction, launchColorMap, info))
    {
        fps.start();

```

```
49          double t = info.getTimer().get_ticks() / 10000.0;
50          if (t > 1.0) {
51              t = 1.0;
52              info.getTimer().pause();
53          }
54          SDL_SetRenderDrawColor(info.getCurrentScreen()->getRenderer(), 0, 0, 0,
        255);
55          info.selectScreen(1);
56          screen = info.getCurrentScreen();
57          screen->startDraw();
58          SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 255, 255);
59          // lissajousPlotDerivate.plot(*screen, 0, 1, 5);
60          drawArrow(*screen, 0, 0, dxt(t) / 2, dyt(t) / 2);
61          double indicePart = calcIndicePart2(xt, yt, t);
62          drawCircleIndice(*screen, {0, 0}, 0, indicePart, 500);
63          filledCircleRGBA(screen->getRenderer(), screen->convPointSDL({dxt(t),
        dyt(t)}).x, screen->convPointSDL({dxt(t), dyt(t)}).y, 15, 255, 0, 0, 255);
64          screen->renderPresent();
65          info.selectScreen(0);
66          screen = info.getCurrentScreen();
67          screen->startDraw();
68          SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 255, 255);
69          lissajousPlot.plot(*screen, 0, 1, 5);
70          screen->renderPresent();
71          lissajousPlot.showDerivate(*screen, t);
72          int indice = calcIndice(xt, yt, screen->convPoint(Point<int>{info.
        getMouseX(), info.getMouseY()}));
73          if (info.getMouseMooved()) {
74              drawIndice(info, indice);
75          }
76          SDL_RenderPresent(screen->getRenderer());
77          SDL_Delay(fmax(0, (1000 / 30) - fps.get_ticks()));
78      }
79  }
```

fichierCPP/LissajousCurve2.cpp

```cpp
#include "draw.hpp"
#include "projet.hpp"
#include "point.hpp"
#include "polynome.hpp"
#include "polygone.hpp"
#include "mathBonus.hpp"
#include "screen.hpp"
#include "indice.hpp"
#include "parametricPlot.hpp"
#include <iostream>
#include <vector>
#include "SDL2_gfxPrimitives.h"

void    calculAire1(std::vector<std::vector<Case>>& map, int w, int h) {
    int aire = 0;
    for (int i = 0; i < w; i++) {
        for (int j = 0; j < h; j++) {
            if (map[i][j].indice != 0) {
                aire++;
            }
        }
    }
    std::cout << aire / (static_cast<double>(w * h)) << std::endl;
}

void    calculAire2(std::vector<std::vector<Case>>& map, int w, int h) {
    int aire = 0;
    for (int i = 0; i < w; i++) {
        for (int j = 0; j < h; j++) {
            if (map[i][j].indice != 0) {
                aire += map[i][j].indice;
            }
        }
    }
    std::cout << aire / (static_cast<double>(w * h)) << std::endl;
}

void    representationAire(Info& info, Polygone &poly) {
    Timer       fps;
    /* Illustrate the two color theoreme improve algorithm */
    VirtualScreen   *screen;
    screen = info.getCurrentScreen();
    std::vector<std::vector<Case>>  map(screen->getVirtualW(), std::vector<Case
    >(screen->getVirtualH()));

    drawPolygoneOnMap(*screen, poly, map);
    // parametricPlot.plotOnMap(*screen, 0, 1, map);
    screen->startDraw();
    int x;
    int y;
    while(findPointNotColored(map, x, y)) {
        Point<double> p = screen->convPoint(Point<int>{x, y});
        int indice = calcIndice(poly, p);
        colorMapIterative(map, x, y, indice);
```

```
54      }
55      int debug = 0;
56      for (size_t i = 0; i < map.size(); i++) {
57          for (size_t j = 0; j < map[i].size(); j++) {
58              if (map[i][j].onCurve == true) {
59                  debug++;
60                  map[i][j].indice = calcIndice(poly, screen->convPoint(Point<int
    >{x, y}));
61              }
62          }
63          std::cout << debug << std::endl;
64      }
65      std::cout << debug << std::endl;
66      calculAire1(map, screen->getVirtualW(), screen->getVirtualH());
67      calculAire2(map, screen->getVirtualW(), screen->getVirtualH());
68
69      info.getTimer().start();
70      SDL_SetRenderDrawColor(screen->getRenderer(), 255, 255, 255, 255);
71      SDL_RenderClear(screen->getRenderer());
72      screen->startDraw();
73      for (size_t i = 0; i < map.size(); i++) {
74          for (size_t j = 0; j < map[i].size(); j++) {
75              if (map[i][j].onCurve == true) {
76                  SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 0, 255);
77                  drawPoint(*screen, {static_cast<int>(i), static_cast<int>(j)},
    1);
78              } else if (map[i][j].colored == true) {
79                  if (map[i][j].indice == 0) {
80                      SDL_SetRenderDrawColor(screen->getRenderer(), 255, 255, 255,
     255);
81                  } else {
82                      SDL_SetRenderDrawColor(screen->getRenderer(), 255, 0, 0,
    255);
83                  }
84                  drawPoint(*screen, {static_cast<int>(i), static_cast<int>(j)},
    1);
85              }
86          }
87      }
88      screen->renderPresent();
89      SDL_RenderPresent(screen->getRenderer());
90      while (handleEvent(noneFunction, noneFunction, info))
91      {
92          fps.start();
93          SDL_Delay(fmax(0, (1000 / 30) - fps.get_ticks()));
94      }
95
96
97      SDL_SetRenderDrawColor(screen->getRenderer(), 255, 255, 255, 255);
98      SDL_RenderClear(screen->getRenderer());
99      screen->startDraw();
100     for (size_t i = 0; i < map.size(); i++) {
101         for (size_t j = 0; j < map[i].size(); j++) {
102             if (map[i][j].onCurve == true) {
103                 SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 0, 255);
```

```
104                     drawPoint(*screen, {static_cast<int>(i), static_cast<int>(j)},
    1);
105                 } else if (map[i][j].colored == true) {
106                     if (map[i][j].indice == 0) {
107                         SDL_SetRenderDrawColor(screen->getRenderer(), 255, 255, 255,
     255);
108                     } else if (map[i][j].indice < 0) {
109                         SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 100 *
    map[i][j].indice, 255);
110                     } else {
111                         SDL_SetRenderDrawColor(screen->getRenderer(), 100 * abs(map[
    i][j].indice), 0, 0, 255);
112                     }
113                     drawPoint(*screen, {static_cast<int>(i), static_cast<int>(j)},
    1);
114                 }
115             }
116         }
117         screen->renderPresent();
118         SDL_RenderPresent(screen->getRenderer());
119         while (handleEvent(noneFunction, noneFunction, info))
120         {
121             fps.start();
122             SDL_Delay(fmax(0, (1000 / 30) - fps.get_ticks()));
123         }
124
125     }
```

fichierCPP/aire.cpp

```cpp
#include "draw.hpp"
#include "projet.hpp"
#include "point.hpp"
#include "polynome.hpp"
#include "mathBonus.hpp"
#include "screen.hpp"
#include "parametricPlot.hpp"
#include "indice.hpp"
#include <iostream>
#include <vector>
#include <tuple>
#include <random>

std::vector<Point<double>>  generateBezierPointsLoop(Point<double> p1, int n,
    double y_min, double y_max) {
    /* Generate the points of the Bezier curve */
    std::vector<Point<double>>  points;
    std::random_device rd;
    std::default_random_engine eng(rd());
    std::uniform_real_distribution<double> distr_y(y_min, y_max);
    std::uniform_real_distribution<double> distr_x(0, 1.5);

    points.push_back(p1);
    for (int i = 0; i < n; i++) {
        points.push_back({distr_x(eng), distr_y(eng)});
    }
    points.push_back(p1);
    return (points);
}


void    bezierCurveCloseLoop() {
    /* Draw the Bezier curve */
    Info            info;
    VirtualScreen   *screen;

    info.addVirtualScreen(Flag::full);
    screen = info.getCurrentScreen();
    screen->createPlan({0, 1}, {1, -1});
    info.setBezierPoints(generateBezierPointsLoop({0, 0}, 10));
    Timer   fps;

    info.getTimer().start();
    while (handleEvent(changeBezierPoint, launchColorMap, info))
    {
        fps.start();
        std::function<double(double)>   xt = info.getXT();
        std::function<double(double)>   yt = info.getYT();
        PPlot   bezierPlot(xt, yt);

        screen->startDraw();
        SDL_SetRenderDrawColor(screen->getRenderer(), 255, 255, 255, 255);
        SDL_RenderClear(screen->getRenderer());
        SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 255, 255);
```

```
54          bezierPlot.plot(*screen, 0, 1, 3);
55          int indice = calcIndice(xt, yt, screen->convPoint(Point<int>{info.
        getMouseX(), info.getMouseY()}));
56          if (info.getMouseMooved()) {
57              drawIndice(info, indice);
58          }
59          if (info.getClicked()) {
60              indiceAnimationLoop(info, screen->convPoint(Point<int>{info.
        getMouseX(), info.getMouseY()}));
61          }
62          screen->renderPresent();
63          SDL_RenderPresent(screen->getRenderer());
64          SDL_Delay(fmax(0, (1000 / 30) - fps.get_ticks()));
65      }
66  }
```

fichierCPP/bezierCloseLoop.cpp

```cpp
#include "draw.hpp"
#include "projet.hpp"
#include "point.hpp"
#include "polynome.hpp"
#include "mathBonus.hpp"
#include "screen.hpp"
#include "parametricPlot.hpp"
#include <iostream>
#include <vector>
#include <tuple>
#include <random>

std::vector<Point<double>>  generateBezierPoints(Point<double> p1, Point<double> p2, int n, double y_min, double y_max) {
    /* Generate the points of the Bezier curve no loop allowed*/
    std::vector<Point<double>>  points;
    std::random_device rd;
    std::default_random_engine eng(rd());
    std::uniform_real_distribution<double> distr_y(y_min, y_max);
    std::uniform_real_distribution<double> distr_x(0, 1);

    points.push_back(p1);
    for (int i = 0; i < n; i++) {
        points.push_back({distr_x(eng), distr_y(eng)});
    }
    points.push_back(p2);
    std::sort(points.begin(), points.end());
    return (points);
}

std::vector<Point<double>>  generateNextPoints(std::vector<Point<double>> p, double t) {
    /* Generate the next points of the Bezier curve*/
    std::vector<Point<double>>  new_p;

    for (size_t i = 0; i < p.size() - 1; i++) {
        new_p.push_back({p[i].getX() * (1 - t) + p[i + 1].getX() * t, p[i].getY() * (1 - t) + p[i + 1].getY() * t});
    }
    return (new_p);
}

std::tuple<std::function<double(double)>, std::function<double(double)>> bezierCurvePoly(std::vector<Point<double>> bezierPoints) {
    /* Generate the polynomial of the Bezier curve*/
    Poly     px;
    Poly     py;

    std::tie(px, py) = BezierPoly(bezierPoints);
    return (std::make_tuple(std::bind( &Poly::operator(), px, std::placeholders::_1), std::bind( &Poly::operator(), py, std::placeholders::_1 )));
}


```

```
50 PPlot    bezierCurvePlot(std::vector<Point<double>> bezierPoints) {
51     /* Generate the plot of the Bezier curve*/
52     Poly     px;
53     Poly     py;
54 
55     std::tie(px, py) = BezierPoly(bezierPoints);
56     return (PPlot{std::bind( &Poly::operator(), px, std::placeholders::_1), std
       ::bind( &Poly::operator(), py, std::placeholders::_1 )});
57 }
58 
59 void     bezierCurveLoop() {
60     /* Draw the Bezier curve */
61     Info             info;
62     VirtualScreen    *screen;
63 
64     info.addVirtualScreen(Flag::full);
65     screen = info.getCurrentScreen();
66     screen->createPlan({0, 1}, {1, -1});
67     std::vector<Point<double>>  bezierPoints = generateBezierPoints({0, 0}, {1,
       0}, 10);
68     Timer    fps;
69     PPlot    bezierPlot(bezierCurvePlot(bezierPoints));
70 
71     info.getTimer().start();
72     while (handleEvent(hideOrShowAlgo, noneFunction, info))
73     {
74         fps.start();
75 
76         double t = info.getTimer().get_ticks() / 10000.0;
77         if (t >= 1.0) {
78             info.getTimer().start();
79             t = 0;
80         }
81         screen->startDraw();
82         SDL_SetRenderDrawColor(screen->getRenderer(), 255, 255, 255, 255);
83         SDL_RenderClear(screen->getRenderer());
84         SDL_SetRenderDrawColor(screen->getRenderer(), 0, 255, 0, 255);
85         if (info.showAlgo()) {
86             drawLines(*screen, bezierPoints);
87             drawPoints(*screen, bezierPoints, 5);
88         }
89         auto nextP = bezierPoints;
90         while (nextP.size() >= 2) {
91             SDL_SetRenderDrawColor(screen->getRenderer(), bezierPoints.size() /
       static_cast<double>(nextP.size()) * 255, 42, (1.0 - bezierPoints.size() /
       static_cast<double>(nextP.size())) * 255, 255);
92             nextP = generateNextPoints(nextP, t);
93             if (info.showAlgo()) {
94                 drawLines(*screen, nextP);
95             }
96         }
97         SDL_SetRenderDrawColor(screen->getRenderer(), 255, 0, 0, 255);
98         SDL_Point p = screen->convPointSDL(nextP[0]);
99         drawPoint(*screen, p, 15);
100        SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 255, 255);
```

```
101             bezierPlot.plot(*screen, 0, t, 3);
102             screen->renderPresent();
103             SDL_RenderPresent(screen->getRenderer());
104             SDL_Delay(fmax(0, (1000 / 30) - fps.get_ticks()));
105         }
106     }
```

fichierCPP/bezierCurve.cpp

```cpp
#include "draw.hpp"
#include "projet.hpp"
#include "point.hpp"
#include "polynome.hpp"
#include "mathBonus.hpp"
#include "screen.hpp"
#include "SDL2_gfxPrimitives.h"
#include <iostream>
#include <vector>
#include <tuple>
#include <random>

std::tuple<std::vector<Sint16>, std::vector<Sint16>> convBezierPointsGFX(VirtualScreen &screen, std::vector<Point<double>> points) {
    /* Convert the points of the Bezier curve to the points of the screen*/
    std::vector<Sint16>    x;
    std::vector<Sint16>    y;

    for (Point<double> p : points) {
        Point<int> p_ = screen.convPoint(p);
        x.push_back(p_.getX());
        y.push_back(p_.getY());
    }
    return (std::make_tuple(x, y));
}

void    bezierCurveLoopGFX() {
    /* Draw the Bezier curve using GFX*/
    Info            info;
    VirtualScreen   *screen;

    info.addVirtualScreen(Flag::full);
    screen = info.getCurrentScreen();
    screen->createPlan({0, 1}, {1, -1});
    std::vector<Point<double>>  bezierPoints = generateBezierPoints({0, 0}, {1, 0}, 5);
    Timer   fps;

    std::vector<Sint16> x;
    std::vector<Sint16> y;
    std::tie(x, y) = convBezierPointsGFX(*screen, bezierPoints);

    info.getTimer().start();
    while (handleEvent(hideOrShowAlgo, noneFunction, info))
    {
        fps.start();

        double t = info.getTimer().get_ticks() / 10000.0;
        if (t >= 1.0) {
            info.getTimer().start();
            t = 0;
        }
        screen->startDraw();
        SDL_SetRenderDrawColor(screen->getRenderer(), 255, 255, 255, 255);
```

```
53          SDL_RenderClear(screen->getRenderer());
54          if (info.showAlgo()) {
55                  drawLines(*screen, bezierPoints, 3, 255, 0, 0);
56          }
57          auto nextP = bezierPoints;
58          while (nextP.size() >= 2) {
59              SDL_SetRenderDrawColor(screen->getRenderer(), bezierPoints.size() /
    static_cast<double>(nextP.size()) * 255, 42, (1.0 - bezierPoints.size() /
    static_cast<double>(nextP.size())) * 255, 255);
60              nextP = generateNextPoints(nextP, t);
61              if (info.showAlgo()) {
62                  drawLines(*screen, nextP, 3, bezierPoints.size() / static_cast<
    double>(nextP.size()) * 255, 42, (1.0 - bezierPoints.size() / static_cast<
    double>(nextP.size())) * 255);
63                  drawPoints(*screen, nextP, 10);
64              }
65          }
66          SDL_SetRenderDrawColor(screen->getRenderer(), 255, 0, 0, 255);
67          SDL_Point p = screen->convPointSDL(nextP[0]);
68          drawPoint(*screen, p, 15);
69          SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 255, 255);
70          bezierRGBA(screen->getRenderer(), &x[0], &y[0], x.size(), 5, 255, 0, 0,
    255);
71          screen->renderPresent();
72          SDL_RenderPresent(screen->getRenderer());
73          SDL_Delay(fmax(0, (1000 / 30) - fps.get_ticks()));
74      }
75  }
```

fichierCPP/bezierCurveWithGFX.cpp

```cpp
#include "draw.hpp"
#include "projet.hpp"
#include "point.hpp"
#include "mathBonus.hpp"
#include "polynome.hpp"
#include "polygone.hpp"
#include <vector>
#include "SDL2_gfxPrimitives.h"

void    drawLine(VirtualScreen& screen, Point<double> p1, Point<double> p2) {
    /* Draw a line between two points*/
    Point<int>  p1_ = screen.convPoint(p1);
    Point<int>  p2_ = screen.convPoint(p2);

    SDL_RenderDrawLine(screen.getRenderer(), p1_.getX(), p1_.getY(), p2_.getX(),
    p2_.getY());
}

void    drawLine(VirtualScreen& screen, Point<double> p1, Point<double> p2, int
   size, int r, int g, int b) {
    /* Draw a line between two points*/
    Point<int>  p1_ = screen.convPoint(p1);
    Point<int>  p2_ = screen.convPoint(p2);

    thickLineRGBA(screen.getRenderer(), p1_.getX(), p1_.getY(), p2_.getX(), p2_.
   getY(), size, r, g, b, 255);
}

void    drawLines(VirtualScreen& screen, const std::vector<Point<double>>&
   points) {
    /* Draw all the lines between the points*/
    std::vector<SDL_Point>  pointsSDL;

    for (Point<double> p : points) {
        pointsSDL.push_back(screen.convPointSDL(p));
    }
    SDL_RenderDrawLines(screen.getRenderer(), &pointsSDL[0], pointsSDL.size());
}

void    drawLines(VirtualScreen& screen, const std::vector<Point<double>>&
   points, int size, int r, int g, int b) {
    /* Draw all the lines between the points*/
    std::vector<SDL_Point>  pointsSDL;

    for (Point<double> p : points) {
        pointsSDL.push_back(screen.convPointSDL(p));
    }
    for (size_t i = 0; i < pointsSDL.size() - 1; i++) {
        thickLineRGBA(screen.getRenderer(), pointsSDL[i].x, pointsSDL[i].y,
   pointsSDL[i + 1].x, pointsSDL[i + 1].y, size, r, g, b, 255);
    }
}

void    drawPoint(VirtualScreen& screen, SDL_Point point, int size) {
```

```
49      /* Draw a point*/
50      std::vector<SDL_Point>  points;
51
52      points.push_back(point);
53      for (int i = 0; i < size; i++) {
54          for (int j = 0; j < size; j++) {
55              points.push_back({point.x + i - size / 2, point.y + j - size / 2});
56          }
57      }
58      SDL_RenderDrawPoints(screen.getRenderer(), &points[0], points.size());
59  }
60
61  void    drawPoints(VirtualScreen& screen, std::vector<Point<double>> points, int
        size) {
62      for (Point<double> p : points) {
63          drawPoint(screen, screen.convPointSDL(p), size);
64      }
65  }
66
67  void    drawArrow(VirtualScreen& screen, double x, double y, double dx, double
        dy) {
68      /* Draw an arrow*/
69      SDL_Point p1 = screen.convPointSDL({x, y});
70      SDL_Point p2 = screen.convPointSDL({x + dx, y + dy});
71
72      thickLineRGBA(screen.getRenderer(), p1.x, p1.y, p2.x, p2.y, 7, 0xFF, 0, 0, 0
        xFF);
73
74      double  x1 = x + dx + (dx * 0.2);
75      double  y1 = y + dy + (dy * 0.2);
76
77      double  x2 = x + dx - (dy * 0.2);
78      double  y2 = y + dy + (dx * 0.2);
79
80      double  x3 = x + dx + (dy * 0.2);
81      double  y3 = y + dy - (dx * 0.2);
82
83      p1 = screen.convPointSDL({x1, y1});
84      p2 = screen.convPointSDL({x2, y2});
85      SDL_Point p3 = screen.convPointSDL({x3, y3});
86      filledTrigonRGBA(screen.getRenderer(), p1.x, p1.y, p2.x, p2.y, p3.x, p3.y,
        255, 0, 0, 255);
87  }
88
89  void    drawArrowAngle(VirtualScreen& screen, Point<double> p, double angle, int
        size) {
90      /* Draw an arrow with an angle*/
91      double  x = p.getX();
92      double  y = p.getY();
93      int     dx = cos(angle) * size;
94      int     dy = -sin(angle) * size;
95
96      SDL_Point p1 = screen.convPointSDL({x, y});
97      SDL_Point p2;
98      p2.x = p1.x + dx;
```

```
99      p2.y = p1.y + dy;
100
101     thickLineRGBA(screen.getRenderer(), p1.x, p1.y, p2.x, p2.y, 14, 0xFF, 0, 0,
        0xFF);
102
103     SDL_Point   p3;
104     p3.x = p1.x + dx * 1.2;
105     p3.y = p1.y + dy * 1.2;
106     SDL_Point   p4;
107     p4.x = p1.x + dx - dy * 0.2;
108     p4.y = p1.y + dy + dx * 0.2;
109     SDL_Point   p5;
110     p5.x = p1.x + dx + dy * 0.2;
111     p5.y = p1.y + dy - dx * 0.2;
112     filledTrigonRGBA(screen.getRenderer(), p3.x, p3.y, p4.x, p4.y, p5.x, p5.y,
        255, 0, 0, 255);
113 }
114
115 void    drawPointCoord(VirtualScreen& screen, Point<double> p, int x, int y) {
116     /* Draw the coordinate value of the mouse*/
117     std::string     text = std::to_string(p.getX()) + ", " + std::to_string(p.
        getY());
118     SDL_Surface     *surface;
119     SDL_Texture     *texture;
120     SDL_Color       color;
121     SDL_Rect        TextRect;
122     SDL_Rect        rect;
123
124     color.r = 255;
125     color.g = 255;
126     color.b = 255;
127     color.a = 255;
128     surface = TTF_RenderText_Blended(my_font, text.c_str(), color);
129     TTF_SizeText(my_font, text.c_str(), &TextRect.w, &TextRect.h);
130     texture = SDL_CreateTextureFromSurface(screen.getRenderer(), surface);
131     TextRect.x = x + TextRect.w;
132     TextRect.y = y - TextRect.h;
133     rect = {TextRect.x - 5, TextRect.y - 2, TextRect.w + 10, TextRect.h + 4};
134     SDL_SetRenderDrawColor(screen.getRenderer(), 0, 0, 0, 255);
135     SDL_RenderFillRect(screen.getRenderer(), &rect);
136     SDL_RenderCopy(screen.getRenderer(), texture, NULL, &TextRect);
137 }
138
139 double  convAngleGFX(double angle) {
140     /* Convert an angle from radian to degree in anti clockwise*/
141     if (angle <= 0) {
142         return(-angle);
143     } else {
144         return(2 * M_PI - angle);
145     }
146 }
147
148 void    drawCircleIndice(VirtualScreen& screen, Point<double> p, double angle0,
    double indice, int radius) {
149     /* Draw a circle with an indice*/
```

```
150     int      color;
151
152     if (indice < 0) {
153         color = 0xFFFF0000;
154     } else {
155         color = 0xFF0000FF;
156     }
157     while (radianToDegree(abs(indice)) >= 360) {
158         if (indice < 0) {
159             indice += 2 * M_PI;
160         } else {
161             indice -= 2 * M_PI;
162         }
163         thickCircleColor(screen.getRenderer(), screen.convPointSDL(p).x, screen.
    convPointSDL(p).y, radius, color, 10);
164         radius += 50;
165     }
166     angle0 = convAngleGFX(angle0);
167     int angleSart = radianToDegree(angle0);
168     int angleFinal = angleSart - radianToDegree(indice);
169     // indice = angle0 - indice;
170     if (angleSart > angleFinal) {
171         std::swap(angleSart, angleFinal);
172     }
173     thickArcColor(screen.getRenderer(), screen.convPointSDL(p).x, screen.
    convPointSDL(p).y, radius, angleSart, angleFinal, color, 10);
174 }
175
176 void    drawCircles(VirtualScreen& screen, std::vector<Point<double>>& points,
    int radius, int r, int g, int b) {
177     /* Draw the circles */
178     for (int i = 0; i < points.size(); i++) {
179         filledCircleRGBA(screen.getRenderer(), screen.convPointSDL(points[i]).x,
     screen.convPointSDL(points[i]).y, radius, r, g, b, 255);
180     }
181 }
182
183 void    drawPolygone(VirtualScreen& screen, Polygone poly, int size, int r, int
    g, int b) {
184     std::vector<Point<double>>  points = poly.getPoints();
185     if (poly.isClosed()) {
186         drawLines(screen, points, size, r, g, b);
187         drawLine(screen, points[0], points[points.size() - 1], size, r, g, b);
188     }
189     if (points.size() > 0) {
190         drawCircles(screen, points, size * 3, r, g, b);
191     }
192 }
193
194 void    drawPointPolygone(VirtualScreen& screen, Polygone poly, int size, int r,
     int g, int b, double t) {
195     std::function<double(double)>   xt;
196     std::function<double(double)>   yt;
197     std::vector<Point<double>>      points;
198     int                             nbLineTravel;
```

```
199
200     points = poly.getPoints();
201     if (points.size() < 3)
202         return ;
203     nbLineTravel = t / (1.0 / static_cast<double>(points.size()));
204     t = (t - nbLineTravel * (1.0 / static_cast<double>(points.size()))) *
        static_cast<double>(points.size());
205     std::cout << "t = " << t << std::endl;
206     tie(xt, yt) = lineFunction(points[nbLineTravel % points.size()], points[(
        nbLineTravel + 1) % points.size()]);
207     SDL_Point   p1 = screen.convPointSDL({xt(t), yt(t)});
208     filledCircleRGBA(screen.getRenderer(), p1.x, p1.y, size, r, g, b, 255);
209 }
210
211 void    drawPolygoneOnMap(VirtualScreen& screen, Polygone& poly, std::vector<std
        ::vector<Case>> &map) {
212     double  pas = 1 / static_cast<double>(screen.getVirtualW()) / 8.0;
213     std::function<double(double)>   xt;
214     std::function<double(double)>   yt;
215     std::vector<Point<double>>      points;
216
217     points = poly.getPoints();
218     if (points.size() < 3)
219         return ;
220     for (int i = 0; i < points.size(); i++) {
221         tie(xt, yt) = lineFunction(points[i], points[(i + 1) % points.size()]);
222         for (double t = 0; t <= 1; t += pas) {
223             double x = xt(t);
224             double y = yt(t);
225             SDL_Point point = screen.convPointSDL({x, y});
226             for (int i = -1; i <= 1; i++) {
227                 for (int j = -1; j <= 1; j++) {
228                     if (point.x + i >= 0 && point.x + i < map.size() && point.y
        + j >= 0 && point.y + j < map[0].size()) {
229                         map[point.x + i][point.y + j].colored = true;
230                         map[point.x + i][point.y + j].onCurve = true;
231                     }
232                 }
233             }
234         }
235     }
236 }
237
238 /*void  PPlot::plotOnMap(VirtualScreen &screen, double tStart, double tEnd, std
        ::vector<std::vector<Case>> &map) {
239     double  w = tEnd - tStart;
240     double  pas = w / static_cast<double>(screen.getVirtualW()) / 8.0;
241
242     if (w == 0)
243         return ;
244     for (double t = tStart; t <= tEnd; t += pas) {
245         double x = xt_(t);
246         double y = yt_(t);
247         SDL_Point point = screen.convPointSDL({x, y});
248         for (int i = -3; i <= 3; i++) {
```

```
249                 for (int j = -3; j <= 3; j++) {
250                     if (point.x + i >= 0 && point.x + i < map.size() && point.y + j
    >= 0 && point.y + j < map[0].size()) {
251                         map[point.x + i][point.y + j].colored = true;
252                         map[point.x + i][point.y + j].onCurve = true;
253                     }
254                 }
255             }
256         }
257 }
258 */
```

fichierCPP/draw.cpp

```cpp
#include <SDL2/SDL.h>
#include <functional>
#include "projet.hpp"
#include "timer.hpp"

bool    handleEvent(std::function<void(Info& info)> f1, std::function<void(Info& info)> f2, Info& info)
{
    /* Handle the events*/
    int         x;
    int         y;
    int         button;
    bool        close_requested = false;
    SDL_Event   event;

    SDL_GetMouseState(&x, &y);
    info.setMouseInfo(x * info.getHighDPI(), y * info.getHighDPI(), true);
    info.setClickInfo(x * info.getHighDPI(), y * info.getHighDPI(), false);
    while (SDL_PollEvent(&event))
    {
        switch (event.type)
        {
            case SDL_QUIT:
                close_requested = true;
                break;
            case SDL_KEYDOWN:
                switch (event.key.keysym.sym)
                {
                    case SDLK_ESCAPE:
                        close_requested = true;
                        break;
                    case SDLK_a:
                        f1(info);
                        break;
                    case SDLK_b:
                        f2(info);
                        break;
                    case SDLK_SPACE:
                        if (!info.getTimer().is_paused()) {
                            info.getTimer().pause();
                        } else {
                            info.getTimer().unpause();
                        }
                        break;
                    default:
                        break;
                }
                break;
            case SDL_MOUSEMOTION:
                SDL_GetMouseState(&x, &y);
                info.setMouseInfo(x * info.getHighDPI(), y * info.getHighDPI(), true);
                break;
            case SDL_MOUSEBUTTONDOWN:
```

```
53                  button = SDL_GetMouseState(&x, &y);
54                  info.setClickInfo(x * info.getHighDPI(), y * info.getHighDPI(),
      button);
55                  break;
56              default:
57                  break;
58          }
59      }
60      return (!close_requested);
61  }
```

fichierCPP/event.cpp

```cpp
#include <SDL2/SDL.h>
#include <functional>
#include "projet.hpp"
#include "timer.hpp"

void    noneFunction(Info& info) {
    /* Do nothing */
    (void)info;
}

void    hideOrShowAlgo(Info& info) {
    /* Hide or show the algo */
    info.hideOrShowAlgo(!info.showAlgo());
}

void    changeBezierPoint(Info& info) {
    /* Change the bezier point */
    info.setBezierPoints(generateBezierPointsLoop({0, 0}, 10));
}

void    launchColorMap(Info& info) {
    /* Launch the color map */
    twoColorMapLoopImprove(info);
}
```

fichierCPP/eventFunction.cpp

```cpp
#include <SDL2/SDL_ttf.h>
#include <tuple>
#include <string>
#include "projet.hpp"
#include "polygone.hpp"
#include "indice.hpp"
#include "mathBonus.hpp"
#include <boost/math/constants/constants.hpp>
#include <tuple>

void    drawIndice(Info& info, int indice) {
    /* Draw the numerival value of the indice near the mouse */
    SDL_Surface     *surface;
    SDL_Texture     *texture;
    VirtualScreen   *screen;
    SDL_Color       color;
    SDL_Rect        TextRect;
    SDL_Rect        rect;

    color.r = 255;
    color.g = 255;
    color.b = 255;
    color.a = 255;
    screen = info.getCurrentScreen();
    surface = TTF_RenderText_Blended(my_font, std::to_string(indice).c_str(), color);
    TTF_SizeText(my_font, std::to_string(indice).c_str(), &TextRect.w, &TextRect.h);
    texture = SDL_CreateTextureFromSurface(screen->getRenderer(), surface);
    TextRect.x = info.getMouseX() - TextRect.w;
    TextRect.y = info.getMouseY() - TextRect.h;
    rect = {TextRect.x - 5, TextRect.y - 2, TextRect.w + 10, TextRect.h + 4};
    SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 0, 255);
    SDL_RenderFillRect(screen->getRenderer(), &rect);
    SDL_RenderCopy(screen->getRenderer(), texture, NULL, &TextRect);
}

void    drawIndice(Info& info, double indice) {
    /* Draw the numerival value of the indice near the mouse */
    SDL_Surface     *surface;
    SDL_Texture     *texture;
    VirtualScreen   *screen;
    SDL_Color       color;
    SDL_Rect        TextRect;
    SDL_Rect        rect;

    color.r = 255;
    color.g = 255;
    color.b = 255;
    color.a = 255;
    screen = info.getCurrentScreen();
    surface = TTF_RenderText_Blended(my_font, std::to_string(indice).c_str(), color);
    TTF_SizeText(my_font, std::to_string(indice).c_str(), &TextRect.w, &TextRect
```

```
    .h);
52      texture = SDL_CreateTextureFromSurface(screen->getRenderer(), surface);
53      TextRect.x = info.getMouseX() - TextRect.w;
54      TextRect.y = info.getMouseY() - TextRect.h;
55      rect = {TextRect.x - 5, TextRect.y - 2, TextRect.w + 10, TextRect.h + 4};
56      SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 0, 255);
57      SDL_RenderFillRect(screen->getRenderer(), &rect);
58      SDL_RenderCopy(screen->getRenderer(), texture, NULL, &TextRect);
59  }
60
61  std::function<double(double)>    inIntegralFunction(std::function<double(double)>
        xt, std::function<double(double)> yt, Point<double> p) {
62      /* return the function used in the integeal */
63      std::function<double(double)> at;
64      std::function<double(double)> bt;
65      tie(at, bt) = shiftFunction(xt, yt, p);
66      tie(at, bt) = normalizeFunction(at, bt);
67      std::function<double(double)> I = {[at, bt](double t) {return (at(t) *
        derivate(bt, t) - bt(t) * derivate(at, t));}};
68      return (I);
69  }
70
71  int      calcIndice(std::function<double(double)> xt, std::function<double(double
        )> yt, Point<double> p) {
72      /* Calculate the indice of the function at the point p*/
73      std::function<double(double)>    I = inIntegralFunction(xt, yt, p);
74      return (round((integrate(0, 1, I, 10000)) / (2 * M_PI)));
75  }
76
77  double  calcIndicePart(std::function<double(double)> xt, std::function<double(
        double)> yt, Point<double> p, double t) {
78      /* Calculate the indice of the function at the point p at the time t*/
79      if (t < 0 || t > 1)
80          return (0);
81
82      std::function<double(double)>    I = inIntegralFunction(xt, yt, p);
83      return (integrate(0, t, I, 1000));
84  }
85
86  int      calcIndice(Polygone poly, Point<double> p) {
87      std::function<double(double)>    xt;
88      std::function<double(double)>    yt;
89      std::vector<Point<double>>       points;
90      double                           indice;
91
92      points = poly.getPoints();
93      if (points.size() < 3)
94          return (0);
95      indice = 0;
96      for (int i = 0; i < points.size(); i++) {
97          tie(xt, yt) = lineFunction(points[i], points[(i + 1) % points.size()]);
98          indice += calcIndicePart(xt, yt, p, 1);
99      }
100     return (round(indice / (2 * M_PI)));
101 }
```

```
102
103 double  calcIndicePart(Polygone poly, Point<double> p, double t) {
104     std::function<double(double)>   xt;
105     std::function<double(double)>   yt;
106     std::vector<Point<double>>      points;
107     double                          indice = 0;
108     int                             nbLineTravel;
109
110     points = poly.getPoints();
111     if (points.size() < 3)
112         return (0);
113     nbLineTravel = t / (1.0 / static_cast<double>(points.size()));
114     for (int i = 0; i < nbLineTravel; i++) {
115         tie(xt, yt) = lineFunction(points[i], points[(i + 1) % points.size()]);
116         indice += calcIndicePart(xt, yt, p, 1);
117     }
118     t = (t - nbLineTravel * (1.0 / static_cast<double>(points.size()))) *
    static_cast<double>(points.size());
119     if (t > 0) {
120         tie(xt, yt) = lineFunction(points[nbLineTravel], points[(nbLineTravel +
    1) % points.size()]);
121         indice += calcIndicePart(xt, yt, p, t);
122     }
123     return (indice);
124 }
125
126 int     calcIndice2(std::function<double(double)> xt, std::function<double(
    double)> yt) {
127     /* Calculate the indice of the function at the point p*/
128     std::function<double(double)>   I = inIntegralFunction(derivateFunction(xt),
     derivateFunction(yt), Point<double>(0, 0));
129     return (round((integrate(0, 1, I, 10000)) / (2 * M_PI)));
130 }
131
132 double  calcIndicePart2(std::function<double(double)> xt, std::function<double(
    double)> yt, double t) {
133     /* Calculate the indice of the function at the point p at the time t*/
134     if (t < 0 || t > 1)
135         return (0);
136     std::function<double(double)>   I = inIntegralFunction(derivateFunction(xt),
     derivateFunction(yt), Point<double>(0, 0));
137     return (integrate(0, t, I, 1000));
138 }
```

fichierCPP/indice.cpp

```cpp
#include "draw.hpp"
#include "projet.hpp"
#include "point.hpp"
#include "polynome.hpp"
#include "mathBonus.hpp"
#include "screen.hpp"
#include "indice.hpp"
#include "parametricPlot.hpp"
#include <iostream>
#include <vector>
#include "SDL2_gfxPrimitives.h"

void    indiceAnimationLoop(Info& info, Point<double> p) {
    /* Illustrate what is the indice of a point */
    VirtualScreen   *screen;
    screen = info.getCurrentScreen();

    Timer       fps;
    Point<int>  p_ = screen->convPoint(p);
    std::function<double(double)> xt = info.getXT();
    std::function<double(double)> yt = info.getYT();
    PPlot       parametricPlot(xt, yt);

    std::cout << p << std::endl;

    double  angle0 = atan2(yt(0) - p.getY(), xt(0) - p.getX());

    info.getTimer().start();
    info.getTimer().pause();
    while (handleEvent(noneFunction, noneFunction, info))
    {
        fps.start();

        double t = info.getTimer().get_ticks() / 10000.0;
        if (t > 1.0) {
            t = 1.0;
            info.getTimer().pause();
        }
        screen->startDraw();
        SDL_SetRenderDrawColor(screen->getRenderer(), 255, 255, 255, 255);
        SDL_RenderClear(screen->getRenderer());
        SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 0, 255);
        parametricPlot.plot(*screen, 0, 1, 5);
        SDL_SetRenderDrawColor(screen->getRenderer(), 255, 0, 0, 255);
        filledCircleRGBA(screen->getRenderer(), screen->convPointSDL({xt(t), yt(t)}).x, screen->convPointSDL({xt(t), yt(t)}).y, 15, 255, 0, 0, 255);
        // parametricPlot.showDerivate(*screen, t);
        double indice = calcIndicePart(xt, yt, p, t);
        if (info.getTimer().is_paused()) {
            std::cout << " ------ " << std::endl;
        }
        drawArrowAngle(*screen, p, angle0 + indice, 100);
        drawCircleIndice(*screen, p, angle0, indice);
        filledCircleRGBA(screen->getRenderer(), p_.getX(), p_.getY(), 15, 0, 0,
```

```
    0, 255);
54          screen->renderPresent();
55          SDL_RenderPresent(screen->getRenderer());
56          SDL_Delay(fmax(0, (1000 / 30) - fps.get_ticks()));
57      }
58  }
59
60
61  void    indiceAnimationLoopPolygone(Info& info, Point<double> p, Polygone poly)
        {
62      /* Illustrate what is the indice of a point */
63      VirtualScreen   *screen;
64      screen = info.getCurrentScreen();
65
66      Timer       fps;
67      Point<int>  p_ = screen->convPoint(p);
68      std::vector<Point<double>>  points = poly.getPoints();
69
70
71      double  angle0 = atan2(points[0].getY() - p.getY(), points[0].getX() - p.
        getX());
72      info.getTimer().start();
73      info.getTimer().pause();
74      while (handleEvent(noneFunction, noneFunction, info))
75      {
76          fps.start();
77
78          double t = info.getTimer().get_ticks() / 10000.0;
79          if (t > 1.0) {
80              t = 1.0;
81              info.getTimer().pause();
82          }
83          screen->startDraw();
84          SDL_SetRenderDrawColor(screen->getRenderer(), 255, 255, 255, 255);
85          SDL_RenderClear(screen->getRenderer());
86          screen->startDraw();
87          drawPolygone(*screen, poly, 5, 0, 0, 0);
88          double indice = calcIndicePart(poly, p, t);
89          drawArrowAngle(*screen, p, angle0 + indice, 100);
90          drawPointPolygone(*screen, poly, 15, 255, 0, 0, t);
91          drawCircleIndice(*screen, p, angle0, indice);
92          filledCircleRGBA(screen->getRenderer(), p_.getX(), p_.getY(), 15, 0, 0,
        0, 255);
93          screen->renderPresent();
94          SDL_RenderPresent(screen->getRenderer());
95          SDL_Delay(fmax(0, (1000 / 30) - fps.get_ticks()));
96      }
97  }
```

fichierCPP/indiceAnimation.cpp

```cpp
#include "projet.hpp"
#include "indice.hpp"
#include <boost/program_options.hpp>

namespace po = boost::program_options;

int     main(int argc, char **argv)
{
    po::options_description desc("Allowed options");
    desc.add_options()
    ("help", "describe arguments")
    ("BCurve", "Launch to see Bezier curve algorithm")
    ("BLoop", "Launch to see Bezier close loop")
    ("LCurve", "Launch to see Lissajous curve")
    ("BCurveGFX", "Launch to see Bezier curve algorithm with GFX")
    ("Polygone", "Launch to see Polygone");
    po::variables_map vm;
    try {
        po::store(po::parse_command_line(argc, argv, desc), vm);
        po::notify(vm);
    } catch(po::unknown_option const & unknown) {
        std::cout << "Unknown option: " << unknown.get_option_name() << std::
    endl;
        std::cerr << desc;
        return (1);
    }
    if (vm.count ("help")) {
        std::cerr << desc << "\n";
        return 1;
    }
    if (vm.count("BCurve")) {
        bezierCurveLoop();
    } else if (vm.count("BCurveGFX")) {
        bezierCurveLoopGFX();
    } else if (vm.count("LCurve")) {
        lissajousCurveLoop2(2, 3);
    } else if (vm.count("Polygone")) {
        polygoneBuilder();
    } else if (vm.count("BLoop")) {
        bezierCurveCloseLoop();
    }
    return (0);
}
```

fichierCPP/main.cpp

```cpp
#include "mathBonus.hpp"
#include "point.hpp"
#include <iostream>

double  fact(int n) {
    /* Compute the factorial of a number */
    double  res = 1;

    while (n > 1) {
        res *= n;
        n--;
    }
    return (res);
}

double  comb(int k, int n) {
    /* Compute the combination of a number */
    return (fact(n) / (fact(k) * fact(n - k)));
}

double  derivate(std::function<double(double)> f, double t) {
    /* Compute the derivate of a function */
    double h = 0.000001;

    return ((f(t + h) - f(t))/(h));
}

std::tuple<double, double>  normalize(double x, double y) {
    /* Normalize a vector */
    double norm = sqrt(x * x + y * y);

    if (norm != 0) {
        return(std::make_tuple(x / norm, y / norm));
    } else {
        return(std::make_tuple(0, 0));
    }
}

double  normalizeX(double x, double y) {
    /* Normalize a vector */
    double norm = sqrt(x * x + y * y);

    if (norm != 0) {
        return (x / norm);
    } else {
        return (0);
    }
}

double  normalizeY(double x, double y) {
    /* Normalize a vector */
    double norm = sqrt(x * x + y * y);

    if (norm != 0) {
```

```
55          return (y / norm);
56      } else {
57          return (0);
58      }
59  }
60
61  double  integrate(double a, double b, std::function<double(double)> f, int N)
62  {
63      /* Compute the integral of a function between a and b*/
64      double  pas = (b - a) / static_cast<double>(N);
65      double  sum = 0;
66      for (int i = 0 ; i < N ; i++) {
67          sum +=  (f(a + i * pas) + 4 * f(a + (i + 0.5) * pas) + f(a + (i + 1) *
    pas));
68      }
69      return (sum * pas / 6);
70  }
71
72  double      polaireAngle(Point<double> p1)
73  {
74      /* Compute the angle of a vector */
75      return (atan2(p1.getY(), p1.getX()));
76  }
77
78  int         radianToDegree(double radian)
79  {
80      /* Convert radian to degree */
81      return (static_cast<int>(radian * 180 / M_PI));
82  }
83
84  std::tuple<std::function<double(double)>, std::function<double(double)>>
    lineFunction(Point<double> p1, Point<double> p2) {
85      /* return the function used in the line */
86      std::function<double(double)> xt = {[p1, p2](double t) {return (p1.getX() +
    (p2.getX() - p1.getX()) * t);}};
87      std::function<double(double)> yt = {[p1, p2](double t) {return (p1.getY() +
    (p2.getY() - p1.getY()) * t);}};
88      return (std::make_tuple(xt, yt));
89  }
90
91  std::function<double(double)>   derivateFunction(std::function<double(double)> f
    ) {
92      /* Return the derivate of the function f */
93      return [f](double t) {
94          return (derivate(f, t));
95      };
96  }
97
98  std::tuple<std::function<double(double)>, std::function<double(double)>>
    shiftFunction(std::function<double(double)> xt, std::function<double(double)>
     yt, Point<double> p) {
99      /* Shift the function to the point p */
100     double  x = p.getX();
101     double  y = p.getY();
102
```

```
103     std::function<double(double)> xt_ = {[x, xt](double t) {return (xt(t) - x)
        ;}};
104     std::function<double(double)> yt_ = {[y, yt](double t) {return (yt(t) - y)
        ;}};
105     return (std::make_tuple(xt_, yt_));
106 }
107
108 std::tuple<std::function<double(double)>, std::function<double(double)>>
        normalizeFunction(std::function<double(double)> xt, std::function<double(
        double)> yt) {
109     /* Normalize the function */
110     std::function<double(double)> at = {[xt, yt](double t) {return (normalizeX(
        xt(t), yt(t)));}};
111     std::function<double(double)> bt = {[xt, yt](double t) {return (normalizeY(
        xt(t), yt(t)));}};
112     return (std::make_tuple(at, bt));
113 }
```

fichierCPP/mathBonus.cpp

```cpp
#include "parametricPlot.hpp"
#include "mathBonus.hpp"
#include "draw.hpp"
#include "SDL2_gfxPrimitives.h"

PPlot::PPlot(std::function<double(double)> xt, std::function<double(double)> yt)
    : xt_(xt), yt_(yt) {};


void    PPlot::plot(VirtualScreen &screen, double tStart, double tEnd, int size)
    {
    /* Plot the parametric function */
    double  w = tEnd - tStart;
    double  pas = w / static_cast<double>(screen.getVirtualW()) / 8.0;

    if (w == 0)
        return ;
    for (double t = tStart; t <= tEnd; t += pas) {
        double x = xt_(t);
        double y = yt_(t);
        SDL_Point point = screen.convPointSDL({x, y});
        drawPoint(screen, point, size);
        // for (int i = -size / 2; i <= size / 2; i++) {
        //  SDL_RenderDrawPoint(screen.getRenderer(), point.x, point.y + i);
        // }
    }
    SDL_Point point = screen.convPointSDL({xt_(0), yt_(0)});
}

void    PPlot::plotOnMap(VirtualScreen &screen, double tStart, double tEnd, std
    ::vector<std::vector<Case>> &map) {
    /* Plot the parametric function on a map*/
    double  w = tEnd - tStart;
    double  pas = w / static_cast<double>(screen.getVirtualW()) / 8.0;

    if (w == 0)
        return ;
    for (double t = tStart; t <= tEnd; t += pas) {
        double x = xt_(t);
        double y = yt_(t);
        SDL_Point point = screen.convPointSDL({x, y});
        for (int i = -3; i <= 3; i++) {
            for (int j = -3; j <= 3; j++) {
                if (point.x + i >= 0 && point.x + i < map.size() && point.y + j
    >= 0 && point.y + j < map[0].size()) {
                    map[point.x + i][point.y + j].colored = true;
                    map[point.x + i][point.y + j].onCurve = true;
                }
            }
        }
    }
}


```

```
51
52 void     PPlot::showDerivate(VirtualScreen &screen, double t) {
53     /* Plot the derivate of the parametric function */
54     double  x = xt_(t);
55     double  y = yt_(t);
56     double  dx = derivate(xt_, t);
57     double  dy = derivate(yt_, t);
58     std::tie(dx, dy) = normalize(dx, dy);
59     dx /= 5.0;
60     dy /= 5.0;
61     drawArrow(screen, x, y, dx, dy);
62 }
63
64 std::function<double(double)>    PPlot::getXt() const {
65     /* Get the function of x */
66     return (xt_);
67 }
68
69 std::function<double(double)>    PPlot::getYt() const {
70     /* Get the function of y */
71     return (yt_);
72 }
```

fichierCPP/parametricPlot.cpp

```cpp
#include "point.hpp"
#include <vector>
#include <iostream>
#include <algorithm>


std::ostream& operator<<(std::ostream &os, const Point<double> &p) {
    os << p.x_ << ", " << p.y_;
    return (os);
}


std::ostream& operator<<(std::ostream &os, const Point<int> &p) {
    os << p.x_ << ", " << p.y_;
    return (os);
}
```

fichierCPP/point.cpp

```cpp
#include "polygone.hpp"

Polygone::Polygone(std::vector<Point<double>>    points) : points_(points),
    closed_(false) {
    if (points_.size() > 2) {
        closed_ = true;
    }
}

Polygone::~Polygone() {
}

Polygone::Polygone() : closed_(false) {}

void    Polygone::addPoint(Point<double> point) {
    points_.push_back(point);
    if (points_.size() > 2) {
        closed_ = true;
    }
}

bool    Polygone::isClosed() {
    return (closed_);
}

std::vector<Point<double>>  Polygone::getPoints() const {
    return (points_);
}
```

fichierCPP/polygone.cpp

```cpp
#include "polygone.hpp"
#include "draw.hpp"
#include "projet.hpp"
#include "point.hpp"
#include "polynome.hpp"
#include "mathBonus.hpp"
#include "screen.hpp"
#include "parametricPlot.hpp"
#include "indice.hpp"
#include <iostream>
#include <vector>
#include <tuple>
#include <random>

void    polygoneBuilder() {
    Info        info;
    Polygone    poly;

    VirtualScreen   *screen;
    info.addVirtualScreen(Flag::full);
    screen = info.getCurrentScreen();
    screen->createPlan({0, 1}, {1 * screen->getRatio(), 0});
    Timer       fps;
    info.getTimer().start();
    //ajout d'un polygone étoilé
    // poly.addPoint(Point(1 * screen->getRatio() / 2, 0.9));
    // poly.addPoint(Point(1 * screen->getRatio() / 2 + 0.4, 0.1));
    // poly.addPoint(Point(1 * screen->getRatio() / 2 - 0.4, 0.65));
    // poly.addPoint(Point(1 * screen->getRatio() / 2 + 0.4, 0.65));
    // poly.addPoint(Point(1 * screen->getRatio() / 2 - 0.4, 0.1));
    // representationAire(info, poly);
    while (handleEvent(noneFunction, noneFunction, info))
    {
        fps.start();

        screen->startDraw();
        SDL_SetRenderDrawColor(screen->getRenderer(), 255, 255, 255, 255);
        SDL_RenderClear(screen->getRenderer());
        screen->startDraw();
        if (info.getClicked() == SDL_BUTTON_LEFT) {
            poly.addPoint(screen->convPoint(Point<int>{info.getMouseX(), info.
    getMouseY()}));
        }
        int indice = calcIndice(poly, screen->convPoint(Point<int>{info.
    getMouseX(), info.getMouseY()}));
        if (info.getMouseMooved()) {
            drawIndice(info, indice);
        }
        if (info.getClicked() == SDL_BUTTON_X1) {
            indiceAnimationLoopPolygone(info, screen->convPoint(Point<int>{info.
    getMouseX(), info.getMouseY()}), poly);
        } else if (info.getClicked()) {
            std::cout << "button: " << info.getClicked() << std::endl;
        }
```

```
52            drawPolygone(*screen, poly, 5, 0, 0, 0);
53            screen->renderPresent();
54            SDL_RenderPresent(screen->getRenderer());
55            SDL_Delay(fmax(0, (1000 / 30) - fps.get_ticks()));
56        }
57    }
```

fichierCPP/polygoneBuilder.cpp

```cpp
#include "polynome.hpp"
#include "point.hpp"
#include "mathBonus.hpp"
#include <cmath>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <tuple>

Poly::Poly() {
    coefs_.push_back(0);
}

Poly::Poly(std::vector<double> coefs) : coefs_(coefs) {
    std::reverse(coefs_.begin(), coefs_.end());
}

Poly::Poly(size_t deg, double val) : coefs_(deg, val)
{}

Poly::Poly(std::string file, int c) {
    std::ifstream    is(file);
    int              deg;
    double           val;

    (void)c;
    is >> deg;
    for (int i = 0; i < deg; i++) {
        is >> val;
        coefs_.push_back(val);
    }
}

Poly::~Poly() {

}

Poly     puissance(const Poly& p, const int& n) {
    /* Return the polynomial p^n */
    if (n == 0)
        return (Poly{{1}});
    Poly     p2;
    p2 = p;
    for (int i = 1; i < n; i++) {
        p2 = p2 * p;
    }
    return (p2);
}

Poly     Poly::operator*(const double& a) const
{
    /* Return the polynomial p * a */
    Poly     p;

```

```
55      p.coefs_ = this->coefs_;
56      for (double& coef : p.coefs_) {
57          coef *= a;
58      }
59      return (p);
60  }
61
62  Poly    Poly::operator*(const Poly& p) const {
63      /* Return the polynomial p * p */
64      Poly    p2(p.coefs_.size() + this->coefs_.size(), 0.0);
65
66      for (size_t i = 0; i < this->coefs_.size(); i++) {
67          for (size_t j = 0; j < p.coefs_.size(); j++) {
68              p2.coefs_[i + j] += this->coefs_[i] * p.coefs_[j];
69          }
70      }
71      return (p2);
72  }
73
74  Poly    Poly::operator+(const double& a) const
75  {
76      /* Return the polynomial p + a */
77      Poly    p;
78
79      p.coefs_ = this->coefs_;
80      p.coefs_[0] += a;
81      return (p);
82  }
83
84  Poly    Poly::operator+(const Poly& p) const {
85      /* Return the polynomial p + p */
86      Poly    p2(std::max(p.coefs_.size(), this->coefs_.size()), 0.0);
87      size_t  i = 0;
88
89      while (i < this->coefs_.size() && i < p.coefs_.size()) {
90          p2.coefs_[i] = this->coefs_[i] + p.coefs_[i];
91          i++;
92      }
93      while (i < this->coefs_.size()) {
94          p2.coefs_[i] = this->coefs_[i];
95          i++;
96      }
97      while (i < p.coefs_.size()) {
98          p2.coefs_[i] = p.coefs_[i];
99          i++;
100     }
101     return (p2);
102 }
103
104 double  Poly::operator()(double x) const {
105     /* Return the value of the polynomial at x */
106     double res = 0;
107
108     for (size_t i = 0; i < coefs_.size(); i++) {
109         res += pow(x, i) * coefs_[i];
```

```
110             // std::cout << pow(x, i) * coefs_[i] << std::endl;
111         }
112         return (res);
113     }
114
115     std::ostream& operator<<(std::ostream &os, const Poly &p)
116     {
117         for (size_t i = 0; i < p.coefs_.size(); i++) {
118             if (p.coefs_[i] != 0) {
119                 os << p.coefs_[i] << " * X^" << i << " + ";
120             }
121         }
122         std::cout << std::endl;
123         return (os);
124     }
125
126     std::tuple<Poly, Poly>  BezierPoly(std::vector<Point<double>> points)
127     {
128         /* Return the Bezier polynomial of the points */
129         Poly    px;
130         Poly    py;
131         size_t  n = points.size();
132
133         for (size_t i = 0; i < n; i++) {
134             px = px + (puissance(Poly{{-1, 1}}, n - 1 - i) * points[i].getX() *
        puissance(Poly{{1, 0}}, i) * comb(i, n - 1));
135             py = py + (puissance(Poly{{-1, 1}}, n - 1 - i) * points[i].getY() *
        puissance(Poly{{1, 0}}, i) * comb(i, n - 1));
136         }
137         return (std::make_tuple(px, py));
138     }
139
140
141     void    Poly::savePoly(std::string name) {
142         std::ofstream   os(name);
143
144         os << coefs_.size() << " ";
145         for (int i = 0; i < coefs_.size(); i++) {
146             os << coefs_[i] << " ";
147         }
148     }
```

fichierCPP/polynome.cpp

```cpp
#include "projet.hpp"
#include "indice.hpp"
#include "mathBonus.hpp"
#include <boost/program_options.hpp>
#include <boost/math/constants/constants.hpp>
#include <SDL2/SDL_ttf.h>
#include <functional>

TTF_Font *my_font = NULL;

Info::Info(bool fullScreen) : S(fullScreen), show(true) {
    TTF_Init();
    my_font = TTF_OpenFont("ressource/HelveticaNeue.ttc", 35);
    setMouseInfo(0, 0, false);
    click = false;
    colorMap = false;
}

Info::~Info() {}

void            Info::addVirtualScreen(Flag f) {
    std::cout << "APPELLE ICI" << std::endl;
    Vscreens.emplace_back(S, f);
    std::cout << "add virtual screen" << std::endl;
    std::cout << "size of Vscreens: " << Vscreens.size() << std::endl;
    std::cout << "size of virtual screen: " << Vscreens.back().getVirtualW() <<
    std::endl;
    std::cout << "size of virtual screen: " << Vscreens.back().getVirtualH() <<
    std::endl;
    screen = &Vscreens.back();
}

VirtualScreen   *Info::getCurrentScreen() const {
    return (screen);
}

void            Info::selectScreen(int index) {
    screen = &(Vscreens[index]);
}

Timer           &Info::getTimer() {
    return (timer);
}

void            Info::hideOrShowAlgo(bool show_) {
    show = show_;
}

void            Info::setMouseInfo(int x, int y, bool m) {
    mouse_x = x;
    mouse_y = y;
    mouseMooved = m;
}

```

```
53  void            Info::setClickInfo(int x, int y, int c) {
54      mouse_x = x;
55      mouse_y = y;
56      click = c;
57  }
58
59  bool            Info::showAlgo() const {
60      return (show);
61  }
62
63  int             Info::getHighDPI() const {
64      return (S.getHighDPI());
65  }
66
67  int             Info::getMouseX() const {
68      return (mouse_x);
69  }
70
71  int             Info::getMouseY() const {
72      return (mouse_y);
73  }
74
75  bool            Info::getMouseMooved() const {
76      return (mouseMooved);
77  }
78
79  int             Info::getClicked() const {
80      return (click);
81  }
82
83  std::vector<Point<double>>  Info::getBezierPoints() const {
84      return (bezierPoints);
85  }
86
87  void                        Info::setBezierPoints(std::vector<Point<double>>
        points) {
88      bezierPoints = points;
89      tie(xt, yt) = bezierCurvePoly(bezierPoints);
90  }
91
92  bool            Info::getColorMap() const {
93      return (colorMap);
94  }
95
96  void            Info::setColorMap(bool b) {
97      colorMap = b;
98  }
99
100 std::function<double(double)>   Info::getXT() const {
101     return (xt);
102 }
103
104 std::function<double(double)>   Info::getYT() const {
105     return (yt);
106 }
```

```
107
108 void                Info::setXT(std::function<double(double)> f) {
109     xt = f;
110 }
111
112 void                Info::setYT(std::function<double(double)> f) {
113     yt = f;
114 }
```

fichierCPP/projet.cpp

```cpp
#include "screen.hpp"
#include <vector>

Screen::Screen(bool fullScreen) {
    int w;
    int h;

    SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER);
    if (fullScreen) {
        window = SDL_CreateWindow("Bezier", 0, 0, 0, 0,
    SDL_WINDOW_FULLSCREEN_DESKTOP | SDL_WINDOW_ALLOW_HIGHDPI);
    } else {
        window = SDL_CreateWindow("Bezier", SDL_WINDOWPOS_CENTERED,
    SDL_WINDOWPOS_CENTERED, 800, 800, SDL_WINDOW_ALLOW_HIGHDPI);
    }
    render = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED |
    SDL_RENDERER_TARGETTEXTURE);
    SDL_SetRenderDrawBlendMode(render, SDL_BLENDMODE_BLEND);
    SDL_GL_GetDrawableSize(window, &window_w, &window_h);
    SDL_GetWindowSize(window, &w, &h);
    highDPI = window_w / w;
    ratio = window_w / static_cast<double>(window_h);
}

Screen::~Screen() {
    SDL_DestroyRenderer(render);
    SDL_DestroyWindow(window);
}

int             Screen::getWindowW() const {
    return (window_w);
}

int             Screen::getWindowH() const {
    return (window_h);
}

double          Screen::getRatio() const {
    return (ratio);
}

SDL_Renderer    *Screen::getRenderer() const {
    return (render);
}

int             Screen::getHighDPI() const {
    return (highDPI);
}

VirtualScreen::VirtualScreen(const Screen &screen, Flag flag) : S(screen) {
    virtualRect.x = 0;
    virtualRect.y = 0;
    virtualRect.w = S.window_w;
    virtualRect.h = S.window_h;
```

```
52      if (flag == Flag::up) {
53          virtualRect.h /= 2;
54      } else if (flag == Flag::down) {
55          virtualRect.h /= 2;
56          virtualRect.y = virtualRect.h;
57      }
58      if (flag == Flag::left) {
59          virtualRect.w /= 2;
60      } else if (flag == Flag::right) {
61          virtualRect.w /= 2;
62          virtualRect.x = virtualRect.w;
63      }
64      ratio = virtualRect.w / static_cast<double>(virtualRect.h);
65      texture = SDL_CreateTexture(S.render, SDL_PIXELFORMAT_RGBA8888,
    SDL_TEXTUREACCESS_TARGET, virtualRect.w, virtualRect.h);
66  }
67
68  VirtualScreen::~VirtualScreen() {
69      // SDL_DestroyTexture(texture);
70  }
71
72  void     VirtualScreen::createPlan(Point<double> planUL, Point<double> planDR) {
73      /* Create a plan */
74      this->planUL = planUL;
75      this->planDR = planDR;
76      planW = abs(planUL.getX() - planDR.getX());
77      planH = abs(planUL.getY() - planDR.getY());
78  }
79
80  Point<int>  VirtualScreen::convPoint(Point<double> p) {
81      /* Convert a point from the plan to the screen */
82      double  ratioX = (p.getX() - planUL.getX()) / planW;
83      double  ratioY = -(p.getY() - planUL.getY()) / planH;
84      Point<int> p_(ratioX * virtualRect.w, ratioY * virtualRect.h);
85      return (p_);
86  }
87
88  Point<double>   VirtualScreen::convPoint(Point<int> p) {
89      /* Convert a point from the screen to the plan */
90      double  ratioX = p.getX() / static_cast<double>(virtualRect.w);
91      double  ratioY = p.getY() / static_cast<double>(virtualRect.h);
92      Point<double> p_(ratioX * planW + planUL.getX(), -ratioY * planH + planUL.
    getY());
93      // Point<double> p_(ratioX * static_cast<double>(planW) + static_cast<double
    >(planUL.getX()),
94      //          ratioY * static_cast<double>(planH) - static_cast<double>(planUL
    .getY()));
95      return (p_);
96  }
97
98  SDL_Point   VirtualScreen::convPointSDL(Point<double> p) {
99      /* Convert a point from the plan to the screen */
100     double  ratioX = (p.getX() - planUL.getX()) / planW;
101     double  ratioY = -(p.getY() - planUL.getY()) / planH;
102     SDL_Point p_ = {static_cast<int>(ratioX * virtualRect.w), static_cast<int>(
```

```
    ratioY * virtualRect.h)};
103     return (p_);
104 }
105
106 void        VirtualScreen::startDraw() {
107     /* Start drawing on the virtual screen */
108     SDL_SetRenderTarget(S.render, texture);
109     SDL_SetRenderDrawColor(S.render, 255, 255, 255, 255);
110     SDL_RenderClear(S.render);
111 }
112
113 void        VirtualScreen::renderPresent() {
114     /* Present the virtual screen on the screen */
115     SDL_SetRenderTarget(S.render, NULL);
116     SDL_RenderCopy(S.render, texture, NULL, &virtualRect);
117 }
118
119 int         VirtualScreen::getVirtualW() const {
120     return (virtualRect.w);
121 }
122
123 int         VirtualScreen::getVirtualH() const {
124     return (virtualRect.h);
125 }
126
127 SDL_Renderer    *VirtualScreen::getRenderer() const {
128     return (S.render);
129 }
130
131 double          VirtualScreen::getRatio() const {
132     return (ratio);
133 }
```

fichierCPP/screen.cpp

```cpp
#include "timer.hpp"

Timer::Timer()
{
    startTicks = 0;
    pausedTicks = 0;
    paused = false;
    started = false;
}

void Timer::start()
{
    started = true;
    paused = false;
    startTicks = SDL_GetTicks();
}

void Timer::stop()
{
    started = false;
    paused = false;
}

int Timer::get_ticks()
{
    if (started == true)
    {
        if (paused == true)
            return pausedTicks;
        else
            return SDL_GetTicks() - startTicks;
    }
    return 0;
}

void Timer::pause()
{
    if (started == true && paused == false)
    {
        paused = true;
        pausedTicks = SDL_GetTicks() - startTicks;
    }
}

void Timer::unpause()
{
    if (paused == true)
    {
        paused = false;
        startTicks = SDL_GetTicks() - pausedTicks;
        pausedTicks = 0;
    }
}

```

```
55 bool Timer::is_started()
56 {
57     return started;
58 }
59
60 bool Timer::is_paused()
61 {
62     return paused;
63 }
```

fichierCPP/timer.cpp

```cpp
#include "draw.hpp"
#include "projet.hpp"
#include "point.hpp"
#include "polynome.hpp"
#include "polygone.hpp"
#include "mathBonus.hpp"
#include "screen.hpp"
#include "indice.hpp"
#include "parametricPlot.hpp"
#include <iostream>
#include <vector>
#include "SDL2_gfxPrimitives.h"

void    twoColorMapLoop(Info& info) {
    /* Illustrate the two color theoreme */
    VirtualScreen   *screen;
    screen = info.getCurrentScreen();

    Timer       fps;
    std::function<double(double)> xt = info.getXT();
    std::function<double(double)> yt = info.getYT();
    PPlot       parametricPlot(xt, yt);

    info.getTimer().start();
    screen->startDraw();
    SDL_SetRenderDrawColor(screen->getRenderer(), 255, 255, 255, 255);
    SDL_RenderClear(screen->getRenderer());
    for (int i = 0; i < screen->getVirtualW(); i += 6) {
        for (int j = 0; j < screen->getVirtualH(); j += 6) {
            Point<double> p = screen->convPoint(Point<int>{i, j});
            int indice = calcIndice(xt, yt, p);
            if (indice % 2 == 0) {
                SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 255, 255);
            } else {
                SDL_SetRenderDrawColor(screen->getRenderer(), 255, 0, 0, 255);
            }
            drawPoint(*screen, {i, j}, 6);
            // screen->renderPresent();
        }
        if (i % (screen->getVirtualW() / 100) == 0) {
            std::cout << i << std::endl;
        }
    }
    SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 0, 255);
    parametricPlot.plot(*screen, 0, 1, 12);
    screen->renderPresent();
    SDL_RenderPresent(screen->getRenderer());
    while (handleEvent(noneFunction, noneFunction, info))
    {
        fps.start();
        SDL_Delay(fmax(0, (1000 / 30) - fps.get_ticks()));
    }
}

```

```
55 bool     findPointNotColored(std::vector<std::vector<Case>>& map, int&x, int &y)
       {
56     for (size_t i = 0; i < map.size(); i++) {
57         for (size_t j = 0; j < map[i].size(); j++) {
58             if (map[i][j].colored == false) {
59                 x = i;
60                 y = j;
61                 return true;
62             }
63         }
64     }
65     return false;
66 }
67
68 void     colorMapRecursive(std::vector<std::vector<Case>>& map, int x, int y, int
       indice) {
69     if (x < 0 || x >= static_cast<int>(map.size()) || y < 0 || y >= static_cast<
       int>(map[x].size())) {
70         return;
71     }
72     if (map[x][y].colored == true) {
73         return;
74     }
75     map[x][y].colored = true;
76     map[x][y].indice = indice;
77     colorMapRecursive(map, x - 1, y, indice);
78     colorMapRecursive(map, x, y - 1, indice);
79     colorMapRecursive(map, x + 1, y, indice);
80     colorMapRecursive(map, x, y + 1, indice);
81 }
82
83 void     colorMapIterative(std::vector<std::vector<Case>>& map, int x, int y, int
       indice) {
84     std::vector<Point<int>> toColor;
85     toColor.push_back(Point<int>{x, y});
86     while (toColor.size() > 0) {
87         Point<int>  p = toColor.back();
88         toColor.pop_back();
89         if (p.getX() < 0 || p.getX() >= static_cast<int>(map.size()) || p.getY()
       < 0 || p.getY() >= static_cast<int>(map[p.getX()].size())) {
90             continue;
91         }
92         if (map[p.getX()][p.getY()].colored == true || map[p.getX()][p.getY()].
       onCurve == true) {
93             continue;
94         }
95         map[p.getX()][p.getY()].colored = true;
96         map[p.getX()][p.getY()].indice = indice;
97         toColor.push_back(Point<int>{p.getX() - 1, p.getY()});
98         toColor.push_back(Point<int>{p.getX(), p.getY() - 1});
99         toColor.push_back(Point<int>{p.getX() + 1, p.getY()});
100        toColor.push_back(Point<int>{p.getX(), p.getY() + 1});
101    }
102 }
103
```

```
104 void    twoColorMapLoopImprove(Info& info) {
105     /* Illustrate the two color theoreme improve algorithm */
106     VirtualScreen   *screen;
107     screen = info.getCurrentScreen();
108 
109     Timer       fps;
110     std::function<double(double)> xt = info.getXT();
111     std::function<double(double)> yt = info.getYT();
112     PPlot       parametricPlot(xt, yt);
113     std::vector<std::vector<Case>>  map(screen->getVirtualW(), std::vector<Case
    >(screen->getVirtualH()));
114 
115     parametricPlot.plotOnMap(*screen, 0, 1, map);
116     info.getTimer().start();
117     screen->startDraw();
118     int x;
119     int y;
120     while(findPointNotColored(map, x, y)) {
121         Point<double> p = screen->convPoint(Point<int>{x, y});
122         int indice = calcIndice(xt, yt, p);
123         std::cout << indice << std::endl;
124         colorMapIterative(map, x, y, indice);
125     }
126 
127     SDL_SetRenderDrawColor(screen->getRenderer(), 255, 255, 255, 255);
128     SDL_RenderClear(screen->getRenderer());
129     screen->startDraw();
130     for (size_t i = 0; i < map.size(); i++) {
131         for (size_t j = 0; j < map[i].size(); j++) {
132             if (map[i][j].onCurve == true) {
133                 SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 0, 255);
134                 drawPoint(*screen, {static_cast<int>(i), static_cast<int>(j)},
    1);
135             } else if (map[i][j].colored == true) {
136                 if (abs(map[i][j].indice) % 2 == 0) {
137                     SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 255,
    255);
138                 } else {
139                     SDL_SetRenderDrawColor(screen->getRenderer(), 255, 0, 0,
    255);
140                 }
141                 drawPoint(*screen, {static_cast<int>(i), static_cast<int>(j)},
    1);
142             }
143         }
144     }
145     screen->renderPresent();
146     SDL_RenderPresent(screen->getRenderer());
147     // SDL_SetRenderDrawColor(screen->getRenderer(), 0, 0, 0, 255);
148     // parametricPlot.plot(*screen, 0, 1, 12);
149     while (handleEvent(noneFunction, noneFunction, info))
150     {
151         fps.start();
152         SDL_Delay(fmax(0, (1000 / 30) - fps.get_ticks()));
153     }
```

```
154 }
```

fichierCPP/twoColorMap.cpp