Rapport – Projet tutoré

Portraits de phase de fonctions complexes :
calcul en C, compilateur et application web

Étienne Collet

Encadrant : Olivier Bouillot

Université Gustave Eiffel

28 octobre 2026

Remerciements

Je souhaite remercier :

Olivier Bouillot,
pour son suivi ainsi que pour le temps qu'il m'a consacré.

Table des matières

1 Introduction **4**

2 Les outils mis en jeu **5**
2.1 L'outil PhasePortrait 5
2.1.1 Portrait de phase 5
2.1.2 L'application 5
2.2 La bibliothèque ctypes 7
2.2.1 Description générale 7
2.2.2 Utilisation 7
2.2.3 Points forts et points faibles 8

3 L'optimisation de la vitesse de calcul **9**
3.1 Les changements apportés au code 9
3.1.1 État des lieux 9
3.1.2 Démarche suivie 9
3.1.3 Le script `fct_to_phase_portrait_update` 9
3.1.4 La bibliothèque `tools.so` 9
3.1.5 Les changements apportés au code Python 12
3.2 Résultats obtenus 13
3.2.1 Autour de la fonction `draw` 14
3.2.2 Autour de la fonction `compute` 14

4 Le compilateur ppcomp **16**
4.1 État des lieux et objectifs 16
4.2 Flex : les lexèmes et les tokens 16
4.2.1 Les lexèmes fondamentaux 16
4.2.2 Les lexèmes optionnels 17
4.2.3 Des lexèmes aux tokens 17
4.3 Bison : la grammaire 17
4.3.1 Description de la grammaire 17
4.3.2 Le programme obtenu 18
4.4 La traduction de l'arbre syntaxique en C 19
4.5 Résultats obtenus 19
4.5.1 Exemple de programme 19
4.5.2 Résultat en C 19
4.6 Limites connues du compilateur 21

5 Refonte de l'application **22**
5.1 Refonte de l'arborescence 22
5.2 Refonte graphique 23

6 Conclusion et perspectives **24**

A Annexes **25**
A.1 Mesures sur les opérations élémentaires 25
A.2 Glossaire 25

Bibliographie **26**

Liste des tableaux

1 Régressions quadratiques du temps de `draw` (en secondes) 14
2 Régressions quadratiques du temps de `compute` (en secondes) 14
3 Coût d'une génération complète pour $z \mapsto \frac{z^2-1}{z^2+1}$, compilation comprise 23
4 Les offres de l'application 23
5 Temps moyen d'une opération sur des entiers, en Python et via `ctypes` 25

Table des figures

1 Représentation HLS sous forme de cylindre 5
2 Portrait de phase de $z \mapsto z$ sur $[-5,5]+i[-5,5]$ 6
3 Capture de l'outil PhasePortrait 6
4 Portrait de phase annoté 6
5 Portrait de phase de la fonction f sur $[-2,2]+i[-2,2]$ 13
6 Portraits de phase de la fonction Γ 13
7 Temps d'exécution de `draw` selon le nombre de pixels par unité 14
8 Temps d'exécution de `compute` selon le nombre de pixels par unité 14
9 Arborescence du projet après la refonte 22

1 Introduction

Lors de nos premières discussions sur mon projet tutoré, nous avons parlé de l'application PhasePortrait, un outil qui permet de visualiser et de manipuler facilement les représentations graphiques des fonctions $f : \mathbb{C} \to \mathbb{C}$, comme on le ferait avec le graphe d'une fonction $f : \mathbb{R} \to \mathbb{R}$.

Les fonctions complexes sont des objets mathématiques importants, utilisés en algèbre, en physique et dans de nombreux autres domaines. Les visualiser est difficile, car leur graphe vit dans un espace à quatre dimensions. C'est pourquoi on le représente par une image en couleurs : la position du pixel code deux dimensions (le point z) et sa couleur code les deux autres (la valeur $f(z)$).

Cette application utilise des nombres complexes enrichis d'un point à l'infini, qui intervient dans certains calculs : $1/0 = \infty$ et $c \times \infty = \infty$ si $c \neq 0$. Ces nombres sont appelés *RiemannSphere*, en hommage au mathématicien Bernhard Riemann. Un tel nombre est décrit par sa partie réelle, sa partie imaginaire et un troisième paramètre qui indique s'il est infini ; dans ce cas, les parties réelle et imaginaire n'ont pas de signification.

Les portraits de phase sont une grande source d'information sur la fonction associée : ils révèlent la présence ou l'absence de zéros et de pôles sur une partie du plan. Ils permettent ainsi au mathématicien de valider ses intuitions ou de formuler de nouvelles conjectures.

Cependant, le calcul de la valeur de chaque pixel et l'affichage de l'image sont très coûteux en temps. Pour un carré unité de n pixels de côté, ces opérations ont une complexité en $\mathcal{O}(f \cdot n^2)$, où f est la complexité d'une évaluation de la fonction étudiée. Ce facteur intervient car il ne faut pas le même temps pour calculer $z \mapsto z$ et $z \mapsto \sin(z)$; le facteur n^2 vient des boucles imbriquées qui parcourent chaque pixel de l'image.

La problématique que j'ai retenue est donc de réduire le temps d'exécution des fonctions les plus coûteuses, afin d'obtenir plus rapidement des portraits avec plus de pixels par unité, ou sur une plus grande partie du plan. Le projet s'est ensuite prolongé par l'écriture d'un compilateur, pour que l'utilisateur n'ait plus à programmer en C, puis par la refonte de l'application sous la forme d'une application web.

Je présenterai d'abord les outils mis en jeu : l'application, son fonctionnement et la bibliothèque `ctypes`. Je détaillerai ensuite les optimisations apportées au calcul et les gains obtenus, puis le compilateur `ppcomp`, et enfin la refonte de l'application. Je terminerai par un bilan et les perspectives d'amélioration.

2 Les outils mis en jeu

2.1 L'outil PhasePortrait

L'outil PhasePortrait facilite l'étude des représentations graphiques d'une fonction complexe $f : \mathbb{C} \to \mathbb{C}$. Il permet de sauvegarder les résultats obtenus et de changer rapidement les paramètres de l'image : sa résolution, sa largeur et sa hauteur.

2.1.1 Portrait de phase

La représentation HLS. La représentation HLS est une représentation des couleurs alternative à RGB, sous forme de cylindre (voir figure 1). Elle repose sur trois paramètres : la teinte, la clarté et la saturation. La teinte s'exprime en degrés : elle commence à 0° avec le rouge, passe par le jaune, le vert, le bleu, puis le violet, et revient au rouge à 360°. La clarté est l'axe vertical : elle est comprise dans $[0, 1]$, la valeur 0 donnant du noir et la valeur 1 du blanc.

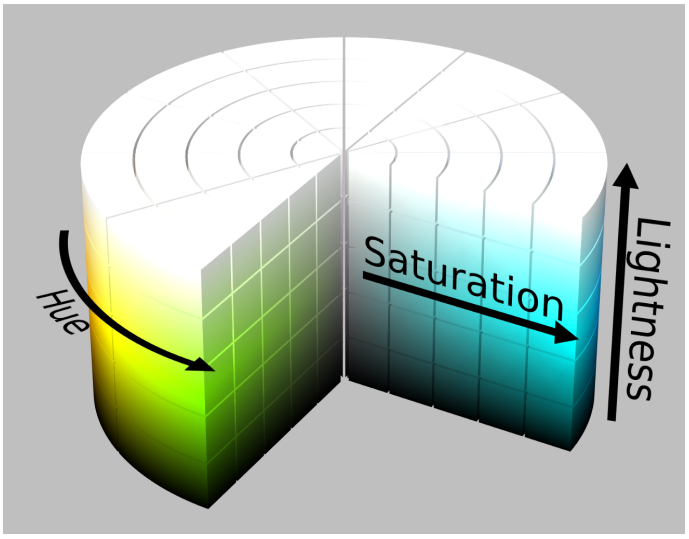


FIGURE 1 – Représentation HLS sous forme de cylindre (source : Wikipédia [6])

Calcul de la couleur. Pour colorer le pixel (a, b), on lui associe le nombre complexe $z = a + ib$, puis on calcule la teinte H, la clarté L et la saturation S à partir de $f(z)$:

$$H = \arg(f(z)) \tag{1}$$

$$L = \frac{1}{2}\left(\frac{\ln|f(z)|}{1 + |\ln|f(z)||} + 1\right) \tag{2}$$

$$S = 1 \tag{3}$$

Le portrait de phase d'une fonction $f : \mathbb{C} \to \mathbb{C}$ est alors l'image dont le pixel de coordonnées (a, b) est coloré en HLS par les formules (1), (2) et (3), selon la correspondance $(a, b) \mapsto z \mapsto f(z)$. Par exemple, la figure 2 est le portrait de phase de la fonction identité $z \mapsto z$ pour $z \in [-5, 5] + i[-5, 5]$: la couleur donne directement l'argument de z.

2.1.2 L'application

L'outil se compose de huit parties (voir figure 3) : les parties 1 à 4 modifient les paramètres de calcul, les parties 6 et 7 servent à créer et manipuler l'image (génération, sauvegarde, visualisation), la partie 5 affiche l'image et la partie 8 affiche les journaux de calcul.

Un portrait de phase contient beaucoup d'informations sur la fonction étudiée ; la figure 4 en annote une partie.

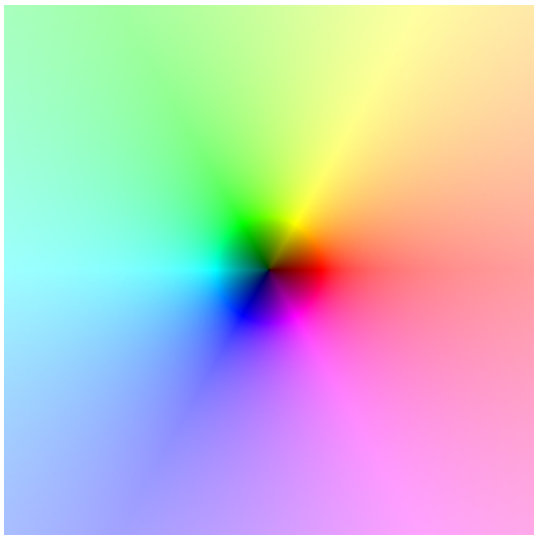


FIGURE 2 – Portrait de phase de $z \mapsto z$ sur $[-5,5]+i[-5,5]$

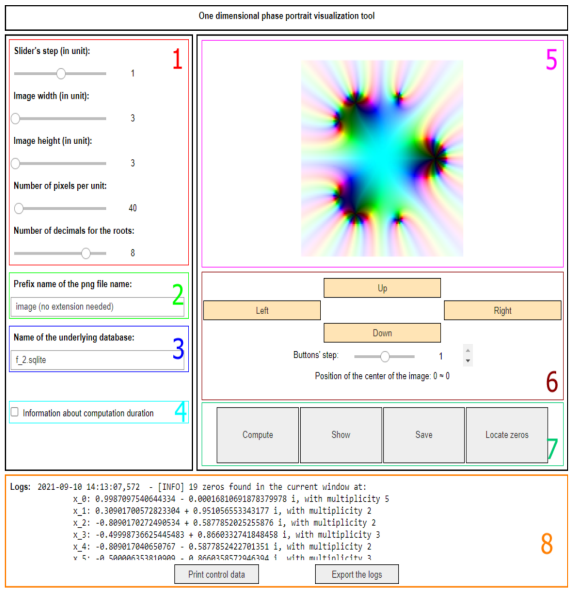


FIGURE 3 – Capture de l'outil PhasePortrait. Source : documentation de l'outil

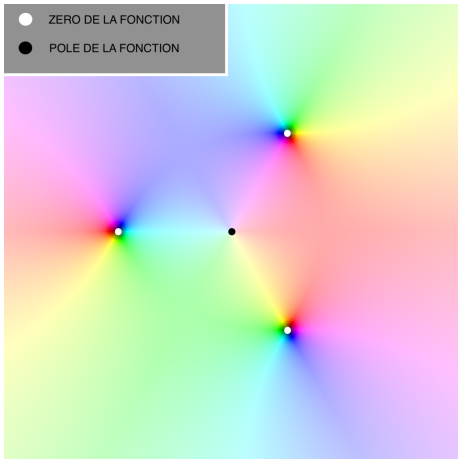


FIGURE 4 – Portrait de phase de $f : z \mapsto \frac{4}{z}\left(z^3+1\right)$ sur $[-2,2]+i[-2,2]$, annoté

2.2 La bibliothèque ctypes

2.2.1 Description générale

`ctypes` est un module de la bibliothèque standard de Python qui permet d'appeler, depuis du code Python, les fonctions d'une bibliothèque C. Rappelons qu'il existe trois façons d'utiliser une bibliothèque :

- **statique** : la bibliothèque est incorporée au programme lors de l'édition de liens, pendant la compilation. Lorsqu'elle est modifiée, chaque programme qui l'utilise doit être recompilé ;
- **partagée** : la bibliothèque n'est pas copiée dans les programmes, mais liée à eux au lancement. Elle peut servir à de nombreux programmes, et sa modification ne nécessite pas de les recompiler ;
- **dynamique** : c'est une bibliothèque partagée que le programme charge lui-même, au moment où il en a besoin.

C'est ce dernier mode qu'utilise `ctypes` : les bibliothèques sont chargées pendant l'exécution par l'objet `cdll`.

2.2.2 Utilisation

Côté C. Pour utiliser `ctypes`, on commence par créer une bibliothèque partagée, par exemple `tools.so`, qui contient les fonctions C à appeler depuis Python. Voici un exemple de source possible :

```c
/* tools.c */
#include <stdlib.h>

int somme(int a, int b){
    return a + b;
}

int * sommeAlloc(int a, int b){
    int *res = (int *) malloc(sizeof(int));
    if (NULL != res){
        *res = a + b;
    }
    return res;
}

void libereSommeAlloc(int * res){
    free(res);
}
```

On compile ensuite ce fichier en bibliothèque partagée avec la commande :

```
gcc -ansi -Wall -pedantic -shared -O2 -fPIC -o tools.so tools.c        (4)
```

Cette commande ne compile qu'un seul fichier. Pour notre bibliothèque, un `makefile` compile chaque module séparément en fichier objet, puis les rassemble en une bibliothèque avec les mêmes options.

Les options passées à gcc. Voici le rôle des options de la commande (4) :

`-ansi` applique la norme ANSI C ;

`-Wall` active la plupart des avertissements ;

`-pedantic` signale tout écart à la norme ISO C ;

`-O2` active presque toutes les optimisations qui n'impliquent pas de compromis entre espace et vitesse (par exemple, gcc ne déroule pas les boucles à ce niveau). Par rapport à `-O`, cette option allonge la compilation mais accélère le code produit ;

`-shared` produit un objet partagé, qui peut être lié à d'autres objets pour former un exécutable ;

`-fPIC` produit du code indépendant de sa position en mémoire, indispensable pour une bibliothèque chargée dynamiquement.

Côté Python. Une fois la bibliothèque compilée, on l'utilise dans un script Python : on importe `ctypes`, on charge la bibliothèque avec `cdll.LoadLibrary`, qui prend le chemin du fichier, puis on déclare les types des paramètres et du résultat des fonctions utilisées, avant de les appeler.

Déclarer ces types est important : sans eux, `ctypes` suppose que la fonction renvoie un `int` et convertit les paramètres à l'aveugle, ce qui peut provoquer des conversions ou des débordements silencieux côté C, et donc des calculs faux.

```
from ctypes import *
from random import randrange

_lib = cdll.LoadLibrary("./tools.so")
_lib.somme.argtypes = [c_int, c_int]
_lib.somme.restype = c_int
M = int(1e9)
x = randrange(M)
y = randrange(M)
a = _lib.somme(x, y)
print("addition ctypes : " + str(x) + " + " + str(y) + " = " + str(a))
print("résultat Python : " + str(x + y))
```

Sur un code aussi court, qui ne réalise qu'une opération en C, `ctypes` n'apporte rien : le coût de l'appel allonge même le temps d'exécution (voir l'annexe A.1).

Pour les pointeurs, `ctypes` fournit le type `POINTER(type)`. Comme Python ne connaît pas la mémoire allouée par le C, toute allocation faite en C doit être libérée en C :

```
from ctypes import *

_lib = cdll.LoadLibrary("./tools.so")
_lib.sommeAlloc.argtypes = [c_int, c_int]
_lib.sommeAlloc.restype = POINTER(c_int)
_lib.libereSommeAlloc.argtypes = [POINTER(c_int)]
_lib.libereSommeAlloc.restype = None

a = _lib.sommeAlloc(2, 3)
print("addition = " + str(a[0]))
_lib.libereSommeAlloc(a)
```

2.2.3 Points forts et points faibles

- Points forts :
  - `ctypes` réduit fortement le temps d'exécution des calculs lourds ;
  - il donne un contrôle fin de la mémoire.
- Points faibles :
  - le code compilé dépend du système d'exploitation et de l'architecture ;
  - le module est délicat à utiliser, à cause des conversions de types entre Python et C et de la manipulation des pointeurs ;
  - il allonge le temps de développement, car il faut écrire et recompiler une bibliothèque (sans parler des erreurs dues à une bibliothèque qu'on a oublié de recompiler après une modification !) ;
  - les types du C sont bornés : un entier de l'ordre de 10^{20} dépasse la capacité d'un `int` comme d'un `unsigned long long`, alors que Python manipule des entiers de taille arbitraire.

3 L'optimisation de la vitesse de calcul

3.1 Les changements apportés au code

3.1.1 État des lieux

Le code d'origine, écrit uniquement en Python, n'utilisait pas `ctypes`. Il était composé de plusieurs modules et séparait le calcul des valeurs de la fonction de leur affichage dans l'image. La gestion des images reposait sur de multiples boucles imbriquées, ce qui coûte très cher en Python, en mémoire comme en temps. Les points à améliorer se trouvaient dans le fichier `phase_portrait.py`, en particulier les fonctions `compute` et `draw`, extrêmement coûteuses. De plus, la taille et la densité du code m'empêchaient d'en comprendre toutes les parties.

3.1.2 Démarche suivie

Nous avons donc réfléchi à la manière de supprimer ces pertes de temps, et l'idée retenue a été d'utiliser `ctypes`. J'ai d'abord expérimenté le module sur de petits programmes : opérations élémentaires et boucles seules.

Pour faciliter la suite, j'ai créé le fichier `phase_portrait_2_0.py`, une version simplifiée de `phase_portrait.py` sans la base de données qui sauvegardait les calculs. Cela m'a permis d'identifier les obstacles. Par exemple, les valeurs calculées étaient stockées dans un dictionnaire Python, une structure incompatible avec le C. Je l'ai remplacé par un tableau à deux dimensions, alloué et libéré en C.

J'ai ensuite réécrit en C la fonction `draw`, qui dessine le portrait de phase à partir des valeurs qui lui sont transmises. Pour la fonction `compute`, j'ai commencé avec une seule fonction à calculer, codée en dur. Pour pouvoir en changer facilement, il fallait transmettre une fonction en paramètre : je n'avais alors pas trouvé comment passer un pointeur de fonction avec `ctypes`.

J'ai donc écrit un script qui compile une fonction écrite en C et l'incorpore à une bibliothèque, ce qui permet de changer de fonction à volonté. Il a fallu pour cela compléter le module `CRiemannSphere.c` avec toutes les opérations sur les *RiemannSphere*, et créer le module `CSpecialFunctions.c`, qui regroupe les fonctions usuelles (sinus, exponentielle...).

Ce travail m'a amené à modifier `phase_portrait.py`, à créer la bibliothèque `tools.so` (aujourd'hui `CphasePortrait.so`) et le script `fct_to_phase_portrait_update`.

3.1.3 Le script `fct_to_phase_portrait_update`

Ce script transforme un fichier texte écrit en C en une bibliothèque partagée utilisée par `tools.so`. Le fichier doit contenir une unique fonction de prototype

```
int operation(RS z, RS * res);
```
(5)

dont le code utilise les fonctions des modules `CRiemannSphere.c` et `CSpecialFunctions.c` pour calculer un *RiemannSphere*. Le script recopie ce fichier dans un fichier C complet, en ajoutant les en-têtes nécessaires, puis le compile en une bibliothèque `operation.so`.

Par exemple, les deux fonctions suivantes calculent respectivement l'identité et le sinus d'un *RiemannSphere* :

```c
int operation(RS z, RS * res){
    return RS_copy(res, z);
}
```

```c
int operation(RS z, RS * res){
    return RS_sin(z, res);
}
```

3.1.4 La bibliothèque `tools.so`

Les sources de la bibliothèque se trouvent dans le répertoire `src_C`, les en-têtes dans `include`. Les modules qui la composent sont `Cimage`, `Ccolor`, `CRiemannSphere`, `CSpecialFunctions` et

`CUnitaryTesting`. Les fichiers `TestsColor.c`, `TestsCRiemannSphere.c` et `TestsCSpecialFunctions.c` exécutent une batterie de tests sur les modules correspondants. Le `makefile`, placé en dehors du répertoire des sources pour que le script Python puisse compiler lui-même la bibliothèque, permet aussi de compiler et lancer ces tests.

Le module `Ccolor`. Ce module manipule des couleurs aux formats HLS et RGB. Il convertit surtout un *RiemannSphere* en couleur HLS, puis cette couleur en RGB, grâce aux fonctions `int HSL(RS z, HSLColor * hsl)` et `int RGB(RS z, RGBColor * rgb)`, qui renvoient la couleur par adresse. Les deux formats sont représentés par les structures suivantes :

```
/** Couleur dans le modèle HLS */
typedef struct {
    double h;
    double l;
    double s;
} HSLColor;

/** Couleur dans le modèle RGB */
typedef struct {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
} RGBColor;
```

Le module `CRiemannSphere`. Ce module définit la structure *RiemannSphere*, ainsi que les fonctions pour l'initialiser, la manipuler et calculer avec :

```
typedef struct RiemannSphere{
    double re;
    double im;
    int infinite;
} RS;
```

L'essentiel du module est constitué des fonctions de calcul : addition, soustraction, multiplication, division... Elles ont presque toutes un prototype de la forme

```
int fun(Type arg1, ..., Type argn, RS * res);
```
(6)

où `Type` est un entier, un réel ou un *RiemannSphere*. Un ou deux arguments précèdent un pointeur sur un *RiemannSphere*, qui reçoit le résultat ; la valeur renvoyée vaut 0 en cas de succès et 1 en cas d'erreur. Seule `RS_copy` a un prototype différent, `int RS_copy(RS * dest, RS src)`, pour rester proche des fonctions de copie Unix.

Le module `Cimage`. Ce module alloue un tableau à deux dimensions de *RiemannSphere* avec la fonction `RS ** initTab(int x, int y)`, où `x` et `y` sont les dimensions de la matrice. Ce tableau reçoit les valeurs calculées, côté Python ou côté C, puis sert à dessiner l'image. La fonction `void libereTab(RS *** tab, int x)` le libère : comme indiqué à la section 2.2.2, toute allocation faite en C doit être libérée en C.

Le module contient aussi la fonction `RGBColor ** draw(RS ** values, int x, int y)`, qui alloue une image et donne à chaque pixel la couleur correspondant à sa valeur. Une fois l'image exploitée, il faut la libérer avec `void libereImage(RGBColor *** img, int x, int y)`. Enfin, la fonction `compute` remplit le tableau en évaluant, pour chaque pixel, la fonction qu'on lui passe en paramètre.

Le module `CSpecialFunctions`. Ce module contient les fonctions plus élaborées sur les *RiemannSphere* : sinus, cosinus, racine carrée, puissance, exponentielle, logarithme et fonction Gamma. Elles respectent toutes le prototype (6).

Le module `CUnitaryTesting`. Ce module contient des fonctions génériques pour tester les modules `Ccolor`, `CRiemannSphere` et `CSpecialFunctions`. Il a été écrit par M. Bouillot, afin de factoriser au maximum les tests que j'avais écrits.

Les modules de tests. `TestsColor`, `TestsCRiemannSphere` et `TestsCSpecialFunctions` testent chacune des fonctions du module correspondant, à l'aide des fonctions génériques de `CUnitaryTesting`. Ils reprennent les tests qui existaient dans les sources Python, en les approfondissant et en corrigeant ceux qui le nécessitaient.

3.1.5 Les changements apportés au code Python

Côté Python, j'ai d'abord créé une structure identique à celle de la bibliothèque, qui fait le lien entre les deux langages. À l'initialisation d'un objet `PhasePortrait`, j'ai ajouté le chargement de la bibliothèque et l'allocation du tableau qui stocke les valeurs de la fonction étudiée.

```
class C_RS(Structure):
    _fields_ = [("re", c_double),
                ("im", c_double),
                ("infinite", c_int)]
```

J'ai aussi réécrit la fonction `draw`, qui n'utilise plus que des fonctions C. Voici son corps :

```
img = Image.new('RGB', self.size, "white")
self._lib.draw.argtypes = [POINTER(POINTER(C_RS)), c_int, c_int]
self._lib.draw.restype = POINTER(POINTER(RGB_COLOR))
img_c = self._lib.draw(self.values, self.size[0], self.size[1])
rebuild_image(img_c, img, self.size)
self.img = img
# Libération de la mémoire allouée en C
self._lib.libereImage(byref(img_c), self.size[0], self.size[1])
```

3.2 Résultats obtenus

L'objectif était de réduire le temps de calcul et d'affichage des portraits de phase grâce à `ctypes`. Les temps d'exécution de la fonction `draw` ont été mesurés avec la fonction

$$f : z \mapsto \frac{4}{z}\left(z^3 + 1\right) \tag{7}$$

dont le portrait de phase est donné par la figure 5. Cette fonction n'est pas définie en $z = 0$, ce qui est bien signalé par un message d'erreur. Une évaluation de f demande un nombre constant de multiplications et de divisions : la fonction `draw` a donc une complexité en $\mathcal{O}(n^2)$, n étant le nombre de pixels par unité.

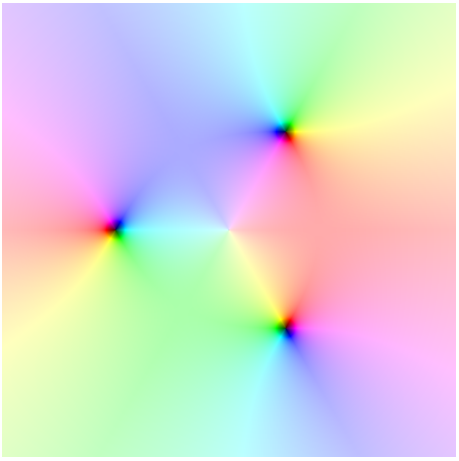
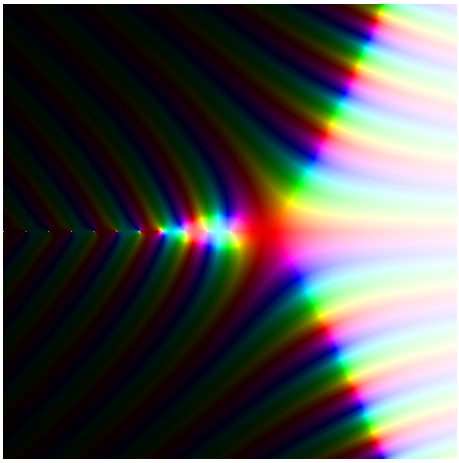


FIGURE 5 – Portrait de phase de la fonction f sur $[-2, 2] + i[-2, 2]$

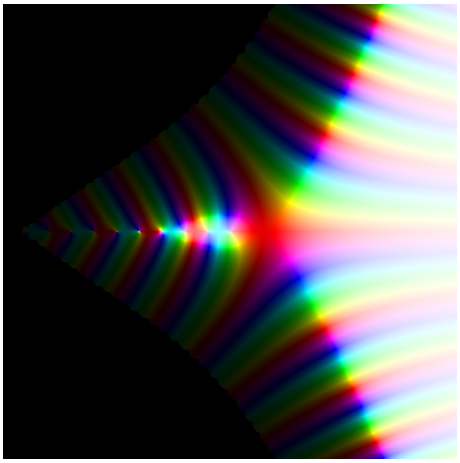
Les temps de la fonction `compute` ont été mesurés avec la fonction Gamma,

$$\Gamma(z) = \int_0^{+\infty} x^{z-1} e^{-x} \, \mathrm{d}x, \tag{8}$$

dont les portraits de phase, calculés en Python puis en C, sont donnés par la figure 6.



(a) Calcul réalisé en Python



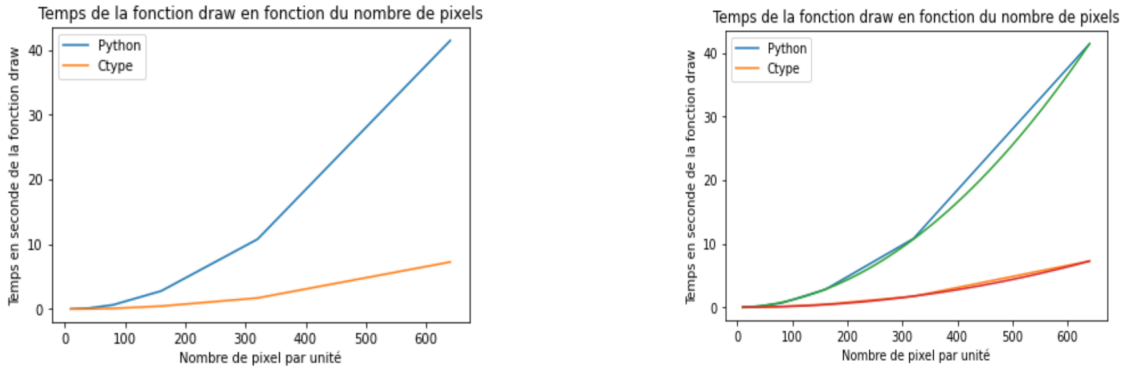
(b) Calcul réalisé en C

FIGURE 6 – Portraits de phase de la fonction Γ

On remarque que les deux images diffèrent loin de l'origine : les nombres flottants du C débordent pour les grandes valeurs de Γ, là où Python continue à calculer. Nous y reviendrons dans les perspectives.

3.2.1 Autour de la fonction `draw`

La fonction `draw` a une complexité en $\mathcal{O}(n^2)$, comme le confirme la figure 7. La nouvelle version ne change pas cette complexité, mais elle en réduit fortement la constante.



(a) Sans la régression (b) Avec la régression

FIGURE 7 – Temps d'exécution de `draw` selon le nombre de pixels par unité

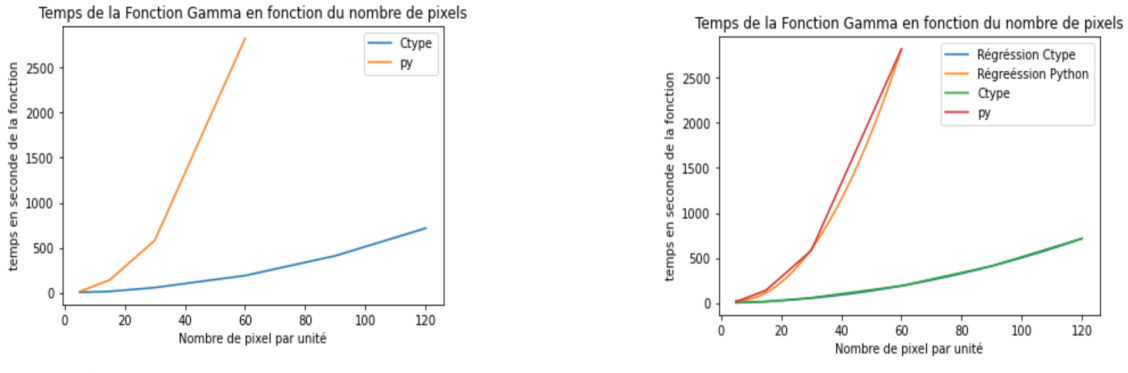
Python	$9{,}734 \cdot 10^{-5}x^2 + 2{,}564 \cdot 10^{-3}x - 7{,}364 \cdot 10^{-2}$
`ctypes`	$1{,}849 \cdot 10^{-5}x^2 - 5{,}285 \cdot 10^{-4}x + 2{,}126 \cdot 10^{-2}$

TABLE 1 – Régressions quadratiques du temps de `draw` (en secondes)

Le coefficient R^2 de ces deux régressions vaut environ 0,9999, ce qui est excellent. Le rapport des coefficients dominants montre qu'avec `ctypes`, la fonction `draw` est environ 5,3 fois plus rapide qu'en Python pur : c'est une amélioration nette.

3.2.2 Autour de la fonction `compute`

La fonction `compute` écrite en C accepte désormais n'importe quelle fonction ; les mesures ont été faites avec $z \mapsto \Gamma(z)$ (figure 8).



(a) Sans la régression (b) Avec la régression

FIGURE 8 – Temps d'exécution de `compute` selon le nombre de pixels par unité

Python	$0{,}9325x^2 - 9{,}779x + 50{,}54$
`ctypes`	$0{,}04649x^2 + 0{,}3822x + 0{,}2184$

TABLE 2 – Régressions quadratiques du temps de `compute` (en secondes)

Le coefficient R^2 de ces régressions vaut environ 0,9997, un peu moins que pour `draw` mais toujours très bon. Avec `ctypes`, la fonction `compute` est environ 20 fois plus rapide qu'en Python pur : c'est une excellente amélioration.

4 Le compilateur ppcomp

4.1 État des lieux et objectifs

Après l'optimisation, le calcul de l'image se fait en C, dans une bibliothèque dynamique. Pour étudier une nouvelle fonction, il faut donc l'écrire en C, avec les opérations de cette bibliothèque, puis la lui transmettre. Cela complique beaucoup la prise en main : l'utilisateur doit savoir programmer en C et connaître la bibliothèque que j'ai écrite.

L'idée est donc d'écrire un compilateur qui, à partir d'une expression arithmétique, produit la fonction C correspondante. L'utilisateur saisit simplement une expression comme `(_Z + _I 2) * 5` et voit l'image apparaître quelques secondes plus tard.

Pour écrire ce compilateur, j'ai utilisé Flex et Bison. Flex est un logiciel libre qui génère des analyseurs lexicaux : des programmes qui reconnaissent des motifs dans un texte. Bison génère des analyseurs syntaxiques : des programmes qui vérifient que le texte respecte une grammaire donnée. En les associant, et en ajoutant un programme C qui interprète l'arbre produit, on obtient le compilateur voulu.

4.2 Flex : les lexèmes et les tokens

Il faut d'abord définir les lexèmes que le compilateur doit reconnaître, puis les associer à des tokens, qui serviront de catégories de mots pour la grammaire. Par analogie, en français, les noms, les verbes ou les adjectifs sont des catégories de mots.

4.2.1 Les lexèmes fondamentaux

Un lexème est un motif lexical, décrit par une chaîne de caractères ou par une expression régulière. Comme le langage décrit des expressions arithmétiques, il doit d'abord reconnaître les opérateurs :

— l'addition ($+$) et la soustraction ($-$) ;

— la multiplication ($*$) et la division ($/$) ;

— la puissance ($\hat{}$).

Le modulo n'est pas proposé, car il n'a pas de sens pour les nombres complexes.

Il faut ensuite reconnaître les nombres. Comme nous l'avons vu (page 10), un nombre complexe est représenté par deux nombres flottants ; le compilateur garde donc la même précision :

— les nombres entiers sont décrits par `[0-9]+` ;

— les nombres décimaux sont décrits par `[0-9]+[.][0-9]+`.

Une fois lu, tout nombre est converti en flottant : le nombre 5 devient 5,0 dans le code produit.

Pour écrire des nombres complexes et les constantes usuelles, j'ai introduit les symboles suivants :

— `_I` pour i ;

— `_INF` pour l'infini ;

— `_E` pour e ;

— `_P` pour π.

Les parenthèses précisent l'ordre des opérations, et les accolades délimitent un nombre complexe :

$$\texttt{3 / ({2 + _I 3} * 2)} \iff 3/((2+3i)\times 2) \tag{9}$$

Le langage permet aussi de définir et d'appeler des fonctions, prédéfinies ou écrites par l'utilisateur. Une fonction de l'utilisateur ne prend qu'un seul paramètre, car le programme final ne sait calculer que des fonctions d'une variable. Les lexèmes correspondants sont :

— `[a-z]+[_a-zA-Z0-9]*` pour un nom de fonction : un mot qui commence par une lettre minuscule et se poursuit par des lettres, des chiffres ou des tirets bas ;

— `[A-Z]+[_A-Z0-9]*` pour un nom de variable : un mot qui commence par une lettre majuscule et se poursuit par des majuscules, des chiffres ou des tirets bas ;

— les fonctions prédéfinies : `_sqrt` (racine carrée), `_log` (logarithme), `_exp` (exponentielle), `_sin` (sinus), `_cos` (cosinus), `_abs` (module), `_sabs` (module au carré), `_conj` (conjugué), `_opp` (opposé) et `_inv` (inverse). Les lexèmes `_tan` et `_arg` sont reconnus, mais pas encore implémentés (voir section 4.6) ;

— `_Z`, la variable prédéfinie de la fonction principale ;

— `return`, pour la valeur à renvoyer.

4.2.2 Les lexèmes optionnels

Pour aller plus loin, j'ai repris quelques éléments du C. Les commentaires d'abord : `//` pour un commentaire sur une ligne, `/*` et `*/` pour un commentaire sur plusieurs lignes. Ensuite, les structures conditionnelles `if` et `else`, avec les opérateurs de comparaison qui leur donnent un sens :

— `&&` pour le ET logique ;

— `||` pour le OU logique ;

— `==` pour l'égalité ;

— `!=` pour la différence.

Les autres chaînes de caractères sont ignorées ou provoquent une erreur.

4.2.3 Des lexèmes aux tokens

Les lexèmes sont associés aux tokens suivants :

— `NUM` : les nombres entiers et décimaux ;

— `II` : `_I` ;

— `INF` : `_INF` ;

— `EXPI` : `_E` et `_P` ;

— `ADDSUB` : $+$ et $-$;

— `DIVSTAR` : $*$, / et $\hat{}$;

— `VARIA` : les noms de variables, dont `_Z` ;

— `FUNCT` : les noms de fonctions, dont les fonctions prédéfinies ;

— `RETURN`, `IF`, `ELSE` : les mots-clés correspondants ;

— `AND`, `OR` : `&&` et `||` ;

— `EQ` : `==` et `!=`.

Comme $*$, / et $\hat{}$ partagent le même token, ils ont la même priorité et sont évalués de gauche à droite : `2 * _Z ^ 2` vaut $(2z)^2$ et non $2z^2$. Il faut donc écrire `2 * (_Z ^ 2)`.

4.3 Bison : la grammaire

4.3.1 Description de la grammaire

À partir des tokens, la grammaire définit les enchaînements valides : un texte qui ne la respecte pas provoque une erreur de syntaxe, comme la phrase « mange pomme Paul la » en français. L'analyseur construit aussi l'arbre syntaxique du programme, qui facilite sa traduction en C.

Un programme se compose de deux parties : la définition facultative de fonctions auxiliaires, puis la fonction principale, que le reste de l'application évalue sur chaque pixel. Commençons par trois éléments de grammaire qui servent partout.

Les éléments fondamentaux.

— `Complexe` représente un nombre complexe, sous l'une des six formes suivantes :
  — `'{' NUM ADDSUB II NUM '}'`, équivalent à $a \pm b\,i$;
  — `II NUM`, équivalent à $b\,i$;
  — `NUM`, équivalent à a ;
  — `II`, équivalent à i ;
  — `EXPI`, équivalent à e ou π ;
  — `INF`, l'infini.

— `Exp` représente une expression arithmétique. Ses opérandes sont des nombres complexes, des variables ou des appels de fonction de la forme `FUNCT '(' Exp ')'`, c'est-à-dire un nom de fonction suivi de son argument entre parenthèses. Ces opérandes sont combinés par les opérateurs et regroupés par des parenthèses. Ainsi, `Exp` décrit aussi bien l'expression `2` que

```
({3 + _I 6} * _log(_P)) + (20 * VARIABLE_0)
```
(10)

— `Equal` représente une condition. La plus simple est `Exp EQ Exp` ; on peut en combiner plusieurs avec des ET et des OU logiques, par exemple :

```
(1 == 3 && 2 == VARIABLE_1) || VARIABLE_2 == 0
```
(11)

Les fonctions auxiliaires. Elles rendent les calculs complexes plus lisibles, mais ne sont jamais obligatoires. Une fonction auxiliaire se compose d'un nom, d'un paramètre et de l'expression qu'elle calcule, d'où la règle :

```
FUNCT '(' VARIA ')' '=' Exp ';'
```
(12)

Par exemple, une fonction qui renvoie le triple de son paramètre s'écrit `triple(PARAM) = PARAM * 3;`.

La fonction principale. C'est elle que l'application appelle. Son unique argument, `_Z`, est prédéfini, donc son en-tête se réduit à `FUNCT '='`. Son corps, entre accolades, est une suite d'instructions (token `SuiteInstr`), chacune de l'une des formes suivantes :

— `RETURN Exp ';'` : termine la fonction en renvoyant la valeur de l'expression ;
— `VARIA '=' Exp ';'` : déclare une variable, qui prend la valeur de l'expression ;
— `IF '(' Equal ')' '{' SuiteInstr '}'` : exécute les instructions si la condition est vraie ;
— `IF '(' Equal ')' '{' SuiteInstr '}' ELSE '{' SuiteInstr '}'` : exécute la première suite d'instructions si la condition est vraie, la seconde sinon.

4.3.2 Le programme obtenu

Le programme issu de Flex et de Bison s'appelle `ppcomp` (*Phase Portrait compiler*). Les fichiers écrits dans ce langage portent l'extension `.fpp` (*file phase portrait*). Il s'utilise de deux façons :

1. `./ppcomp [OPTION]` : sans fichier, il lit l'entrée standard, que l'on termine par Ctrl + D ;
2. `./ppcomp [OPTION] fichier.fpp` : il compile le fichier donné.

Ses options sont les suivantes :

`-t`, `-tree`	affiche l'arbre syntaxique ;
`-s`, `-symtab`	affiche les tables des symboles, présentées plus loin ;
`-v`, `-verbose`	affiche toutes les étapes de la compilation (et active `-t` et `-s`) ;
`-o` *nom*, `-out` *nom*	donne le nom du fichier produit ;
`-h`, `-help`	affiche l'aide.

Par exemple, la commande suivante compile `test14.fpp` en un fichier `exemple.c` :

```
./ppcomp -o exemple test14.fpp
```
(13)

4.4 La traduction de l'arbre syntaxique en C

Le programme C qui suit l'analyse parcourt l'arbre syntaxique et le traduit en code C. Il s'appuie sur des tables des symboles pour détecter les erreurs sémantiques. Une table des symboles répertorie tous les symboles (variables et fonctions) d'une portée ; une erreur sémantique est, par exemple, l'utilisation d'un symbole non déclaré. Le programme signale aussi des avertissements, comme un symbole déclaré mais jamais utilisé. Il est découpé en quatre modules :

1. **tree** : repris du cours de compilation et adapté au projet, il crée l'arbre syntaxique, l'affiche et libère la mémoire qu'il occupe ;
2. **table** : il alloue, remplit, affiche et libère les tables des symboles. Une entrée de table est représentée par la structure suivante :

```c
#define MAXDICO 150

/* Types possibles d'un symbole */
#define C_CONSTANTE 0
#define C_VARIABLE 1
#define C_FONCTION 2

/* Une ligne d'une table des symboles */
typedef struct {
    char identif[MAXDICO];
    int type;
    char complement[MAXDICO];
    int nbUse;
} desc_identif;
```

   Une entrée contient le symbole, son type (constante, variable ou fonction), un complément qui donne, pour une fonction, le nom de son paramètre, et le nombre d'utilisations du symbole en dehors de sa définition ;
3. **write** : il écrit le fichier C produit, en respectant l'indentation et en documentant les fonctions, pour que ce fichier reste lisible par un humain ;
4. **compile** : c'est le point d'entrée du programme. Il contient les fonctions récursives qui parcourent l'arbre, détectent les erreurs sémantiques et pilotent les trois autres modules.

4.5 Résultats obtenus

4.5.1 Exemple de programme

Voici le fichier source `test.fpp` :

```
/* add a first comment */
fun(A1) = -(_cos(A1) + _I _P);
var = {
    if(_Z != 0){
        return _Z;
    }else{
        FUN = fun(_Z);
        return FUN;
    }
}
```

4.5.2 Résultat en C

La commande `./ppcomp -o exemple test.fpp` produit le fichier suivant :

```c
/**
 * File : exemple.c
 * cree par le compilateur ppcomp
 */
#include "../../include/CRiemannSphere.h"
#include "../../include/CSpecialFunctions.h"

/**
 * name   : fun
 * param  : A1
 * param  : RS* res (resultat)
 * return : int 0 for success and 1 for error
 */
int C_fun ( RS A1, RS * res ) {
    RS rsVars0, rsVars1;
    RS rsStat0;
    if (RS_cos(A1, &(rsVars1))){
        RS_copy(res, RS_INFINITE());
        return 1;
    }
    if (RS_initialization(&(rsStat0), 0.000000, 3.141593, 0)){
        RS_copy(res, RS_INFINITE());
        return 1;
    }
    if (RS_add(rsVars1, rsStat0, &(rsVars0))){
        RS_copy(res, RS_INFINITE());
        return 1;
    }
    if (RS_opposite(rsVars0, &(*res))){
        RS_copy(res, RS_INFINITE());
        return 1;
    }
    return 0;
}

/**
 * name   : var
 * param  : _Z
 * param  : RS* res (resultat)
 * return : int 0 for success and 1 for error
 */
int C_var ( RS _Z, RS * res ) {
    RS rsVars0, rsVars1, rsVars2;
    RS rsStat0;
    if (RS_initialization(&(rsStat0), 0.000000, 0.000000, 0)){
        RS_copy(res, RS_INFINITE());
        return 1;
    }
    if(RS_neq(_Z, rsStat0)){
        if(RS_copy(res, rsVars0)){
            RS_copy(res, RS_INFINITE());
            return 1;
        }
        return 0;
    }else{
        RS FUN ;
        if (C_fun(_Z, &(rsVars1))){
            RS_copy(res, RS_INFINITE());
            return 1;
```

```
        }
        if(RS_copy(&(FUN), rsVars1)){
            RS_copy(res, RS_INFINITE());
            return 1;
        }
        if(RS_copy(res, rsVars2)){
            RS_copy(res, RS_INFINITE());
            return 1;
        }
        return 0;
    }
}
```

La fonction auxiliaire `C_fun` est correcte. En revanche, la fonction principale contient deux erreurs de traduction : `return _Z;` copie `rsVars0` au lieu de `_Z`, et `return FUN;` copie `rsVars2` au lieu de `FUN`. Dans les deux cas, la valeur renvoyée est une variable temporaire jamais initialisée. Ces limites sont détaillées ci-dessous.

4.6 Limites connues du compilateur

La refonte de l'application (section 5) a été l'occasion de tester `ppcomp` de façon systématique. Quatre défauts sont apparus :

- une expression renvoyée qui n'est qu'une variable ou qu'une expression entièrement parenthésée, comme `return (_Z * 2);`, renvoie une variable temporaire non initialisée ;
- des parenthèses doublées, comme `_Z * ((_Z + 1))`, produisent la même erreur ;
- les fonctions `_tan` et `_arg` sont reconnues par l'analyseur lexical, mais absentes de la table des fonctions prédéfinies : le compilateur s'arrête alors brutalement (erreur de segmentation) ;
- comme indiqué plus haut, $*$, / et $\hat{}$ ont la même priorité, ce qui ne correspond pas à l'usage mathématique.

Ces défauts produisaient des images fausses sans aucun message d'erreur, ce qui est le pire cas pour un outil d'étude. En attendant leur correction dans les sources de `ppcomp`, l'application les contourne : le code C produit est analysé avant d'être compilé, et toute lecture d'une variable temporaire non initialisée est refusée avec un message clair. Le mode « formule », présenté plus loin, génère quant à lui du code qui évite ces constructions.

5 Refonte de l'application

L'outil d'origine était une application de bureau écrite en Python. Pour le rendre accessible sans installation, en particulier aux étudiants, je l'ai transformé en application web : un serveur calcule les images, et une interface dans le navigateur permet d'écrire les fonctions, de régler la zone affichée et de consulter les résultats.

5.1 Refonte de l'arborescence

Le projet est désormais organisé autour de trois composants : le moteur de calcul (bibliothèque C et compilateur), une API web et une interface web.

```
Projet
 |- include, src_C     bibliothèque C de calcul et ses tests
 |- src_comp           compilateur ppcomp (Flex, Bison, C)
 |- src_api            API web (Python, FastAPI)
 |    |- phaseportrait   moteur de rendu : C, ctypes, traduction des
    formules
 |    |- database        schéma de la base de données (SQLAlchemy,
    MySQL)
 |    |- billing         offres et paiement (Stripe)
 |    |- admin           administration
 |    |- tests           tests automatisés
 |- src_client         interface web (React, TypeScript)
 |- src_py             ancienne application de bureau
 |- report             ce rapport
```

FIGURE 9 – Arborescence du projet après la refonte

L'API. Écrite en Python avec le framework FastAPI, elle gère les comptes, les projets, les scripts et les images, stockés dans une base MySQL. À chaque génération, le module `renderer` enchaîne les étapes suivantes :

1. traduire le script en C avec `ppcomp`, puis vérifier le code produit (section 4.6) ;
2. compiler ce code en bibliothèque partagée avec `gcc` et `make` ;
3. calculer la valeur de chaque pixel avec la fonction `compute` de la bibliothèque, à laquelle la fonction compilée est désormais passée directement sous forme de pointeur, sans repasser par Python à chaque pixel ;
4. dessiner l'image avec `draw`, l'enregistrer en PNG, puis supprimer les fichiers intermédiaires.

Chaque compilation porte un nom unique, ce qui permet de servir plusieurs utilisateurs en même temps ; le nombre de rendus simultanés est limité pour borner la mémoire utilisée.

Les corrections de la bibliothèque C. En la connectant à l'application web, j'ai corrigé deux défauts de la bibliothèque :

— l'initialisation du point à l'infini s'appelait elle-même sans fin, ce qui faisait planter le serveur dès qu'un pixel tombait sur un pôle ;

— les opérations élémentaires (multiplication, division, inverse, module) convertissaient leurs opérandes en texte à chaque appel, un reste de débogage. Les supprimer rend la division environ 64 fois plus rapide (8,3 s contre 0,13 s pour un million de divisions).

Les fonctions `compute` et `draw` acceptent aussi désormais un centre et des dimensions non entiers, ce qui permet de zoomer sur une partie du plan.

Taille de l'image	Temps de génération	Mémoire maximale	Taille du fichier
1000×1000 px	1,2 s	56 Mo	0,3 Mo
2000×2000 px	3,5 s	168 Mo	0,8 Mo
3000×3000 px	7,2 s	351 Mo	1,4 Mo

TABLE 3 – Coût d'une génération complète pour $z \mapsto \frac{z^2-1}{z^2+1}$, compilation comprise

L'écriture des fonctions. L'interface propose deux façons d'écrire une fonction :

— le **mode formule**, pensé pour les étudiants : on écrit la fonction comme sur une copie, par exemple `(z^2 - 1)/(z^2 + 1)`, `e^(1/z)`, `sqrt(z)` (ou $\sqrt{z}$ au clavier) ou `2iz^3`, avec la multiplication implicite et les priorités usuelles. La formule est affichée comme dans un manuel pendant la frappe, les erreurs sont localisées (ligne et colonne), et un clavier de symboles évite d'avoir à connaître la syntaxe. Un traducteur, écrit en Python, transforme la formule en programme `.fpp` correctement parenthésé, que `ppcomp` compile ensuite ;

— le **mode code**, qui donne accès à tout le langage de `ppcomp` (fonctions auxiliaires, variables, conditions), avec coloration syntaxique et numéros de ligne.

Les deux modes sont reliés : passer de la formule au code affiche sa traduction, et un code simple peut être reconverti en formule, en respectant les priorités de `ppcomp`.

Les offres et l'administration. Pour financer l'hébergement, l'application propose une offre gratuite et deux abonnements mensuels payés par Stripe, qui diffèrent par le nombre de générations par mois, le nombre d'images conservées et la taille maximale des images (tableau 4). Une interface d'administration présente les statistiques de la plateforme et permet de gérer les comptes, d'offrir une offre (à un enseignant, par exemple) et de modifier les paramètres. L'interface et l'API sont disponibles en français et en anglais.

	Gratuit	Plus	Pro
Prix mensuel TTC	0 €	15 €	40 €
Générations par mois	30	500	illimitées
Images conservées	15	300	3 000
Taille maximale	1000 px	2000 px	3000 px

TABLE 4 – Les offres de l'application

5.2 Refonte graphique

L'interface a été entièrement redessinée autour d'un système de design commun : couleurs, typographie, espacements, boutons, formulaires et tableaux sont définis une seule fois, ce qui garantit une apparence cohérente sur toutes les pages. Un thème clair et un thème sombre sont proposés, et l'interface s'adapte aux écrans de téléphone. Le logo et les illustrations reprennent le portrait de phase de $z \mapsto z$.

L'espace de travail d'un projet regroupe sur un seul écran les éléments de l'ancienne application (figure 3) :

— à gauche, la liste des scripts et le réglage de la zone affichée (centre, largeur, hauteur, résolution), avec un résumé de l'intervalle couvert et de la taille de l'image ;

— au centre, l'éditeur (formule ou code) et une console qui affiche les messages du compilateur ;

— à droite, l'aperçu de l'image, l'historique des générations du projet et une aide qui rappelle la syntaxe, explique la lecture des couleurs et propose des exemples à insérer.

Des raccourcis clavier permettent d'enregistrer (Ctrl + S) et de générer (Ctrl + Entrée) sans quitter l'éditeur.

6 Conclusion et perspectives

— Après avoir fait l'état des lieux de l'application, et en particulier de ses fonctions de calcul et d'affichage, avec un profileur, j'ai appris à utiliser finement la bibliothèque `ctypes`. Ce travail m'a permis de réduire fortement le temps d'exécution des fonctions `compute` et `draw` : environ 20 fois pour la première et 5 fois pour la seconde. Par exemple, le portrait de phase de $z \mapsto z^3$ sur une zone de $10\,000 \times 10\,000$ pixels a pu être calculé en moins de 4 minutes.

— Les portraits de phase de la fonction Gamma montrent cependant les limites du C : ses nombres flottants débordent pour les grandes valeurs, là où Python continue à calculer. Il serait intéressant de combiner les deux langages, en confiant à Python les valeurs extrêmes.

— J'ai traduit intégralement le module `special_functions.py` en C, afin de pouvoir transmettre n'importe quelle fonction au `compute` de la bibliothèque.

— J'ai ensuite défini une grammaire et écrit le compilateur `ppcomp`, pour que l'utilisateur n'ait plus à manipuler la bibliothèque en C : il écrit une expression, et le compilateur produit la fonction C correspondante.

— Enfin, la refonte en application web rend l'outil utilisable sans installation, et son mode formule le rend accessible aux étudiants qui ne programment pas. Elle a aussi permis de découvrir et de contourner plusieurs défauts de la bibliothèque et du compilateur.

Plusieurs pistes d'amélioration se dégagent :

— corriger les défauts de `ppcomp` dans ses sources (section 4.6), et donner à la puissance une priorité supérieure à celle de la multiplication ;

— ajouter au langage les sommes et produits indexés, de la forme $\sum_{k=K}^{N}$ et $\prod_{k=K}^{N}$, pour étudier des séries et des produits infinis tronqués ;

— remettre en place un cache des valeurs déjà calculées, par exemple dans la base de données, et ajouter un mode verbeux qui détaille l'avancement du calcul dans la bibliothèque ;

— paralléliser le calcul des pixels, qui sont indépendants les uns des autres, pour profiter de tous les cœurs du processeur.

A Annexes

A.1 Mesures sur les opérations élémentaires

Le tableau 5 compare le temps d'une addition et d'une multiplication d'entiers en Python et via `ctypes`. Pour une opération isolée, `ctypes` est environ trois fois plus lent, à cause du coût de l'appel. Au-delà de 10^{19}, les entiers dépassent la capacité des types du C : `ctypes` ne peut plus calculer, même avec le type `unsigned long long`.

Taille des entiers	Addition Python (10^{-7} s)	Addition `ctypes` (10^{-7} s)	Multiplication Python (10^{-7} s)	Multiplication `ctypes` (10^{-7} s)
10^5	2,08	6,26	1,92	6,42
10^6	2,08	6,38	1,86	6,11
10^7	2,11	5,66	1,96	7,06
10^8	1,97	5,48	1,80	7,03
10^9	2,07	6,14	2,02	6,48
10^{10}	2,19	6,41	2,50	6,73
10^{11}	2,18	6,45	2,24	6,95
10^{12}	2,23	6,75	2,27	7,78
10^{13}	2,21	6,32	2,15	6,68
10^{14}	2,12	6,25	2,58	6,92
10^{15}	2,01	6,52	2,38	6,80
10^{16}	2,18	6,74	2,58	6,87
10^{17}	2,10	6,34	2,35	6,76
10^{18}	2,13	6,27	2,20	7,17
10^{19}	2,23	6,74	2,28	7,21
10^{20}	2,47	–	2,59	–

TABLE 5 – Temps moyen d'une opération sur des entiers, en Python et via `ctypes`

A.2 Glossaire

API
(*Application Programming Interface*) ensemble de points d'accès par lesquels un programme en utilise un autre ; ici, le serveur que l'interface web interroge.

Arbre syntaxique
représentation d'un programme sous forme d'arbre, construite par l'analyseur syntaxique.

Bison
générateur d'analyseurs syntaxiques à partir d'une grammaire.

ctypes
module de la bibliothèque standard de Python qui permet d'appeler des fonctions d'une bibliothèque C.

Flex
générateur d'analyseurs lexicaux à partir d'expressions régulières.

HLS
représentation des couleurs par la teinte (*hue*), la clarté (*lightness*) et la saturation.

Lexème
motif lexical reconnu par l'analyseur lexical, comme un nombre ou un nom de fonction.

Pôle
point où une fonction tend vers l'infini ; autour d'un pôle, les couleurs du portrait de phase tournent dans le sens inverse de celui d'un zéro.

RiemannSphere
: nombre complexe enrichi d'un point à l'infini.

Table des symboles
: liste des variables et des fonctions d'une portée, utilisée pour détecter les erreurs sémantiques.

Token
: catégorie de lexèmes utilisée par la grammaire.

Bibliographie

[1] E. Wegert, *Visual Complex Functions : An Introduction with Phase Portraits*, Birkhäuser, 2012.

[2] Python Software Foundation, *ctypes — A foreign function library for Python*, `https://docs.python.org/3/library/ctypes.html`.

[3] Free Software Foundation, *GCC Command Options*, `https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html`.

[4] V. Paxson et al., *Flex, a fast scanner generator*, `https://github.com/westes/flex`.

[5] Free Software Foundation, *Bison — GNU Parser Generator*, `https://www.gnu.org/software/bison/manual/`.

[6] Wikipédia, *Teinte saturation luminosité*, fr.wikipedia.org/wiki/Teinte_saturation_luminosité.