Report – Student Project

Phase portraits of complex functions:
computing in C, a compiler and a web application

Étienne Collet

Supervisor: Olivier Bouillot

Université Gustave Eiffel

October 28, 2026

Acknowledgements

I would like to thank:

Olivier Bouillot,
for his guidance and for the time he gave me.

Contents

1 Introduction **4**

2 The tools involved **5**
2.1 The PhasePortrait tool . . . 5
2.1.1 Phase portrait . . . 5
2.1.2 The application . . . 5
2.2 The ctypes library . . . 7
2.2.1 Overview . . . 7
2.2.2 Usage . . . 7
2.2.3 Strengths and weaknesses . . . 8

3 Optimising the computation speed **9**
3.1 The changes made to the code . . . 9
3.1.1 Initial state . . . 9
3.1.2 Approach . . . 9
3.1.3 The `fct_to_phase_portrait_update` script . . . 9
3.1.4 The `tools.so` library . . . 9
3.1.5 The changes made to the Python code . . . 12
3.2 Results . . . 13
3.2.1 About the `draw` function . . . 14
3.2.2 About the `compute` function . . . 14

4 The ppcomp compiler **16**
4.1 Initial state and goals . . . 16
4.2 Flex: lexemes and tokens . . . 16
4.2.1 The basic lexemes . . . 16
4.2.2 The optional lexemes . . . 17
4.2.3 From lexemes to tokens . . . 17
4.3 Bison: the grammar . . . 18
4.3.1 Description of the grammar . . . 18
4.3.2 The resulting program . . . 19
4.4 Translating the syntax tree into C . . . 19
4.5 Results . . . 20
4.5.1 Example program . . . 20
4.5.2 Result in C . . . 20
4.6 Known limits of the compiler . . . 21

5 Rebuilding the application **22**
5.1 New project structure . . . 22
5.2 Visual redesign . . . 23

6 Conclusion and outlook **24**

A Appendices **25**
A.1 Measurements on elementary operations . . . 25
A.2 Glossary . . . 25

Bibliography **26**

List of Tables

1 Quadratic regressions of the time of `draw` (in seconds) . . . 14
2 Quadratic regressions of the time of `compute` (in seconds) . . . 14
3 Cost of a complete generation for $z \mapsto \frac{z^2-1}{z^2+1}$, compilation included . . . 23
4 The plans of the application . . . 23
5 Average time of an operation on integers, in Python and through `ctypes` . . . 25

List of Figures

1 HLS representation as a cylinder . . . 5
2 Phase portrait of $z \mapsto z$ over $[-5,5]+i[-5,5]$ . . . 6
3 Screenshot of the PhasePortrait tool . . . 6
4 Annotated phase portrait . . . 6
5 Phase portrait of the function f over $[-2,2]+i[-2,2]$ . . . 13
6 Phase portraits of the function Γ . . . 13
7 Execution time of `draw` by number of pixels per unit . . . 14
8 Execution time of `compute` by number of pixels per unit . . . 14
9 Structure of the project after the rebuild . . . 22

1 Introduction

In our first discussions about my student project, we talked about the PhasePortrait application, a tool for easily visualising and handling the graphical representations of functions $f : \mathbb{C} \to \mathbb{C}$, just as one would with the graph of a function $f : \mathbb{R} \to \mathbb{R}$.

Complex functions are important mathematical objects, used in algebra, in physics and in many other fields. Visualising them is hard, because their graph lives in a four-dimensional space. This is why it is represented by a colour image: the position of the pixel encodes two dimensions (the point z) and its colour encodes the other two (the value $f(z)$).

This application uses complex numbers extended with a point at infinity, which appears in some computations: $1/0 = \infty$ and $c \times \infty = \infty$ if $c \neq 0$. These numbers are called *RiemannSphere*, after the mathematician Bernhard Riemann. Such a number is described by its real part, its imaginary part and a third parameter telling whether it is infinite; in that case, the real and imaginary parts have no meaning.

Phase portraits are a rich source of information about the associated function: they reveal the presence or absence of zeros and poles over a part of the plane. They thus let mathematicians check their intuitions or state new conjectures.

However, computing the value of each pixel and displaying the image take a lot of time. For a unit square of n pixels per side, these operations have a complexity in $\mathcal{O}(f \cdot n^2)$, where f is the complexity of one evaluation of the function studied. This factor appears because computing $z \mapsto z$ does not take the same time as computing $z \mapsto \sin(z)$; the factor n^2 comes from the nested loops that go through each pixel of the image.

The question I chose is therefore how to reduce the execution time of the most costly functions, so as to get portraits with more pixels per unit, or over a larger part of the plane, more quickly. The project then went on with the writing of a compiler, so that users no longer have to program in C, and with the rebuilding of the tool as a web application.

I will first present the tools involved: the application, how it works and the `ctypes` library. I will then detail the optimisations made to the computation and the gains obtained, then the `ppcomp` compiler, and finally the rebuilding of the application. I will end with an assessment and the possible improvements.

2 The tools involved

2.1 The PhasePortrait tool

The PhasePortrait tool makes it easier to study the graphical representations of a complex function $f : \mathbb{C} \to \mathbb{C}$. It saves the results obtained and lets the user quickly change the parameters of the image: its resolution, its width and its height.

2.1.1 Phase portrait

The HLS representation. The HLS representation is an alternative to RGB for representing colours, in the form of a cylinder (see figure 1). It relies on three parameters: hue, lightness and saturation. Hue is expressed in degrees: it starts at 0° with red, goes through yellow, green, blue, then purple, and comes back to red at 360°. Lightness is the vertical axis: it lies in $[0, 1]$, the value 0 giving black and the value 1 white.

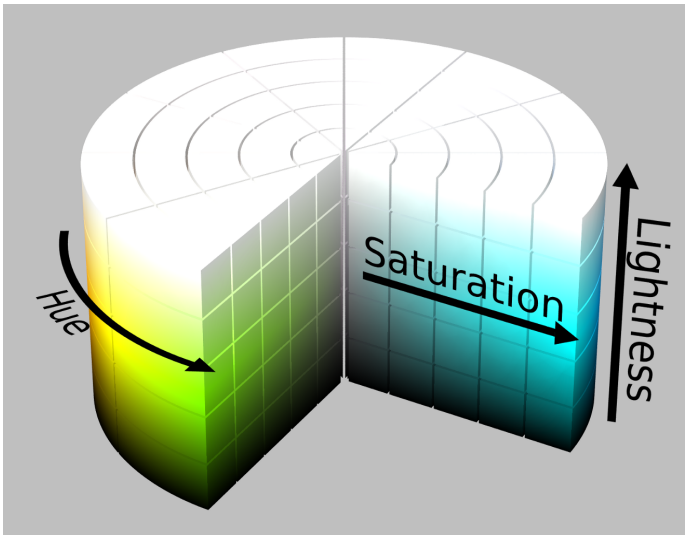


Figure 1: HLS representation as a cylinder (source: Wikipedia [6])

Computing the colour. To colour the pixel (a, b), we associate it with the complex number $z = a + ib$, then compute the hue H, the lightness L and the saturation S from $f(z)$:

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

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

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

The phase portrait of a function $f : \mathbb{C} \to \mathbb{C}$ is then the image whose pixel of coordinates (a, b) is coloured in HLS by formulas (1), (2) and (3), through the mapping $(a, b) \mapsto z \mapsto f(z)$. For example, figure 2 is the phase portrait of the identity function $z \mapsto z$ for $z \in [-5, 5] + i[-5, 5]$: the colour directly gives the argument of z.

2.1.2 The application

The tool is made of eight parts (see figure 3): parts 1 to 4 change the computation parameters, parts 6 and 7 are used to create and handle the image (generation, saving, viewing), part 5 displays the image and part 8 displays the computation logs.

A phase portrait holds a lot of information about the function studied; figure 4 annotates part of it.

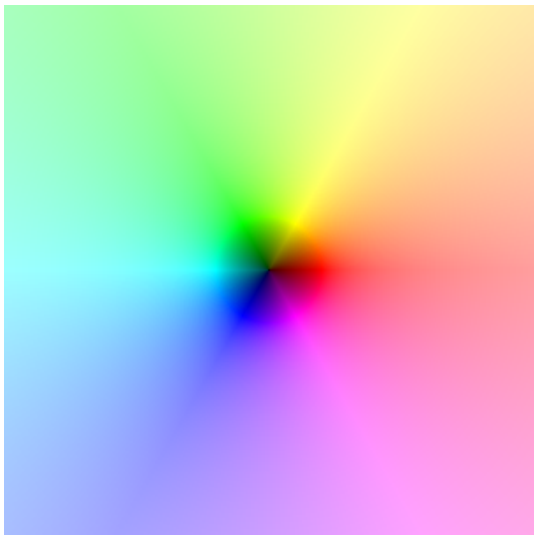


Figure 2: Phase portrait of $z \mapsto z$ over $[-5,5]+i[-5,5]$

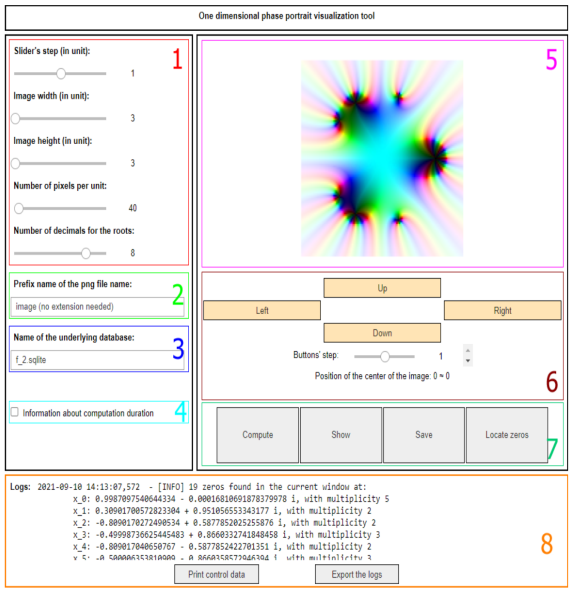


Figure 3: Screenshot of the PhasePortrait tool. Source: documentation of the tool

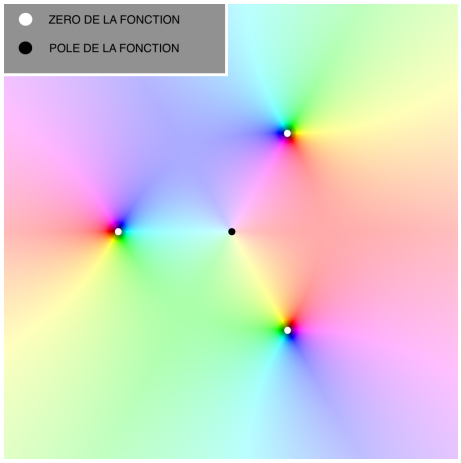


Figure 4: Phase portrait of $f : z \mapsto \frac{4}{z}\left(z^3+1\right)$ over $[-2,2]+i[-2,2]$, annotated

2.2 The ctypes library

2.2.1 Overview

`ctypes` is a module of the Python standard library that calls, from Python code, the functions of a C library. Recall that there are three ways of using a library:

- **static**: the library is built into the program at link time, during compilation. When it changes, every program using it must be recompiled;
- **shared**: the library is not copied into the programs, but linked to them when they start. It can serve many programs, and changing it does not require recompiling them;
- **dynamic**: a shared library that the program loads itself, when it needs it.

`ctypes` uses this last mode: libraries are loaded at run time by the `cdll` object.

2.2.2 Usage

On the C side. To use `ctypes`, we first create a shared library, for example `tools.so`, which contains the C functions to call from Python. Here is a possible source:

```
1  /* tools.c (somme: sum, libere: free) */
2  #include <stdlib.h>
3
4  int somme(int a, int b){
5      return a + b;
6  }
7
8  int * sommeAlloc(int a, int b){
9      int *res = (int *) malloc(sizeof(int));
10     if (NULL != res){
11         *res = a + b;
12     }
13     return res;
14 }
15
16 void libereSommeAlloc(int * res){
17     free(res);
18 }
```

This file is then compiled into a shared library with the command:

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

This command compiles a single file. For our library, a `makefile` compiles each module separately into an object file, then gathers them into a library with the same options.

The options given to gcc. Here is the role of the options of command (4):

`-ansi` applies the ANSI C standard;

`-Wall` turns on most warnings;

`-pedantic` reports any departure from the ISO C standard;

`-O2` turns on nearly all the optimisations that do not involve a trade-off between space and speed (for example, gcc does not unroll loops at this level). Compared with `-O`, this option makes compilation longer but the code produced faster;

`-shared` produces a shared object, which can be linked with other objects to form an executable;

`-fPIC` produces position-independent code, required for a dynamically loaded library.

On the Python side. Once the library is compiled, it is used in a Python script: we import `ctypes`, load the library with `cdll.LoadLibrary`, which takes the path of the file, then declare the types of the parameters and of the result of the functions used, before calling them.

Declaring these types matters: without them, `ctypes` assumes that the function returns an `int` and converts the parameters blindly, which can cause silent conversions or overflows on the C side, and therefore wrong results.

```
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("ctypes addition: " + str(x) + " + " + str(y) + " = " + str(a))
print("Python result: " + str(x + y))
```

On such short code, which performs a single operation in C, `ctypes` brings nothing: the cost of the call even increases the execution time (see appendix A.1).

For pointers, `ctypes` provides the type `POINTER(type)`. As Python does not know about the memory allocated by C, any allocation made in C must be freed in 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 Strengths and weaknesses

- Strengths:
  - `ctypes` greatly reduces the execution time of heavy computations;
  - it gives fine control over memory.
- Weaknesses:
  - the compiled code depends on the operating system and the architecture;
  - the module is tricky to use, because of the type conversions between Python and C and the handling of pointers;
  - it lengthens development time, since a library has to be written and recompiled (not to mention the errors caused by a library one forgot to recompile after a change!);
  - C types are bounded: an integer of the order of 10^{20} exceeds the capacity of an `int` as well as of an `unsigned long long`, whereas Python handles integers of arbitrary size.

3 Optimising the computation speed

3.1 The changes made to the code

3.1.1 Initial state

The original code, written only in Python, did not use `ctypes`. It was made of several modules and separated the computation of the values of the function from their display in the image. The handling of images relied on many nested loops, which is very costly in Python, in memory as well as in time. The points to improve were in the file `phase_portrait.py`, in particular the extremely costly functions `compute` and `draw`. Moreover, the size and density of the code kept me from understanding all its parts.

3.1.2 Approach

We therefore thought about how to remove these time losses, and the idea chosen was to use `ctypes`. I first experimented with the module on small programs: elementary operations and bare loops.

To make the next steps easier, I created the file `phase_portrait_2_0.py`, a simplified version of `phase_portrait.py` without the database that saved the computations. This let me identify the obstacles. For example, the computed values were stored in a Python dictionary, a structure incompatible with C. I replaced it with a two-dimensional array, allocated and freed in C.

I then rewrote in C the function `draw`, which draws the phase portrait from the values given to it. For the function `compute`, I started with a single hard-coded function to compute. To be able to change it easily, a function had to be passed as a parameter: at the time, I had not found how to pass a function pointer with `ctypes`.

So I wrote a script that compiles a function written in C and adds it to a library, which makes it possible to change functions at will. This required completing the module `CRiemannSphere.c` with all the operations on *RiemannSphere* numbers, and creating the module `CSpecialFunctions.c`, which gathers the usual functions (sine, exponential...).

This work led me to change `phase_portrait.py`, and to create the library `tools.so` (now `CphasePortrait.so`) and the script `fct_to_phase_portrait_update`.

3.1.3 The `fct_to_phase_portrait_update` script

This script turns a text file written in C into a shared library used by `tools.so`. The file must contain a single function with the prototype

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

whose code uses the functions of the modules `CRiemannSphere.c` and `CSpecialFunctions.c` to compute a *RiemannSphere*. The script copies this file into a complete C file, adding the required headers, then compiles it into a library `operation.so`.

For example, the following two functions compute the identity and the sine of a *RiemannSphere* respectively:

```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 The `tools.so` library

The sources of the library are in the directory `src_C`, the headers in `include`. Its modules are `Cimage`, `Ccolor`, `CRiemannSphere`, `CSpecialFunctions` and `CUnitaryTesting`. The files `TestsColor.c`, `TestsCRiemannSphere.c` and `TestsCSpecialFunctions.c` run a set of tests on the corresponding modules. The `makefile`, placed outside the source directory so that the Python script can compile the library itself, also compiles and runs these tests.

The `Ccolor` module. This module handles colours in the HLS and RGB formats. Above all, it converts a *RiemannSphere* into an HLS colour, then this colour into RGB, with the functions `int HSL(RS z, HSLColor * hsl)` and `int RGB(RS z, RGBColor * rgb)`, which return the colour by address. Both formats are represented by the following structures:

```
/** Colour in the HLS model */
typedef struct {
    double h;
    double l;
    double s;
} HSLColor;

/** Colour in the RGB model */
typedef struct {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
} RGBColor;
```

The `CRiemannSphere` module. This module defines the *RiemannSphere* structure, along with the functions to initialise it, handle it and compute with it:

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

Most of the module consists of computation functions: addition, subtraction, multiplication, division... Nearly all of them have a prototype of the form

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

where `Type` is an integer, a real or a *RiemannSphere*. One or two arguments come before a pointer to a *RiemannSphere*, which receives the result; the value returned is 0 on success and 1 on error. Only `RS_copy` has a different prototype, `int RS_copy(RS * dest, RS src)`, to stay close to the Unix copy functions.

The `Cimage` module. This module allocates a two-dimensional array of *RiemannSphere* with the function `RS ** initTab(int x, int y)`, where `x` and `y` are the dimensions of the matrix. This array receives the computed values, on the Python side or on the C side, then is used to draw the image. The function `void libereTab(RS *** tab, int x)` frees it: as stated in section 2.2.2, any allocation made in C must be freed in C.

The module also contains the function `RGBColor ** draw(RS ** values, int x, int y)`, which allocates an image and gives each pixel the colour matching its value. Once the image has been used, it must be freed with `void libereImage(RGBColor *** img, int x, int y)`. Finally, the function `compute` fills the array by evaluating, for each pixel, the function given to it as a parameter.

The `CSpecialFunctions` module. This module contains the more elaborate functions on *RiemannSphere* numbers: sine, cosine, square root, power, exponential, logarithm and the Gamma function. They all follow prototype (6).

The `CUnitaryTesting` module. This module contains generic functions to test the modules `Ccolor`, `CRiemannSphere` and `CSpecialFunctions`. It was written by Mr Bouillot, to factor out the tests I had written as much as possible.

The test modules. `TestsColor`, `TestsCRiemannSphere` and `TestsCSpecialFunctions` test each function of the corresponding module, using the generic functions of `CUnitaryTesting`. They take up the tests that existed in the Python sources, going further and fixing those that needed it.

3.1.5 The changes made to the Python code

On the Python side, I first created a structure identical to the one of the library, which links the two languages. When a `PhasePortrait` object is initialised, I added the loading of the library and the allocation of the array that stores the values of the function studied.

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

I also rewrote the function `draw`, which now only uses C functions. Here is its body:

```
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
# Free the memory allocated in C
self._lib.libereImage(byref(img_c), self.size[0], self.size[1])
```

3.2 Results

The goal was to reduce the time taken to compute and display phase portraits thanks to `ctypes`. The execution times of the function `draw` were measured with the function

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

whose phase portrait is given in figure 5. This function is not defined at $z = 0$, which is properly reported by an error message. One evaluation of f takes a constant number of multiplications and divisions: the function `draw` therefore has a complexity in $\mathcal{O}(n^2)$, n being the number of pixels per unit.

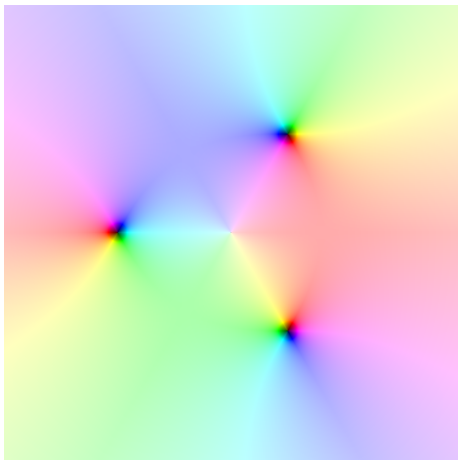
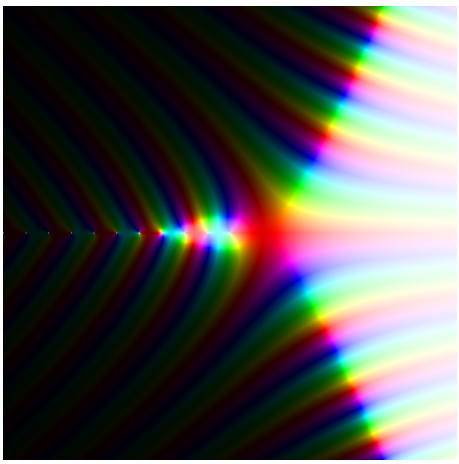


Figure 5: Phase portrait of the function f over $[-2, 2] + i[-2, 2]$

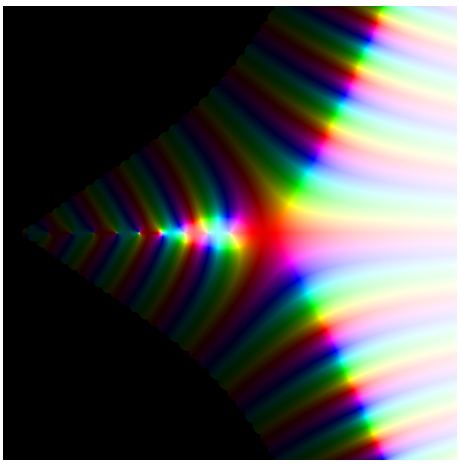
The times of the function `compute` were measured with the Gamma function,

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

whose phase portraits, computed in Python then in C, are given in figure 6.



(a) Computed in Python



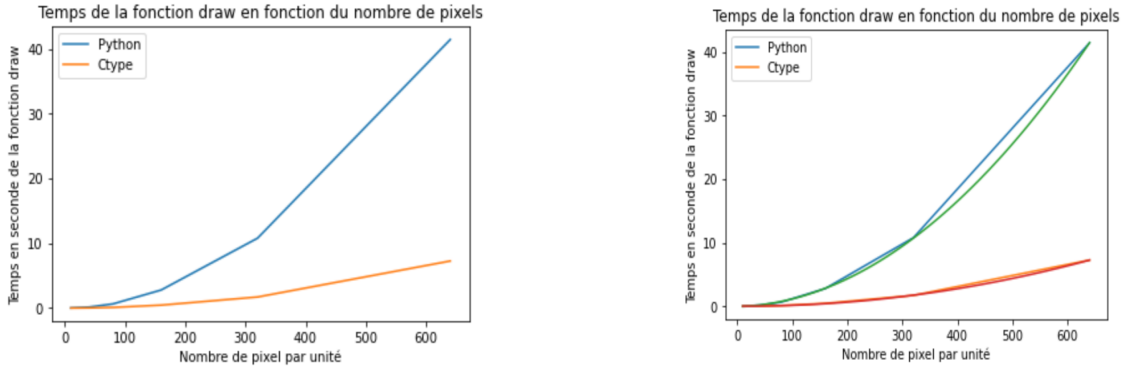
(b) Computed in C

Figure 6: Phase portraits of the function Γ

Note that the two images differ far from the origin: C floating-point numbers overflow for large values of Γ, where Python keeps computing. We will come back to this in the outlook.

3.2.1 About the `draw` function

The function `draw` has a complexity in $\mathcal{O}(n^2)$, as figure 7 confirms. The new version does not change this complexity, but greatly reduces its constant.



(a) Without the regression

(b) With the regression

Figure 7: Execution time of `draw` by number of pixels per unit

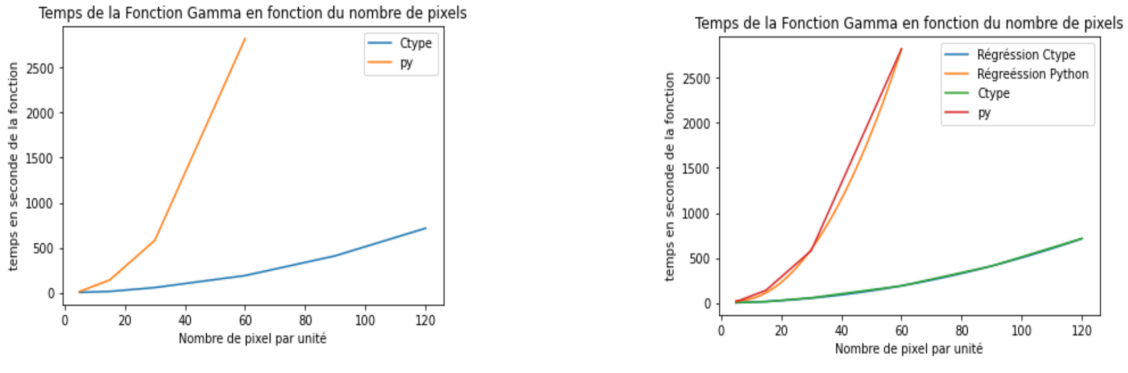
Python	$9.734 \cdot 10^{-5}x^2 + 2.564 \cdot 10^{-3}x - 7.364 \cdot 10^{-2}$
`ctypes`	$1.849 \cdot 10^{-5}x^2 - 5.285 \cdot 10^{-4}x + 2.126 \cdot 10^{-2}$

Table 1: Quadratic regressions of the time of `draw` (in seconds)

The R^2 coefficient of these two regressions is about 0.9999, which is excellent. The ratio of the leading coefficients shows that with `ctypes`, the function `draw` is about 5.3 times faster than in pure Python: a clear improvement.

3.2.2 About the `compute` function

The function `compute` written in C now accepts any function; the measurements were made with $z \mapsto \Gamma(z)$ (figure 8).



(a) Without the regression

(b) With the regression

Figure 8: Execution time of `compute` by number of pixels per unit

Python	$0.9325x^2 - 9.779x + 50.54$
`ctypes`	$0.04649x^2 + 0.3822x + 0.2184$

Table 2: Quadratic regressions of the time of `compute` (in seconds)

The R^2 coefficient of these regressions is about 0.9997, slightly less than for `draw` but still very good. With `ctypes`, the function `compute` is about 20 times faster than in pure Python: an excellent improvement.

4 The ppcomp compiler

4.1 Initial state and goals

After the optimisation, the image is computed in C, in a dynamic library. To study a new function, it must therefore be written in C, with the operations of this library, then passed to it. This makes the tool much harder to pick up: users must know how to program in C and know the library I wrote.

The idea is therefore to write a compiler that, from an arithmetic expression, produces the corresponding C function. Users simply type an expression such as `(_Z + _I 2) * 5` and see the image appear a few seconds later.

To write this compiler, I used Flex and Bison. Flex is free software that generates lexical analysers: programs that recognise patterns in a text. Bison generates parsers: programs that check that the text follows a given grammar. By combining them, and adding a C program that interprets the tree produced, we get the desired compiler.

4.2 Flex: lexemes and tokens

We must first define the lexemes the compiler has to recognise, then associate them with tokens, which serve as word categories for the grammar. By analogy, in English, nouns, verbs and adjectives are word categories.

4.2.1 The basic lexemes

A lexeme is a lexical pattern, described by a character string or a regular expression. As the language describes arithmetic expressions, it must first recognise the operators:

- addition ($+$) and subtraction ($-$);
- multiplication ($*$) and division ($/$);
- power ($\hat{}$).

Modulo is not offered, as it makes no sense for complex numbers.

Numbers must then be recognised. As we have seen (page 10), a complex number is represented by two floating-point numbers; the compiler therefore keeps the same precision:

- integers are described by `[0-9]+`;
- decimal numbers are described by `[0-9]+[.][0-9]+`.

Once read, every number is converted to a floating-point number: the number 5 becomes 5.0 in the code produced.

To write complex numbers and the usual constants, I introduced the following symbols:

- `_I` for i;
- `_INF` for infinity;
- `_E` for e;
- `_P` for π.

Parentheses set the order of operations, and braces delimit a complex number:

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

The language also lets users define and call functions, predefined or written by the user. A user function takes a single parameter, since the final program can only compute functions of one variable. The corresponding lexemes are:

- `[a-z]+[_a-zA-Z0-9]*` for a function name: a word that starts with a lowercase letter and goes on with letters, digits or underscores;
- `[A-Z]+[_A-Z0-9]*` for a variable name: a word that starts with an uppercase letter and goes on with uppercase letters, digits or underscores;
- the predefined functions: `_sqrt` (square root), `_log` (logarithm), `_exp` (exponential), `_sin` (sine), `_cos` (cosine), `_abs` (modulus), `_sabs` (squared modulus), `_conj` (conjugate), `_opp` (opposite) and `_inv` (inverse). The lexemes `_tan` and `_arg` are recognised, but not implemented yet (see section 4.6);
- `_Z`, the predefined variable of the main function;
- `return`, for the value to return.

4.2.2 The optional lexemes

To go further, I took a few elements from C. Comments first: `//` for a one-line comment, `/*` and `*/` for a comment over several lines. Then the conditional structures `if` and `else`, with the comparison operators that give them meaning:

- `&&` for logical AND;
- `||` for logical OR;
- `==` for equality;
- `!=` for difference.

Other character strings are ignored or cause an error.

4.2.3 From lexemes to tokens

The lexemes are associated with the following tokens:

- `NUM`: integers and decimal numbers;
- `II`: `_I`;
- `INF`: `_INF`;
- `EXPI`: `_E` and `_P`;
- `ADDSUB`: $+$ and $-$;
- `DIVSTAR`: $*$, / and $\hat{}$;
- `VARIA`: variable names, including `_Z`;
- `FUNCT`: function names, including the predefined functions;
- `RETURN`, `IF`, `ELSE`: the corresponding keywords;
- `AND`, `OR`: `&&` and `||`;
- `EQ`: `==` and `!=`.

As $*$, / and $\hat{}$ share the same token, they have the same priority and are evaluated from left to right: `2 * _Z ^ 2` means $(2z)^2$ and not $2z^2$. One must therefore write `2 * (_Z ^ 2)`.

4.3 Bison: the grammar

4.3.1 Description of the grammar

From the tokens, the grammar defines the valid sequences: a text that does not follow it causes a syntax error, like the sentence "eats apple Paul the" in English. The parser also builds the syntax tree of the program, which makes its translation into C easier.

A program has two parts: the optional definition of helper functions, then the main function, which the rest of the application evaluates on each pixel. Let us start with three grammar elements used everywhere.

The basic elements.

- `Complexe` represents a complex number, in one of the following six forms:
  - `'{' NUM ADDSUB II NUM '}'`, equivalent to $a \pm b\,i$;
  - `II NUM`, equivalent to $b\,i$;
  - `NUM`, equivalent to a;
  - `II`, equivalent to i;
  - `EXPI`, equivalent to e or π;
  - `INF`, infinity.
- `Exp` represents an arithmetic expression. Its operands are complex numbers, variables or function calls of the form `FUNCT '(' Exp ')'`, that is, a function name followed by its argument in parentheses. These operands are combined by the operators and grouped by parentheses. Thus `Exp` describes the expression `2` as well as

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

- `Equal` represents a condition. The simplest is `Exp EQ Exp`; several can be combined with logical ANDs and ORs, for example:

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

Helper functions. They make complex computations more readable, but are never required. A helper function is made of a name, a parameter and the expression it computes, hence the rule:

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

For example, a function that returns three times its parameter is written `triple(PARAM) = PARAM * 3;`.

The main function. This is the one the application calls. Its only argument, `_Z`, is predefined, so its header comes down to `FUNCT '='`. Its body, in braces, is a sequence of instructions (token `SuiteInstr`), each of one of the following forms:

- `RETURN Exp ';'`: ends the function by returning the value of the expression;
- `VARIA '=' Exp ';'`: declares a variable, which takes the value of the expression;
- `IF '(' Equal ')' '{' SuiteInstr '}'`: runs the instructions if the condition is true;
- `IF '(' Equal ')' '{' SuiteInstr '}' ELSE '{' SuiteInstr '}'`: runs the first sequence of instructions if the condition is true, the second otherwise.

4.3.2 The resulting program

The program built from Flex and Bison is called `ppcomp` (*Phase Portrait compiler*). Files written in this language have the extension `.fpp` (*file phase portrait*). It can be used in two ways:

1. `./ppcomp [OPTION]`: without a file, it reads the standard input, ended with Ctrl + D;
2. `./ppcomp [OPTION] file.fpp`: it compiles the given file.

Its options are the following:

`-t`, `-tree`	displays the syntax tree;
`-s`, `-symtab`	displays the symbol tables, presented below;
`-v`, `-verbose`	displays every step of the compilation (and turns on `-t` and `-s`);
`-o` *name*, `-out` *name*	gives the name of the file produced;
`-h`, `-help`	displays the help.

For example, the following command compiles `test14.fpp` into a file `exemple.c`:

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

4.4 Translating the syntax tree into C

The C program that follows the parsing goes through the syntax tree and translates it into C code. It relies on symbol tables to detect semantic errors. A symbol table lists all the symbols (variables and functions) of a scope; a semantic error is, for example, the use of an undeclared symbol. The program also reports warnings, such as a symbol declared but never used. It is split into four modules:

1. **tree**: taken from the compilation course and adapted to the project, it creates the syntax tree, displays it and frees the memory it uses;
2. **table**: it allocates, fills, displays and frees the symbol tables. A table entry is represented by the following structure:

```c
#define MAXDICO 150

/* Possible types of a symbol */
#define C_CONSTANTE 0
#define C_VARIABLE 1
#define C_FONCTION 2

/* One row of a symbol table */
typedef struct {
    char identif[MAXDICO];
    int type;
    char complement[MAXDICO];
    int nbUse;
} desc_identif;
```

   An entry holds the symbol, its type (constant, variable or function), a complement that gives, for a function, the name of its parameter, and the number of uses of the symbol outside its definition;

3. **write**: it writes the C file produced, keeping the indentation and documenting the functions, so that this file stays readable by a human;
4. **compile**: the entry point of the program. It contains the recursive functions that go through the tree, detect semantic errors and drive the other three modules.

4.5 Results

4.5.1 Example program

Here is the source file `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 Result in C

The command `./ppcomp -o exemple test.fpp` produces the following file:

```
/**
 * 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;
    }
}
```

The helper function `C_fun` is correct. However, the main function contains two translation errors: `return _Z;` copies `rsVars0` instead of `_Z`, and `return FUN;` copies `rsVars2` instead of `FUN`. In both cases, the value returned is a temporary variable that was never initialised. These limits are detailed below.

4.6 Known limits of the compiler

The rebuilding of the application (section 5) was an opportunity to test `ppcomp` systematically. Four flaws came to light:

- a returned expression that is only a variable or an expression fully in parentheses, such as `return (_Z * 2);`, returns an uninitialised temporary variable;
- doubled parentheses, such as `_Z * ((_Z + 1))`, produce the same error;
- the functions `_tan` and `_arg` are recognised by the lexical analyser, but missing from the table of predefined functions: the compiler then crashes (segmentation fault);
- as stated above, $*$, / and $\hat{}$ have the same priority, which does not match mathematical usage.

These flaws produced wrong images without any error message, which is the worst case for a study tool. Until they are fixed in the sources of `ppcomp`, the application works around them: the C code produced is analysed before being compiled, and any read of an uninitialised temporary variable is refused with a clear message. The "formula" mode, presented below, generates code that avoids these constructions.

5 Rebuilding the application

The original tool was a desktop application written in Python. To make it available without installation, especially to students, I turned it into a web application: a server computes the images, and an interface in the browser is used to write the functions, set the displayed area and view the results.

5.1 New project structure

The project is now organised around three components: the computation engine (C library and compiler), a web API and a web interface.

```
Project
 |- include, src_C     C computation library and its tests
 |- src_comp           ppcomp compiler (Flex, Bison, C)
 |- src_api            web API (Python, FastAPI)
 |    |- phaseportrait   rendering engine: C, ctypes, formula
    translation
 |    |- database        database schema (SQLAlchemy, MySQL)
 |    |- billing         plans and payment (Stripe)
 |    |- admin           administration
 |    |- tests           automated tests
 |- src_client         web interface (React, TypeScript)
 |- src_py             former desktop application
 |- report             this report
```

Figure 9: Structure of the project after the rebuild

The API. Written in Python with the FastAPI framework, it manages the accounts, projects, scripts and images, stored in a MySQL database. For each generation, the `renderer` module goes through the following steps:

1. translate the script into C with `ppcomp`, then check the code produced (section 4.6);
2. compile this code into a shared library with `gcc` and `make`;
3. compute the value of each pixel with the function `compute` of the library, to which the compiled function is now passed directly as a pointer, without going back through Python for each pixel;
4. draw the image with `draw`, save it as PNG, then delete the intermediate files.

Each compilation has a unique name, so several users can be served at the same time; the number of simultaneous renders is limited to bound the memory used.

Fixes to the C library. While connecting it to the web application, I fixed two flaws of the library:

- the initialisation of the point at infinity called itself endlessly, which crashed the server as soon as a pixel fell on a pole;
- the elementary operations (multiplication, division, inverse, modulus) converted their operands to text at each call, a leftover from debugging. Removing this makes division about 64 times faster (8.3 s against 0.13 s for a million divisions).

The functions `compute` and `draw` now also accept a non-integer center and dimensions, which makes it possible to zoom into part of the plane.

Image size	Generation time	Peak memory	File size
1000×1000 px	1.2 s	56 MB	0.3 MB
2000×2000 px	3.5 s	168 MB	0.8 MB
3000×3000 px	7.2 s	351 MB	1.4 MB

Table 3: Cost of a complete generation for $z \mapsto \frac{z^2-1}{z^2+1}$, compilation included

Writing functions. The interface offers two ways of writing a function:

- the **formula mode**, designed for students: the function is written as on paper, for example `(z^2 - 1)/(z^2 + 1)`, `e^(1/z)`, `sqrt(z)` (or $\sqrt{z}$ from the keyboard) or `2iz^3`, with implicit multiplication and the usual priorities. The formula is displayed as in a textbook while typing, errors are located (line and column), and a symbol keyboard removes the need to know the syntax. A translator, written in Python, turns the formula into a correctly parenthesised `.fpp` program, which `ppcomp` then compiles;
- the **code mode**, which gives access to the whole `ppcomp` language (helper functions, variables, conditions), with syntax highlighting and line numbers.

The two modes are linked: switching from formula to code shows its translation, and simple code can be converted back into a formula, respecting the priorities of `ppcomp`.

Plans and administration. To pay for the hosting, the application offers a free plan and two monthly subscriptions paid through Stripe, which differ in the number of generations per month, the number of images kept and the maximum size of the images (table 4). An administration interface shows the statistics of the platform and is used to manage the accounts, grant a plan (to a teacher, for example) and change the settings. The interface and the API are available in French and in English.

	Free	Plus	Pro
Monthly price	€0	€15	€40
Generations per month	30	500	unlimited
Images kept	15	300	3,000
Maximum size	1000 px	2000 px	3000 px

Table 4: The plans of the application

5.2 Visual redesign

The interface was entirely redesigned around a common design system: colours, typography, spacing, buttons, forms and tables are defined only once, which ensures a consistent look on every page. A light theme and a dark theme are offered, and the interface adapts to phone screens. The logo and illustrations use the phase portrait of $z \mapsto z$.

The workspace of a project brings together on a single screen the elements of the former application (figure 3):

- on the left, the list of scripts and the settings of the displayed area (center, width, height, resolution), with a summary of the interval covered and of the image size;
- in the middle, the editor (formula or code) and a console that displays the compiler messages;
- on the right, the preview of the image, the history of the generations of the project and a help panel that recalls the syntax, explains how to read the colours and offers examples to insert.

Keyboard shortcuts save (Ctrl + S) and generate (Ctrl + Enter) without leaving the editor.

6 Conclusion and outlook

- After assessing the application, and in particular its computation and display functions, with a profiler, I learnt to make fine use of the `ctypes` library. This work greatly reduced the execution time of the functions `compute` and `draw`: about 20 times for the first and 5 times for the second. For example, the phase portrait of $z \mapsto z^3$ over an area of 10,000 × 10,000 pixels could be computed in less than 4 minutes.
- The phase portraits of the Gamma function, however, show the limits of C: its floating-point numbers overflow for large values, where Python keeps computing. It would be interesting to combine both languages, leaving the extreme values to Python.
- I fully translated the module `special_functions.py` into C, so that any function can be passed to the `compute` of the library.
- I then defined a grammar and wrote the `ppcomp` compiler, so that users no longer have to handle the library in C: they write an expression, and the compiler produces the corresponding C function.
- Finally, rebuilding the tool as a web application makes it usable without installation, and its formula mode makes it accessible to students who do not program. It also brought to light, and worked around, several flaws of the library and the compiler.

Several possible improvements stand out:

- fix the flaws of `ppcomp` in its sources (section 4.6), and give power a higher priority than multiplication;
- add indexed sums and products to the language, of the form $\sum_{k=K}^{N}$ and $\prod_{k=K}^{N}$, to study truncated series and infinite products;
- bring back a cache of the values already computed, for example in the database, and add a verbose mode that details the progress of the computation in the library;
- parallelise the computation of the pixels, which are independent of each other, to take advantage of all the cores of the processor.

A Appendices

A.1 Measurements on elementary operations

Table 5 compares the time of an addition and of a multiplication of integers in Python and through `ctypes`. For a single operation, `ctypes` is about three times slower, because of the cost of the call. Beyond 10^{19}, integers exceed the capacity of C types: `ctypes` can no longer compute, even with the type `unsigned long long`.

Size of the integers	Python addition (10^{-7} s)	`ctypes` addition (10^{-7} s)	Python multiplication (10^{-7} s)	`ctypes` multiplication (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: Average time of an operation on integers, in Python and through `ctypes`

A.2 Glossary

API
(*Application Programming Interface*) set of access points through which one program uses another; here, the server that the web interface queries.

Bison
generator of parsers from a grammar.

ctypes
module of the Python standard library that calls functions of a C library.

Flex
generator of lexical analysers from regular expressions.

HLS
representation of colours by hue, lightness and saturation.

Lexeme
lexical pattern recognised by the lexical analyser, such as a number or a function name.

Pole
point where a function tends to infinity; around a pole, the colours of the phase portrait turn in the opposite direction to that of a zero.

RiemannSphere
: complex number extended with a point at infinity.

Symbol table
: list of the variables and functions of a scope, used to detect semantic errors.

Syntax tree
: representation of a program as a tree, built by the parser.

Token
: category of lexemes used by the grammar.

Bibliography

[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] Wikipedia, *HSL and HSV*, `https://en.wikipedia.org/wiki/HSL_and_HSV`.