TD 2

Classes

2.1 Un peu d'organisation...

À partir de ce TD il est demandé de respecter les noms de classes et de variables donnés par l'énoncé, ainsi que la signature des fonctions. Aussi, pour chaque classe, on créera deux fichiers, *nomdelaclasse.h*, qui contiendra toutes les déclarations, et *nomdelaclasse.cpp*, qui contiendra toutes les définitions (sauf mention contraire de l'énoncé). On n'oubliera pas de créer un troisième fichier *test_nomdelaclasse.cpp* qui fournira un programme principal de test de la classe. On rappelle qu'un fichier **.h* doit nécessairement être "gardé" : il commencera par les lignes

```
#ifndef NOMDEFICHIER_H
#define NOMDEFICHIER_H
```

et finira par la ligne

```
#endif // NOMDEFICHIER_H
```

Ces lignes, appelées *header guard* assurent que le contenu du fichier sera déclaré une et une seule fois, quelque soit la manière dont il est inclus. Aussi, pour inclure les fichiers **.h* créés dans d'autres fichiers **.h* ou **.cpp*, on utilisera la syntaxe `#include "nomdufichier.h"`, avec des guillemets `""`, et non des chevrons `<>` (qui sont pour les headers externes au projet, par exemple dans */usr/include*).

Lorsqu'on travaille avec un grand nombre de classes (et donc un grand nombre de fichiers), compiler à la main devient pénible et long. Pour rendre la compilation plus aisée il existe un outil très utilisé : `make`. Copiez le fichier *Makefile_template* présent sur e-campus dans le dossier contenant votre code et renommez-le en *Makefile*. Alternativement, vous pouvez utiliser la version plus simple mais moins automatique, *Makefile_simple*.

Enfin, il existe des outils plus modernes et automatiques comme `cmake`. Pour les plus courageux, un tutoriel simple se trouve ici : `https://alexandre-laurent.developpez.com/tutoriels/cmake/`.

Pour *Makefile_template*, les premières lignes sont à modifier en fonction des fichiers à compiler :

```
#PARTIE A MODIFIER : Liste des fichiers .cpp (et uniquement les .cpp) à compiler
SOURCES=
#FIN DE LA PARTIE A MODIFIER
```

Une fois le *makefile* modifié, le programme peut être compilé en lançant la commande

```
$  make
```

dans le répertoire où est situé le *makefile*. Une fois la compilation terminée avec succès, le programme peut-être exécuté via

```
$  ./run
```

Plutôt que d'utiliser les makefile fournis, il est conseillé de l'écrire vous même en vous inspirant de :

```
# arguments pour l'étape de compilation : tous les avertissements
CPPFLAGS := -Wall -Wextra
# arguments pour l'édition de liens, bibliothèques externes
LDFLAGS := -lm
# nom du compilateur
CPP := g++

# cible par défaut
all: monprogramme

# cible monprogramme : compilation des .cpp (dépendances .o), édition des liens, puis
↪  exécution
monprogramme: main.o uneclasse.o
  $(CPP) -o monprogramme $^ $(LDFLAGS)
  ./monprogramme

# compilation de main.cpp
main.o: main.cpp uneclasse.h
  $(CPP) -o $@ -c $< $(CPPFLAGS)

# compilation de uneclasse.cpp
uneclasse.o: uneclasse.cpp uneclasse.h
  $(CPP) -o $@ -c $< $(CPPFLAGS)
```

Attention si vous copiez-collez ce contenu dans votre *Makefile* : les indentations doivent être des tabulations, et non des espaces !

2.2 Déclaration et utilisation de classe en C++

Pré-requis : cours 4 sur les classes.

2.2.1 Classe `point`

Créer une classe `point` composée de deux coordonnées réelles puis déclarer et définir une méthode pour initialiser ces valeurs ainsi qu'une méthode affichant les deux coordonnées. Testez votre classe dans un `main()` en créant un point et en l'affichant.

En attendant le cours sur l'encapsulation, déclarer tous les membres avec une visibilité publique :

```
class point {
public:
  ...
};
```

Libre à vous de choisir une convention de nommage. Il est courant de nommer les classes par une majuscule et `EnCamelCase`, mais la bibliothèque standard utilise plutôt `le_snake_case`. De plus, les attributs d'une classe peuvent être préfixés par `m_` (pour *membre*) afin bien les différentier des variables locales.

Correction

Classe `point`

Déclaration (`point.h`)

Comme indiqué dans le sujet, la déclaration de la classe `point` *i.e.* les attributs de la classe ainsi que les prototypes des méthodes, se fait dans un fichier d'en-tête dédié appellé `point.h`

```
#ifndef _POINT_H_
#define _POINT_H_ 1

class point {
public:
  // Attributs
  double x;
  double y;

  // Méthode d'initialisation
  void initialise(const double x_, const double y_);

  // Méthode d'affichage
  void affiche() const;
};

#endif
```

On retrouve en préambule de la déclaration, les directives de préprocesseur `#ifndef/#define` qui interdisent l'inclusion multiple de ce fichier. Le cours sur l'encapsulation n'ayant pas été abordé, les méthodes `initialise` et `affiche` de même que les données membres ou attributs ont tous une visibilité publique. Sans présager de comment les méthodes d'initialisation et d'affichage seront implémentées, toutes deux ne retournent pas de données (type `void`), la fonction `initialise` prenant deux arguments `x_` et `y_`.
Un point à noter est la présence du `const` à la fin de la déclaration de la méthode `affiche` qui indique que cette fonction-membre ne modifiera pas les données membres. Ça n'a rien d'obligatoire, et on verra plus en détails cette notion dans la suite du cours. En revanche, la fonction d'initialisation ne présente pas un tel attribut du fait que, par construction, cette méthode assignera des valeurs aux membres `x_` et `y_`.

Déclaration (`point.cpp`)

Le fichier `point.cpp` qui contient la définition des méthodes se présente sous la forme suivante

```
#include "point.h"

#include <iostream>
using namespace std;

void point::initialise(const double x_, const double y_) {
  x = x_;
  y = y_;
}

void point::affiche() const {
  cout << "(x,y) = (" << x << "," << y << ")" << endl;
}
```

Ce fichier devant être indépendamment compilé, la déclaration de la classe doit être connue. C'est l'objet de l'inclusion du fichier `point.h` (ligne 1). Les définitions des méthodes reprennent les prototypes déclarés dans le fichier `point.h` en préfixant néanmoins leurs intitulés du nom de la classe suivi de l'opérateur de résolution de portée `::`. Ce préfixe assure au compilateur que les fonctions `initialise` et `affiche` sont des fonctions membres de la classe `point` et non des fonctions externes à cette classe : la présence de ce préfixe est donc primordiale dans l'association entre la déclaration des méthodes et leurs définitions.
Comme expliqué lors du cours sur les classes en C++ , le principe sur lequel repose la programmation orientée objet tient en l'association de données membres et de fonctions membres. Ainsi, les fonctions membres ou méthodes de la classe `point` que sont `initialise` et `affiche` ont, de par leur statut de méthodes, la possibilité de manipuler les membres de la classe `point` que sont `x` et `y`. La méthode `initialise` assigne ainsi des valeurs aux membres `x` et `y` par le biais des arguments `x_` et `y_` tandis que la méthode `affiche` se contente de présenter ces valeurs sans les modifier (d'où la présence du mot-clé `const`).
Dernier point, l'inclusion de la librairie `iostream` de gestion des flux d'entrée et sortie (`cout/cin`) est nécessaire puisque la méthode `affiche` utilise la sortie standard `cout`.

Programme test (`main.cpp`)

Afin de tester la classe `point` indépendamment de toute développement ou utilisation futur, il convient de créer pour chaque classe ce que l'on appelle un test unitaire. Ces programmes-tests permettent donc de tester les fonctionnalités d'une classe et de s'assurer de son bon fonctionnement en dehors de toute utilisation dans un cadre plus large faisant intervenir d'autres classes, d'autres programmes... C'est donc une étape indispensable et malheureusement, trop souvent négligée. Un programme test de la classe `point` peut prendre la forme suivante

```
#include "point.h"

int main() {
  // Création d'une instance de la classe point
  point my_point;
  my_point.initialise(6.5, 8.5);
  my_point.affiche();
  return 0;
}
```

Ce programme teste successivement la création d'un objet de type `point`, son initialisation puis l'affichage des valeurs d'abscisse et d'ordonnée. La présence de la directive `#include "point.h"` en préambule du programme permet de s'assurer d'une part, que la classe `point` existe en soi et d'autre part, que l'utilisation des méthodes associées à cette classe est correctement faite. La liaison entre l'utilisation des méthodes et leurs définitions est réalisée par le compilateur lors de l'édition de lien, troisième étape de la compilation (*cf.* cours sur la compilation).

Tant que les attributs sont publiques, il est aussi possible de créer le point en l'initialisant directement comme dans une structure en C :

```
#include "point.h"

int main() {
  // Création d'une instance de la classe point
  point my_point { .x=6.5, .y=8.5 };
  my_point.affiche();
  return 0;
}
```

Mais nous allons rapidement remplacer l'initialisation par un constructeur dans le TD suivant.

Makefile

Avec `Makefile_template`

On n'oubliera pas de modifier le *makefile*

```
#PARTIE A MODIFIER : Liste des fichiers .cpp (et uniquement les .cpp) à compiler
SOURCES=main.cpp point.cpp
#FIN DE LA PARTIE A MODIFIER
```

On peut ensuite compiler et exécuter notre programme test avec

```
$  make && ./run
```

Avec `Makefile_simple`

Un makefile minimaliste serait :

```
CXXFLAGS = -Wall
HEADERS = point.h
OBJETS = main.o point.o

test_point: $(OBJETS)
  g++ -o $@ $^ $(CXXFLAGS)
  ./test_point
```

Correction

```
%.o: %.cpp $(HEADERS)
  g++ -c -o $@ $< $(CXXFLAGS)
```

Que fait make lorsqu'on l'invoque ? Dans l'ordre :

1. Il veut construire la cible `test_point`. Pour ça, il voit qu'il faut les dépendances `$(OBJETS)`, c'est à dire les fichiers objets `main.o` et `point.o`.
2. Il regarde donc si `main.o` existe et si il est à jour. Sinon, il construit la cible `main.o`. Il voit que c'est possible avec la cible "joker" `%.o`. Il voit qu'il a besoin du fichier source correspondant `%.cpp` (c'est à dire `main.cpp`), ainsi que les headers. Il exéctera les commandes de cette cibles que si ces dépendances ont étés modifiées depuis la dernière invocation.
3. Exécution de la commande `g++ -c -o $@ $< $(CXXFLAGS)` : il s'agit de la compilation du fichier source `$<` et de la génération du fichier objet `$@`. La variable `$@` correspond au nom de la cible (`main.o` ici). La variable `$<` correspond au nom de la première dépendance (`main.cpp` ici).
4. Même chose pour `point.o`.
5. Maintenant que toutes les dépendances de `test_point` sont satisfaites, on peut exécuter les commandes de cette cible. La première est `g++ -o $@ $^ $(CXXFLAGS)` : il s'agit de l'édition de lien, générant un exécutable `$@` à partir des fichiers objets `$^`. La variable `$@` correspond au nom de la cible (`test_point` ici). La variable `$^` correspond au nom de toutes les dépendances (`main.o point.o` ici).
6. La deuxième commande est `./test_point` : il s'agit de l'exécution de l'exécutable.

Encore plus élémentaire...

Un makefile avec des noms explicites serait :

```
CXXFLAGS = -Wall

test_point: main.o point.o
  g++ main.o point.o -o test_point $(CXXFLAGS)
  ./test_point

main.o: main.cpp point.h
  g++ -c main.cpp -o main.o $(CXXFLAGS)

point.o: point.cpp point.h
  g++ -c point.cpp -o point.o $(CXXFLAGS)
```

2.2.2 Tableau d'objets

Déclarer un tableau (`std::vector`) contenant cinq objets de type `point`. Initialiser chacune des entrées du tableau puis créer une fonction affichant les coordonnées du point le plus éloigné de l'origine.
Pour cela, il sera utile de créer une nouvelle méthode `point::dist_origine()` qui renvoie la distance à l'origine.

Correction

Tableau d'objets

La notion de classe n'étant qu'une extension de la notion de type, on peut stocker des objets dans des tableaux. Le programme ci-dessous propose ainsi de stocker 5 instances de type `point` dans un tableau dynamiquement alloué.

```cpp
#include "point.h"

int main()
{
  const unsigned int npoint = 5;
  point* my_points = new point[npoint];
  // Initialisation arbitraire des 5 objets
  for (unsigned int i = 0; i < npoint; i++)
    my_points[i].initialise(i, 2*i);
  for (unsigned int i = 0; i < npoint; i++)
    my_points[i].affiche();
  // Désallocation
  delete[] points;

  return 0;
}
```

La même chose peut être faite plus simplement avec `std::vector` :

```cpp
#include "point.h"

int main()
{
  std::vector<point> my_points(5);
  // Initialisation arbitraire des 5 objets (for i loop)
  for (unsigned int i = 0; i < my_points.size(); i++)
    my_points[i].initialise(i, 2*i);
  // Affichage (range-based loop)
  for (point p : my_points)
    p.affiche();

  return 0;
}
```

La recherche du point le plus éloigné par rapport à l'origine ne peut se faire qu'au travers d'une fonction externe. En effet, c'est un non sens que de chercher pour un objet de type `point` s'il est plus éloigné que lui-même de l'origine. Ce genre de raisonnement permet ainsi de déduire que cette fonction ne peut être une méthode de la classe `point`, et est donc nécessairement une fonction externe. On peut en revanche, envisager d'ajouter à la classe `point` une méthode permettant de retourner la distance à l'origine. Ainsi, la classe `point` verra sa déclaration modifiée en conséquence

```cpp
class point {
public:
  ...
  // Calcul de la distance à l'origine
  double dist_origine() const;
};
```

dont la définition sera la suivante

```cpp
#include <cmath>
...
double point::dist_origine() const {
  return sqrt(x*x + y*y);
}
```

Quant à la fonction externe que l'on placera dans le programme principal, on pourra l'écrire de la façon suivante

```cpp
unsigned int get_farthest(std::vector<point> points)
{
  double max_dist = 0.0;
  unsigned int max_idx = -1;
  for (unsigned int i = 0; i < points.size(); i++) {
    double dist = points[i].dist_origine();
    if (dist > max_dist) {
      max_idx = i;
      max_dist = dist;
    }
  }
  return idx;
}
```

et l'utiliser ainsi

Correction

```cpp
#include "point.h"
#include <iostream>
using namespace std;

// Fonction get_farthest

int main()
{
  std::vector<point> my_points(5);
  // Initialisation arbitraire des 5 objets (for i loop)
  for (unsigned int i = 0; i < my_points.size(); i++)
    my_points[i].initialise(i, 2*i);

  unsigned int i = get_farthest(my_points);
  cout << "Le point le plus éloigné de l'origine est le point #" << i << " de coordonnées ";
  my_points[i].affiche();
}
```

2.2.3 Classe `Polygone`

Initialisation

Créer une classe `Polygone` possédant un unique attribut : un tableau de `point` contenant les coordonnées des sommets.

Définir une méthode `initialise_régulier(unsigned int ordre, double taille)` permettant l'initialisation de la classe `Polygone` comme un polygone d'ordre `ordre` régulier, c'est-à-dire équilatéral et équiangle, et dont la distance des sommets à l'origine est `taille`.

Définir une méthode `affiche_sommets` affichant la liste des coordonnées des sommets.

Faire un programme test créant un polygone régulier, de taille aléatoire [1] si vous voulez, et affichant ses sommets.

Périmètre

Définissez ensuite une méthode qui calculera le périmètre d'un polygone. On supposera que les segments entre chaque sommet consécutif ne se croisent pas. On pourra s'aider d'une fonction auxiliaire pour calculer la distance entre deux points.

Tester votre méthode sur un carré de "taille" $\sqrt{2}$.

Enfin, écrivez un programme calculant les périmètres de polygones réguliers de "taille" $1/2$ pour des ordre de $n = 1$ à $n \gg 1$, et enregistrant séquentiellement les résultats dans un fichier texte. Utilisez ensuite numpy et matplotlib en Python pour lire le fichier généré (avec `numpy.loadtxt` par exemple) et afficher le périmètre p_n en fonction de n. Vers quelle valeur p_∞ le périmètre converge-t-il ? Quelle est la vitesse de convergence (afficher $|p_\infty - p_n|$ sur un plot log-log) ?

Correction

Classe `Polygone`

De nouveau, la notion de classe étant une extension de la notion de type, il est possible que des classes soient composées d'autres objets. Ainsi, la classe `polygone` est, par construction, composée d'un ensemble de points *i.e.* d'objets de type `point`. Le code ci-dessous propose une solution au problème posé en utilisant un `std::vector`.

1. On peut utiliser les objets de `<random>` de la bibliothèque standard, par exemple `std::uniform_real_distribution`. Vous pouvez aussi utiliser `rand()`.

Déclaration de la classe `polygone` : fichier `polygone.h`

```
#ifndef _POLY_H_
#define _POLY_H_

#include "point.h"
#include <vector>

class polygone {
public:

  // Méthode d'initialisation d'un polygone régulier
  void initialise_régulier(unsigned int ordre, double taille);

  // Méthode d'affichage des sommets
  void affiche_sommets() const;

  // Calcul du périmètre
  double calc_perimetre() const;

  // Attributs
  std::vector<point> sommets;
};

#endif
```

Définition de la classe `polygone` : fichier `polygone.cpp`

```cpp
#include "polygone.h"
#include <iostream>
#include <cmath>
using namespace std;

void polygone::initialise_régulier(unsigned int n, double taille)
{
  // Faire en sorte que le tableau de points ait bien une taille n
  sommets = std::vector<point>(n);

  // Initialisation de chacun des sommets via la méthode initialise de la classe point
  for (unsigned int k = 0; k < n; k++) {
    double angle = (2*M_PI * k) / n;
    sommets[k].initialise(
      taille * cos(angle),
      taille * sin(angle)
    );
  }
  // Note : on peut aussi partir du tableau vide, puis faire .push_back() dans la boucle
}

void polygone::affiche_sommets() const
{
  unsigned int ordre = sommets.size();
  cout << "Nombre de sommets : " << ordre << endl;
  for (unsigned int i = 0; i < ordre; i++) {
    cout << "Sommet #" << i << " : ";
    sommets[i].affiche();
  }
}

double distance_points (point a, point b)
{
  double dx = a.x - b.x;
  double dy = a.y - b.y;
  return sqrt( dx*dx + dy*dy );
}

double polygone::calc_perimetre() const
{
  unsigned int ordre = sommets.size();

  double perimetre = distance_points(sommets[0], sommets[ordre-1]);

  // on considère que les sommets sont classés dans le tableau selon le périmètre
  // (pas de croisement des côtés du polygone)
  for (unsigned int i = 0; i < ordre-1; i++) {
    perimetre += distance_points(sommets[i], sommets[i+1]);
  }

  return perimetre;
}
```

Il est possible de légèrement raccourcir le code de `calc_perimetre()` :

```cpp
double polygone::calc_perimetre() const
{
  unsigned int ordre = sommets.size();
  double perimetre = 0;

  for (unsigned int i = 0; i < ordre; i++)
    perimetre += distance_points(sommets[i], sommets[(i+1)%ordre]);

  return perimetre;
}
```

Programme test `test_polyone.cpp`

```cpp
#include "polygone.h"
#include <random>

int main()
{
  std::mt19937 gen;
  std::uniform_real_distribution<> distrib_unif (0.5, 2.0);
  polygone poly;
  poly.initialise_régulier(6, distrib_unif(gen));
  poly.affiche_sommets();
  return 0;
}
```

Programme de calcul de périmètres `perimetres.cpp`

```cpp
#include "polygone.h"
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ofstream fichier ("perimetres.txt");

  for (unsigned int n = 1; n < 20000; n++) {
    polygone poly;
    poly.initialise_régulier(n, 0.5);
    double p_n = poly.calc_perimetre();

    // écriture dans le fichier
    fichier << n << " " << p_n << endl;

    // affichage de temps en temps
    if (n < 1000 or n%1000 == 0)
      cout << n << " -> " << p_n << endl;
  }

  fichier.close();
  return 0;
}
```

Attention, si la méthode `polygone::initialise_régulier` ne fait que des `push_back` sans effacer le tableau au préalable, il faut bien re-créer un polygone à chaque fois en déclarant `polygone poly` *dans* la boucle. En sortant de chaque itération, il sera détruit comme n'importe quelle autre variable déclarée au sein des accolades. Sinon, le nombre de points explose.

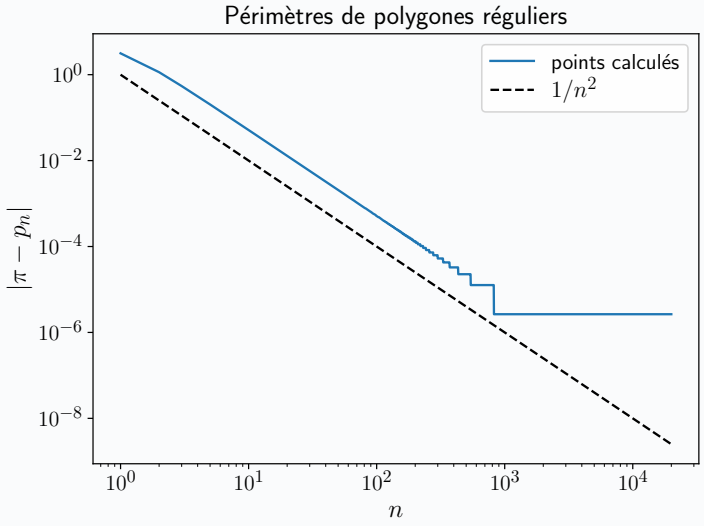
Programme Python pour afficher les périmètres

```python
import numpy as np
import matplotlib.pyplot as plt

data = np.loadtxt("perimetres.txt");
n = data[:,0]
p_n = data[:,1]

plt.plot(n, p_n, label="points calculés")
plt.xlabel("$n$")
plt.ylabel("périmètre $p_n$")
plt.title("Périmètres de polygones réguliers")
plt.axhline(y=np.pi, linestyle='--', color='black', label=r"$\pi$")
plt.xscale('log')
plt.legend()
plt.show()
```

Le périmètre de polygones réguliers de taille r converge bien évidemment vers le périmètre d'un cercle de rayon r, qui vaut $p_\infty = 2\,\pi\,r$, et simplement π ici. Les périmètres p_n convergent en $1/n^2$ vers π, comme on peut le voir sur un plot log-log :

Correction

```
plt.plot(n, np.abs(np.pi-p_n), label="points calculés")
plt.plot(n, 1/n**2, label="$1/n^2$", linestyle='--', color='black')
plt.xscale('log')
plt.yscale('log')
plt.xlabel("$n$")
plt.ylabel(r"$|\pi-p_n|$")
plt.title("Périmètres de polygones réguliers")
plt.legend()
plt.show()
```



Pourquoi, à votre avis, la courbe finit-elle ici en escalier au lieu de suivre une belle loi d'échelle ?

2.3 La classe `complexe`

Dans le précédent TD, nous représentions un nombre complexe par sa partie réelle et sa partie imaginaire. Chaque fois que nous passions un nombre complexe en argument d'une fonction nous devions passer sa partie réelle et sa partie imaginaire... En utilisant le concept d'objet, créer une classe `complexe` qui représente un nombre complexe. Transformer les fonctions `affiche`, `norme` et `argument` écrites précédemment afin que ces dernières soient des membres de la classe `complexe`. On n'oubliera pas le mot-clé `public`, et on pourra se renseigner sur la fonction `atan2` de `<cmath>`. Bien évidemment, on testera le code écrit.

Correction

On défini ensuite la classe `complexe` de la manière suivante, dans *complexe.h* (le mot-clé `public` est essentiel ; par défaut les membres et méthodes d'une classe sont privés (`private`), non accessibles depuis l'extérieur de la classe)

```
#ifndef COMPLEX_H
#define COMPLEX_H

class complexe {
public:
  double re, im;

  void affiche () const;

  double norme () const;
  double argument () const;
};

#endif // COMPLEX_H
```

Elle contient deux membres, la partie réelle et la partie imaginaire, et déclare les trois méthodes demandées par l'énoncé. On aura pris soin d'inclure le *header guard*. Notons que l'on a déclaré nos trois méthodes comme constantes. En effet, elles ne modifient pas l'objet, et c'est donc une bonne

Correction

pratique de les déclarer constantes (ce n'est ceci dit pas obligatoire).
L'implémentation *doit* être faite dans le fichier *complexe.cpp* (en général, sauf pour les méthodes dites en ligne, `inline`). On commence par inclure les fichiers d'en-tête donnant accès à la classe `complexe`, à `cout` et aux fonctions mathématiques

```
#include "complexe.h"

#include <iostream>
#include <cmath>
using namespace std;
```

Puis on définit les méthodes de la classe `complexe` dans *complexe.cpp*

```
void complexe::affiche () const {
  cout << re << " + " << im << " i" << endl;
}

double complexe::norme () const {
  return sqrt(re*re + im*im);
}

double complexe::argument () const {
  return atan2(im, re);
}
```

On n'oubliera pas la résolution de portée `complexe::` devant le nom des méthodes.
Finalement, on teste ce qu'on vient d'écrire dans *main.cpp* :

```
#include <iostream>
using namespace std;

#include "complexe.h"

int main()
{
  complexe a;

  a.re = sqrt(2);
  a.im = sqrt(2);

  a.affiche();

  cout << "Norme : " << a.norme() << endl;
  cout << "Argument :" << a.argument() << " (" << 180.0 * a.argument() / M_PI <<
  ↪  "deg)" << endl;

  return 0;
}
```

On n'oubliera pas de modifier le *makefile*.

2.4 La fonction exponentielle

Déclarer et définir une fonction *non-membre* `exponentielle(complexe)` qui prend en argument un nombre complexe et qui renvoie un nombre complexe.

Correction

Comme nous avons une classe `complexe` à notre disposition, nous pouvons retourner le résultat d'une opération via `return` au lieu de devoir passer des argument par référence. La fonction `exponentielle` a pour déclaration (dans *complexe.h*)

Correction

```
complexe exponentielle(complexe a);
```

et pour définition (dans *complexe.cpp*)

```
complexe exponentielle(complexe a)
{
  complexe result;

  result.re = exp(a.re) * cos(a.im);
  result.im = exp(a.re) * sin(a.im);

  return result;
}
```

Comme la fonction `exponentielle` n'est pas une fonction membre de la classe `complexe`, il n'y a pas d'opérateur de résolution de portée `complexe::` lors de la définition de la fonction dans le fichier `complexe.cpp`
Sans rien changer à la définition de `exponentielle` (mis à part la signature de la fonction), nous aurions pu la déclarer de la manière suivante

```
complexe exponentielle(const complexe & a);
```

Cette définition signifie que l'argument est passé par référence, mais que l'utilisateur de la fonction est assuré que la valeur de l'argument ne sera pas modifié par la fonction.

2.5 Tableaux de complexes

Dans le TD précédent (exercices 1.6 et 1.7), nous avons utilisé des tableaux de parties réelles et imaginaires. Ré-écrire le programme de l'exercice 1.6 ou 1.7 du TD1 en utilisant la classe `std::vector<complexe>`.

Rappel de méthodes qui peuvent être utiles :
- `vector<complexe> monTableau;` pour déclarer un tableau contigu d'objets de type `complexe`.
- `monTableau.push_back(nombre)` pour rajouter le un nombre à la fin du tableau. La taille du tableau est ajustée automatiquement.
- `monTableau.size()` pour connaître le nombre d'éléments dans le tableau.
- `monTableau[i]` ou `monTableau.at(i)` permet d'accéder au i-ème élément du tableau (possiblement pour le modifier). Comme corollaire évident :
  - Si la classe `complexe` possède une méthode `truc()`, `monTableau[i].truc();` permet d'appeller la méthode `truc()` sur le i-ème élément du tableau. On peut aussi utiliser `monTableau.at(i).affiche();` si la méthode `affiche()` a été déclarée `const`),
  - Si la classe `complexe` possède un membre nommé `re`, `monTableau[i].re = 1.2;` permet de modifier la valeur du membre `re` du i-ème nombre complexe du tableau.

Correction

On commence par inclure la bibliothèque `<vector>`, et on s'assure d'utiliser l'espace de nom `std`.

```
#include <vector>
using namespace std;
```

Ici on ne fait pas d'allocation de mémoire à la main ; on déclare simplement un tableau automatique `vector<complexe>`.

```
int main()
{
  unsigned int N = 0;
  cout << "N = ";
  cin >> N;

  vector<complexe> nombres;
```

Pour remplir le tableau avec $e^{\frac{2i\pi n}{N}}$, nous allons faire appel à la fonction `exponentielle` ; nous créons

Correction

donc d'abord le nombre complexe $0 + \frac{2i\pi n}{N}$, puis nous ajoutons l'exponentielle de ce nombre à la fin du tableau grâce à `push_back`.

```
for (unsigned int n = 0; n < N; ++n) {
  complexe nombre;
  nombre.re = -0.4242; // parce ça ne change rien, pourquoi pas
  nombre.im = 2 * n * M_PI / N;

  nombres.push_back(exponentielle(nombre));
}
```

Finalement, nous accédons aux parties réelles et imaginaire des nombres complexes du tableau afin de calculer leur somme

```
int main () {
  complexe somme;
  somme.re = 0;
  somme.im = 0;

  for (unsigned int n = 0; n < nombres.size(); ++n) {
    // deux façons différentes d'accéder à un élément d'un vector :
    somme.re += nombres[n].re;  // accès par indice
    somme.im += nombres.at(n).im;   // accès avec la méthode .at(n), qui vérifie en
    ↪  plus si n est un indice valide et affiche une erreur sinon (plus sécurisé
    ↪  mais un tout petit peu plus lent)
  }

  somme.affiche();

  return 0;
}
```

On remarque que grâce à l'utilisation de la classe `complexe`, le code est plus court et plus simple. On verra plus tard que, avec la surcharge d'opérateurs, on pourra écrire un code encore plus concis.

2.6 Surcharge d'opérateurs

Pré-requis : cours 5 sur la surcharge d'opérateurs.

Cette partie a pour but d'introduire la notion de surcharge d'opérateur afin de pouvoir écrire des expressions naturelles du type `complexe moins_un = complexe(0,1) * complexe(0,1)`.

Nous allons surcharger les opérateurs binaires (c'est-à-dire prenant deux arguments) comme `operator+`, `operator*`, ainsi l'opérateur unaire (c'est-à-dire prenant un seul arguments) `complexe operator-(complexe)`. Il y a deux façons de faire :

1. Déclarer les opérateurs comme des fonctions et non des membres de la classe `complexe`. Ils prennent alors tous leurs opérandes par argument. Exemple : `complexe operator+ (complexe a, complexe b)`. C'est la façon la plus intuitive. On pourra soit les déclarer à l'extérieur de la classe `complexe`, soit les déclarer comme fonctions amies[2] à l'intérieur de la classe `complexe`. On parle de *non-member operator overloading*.
2. Déclarer les opérateurs comme méthodes membres de la classe `complexe`, prennant un unique argument `complexe`. Le premier opérande est implicitement l'objet lui même, ie. `*this`, et le deuxième opérande est l'argument. Exemple : `complexe complexe::operator+ (complexe b) const`. On parle de *member operator overloading*.

Ces deux façons sont équivalentes dans la plupart des cas. Une exception : si `a` n'est pas un `complexe` mais est *convertible en* `complexe` (par exemple `1 + complexe(0,1)`), l'opération sera valide avec du *non-member*

2. Les fonctions amies ne sont pas au programme du cours mais il est possible de tomber sur du code les utilisant. Pour faire bref : une fonction amie n'est pas une fonction membre (même s'il faut la déclarer à l'intérieur de la classe avec le mot clé `friend` en préfixe), mais elle a accès aux attributs privés de la classe. Ceci ne fera sens pour vous que dans la suite du cours. Pour l'instant tous les attributs sont publiques, ça ne change donc rien.

overloading, alors qu'elle ne le sera pas avec du *member overloading*. Autre exemple : la multiplication scalaire à gauche, comme `2 * complexe(0,1)`, ne peut pas être implémenté comme membre de la classe `complexe`. En général, il faut imaginer les cas d'utilisation et choisir la façon qui ne pose le moins de problèmes.

Nous allons également surcharger les opérateurs membres comme `operator+=` ou `operator*=`. Ces opérateurs ont une signature du type ou `complexe& operator+= (const complexe&)`, et *sont* des fonctions membres, il faut donc utiliser l'opérateur de résolution de portée `complexe::` lors de la définition. Rappel : ces opérateurs de modifications renvoient non pas `void`, mais `complexe&`, renvoyant une référence sur soi-même. Ceci pour permettre d'écrire des expressions tordues comme `(a *= 2) *= 3;` ou `f(a += 1);`. Pour cela, il faut renvoyer `*this`. Ce n'est pas bien grave si vous ne voyez pas l'intérêt pour le moment. Pour d'autres opérateurs (par exemple l'affectation, cf. suite du cours), cela devient clair.

Les opérations effectuées ici sont très rapide à écrire. Mais pour des opérations un peu plus complexes et lorsque l'on choisit le *non-member operator overloading*, il est avantageux ou même nécessaire [3] d'implémenter l'opération dans (par exemple) `operator+=` et de simplement utiliser `+=` dans le corps de `complexe operator+ (complexe a, complexe b)`.

Définir les opérateurs `operator+=` et `operator-=` (argument `complexe`), ainsi que `operator*=` avec argument `complexe` ou scalaire (`double`). Pour les opérateurs binaire, afin d'explorer les différentes façon de faire :

- définir les additions et multiplications entre deux complexes comme fonctions externes
- définir la multiplication entre un scalaire et un complexe, aussi bien à droite qu'à gauche, de la façon qui vous parait la plus appropriée
- définir les soustractions et divisions entre deux complexes comme méthodes de la classe `complexe`
- définir l'opération d'inversion de signe (unaire, pour écrire par exemple `complexe b = -a;`) comme méthode

Bien sûr, tester chacune de ces opérations. Enfin, on adaptera le code de l'exercice précédent afin d'utiliser les opérateurs surchargés.

Correction

```
class complexe
{
public:
  ...
  // addition de `b' à soi
  complexe& operator+= (complexe b);
  // soustraction de `b' à soi
  complexe& operator-= (complexe b);
  // multiplication de `b' à soi
  complexe& operator*= (complexe b);
  complexe& operator*= (double b);
  // division de soi par `b' :
  complexe& operator/= (complexe b);
  complexe& operator/= (double b);
  // opposé de soi
  complexe operator- () const;
  // soustraction de `a' et `b'
  complexe operator- (complexe b) const;
  // division de `a' par `b' :
  complexe operator/ (complexe b) const;
  complexe operator/ (double b) const;  // note: non demandé dans l'énoncé, mais
  ↪  utile
  ...
};

// multiplication de `a' par `b' (que `a' et `b' soient scalaires ou complexes)
complexe operator* (complexe a, complexe b);
complexe operator* (complexe a, double b);
complexe operator* (double a, complexe a);
```

3. lorsque des attributs privés de la classe sont en jeu (cf. cours suivant)

```
// addition de `a' et `b'
complexe operator+ (complexe a, complexe b);
```

Pour les opérateurs membres, l'implémentation est directe et ressemble à :

```
complexe& complexe::operator+= (complexe right) {
  re += right.re;
  im += right.im;
  return *this;
}

complexe& complexe::operator*= (double scalar) {
  re *= scalar;
  im *= scalar;
  return *this;
}
```

De même pour la autres opérations à soi. Nous pouvons aussi prendre le parti d'utiliser les opérations binaires externes pour définir les opérations à soi. Par exemple :

```
complexe& complexe::operator*= (complexe right) {
  *this = (*this) * right;
  return *this;
}
```

Ce n'est cependant en général pas recommandé pour des raisons de performance (copie inutile, que le compilateur peut cependant souvent optimiser). On implémente alors la multiplication dans la fonction externe :

```
complexe operator* (complexe a, complexe b) {
  complexe c;
  c.re = a.re * b.re - a.im * b.im;
  c.im = a.re * b.im + a.im * b.re;
  return c;
}
```

Notons que nous avons pris les arguments par copie. Il est possible, et même recommandé pour éviter des copies inutiles, de prendre les objets `complexe` par référence constante :

```
class complexe {
  ...
  complexe& operator+= (const complexe& b);
  ...
};

complexe operator* (const complexe& a, const complexe& b);
...
```

Toutefois, par souci de clarté, on a gardé un passage par copie dans cette correction. Un objet de type `complexe` ne prennant que 2×8 octets (contre 8 pour un pointeur ou référence), la différence de performance est probablement peu perceptible, d'autant plus que le compilateur effectue certaines optimisations.

Pour les opérateurs non-membres, on peut simplement faire :

```
complexe operator+ (complexe a, complexe b) {
  complexe c;
  c.re = a.re + b.re;
```

```
  c.im = a.im + b.im;
  return c;
}
```

Mais il est possible de faire plus efficace en évitant la création d'un intermédiaire :

```
complexe operator+ (complexe a, complexe b) {
  a.re += b.re;
  a.im += b.im;
  return a;
}
```

Ici, on a pris l'argument `a` par copie, on peut donc le modifier sans rirsque de modifier la variable de l'utilisateur ! Grâce à notre définition de `complexe::operator+=(complexe)`, il est possible d'être encore plus expéditif :

```
complexe operator+ (complexe a, complexe b) {
  return (a += b);
}
```

Noter que l'on a renversé les rôles par rapport à notre implémentation de la multiplication ci-dessus : c'est `operator+` qui utilise `operator+=` et non l'inverse. C'est la façon de faire recommandée, et même nécessaire lorsqu'on agit sur des attributs privés (cf. cours suivant).

Finissons avec les divisions, toutes définies comme méthodes :

```
complexe& complexe::operator/= (double scalar) {
  re /= scalar;
  im /= scalar;
  return *this;
}

complexe complexe::operator/ (double scalar) const {
  return { re/scalar, im/scalar };
}

complexe complexe::operator/ (complexe right) {
  return (*this) * right.conj() / right.abssquare();
}

complexe& complexe::operator/= (complexe right) {
  // parce que l'on est un peu paresseux :
  *this = (*this) / right;
  return *this;
}
```

Notons que nous avons pris la liberté, pour clarifier l'implémentation de la division, de définir deux méthodes `complexe::conj()` et `complexe::abssquare()`, déclarées comme suit :

```
class complexe {
  ...
  complexe abssquare () const;
  complexe conj () const;
  ...
};
```

et définies ainsi :

```
complexe complexe::abssquare () const {
  return re * re + im * im;
}
```

Correction

```
complexe complexe::conj () const {
  return { re, -im };
}
```

On note l'utilisation de l'initialisation de structures "à la C" (`complexe {re, im}`), qui sera remplacé par la notion bien plus générale de *constructeur* (et qui ne dépend pas de l'ordre dans lequel sont définit les attributs) dans la suite du cours.

2.7 L'opérateur de flux <<

L'opérateur de flux `operator<<` peut également être surchargé pour pouvoir écrire

```
complexe i(0,1);
cout << i << endl;
```

On l'utilise ici comme un opérateur binaire dont l'opérande de gauche est `cout` et dont l'opérande de droite est notre nombre complexe. La signature à utiliser est alors `std::ostream& operator<< (std::ostream& out, const complexe& nombre)`. En effet, `std::cout` est un objet de type `std::ostream` spécialement implémenté pour afficher dans la console [4].

Pourquoi renvoit-on une référence sur le flux `out` que l'on prend en argument, au lieu de `void` ? Simplement pour pouvoir chaîner les opérations d'affichage (ie. `cout << i << endl;` au lieu de `cout << i; cout << endl`).

Correction

Dans *complexe.h*, assurons-nous d'abord que le type `ostream` est déclaré, via

```
#include <ostream>
```

Ensuite il faut déclarer l'opérateur de flux via

```
std::ostream& operator<< (std::ostream& out, const complexe& z);
```

On notera l'utilisation de `std::` car utiliser `using namespace` à l'intérieur d'un fichier **.h* est une très mauvaise idée, et doit être évité, sinon, tous les fichiers qui incluent ce fichier **.h* subiront automatiquement l'usage de cet espace de nom.

Finalement, une possible implémentation de l'opérateur de flux peut-être

```
ostream& operator<< (ostream& out, const complexe& z) {
  if (z.im >= 0) {
    out << z.re << " + " << z.im << "i";
  } else {
    out << z.re << " - " << -z.im << "i";
  }
  return out;
}
```

à écrire dans *complexe.cpp* bien sûr. Ici, nous n'avons plus besoin d'utiliser `std::` car nous sommes dans un fichier source **.cpp* dans lequel on aura pris soin d'insérer `using namespace std;`. Ce n'est pas un problème ici, car un fichier **.cpp* ne peut pas être inclus dans un autre, donc l'instruction `using namespace std;` a de l'effet uniquement à l'intérieur de ce fichier.

4. D'ailleurs, `std::ofstream` étant aussi de type `std::ostream`, cette surcharge de `operator<<` fonctionnera aussi pour l'écriture de fichiers. Même chose pour `std::ostringstream`.

2.8 Méthode de Newton et fractales

Concluons ce TD par une étude des propriétés de convergence de la méthode de Newton, en utilisant la classe `complexe` que l'on a précedemment écrite [5].

2.8.1 Introduction

La méthode de Newton permet de trouver numériquement un zéro d'une fonction $f(z)$, à condition de connaître sa dérivée $f'(z)$. Cette méthode est souvent présentée avec des fonctions réelles, mais elle est tout aussi bien applicable aux fonctions holomorphes.

Pour rappel, la méthode de Newton est une méthode itérative consistant à approximer la fonction $f(z)$ par sa linéarisation autour d'un point de travail z_n :

$$f(z) \simeq f(z_n) + (z - z_n)\, f'(z_n)$$

dont il est facile de connaitre le zéro $z^{(0)}$ (équation affine) :

$$0 = f(z_n) + (z^{(0)} - z_n)\, f'(z_n) \quad \Leftrightarrow \quad z^{(0)} = z_n - \frac{f(z_n)}{f'(z_n)}$$

Dit autrement, $z^{(0)}$ est l'intersection de la tangeante de f en z_n avec l'axe des abscisses, comme le montre le figure ci-dessous. Ce n'est évidemment pas un zéro de f, mais on a bon espoir que $z^{(0)}$ soit plus proche d'un zéro que z_n. On itère alors cette prodécure en prenant ce $z^{(0)}$ comme un nouveau point de travail :

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)}$$

en espérant converger vers un point fixe de cette relation de récurrence. Il est immédiat de voir que les points fixes de cette relation sont les zéros $z_\star$ de la fonction ($f(z_\star) = 0$).

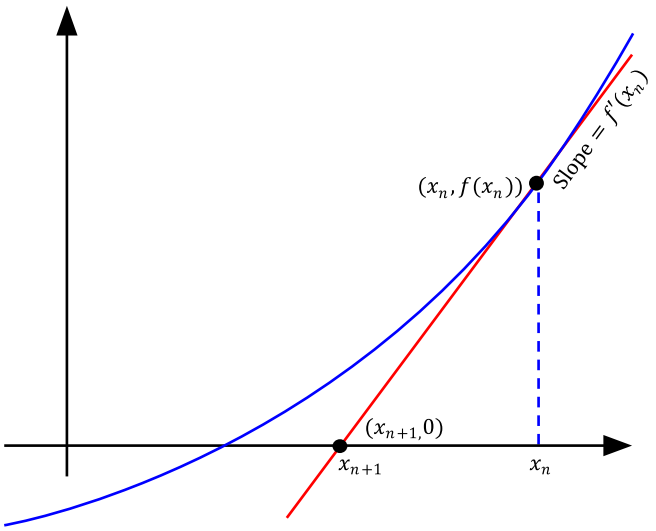


FIGURE 2.1 – Illustration d'une itération de la méthode de Newton sur une fonction réelle.

On peut montrer que dans un certain voisinage d'un zéro $z_\star$, la méthode converge bien : $z_n \xrightarrow[n\to\infty]{} z_\star$ pour tout point initial z_0 dans ce voisinage. Il est bien connu que la méthode de Newton converge extrêmement rapidement : si le zéro n'est pas aussi un point de rebroussement (ie. si $f'(z_\star) \neq 0$, alors *on double le nombre de chiffres significatifs à chaque étape !*

Ceci dit, ce voisinage dépend fortement de la fonction. En dehors de ce voisinage, tout peut arriver : ça peut ne pas converger, ou ça peut converger vers n'importe quel autre zéro de la fonction. La vitesse de convergence

5. Vous pouvez aussi utiliser la classe `std::complex<double>` de la bibliothèque standard au cas où vous n'auriez pas une classe `complexe` entièrement fonctionnelle. Son utilisation est presque identique, consultez une documentation en ligne pour plus de détails.

peut bien évidemment dépendre du point de départ. C'est ce que nous voulons étudier.

2.8.2 Implémentation

Pour cela, nous voulons écrire une fonction `methode_newton` effectuant cette prodédure (en n'allant bien sûr pas jusqu'à l'infini...), sur une fonction holomorphe f quelconque. Il nous faut donc passer en argument une fonction à `methode_newton`. Comment faire en C++ ?

Le "type" (ou plutôt conteneur standard) de fonction en C++ est `std::function<...>`, dont l'argument template est le prototype de la fonction, sous forme `type_de_retour(type_arg1,type_arg2,...)`. Ainsi, pour une fonction complexe, on peut utiliser `std::function<complexe(complexe)>`. Pour éviter d'écrire de nombreuses fois ce type un peu long, on peut définir un alias, par exemple :

```
#include <functional>
using fonc_complexe_t = std::function<complexe(complexe)>;
```

Un objet de type `std::function` peut être appelé avec les parenthèses, comme n'importe quelle fonction. Écrire une fonction de prototype

```
std::tuple<complexe,int> methode_newton (
  fonc_complexe_t f,
  fonc_complexe_t f_deriv,
  complexe z_init,
  unsigned int max_it,
  double epsilon
)
```

où `f` et `f_deriv` sont f et f' respectivement, où `z_init` est le point initial de l'itération, où `max_it` est le nombre maximal d'itérations (pour arrêter les itérations au cas où ça ne converge pas), et où `epsilon` est le seuil de convergence (lorsque $|f(z_{n_{\text{fin}}})| < \epsilon$, on dit que l'on a trouvé un zéro). La fonction `methode_newton` retourne le zéro estimé $z_{n_{\text{fin}}}$ ainsi que le numéro d'itération n_{fin} auquel l'algorithme a convergé, ou -1 autrement. On profitera des opérateurs surchargés précédemment pour simplifier l'écriture.

Ces deux informations sont retournées sous forme d'un tuple `std::tuple<complexe,int>`. Pas de panique, c'est très simple à utiliser. On peut créer un tuple de la façon suivante :

```
std::tuple<complexe,int> t = { complexe(0,1), 13 };
```

Pour dépaqueter un tuple, on peut utiliser la syntaxe

```
auto [z,n] = t;
```

où les variables `complexe z` et `int n` sont déclarées puis assignées aux valeurs contenus dans le tuple.

Pour passer une fonction à `methode_newton`, on peut

- Soit déclarer une fonction de façon habituelle `complexe f (complexe z);` puis simplement écrire `f` comme argument.
- Soit créer une fonction lambda, par exemple

  ```
  std::function<complexe(complexe)> f = [] (complexe z) -> complexe {
    ...
  };
  ```

  Il est aussi possible d'écrire directement la lambda en argument sans déclarer la variable `f` intermédiaire.

N'hésitez pas à temporairement ajouter un affichage des valeurs durant l'itération pour voir ce qu'il se passe et observer la convergence. On pourra faire en sorte d'augmenter le nombre de chiffres affichés avec `std::setprecision`. N'hésitez pas non plus à converger jusqu'à 10^{-25}. Testez votre fonction sur, disons, $f(z) = z-1$ et $f(z) = (z-1)^2$ à partir d'un point relativement éloigné du zéro $z_\star = 1$. Voyez-vous une différence de vitesse de convergence ? Pourquoi ?

Correction

Implémentation possible :

```
#include <iostream>

std::tuple<complexe,int> methode_newton (fonc_complexe_t f, fonc_complexe_t
↪  f_deriv, complexe z, unsigned int max_it, double epsilon, bool debug = false) {
  int n = 0;
  complexe fz = f(z);
  epsilon *= epsilon;
  // on regarde plutôt |f(z)|^2 < ε^2
  while (fz.abssquare() > epsilon) {
    if (n >= max_it)
      return { z, -1 };
    z -= fz / f_deriv(z);
    n += 1;
    fz = f(z);
    if (debug)
      std::cout << "newton iter " << n << " : f(z) = " << fz << " at z = " << z <<
      ↪  std::endl;
  }
  return { z, n };
}
```

On observe une petite optimisation : on effectue la comparaison $|f(z)|^2 < \epsilon^2$ car la racine carrée nécessaire au calcul de $|f(z)|$ est coûteuse.

Notez qu'il n'y a aucun besoin de déclarer une variable locale complexe pour z_n, l'argument `z` (passé par copie) fait très bien l'affaire. Par contre, comme $f(z)$ rentre à la fois dans le test de convergence et la relation de récurrence, autant donc définir une variable `fz` pour éviter de calculer $f(z)$ deux fois par itération.

Enfin, on s'est permis d'ajouter un flag `debug` pour activer l'affichage des itérations au besoin. Notre `main()` ressemble à :

```
#include <iomanip>

complexe f (complexe z) {
  complexe zm1 = (z-complexe(1,0));
  return zm1*zm1;
};

complexe f_deriv (complexe z) {
  return 2.*(z-complexe(1,0));
};

int main () {
  std::cout << std::setprecision(25) << std::fixed;
  complexe z_0 = {3,-10};
  auto [z_zero, nit] = methode_newton(f, f_deriv, z_0, 100, 1e-25, true);
  std::cout << z_zero << ", " << nit << std::endl;
  return 0;
}
```

Pour $f(z) = (z-1)^2$, la convergence vers le zéro $z_\star = 1$ est plus lente car $f'(z_\star) = 0$. On ne gagne qu'un chiffre de précision toutes les ~ 4 itérations. Autrement dit, la convergence est "seulement" exponentielle au lieu de super-exponetielle.

2.8.3 Comportement fractal

Dans la suite, nous allons étudier ce que donne la méthode de Newton sur un polynôme. Prenons par exemple $f(z) = z^3 - 1$, qui est un grand classique. Combien de zéro possède-t-il sur $\mathbb{C}$, et quels sont-ils ? Libre à vous d'en étudier un autre, ou même une fonction plus exotique, à conditions que vous connaissiez ses zéros.

Correction

De façon générale, les zéros de $f(z) = z^n - 1$ sont les racines n-èmes de l'unité, $\{e^{2\pi i\, k/n}\}_{0\leq k<n}$, au nombre de n. Pour $f(z) = z^3 - 1$, on a donc

$$1, \quad e^{2\pi i/3} = \frac{-1+i\sqrt{3}}{2}, \quad e^{2\pi i\, 2/3} = \frac{-1-i\sqrt{3}}{2}$$

Sa dérivée est $f'(z) = 3z^2$. Nous définissons

```
fonc_complexe_t f = [] (complexe z) { return z*z*z - complexe(1,0); };
fonc_complexe_t f_deriv = [] (complexe z) { return 3.*z*z; };
```

Notons que le calcul `z*z` est effectué deux fois par itération, ce qui n'est pas optimal. Ceci dit, il est possible que le compilateur le remarque et évite ce double calcul.

Mettez-vous proche de $z_0 = 0$ et variez légèrement cette valeur de départ dans différentes direction. Sur quel zéro tombez vous ? Êtes-vous d'accord avec votre voisin ? Que fait le temps de convergence à mesure que l'on rapproche z_0 de 0 ? Pourquoi ? [6]

Pour y voir un plus clair, nous allons parcourir z_0 de façon systématique sur une grille de taille $N \times N$ maillant $\{x + iy,\ x \in [-a, +a],\ y \in [-a, +a]\}$, et afficher le résultat avec matplotlib. Nous allons numéroter chaque zéro ainsi que lui définir une couleur, afin d'afficher une carte des bassins de convergence (ie. l'ensemble des points qui convergent vers un zéro donné).

Pour chaque point de la grille, exécutez la méthode de Newton et déterminez sur quel zéro la méthode a convergé. Choisissez -1 lorsqu'il n'y a pas eu convergence. Pour stocker les résultats, au lieu de définir des matrices dans notre programme, nous pouvons écrire directement dans un fichier lisible par `numpy.loadtxt`. Écrivez donc le numéro du zéro obtenu au fur et à mesure dans un fichier, en respectant le format demandé par `numpy.loadtxt` (c'est à dire une grille, avec des espaces horizontalement et des retours à la ligne verticalement). Compilez avec le flag `-O3` pour activer les optimisations du compilateur.

rrection

On a simplement à effectuer deux boucles imbriquées, en effectuant des pas $da = a/N$ sur la partie réelle et imaginaire de z_0 :

6. Regardez f'...

Correction

```cpp
#include <fstream>

int main () {
  std::ofstream fichier_bassins_converg ("newton_bassins_converg.txt");

  fonc_complexe_t f = [] (complexe z) { return z*z*z - complexe(1,0); };
  fonc_complexe_t f_deriv = [] (complexe z) { return 3.*z*z; };
  constexpr unsigned int n_zeros = 3;
  complexe zeros [n_zeros] = { {1,0}, {-1/2.,sqrt(3)/2}, {-1/2.,-sqrt(3)/2} };

  int N = 1000;
  double a = 1.0, da = a/N;

  for (int i = -N; i <= N; i++) {
    for (int j = -N; j <= N; j++) {

      complexe z_0 = {i*da, j*da};
      auto [z_zero, nit] = methode_newton(f, f_deriv, z_0, 50, 1e-15, false);

      // détermination du numéro du zéro
      int num_zero = -1;
      for (int k = 0; k < n_zeros; k++) {
        if ((z_zero-zeros[k]).abssquare() < 1e-10)
          num_zero = k;
      }

      fichier_bassins_converg << num_zero << " ";
    }
    fichier_bassins_converg << "\n";
    std::cout << i << std::endl;
  }

  return 0;
}
```

On utilise `std::ofstream` pour écrire dans un fichier.
Note : nous avons omis `fichier_bassins_converg.close()`. C'est tout à fait acceptable, le fichier est automatiquement fermé lorsque la variable `fichier_bassins_converg` est détruite. Nous allons voir comment cela fonctionne dans la suite du cours.

Le fichier complet de correction est *TD2/methode_newton_manualzero.cpp*, et le code python pour afficher les figures est disponible dans un notebook jupyter *TD2/methode_newton.ipynb*.

Avec Python, lire la grille résultante et la transformer en une image RBG, c'est-à-dire une grille de taille `(N,N,3)`. Ainsi, `img[42,24,:]` correspond au pixel $(42, 24)$, et vaut `[r,g,b]` où `r`, `g`, `b` sont entre 0 et 1 (`[0,0,0]` pour du noir, `[1,1,1]` pour du blanc). Pour les points qui n'ont pas convergés, choisir du noir par exemple. Astuce : forcer `dtype=int` pour `numpy.loadtxt` et utiliser des constructions du type `img[ carte == ..., : ]` pour générer l'image. Ensuite, `plt.imshow` permet d'afficher directement cette image. On prendra soin d'utiliser l'argument `extent` pour mettre les axes à la bonne échelle.

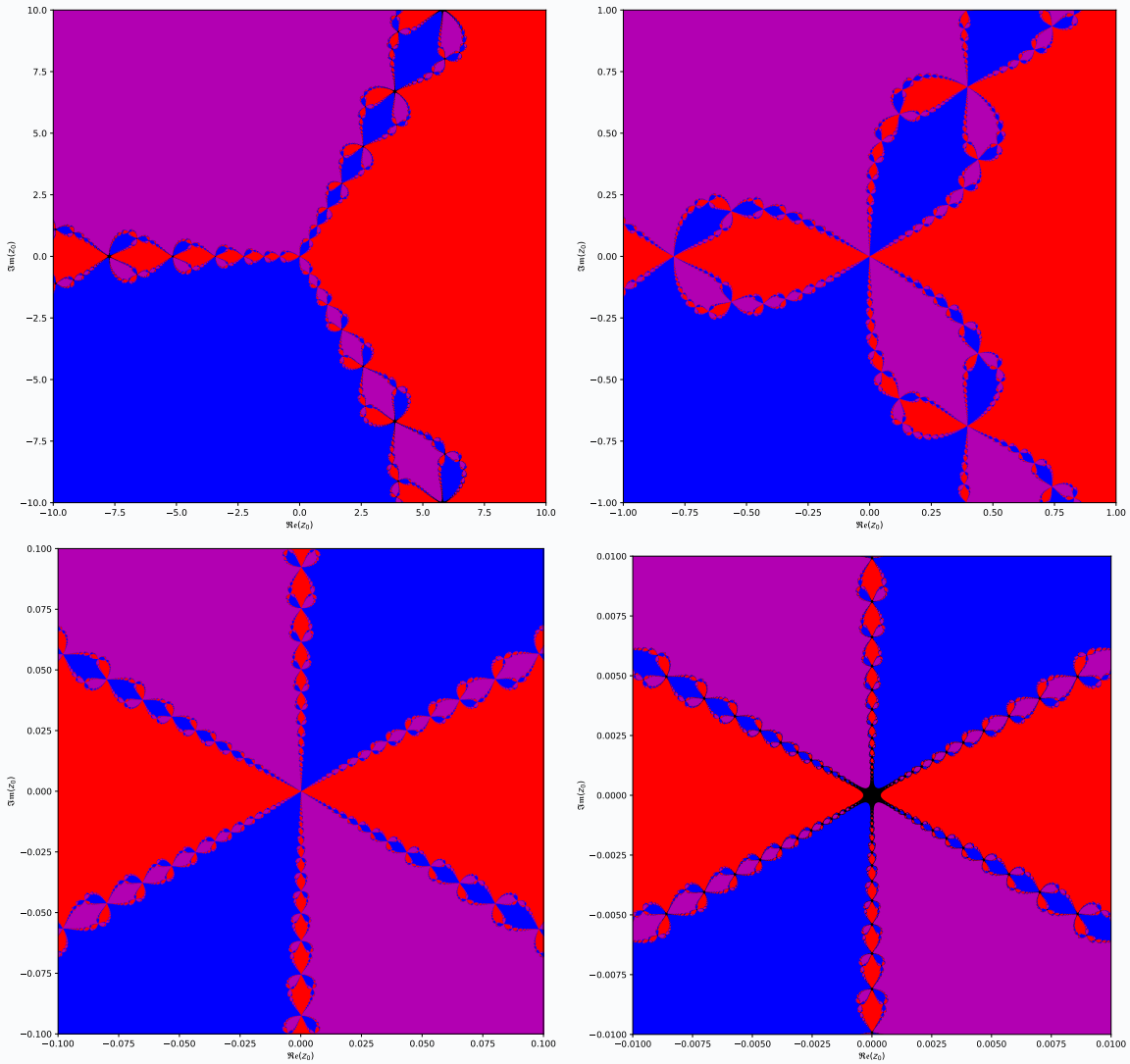
On obtient une magnifique fractale, la *fractale de Newton*. Une caractéristique typique des fractales est l'auto-similarité. Vérifiez que notre fractale est bien auto-similaire, par exemple en dézoomant ou en zoomant autour du point $z_0 = 0$, et en zoomant encore, et encore, et encore...

Correction

```
import numpy as np
import matplotlib.pyplot as plt

Iz = np.loadtxt("newton_bassins_converg.txt", dtype=int)
RGB = np.zeros((Iz.shape[0],Iz.shape[1],3))
Iz = Iz.transpose()[::-1,:]

RGB[ Iz == 0, : ] = [1,0,0]
RGB[ Iz == 1, : ] = [0.7,0,0.7]
RGB[ Iz == 2, : ] = [0,0,1]

a = 1.0
plt.figure(figsize=(10,10))
plt.imshow(RGB, extent=[-a,a,-a,a])
plt.show()
```

La seule subtilité est au niveau de `Iz.transpose()[::-1,:]`. C'est nécessaire pour deux raisons : transposition parce que imshow n'a pas l'orgine standard, et inversion de l'axe vertical car on a parcouru de bas en haut.

Voici les bassins de convergence de la méthode de Newton pour les trois zéros de $z^3 - 1$, pour $a = 10$, $a = 1$, $a = 0.1$ et $a = 0.01$:

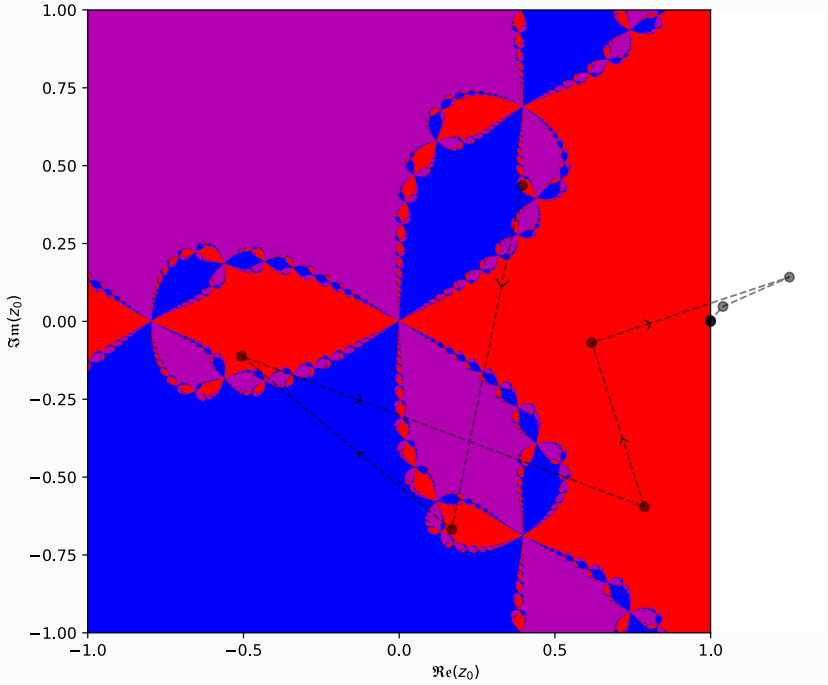


chaque zéro ayant sa couleur. On a bien une auto-similarité, quoi que différente à grande échelle (3

Correction

"bras" pour $|z| > 1$) qu'à petite échelle (6 "bras" pour $|z| < 1$). On remarque un petit défaut de convergence pour $a = 0.01$, simplement car `max_it` était un peu faible ici (40).

Sans rentrer dans les détails, tentez d'intuiter la raison de cette auto-similarité en pensant à la trajectoire de z_n. Voici une telle trajectoire, permetant de remarquer que les lobes des régions fractales ne sont que des images (éventuellement multiples) du bassin de convergence principal :



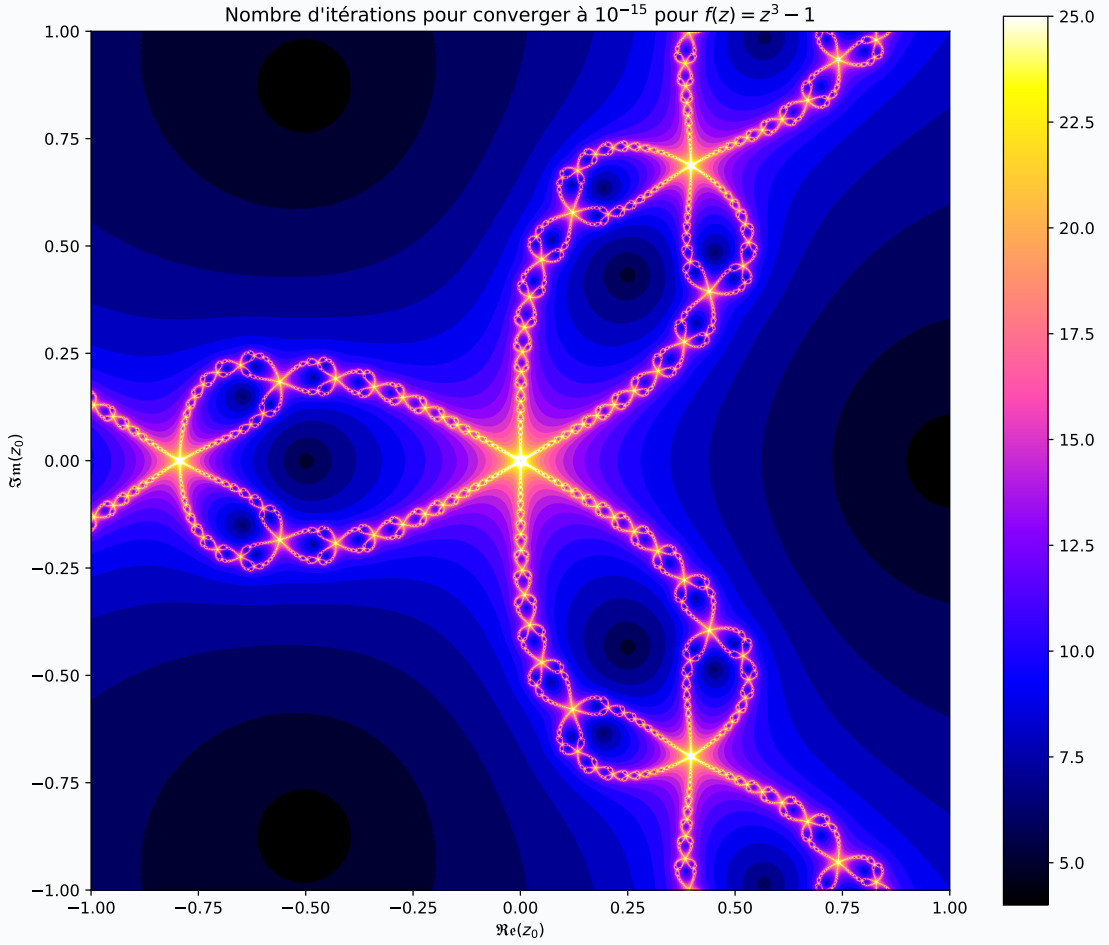
Pour les motivés, on peut afficher une autre carte, possiblement plus belle encore, celle du nombre d'itérations nécessaires pour atteindre le seuil de convergence, $n_{\text{fin}}(z_0)$. Au lieu de créer une carte RBG, affichez directement la matrice avec `plt.imshow` et une colormap de votre choix. Mieux encore, on peut s'amuser à combiner les deux cartes.

Correction

Résultat en écrivant `nit` au lieu de `num_zero`, et en exécutant

```
Nit = np.loadtxt("newton_niter.txt", dtype=int)
plt.imshow(Nit, extent=[-a,a,-a,a], cmap='gnuplot2', vmin=3, vmax=25)
```

Correction



Note : Les factales de Newton s'inscrivent dans le domaine plus général de la *dynamique holomorphique*. Pour une compréhension plus profonde des mécanismes à l'œuvre et de l'émergence de ce comportement fractal, on pourra visionner l'excellente vidéo de 3Blue1Brown sur le sujet : `https://www.youtube.com/watch?v=-RdOwhmqP5s`

2.8.4 Amortissement [bonus]

Avec le polynôme z^3-1, on observe une convergence de la méthode de Newton presque partout. Mais il n'est pas garanti que ça soit toujours le cas. Étudions $f(z) = z^4 - 2z^2 - 1$. Combien de zéros ce polynôme a-t-il ? Au lieu de déterminer les zéros à la main, on va laisser notre programme le faire... puisque c'est justement l'utilité de la méthode de Newton. Modifiez le programme pour découvrir les zéros au fil du parcours de la grille. Retracez une carte des bassins de convergence pour ce nouveau polynôme. Avons-nous toujours convergence (quelque soit `max_it`) partout ?

Correction

Remplaçons notre tableau statique de zéros par un tableau dynamique :

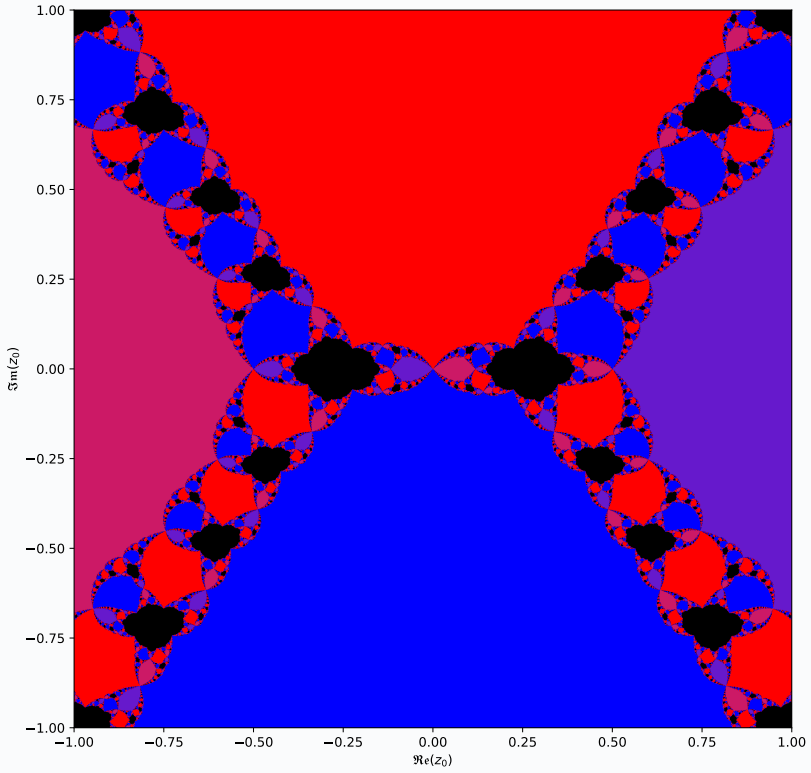
```
std::vector<complexe> zeros;
```

Il suffit d'adapter le code de détermination du numéro du zéro pour ajouter un éventuel nouveau zéro à la liste. Par exemple :

Correction

```
// détermination du numéro du zéro
int num_zero = -1;
if (nit >= 0) {
  for (int k = 0; k < zeros.size(); k++) {
    if (std::norm(z_zero-zeros[k]) < 1e-10)
      num_zero = k;
  }
  if (num_zero == -1) {
    zeros.push_back(z_zero);
    num_zero = zeros.size() - 1;
  }
}
```

On rajoute bien sûr une quatrième couleur à notre programme Python. On obtient :



On remarque des zones, ici en noir, où la méthode de Newton ne converge jamais. Elle oscille en fait entre deux lobes noirs indéfiniement.

Remplaçons la relation de récurrence de la méthode de Newton par une version amortie

$$z_{n+1} = z_n - h \cdot \frac{f(z_n)}{f'(z_n)}$$

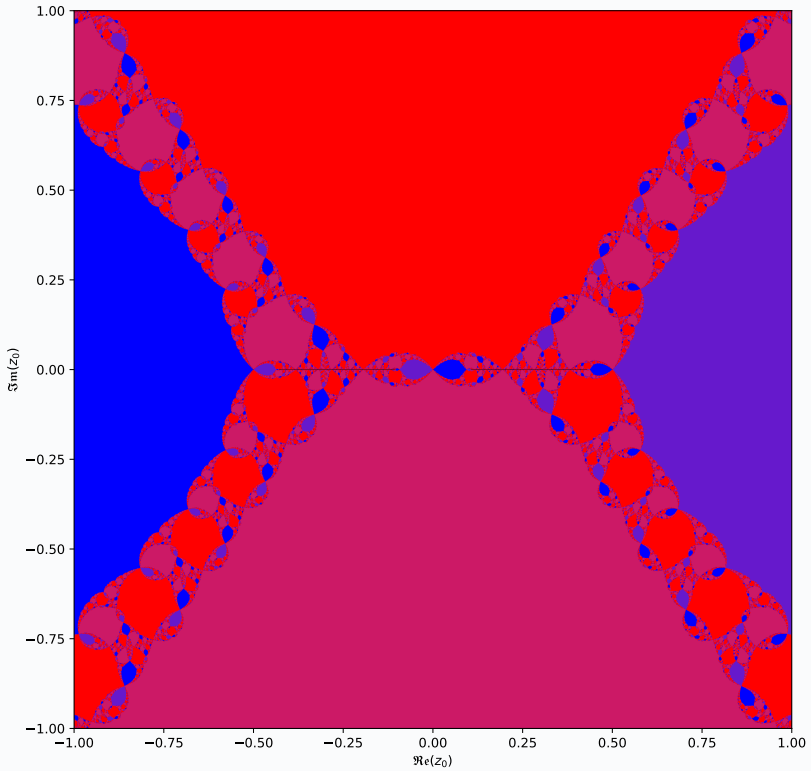
où $h \in]0, 1]$. On parle d'amortissement car à chaque itération, le pas $|z_{n+1} - z_n|$ est plus petit quand $h < 1$.

Modifiez votre programme et retracez une carte des bassins de convergence. Comme la méthode est nécessairement plus lente à converger, il peut être nécessaire d'augmenter le nombre d'itérations maximales. Qu'observez-vous ? En quoi cela peut-il est considéré comme une amélioration ?

rection

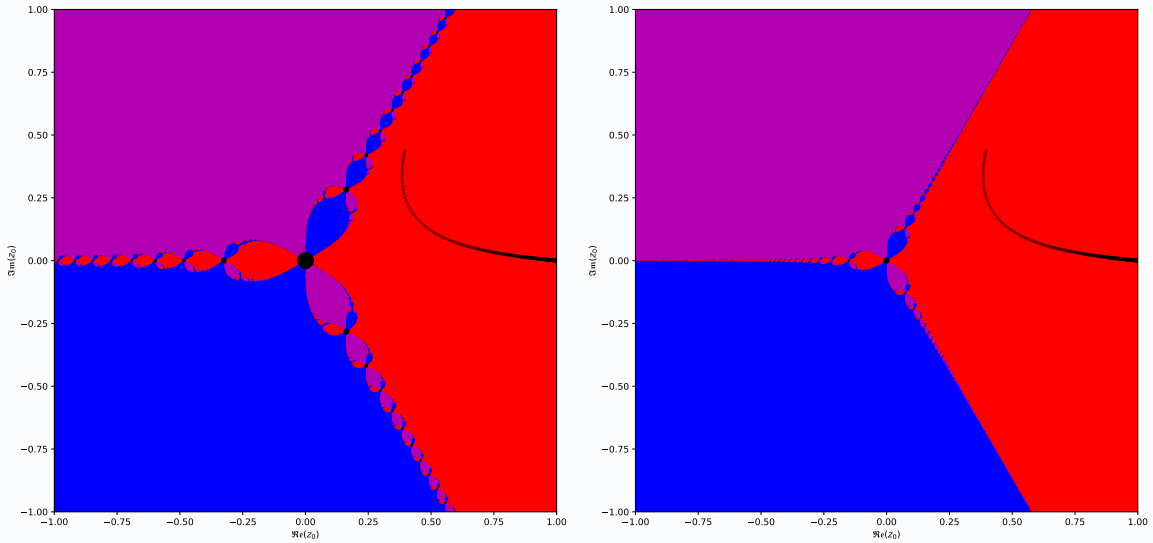
Même avec un amortissement modeste de $h = 0.8$, on élimine totalement les oscillations de la méthode de Newton (pour cette fonction du moins), et on a une convergence presque partout :

Correction



Au vu du faible ralentissement induit par cet amortissement, il est clairement avantageux de l'introduire pour améliorer les propriétés de convergence de la méthode de Newton.

Pour conclure, revenons à $f(z) = z^3 - 1$. Voici les bassins de convergence pour $a = 1$, et pour des facteurs d'amortissement de $h = 0.1$ (gauche ; nécessite d'augmenter le nombre d'itérations à 200 pour $\epsilon = 10^{-8}$) et $h = 0.01$ (droite ; nécessite d'augmenter le nombre d'itérations à 2000 pour $\epsilon = 10^{-8}$) :



On voit que les zones chaotiques diminuent en taille. Cela veut dire que le zéro que l'on va obtenir est bien plus prévisible. On remarque en fait que les bassins de convergence se rapprochent des "bassins versants" (au sens de l'écoulement de l'eau sur une surface $|f(z)|$ vers les zéros). Tracez $|f(z)|$ pour vous en convaincre. Pour $h \ll 1$, on effectue en fait une descente de gradient, qui est beaucoup moins efficace mais aussi beaucoup plus prédictible.