Sorbonne Université | **M1 Mathématiques**
2018–2019 | **4M056 : Programmation en C++**

TP noté 2 : recherche de zéros par la méthode de Newton

À l'issue du TP, chaque binôme envoie par email ses fichiers `.hpp` et `.cpp`. **Il est impératif de mettre en commentaire, dans tous les fichiers, les nom, prénom et n° d'étudiant de chacun des membres du binôme.**

Description Soit $f : \mathbb{R} \to \mathbb{R}$ une fonction de classe C^1. Soit $x \in \mathbb{R}$. On définit alors la suite $(x_n)_{n\in\mathbb{N}}$ par $x_0 = x$ et

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

Si α est un zéro de f, alors il existe un voisinage de α tel que, pour tout x dans ce voisinage, la suite $(x_n)_{n\in\mathbb{N}}$ définie précédemment converge vers α.

Cela permet, étant donnée une estimation a priori d'un zéro d'une fonction, de construire une suite qui converge vers ce zéro. Si l'estimation de départ est trop mauvaise, alors la suite peut ne pas converger. Cette méthode marche également sur les nombres complexes pour les fonctions analytiques.

Le but de ce TP est de programmer un modèle de fonction qui implémente cette suite et de le tester sur quelques cas simples.

1. Dans un fichier `newton.hpp`, écrire un *template* de fonction

```cpp
template <class Func, class K>
K newton_method(const Func & f, K x0, unsigned max_iter, double precision);
```

qui itère la suite (x_n) initialisée à `x0` jusqu'à ce que $|f(x)|$ soit inférieur à `precision` ou bien que le nombre d'itérations dépasse `max_iter`. Nous supposerons que :

- sur tout corps `K` admissible dans le *template*, `std::abs(x)` (défini dans `<cstdlib>`, `<cmath>` ou `<complex>` selon les types) calcule $|x|$.
- les objets `f` de type quelconque `F` possède un opérateur `()` tel que `f(x)` soit de type `K` pour `x` de type `K`, ainsi qu'une méthode `deriv` telle que `f.deriv(x)` donne la dérivée (de type `K` également) au point `x`.

2. Faire ce qu'il faut pour que, si l'argument `precision` n'est pas utilisé, alors `precision` prend la valeur par défaut 10^{-12}.

3. Dans un fichier `test_sin.cpp`, on souhaite tester notre *template* sur la fonction $s(x) = \sin(\pi x)$ dont les zéros sont les nombres entiers. Pour cela, on crée une courte classe :

```cpp
class Sinclass {
        public:
                double operator() (double x) ..........
                double deriv (double x) ..........
};
```

qui implémente la fonction s.

4. Dans ce même fichier `test_sin.cpp` , ajouter une fonction `main()` qui, partant d'un point $x = 5.4$, cherche un zéro de la fonction s par la méthode de Newton (en utilisant bien évidemment le *template* précédent). Vous utiliserez pour `max_iter` la valeur qui vous semblera raisonnable (testez !) pour atteindre la précision par défaut.

5. La méthode de Newton est très efficace pour calculer des racines carrées réelles ou complexes. En effet, pour calculer $\sqrt{a}$ pour $a > 0$, il suffit de chercher un zéro de $x \mapsto x^2 - a$. Dans un fichier `sqrt.cpp` , on déclare une classe `Poly2` pour décrire les fonctions de type $x \mapsto x^2 - a$ et une fonction `sqrt_approx` :

```
class Poly2 {
	protected:
		double a;
	public:
		Poly2(double a0);
		...
};
double sqrt_approx(double a, unsigned max_iter, double precision);
```

Compléter le code de la classe `Poly2` pour qu'elle soit utilisable comme premier argument du template précédent.

6. Écrire le code de `sqrt_approx` qui ne doit faire appel qu'à un objet de la classe `Poly2` et au template `newton_method` et calcule une approximation de $\sqrt{a}$.

7. Ajouter une classe `Poly2C` et une fonction `sqrt_approx_c` qui fassent les mêmes calculs dans le cas complexe, et non plus réel. Nous vous rappelons que `<complex>` permet de manipuler les nombres complexes avec un type `std::complex<double>` , un constructeur à deux arguments (parties réelle puis imaginaire) et les opérations arithmétiques usuelles.

8. Dans le fichier `sqrt.cpp` toujours, écrire une fonction `main()` qui teste les fonctions précédentes qui calculent les racines carrées de 4, 9, i et $(1/2) - (\sqrt{3}/2)i$ avec une précision de 10^{-12} et affichent le résultat dans le terminal.

9. **Zéros de polynômes.** Dans un fichier `polynome.hpp` , soit la classe :

```
template <class K>
class Polynome {
	protected:
		int degree;
		K * coeff;
	public:
		...
};
```

pour coder les polynômes à coefficients dans `K` : `degree` donne le degré d du polynôme et `coeff` est un tableau de taille `degree+1` tel que `coeff[i]` est le coefficient du monôme de degré `i` du polynôme. Écrire le code de l'opérateur `()` et de la méthode `deriv()` .

10. Écrire un constructeur `Polynome(const std::vector<K> & v)` à partir d'un vecteur qui contient la suite des coefficients (faites attention au fait que le degré est la taille de `v` diminuée de 1). Nous supposerons que `v[i]` est le coefficient du monôme de degré `i`.

11. Écrire également le constructeur par copie, le destructeur de cette classe et l'opérateur d'affectation.

12. Dans un programme `test_poly.cpp`, chercher les zéros réels de

$$P(X) = X^5 - 4X^4 + 4X^3 - 16X^2 + 3X - 12$$

dans l'intervalle $[-10, 10]$ en essayant de faire converger la méthode de Newton à partir de conditions de départ idoines.