POLYTECH
NICE-SOPHIA

TP « Fractal »

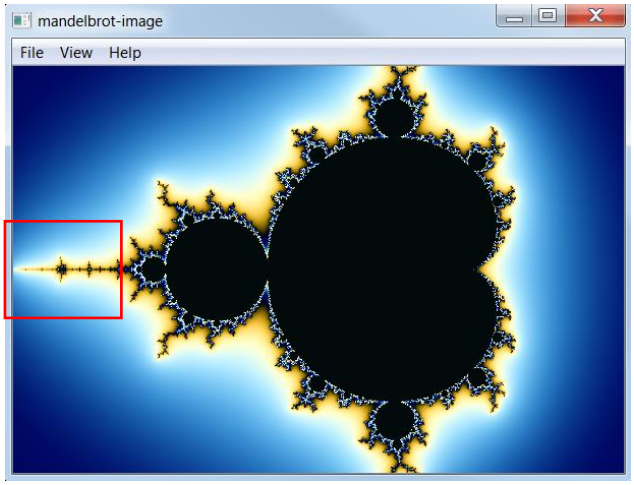
Dans ce TP, on s'appuie sur le dessin d'une fractale pour introduire des éléments nouveaux du langage C++ : les fonctions « *friends* », les nombres complexes, la mesure du temps cpu, la programmation multi-cœurs.

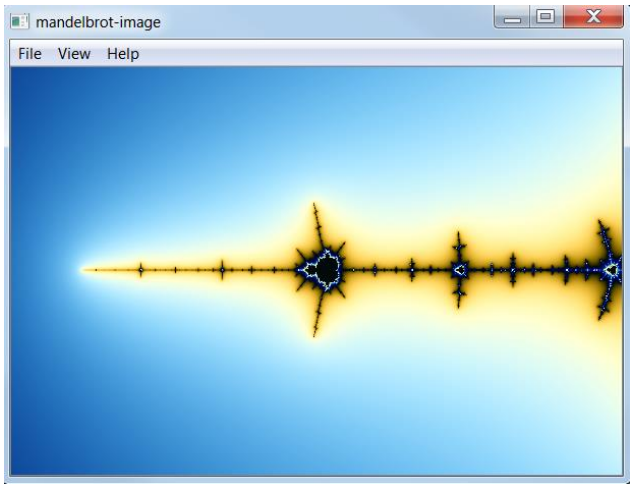
1 - Dessin d'une fractale.

1.1 Introduction

A lire : http://en.m.wikipedia.org/wiki/Mandelbrot_set
A voir : http://vimeo.com/12185093

Dans ce TP, on souhaite dessiner une fractale et obtenir des résultats tels que ci-dessous.

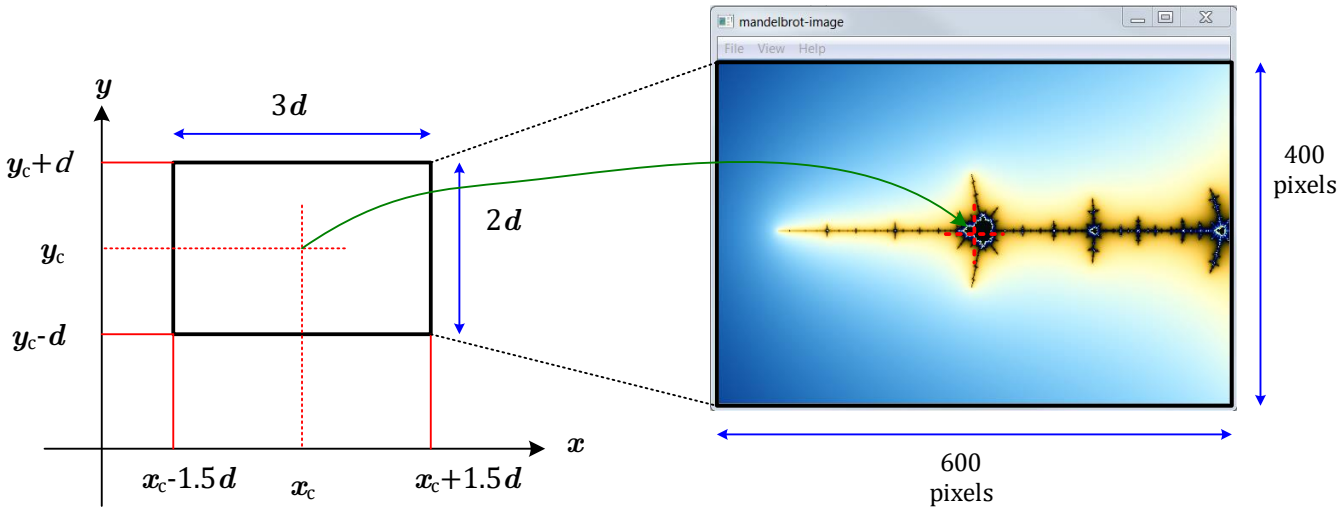




L'image du bas est un zoom sur le rectangle rouge de l'image du haut.

Plusieurs variables contrôlent le dessin :
x_c, y_c le centre du rectangle,
d la demi hauteur du rectangle,
La demi longueur sera fixée à $1.5d$,
La largeur et hauteur de la fenêtre en pixels w et h.

La figure ci-dessous permet de mieux comprendre comment passer des coordonnées réelles vers/depuis les coordonnées pixels.



Domaine « réel » Domaine « image »

Une simple transformation linéaire permet de passer d'un domaine à l'autre.
Ce sera à vous d'établir ces transformations.

Pour dessiner une fractale de Mandelbrot, on doit calculer, pour chaque pixel du domaine image, le point de coordonnée (x_0, y_0) dans le domaine réel. On a donc

$$\begin{cases} x_c - 1.5d \leq x_0 \leq x_c + 1.5d \\ y_c - d \leq y_0 \leq y_c + d \end{cases}$$

Avec le nombre complexe $c_0 = x_0 + iy_0$, on construit la série complexe $z_n$tel que

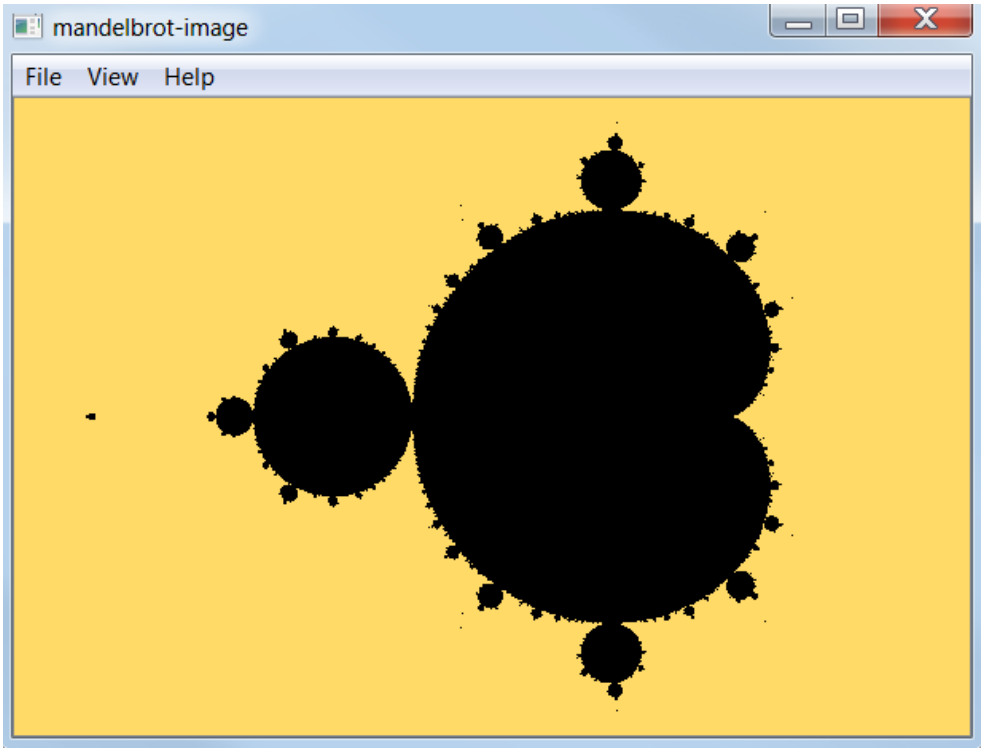
$z_{n+1} = z_n^2 + c_0$, avec $z_0 = 0$.

Dès que $|z_n| \geq 2$, le point c_0 est à l'extérieure de l'ensemble de Mandelbrot. A l'inverse, les points c_0 pour lesquels $|z_n| < 2$ pour toute valeur de n sont des points de l'ensemble de Mandelbrot.
Pour limiter les temps de calcul, on définit un nombre maximum d'itération n_{MAX}, si $|z_{n_{MAX}}| < 2$, alors le point c_0 est considéré comme appartenant à l'ensemble de Mandelbrot.
Pour la suite de ce TP, on prendra $n_{\text{MAX}} = 512$.

1.2 Premier Code

Prendre comme point de départ votre projet `TriangleImage`. Il vous faut ajouter une classe `MandelbrotImage` dérivée de `QImage` sur le même modèle que la classe `TriangleImage`.

Votre résultat final est une image de dimension 600x400 qui doit être identique à l'image ci-dessous.



Les dimensions de votre image doivent être des paramètres : l'utilisateur peut par exemple agrandir la fenêtre.

Vous utiliserez les nombres complexes de type `std::complex<double>` de la librairie STL, qui est documentée avec le lien http://en.cppreference.com/w/cpp/numeric/complex.

Pour cette image, les paramètres sont les suivants :

$$\begin{cases} x_c = -1/2 \\ y_c = 0 \\ d = 1 \end{cases}$$

A titre indicatif, les points de l'ensemble de Mandelbrot sont en noir, les points hors de l'ensemble sont avec la couleur RGB = (255, 218, 103).

Pour vous aider, voici un extrait de mon code source.

```
...
// iterate on each point of the image
for (int yp = 0; yp < height; ++yp) {
  // convert vertical pixel domain into real y domain
  double y0 = v_pixel2rect (yp);
  for (int xp = 0; xp < width; ++xp) {
    // convert horizontal pixel domain into real x domain
    double x0 = h_pixel2rect (xp);
    std::complex<double> c0(x0, y0);
    ...
```

Comment appelle-t-on les fonctions `v_pixel2rect` et `h_pixel2rect` ?
Indices : bien lire la dernière section.

1.3 Temps de calcul pour une image

On souhaite mesurer le temps utilisé pour calculer l'image.
STL offre de nombreuses fonctions pour cela, voir http://en.cppreference.com/w/cpp/chrono qui donne un exemple qui ne passe pas une « *code review* » moderne. En adaptant le code, j'ai barré des bouts de code et ajouté des … que vous devrez compléter et/ou remplacer.

Vous utiliserez par contre l'horloge `high_resolution_clock` pour plus de précision et le temps écoulé sera donné en microsecondes

```
...
std::chrono::time_point<std::chrono::system_clock> start, end;
... start = std::chrono::system_clock::now();
MandelbrotImage mandelbrot_image(mandelbrot_width, mandelbrot_height);
... end = std::chrono::system_clock::now();
... elapsed_time = ...
std::cout << "Info: image calculated in " << ... << ...;
```

Pour information, j'obtiens sur la console pour le calcul de l'image 600x400 le résultat suivant :

```
INFO: image calculed in 4751271us
```

1.4 Les fonctions « friend »

Pour améliorer la lisibilité du résultat, on se propose d'ajouter des séparateurs pour les milliers (une virgule en anglais), pour obtenir :

```
INFO: image calculed in 4,751,271us
```

On peut donc envisager une classe, appelé `Commify`, qui sera utilisé comme ceci

```
std::cout << "Info: image calculated in " << ELEC4::Commify(my_value) << ...
```

La fonction `Commify()` est un constructeur qui retourne donc un objet de la classe. Cette objet est ensuite affiché avec l'opérateur `<<`. Nous avons déjà utilisé la surcharge de l'opérateur `<<` pour qu'il affiche correctement un objet d'une classe. (Voir le mini-projet pour plus d'information)

Dans notre cas, on peut donc écrire

```
class Commify {
private:
  ...
public:
  explicit Commify(int value) {
    ...
  }
  friend std::ostream& operator<<(std::ostream &os, const Commify &c) {
    ...
    return os;
  }
};
```

A vous de compléter cette classe pour obtenir le résultat souhaité. On pourra utiliser la classe `ostringstream` qui permet de récupérer une chaine de caractère, voir exemple avec ce lien http://en.cppreference.com/w/cpp/io/basic_ostringstream/str.

1.5 Livrables pour cette section

Aucun, voir section 3

2 - Fractal avec des couleurs

Bien relire la page wikipedia sur les fractales en particulier la section « *Smooth Coloring* ». On va implémenter une variante de l'algorithme présentée en s'appuyant sur les courbes splines (vues en TP) pour obtenir des dégradés de couleurs.

2.1 Tableau de couleurs

Vous construirez un tableau de 2048 couleurs en utilisant les informations de la Section 6. Bien réfléchir comment construire ce tableau, environ 20 lignes de code à écrire, avec du copier/coller.

2.2 Calcule de la couleur

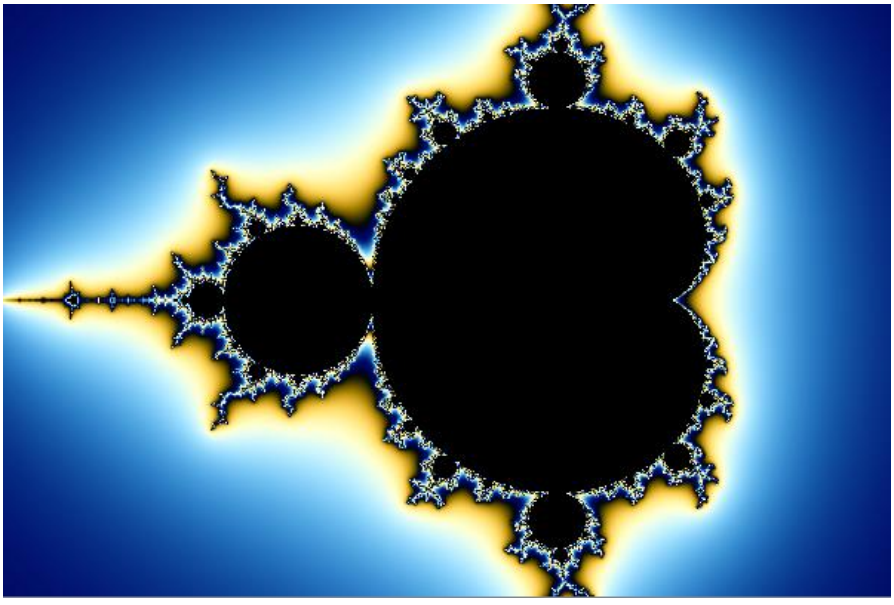
L'algorithme de wikipedia n'utilise pas la condition $|z_n| \geq 2$ pour sortir de la boucle de calcul mais $|z_n| \geq 256$. La couleur du point c_0 est une fonction de n, le nombre d'itération, et de $|z_n|$.

On utilisera la variante ci-dessous pour calculer i, l'index dans le tableau de couleurs :

$$v = \log_2(\log_2(|z_n|^2))$$
$$i = 1024 \cdot \sqrt{(n + 5 - v)}$$

Vous utiliserez un modulo 2048 si i est supérieur ou égale à 2048 avant de trouver la couleur dans le tableau.

Modifiez votre programme, compilez et vérifiez que pour les paramètres identiques à la section 1, vous obtenez bien cette image :



Relevez le temps de calcul.

3 - Amélioration des performances

3.1 Votre machine

Trouver pour votre ordinateur les éléments suivants (utilisez par exemple CPUZ)

Nom du Processeur	*Intel Core i7 4800MQ*
Nombre de cœurs	*4*
Nombre de threads	*8*
Fréquence de base	*2.7GHz*
Instructions SSE	*SSE 4.1 / 4.2, AVX 2.0*

3.2 Calcul Parallèle avec les « *threads* »

A lire et voir
http://www.cs.fsu.edu/~lacher/courses/DOCS/c++_threads.html
https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial.
https://youtu.be/M80e32We_Xc
https://youtu.be/13dFggo4t_I

On peut vite perdre énormément de temps à optimiser une méthode, il faut donc s'assurer que la fonction choisie représente une part significative du temps total (voir loi de Amdahl)

On fait l'hypothèse que la boucle principale représente bien la majorité du temps passé.
Pour mémoire la boucle principale fait une itération sur tous les pixels de l'image

...

```
        // iterate on each point of the image
        for (int yp = 0; yp < height; ++yp) {
          // convert vertical pixel domain into real y domain
          double y0 = v_pixel2rect(yp);
          for (int xp = 0; xp < width; ++xp) {
            // convert horizontal pixel domain into real x domain
            double x0 = h_pixel2rect(xp);
            std::complex<double> c0(x0, y0);
            ...
          }
        }
```

On va donc créer une fonction : `process_sub_image(int i, int m)` qui réalisera une fraction de la boucle principale. Implémentez cette fonction, le code ci-dessus devient donc

```
...
for(int i= 0; i < max_threads; i++) {
   process_sub_image(i, max_threads);
}
```

Après ces transformations, votre code doit compiler et s'exécuter comme en section 2.

La mise en place des threads se fait ensuite facilement, vous pouvez utiliser l'exemple suivant. Il sera à compléter avec un objectif simple : lorsque vous développez une application multi-thread, il est important de pouvoir repasser en version avec un thread très facilement et donc ici en mettant la variable `max_thread` à 1 sans autre changement dans le code.

```
std::vector<std::thread> threads;
int max_threads = 4;
for(...) {
  threads.emplace_back([=]() {
    process_sub_image(i, max_threads);
  });
}
...
for (auto &thread_elem : threads) {
  thread_elem.join();
}
```

Faire varier le nombre de threads et tracez une courbe : temps d'exécution en fonction du nombre de threads

POLYTECH
NICE-SOPHIA

3.3 Livrables pour les Sections 1, 2 & 3

1) Une archive avec l'ensemble de votre projet qui doit compiler sans erreur dans l'environnement QT et donne la fractale attendu.
2) Un document .pdf décrivant les points suivants :
   - La classe `MandelbrotImage` : constructeur, destructeur, ses membres, …
   - Les fonctions de la librairie `std::complex` que vous avez utilisées.
   - Les foncteurs utilisés pour le passage des coordonnées pixel aux coordonnées x et y
   - 2 captures d'écran avec les paramètres suivants :

```
d = 1.0,     x_c = -0.5,                 y_c = -0.0
d = 0.0002, x_c = -1.7685736562992577, y_c = -0.000964296850972570525;
```

   - Explication sur le paramètre de `emplace_back()` de la section 3.2.
   - Courbe du temps d'exécution en fonction du nombre de threads avec vos commentaires.

4 - Amélioration des performances

4.1 Profilage

Plusieurs outils permettent de faire un profilage du temps passé dans chaque fonction.
J'ai obtenu le tableau suivant avec l'outil linux gprof

```
Flat profile:

Each sample counts as 0.01 seconds.
  %
 time   name
 20.00  double std::norm(std::complex<double> const&)
 13.33  std::complex<double> std::operator+<double>(std::complex<dou
  6.67  std::complex<double>::imag[abi:cxx11]() const
  6.67  std::complex<double>& std::complex<double>::operator+=<doubl
  6.67  std::complex<double> std::operator*<double>(std::complex<dou
```

Il est donc pertinent de travailler sur la fonction `std::norm()` qui calcule $|z|^2 = x^2 + y^2$.

Une piste possible est de ne pas utiliser la librairie `std::complex`, et de coder explicitement le calcul des parties réelles et imaginaires.

Notez que :

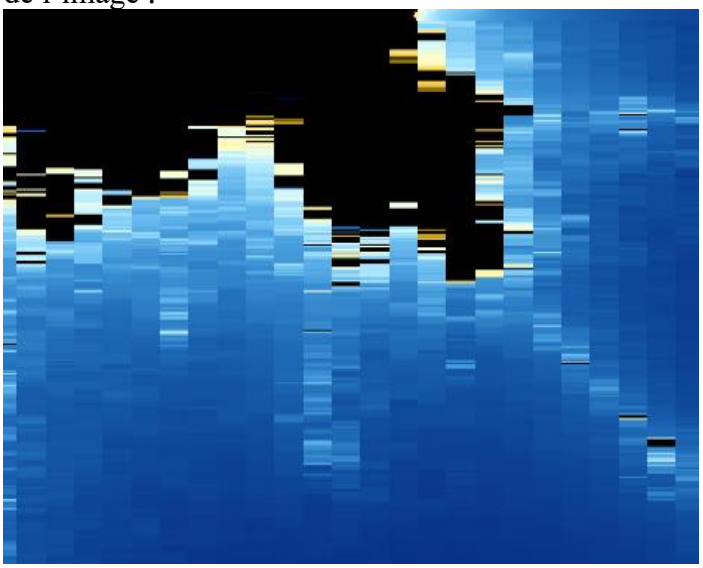
$$|z|^2 = x^2 + y^2$$
$$\Re(z^2) = x^2 - y^2$$

On peut optimiser les étapes du calcul pour ne calculer x^2 et y^2 qu'une fois.
A vous ! Faire bien attention que l'image doit être absolument identique avec celle de la section 3.

Mesurez vos performances, comparez avec la section 3.

4.2 Dégradation de la Qualité

Lorsque le facteur de zoom (d) est inférieur à 10^{-15}, on observe une dégradation de la qualité de l'image :



Est-ce normal et savez-expliquer pourquoi ?

4.3 Livrables pour la Section 4

1) Une fonction `process_sub_image_optimized()` sera ajoutée à votre code.
2) Un chapitre supplémentaire dans votre rapport avec les points suivants :
   Comment avez-vous vérifié que l'image est identique à celle de la section 3.
   Quel gain de performance observez-vous ?
3) Expliquer pourquoi la qualité de l'image se dégrade lorsque l'on zoom trop fortement

5 - Image Dynamique

Avec quelques modifications mineurs dans votre code, les coordonnées de base :

$$\begin{cases} x_c \\ y_c \\ d \end{cases}$$

peuvent être passées en argument du constructeur de l'image.

On peut ajouter une fonction virtuelle `keyPressEvent` (voir ici pour un exemple) dans la classe `MainWindow` qui est appelée dès qu'une touche du clavier est enfoncée. On peut donc détecter facilement si les touches de navigation (*up, down, left, right*) ou les touches + et – ont été appuyées, puis recalculer x_c, y_c, d et recalculer et afficher une nouvelle image. On implémente ainsi un effet de « *pan* » et de « *zoom* ».

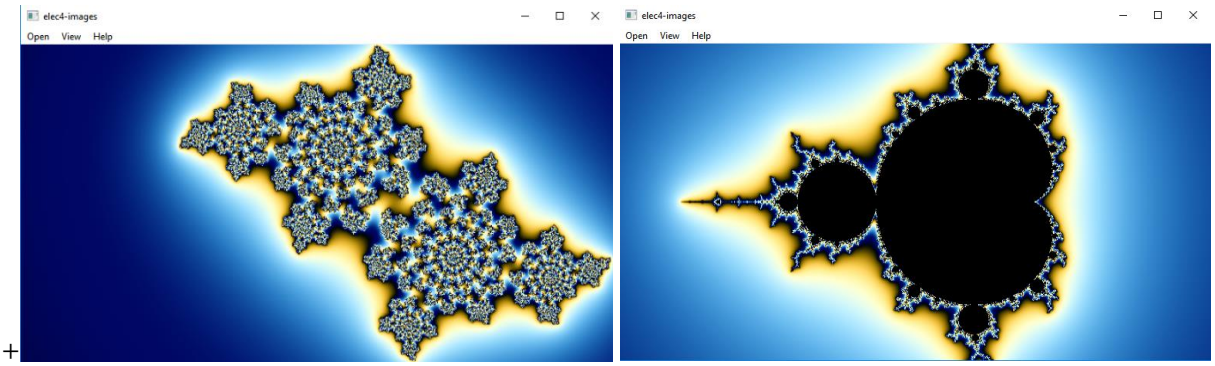
En Option :

En fouillant la doc QT, vous pouvez facilement trouver comment passer en mode « f*ull screen* » et obtenir ainsi des dessins hautes résolutions. Attention au temps de calcul qui ne permet pas d'avoir des images fluides.

5.1 Changement de Fractal

Ajouter une touche (« t ») pour permettre le passage d'une fractale de Mandelbrot à une fractale Julia :

$z_{n+1} = z_n^2 + c_0$, avec $z_0 = (x + iy)$ *et* $c0 = -0.4 + 0.6i$

+ 

5.2 Livrables pour cette Section

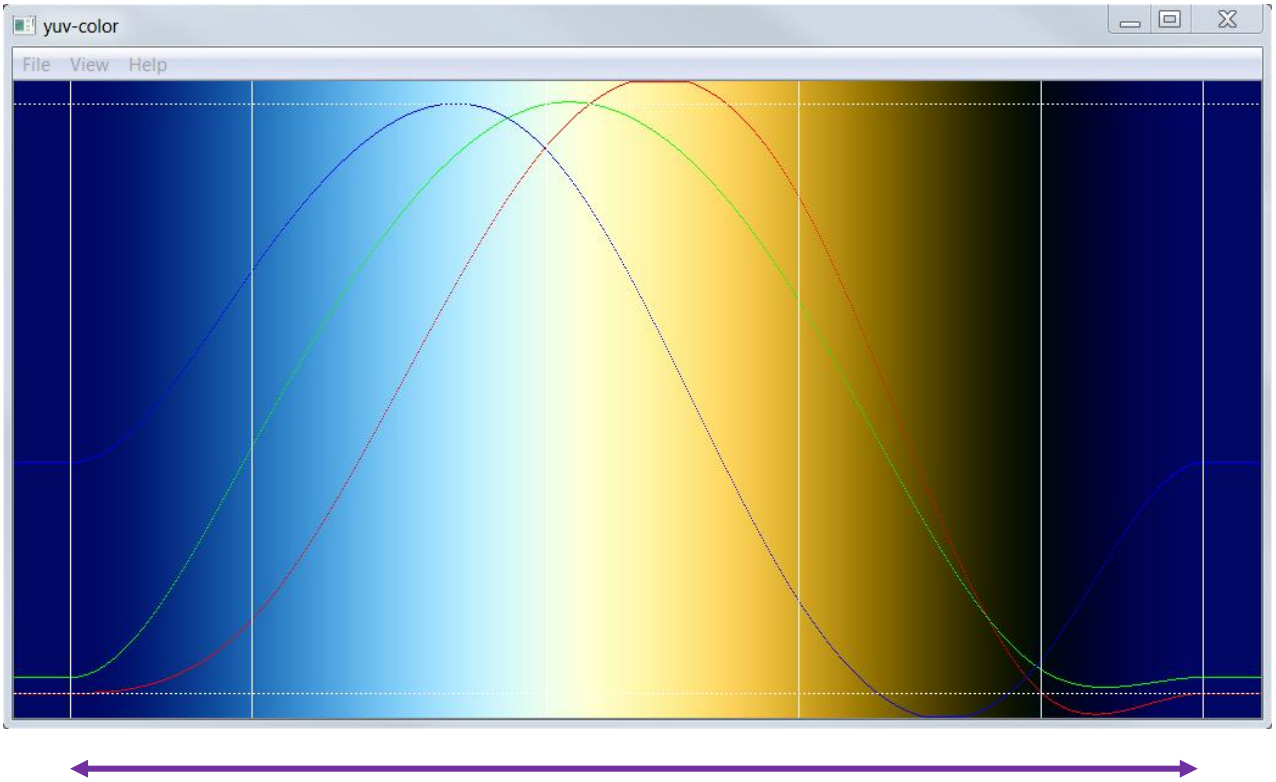
1) Votre code avec les modifications permettant de naviguer et de zoomer dans l'image (up, down, left, right, zoom in, zoom out, toggle)

2) 2 Capture d'ecran pour Mandelbrot et Julia avec les coordonnées suivantes

```
xc      = -1.02390549069711123
yc      =  0.249109414132206636
zoom    =  0.00737869762948382968
```

6 - Annexe : Gradient de Couleur

Le gradient de couleur ci-dessous a été obtenu simplement en utilisant des courbes splines du TP précédent.



Les 3 courbes (rouge, verte et bleue) nommées $y_r(x), y_g(x), y_b(x)$ sont des splines définies par les points suivants :

```
vector<double> xs{   0.,   0.16,   0.42,   0.6425,  0.8575};

vector<double> yr{   0.,  32.  , 237.  , 215.    ,  0.    };
vector<double> yg{   7., 107.  , 255.  , 170.    , 10.    };
vector<double> yb{ 100., 183.  , 235.  ,  40.    , 15.    };
```

POLYTECH
NICE-SOPHIA

7 - Annexe : Vers un Code Pro

7.1 Les Foncteurs

A Chercher : Google sur foncteurs en anglais : *function objects* ou *functors*

Vous avez sans doute écrit plusieurs fonctions d'écrêtage comme ci-dessous avec un copier-coller des fonctions de fichier en fichier

```
static int clamp_to_rgb(double v) {
  return (v <= 0.0) ? 0 : ((v >= 255.0) ? 255 : static_cast<int>(v));
}
static int clamp3(int v, int v_min, int v_max) {
  return (v < v_min) ? 0 : ((v > v_max) ? v_max : v);
}
```

On souhaite donc mutualiser ces fonctions dans un seul fichier : `elec4_util.h`
Faire les copier-coller nécessaire.
Compiler, avez-vous warning ?
Expliquez le rôle du mot clef `static` ?

Pour remplacer ces fonctions et obtenir un code réutilisable on se propose de créer un « *functor* » avec template c'est-à-dire une classe comme ci-dessous

```
template<typename tpl_t>
class Clamp {
private:
  tpl_t min_;
  tpl_t max_;
public:
  int operator()( …à vous de completer cette classe…

};
```

Modifiez votre code pour utiliser votre foncteur, par exemple, je trouve dans mon code après modification :

```
ELEC4::Clamp<double> clamp_to_rgb(0., 255.);
QColor vertical_color(clamp_to_rgb(yr), clamp_to_rgb(yg), clamp_to_rgb(yb));
..
ELEC4::Clamp<int> clamp_to_height(0, height - 1);
setPixel(xp, clamp_to_height(yp_r), qRgb(255, 0, 0));
```

7.2 D’autres foncteurs

Il est aussi intéressant de construire un foncteur pour une interpolation linéaire, par exemple

```
double x_min = -0.05;
double x_max =  1.05;
ELEC4::LinearInterpolate h_interpolator(0.0, static_cast<double>(width - 1),
                                        x_min, x_max);

for (int xp = 0; xp < width; ++xp) {
  double  x = h_interpolator(xp);
  …
```