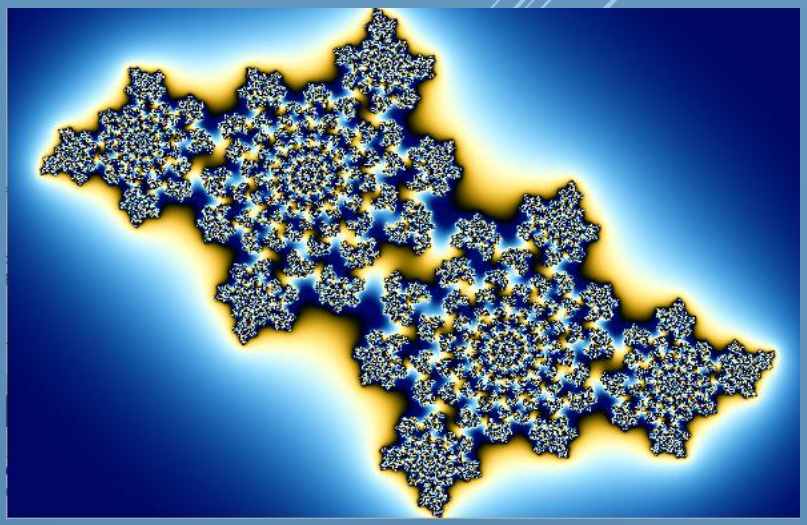
TP N°5
C++ POUR L'EMBARQUÉ



GUO Ran – BARRIGA Ricardo

Elec4 2019-2020

Professeur Bernard PLESSIER

04/04/2020

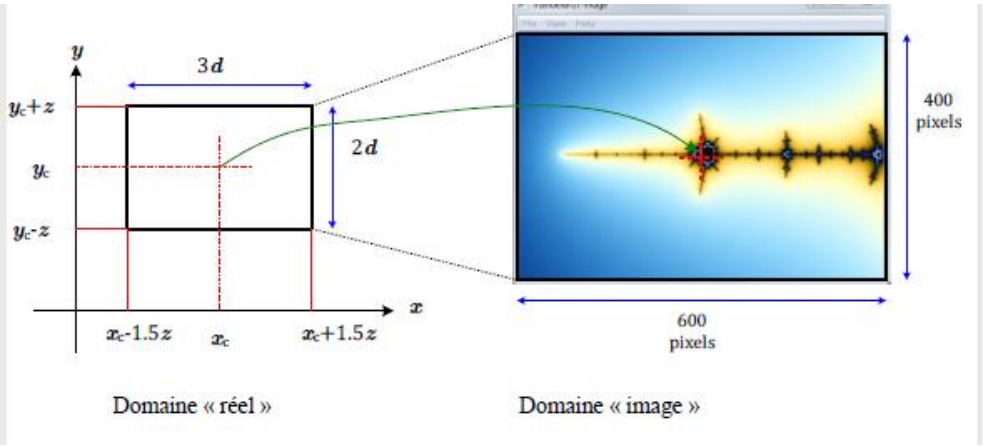
1) Dessin d'une fractale :

1.1) Introduction :

Dans ce TP nous allons dessiner une image fractale pour introduire des éléments nouveaux du langage C++ tels que les fonctions « friends », les nombres complexes, la mesure du temps CPU, la programmation multi-coeurs.

1.2) Premier code :

Nous avons pris comme point de départ le projet Triangle Image. On a utilisé une conversion linéaire pour passer du plan de pixel au plan réel, comme l'image ci-dessous :




```
1.  double v_pixel2rect (const int yp){
2.  return (static_cast <double>(Pheight)-
    (static_cast<double>(yp)))/static_cast <double>(Pheight) *height -((height/2)-
    yc);
3.  }
4.
5.  //
6.  double h_pixel2rect (const int xp){
7.  return (static_cast <double>(xp))/static_cast <double>(Pwidth) * width -
    ((width/2)-xc);
8.  }
```

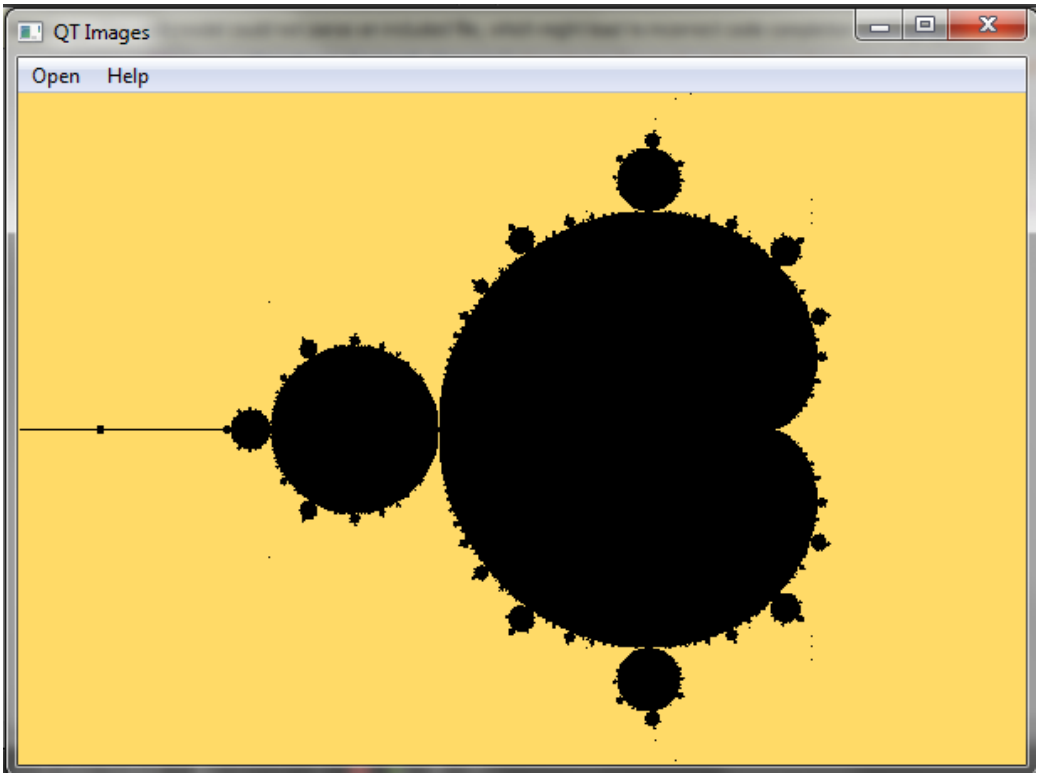
Ces deux méthodes vont permettre passer de x et y dans le domaine pixel au domaine réel. Ensuite on déclare une variable complexe "z" en utilisant une classe prédéfinie dans la bibliothèque std. On fait donc: std::complex<double> z(0,0);

De même on déclare la variable complexe $c0 = x0 + iy0$. Où x0 et y0 sont les points du plan réel. On doit satisfaire la condition Zn < 2 pour être dans l'ensemble de Mandelbrot. Ensuite on fait une itération pour diminuer n qui est initialisé à 512. On va arrêter la boucle dans le cas où le

module de Zn est supérieur à 2 et donc on est dehors de l'ensemble de Mandelbrot. Si n est négative on va peindre le pixel de couleur noire, et sinon on utilise la couleur : QColor(255, 218, 103). Cette logique est dans le code ci-dessous:

```
1.              while(n--){
2.                 z=std::pow(z,2)+c0;
3.                 if(std::abs(z)>=2)
4.                     break;
5.              }
6.              if(n<0){
7.                  painter.setPen(QPen(QColor(0,0,0)));
8.                  painter.drawPoint(xp,yp);
9.              }
10.             else{
11.                 painter.setPen(QPen(QColor(255, 218, 103)));
12.                 painter.drawPoint(xp,yp);
13.             }
```

Avec ce code on a obtenu l'image ci-dessous :



1.3) Temps de calcul pour une image

Dans cette partie on s'intéresse à mesurer le temps utilisé pour calculer l'image.

On utilise ces lignes de code dans le mainwindows.cpp :

```
std::chrono::time_point<std::chrono::system_clock> start = std::chrono::system_clock::now();
MandelbrotImage Mandelbrot(2,3,-0.5,0,512);
```

```
std::chrono::time_point<std::chrono::system_clock> end = std::chrono::system_clock::now();
std::chrono::duration<double> elapsed_time = end - start;
std::cout << "Info: image calculated in " << elapsed_time.count() * 10e6 <<"us" << std::endl;
//
```

On obtient 2.71573 *10^7 us.

```
Info: image calculated in 2.71573e+007us
```

1.4) Les fonctions friend :

Maintenant on veut une meilleure lisibilité du résultat. Pour cela on veut ajouter des séparateurs pour les milliers. On va utiliser une classe appelé Commify. Celui-ci est un constructeur qui retourne donc un objet de la classe. Cette objet est ensuite affiche avec l'opérateur <<.

On peut voir ci-dessous une partie de la classe Commify. On utilise la fonction-déclaration friend pour donner une fonction à l'opérateur <<. Friend accorde à la classe Commify l'accès aux membres privés et protège à la classe où la déclaration friend apparaît.

```
1.   friend std::ostream& operator<<(std::ostream &os, const Commify &c) {
2.   std::string s=c.toString();
3.  // std::ostringstream ss;
4.   size_t len=s.length();
5.   for(long int i=(int)len-3;i>0;i-=3)
6.       s.insert(i, ",");
7.       os<<s;
8.
```

2) Fractal avec des couleurs :

Dans cette partie nous implémenter une variante de l'algorithme faite dans l'indice antérieur pour ajouter un dégradé de couleurs basé dans le TP4.

2.1) Tableau de couleurs

Nous allons construire un tableau de 2048 couleurs pour chaque spline(RVB). Ci-dessous on montre le code pour créer le tableau du spline 1(rouge).

```
double sp1[2048];
double sp2[2048];
double sp3[2048];
//sp1 red
double x=-0.05;
for (int i=0; i<2048; i++){
    if (spline1.get_value(x)>255){
        sp1[i]=255;
    }
    else if (spline1.get_value(x)<0){
        sp1[i]=0;
    }
    else {
        sp1[i]=spline1.get_value(x);
    }
    x=x+(1.10/2048);
}
```

2.2 Calcule de la couleur

Dans cette partie, notre condition pour sortir de la boucle est |Zn|>=256. Ensuite on cherche l'indice dans nos tableaux de couleurs, et on applique les affectations ci-dessous :

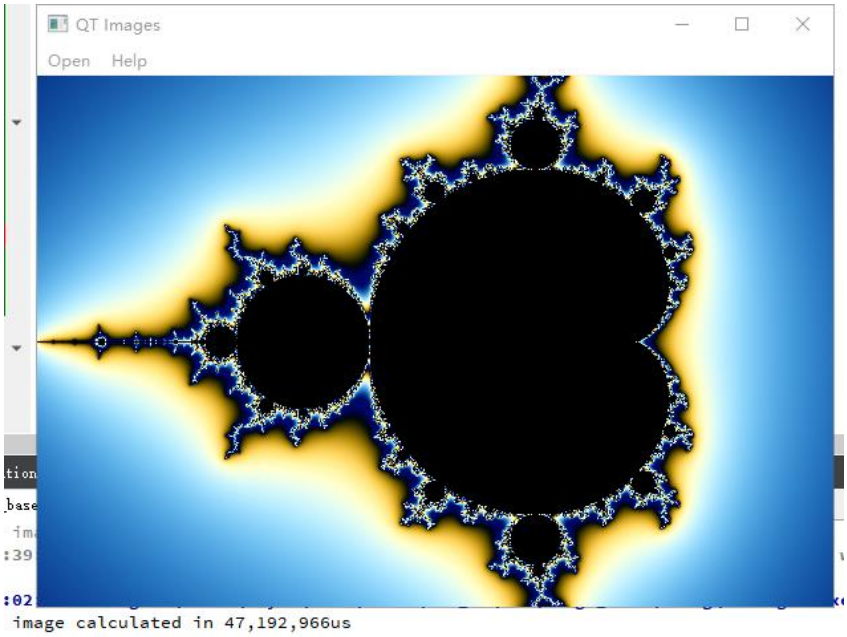
v=log2(log2(|zn|2)) et i=1024·√(n+5−v)

On implémente le code ci-dessous :

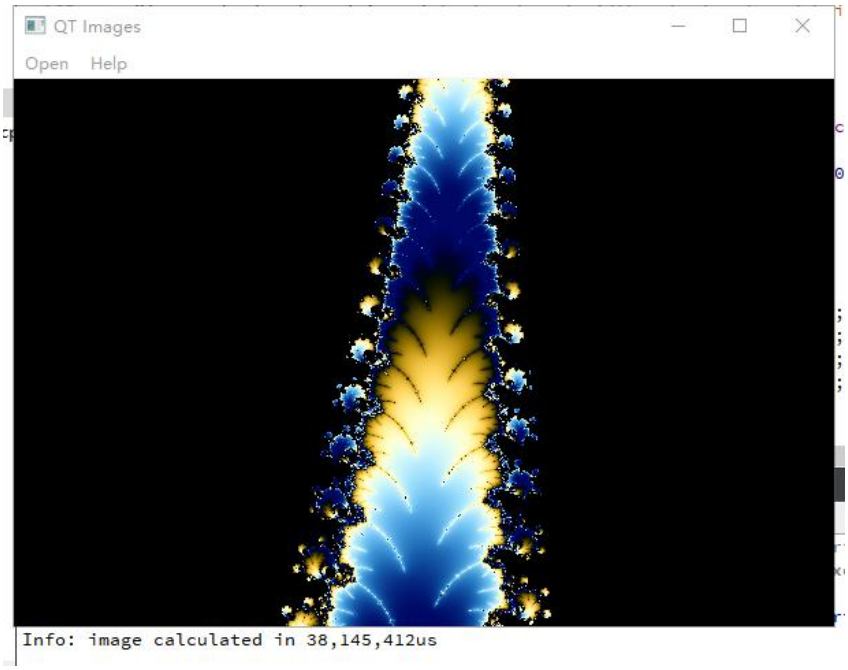
```
double v=log(log(std::abs(z)*std::abs(z))/log(2))/log(2);
int i=1024*sqrt((Mandelbrot.getNmax()-n)+5-v);

if(i>=2048) i=i%2048;
painter.setPen(QPen(QColor(sp1[i], sp2[i] , sp3[i])));
painter.drawPoint(xp,yp);
```

On a utilisé std::complex pour déclarer les variables complexes Z et C0.

Et on obtient cette image dans 47,192,966 us :



On change les paramètres : d = 0.0002, x_c = -1.7685736562992577, y_c = -0.000964296850972570525 et on obtient l'image ci-dessous dans 38,145,412 us :



On présente ensuite notre class MandelbroImage. Cette class a comme attributs : height, widht, xc,yc et n_max.

```
1.  class MandelbrotImage : public QImage
2.  {
3.  private:
4.      double height;
5.      double width;
6.      double xc;
7.      double yc;
8.      int  n_max;
9.
10. public:
11.     MandelbrotImage(const int h=2,const int w=3,const double x=-
    0.5,const double y=0,const int nmax=512):
12.         height(h),width(w),xc(x),yc(y),n_max(nmax){};
13.
14.     double v_pixel2rect (const int yp);
15.
16.     double h_pixel2rect (const int xp);
17.
18.     int getNmax(){return this->n_max;}
19. };
```

3) Amélioration des performances :

3.1) Votre machine

On a utilisé CPUZ pour trouver les éléments ci-dessous :

```
--------------------------------------------------------------------------

Socket 1            ID = 0
    Number of cores     2 (max 2)
    Number of threads   4 (max 4)
    Manufacturer        GenuineIntel
    Name            Intel Core i5 7200U
    Codename        Kaby Lake-U/Y
    Specification       Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
    Package (platform ID)   Socket 1515 FCBGA (0x7)
    CPUID           6.E.9
    Extended CPUID      6.8E
    Core Stepping       B0
    Technology      14 nm
    TDP Limit       15.0 Watts
    Tjmax           100.0 °C
    Core Speed      1396.9 MHz
    Multiplier x Bus Speed  14.0 x 99.8 MHz
    Base frequency (cores)  99.8 MHz
    Base frequency (ext.)   99.8 MHz
    Stock frequency     2700 MHz
    Max frequency       3100 MHz
    Instructions sets   MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3
    Microcode Revision  0x8E
```

3.2) Calcul Parallèle avec les « threads »

Dans cette partie on veut s'assurer que la fonction choisie représente une part significative du temps total. Nous allons fait l'hypothèse que la boucle principale représente bien la majorité du temps passé.

On a créé une fonction process_sub_image(int i, int m) qui réalisera une fraction de la boucle principale.

```
1.  void process_sub_image(int i,int m){
2.      int n_[(Pwidth/m)][Pheight];
3.      std::complex<double> zn_[Pwidth/m][Pheight];
4.
5.      calculer(n_,zn_,i*Pwidth/m,i);
6.      for(int p=0;p<Pheight;p++)
7.          for(int q=0;q<Pwidth/m;q++){
8.              n[q+i*Pwidth/m][p]=n_[q][p];
9.              zn[q+i*Pwidth/m][p]=zn_[q][p];
10.       }
11. }
```

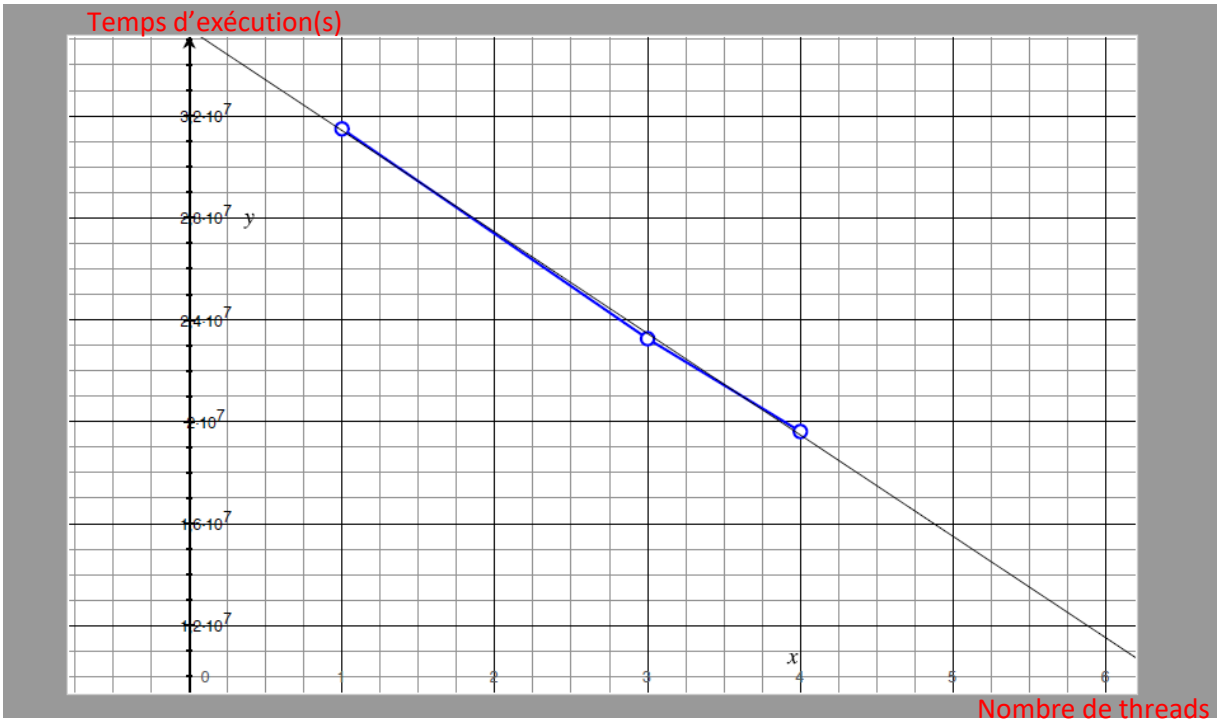
On utilise la fonction process_sub_image comme ci-dessous :

```
1.  std::vector<std::thread> threads;
2.        for(int i=0;i<max_threads;i++){
3.            threads.emplace_back([=](){
4.            process_sub_image(i,max_threads);
5.            });
6.         }
```

Explication sur le paramètre de emplace_back()

Emplace_back() sert pour traiter les données séparés en parallèle. Chaque threads fait la fonction de process_sub_image(i),max_threads). Pour cela on a utilisé une fonction anonyme [=] qui capture tous les occurrences libres par copie et qui ne peuvent être modifiées.

On trace une courbe en faisant varier le nombre de threads et en voyant le temps d'exécution.

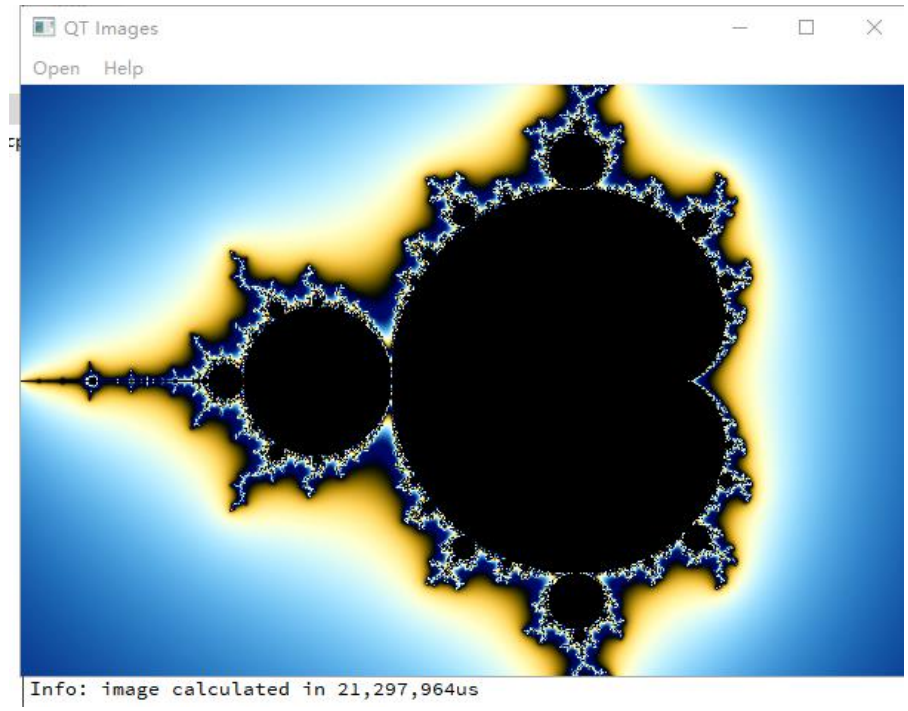


4) Amélioration des performances (2ème méthode) :

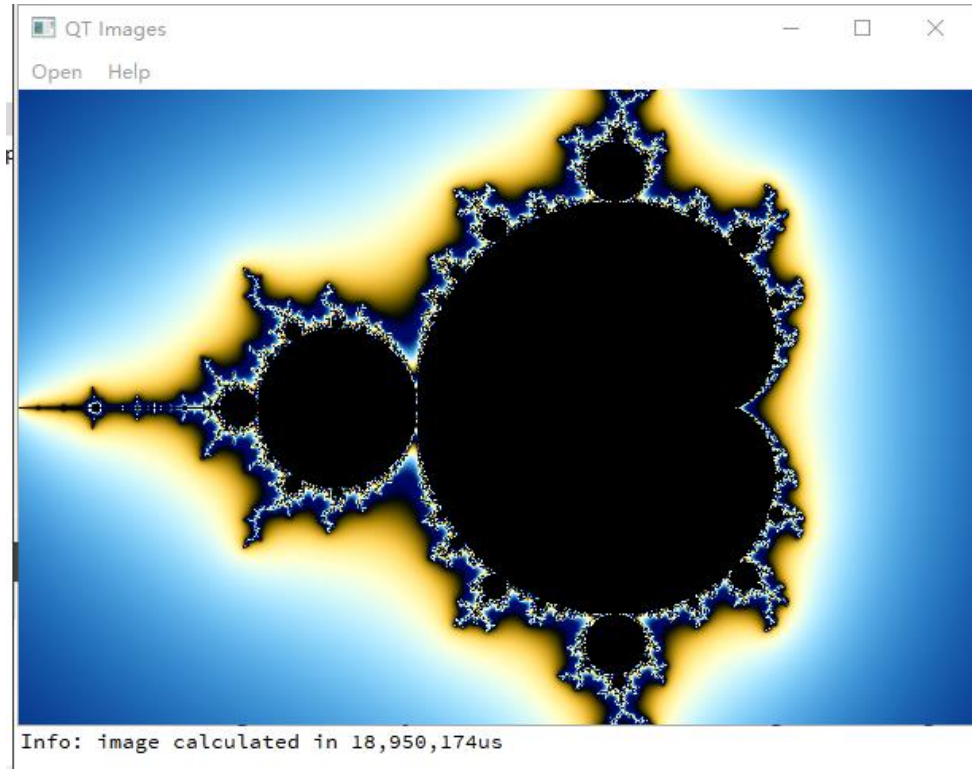
Dans cette partie on veut optimiser en codant explicitement le calcul des parties réelles et imaginaires.

On obtient ces deux images :

a) Image initial



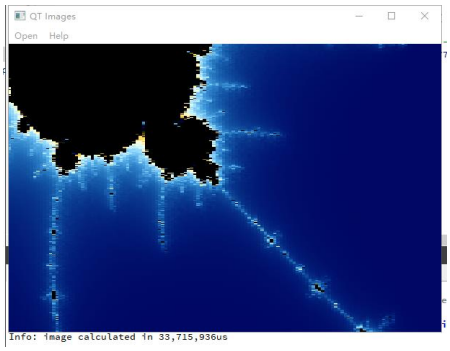
b) Image optimisée :



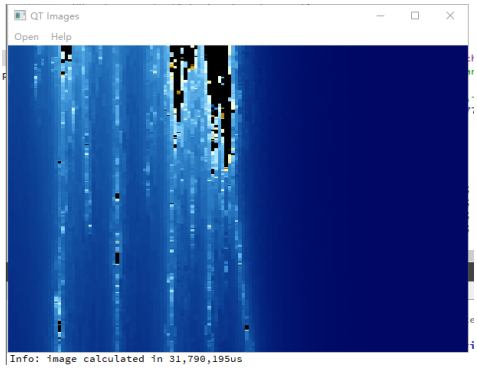
On peut remarquer qu'on obtient deux images identiques. L'amélioration de performance est dans le temps de calcul.

On fait différents facteurs de zoom à notre image.

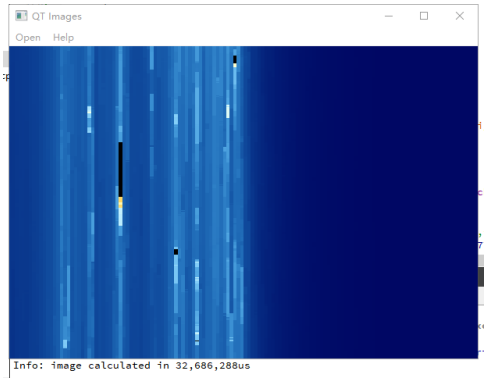
a) Zoom d= 10 e-15



b) Zoom d= 10 e-16



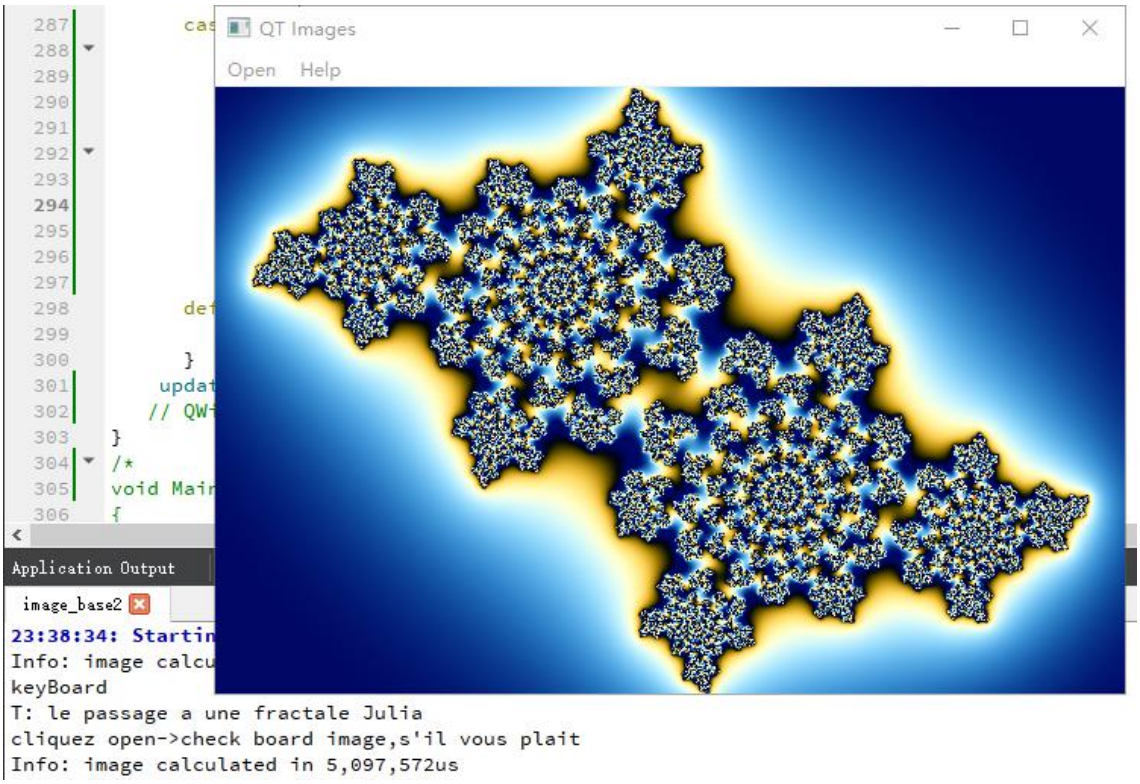
c) Zoom d= 10 e-17



Quand on zoom trop fort on voit que l'image se dégrade parce qu'on demande plus de détaille et on est dans la limite de résolution car on est proche de la région de Mandelbrot.

5) Image Dynamique

Dans cette partie on va implémenter une variation à notre code pour passer d'un fractale de Mandelbrot à une fractale de Julia.



On a ajouté la méthode *KeyPressEvent* dans *mainwindow.cpp,* comme le figure ci-dessous :

```
void MainWindow::keyPressEvent(QKeyEvent *event){

    setFocusPolicy(Qt::StrongFocus);
    installEventFilter(this);
    this->setFocus();
    this->grabKeyboard();
    std::cout<<"keyBoard"<<std::endl;
    switch (event->key()) {
      case Qt::Key_Left:
          xc=xc-0.5;
         std::cout<<"left: xc="<<xc<<std::endl;
         std::cout<<"cliquez open->check board image,s'il vous plait"<<std::endl;
          break;
      case Qt::Key_Right:
          xc=xc+0.5;
          std::cout<<"right: xc="<<xc<<std::endl;
          std::cout<<"cliquez open->check board image,s'il vous plait"<<std::endl;
          break;
```

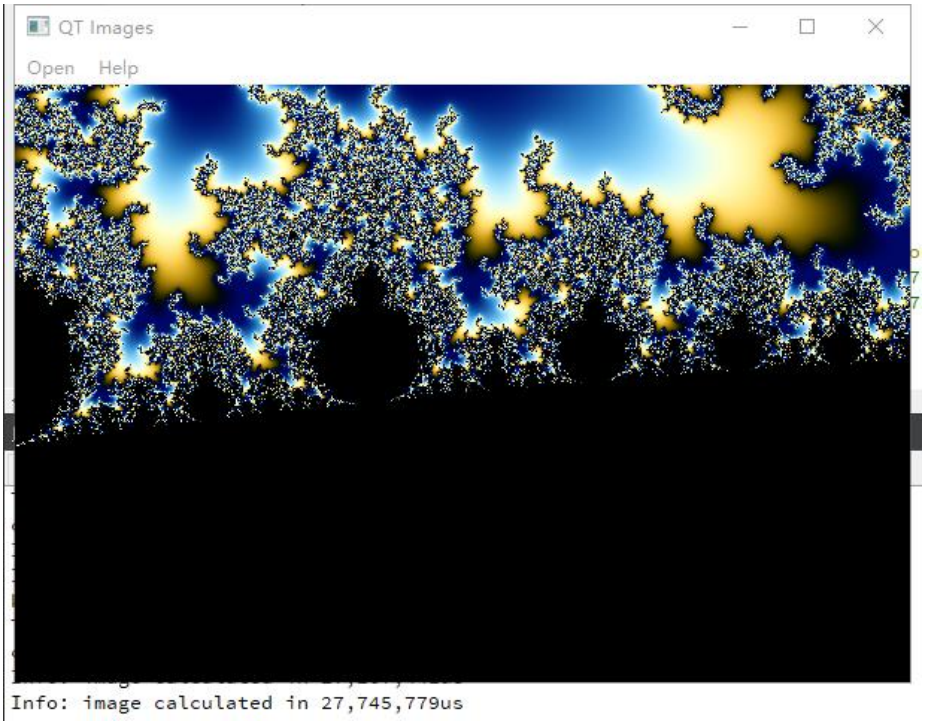
L'image va changer quand on presse *up*, *down*, *left* ou *right* ou *T.*

Pour la configuration :

xc = -1.02390549069711123 yc = 0.249109414132206636 zoom = 0.00737869762948382968

on obtient les images ci-dessous :

Un fractale de Mandelbrot：



Un fractale de Julia：