COMPTE RENDU LAB FRACTALE

CLASSE MANDELBROTIMAGE

```
153 class MandelbrotImage: public FractalImage {
154  public:
155     MandelbrotImage(double xc, double yc, double d, int width, int height):
156         FractalImage(xc, yc, d, width, height) {}
157  protected:
158     void make_fractal_pixel(const double &x0, const double &y0,
159                             const int &x, const int &y) override {
160         compute_pixel(0, 0, x0, y0, x, y);
161     }
162 };
```

La classe **MandelbrotImage** hérite de la classe **FractalImage** ci-dessous. On va implémenter uniquement la méthode virtuelle **make_fractal_pixel** dans la classe **MandelbrotImage**, et tout le reste sera décrit dans la classe **FractalImage**. Les méthodes **resize**, **width** et **height** sont utiles si on implémente une fonction de redimensionnement de la fenêtre. Les autres méthodes sont décrites plus bas.

```
26 class FractalImage: public QImage {
27  protected:
28     const int nmax_ = 512;
29     const int ncolor_ = 65536;
30     const Gradient grad = Gradient(2048);
31
32     double xc_;
33     double  yc_;
34     double d_;
35
36     int w_;
37     int h_;
38
39     void compute_pixel_complex(const double &xn0, const double &yn0,
40                         const double &x0, const double &y0,
41                         const int &x, const int &y) { ...}
62
63     void compute_pixel(const double &xn0, const double &yn0,
64                         const double &x0, const double &y0,
65                         const int &x, const int &y) { ...}
93
94
95     void process_sub_image(int i, int m) { ...}
110
111    virtual void make_fractal_pixel(const double &x0, const double &y0,
112                                    const int &x, const int &y) = 0;
113
114  public:
115     FractalImage(double xc, double yc, double d, int width, int height):
116         QImage(width, height, QImage::Format_RGB32),
117         xc_{xc}, yc_{yc}, d_{d},
118         w_{width}, h_{height} {}
119
120    void make_fractal() { ...}
131    void resize(const int &w, const int &h){ ...}
135    int height() {return h_;}
136    int width() {return w_;}
137 };
```

CONSTRUCTEUR

Le constructeur fait appel à celui de **FractalImage**. Ce dernier initialise ses variables internes. On a le constructeur de la classe mère **QImage**, les variables *xc_*, *yc_*, *d_*, *w_* et *h_*.

```
MandelbrotImage(double xc, double yc, double d, int width, int height):
    FractalImage(xc, yc, d, width, height) {}

FractalImage(double xc, double yc, double d, int width, int height):
    QImage(width, height, QImage::Format_RGB32),
    xc_{xc}, yc_{yc}, d_{d},
    w_{width}, h_{height} {}
```

VARIABLES INTERNES

```
protected:
   const int nmax_ = 512;  // for maximum iteration of loop
   const int ncolor_ = 65536;  // threshold for coloring pixel
   const Gradient grad = Gradient(2048);   // color gradient for coloring pixel

   double xc_;
   double  yc_;
   double d_;

   int w_;
   int h_;
```

Nous avons plusieurs variables qui décrivent notre fractale.

Les constantes *nmax_* et *ncolor_* servent de conditions limites pour le calcul et la coloration de la fractale. La constante *ncolor_* est fixé à 65536 car $65536=256^2$, et on veut vérifier que $|z_n| > 256$ avant de colorier le pixel (car tant que $|z_n| < 256$, on considère qu'on est dans la fractale).

La constante *grad* correspond au tableau de couleurs qui s'étend sur 2048 valeurs.

Les variables *xc_*, *yc_* et *d_* permettent de déplacer la fractale dans la fenêtre et de zoomer.

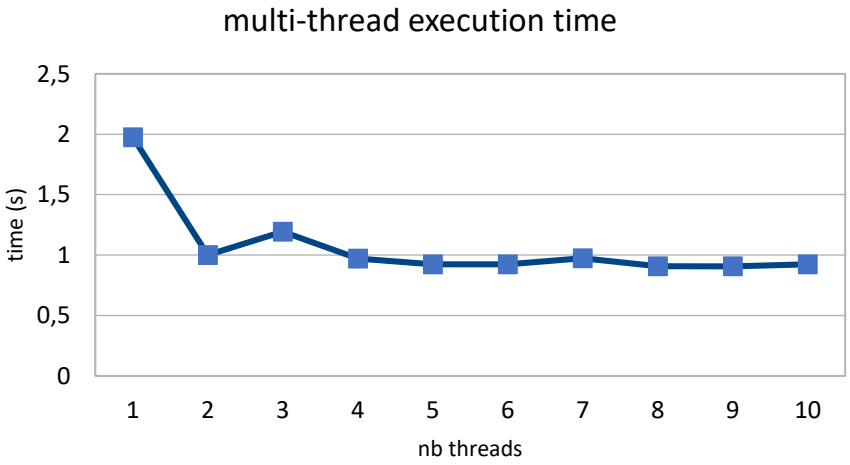
Les variables *w_* et *h_* permettent de connaître la taille de la fenêtre dans laquelle on dessine la fractale.

METHODES

MAKE_FRACTAL

```
// multi-threaded method
// drawing fractal image
void make_fractal() {
    std::vector<std::thread> threads;
    int max_threads = 4;
    for (int i = 0; i < max_threads; i++) {
        threads.emplace_back([=]() {
            process_sub_image(i, max_threads);
        });
    }
    for (auto &thread_elem : threads)
        thread_elem.join();
}
```

Cette méthode utilise le multithreading pour dessiner la fractale. On utilise 4 threads car notre machine tourne avec 4 cœurs. On peut voir ci-dessous la courbe de l'évolution du temps d'exécution en fonction du nombre de cœurs. On peut voir qu'après 4 threads, le temps d'exécution n'est pas fortement réduit.



La méthode **emplace_back** prend en argument une fonction anonyme qui exécute la méthode **process_sub_image**. On doit passer par cette fonction anonyme car **emplace_back** prend en argument une fonction, or en mettant directement le nom d'une fonction, on sous-entend qu'on passe le résultat de cette dernière en paramètre (ici **void**). Il y aura donc incompatibilité de type.

PROCESS_SUB_IMAGE

```
// draw one portion of fractal image
void process_sub_image(int i, int m) {
    Map<double> y_displ_to_plan(0,  h_, yc_+d_, yc_-d_);
    Map<double> x_displ_to_plan(0,  w_, xc_-1.5*d_, xc_+1.5*d_);

    const int ypos_start = i*(h_)/m;
    const int ypos_stop  = (i+1)*(h_)/m;

    for (int y = ypos_start; y < ypos_stop; ++y) {
        auto y0 = y_displ_to_plan(y);
        for (int x = 0; x < w_; ++x) {
            auto x0 = x_displ_to_plan(x);
            make_fractal_pixel(x0, y0, x, y);
        }
    }
}
```

Cette méthode dessine une partie de l'image. On utilise des foncteurs pour mapper les coordonnées du plan *(w,h)* de la fenêtre dans le plan *(x,y)* de la fractale. On voit notamment que le passage sur l'axe *y* est inversé car l'axe *h* de la fenêtre pointe vers le bas et l'axe y pointe vers le haut.

MAP

La formule utilisée pour mapper les données est décrite dans la classe ci-dessous.

Avec cette classe on peut passer d'un ensemble linéaire à un autre en gardant toute proportionnalité.

```
30  // Map a value from one interval to another
31  // Can map all values of vector
32  template <typename T>
33  class Map {
34   private:
35      T min_from_;
36      T max_from_;
37      T min_to_;
38      T max_to_;
39   public:
40      Map() = default;
41      Map(T min1, T max1, T min2, T max2):
42          min_from_{min1}, max_from_{max1},
43          min_to_{min2}, max_to_{max2} {}
44
45      T operator ()(const T &p) {
46          return (p - min_from_)/(max_from_-min_from_)
47                  *(max_to_-min_to_) + min_to_;
48      }
49
50      vector<T> operator () (const vector<T> &p_vect){
51          vector<T> p_maped;
52          p_maped.reserve(p_vect.size());
53          for (auto p : p_vect) {
54              p_maped.push_back((p - min_from_)/(max_from_-min_from_)
55                                *(max_to_-min_to_) + min_to_);
56          }
57          return p_maped;
58      }
59  };
```

COMPUTE_PIXEL_COMPLEX & COMPUTE_PIXEL

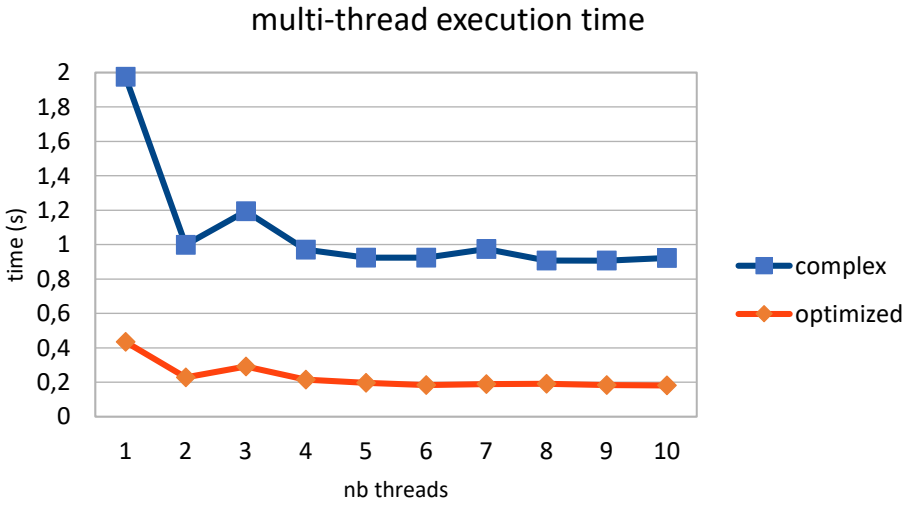
```cpp
// draw (x,y) pixel using complex formula zn = zn*zn + c
// xn0, yn0 are initial values of zn
// x0, y0 are values of c
void compute_pixel_complex(const double &xn0, const double &
                  const double &x0, const double &y0,
                  const int &x, const int &y) {

    std::complex<double> zn(xn0, yn0);
    std::complex<double> z(x0, y0);
    bool inside_fract = true;
    int n;
    for (n = 0; n <nmax_; ++n) {
        zn = zn*zn + z;
        if (!(std::norm(zn) < ncolor_)) {
            inside_fract = false;
            break;
        }
    }
    if (inside_fract) {
        setPixel(x, y, qRgb(0, 0, 0));
    } else {
        double v = log2(log2((std::norm(zn))));
        int i = (1024*sqrt(n+5-v));
        setPixel(x, y, grad[i%2048]);
    }
}
```

```cpp
// draw (x,y) pixel using same formula as compute_pixel_complex
// but without complex objects
void compute_pixel(const double &xn0, const double &yn0,
                   const double &x0, const double &y0,
                   const int &x, const int &y) {
    double xn = xn0;
    double yn = yn0;
    double xn_square = xn*xn;
    double yn_square = yn*yn;
    auto abs_zn = xn_square+yn_square;
    bool inside_fract = true;
    int n;
    for (n = 0; n <nmax_; ++n) {
        yn = 2*xn*yn +y0;
        xn = xn_square - yn_square + x0;
        xn_square = xn * xn;
        yn_square = yn * yn;

        abs_zn = xn_square+yn_square;
        if (!(abs_zn < ncolor_)) {
            inside_fract = false;
            break;
        }
    }
    if (inside_fract) {
        setPixel(x, y, qRgb(0, 0, 0));
    } else {
        double v = log2(log2((abs_zn)));
        int i = (1024*sqrt(n+5-v));
        setPixel(x, y, grad[i%2048]);
    }
}
```

Ces méthodes calculent si le point *(x,y)* est inclus dans la fractale en calculant la convergence de la suite $z_n = z_n^2+c$. Si la suite ne converge pas, on calcule la couleur correspondante du pixel. **compute_pixel_complex** utilise pour cela la méthode *std::norm* qui va calculer la norme de zn au carré, qui est plus rapide que *std::abs*.

Les valeurs *x0, y0, xn0, yn0* permettent de sélectionner les conditions initiales de la suite.

Pour des raisons de rapidité d'exécution, nous réécrivons cette méthode dans **compute_pixel** mais sans utiliser la classe complexe. On va donc devoir calculer indépendamment la partie réelle et la partie imaginaire. On peut voir ci-dessous que le gain en temps de calcul que cette opération permet est de 80% environ.



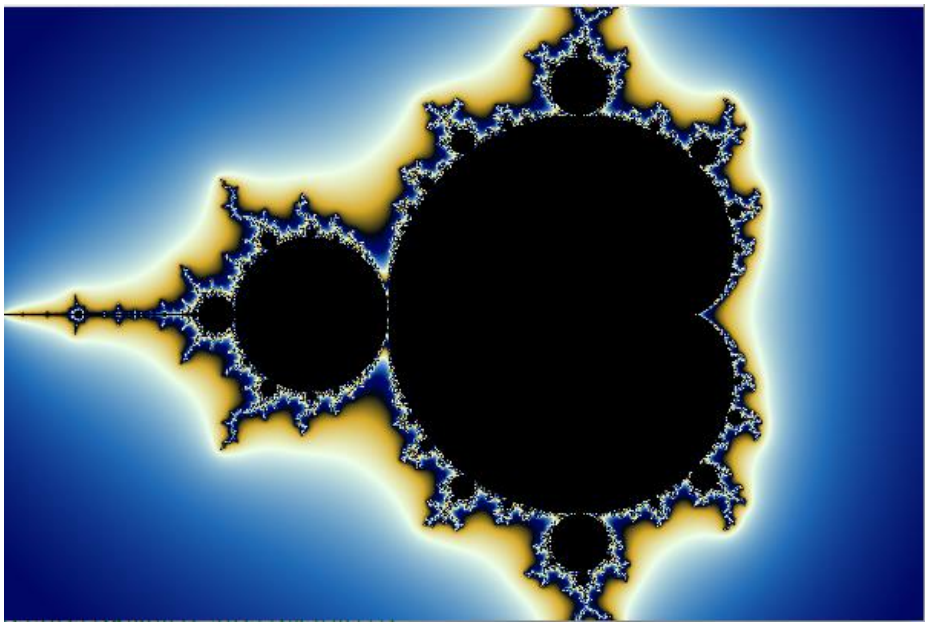
MAKE_FRACTAL_PIXEL

La méthode virtuelle make_fractal_pixel est réécrite pour correspondre à la formule de Mandelbrot.

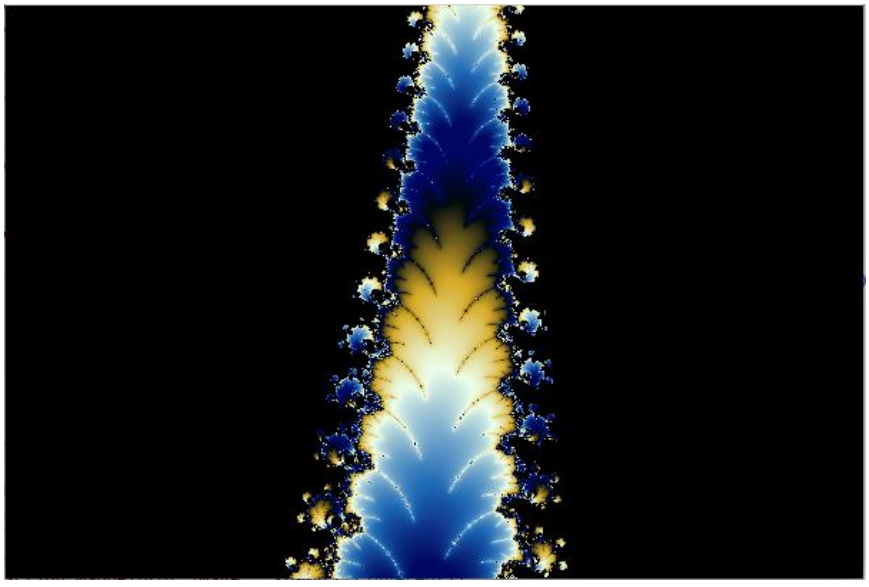
```
void make_fractal_pixel(const double &x0, const double &y0,
                        const int &x, const int &y) override {
    compute_pixel(0, 0, x0, y0, x, y);
}
```

RESULTATS

Pour *d=1.0, xc_=-0.5* et *yc=0*, on obtient l'image suivante.

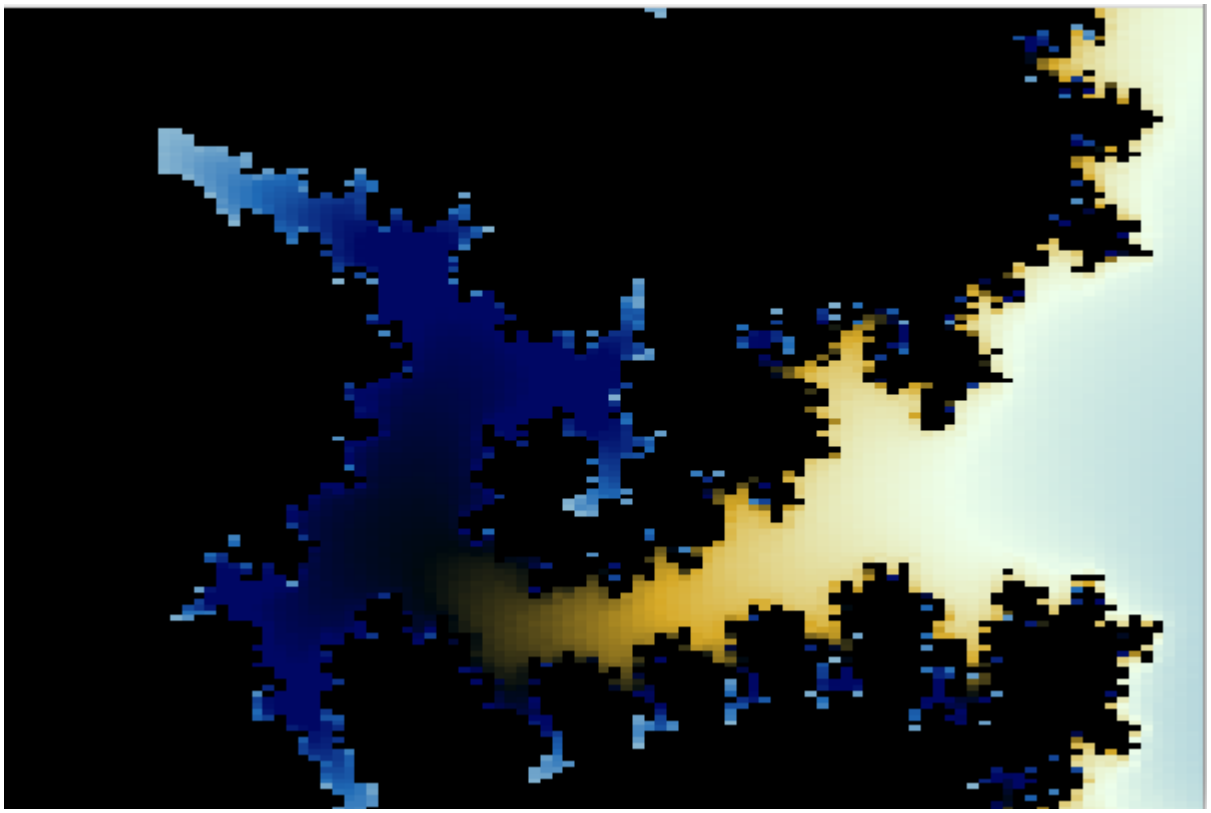


Pour *d = 0.0002, xc_ = -1.7685736562992577, yc_ = -0.00096429685097257052*5 on obtient.



DEGRADATION DE LA QUALITÉ

On peut voir sur cette photo prise à $d = 3.84434e\text{-}15$ que la qualité d'image se détériore pour des petites valeurs de d. Cela est dû au fait que le calcul de la convergence de notre suite se fait avec des valeurs finies. Donc pour des petites variations de x ou y, le résultat du calcul sera identique et la zone sera coloriée de la même couleur.



NAVIGATION

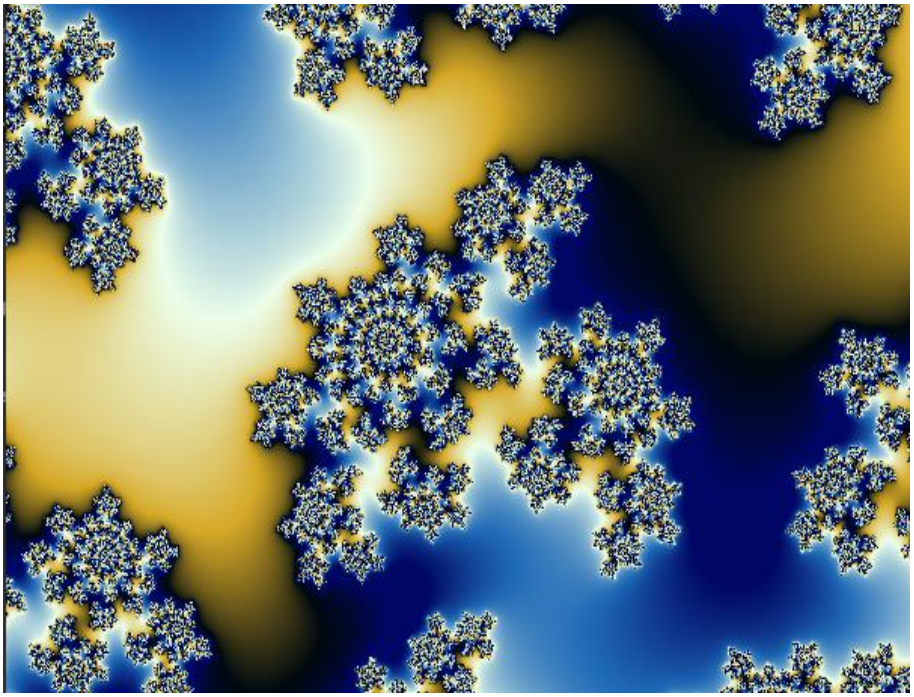
Pour se déplacer dans l'image, il faut modifier les valeurs de *xc_*, *yc_* et *d_*. Pour cela, nous allons *override* la méthode **keyPressEvent** héritée par notre classe **MainWindow** et définir les actions à réaliser en fonction de la touche pressée. Pour passer d'une fractale de Mandelbrot à une fractale de Julia, on utilise un pointeur polymorphique de type **FractalImage**.

```
// event for fractal navigation
void MainWindow::keyPressEvent(QKeyEvent *event) {
  if (current_window_fractal_) {
      static bool fract_julia = false;  // choose wich fractal to display
      static std::unique_ptr<FractalImage> current_fract;
      static double xc = -0.5;  // initial values
      static double yc = 0;
      static double d = 1;
      bool pressed = false;

      switch (event->key()) {
      case Qt::Key_T:   // toggle
          fract_julia = !fract_julia;
          pressed = true;
          break;
      case Qt::Key_Left:    // go left
              xc -= 0.1*d;
              pressed = true;
          break;
      case Qt::Key_Right:   // go right
              xc += 0.1*d;
              pressed = true;
          break;
      case Qt::Key_Up:  // go up
              yc += 0.1*d;
              pressed = true;
          break;
      case Qt::Key_Down:    // go down
              yc -= 0.1*d;
              pressed = true;
          break;
      case Qt::Key_Plus:    // zoom in
              d = d/10;
              pressed = true;
          break;
      case Qt::Key_Minus:   // zoom out
              d = d*1.5;
              pressed = true;
          break;

      default:
          QMainWindow::keyPressEvent(event);
        }
```

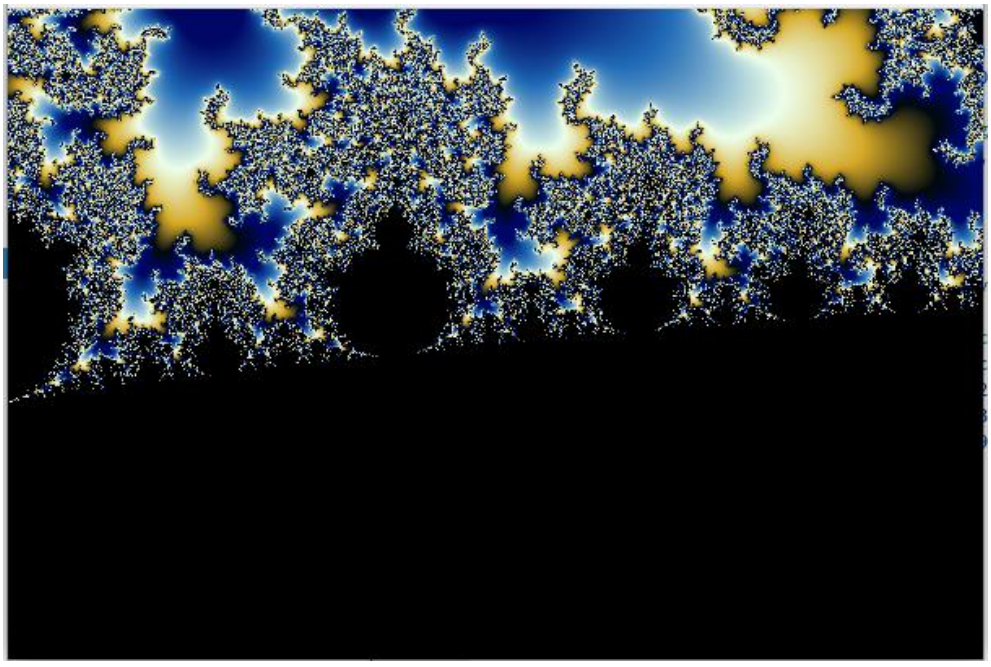
RESULTATS

On regarde les résultats des fractales pour *xc_ = -1.02390549069711123*, *yc_=0.249109414132206636* et *d_= 0.00737869762948382968*.

JULIA



MANDELBROT



JULIA ENTIERE