

# Architecture Interne des Systèmes d'Exploitation (AISE)

Master CHPS

Jean-Baptiste BESNARD  
[jbbesnard@paratools.fr](mailto:jbbesnard@paratools.fr)

# Organisation du Cours

## Cour MATIN, TD après-midi

➡ 10/01: Généralités sur les OS et Utilisation de base

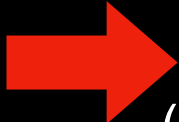
➡ 15/01 : La Chaine de Compilation et l'exécution d'un programme

➡ 16/01 : Les I/Os POSIX et Introduction aux Sockets

➡ 01/02 : Méthodes de communication Inter-Processus

➡ 06/02 : Programmation et reverse et Q/A projets (journée TD)

➡ 14/02 : Mémoire avancée (mmap, madvise, pages, TLB, ...)

 13/02 : TD Débogage (gdb, valgrind) && TD mesure du temps et profilage (perf, kcache, grind) et Q/A projets (journée TD)

➡ Un examen final (25 Mars matin)

# Création d'une Bibliothèque de Dessin

# Objectif de la Journée

## **Nous souhaitons dessiner l'ensemble de Mandelbrot**

- Pour ce faire nous allons créer notre bibliothèque de dessin générant des images au format Portable PixMap (PPM)
- Ensuite nous allons utiliser cette bibliothèque pour générer l'ensemble de Mandelbrot
- Nous terminerons le cours avec un profilage de ce code avec valgrind (callgrind) et Linux Perf

# Le Format PPM

**PPM signifie Portable PixMap c'est un format d'image extrêmement portable.**

- Nous l'avons retenu car il est extrêmement simple.
- Il permet de générer des images en quelques lignes de C

**Notre but est de définir une bibliothèque de dessin en PPM**

[https://fr.wikipedia.org/wiki/Portable\\_pixmap](https://fr.wikipedia.org/wiki/Portable_pixmap)

**Sur la base de ces explications nous souhaitons générer un PPM en format Binaire**

**Spécifications: <http://netpbm.sourceforge.net/doc/ppm.html>**

# Le Format PPM

P6

[LARGEUR] [HAUTEUR]

[VALMAX au plus 65536]

RASTER BINAIRE DE PIXELS

Par simplicité nous avons retenu le retour à la ligne comme séparateur. Le raster de pixel, est tout simplement un tableau de  $LARGEUR * HAUTEUR$  éléments. Le type de l'élément, comme indiqué dans la norme est une structure regroupant les trois composantes sur un octet pour un MAXCOULEUR inférieur ou égal à 256, et deux octets pour une valeur maximale de 65536. Nous allons couvrir le cas où nous encodons sur un octet, très classiquement pour une image. Les trois composantes sont Rouge, Vert et Bleu, notées R, G et B dans le reste de cet article.

# Le Format PPM

On définit alors un pixel comme une structure C:

```
struct ppm_pixel
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
};
```

Nous ajoutons également une fonction inline pour remplir un pixel. Une fonction quand définie dans le header peut être « copiée » dans les différents fichiers, devenant ainsi plus efficace. Il faut lui ajouter le mot-clef « static » pour ne pas créer de conflits de noms, cette fonction étant alors définie dans tous les fichiers où le header a été inclus. Pour le pixel définissons un setter:

```
static inline void ppm_setpixel( struct ppm_pixel * px,
                                unsigned char r ,
                                unsigned char g ,
                                unsigned char b)
{
    px->r = r;
    px->g = g;
    px->b = b;
}
```

# Le Format PPM

Ensuite définissons une image:

```
struct ppm_image
{
    unsigned int width;
    unsigned int height;
    struct ppm_pixel * px;
};
```

Simple également, une largeur (width), une hauteur (height) et des pixels.



# Le Format PPM

Maintenant définissons les fonctions qui seront le point d'entrée de l'utilisateur:

```
//Crée une image de taille w X h
int ppm_image_init( struct ppm_image *im , int w , int h );
//Libère une image
int ppm_image_release( struct ppm_image *im );
//Dessine un pixel de la couleur R,G,B au point X,Y
void ppm_image_setpixel( struct ppm_image * im,
                        int x, int y,
                        unsigned char r,
                        unsigned char g,
                        unsigned char b);
//Sauvegarde l'image dans le fichier au chemin 'path'
int ppm_image_dump( struct ppm_image *im, char * path );
```

# PPM\_Image\_init

```
//Crée une image de taille w X h  
int ppm_image_init( struct ppm_image *im , int w , int h );
```

Cette fonction doit instancier une image PPM, en particulier il s'agit de procéder à l'allocation mémoire de cette image pour la rendre manipulable par les primitives de des sin. On vide le contenu de la structure, puis on sauve largeur et hauteur dans la structure. Enfin on utilise la fonction calloc pour allouer de la mémoire constituant notre raster de pixel. Calloc étant une manière optimisée d'allouer de la mémoire à « 0 ». (pas de malloc + memset!). Et voilà la base de l'image est prête.

# PPM\_image\_release

```
//Libère une image  
int ppm_image_release( struct ppm_image *im );
```

Ici on veut simplement libérer une image, tout en rendant la mémoire associée au système. On prends le temps de tout mettre à zéro dans la structure. Ensuite on fait « free » de la mémoire. Il est important de mettre le pointeur à NULL. En effet free ne produit pas d'erreur si on lui passe un pointeur NULL. Ainsi vous couvrez le cas où une image est libérée (par erreur) deux fois sans crasher.

# PPM\_image\_dump

```
//Sauvegarde l'image dans le fichier au chemin 'path'  
int ppm_image_dump( struct ppm_image *im, char * path );
```

Cette fonction va se charger d'écrire notre fichier au format PPM. Le processus est assez simple, il suffit de se reporter au standard. Ici nous ouvrons tout d'abord le fichier « path » en écriture. Ensuite, nous écrivons le « magic number » de PPM, le P6. Puis les largeurs et hauteurs. Enfin après avoir donné la valeur maximum de 255 suivie d'un retour chariot, nous écrivons le raster de pixel directement en binaire. Et voilà, l'image est sauvegardée.

# PPM\_image\_setpixel

```
//Dessine un pixel de la couleur R,G,B au point X,Y  
void ppm_image_setpixel( struct ppm_image * im,  
                          int x, int y,  
                          unsigned char r,  
                          unsigned char g,  
                          unsigned char b);
```

Il faut bien sûr une fonction pour dessiner un pixel. Cette fonction étant un accesseur soumis à des contraintes de performances (de nombreux pixels modifiés par cet appel de base), nous le placerons dans le header en static inline. La seule complexité est le calcul de l'indice du pixel (tableau 2D) dans un tableau C issu d'une allocation linéaire. Le principe est de se déplacer selon Y en blocs de la largeur et ensuite d'indicer ce déplacement par la position en X. Ensuite, nous appelons le setter de pixel qui est également static inline.

# Test de la Bibliothèque

```
#include "ppm.h"
```

```
int main(int argc, char *argv[])  
{
```

```
    struct ppm_image im;
```

```
    ppm_image_init( &im, 1024, 1024 );
```

```
    int i,j;
```

```
    for (i = 0; i < 1024; ++i) {  
        for (j = 0; j < 1024; ++j) {  
            ppm_image_setpixel( &im, i, j, i%255, j%255, (i+j)%255);  
        }  
    }
```

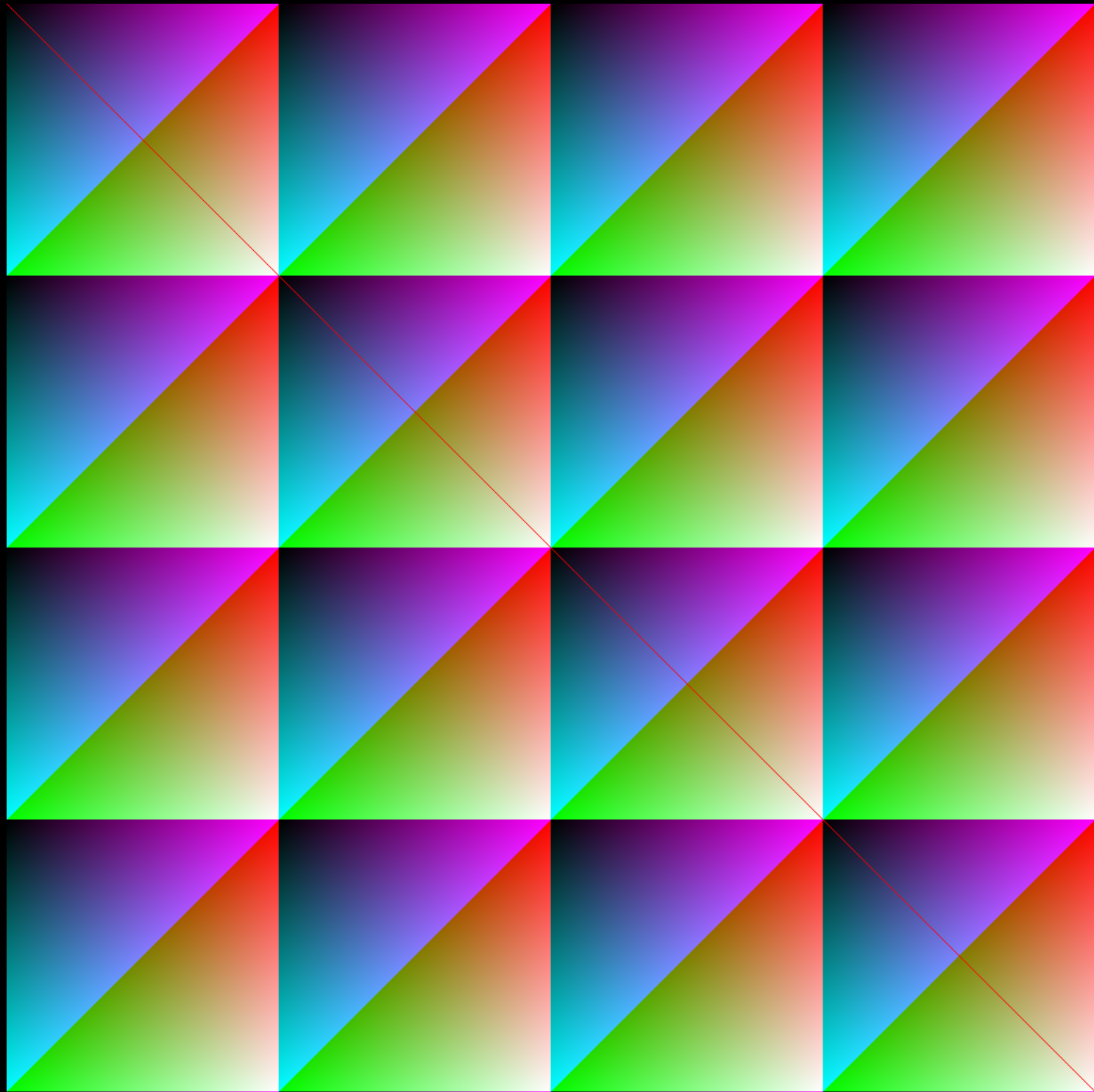
```
    for (i = 0; i < 1024; ++i) {  
        ppm_image_setpixel( &im, i, i, 255, 0, 0 );  
    }
```

```
    ppm_image_dump( &im , "test.ppm");  
    ppm_image_release( &im );
```

```
    return 0;
```

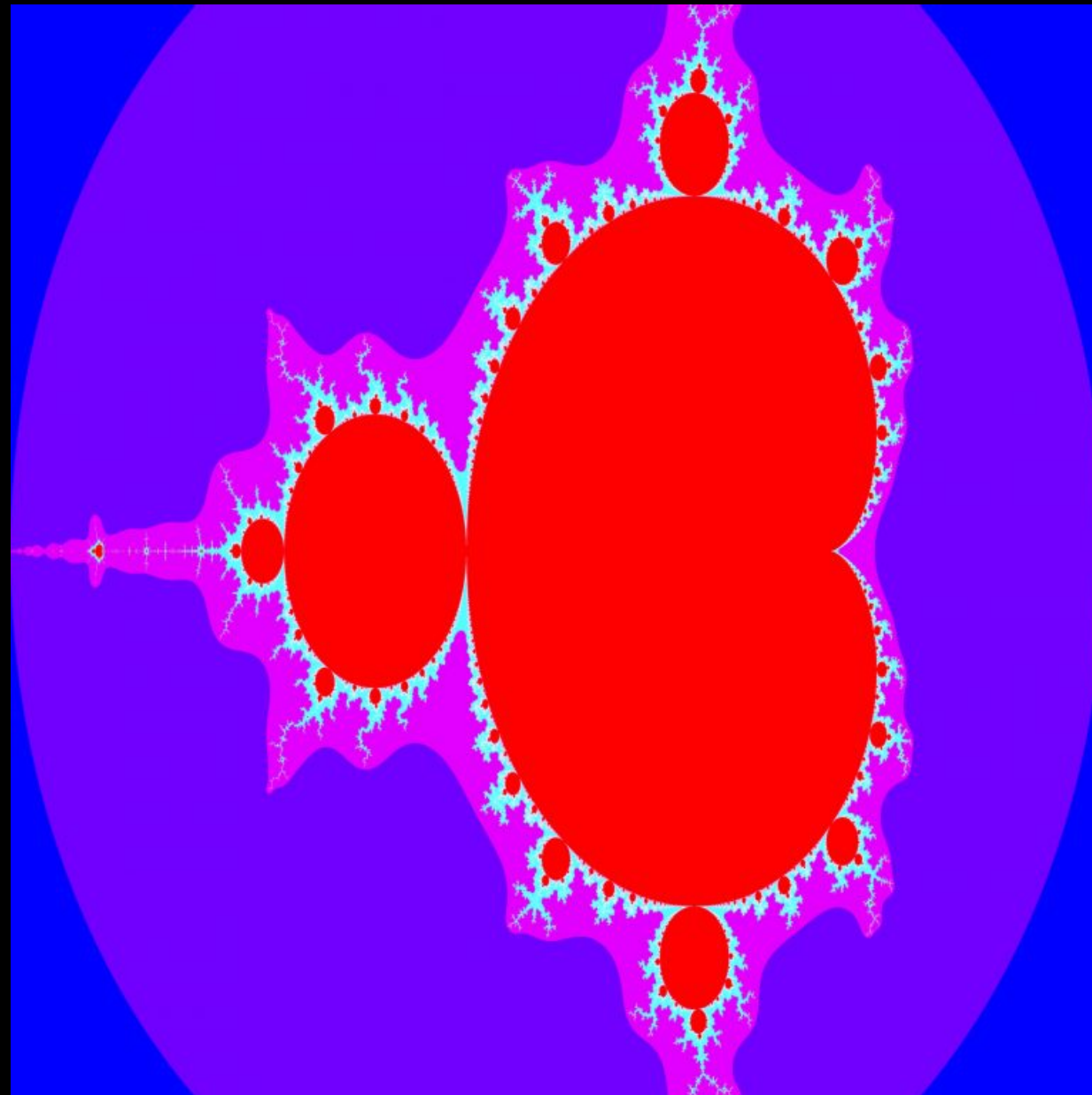
```
}
```

# Test de la Bibliothèque



# L'Ensemble de Mandelbrot

<https://youtu.be/ay8OMOs6AQ>





# L'Ensemble de Mandelbrot

nous allons explorer l'ensemble de Mandelbrot. Le but ici est de se familiariser avec le développement C tout en mettant en place une version initiale d'un calcul relativement intensif. L'exemple retenu ici est le dessin de l'ensemble de Mandelbrot en reposant sur la petite bibliothèque de dessin PPM que nous avons précédemment introduite.

# L'Ensemble de Mandelbrot

Benoit Mandelbrot, de nationalité Française présente la notion de rugosité dans la vidéo ci-dessus. Nous allons ici nous intéresser à la fractale qui porte son nom. Cet objet de curiosité est d'une infinie complexité bien que comme nous allons voir naissant d'une simple formule répétée infiniment. Où comme le dit Mr. Mandelbrot à la fin de la vidéo ci-dessus: « Des merveilles insondables naissent de règles simples.. répétée indéfiniment ».

# Définition de L'ensemble

Ici nous allons tout d'abord définir la méthode pour générer cet ensemble qui est défini dans le plan complexe. On note  $Z$  un nombre complexe et  $^2$  le carré de ce nombre. Enfin on définit  $|Z|$  comme le module d'un nombre complexe.

L'ensemble de Mandelbrot est défini de manière récursive pour tout point du plan complexe  $c$  comme suit:

$$Z(0) = 0$$

$$Z(n+1) = Z(n)^2 + c$$

L'ensemble de Mandelbrot  $M$  est tel que cette suite est bornée. Attachons nous maintenant à définir en termes informatiques cette abstraction mathématique.

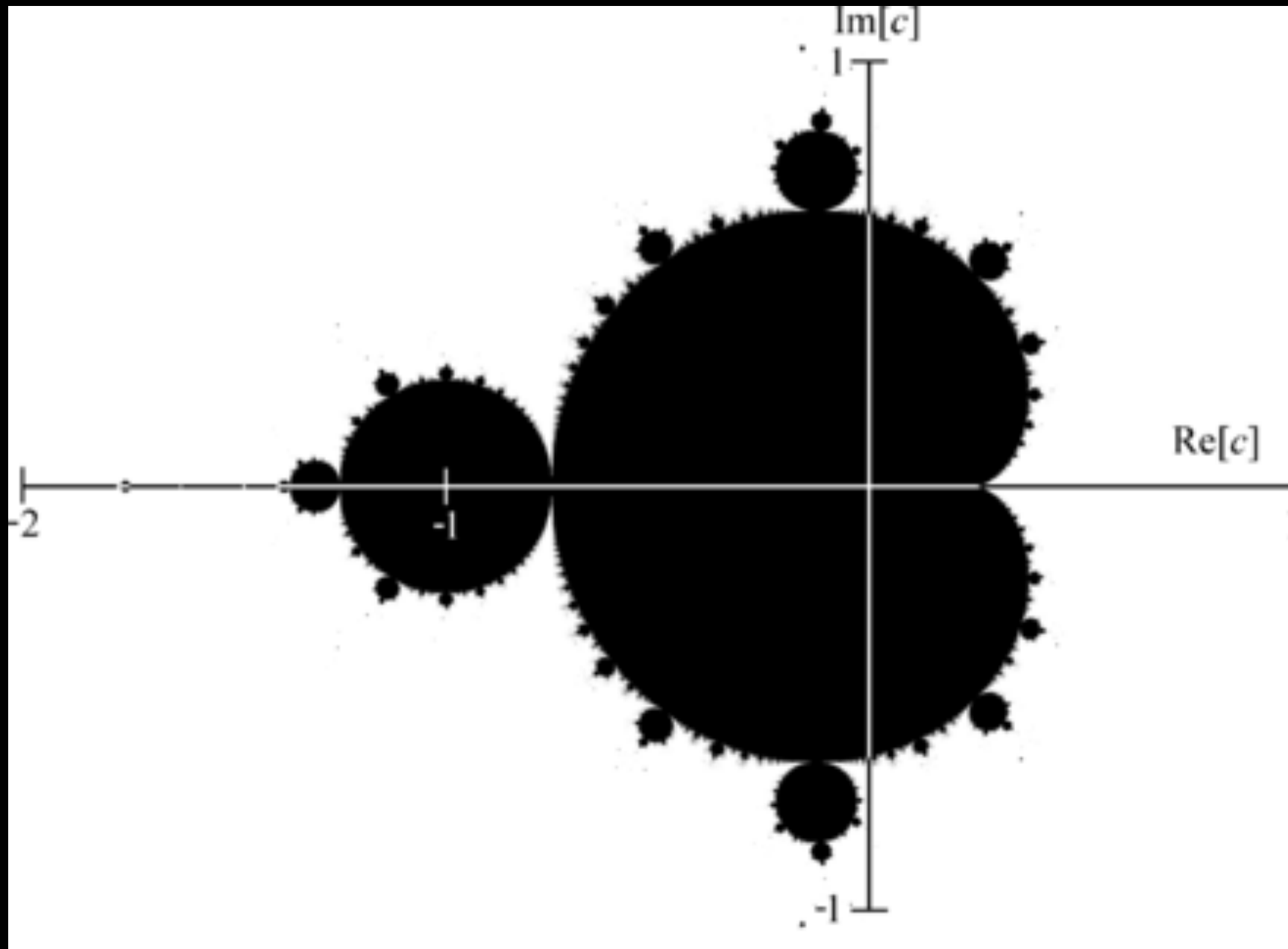
# Complexes en C

**On l'oublie parfois mais le C propose en standard des fonction et un support des nombres complexes. Pour le vérifier vous pouvez consulter le manuel associé:**

**man complex**

# Mandelbrot en C

Nous allons donc utiliser ces nombres directement, évitant ainsi de réinventer des complexes et de nous tromper dans les calculs de modules et de produits. Représentons nous maintenant l'ensemble de Mandelbrot:



# Mandelbrot en C

On voit ici qu'il est présent dans le plan complexe entre -2 et 1 sur l'axe des réels et entre -1 et 1 sur l'axe des imaginaires. Dans une approche informatiques nous allons bien sûr devoir discrétiser cet espace pour y opérer les calculs, la mémoire de nos machines étant limitée. La première tâche est donc d'associer l'espace de définition de M à un tableau 2D de nombre complexes C, indexés de 0 à N.

Pour ce faire, il faut réaliser des fonction donnant pour un point du tableau ses coordonnées complexes telles qu'elle couvrent de manière uniforme l'espace cible. Cela revient à diviser uniformément, largeur et hauteur par le nombre de cases dans chaque dimension pour définir un pas q correspondant à la « largeur » d'une case.

```
#define SIZEX 5000  
#define SIZEY 5000
```

```
double cx( int x )  
{  
    /* -2 ----> 1 */  
    static const double qx = 3.0 / (double)SIZEX;  
    return -2.0 + x * qx;  
}
```

```
double cy( int y )  
{  
    /* -1 ----> 1 */  
    static const double qy = 2.0 / (double)SIZEY;  
    return -1.0 + y * qy;  
}
```

# Boucle de Calcul

Voici la formule qui sera  
à **appliquer à chaque**  
**point du plan complexe**  
**(SIZEX x SIZEY)** définit  
au précédent slide:

```
#define TRSH 2.0  
#define ITER 150ull
```

```
unsigned long int iter = 0;
```

```
double complex c = cx(x) + cy(y) * I;  
double complex z = 0;
```

```
while(iter < ITER)  
{  
    double mod = cabs(z);
```

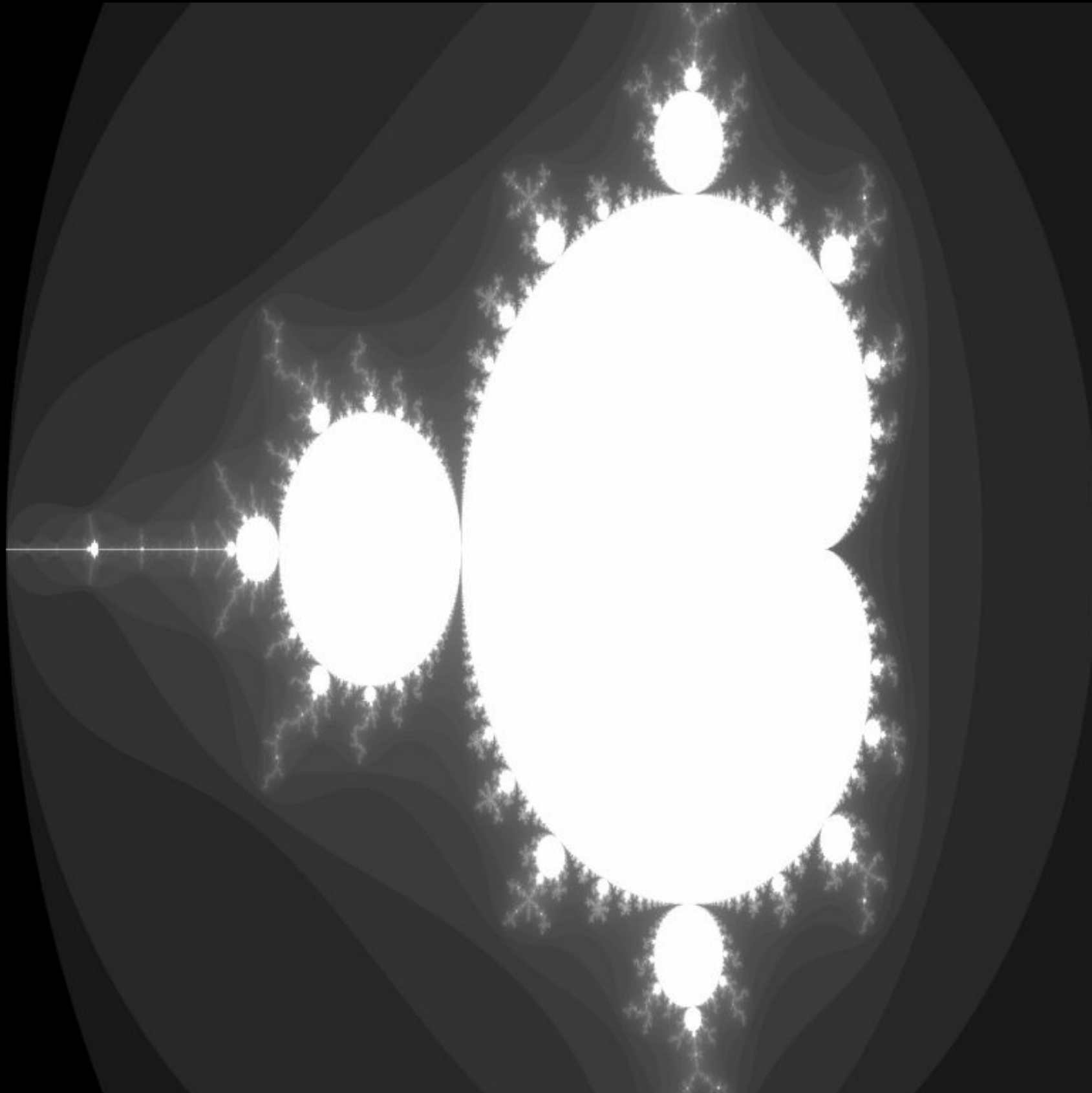
```
    if( TRSH < mod )  
    {  
        break;  
    }
```

```
    z = z*z + c;
```

```
    iter++;  
}
```

```
color[x][y] = iter;
```

# Votre Premier Ensemble de Mandelbrot





# Le code du Mandelbrot Noir & Blanc

Dans ce code finalement très simple:

- on crée une image PPM aux dimensions **SIZEX** et **SIZEY**.
- Ensuite on parcourt le plan complexe et on calcule « **c** » les coordonnées du point d'origine.
- Puis, on applique la récursivité tant que le module est inférieur à 2.0 et que le nombre limite d'itération n'est pas dépassé.
- Enfin, pour chaque point du plan retenu, on définit une couleur en niveau de gris dans l'image PPM.
- Notez que la valeur doit être entre 0 et 255 pour le PPM, je vous recommande de lui appliquer un LOG pour obtenir la sortie ci-dessus.

# Le code du Mandelbrot Noir & Blanc

```
#include <stdio.h>
#include <complex.h>
#include <math.h>
#include "ppm.h"
```

```
#define TRSH 2.0
#define ITER 1024ull
```

```
#define SIZEX 1500
#define SIZEY 1500
```

```
double cx( int x )
{
    /* -2 ---> 1 */
    static const double qx = 3.0 /
(double)SIZEX;
    return -2.0 + x * qx;
}
```

```
double cy( int y )
{
    /* -1 ---> 1 */
    static const double qy = 2.0 /
(double)SIZEY;
    return -1.0 + y * qy;
}
```

```
int main(int argc, char *argv[]) {
    struct ppm_image im;
    ppm_image_init( &im , SIZEX , SIZEY );
```

```
    int i,j;
    double colref = 255.0/log(ITER);
```

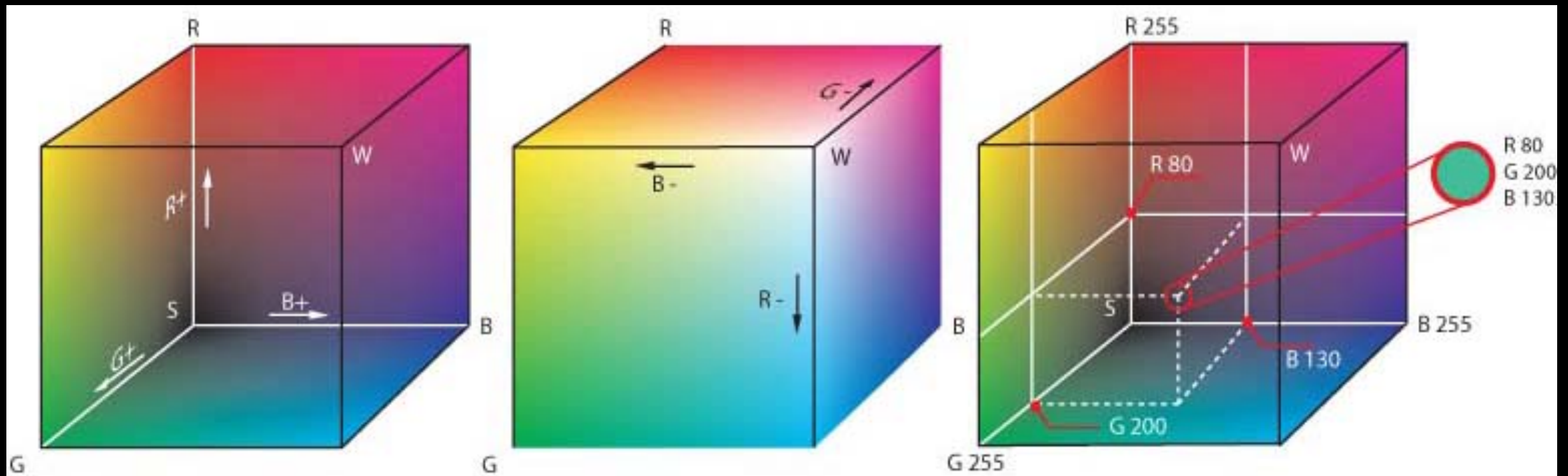
```
    for (i = 0; i < SIZEX; ++i) {
        for (j = 0; j < SIZEY; ++j) {
            unsigned long int iter = 0;
            double complex c = cx(i) + cy(j) * I;
            double complex z = 0;
```

```
                while(iter < ITER){
                    double mod = cabs(z);
                    if( TRSH < mod ){
                        break;
                    }
                    z = z*z + c;
                    iter++;
                }
```

```
                int grey = colref*log(iter);
                ppm_image_setpixel(&im,i,j,
                                grey, grey , grey);
            }
        }
```

```
    ppm_image_dump( &im, "m.ppm");
    ppm_image_release( &im );
    return 0;
}
```

# Définition d'une Echelle de Couleur



Source wikipedia

**Voici le Cube RGB.**

**Pourquoi ne pas mettre un peu de couleur dans votre Mandelbrot ?**

# Echelle de Couleur Possible

```
struct col {  
    int r;  
    int g;  
    int b;  
};
```

```
struct col getcol( int val , int max ) {  
    double q = (double)val/(double)max;  
    struct col c = { 0, 0, 0 };
```

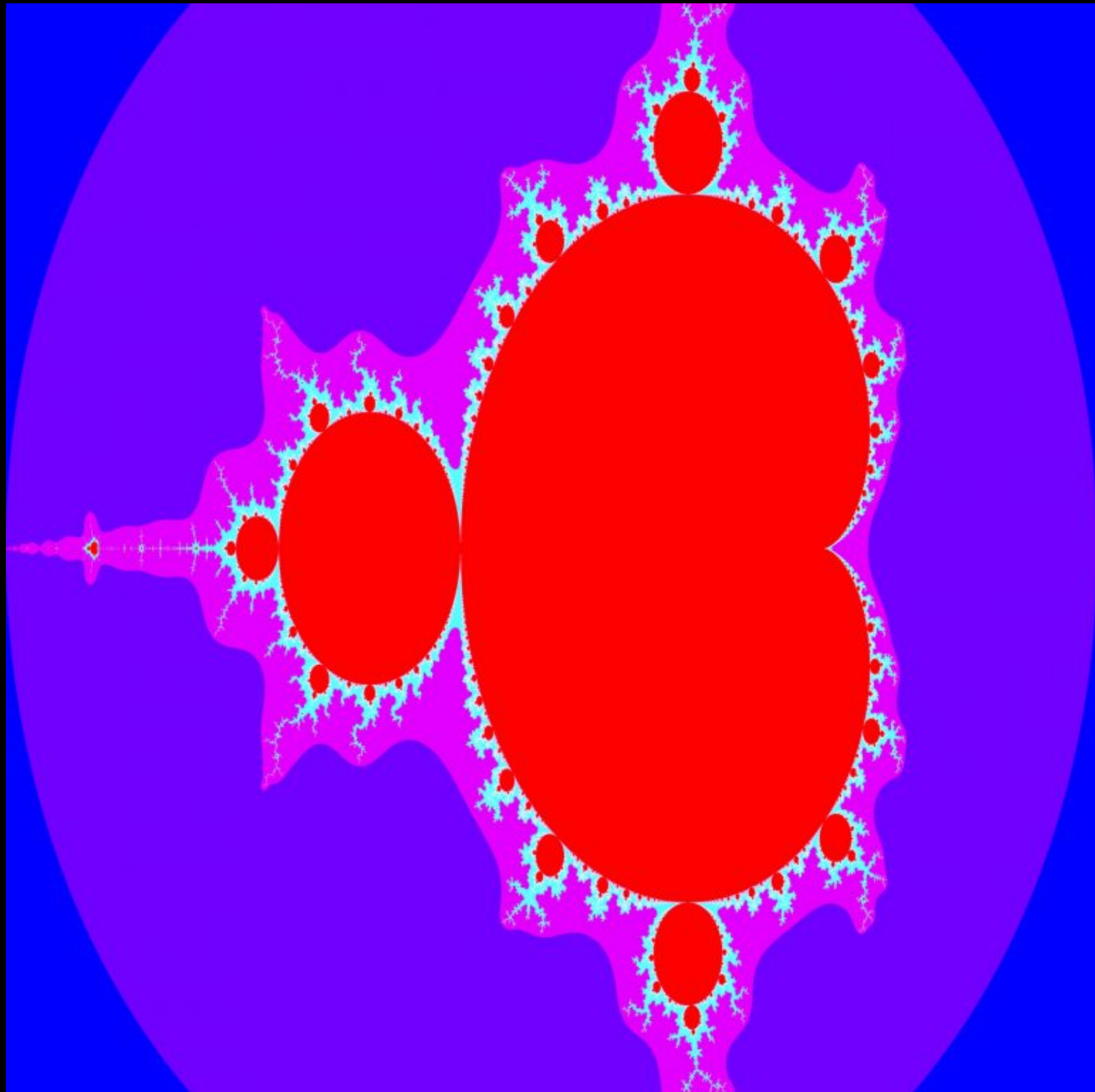
```
    if( q < 0.25 ){  
        c.r = ( q * 4.0 ) * 255.0;  
        c.b = 255;  
    }  
    else if( q < 0.5 ) {  
        c.b = 255;  
        c.g = 255;  
        c.r = (q-0.25)*4.0*255.0;  
    }  
    else if( q < 0.75 ){  
        c.b = 255;  
        c.r = 255;  
        c.g = 255.0 - (q-0.5)*4.0*255.0;  
    }  
    else{  
        c.b = 255-(q-0.75)*4.0*255.0;  
        c.g = 0;  
        c.r = 255;  
    }  
    return c;  
}
```

# Echelle de Couleur Possible

Et on remplace le calcul de couleur:

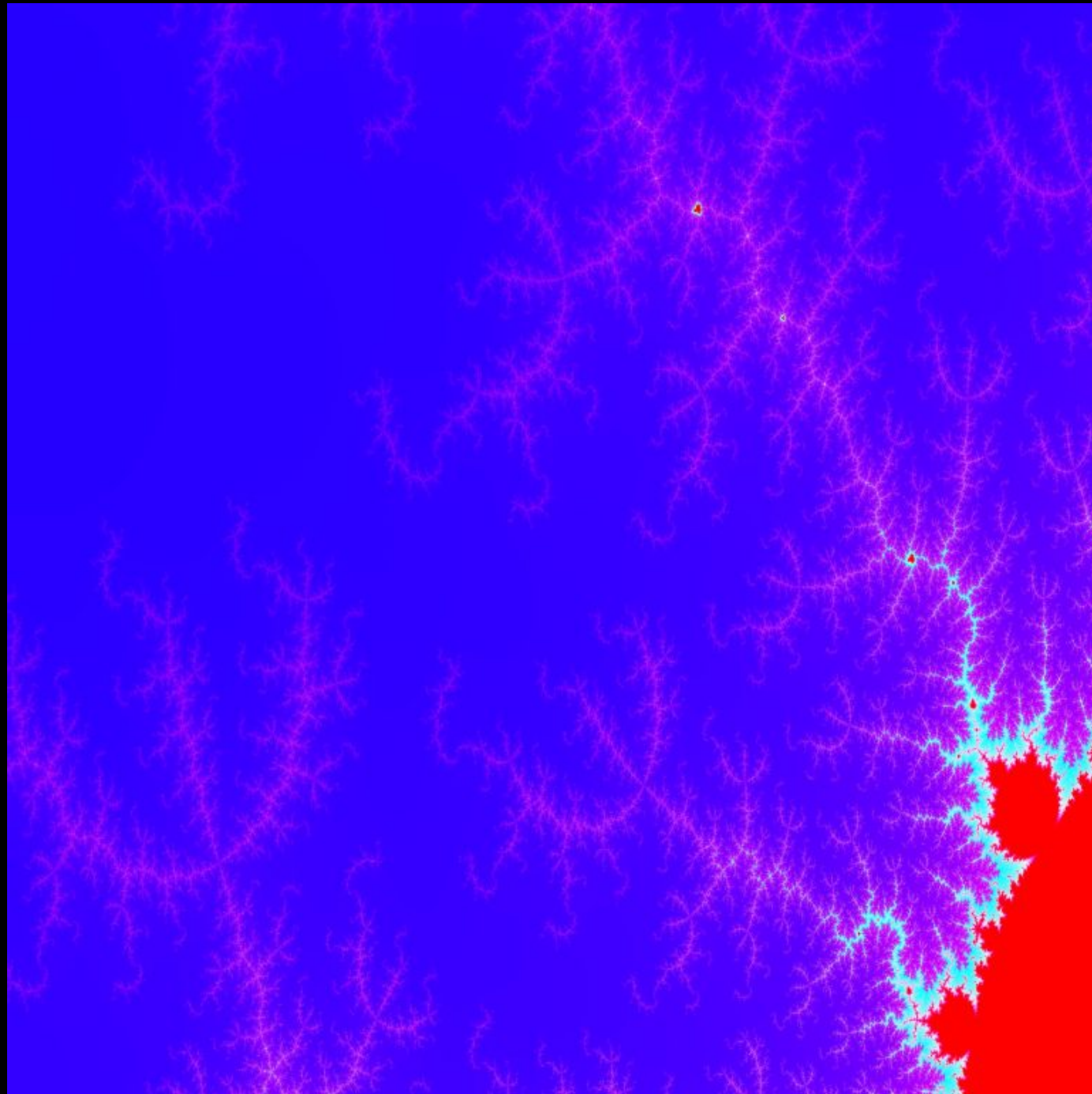
```
double colref = log(ITER);  
//...  
struct col cc = getcol( log(iter), colref );  
ppm_image_setpixel(&im, i,j, cc.r, cc.g , cc.b );
```

# Mandelbrot en Couleur

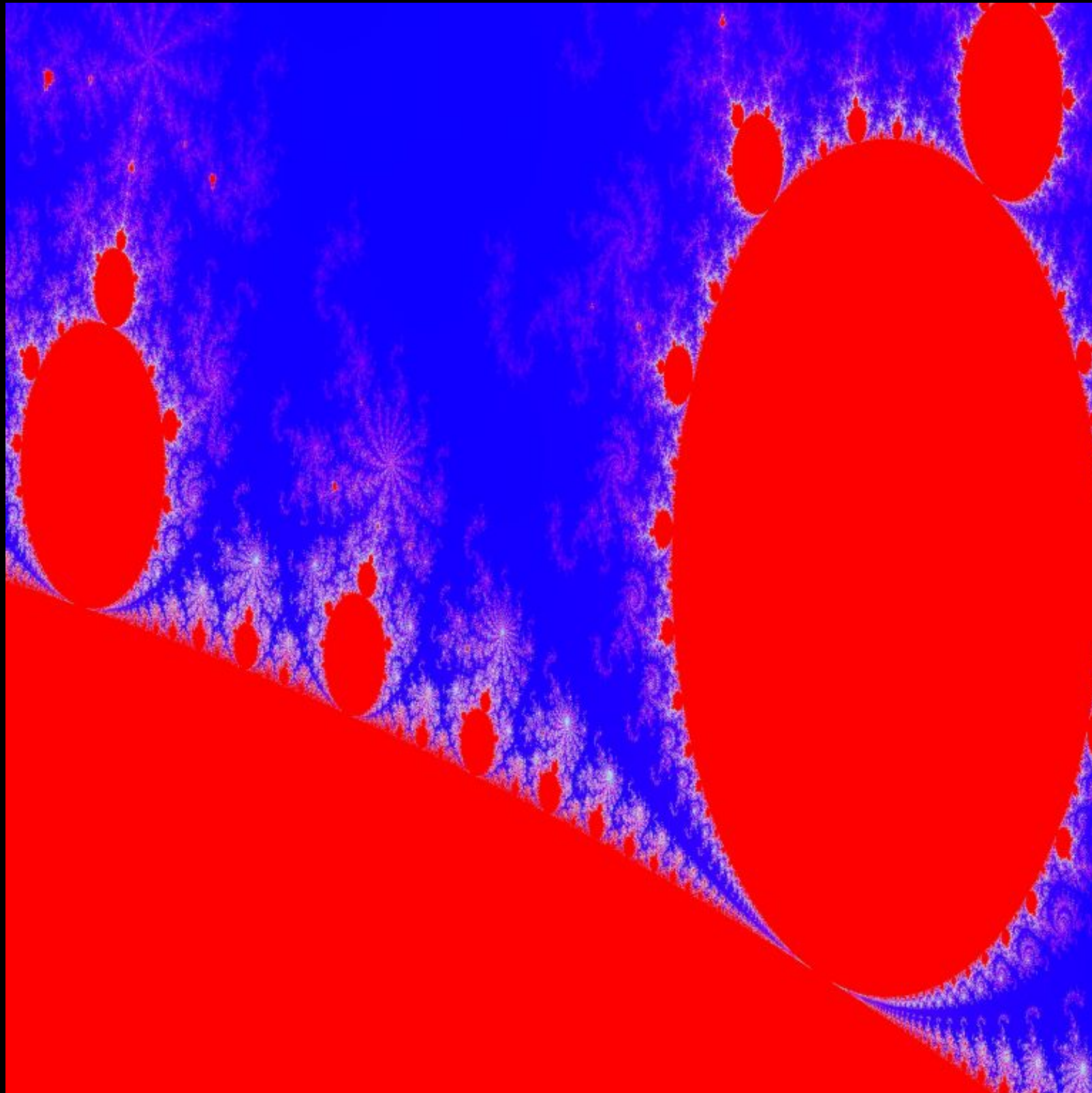




# Faites des Zooms !



# Faites des Zooms !





# Mandelbrot

**Pour aller plus loin, parallélisez le code en OpenMP**

# Profilage avec Callgrind

Nous allons utiliser deux outils, valgrind qui fournit Callgrind et Kcachegrind qui est une interface graphique permettant de visualiser les fichiers de sortie générés par Callgrind.

**Pour les installer sur Centos / RedHat:**

```
yum install kdesdk-kcachegrind valgrind graphviz
```

**Pour les installer sur Debian / Ubuntu:**

```
apt-get install kcachegrind valgrind graphviz
```

# Profilage avec Callgrind

Le processus d'instrumentation de callgrind est très simple. Il suffit de rajouter l'invocation de Callgrind avant le programme cible.

Par exemple:

```
valgrind --tool=callgrind ./test
```

# Profilage avec Callgrind

On peut afficher des sorties en console:

```
callgrind_annotate ./callgrind.out.XXXXXX
```

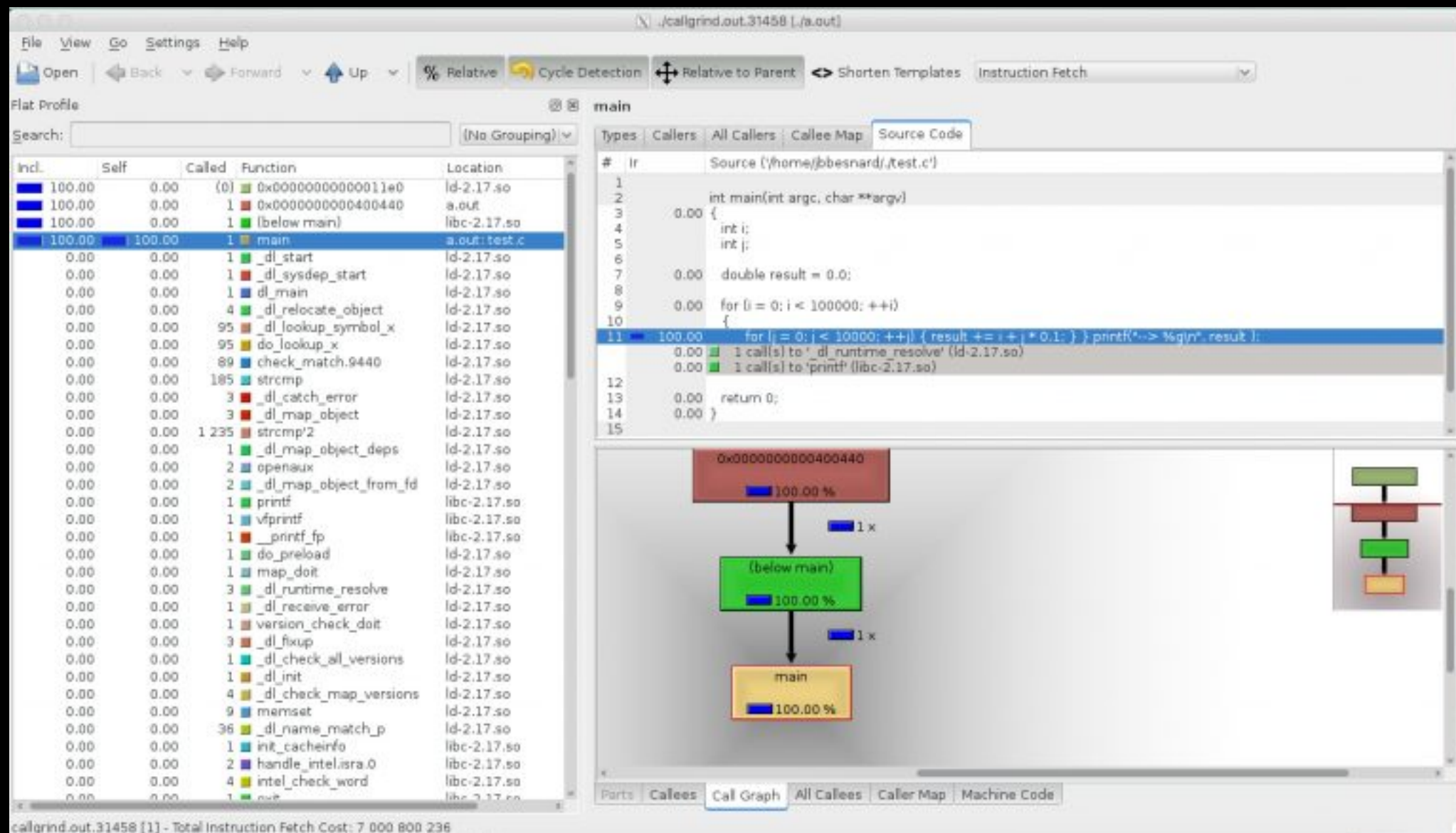
```
callgrind_annotate --tree=both ./callgrind.out.XXXXXX
```

```
callgrind_annotate --auto=yes ./callgrind.out.XXXXXX
```

# Visualiser dans Kcachegrind

Il existe une interface graphique:

**kcachegrind ./callgrind.out.XXXXX**



# Mandelbrot

**Optimisez votre Mandelbrot !**