Mandelbrot Generator

Michael Bowers

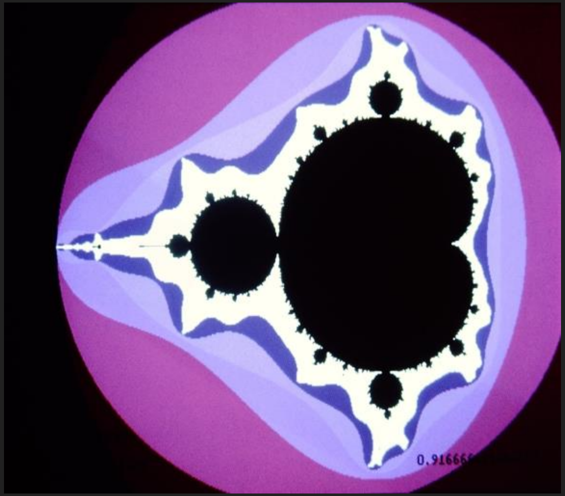
Background

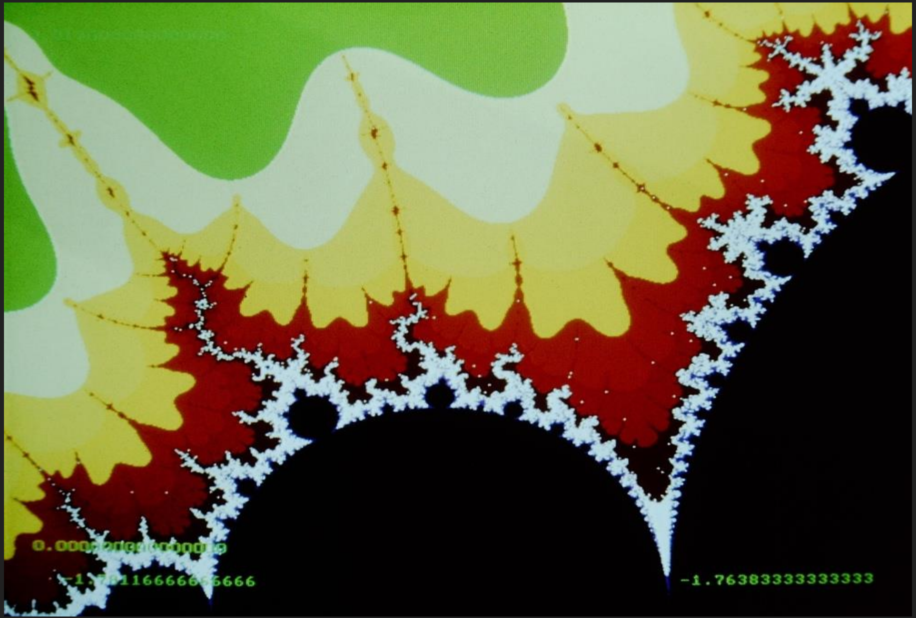
Mandelbrot - fractal produced by performing $Z_n = Z_{n-1}^2 + C$ for n number of iterations

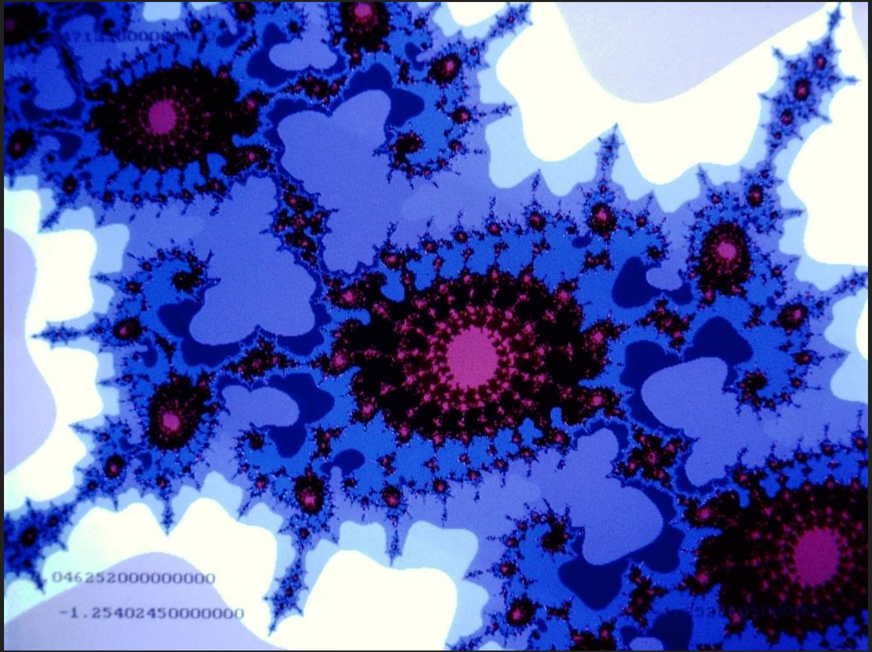
If the absolute value of $Z_n < 2$ for an N number of iterations, the number is considered within the set.

// c = cx + cy

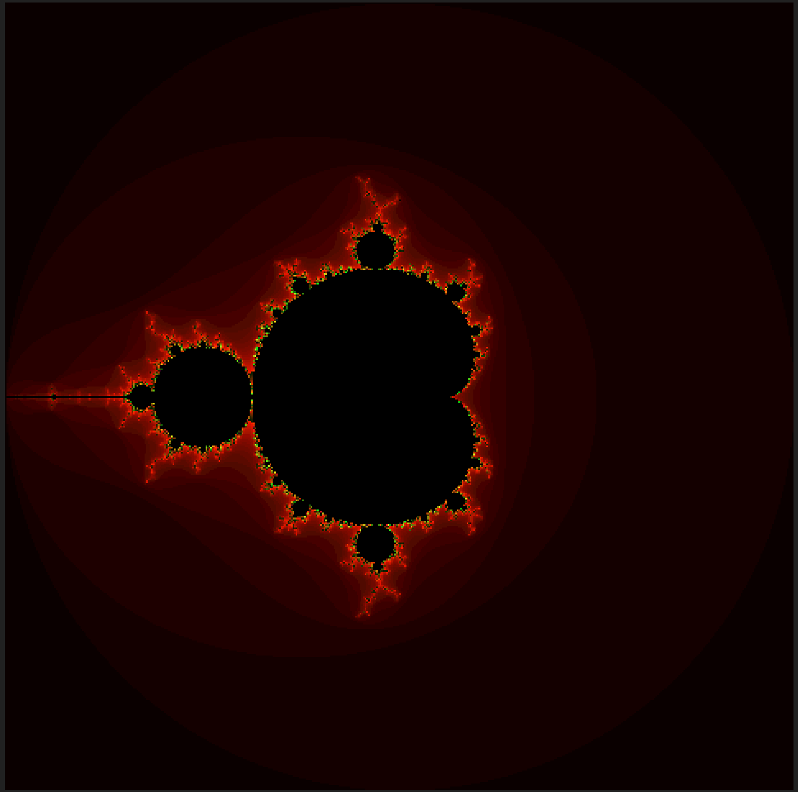
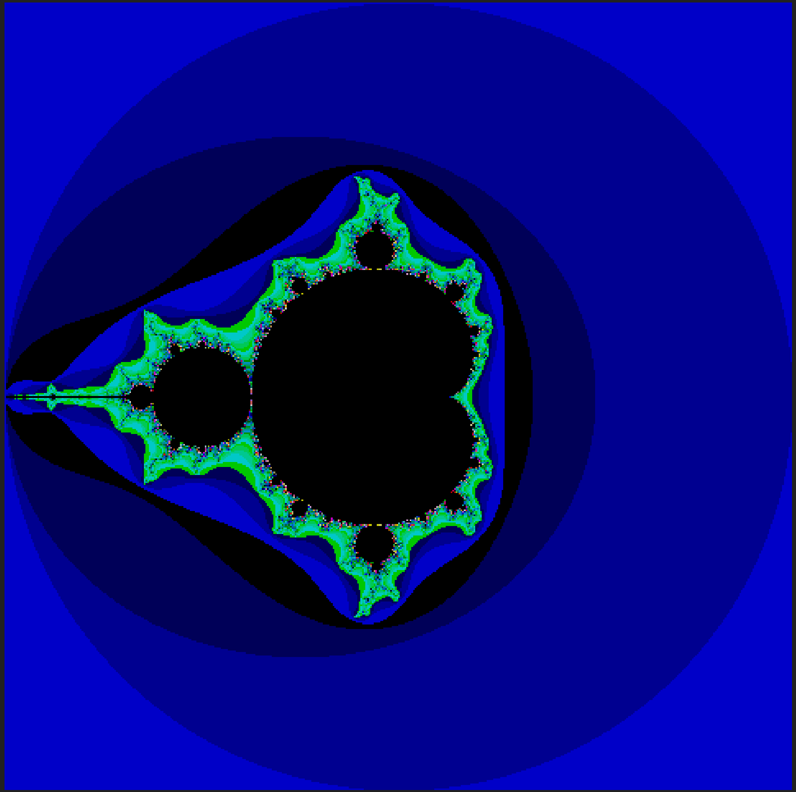
If $(zx^2 + zy^2 > 4)$ and (iter < max)
tempx = $zx^2 - zy^2 + cx$;
zy = 2(zx)(zy) + cy
Zx = tempx
iter++



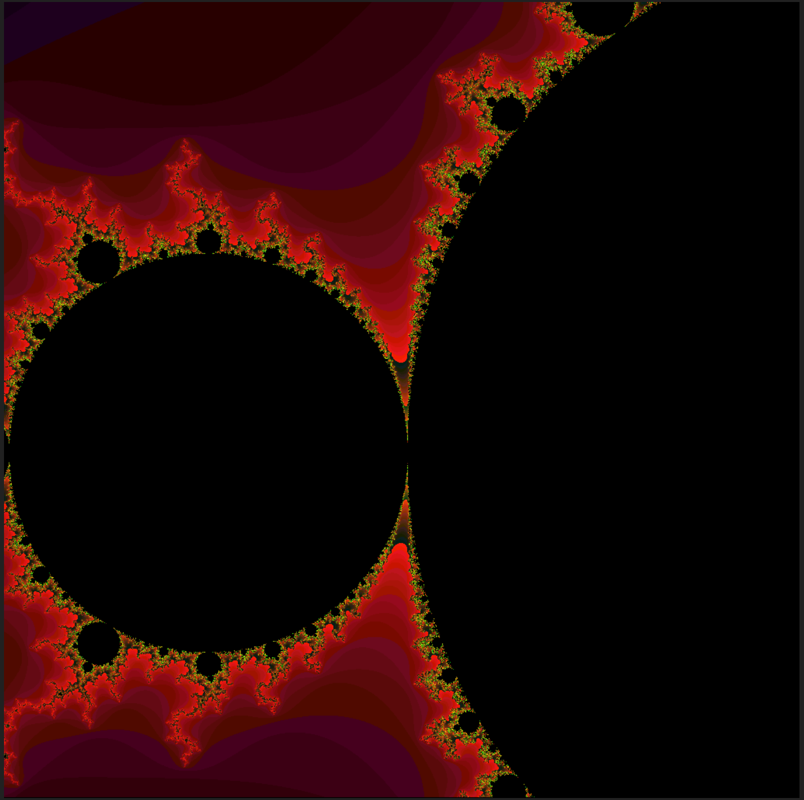


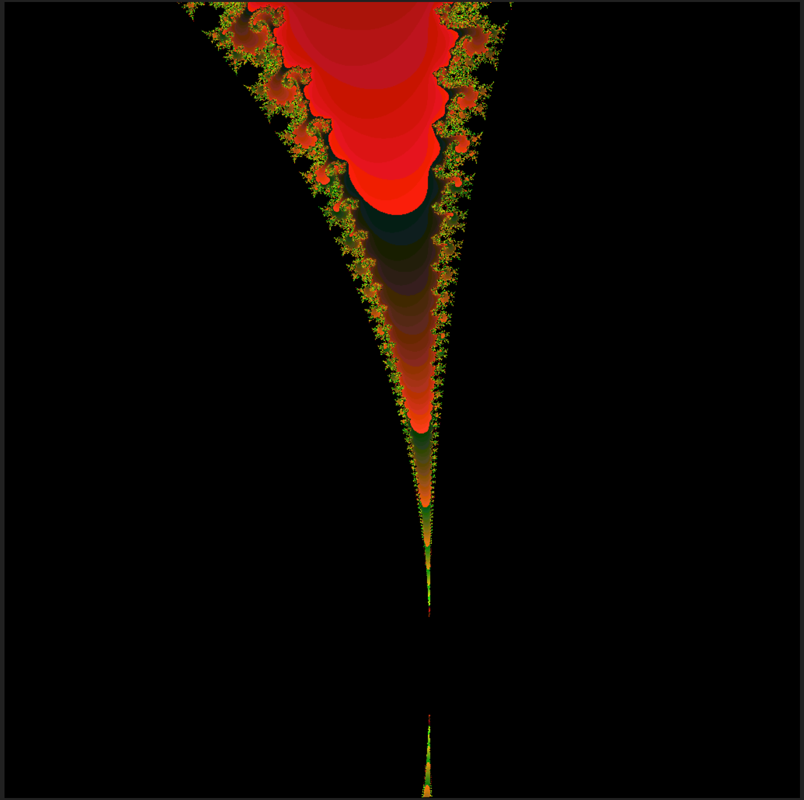


Keith Shuert, 1991

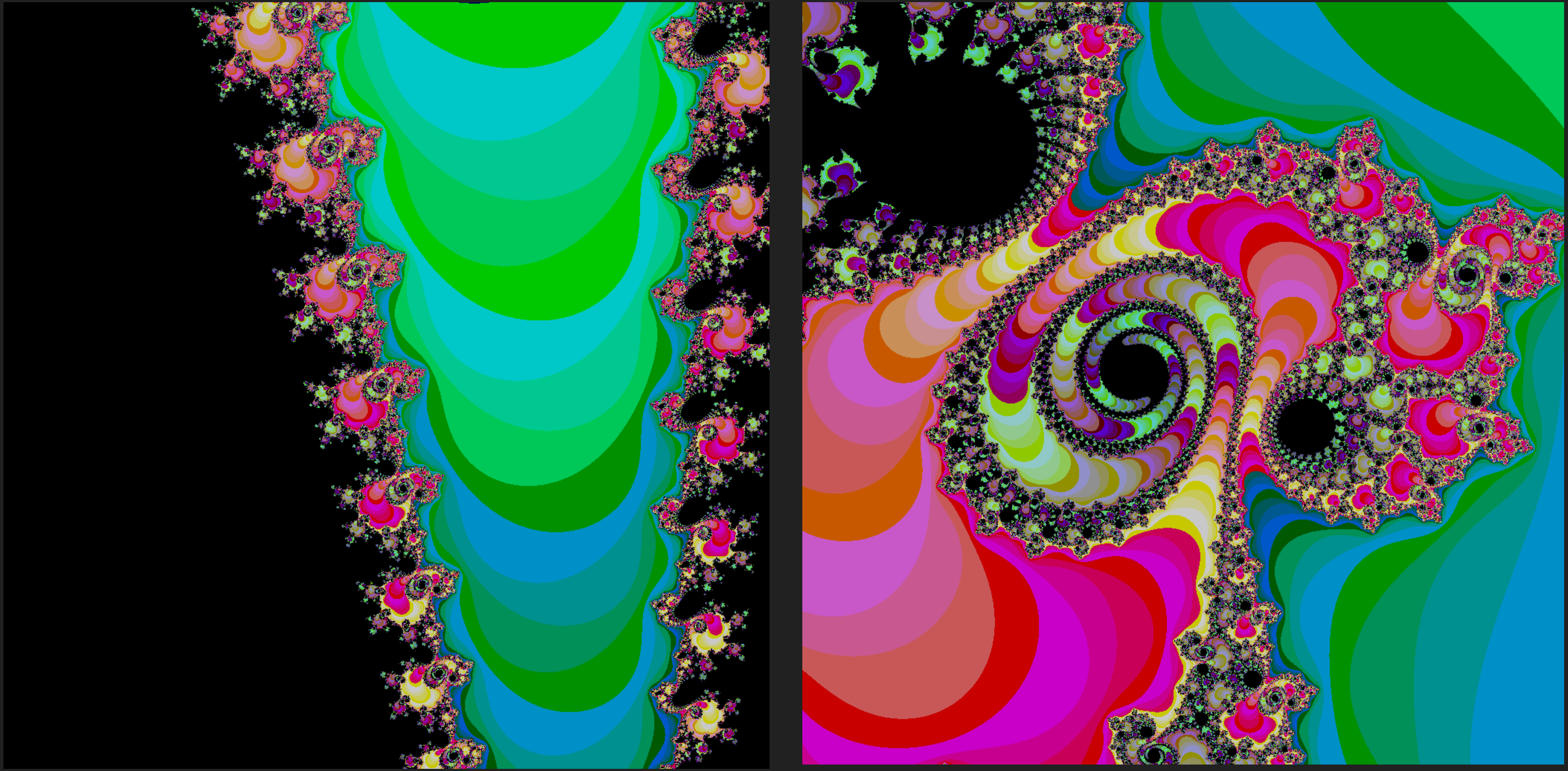
 

X = 0, Y = 0, R = 2

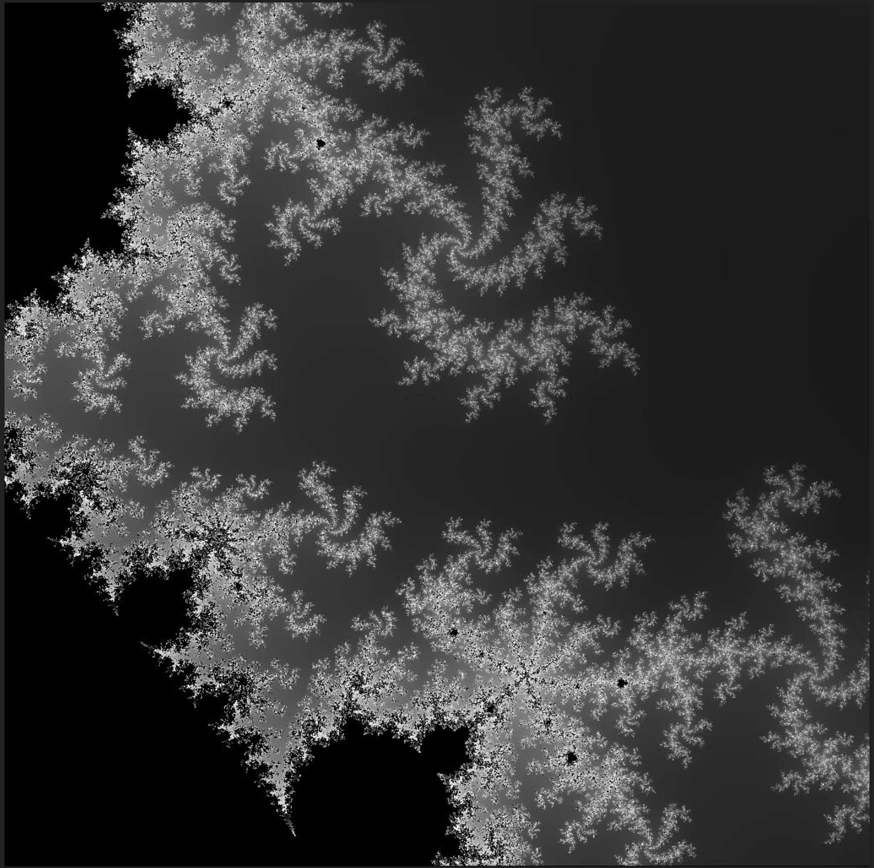
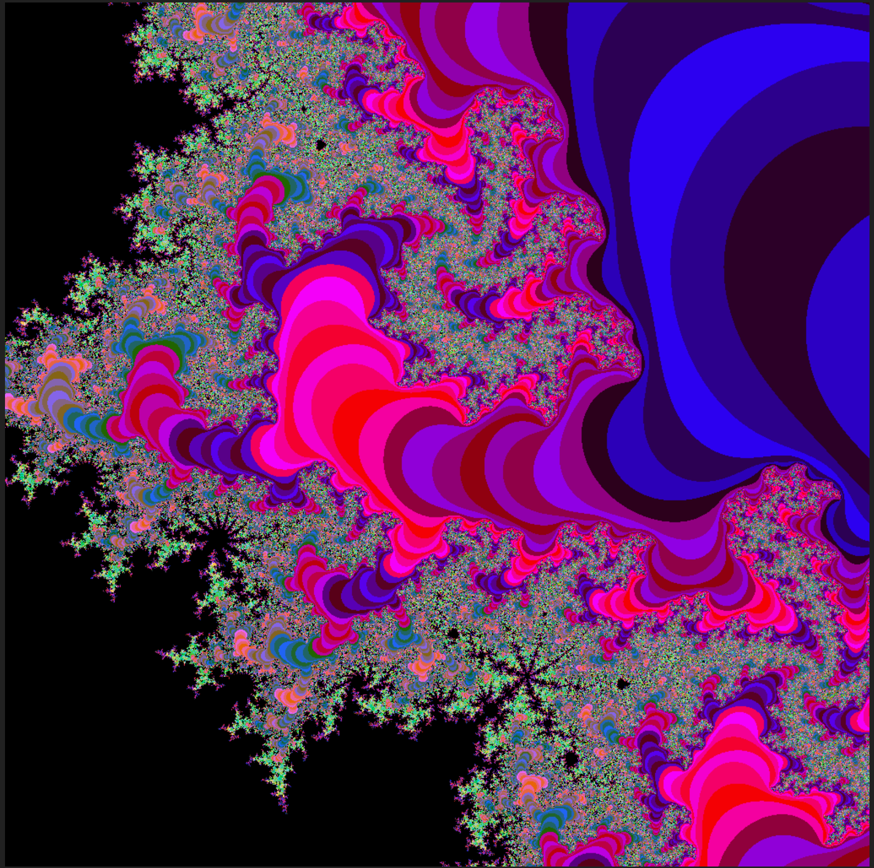




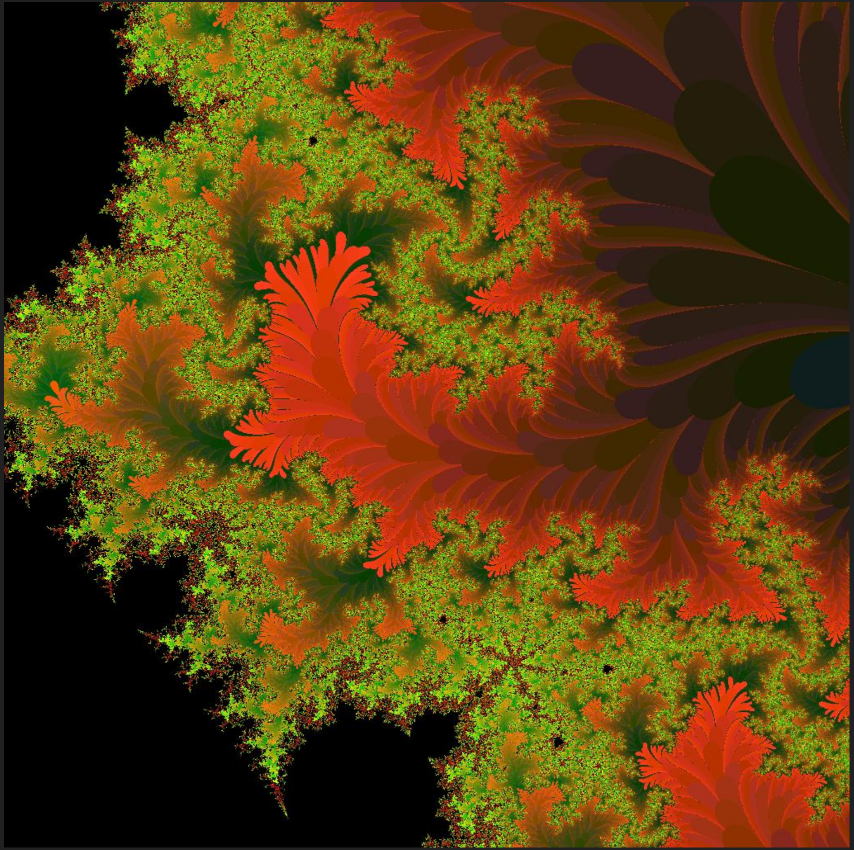
X = -0.066879524349399, Y = -0.7568097768953, R < 2

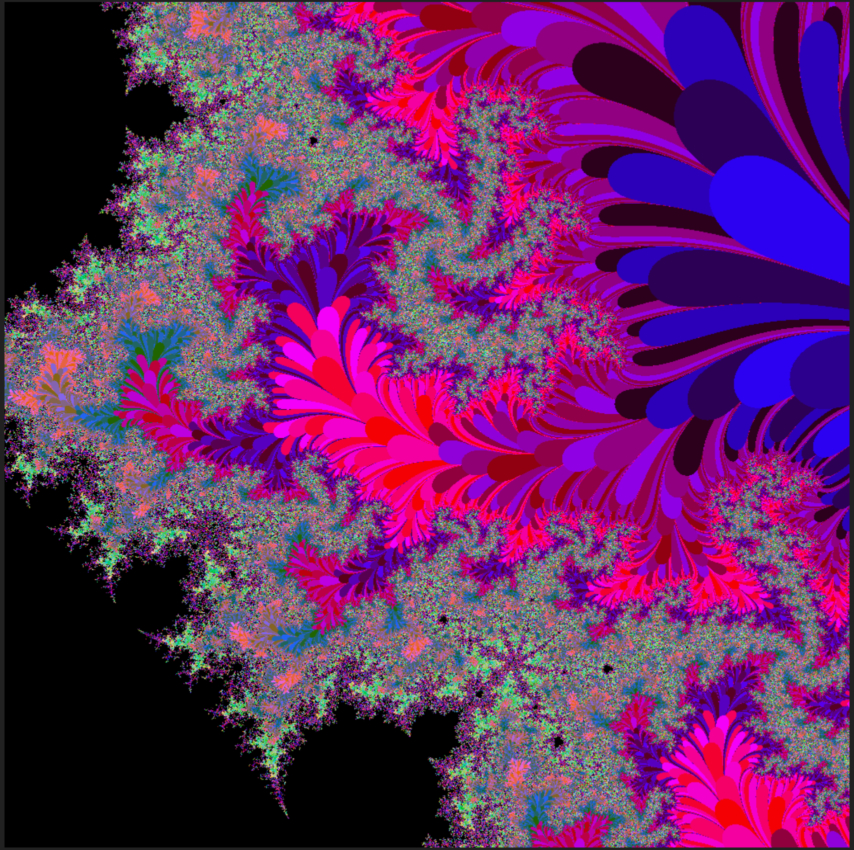


X = -0.066879524349399, Y = -0.7568097768953, R < 2

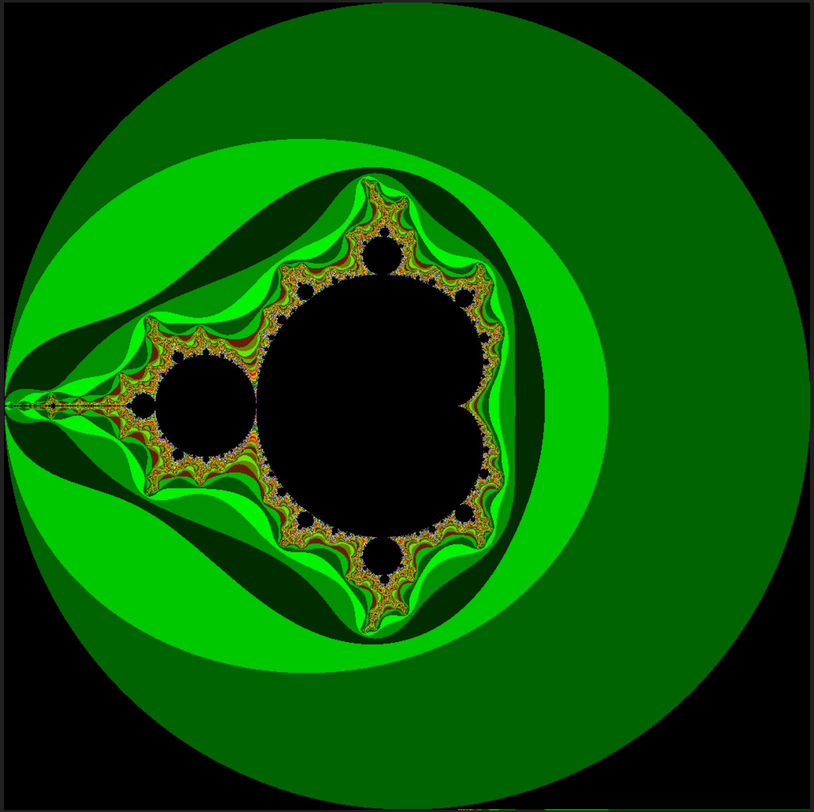



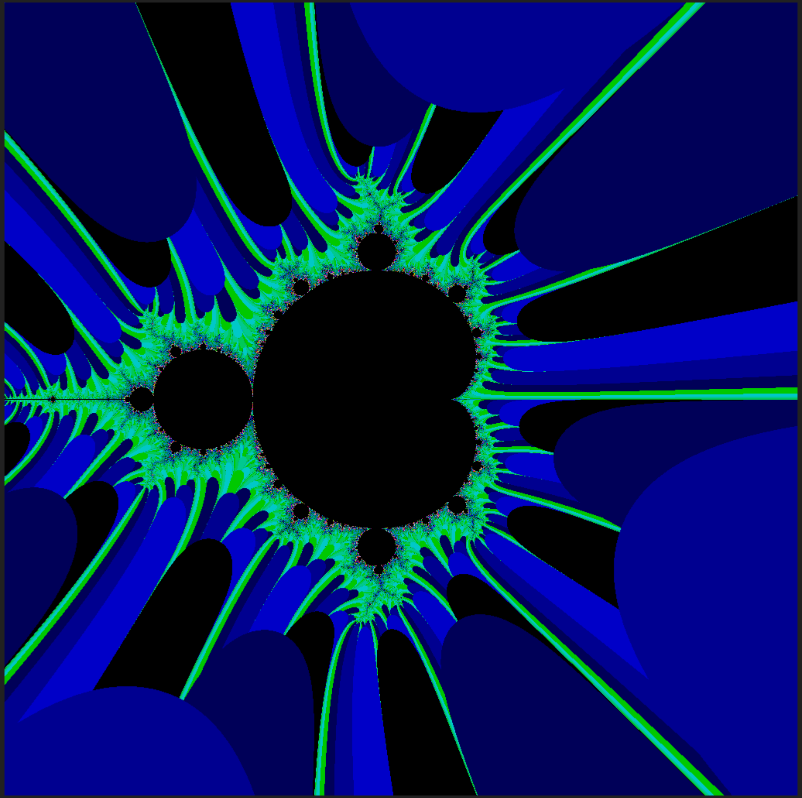
X = 0.5, Y = 0.318, R = 0.005





X = 0.5, Y = 0.318, R = 0.005





Four Mandelbrot Runs

- Sequential Run
  - Loop through for each point
- Parallel Run (TBB)
  - Parallel_for

```
tbb::parallel_for (int(0), int(height), [&] (int j)
{
    tbb::parallel_for (int(0), int(width), [&] (int i){
        pix[i+(height*j)] = GetIterNum(i, j, y, x, max);
    });
});
```

Four Mandelbrot Runs

- Vectorized Run with SSE
  - Streaming SIMD Instructions (SSE)
  - Utilizes 64/128 bit XMM registers to perform vectored operations
    - Tailored for performing same operations on multiple piece of data.
  - Intrinsics on emmintrin.h
- Vectorized and Parallelized (SSE+TBB)
- Intel Atom N2600 has Intel® SSE2, Intel® SSE3, Intel® SSSE3

`__m128i _mm_add_epi8 (__m128i a, __m128i b)`	paddb
`__m128d _mm_add_pd (__m128d a, __m128d b)`	addpd
`__m128 _mm_add_ps (__m128 a, __m128 b)`	addps
`__m128d _mm_add_sd (__m128d a, __m128d b)`	addsd
`__m64 _mm_add_si64 (__m64 a, __m64 b)`	paddq
`__m128 _mm_add_ss (__m128 a, __m128 b)`	addss
`__m128i _mm_adds_epi16 (__m128i a, __m128i b)`	paddsw
`__m128i _mm_adds_epi8 (__m128i a, __m128i b)`	paddsb

Retrieved from Intel® Intrinsics Guide

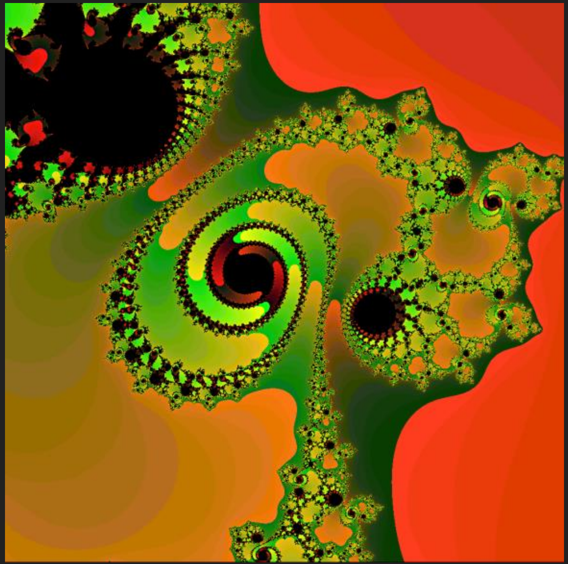
```
__m128 xsqr = _mm_mul_ps(x, x);   // xsqr = x*x
__m128 ysqr = _mm_mul_ps(y, y);   // ysqr = y*y
__m128 xy   = _mm_mul_ps(x, y);   // xy = x*y

__m128 xsqr_plus_ysqr = _mm_add_ps(xsqr, ysqr); // xsqr_plus_ysqr = x*x + y*y

__m128i check    = (__m128i)_mm_cmplt_ps(xsqr_plus_ysqr, threshold); // 0 if done, 0xFFFFFFFF if not
__m128i add_iter = _mm_and_si128(check, ones);                       // 0 if done, 1 if not done
```

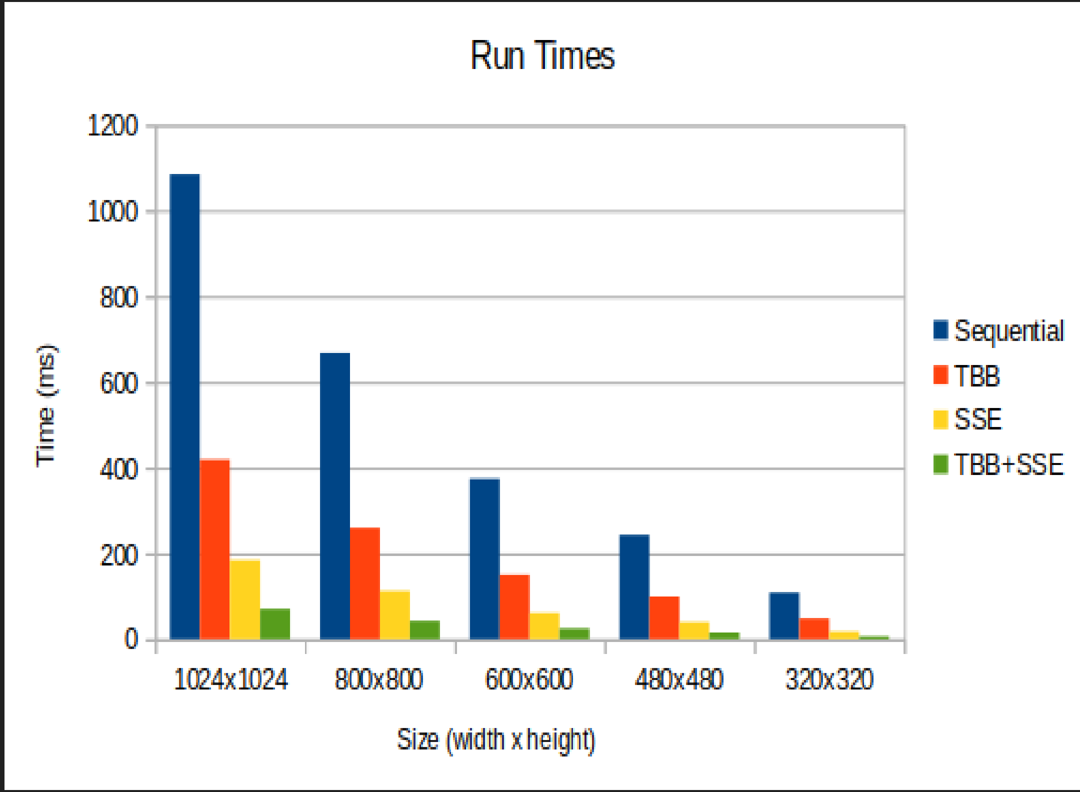
Coloring Algorithm

```
//generating parallel brot
FILE *fp_p = fopen("pbrot.ppm", "wb"); /* b - binary mode */
(void) fprintf(fp_p, "P6 %d %d 255\n", MB.width, MB.height); //P6 means binary, 255 max
uint8_t cfactor_p = 10; // This is a multiplier to change colors
uint8_t color_p[3];
for(i=0; i < SIZE; i++){
    color_p[2] = ((MB.pixarr_p[i] & 0xff) >> 6 )*cfactor_p;    //less blue
    color_p[1] = ((MB.pixarr_p[i] & 0xff) >> 3 )*cfactor_p;    //some green
    color_p[0] = ((MB.pixarr_p[i] & 0xff)      )*cfactor_p;    //more red
    (void) fwrite(color_p, 1, 3, fp_p);
}
(void) fclose(fp_p);
```



Adapted from: https://solarianprogrammer.com/2017/10/25/ppm-image-python-3/

Results (at x=0, y=0, r=2, i=256)



(1024x1024) at
X = 0, Y = 0, radius = 2
Iterations = 256

	Time (ms)	Performance Ratio
Sequential	1084.93	-
TBB	419.15	~2.59
SSE	185.15	~5.86
TBB+SSE	71.05	~15.27