

Fractal:

The Mandelbrot Set's Image Generation through Accurate Computation - C++/GMP Implementation and Verification

There is a program named "backfract" on the web (its link: [Backfract](#). Its author also released a Go game front end named "cgoban", which can be found by Baidu search engine while backfract can't). It can render Mandelbrot set fractals. This program can display various areas of Mandelbrot set in full screen automatically and the visual effect is very good (Here I list some of them: [backfract examples](#)). It runs on _unix/X. On its homepage it is said that the program "randomly" selects "interesting" areas to display, but does not purely zoom in, sometimes it zooms in, sometimes it zooms out next. There are also some videos on the web alleged to be able to zoom in by a factor of 10's power "N". Usually they finish playing back in several minutes. I consider whether I myself can zoom it in infinitely, at least as most as possible.

In mathematics, there are two kinds of definitions for Mandelbrot set. In the book Chaos Fractals And The Applications and the book The Computer Images of Fractals And Its Applications, roughly it speaks like this: The Mandelbrot set is related to another kind of set: Julia set. Notate the quadratic mapping with point c as its parameter as $f_c = z^2 + c$, for any point c on the complex plane there is a corresponding Julia set. Some c points' corresponding Julia sets are connected while others are not, by this, there is below definition for Mandelbrot set:

Mandelbrot set M is a set of parameters c that each makes the Julia set corresponding to the f_c mapping of it a connected set, i.e.

$$M = \{c \in \mathbb{C} \mid J(f_c) \text{ is a connected set}\}$$

If we plot every c that is accompanied by a connected corresponding Julia set on the complex plane, then the set of c 's is M set fractal. This definition is not good for computer processing, luckily, another equivalent definition of Mandelbrot set has been proved in mathematics:

$$M = \left\{c \in \mathbb{C} \mid \lim_{n \rightarrow \infty} Z_n \nrightarrow \infty\right\}$$

in which,

$$\begin{aligned} Z_0 &= 0 \\ Z_{n+1} &= Z_n^2 + c \end{aligned}$$

This web page ([The Mandelbrot Set](#)) talked about handling of some problems met in the actual plotting:

(1) Judgement of infinity

In mathematics, it can be proved that the Mandelbrot set is fully enclosed inside the closed circle which takes the origin as its center and is of radius 2, that is:

$$M \subset \{c \in \mathbb{C} \mid |c| \leq 2\}$$

So if one intermediate result of the iteration is of a modulus larger than 2, then it will definitely go to infinity if the iteration continues. So we use modulus 2 as a threshold value.

(2) Times of iteration

The author of the above web page gave an example applying 50 times. But I ever met the case of different results between 50 times iteration and 200 or 1000 times iteration. So I haven't been clear about the mathematical basis of this question yet.

(3) Coloring

In fact the Mandelbrot set is only of the form of the middle (black) part. All the points outside of it do not belong to Mandelbrot set. It is not very visually appealing by simply plotting a black and white image marking whether each point belongs to Mandelbrot set or not. The various colorful details we have usually seen depend on coloring. One common method is to select the color according to the modulo arithmetic of iteration times of the "escaped" point. (One other method is similar to contour line plotting, which is not used here.) But the effect of a rather simple coloring cannot compare with that produced by a program like backfract.

(The pseudo-code that calculates whether the point of coordinate (i, j) in the drawing window belongs to the M set or not and then colors it):

```
x0 = left_coord + delta * i;
y0 = above_coord - delta * j;
x = x0; // or x = 0
y = y0; // or y = 0
int k = 0;
while( k < MAX_ITERATION )
{
    real = x^2 - y^2 + x0;
    imag = 2xy + y0;
    |z|^2 = real*real + imag*imag;
    if( |z|^2 > 4 )
        break;
    else{ x = real; y = imag; }
    k++;
}
if( k < MAX_ITERATION )
    SetPixel(hdc, i, j, colors[ k % 50 ] );
else
    SetPixel(hdc, i, j, RGB(0, 0, 0));
```

The floating point computation got the IEEE 754 standard, and books on computer architecture generally will talk about it. There are also books such as Floating Point Computation Programming Principles Implementation And Application by Chun-Gen Liu (in Chinese) that illustrate it. And there is a famous article on the web: What Every Computer Scientist Should Know About Floating-Point Arithmetic.

This article got three related programs:

The first one uses machine float, that is, the double data type in c/c++ and plots Mandelbrot set in VC++. Besides, it lets the user select an area to zoom in various parts by "rubber band";

The second one adds into the above code my class implementation for floating-point number, and compare the result to that of the above machine float. But I haven't modified and finished the function of "rubber band" yet;

The third one doesn't take a GUI, it purely compare the computation result of my floating-point implementation with that of GMP fraction implementation so to check their consistency.

The three programs were developed under WinXP + VC++6 on VMWare3 at the very beginning. Later they are updated to Win7 + vs2015(MFC installed) on VMWare6, with minor modifications. Both versions are provided here:

[VC++6 version](#) [vs2015 version](#)

(The code had not been cleaned up, :-(.)

The 1st program

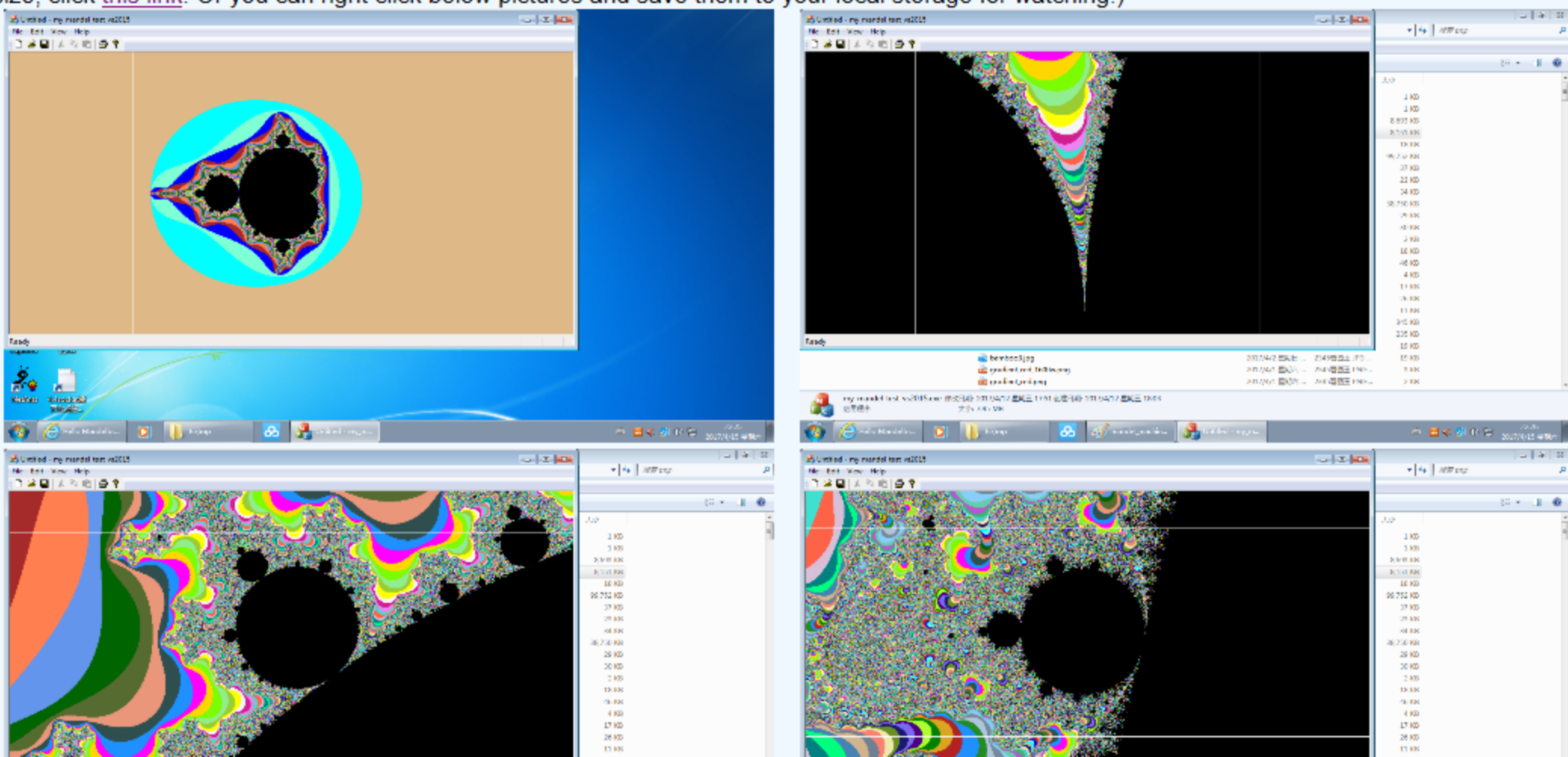
(1)double type implementation

This code part of operation is nearly the same as the above pseudo-code:

my_mandel_test_3(machine_float)my_mandel_testView.cpp:

```
1   for(i = 0; i < w; i++)
2   {
3       for(j = 0; j < h; j++)
4       {
5           BOOL b_is_boundary = FALSE;
6           if(delta_w > delta_h)
7           {
8               if( j == (int)( (h*delta - (old_above_coordinate - \
9                           old_below_coordinate))/(2*delta) )
10                  || j == ( h-(int)( (h*delta - (old_above_coordinate - \
11                           old_below_coordinate))/(2*delta) ) )
12                  )
13                   b_is_boundary = TRUE;
14           }
15           else
16           {
17               if( i == (int)( (w*delta - (old_right_coordinate - \
18                           old_left_coordinate))/(2*delta) )
19                  || i == ( w-(int)( (w*delta - (old_right_coordinate - \
20                           old_left_coordinate))/(2*delta) ) )
21                  )
22                   b_is_boundary = TRUE;
23           }
24
25           double x0 = left_coordinate + delta * (double)i;
26           double y0 = above_coordinate - delta * (double)j;
27           double x = x0;
28           double y = y0;
29
30           int k = 0;
31           #define MAX_ITERATION 1000
32           while(k < MAX_ITERATION)
33           {
34               double real_part = x*x-y*y+x0;
35               double imaginary_part = 2*x*y+y0;
36               double z_abs_square = real_part*real_part + imaginary_part*imaginary_part;
37               if(z_abs_square > 4.0)
38               {
39                   break;
40               }
41               else
42               {
43                   x = real_part;
44                   y = imaginary_part;
45               }
46               k++;
47           }
48           if(k < MAX_ITERATION)
49           {
50               //unsigned long colors[16] =
51               unsigned long colors[50] =
52               {
53                   RGB(0xDE , 0xB8 , 0x87),
54                   ...
55                   RGB(0x19 , 0x19 , 0x70 )
56               };
57               //SetPixel(hdc, i, j , colors[k%16]);
58               SetPixel(hdc, i, j , colors[k%50]);
59               if(b_is_boundary)
60                   SetPixel(hdc, i, j , RGB(255, 255, 255));
61           }
62           else
63           {
64               SetPixel(hdc, i, j , RGB(0, 0, 0));
65               if(b_is_boundary)
66                   SetPixel(hdc, i, j , RGB(255, 255, 255));
67           }
68       }
69   }
70 }
```

Its running result is like below:(To save the space, below pictures displayed are those zoomed out by the browser. To browse them in their original size, click [this link](#). Or you can right click below pictures and save them to your local storage for watching.)





The proportion of the area selected to zoom in by the user is very likely not the same as that of the drawing window. So I make some extension to it to adapt to the drawing window so to make every pixel point inside the drawing window being plotted. The inner part within the two white lines in each of the above images is the area selected by the user (the visual effect maybe is poor for small pictures, you can turn to the original ones), which may appear too flat or too high for the drawing window. The judgement processing of "b_is_boundary" in the above code is for this reason.

(Two intermediate text files draw_coordinates.txt and arguments.txt are generated after you run the program. You need to delete them manually before the next run.)

The machine double float is of a binary precision of 53 bits, roughly 16 decimal places of accuracy. The starting horizontal and vertical coordinates are both of a magnitude of about (-2,2). If each time the dimension of the selected area for zooming in is about one tenth of that of the original image, we may find that after about 16 times of selection the image went "mosaic". Bare 16 times, this is far less than infinitely zooming it in. Besides, we will see behind that after about 40 times of iteration, the log already shows that the accumulation of error from machine float operation leads to obvious fault.

The 2nd program

(2) my floating-point class implementation

Since the calculation is only concerned with addition and multiplication operation, we should be able to do completely accurate calculation for the terminating decimals. The arithmetic taught in primary school should suffice. In order that we only need to compute terminating decimals, we only use starting coordinates and step values with finite fraction part. The latter is achieved by specifying the width and height of the drawing window as some particular values such as 500/512/800/1000/1024 so that the gap value between two pixel points is also a simple terminating decimal.

That's why I modified the code in View.cpp to support only these particular values.

my_mandel_test_my_floats_vs2015\my_mandel_test_my_floats_vs2015View.cpp:

```
1  if (!(w == 200 || w == 250 || w == 400 || w == 500
2      || w == 800 || w == 1000 /*|| w == 1024*/))
3      || !(h == 200 || h == 250 || h == 400 || h == 500
4          || h == 800 || h == 1000 /*|| h == 1024*/))
5      )
6
7      {
8          AfxMessageBox(L"width and height not supported");
9          return;
10     }
11
12     char* one_w_strI = (w == 200) ? "0.005" : (
13         w == 250 ? "0.004" : (
14             w == 400 ? "0.0025" : (
15                 w == 500 ? "0.002" : (
16                     w == 800 ? "0.00125" : (
17                         w == 1000 ? "0.001" : ("error characters")
18                     )
19                 )
20             )
21         )
22     );
23
24     char* one_h_strI = (h == 200) ? "0.005" : (
25         ...
```

The reader maybe still needs to modify the code in View.h, setting the width and height of the program window that suit your display area and meet the above specification.

my_mandel_test_my_floats_vs2015\my_mandel_test_my_floats_vs2015View.h:

```
1  #define DEFAULT_CLIENT_AREA_WIDTH  500//1000
2  #define DEFAULT_CLIENT_AREA_HEIGHT 250// 800
```

The starting coordinates are: x: -2.5 to 2.5 , y: -2.0 to 2.0 :

my_mandel_test_my_floats_vs2015\my_mandel_test_my_floats_vs2015View.h:

```
1  fI fI_left_coord("-2.25");
2  fI fI_right_coord("2.25");
3  fI fI_above_coord("2.0");
4  fI fI_below_coord("-2.0");
```

Three new files(my_floats.h, my_floatI.cpp and my_floatII.cpp) that implement my floating-point class and several helper functions were added.

(Sometimes it reported error when I built the project under VC6 but passed after I re-added those files to the project. And it could never pass building under vs2015, reporting re-definitions found in nafxcd.lib/uafxcwd.lib and libcmdt.lib. To solve this issue, one maybe need to adjust the project. First, ignore these two libraries in the "input" entry of the "link" option of the project; then, add these two libraries as per the above order to the "Other library dependencies".) Two different kinds of data structure for floating-point were implemented successively. Because it occurred to me that the latter new one maybe would be better than the former one that had been implemented, I did them one after another.

The first implementation:

my_floats.h:

```
1  class float_number{
2  public:
3      bool    minus;
4      char*   number; //in absolute value form
5      int     point_pos;
6
7      ...
8  };
```

The second implementation:

my_floats.h:

```
1  class floatII{
2  public:
3      bool    minus; //with '-' sign?
4      char*   digits; //no '.' inside
5      int     point_pos_r; //counting from the right side
6
7      ...
8  };
```

Sometimes abridged as:

my_mandel_test_my_floats_vs2015\my_mandel_test_my_floats_vs2015View.cpp:

```
1 typedef float_number fI;
2 typedef floatII fII;
```

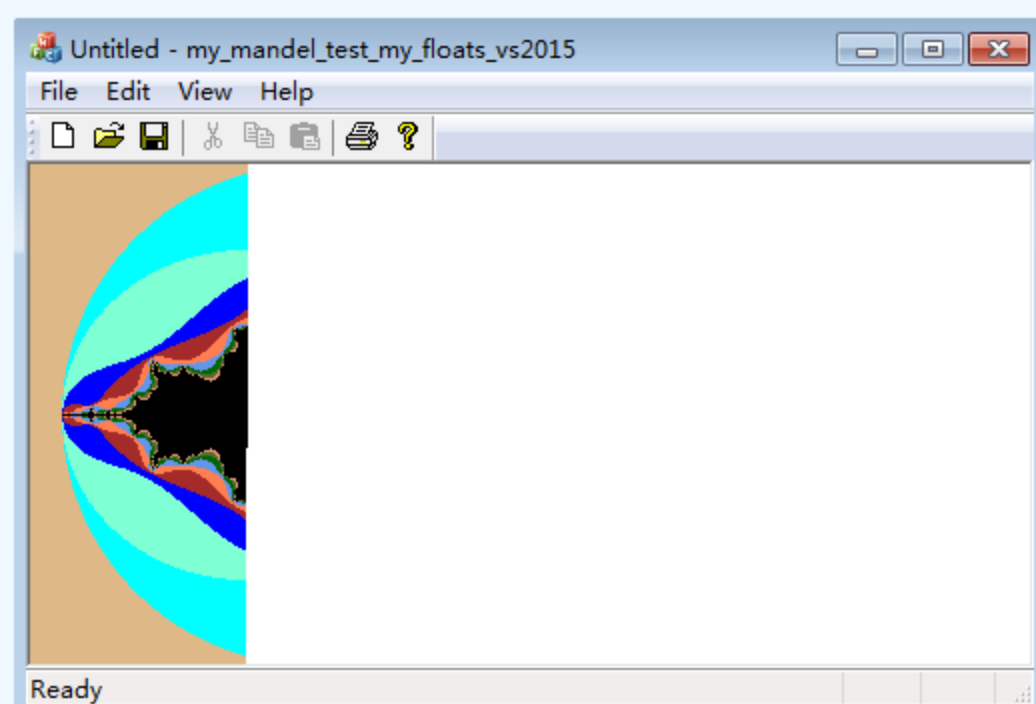
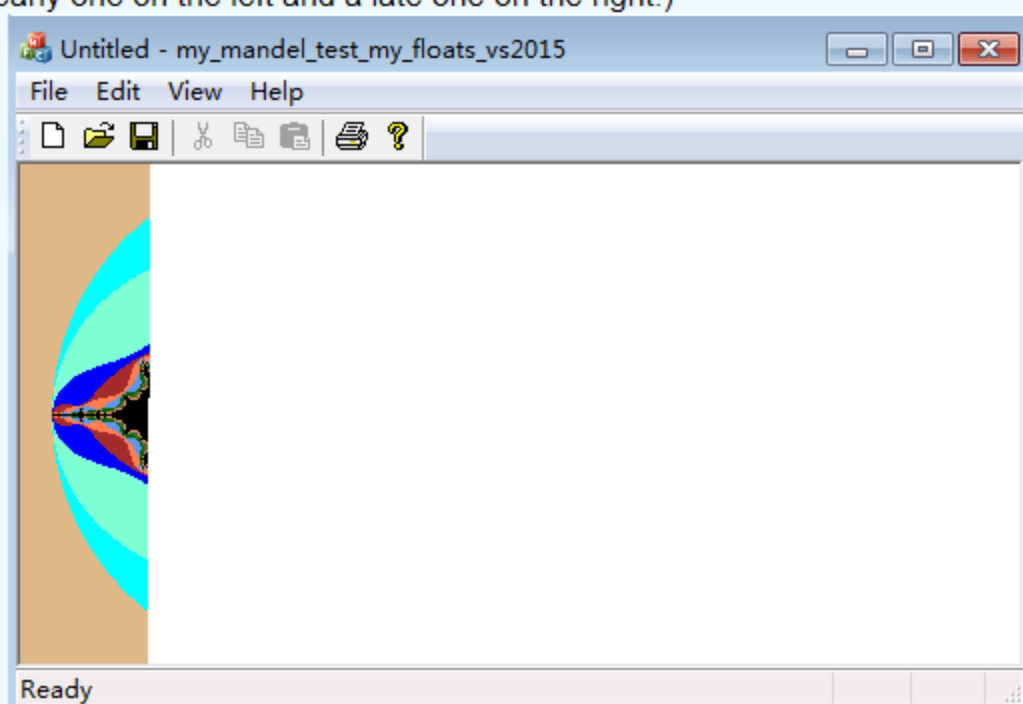
Both classes use separated member "minus" to represent whether the floating-point number is positive or negative, so the digits string of the number (member "number" or "digits") all goes without symbol of sign. The first difference between the two classes is that the decimal point was reserved in the digits string in the previous class implemented, but it vanishes in the latter class. The second difference is the mark of the place of the decimal point, in the previous class it is counted from left to right(point_pos), in the latter class, however, is counted from right to left(point_pos_r). Thus the implementation code of the latter class is more concise. The computation results of these two implementations can be compared to each other to verify the code. This is a second way for verification. But the cores of the algorithm of these two classes are the same, so probably its function for verification is limited. A complete verification relies on the third program below which implements the contrast of the computation results herein to that of the GMP calculation. That is a third way for verification, also can be said as the real verification. (A first way for verification is a rather local one, it only verifies addition, i.e. do a reverse operation back after an addition finishes and see if the result can be restored correctly. This way is little used.)

Floating-point classes being accomplished, take fI class implementation as an example, the code is changed to roughly like this:(The actual code will interleave a little because of the need for result comparison. The example here is of an early version of code.)

```
my_mandel_test_my_floats_0\my_mandel_testView.cpp:
1 fI fI_left_coord ("-2.25");
2 fI fI_right_coord( "2.25");
3 fI fI_above_coord( "2.0" );
4 fI fI_below_coord("-2.0" );
5
6 fI delta_wI, delta_hI, deltaI;
7
8 #if 1
9 fI w_factorI(one_w_strI);
10 fI h_factorI(one_h_strI);
11
12 delta_wI = (fI_right_coord - fI_left_coord) * w_factorI;
13 delta_hI = (fI_above_coord - fI_below_coord) * h_factorI;
14
15 if(delta_wI > delta_hI)
16     deltaI = delta_wI;
17 else
18     deltaI = delta_hI;
19 #endif
20
21 #if 1
22 fI x0I(fI_left_coord );
23 fI y0I(fI_above_coord);
24 fI aI("2");
25 fI threshold("4.0");
26 #endif
27
28 #if 1
29 for(i = 0; i < w; i++)
30 {
31     y0I = fI_above_coord;
32     for(j = 0; j < h; j++)
33     {
34         fI xI("0");
35         fI yI("0");
36
37         int k = 0;
38         #define MAX_ITERATION 10//20//100
39         while(k < MAX_ITERATION)
40         {
41             fI real_part      = xI*xI-yI*yI+x0I;
42             fI imaginary_part = aI*xI*yI +y0I;
43             fI z_abs_square = real_part*real_part + imaginary_part*imaginary_part;
44             if(z_abs_square > threshold)
45             {
46                 break;
47             }
48             else
49             {
50                 xI = real_part;
51                 yI = imaginary_part;
52             }
53             k++;
54         }
55
56         if(k < MAX_ITERATION)
57         {
58             SetPixel(hdc, i, j , colors[k%50]);
59         }
60         else
61         {
62             SetPixel(hdc, i, j , RGB(0, 0, 0));
63         }
64
65         y0I = y0I - deltaI;
66     }
67     x0I = x0I + deltaI;
68 }
69 #endif
```

The fII class code is similar, basically it just changes "I" to "II" or appends a suffix "II".

The illustrations showing the running result:(Now one can see the program is drawing the pixels of the screen one by one. Below are two images, an early one on the left and a late one on the right.)



(The green wireframes orderly marked the real part, the imaginary part and (the high-precision approximation value of) the square of the modulus of z computed using my class code(`fl` and `fi`, which have the same result values).)

```
log.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

To calc delta...
Ended calc delta
(i,j)(0 0)
(x0I y0I) is (-2.25, 2.0)
(x0II y0II) is (-225, 20)
[machine float(k = 0)]real:-2.25000000000000000000000000000000,
imag:2.00000000000000000000000000000000,z2:9.06250000000000000000000000000000
my floatI(==II) approx: real_I_approx(len:4): -2.25,
my floatI(==II) approx: imaginary_I_approx(len:3): 2.0,
z2_I(len:6): 9.0625
z2_II(len:5, point_pos_r:4): 90625

I.THREE NUMBERS LENGTHS ARE:4,3,6
II.THREE NUMBERS LENGTHS ARE:3,2,5

Iterate 0 rounds.
(i,j)(0 1)
(x0I y0I) is (-2.25, 1.9840)
(x0II y0II) is (-225, 19840)
[machine float(k = 0)]real:-2.25000000000000000000000000000000,
imag:1.98399999999999999999999999999999,z2:8.99875600000000000000000000000000
my floatI(==II) approx: real_I_approx(len:4): -2.25,
my floatI(==II) approx: imaginary_I_approx(len:6): 1.9840,
z2_I(len:10): 8.99875600
z2_II(len:9, point_pos_r:8): 899875600

I.THREE NUMBERS LENGTHS ARE:4,6,10
II.THREE NUMBERS LENGTHS ARE:3,5,9

Iterate 0 rounds.
(i,j)(0 2)
(x0I y0I) is (-2.25, 1.9680)
```

Below we give one more example, the pixel point (16 125):

[illegible]

It can be seen that the point lies at (16 125) is on the real axis and belongs to M set. So the iteration reached the maximum(here it is 10 times) and the value still didn't exceed the threshold. The code did no more calculation and went to next point (16 126).

It brings dense computation to conduct accurate calculation using my floating-point classes. In my code, the calculation code was simply placed inside the OnDraw() function, this resulted in that the program didn't respond to external messages. This no responding maybe doesn't matter much under Win XP yet, one can still see the program is still drawing on the screen one pixel by one pixel. But it becomes worse under Win 7: the program draws no more than a few pixels and then just displays "The program does not respond".

To avoid blocking message loop because of dense computation (Blocking message loop is not blocking process. OnDraw() is located in the main thread, dense computation inside of it will make the main thread down inside this function for a long time, and will delay the message loop (that

thread, dense computation inside of it will make the main thread run inside this function for a long time, and will delay the message loop (that implemented by MFC itself), I need to either start another worker thread to compute, or insert message handler loop code by myself into the dense computation. Here the latter method is used. But it met a problem when closing the window, the process doesn't exit and the OnDraw() is still running, so I can only add some handling in the OnDestroy() function in View class.

Being able to conduct accurate calculation using my floating-point classes, it may be found that how the error from machine floating-point (i.e. double type) computation which is intrinsically approximate could accumulate. Below is the log example. ([log files for download](#) (Below extracted part is at the very end of the log file.))

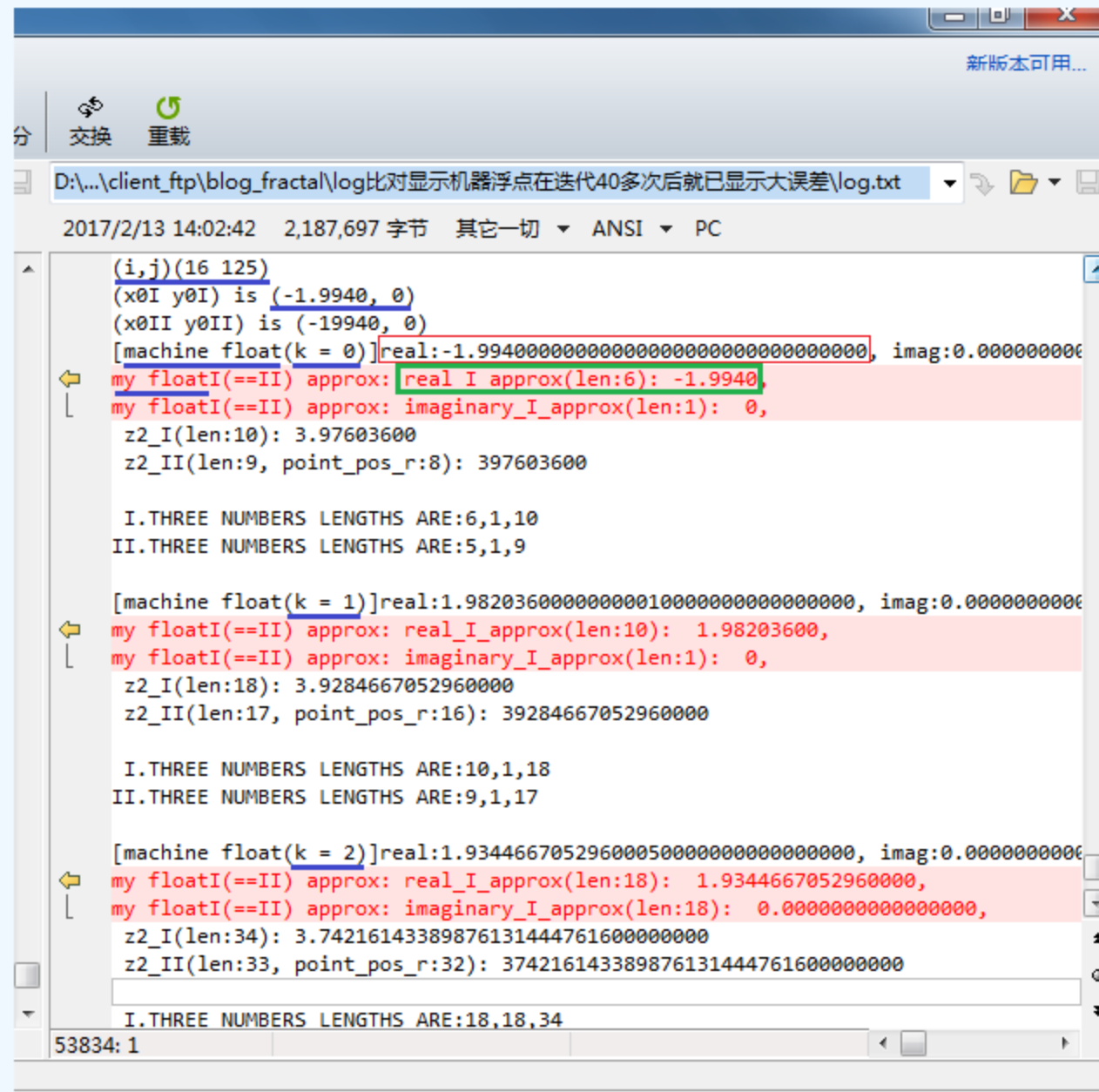
(Since the iteration times is large, here it is the approximation in some certain precision of the calculated result using my floating-point class that is compared to the calculated result using machine floating-point, of course, this precision is highly better than that of double type.)

Here it is the iteration of the point of coordinate (16, 125) that is logged:

(The value inside the red rectangle is machine floating-point result. The value inside the green rectangle is my floating-point result.)

(P.S. In the path showed in below image, the name of the directory that log.txt lies in is in Chinese, which means "logs comparison that shows machine floating-point got obvious error after more than 40 times of iteration"; see the sub-directory name of the above log file for download.)

When $k = 0$, the results are the same;



```
(i,j)(16 125)
(x0I y0I) is (-1.9940, 0)
(x0II y0II) is (-19940, 0)
[machine float(k = 0)]real:-1.99400000000000000000000000000000, imag:0.00000000000000000000000000000000
my floatI(==II) approx: real_I_approx(len:6): -1.9940
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
z2_I(len:10): 3.97603600
z2_II(len:9, point_pos_r:8): 397603600

I.THREE NUMBERS LENGTHS ARE:6,1,10
II.THREE NUMBERS LENGTHS ARE:5,1,9

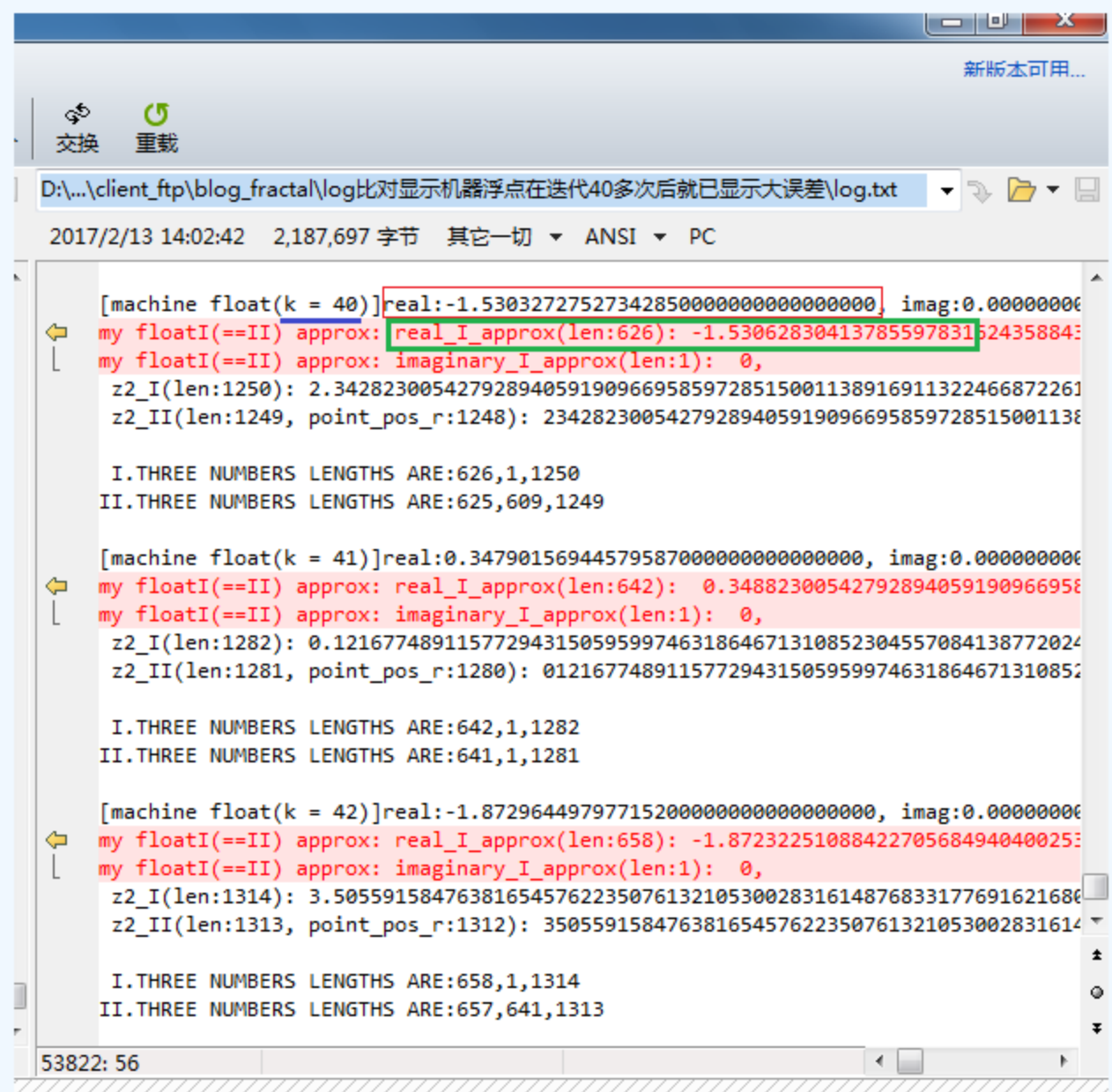
[machine float(k = 1)]real:1.98203600000000001000000000000000, imag:0.00000000000000000000000000000000
my floatI(==II) approx: real_I_approx(len:10): 1.98203600,
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
z2_I(len:18): 3.9284667052960000
z2_II(len:17, point_pos_r:16): 39284667052960000

I.THREE NUMBERS LENGTHS ARE:10,1,18
II.THREE NUMBERS LENGTHS ARE:9,1,17

[machine float(k = 2)]real:1.93446670529600050000000000000000, imag:0.00000000000000000000000000000000
my floatI(==II) approx: real_I_approx(len:18): 1.9344667052960000,
my floatI(==II) approx: imaginary_I_approx(len:18): 0.0000000000000000,
z2_I(len:34): 3.74216143389876131444761600000000
z2_II(len:33, point_pos_r:32): 37421614338987613144476160000000

I.THREE NUMBERS LENGTHS ARE:18,18,34
53834: 1
```

When $k = 40$, the difference isn't large;



```
[machine float(k = 40)]real:-1.53032727527342850000000000000000, imag:0.00000000000000000000000000000000
my floatI(==II) approx: real_I_approx(len:626): -1.53062830413785597831524358843
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
z2_I(len:1250): 2.3428230054279289405919096695859728515001138916911322466872261
z2_II(len:1249, point_pos_r:1248): 23428230054279289405919096695859728515001138

I.THREE NUMBERS LENGTHS ARE:626,1,1250
II.THREE NUMBERS LENGTHS ARE:625,609,1249

[machine float(k = 41)]real:0.34790156944579587000000000000000, imag:0.00000000000000000000000000000000
my floatI(==II) approx: real_I_approx(len:642): 0.34882300542792894059190966958
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
z2_I(len:1282): 0.1216774891157729431505959974631864671310852304557084138772024
z2_II(len:1281, point_pos_r:1280): 01216774891157729431505959974631864671310852

I.THREE NUMBERS LENGTHS ARE:642,1,1282
II.THREE NUMBERS LENGTHS ARE:641,1,1281

[machine float(k = 42)]real:-1.87296449797715200000000000000000, imag:0.00000000000000000000000000000000
my floatI(==II) approx: real_I_approx(len:658): -1.87232251088422705684940400253
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
z2_I(len:1314): 3.5055915847638165457622350761321053002831614876833177691621680
z2_II(len:1313, point_pos_r:1312): 35055915847638165457622350761321053002831614

I.THREE NUMBERS LENGTHS ARE:658,1,1314
II.THREE NUMBERS LENGTHS ARE:657,641,1313
53822: 56
```

When $k = 47$, there is difference at the first decimal place;

```
[machine float(k = 46)]real:1.63535399957202790000000000000000, imag:0.000000000000000000
my floatI(==II) approx: real_I_approx(len:722): 1.65170099296666312452490142093
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
z2_I(len:1442): 2.7281161701670609483496360076288512863138199408583340321828907
z2_II(len:1441, point_pos_r:1440): 27281161701670609483496360076288512863138199

I.THREE NUMBERS LENGTHS ARE:722,1,1442
II.THREE NUMBERS LENGTHS ARE:721,1,1441

[machine float(k = 47)]real:0.68038270391622802000000000000000, imag:0.000000000000000000
my floatI(==II) approx: real_I_approx(len:738): 0.73411617016706094834963600762
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
z2_I(len:1474): 0.5389265513007531871459145740926459255176174200209479090983249
z2_II(len:1473, point_pos_r:1472): 05389265513007531871459145740926459255176174

I.THREE NUMBERS LENGTHS ARE:738,1,1474
II.THREE NUMBERS LENGTHS ARE:737,721,1473

[machine float(k = 48)]real:-1.53107937621164240000000000000000, imag:0.000000000000000000
my floatI(==II) approx: real_I_approx(len:754): -1.45507344869924681285408542596
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
z2_I(len:1506): 2.1172387411095196464541557084097208106400794744318897322720201
z2_II(len:1505, point_pos_r:1504): 21172387411095196464541557084097208106400794

I.THREE NUMBERS LENGTHS ARE:754,1,1506
II.THREE NUMBERS LENGTHS ARE:753,1,1505
```

When $k = 52$, already we cannot think they are close ;

```
I.THREE NUMBERS LENGTHS ARE:770,1,1538
II.THREE NUMBERS LENGTHS ARE:769,1,1537

[machine float(k = 50)]real:-1.87135711897860000000000000000000, imag:0.000000000000000000
my floatI(==II) approx: real_I_approx(len:786): -1.97881221268974079230045491959
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
z2_I(len:1570): 3.9156977730900679503131338966665635900816493725360547218514042
z2_II(len:1569, point_pos_r:1568): 39156977730900679503131338966665635900816493

I.THREE NUMBERS LENGTHS ARE:786,1,1570
II.THREE NUMBERS LENGTHS ARE:785,769,1569

[machine float(k = 51)]real:1.50797746675188620000000000000000, imag:0.000000000000000000
my floatI(==II) approx: real_I_approx(len:802): 1.92169777309006795031313389666
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
z2_I(len:1602): 3.6929223310993262880789603594235039056401722441869425498145082
z2_II(len:1601, point_pos_r:1600): 36929223310993262880789603594235039056401722

I.THREE NUMBERS LENGTHS ARE:802,1,1602
II.THREE NUMBERS LENGTHS ARE:801,1,1601

[machine float(k = 52)]real:0.27999604023143587000000000000000, imag:0.000000000000000000
my floatI(==II) approx: real_I_approx(len:818): 1.69892233109932628807896035942
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
z2_I(len:1634): 2.8863370871079688587552354032802987659232106168329333404131896
z2_II(len:1633, point_pos_r:1632): 28863370871079688587552354032802987659232106
```

Since $k = 53$, the results might have got different plus or minus signs!

```
[machine float(k = 53)]real:-1.91560221745471600000000000000000, imag:0.000000000000000000
my floatI(==II) approx: real_I_approx(len:834): 0.89233708710796885875523540328
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
z2_I(len:1666): 0.7962654770283348028283796859857320631442923927033780492482284
z2_II(len:1665, point_pos_r:1664): 07962654770283348028283796859857320631442923

I.THREE NUMBERS LENGTHS ARE:834,1,1666
II.THREE NUMBERS LENGTHS ARE:833,1,1665

[machine float(k = 54)]real:1.67553185551742500000000000000000, imag:0.000000000000000000
my floatI(==II) approx: real_I_approx(len:850): -1.19773452297166519717162031401
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
```

```
z2_I(len:1698): 1.4345679875181623859009060729373019985407988315280917260401238
z2_II(len:1697, point_pos_r:1696): 14345679875181623859009060729373019985407988

I.THREE NUMBERS LENGTHS ARE:850,1,1698
II.THREE NUMBERS LENGTHS ARE:849,833,1697

[machine float(k = 55)]real:0.81340699885366541000000000000000, imag:0.00000000
my floatI(==II) approx: real_I_approx(len:866): -0.55943201248183761409909392706
my floatI(==II) approx: imaginary_I_approx(len:1): 0,
z2_I(len:1730): 0.3129641765894789160576388523480401449747801644140285524498936
z2_II(len:1729, point_pos_r:1728): 03129641765894789160576388523480401449747801

I.THREE NUMBERS LENGTHS ARE:866,1,1730
II.THREE NUMBERS LENGTHS ARE:865,1,1729

54280: 46 默认文本
```

(It seems that "backfract" uses "long double" type instead of "double". Still this is too little.)

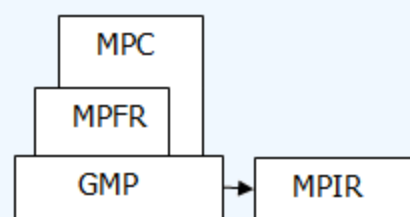
The function to select a region to zoom in is not implemented in this version of code. Besides, the zooming-in of M set is endless, I need to consider how to zoom it in on earth. Manual selection? Or automatic selection? How should I present the image patterns of the M set?

The 3rd program

(3) GMP implementation

GMP library is the GNU "big number" library. The GMP library I used was built under MinGW on WinXP on VMWare3, it can then be directly linked into the applications developed under VC++6, along with some more libraries of MinGW and Windows. When I upgrade/update my program on Win 7+vs2015 on VMWare6, the library itself is still usable, only that the fprintf() function cannot be used for the reason of libmsvcrt.a of MinGW. So I replaced all the "fprintf"s with "fwrite"s. For running, some VC libraries from VS and some libraries from MS SDK that are prompted to be in need are added.

GNU big number libraries are composed of GMP, MPFR, MPC. GMP is big integer and fraction number library. It also included function part for floating-point operation but that now is replaced by MPFR(said in GMP manual). MPFR is arbitrary precision big floating-point number library, it is based on GMP. MPC is big complex number library and it is based on the former two libraries. (It is interesting that GCC building is dependent on these libraries.) Besides, There is another library [MPIR](#) branching from GMP. It is said that one of MPIR's goals is to become a big number library that can be built with VS under Windows because GMP cannot. Here MPIR is not used. I ever thought I would need MPFR to realize the function needed, thus "mpfr" is used in the file name ("mandel_with_mpfr_test") but later I found that mere the fraction part of GMP library could suffice. And the latter calculates in absolute precision, not limited to the arbitrary precision of the former. Yet the file names are not modified. Actually nearly my whole program has nothing to do with MPFR except only at one place I use it to print a log record.



I installed a MinGW+MSYS offline-install package downloaded from the web on WinXP on VMWare. As the manual and "configure" help guided, I configured and compiled GMP and MPFR(The source package of MPFR got a file(INSTALL) talking about three ways for building it under Windows (MinGW, Cygwin and vs2015)). There is nothing peculiar. But the library to be built can only be specified either as static or as shared/dynamic each time. I just built static ones. The two static libraries(libgmp.a and libmpfr.a) obtained can directly be used by VC++6 and vs2015:

```
mandel_with_mpfr_test_vs2015\mpfr_inc.h:
1 #pragma comment(lib, "libmpfr.a")
2 #pragma comment(lib, "libgmp.a")
3 #pragma comment(lib, "libgcc.a")
4 #pragma comment(lib, "libgcc_s.a")
5 #pragma comment(lib, "libmingwex.a")
6 #pragma comment(lib, "libmsvcrt.a")
```

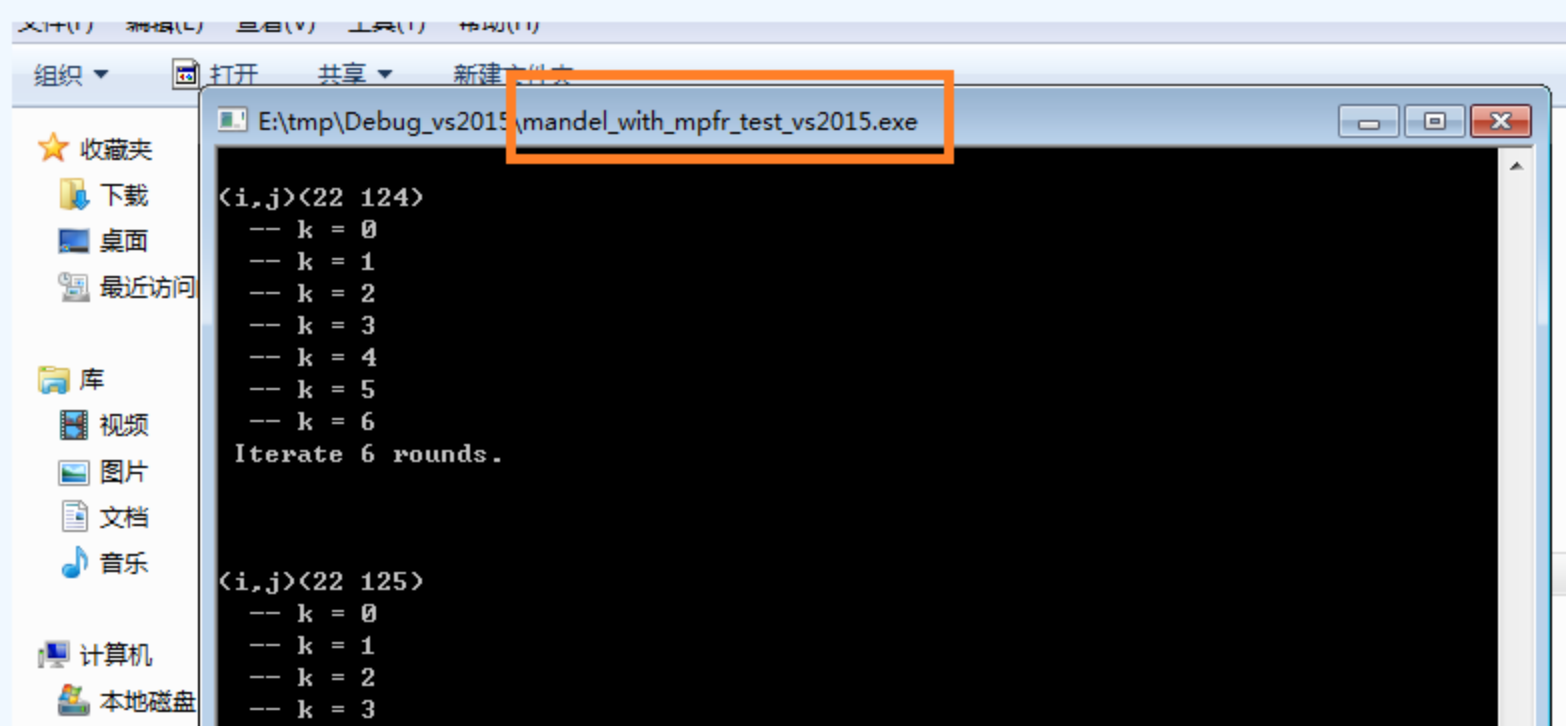
All other lib*.a files are MinGW libraries that are needed when building the program. Besides, to run the program one need libgcc_s_dw2-1.dll of MinGW and some dlls of Windows.

GMP is implemented in c. It got a C++ wrapper library that could be configured and built. But the manual said that this c++ library generally can only be used by the application built with the same (c++) compiler. In development I found that I must use c compiler instead of c++ compiler to compile one source file that includes gmp.h, so my mpfr_lib.c has to be .c file instead of .cpp file.

Link order is required for MPFR library, first libmpfr.a, then libgmp.a. And the manual says that in some cases one must specify /MD option for the Windows cl compiler to build VC++ application using GMP library.

(There is an open source project on fractals named "FractalNow". It got MPFR computation joined into its code since version 8.0. This project is dependent on Qt. I built this project under both Ubuntu and WinXP MinGW. It got a debian package in the Ubuntu repository and was rather easy for building. On Windows, however, the Qt4.8 it depends on requires GCC4.4 for building. I could only search for and download one MinGW package with GCC4.4, then copy the earlier MSYS directory, moreover, download the "pthreads-w32" package, thus accomplished its building.)

Running effect:



[illegible]

In this second illustration below it accomplished the comparison between the exact value of the imaginary part and the approximate value of the imaginary part. For this point is on the real axis with an imaginary part of 0, the log is short.

```
log1.txt - 记事本
```

```
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
```

```
---- to check against real_partII_approx...  
  
(续上)  
my floatI(II): imaginary I(len:1): 0,  
my floatI(==II) approx: imaginary I approx(len:1): 0,  
MPFR value of imag:00000000000000000000000000000000, exp:0, [FRACTION: 0 / 1]  
---- to check against imaginary_partII...  
MPFR value of imag approx:00000000000000000000000000000000, exp:0, [FRACTION: 0 / 1]  
---- to check against imaginary_partII_approx...  
  
z2_I(len:4098):  
2.02777727626111620407047246204000740440112002205212621020006507278264126122014225454247125162041000552774
```

Finally in this third illustration below it accomplished the comparison between the exact value of the square of z 's modulus and the approximate value of it.

[illegible]

The unmarked text part with words like "[FRACTION ... / ...]" in these illustrations is the representation in fraction form that GMP used, i.e., numerator / denominator, and it is irreducible.

This third program goes without GUI and got only pure calculation. The code now lies in main.cpp instead of View.cpp. All the newly added functions for GMP implementation begin with "GMP_" and lie in two files(mpfr_inc.h and mpfr_lib.c)(as above said, named after "mpfr"). I illustrate it below what to do to turn the pseudo-code before to GMP code:

(pseudo-code:)	(corresponding GMP implementaion)
x0(left);	
y0(above);	
a("2");	
threshold("4.0");	GMP_init_calc()
for(i = 0; i < w; i++)	
{	
y0 = above;	GMP_reset_y0()
for(j = 0; j < h; j++)	
{	
x("0");	GMP_reset_x()
y("0");	GMP_reset_y()
k = 0;	
while(k++ < MAX_ITER)	
{	
real = x*x-y*y+x0;	GMP_calc_real()
imag = a*x*y +y0;	GMP_calc_imag()
z_square = real*real + imag*imag;	GMP_calc_z_square()
if(z_square > threshold)	if(GMP_cmp_threshold() > 0)
break;	
else{	
x = real;	GMP_set_x_as_real()
y = imag;	GMP_set_y_as_imag()
}	
}	
y0 = y0 - delta;	GMP_update_y0()
}	
x0 = x0 + delta;	GMP_update_x0()

GMP_clear()

The only modification to my floating-point class code this third program made is: when an approximation is made to the (intermediate) calculated result of my floating-point implementation, to continue to compare the results and check the consistency of the two implementations, I need to make an corresponding approximation to the (intermediate) calculated result of GMP implementation. That is, I add one more parameter(p_pos) into the truncate_floatII() function to tell GMP implementation side: which decimal place I truncated at here, so that you must truncate correspondingly at that same decimal place of your value there.

Code before and after modification:

The complete code of the above right side:

```
mandel_with_mpfr_test_vs2015\my_floats.h:
1 extern floatII truncate_floatII(int x_precision, int y_precision, int delta_precision,
2 int min_precision_to_truncate,
3 floatII& f,
4 int add_to_tail_precision,
5 int* p_pos);
```

The corresponding modification to my_floats.cpp will be seen soon below.

Currently truncate_floatI() is not modified. Because the results of floatI and floatII are the same, I only need to use one of them, currently I use floatII.

After GMP implementation was added, I found and corrected one (preliminary level, :-() bug in my_floatI.cpp(my_floatII.cpp is the same):

Why we need approximation yet? Because with absolutely accurate computation(not a bit approximation) the need for storage for the intermediate decimal result is incrementing exponentially by 2 to the power of the times of iteration. It counts in K for 10 times of iteration, and counts in M for 20 times of iteration. For 30 times of iteration it counts in G already. Besides, with computation like this on PC one becomes to feel a little slow already after about 10 times of iteration (i.e. a magnitude of 1000 digits times 1000 digits). So I have to do some approximation in the actual computation, for example, to let the need for storage increment linearly with the times of iteration. And take the speed of getting result into consideration.

Presently the algorithm for doing approximation I considered is roughly like this:

Step (0): Specify a requirement for the promotion of precision, e.g., 16 more decimal places of accuracy than the old ones;

Step (1): Specify the decimal places of accuracy, need to take these into consideration:

- (A) the precision of starting coordinate,
- (B) the precision of the step value between coordinates,
- (C) incrementing with the times of iteration(?) ;

Step (2): Does the calculated result exceed the precision requirement? If not, do not modify the value of the result;

Step (3): If the answer is "yes" in step (2), count to the wanted decimal place of accuracy, then:

- (A) if there is already significant digit in its front, then add at most e.g. 16 more decimal places after it.
- (B) if there isn't any significant digit(i.e. all are '0's) in its front, unless being of value 0, find the first significant digit(non-zero one) after it, then add at most e.g. 16 more decimal places after that digit.

Its implementation:(taking floatII as the example) ((Highlighted lines of code is the modification of this function as said above.))

```
mandel_with_mpfr_test_vs2015\my_floatII.cpp:
1 floatII truncate_floatII(int x_precision, int y_precision, int delta_precision,
2 int min_precision_to_truncate,
3 floatII& f,
4 int add_to_tail_precision,
5 int* p_pos)
6 {
7     *p_pos = 0;
8
9     int truncate_part_len = 0;
10    int i;
```



```

11 //if(f.point_pos_r < min_precision_to_truncate)
12 // return f;
13
14 floatII approximation = f;
15
16 int max = x_precision ;
17 max = y_precision > max ? y_precision : max ;
18 max = delta_precision > max ? delta_precision : max ;
19
20 if(f.point_pos_r <= max)
21     return approximation;
22
23 max += add_to_tail_precision;
24 if(f.point_pos_r <= max)
25     return approximation;
26 else
27     truncate_part_len = f.point_pos_r - max;
28
29 max -= add_to_tail_precision;
30 //go to NO.`max' after `.' of f
31 /*
32     result (= x * y) ((precison of a by b )) :
33     (a) common case: (some non-'0' ahead)
34         0.02003004080509760000000001300xxxxxxxxxxxxxxxxxxx
35     (b) rare case: (all '0's ahead)
36         0.000000000000000000000000013000000050000xxxxxxxx
37         or 0.0000000000000000000000000130    (shorter vs. above)
38 */
39 int len = strlen(f.digits);
40 int int_len = len - f.point_pos_r;
41 char* p = approximation.digits;
42 bool is_there_nonzero = false;
43 for(i = 0; i < (int_len + max); i++)
44 {
45     if(p[i] != '0')
46     {
47         is_there_nonzero = true;
48         break;
49     }
50 }
51
52 if(is_there_nonzero)
53 {
54     approximation.point_pos_r -= truncate_part_len;
55     int len2 = len - truncate_part_len;
56     memset(&p[len2], 0, truncate_part_len);
57     p[len2] = '\0'; //clear again ( redundant though)
58
59     *p_pos = len2 - int_len + 1; // *p_pos = &p[len2] - &p[int_len] + 1;
60     return approximation; // (.)
61                             //531415
62                             //0123456
63                             //      ^
64 }
65 else //TODO: to log and check this rare/corner case.
66 {
67     for(i = (int_len + max); i < len; i++)
68     {
69         if(p[i] != '0')
70             break;
71     }
72     if(i == len)
73     {
74         floatII result("0");
75         return result;
76     }
77     else
78     {
79         if( (len - i - 1) <= add_to_tail_precision )
80             return approximation;
81         else
82         {
83             truncate_part_len = len - (i + 1 + add_to_tail_precision);
84             approximation.point_pos_r -= truncate_part_len;
85             memset(&p[i+1+add_to_tail_precision], 0, truncate_part_len);
86
87             *p_pos = i+1+add_to_tail_precision - int_len + 1;
88
89             return approximation;
90         }
91     }
92 }
93 }
94 }
95 }
96 }

```

The modification needed for the code that includes approximation:

```

for(i = 0; i < w; i++)
{
    y0 = above;
    for(j = 0; j < h; j++)
    {
        x=0;
        y=0;
        k = 0;
        while(k++ < MAX_ITERATION)
        {
            real = x*x - y*y + x0;
            imag = 2*x*y + y0;
            z_square = real*real + imag*imag;
            if(z_square > threshold)
                break;
            else{
                x = real;
                y = imag;
            }
        }
    }
}

```

f1/f11 实现

truncate_floatI/ II

truncate_floatI/ II

GMP 实现

GMP_truncate_real()

GMP_truncate_imag()

$\left. \begin{array}{l} \text{近似: real(approx)} \\ \text{近似: imag(approx)} \end{array} \right\}$

$\left. \begin{array}{l} -x = \text{real}; \quad x = \text{real(approx)}; \\ -y = \text{imag}; \quad y = \text{imag(approx)}; \end{array} \right\}$

```

    }
    x0 = x0 + delta;
}

```

The implementation of GMP truncating function in it: (taking GMP_truncate_real() as the example; the GMP_truncate_imag() code just replace "real" with "imag".)

```

mandel_with_mpfr_test_vs2015\mpfr_lib.c:
1  void GMP_truncate_real(char* str_numerator, char* str_denominator,
2      char* str_gmp_mpfr, int* p_exp, int mpfr_precision,
3      int len, int trunc_pos)
4  {
5      mpz_t  numerator;
6      mpz_t  denominator;
7
8      mpq_t  a;
9
10     mpfr_t  f1, f2;
11     mpfr_exp_t  exp;
12
13     mpz_init(numerator);
14     mpz_init(denominator);
15
16     mpq_init(a);
17
18
19     mpq_set(gmp_real_approx, gmp_real);
20
21     if(trunc_pos == 0)
22     {
23         //return;
24     }
25     else
26     {
27         int n;
28         char* str;
29         char* p;
30         mpz_set(denominator, mpq_denref(gmp_real_approx));
31         //gmp_printf("denominator is %Zd\n", denominator);
32         mpq_set_si(a, 10, 1);
33         n = 0;
34         while( mpz_cmp_si(denominator, 1) != 0)
35         {
36             mpq_mul(gmp_real_approx, gmp_real_approx, a);
37             //gmp_printf("%#040Qd\n", gmp_real_approx);
38
39             mpz_set(denominator, mpq_denref(gmp_real_approx));
40             n++;
41         }
42         //printf("\n multiplied by 10^ %d\n", n);
43         mpz_set(numerator, mpq_numref(gmp_real_approx));
44         //gmp_printf("numerator is %Zd\n", numerator);
45
46         str = (char*)malloc(len+16); //some more space
47         memset(str, 0, len+16); //buggy: memset(str, 0, sizeof(str));
48         mpz_get_str(str, 10, numerator);
49         //printf("str got:%s\n", str);
50         p = str + strlen(str);
51         p -= (n-trunc_pos)+1; //53.1415926
52         while( *p != '\0')
53             *p++ = '0';
54
55         mpq_set_str(gmp_real_approx, str, 10);
56         mpq_canonicalize(gmp_real_approx);
57         while(n > 0)
58         {
59             mpq_div(gmp_real_approx, gmp_real_approx, a);
60             //gmp_printf("%#040Qd\n", approx);
61             n--;
62         }
63
64         free(str);
65     }
66
67     mpz_set(numerator, mpq_numref(gmp_real_approx));
68     mpz_get_str(str_numerator, 10, numerator);
69
70     mpz_set(denominator, mpq_denref(gmp_real_approx));
71     mpz_get_str(str_denominator, 10, denominator);
72
73     //for visualizing the string
74     mpfr_init2(f1, (mpfr_prec_t)mpfr_precision);
75     mpfr_init2(f2, (mpfr_prec_t)mpfr_precision);
76
77     mpfr_set_z(f1, numerator, MPFR_RNDZ);
78     mpfr_set_z(f2, denominator, MPFR_RNDZ);
79     mpfr_div(f1, f1, f2, MPFR_RNDZ);
80     mpfr_get_str(str_gmp_mpfr, &exp, 10, 0, f1, MPFR_RNDZ);
81     *p_exp = (int)exp;
82
83     mpfr_clear(f1);
84     mpfr_clear(f2);
85     mpz_clear(numerator);
86     mpz_clear(denominator);
87
88     mpq_clear(a);
89
90
91 }

```

The logic of this implementation of truncation is such: since all of the fractions in my program are designed to be terminating decimals(not repeating decimals, needless to say irrational numbers), as to a calculated result which has the form like "53.1415926" etc., let it times 10 to the power of an incrementing integer variable(which will be equal to its decimal places) until the product becomes an integer (the rule for judgement is the denominator becomes 1), then go to the position to truncate as specified, set the digit itself and those after it to '0', use this new string to construct a new mpq_t type number, then divide this new number by 10 to the power of the corresponding number. Thus we accomplished the truncation of the calculated result.

In the function code, the highlighted line 30 obtained the denominator, line 32 is the multiplier 10 to be used, we can see the loop of multiplying by 10 and checking if the denominator becomes 1 at line 34 and line 36. Line 43 obtained the current numerator, the highlighted line 48 obtained the current string of the numerator. Line 50 to 53 set the part to be truncated of this string to all '0'. After constructing the new GMP fraction at line 55, we can see the highlighted line 57 and line 59 loop to divide the new number by 10 to the power n.

So far, we haven't looked at the addition and multiplication code yet. I thought it's only primary school arithmetic, below we'll take a rough look at it. Although on the above the GMP shows it is really powerful so maybe to use GMP(and MPFR) to do more computation will be a better choice later (because it allows people to use fractions, that is, repeating decimals, and other functions etc. that I myself haven't got), it might be better to have one more kind of implementation in parallel when available. However, if not available, I can just use GMP. Here I take the floatII class as the example, the implementation code of floatI class is longer, the floatII code is more concise, both of their algorithms are the same. There isn't very much to say about addition:

```
mandel_with_mpf_test_vs2015\my_floatII.cpp:
1 floatII floatII::operator+ ( floatII& f2)
2 {
3     //TODO: 0?
4
5     int sign, pos;
6
7     //align point
8     floatII f1 = *this;
9     int int_len1 = strlen(f1.digits) - f1.point_pos_r;
10    int int_len2 = strlen(f2.digits) - f2.point_pos_r;
11    int frac_len1 = f1.point_pos_r;
12    int frac_len2 = f2.point_pos_r;
13    int max_int_len = int_len1 >= int_len2 ? int_len1 : int_len2;
14    int max_frac_len = frac_len1 >= frac_len2 ? frac_len1 : frac_len2;
15
16    floatII f1_old = f1;
17    floatII f2_old = f2;
18
19    //pos
20    pos = max_frac_len;
21
22    int len = max_int_len + max_frac_len;
23
24    char* s1 = f1.extend_zero2( max_int_len-int_len1, max_frac_len-frac_len1);
25    char* s2 = f2.extend_zero2( max_int_len-int_len2, max_frac_len-frac_len2);
26
27    if(f1.minus == f2.minus)
28    {
29        //minus assign
30        sign = f1.minus? -1 : +1;
31
32        int* carry = (int*)malloc( (len+1) *sizeof(int));
33        char* r = (char*)malloc( len );
34        memset(carry, 0, (len+1)*sizeof(int));
35        char* q = s1+len-1;
36        char* p = s2+len-1;
37        int m;
38        int k = 0;
39        while(q >= s1)
40        {
41            m = chtoi(*q) + chtoi(*p) + carry[k];
42            if(m >= 10)
43            {
44                carry[k+1] += 1;
45                r[k] = itoch(m-10);
46            }
47            else
48            {
49                r[k] = itoch(m);
50            }
51
52            p--;
53            q--;
54            k++;
55        }
56
57        char* str = (char*)malloc(len+1+1); //doesn't care if more
58        int i = 0;
59        int j;
60        if(carry[k] > 0)
61            str[i++] = itoch(carry[k]);
62        for(j = 0; j < len; j++)
63            str[i++] = r[len-1-j];
64        str[i] = '\0';
65
66        floatII str_result(str, sign, pos);
67
68        free(str); //? do it in destructor?
69
70        free(r);
71        free(carry);
72        free(s1);
73        free(s2);
74
75        return str_result;
76    }
77    else
78    {
79        //to decide sign
80        char* p = s1;
81        char* q = s2;
82        while(*p != '\0') // && *q != '\0'
83        {
84            if(chtoi(*p) != chtoi(*q))
85                break;
86            p++;
87            q++;
88        }
89        if(*p == '\0')
90        {
91            free(s1);
92            free(s2);
93
94            floatII f4("0");
95            if(MY_DEBUG) printf("they are equal,minus result is 0\n");
96            return f4;
97        }
98        else
99        {
100            bool t = chtoi(*p) > chtoi(*q) ? f1.minus : f2.minus;
101            sign = t? -1 : +1;
102
103            char* big = (*p > *q) ? s1 : s2;
104            char* small = (*p > *q) ? s2 : s1;
105
106            p = big + len-1;
107            q = small + len-1;
108
109
110
```



```

110     int* borrow = (int* )malloc( (len) *sizeof(int));
111     int n;
112     char* r = (char* )malloc( (len));
113     memset(borrow, 0, (len)*sizeof(int));
114     int k = 0;
115
116     while(p >= big)
117     {
118         if( (ctoi(*p) + borrow[k]) >= ctoi(*q))
119         {
120             n = ctoi(*p) + borrow[k] - ctoi(*q) ;
121         }
122         else
123         {
124             char* l = p-1;
125             while( ctoi(*l) == 0 )
126             {
127                 *l = '9';
128                 l--;
129             }
130             borrow[k] = 10;
131             *l -= 1;
132             n = ctoi(*p) + borrow[k] - ctoi(*q) ;
133         }
134         r[k] = itoch(n);
135
136         p--;
137         q--;
138         k++;
139     }
140     //TODO: to remove leading 0s in integer part
141
142     char* str = (char* )malloc(len+1);
143     int i = 0;
144     int j;
145     for(j = 0; j < len; j++)
146         str[i++] = r[len-1-j];
147     str[i] = '\0';
148
149     floatII str_result2(str, sign, pos);
150
151     free(str);
152
153     free(r);
154     free(borrow);
155
156     free(s1);
157     free(s2);
158
159     return str_result2;
160 }
161 }
162 }

```

It prepares to align the decimal point beginning from the highlighted line 7. Line 20 obtained the position of the decimal point. Line 24 and 25 supplement '0's in an aligning style. Line 27 is about to do addition if the signs of the two numbers are the same. Its next line of code begins to do the addition, recording the carry value. Beginning from line 57 it prepares for the result string, which reads from left to right, while the addition itself is done from the lower to the higher, i.e., from right to left. Line 66 constructs the object of `floatII` class using the string and returns from the function. Line 84 is about to do subtraction for the signs of the two numbers are different. First it does comparison between their absolute values to decide which one is larger and what the sign of the difference will be. Line 94 shows that they are equal so the code returns 0. Line 104 is about to subtract the smaller absolute value from the larger one. Next it does subtraction, borrowing, etc. When done it likewise constructs the string, then constructs the object of `floatII` class. Thus done.

The multiplication costs two `""` operator overloading functions, however. The multiplication of two objects of the class is transformed to first doing multiplications of one object of the class and one integer variable (i.e., a number multiplied by every digit of another number), then summing them up after shifting their decimal points correspondingly. (For example, if we got to calculate $f_1 * f_2$, in which f_2 is of the form "xyzwr.st0", then the result would be

$$\begin{aligned}
 result &= f_1 * x * 10^m & (m = \text{max_int_len} - 1) \\
 &+ f_1 * y * 10^{m-1} \\
 &+ f_1 * z * 10^{m-2} \\
 &+ \dots \\
 &+ f_1 * t * 10^{-2} \\
 &+ f_1 * 0 * 10^{-3} & (-3 = -\text{max_frac_len})
 \end{aligned}$$

). Thus I used two operator overloading functions.

mandel_with_mpf_test_vs2015\my_floatII.cpp:

```

1  floatII floatII::operator* (int m) // 0<=m<=9
2  {
3      //TODO: 0?
4
5      int sign, pos;
6
7      floatII& f1 = *this;
8      sign = f1.minus? -1: +1; //does not care
9
10     //pos
11     pos = f1.point_pos_r; //doesn't change
12
13     int len = strlen(f1.digits);
14     char* s = f1.digits;
15
16     int* carry = (int* )malloc( (len+1) *sizeof(int));
17     char* r = (char* )malloc( (len));
18     memset(carry, 0, (len+1)*sizeof(int));
19     char* q = s+len-1;
20     int d;
21     int k = 0;
22     while(q >= s)
23     {
24         d = ctoi(*q) * m + carry[k];
25         if(d >= 10)
26         {
27             carry[k+1] += d/10;
28             d = d % 10;
29         }
30         r[k] = itoch(d);
31
32         q--;
33         k++;

```

```

34     }
35     //copied from add operation code part
36     char* str = (char* )malloc(len+1+1); //doesn't care if more
37     int i = 0;
38     int j;
39     if(carry[k] > 0)
40         str[i++] = itoch(carry[k]);
41     for(j = 0; j < len; j++)
42         str[i++] = r[len-1-j];
43     str[i] = '\0';
44
45     floatII str_result(str, sign, pos);
46
47     free(str); //? do it in destructor?
48     //copied from add operation code part -- end
49
50     free(r);
51     free(carry);
52
53     return str_result;
54 }
55
56
57 floatII floatII::operator* (const floatII& f2)
58 {
59     //TODO: 0?
60
61     int sign; // pos;
62
63     floatII& f1 = *this;
64     if(f1.minus == f2.minus)
65         sign = +1;
66     else
67         sign = -1;
68
69     char*p = f1.digits;
70     char*q = f2.digits;    //(choose the shorter to be multiplier?)
71     floatII multi("0");
72     int len2 = strlen(f2.digits);
73     for(int i = 0; i < len2; i++)
74     {
75         floatII t1 = f1 * chtoi(q[i]);
76
77         floatII t2 = shift_floatII(t1, len2 - f2.point_pos_r - 1 - i);
78         multi = multi + t2;
79     }
80
81     multi.minus = (sign==-1)? true : false;
82     return multi;
83 }
84
85
86 floatII& shift_floatII (floatII& f, int m)
87 {
88     if(m > 0)
89     {
90         if(m >= f.point_pos_r)
91         {
92             f.extend_zero(0, m - f.point_pos_r);
93             f.point_pos_r = 0;
94         }
95         else
96             f.point_pos_r -= m;
97     }
98     else if(m < 0)
99     {
100         int int_len = strlen(f.digits) - f.point_pos_r;
101         if(-m >= int_len)
102             f.extend_zero(-m-int_len+1, 0);
103
104         f.point_pos_r += -m;
105     }
106
107     return f;
108 }

```

The highlighted lines are those illustrated above. In the first function it does carrying similar to that in addition.

Having long been a c programmer up until now, I know very little about c++. It looks more natural using operator overloading in c++ to operate on the floating-point number class. However, it feels more portable for c libraries than c++ libraries when one is using GMP.

Once I found the memory usage of the third program soon went too high, and it kept increasing. After debugging I found it out that some pieces of code modifying "malloc/free" as "new/delete" led to this problem. This is not memory leak yet, for the memory is restored as soon as one closes the program. But under such memory usage the program will soon be not able to run. It seems the memory allocated through "new" has not been returned to the OS quickly after "delete", while "free" does. Since the program does large amount of computation persistently, no memory leak is crucial to the program.

It is rather good that the result of my implementation is consistent with GMP computation, using not very much code. GMP uses the fraction form, this enlarges the range of operation greatly compared to using decimal form by me. Surely it got a more complicated algorithm. The algorithm of mine is only for multiplication and addition, so it is just suitable for computing Mandelbrot set(I don't know if it is well suitable for computing Julia set.) and may not be enough for computation of other fractals.

Besides Win/VC, I would like to implement it on Linux/GTK.

I ever thought it was best to distribute the programs on ReactOS. But I tried ReactOS on VMWare and found its compatibility and stability were not suitable for actual use yet.

(P.S.: The precision in MPFR means binary precision, this may not easily match the decimal precision I adopted in my floating-point implementation.)