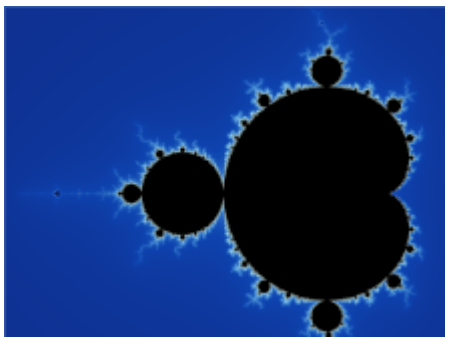
Mandelbrot

Project 1: Mandelbrot Fractal Zoomer

Part B DUE 9/30 11:59PM

Part A DUE 9/23 11:59PM

Autograder for partB will be released soon



Simpsons contributor at English Wikipedia [Public domain]

Setting up your computer for part A

Please accept the github classroom assignment by clicking this link. Once you've created your repository, you'll need to clone it to your instructional account. You'll also need to add the starter code:

```
$ git remote add starter https://github.com/61c-teach/fa19-proj1-starter
$ git pull starter master
```

If we publish changes to the starter code, you may get them by running `git pull starter master`

Please read the entire spec before beginning the project.

For this project, you can work locally, however your code is expected to run correctly on the Hive machines. In addition, some of the tests utilize software downloaded on the Hive machines, so be sure to test there often. If your code does not compile or execute on the Hive machines, you will get a 0. As a reminder, check Hivemind to see the traffic across the Hive machines.

Setting up your computer for part B

If you already set up your starter repo simply run this. The testing files take up some space, so you might need to wait a couple minutes to update your repo.

```
$ git pull starter master
```

We've compressed the files for the BigTest in PartB so that the overall size of the repository is smaller. When it's time for you to test PartB, you can unzip them on the hive using the following command from your proj1 root directory (you can copy it exactly into your terminal)

```
$ cd testing/BigTest && unzip BigTest0.zip && unzip BigTest1.zip && unzip
BigTest2.zip && unzip BigTest3.zip && unzip BigTest4.zip && unzip BigTest5.zip
```

I'd highly recommend **NOT using** `git add -A` or `git add .` while working on this project. Instead, use `git status` to display which `.c` files you've changed and **only add, commit, and push those**.

Inspiration

This is arguably (ref: Dan Garcia) the most beautiful, interesting recent discovery of mathematics, the Mandelbrot Fractal. Here are some videos describing why it's so amazing if you are interested:

- Michael Stevens description (of Vsauce fame)
- Ben Sparks (on Numberphile)
- Holly Krieger (on Numberphile)

Background

In this project, you will be implementing the Mandelbrot function and translating the result to an image! As your first major foray into the world of C programming, we hope to give you practice working with structs, pointers, memory, and more. The project will be split into 2 parts: 1) evaluating the Mandelbrot function given some inputs, and 2) creating images from the Mandelbrot output.

The core idea behind this project is the following function:

$$M(Z, C) = Z^2 + C$$

... and how this behaves if you start with Z as zero (i.e., 0 + 0i) and iterate on itself. (Z and C are assumed to be complex numbers.) That means after you get a value out, you stick it back in as Z and do it again, and again, etc. C always stays the same across iterations, though we are interested in how different C's affect how the function behaves. If the output stays bounded (the absolute value of the complex number stays less than an arbitrary threshold, which we choose to be 2), then it's in the Mandelbrot Set! If it isn't, then what's interesting to calculate is how many times does it needs to iterate until it surpasses the threshold.

Let's start with an example. Let's say $C = 1$. Now let's see how it behaves with $Z = 0 + 0i$ and iterate.

$M(0, C = 1) = 0^2 + 1 = 1$ Now, let's set that to be our new Z and keep going:

$M(1, C = 1) = 1^2 + 1 = 2$ Now, let's set that to be our new Z and keep going:

$M(2, C = 1) = 2^2 + 1 = 5$ At this point, the result is bigger than 2, and if we keep going it becomes 26 and even bigger!

So that means $C = 1$ is NOT in the set, since it blows up. If we were to write this more succinctly, we would say that the Z values for iterations for $C = 1$ starting with $Z = 0$ are the following: $0 -> 1 -> 2 -> 5$. So it went through 3 iterations (there were three -> arrows) until we got farther than 2 away from the origin. Let's try it with other values:

C	Pattern (values of Z as we iterate)	In Mandelbrot set? (i.e., bounded as we iterate forever?)	Iterations until farther than 2 and becomes unbounded	
-1	0-> -1 -> 0 -> …	yes	∞ (never happens)	
-½	0 -> -½ -> -¼ -> -0.4375 -> -0.30859375 -> -0.4047698974609375 -> -0.33616133010946214 -> -0.38699	yes	∞(never happens)	
0	0 -> 0 -> …	yes	∞(never happens)	
-1+i	0 -> -1+i -> -1-i->-1+3i	no	3	
1	0 -> 1 -> 2	no	2	

Like the halting problem, if we're iterating some large number of times and it stays bounded, we don't know whether it'll go outside the bounds on the next iteration or never go outside. Thus, in order to prevent infinite loops, we declare some cutoff max_iterations that we'll iterate up to; upon hitting max_iterations, we assume that the value stays bounded indefinitely. We can then visualize those values (the number of iterations) as colors. We usually reserve black for the color of the points that reach max_iterations before passing the threshold. These colors, when put together, become the beautiful images associated with the Mandelbrot function!

Part A:

Part A.1:

In this part, you are responsible for evaluating the Mandelbrot function. First, you should complete ComplexNumber.c: a file that looks familiar to a Java class file! Recall that in C, objects do not exist, so the closest data type a ComplexNumber can be is a struct with a real and imaginary field. You will be implementing getter functions and various arithmetic operations to make working with complex numbers easier. It may be helpful to review complex numbers to implement these correctly.

Similar to how interfaces in Java describe a class, we can use .h (header) files in C to describe a file and its associated variables and functions. Their existence allows functions written in one file to be called in another. Header files are standard when programming in C, so it is definitely worth your time to understand the ones provided in this project. ComplexNumber.h looks somewhat like this:

- ## ComplexNumber.h

  This file describes the struct `Complexnumber {double real, double imaginary}`.

```
typedef struct ComplexNumber ComplexNumber;

// Returns a pointer to a new Complex Number with the given real and imaginary
components
ComplexNumber* newComplexNumber(double real_component, double
imaginary_component);

// Returns a pointer to a new Complex Number equal to a*b
ComplexNumber* ComplexProduct(ComplexNumber* a, ComplexNumber* b);

// Returns a pointer to a new Complex Number equal to a+b
ComplexNumber* ComplexSum(ComplexNumber* a, ComplexNumber* b);

// Returns the absolute value of Complex Number a
double ComplexAbs(ComplexNumber* a);

// Frees the complex number
void freeComplexNumber(ComplexNumber* a);

// Gets the real component of the complex number
double Re(ComplexNumber* a);
// Gets the imaginary component of the complex number
double Im(ComplexNumber* a);
```

We WILL be enforcing the abstraction barrier by testing your code on a more precise version of ComplexNumber. In addition, we will be running memory tests to ensure no memory leaks, so be conscious of your memory use here. **Do not touch the code in test_complex_number.**

Part A.2

Once you have your ComplexNumber completed, you are now ready to evaluate the Mandelbrot function. Again, the header file looks like this:

- ## Mandelbrot.h

  This file contains the main coding part of part A:

```
/*
This function returns the number of iterations that cause the initial point to
exceed the threshold.
If the threshold is not exceeded after maxiters, the function returns 0.
*/
u_int64_t MandelbrotIterations(u_int64_t maxiters, ComplexNumber * point, double
threshold);

/*
This function calculates the Mandelbrot plot and stores the result in output.
The number of pixels in the image is resolution * 2 + 1 in one row/column. It's a
square image.
Scale is the the distance between center and the top pixel in one dimension.
*/
void Mandelbrot(double threshold, u_int64_t max_iterations, ComplexNumber*
center, double scale, u_int64_t resolution, u_int64_t * output);
```

(Note that we use u_int64_t instead of integers, so that we can use bigger numbers. u_int_64 stands for unsigned integer of 64 bits)

Mandelbrot should, given the above inputs, output the result of running the Mandelbrot function iteratively (which returns the number of iterations) over a grid of points (C values) into the output array which will later become the image. MandelbrotIterations is where the iterative evaluation of the function should take place without breaking the abstraction barrier of ComplexNumber!

A bit on the layout of the final image: the center-most pixel will contain the result of evaluating the Mandelbrot function (with given threshold and max iterations) with C as the input variable center and Z as 0. The bottom-left corner will evaluate the Mandelbrot function at C = (center - (scale + scale * i)), with other pixels linearly interpolated between the edge and the center. This is important when planning how to populate the output array.

This image should be outputted into the array output, which will be a preallocated (2 * resolution + 1)^2 size 1D array. The 2D “image” mentioned above should be stored in this array row by row.

For example, setting center to 5+3i, scale to 5, resolution to 2, (with the result stored in the given location in output) as follows, with M(Z, C) meaning the number of interactions of evaluating the Mandelbrot function with those initial Z and C:

output[0]: M(0, 0+8i)	output[1]: M(0, 2.5+8i)	output[2]: M(0, 5+8i)	output[3]: M(0, 7.5+8i)	output[4]: M(0, 10+8i)
output[5]: M(0, 0+5.5i)	output[6]: M(0, 2.5+5.5i)	output[7]: M(0, 5+5.5i)	output[8]: M(0, 7.5+5.5i)	output[9]: M(0, 10+5.5i)
output[10]: M(0, 0+3i)	output[11]: M(0, 2.5+3i)	output[12]: M(0, 5+3i)	output[13]: M(0, 7.5+3i)	output[14]: M(0, 10+3i)
output[15]: M(0, 0+0.5i)	output[16]: M(0, 2.5+0.5i)	output[17]: M(0, 5+0.5i)	output[18]: M(0, 7.5+0.5i)	output[19]: M(0, 10+0.5i)
output[20]: M(0, 0-2i)	output[21]: M(0, 2.5-2i)	output[22]: M(0, 5-2i)	output[23]: M(0, 7.5-2i)	output[24]: M(0, 10-2i)

Part A.3

The “parent” file which calls your Mandelbrot function is in MandelFrame.c. For this part, you do not need to understand how it works, though it may be useful in Part B. However, we have included a memory leak in this file. In order to pass the memory tests for part A, you **must** fix this leak in addition to ensuring there are no leaks in your written code. Other than this, there should be no edits made to this file.

- ## MandelFrame.c

  This file contains a main function, which receives command line arguments and converts them into the inputs for Mandelbrot.c.

Part B

One amazing part about the Mandelbrot set is that it is a fractal (geometrically self-similar); when you zoom in on it, it looks similar to the original shape. By using a color map from integers to colors and a steady zoom, you can get really cool patterns.

Part B.1

For this part, you will be in charge of handling the file input/output. As such, the first half of part B will focus on providing some practice with file I/O in C.

File I/O in C is handled through the FILE object, which are opened through the built-in function `FILE* fopen(const char* filename, const char* mode)`. `filename` should be the name of the file, and `mode` changes the permissions for that file. For example, setting `mode` to "r" makes it so that the file can be read, but can't be written. Once you finish using a file, be sure to close the file with `int fclose(FILE* file)`; this will free all memory associated with file, among other tasks needed to close the file. A list of all possible modes, as well as more details on file I/O, can be found at https://www.tutorialspoint.com/cprogramming/c_file_io.htm .

=======

Part B.1.1: Input

In ColorMapInput.c, implement the following function:

- ## FileToColorMap

```
uint8_t** FileToColorMap(char* colorfile, int* colorcount)
```

This function receives as input a file name containing a color map, and reads it. It then mallocs and outputs a 2D array containing the colors described in the color file.

The first line of colorfile will have the number of colors specified by Colorfile. Remaining lines will contain three integers between 0 and 255 separated by spaces, which corresponds to the RGB value of that color. For example, the following is a map that would yield a return value of `((255,0,0), (255,255,0), (0,255,0), (0,255,255), (0,0,255), (255,0,255))`:

```
6
255 0 0
255 255 0
0 255 0
0 255 255
0 0 255
255 0 255
```

For another example, look at defaultcolormap.txt.

In addition, the input colorcount is a blank integer pointer; when running this code, colorcount should be modified so that it stores the number of colors in Colorfile. This is because there is no easy way to find the length of an array in C; as such, this method needs to return the length of the color array as well as the array. Since C doesn't allow for multiple return values, we use an integer pointer to allow for this (alternatively, we could have used a struct).

If the colorfile is malformed (i.e. It doesn't fit the pattern specified above) or doesn't exist, this function should return a null pointer, and free any data it had already allocated. An exception to this rule is if the specified length of the colorfile is less than the actual number of colors in the colorfile; in that case, only the first n colors should be read and returned, where n is the specified length.

Also note that we are using a 2D array instead of a 1D array for this function, which is the same as an array of pointers.

For this part, we suggest use of the built-in function `int fscanf(FILE* ptr, const char *format, ...)`, which is designed to read lines from the file specified. Read more about it here provides a really good tutorial for fscanf, as well as all the customization that can be used with this function.

Part B.1.2: Output and the .ppm format

This section will allow for practice with file output, as well as the .ppm format which will be used Part B.2. For this part, we will be creating a color palette in both P3 and P6 format.

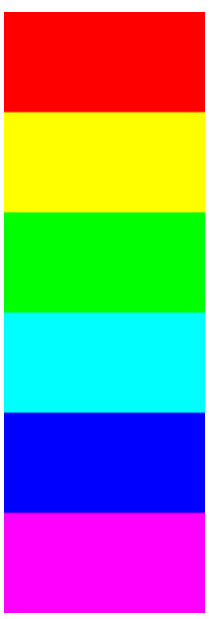
Implement the following functions within colorPalette.c:

- ## P3colorpalette

```
int P3colorpalette(char* colorfile, int width, int heightpercolor, char*
outputfile)
```

This function should create a .ppm P3 image in outputfile, with the following specifications:

- The image's horizontal dimension should be equal to `width`.
- The image's vertical dimension should be equal to `heightpercolor` * the length of the colorfile.
- The first `heightpercolor` rows should be colored entirely with the first color in colorfile. The next `heightpercolor` rows after that should be colored with the second color in colorfile, and so on. Thus, using the example colormap from FileToColorMap, with `width=4` and `heightpercolor=2` should yield the following image (zoomed up by a factor of 25):



If colorfile is invalid, width or heightpercolor is less than 1, or any other error occurs while running, this function should return with value 1 (after freeing everything); otherwise, this function should return 0. You may assume that outputfile is a valid .ppm file location.

The .ppm P3 format is very similar to the .pbm P1 format that was used for HW 2.5. Here are the specifications of the P3 format:

- The first line contains metadata for the image. Consider the example header "P3 4 12 255":
  - "P3" refers the file format. In this case, "P3" are color images.
  - "4" refers to the horizontal width of the image.
  - "12" refers to the vertical height of the image.
  - "255" refers to the maximum value of a pixel. Since we are using RGB colors, with each component going from 0 to 255, we will always set this to 255.
- The rest of the file consists of a number of lines, each of which specifies a single row in the image.
  - Each pixel is represented by three integers between 0 and 255 separated by spaces; together, they form the RGB value of that pixel. Pixels on the same row are separated by spaces, and rows are separated by new lines.

For example, the .ppm P3 format image of the above image is:

```
P3 4 12 255
255   0   0  255   0   0  255   0   0  255   0   0
255   0   0  255   0   0  255   0   0  255   0   0
255 255   0  255 255   0  255 255   0  255 255   0
255 255   0  255 255   0  255 255   0  255 255   0
  0 255   0    0 255   0    0 255   0    0 255   0
  0 255   0    0 255   0    0 255   0    0 255   0
  0 255 255    0 255 255    0 255 255    0 255 255
  0 255 255    0 255 255    0 255 255    0 255 255
  0   0 255    0   0 255    0   0 255    0   0 255
  0   0 255    0   0 255    0   0 255    0   0 255
255   0 255  255   0 255  255   0 255  255   0 255
255   0 255  255   0 255  255   0 255  255   0 255
```

Note that we added variable spacing in this example to be clearer; in reality, the .ppm format doesn't specify any rules regarding the number of whitespace characters between pixels. In order to autograde this, we will require that your output separates numbers in the same line with exactly one space, and no space at the end of the line. As such, the first line of the above would be:

```
255 0 0 255 0 0 255 0 0 255 0 0
```

Be sure to open your file with the correct mode, and to close the file when you finish. For outputting to this file, the function `int fprintf(FILE* file, const char* format, ...)` works really well; this function essentially works the same way as `printf`, but outputs data to the specified file instead of to the terminal. Make sure you use the correct format specifier; putting in a char sized object when printf expects an int will sign extend the value, so `printf("%d", (char) 0xFF)` will interpret this as `printf("%d", 0xFFFFFFFF)`, which will output -1. http://man7.org/linux/man-pages/man3/printf.3.html provides a good resource on the available format specifiers.

• P6colorpalette

```
int P6colorpalette(char* colorfile, int width, int heightpercolor, char*
outputfile)
```

This function works the same way as P3colorpalette, but this time, we want the output to be in .ppm P6 format. Here are the specifications of P6:

- Like in P3 format, the first line contains metadata. The only difference between P6 format and P3 format on this line is that instead of writing "P3", we write "P6".
- After the header, the rest of the file consists of the image data. Instead of writing a number between 0 and 255 in human readable format for each component of a pixel, we instead write the value in binary. Thus, each component of a pixel turns into one character. Since the header already specifies the image size and each pixel is written with the same number of characters, there is no need for any whitespace between pixels nor newlines between rows, so the P6 format does not have them.

For example, the .ppm P6 format image of the above image is:

```
P6 4 12 255
ÿ\0\0ÿ\0\0ÿ\0\0ÿ\0\0ÿ\0\0ÿ\0\0ÿ\0\0ÿ\0\0ÿ\0\0ÿÿ\0ÿÿ\0ÿÿ\0ÿÿ\0ÿÿ\0ÿÿ\0ÿÿ\0ÿÿ\0ÿÿ\0\0ÿ\0\0ÿ\0\
```

Note that ÿ is the character that corresponds to 255, and \0 is being used to represent to character that corresponds to 0.

Instead of using fprintf for this, the function `size_t fwrite(const void* ptr, size_t size, size_t nmemb, FILE* file)` is better suited for binary output. The usage of this slightly differs from that of printf; https://www.tutorialspoint.com/c_standard_library/c_function_fwrite.htm provides a good tutorial on the usage of this function.

When you finish both P3colorpalette and P6colorpalette, try and run both programs with the same parameters and different output files. You should notice that the P6 version takes up much less memory.

Part B.2: Back to the Mandelbrot Set

Implement the following function within MandelMovie.c:

- ## MandelMovie

```
void MandelMovie(double threshold,
u_int64_t max_iterations, Complexnumber* center,
double initialscale, double finalscale,
int framecount, u_int64_t resolution,
u_int64_t** output)
```

The goal of this function is to generate a sequence of Mandelbrot iteration images, so that we can make a video out of zooming into one spot on the Mandelbrot fractal.

`output` is an array of length `num_frames`, each pointing to an appropriate-sized int array. Each element of `output` should be the result of running Mandelbrot; each frame should have `threshold`, `max_iterations`, `center`, and `resolution` identical to the variables passed into MandelMovie (so everything except for the `scale` stays constant through the movie we're generating)

The `scale` of the first frame should be `initialscale`, and the scale of the last frame should be `finalscale` (inclusive on both ends). The scales of intermediate frames form a geometric sequence (that is, $\frac{\text{scale(frame } n)}{\text{scale(frame } (n-1))} = \frac{\text{scale(frame } (n+1))}{\text{scale(frame } n)}$. You may find the `log` and `exp` or the `pow` function useful for generating the correct scales.

This part should not be very long; our solution uses two lines of code inside a `for` loop.

Part B.3: Crescendo

Everything we've made so far, brought together in one final function.

Implement the following function within MandelMovie.c:

- ## main

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

Inputs are provided on command line; read through the provided function printUsage for the expected sequence of inputs. Most inputs correspond to the ones in void MandelMovie. The ones which don't are:

- outputfolder: This is a folder in which you will store the output images.
- colorfile: This is a file containing information on what colors to use for the output.

The goal of this main function is to run MandelMovie with the given command line arguments, convert each iteration image into a color image using the mapping provided in colorfile, and output each color image into files in outputfolder.

We suggest using the following general steps for this function:

1. Check inputs to confirm that they are valid, then convert them into variables.
2. Run MandelMovie.
3. Convert from iteration count to colors, and output the results into output files.

Your code should return early with return value 1 in the event that the input is incorrect. This includes when, but is not limited to:

1. The number of arguments provided does not match the needed number.

2. threshold, maxiterations, or scale is 0 or less.
3. Insufficient memory for output. (Because of this, you should allocate enough memory for output in your main function).
4. The number of frames exceeds 10,000, or is 0 or less.
5. resolution is less than 0.
6. colorfile is not properly formatted, as specified in the above description of colorfile.
7. The number of frames is 1 AND initialscale != finalscale.

Note that the following are allowed:

1. initialscale $\leq$ finalscale. In this case, the image will zoom out or stay constant.
2. The number of frames is 1, and initialscale == finalscale

Outputs should be made in .ppm P6 (NOT P3) file format, with image names frame00000.ppm for the first frame, frame00001.ppm for the second frame, and so on. The function `sprintf` may come in handy here; sprintf works the same as printf and fprintf, but instead outputs into a string.

- It is recommended to use the main function found in MandelFrame.c as a starting point, since it follows a similar procedure to run Mandelbrot and store the output.

Pixels with iteration value 0 will be colored with black (RGB value 0,0,0). Pixels with iteration value 1 through `len(colormap)` will be colored with their corresponding colormap color. For pixels with greater values, the colors used will loop; thus, if colormap = `((255,0,0), (255,255,0), (0,255,0), (0,255,255), (0,0,255), (255,0,255))`, pixels with value 1, 7, 13, ... will be colored red (RGB value 255,0,0), pixels with value 2,8,14, ... will be colored yellow (RGB value 255,255,0), etc.

Lastly, we provided a method convert from your sequence of ppm files into an animation

In order to generate the video, run `ffmpeg -framerate 24 -i student_output/partB/frame%05d.ppm output.mp4` in your project folder. In order to actually view the movie file, we recommend you `git push` and download the video to your own computer so you can open the video with whatever video software you have. Windows computers may have difficulties opening .mp4 files; in this case, you can view your video by uploading a private or unlisted video to Youtube.

Testing & Running Your Code

Part A

Run `make testA` to run a small case for part A, and run `make mem-checkA` to check for memory leaks.

`make testASimple` runs a small test (threshold 2, center 5+3i, scale 5, resolution 2, similar to the example table above), which is useful for debugging.

Unless you have Valgrind installed locally, you need to run the memory check on the Hive machines.

For debugging you can use CGDB on Mandelframe. You will need to set the initial args. Use `set <args>`

Part B.1

`make testB1Small` will run a small case for part B.1, and `make testB1Big` will run a normal case for part B.1. We also provide a memory check by running `make memcheckB1`.

Part B.3

Run `make testB2Small` to run a small case for part B, and run `make mem-checkB2Small` to check for memory leaks.

Run `make testB2` to run an average case for part B, and run `make mem-checkB2` to check for memory leaks.

After you've unzipped the tests (if you pulled PartB after 9pm on 9/19), `make BigTest` will run a giant test of part B. It is suggested to run this on the hive computer, as this test may take up to an hour to run.

Note that we do not have tests for B.2.

Windows computers don't have a method to view .ppm files, so it is suggested that you use internet resources for this. https://paulcuth.me.uk/netpbm-viewer/ , for example, is a good tool that can be used to help debug.

For Macs, the default image viewer should have a way to view PPM Files.

Note that a majority of our Gradescope tests will be hidden, and not provided until after the deadline. Be sure to check for edge cases (ex. Really low threshold, resolution = 0, using freeComplexNumber to free complex numbers).

A note on testing:

Small differences in computation result (such as those introduced due to floating point error) can lead to significant differences in the complex number over hundreds of iterations; as a result, there might be slight differences in your output depending on implementation, operating system, or even when you run your code. Due to this, we will consider correct any solution whose output matches the reference in 99.9% of all pixels. We have provided the python script to check the similarity between files or folders. Run `python verify.py <reference file/folder> <output file/folder>` to calculate this score, in case you wish to debug further. `Make test` automatically runs `verify.py` for all tests we provide.

Note that a majority of our Gradescope tests will be hidden, and not provided until after the deadline. Be sure to check for edge cases (ex. Really low threshold, resolution = 0, using freeComplexNumber to free complex numbers).

A few tips on reducing floating point error:

- Try to make it so that each calculation is done independently, and isn't dependent on a previous calculation. In general, the more operations you do in series, the greater the potential error (You can see this for yourself with a drawing exercise. Try drawing a seven-pointed star without lifting your pencil from the paper. Now try drawing a seven-pointed star by first picking seven points in a regular heptagon shape, then connecting them with lines. The second one should be more symmetrical.) This is especially true for floating point multiplication.
- Don't use pow if you can avoid it; this is both slower and less accurate than sqrt and simple multiplication, because pow has to work for arbitrary exponent.

Submitting your work

Submission will be done via Gradescope. The initial submission will run a few simple tests, checking for the validity of files. The majority of tests will be hidden, and will be run only after the deadline. Thus, it is recommended to test your code heavily.

When testing your code, we will only be interfacing with your code via the main function of MandelFrame, the test function in ComplexNumber, and the function Mandelbrot in Mandelbrot.c. As such, you may choose to modify any other functions from their current specifications. You may also declare any additional helper functions you desire, but they should stay within Mandelbrot. Besides the memory leak fix, you should not modify MandelFrame.

Extra Credit

We will be posting a thread on Piazza for an art gallery of images obtained from your programs. Experiment with varying threshold, color maps, center, zoom level, etc. If you want to be really ambitious, you may even choose to modify the design choices we specified (ex. Instead of linearly interpolating from center to edge, what would it look like with an exponential interpolation). Videos may take a long time to generate, so be reasonable with hive machine usage if you wish to create long videos. Here is an example video created by staff (the same file as in BigTest, 1.5 hours runtime on hive machine):

We will be offering 1 extra credit point to the three best submissions by vote.

CS 61C Calendar Staff Policies Piazza Venus Resources Semesters Back to top