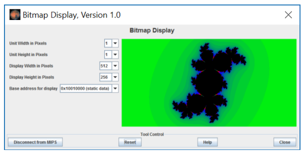
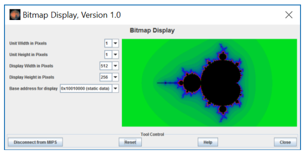
COMP 273 Fall 2016 Assignment 3
Julia and Mandelbrot sets in MIPS

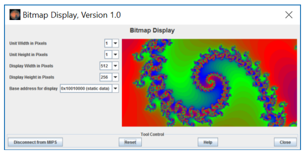
Due: 11:59 pm, Wednesday 14 March 2023



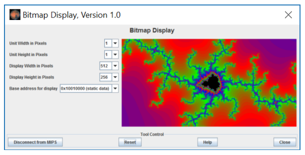
Unit Width in Pixels: 1
Unit Height in Pixels: 1
Display Width in Pixels: 512
Display Height in Pixels: 256
Base address for display: 0x10010000 (static data)



Unit Width in Pixels: 1
Unit Height in Pixels: 1
Display Width in Pixels: 512
Display Height in Pixels: 256
Base address for display: 0x10010000 (static data)



Unit Width in Pixels: 1
Unit Height in Pixels: 1
Display Width in Pixels: 512
Display Height in Pixels: 256
Base address for display: 0x10010000 (static data)



Unit Width in Pixels: 1
Unit Height in Pixels: 1
Display Width in Pixels: 512
Display Height in Pixels: 256
Base address for display: 0x10010000 (static data)

1 Introduction

The term *fractal* was introduced by French mathematician Benoît Mandelbrot in the mid-1970s to refer to strange and beautiful mathematical sets that have similar structures at all scales. In this assignment you will draw images of fractals using a memory mapped display within MARS.

While there are many types of fractals, some of the most widely known are related to the behaviour of a given function when the results of the function are repeatedly given back to the function as arguments. This process is known as iteration. Given a function $f(x)$, the iteration of a starting value x first produces $f(x)$, then $f(f(x))$, then $f(f(f(x)))$, then $f(f(f(f(x))))$, and so on.

Consider the simple one dimensional example, $f(x) = 1 - x^2$, and let us start with $x_0 = 0.5$. After one iteration of the function we get $x_1 = 0.75$, then $x_2 = 0.4375$, and so on. If we start with the value $x_0 = 0.5$, the generated sequence will always remain between 0 and 1. In fact this happens if we start with any number x_0 in the interval $[0, 1]$ because when we square a number between 0 and 1 we always get another number between 0 and 1 (likewise, subtracting this number from 1 also produces a result between 0 and 1). In contrast, if we start with $x_0 = 2$, the sequence is $\{2, -3, -8, -63, -3968, ...\}$.

This simple example shows that for some starting values the numbers in the sequence get very big (that is, the sequence grows without bound), while for others the sequence remains bounded. *We can thus define a set:* the set of values that have the property that the sequence starting at any value in the set and produced by iterating the given function does not grow without bound.

This is exactly the principle on which the well known Julia sets are based, except that rather than having a one dimensional function, we have a function which maps two dimensional coordinates of the plane back onto the plane (the plane in this case is actually the complex plane). Specifically, this function is

$$f(x, y) = (x^2 - y^2 + a,\ 2xy + b), \tag{1}$$

where a and b are constants. For example, with $(a, b) = (0.25, 0.5)$, consider the initial point $(0, 0)$.

$$\begin{aligned}
(x_0, y_0) &= (0, 0) \\
(x_1, y_1) &= (0^2 - 0^2 + 0.25,\ 2 \times 0 \times 0 + 0.5) \\
&= (0.25, 0.5) \\
(x_2, y_2) &= (0.25^2 - 0.5^2 + 0.25,\ 2 \times 0.25 \times 0.5 + 0.5), \\
&= (0.0625, 0.75) \\
(x_3, y_3) &= (-0.30859375, 0.59375) \\
(x_4, y_4) &= (-0.0073089599609375, 0.133544921875) \\
(x_5, y_5) &= (0.23221917473711073, 0.4980478510260582)
\end{aligned}$$

$$\begin{aligned}
(x_6, y_6) &= (0.0558740832039101, 0.7313125218897254)\\
(x_7, y_7) &= (-0.2816960914988327, 0.5817228333922557)\\
(x_8, y_8) &= (-0.009048766924195428, 0.1722619029955499)\\
(x_9, y_9) &= \ldots
\end{aligned}$$

Now let us try a different example. Consider the starting point $(x_0, y_0) = (1, 0)$.

$$\begin{aligned}
(x_0, y_0) &= (1, 0)\\
(x_1, y_1) &= (1^2 - 0^2 + 0.25,\ 2 \times 1 \times 0 + 0.5)\\
&= (1.25, 0.5)\\
(x_2, y_2) &= (1.25^2 - 0.5^2 + 0.25,\ 2 \times 1.25 \times 0.5 + 0.5),\\
&= (1.5625, 1.75)\\
(x_3, y_3) &= (-0.37109375, 5.96875)\\
(x_4, y_4) &= (-35.23826599121094, -3.929931640625)\\
(x_5, y_5) &= (1226.5410273673479, 277.4679529592395)\\
(x_6, y_6) &= (1427414.6768959584, 680652.1561682811)\\
(x_7, y_7) &= \ldots
\end{aligned}$$

Notice that with this starting point the sequence grows without bound.

For a given (a, b), the corresponding Julia set contains the points (x, y) that do not grow without bound upon iteration of Equation 1. This is the black region in the left most region at the top of the image, while the coloured regions show points that do grow without bound, and the colour gives an indication of how fast. In contrast to the definition of a Julia set, the Mandelbrot set is the set of points (a, b) in the plane for which the corresponding Julia set function does not exhibit unbounded growth when iterated starting at (0,0). Finally, notice that Equation 1 is very simple when written with complex numbers. For a given complex constant $c = a + bi$, and starting point $z = x + yi$, we iterate the function $z^2 + c$.

2 Before getting started

This assignment is organized in a manner that should help you make progress toward the final objective while testing small parts of your code along the way. However, there is also some important details and advice you should review in this section before getting started.

2.1 Float register conventions

It is important to implement the assignment as specified and follow register conventions for grading. Use the following conventions for the floating point registers.

`$f0 $f1 $f2 $f3`	Function return values
`$f12 $f13 $f14 $f15`	Function arguments
`$f4` through `$f18`	Temporary registers
`$f20` through `$f31`	Save registers

Note that the floating point save registers are the only registers you can assume are preserved across function calls. In your functions, you must save these registers to the stack with a `swc1` instruction before using them, and restore them before returning. The standard convention for the argument registers is to only use `$f12` and `$f14` when passing float values, which is likely to simplify assigning argument registers to functions when there is a mix of double and single precision arguments. However, you will **only use single precision floating point in this assignment**, thus, we will bend this rule and allow the use of all four float arguments. If a function takes both int and float arguments, you should simply assign them in order. For instance, in the function `f(int a, float b, int c, float d, float e)` you should assume that integer `a` is in `$a0`, float `b` is in `$f12`, integer `c` is in `$a1`, float `d` is in `$f13`, and float `e` is in `$f14`.

2.2 MIPS coding advice and avoiding pitfalls

Write good comments in your code to help you keep track of what you are trying to accomplish. This will help you create your solution, and will help the TAs better understand your code. You might find it useful to keep a number or references close at hand when working on your solution: the register conventions, the SYSCALL reference in the MARS help, the documentation of instructions on the green quick reference sheet in the textbook, and the definitive instruction reference in the appendix of the textbook (e.g., Appendix B10 of the 4th edition). Note that MARS may occasionally get stuck in a bad state and run slowly, but this is easily resolved by restarting MARS. Finally, here is a bit of advice to help you avoid some common pitfalls.

- Watch for errors in register usage when cutting and pasting or in code that is doing similar operations to different variables. For instance, repeating a register in wrong places, which can happen when you want to do something to both x and y, and instead you do it to x and x.
- Careful not to forget or confuse the definition of an instruction. It is easy enough to confuse `move a b` with `move b a`, given that the copy direction is different for other instructions, for instance, consider what gets modified when invoking `mtc1`, `mfc1`, `lw` and `sw`.
- Careful if you copy and paste code that saves registers on the stack from the beginning of your function to the end of your function. A common error is to forget to change the `sw` instructions to `lw` in the pasted code.
- Be careful to decrease and increase your stack pointer by the same amount when writing the start and end of your functions.
- Always write function headers to show where your procedures start, and what they do. That is, you will also have labels for both internal branches and loops, so it helps if the procedure entry labels have greater visibility.
- Choose good names for labels within a procedure, that is, for your conditional code and loops. Moreover, give them a prefix to identify the current function. For example, if you want a `loop` label in functions `f1` and `f2`, then `f1Loop` and `f2Loop` would make good unique labels.

3 Assignment objectives (2^4+i marks total)

You will write a program to compute iterations of the function defined above, by first writing some helper functions. Once your are able to compute iterations, you will write some additional helper functions, and eventually draw the structure of different Julia sets, and the Mandelbrot set. Use single precision floating point numbers for all your calculations. Please follow function specifications exactly and implement label names as requested to allow your code to be tested during marking.

1. **printComplex and printNewLine** (2 marks)

   Implement a function `printComplex` that takes two float arguments, `$f12` and `$f13`, and prints them to the RunIO console as a complex number using `SYSCALL`s. For instance, if you make the call `printComplex(1.1, 2.3)` then you should see "`1.1 + 2.3 i`". Implement another function called `printNewLine` to print a newline character `"\n"`. Use the `.asciiz` directive in your `.data` segment to define the null terminated ASCII strings you need for both functions. Test your functions by adding code at the top of the `.text` section that uses your functions to print the complex number stored as a pair of floats stored at label `JuliaC1`. Note we will always store the real part in the lower register number or the lower memory address, while the imaginary part will be stored in the higher register number of higher memory address.

2. **multComplex** (2 marks)

   Implement function with signature

   ```
   (float, float) = multComplex( float a, float b, float c, float d )
   ```

   that computes the complex product $(a+bi)(c+di) = (ac-db)+(ad+bc)i$. Following the register conventions as described in the previous section, the real and complex parts of the product will go into *return* registers `$f0` and `$f1`. Test your function by adding code to your `.text` segment that computes the square of the complex number stored at label `JuliaC1` and prints the result. Did you get the correct result?

3. **iterateVerbose** (4 marks)

   Implement a verbose iteration function with the following signature.

   ```
   int iterateVerbose( int n, float a, float b, float x0, float y0 )
   ```

   This function should compute the first n iterations of Equation 1 using the the starting point (x_0, y_0). Your implementation should use your `multComplex` function. Your program should check if the squared magnitude of the current iteration exceeds the global constant defined at label `bound`, and if you discover that `bound` $< x^2 + y^2$ then stop iterating and return the number of iterations that were below the bound.

   Because this is the *verbose* implementation, you should print all the iterations as you go, and likewise, print the return value iteration count before returning.

   Add code to the main `.text` segment to test function. For example, calling `iterateVerbose( 10, 0.25, 0.5, 0.0, 0.0 )` will return 10 in `$v0`, and generate the following output.

   ```
   x0 + y0 i = 0.0 + 0.0 i
   x1 + y1 i = 0.25 + 0.5 i
   x2 + y2 i = 0.0625 + 0.75 i
   x3 + y3 i = -0.30859375 + 0.59375 i
   x4 + y4 i = -0.00730896 + 0.13354492 i
   x5 + y5 i = 0.23221917 + 0.49804786 i
   x6 + y6 i = 0.05587408 + 0.7313125 i
   x7 + y7 i = -0.28169608 + 0.58172286 i
   x8 + y8 i = -0.00904879 + 0.1722619 i
   x9 + y9 i = 0.22040772 + 0.49688247 i
   10
   ```

   In another example, calling `iterateVerbose( 10, 0.25, 0.5, 1.0, 0.0 )` will return 3 and produce the following output.

   ```
   x0 + y0 i = 1.0 + 0.0 i
   x1 + y1 i = 1.25 + 0.5 i
   x2 + y2 i = 1.5625 + 1.75 i
   x3 + y3 i = -0.37109375 + 5.96875 i
   3
   ```

Remember that it is possible that your output may not match *exactly* because floating point computations are approximate.

4. **iterate** (i marks)

   Create a function called `iterate` that is identical to `iterateVerbose`, except that it does not print anything to the RunIO console. Because this is so easy given the previous step, this step is only worth one imaginary mark.

5. **pixel2ComplexInWindow** (2 marks)

   To draw Julia and Mandelbrot sets, we require a way of mapping a region of the complex plane to the array of pixels that make up the bitmap display. Create a function with the signature

   ```
   ( float x, float y ) = pixel2ComplexInWindow( int col, int row )
   ```

   that takes an integer pixel location (col, row) where col is the column and row is the row within the bitmap display. The bitmap width w and height h are provided as constants in the static `.data` segment at the label `resolution`. The return value, a complex number $x + yi$, is computed using the window defined at label `windowlrbt`, where the letters `lrbt` refers to, in order, the left, right, bottom, and top coordinates. Left and right provide the real-axis range, while bottom top provide the imaginary-axis range. Ultimately, your function should compute

   $$x = \frac{col}{w}(r - l) + l \, ,$$
   $$y = \frac{row}{h}(t - b) + b \, .$$

   For integer pixel coordinates and resolution width and height, use `mtc1` and `cvt.s.w` to move them to the coprocessor and to cast them to floats so that you can use them in your computation.

   Note that we ignore a small problem in the equations above, which is that the upper left of the bitmap display corresponds to $(col, row) = (0, 0)$, but this will map to the lower left corner of the complex window defined by `windowlrbt` (this is not a big issue with fractals because we are not likely to notice the image is upside down). Keep this in mind when you test your function. Test your function for $(0, 0)$, $(256, 128)$, and $(512, 256)$, printing the results to the console, and make sure that you get the lower-left, middle, and upper-right, as expected, based on `windowlrbt`.

6. **drawJulia** (4 marks)

   Implement a function `drawJulia( float a, float b )`, which receives the complex constant $a+bi$ in two float arguments. The function should loop over every integer pixel location (col, row) in the bitmap display, and compute a starting point for iteration using `pixel2ComplexInWindow`. Given the complex constant and starting point, use `iterate` to check if the staring point grows without bounds. Load the static constant `maxIter` into the n parameter for the `iterate` function.

   If the `iterate` function returns before the maximum number of iterations, then you will know that the bound was reached, and that the point is *not* in the Julia set. In this case, compute a colour by calling the provided function `computeColour( iterations )`. The return value is an `ARGB` encoded word that slowly changes between green, blue, and red in a loop as the iterations increase. The `scale` static data constant allows the speed of colour changes to be tuned (feel free to experiment, but the default 16 should work fine). If the maximum number of iterations is reached, you should instead choose the colour black (zero). To set the pixel at location (col, row) in the bitmap display of resolution w by h you write a word to the location

   $$\texttt{bitmapDisplay} + 4(w \times row + col). \qquad (2)$$

   However, since you are visiting each pixel in the display in order, you will probably organize your code to simply add 4 to a pointer to step between pixels.

Note that lowering the maximum iterations constant will make your program run faster, but will result in lower quality images (missing details at the set boundary). Also note that by default, the resolution is set to the bitmap display tool default of 512 wide by 256 high, but feel free to edit these and adjust the tool settings to draw smaller images when debugging.

Test your function with various constants. Note that the there are several complex window regions defined, but commented out. Note the comments that identify the best window for viewing a typical Julia set. See the following link for examples of what you should expect to see.

`http://en.wikipedia.org/wiki/Julia_set#Quadratic_polynomials`

7. **drawMandelbrot** (2 marks)

   The Mandelbrot set is the set of points (a, b) in the plane for which the corresponding Julia set function does not exhibit unbounded growth when iterated with the starting point $(0, 0)$. Write a function `drawMandelbrot()` to draw the Mandelbrot set in the bitmap display in a manner similar to drawJulia. That is, use the defined `resolution` of the bitmap display, and let the region of the complex plane be defined by `windowlrbt`. However, for a Mandelbrot set you will now compute constants (a, b) with `pixel2ComplexInWindow` and iterate starting with $(0, 0)$.

   Several interesting window regions are defined for viewing different parts of the Mandelbrot set. Test your code by commenting out all but the desired window region and adding a `jal drawMandelbrot` to your main `.text` segment. Finally note that if you are using a zoomed window region, you may want to increase the number of iterations to better identify points that are not in the set! In all the windows provided, a maximum iteration count of 512 should probably be more than enough.

8. **Bonus** (4 marks)

   This bonus is not for the faint of heart, as there is already an ample number of challenges to complete the assignment steps above. However, if you really have the time and curiosity, write the following three functions.

   ```
   ( float p, float q ) = complexSqrt( float a, float b )
   plotComplex( float x, float y )
   juliaReverse( int n, float a, float b, float x0, float y0 )
   ```

   The function `complexSqrt` computes the square root. The function `plotComplex` takes a point (x, y) in the complex plane, and uses `windowlrbt` and `resolution` to compute a pixel location, and provided that it is in bounds of the bitmap display, sets the pixel to the colour of your choice (choose wisely to contrast the colours generated by `computeColour`). Given an arbitrary starting point, `juliaReverse` computes $\sqrt{x + yi - (a + bi)}$ at each iteration, choosing randomly the positive or negative square root on each iteration (see SYSCALL for random number generation), and drawing the point with `plotComplex`. The starting point can be arbitrary (e.g., zero), and subsequent iterations will quicly converge to the boundary of the Julia set. You can use the function to apply frosting to the tips of your Julia sets by adding `juliaReverse` as a second drawing pass after calling `drawJulia`.

What to Submit

Submit your `.asm` file through MyCourses. Be sure you have your name and student number in a comment at the top of your file. Be sure to check that your submission is correctly submitted by downloading and examining your submission from MyCourses. You cannot receive any marks for assignments with missing or corrupt files. Note that you are encouraged to discuss assignments with your classmates, but not to the point of sharing code and answers. All code and written answers must be your own.