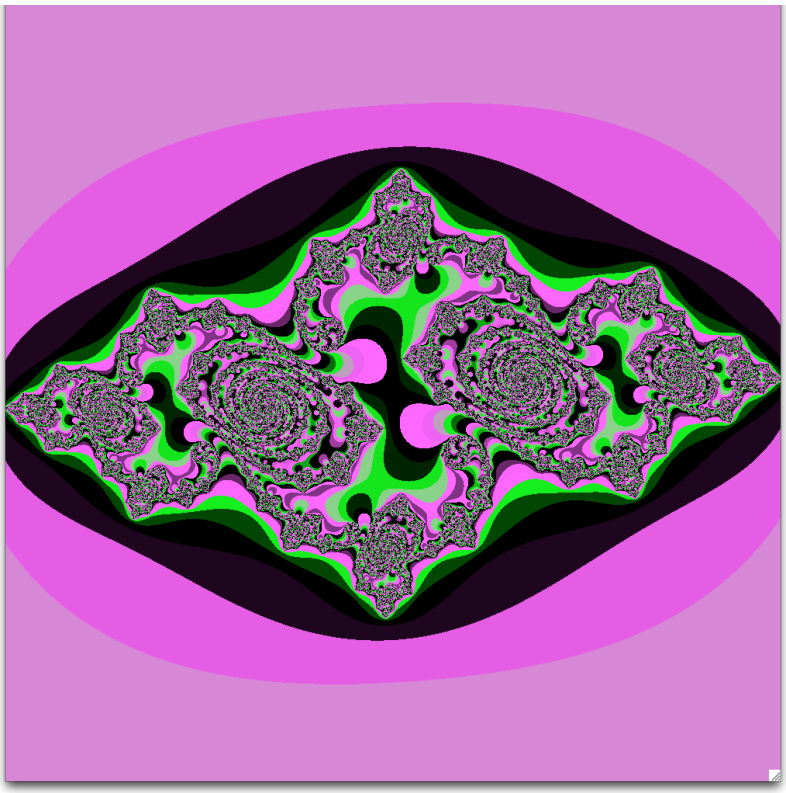
Julia Sets

Unknown macro: {style}

cite

Unknown macro: {font-family}

Project Overview



Given a function such as f(x) = x^2 + c, you can iterate it as many times as you like by computing f(f(...f(f(x))...)). In general, inserting a real number into such an equation will result in the iteration growing towards infinity. If c=0 and x<1, though, it will shrink to zero. In fact, when c=0 and x=1, iterating the function does not change the value at all!

When you generalize this to allow for complex numbers, then the behavior becomes more interesting. A given value for c may have several *attractors*, such as zero and infinity in the example above. Sometimes iterating a certain value enters into an orbit – perhaps it rotates between a finite number of points. Usually nearby points have the same behavior. The boundaries between regions with different behavior compose the Julia set of the function.

Recall that complex numbers, written x + iy, can be thought of as coordinates on a two-dimensional plane, such as an image. Using this relationship, we can visualize the Julia set of a function by coloring pixels (which correspond to a sampled complex number) based on their behavior over iterations in the function.

It is difficult (though by no means impossible) to detect orbiting points, but it is quite simple to tell if a value is increasing without bound.

Lisp's functionality provides convenient aspects to visualize Julia sets in this way. Our program is divided into two files: one that contains a function that writes a ppm file, and one that creates a multidimensional list representation of the image that a function generates. The files are linked by a *—load* command at the top of the file-writer which imports the functions from the Julia set file.

Writing an Image File

In order to display a Julia set, we first need a way to output an image. The file format ppm is simple to create. After a header which specifies a few parameters (including the dimensions of the image), you just have to write the rgb values in bytecode. We open a stream with *—with-open-file*, which is described in Input&Output. Here we open it with a specified *—element-type* of *—unsigned-byte 8* since we need to write bytecode.

Unknown macro: {table}

Unknown macro: {tr}

Unknown macro: {td}

```
(defun write-ppm(filename colors cols rows)
  ; open file st
  (with-open-file (st filename
  :element-type '(unsigned-byte 8)
  :direction :output
  :if-exists :supersede
  :if-does-not-exist :create)
      ; P6 means colors are in byte-code (P3 for ASCII)
      ; \~a prints object without escape chars, such as quotes
      ; 255 indicates the max color (intensity)
      (let (( header (format nil
            "~&P6~&~a ~a~&~a~%" cols rows 255) ))
  (loop for char across header do
     (write-byte (char-code char) st)))

      (loop for x from 0 to (- cols 1) do
   (loop for y from 0 to (- rows 1) do
       (let ((pixel (nth x (nth y colors)) ))
    (write-byte (max 0 (min 255
                     (floor (* 255 (car pixel))))) st)
    (write-byte (max 0 (min 255
                     (floor (* 255 (cadr pixel))))) st)
    (write-byte (max 0 (min 255
                     (floor (* 255 (caddr pixel))))) st)
        )
    )
       )
   )
filename)
```

Unknown macro: {td}

Lisp has a *—write-byte* function which writes the binary representation of an integer to the specified stream. The format is *—(write-byte integer stream)*. To write characters we use the *—char-code* function to convert characters into integers which can be written. Entire strings (such as the header) must be written out one character at a time. To do so we use a for-each loop, which is an example of an imperative structure which has worked its way into the Common Lisp language. The syntax is similar to that of any other language. Semantically, it just executes the body for each character in the string *—header*.

To write out every pixel's rgb values we use a different loop structure. This one is like the conventional *—for* loop that iterates over a specified range of integers. Lisp provides other looping options as well. The nested loops access every pixel in a two-dimensional list, which is a large list containing lists of rows of pixels (which are themselves lists of three elements). That is, colors = [row1 row2 ... rowN] where rowi = [pixeli1 pixeli2 ... pixeliN] and pixelij = [r g b]. In Lisp, an image with one row and three columns is written as *—(list (list '(.7 .3 .13) '(.45 .8 .1) '(.23 0 .73)))*.

The colors need to be integers between 0 and 255 for the ppm, but our functions for visualizing the Julia set produce numbers between 0 and 1 to be interpreted as colors. Before we can write out the colors we scale them to 255 and then cap the values at 0 and 255 with *—max* and *—min* in case some numbers were outside the expected range.

Generating the Image

Unknown macro: {table}

Unknown macro: {tr}

Unknown macro: {td}

```
;calculates a Julia Set from -1.5(1+i) to 1.5(1+i)
;with precision corresponding to rows&cols
(defun calculateMatrix (rows cols)
    (labels (
       (calcMatrix-util (i rows cols)
          (if (= i rows)
            '()
            (cons (calculateRow i 0 rows cols)
                  (calcMatrix-util (+ i 1) rows cols) )
        ))
        (calculateRow (i j rows cols)
           (if (= j cols)
              '()
               (cons (getColor (+(* (/ i rows) 3) -1.5)
                               (+ (* (/ j cols) 3) -1.5))
                     (calculateRow i (+ j 1) rows cols) )
           )
        ))
 (calcMatrix-util 0 rows cols) )
)
```

Unknown macro: {td}

In Lisp you usually create lists recursively: one element at a time, which you add to the beginning of the rest of the list. To make a two-dimensional list in an imperative style, you could use nested for-loops (similarly to how we read the list above, and discussed in Iteration). In the functional style, this requires two recursive functions. We define these helper functions in a *—labels* block, which is a special operator described in Functions.

—calculateMatrix is the user-friendly wrapper for the recursive function *—calcMatric-util*. The utility function builds the image one row at a time by calling *—calculateRow*. Since both of these functions are only useful to calculate an entire matrix, it is handy to be able to group them with *—label s* into a single function. *—calculateRow* is also recursive: if the current column is not the last one, then it gets the pixel color for the specified row and column, and then calls itself to create the rest of the row.

The arithmetic transformation in the call to *—getColor* converts row and column values into x and y values between -1.5 and 1.5, which are good bounds for a Julia set.

Unknown macro: {tr}

Unknown macro: {td}

```
;finds how long it takes for the complex number c to become large
(defun iter (c)
    (labels ((iterate-util (c n max f)
                (if (= n max)
                  n
                  (let ((next (funcall f c)))
                    (if (>= (abs next) 2)
                       n
                       (iterate-util next (+ n 1) max f)
                    )
                  )
                )
            ))
       (iterate-util c 1 500
          (lambda (x) (+ (expt x 2) (complex -.75 .11)) ))
 )
)
```

Unknown macro: {td}

This function also uses labels to use recursion and reduce the caller's argument requirement. *—n* is the control variable to keep track of how many times the function has been iterated. *—max* provides a stopping point, since some numbers converge to a point or join a finite orbit. *—f* is the function (used as a variable, as in First-class functions) whose Julia set we are calculating. We provide this with a *—lambda* expression (see F unctions) so that it is easy to modify the code to use a different function. It also separates the function from the algorithm of iteration which increases clarity.

Inside the *—lambda* expression we create a complex number, which is a built-in type in Lisp, unlike many other languages. This is very useful because in the condition of the if-statement we find the "abs" of the complex number *—next*, and lisp calculates the norm. In C you would have to deal with x and y values and compute the norm yourself.

Notice that we use a *—let* statement to store the value of the next value in the sequence. Since we use *—next* twice, this prevents us from having to compute the function twice.

Unknown macro: {tr}

Unknown macro: {td}

```
; gets the color for x+iy
(defun getColor (x y)
  (let ((n (iter (complex x y)) ))
     (cond ((= n 500)
                '(0 0 0))
           ((= (mod n 7) 0)
                (list (log n)
                      (sqrt (/ n 50))
                      (/ (expt (mod n 50) 2) 10) ))
           ( t
                (list (sin n) (cos n) (sin n)))
    )
  )
)
```

Unknown macro: {td}

—getColor is the link between *—iter* and *—calculateMatrix*. It converts the x and y coordinates into a complex number and uses *—iter* to determine how many iterations it takes before it is large. Then it chooses a color for the pixel based on the value of *—n*. Any points which stayed

small are colored white, and then the others are sorted into classes (here, just ?? mod 7??) and colored with different strategies.

This uses the macro —*cond* to select among several conditions. This is like an if-elseif block in many other languages. If the first condition (the first element of the first list) is not true, then it tries the next condition (the first element of the second list). It executes statement following the first condition to be true, and exits the —*cond* afterward. By making the last condition —*t* you get an else clause, which executes if none of the others are true.