EA1 Homework Program 4: Iterated functions and Julia sets

1 Complex numbers in Matlab

In this assignment you will be working with complex numbers, that is, numbers having both real and imaginary parts. Don't worry if you don't remember much about complex numbers—Matlab will be doing all the work for you! In Matlab, you can create complex numbers using either `i` or `j` to denote the imaginary unit $i = \sqrt{-1}$. For example, typing `z=2-5i` in the command window will create a variable `z` whose value is a complex number having real part 2 and imaginary part -5. Vectors and matrices can contain complex elements as well: typing `rand(2,3)+1i*rand(2,3)` will create a 2-by-3 matrix having random real and imaginary parts. Note that we used `1i` instead of just `i` in this expression; this is because if you would happen to redefine `i` as a variable (thus overwriting Matlab's definition), then `1i` would still mean $\sqrt{-1}$ whereas `i` would mean whatever you assigned it to mean. For this reason it is best to avoid using `i` or `j` unless they come immediately after a number: `1i` or `0.6j` are fine, but avoid `3*i`.

The ordinary arithmetic operators in Matlab work as expected with complex numbers. Also, the absolute value function `abs` works with complex numbers: if x and y are real, then $|x+iy|$ is defined to be $\sqrt{x^2+y^2}$, that is, the distance from $x+iy$ to the origin in the complex plane. For example, `abs(3+4i)` returns `5`.

2 Julia sets

Consider the function f defined as $f(z) = z^2 + c$, where z is the independent variable and c is a constant offset. Here z and c can be complex numbers having both real and imaginary parts. Given this function f and a starting point z_0, we can generate a sequence $\{z_0, z_1, z_2, \dots\}$ of complex numbers as follows: we set $z_1 = f(z_0)$, $z_2 = f(z_1)$, $z_3 = f(z_2)$, and so on. In other words, for each index $k = 1, 2, 3, \dots$ we set $z_k = f(z_{k-1}) = z_{k-1}^2 + c$. Thus starting from some z_0 we obtain

$$\begin{aligned} z_1 &= z_0^2 + c \\ z_2 &= z_1^2 + c = (z_0^2 + c)^2 + c \\ z_3 &= z_2^2 + c = ((z_0^2 + c)^2 + c)^2 + c \\ z_4 &= z_3^2 + c = (((z_0^2 + c)^2 + c)^2 + c)^2 + c \end{aligned}$$

and so on. The following table shows the beginning part of the sequence for some different choices for the starting point z_0 when $c = -1$ (with approximate values shown as needed to save space):

z_0	z_1	z_2	z_3	z_4	z_5	z_6
0	-1	0	-1	0	-1	0
0.5	-0.75	-0.4375	-0.8086	-0.3462	-0.8802	-0.2253
$0.6 + 0.4i$	$-0.8 + 0.5i$	$-0.6 - 0.8i$	$-1.2 + 0.9i$	$-0.3 - 2.3i$	$-6.0 + 1.3i$	$33 - 15i$
i	-2	3	8	63	3968	15745023

Notice that for the first two choices for z_0 in this table, the resulting sequences are *bounded*, that is, the numbers do not get larger and larger as the sequences progress. In contrast, the sequences for the third and fourth choices for z_0 are *unbounded*, that is, the numbers explode in size as the sequences progress. Let us now define the following set of complex numbers:

$$\mathcal{J}_c = \text{ the set of all complex numbers } z_0 \text{ that generate a bounded sequence } \{z_0, z_1, z_2, \dots\}.$$

According to our table, the numbers 0 and 0.5 belong to $\mathcal{J}_c$ when $c = -1$, but the numbers $0.6 + 0.4i$ and i do not. These sets $\mathcal{J}_c$ are called *filled Julia sets*, and they are famous for being remarkably complicated considering the simplicity of the iteration $z_k = z_{k-1}^2 + c$. The purpose of this assignment is to write a Matlab function that will compute visualizations of these Julia sets $\mathcal{J}_c$.

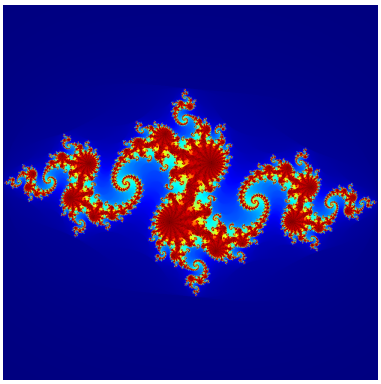
3 Visualizing Julia sets

It turns out that for any choice for c, the generated sequence $\{z_0, z_1, z_2, \dots\}$ is bounded (that is, z_0 belongs to the Julia set $\mathcal{J}_c$) if and only if $|z_k| \leqslant R_c$ for all k, where R_c is the *escape radius* given by the formula

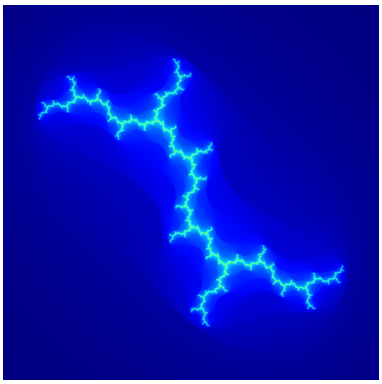
$$R_c = \frac{1 + \sqrt{1 + 4|c|}}{2}.$$

We can therefore assign two numbers to each complex value of z_0, its *escape time* and its *escape value*: if $\{z_0, z_1, z_2, \dots\}$ is the sequence generated by z_0 as described above, then the escape time for z_0 is the smallest index k such that $|z_k| > R_c$, and its escape value is $|z_k|$. For example, when $c = -1$ we have $R_c = 1.618$, so looking at the above table we see that $z_0 = i$ has an escape time of $k = 1$ and an escape value of $|z_1| = 2$. Similarly, $z_0 = 0.6 + 0.4i$ has an escape time of $k = 4$ and an escape value of $|z_4| \approx |-0.3 - 2.3i| \approx 2.3$. If z_0 belongs to $\mathcal{J}_c$ (as do the first two choices for z_0 in the table when $c = -1$), then we say its escape time is infinite and its escape value is undefined. We can represent infinite and undefined values in Matlab using the constants `Inf` (meaning "infinity") and `NaN` (meaning "not a number"), respectively.

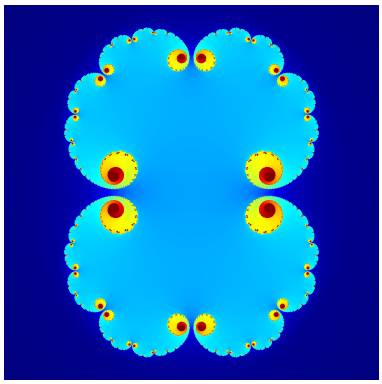
The escape time measures how slowly or quickly the sequence blows up: if the escape time is small, it blows up quickly, and if the escape time is large, it blows up slowly. We can visualize $\mathcal{J}_c$ over a rectangular region in the complex plane by dividing the region into equally spaced points (with each point representing a single pixel in the image), calculating the escape time for each point, and assigning different colors to different escape times. Using the escape values as well in the formula for the color assignment reduces color quantization and generally results in nicer images. The following images show such visualizations of $\mathcal{J}_c$ for different values of c. In these images, the black points represent values of z_0 that actually belong to $\mathcal{J}_c$ (i.e., values of z_0 whose escape times are infinite), the dark red points represent values of z_0 that have large escape times (the corresponding sequences blow up slowly), and the dark blue points represent values of z_0 that have small escape times (the corresponding sequences blow up quickly). Colors in between (like yellow) represent escape times in between.



$c = -0.8 + 0.156i$



$c = i$



$c = 0.26$

To calculate the escape time for a particular value of z_0, we simply generate elements of the sequence $\{z_0, z_1, z_2, \dots\}$ one by one until we reach a value of k for which $|z_k| > R_c$. This final value of k is then the escape time and $|z_k|$ is the escape value. In practice we must stop at some maximum value of k (otherwise this

would go on forever if c happened to belong to $\mathcal{J}_c$). For example, the following code will calculate the escape time and escape value for $z_0 = 1.27 + 0.004i$ and store them in variables called `escTime` and `escVal` (respectively), using a maximum of 1000 iterations:

```
1  z = 1.27 + 0.004i;
2  c = -1;
3  R = (1+sqrt(1+4*abs(c)))/2;
4  escTime = Inf;
5  escVal = NaN;
6  for k = 1:1000
7      z = z^2 + c;
8      if abs(z) > R
9          escTime = k;
10         escVal = abs(z);
11         break
12     end
13 end
```

If you run this in Matlab, the variables `escTime` and `escVal` should end up having the values `24` and `3.5244`, respectively. Likewise, if you change the initial value of `z` in line 1 to be something inside $\mathcal{J}_c$ (such as `z=0`), then after running lines 2 through 13 again, the variables `escTime` and `escVal` should end up having the values `Inf` and `NaN`, respectively (corresponding to an infinite escape time with an undefined escape value). Thus if you put this code inside a loop which loops over all pixels in an image, storing each calculated escape time and escape value in a corresponding array, then at the end you will have the data you need to create pictures like those above. The problem with this approach is that it will be very slow in Matlab for large-resolution images. For example, for an image of size 1024 by 1024 pixels, there will be $2^{20} \approx 10^6$ different z_0 values, which means we would need to execute the above `for` loop a total of 2^{20} different times. Fortunately, there is a much faster way that uses *logical indexing* and *vectorization*. Logical indexing lets you select/access only those elements of an array that satisfy some logical criterion. Vectorization lets you operate on entire arrays at once without explicitly looping over the elements.

4 The assignment

You will create a function m-file in this assignment, unlike the script m-files you created in previous assignments. A function m-file begins with a function declaration, which is a line starting with the keyword `function` and containing information about the inputs, outputs, and name of the function. Your function should be called `julia`, and the file name should be `julia.m`. There will be five inputs to your function, one mandatory input and four optional inputs:

`c` A scalar representing the value of the parameter c. This input is mandatory, so there is no default value.

`limits` This is a 4-element vector specifying a rectangular region in the complex plane. It has the form `[XMIN XMAX YMIN YMAX]`, where `XMIN` and `XMAX` are the minimum and maximum real parts and `YMIN` and `YMAX` are the minimum and maximum imaginary parts. For example, `[-1 2 -2 3]` specifies a rectangular region in the complex plane with $-1 - 2i$ as the lower left corner and $2 + 3i$ as the upper right corner. This is similar to the input to the `axis` function. Default is `[-R R -R R]`, where `R` is the escape radius for `c`.

`nx` The number of points (pixels) in the x-direction. Default is `1024`.

`ny` The number of points (pixels) in the y-direction. Default is `1024`.

`maxEscTime` The maximum number of iterations in the sequence allowed when calculating the escape times. This will be the maximum effective escape time besides `Inf`. Default is `1000`.

There will also be three outputs to your function:

`EscTime` An `ny`-by-`nx` array containing the escape times for each pixel.

`EscVal` An `ny`-by-`nx` array containing the escape values for each pixel.

`Image` An array containing the color data for the image.

You will write your own code to calculate `EscTime` and `EscVal`, and we will provide you with the code to calculate `Image`.

1. Begin your file with the function declaration, followed by comments to document your function. After the function documentation, enter a blank line followed by comments with your name, etc., as in

```
% Homework Program 4
%
% Name:       Kent, Clark
% Section:    30
% Date:       10/10/2019
```

2. Set each of the four optional input variables to its default value if it is either missing or empty `[]`. For example, if we had an input `X` and wanted to give it a default value of `3`, we could include the code

```
if ~exist('X','var') || isempty(X)
    X = 3;
end
```

   This will assign `X` its default value if either it does not yet exist as a variable (hence the `'var'`) or if it exists but is empty. Include code like this for each of the four optional inputs. Note that this method of assigning default values to inputs is more flexible than the method demonstrated in Example 6.3 of the Chapman text (pages 254–256), which is based on simply checking the value of `nargin`. For example, the above method allows you to call your function as `julia(-1,[],64,64)` to get a low-resolution image with the default `limits` and `maxEscTime`.

3. Create an `ny`-by-`nx` array called `Z` that contains all of the z_0 values in the desired rectangular region. The functions `linspace` and `meshgrid` are useful here, and in fact the following code will do what you want (just copy and paste it):

```
x = linspace(limits(1),limits(2),nx);
y = linspace(limits(4),limits(3),ny);
[X,Y] = meshgrid(x,y);
Z = X + 1i*Y;
```

   This code will produce an array `Z` of complex numbers. The element `Z(1,1)` represents the upper left corner of the region, which has the value `XMIN+1i*YMAX`. Likewise, `Z(ny,1)` represents the lower left corner of the region, which has the value `XMIN+1i*YMIN`. Similarly, `Z(1,nx)` represents the upper right corner of the region, which has the value `XMAX+1i*YMAX`. Finally, `Z(ny,nx)` represents the lower right corner of the region, which has the value `XMAX+1i*YMIN`. All other elements of `Z` are equally spaced between these corner values.

4. Create the `ny`-by-`nx` arrays `EscTime` and `EscVal` containing the escape times and escape values for each corresponding point in `Z`. This main part of the assignment consists of the following steps:

   (a) Initialize the array `EscTime` to be an `ny`-by-`nx` array of all `Inf`, and initialize the array `EscVal` to be an `ny`-by-`nx` array of all `NaN`. Note that `Inf(3,5)` and `NaN(3,5)` will create 3-by-5 arrays of all `Inf` values and all `NaN` values, respectively.

   (b) Create an `ny`-by-`nx` array called `done` of all logical `false` values. This array will flag as "done" any values of z_0 for which we already know the escape time. In the beginning we don't know anything, so `done` should contain all `false` values. Note that `false(3,5)` will create a 3-by-5 array of all `false` values.

   (c) Create a `for` loop with a loop variable `k` that counts from `1` to `maxEscTime`. Inside the loop body, do the following:

      i. Replace each element of `Z` with its square plus the value of the parameter `c`. In other words, replace the current z with z^2+c for each different value of z in `Z`. Do this using vectorization: do not explicitly loop over all values of `Z`, but instead use appropriate array operations to process the whole array `Z` at once.

      ii. Create an `ny`-by-`nx` logical array called `new` whose elements are `true` when the corresponding element of `abs(Z)` is greater than `R` and the corresponding element of `done` is `false`. Again, use vectorization here instead of explicitly looping through all values of `Z` and `done`. The `true` elements of this array `new` will represent all "newly escaped" sequences, that is, all sequences whose escape times are equal to the current value of the loop variable `k`.

      iii. Use logical indexing to assign the current value of `k` to those elements of `EscTime` given by the `true` elements of `new`. Likewise, assign the current absolute values of those elements of `Z` given by the `true` elements of `new` to the corresponding elements of `EscVal`.

      iv. Update the `done` array by setting each element to `true` if either it was already `true` or the corresponding element of `new` is `true`. Again, use vectorization, not explicit loops.

      v. Use `break` to end the loop if all elements of the `done` array are true. Note that when the logical expression used in conjunction with the `if` keyword is an array of logical values, then it is considered true for the purpose of branching when *all* of its elements are true.

5. Create the `Image` array and display the image. You do not have to understand how to do this. Instead, just call the function `showJulia` provided for you in the file `showJulia.m`, which you can download from Canvas from the same folder you got this assignment. The function call is simply

```
% Plot result as a color image
%
Image = showJulia(EscTime,EscVal,limits);
```

Test your function by running the Matlab script in the file `exploreJulia.m`, which you can download from Canvas from the same folder you got this assignment. This script should display the Julia set and wait for you to select a new rectangular region by clicking and dragging on the figure. Be patient—it can take a while to generate the image (perhaps tens of seconds on a slower computer). Once you select a new region, the script will then recompute and show the set on the new region, etc., until you abort by pressing `CTRL-C` or by closing the figure window. See if you can zoom in and find some interesting parts of the set using this script (hint: explore the boundary regions). Note that if you zoom in too far, your function may stop working as expected because of round-off errors.

Finally, submit your function m-file `julia.m` through Canvas. Do not submit any of the figures or pictures.