Julia Set Class

Introduction

The Julia set is a mathematical set defined from a function. In addition to its merits in complex dynamics, it can be used to generate interesting images. Every point in a 2 dimensional plane can be categorized as inside or outside of the Julia set for a particular function choice.

Finding Points in the Julia Set

For our purposes, this is a good enough definition of the Julia set for a function \((x', y') = f(x, y)\), where \((x, y)\) and \((x', y')\) are the coordinates of points in the 2 dimensional plane.

Take a point \((x_0, y_0)\). Using \(x_0\), and \(y_0\) as input to \(f(x, y)\), we receive a new point from the function.

\[
x_1, y_1 = f(x_0, y_0)
\]

Repeat that process by using the output of the previous call of the function as the input to the current call. We could repeat up to \(n\) times:

\[
x_2, y_2 = f(x_1, y_1) \\
x_3, y_3 = f(x_2, y_2) \\
x_4, y_4 = f(x_3, y_3) \\
\vdots \\
x_{n-1}, y_{n-1} = f(x_{n-2}, y_{n-2}) \\
x_n, y_n = f(x_{n-1}, y_{n-1})
\]

For large enough values of \(n\), the resulting point at \(x_n, y_n\) will fall into one of two categories:

1- The point will still be close to the origin \((0,0)\) of the plane. We define close to mean a distance less than or equal to 2.

2- The point will be far from the origin. We define far to mean a distance greater than 2.

If the point is close to the origin, it is part of the Julia set for \(f(x,y)\). Otherwise it is not part of the Julia set.

From Functions to Images

So, where do the interesting pictures come from?

We choose an actual function for \(f(x,y)\), then color each point in the plane based on how far it is from being in the Julia set for the selected function.

Actually, there are infinitely many points in the plane, so we can’t do every point. Instead, we select a regular grid of points each representing the points near it in the plane. For each point in the grid:

- Calculate its \(x_0, y_0\) value, from its position in the grid.
- Iterate the application of \(f(x, y)\) up to \(n\) times.
- If the result becomes far away, remember the iteration number when it first escaped (became far away from the origin). We call this the “escape value”. Points with lower escape values are further from the Julia set than those with higher escape values.
- If the result doesn’t become far away (escape) within \(n\) iterations, assume it did not escape and remember \(n\) as the escape value. These points are assumed to be part of the Julia set for \(f(x,y)\).

Now, we have a set of integers (the escape values) that represent how close each of the points in the grid are to the Julia set for \(f(x,y)\). Larger numbers mean they are closer to the Julia set.

Since the points selected are in a regular grid, we can pair each grid point with a pixel in a rectangular image. We assign the color of pixel based on the escape value of the corresponding grid point. All pixels associated with the same escape value will have the same color.

Defining a Regular Grid

Our end goal is to have a rectangular image with pixel colors assigned based on Julia set escape values. We
make an image that has dimensions of \[\text{width}\] by \[\text{height}\]. The rows of the image are numbered 0 through \[\text{height}-1\] and the columns are numbered 0 through \[\text{width}-1\]. For each of these pixels, we want to calculate a point \([x, y]\) that represents the center of the pixel. These points that are in the center of pixels will be a regular grid, just what we need for our image.

Depending where we want our image to look in the Julia set plane, we will set minimum and maximum x and y values for these points. For example, we may want to have our image look at the region of the plane from -1.5 to 0.5 in the x axis and -0.3 to 1.2 in the y axis. For our example, we’ll make an image that is 4 pixels wide and 3 pixels high. (Very small, but a good example of the calculations.)

Let’s calculate the x coordinates of our image. First, the left most column (0) will have \(x = -1.5\), because that is the left side of the region of the image. Next, the right most column (3) will have \(x = 0.5\), the right side of the region. What are the x coordinates of columns 1 and 2? They need to be regularly spaced. There are \([\text{width}-1 = 3]\) gaps between our columns. Draw a picture to convince yourself. Since there are 3 gaps to travel from -1.5 to 0.5, how big is each gap? The generic formula for this is \(\text{gap}_x = (\text{max}_x - \text{min}_x) / (\text{width} - 1)\).

\[
\text{gap}_x = (0.5 - (-1.5)) / (4 - 1) = 2.0 / 3 = 0.666666
\]

With the gap between columns calculated, we can easily calculate the x coordinate for any column. The generic formula for this calculation is \(x = \text{min}_x + \text{column} \times \text{gap}_x\).

\[
x_{\text{column 0}} = \text{min}_x + 0 \times \text{gap}_x = -1.5 \\
x_{\text{column 1}} = \text{min}_x + 1 \times \text{gap}_x = -0.833333 \\
x_{\text{column 2}} = \text{min}_x + 2 \times \text{gap}_x = -0.166666 \\
x_{\text{column 3}} = \text{min}_x + 3 \times \text{gap}_x = 0.5
\]

Using these two formulas, we can calculate the x coordinate of any column.

A similar discussion for calculating the y coordinate of any row yields these results.

\[
\text{gap}_y = (1.2 - (-0.3)) / (3 - 1) = 1.5 / 2 = 0.75
\]

\[
y_{\text{row 0}} = \text{max}_y - 0 \times \text{gap}_y = 1.2 \\
y_{\text{row 1}} = \text{max}_y - 1 \times \text{gap}_y = 0.45 \\
y_{\text{row 2}} = \text{max}_y - 2 \times \text{gap}_y = -0.3
\]

Now, for every pixel, we can calculate the point in the plane \((x_0, y_0)\), from the \((\text{column, row})\) of the pixel. For each pixel, these values are used to calculate an escape value.

**Assignment**

Create a class named \[\text{JuliaSet}\], to represent a collection of Julia set calculations.

For our function \(f(x, y)\) we will use this definition:

\[
x' = x^2 - y^2 + a \\
y' = 2xy + b
\]

where, \(a\) and \(b\) are parameters used to configure a particular version of the family of functions described by these equations.

Also create a program that calculates a \[\text{JuliaSet}\], creates a \[\text{PPM}\] from the set, and saves the image to a file.

**JuliaSet Class Programming Requirements**

This class must be declared in \[\text{JuliaSet.h}\] and implemented in \[\text{JuliaSet.cpp}\].

Your \[\text{JuliaSet}\] class must store the following data:

- The width and height (in pixels) of the desired image (\[\text{int}\]s).
- The minimum and maximum values for x and y to define the rectangular space in the plane (\[\text{double}\]s).
- The two parameters \(a\) and \(b\) used in \(f(x, y)\) above (\[\text{double}\]s).
- The maximum allowed escape count that corresponds to non-escaping points (\[\text{int}\]).

Your \[\text{JuliaSet}\] class must have the following methods.

**Constructor and Getters**
• **JuliaSet();** Default constructor. Sets up for a 400x300 image using the 4x3 rectangle centered on the origin. Sets \( a \) to -0.650492, \( b \) to -0.478235 and maximum escape count to 255.

- int getWidth() const; Returns the pixel width.
- int getHeight() const; Returns the pixel height.
- double getMinX() const; Returns the plane minimum x value.
- double getMaxX() const; Returns the plane maximum x value.
- double getMinY() const; Returns the plane minimum y value.
- double getMaxY() const; Returns the plane maximum y value.
- double getA() const; Returns the \( a \) parameter.
- double getB() const; Returns the \( b \) parameter.
- int getMaxEscapeCount() const; Returns maximum allowed escape count.

### Setters

- void setPixelSize( const int& width, const int& height ); Sets the pixel width and height of the desired image. Only makes a change if the width and height are both at least 2.
- void setPlaneSize( const double& min_x, const double& max_x, const double& min_y, const double& max_y ); Sets the plane boundaries. If the minimum value for a dimension is greater than maximum value, automatically swap them. Only allows values in the range -2.0 to 2.0 for each. If any is out of range, change nothing. If the minimum value for a dimension is identical to the maximum value for the dimension, then change nothing.
- void setParameters( const double& a, const double& b ); Sets values for \( a \) and \( b \). Only allows values in the range -2.0 to 2.0 for each. If either is out of range, change nothing.
- void setMaxEscapeCount( const int& count ); Sets the maximum allowed escape count. Only makes the change if count is at least 0.

### Coordinate Methods

- Create new data members to store the horizontal and vertical gap sizes. These new data members must be initialized to 0.01 in the default constructor.
- double getDeltaX() const; Returns the horizontal gap value.
- double getDeltaY() const; Returns the vertical gap value.
- void setDeltas( const double& delta_x, const double& delta_y ); Assigns the gaps to data members. Only assigns if both values are positive.
- double calculateDeltaX() const; Calculate the horizontal plane distance between neighboring pixel columns. This is the gap value discussed above. Note this method calculates the value and returns it. It does not set the data member.
- double calculateDeltaY() const; Calculate the vertical plane distance between neighboring pixel rows. This is the gap value discussed above.
- Modify [setPixelSize] and [setPlaneSize] to calculate new gap values and set them, whenever a change is made to the image dimensions or plane dimensions.
- void calculatePlaneCoordinatesFromPixelCoordinates( const int& row, const int& column, double& x, double& y ) const; Sets x and y to the plane coordinates for the row and column. If either row or column is out of range, set both x and y to 0. Notice x and y are return by reference.

### Escape Value Methods

- Add a new data member to store all calculated escape counts. (**std::vector<int>**) The default constructor should initialize it to have 120000 items.
- Modify [setPixelSize] to resize the vector when changes are made.
- const std::vector<int>& getEscapeCounts() const; Returns the vector of escape counts.
- int getEscapeCount( const int& row, const int& column ) const; Returns the escape count stored at the row, column position. If the row or column is out of range, return -1.
- void setEscapeCount( const int& row, const int& column, const int& count ); Stores the escape count in the position for row, column. If row or column are out of range, do nothing. If the escape count is out of range, do nothing.
- void calculateNextPoint( const double& x0, const double& y0, const double& x, const double& y ) const; Calculate the next escape point after \( x_0, y_0 \) and store in \( x_1, y_1 \). Note that \( x_1 \) and \( y_1 \) are return by reference.
- int calculatePlaneEscapeCount( const double& x0, const double& y0 ) const; Calculate the number of iterations required for \( x_0, y_0 \) to escape. The return value should be in the range 0 to maximum escape count, inclusive. 0 means immediately escaped. Maximum escape count means never escaped, or escaped on the last step. Escape means the distance from the origin is more than 2.
- int calculatePixelEscapeCount( const int& row, const int& column ) const; Calculate the number of iterations required for row, column to escape. The return value should be in the range 0 to maximum.
escape count, inclusive. 0 means immediately escaped. Maximum escape count means never escaped, or escaped on the last step. If row or column is out of range, return -1.

- `void calculateAllEscapeCounts();` Calculate escape counts for all pixel locations, storing them.

**Image Methods**

- `void setPPM( PPM& ppm ) const;` Uses the currently calculated escape values to configure an image in the PPM object. Sets the width and height of the image to match the width and height of the Julia set. Sets the maximum color value to 63. For each pixel in the PPM object, sets the color based on the escape value for the pixel. See the following table:

<table>
<thead>
<tr>
<th>Escape Value</th>
<th>Color (R, G, B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0, 0, 0)</td>
</tr>
<tr>
<td>max escape count</td>
<td>(63, 31, 31)</td>
</tr>
<tr>
<td>count % 8 == 0</td>
<td>(63, 63, 63)</td>
</tr>
<tr>
<td>count % 8 == 1</td>
<td>(63, 31, 31)</td>
</tr>
<tr>
<td>count % 8 == 2</td>
<td>(63, 63, 31)</td>
</tr>
<tr>
<td>count % 8 == 3</td>
<td>(31, 63, 31)</td>
</tr>
<tr>
<td>count % 8 == 4</td>
<td>(0, 0, 0)</td>
</tr>
<tr>
<td>count % 8 == 5</td>
<td>(31, 63, 63)</td>
</tr>
<tr>
<td>count % 8 == 6</td>
<td>(31, 31, 63)</td>
</tr>
<tr>
<td>count % 8 == 7</td>
<td>(63, 31, 63)</td>
</tr>
</tbody>
</table>

**Program Programming Requirements**

This program must be written in `julia.cpp`, and the final program must be named `julia`.

The program must:

- Use a menu system.
- Menu loops until user chooses quit
- Options in menu
  - w: set width
  - h: set height
  - x: set minimum x
  - X: set maximum x
  - y: set minimum y
  - Y: set maximum y
  - a: set parameter a
  - b: set parameter b
  - e: set maximum escape value
  - c: calculate all escape values
  - p: assign colors to the PPM
  - s: save the PPM to file
  - S: show Julia set meta data
  - q: quit

**Example Input**

This sample input produces these three images:

- `julia_example1.ppm`
- `julia_example2.ppm`
- `julia_example3.ppm`

This Makefile and `julia.cpp` may be used.

**Additional Documentation**

- [Julia set on Wikipedia](https://en.wikipedia.org/wiki/Julia_set)

**Show Off Your Work**

To receive credit for this assignment, you must complete the unit tests available in CodeGrinder, and you
must upload the source code (.cpp and .h files) and the Makefile to the Canvas submission system. Additionally, the program must build, run and give correct output.