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.

For our purposes, this is a good enough definition of the Julia set for a function \( f(x, y) \). Take a point on the 2 dimensional plane, \((x_0, y_0)\). Use \( x_0 \) and \( y_0 \) as input to \( f(x, y) \), and receive a new point from the function.

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

Repeat that process up to \( n \) times:

\[
x_{2}, y_{2} = f(x_{1}, y_{1})
\]
\[
x_{3}, y_{3} = f(x_{2}, y_{2})
\]
\[
x_{4}, y_{4} = f(x_{3}, y_{3})
\]
\[
...
\]
\[
x_{n-1}, y_{n-1} = f(x_{n-2}, y_{n-2})
\]
\[
x_{n}, y_{n} = f(x_{n-1}, y_{n-1})
\]

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 \((0,0)\). We define close to mean a distance less 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.

So, where do the interesting pictures come from? Build an evenly spaced grid of points near the origin. For each point:

- 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 time step when it first became far away
- record the time step of the escape, or \( n \) if it did not escape

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

Finally, assign a color for each of the escape values, and create an image file with these colors in the correct pixel locations for your grid points. Thus, all points that escape in the same number of steps will have a pixel with the same color.

**Assignment**

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

For your function \( f(x,y) \) use this definition:

\[
\text{new}_x = x^2x - y^2y + a
\]
\[
\text{new}_y = 2xy + b
\]

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

**JuliaSet Programming Requirements**

Your \texttt{JuliaSet} class must store the following data.

- The width and height (in pixels) of the desired image.
- The minimum and maximum values for \( x \) and \( y \) to define the rectangular space in the complex plane.
- The two parameters \( a \) and \( b \) used in \( f(x,y) \) above.
- The maximum allowed escape count that corresponds to non-escaping points.
- All calculated escape counts.
Your JuliaSet class must have the following methods.

**Constructor and Getters**

- **JuliaSet();** Default constructor. Sets up for a 400x400 image using the 4x4 rectangle centered on the origin. Sets \(a\) and \(b\) to 0 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 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 that than maximum value, automatically swap them.
- **void setParameters( const double& a, const double& b );** Sets values for \(a\) and \(b\).
- **void setMaxEscapeCount( const int& count );** Sets the maximum allowed escape count. Only makes the change if count is at least 0.

**Coordinate Methods**

- **double calculateDeltaX() const;** Calculate the horizontal plane distance between neighboring pixel columns.
- **double calculateDeltaY() const;** Calculate the vertical plane distance between neighboring pixel rows.
- **double calculatePlaneXFromPixelColumn( const int& column ) const;** Calculate the plane x value for a given column. If the column is out of range, return 0.
- **double calculatePlaneYFromPixelRow( const int& row ) const;** Calculate the plane y value for a given row. If the row is out of range, return 0.
- **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**

- **void calculateNextPoint( const double x0, const double y0, double& x1, double &y1 ) 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.
- **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 internally.
- **int getPixelEscapeCount( const int& row, const int& column ) const;** Fetch an already calculated escape value, that is stored internally. If row or column is out of range, return -1.

**Program Programming Requirements**

The program must:

- prompt the user for the size of the image to create, the dimensions of the plane to calculate, the values of
a) and b), and the name of the file to store the PPM in. The order must be the same as shown in the examples
- calculate a Julia set using the values given by the user, and a maximum escape count of 255
- create a grayscale PPM using the escape counts from the Julia set as the channel brightness
- save the PPM object to the file specified by the user.

Example Execution 1

```
$ ./main
Pixel Width 768
Pixel Height 768
Min x -0.3
Max x 0.1
Min y 0.1
Max y 0.5
A -0.835
B -0.2321
Filename julia-zoom.ppm
```

Produces [this image](#).

Example Execution 2

```
$ ./main Pixel Width 512
Pixel Height 512
Min x -2.0
Max x 2.0
Min y -2.0
Max y 2.0
A -0.835
B -0.2321
Filename julia-full.ppm
```

Produces [this image](#).

Additional Documentation

- [Julia set on Wikipedia](#)

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.