Julia Set

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:

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

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 the function \(f(x,y)\). Otherwise it is not part of the Julia set for the function \(f(x,y)\). A point that is not part of the Julia set “escapes” at the first iteration where the new \((x,y)\) point is a distance of more than 2 from the origin. For points that escape, we will want to remember the “escape count”, or which iteration it escaped.

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. “How far” is the “escape count”.

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 \( \text{height} \times \text{width} \). 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. The will be regular because we will space them at fixed distances from each other.

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. We will use the term delta to indication the change in a coordinate over a gap. 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 \( \text{delta}_x = \frac{\text{max}_x - \text{min}_x}{\text{width} - 1} \).

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

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

\[
\begin{align*}
\text{x\_column\_0} &= \text{min}_x + 0 \times \text{delta}_x = -1.5 \\
\text{x\_column\_1} &= \text{min}_x + 1 \times \text{delta}_x = -0.833333 \\
\text{x\_column\_2} &= \text{min}_x + 2 \times \text{delta}_x = -0.166666 \\
\text{x\_column\_3} &= \text{min}_x + 3 \times \text{delta}_x = 0.5 \\
\end{align*}
\]

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.

\[
\begin{align*}
\text{delta}_y &= \frac{1.2 - (-0.3)}{3 - 1} = \frac{1.5}{2} = 0.75 \\
\text{y\_row\_0} &= \text{max}_y - 0 \times \text{delta}_y = 1.2 \\
\text{y\_row\_1} &= \text{max}_y - 1 \times \text{delta}_y = 0.45 \\
\text{y\_row\_2} &= \text{max}_y - 2 \times \text{delta}_y = -0.3 \\
\end{align*}
\]

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

**Our Function \( f(x, y) \)**

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

\[
\begin{align*}
x' &= x^2 - y^2 + a \\
y' &= 2x*y + b
\end{align*}
\]

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

**Assignment**

In this assignment you will create a class to create and store a Julia set’s escape values. It will inherit from \texttt{NumberGrid} for the height, width, max value, and number storage.

You will also extend the \texttt{ppm_menu} program to add a few new commands.

The new commands required are:

- \texttt{fractal-plane-size}: Set the dimensions of the grid in the complex plane.
- \texttt{fractal-calculate}: Calculate the escape values for the fractal.
- \texttt{julia-parameters}: Set the parameters of the Julia Set function.

**Potential Session**
Programming Requirements

Below, the functions and methods may have a symbol CG? before them, where ? is a number. This indicates which Code Grinder step requires the function or method to be implemented.

Modify **NumberGrid.h**

The following methods must be modified or created in the `NumberGrid` class.

- CG1 `virtual ~NumberGrid() {};
- CG1 `virtual void setGridSize( const int& height, const int& width );
- CG1 `virtual int calculateNumber( const int& row, const int& column ) const = 0;
- CG1 `virtual void calculateAllNumbers();`

Modify **NumberGrid.cpp**

The following methods must be implemented or changed for the `NumberGrid` class.

- CG1 `virtual ~NumberGrid() {};` This destructor only needs an empty block of code. But, it must exist.
- CG1 `virtual void setGridSize( const int& height, const int& width );` No changes in the implementation. Added `virtual` in declaration.
- CG1 `virtual int calculateNumber( const int& row, const int& column ) const = 0;` No implementation. This is a pure virtual method.
- CG1 `virtual void calculateAllNumbers();` For every row, column pair, uses `calculateNumber` to get a number and `setNumber` to store it.

Create **JuliaSet.h**

The `JuliaSet` class must inherit publicly from `NumberGrid`.

The `JuliaSet` class must store the following information:

- minimum and maximum values for x and y to define the plane size
- delta values for x and y to define the grid points in the plane
- a and b parameter values for the Julia set function

The following methods must be created in the `JuliaSet` class.

- CG1 `JuliaSet();`
- CG1 `JuliaSet( const int& height, const int& width, const double& min_x, const double& max_x, const double& min_y, const double& max_y, const double& a, const double& b );`
- CG1 `virtual ~JuliaSet();`
- CG1 `double getMinX() const;`
- CG1 `double getMaxX() const;`
- CG1 `double getMinY() const;`
- CG1 `double getMaxY() const;`
- CG1 `double getA() const;`
- CG1 `double getB() const;`
- CG1 `virtual void setGridSize( const int& height, const int& width );`
- CG1 `void setPlaneSize( const double& min_x, const double& max_x, const double& min_y, const double& max_y );`
- CG1 `void setParameters( const double& a, const double& b );`
Create **JuliaSet.cpp**

- **JuliaSet();** Default constructor. Sets up for a 300x200 grid. For the plane coordinates uses the 3x2 rectangle centered on the origin. Sets a to -0.650492, b to -0.478235 and maximum number count to 255. Sets the default value for delta_x and delta_y to 0.01. What values of minx and max would give you a rectangle of width 3 and centered on the origin?
- **JuliaSet( const int& height, const int& width, const double& min_x, const double& max_x, const double& min_y, const double& max_y, const double& a, const double& b );** Constructor. Sets up the **NumberGrid** and **JuliaSet** data members from parameters.
- **virtual ~JuliaSet();** Must exist, but has empty code block.
- **double getMinX() const;** Return the minimum X value for the plane coordinates.
- **double getMaxX() const;** Return the maximum X value for the plane coordinates.
- **double getMinY() const;** Return the minimum Y value for the plane coordinates.
- **double getMaxY() const;** Return the maximum Y value for the plane coordinates.
- **double getA() const;** Return the a parameter for the Julia set.
- **double getB() const;** Return the b parameter for the Julia set.
- **virtual void setGridSize( const int& height, const int& width );** This method overrides the **NumberGrid** version. Only makes changes if both height and width are at least 2. If so, it calls **NumberGrid::setGridSize()**.
- **void setPlaneSize( const double& min_x, const double& max_x, const double& min_y, const double& max_y );** Sets the 4 plane coordinates. Only makes a change if all of the coordinate values are between -2.0 and 2.0, inclusive. Only make changes if the minimum and maximum value for a dimension are different. If the minimum value for a dimension is greater than the maximum value for the dimension, automatically swap them.
- **void setParameters( const double& a, const double& b );** Sets a and b parameters. Only allows values in the range -2.0 to 2.0 for each. If either is out of range, change nothing.
- **double getDeltaY() const;** Returns the vertical delta value.
- **double getDeltaX() const;** Returns the horizontal delta value.
- **void setDeltas( const double& delta_x, const double& delta_y );** Assigns the deltas to data members. Only assigns if both values are positive.
- **double calculateDeltaY() const;** Calculate the vertical plane distance between neighboring pixel rows. This is the delta value discussed above. Note this method calculates the value and returns it. It does not set the data member.
- **double calculateDeltaX() const;** Calculate the horizontal plane distance between neighboring
The following functions will require updates to their functionality and/or declarations.

- **CG1** Modify `setGridView` and `setPlaneSize` to calculate new delta values and set them, whenever a change is made to the image dimensions or plane dimensions.

- **CG1** `double calculatePlaneYFromPixelRow( const int& row ) const;` Calculate the plane y value for a given row. If the row is out of range, return 0. Do not call `calculateDeltaY()` here. Use `getDeltaY()` or directly access the data member. The value should have already been calculated previously.

- **CG1** `double calculatePlaneXFromPixelColumn( const int& column ) const;` Calculate the plane x value for a given column. If the column is out of range, return 0. Do not call `calculateDeltaX()` here. Use `getDeltaX()` or directly access the data member. The value should have already been calculated previously.

- **CG1** `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. This is the Julia set function $f(x,y)$.

- **CG1** `int calculatePlaneEscapeCount( const double& x0, const double& y0 ) const;` Calculate the number of iterations required for x0, y0 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.

- **CG1** `virtual int calculateNumber( 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. Should use `calculatePlaneXFromPixelColumn()`, `calculatePlaneYFromPixelRow()`, and `calculatePlaneEscapeCount()`.

**Update** `image_menu.cpp`

- **CG1** `int imageMenu( std::istream& is, std::ostream& os );` Change the declaration of the `NumberGrid` object that is passed to `takeAction`. We now want a `NumberGrid*` that is allocated from the heap, and points to a `JuliaSet` object. Such as `NumberGrid* gptr = new JuliaSet( );`. This also requires the call to `takeAction` to pass `*gptr`, instead of `grid`. Finally, after the while loop finishes, if `gptr` is not 0, then delete it.

**Update** `image_menu.h`

Add the following function declarations to the file.

- **CG2** `void setFractalPlaneSize( std::istream& is, std::ostream& os, NumberGrid& grid );`
- **CG2** `void calculateFractal( std::istream& is, std::ostream& os, NumberGrid& grid );`
- **CG2** `void setJuliaParameters( std::istream& is, std::ostream& os, NumberGrid& grid );`

**Update** `image_menu.cpp`

This file must include the implementations for the new functions declared in `image_menu.h`.

- **CG2** `void setFractalPlaneSize( std::istream& is, std::ostream& os, NumberGrid& grid );` Asks the user for the `doubles` "Min X?", "Max X?", "Min Y?", and "Max Y?", then sets the plane size. Only does this work if the `grid` is actually a JuliaSet object. Otherwise, gives a message "Not a JuliaSet object. Can’t set plane size."

- **CG2** `void calculateFractal( std::istream& is, std::ostream& os, NumberGrid& grid );` Calculates all numbers for the `grid`.

- **CG2** `void setJuliaParameters( std::istream& is, std::ostream& os, NumberGrid& grid );` Asks the user for the `doubles` "Parameter a?" and "Parameter b?". Then sets the parameters. Only does this work if the `grid` is actually a `JuliaSet` object. Otherwise gives the message "Not a JuliaSet object. Can’t set parameters."

The following functions will require updates to their functionality and/or declarations.

- **CG2** `void showMenu( std::ostream& os );` Add to the menu to include the following messages: "fractal-
Set the dimensions of the grid in the complex plane.

```
plane-size)
```

Calculate the escape values for the fractal.

```
fractal-calculate)
```

Set the parameters of the Julia Set function.

```
julia-parameters)
```

CG2 `void takeAction( std::istream& is, std::ostream& os, const std::string& choice, PPM& input_image1, PPM& input_image2, PPM& output_image, NumberGrid& grid );` Add to the recognized commands to recognize the new actions in the menu, and take the correct action.

**Update ppm_menu.cpp**

No changes are required for `ppm_menu.cpp`.

**Update Makefile**

- This file must include the rules to build the program `ppm_menu`.
- A developer must be able to use the command `make` to compile all necessary files and link them to the executable program `ppm_menu`.
- Additionally, add the `clean` target that has no dependencies, but will remove any `.o` files and `ppm_menu`.
- Add automatic source and object file calculation to the `Makefile`.
- Add automatic dependency calculations to the `Makefile`.

**Additional Documentation**

- [C++ Reference](#)
- [Examples from class](#)
- [Sample Session Input File](#)
- [Julia set on Wikipedia](#)

**Sample PPM Images**

- [Sample Output1](#)
- [Sample Output2](#)
- [Sample Output3](#)

**Show Off Your Work**

To receive credit for this assignment, you must

- complete the unit tests available in CodeGrinder
- zip the source code (.cpp and .h files) and the Makefile and upload to the Canvas submission system
- use git to add, commit and push your solution to your repository for this class.

Additionally, the program must build, run and give correct output.

**Extra Challenges (Not Required)**

- Create classes that inherit from `JuliaSet` that have different functions, $\ell(x,y)$, for calculating escape values. Add the ability to use them from the `imageMenu()`.
- Make methods of the `JuliaSet` class that allow for zooming in or out in the plane. Add the ability to use them from the `imageMenu()`.
- Try other ways to modify the plane and parameters that would make it easier to create interesting Julia set images.