**CS 3005: Programming in C++**

**Turing Pattern Image Generation**

**Introduction**

Turing patterns have been proposed as a mechanism to explain how patterns develop in nature. In particular, leopard spots, tiger stripes, puffer fish spots and angel fish stripes have been described as biological instances of reaction-diffusion systems.

For this assignment, you will write code to simulate a reaction-diffusion system, converting the result into an image, or a series of images to show the generation of the patterns.

**Overview**

**Diffusion Calculation**

In a 2-dimensional space, a chemical with a high concentration at one location will diffuse over time and spread to the rest of the space. This is called diffusion, and can be simulated using a diffusion equation. We'll assume a rectangular grid of locations, each location is a square of length $dx$ on each side. We calculate updates every $dt$ seconds, where $dt$ is usually a small number like 0.001. Using the previous time step's values to calculate the current time step's values.

The algebraic version of the equation of diffusion update for chemical species A at grid position $(row, column)$ is:

```
DeltaA(row,column) = -4.0 * A(row, column) 
    + A( row - 1, column ) 
    + A( row + 1, column ) 
    + A( row, column - 1 ) 
    + A( row, column + 1 )
```

And the divergence (actual change) is

```
DivergenceA( row, column ) = DeltaA / (dx*dx)
```

The final concentration for the current time step will be

```
Acurrent( row, column ) = Aprevious + dt * Da * DivergenceA( row, column )
```

**Reaction Calculation**

In order to see the Turing patterns, we need to have 2 species of chemical. We’ll call them A and B. There will be a similar set of equations for the diffusion of B.

The patterns come when A and B react to each other. This can result in more A and B begin generated or consumed at particular grid locations, depending on the relative concentrations of the two chemicals. We define two reaction functions for a grid location $(row, column)$. We introduce two more parameters, $\alpha$ and $\beta$.

```
Racurrent(row, column) = Aprevious(row, column) - Aprevious(row, column)*Aprevious(row, column)*Aprevious(row, column) - Bprevious(row, column) + alpha

Rbcurrent(row, column) = ( Aprevious(row, column) - Bprevious(row, column)) * beta
```

We update the diffusion equations to be reaction-diffusion equations by adding in the reaction functions.

```
Acurrent( row, column ) = Aprevious(row, column) + dt * Da * DivergenceA( row, column ) + dt * Ra( Aprevious( row, column ), Bprevious( row, column ) )

Bcurrent( row, column ) = Bprevious(row, column) + dt * Db * DivergenceB( row, column ) + dt * Rb( Aprevious( row, column ), Bprevious( row, column ) )
```

**Iterations**
These equations are meant to simulate the gradual process of diffusion and reaction. It takes time for the final state of the system to be reached. We need to repeat the process many times to reach the stable state of the system.

This is implemented by looping over the update, trading between two sets of vectors, using one as the previous and one as the current values. Trading between each iteration.

### Concentrations to Pseudo Escape Values

We need to calculates fake (pseudo) escape values to use as indexes into the color table by the `setPPM` method. Here’s the process:

Let $d$ be the difference between $a$ and $b$ concentrations for a pixel. We’ll convert that difference to a number $s$ which is between 0 and 1 using the maximum and minimum differences in the whole grid. Let $\text{min}_d$ be the minimum difference and $\text{max}_d$ be the maximum difference. Then:

$$s = \frac{d - \text{min}_d}{\text{max}_d - \text{min}_d}$$

Let $\text{max}_e$ be the maximum escape value. The escape value $e$ will be:

$$e = 1 + (\text{max}_e - 2) \times s$$

That way, we’ll rule out the “immediately escaped” and “never escaped” values of 0 and $e$, allowing for smooth gradient coloring of the results.

### The TuringPattern Class

The `TuringPattern` class will be made to inherit the functionality of the `ComplexFractal` class adding the ability to simulate a reaction-diffusion system with 2 chemical species.

### Suggested Data Members

These data members are suggested to be added to the class.

- `std::vector< std::vector< std::vector< double > > > mData;` The outer vector (and first index in []) will be for switching between current and previous values. The next vector (and second index in []) will be for the chemical species (A == 0, B == 1). The inner vector (and third index in []) is for the actual concentration values. An example usage may be `this->mData[mCurr][1][i]` to get the $i$th value of B in the current data set.
- `int mCurr, mPrev;` One of these will be 0, the other will be 1. They trade at the beginning of a new time step.
- `double mDx, mDt;` Physical parameters for use in the reaction-diffusion equations.
- `double mAlpha, mBeta;` Parameters used by the $R_a$ and $R_b$ reaction functions.
- `double mDa, mDb;` Diffusion constants for use in the reaction-diffusion equations.
- `double mMinDifference, mMaxDifference;` Values used to translate concentration values to pseudo escape values.

### Required Methods

- `TuringPattern( );` Constructor chains to `ComplexFractal()`, then initializes data members $mCurr(0)$, $mPrev(1)$, $mDx(1.0)$, $mDt(0.001)$, $mAlpha(-0.005)$, $mBeta(10.0)$, $mDa(1.0)$, $mDb(100.0)$, $mMinDifference(1.0e10)$, $mMaxDifference(-1.0e10)$. Also calls `setPreferredSize()` in the body of the constructor to resize `mData`.
- `double getDx() const;`
- `double getDt() const;`
- `double getAlpha() const;`
- `double getBeta() const;`
- `double getDa() const;`
- `double getDb() const;`

- `void setParameters(const double& dx, const double& dt, const double& alpha, const double& beta, const double& Da, const double& Db );` Only makes changes if all 6 parameters have values greater than 0.

- `double getCurrValue(const int& row, const int& column, const int& species ) const;` Returns the current value for the specified species. Only returns the value if `species` is 0 or 1, and `row` and `column` are in legal ranges. Uses this syntax: `this->mData[mCurr][species][i]`, where $i$ is calculated in the
same way as the escape value indexes in the ComplexFractal class.

- double getPrevValue( const int& row, const int& column, const int& species ) const; Similar to getCurValue.

- void setCurValue( const int& row, const int& column, const int& species, const double& value ); Reverse of getCurValue. Assigns to the correct location. Any value is valid.

- void setPrevValue( const int& row, const int& column, const int& species, const double& value ); Similar to setCurValue.

- void swapCurPrev( ); Swaps the values of mCurr and mPrev.

- double randomizesValues( ); For each species (0,1) for every row and column, generate a random number between -0.01 and 0.01, assign this value into both current and previous data, using setCurValue and setPrevValue. Generate a random floating point number using \[(\text{std::rand()} \times \% \text{big_value}) - (\text{big_value} / 2) \] / (50. * \text{big_value}), where \text{int big_value = 1000000}; or so.

- double calculateDivergence( const int& row, const int& column, const int& species ); Calculates the Divergence value for species using the formulas in the discussion above. Be sure to use the previous values. Assumes input values are valid.

- double calculateCurValue( const int& row, const int& column, const int& species ); Calculates the Current value using the equation shown in the discussion above. Note this should call calculateDivergence and call the correct Ra or Rb function depending on the value of species. Assumes input values are valid. The values of \(a\) and \(b\) passed to \(Ra\) and \(Rb\) are from getPrevValue for the current \(row\) and \(column\).

- void updateValues( int steps ); Loops \(steps\) times, each time, swapping current and previous, and calculating current value for all rows and columns, storing the result with setCurValue.

- double Ra( const double& a, const double& b ) const; Calculates the reaction value for species A, with current values of \(a\) and \(b\). Use the equation shown in the discussion above. Look for \(Racurrent\).

- double Rb( const double& a, const double& b ) const; Calculates the reaction value for species B, with current values of \(a\) and \(b\). Use the equation shown in the discussion above. Look for \(Rbcurrent\).

- void findMinMaxDifference( ); For each row and column, calculate \(a - b\). Record the minimum value found in \(mMinDifference\) and the maximum value found in \(mMaxDifference\). Be sure to use getCurValue.

- virtual void setPixelSize( const int& width, const int& height ); Go to ComplexFractal.h and add virtual to the declaration of this method. Then, in this class, make this method call ComplexFractal::setPixelSize( width, height ); then resize mData's parts correctly. Only make changes if width and height are both greater than 0.

- virtual int calculatePixelEscapeCount( const int& row, const int& column ) const; Uses the process described above to calculate the pseudo escape values.

- virtual void calculateAllEscapeCounts( ); Go to ComplexFractal.h and add virtual to the declaration for this method. In this class, have the method call findMinMaxDifference, then call ComplexFractal::calculateAllEscapeCounts();

**Command**

Rename image_command.cpp to turing_command.cpp. Then add to it these options:

- -P // choose Turing pattern image
- -t dx,dt:alpha, beta:Da,Db // assign the Turing pattern parameters
- -I // randomly initialize the Turing pattern
- -U steps // update the Turing pattern steps times

You'll need to update the Makefile after changing the file name.

**Example Calls**

This command will create the following images:

```
./turing_command -T 256 -a 0:255,229,106:100:255,214,15 -a 101:255,214,15:160:206,137,0 -a 161:206,137,0:255:0.0 -p -p 1000,1000 -m 255 -p 100,100 -m 255 -t 1.0,0.001:-0.005,10.0:1.0,100.0 -I -f tp010.ppm -U 500 -f tp011.ppm -U 500 -f tp012.ppm -U 500 -f tp013.ppm -U 500 -f tp014.ppm -U 500 -f tp015.ppm -U 500 -f tp016.ppm
```
Note that due to random numbers, your actual pattern will look different than these.

**Additional Documentation/Hints**

- The `turing.cpp` file is an updated version of the menu based image generation. It might give examples of how to call methods of the `TuringPattern` class.
- [Turing patterns on Wikipedia](https://en.wikipedia.org/wiki/Turing_pattern)

**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.