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.

Let $A_{\text{previous}}(row, column)$ be the concentration of chemical species A at position $row$, $column$, at the end of the previous time step.

Let $A_{\text{current}}(row, column)$ be the current concentration of chemical species A at position $row$, $column$, at the end of this time step.

Let $\Delta A_{\text{row, column}}$ be the change in concentration of chemical species A at the position $row$, $column$.

Let $\text{Divergence}_{A}(row, column)$ be the divergence, or change over area, of concentration of chemical species A at position $row$, $column$.

Let $Da$ be the efficiency, or speed, with which chemical species A moves.

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

$$\Delta A_{\text{row, column}} = -4.0 \times A_{\text{previous}}(row, column) + A_{\text{previous}}(row - 1, column) + A_{\text{previous}}(row + 1, column) + A_{\text{previous}}(row, column - 1) + A_{\text{previous}}(row, column + 1)$$

Thus, $\Delta A$ is the sum of concentration differences between position $row$, $column$ and its four direct neighbors.

And the divergence (actual change) is

$$\text{Divergence}_{A}(row, column) = \Delta A / (dx^2)$$

Thus, it’s the $\Delta A$ spread over the area of the grid location.

The final concentration for the current time step will be

$$A_{\text{current}}(row, column) = A_{\text{previous}}(row, column) + dt \times Da \times \text{Divergence}_{A}(row, column)$$

In other words, the amount changes based on how much time, $dt$, how fast A is, $Da$, and how different the concentration is from its neighbors, $\text{Divergence}_{A}$.

#### 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. Just like those described above for A.
The patterns come when A and B react with each other. This can result in more A and B being 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).

Let \[ \alpha \] and \[ \beta \] be parameters that help describe the interaction between the two chemical species.

Let \[ Ap \] be short hand for \[ A_{\text{previous}}( \text{row, column} ) \], and \[ Bp \] be short hand for \[ B_{\text{previous}}( \text{row, column} ) \].

Let \[ Ra( Ap, Bp ) \] be the concentration of chemical species A that is produced when A and B interact at position \( \text{row, column} \). If the number is positive, then A was produced, and if it is negative, then A was consumed.

Let \[ Rb( Ap, Bp ) \] be the concentration of chemical species B that is produced when A and B interact at position \( \text{row, column} \). If the number is positive, then B was produced, and if it is negative, then B was consumed.

These are the equations we will use for \( Ra \) and \( Rb \).

\[
Ra( Ap, Bp ) = Ap - Ap \cdot Ap \cdot Ap - Bp + \alpha \\
Rb( Ap, Bp ) = ( Ap - Bp ) \cdot \beta
\]

We update the diffusion equations from the previous discussion to be reaction-diffusion equations now by adding in the reaction functions.

\[
A_{\text{current}}( \text{row, column} ) = A_{\text{previous}}( \text{row, column} ) + \Delta t \cdot Da \cdot \text{Divergence}A( \text{row, column} ) + \Delta t \cdot Ra( A_{\text{previous}}( \text{row, column} ), B_{\text{previous}}( \text{row, column} ) )
\]

\[
B_{\text{current}}( \text{row, column} ) = B_{\text{previous}}( \text{row, column} ) + \Delta t \cdot Db \cdot \text{Divergence}B( \text{row, column} ) + \Delta t \cdot Rb( A_{\text{previous}}( \text{row, column} ), B_{\text{previous}}( \text{row, column} ) )
\]

**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 Grid Numbers**

To create images, we need to calculate grid numbers to use as indexes into the color table by the \( \text{setPPM} \) 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:

\[
s = (d - \text{min}_d) / (\text{max}_d - \text{min}_d)
\]

Let \( \text{max}_n \) be the maximum grid number. The number \( n \) will be:

\[
n = 1 + (\text{max}_n - 2) \cdot s
\]

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

**Assignment**

In this assignment you will create a class, \( \text{TuringPattern} \), that inherits from the \( \text{ThreadedGrid} \) class. The purpose this class is to simulate the diffusion-reaction process of two chemical species.

The menu will have several new commands.

- \( \text{turing) Choose to make a Turing pattern.} \)
- \( \text{turing-parameters) Set the parameters of the Turing pattern.} \)
- \( \text{turing-randomize) Randomize the chemical species concentrations.} \)
Potential Session

# To run all of the commands from a script, throwing away the prompts
$ ./ppm_menu < ppm_menu_assignment_13_sample_session_a.txt >> /dev/null
$ ls -l *.ppm
-rw-r--r-- 1 cgl cgl 30015 Apr 13 14:43 ls010.ppm
-rw-r--r-- 1 cgl cgl 30015 Apr 13 14:43 ls011.ppm
-rw-r--r-- 1 cgl cgl 30015 Apr 13 14:43 ls012.ppm
-rw-r--r-- 1 cgl cgl 30015 Apr 13 14:43 ls013.ppm
-rw-r--r-- 1 cgl cgl 30015 Apr 13 14:43 ls014.ppm
-rw-r--r-- 1 cgl cgl 30015 Apr 13 14:43 ls015.ppm
-rw-r--r-- 1 cgl cgl 30015 Apr 13 14:43 ls016.ppm
-rw-r--r-- 1 cgl cgl 30015 Apr 13 14:43 ls017.ppm
-rw-r--r-- 1 cgl cgl 30015 Apr 13 14:43 ls018.ppm
-rw-r--r-- 1 cgl cgl 30015 Apr 13 14:43 ls019.ppm

Programming Requirements

Create [TuringPattern.h](#) and [TuringPattern.cpp](#)

These files will be used to declare and define the [TuringPattern](#) class.

**TuringPattern class**

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 value sets. 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 mA, mB; Parameters used by the Ra and Rb 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.

- TuringPattern(); Constructor chains to ThreadedGrid, then initializes data members mCurr(0), mPrev(1), mDx(1.0), mDt(0.001), mA(-0.005), mB(10.0), mDa(1.0), mDb(100.0), mMinDifference(1.0e10), mMaxDifference(-1.0e10). Also calls setGridSize() 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 dx, dt, alpha, beta, Da, and Db all have values greater than 0. alpha and beta may be positive or negative.

- 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 grid value indexes in the [NumberGrid](#) class.
The following functions will require updates to their functionality and/or declarations.

- `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 swapCurrPrev();` Swaps the values of `mCur` and `mPrev`.
- `void randomizeValues();` 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 `( ( std::rand() % big_value ) - ( big_value / 2 ) ) / ( 50. * big_value )`, where `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 calculateCurrValue( 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( const 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.
- `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.
- `void findMinMaxDifference();` For each row and column, calculate `a - b`. Record the minimum value found in `mMinDifference` and the maximum value found `mMaxDifference`. Be sure to use `getCurValue`.
- `virtual void setGridSize( const int& height, const int& width );` Make this method call `NumberGrid::setGridSize( height, width );` then resize `mData`'s parts correctly. Only make changes if `width` and `height` are both greater than 0.
- `virtual int calculateNumber( const int& row, const int& column ) const;` Uses the process described above to calculate the grid numbers from concentrations.
- `virtual void calculateAllNumbers();` Have the method call `findMinMaxDifference`, then call `ThreadedGrid::calculateAllNumbers();`

**Update** `image_menu.h` and `image_menu.cpp`

Add the following function declarations to the header file and implementations to the `.cpp` file.

- `void setTuringParameters( std::istream& is, std::ostream& os, NumberGrid& grid );` Verifies `grid` is actual a `TuringPattern` object, then asks the user for the double numbers "Parameter dx? ", "Parameter dt? ", "Parameter alpha? ", "Parameter beta? ", "Parameter Da? ", "Parameter Db? ", and calls `setParameters()` if it is. Otherwise gives an error message.
- `void randomizeTuringValues( std::istream& is, std::ostream& os, NumberGrid& grid );` Verifies `grid` is actual a `TuringPattern` object, then calls `randomizeValues()` if it is. Otherwise gives an error message.
- `void updateTuringValues( std::istream& is, std::ostream& os, NumberGrid& grid );` Verifies `grid` is actual a `TuringPattern` object, then asks the user for the integer number of "Steps? ", and calls `updateValues()` if it is. Otherwise gives an error message.
- `void calculateTuring( std::istream& is, std::ostream& os, NumberGrid& grid );` Verifies `grid` is actual a `TuringPattern` object, then calls `calculateAllNumbers()` if it is. Otherwise gives an error message.
- `void setJuliaFractal( std::istream& is, std::ostream& os, NumberGrid& grid );` Similar to `setTuringPattern` and `setMandelbrotFractal`, but for `TuringPattern`.

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

- `void showMenu( std::ostream& os );` Change/add to the menu to make the Turing pattern related options displayed as shown above.
void takeAction( std::istream& is, std::ostream& os, const std::string& choice, PPM& input_image1, PPM& input_image2, PPM& output_image, NumberGrid& grid, ColorTable& table); Add clauses to handle the commands listed above. Note that the “turing” command will have an empty clause here.

int imageMenu( std::istream& is, std::ostream& os ); Inside the while loop, after where you already check for “julia” and “mandelbrot”, add a check for “turing”, and call setTuringPattern if appropriate.

**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.
- The all target is expected in the Makefile.
- The clean target is expected in the Makefile.
- Automatic source and object file calculation are expected in the Makefile.
- Automatic dependency calculations are expected in the Makefile.
- The linker will need to know to add the thread library for consideration. This is usually accomplished with the LD_FLAGS make variable. See the class examples for a concrete example.

**Additional Documentation**

- C++ Reference
- Examples from class
- Sample Session A
- Sample Session B
- Turing patterns on Wikipedia
- Initialize Random Numbers in imageMenu().

**Sample PPM Images**

Output from a run of the “a” sample session.

- ls010.ppm
- ls011.ppm
- ls012.ppm
- ls013.ppm
- ls014.ppm
- ls015.ppm
- ls016.ppm
- ls017.ppm
- ls018.ppm
- ls019.ppm

Output from a run of the “b” sample session.

- zl010.ppm
- zl011.ppm
- zl012.ppm
- zl013.ppm
- zl014.ppm
- zl015.ppm
- zl016.ppm
- zl017.ppm
- zl018.ppm
- zl019.ppm

**Show Off Your Work**

To receive credit for this assignment, you must

- 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 additional command(s) that let the user select how many threads to use.
- Override the $\text{setPPM}$ method to be threaded.
- Are there other compute-resource bound parts of the code that take long enough to be worth the effort of converting to threaded?