NE 451

DEREK BENNEWIES | 20291856

FINAL PROJECT

DECEMBER 3, 2012

INSTRUCTOR: DR. MIKKO KARTTUNEN

Introduction

As computing power improves, numerical simulations of physical phenomena can be made increasingly sophisticated and powerful, and occasionally, they enable the discovery of entirely new, previously unobserved phenomena. Such is the case with Gray-Scott Diffusion, in which complex and often-unstable patterns arise from an initially near-homogeneous reaction mixture. These patterns are seen often in nature, from striping on zebras to patterns of growth in cell colonies.

While Turing first showed the mathematics of this pattern formation as early as sixty years ago, it took a further forty years before the particular mathematics of this reaction scheme were rigorously tested using computerized simulations [1][2]. Pearson's seminal work from 1993 was among the first to attempt to catalog all possible patterns arising from the Gray-Scott Diffusion scheme, which is as follows:

$$U + 2V \rightarrow 3V$$
$$V \rightarrow P$$

Which is an autocatalytic reaction scheme studied in depth by Gray and Scott [3]. In this scheme, reactant *U* continuously diffuses through a membrane into the reaction cell at some rate *F*, while *V* diffuses out of the system at some rate (*F* + *k*). This yields the following system of partial differential equations to model the concentrations of reaction components:

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u - uv^2 + F(1-u)$$
$$\frac{\partial v}{\partial t} = D_v \nabla^2 v + uv^2 - (F+k)v$$

In which *u* and *v* are the concentrations of *U* and *V*, and D_u and D_v are their diffusion coefficients, all respectively. The ∇^2 operator is the Laplacian.

Since Pearson's work, the literature on Gray-Scott Diffusion has expanded considerably, with many authors studying particular behaviors of the system at certain points in the (*F*, *k*) parameter space to better characterize each pattern in Pearson's catalog [4][5]. However, perhaps the most accessible and comprehensive exploration of parameter space is a map compiled by Mr. Robert Munafo and available on his website [6]. This map is shown in Figure 1 and includes most of the patterns that can form in the Gray-Scott Diffusion scheme.

Pearson's work established a convention for visualizing Gray-Scott systems in which high concentrations of *U* are colored red, while low concentrations of *U* are colored blue. A vocabulary has also grown to describe the observed patterns in two dimensions, including the following:

- Solitons – stable, low-*U* spots in a high-*U* field
- Negatons – stable, high-*U* spots in a low-*U* field
- Worms – contiguous curvilinear bands of either high-*U* (negative worms) or low-*U* (positive worms)
- U-Skate - a mobile, stable, U-shaped region of high-*U*

The code developed for this project provides a simple and highly usable means to explore this parameter space to identify and observe the evolution of the elements mentioned above. The next section will present the code and explain its usage, while the following sections will highlight and discuss some interesting results that it can generate.

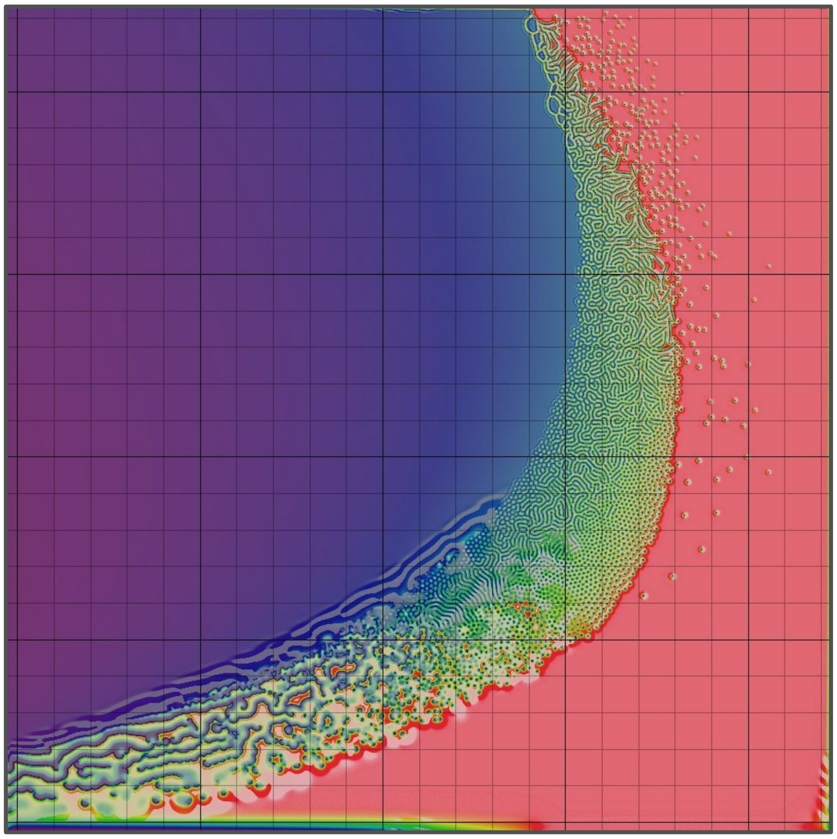

Fig 1. A map of obtainable patterns in (*F*, *k*) parameter space. *F* is the vertical dimension and ranges from 0.0020 (bottom) to 0.0860 (top). *k* is the horizontal dimension and ranges from 0.0310 (left) to 0.0730 (right) [6].

Code Usage and Structure

The code for this two-dimensional Gray-Scott Diffusion simulation is constructed in Python and consists of two primary files: `main.py` and `utils.py`. The main file contains the structure of the simulation and is the entry point for the user. The utilities file contains commonly-used functions in the simulation, such as the calculation of the Laplacian, and numerical integration schemes for the partial differential equations. For a quick introduction on how to use these files, the reader is directed to the *Usage* subsection. For a deeper understanding of how the simulation is structured and how passed arguments are applied, the reader is directed to the *Structure* subsection.

Dependencies

The simulation code depends on the following non-core Python libraries and software:

Python Libraries

- numpy – for mathematical constructs. Available at numpy.scipy.org, or with pip.
- matplotlib – for plotting images. Available at matplotlib.org, or with pip. Matplotlib has several dependencies of its own – see matplotlib.org for more information.

Other Software

- ffmpeg – for constructing videos. Available at ffmpeg.org.

Usage

To perform a simulation, the user need only call a single function, with as few as two arguments. To run a simulation directly from the console, navigate to the directory containing the simulation code and then enter Python. From there, the user can `import main` and then call the function `main.main`. This function is structured as follows:

`main.`**`main`**(*feed, k, feedmax=0, kmax=0, h=0.01, grid=256, perturb_num=1, perturb_u=0.5, perturb_v=0.25, perturb_mag=0.05, dt=0.5, timesteps=50000, out_freq=100, du=0.00002, dv=0.00001, framerate=20, laplacian=utils.lap5, visualize=False, integrate=utils.exp_euler, , dpi=None, saveims=False*)

In which *feed* and *k* are the only mandatory arguments, and arguments have the following meanings:

- feed – parameter *F* from the system of differential equations
- k – parameter *k* from the system of differential equations
- feedmax=0 – if set, will produce a reaction cell with a range of *F* and *k* values
- kmax=0 – if set, will produce a reaction cell with a range of *F* and *k* values
- h=0.01 – the unitless length step size for computing the Laplacian
- grid=256 – the number of pixels per dimension in the output images and movies
- perturb_num=1 – the number of perturbations from the homogeneous state to perform upon initialization
- perturb_u=0.5 – in the perturbations, the value at which to set *u*
- perturb_v=0.25 – in the perturbation, the value at which to set *v*
- perturb_mag=0.05 – the size of each dimension of the perturbation, as a proportion of the number of pixels in each dimension
- dt=0.5 – the unitless value of the timestep
- timesteps=50000 – the number of timesteps for which to run the simulation
- out_freq=100 – the period (in timesteps) at which to output a frame of video
- du=0.00002 – the diffusion coefficient D_u
- dv=0.00001 – the diffusion coefficient D_v
- framerate=20 – the number of frames per second in the output video
- lacplacian=utils.lap5 – the type of Laplacian stencil to use
- visualize=False – if True, displays video frames as the simulation runs
- integrate=utils.exp_euler – the type of numerical integration scheme to use
- dpi=None – the dots per inch (i.e. resolution) of the output images and video – a value of None provides a default dpi value of about 80
- saveims=False – if true, will keep the working folder containing still frames

The output from this function is an .mp4 video of the simulation, whose name includes the values of *feed* and *k*, and is placed in the current active directory. This video is a representation of the time evolution of the parameter *U* in which red areas are high in *U*, and blue areas are low. During simulation, progress is displayed (in terms of completed timesteps and estimated time remaining) in the console for convenience.

For multiple simulations, the user is encouraged to develop a script file that imports the `main` module and calls the above function several times. If the user intends to change the functions for the Laplacian or for temporal integration, note that the `utils` module must also be imported.

STRUCTURE

The simulation structure is discussed in *Main.py*. Some finer details on the numerics of the simulation are discussed in *Utils.py*

MAIN.PY

After setting up a working directory to store movie frames as they are generated, the simulation initializes a reaction cell. Within the reaction cell, the concentration of each reactant (*U* and *V*) is stored in its own numpy array. Initially, all values of *u* across the cell are set to one, and all values of *v* are set to 0.

Then, the perturbation is performed. The simulation randomly selects a corner point of the region to be perturbed and the size of this region based on user parameters. The simulation then perturbs the values of *u* and *v* in this region to the specified values, and introduces 1% random noise for symmetry-breaking. This perturbation process is repeated until the user-specified number of perturbations is reached.

The initialization function outputs the arrays of concentrations, *u* and *v*, which are then passed to the simulation function. After initializing plots and a timer for the simulation, the simulation function calculates the Laplacians of *u* and *v* as numpy arrays, and then performs numerical integration to determine the values of *u* and *v* at the next timestep. This process is repeated until the specified number of timesteps is reached. At regular intervals, a new image is generated using matplotlib and saved to the working directory (as well as on screen, if *visualize* is set to *True*), and progress is written to the console.

UTILS.PY

The utilities file contains small, often-used functions called by `main.py`. The first two functions are `lap5d` and `lap9d`, which are the traditional 5-point and 9-point stencils for calculating the Laplacian, as described in [7]. Both stencils have a cumulative error of $\mathcal{O}(h^2)$.

The second two functions, `u_fun` and `v_fun`, calculate the right side of the partial differential equations listed in the introduction. These are defined as functions in the utilities file rather than calculated directly in the main file so that they can each be passed to a variety of integrator functions.

The next three functions are integration algorithms. `exp_euler` implements the Explicit Euler method, commonly used in the literature despite its high cumulative error of $\mathcal{O}(h)$. Next is the midpoint method, `midpoint`, which has a smaller cumulative error of $\mathcal{O}(h^2)$. Finally, the fourth-order Runge-Kutta method, `rk4`, is included. This has a small cumulative error of $\mathcal{O}(h^4)$, but is more expensive to compute.

Results and Discussion

Literature Validation

In this section, the validity of the simulations generated using this code are examined, ensuring that the outputs are reasonable given the literature. The simplest way to perform such validation is to compare frames of video in the steady state of a simulation to the expected resulting structures given some *F* and *k* parameters, according to [6], as this is the most thorough mapping of parameters space available.

For the simulations below, optional arguments in `main.main` were left at default values unless explicitly stated otherwise, with the exception of *saveims*, which was always set to *True* in order to collect snapshots of the simulation.

Solitons – F=0.0260, K=0.0610

At these values, according to [6], solitons should form via "mitosis" to fill the entire reaction cell. As seen below in Figure 2, this is indeed the case with this simulation.

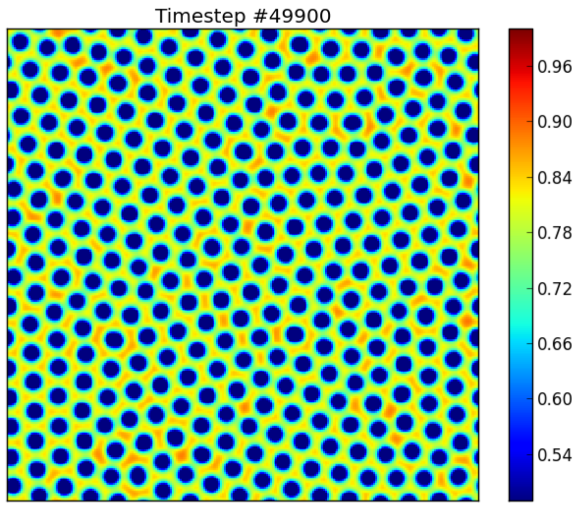

Fig 2. Solitons at (*F*, *k*) = (0.0260, 0.0610)

Wavefronts – F=0.0060, K=0.0410

Wavefronts are propagating bands of a dramatic *U*-concentration differences. At these parameters, according to [6], these bands should spontaneously form, propagate throughout the reaction cell, and annihilate each other. Figure 3, below, shows a snapshot of this behaviour. Using this code with default parameters, wavefronts will eventually converge to a blue (low-*U*) system, as the reaction cell is small enough to allow all wavefronts will annihilate each other.

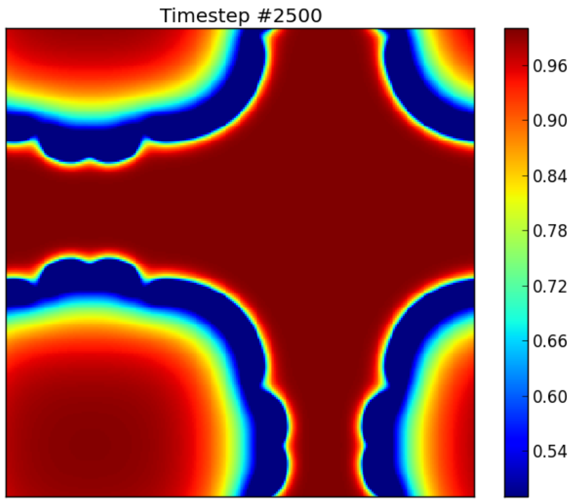

Fig 3. Propagating wavefronts at (*F*, *k*) = (0.0060, 0.0370). All wavefronts eventually annihilate one another.

Chaotic Oscillations – F=0.0220, K=0.0490

There should be no order or structure at these values. Instead, continuous, chaotic oscillations between high- and low-*U* should occur randomly throughout the reaction cell [6]. Figure 4 is a snapshot of this type of system.

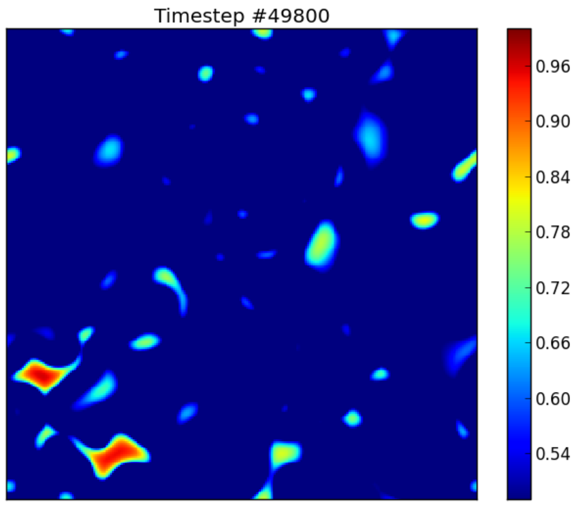

Fig 4. Chaotic oscillations at (*F*, *k*) = (0.0220, 0.0490)

Worms – F=0.0460, K=0.0630

This choice of *F* and *k* should yield a reaction cell completely filled with negative worms, which gradually straighten and lengthen over time [6]. This code is capable of producing such behaviour, as shown in Figure 5.

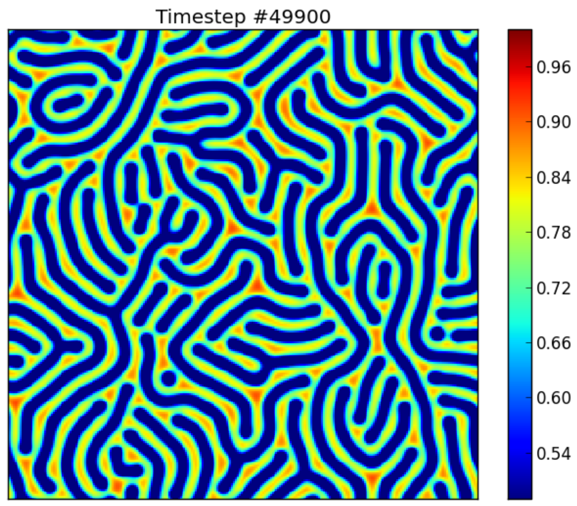

Fig 5. Worms at (*F*, *k*) = (0.0460, 0.0630)

NEGATONS – F=0.0620, K=0.0609

This point in parameter space can yield a combination of positive worms, negatons, and the rare u-skate phenomenon [6]. After a number of simulations at these parameters, negatons were indeed observed, and are shown below in Figure 6. Note the presence of positive worms as well.

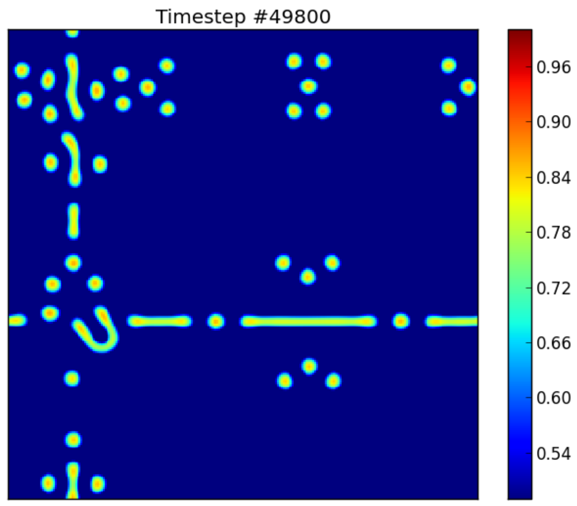

Fig 6. Negatons at (*F*, *k*) = (0.0620, 0.0610). Image taken after 50,000 timesteps.

U-Skate – F=0.0620, K=0.0609

Mr. Munafo dedicates an entire page to the behaviours he has observed at this point in parameter space. He identifies here u-skate and u-skate-like patterns that propagate throughout the reaction cell in modes reminiscent of "gliders" in the popular automaton "Conway's Game of Life" [6]. Despite repeated attempts with a variety of numerics, this package is unable to consistently generate u-skates. Figure 6 displays a single u-skate near the bottom-left corner, but the presence of this structure is atypical with this package. This is a significant drawback in this project and requires further investigation.

Mapping Parameter Space

To give a more thorough picture of the accuracy of this simulation package, a simulation was run over the parameter space specified in Figure 1. The resulting simulation, a snapshot of which is shown below in Figure 7, shows that the results are indeed remarkable similar. Note that the figure generated by this package appears to have a narrower band of patterns near the top of the image. This is likely due to poorly timed truncation of the simulation, as a qualitative analysis of the video shows that this region has not yet completed evolving, even after 100,000 timesteps. While a more complete map of parameter space would be ideal, 100,000 timesteps with the *grid* parameter set to 2048 already takes nearly 20 hours to run on a Core i5 processor.

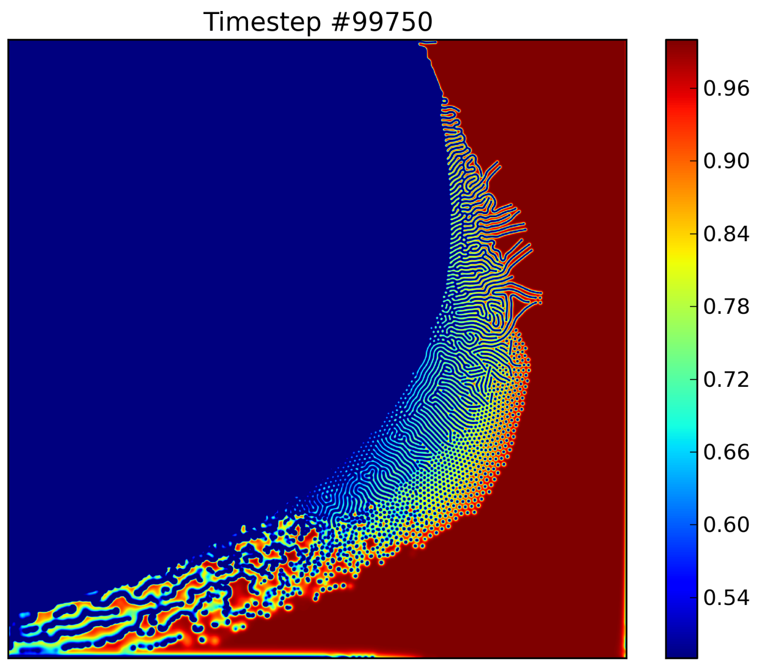

Fig 7. A map of parameter space similar to Figure 1. Non-default parameters include *timesteps*=100000, *grid*=2048, *dpi*=300, *out_freq*=250, *perturb_mag*=0.2.

Numerics

While many Gray-Scott diffusion simulations in the literature are executed using the explicit Euler method and the simple, 5-point Laplacian, it is possible to use alternative numerical integration schemes. Interestingly, no significant differences have been noted while using the various integration schemes (9-point Laplacian, 4th-order Runge Kutta, etc.) available with this project. To demonstrate such stability to varying numerics, Figure 7 shows 6 snapshots, taken at identical timesteps, at *F* = 0.0260 and *k* = 0.0610, which typically contains solitons, as shown in the previous section. It's apparent that numerics have no visible, qualitative effects on the resulting simulations.

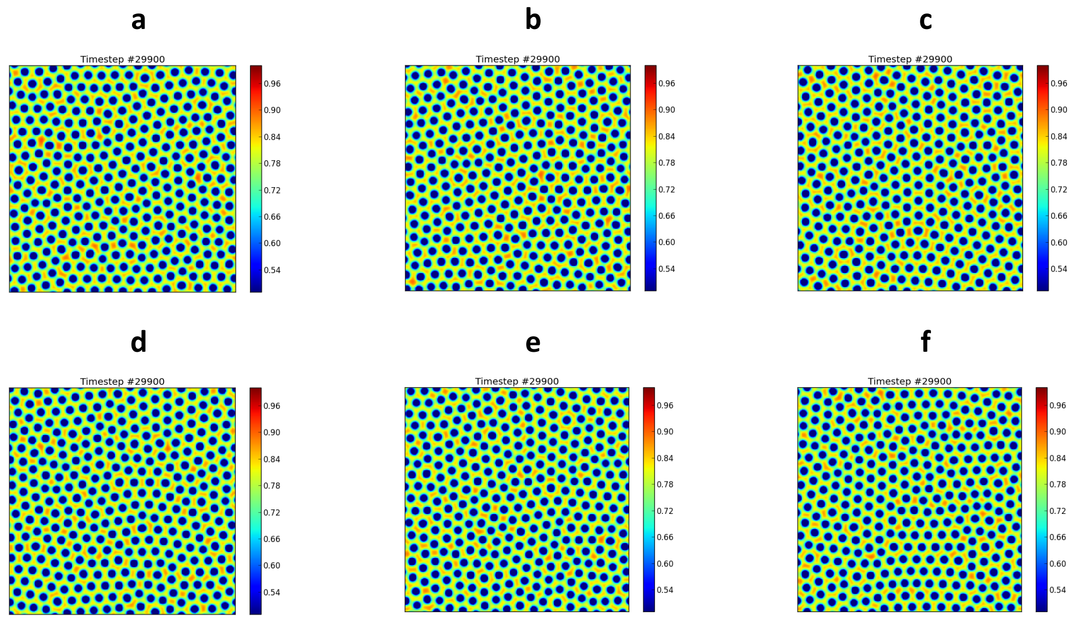

Fig 8. Stability to numerics at (*F*, *k*) = (0.0260, 0.0610). **a, b, c** use the 5-point Laplacian. **d, e, f** use the 9-point Laplacian. **a, d** use Explicit Euler. **b, e** use the Midpoint method. **c, f** use 4th-order Runge Kutta. All other optional arguments left at defaults.

CONCLUSIONS

SUMMARY

The Gray-Scott reaction system is a widely-studied autocatalytic reaction that provides insight into the formation of many complex biological structures, including the striping and spotting seen on many animals. This package is an easy-to-use implementation of a Gray-Scott reaction simulation, and is capable of producing time-evolution videos of a reaction cell given various input parameters.

As shown above, this package will yield most expected structures at the parameter values specified in the literature, with a few exceptions. However, to become a research-grade tool, some further development still needs to be completed.

WORK TO BE DONE

Based on the results achieved above, there are two major areas of work to be done regarding this simulation package. First is building a set of quantitative analysis tools. For example, functionality to determine the pair correlation function (i.e. g(r)) between solitons or negatons, given simulations that yield such structures. This sort of analysis can provide more meaningful feedback on how slight changes in parameters can affect the resultant simulation.

Secondly, more research needs to be done into u-skates. U-skates are amongst the most complex and highly-studied structures in Gray-Scott systems, yet this simulations package, as used in this report, has not been able to generate any such structures. Being able to generate u-skates at the expected parameters would be further validation for the accuracy of this simulation package, and open the door for interesting quantitative analysis, including the speed of such structures and their autocorrelation functions.

With a robust set of quantitative analysis tools, and verification that this package can indeed generate complex structures, simulations generated herein may become applicable to current research.

REFERENCES

[1] A. M. Turing, “The Chemical Basis of Morphogenesis,” *Society*, vol. 237, no. 641, pp. 37–72, 1952.

[2] J. E. Pearson, “Complex Patterns in a Simple System,” *Science*, vol. 261, no. 5118, pp. 189–192, 1993.

[3] P. Gray and S. K. Scott, “Autocatalytic reactions in the isothermal, continuous stirred tank reactor: Oscillations and instabilities in the system A + 2B → 3B; B → C,” *Chemical Engineering Science*, vol. 39, no. 6, pp. 1087–1097, 1984.

[4] M. P. Zorzano, D. Hochberg, and F. Moran, “Consequences of imperfect mixing the Gray-Scott model.,” *Physical Review E - Statistical, Nonlinear and Soft Matter Physics*, vol. 74, no. 5 Pt 2, p. 4, 2006.

[5] C. B. Muratov and V. V Osipov, “Spike autosolitons in the Gray-Scott model,” *Arxiv Preprint*, vol. 33, no. January, p. 96, 1998.

[6] Robert Munafo, “Reaction-Diffusion by the Gray-Scott Model: Pearson’s Parameterization,” 1996. [Online]. Available: http://mrob.com/pub/comp/xmorphia. [Accessed: 09-Nov-2012].

[7] M. Patra and M. Karttunen, “Stencils with isotropic discretization error for differential operators,” *Numerical Methods for Partial Differential Equations*, vol. 22, no. 4, pp. 936–953, 2006.

APPENDIX: CODE

MAIN.PY

```
### NE 451 Final Project - Gray-Scott Reaction/Diffusion ###

# Submitted by: Derek Bennewies
# Last Edited: November 29, 2012

import numpy as np
import random
import math
import os
import sys
import datetime
import shutil
import matplotlib.pyplot as plt
import utils
import time
import functools as ft


# Script entry point
def main(feed, k, feedmax=0, kmax=0, h=0.01, grid=256, perturb_num=1, perturb_u=0.5,
    perturb_v=0.25, perturb_mag=0.05, dt=0.5, timesteps=50000, out_freq=100, du=0.00002,
    dv=0.00001, framerate=20, laplacian=utils.lap5, visualize=False,
    integrate=utils.exp_euler, dpi=None, saveims=False):

    # Change to working directory for images and videos
    imgdir = str(datetime.datetime.today())
    os.mkdir(imgdir)
    os.chdir(imgdir)

    # Initiailize grid
    [grid_u, grid_v] = initialize(grid, perturb_num, perturb_u, perturb_v, perturb_mag)

    # Run simulation
    simulate(h, du, dv, grid, grid_u, grid_v, dt, timesteps, k, feed, out_freq,
laplacian, visualize, integrate, feedmax, kmax, dpi)

    # Create movie
    namestring = make_movie(framerate, k, feed)

    # Return to original directory and delete working directory
    if saveims == False:
        shutil.move('%s.mp4' % namestring, os.pardir)
        os.chdir(os.pardir)
        shutil.rmtree(imgdir)


# Initialize system
def initialize(grid, perturb_num, perturb_u, perturb_v, perturb_mag):

    # Initiailize the grid with all U
    grid_u = np.ones((grid, grid))
    grid_v = np.zeros((grid, grid))

    # Apply the specified number of randomly-sized perturbations
    for i in range(0, perturb_num):
        x_start = int(math.floor(random.random() * grid * 0.9))
        y_start = int(math.floor(random.random() * grid * 0.9))
        x_end = int(math.floor(x_start + (random.random() * grid + grid) * perturb_mag))
        y_end = int(math.floor(y_start + (random.random() * grid + grid) * perturb_mag))

        # Apply perturbations
        for x in range(x_start, x_end):
            for y in range(y_start, y_end):

                # Constant perturbation
                grid_u[y, x] = perturb_u
                grid_v[y, x] = perturb_v

                # Random perturbation
                perturb = 0.01 * random.random()
                grid_u[y, x] -= perturb
                grid_v[y, x] += perturb

    # Return grids
    return (grid_u, grid_v)


# Run simulation on initialized system
def simulate(h, du, dv, grid, grid_u, grid_v, dt, timesteps, k, feed, out_freq,
laplacian, visualize, integrate, feedmax, kmax, dpi):

    # If requested, initiailize live visualization
    if visualize == True:
        plt.ion()

    # Initiailize figure for movie frames
    fig, ax = plt.subplots()
    ax.axes.get_yaxis().set_visible(False)
    ax.axes.get_xaxis().set_visible(False)
    im = ax.imshow(grid_u, interpolation='nearest')
    fig.colorbar(im)

    # Initiailize time
    start_time = time.time()

    # Calculate h^2 (for speed)
    h2 = h * h

    # If requested, reconfigure feed and k as linspace matrices
    if feedmax != 0 or kmax != 0:
        feedlin = np.linspace(0.086, 0.002, num=grid)
        klin = np.linspace(0.031, 0.073, num=grid)
        k = np.tile(klin, (grid, 1))
        feed = np.tile(feedlin[:, np.newaxis], (1, grid))

    # Time loop
    for t in range(1, timesteps):

        # Calculate Laplacians
        lap_u = laplacian(grid_u, h2)
        lap_v = laplacian(grid_v, h2)

        # Calculate central term in integration calculations (for speed)
        grid_uvv = grid_u * grid_v * grid_v

        # Calculate new U and V values using selected integration method
        u_fun = ft.partial(utils.u_fun, du=du, lap_u=lap_u, grid_uvv=grid_uvv, feed=feed)
        v_fun = ft.partial(utils.v_fun, dv=dv, lap_v=lap_v, grid_uvv=grid_uvv, feed=feed,
k=k)
        grid_u += integrate(grid_u, dt, u_fun)
        grid_v += integrate(grid_v, dt, v_fun)

        # If it's the right timestep, produce output
        if t % out_freq == 0:

            # Generate the plot
            im.set_data(grid_u)
            plt.title('Timestep #%s' % t)

            # Save the figure as a png
            fig.savefig('img%06d.png' % (t / out_freq), dpi=dpi)

            # If requested, visualize the output as it's generated
            if visualize == True:
                fig.canvas.draw()

            # Calculate time remaining
            rem_time = (timesteps - t) * ((time.time() - start_time) / t)
            rem_hours = math.floor(rem_time / 3600)
            rem_mins = math.floor((rem_time % 3600) / 60)
            rem_secs = math.floor(rem_time % 60)

            # Write current progress to console
            sys.stdout.write('\r Timestep %s of %s is done, %02d:%02d:%02d remaining' %
(t, timesteps, rem_hours, rem_mins, rem_secs))
            sys.stdout.flush()

    # Newline for readability
    print '\n'


# Function to generate the movie out of output images
def make_movie(framerate, k, feed):

    # The name of the movie file
    namestring = 'GSD-k=%s-feed=%s' % (k, feed)

    # Compile the movie from the images
    moviestring = 'cat *.png | ffmpeg -f image2pipe -r %s -vcodec png -i - -vcodec
libx264 %s.mp4 > rubbish 2>&1' \
        % (framerate, namestring)
    os.system(moviestring)

    # Pass the filename back to main to move to original directory
    return namestring
```

UTILS.PY

```
### NE 451 Final Project - Gray-Scott Reaction/Diffusion ###

# Submitted by: Derek Bennewies
# Last Edited: November 29, 2012

import numpy as np

# Utility Functions #


# Compute the 5-point Laplacian with PBC - O(h^2)
def lap5(X, h2):

    # Compute the stencil matrices
    X_up = np.roll(X, 1, axis=1)
    X_down = np.roll(X, -1, axis=1)
    X_left = np.roll(X, 1, axis=0)
    X_right = np.roll(X, -1, axis=0)

    # Compute the Laplacian
    lap = (X_up + X_down + X_left + X_right - 4 * X) / h2

    return lap


# Compute the 9-point Laplacian with PBC - O(h^2)
def lap9(X, h2):

    # Compute adjacent the stencil matrices
    X_up = np.roll(X, 1, axis=1)
    X_down = np.roll(X, -1, axis=1)
    X_left = np.roll(X, 1, axis=0)
    X_right = np.roll(X, -1, axis=0)

    # Compute the diagonal stencil matrices
    X_r_up = np.roll(X_right, 1, axis=1)
    X_r_down = np.roll(X_right, -1, axis=1)
    X_l_up = np.roll(X_left, 1, axis=1)
    X_l_down = np.roll(X_left, -1, axis=1)

    # Compute the Laplacian
    lap = ((X_up + X_down + X_left + X_right) * 2 / 3 + \
        (X_l_up + X_l_down + X_r_up + X_r_down) * 1 / 6 - \
        X * 10 / 3) / h2

    return lap


# Evaluate the concentration of u given parameters
def u_fun(grid_u, lap_u, du, grid_uvv, feed):
    return du * lap_u - grid_uvv + feed * (1 - grid_u)


# Evaluate the concentration of v given parameters
def v_fun(grid_v, lap_v, grid_uvv, dv, feed, k):
    return dv * lap_v + grid_uvv - (feed + k) * grid_v


# Explicit Euler integration - O(h)
def exp_euler(y, dt, fun):
    return dt * fun(y)


# Midpoint Method integration - O(h^2)
def midpoint(y, dt, fun):

    k1 = dt * 0.5 * fun(y)

    return dt * fun(y + k1)
```

```
# Runge-Kutta integration - O(h^4)
def rk4(y, dt, fun):

    k1 = dt * fun(y)
    k2 = dt * fun(y + 0.5 * k1)
    k3 = dt * fun(y + 0.5 * k2)
    k4 = dt * fun(y + k3)

    return (k1 + 2 * k2 + 2 * k3 + k4) / 6.0
```