UNIVERSITY OF MANCHESTER

SCHOOL OF COMPUTER SCIENCE

THIRD YEAR PROJECT

Alan Turing's morphogenesis: on the wonders of nature

Author:
Vasilis NICOLAOU

Degree Programme:
MEng(Hons) Software Engineering

Supervisor:
Dr. Eva M. NAVARRO-LOPEZ

April 2013

Abstract

Alan Turing in 1952 described mathematically how cells can self-organise to form a variety of structures. He called this phenomenon morphogenesis, which means the creation of shape. Cells communicate with each other by diffusion, a process where chemical compounds or morphogens move from higher to lower concentrations. The project studies various aspects of morphogenesis and analyses several reaction-diffusion mathematical models. The main goal was to translate the simulations of such dynamical systems into descriptive animations of the processes involved. The project exploits morphogenesis exploring the phenomenon of cell mutation. Producing sound from reaction-diffusion mathematical models is also introduced. Finally, the project proposes applications of morphogenesis in engineering achieving the merging of the separate worlds of biology, mathematics, computer science, engineering and arts.

Title of project: Alan Turing's morphogenesis: on the wonders of nature.

Author: Vasilis Nicolaou

Supervisor: Eva M. Navarro Lopez

April 2013

Acknowledgements

I would like to thank my supervisor, Dr. Eva M. Navarro Lopez, for her academic wisdom, her caring manner and her constant help throughout the year which was essential for completing this project.

I also thank my friends and co-students for the funny moments and inspirational conversations throughout a challenging and difficult year.

Last but not least, I thank my family for believing and supporting me throughout my University studies.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Morphogenesis

Morphogenesis is the biological process that defines how a system of cells is organised and shaped. The word originates from the Greek words 'μορφή' which means shape and 'γένεσις' which means birth. Alan Turing in his paper 'The chemical basis of morphogenesis' gives a mathematical model that can approximate how an embryo develops its various organs [1].

The challenge was to show how from a single cell that replicates into identical cells, various organs are developed with different functionalities. For example, in a human embryo, there is initially a single cell that was produced from half the characteristics of the mother and half of the father. Eventually, millions of cells are generated. Nevertheless, they manage to differentiate into groups that form parts of different organs.

Turing defined a hypothetical chemical substance called morphogen. A morphogen is an abstract term which represents a chemical compound that gives the cell certain properties according to its concentration. A cell, therefore, may be defined as a vector of morphogens that react together forming the state of a cell. In addition, each cell has several neighbouring/adjacent cells.

When cells with different concentrations of chemical compounds are connected, the phenomenon of diffusion is observed. The membrane of a cell has a property that allows chemical substances to move from higher to lower concentrations. This is how cells communicate. Diffusion is further explained in Section 2.2.

By constructing mathematical equations that approximate chemical reactions and diffusion, one can define models that approximate the phenomenon of morphogenesis. Such models are described in Chapter 4.

1.2 Project objectives

This project studies the idea of morphogenesis that Alan Turing proposed and subsequently produces software applications that simulate various models of the reaction-diffusion biological process. The simulations combine graphs that may be used for analysis or testing and movies[1] that show how cells are organised forming structural patterns. Simulations also visualise the effects of cell mutations, processed as random changes in morphogen concentrations after a system develops a structured shape.

[1] Movies are available at: `http://www.youtube.com/watch?v=2JppD_Mw_3k`

The project goes a step further and introduces applications of morphogenesis in engineering. A process scheduling algorithm inspired by the reaction-diffusion concept has been developed and compared to traditional schedulers such as the Round Robin. The idea of how a scheduler is related to morphogenesis is not obvious and is described in section 5.2.3.

Some experiments study the possibility of producing sound by the use of mathematical models that produce continuous oscillations. Due time limitations, experiments are not extensive and no meaningful sound output was produced. However, readers and future researches of the subject are encouraged into further study of such experiments, since graphs present a behaviour similar to sound waves.

1.3 Background

Morphogenesis is defined by Alan Turing as the process of how cells of an embryo create shape out of an initially chaotic behaviour [1]. Cells are interacting with each other by diffusing chemical substances which in Turing's morphogenesis model are defined as morphogens. The term morphogen is abstract and should not be confused with DNA genes, chromosomes or certain chemical elements. A morphogen is considered to be a chemical substance that has some theoretical properties which give to cell specific characteristics. Morphogens may also react with each other in the same way chemical molecules do. Reactions and diffusion happen over time and thus morphogenesis process can be a dynamical system. Thus, it can be modelled mathematically using differential equations, by suggesting that each cell has a state according to its morphogen concentrations and that state is subject to change over time.

The approach followed for simulation and analysis of morphogenesis' models uses a variety of concepts of the mathematical, biochemical and programming domain. Such concepts are discussed in this report providing the knowledge basis that is necessary for the reader to follow the approach and the implementation procedure of the project.

Finally, various papers and articles [2][3][4] that study the phenomenon of morphogenesis and pattern generation were used to identify mathematical models of the reaction-diffusion process and were the spark for further exploration of ideas related to morphogenesis.

1.4 Approach

The approach was partitioned in several stages. Since the project has a strong mathematical component, background knowledge of the mathematical concepts that were involved are presented. The information on the mathematical knowledge that is required is given by Alan Turing in his paper [1]. To sum up, this subject merges ordinary differential equations, integration methods, algebra and computational theory.

Application of mathematical concepts from dynamical systems theory is required to build an understanding on how software applications of mathematical models are developed. Since the main tool to implement such models was Matlab, suitable software design methodologies were identified to facilitate a concrete development of the software applications.

An agile development methodology was followed [5]. A basic application was built at an early stage. This enabled concurrency in researching and programming which resulted in identifying and implementing features, re-factoring code and fixing bugs, achieving the implementation of the variety of applications to be well defined, maintainable and reasonably documented.

In brief, the project is researched based, involving various concepts from chemistry, biology, mathematics and software design. The absence of requirement gathering from users enabled the intense research and tutorial discussions to embrace the study of various ideas that were presented throughout the life-cycle of the project.

1.5 Summary of results

Results are separated into categories according to the purpose of each application. Information about chemical reactions and diffusion is provided. The data from the applications and experiments that took place are discussed. After that, various mathematical models are simulated for studying their parameters and how the latter affect their behaviour. Finally, experimental ideas that involve cell mutation, sound producing and process scheduling using diffusion are introduced.

The presentation of results shows at which circumstances cells form structures and how different shapes are produced by altering the parameters of a model or by using different models. Furthermore, perturbations are introduced in the form of cell mutations and explore whether the structure of the system is altered. Finally, the possibility of producing sound is introduced along with a study on how morphogenesis might be applied in computer science as a nature inspired algorithm.

1.6 Report structure

The report consists of three parts.

1. The background information for the reader to build the basic concepts involved in this project.

2. The applications that were implemented and the results obtained in each stage of the project.

3. The development process and methodologies that were followed to build the software artifacts. Thus, offering a point of reference to the reader for building extensions or similar solutions.

The main morphogenesis models are supplied in Chapter 4. Those are the L-systems equations, the Gray-Scott model and the Gizburg-Landau model. The background information provides all the necessary knowledge on the concepts that are studied by the project. The mathematical background is the most important and advanced topic, since all the biochemical related issues, such as the chemical reactions and cell diffusion are described by differential equations. Thus, Chapter 2 consists of basic information about how chemical compounds react and what assumptions are made for both chemical reactions and diffusion when defining a mathematical model that approximates such behaviours. Finally, the programming background is given, with information on Matlab and Java, including key features of each language and pseudo-code of the algorithms that were implemented.

The development process that is discussed in Chapter 5 gives the methodology used to implement each software application and help in reading the code for the purpose of maintaining or expanding it. It also includes information about testing and visualising data by using script

languages that enable automation and minimise the amount of time spent by combining plotting and testing.

Results are given in Chapter 6 and are presented in a way that reflects the progress of the project, starting from the study of chemical reactions and cell diffusion and then progressing to complete mathematical models of morphogenesis. At the end, the results of experimenting with process management combined with morphogenesis are supplied. Results include images that show snapshots of how the reaction-diffusion process evolves in time. Graphs showing the numerical state of a system or statistics about comparison of algorithms are also given.

Finally, the appendices contain the important code implementations of the project. Since various papers on morphogenesis [3][4] contain examples of their code artifacts, it is important that alternative solutions are given.

Chapter 2

Biochemical concepts

An elementary level of knowledge in chemistry is needed to understand the concepts that are discussed in this report. As mentioned previously, the reactions do not include real chemical elements, but rather some abstract compounds called morphogens. The process however, is the same as when real chemical elements react. This enables the use of various reaction equations with different rates and therefore different models, that widen the range of the experiments in order to approximate physical systems.

2.1 Chemical reactions

A reaction is defined as the interaction of a set of two or more chemical compounds to produce another, different set of one or more chemical compounds. The equation to describe such a process is:

$$X + Y \underset{r}{\rightarrow} Z \tag{2.1}$$

Equation (2.1) means, that the chemical compound X is reacting with the compound Y and both produce Z at a rate r. Thus, when one of the concentrations of either X or Y runs out, the reaction stops. From the chemical equation that describes a reaction, a mathematical description may be constructed. Since reaction depends on time (the concentration of Z is increased over time according to a rate r, X and Y concentrations are decreased at the same rate), a chemical reaction should be considered as a dynamical system. As described in section 3.1 a dynamical system is defined by ordinary differential equations.

The system of equation (2.1) has 3 states, X, Y and Z. Therefore, three equations are needed to describe each state of the system in time. Z is produced at a rate r. In addition, concentrations of X and Y that are decreased in the same rate (for instance, one unit of both X and Y is needed to produce one unit of Z). The equation that gives the state of Z at any time, is:

$$Z = rXYt + Z_0 \tag{2.2}$$

Likewise for X and Y with the main difference that those compounds are expended to produce Z, so the equations are:

$$X = -rXYt + X_0 \tag{2.3}$$

$$Y = -rXYt + Y_0 \tag{2.4}$$

Z_0, X_0 and Y_0 are the initial concentrations of each chemical substance and r is the parameter that represents the reaction rate. Differentiating equations (2.2), (2.3) and (2.4) the

mathematical model is obtained as:

$$\frac{dZ}{dt} = rXY$$
$$\frac{dX}{dt} = -rXY$$
$$\frac{dY}{dt} = -rXY$$

2.2 Cell diffusion

Diffusion is the process where chemical substances move from their higher to lower concentrations. For example, if two cells A and B are attached, containing 10 units and 2 units respectively of a chemical X, then the chemical compound X will move from A to B until the chemical concentrations are balanced (6 units each). Turing suggested in his paper that the diffusion occurs similarly with the conduction of heat [1, p. 40]. By replacing the conduction with diffusion, Turing gave the equation of diffusion in a ring of cells [1, p. 47] from which a general equation of diffusion can be extracted as shown in equation (2.5). The latter helps in experimenting with cells that have any number of cells attached to them.

$$F = \delta^2 K .* (\sum_{i=a}^{A} x_i - N x_j) \tag{2.5}$$

Where:

- A is the set of cells that are directly attached to cell j,
- N is the size of A (the number of cells attached to j),
- j is a cell of the system under study and it does not exist in set A,
- x_i is a vector that contains the concentration of the chemical substances in cell i,
- K is a vector which contains diffusion coefficients for each chemical substance,
- δ is the size of the sides of the cells.

The binary operation '.*' means that the vector multiplication is pair-wise (adopted from Matlab semantics [11]). That is:

$$\begin{pmatrix} a \\ b \end{pmatrix} .* \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} ac \\ bd \end{pmatrix}$$

Turing considers the scenario of cells connected in a ring communicating with diffusion. That means that every cell is connected to another two cells as shown in Figure 2.1. Therefore, the equation becomes:

$$F_{ring} = \delta^2 * K .* (x_{i-1} - 2x_i + x_{i+1}) \tag{2.6}$$

Diffusion in a torus is also considered in order to tackle the problem that arises by working on a surface structure. For instance, consider a 2-dimensional matrix. The cells on the edges are connected with three other cells. If a cylinder connection is made, that leaves the top and bottom cells to be connected with only three other cells and all others to be connected with four. The solution is to connect every cell on an edge with a cell in the other edge-end. This

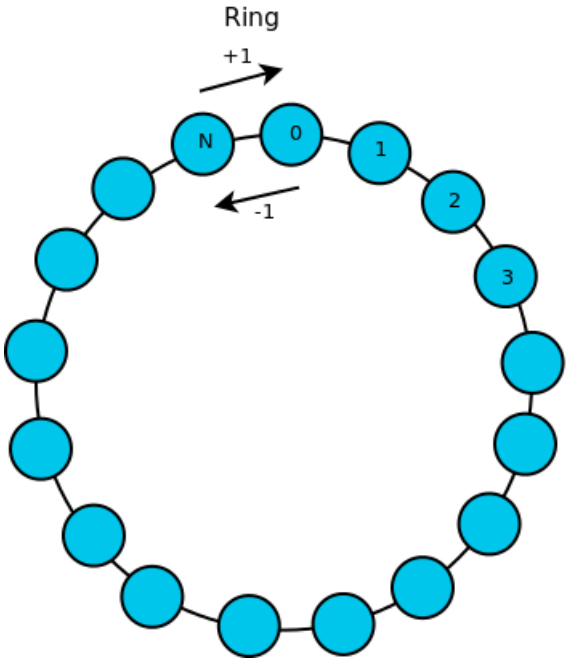

Figure 2.1: Interconnection of cells in a ring-shaped structure.

forms the shape of torus shown in Figure 2.2. Another advantage by the use of a torus structure is that it allows a straightforward way of converting a surface of cells into an image, since it is easily representable by using 2-dimensional computer graphics.

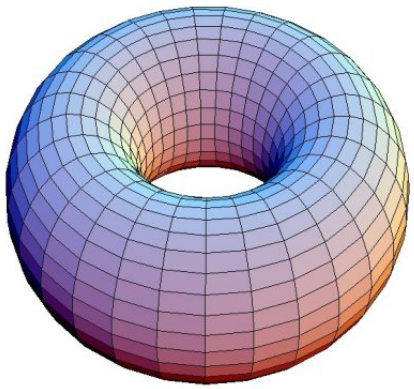

Figure 2.2: 2-dimensional surface wrapped in a torus-shaped structure.

Chapter 3

Mathematical concepts

This chapter gives information about the mathematical concepts that were used during this project. A basic knowledge of ordinary differential equations (ODE's) and matrix algebra is needed. Main concepts such as dynamical systems are explained. Linear and non-linear systems are also discussed, including the linearisation technique.

3.1 Dynamical systems

A dynamical system gives a functional description of a solution which varies in time. Mathematically a dynamical system is a function $f(t,x) \forall t \in \mathbb{R}$ and $x \in \mathrm{E} \subset \mathbb{R}^n$, that describes how points $x \in E$ change with respect to time [8, p. 182].

A mathematical definition to the problem based on dynamical systems is needed in order to construct the mathematical model that describes the behaviour of the phenomenon of Morphogenesis.

Morphogenesis is the process that gives shape to biological organisms. Mathematically a biological system may be defined as a set of cells which contain a set of morphogens. Thus, a cell is defined as a vector containing real values that represent the morphogen concentrations. Therefore, the functional model will be defined as a matrix of N vectors of size M, where N is the number of cells and M is the number of morphogens that each cell contains. This is shown in Table 3.1.

Table 3.1: Mathematical representation of biological elements.

Biological element	Mathematical Representation	Mathematical Definition
Morphogen	A real value variable that represents the concentration of the morphogen	$X \in \mathbb{R}$
Cell	Vector of size M equal to the number of morphogens that it contains	$x \in \mathbb{R}^M$
Physical System	A matrix of size N equal to the number of cells that contains M 1-dimensional vectors of real values	$S \in \mathbb{R}^{N \times M}$

3.1.1 Non-linear systems

The purpose of a mathematical model is the aid to the study of physical systems. It should be noted, that any mathematical model gives an approximate solution of the physical system it describes. Therefore, the accuracy of any mathematical model is measured by the closeness of that model to the behaviour of that physical system. Such models are constructed by a set of non-linear differential equations.

This project focuses on the study of known mathematical models by analysing and simulating their behaviour. Thus, some concepts of ordinary differential equations, integration algorithms and numerical stability theory are required.

The format of non-linear functions studied in this project is:

$$\frac{dx}{dt} = F(x(t)) = F(x_1, ..., x_n)$$

Where:

- x is a vector of size n.
- n is the number of morphogens.
- F is a non-linear function.

Such systems approximate the internal chemical reactions in each cell. To have a complete reaction-diffusion model, the diffusion equation is added to the non-linear equations. Then the mathematical model becomes:

$$\frac{dx_i}{dt} = F(x_i(t)) = F(x_i) + \delta(\sum_{j=a}^{A} x_j - Nx_i) \tag{3.1}$$

$\forall i \in$ `Cells`.

The construction of the diffusion equation is described in Section 2.2.

Analysis

There are two basic ideas concerning the analysis of non-linear systems; stability and oscillation. Stability is associated with the behaviour of the system around the equilibrium points. The main interest is to penetrate the system with small or large perturbations at its equilibrium points and observe if its state changes [9]. Such perturbations are introduced in the form of mutation by altering the morphogen concentrations with random values. Other mutations of great interest involve altering the parameters or the diffusion coefficients in the models of morphogenesis.

In terms of oscillating, the aim is to detect if the system shows an oscillatory behaviour and analyse what properties that oscillation has: periodic/aperiodic, frequency, amplitude and how long does it take for the system to stop oscillating and reach an equilibrium state [9, p. 7].

Simulation

Simulation means that the mathematical model is integrated in time and the results are shown visually. Graphs, bar charts and finally movies show a representation of how cells are organised into shaped structures by converting the morphogen concentrations into colours.

The integration of such models is done by programming in Matlab and the Java programming language. In order to develop solutions -in respect to mathematical models- in Matlab and Java, knowledge of integration algorithms and numerical stability is required.

The goal of the simulations is not only to show graphically how the state of a system changes, but also to prove that cells create shapes and ordered structures out of a chaotic and random initial state, according to the mathematical model that describes such a system.

Summarising, the simulations result from the integration of ordinary differential equations that define the mathematical model under study and then converting the solution into human understandable visualisations.

3.1.2 Linear systems

This project also studies linear systems of ordinary differential equations of the form:

$$\dot{x} = Ax = A \begin{pmatrix} \dot{x}_1 \\ \vdots \\ \dot{x}_n \end{pmatrix}$$

where $A \in \mathbb{R}^{n \times n}$ and $x \in \mathbb{R}^n$.

In general, a linear system can be viewed as

$$\dot{x} = F(x)$$

where $F(x)$ is a linear function. The reason behind the transformation to $\dot{x} = Ax$ is to exploit the properties of any linear function. Those are the superposition property

$$F(x+y) = f(x) + f(y)$$

and the homogeneity

$$F(ax) = aF(x)$$

By substituting F(x) by Ax it is directly shown that those properties are satisfied since:

$$F(x) = Ax = \begin{pmatrix} a_{11}x_1 + \ldots + a_{1n}x_n \\ \vdots \\ a_{n1}x_1 + \ldots + a_{nn}x_n \end{pmatrix} = \begin{pmatrix} a_{11}x_1 \\ \vdots \\ a_{n1}x_1 \end{pmatrix} + \ldots + \begin{pmatrix} a_{1n}x_n \\ \vdots \\ a_{nn}x_n \end{pmatrix} = c \begin{pmatrix} a'_{11}x_1 + \ldots + a'_{1n}x_n \\ \vdots \\ a'_{n1}x_1 + \ldots + a'_{nn}x_n \end{pmatrix}$$

Linearisation process

It is possible to approximate a non-linear system around its equilibrium points by exploiting a property that most non-linear systems have. That property states that in the neighbourhood of equilibrium solutions, non-linear systems behave as the linear system that approximates them. This means that a similar behaviour may be retrieved (for example, oscillatory behaviour with certain amplitude and frequency), but the numerical values do not represent the values that the non-linear system would have at that certain points in time. For example, the results of linear systems in Section 6.3.1 show that there are negative values representing chemical concentrations. This is not a problem, since only the behaviour is studied when approximating a system by linearisation.

The equilibrium points of the system are the values of vector $x \in \mathbb{R}^n$ at which $\frac{dx}{dt} = 0$. When all the derivatives of a system are evaluated to 0 then the system is unable to change as time passes, since the integration step will always add zero. Exceptions are not studied.

The linearisation technique process as given from Florin Diacu [10, p. 227] is:

1. Shift equilibrium points to the origin, and define new variables, such as:

$$u = x - x_0$$

2. Substitute the variables of the system with the new variables u from step 1 and obtain the equations for $(\dot{u}_1, \ldots, \dot{u}_n)^T$.

3. Define G as the new form of the vector field such that

$$\dot{u} = G(u)$$

The origin is an equilibrium such that:

$$G(0) = 0$$

The partial-derivative matrix A is formed as:

$$A = \begin{pmatrix} \frac{\partial G_1}{\partial u_1}|_{u=0} & \cdots & \frac{\partial G_1}{\partial u_n}|_{u=0} \\ \vdots & \ddots & \vdots \\ \frac{\partial G_n}{\partial u_1}|_{u=0} & \cdots & \frac{\partial G_n}{\partial u_n}|_{u=0} \end{pmatrix}$$

4. Apply the derivatives shown in the corresponding matrix A (step 3) in terms of u_i.

Example Let the non-linear system be:

$$\dot{x}_1 = (x_1 - 3)(x_2 - 1) \tag{3.2}$$

$$\dot{x}_2 = (x_1 + 2)(x_2 + 5) \tag{3.3}$$

The first step is to find the equilibrium point where $\dot{x}_1 = \dot{x}_2 = 0$. It is easily observable that with $x_1 = -2$ and $x_2 = 1$ then

$$\dot{x}_1 = \dot{x}_2 = 0 \tag{3.4}$$

A linearised system can approximate the behaviour of a non-linear system near the equilibrium points in (3.4). Thus, an isolated equilibrium should be used to shift the equilibrium points to the origin.

To achieve this, define:

$$u = x - x_0 \tag{3.5}$$

x_0 being the equilibrium points of the system.

Equation (3.5) is the general form to work with the two variables of the system. Applying (3.5) on (3.2) and (3.3), the two shifting variables are obtained:

$$u_1 = x_1 + 2 \tag{3.6}$$

$$u_2 = x_2 - 1 \tag{3.7}$$

Using (3.6) and (3.7) on (3.2) and (3.3) the new system with the altered coordinates is obtained:

$$\dot{u}_1 = \dot{x}_1 \text{ and } \dot{u}_2 = \dot{x}_2 \iff$$

$$\dot{u}_1 = -5u_2 + u_1u_2,$$
$$\dot{u}_2 = 6u_1 + u_1u_2$$

with

$$G = \begin{pmatrix} \dot{u}_1 \\ \dot{u}_2 \end{pmatrix}$$

Following the steps 3 and 4 of the linearisation technique, the real values of the corresponding matrix A are computed:

$$A = \begin{pmatrix} \frac{\partial G_1}{\partial u_1}|_{u=0} & \frac{\partial G_1}{\partial u_2}|_{u=0} \\ \frac{\partial G_2}{\partial u_1}|_{u=0} & \frac{\partial G_2}{\partial u_2}|_{u=0} \end{pmatrix} = \begin{pmatrix} \frac{\partial(-5u_2+u_1u_2)}{\partial u_1}|_{u=0} & \frac{\partial(-5u_2+u_1u_2)}{\partial u_2}|_{u=0} \\ \frac{\partial(6u_1+u_1u_2)}{\partial u_1}|_{u=0} & \frac{\partial(6u_1+u_1u_2)}{\partial u_2}|_{u=0} \end{pmatrix} \Rightarrow A = \begin{pmatrix} 0 & -5 \\ 6 & 0 \end{pmatrix}$$

The linearized system around $(u_1, u_2) = (0,0)$ is $\dot{u} = Au$, with

$$u = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$$

This system, approximates the behaviour of the non-linear system defined by the equations (3.2) and (3.3)

3.2 Numerical stability

A system is numerically unstable when its state grows uncontrollably to infinity. Instability may occur for three reasons.

1. The system is unstable.
2. The system is unstable for the particular initial conditions and parameters.
3. Integration flaws of the ODE solver show that the system is unstable when it is not.

The interest of this project lies on reasons 2 and 3. This is because the construction of a mathematical model does not concern this project, and thus, already proposed mathematical models are used.

Numerical stability is a very important aspect to consider when using an ODE solver algorithm. Depending on the stiffness of the differential equations, a decision for the length of the time step must be made. The decision of the time step is crucial when having oscillating systems. This is due to the fact that, if the time step is not small enough and the amplitude grows with large errors on each step then the amplitude of that oscillation will grow to infinity resulting in an unstable system.

In order to have a better understanding of why it is important to decide on the right time-step, consider the following situation: Think of a labyrinth with arbitrary length paths and walls of arbitrary width. Someone wants to pass through the labyrinth by making steps of arbitrary size. After a step, he observes if he arrived at a wall-end. If the step is big, it could pass mistakenly through a wall, without noticing (in a virtual world) or crash at it (in a real world). How should the decision be made about the right size of the step? The example here is of a step in space, but the problem is of the same nature: if the step is big, the solver will present a wrong or unstable behaviour.

On the other hand, having a very small step, increases the time that an ODE solver needs to complete the integration. In the labyrinth example, doing the smallest possible step each time to avoid wall collision, will result in a correct solution, but the time complexity would be huge. A dynamic approach, however, introduces variations on the time steps. An algorithm that changes the time step is called implicit and an algorithm that uses a constant time step is called explicit. Most Matlab ODE solvers use an implicit approach of the Runge-Kutta family algorithms [11], reducing the time-steps when, for example, oscillations have low amplitude.

The Java implementation uses the explicit implementation of the Euler method since the range of the numbers is known, as the time slices that are given to a process by a scheduler are fixed. In addition, float point underflows are avoided by comparing the time slice given to a process with the smallest execution time-step a process is allowed to make.

Chapter 4

Models of morphogenesis

This project studies already-known reaction-diffusion models to produce different patterns and explore various ideas and concepts. The Gray-Scott model and the L-Systems equations are the core of the visualisation applications. The applications integrate those models and then by mapping the computed values into colours, images are generated and show the process of the cells producing a wide variety of structures.

4.1 The Gray-Scott model

The Gray-Scott model describes a reaction-diffusion system [6]. The original model involves partial derivatives and is given by the two partial differential equations:

$$\begin{aligned} \dot{u} &= d_u \bigtriangledown u - uv^2 + F(1-u), \\ \dot{v} &= d_v \bigtriangledown v + uv^2 - (F+k)v \end{aligned} \tag{4.1}$$

Where:

- u and v represent the morphogen concentrations.
- d_u and d_v are the diffusion coefficients of the morphogens u and v respectively.
- F and k are real constant values which represent the reaction rate.
- $\bigtriangledown$ is the laplacian operator of u for calculating diffusion in Euclidian space [3].

The analysis given by Benjamin M. Heineike in [3] finds several parameter ranges and values to produce several patterns. The equations were modified for the purposes of this project; the laplacian of the concentrations in respect to space was substituted by a linear diffusion equation as described in Section 2.2. Reaction functionsremain exactly the same as defined by Gray-Scott in equations (4.1).

$$\begin{aligned} U + 2V &\underset{F}{\rightarrow} 3V, \\ V &\underset{F+k}{\rightarrow} P \end{aligned} \tag{4.2}$$

Thus, from equation (4.1), replacing the laplacian operator by the diffusion equation (2.5) the model becomes:

$$\dot{u}_j = d_u((\sum_{i=a}^{A} u_i - Nu_j)) - u_j v_j^2 + F(1-u_j),$$
$$\dot{v}_j = d_v(\sum_{i=a}^{A} v_i - Nv_j)v + uv^2 - (F+k)v_j, \tag{4.3}$$
$$\forall j \in Cells$$

Where:

- A is the set of adjacent cells to the current cell j.
- N is the number of elements of the set A.

If, for example, the structure of the cell interconnections is a ring then the equations (4.3) become:

$$\dot{u}_i = d_u(u_{leftof(i)} - 2u_i + u_{rightof(i)}) - u_i v_i^2 + F(1-u_i),$$
$$\dot{v}_i = d_v(v_{leftof(i)} - 2v_i + v_{rightof(i)}) + u_i v_i^2 - (F+k)v_i, \tag{4.4}$$
$$\forall i \in Cells$$

The aim is to test the model with different parameters that were introduced by Benjamin M. Heineike [3] and discover if the Gray-Scott reaction function can be combined with diffusion, according to the conduction of heat as proposed by Alan Turing [1], to generate patterns. The results are presented in Section 6.3.2.

4.2 The L-Systems equations

The L-Systems equations are proposed by Christopher G. Jennings [7]. His work involves a Java applet that generates patterns. Examples of such patterns involve cheetah skin patterns and fingerprints. The Java applet uses the explicit Euler's method to integrate the reaction equations and the diffusion. The reaction equations are:

$$\dot{u} = uv - u - 12,$$
$$\dot{v} = 16 - uv \tag{4.5}$$

The same concepts were used and expanded into testing different ideas. The first idea was to create patterns by applying different colour mappings and image processing techniques. The second was to study how cell mutation can affect the normal structure of the system. Another aspect was to observe if the Euler's method used in [7] was indeed accurate and what were the limitations in terms of the produced patterns and the numerical computations.

Results are shown in paragraph 6.3.2 comparing the L-Systems equations to the Gray-Scott reaction function.

4.3 The Gizburg-Landau model

The Gizburg-Landau model consists of only one equation. Its continuous oscillatory behaviour makes it ideal for studying the possibility of producing sound. Unfortunately, time limitations did not enable research into finding a suitable way for mapping morphogen concentrations to sound spectra. Nevertheless, it might be possible to generate sound since the graphs that were generated look very similar to sound wave patterns. Such a graph is shown in paragraph 6.4.

The equation is:

$$\dot{u} = (1+ib) \bigtriangledown^2 u - (1+ia)u|u|^2 \tag{4.6}$$

Where:

- u is a vector of cells (each containing a single morphogen).
- a and b are real constants.
- i is the imaginary unit.

The equation (4.6) is a partial differential equation. The solution is given by Aly-Khan Kassam [4]. The aim was to find the correct mapping to exploit the long wave oscillation and consequently produce sound patterns.

Chapter 5

Software development tools and concepts

The main deliverables of this project are two software artifacts.

1. A Matlab application that visualises the generation of patterns obtained from the interconnections of cells.

2. An environment for testing process schedulers, including the implementation of a diffusion-inspired scheduler.

Various programming and script languages were used. The first application is written in Matlab. The second application includes two parts: a process management framework written in Java on the Eclipse IDE and a set of bash scripts that use gnuplot [12] to generate bar graphs for analysing and testing the various scheduler implementations. Basic knowledge of the concepts, the languages and the algorithms used are provided throughout this chapter.

5.1 Matlab

Matlab is a programming environment developed by MathWorks [11]. It is ideal for working with vectors and matrices since it has build in matrix operations. It provides basic and advanced mathematical procedures and functions, such as sin, cos, Fourrier transformations and ordinary differential equation solvers. Its capability of producing all kinds of graphs and supplying a plethora of image processing functions makes it the ideal tool for studying the phenomenon of morphogenesis.

The approach was to study and test models by solving differential equations and producing representational graphs and at the end, generating movies with image time frames to show how cells get organised and what shapes they produce. An example of such image frame is shown in Figure 5.1.

The functions in Matlab are written according to the format below:

```
function <output> = name(<arguments>)
    body
end
```

Note that the output can be a vector, a 2-dimensional array or an abstract structure, meaning that the type of the output value will be given during the execution of the program.

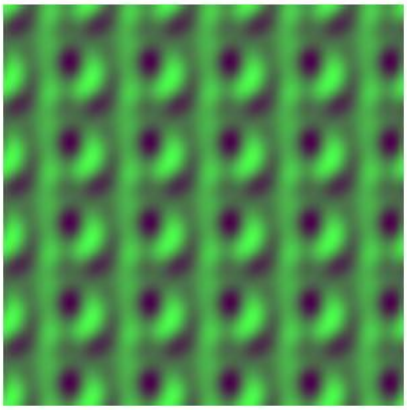

Figure 5.1: An image representation of the structure produced by a system of cells.

5.1.1 Modelling morphogenesis

As explained in Chapter 4, there are several dynamical models of the reaction-diffusion process in morphogenesis. The reaction model has the form $\dot{x} = F(x)$ and the diffusion model is according to conduction of heat as explained in paragraph 2.2 with cell interconnections of ring or torus-shaped structures. The Matlab program that was generated to solve such a model has two main parts.

The first part is the main function, where the initial conditions are given. Those conditions are the number of cells, the initial concentrations of each chemical compound and the diffusion coefficients. The main function constructs the data structures - which are vectors - and calls the appropriate ODE solver with arguments a function, a column vector (as the initial conditions) and a time span. Other arguments that are used by the ODE algorithm are explained in the next paragraph. Any function can have inside the definition and implementation of another function which is called a nested function.

The main purpose of the nested function is to make sure that the implementation of the mathematical model is evaluated correctly. The equations of such a model should be implemented in the nested function in a way that the vector operations are evaluated correctly. From a software development point of view, using a nested function instead of an external one is better, since the defined constants are easily shared from the main to its nested function. An example of a template code for solving ordinary differential equations is shown in Figure 5.2.

5.1.2 Integrating ordinary differential equations

Matlab provides a variety of ODE solver implementations each of them having different properties. Note that there are some constraints in evaluating a model that describes fluid concentrations. The main constraint is that the concentrations must never get a negative value otherwise the model will have no physical meaning.

The ODE-solver used in this project is the `ode45`. According to the Matlab documentation `ode45` is for solving non-stiff differential equations of medium order and it is the most common used among the provided ode solvers [11].

Apart from the ODE arguments shown on line 5 in Figure 5.2, the solver can also be given

```
function out = f()
    %body, initialise constants
    %call an ODE solver, such as ode45, ode23 etc...
    %example:
    [t, x] = ode45(@odefun, [0 100]);
    %that will solve equations in a time span 0-100 seconds

    plot x in respect to t as x(t(i))=x(i)

    function odefun = integration(tspan, init)
        body evaluate equations
        ode = equation results (column vector)
    end
end
```

Figure 5.2: Template of Matlab code for solving ODE's.

a structure of options as an extra argument. Options are used to override any default behaviour and explicitly define the constraints that the solver must take into account. The options are declared with the use of the `odeset` function.

The `odeset` function takes an arbitrary length of arguments. First, a string value is given which represents the key of the parameter that is intended to be overridden following by a real numerical value. For example, any reaction-diffusion model requires that no cell has a negative concentration of any of its morphogens. Thus, the options are constructed with the code below:

$$options = odeset('NonNegative', (1:N));$$

Where N is the size of the solution vector (cells$\times$morphogens). Note that all matlab ode solvers accept a column vector as the initial conditions and thus extra care should be taken for converting a 2-dimensional array to a column vector and the opposite correctly.

5.1.3 Plotting the results

Matlab provides easy to use methods that make plotting graphs convenient. The interest lies on observing graphically the equilibrium points of the system, how the system oscillates and for how long. Thus, plotting the results helps in both the analysis of the system and the testing of the programs.

The plot function

The ODE solvers can plot a graph by default after the solution is calculated or in real time. However, it is better to store the results of the ODE solvers in arrays to have full control of the data. After executing an ODE solver two matrices can be retrieved; the time vector and an array that can be 1-dimensional or 2-dimensional and contains the results of the variables in each column. This is shown representatively in Table 5.1.

Table 5.1: Data as obtained from integrating a function with input array X of size $4\times$ `sizeOf(t)`.

Time (t)	X_1	X_2	X_3	X_4
0	0.2373	0.4588	0.9631	0.5468
0.0061	0.2402	0.4489	0.9699	0.5194
0.0122	0.2431	0.4389	0.9767	0.4921
⋮	⋮	⋮	⋮	⋮
0.9622	0.5530	-0.7833	1.5869	-2.3828
0.9811	0.5564	-0.8006	1.5916	-2.4167
1.0000	0.5597	-0.8177	1.5962	-2.4498

The plot can be used to show a graph of all the variables according to time or for each variable according to another variable. For example, let t be the time array and X be a 2-dimensional array representing morphogen X values of a cell in each column. Now the row index is associated with the time index according to the relation:

$$X[cell][index] = X(t_{index})$$

Executing the plot with arguments t and X as shown in Table 5.1 will give a plot of a graph containing a number of curves equal to the width of the array X. Note that X and t must have an equal number of rows. The generated plot is shown in Figure 5.3.

$$plot(t,X);$$

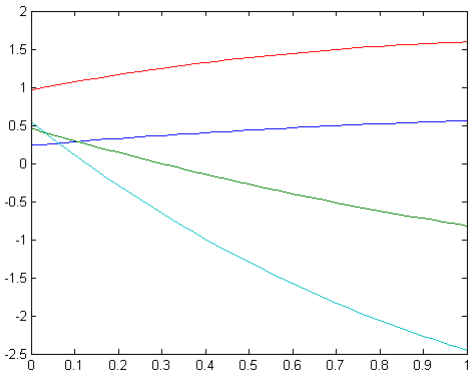

Figure 5.3: Graph presenting data from Table 5.1 , Time$\times X$.

5.1.4 Creating movies

Matlab provides a tool set for creating animated movies. The concept is that a movie is a set of images called frames. Each frame, which is a 3-dimensional image for coloured images, is stored in a 2-dimensional array which contains 3-dimensional frames. Eventually, the result is

a 4-dimensional array of size $F \times X \times Y \times 3$, with F being the number of frames, $X \times Y$ the size of each frame and number 3 the vector size of colour values in RGB colour model.

To create a movie, Matlab needs to record images from a figure type window. Thus, for every frame stored, an image must be shown in the same figure window. The function that creates and shows an image out of a 2-3D matrix is called imshow(M), M is the argument that represents a 2-dimensional or 3-dimensional matrix (2D for grey-coloured images, 3D for coloured). Then the function `getFrame(Figure,window-size)` is called, in order to grab the frame. All frames must be stored in a matrix that was initialised by the function `moviein(numberOfFrames,Figure,windowSize)`.

Every structure of type 'Figure' has an identifier. Thus, keeping the same identifier for the figure that hosts the image from `imshow()` avoids the creation of many windows. The result is one figure-window that hosts a different image while time passes, resulting in an animated movie show. A structure of type 'Figure' is constructed by calling the function `figure(id)` with `id` being a natural number. The general function playMovie that was developed can be viewed in Appendix section A.2.

Creating the images

Alan Turing states that morphogens are chemical compounds that give certain characteristics to the cells according to their type and concentration [1]. A way to identify and differentiate cells in a computer simulation is to give them different colours according to their morphogen concentrations. Many ideas and combinations were used on how to match morphogen concentrations to different colours.

The first problem in matching morphogen concentrations to colour values is to normalise the concentrations, which are arbitrary real values greater than 0, to real values in the range $[0, 1]$. The Algorithm 1 describes the colour-mapping procedure.

Algorithm 1 Colour mapping

```
let MORPHOGENS be a 2D array of positive real values
let MIN be the minimum value in MORPHOGENS
let MAX be the maximum value in MORPHOGENS
let an array COLOURS be the same dimensions as the MORPHOGENS array's
for each VALUE in MORPHOGENS do
   NEWVALUE=(VALUE-MIN)/(MAX-MIN)
   add the NEWVALUE to COLOURS
end for
```

After converting the concentration values in colour values and storing them in a 3-dimensional array, the `imshow()` method will output a coloured image of a representation of the cells at a certain point in time. This is called a snapshot of the state of the physical system under study.

5.1.5 Producing sound

Since models of morphogenesis present an oscillatory behaviour, the wave patterns that are obtained can be mapped into sound waves. A model with continuous/persisting oscillations is the Gizburg-Landau model which involves partial differential equations. The template for computing the mathematical model is given by Aly-Khan Kassam [4].

The morphogen values can be imported directly to the Matlab function `soundsc(values, sample rate, bit depth)`, that normalises and scales them automatically to sound values. Decreasing the sample rate makes the sound heavier/slower, while higher sample rates make the sound accute. Matlab documentation states that the `soundsc` function accepts sample rates of the range $[80, 10^6]$ [11] but it is bound on what the sound card on the machine can support.

The `soundsc` is available on both Windows and UNIX architectures. The implementation was done on a Linux-based operating system (Fedora 17) and the method used the ALSA drivers to generate sound.

5.2 Java

One of the goals of the project was to identify ideas based on morphogenesis that can be applied to computer science or other engineering problems. Java was chosen as the tool to deliver such solutions, since it can work on many machines and on various operating systems. In addition, Java is more suitable for Object Oriented Programming (OOP) than Matlab. Matlab supports OOP as well, but two main reasons led to the decision of chosing Java over Matlab.

1. The fact that type handling in Java is explicit, helps in a better structural software design, with the use of various patterns as described by the Gang of Four [13] and GRASP[1] principles [5] resulting to a more understandable, easy to maintain source code.

2. Matlab is proprietary software and since no explicit Matlab tools are needed, Java is the best way to deliver, develop and expand free, easy-to-implement and use software framework.

The solution is broken into two main concepts: Building an ODE solver and a simulation of process management system. The idea is described in paragraph 5.2.3, Diffusion inspired scheduler.

5.2.1 Implementing Euler's method

A method for integrating ordinary differential equations is the Euler's method. Euler's method is a first order method [14]. Two ways to implement the Euler's method are suggested. The approach that was proposed originally is the explicit method where the time-step is fixed. A more advanced solution is the implementation of implicit methods which change dynamically the time steps to reduce integration errors during steps.

Euler's method is used for the purpose of the diffusion scheduling. Since there is a limit on the time slice, the explicit method is used to avoid the disadvantage of increasing the total complexity of the algorithm. That means that the error per step is proportional to the square of the step size. To approximate the reaction-diffusion model of L-systems, a step of 0.1 time units is considered as suggested and implemented from Christopher G. Jennings, PhD in his work on Turing's Reaction-Diffusion Model of Morphogenesis [7].

The method takes the initial values for every variable in a vector x_0 of size N.

$$\dot{x}_0 = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

[1] GRASP is an acronym for General Responsibility Assignment Software Patterns [5]

Then, for each time step, starting from $n = 0$ it calculates the next state in the next time step.

$$\dot{x}_{n+1} = \dot{x}_n + h * f(t_n, \dot{x}_n)$$

Where:

- h is the step size and it should be less than 1.
- n is the index to the current step.

The algorithm that describes the implementation of the Euler's method is shown in Figure 2. The algorithm depends on the function and its number of variables. For that reason, the Template pattern was used to protect the implementation of the Euler's method from function variations. As shown in Appendix B.5, the abstract class AbstractFunction provides a default implementation of an equation. In order to add a new equation, a new class has to be implemented that extends the AbstractFunction. Thus, by the means of inheritance, the variations of different equations are hidden from the ODE solver resulting in a coherent code that does not need to change every time a new equation is added.

Likewise, ODE solvers can be added, as defined by the ODE interface, without the need of changing the functions since the ODE interface and the AbstractODE is bound to the AbstractFunction. In other words, the principle of GRASP for protecting variations was widely taken into account to construct a well coherent framework.

Algorithm 2 Euler's method

```
results[0]=initial conditions
for i in 1 to number of total steps do
  results[i] = results[i-1] + timestep*function()
end for
return results
```

5.2.2 Process management

A process is a portion of code that needs to be executed by a processor to accomplish a task. This execution needs time to be completed, proportional to the cycles per second that the processor runs.

The process management is a responsibility of the operating system. Each operating system can have one or more scheduling algorithms to manage the execution of a queue of processes. Modern operating systems, support preemptive multitasking [15]. This means, that a process can be paused without its permission and another process can take its place. The scheduling algorithms become more sophisticated in order to handle pseudo-parallelism. The general goal is to keep each process as less idle as possible.

This project explores the behaviour of executing a fixed amount of processes by using a diffusion-inspired scheduler as described in paragraph 5.2.3 against other process schedulers. The study is just an introduction to the idea of using the concept of reaction-diffusion to help the algorithm decide on how much time each process must be given each turn. Each process has the properties described in Figure 5.4.

The processes are arranged in a queue. No new processes are added in the queue and a process is removed from the queue when it exits with a finished or a crash message.

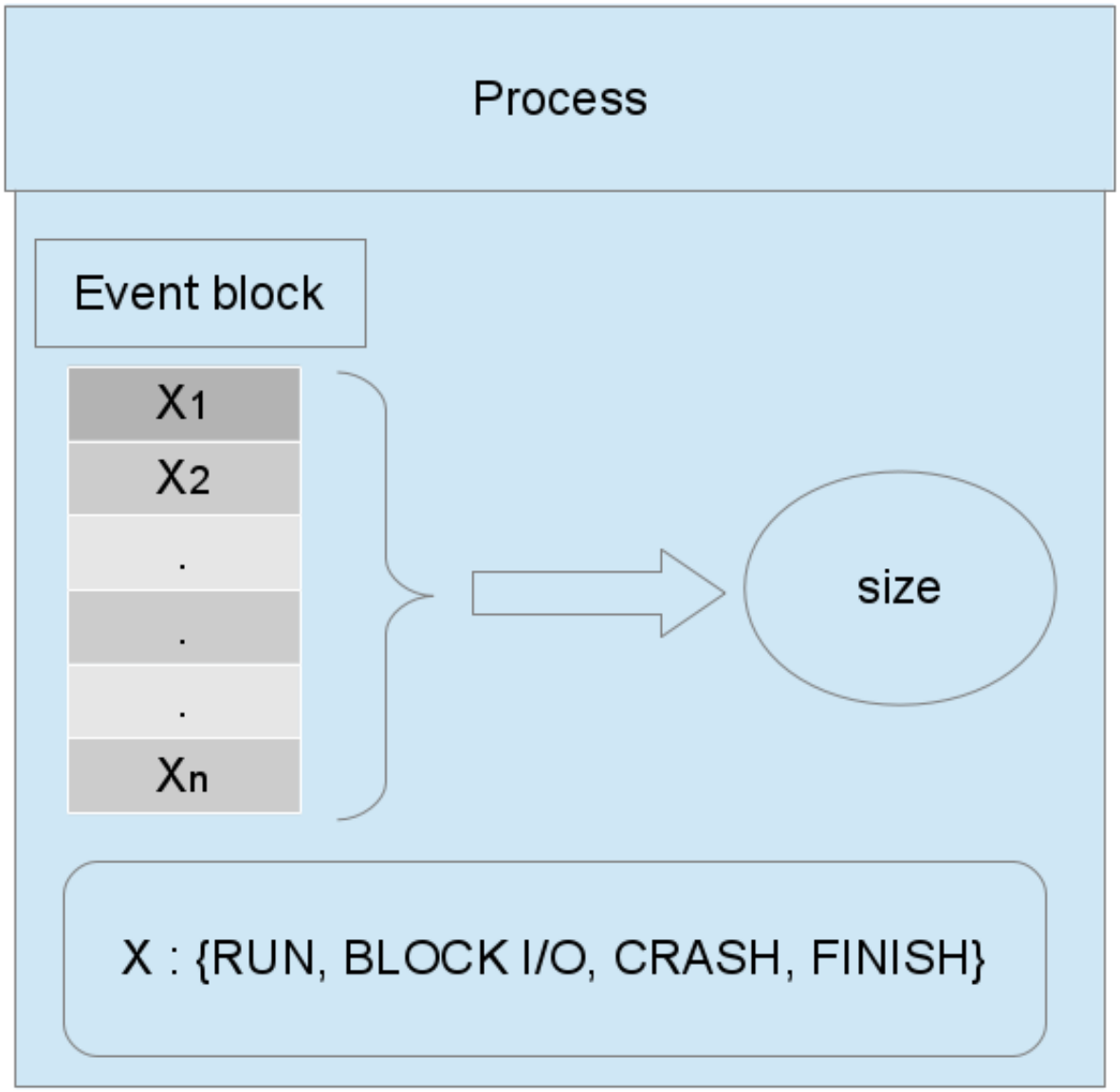

Figure 5.4: Scheme of a process as handled by the implementation of the process management framework.

5.2.3 The scheduling problem

The responsibility of a scheduler is to find a way of executing all the processes that seek execution time with maximum efficiency. The term efficient is ambiguous but justifiably so. The goal of a scheduler differs from situation to situation and from system to system. Despite the fact that in a distributed system, prioritising might be less important than executing client requests without the concern of which client should get the response first. There are also some general specifications that a scheduler shall achieve. Some general goals include:

- To finish as early as possible.
- To execute system processes with higher priority.
- To avoid starvation.
- To detect deadlocks.

Finishing fast is not as crucial as it may seem. It is clear, that the fastest solution in terms of time is to execute the processes serially (assuming all finish at some point). However, the most important concept in a real-world situation is to present pseudo-parallelism, that is to make the processes look that they are being executed in parallel. Avoiding starvation is another specification a scheduler has to offer. Assuming that the body of the code under processing does not contain any deadlock situations, then all processes must be executed and finished at some point in time. Thus, each process should take some time in every short amount of cycles. The phenomenon on which a process is not given processing time at all is called starvation.

Deadlocks are not considered in this project but they should be referenced to avoid conflicts with starvation. A deadlock can occur when two or more processes wait for each other to continue their processing. If this happens, the scheduler will be running those processes forever. Detection of deadlocks is important but is not always the responsibility of the scheduler. Usually, the program that detects deadlocks is called 'watchdog' [15]. However, it is crucial for a scheduler to work smoothly if deadlocks are detected to avoid wasting time with frozen processes.

The rest of the Chapter describes different scheduler algorithms used to assess the performance of the diffusion-inspired scheduler. These are the random sceduler, the Round-Robin and the priority scheduler. The algorithm of the diffusion-inspired scheduler is also given.

Random scheduler

The random scheduler picks randomly a process from the queue and executes it for a fixed amount of time units. The pseudo-code is in Algorithm 3 below.

Round Robin scheduler

The Round Robin scheduler picks serially each process in the queue and executes it for a fixed amount of time units. It guarantees that eventually all processes will be executed and exit. The pseudo-code is shown in the Algorithm 4.

Algorithm 3 Random scheduler

```
while Process queue is not empty do
  choose a process at random
  ask the process to run up to a pre-defined number of time units
  get acknowledgment message from process
  get time used by the process
  increase for each other process their idle time
  increase execution time for the process chosen
  if the process finished then
    add execution time, idle time and total time to the scheduler statistics
    remove process from queue
  end if
end while
display statistics
```

Algorithm 4 Round Robin scheduler

```
while Process queue is not empty do
  choose next process in queue
  ask the process to run up to a pre-defined number of time units
  get acknowledgment message from process
  get time used by the process
  increase for each other process their idle time
  increase execution time for the process chosen
  if the process finished then
    add execution time, idle time and total time to the scheduler statistics
    remove process from queue
  end if
  if last process in queue then
    set pointer back to the start
  end if
end while
display statistics
```

Priority scheduling

The priority scheduler is based on the fact that some processes are more important than others. The implementation assigns a random integer 0-39 to each process in the waiting queue. This is the priority of the process; the smaller the number the higher the priority. The scheduler executes serially the processes with the minimum priority number until they are all finished and then moves on to the next minimum priority number until the queue is empty. The pseudo-code is shown in Algorithm 5.

Diffusion-inspired scheduler

The implementation of the diffusion-inspired scheduler exploits the idea of the Round Robin in an alternative implementation of a weighted Round Robin scheduler. That is, instead of having

Algorithm 5 Priority scheduler

```
set a priority value from 0-39 for each process
set minimum value to 0
while Process queue is not empty do
  if a process with priority equal to the minimum value does not exist in the queue then
    find the minimum priority value of a process existing in the process queue
  end if
  find the next process with priority value equal to the minimum value
  ask the process to run up to a pre-defined number of time units
  get acknowledgment message from process
  get time used by the process
  increase for each other process their idle time
  increase execution time for the process chosen
  if the process finished then
    add execution time, idle time and total time to the scheduler statistics
    remove process from queue
  end if
end while
display statistics
```

a fixed amount of time units that a process takes to be executed, the time varies according to Diffusion dynamics. The advantage is that because of diffusion all processes are guaranteed to take some time but not necessarily on every execution round.

The processes are initially given a certain random time unit value that is in reality the amount of time that it will be given for execution. That amount of time represents the concentration of a morphogen in a physical system of cells, and thus, processes represent cells. The processes are executed for the amount of time units given. After the execution round ends, the reaction-diffusion dynamics are calculated and the system of equations is integrated giving as a result new time units for each process.

An exceptional scenario can be encountered where processes starve. This can happen when all the processes that have a positive time unit are executed and finish, leaving in the queue processes that have 0 time units and are thus unable to diffuse. This is solved by following a simple rule: if a process is called to be executed for 0 time cycles, then it is always executed for one.

A more sophisticated way to avoid starvation is to assign two fixed pseudo-processes to certain places in the queue that have a fixed amount of time units that never change, and thus feeding the rest of the processes avoiding starvation. These scenarios and solutions were not studied. Nevertheless, the fact that starvation did not happen during the experiments points that further investigation is needed. The implemented framework is freely available for expansions and testing various ideas.

The algorithm of the Diffusion scheduler is provided below (Algorithm 6).

Development description of the process management framework is given in Chapter 7.

Algorithm 6 Diffusion-inspired scheduler

```
set randomly time units to a process in a range close to the other ranges
(alternatively, set equal time units to a process equal to the pre-defined value)
while Process queue is not empty do
    choose next process in queue
    ask the process to run up to its given number of time units
    get acknowledgment message from process
    get time used by the process
    if process used all the time then
        set the reaction variable of that process to 0
    end if
    if process was last in queue then
        set Euler time step to 0.01 as an imaginary time step
        calculate reaction-diffusion differential equations by applying Euler's method and as-
        sign to each process a new amount of time units
    end if
    increase for each other process their idle time
    increase execution time for the process chosen
    if the process finished then
        add execution time, idle time and total time to the scheduler statistics
        remove process from queue
    end if
    if last process in queue then
        set pointer back to the start
    end if
end while
display statistics
```

Chapter 6

Applications and results

The results of this project involve analysis of dynamical systems that describe chemical reactions, cell diffusion and complete reaction-diffusion process as described in Chapter 4. The latter is a set of linearised and non-linear models. The non-linear models are used to visualise cell structures and to study mutations. Results also show the rationale behind sound generation and some attempts to generate sound. Finally, the statistical information obtained from the scheduling algorithms demonstrate why scheduling using the concept of diffusion is promising. The results are obtained by the software applications that were developed for the project.

6.1 Chemical reactions

Various chemical reactions were studied to identify how altering various parameters affects their behaviour. The main interest is to observe oscillatory behaviour, if any, and the amount of time needed for a reaction to be completed. Parameters are related to the concentration of each chemical substance, the reaction rate or coefficient which show the amount of each compound needed to produce another compound.

The experiments use the chemical reaction equations described by the Michaelis-Menten mechanism as presented in the case study of Stephen Childress [16]. Four chemical substances are involved: one acting as a catalyst (enzyme E) interacting with a chemical substance X for composing the product P. There is a complex chemical compound C as a middle state which breaks into E+X or E+P according to the reaction rates k_{-1}, and k_{+2} respectively. Note that the index in each rate k indicates the step (backwards, forward) needed to reach a certain state of the reaction.

The chemical reaction equation of the Michaelis-Menten mechanism is:

$$E + X \underset{k_{-1}}{\overset{k_{+1}}{\underset{\leftarrow}{\rightarrow}}} C \overset{k_{+2}}{\rightarrow} E + P \tag{6.1}$$

It is important to note that predictions are made before running an experiment. This helps in both testing and building an understanding of how, in this example, a complex reaction works.

Prediction

It is clear from the equation (6.1) that the final products will be the enzyme E and the product P. The question arising is how can the maximum possible concentration of P be retrieved at the minimum amount of time?

Table 6.1: Parameters used by solving the chemical equation (6.1).

	Initial Concentrations				Reaction rates		
Experiment Number	E	X	C	S	k_{+1}	k_{-1}	k_{+2}
1	20	20	0	0	0.01	0.01	0.01
2	20	20	0	0	0.01	0.1	0.01
3	20	20	0	0	0.01	0.01	0.1
4	20	20	0	0	0.1	0.01	0.01

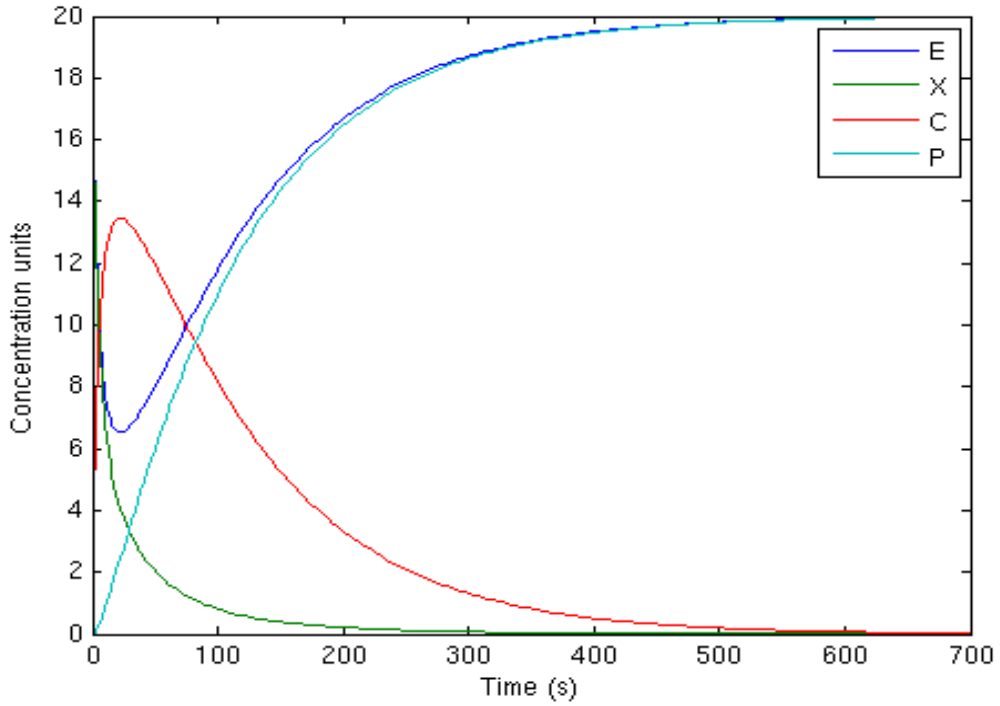

Figure 6.1: Graph showing the behaviour of the chemical reaction when there is a balance in all rates (experiment 1,Table 6.1).

The complex compound C breaks into E and X at a rate k_{-1} into E and P at a rate k_{+2}. When X runs out the reaction stops. Thus, the rate k_{+2} must be greater or equal to k_{-1} to accomplish the minimum time.

Table 6.1 shows the parameters of the try-outs for testing the Michaelis-Menten mechanism as described by the chemical equation (6.1).

The rationale behind the parameter values is based on discovering how each parameter affects the reaction. Starting with equal reaction rates and then changing only one parameter for each test, the altered behaviour can lead to conclusions which may verify the predictions. If the prediction is proven wrong, then either there is a bug in the code which integrates the differential equations or the logic behind the prediction was false or the mathematical model that described the chemical reaction was wrong.

The results show that, as predicted, that the higher the rate in which C breaks to E and P the less time is needed for chemical X to run out and cause the reaction to end. At the same time the production of E and X must be lower. So, generally the significance at the difference

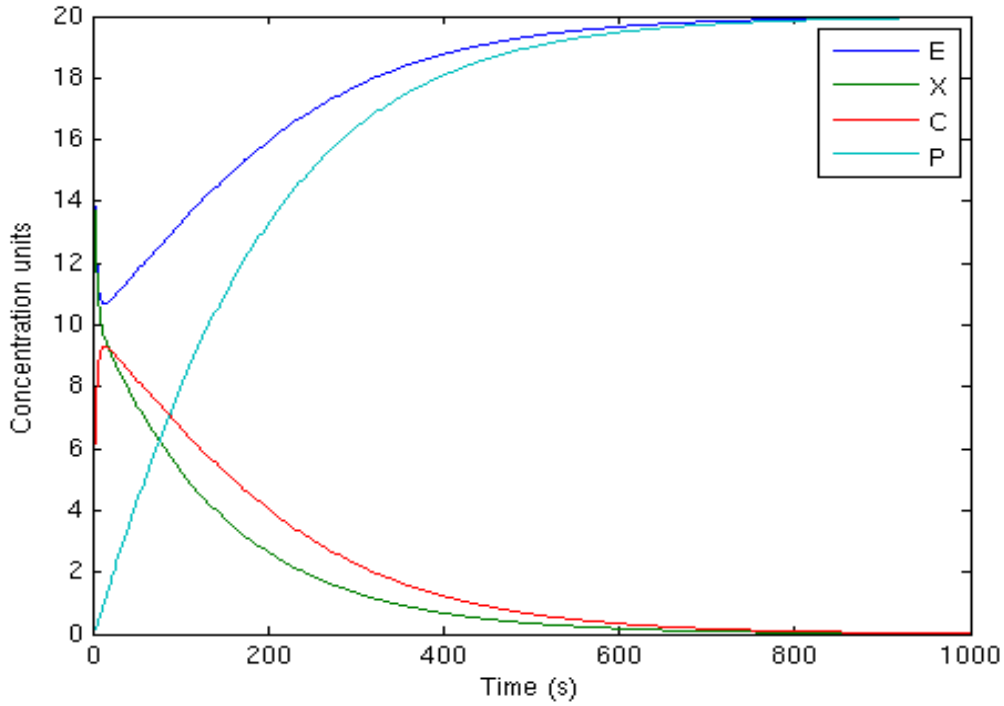

Figure 6.2: Graph showing how a higher rate of C breaking into X affects Michaelis-Menten mechanism (experiment 2, Table 6.1).

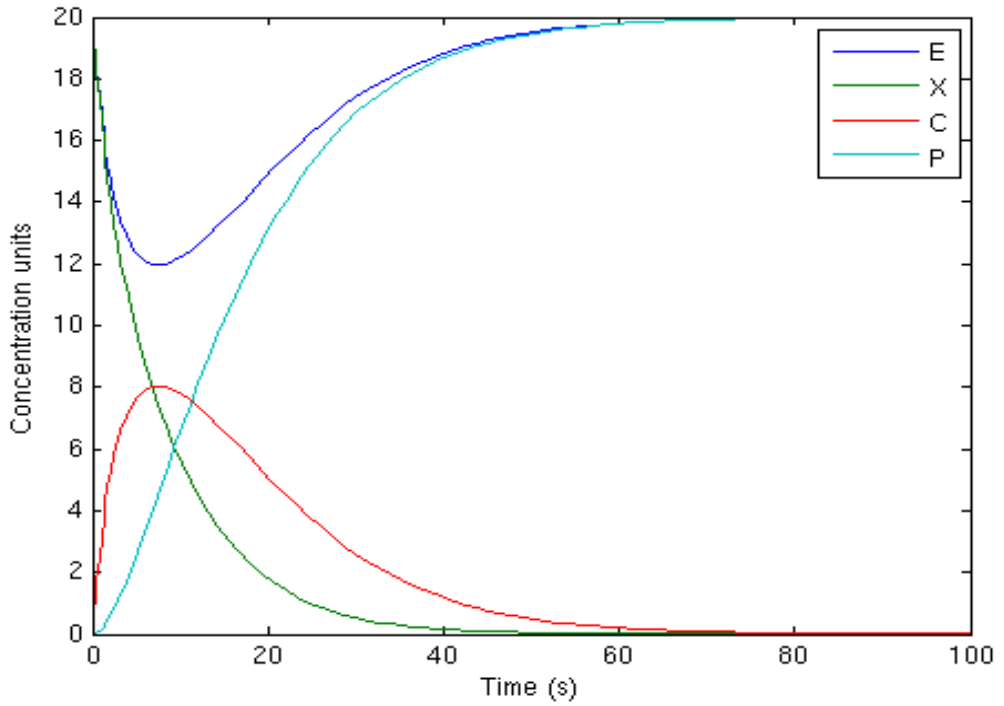

Figure 6.3: Graph showing how a higher rate of C producing product P affects the Michaelis-Menten mechanism (experiment 3, Table 6.1).

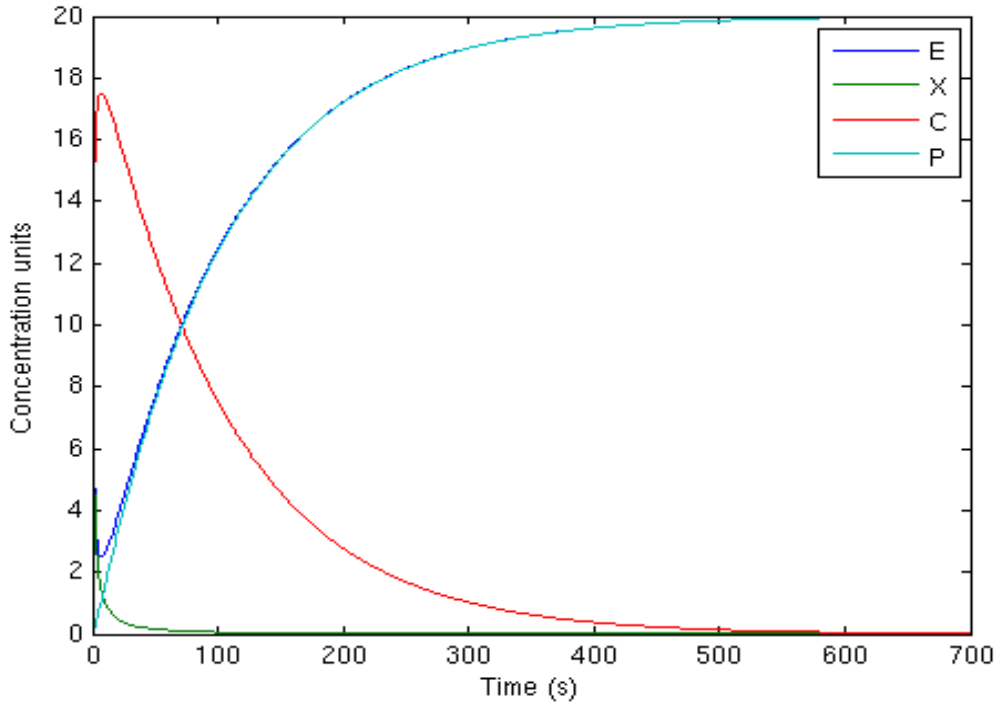

Figure 6.4: Graph showing how a higher rate of producing C affects the Michaelis-Menten mechanism (experiment 4, Table 6.1).

between k_{+2} and k_{-1} affects the overall production rate of P. The reaction rate of the production of the complex compound C also affects the time the reaction needs to complete. It can be high enough to satisfy the production rates of both ends. However, if it exceeds a certain threshold it has no effect, as the production rates of X and P are not sufficient to cover the concentration of C.

The experiments from chemical reactions show that:

- The chemical reactions can be easily modelled with differential equations.
- Compounds that break into different chemical substances present more complex behaviours and are closest to produce oscillations rather than equations of the form showed in the first example.
- Different reaction rates affect the time for the completion of a chemical reaction but values exceeding a threshold may exist that have no effect.

6.2 Diffusion

Diffusion depends on the interconnections between cells. Two types of interconnection were studied: cells connected in a ring-shaped structure and a torus-shaped connection. The difference between the two is the number of cells each cell is attached to. In a ring, two cells are attached to every other cell. In a torus, four cells are attached to every other cell.

Diffusion was implemented according to the conduction of heat as described in Section 2.2. Experiments were done to identify the role of the parameters of the cells, the diffusion

Table 6.2: Parameters used to run tests on diffusion.

No.	Cell side size	Number of Cells	Shape	Diffusion Coeff.	Initialisation method
1	0.5	3	Ring	0.1	Random
2	0.1	3	Ring	0.1	Random
3	0.5	9	Ring	0.1	Random
4	0.5	4	Torus	0.1	Random

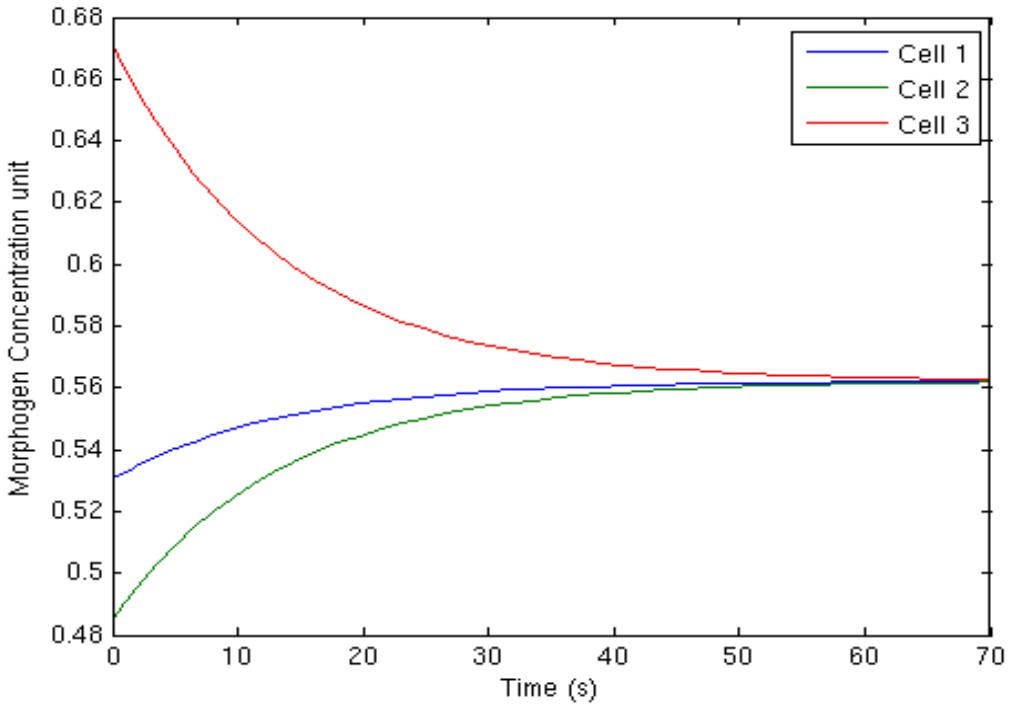

Figure 6.5: Graph showing chemical concentrations during the diffusion process (experiment 1, Table 6.2).

coefficients and the shape of the system. Table 6.2 shows the parameters under study and figures 6.5-6.8 show the graph representations of the behaviour of such system.

The expected result after the diffusion process ends is that every cell has the same concentration of a particular morphogen as shown in figures 6.5-6.8. The diffusion coefficient and the size of the cell affect the fluid flow and therefore smaller cells need more time to diffuse. The properties of a ring structure against a torus structure are also studied.

Conclusions obtained from the results of tests 2, 3 and 4 compared to 1 (Table 6.2):

- Experiment 2 shows that with smaller cells, diffusion needs more time to be completed.
- Experiment 3 shows that more cells need more time to complete the diffusion process.
- Experiment 4 shows that a torus structure diffuses faster that a ring one.

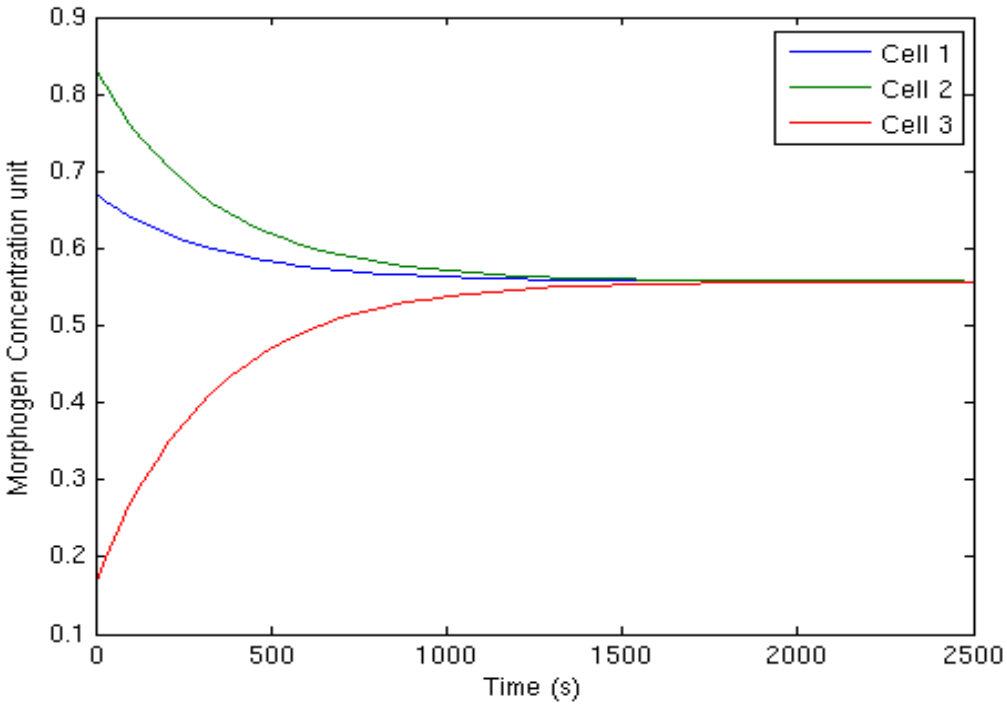

Figure 6.6: Graph showing how the size of the cells affects the diffusion process (experiment 2, Table 6.2).

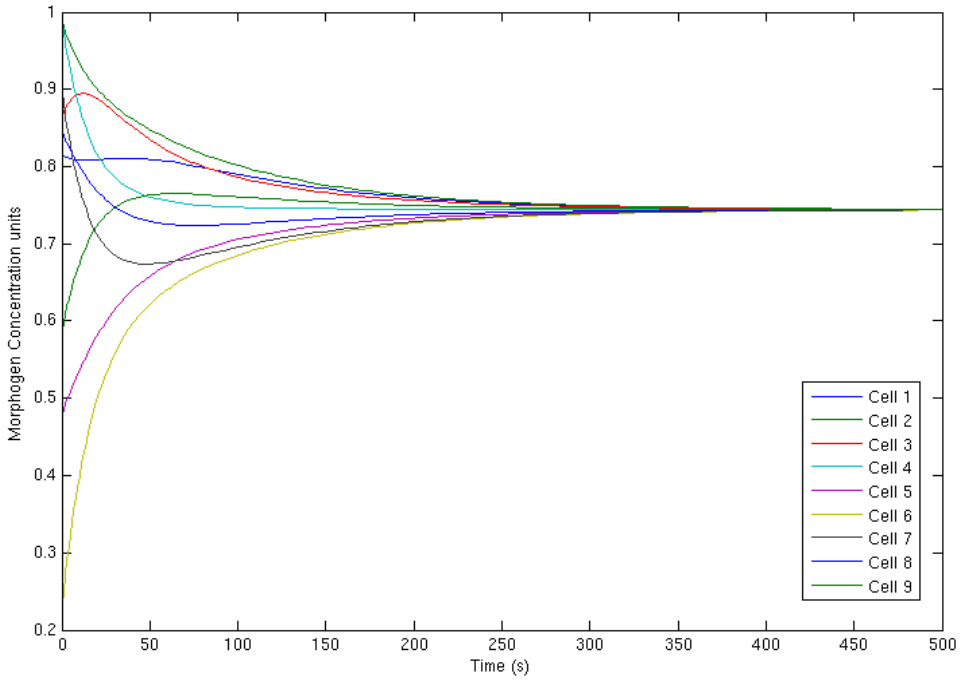

Figure 6.7: Graph showing how the number of the cells affects the diffusion process (experiment 3, Table 6.2).

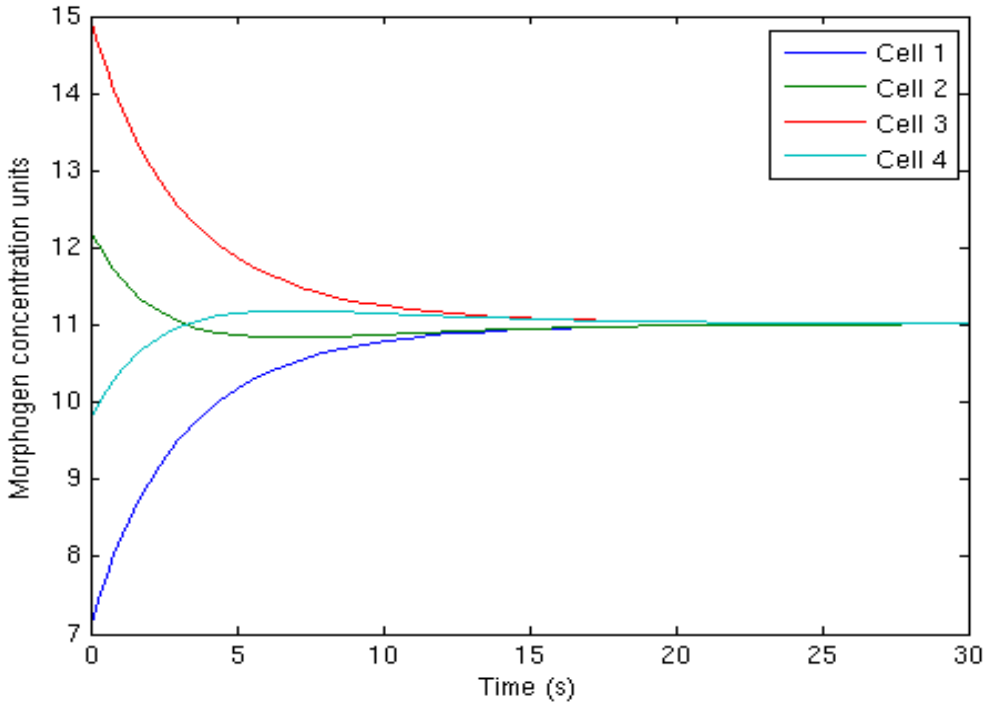

Figure 6.8: Graph showing the diffusion process in a torus structure (experiment 4, Table 6.2).

6.3 Reaction-diffusion

Reaction-diffusion models can be considered as complete models of morphogenesis. They result from merging reaction and diffusion models. Depending on the type of the function, linear or non-linear, that describes a chemical reaction, the models are categorised into linear or non-linear.

The linearised model that is tested in Section 6.3.1 produces negative solutions as well. Negative solutions are not accepted when working with chemical substances, since a negative concentration does not have a physical meaning. There is an option for forcing the ODE solver to always produce non-negative solutions. However, this results in unstable behaviour. Nevertheless, as explained in Section 3.1.2 the goal is to approximate the behaviour of a system near the equilibrium points. In other words, the study is about the qualitative behaviour of the system rather than the quantitative one.

Modelling with non-linear systems gives an approximation closer to reality since chemical substances can't be negative. Results for non-linear systems include image snapshots of the cell structure at particular time points.

6.3.1 Implementing and testing with linearised models

The linear system that was studied is proposed by Alan Turing [1] and analysed in the case study of Stephen Childress [16]. The main interest lies on the stability conditions. The equation is of the form:

$$\dot{x} = Ax$$

Table 6.3: Experiments using the Jacobian matrix and diffusion for modelling the process of reaction-diffusion in a ring of cells.

No.	A	No. of cells	Diffusion included	Initial state	Satisfies conditions
1	$\begin{pmatrix} 0.6171 & 0.8244 \\ -1.6432 & -0.8824 \end{pmatrix}$	3	Yes	Random	Yes
2	$\begin{pmatrix} 0.6834 & 0.4423 \\ -2.1634 & -1.3875 \end{pmatrix}$	3	Yes	Random	Yes
3	$\begin{pmatrix} -1.6364 & 0.8824 \\ -0.5 & -0.312 \end{pmatrix}$	3	Yes	Random	No
4	$\begin{pmatrix} 0.6171 & 0.8244 \\ -1.6432 & -0.8824 \end{pmatrix}$	3	No	Random	Yes

Where A is the Jacobian matrix evaluated at the equilibrium point as described in 3.1.2 and has size $N \times N$. The case studies a system with 2 morphogens.

$$N = 2,$$

$$A \in \mathbb{R}^{2\times 2}$$

The stability conditions are computed by finding the roots of the characteristic equation obtained from equation (6.3) and analysed in [16, p. 5] computing the conditions shown in equation (6.4).

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \tag{6.2}$$

$$det(A - \lambda I) = 0 \tag{6.3}$$

$$\begin{aligned} a_{11} + a_{22} &< 0, \\ a_{11}a_{22} - a_{12}a_{21} &> 0 \end{aligned} \tag{6.4}$$

In other words, the linearised matrix can have several random forms, but always satisfy the given conditions in 6.4. The program in Appendix Section A.3 is responsible for the generation of such random matrices. Table 6.3 shows the different values of the matrix. The Figures 6.9-6.12 show the behaviour of the system. Note that the Jacobian matrix is used to approximate the behavior of the reaction process. Merging the corresponding matrix with the diffusion in a ring of cells from equation (2.6), the system becomes as shown by equation (6.5).

$$\dot{x_i} = Ax_i + \begin{pmatrix} \delta^2 d_1 & 0 \\ 0 & \delta^2 d_2 \end{pmatrix} (x_{leftof(i)} - 2x_i + x_{rightof(i)}) \tag{6.5}$$

The experiments show that:

- The absence of diffusion as opposed in experiment 4 of table 6.3 does not affect the oscillatory behaviour.
- Not satisfying the stability conditions leads to an unstable system with infinity values, as shown in test 3 of table 6.3.

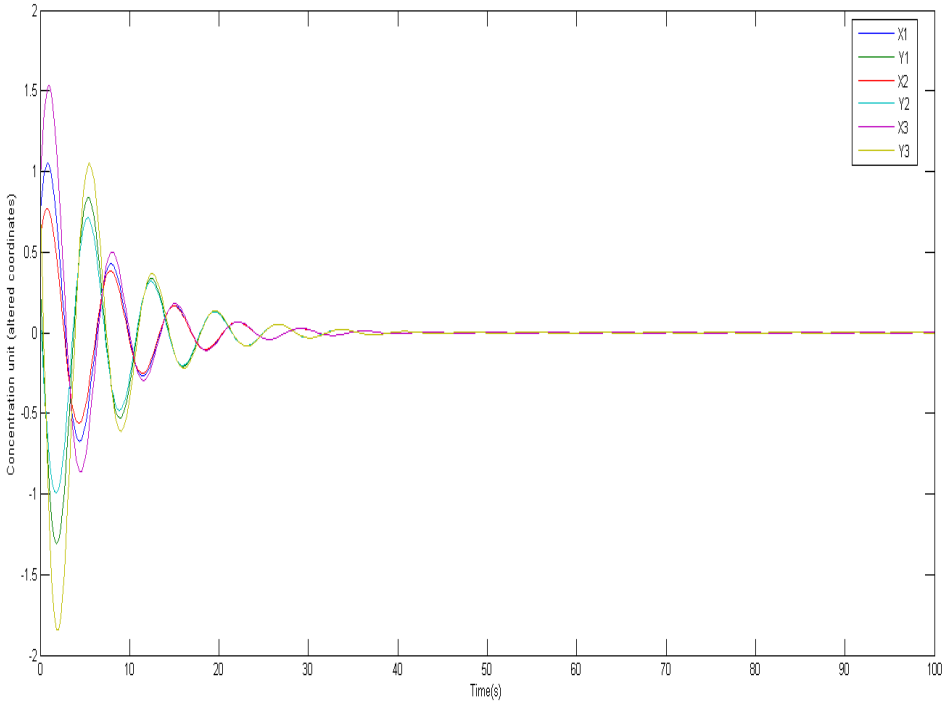

Figure 6.9: Graph showing the behaviour of a linear reaction-diffusion system (experiment 1, Table 6.3).

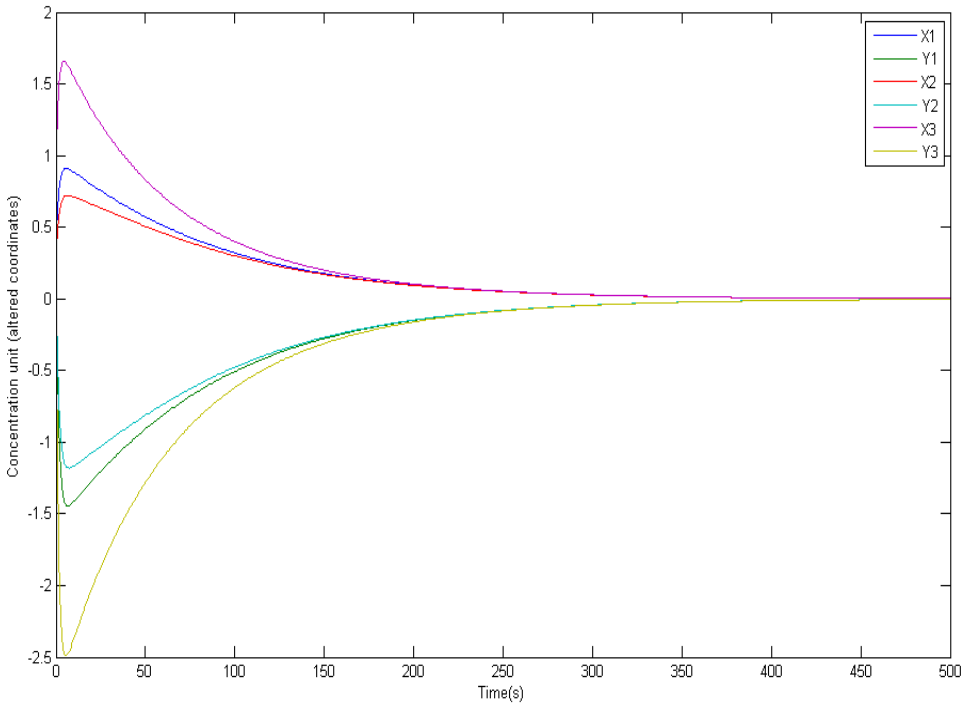

Figure 6.10: Graph showing the behaviour of a linear reaction-diffusion system with different values of the Jacobian matrix (experiment 2, Table 6.3).

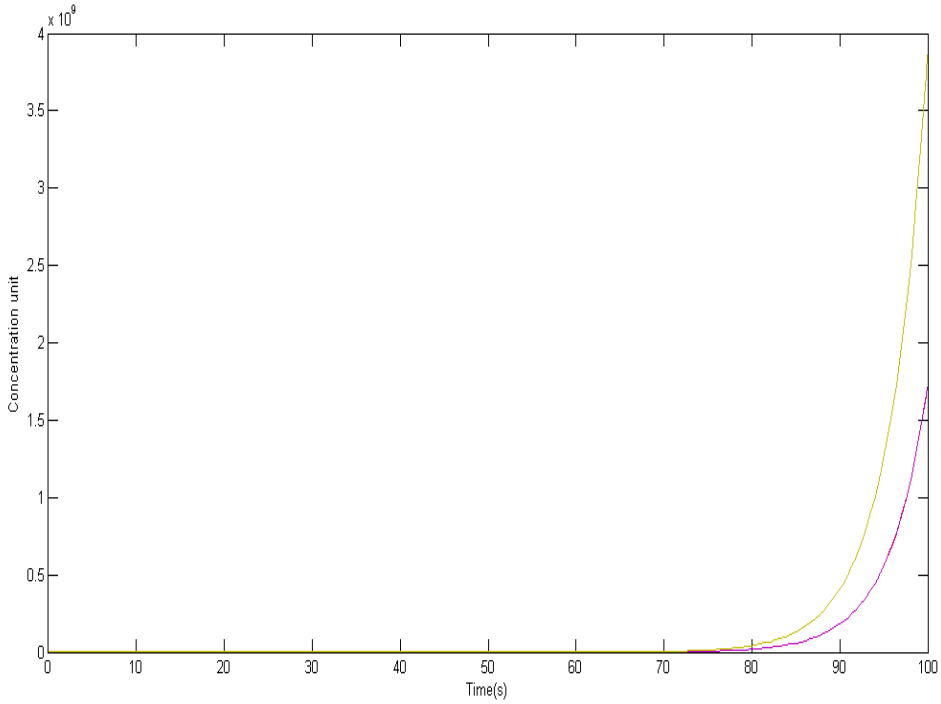

Figure 6.11: Graph showing that a linearised reaction-diffusion system is unstable if stability conditions are not satisfied (experiment 3 from Table 6.3).

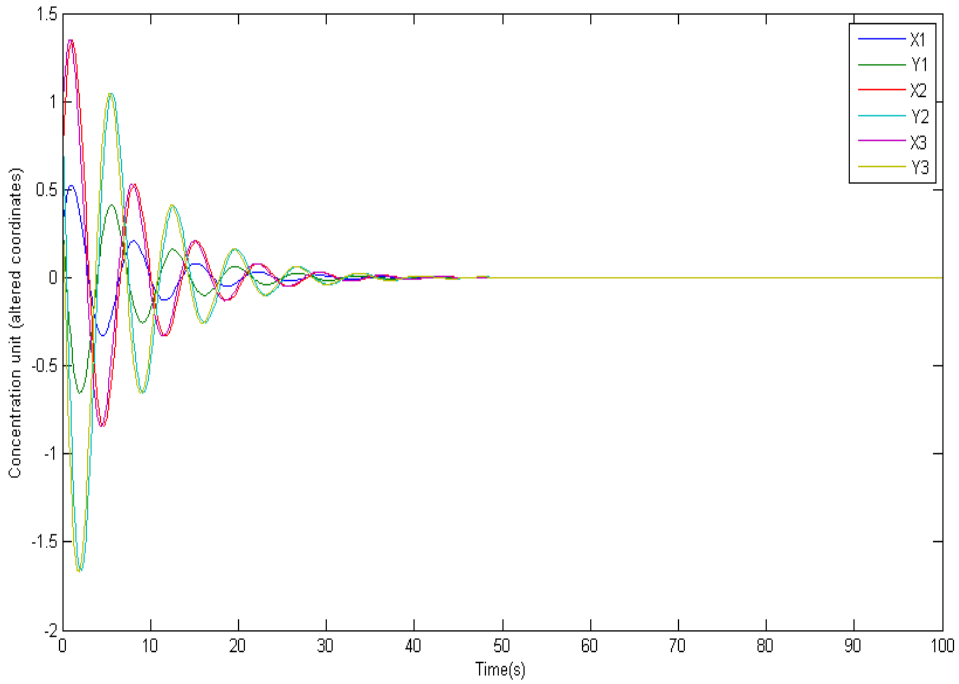

Figure 6.12: Graph showing the same linear system as in experiment 1 but without diffusion (experiment 4, Table 6.3).

Table 6.4: Experiments on how a linear system is affected from diffusion.

No.	A	No. of cells	Diffusion coefficients (M)	Initial state
1	$\begin{pmatrix} 0.6171 & 0.8244 \\ -1.6432 & -0.8824 \end{pmatrix}$	3	$\begin{pmatrix} 0.025 & 0 \\ 0 & 0.05 \end{pmatrix}$	Random
2	$\begin{pmatrix} 0.6171 & 0.8244 \\ -1.6432 & -0.8824 \end{pmatrix}$	3	$\begin{pmatrix} 2.5 & 0 \\ 0 & 5 \end{pmatrix}$	Random
3	$\begin{pmatrix} 0.6834 & 0.4423 \\ -2.1634 & -1.3875 \end{pmatrix}$	3	$\begin{pmatrix} 0.025 & 0 \\ 0 & 0.05 \end{pmatrix}$	Random
4	$\begin{pmatrix} 0.6834 & 0.4423 \\ -2.1634 & -1.3875 \end{pmatrix}$	3	$\begin{pmatrix} 40 & 0 \\ 0 & 80 \end{pmatrix}$	Random

Results do not prove or show regarding the way that diffusion affects the system. Trying to perform the same tests by changing the diffusion coefficients only is shown in Table 6.4 and figures 6.13, 6.14 opposed to 6.9 and 6.10 respectively.

Notice now that only two colours are visible in the graph 6.10. That means that morphogen concentrations in cells are equal and thus, with high diffusion coefficients (40 and 80) all cells rapidly become identical. This means that large diffusion coefficients must be avoided in order to differentiate cells.

The general conclusion is that diffusion is very important in morphogenesis and has a high impact on the behaviour of the system. Despite the fact that graphs show identical behaviour, it should not be neglected that the purpose of the project is to generate software applications that present how cells create structures. Cells must have properties that differentiate one-another. By setting a large diffusion coefficient value, cells may become quickly identical and the internal chemical reactions will not lead into different results. The choice of diffusion coefficients is therefore critical.

6.3.2 Non-linear models

The non-linear models discussed in Chapter 4 were used to the final software solutions. The main purpose was to show how cells are organising and form structures out of a random chaotic state. The results are shown in images that represent the structure of the system. Those images are snapshots of the system at time zero, middle and final point snapshots are arranged in a timeline.

Table 6.5 shows the parameters and the models that were used. Notice that diffusion coefficients and the number of cells affect the structure of the system.

As shown in Figures 6.15-6.20, a variety of different structures is obtained. The cells start with a randomly initialised state, but they form shapes and patterns progressively in time.

6.3.3 Introducing mutation

A physical system in the real world is never isolated. External factors or random events might affect the cells by changing their state. This phenomenon is called mutation and one of the reasons, for example, for producing cancerous masses in advanced biological organisms or causing changes of characteristics (for instance, change in the eye colour). In some organisms,

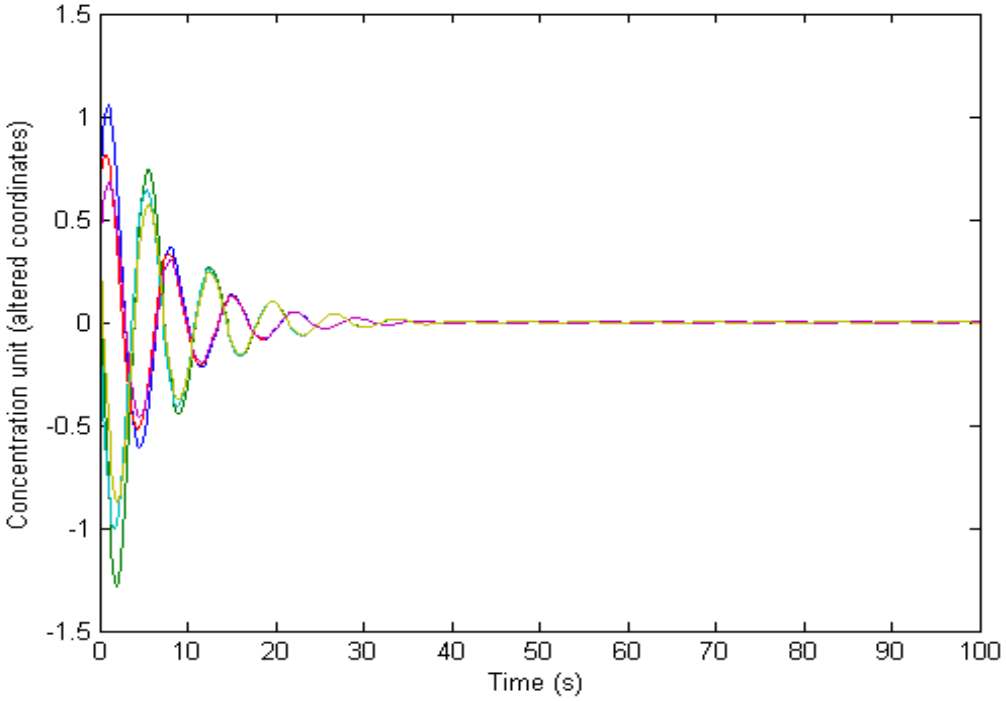

Figure 6.13: Graph showing how the behaviour of the linear system differantiates with large diffusion coefficients (experiment 2, Table 6.4 opposed to experiment 1, Figure 6.9).

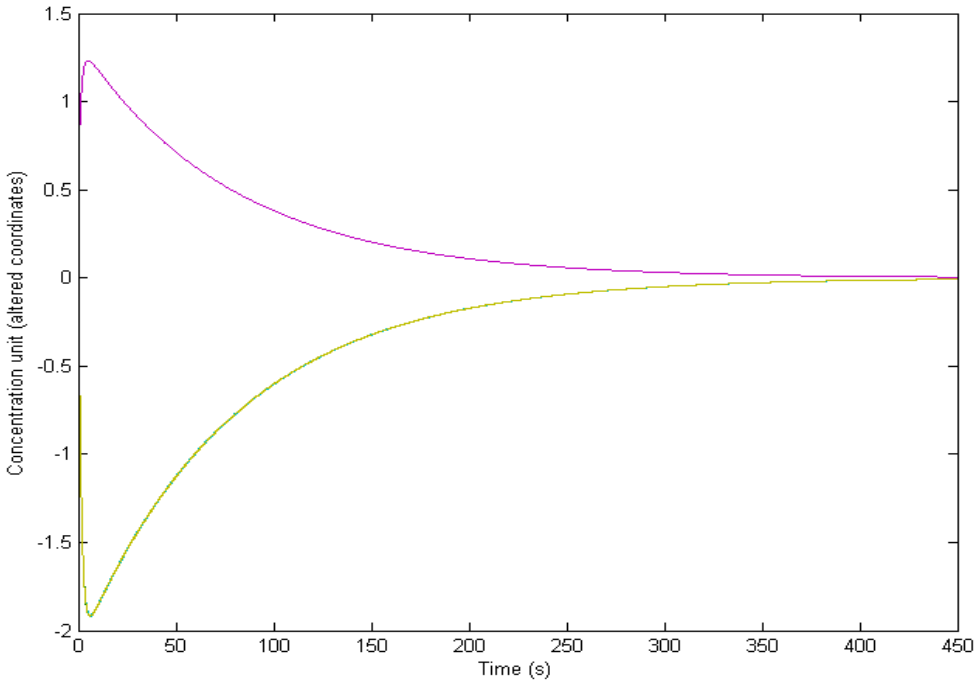

Figure 6.14: Graph showing how the behaviour of the linear system differantiates with very large diffusion coefficients (experiment 4, Table 6.4 opposed to experiment 3, figure 6.10).

Table 6.5: Experiments of non-linear models with various parameters.

No.	Model	Parameters	Diffusion Coefficients	Number of Cells
1	L-Systems	None	$D_u = 1.6, D_v = 6$	6400
2	L-Systems	None	$D_u = 1.6, D_v = 6$	22500
3	L-Systems	None	$D_u = 3.5, D_v = 16$	6400
4	L-Systems	None	$D_u = 2, D_v = 5$	6400
5	Gray Scott	$F = 40, k = 60$	$D_u = 1, D_v = 16$	22500
6	Gray Scott	$F = 30, k = 40$	$D_u = 1, D_v = 16$	22500

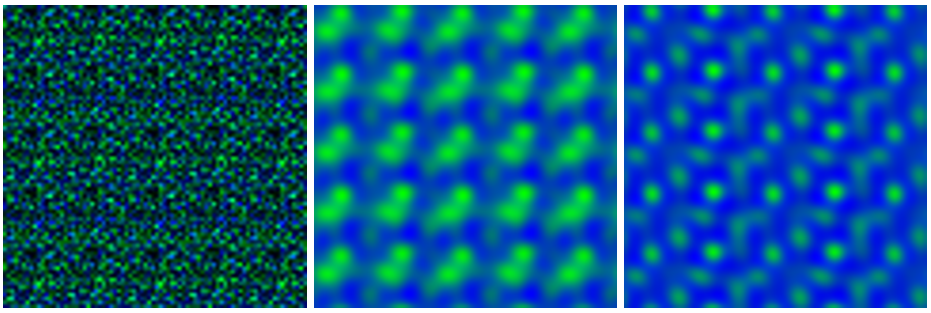

Figure 6.15: Initial, middle and final snapshots of the structure of the L-systems of experiment No. 1, Table 6.5.

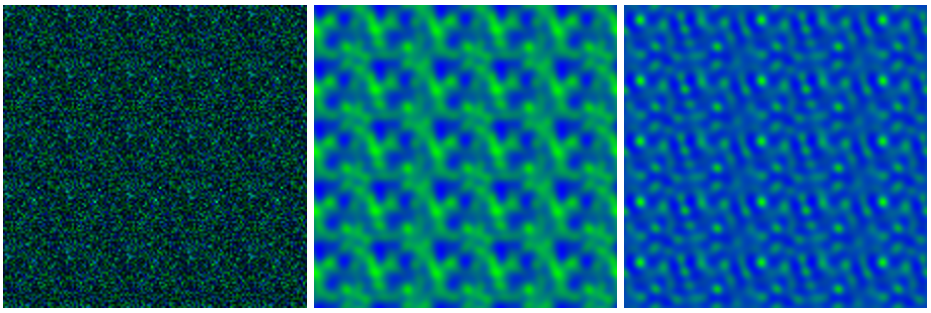

Figure 6.16: Initial, middle and final snapshots of the structure of the L-systems with more cells as shown in experiment 2, Table 6.5.

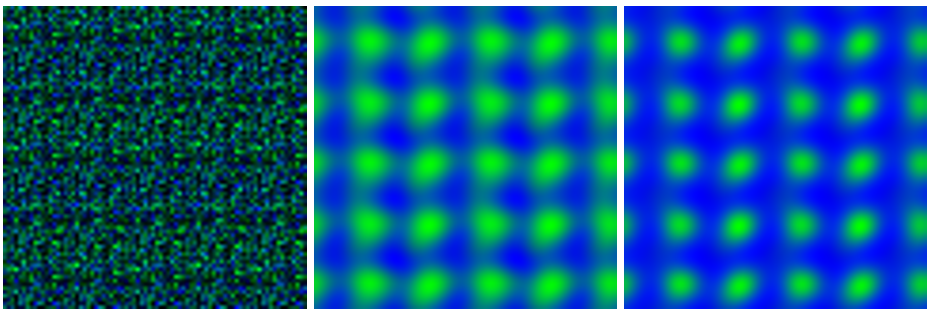

Figure 6.17: Initial, middle and final snapshots of the structure of the L-systems with different diffusion coefficients (experiment 3, Table 6.5).

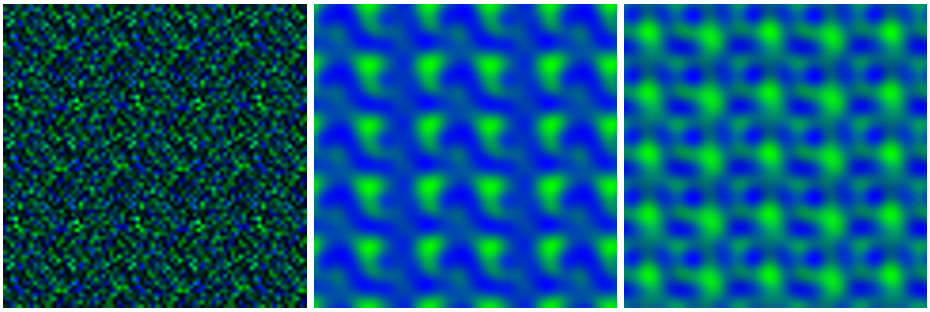

Figure 6.18: Initial, middle and final snapshots of the structure of the L-systems with different diffusion coefficients (experiment 4, Table 6.5).

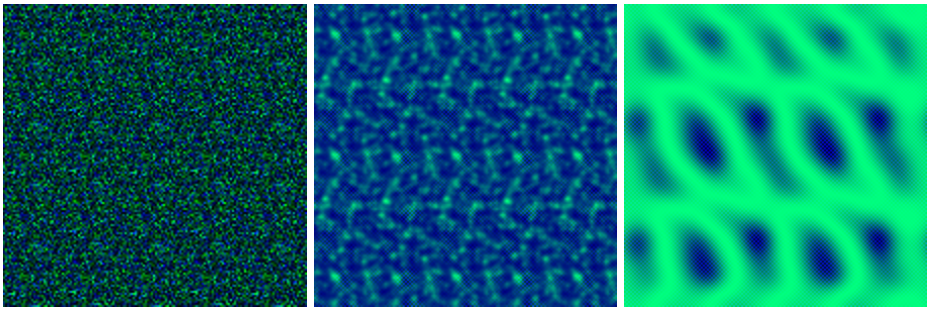

Figure 6.19: Initial, middle and final snapshots of the structure of the Gray-Scott model with parameters shown in experiment No.5, Table 6.5.

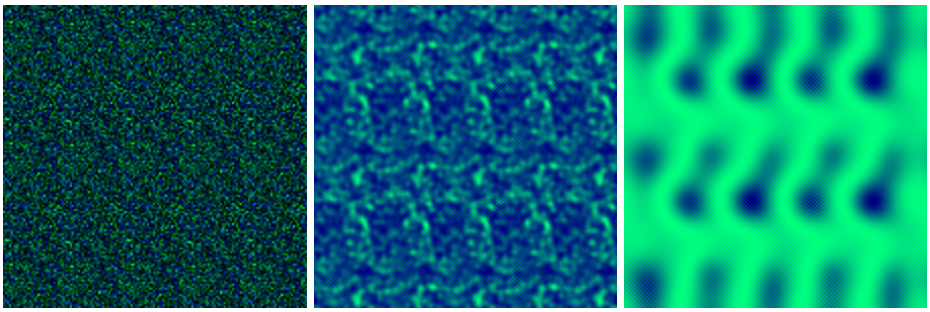

Figure 6.20: Initial, middle and final snapshots of the structure of the Gray-Scott model with different reaction rates (experiment No.5, Table 6.5).

Table 6.6: Showing the effect of mutation on different probabilities.

No.	Diffusion Coefficients	Number of Cells	Mutation Probability	Number of mutated cells
1	$D_u = 1.6, D_v = 6$	3600	0.001	2
2	$D_u = 1.6, D_v = 6$	3600	0.01	39

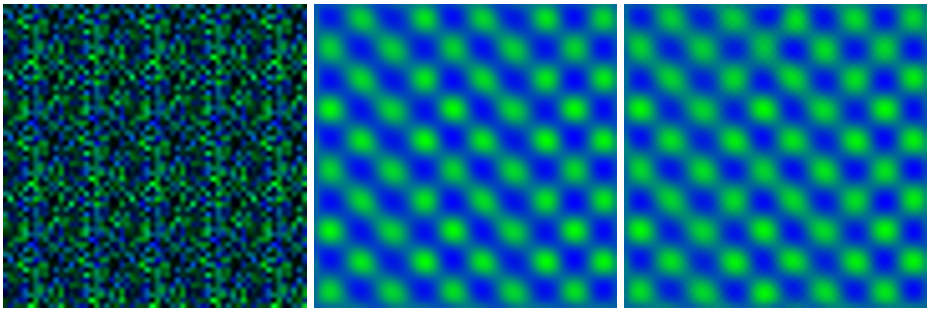

Figure 6.21: Initial state, final structure before mutation , and recovered structure after mutation with mutation probability 0.001 (experiment 1, Table 6.6).

controlled mutations may have desirable effects. In agriculture, grafting takes place in plants to make them produce better fruit [17]. Mutation occurs under different circumstances, but the concept remains the same; the behaviour of a system is changed by altering some of its properties.

A study on altering the system randomly after its development is proposed. The process is that the system is left to evolve for a specific amount of time. Then, random perturbations change some cells by altering their morphogen concentrations and the system is integrated again. The purpose is to find at what mutation probabilities the structure of the system is changed dramatically beyond recovery.

This can be done by observing the structures prior to and following mutation. Due to the colour mapping, a single cell mutation may lead to the false conclusion that the structure changed rapidly but this is not the case. What one should observe is whether the change in some cells can be absorbed by the system and if the system recovers to its previous structure.

The results show that a probability of cell mutation equal to 0.01 is sufficient to alter the

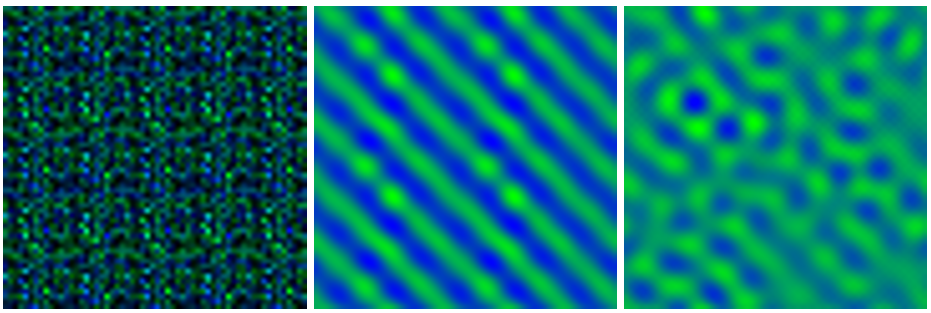

Figure 6.22: Initial state, final structure before and after mutation showing that with probability 0.01 the structure changes (experiment 2, Table 6.6).

Figure 6.23: Pattern produced by the Gizburg-Landau model.

final structure of a system.

More ideas arise on how to experiment with mutation. One is to find the highest probability at which a system can recover. As shown, a probability of 0.01 caused the structure of the system to change while the probability of 0.001 did not affect the system. What may be the threshold that brings the system to its limit?

Another idea would be to simulate the method of grafting. In this method, a tissue is taken from a plant and is inserted into another one. That process may transform, for example, a part of a mandarin tree to become an orange tree. A similar approach can be used to visualise this concept. A way of accomplishing this could be to generate two structures. In practice, the structures are matrices, in theory they are systems of cells. Then, take from the two structures a portion (a column, a row, or a small square portion) of each and substitute them. How will each structure be after integration? This is an interesting experiment that expands the results of this section.

6.4 Producing sound

Mathematical models of morphogenesis have oscillatory behaviour. Thus, they share a common behaviour with sound. The sinusoidal waves of a mathematical reaction-diffusion model can be converted into frequency spectra and then sound may be produced by driving the spectra values to the sound card.

A mathematical model that presents continuous waves is needed. The Gizburg-Landau was used as was analysed and implemented by Aly-Khan Kassam [4] Figure 6.23 shows the pattern that this model generates.

There are various ways for extracting frequency spectra values from the waves that the Gizburg-Landau model generates. The problem is the same with colour values: to find the right way for mapping the morphogen concentrations into frequency values. The method that was followed was to get the mean of the morphogen concentrations for each point in time. The result is shown in Figure 6.24.

6.5 Comparing schedulers

The scheduling solution described in Section 5.2.3 involved four schedulers. The random scheduler, a priority scheduler, the Round Robin Scheduler and the project-related diffusion-scheduler. The results show a similar performance of the Round Robin and the Diffusion

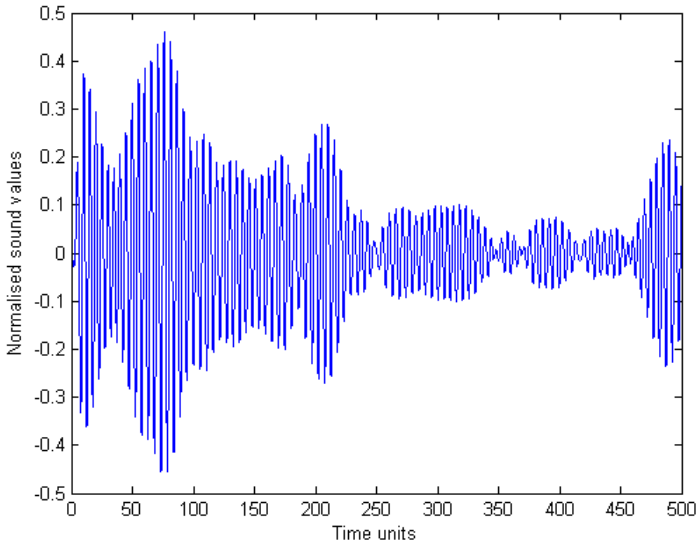

Figure 6.24: Sound spectra by calculating the mean values of the morphogens.

Table 6.7: Experiment parameters for testing the diffusion-inspired scheduler against other schedulers.

No.	Diffusion Coefficient	Number of Processes
1	1	100
2	1	1000
3	5	1000
4	15	1000

scheduler. As shown in the bar graph Figures 6.25, 6.26, 6.27, 6.28 the diffusion-inspired scheduler can execute processes faster than the other schedulers.

However, the main problem is the complexity of the implementation of the Euler's method inside the scheduler which demotivates the import of the diffusion scheduler to a real operating system. Further study can be made on how it can be imported to a distributed system by handling concurrency.

Table 6.7 shows a short amount of testing and statistical analysis which proves why further study on diffusion-inspired scheduling is needed.

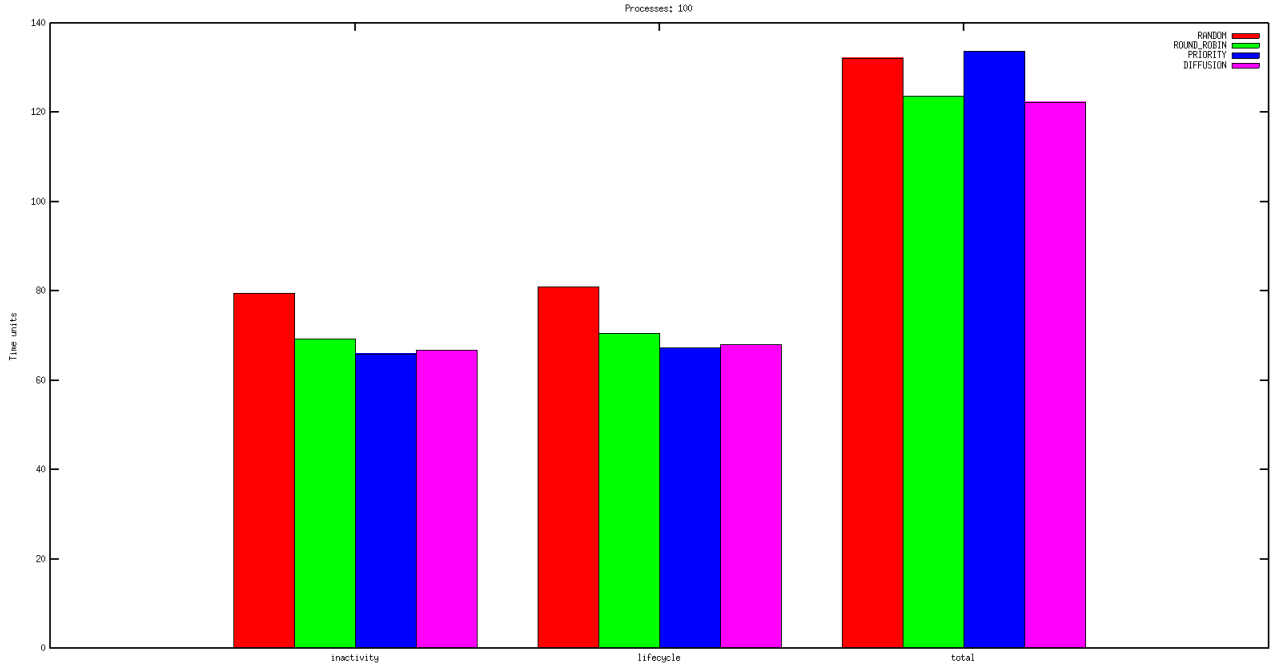

Figure 6.25: Comparison of the scheduler implementations with 100 processes in the queue (experiment No. 1 of Table 6.7).

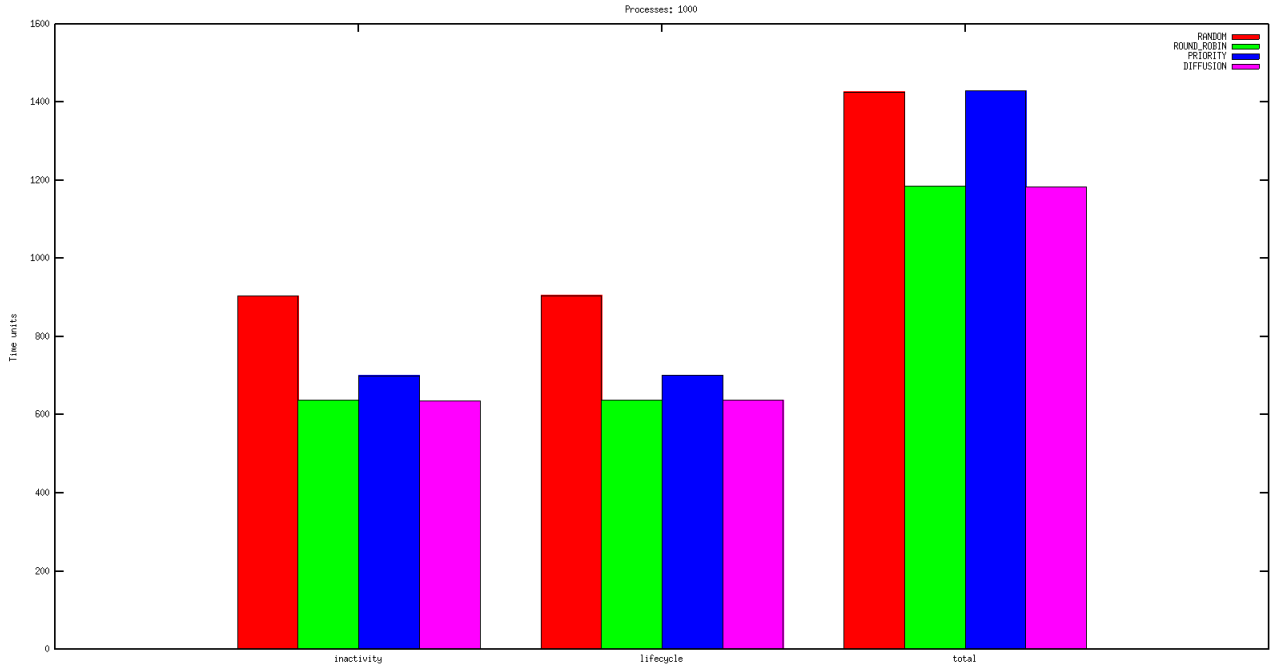

Figure 6.26: Comparison of the scheduler implementations with 1000 processes in the queue (experiment No. 2 of Table 6.7).

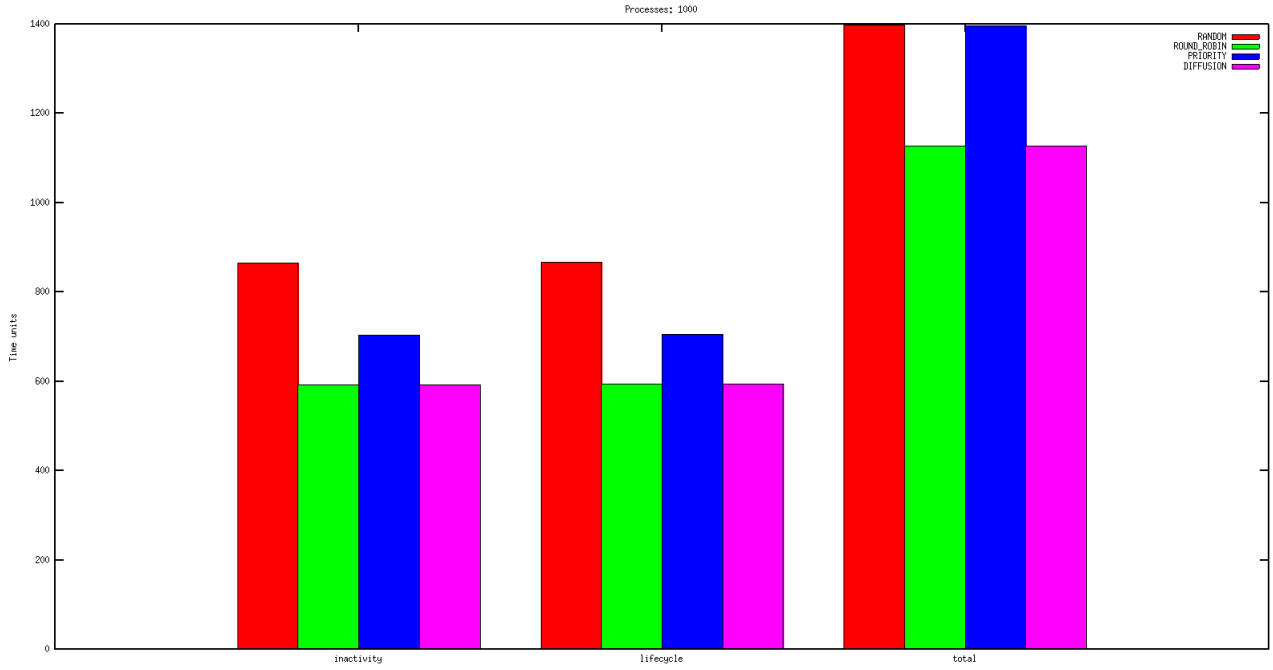

Figure 6.27: Comparison of the scheduler implementations with 1000 processes in the queue and a different diffusion coefficient (5) for the diffusion-inspired scheduler (experiment No. 3 of Table 6.7).

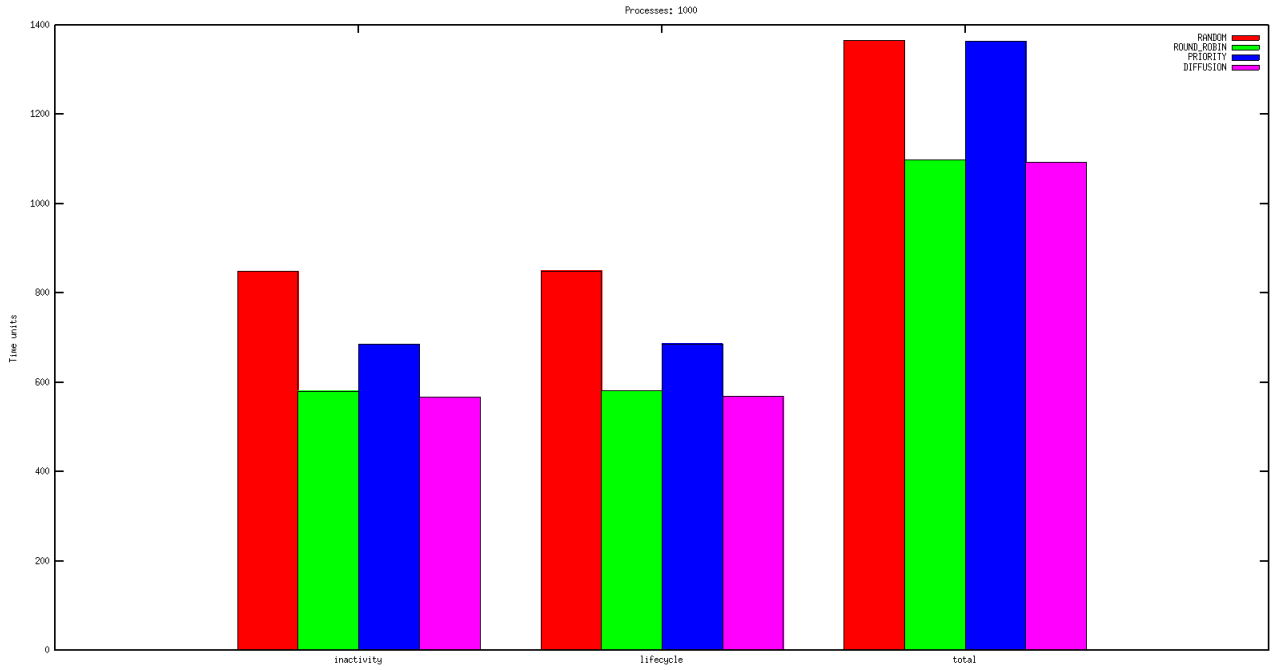

Figure 6.28: Comparison of the scheduler implementations with 1000 processes in the queue and the diffusion coefficient of the diffusion-inspired scheduler being 15 (experiment No. 4 of Table 6.7).

Chapter 7

Software development

The software development process involved implementing simulations for the mathematical models described in Chapter 4, testing that involved a basic mathematical analysis of each model and documentation of the work. The methodologies that were followed during the project can be used as a reference for the reader in order to follow or improve such a method for future research or a related project. It can also be used as a method of understanding the coding structure and then expanding the ideas that are introduced in this project.

7.1 Model simulations

The main deliverable concerning model simulations is a Matlab application. The methodology is discussed to help the reader in using the application and its source code. It might also be used as a guide for solving similar problems.

7.1.1 Code structure

The code has three main components:

- The main function that prompts the user to initialise certain parameters of the model, such as the number of cells and the amount of time to be used in integrating the equations.
- A nested function in the body of the main function that calculates the equations of a certain model. Whether it can be uncoupled from the main function is debatable; it is easier to share variables from the main function to the nested one.
- Global external functions that can be used by any application and can be easily imported by anyone who is interested in using them. Examples of such functions are the `playMovie` and `scaleColors`.

Globally available functions are shown in appendixA.

7.1.2 Testing

Testing mathematical models is challenging. In general, software testing is done by comparing the output of a body of code with hard-coded data (that is, the expected output). For testing a mathematical model, an analysis of that model needs to be done first, in order to identify what

behaviour is expected. The analysis can be automated with the use of the symbolic computational capabilities of Matlab [11] or Mathematica [18].

Testing of function implementation that do not involve integration of a system of equations is done in a traditional approach; the output results of a function for certain inputs are compared to expected outputs for those input values. The `scaleColors` function is a crucial function to the system, since it is a key part in mapping the morphogen concentrations to color interpreted values. It takes an array of morphogen concentrations and normalises their values to the range of $[0, 1]$ according to the maximum and minimum concentrations.

7.1.3 Documentation

Documentation of the work is presented in HTML format. The purpose of documenting the Matlab functions is to provide a quick reference of the functions implemented, their arguments and their interdependence. In order to automate the process of documentation a Bash script was created that gets a set of Matlab files and creates HTML files that provide information of the functions in each file (see bash scripts C.2 and C.1).

7.2 Process scheduling

The deliverable is a framework written in Java that can be used for:

1. Creating Scheduler algorithms.
2. Implementing ODE solvers.
3. Defining mathematical models.
4. Running and retrieving statistical results for each scheduler.

An additional tool written in bash script and gnuplot script is provided for converting the statistical information generated by the Java implementation into bar graphs. All scripts can be found in Appendix C.

7.2.1 Methodology

The framework is meant to be used by anyone that wants to experiment with the idea of scheduling by solving differential equations. Thus, the use of software design patterns was essential in order to make the code readable and easy to be expanded. Documentation was generated with proper commenting of the code and then the use of `javadoc` [21] and testing was done with graph generation by using gnuplot scripts.

7.2.2 Code structure

All class files are included in a single package. The naming of the files is based on software design patterns. Testing was done by the use of bash scripts and the use of the test class ”TestScheduler” for the schedulers. The graphs for analysing the effectiveness of each algorithms were generated with the program gnuplot and bash scripts to create data files for the gnuplot scripts.

Software design patterns

A fair amount of software design patterns were used in order to make the code more readable and easier to maintain. The patterns were used in reference of the 'Gang of Four' [13] and GRASP [5].

The Template pattern was used to protect variations among various ODE solvers, equations/functions and schedulers. A default implementation is given by either an interface or an abstract class and every class-object that provides a specific implementation extends that interface or abstract class. This helps in extending the framework with more ode solvers, schedulers or mathematical models.

A Memento pattern is used for the process classes, allowing them to restore to their initial state and be reused different schedulers. This helps in the statistical analysis, because schedulers must be compared to each other under the same circumstances. For instance, a process is created with random properties such as how many time cycles are needed to finish at which points in time it blocks or crashes etc. All process properties are shown in Figure 5.4.

7.2.3 Testing

Testing was done by producing graph plots and bar graphs in order to observe abnormalities in the parameters of the processes or schedulers. Abnormalities were expected by the use of wrong time step in the Euler's method implementation or bugs in general that restricted the program to run according to its requirements. The graphs and plots were generated with gnuplot with a middle layer script written in bash. The role of the script was to convert the data of the scheduler statistics into gnuplot data format.

7.3 Scripts and tools

A set of scripts were implemented in order to automate the process of testing, documentation and statistical analysis of the scheduling solutions. All scripts were written in Bash on a Linux based operating system. In addition, gnuplot was used in order to import data and generate graphs to visualise the statistical results. The decision for implementing separated scripts to automate the tasks described was proved to have a significant positive impact during the project development. Each script and its role is shown in Table 7.1.

File-name	Language	Description	Source-code
matdoc	Bash	Reads a matlab file that contains and creates a representative HTML file that contains the functions defined in the matlab file. Each function is given a description according to the comments it has before its definition.	C.1
matdoc_pages	Bash	Gets all HTML files and outputs an HTML body containing a list of links for those files. If its output is piped to a file called index, the whole documentation is ready to be read.	C.2
run-test	Bash	Executes the TestScheduler class of the process scheduling solution with a given argument that represents the number of processes existing initially in the queue and then, converts the output data into data that can be read by gnuplot script.	C.4
run-tests	Bash	Gets 3 arguments: number of processes to start, number of processes to finish and the step to reach that number of processes. Then it calls run-tests and gets the generated data to plot them using the gnuplot script totalTime.p	C.5
totalTime.p	gnuplot	Reads a data file and generates the bar plots to visualise the results as shown in paragraph 6.5.	C.3

Table 7.1: Description of the implemented scripts that automate documentation and testing.

Chapter 8

Conclusion

The project generated animations and movies that visualise how cells can form complex structures. This was done with the use of two mathematical models, the Gray-Scott model and the L-Systems equations. A third model, the Gizburg-Landau was used to explore the possibility of sound generation. The ordinary differential equations of those models, were integrated with the ODE solvers of matlab and matrix operations. The results were shown as images by using conversion algorithms to normalise chemical concentrations into ranges of colour values. Results were shown in the form of plotted graphs as well.

Further goals of the project were to exploit the idea of morphogenesis to research other concepts. One was to generate sound instead of images. Experimentation with sound led to the definition of ideas and implementation of tools which may be used in the future to explore the possibility of audio production by the use of models of morphogenesis. The graph results of the project in terms of audio production present similarities to the sound waves. Thus, further analysis and exploration of the concept is encouraged.

Next, the project studied how mutations affect the state of a system. The experiments take a finished structure of a system and alter the morphogen concentrations of some cells with a given probability. Then, the system is integrated again. It has been shown that with a very low probability (0.001) that affects only a tiny amount of cells, the system recovers the structure. On the other hand, a probability of 0.01 is able to alter the structure of a system significantly.

The latter experiment arises further questions or ideas on how to use mutation to study morphogenesis. What will happen if the mutation is done according to the values of a different system, in the same way that grafting is done for plants? What would be the outcome if the mutation changes the diffusion coefficients as well? Is there a way to recover the system to regain its pre-mutant structure?

The last goal was to find a way to apply the concepts of morphogenesis in a computer science problem. The ‘diffusion-inspired algorithm’ was to create a process scheduler that uses diffusion to give each process different time slices on which processes are allowed to be executed. Although the algorithm has a high computational complexity it seems to be more effective than a random scheduler or a priority scheduler. The difference between the diffusion-inspired scheduler and the Round Robin is very small. For some parameters the diffusion-inspired scheduler even seems to have less inactivity and less total time than the Round Robin.

There are a lot of parameters in the scheduling problem that have not been addressed. The diffusion-inspired scheduler was not compared to itself with different parameters. In addition, schedulers may be tested with processes that have high or low rates in requesting input/output time. Thus, exploring on which environments each scheduler is best, or which parameters make

the diffusion-inspired scheduler better in different environments is suggested. The software framework that was developed provides easy to maintain and expand code motivating further experiments.

The project combined a mixture of biology, chemistry, mathematics, engineering and computer science. Time limited further investigation of more conepts and ideas. The conclusion of all this work, is that morphogenesis is not just a mathematical model that generates patterns. It is a framework enabling the testing of physical systems, which has the potential to give new ideas and applications for researching or experimenting with concepts that reflect in the real life world.

Bibliography

[1] A. M. Turing, "The chemical basis of morphogenesis," *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences*, vol. 237, no. 1, 1952.

[2] M. R. Garvie, P. K. Maini, and C. Trenchea, "A Methodology for Parameter Identification in Turing Systems,"

[3] B. M. Heineike, "Modeling morphogenesis with reaction-diffusion equations using galerkin spectral methods," tech. rep., DTIC Document, 2002.

[4] A.-K. Kassam, "Solving reaction-diffusion equations 10 times faster," 2003.

[5] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, 3 ed., Oct. 2004.

[6] J. S. McGough and K. Riley, "Pattern formation in the GrayScott model," vol. 5, no. 1, p. 105121, 2004.

[7] C. G. Jennings, "Turing's reaction-diffusion model of morphogenesis," January 2011.

[8] L. Perko, *Differential Equations and Dynamical Systems*. New York: Springer, 2001.

[9] J. K. Aggarwal, *Notes on Nonlinear Systems*. Van Nostrand Reinhold, Jan. 1972.

[10] F. Diacu, *Differential Equations: Order and Chaos: An Introduction to Differential Equations*. W. H. Freeman, Oct. 2000.

[11] MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[12] T. Williams and C. Kelley, "Gnuplot."

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns CD: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1 ed., May 1998.

[14] O. Koch, P. Kofler, and E. B. Weinmller, "The implicit euler method for the numerical solution of singular initial value problems," *Applied numerical mathematics*, vol. 34, no. 2-3, p. 231252, 2000.

[15] A. S. Tanenbaum, *Modern Operating Systems*. Pearson, 3 ed., Dec. 2007.

[16] S. Childress, "Case sudy 2: Turings model of chemical morphogenesis," 2005.

[17] A. Nelson, *Principles of Agricultural Botany*. Lightning Source Incorporated, Mar. 2007.

[18] Mathematica, *version 8.0.4*. Wolfram, 2011.

[19] J. P. La Salle, *Lecture notes on stability and control*. Center for Dynamical Systems; Division of Applied Mathematics, Brown University, 1966.

[20] S. Lynch, S. Lynch, and Birkhauser, *Dynamical Systems with Applications using MATLAB*. Birkhuser, 2004 ed., June 2004.

[21] javadoc, *version 5.0*. Oracle.

Appendices

Appendix A

Matlab programs

A.1 Template reaction_diffusion_model.m

```
%%Call this function to run a reaction-diffusion model from
%%L-Systems and have visual view of how the cells get organised
function nlm = reaction_diffusion_model()

    Du=1.6; %diffusion coefficient 1
    Dv=6; %diffusion coefficient 2
    NumberOfCells=input('Number of cells: ');
    side=sqrt(NumberOfCells);
    h=0.01;


    systemU=imageToPattern(random('norm', 0, 1, side,side)*12 + random('unif', 0, 1, side, side)*2, 5);
    systemV=imageToPattern(random('norm', 0, 1, side,side)*12 + random('unif', 0, 1, side, side)*2, 5);
    side=size(systemU, 1);
    NumberOfCells = side*side;
    system=[Two2oneD(systemU); Two2oneD(systemV)];
    tLimit = input('Time length: ');
    tspan = [0 tLimit];
    options = odeset('NonNegative',(1:side*side));

    [t, uv] = ode45(@im, tspan, abs(system), options);
    plot(t, uv);

    input('Press enter to proceed to movie');
    fig1=figure(1);
    winsize = get(fig1,'Position');
    winsize(1:2) = [0 0];
    numframes=length(t);
    A=moviein(numframes,fig1,winsize);
    set(fig1,'NextPlot','replacechildren');



    for i=1:numframes
        imshow(imresize(cellsToSurf(i, uv, side), [300, 300]));

        A(:,i)=getframe(fig1,winsize);
    end
    movie(fig1,A,1,5,winsize);

    movie2avi(A, 'movie.avi');



    %%Call this inside an ode solver to solve an L-Systems equation with diffusion in a torus
    function integrate_model = im(tspan, cells)
        U=cells(1:length(cells)/2);
        V=cells(length(cells)/2 + 1:end);
```

```
        for c=1:length(U);

                adj=get_adjacents(c, side);
                diu(c)=Du*(U(adj(1)) + U(adj(2)) + U(adj(3)) + U(adj(4)) - 4*U(c)); %diffusionU
                div(c)=Dv*(V(adj(1)) + V(adj(2)) + V(adj(3)) + V(adj(4)) - 4*V(c)); %diffusionV
                Reu(c)=U(c)*V(c) - U(c) - 12;
                Rev(c)=16-U(c)*V(c);
                dU(c)=Reu(c)+diu(c);
                dV(c)=Rev(c) + div(c);

        end


        integrate_model = [dU'; dV'];
    end

end
```

A.2 Template playMovie.m

```
%%gets an Array of size size(time) containing frames nxm and
function playMovie(gl, time)
    input('Press enter to proceed to movie');
    fig1=figure(1);
    winsize = get(fig1,'Position');
    winsize(1:2) = [0 0];
    numframes=length(time);
    A=moviein(numframes,fig1,winsize);
    set(fig1,'NextPlot','replacechildren');
    size(gl(1))
    for j=1:numframes
        imshow(gl(:, :, j));

        A(:,j)=getframe(fig1,winsize);
    end
    movie(fig1,A,1,5,winsize);

    movie2avi(A, 'movie.avi');
    %gl=real(ifftn(v));
end
```

A.3 Template initialiseA.m

```
function initialiseA = eq_point()
%currently works with 2 chemicals
a11=rand();
a22=-a11-rand();
a12=rand();
a21=(a11*a22)/a12 - rand();
initialiseA=[a11 a12; a21 a22];
```

Appendix B

Java programs

B.1 AbstractFunction.java

```
/**
 *
 *Provides an implementation of a function so it can be
 *used in various ode's. Instead of developing a symbolic
 *solver this is the easiest way to protect variations for each function.
 * @author nicolav0
 *
 */
public abstract class AbstractFunction
{
  protected double result = 0;

  /**
   *
   * @param elems
   * @param params
   * @return result
   * Defines the actual function and calculates the results
   */
  protected abstract double calculate(int elems, double... params);

  /**
   *
   * @param elems
   * @param params
   * @return a new calculated result
   * Call this method to run a calculation of the function and get a new result
   */
  public double getResult(int elems, double... params)
  {
    result = calculate(elems, params);
    return result;
  }

  /**
   *
   * @param elems
   * @param params
   * @return a new calculated result
   * Overload to ensure that the use of Double instead of double does
   * not affect the functionality
   */
  public double getResult(int elems, Double... params)
  {
    int i = 0;
    double[] values = new double[params.length];
    for (Double d : params)
    {
```

```
        values[i++] = d.doubleValue();
      }
      result = calculate(elems, values);
      return result;
    }

    /**
     * get the last calculated result, useful to check the previous
     * value if there is one before running another calculation
     * @return previous result
     */
    public double getResult()
    {
      return result;
    }


}
```

B.2 ODE.java

```
/**
 * interface for implementing ODE solvers
 * @author nicolav0
 *
 */
public interface ODE
{

  void integrate(int steps, double[] initialConditions, double h);

  double[][] solve(int steps, double[] initialConditions, double h);

  double[] getResult(int time);


  double[][] getResults();


  public String toString();

}
```

B.3 AbstractODE.java

```
/**
 * class that provides a framework for various ODE solvers
 * to be implemented.
 * @author nicolav0
 *
 */
public abstract class AbstractODE implements ODE
{
  protected double[][] result;
  AbstractFunction f;
  /**
   * each solver requires a function that will integrate
   * @param f
   */
  public AbstractODE(AbstractFunction f)
  {
    this.f = f;
    result = null;
  }

  /**
   * @param steps
   * @param initialConditions
```

```
   * @param h
   * Call this to integrate your equation. It needs the implementation of solve()
   */
  public void integrate(int steps, double[] initialConditions, double h)
  {
    result = new double[steps+1][initialConditions.length];
    result=solve(steps, initialConditions, h);
  }

  /**
   * implement this so you can call integrate
   */
  public abstract double[][] solve(int steps, double[] initialConditions, double h);

  public double[] getResult(int time)
  {
    return result[time];
  }

  /**
   * get the results in an array of double time x results
   */
  public double[][] getResults()
  {
    return result;
  }

  /**
   * Human readable representation of the solution
   */
  public String toString()
  {
    String output = "Result\n========\n";
    for (int i=0; i < result.length; i++)
    {
      output+="Step " + i + ": ";
      for (int j=0; j < result[i].length; j++)
        output+=result[i][j] + " ";
      output+="\n";
    }
    return output;
  }
}
```

B.4 Scheduler.java

```
import java.util.ArrayList ;
import java.util.Arrays;
import java.util.List;
/**
 * provides a framework for implementing various schedulers.
 * @author nicolav0
 *
 */
public abstract class Scheduler {

  protected static double sliceSize = 0.001;

  protected List<Process> processes;
  public Scheduler()
  {
    processes = new ArrayList<Process>();
  }

  public void addProcess(Process p)
  {
    processes.add(p);
  }

  public Process getProcess(int index)
```

```
  {
    return processes.get(index);
  }

  public abstract void run();


}
```

B.5 Process.java

```
import java.util.ArrayList;

/**
 * A definition of a process that has various characteristics such as
 * process events, size in cycles, probability for IO blocking or crashing.
 * @author nicolav0
 *
 */
public class Process {

  public enum ProcessEvent {RUN, IO, CRASH};
  public enum ProcessMessage {OK, PAUSED, FINISHED, ERROR}
  protected double initialSize;
  protected double size;
  private ArrayList<ProcessEvent> path;
  private double ioProbability;
  private double crashProbability;
  public final int id;
  private boolean isReset = false;
  private int count;
  public Process(int id)
  {
    size = Math.random()*3;
    path = new ArrayList<ProcessEvent>();
    ioProbability = Math.random() / 50; //at most 1/50 prob of expecting io
    crashProbability = Math.random()/100;// / 100; //at most 1/100 prob of crashing
    this.id = id;
    initialSize = size;
    isReset = false;
  }

  /**
   * create a process with a priority count
   * @param id
   * @param count
   */
  public Process(int id, int count)
  {
    this(id);
    this.count = count;
  }

  /**
   * reset the current process for reuse
   */
  public void reset()
  {
    size = initialSize;
  }

  /**
   * execute the Process for time amount of cycles
   * @param time
   * @return
   */
  public ProcessMessage execute(double time)
  {
    //System.out.println("Given a time slice " + time);
```

```
    int tStep = 0;
    for (double t=0; t <= time; t+=Scheduler.sliceSize)
    {
      if (isReset)
      {
        switch (path.get(tStep++))
        {
        case IO:
          path.add(ProcessEvent.IO);
          size-=t;
          return ProcessMessage.PAUSED;
        case CRASH:
          return ProcessMessage.ERROR;
        }

        if (size - t <= Math.exp(-10))
        {
          return ProcessMessage.FINISHED;
        }
      }
      double condition = Math.random();
      if (condition < ioProbability)
      {
        path.add(ProcessEvent.IO);
        size-=t;

        return ProcessMessage.PAUSED;
      }
      if (condition < crashProbability)
      {
        path.add(ProcessEvent.CRASH);
        return ProcessMessage.ERROR;
      }
      if (size - t <= Scheduler.sliceSize*0.1)
      {
        return ProcessMessage.FINISHED;
      }
    }

    size -= time;
    return ProcessMessage.OK;

  }

  /**
   * get the priority of this process
   * @return
   */
  public int getCount() {

    return count;
  }

}
```

Appendix C

Bash scripts and gnuplot scripts

C.1 matdoc

```
#!/bin/bash
#%% are description comments
#main function
# %%
# nested functions

h=1
IFS=$'\n'
echo "<html>"
for line in `egrep -w "␣*%%.*|␣*function.*+" $1`; do
  comm=`echo $line | grep "%%"`
  if [ -n "$comm" ]; then
    next_description=`echo $comm | tr -d '%'`;
  else
    if [ "$h" -eq "1" ]; then

      echo "<h$h>$line</h$h>"
      echo "<p>$next_description</p>"
      echo "<ul><h2><b>Nested␣Functions</b></h2>"
      h=3;
    else
      echo "<li>"
      echo "<h$h>$line</h$h>"
      echo "<p>$next_description</p>"
      echo "</li>"
    fi

  fi
done
echo "</ul>"
echo "</html>"
```

C.2 matdoc_pages

```
#!/bin/bash
cd $1;
curr_dir=`pwd`
echo "<html>"
for file in *.html; do
    echo  "<a␣href=\"$curr_dir/$file\">$file</a><br>"
done
echo "</html>"
```

C.3 totalTime.p

```
set title window_title
set style data histogram
set style histogram cluster gap 1

set style fill solid border rgb "black"
set auto x
set yrange [0:*]
set ylabel 'Time units'
plot 'times_data.dat' using 2:xtic(1) title col, \
        '' using 3:xtic(1) title col, \
        '' using 4:xtic(1) title col, \
        '' using 5:xtic(1) title col
```

C.4 run-test

```
#!/bin/bash

#set gnuplot data for total time
totalTimes=`java TestScheduler $1 | cat >result.log`
totalTimes=`cat result.log | grep "Total time: " | cut -d ':' -f 2 | tr -d ' '`
averages=`cat result.log | grep "Average: " | cut -d ':' -f 2 | tr -d ' '`
avg_line1=`echo $averages | cut -d ' ' -f 1,3,5,7`
avg_line2=`echo $averages | cut -d ' ' -f 2,4,6,8`
total_time=`echo $totalTimes | cut -f 1-4`
{
echo "Time_mesure" `cat algorithms`
echo "inactivity $avg_line1"
echo "lifecycle $avg_line2"
echo "total $total_time"
} | cat >times_data.dat
#gnuplot -p totalTime.p
```

C.5 run-tests

```
#!/bin/bash
for ((i=$1; i <=$2; i+=$3)); do
    ./run-test $i
    gnuId=`pgrep gnuplot`
    if [ -n "$gnuId" ]; then
        kill $gnuId
    fi

    gnuplot -geometry 1600x900 -display :0.0 -e "window_title='Processes: $i'" -p totalTime.p
    sleep 2
done
```