Object Oriented Programming for Scientific Computing

Parag Patel
University of Edinburgh – Own Work Declaration

This sheet must be filled in (each box ticked to show that the condition has been met), signed and dated, and included with all assessments - work will not be marked unless this is done

This sheet will be removed from the assessment before marking

Name:  ………………………………………  Number:   …………………  
Course/Programme:       ……………………………………………………………………...
Title of work:       ……………………………………………………………………………….

I confirm that all this work is my own except where indicated, and that I have:

- Clearly referenced/listed all sources as appropriate  □
- Referenced and put in inverted commas all quoted text (from books, web, etc)  □
- Given the sources of all pictures, data etc. that are not my own  □
- Not made any use of the report(s) or essay(s) of any other student(s) either past or present  □
- Not sought or used the help of any external professional agencies for the work  □
- Acknowledged in appropriate places any help that I have received from others (e.g. fellow students, technicians, statisticians, external sources)  □
- Complied with any other plagiarism criteria specified in the Course handbook  □

I understand that any false claim for this work will be penalised in accordance with the University regulations  □

Signature ………………………………………
Date ………………………………………

Please note: If you need further guidance on plagiarism, you can
1. Consult your course book
2. Speak to your course organiser or supervisor
3. Check out http://www.aaps.ed.ac.uk/regulations/Plagiarism/Intro.htm

Please read the notes about the use of plagiarism detection software overleaf.
Abstract

Software industry has benefited from the object oriented programming paradigm but use of object oriented programming in scientific computing is experimental. One of the main issues of scientific computing is performance. Many scientific codes have been generated over the large period of time and optimized for particular machines. The object oriented programming is observed slower than functional programming. This legacy of code and observed performance loss of the object oriented programming has lead to the experimental use of object oriented programming in scientific community. But object oriented programming provides features to develop larger and more complex numerical codes. In this report different object oriented implementations of the predator prey algorithm will be discussed. The main goal of this project will be to carry out object oriented features like robustness, easy maintenance and reusability for scientific codes.
Contents

1 Introduction 1

1.1 Previous Work 1

2 Object Oriented Programming for Scientific Computing and Predator Prey Algorithm 3

2.1 Why Object Oriented Programming 3

2.1.1 Encapsulation 3

2.1.2 Inheritance 3

2.1.3 Polymorphism 4

2.2 Object Oriented Programming for Scientific Computing 4

2.3 Predator Prey Simulation 4

2.3.1 Application of Predator Prey Simulation 5

2.4 Predator Prey Model 5

2.5 Why Predator Prey Model 6

3 Design 7

3.1 Mathematical Model 7

3.2 Design 8

3.2.1 Design 1 8

3.2.2 Design 2 9

3.2.3 Design 3 10

3.2.4 Design 4 12

3.2.5 Design 5 13

3.2.6 Design 6 15

3.2.7 Extension to all the designs 16

4 Implementation 18

4.1 Functional Implementation 18

4.1.1 Fictional C Implementation: 18

4.1.2 Functional Java Implementation 19
4.2 Object Oriented Implementation .................................................. 20
  4.2.1 Implementation 1 ............................................................... 21
  4.2.2 Implementation 2 ............................................................... 24
  4.2.3 Implementation 3 ............................................................... 27
  4.2.4 Implementation 4 ............................................................... 30
  4.2.5 Implementation 5 ............................................................... 32
  4.2.6 Implementation 6 ............................................................... 33
  4.2.7 Implementation 7 ............................................................... 34

5 Performance ................................................................................. 36
  5.1 Performance of Functional Implementations ................................. 36
  5.2 Performance of Object Oriented Implementations ......................... 37

6 Conclusion and future work .......................................................... 44

A Compilation of Programs and Submission of Jobs ........................................ 46
  A.1 Compilation and Jobs Submission for C Program ............................. 46
  A.2 Compilation and Job Submission of Java Program: ............................ 47

B Default Parameter List ..................................................................... 48

C Work Plan ..................................................................................... 49
  C.1 The following was the initial Work Plan for the project ...................... 49
  C.2 Implementation of the Work Plan ................................................ 49
## List of Figures

1. Predator Prey Cycle ........................................... 6
2. Conceptual Design 1 ............................................ 8
3. Class Diagram of Design 1 ................................. 9
4. Conceptual Design 2 ........................................... 10
5. Class Diagram of Design 2 ................................. 10
6. Conceptual Design 3 ........................................... 11
7. Class Diagram of Design 3 .................................. 11
8. Conceptual Design 4 ........................................... 12
9. Class Diagram of Design 4 .................................. 13
10. Conceptual Design 5 .......................................... 14
11. Class Diagram of Design 5 ................................. 14
12. Conceptual Design 6 .......................................... 15
13. Class Diagram of Design 6 .................................. 16
14. Flowchart for C Functional Implementation ............. 18
15. Flowchart for Java Functional Implementation .......... 20
16. Class Diagram of Implementation 1 ....................... 21
17. Communication Diagram of Implementation 1 .......... 22
18. Class Diagram of Implementation 2 ....................... 25
19. Communication Diagram of Implementation 2 .......... 26
20. Class Diagram of Implementation 3 ....................... 28
21. Communication Diagram of Implementation 3 .......... 29
22. Class Diagram of Implementation 4 ....................... 30
23. Communication Diagram of Implementation 4 .......... 31
24. Class Diagram of Implementation 5 ....................... 32
25. Class Diagram of Implementation 6 ....................... 33
26. Class Diagram of Implementation 7 ....................... 34
27. Communication Diagram of Implementation 7 .......... 35
28. Performance Chart for Functional Code ................. 37
List of Tables

1. Calculation and Total Time for C Functional Implementation .................. 36
2. Calculation and Total Time for Java Naive Functional Implementation .............. 36
3. Average Calculation and Total Time of C and Java Functional Codes ................ 36
4. Calculation and Total Time for Implementation 1 ................................ 37
5. Calculation and Total Time for Implementation 2 ................................ 38
6. Calculation and Total Time for Implementation 3 ................................ 38
7. Calculation, Total and Initialisation time for Implementation 4 .................... 38
8. Calculation and Total Time for Implementation 5 ................................ 38
9. Calculation, Total and Initialisation Time for Implementation 6 .................... 38
10. Calculation, Total and Initialisation Time for Implementation 7 .................... 38
11. Average calculation and Total time of all OO Implementation ........................ 39
12. Calculation and Total Time for Implementation 2 with tiling .......................... 41
13. Calculation, Total and Initialisation Time for Implementation 4 with tiling .......... 41
14. Calculation, Total and Initialisation Time for Implementation 6 with tiling .......... 41
15. Performance Chart with and without Tiling ....................................... 42
Acknowledgements

First, I would like to thank my supervisors Judy Hardy and Lorna Smith for their guidance and support throughout this project. I would also like to thank Balaji Lalshman and classmates for their assistance. And lastly, I would like to thank my parents who have been continuous source of inspiration for my project work.
1 Introduction

In recent years, parallel processing has improved immensely which has led to availability of a very powerful tool for solving many scientific problems[3]. For example Numerical Weather Prediction (NWP), The Integrated Forecast System (IFS) from European Centre for Medium-range Weather Forecast (ECMWF) runs at 400km resolution over 60 levels to calculate for 10 days, on 256 processors on scalar computer (IBM p600) in 1 hour at 80Gflops (317 TFlops)[1]. Programming scientific applications and debugging simulation code like NWP involves large and complex codes. The complexity and resource demands of present day scientific codes create the need for more flexible solutions than those offered by conventional programming style based on a succession of subroutine calls. Moreover reusability of scientific codes lead to more difficulties towards the development of scientific code. As code generated for one parallel machine can not be reused easily for another parallel machine. To reduce the efforts spent on software issues, one can advantageously apply modern software development techniques to make the implementation aspect clearer, simpler and more effective.

One of the possible solutions can be object oriented programming (OOP), because OOP offer features that can improve the design, implementation, and maintenance of large scientific codes[2]. Scientific code also involves parallel programming. It would be interesting to check how the use of object oriented features can benefits scientific parallel codes. In parallel code, processes are created which run on different processors. These processes have their own data, which can be shared between different processes by messages. This paradigm of process closely relates to the notation of object in object oriented programming language. In object oriented programming, objects have their own data and methods and objects communicate with other objects through interface. So object oriented programming can be a favourable choice for parallel scientific application[3]. OOP provide the concept of subclasses and virtual function, these concepts are respectively known in object oriented terminology as inheritance and dynamic binding. These features of OOP make it reusable code whereas data encapsulation and templates provide robust code.

The object oriented programming has been used for scientific computing as well. For example POET[4] and POOMA[4] are framework created for parallel computing using C++. Charm++ [3] provides object oriented programming for parallel computing. Another example is Application Simulation Software (AVS) [7] which is used to simulate physical problem using High performance computing created using C++. The Globus Toolkit is open source software used to build grid system and applications. One of the main components of this toolkit is web services which are being developed using object oriented programming language like java.

To represent the features of object oriented programming for scientific computing, we will consider predator-prey algorithm. The predator-prey algorithm is very important problem in ecology. It studies how different species of animals depend on each other for their existence and how the population of one species of animal effects the populations of the other species of animals. [8]. We choose the predator-prey algorithm, because it is solved using Partial Differential Equations (PDE). Many scientific codes involve numerical solution of PDEs and often take an undesirably large amount of time. Moreover in the predator-prey algorithm, hare, puma and world surface can be seen as different objects, which would be favourable for different object oriented designs for predator-prey algorithm. The main reason to choose the predator-prey algorithm that it is mathematically complicated enough to carry out the basic programming issues, but on the other hand, it is simple enough to keep focus on basic implementation issues.

1.1 Previous Work

In this section, we will discuss work done by other researcher in object oriented programming for High Performance Computing.
“Treatment Planning System” [5] is a large scientific application, which is used to do graphic simulation of radiation treatments for cancer. The main goal of application is to provide three dimensional anatomy and treatment geometry, graphics editing of the input data, multiple views in different windows and some other features, which should be easy to use, as the application is going to be used in clinics with no computer expertise.

This system is divided into one object for each anatomical structure and radiation source, while BLOCK, STRUCTURE, CONTOUR and others are sub-objects of these two objects and linked to each other via pointer. This object structure has provided better representation of data in compare to array in traditional procedural programming, as objects were allocated when they were needed and dispose of them when work is done. Objects with correct and constant results were added to data structure and others were dropped.

In initial procedural style code, each operation was written in procedure and objects were passed in as parameter. These procedures were big case statement with a case branch for each object. Instead of adopting this model for Object oriented code, message passing is used as an alternative. Instead of passing objects to operation, operation was passed as message to objects, specifying operation to be performed on objects.

Object oriented programming is more convenient for providing multi window display. This was accomplished by treating each views as a different global objects, which describes where windows lies, scale from real world to screen view and other details. Object oriented programming has helped to provide much simpler implementation of system with better data representation. Though it required recompiling whole system for addition of new features but it is easier to add new feature as it is not altering implementation of other methods.

“Solving System of Partial Differential Equations using Object-Oriented Programming Techniques with Coupled Heat and Fluid Flow as Example” [2] by Hans Petter Langtangen and Otto Munte is another scientific application developed in Object Oriented Programming. The main goal of application is to provide a PDE solver as a standalone independent object in order to easily assemble with other software. PDE solver has been implemented using utilities available from the Diffpack software library.

Diffpack library provides basic building blocks for PDE solvers as C++ classes. Apart from PDE solver to solve PDE equations, it provides GridFE class to hold information about nodal coordinates, FieldFE to represent scalar finite elements field, LinEqAdm class for representation and solution of linear systems and other classes. Diffpack library also provide NonLinEqSolverUDC class, which is defined by user to solve non linear equations. This class also define interface called makeAndSolveLinearSystem, which allows creating non linear algorithm without affecting general library general library functionality. For example NewtonRaphson subclass implements non linear method with affecting other nonlinear solutions currently in use.

For solving coupled heat and fluid flow system it required to solve momentum and energy equations. To solve these system three classes were created called momentum, energy and manager. Manager class acts as a solver while momentum and energy classes solve momentum and energy equation.

This object oriented approach has provided more flexible solution without affection general implementation of PDE solver. Object oriented programming has provided a rapid, robust and extensible PDE solver.
2 Object Oriented Programming for Scientific Computing and Predator Prey Algorithm

This chapter is divided into two sub sections. The first subsection will be on object oriented programming. This subsection will discuss object oriented programming in general, what features it has, and how those features have helped to improve process of software engineering. It will also discuss object oriented programming in scientific computing and parallel programming. The second subsection will discuss the predator-prey algorithm and its applications for scientific computing.

2.1 Why Object Oriented Programming

The approach to programming has changed dramatically since the invention of the computer. For example, initially programming was done manually toggling in the binary language by use of front panel. But as program size increased assembly language was invented. Assembly language was able to deal with larger program using symbolic representation of machine instruction. As program size continued to grow, high level programming like C, Basic, FORTRAN and other languages were introduced. But once the program exceeds 25000 to 100000 lines of code, it becomes complex and it becomes harder to understand the concept of a program [9]. This problem has lead to structured programming. Object oriented programming provides barriers to break down large program into smaller structure and that enable programmer to write large complex programs fairly easily.

The essential elements of Object Oriented programming are encapsulation, inheritance and polymorphism

2.1.1 Encapsulation

Encapsulation can be explained as a grouping of similar ideas into one unit, which can then be referred as a single name [10]. As similar parts in larger programs are grouped in one unit it reduces the complexity of program. Encapsulation can be classified into low level and high level encapsulation. Encapsulating a constant into a meaningful name, encapsulating an idea into function or subroutine are examples of low level encapsulation. This low level encapsulation is also applied in functional programming but it cannot cope with dependency between code and data. Classes in object oriented programming provide higher level encapsulation. When class is defined in program it also defines data (variables) and methods. This data can be made private to class or public to whole program. Public data can be accessed by any class of the program while private data can only be accessed by the methods of that class and this way it ensures that no improper access of data is taking place. Encapsulation allows changing method details or objecting details without affecting the functionality of the rest of the code.

2.1.2 Inheritance

Inheritance enables one object to acquire the properties of other object. Inheritance supports the idea of hierarchical classification and allows to classes to uses other classes attributes and methods. For example Labrador is a dog, which is part of class mammals, which is under class animal and if Hierarchy is not used, each class is required to declare all attributes and methods explicitly. By using inheritance parent class encapsulate general attributes and methods and subclasses implement specialise attributes and methods for those objects. Hence inheritance allows object oriented programs to include more complex code and reuse of code.
2.1.3 Polymorphism

Polymorphism allows single name for general set of functions or actions. The specific method or action is determined at run time according to the exact nature of program. So polymorphism allows to write a default behaviour of the object, that can be manipulated. This means it becomes easier to extend the implementation of interface if it has more features or attributes.

So as discussed above, encapsulation can provide abstraction of well designed functions and classes which can be reused. Inheritance provides hierarchy to the software, which make debugging, testing and scaling easier and polymorphism provide clean, sensible and readable code.[10] So when these feature of object oriented programming is applied properly, it provides robust and scalable program.

These features of object oriented programming are used by a wide range of applications and different object languages provide features to develop specific software. Mostly object oriented programming is used for System design and industrial software building. For example Simula has been used for VLSI, embedded systems, protocol etc. Java is used most for Web applications and embedded systems. J2EE and .Net provide frameworks to write system software and their modules.

2.2 Object Oriented Programming for Scientific Computing

One of the problem with scientific computing is how to make scientific code easy to maintain and easy to adopt to increased size and complexity [11]. Object oriented programming offer features which allows the programmer to write better scientific codes. Over the last decade, object oriented programming has been very popular among developers and there is lots of work done in object oriented programming for scientific computing as well. Here some of application will be discussed:

Frameworks are becoming more popular in software engineering and scientific computing. Framework is support structure using which other applications or software can be developed. POOMA [6] (Parallel Object Oriented Methods and Applications) and POET (Parallel Object oriented Environment and Toolkit)[5] are example of existing frameworks for parallel computing. Both Frame works are created in C++, with primary goal of code reuse. As parallel numeric algorithms are more complex and very machine dependent, the main motivation for both frameworks is to preserve the working implementation of this code.

It is very useful to create a simulation to represent and to deal with various problem of computational science effectively. Application Visualisation Software (AVS)[7] is a software which provide a platform to create an application to combine 3D graphic system with computation program. AVS is written in object oriented programming C++. The object oriented programming feature provides required extensibility to AVS and hide low level details from high level module interface.

2.3 Predator Prey Simulation

A simulation attempting to predict the relationship in population between a population of predators and population of prey isolated on an island is referred to as the predator-prey problem [12]. The predator-prey algorithm explains the mechanism of how the population of different species depend on each other. It is one of the oldest mathematical algorithms of predation in ecology. This phenomena was first observed by Volterra. He conceived the predator prey idea after observing the fishing fleets in the Adriatic. He observed that when the population of fishes began to increase there was an increase in the number of fishermen as more and more of them were drawn by the success of others. But as soon as the population of fishes declined, the number of fishermen also declined [13]. This was cycle which kept on repeating itself. Due to a number of complex interactions taking place throughout the ecological system
there is a balance maintained between the predator and prey. There are also a number of important stabilising factors that must be taken into account such as [13].

1. Prey defenses: A prey that can adapt quickly and reduce its vulnerability can survive better than those which can be preyed upon and eliminated easily due to the natural selection process. Ex: Bats use sonar to capture moths. Some moths have methods of detecting this sonar and can take immediate defensive action [13].

2. Existence of another prey type: The availability of another prey type can also help balance prey population as the predator feed on more than one type of prey. Thus, when the population of one type of prey reduces then the predator starts feeding on another type which helps the population of the former to recover thereby restoring the balance. However, there maybe cases wherein presence of an alternative prey in large numbers might give rise to high predator density. If the predator doesn’t succeed in reducing the population of prey then such a high predator density might result in the elimination of the other prey type. This is a de-stabilising effect. Ex: Hare Caribou and Lynx relationship in New Foundland [13].

2.3.1 Application of Predator Prey Simulation

Predator Prey algorithm is used in a wide range of applications like Artificial intelligence, economics, stock market, game industry etc. Here some references are given to applications which have used predator prey algorithm.

The intelligence Reconnaissance division of the Air development centre at Rome have devised a method of path recognition and target recognition with the help of predator prey neural network paradigm[16].

Stock markets are primarily driven by the market forces of supply and demand. When the demand increases, it creates a bubble in market values and market valuation increases as well, which eventually dries out the demand from the market and that creates an instance of crash. When there is crash in market supply becomes higher and which causes devaluation of market and demand again starts increasing[6]. This phenomena of stock market is conceptualised by predator prey algorithm

2.4 Predator Prey Model

Nature presents us with many instances where certain species of animal feed on other species depending on their level in the food chain. These animals may in turn be feed upon by other species of animals which are higher up in the food chain. Here first species of animals are known as predator and second species of animals are known as prey. The prey is essential for the existence of the predator. So in scenario wherein prey is completely made extinct by the predator, predator would ultimately become extinct because of starvation[8]. The figure below shows how the population of predator and prey depend on each other.
As shown in figure 1, in nature a cycle is developed where at some time number of prey are more and predators are less. Because of more number of preys the population of predator grows and reduces the population of prey. These cause reduction in predators and increase in prey, and the cycle goes on. As shown in figure it can be noticed that as the population of puma increases population of hare decreases. And as because of less food for prey it cause drop in population of puma. Due to less predation of hare because of less population it cause the hare population to increase and this cycle goes on.

2.5 Why Predator Prey Model

This section discusses why the predator-prey algorithm is being considered to represent object oriented programming features for scientific computing.

Here the predator-prey algorithm is solved by partial differential equation on 2D grid of map. Many physical problems can be solved using partial differential equations which are imposed on 2D grid or 3D grid. For example image processing, weather forecasting etc. Moreover the predator-prey algorithm would be ideal algorithm for object oriented programming as it would be easy to conceptualised the predator-prey model using objects. In the predator-prey model hare, puma and world surface can be seen as different objects.

As the predator-prey model is used in wide range of applications representing many different scientific problems, the predator-prey model is chosen for carrying out features of object oriented programming for scientific computing.
3 Design

In this chapter mathematical formulation of the predator-prey algorithm and different possible designs for predator-prey algorithm will be discussed. In the first subsection mathematical formulation of predator-prey algorithm will be discussed and in the second subsection different issues related to design like scalability, maintenance and reusability will be discussed.

3.1 Mathematical Model

We will consider implementing a sequential version of a two-dimensional predator-prey model with spatial diffusion. The density equations for hare and puma for the predator-prey algorithm are given below:

\[
\frac{\partial H}{\partial t} = rH - aHP + k \left( \frac{\partial^2 H}{\partial x^2} + \frac{\partial^2 H}{\partial y^2} \right) \tag{1}
\]

\[
\frac{\partial P}{\partial t} = bHP - mP + l \left( \frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} \right) \tag{2}
\]

Where,

- \(H\) density of hare
- \(P\) density of puma
- \(r\) intrinsic rate of prey population increase
- \(a\) predation rate coefficient
- \(b\) reproduction rate of predators per one prey eaten
- \(m\) the predator mortality rate
- \(l\) diffusion rate of hare
- \(l\) diffusion rate of puma

In the formulation of these equations there are assumptions:

- all hares are killed and eaten by pumas. They do not die because of other causes.
- population increase rate for hares is constant.
- mortality rate for predator is constant.

The equation 1 and 2 will be solved for a given map of world surface. The world surface will be divided into regular domain of \(N_x\) and \(N_y\) grid points. This grid points will give the state of the world surface, which could be either land or water. The distance between two grid point is assumed as \(l\) and the density of hare and puma will be calculated at each \(\Delta t\) time step.

New values of \(H\) and \(P\) at time \(t + \Delta t\) are given by:

\[
H_{ij}^{\text{new}} = H_{ij}^{\text{old}} + \Delta t \left( r H_{ij}^{\text{old}} - a H_{ij}^{\text{old}} P_{ij}^{\text{old}} + k \left( \frac{H_{i-1,j}^{\text{old}} + H_{i+1,j}^{\text{old}} + H_{i,j-1}^{\text{old}} + H_{i,j+1}^{\text{old}} - 4H_{ij}^{\text{old}}}{l} \right) \right) \tag{3}
\]

\[\text{Mathematical formulation of the predator-prey algorithm has been taken from Dr. David Henty, Dr. Mark Bull and Dr. Allan Simpson from MSc course work for High Performance Computing.}\]
\[ P_{ij}^{\text{new}} = P_{ij}^{\text{old}} + \Delta t \left( b_i P_{ij}^{\text{old}} + m P_{ij}^{\text{old}} + l \left( (P_{i-1,j}^{\text{old}} + P_{i+1,j}^{\text{old}} + P_{ij-1}^{\text{old}} + P_{ij+1}^{\text{old}}) - N_{ij} P_{ij}^{\text{old}} \right) \right) \] (4)

Here \( N_{ij} \) give the sum of neighbouring grid point with the state land.

This is very straightforward algorithm of predator-prey but this algorithm can be made more complex by introducing more species of animal. Introduction of new species will make the algorithm more complex as the population of one species of animal will depend on more number of species of animal. This algorithm can be made more complex by introducing other state of world surface like rocks, grass, valley etc and the population of animals will depend on the surface. For example numbers of hare will be more where the surface of world is grass.

### 3.2 Design

Each design will discuss possible conceptual design and class diagrams and implementation of those designs. During this project predator-prey algorithm is being solved on given maps. During this work maps will be referred to a world. To solve predator-prey algorithm, the world could be looked up as 2-D Grid. The objects will be created according to these 2-D grid maps.

#### 3.2.1 Design 1

For this design concept, the world is considered as one object and it will contain all the information about elements in the world. In the predator-prey algorithm the elements of world would be surface, hares and pumas. Following diagram shows how the object will be created for this design.

![Map viewed as 2-D grid](image)

Figure 2: Conceptual Design 1

As shown in figure 2 only one Object called world will be created which will implement 2-D arrays surface, hare and puma density according to the 2-D grid of world. The following class diagram gives more detail about the world object:
As explained earlier, only one object called world will be created and it will implement all elements and methods required to solve the predator-prey algorithm.

As shown in the class diagram in figure 3, Object world will implement 2-D integer array surface according to 2-D grid of world. Each element in the array will represent whether the surface is land or water. Density of hare and puma on each cell of grid will be stored in 2-D float arrays hare and puma respectively. Both arrays will be initialised random numbers using method initHarePuma. Methods calHare and calPuma will calculate the density of hare and puma for each time step and average density of hare and puma in world will be calculated by method aveHarePuma.

This design would not be ideal for an object oriented solution of predator-prey algorithm. As all attributes and methods are implemented in a single class, it will result in a series of method calls to solve the problem. This mentions that the design is conceptually very similar to functional code. Even scaling of this design wouldn’t be efficient and easy. For example introducing a new species of animal in algorithm would require changing the entire design of class. It would require new attributes and methods for new species of animal. It would also force changes to function initHarePuma to initialise a new species of animal. This design is not robust as all attributes and methods are implemented in one class. It offers access of all attributes to all methods and so no data abstraction is provided.

3.2.2 Design 2

Same as design 1, in this design there is only one class world. But instead of one object, multiple objects of type world will be created. The following figure gives more detail about the conceptual design:
As shown in the figure 4 objects world will contain information about a particular cell of the grid. The following class diagram show design of class:

```
<table>
<thead>
<tr>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>surface: int</td>
</tr>
<tr>
<td>hare: float</td>
</tr>
<tr>
<td>puma: float</td>
</tr>
<tr>
<td>daRead()</td>
</tr>
<tr>
<td>initHarePuma()</td>
</tr>
<tr>
<td>callHare()</td>
</tr>
<tr>
<td>callPuma()</td>
</tr>
</tbody>
</table>
```

Figure 5: Class Diagram of Design 2

As shown in the class diagram in figure 5, the object world will have attribute called surface that will store information about the world surface, which will be initialised by method daRead. The density of hare and puma would be initialised by method initHarePuma. The density of hare and puma for each time step would be calculated by methods callHare and callPuma respectively. During this implementation 2-D instances of object world would be implemented to according to the 2-D grid of map. The average density of both hare and puma would be calculated through the main method. The main method will collect the density of hare and puma from every object and calculates average density.

In this design the degree of object is increased in comparison to design 1. But still it doesn’t the solve problem of data abstraction. The introduction of new species of animal in this design would be more difficult as explained in design 1.

### 3.2.3 Design 3

During this concept of design, all elements surface, hare and puma will be considered as an object. The following conceptual figure explain the design in detail:
As shown in the figure 6 above three different class will be defined for world, hare and puma. The object world will contain information about the surface and will implement the method to change its attribute. The hare and puma objects will have the density for hare and puma respectively for each cell of the grid, and methods to initialise and modify those attributes. The following class diagram explain the objects in more detail.
As shown in the class diagram in figure 7, the object world will have a 2-D array called surface, which will contain information about whether the surface is land or water. This information will be assigned to a 2-D array through the dataRead method, which will read the data file which contains the map. While objects hare and puma will have a 2-D array called hare and puma respectively, these will store density of hare and puma for each cell of grid. The density of hare and puma for each will be initialised randomly through a function called initHare and initPuma, while methods calHare and calPuma will calculate the density of hare and puma for the next time step respectively. For each time step average value of hare and puma will be calculated through methods avHare and avPuma respectively. 

In this design it could be noticed that both hare and puma have similar attributes, which store the density of hare and puma respectively. Even methods for the initialisation of density and calculation of the average value of both species are the same but reusability of code is not being exploited in this design. It would be easy to scale this implementation. For example introducing new species of animal will be implemented through new class; this will not affect the implementation of other classes. But it would result in a more redundant code because of the similar functions for initialisation and average density of the hare and puma array.

3.2.4 Design 4

During this possible design, each element in the world would be seen as an object. So for each cell in the grid three different objects for surface, hare and puma will be created. The following figure explain conceptual design in detail:

![Conceptual Design 4](image)

As shown in figure 8, for each surface, hare and puma different objects will be created, for each cell in the grid. The following design explain the detail design of these classes:
As shown in the class diagram in figure 9, class world will have an attribute called surface which will store information about the surface, which will be initialised by the method called datRead. Classes hare and puma will store the density of hare and puma in a particular cell of the grid respectively. The density of hare and puma will be assigned through the method calls initHare and initPuma respectively. For each time step, the density of hare and puma will be calculated by methods callHare and callPuma respectively. The average density of hare and puma will be calculated from the main method. The main method will collect the density of hare and puma from every object and calculate the average density of hare and puma in the world.

As explained earlier in the preceding design, this design doesn’t exploit the reusable code. For example both hare and puma classes have similar function to initialise random numbers for their attributes. It would be easy to introduce new species of animal in this design but it would not be ideal as it would result in more redundant code.

3.2.5 Design 5

The previous design was not efficient in terms of the reusability of common attributes and methods for hare and puma. During this design it will be seen how object oriented programming exploits reusability of code through inheritance. In this design, the world would be seen as two objects world and animal. The following figure explains the conceptual design:
As shown in figure 10, two classes, world and animal will be defined. Only one object world will store information about world surface and methods to initialise that. During this design, a super class called Animal will be defined which will implement all the basic attributes of animal and subclass will be defined to implement each animal’s special attributes. The following class diagram explain the design in more detail:

Figure 10: Conceptual Design 5

Figure 11: Class Diagram of Design 5
As shown in figure 11, the abstract class called Animal will be created to implement common attributes and method for all species of animals. As shown in the class diagram in figure 11, the abstract class animal will implement attributes density and aveDensity, which will represent the density of animal on each cell of the grid and the average density of animal in world respectively. This attributes will be inherited in objects hares and puma. Objects hare and puma then will implement methods to calculate the density of hares and puma for each time step.

This design would not only be efficient in reusing the code but also in expanding the algorithm. For example introduction of new animals would be very easy, as it will only require the creation of a subclass of abstract class animal and the implementation of method to calculate the density for each time step without changing implementation of other species of animals. But this design would result in additional overhead due to objects creation and inter object communications.

3.2.6 Design 6

As explained in the previous design, in this design surface and animal would be regarded as a objects. But instead of one object for the whole grid, for each cell of the grid one object of each surface and animal would be created. The following figure shows the conceptual design:

![Map viewed as 2-D grid.](image)

**Figure 12: Conceptual Design 6**

As shown in the figure 12 for each of cell of the grid an object world would be created and two abstract animal objects would be created for hare and puma. The following class diagram explain these classes in detail:
As shown in the figure 13, the abstract class animal would implement the common attribute density and method initDensity to initialise random number to that. The subclasses hare and puma would implement the method calHare and calPuma respectively which would calculate the density of hare and puma for different time step. The average density of both hare and puma would be calculated through the main method. The main method will collect the density of hare and puma from every object and calculate density.

This design would be ideal among all the designs discussed before and is easy to scale. For example, the introduction of a new species of animal would be implemented by creating a subclass of abstract class animal and will not affect the other classes. But creating an object for each animal and surface for each cell of the grid will increase the degree of object which will cause more communication between different objects and class. This communication may lead to poor performance of complex code.

3.2.7 Extension to all the designs

In the predator prey algorithm, density for hare and puma for each time step is calculated through discretised Partial differential equation 3 and 4:

Here the first equation calculates the density of hare for $t + \Delta t$ time step. During this it calculates the sum of all its neighbouring cells hares’ density and pumas’ density. Following equation exploit similar calculation in equations 3 and 4.

\[
\left( \left( H_{i-1,j}^{\text{old}} + H_{i+1,j}^{\text{old}} + H_{ij-1}^{\text{old}} + H_{ij+1}^{\text{old}} \right) - N_{ij}H_{ij}^{\text{old}} \right)
\]

\[
\left( \left( P_{i-1,j}^{\text{old}} + P_{i+1,j}^{\text{old}} + P_{ij-1}^{\text{old}} + P_{ij+1}^{\text{old}} \right) - N_{ij}P_{ij}^{\text{old}} \right)
\]
These sub equations can be solved a by single method, this will not only provide reusability of code but it will also provide scalability to code. These equation are discretised using a five point stencil. But it is possible to overload the method to implement a nine point stencil on same equation without changing implementation of other part of equation.
4 Implementation

During first subsection of this chapter, it will discuss functional implementations of predator-prey algorithm in C and Java. And second half will discuss the java OO codes.

4.1 Functional Implementation

4.1.1 Fictional C Implementation:

Implementation of predator-prey algorithm in C is very straight forward, and it is based on successive subroutine calls. The following diagram shows the flow chart of C implementation of predator-prey algorithm:

![Flowchart for C Functional Implementation]

As shown in the flow chart above, first it read islands.dat data file into array called island using function called datread. Following pseudo code shows how data is read into array
void datread(char *filename, void *vx, int xz, int sy) {
    FILE *fp;
    int *vz = (int *) vx;
    if (NULL == (fp = fopen(filename, "r"))){
        print error message and exit from program
    }
    fscanf(fp, "%d %d", &x, &y);
    if (x != xz || y != sy) {
        print error message and exit from program
    }
    for (j=0; j<sy; j++){
        for (i=0; i<xz; i++){
            fscanf(fp, "%d", &t);
            vz[(y-j-1)*xz+y] = t;
        }
    }
    fclose(fp);
}

As shown in code, First it creates a file pointer using structure File. File is a structure which provides
collection to operating system to open a file. Once file is opened it reads the dimension of file and
checks with array, that both are identical to each other. After that by using nested for loop it reads
integer values in array.

Once surface values are read into array land it assigns initial random density to hare and puma array H
and P respectively using function initPP.

After initial density are assigned to hare and puma, it calculates density of hare and puma for next time
stamp using method calPP. Following code give detail implementation of calPP method.

for(i=1; i<N-1; i++){
    for(j=1; j<N-1; j++){
        if(island[i][j] == 1){
            neighbour = island[i-1][j]+island[i+1][j]+island[i][j-1]+island[i][j+1];
            Hnew[i][j] = H[i][j] + deltat * ( r*N[i][j] - a*N[i][j]*P[i][j] ) + k*(O[i-1][j]+O[i+1][j]+O[i][j-1]+O[i][j+1]-O[i][j]);
            Pnew[i][j] = P[i][j] + deltat * ( b*N[i][j]*P[i][j] - n*P[i][j] + 1*(N[i-1][j]+N[i+1][j]+N[i][j-1]+N[i][j+1]-N[i][j]) -
            neighbour*P[i][j]);
        }
    }
}

As shown in code it calculates density of hare and puma for next timestamp where surface is land. After
each calculation new density of hare and puma, it swaps H,P arrays to Hnew and Pnew arrays value.
After each 10 calculation it calculate average values of hare and puma using average function.

4.1.2 Functional Java Implementation

This is naïve implementation of java code to perform as a functional code. All methods and attributes
are implemented in one class and successive method calls solve predator-prey algorithm. The following
diagram shows flowchart of java code.
During this implementation all the methods and attributes are declared in the main method and so the flowchart of this implementation very similar to C. the detail implementation of this functions are explained in implementation 1 later in this chapter.

4.2 Object Oriented Implementation

Main focus during this section would be on implementation issues of the different designs discussed in the previous section. It will discuss different object oriented implementation issues like data abstraction, inheritance, abstract class etc and how it provides robust, easy to maintain and reusable code. During the discussion first it will discuss changes made to design during implementation and detail class diagram.
4.2.1 Implementation 1

In this implementation only one object of class world is created and which implements all the methods required to solve predator prey algorithm. The figure below shows the class diagram of world after implementation:

![Class Diagram](image)

**Figure 16: Class Diagram of Implementation 1**

The Figure 16 shows detail class diagram after implementation of Design 1. Java provides a rich collection of classes and utilities, some of these classes which are being used for implementation of all designs will be discussed briefly here.
As shown in the communication diagram in figure 17, the first method called is datRead, which transfers the data file name as an argument.

```java
    pp.datRead("islands.dat");
```

Java provides classes for IO and the following code explain how the dat file can be read using Java predefine classes.

```java
    File file = new File(s);
    FileInputStream file_input = new FileInputStream(file);
    BufferedReader data_in = new BufferedReader(new InputStreamReader(file_input));
```

After the first statement executes in the above code, it creates an abstract path of islands.dat file. This abstract path of file is transferred to FileInputStream which actually creates an input stream connection to islands.dat file. Class BufferedReader create the character stream which reads character stream from File_input stream, which is created by FileInputStream class.
temp = data_is.readLine();
StringTokenizer st = new StringTokenizer(temp);
while(!temp.equals(null)){
    st = new StringTokenizer(temp);
    j=0;
    while(st.hasMoreTokens()){
        surface[i][j] = Integer.parseInt(st.nextToken());
        j++;
    }
    i++;
}

Once a BufferedReader object is created, it reads lines from the input file using method the readLine() into a string temp. String temp contains a whole line of data, which is tokenized into token using object of class StringTokenizer. These tokens are converted into integer values using the Integer.parseInt() method. As shown in the code above by using two while loops the whole islands.dat file is read into a 2D integer array surface.

Random gn = new Random();
For(int i=0; i<N; i++){
    For(int j=0; j<N; j++){
        if(surface[i][j] == land){
            hare[i][j] = hareev[i][j] = gn.nextDouble();
            puma[i][j] = pumaev[i][j] = gn.nextDouble();
            landpoint++;
        }
    }
}

After the data is read in array surface it called initHarePuma(), which assigns random numbers to the hare and puma arrays. First it create an object named gn of the predefine class Random. Then it generates a series of random numbers between 0.0 to 5.0 and assign to hare and puma according to the surface, where it is land. Whilst assigning random number, it also calculates landpoints in world which later used to calculate average density of hare and puma.

The density of hare and puma is calculated by calling method callHare and calPuma respectively. The equations given below calculate the density of a particular cell of hare and puma respectively for next timestamp t + deltatt.

void callHare(){
    int neighbour;
    for(int i=1; i<N-1; i++){
        for(int j=1; j<N-1; j++){
            if(surface[i][j] == land){
                neighbour = surface[i-1][j] + surface[i+1][j] + surface[i][j-1] + surface[i][j+1] + surface[i-1][j-1] + surface[i-1][j+1] + surface[i+1][j-1] + surface[i+1][j+1];
                hare[i][j] = hare[i][j] + deltatt*(r-hare[i][j] - a*hare[i][j]*puma[i][j] + k*(hare[i-1][j] + hare[i+1][j] + hare[i][j-1] + hare[i][j+1] - neighbour*hare[i][j]));
            }
        }
    }
}

void calPuma(){
    int neighbour;
    for(int i=1; i<N-1; i++){
        for(int j=1; j<N-1; j++){
            if(surface[i][j] == land){
                neighbour = surface[i-1][j] + surface[i+1][j] + surface[i][j-1] + surface[i][j+1] + surface[i-1][j-1] + surface[i-1][j+1] + surface[i+1][j-1] + surface[i+1][j+1];
                puma[i][j] = puma[i][j] + deltatt*(r-puma[i][j] - a*puma[i][j]*hare[i][j] + k*(puma[i-1][j] + puma[i+1][j] + puma[i][j-1] + puma[i][j+1] - neighbour*puma[i][j]));
            }
        }
    }
}
The codes above give the implementation of both methods calHare and calPuma. Here it traverses through the whole 2D grid and calculates the density of hare and puma for the next timestamp for the world surface is land. Here it can be noticed that this function doesn’t calculate the density for outer cells of 2D grid, because as a boundary condition it is assumed that outer cells are always water. The new density of hare and puma are stored into harenew and pumanew arrays respectively. Once for each cell of the grid new density for next time stamp is calculated and then swapped by method swap() given below.

```java
void swap(){
  for(int i=1; i<N-1; i++){
    for(int j=1; j<N-1; j++){
      hare[i][j] = harenew[i][j];
      puma[i][j] = pumanew[i][j];
    }
  }
}
```

After each n calculations of hare and puma the average density of hare and puma are calculated by method aveHarePuma() given below.

```java
void aveHarePuma(){
  double predator=0, prey=0;
  for(int i=1;i<N-1;i++){
    for(int j=1; j<N-1; j++){
      predator += puma[i][j];
      prey += hare[i][j];
    }
  }
  System.out.println("Hare: "+prey/landpoint + " Puma: "+ predator/landpoint);
}
```

In this implementation, as shown in the communication diagram in figure 17, the whole algorithm is solved by a series of functions call which results in a similar code to functional code. The methods initHarePuma and aveHarePuma are too specific for implementation of design and do not provide any flexibility and reusability to code. For example, the calculation of average density of hare and puma is inscribed into a single function. In this case introduction of a new species of animal would force a change in aveHarePuma method and same is true for initHare and Puma method.

```java
harenew[i][j] = hare[i][j] + deltat * ((r+hare[i][j] - a*hare[i][j]*puma[i][j] + k*(hare[i-1][j] + hare[i+1][j] + hare[i][j-1] + hare[i][j+1] - neighbour*hare[i][j]));
Pumanew[i][j] = Puma[i][j] + deltat * ((d+Puma[i][j]*puma[i][j] - b*Puma[i][j]*puma[i][j] - n*Puma[i][j] + (puma[i-1][j] + puma[i+1][j] + puma[i][j-1] + puma[i][j+1] - neighbour*puma[i][j]));
```

The above equations calculate the density of hare and puma for timestamp t+delat time stamp. In the hare equation, it can be noted that it can directly access its four neighbouring cells hare density and puma density array values. It is therefore possible to change the density of the neighbouring haress or the related puma density. This could result in inconsistent code.

4.2.2 Implementation 2

During the implementation of Design 2, a 2D array of object world is created. Each object will implement attributes for each cell of the grid. The figure below shows the class diagram of class world after implementation:
Figure 18: Class Diagram of Implementation 2

Though this design is being implemented by multiple objects of same class, the algorithm is still being solved by a series of method calls. But as the implementation is being changed from one object to multiple objects there are quite a few changes in implementation of methods and method calls. In class world surface, hare and puma attributes are declared as private variables which mean that those attributes can not be accessed from outside that class and only member functions of that object can access those variables. This provides data abstraction to these variables and avoids possible misuse of these variables. The following communication diagram shows how predator-prey algorithm is being solved:
Figure 19: communication Diagram of Implementation 2

It is important to look at how multiple objects of world are initialised in the main method. The following code shows how objects are created:

```java
World pp[0][0] = new World[0][0];
for(int i=0; i<N; i++)
    for(int j=0; j<M; j++){
        pp[i][j] = new World();
    }
```

Here the first line of code creates pp[M][N] references to the class world and then by the use of two nested loops pp[M][N] reference of object world are assigned to actual objects.

As shown in the communication diagram in figure 19, only one object pp[0][0] reads the whole file into array surfaceTemp. Here the surfaceTemp array is declared as a static integer, which allow access to all object of class world with out reference to any object and the array is shared by all objects of class world.
void assignSurface(int i, int j)
{
    surface = surfaceTemp[i][j];
}

Once data is read into the surfaceTemp array, the surface value is assigned to each object by the method assignSurface(). After the surface is assigned to all objects, the initial density of hare and puma are assigned by method call initHarePuma.

for(int i=1; i<N-1; i++)
{
    for(int j=1; j<M-1; j++)
    {
        if(pp[i][j].surface() == land)
        {
            neighbour = pp[i-1][j].surface() == pp[i+1][j].surface() || pp[i][j-1].surface() == pp[i][j+1].surface();
            pp[i][j].calHare(neighbour, pp[i-1][j].puma(), pp[i+1][j].puma(), pp[i][j-1].puma(), pp[i][j+1].puma());
        }
    }
}

As shown in the code above, calHare and calPuma are called for each object using a nested for loop. Here in the calHare method, it passes the number of cells with surface land and density of four neighbouring cells hare. Method calHare implements equation to calculate the density of hare for the next time stamp. Similarly calPuma calculates the density of puma for the next timestamp for each cell.

double predator=0, prey=0;
for(int i=1; i<N-1; i++)
{
    for(int j=1; j<M-1; j++)
    {
        if(pp[i][j].surface() == land){
            predator = pp[i][j].puma();
            prey = pp[i][j].hare();
        }
    }
}

In this design the main method calculates average density of both hare and puma instead of class world. It accumulates density of hare and puma from each cell with surface land into temporary variables prey and predator respectively.

This design has a higher numbers of object than Design 1 but still it solves the predator-prey algorithm with series of method calls and still its not an efficient object oriented implementation of predator prey algorithm. It provide data abstraction between different objects of World but still density of hare and puma of each cell are accessible which is not desirable. Moreover as explained in Design 1 implementation, initHarePuma isn’t flexible and reusable.

4.2.3 Implementation 3

The previous two designs consisted a single object and multiple objects respectively of a single class. In this design world, hare and puma all will be looked up as objects. The figure below shows the class diagram after implementation of this design.
During this implementation, one object of each world, hare and puma is created. Here object world will read data form islands.dat file and store into array surface array while hare and puma will be responsible for initialisation and calculation of hare's and puma's density. Hare and Puma classes will also calculate the average density of hare and puma. The following communication diagram explains the implementation in detail:
As shown in the communication diagram in figure 21, the object `pp` of class `world` reads data file `islands.dat` into the array surface. Methods `datRead()` is implemented in the same way as explained in the design 1 implementation. But in this implementation, surface `world` is declared as a static and public integer array. As the array surface is declared as static and public, object of hare and puma classes can access the value of surface with use of class name as shown in code below.

```c
neighbour = world.surface[i-1][j] + world.surface[i+1][j] + world.surface[i][j-1] + world.surface[i][j+1];
```

This program provides a better object oriented implementation than the previous two implementation discussed before. It provide data abstraction between data of world, hare and puma objects. For example both the hare and puma require each other's density to calculate the density for next time stamp, so both `calHare` and `calPuma` methods are passing the arrays of the puma and hare density respectively. This implementation is easier to expand and be maintained. For example, the introduction of new species can be implemented without changing implementation of hare and puma classes. Though this implementation provides easy to expand and maintained code, it fails to provide abstraction between neighbours of hare and puma. Consider the code given below:

```c
newDensity[i][j] = density[i][j] + deltat * (r*density[i][j] - a*density[i][j]*puma[i][j]);
```
Here the first line of code calculates the density of hare for the next time stamp and the second line of code calculates for puma. Here it can also be noted that both hare and puma can access the density of hare and puma directly and mised code program could result in inconsistent code.

### 4.2.4 Implementation 4

During this implementation for each cell of the 2D grid, one object of each world, hare and puma will be created and this object will implement all the attributes and methods required. The following figure shows class diagram after implementation.

![Class Diagram of Implementation 4](image)

During this implementation, 2D dimensional arrays of objects world, hare and puma are created. As shown in the communication diagram in figure 23, a series of nested for loops call different function to solve predator-prey algorithm.

30
As shown in the code above, a nested loop surface is assigned to each objects of world class, using the static surfaceTemp array. And wherever surface is land, object of hare and puma are assigned to initial random density, using the initDensity method of hare and puma class respectively.
As shown in the code above for each object of hare and puma calHare and calPuma methods are called. Here it can be noted that both hare and puma density are accessed through the method density of hare and puma respectively. This implementation provide data abstraction between different objects of hare and puma, as well as providing abstraction between objects of different class.

This implementation provides a higher numbers of objects. This implementation provides scalable, robust and easy to maintain code. But still in this implementation there exits redundancy of code. For example initDensity, density methods are identical in both classes and same code will need to be rewrite to introduce of new species of animal.

4.2.5 Implementation 5

All previous implementation had problem with reusability of code, However this implementation, by means of abstract class and inheritance will show how Object Oriented programming exploits reusability of code. During this implementation world and animals are looked as a main object. The following figure shows class diagram after implementation of design 5:

![Class Diagram](image)

Figure 24: Class Diagram of Implementation 5

As shown in the class diagram in figure 24, World and Animal class are created and from Animal class two child classes Hare and Puma will be created. Classes hare and puma inherited all the attributes and methods of Animal class. The communication diagram for implementation 5 will be the same as in implementation 3 shown in figure 21.

```
abstract class Animal{
    double density[][] = new double[8][8];
    double seaDensity[][] = new double[8][8];
    double areDensity;
    int landpoint;
    ... ...
    void initDensity(){
        ... ...
    }
}
```
The above given code explains the structure of the abstract class Animal. Here in the predator prey algorithm both the hare and puma have common attributes like density and average density and methods to modify those attributes like method to initialise density etc. The abstract class is a method which provides abstraction for all attributes and methods without the complete implementation of classes but it allows creation of a child classes. In this implementation hare and puma are created as child classes of the Animal classes and they implement methods to calculate density of hare and puma.

4.2.6 Implementation 6

In previous implementation one object of world and one object of hare and puma were created. During this implementation multiple objects of world, hare and puma will be created. The following figure shows the class diagram of this implementation:

![Class Diagram of Implementation 6](image)

As shown the in class diagram in this implementation, multiple objects of world, hare and puma are being created. The communication diagram for implementation 6 will be same as implementation 4 as shown in figure 23.
This implementation provides robust, easier to expand and easier to maintain code. But as multiple objects of world, hare and puma requires lots of communication between them which might result in slower code. This implementation provides abstraction of data between different objects of different class as well as between different objects of same class. As all the common attributes of all animals are implemented by the abstract class animal, it would be easy to introduce new species of animal in this implementation.

4.2.7 Implementation 7

During all the implementation above, either one object of all world, hare and puma or multiple objects of all world, hare and puma classes are being used. But this implementation, it attempts to implement a hybrid implementation of design 5 and 6 explained in chapter 3. During this implementation only one object of class world and multiple objects of classes hare and puma will be created. The following figure shows the class diagram of this implementation.

![Class Diagram of Implementation 7](image)

Figure 26: Class Diagram of Implementation 7

As shown in the class diagram in figure 26, only one object of class world will implement the surface detail of the 2D world grid, while for each cell of the grid different objects of Hare and Puma will be created. The Communication diagram below shows how the predator prey problem is being solved:
During all implementation of predator-prey algorithm, object or objects of class world are used for condition check of surface. Once the value of surface is assigned it is not changed or modified. So creating multiple objects for class world result in overhead of communication and may hit the performance of code. So during this implementation of the predator-prey only one object of class world is being created and multiple objects of classes hare and puma are being created.

This implementation also results in a easy to maintain and robust code with less amount of communication in comparision to implementation 6 discussed above.
5 Performance

This chapter will focus on discussing performance comparison and issue of different implementations of predator-prey algorithm. In first subsection performance of C and Java functional code will be compared and in second subsection performance of different implementation of Java Object Oriented code will be discussed.

All the programs are run on Sunfire 15k machine, known as lemond in university, which has 52 Ultra-SPARC III Cu 1.2 GHz processors. From this 52 processor, 4 processors are used for front-end tasks like editing, compiling etc, while 48 processors are used as back-end for processes. Following compilers are used for Java and C;

- java version "1.4.2_05" . Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_05-b04).
- GCC: gcc version 3.1. Thread model: posix

5.1 Performance of Functional Implementations

To check the correctness of all implementation, initial density of hare and puma were assigned the same number for all cell of the 2D grid. Then average density of hare and puma is checked for different time steps. The average density for the different time steps for different implementation match with one another. That verify the correctness of the different implementation of predator-prey algorithm.

Performance of functional C and naive functional Java code are given in table below. It provides calculation time of density of hare and puma and total time required to solve predator prey algorithm.

C Functional Implementation:

<table>
<thead>
<tr>
<th>Calculation Time</th>
<th>79.087</th>
<th>78.782</th>
<th>78.919</th>
<th>78.560</th>
<th>78.642</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>81.527</td>
<td>81.214</td>
<td>81.330</td>
<td>80.907</td>
<td>81.070</td>
</tr>
</tbody>
</table>

Table 1: Calculation and Total Time for C Functional Implementation

Java Naive Functional Implementation:

<table>
<thead>
<tr>
<th>Calculation Time</th>
<th>120.285</th>
<th>121.070</th>
<th>120.523</th>
<th>120.151</th>
<th>120.134</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>120.537</td>
<td>125.342</td>
<td>124.783</td>
<td>124.416</td>
<td>124.393</td>
</tr>
</tbody>
</table>

Table 2: Calculation and Total Time for Java Naive Functional Implementation

Following table shows average time for calculation of density hare and puma and total time elapsed in program.

<table>
<thead>
<tr>
<th></th>
<th>Calculation Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Functional Implementation</td>
<td>78.798</td>
<td>81.229</td>
</tr>
<tr>
<td>Java Naive Functional Implementation</td>
<td>120.432</td>
<td>124.696</td>
</tr>
</tbody>
</table>

Table 3: Average Calculation and Total Time of C and Java Functional Codes.
And following graph plot values shown in above table:

![Performance Chart](image)

Figure 28: Performance Chart for Functional Code

Here in the performance chart, it can be noticed that calculation time and total time required for Java implementation is higher than C implementation. The reasons why Java is code is slower than C codes is C codes are directly compiled into machine code which is then run by operating system directly, while java is compiled into binary code with Java Virtual Machine (JVM), then interpreters or compilers convert that byte code into machine code and than it is run by the operating system. Conversation from Byte code to machine code consumes time and which make java code slower to run than C code.

5.2 Performance of Object Oriented Implementations

During this subsection, performance of object oriented Java code will be discussed. Following tables shows calculation time of different implementation of java code discussed in chapter 7. It shows calculation time of density for hare and puma and total time elapsed in program. It also show object creation time for implementation 2, 4, 6 and 7 where multiple objects are created for solving program.

Implementation 1:

<table>
<thead>
<tr>
<th>Calculation Time</th>
<th>150.502</th>
<th>150.300</th>
<th>150.437</th>
<th>150.483</th>
<th>150.480</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>158.134</td>
<td>157.965</td>
<td>157.975</td>
<td>158.079</td>
<td>158.088</td>
</tr>
</tbody>
</table>

Table 4: Calculation and Total Time for Implementation 1

Implementation 2:

<table>
<thead>
<tr>
<th>Calculation Time</th>
<th>345.812</th>
<th>336.146</th>
<th>336.086</th>
<th>337.337</th>
<th>338.157</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>356.638</td>
<td>356.618</td>
<td>356.486</td>
<td>357.847</td>
<td>358.7</td>
</tr>
</tbody>
</table>

37
Table 5: Calculation and Total Time for Implementation 2

<table>
<thead>
<tr>
<th></th>
<th>Calculation Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation 3</td>
<td>149.484</td>
<td>149.869</td>
</tr>
<tr>
<td></td>
<td>149.887</td>
<td>149.440</td>
</tr>
<tr>
<td></td>
<td>149.565</td>
<td>156.544</td>
</tr>
<tr>
<td></td>
<td>156.944</td>
<td>156.97</td>
</tr>
<tr>
<td></td>
<td>156.518</td>
<td>156.561</td>
</tr>
</tbody>
</table>

Table 6: Calculation and Total Time for Implementation 3

<table>
<thead>
<tr>
<th></th>
<th>Calculation time</th>
<th>Total time</th>
<th>Initialisation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation 4</td>
<td>285.084</td>
<td>334.88</td>
<td>6.814</td>
</tr>
<tr>
<td></td>
<td>285.317</td>
<td>334.967</td>
<td>6.807</td>
</tr>
<tr>
<td></td>
<td>283.876</td>
<td>333.475</td>
<td>6.795</td>
</tr>
<tr>
<td></td>
<td>283.969</td>
<td>333.573</td>
<td>6.790</td>
</tr>
<tr>
<td></td>
<td>286.394</td>
<td>336.866</td>
<td>6.973</td>
</tr>
</tbody>
</table>

Table 7: Calculation, Total and Initialisation time for Implementation 4

<table>
<thead>
<tr>
<th></th>
<th>Calculation time</th>
<th>Total time</th>
<th>Initialisation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation 5</td>
<td>150.947</td>
<td>155.852</td>
<td>7.251</td>
</tr>
<tr>
<td></td>
<td>149.584</td>
<td>155.334</td>
<td>7.227</td>
</tr>
<tr>
<td></td>
<td>149.040</td>
<td>154.789</td>
<td>7.29</td>
</tr>
</tbody>
</table>

Table 8: Calculation and Total Time for Implementation 5

<table>
<thead>
<tr>
<th></th>
<th>Calculation time</th>
<th>Total time</th>
<th>Initialisation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation 6</td>
<td>281.859</td>
<td>326.055</td>
<td>7.28</td>
</tr>
<tr>
<td></td>
<td>282.741</td>
<td>327.036</td>
<td>7.339</td>
</tr>
<tr>
<td></td>
<td>280.663</td>
<td>324.857</td>
<td>7.227</td>
</tr>
<tr>
<td></td>
<td>279.072</td>
<td>323.164</td>
<td>7.29</td>
</tr>
<tr>
<td></td>
<td>278.778</td>
<td>322.836</td>
<td>7.297</td>
</tr>
</tbody>
</table>

Table 9: Calculation, Total and Initialisation Time for Implementation 6

<table>
<thead>
<tr>
<th></th>
<th>Calculation time</th>
<th>Total time</th>
<th>Initialisation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation 7</td>
<td>133.890</td>
<td>144.735</td>
<td>4.283</td>
</tr>
<tr>
<td></td>
<td>133.777</td>
<td>144.665</td>
<td>4.329</td>
</tr>
<tr>
<td></td>
<td>134.374</td>
<td>145.205</td>
<td>4.282</td>
</tr>
<tr>
<td></td>
<td>134.173</td>
<td>144.979</td>
<td>4.276</td>
</tr>
<tr>
<td></td>
<td>134.006</td>
<td>144.838</td>
<td>4.288</td>
</tr>
</tbody>
</table>

Table 10: Calculation, Total and Initialisation Time for Implementation 7

Following table shows average time for calculation of density hare and puma and total time elapsed in program.

Average Time for all Implementation:

38
<table>
<thead>
<tr>
<th>Implementation</th>
<th>Calculation Time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>150.456</td>
<td>158.0482</td>
</tr>
<tr>
<td>2</td>
<td>338.7064</td>
<td>359.2578</td>
</tr>
<tr>
<td>3</td>
<td>149.649</td>
<td>156.7214</td>
</tr>
<tr>
<td>4</td>
<td>284.928</td>
<td>334.7522</td>
</tr>
<tr>
<td>5</td>
<td>150.884</td>
<td>156.737</td>
</tr>
<tr>
<td>6</td>
<td>280.622</td>
<td>324.849</td>
</tr>
<tr>
<td>7</td>
<td>134.044</td>
<td>144.8844</td>
</tr>
</tbody>
</table>

Table 11: Average calculation and Total time of all OO Implementation

And following graph plot values shown in above table:

![Performance Chart](image)

Figure 29: Performance Chart of All OO Implementation

Here performance of implementation 1 is very similar to performance of java functional code discussed in previous section. Because in implementation 1, an object of class world is created, this implements all methods for predator prey algorithm. This implementation is very similar to java functional code and it can be expected to have similar performance for both programs. In design 3 and design 5 one object for each world, hare and puma are created. In these implementations, hare and puma objects are required to check world surface for each cell and if surface is land, it calculates density of hare and puma for next timestamp. Calculation of density of hare and puma requires corresponding puma and hare density respectively. And because of these communications between objects, performance of these implementations can be expected to be poorer than implementation 1. But to avoid communication between hare and puma objects density array of hare and puma are transferred to calHare and calPuma function as shown in below:

```java
calHare(P.density); P.calPuma(P.density);
```
By transferring density array as an argument in calHare and calPuma function it avoids communication between both objects and also avoids overhead of communication because of which performance of implementation 1, implementation 3 and implementation 5 are similar.

In implementation 4 and implementation 6 multiple objects of class world, hare and puma are being created. And for calculation density of hare or puma it requires to communicate with its four neighbouring objects of same class and corresponding object of puma class and corresponding 4 neighbouring object of world class to calculate no of neighbour. So it require higher amount of time for calculation of hare and puma.

But time required for calculation of hare is higher than implementation 4 and implementation 6 even though it requires communication with four neighbouring objects of same class. The reason is explained in detail by following figure:

![Figure 30: Memory Cache Map](image)

In Java, the 2D arrays are the arrays of array; that is each row is contiguous in memory and so columns are contiguous to in memory. So the storage of arrays are row major, like C. As shown in figure 30, grey cell represent objects of class world in cache of program, while white cells represent objects of class world in main memory. To calculate density of hare and puma for last row in cache it requires lots of main memory I/O operation which can possibly make program to run slower. So to improve the performance loop tiling is applied to this implementation.

Loop tiling increases the temporal locality by reordering traversal of iteration into smaller blocks as shown in figure below:
As shown in the figure 31, tiling reuses the data in tile and improves the temporal locality to improve the performance of code. Following code shows general structure of how two one can apply tiling on 2D dimension array.

```c
for (i=0; i<n; i++)
    for (j=0; j<m; j++)
        for (i1=0; i1<n1; i1++)
            for (j1=0; j1<m1; j1++)
                sum[i1][j1] = a[i1][j1] + b[i][j][j1];
```

Here as shown in above code, array is tiled into 2x2 array, and sum, a and b are referenced by ii and jj variables.

Following table show performance of implementation 2, implementation 4 and implementation 6 after tiling:

**Implementation 2 with tiling:**

<table>
<thead>
<tr>
<th>Calculation time</th>
<th>189.563</th>
<th>189.465</th>
<th>190.485</th>
<th>189.528</th>
<th>198.048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time</td>
<td>200.716</td>
<td>209.536</td>
<td>210.667</td>
<td>209.588</td>
<td>218.808</td>
</tr>
</tbody>
</table>

Table 12: Calculation and Total Time for Implementation 2 with tiling

**Implementation 4 with tiling:**

<table>
<thead>
<tr>
<th>Calculation time</th>
<th>288.501</th>
<th>289.340</th>
<th>290.044</th>
<th>291.864</th>
<th>289.693</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time</td>
<td>338.033</td>
<td>339.083</td>
<td>339.044</td>
<td>341.653</td>
<td>339.422</td>
</tr>
</tbody>
</table>

Table 13: Calculation, Total and Initialisation Time for Implementation 4 with tiling

**Implementation 6 with tiling:**

<table>
<thead>
<tr>
<th>Calculation time</th>
<th>284.803</th>
<th>285.234</th>
<th>284.877</th>
<th>285.076</th>
<th>284.850</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time</td>
<td>328.910</td>
<td>329.343</td>
<td>329.003</td>
<td>329.207</td>
<td>329.190</td>
</tr>
<tr>
<td>Initialisation time</td>
<td>7.290</td>
<td>7.277</td>
<td>7.314</td>
<td>7.301</td>
<td>7.281</td>
</tr>
</tbody>
</table>

Table 14: Calculation, Total and Initialisation Time for Implementation 6 with tiling

Following table show average time of calculation time and total time required for different implementation.

**Average Time for all Implementation:**

41
<table>
<thead>
<tr>
<th>Implementation</th>
<th>calculation time</th>
<th>calculation time with tiling</th>
<th>Total Time</th>
<th>Total Time with tiling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation 1</td>
<td>150.502</td>
<td></td>
<td>158.134</td>
<td></td>
</tr>
<tr>
<td>Implementation 2</td>
<td>338.706</td>
<td>191.4178</td>
<td>359.2578</td>
<td>211.663</td>
</tr>
<tr>
<td>Implementation 3</td>
<td>149.649</td>
<td></td>
<td>156.7214</td>
<td></td>
</tr>
<tr>
<td>Implementation 4</td>
<td>284.928</td>
<td>289.8884</td>
<td>334.7522</td>
<td>339.447</td>
</tr>
<tr>
<td>Implementation 5</td>
<td>150.884</td>
<td></td>
<td>156.737</td>
<td></td>
</tr>
<tr>
<td>Implementation 6</td>
<td>280.622</td>
<td>284.968</td>
<td>324.849</td>
<td>329.1306</td>
</tr>
<tr>
<td>Implementation 7</td>
<td>134.044</td>
<td></td>
<td>144.8844</td>
<td></td>
</tr>
</tbody>
</table>

Table 15: Performance Chart with and without Tiling

Following diagram plot values shown in above table:

![Performance Chart](image)

Figure 32: Performance Chart of All OO Implementations with Tiling

It can be noticed that after tiling calculation time is improved by almost 55.31% and total time is improved by 57.73% but it doesn’t improve the performance of implementation 4 and implementation 6. Even though tiling improves temporal locality, implementation 4 and implementation 6 are required to communicate with 4 neighbouring objects of same class and 4 corresponding neighbouring objects of class world. Thus communication overhead doesn’t improve the performance of those implementations.

Amongst all the implementation, implementation 7 requires lowest time for calculation and total execution, even though it is creates multiple objects for hare and puma classes. To explain this consider following code:

```cpp
for(int i=1; i<81; i++)
    for(int j=1; j<81; j++)
        if(World.surface[i][j] == land){
```

42
As shown in above code, both calHare and calPuma methods transfer sum of four neighbouring object surface value, four neighbouring object density of same class and corresponding density of puma and hare objects. As all the values required for solving density of hare and puma for next time stamp are transferred as argument in function it avoids all communication overheads inside calHare and calPuma methods. Moreover as shown in the code above it requires to check world surface condition and sum of neighbouring object surface only once for both hare and puma objects unlike of twice in implementation 1. These minimise overhead of $M \times N$ if condition checks and sum of neighbouring objects surface value which makes implementation 7 to run faster than implementation 1 which is almost similar to the serial implementation.
6 Conclusion and future work

This dissertation report has presented different strategies for solving predator prey algorithm using object oriented programming. During the implementation of different design, potential of object oriented programming for much greater flexibility, extensibility and re-use has been explored. For example introduction of new species for extending predator prey algorithm would be easy for implementation 5, 6 and 7. The design discussed for extension to all design in section 3.2.7 would explore reusability of code for different implementation of partial differential equations and implementation 4 and 6 provide the most robust code by exploring data abstraction between different objects of hare and puma.

During this report performance is focused on entire code efficiency rather than computational efficiency. But in terms of efficiency, performances of object oriented codes are more inefficient as compared to functional code because of suggested flexibility, robustness and reusability. However it is affordable to some performance loss to get a flexible and robust code. Moreover reusability of code could save considerable amount of time for development of new code.

The future agenda for this project would be comparing features and performance of implementations of these designs into other object oriented programming languages like C++ or FORTRAN90, as these languages provide different features. For example FORTRAN90 provides arrays which are not provided by C++, while templates are not offered by FORTRAN90[14].

As software developed for parallel machine are sometimes more specific to machines and can not be used for other parallel machine. So it would be interesting to parallelise some implementations of predator prey algorithm and comparing performance of these implementation for different architectures.
References


A  Compilation of Programs and Submission of Jobs

The appendix A will provide details about how the C and Java programs are being compiled and how jobs are submitted to back-end machine.

A.1  Compilation and Jobs Submission for C Program

C program can be compiled by command given below:

```bash
c c -o main cio.c main.c -lm -lrt
```

here tags -lm and -lrt are added for maths library and system clock time. This command should be run from the same directory, where cio.c and main.c file are present.

During this programs 4 floating point arrays and 1 integer arrays are being used of size 800x1000, which required to increase the size of program stack. As 4 bytes are required to store floating point value and for an array of size 800x1000 it would required 3125 Kbytes of memory and 5 floating point arrays would require 15625 kbytes of memory.

```bash
ulimit -s 1000000
```

This command allows assigning maximum 1000000 Kbytes to program stack, which can be checked by the following command.

```bash
ulimit -a
```

Once program is compiled, the job is submitted to lomonad back-end using sun grid engine batch scheduler. It can be submitted to lomonad machine using PredatorPrey.sge batch file. This file contains only one command to run predator prey program.

```bash
./main
```

Job can be submitted by a following command

```bash
qsub -cud PredatorPrey.sge
```

As output of this job two file will be created, one for error code and one for output. Both will be named as below

```
PredatorPrey.sge.e[Job ID]
PredatorPrey.sge.o[Job ID].
```
A.2 Compilation and Job Submission of Java Program:

Java code can be compiled from the source directory using following command.

```
javac PredatorPrey.java
```

Once program is compiled, the job is submitted to lomonq back-end using sun grid engine batch scheduler. It can be submitted to lomonq machine using PredatorPrey.sge batch file. This file contains only one command to run predator prey program.

```
java PredatorPrey
```

Job can be submitted by a following command

```
qsub -csl PredatorPrey.sge
```

As output of this job two file will be created, one for error code and one for output. Both will be named as below

```
PredatorPrey.sge.e[Job ID]
PredatorPrey.sge.o[Job ID].
```

Java code with tiling are named as PP.java and which can be compiled and submitted to back-end by following commands respectively.

```
javac PP.java
qsub -csl PP.sge
```
## B Default Parameter List

To solve the density equation for hare and puma for the different time step, some default parameter are used. The value of this parameters are given below:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>intrinsic rate of prey population increase</td>
<td>0.08</td>
</tr>
<tr>
<td>a</td>
<td>predation rate coefficient</td>
<td>0.04</td>
</tr>
<tr>
<td>b</td>
<td>reproduction rate of predators per one prey eaten</td>
<td>0.02</td>
</tr>
<tr>
<td>m</td>
<td>the predator mortality rate</td>
<td>0.06</td>
</tr>
<tr>
<td>k</td>
<td>diffusion rate of hare</td>
<td>0.02</td>
</tr>
<tr>
<td>l</td>
<td>diffusion rate of puma</td>
<td>0.02</td>
</tr>
<tr>
<td>deltat</td>
<td>discrete value of timestep</td>
<td>0.4</td>
</tr>
<tr>
<td>M</td>
<td>No of rows</td>
<td>800</td>
</tr>
<tr>
<td>N</td>
<td>No of columns</td>
<td>1000</td>
</tr>
</tbody>
</table>
C  Work Plan

C.1  The following was the initial Work Plan for the project

- Background Information about predator-prey algorithm and Object Oriented Programmin -17th May-23rd May
- Functional code in C and Java ( naive solution for performance ) -24th May-6th June
- Object Oriented Design -7th June-14th June
- Mid term Dissertation submission + Java Code -15th June- 30th June
- Parallelisation of Java object code to carry out design issue of parallel code -1st July-1st Aug
- Writing and Formatting Dissertation - 2nd Aug-24th Aug

C.2  Implementation of the Work Plan

Work stated till 6th June went according to the initial plan. But soon it was realised that the predator-prey algorithm calculates the density of hare and puma in floating point but was required in an integer form. Two weeks were spent on this task but still no success was seen. This forced a modification in the mathematics of the algorithm, but this was beyond the scope of this project, and by the time it was decided to stick to the floating point format, the mid-term report submission had approached (i.e. 30th June). Hence the focus was shifted towards completion of the mid-term report.

So the Object Oriented Design process started on 1st July. Time allocated for this task was not enough and it took a week more than anticipated. Hence by this time, three unplanned extra weeks were lost. So instead of implementing only one object oriented code and carrying out the Parallelisation on that code, seven different object oriented designs were implemented by the 1st of August.

After finishing all the above stated tasks, the writing process began from 2nd August which was completed on 24th August. During this, the formatting and other required tasks for project presentation were also performed.