Ørjan Bergmann

Discrete event simulation in Java with applications in rodent navigation

Department of Informatics
University of Bergen
N-5020 Bergen, Norway

and

Neuroinformatics and Image Analysis Group
Department of Physiology
University of Bergen
Årstadveien 19, N-5009 Bergen, Norway

Thesis for the Degree of Candidatus Scientiarum in Informatics

6th September 2002

Abstract

In this thesis we will examine models which describe navigation behavior in a rat. Models describing cognitive behavior often use neural networks to account for the learning aspects of the animals they aim to describe. We will in this thesis examine a model which instead introduce abstractions taken directly from rat physiology (such as PlaceCells and GoalCells) and which facilitates learning in a more explicit way.

In order to examine this model we will create a discrete event simulation, implemented in a relatively new object-oriented programming language called Java. Simulations can be implemented effectively in many different languages, but object-orientation is a methodology which promises several advantages, especially for discrete event simulation. Although implemented in Java, we will present our discussion as language-neutrally as possible so that the analysis holds true for other object-oriented languages as well.

Using data extracted from real rats, we will demonstrate that it is possible for simulated animals to navigate successfully in an environment containing obstacles, using the model we describe.

Contents

List of Figures

Acknowledgements

Upon completion of this thesis I would like to thank Professor Trond Steihaug for always being available for answering my questions, for guiding this thesis in the right direction and for giving invaluable feedback. I would also like to express my gratitude to Associate Professor Arvid Lundervold for inspiring me to choose an exciting field of study, for supplying me with numerous papers and articles and for all comments over the course of this thesis.

Also thanks to Associate Professor Bolek Srebro and Eirik Thorsnes for ideas and for allowing me to base parts of this thesis on their results.

Finally, I would like to thank Silje M. Kalsaas for always being patient, and for reading this thesis.

I dedicate this thesis to my father

Ørjan Bergmann
Bergen, 3rd September 2002

Chapter 1

Introduction

1.1 Why this work?

In the fall of 1998 a team of researchers lead by Bolek Srebro opened a lab at the Department of Physiology at the University of Bergen devoted to the exploration and examination of rat brain activity. Among the results produced from this lab was the Master thesis of Eirik Thorsnes [Thorsnes, 2001], describing the dynamic behavior of rat brain when introduced to a new and open environment. An important part of that thesis was describing how a special type of cells in the rat brain seemed to encode the physical position of the rat in the cage. These cells are appropriately called "place-cells", and there has been many attempts to model their behavior since they were first discovered in 1971.

Today results describing place-cell behavior has become readily available for examination from several research facilities around the world. This has prompted the need to find and examine models describing place-cells and their uses.

1.1.1 Motivation

Modeling is a powerful tool for examining complex systems. We model systems in order to learn more about a systems behavior. Modeling is one of our most fundamental tools to that end.

Complex systems which depend on input and produce output are sometimes termed information-processing systems.

Depending on how much details is known about the transformation from the input data to the output data, one can attempt to model the system either as a input-output mapping without implementing the actual processes taking place in between or, if enough information is available, by modeling each of the systems constituent parts.

David Marr [Marr, 1982] formalizes these two mentioned approaches, and introduces a third level at which information-processing systems can be understood.

Computational theory A top down view of the system describing it as a mapping of one type of information to another. At this level of understanding it is important to consider what the goal of system is and why it is appropriate.

Representation and algorithm This level of understanding is primarily concerned with how the computational theory can be achieved. In particular, what is the representation of the input and output, and what is the algorithm for the transformation?

Implementation At the most detailed level of understanding we are concerned with how the representation and algorithm can be realized in the context of interest. The context may require either physical (hardware) or software implementation.

Biological motivation

There exist several models describing navigation in rodents. Some models are very specific and model the entities they describe in a detailed mathematical language and focus on hardware specific implementation. Other models instead propose general principles and focus on *why* they are appropriate.

We will think of the rat as a machine which, provided with input, will produce response (output) characteristics. The purpose of this is to gain deeper insight into the information-processing system which is the rat; *we want to know how the rat navigates.* Knowledge of this would probably provide important insight into how the brain of rat works, and could eventually lead to the tantalizing prospect of direct animal-machine interaction at the physiological level.

Additional thoughts But we might also wonder about what the purpose of the rat is. For each input, the rat will perform an action, but what is the rats ultimate goal in life? Why does it have behavior? It is generally believed that the rat has *instincts* which helps it survive. When hungry, it will search for food. When tired, it will find a place to sleep. But these simple motivations cannot explain all of the rats choices. There might be an element of randomness involved, or there might be incentives that we as humans simply cannot understand. The truth is that we do not know what motivates the rat, we simply assume that something does. Later on we will assume that the rat wants to go to a given place in the cage, but why it would want to do so we leave for others to speculate on.

Often when examining models of biological origin, the principles learned are applicable to a other areas of science as well.

The case of rat navigation can in many instances have direct implications to navigation in robotics. Experiments are continually being carried out by the military to find new and efficient ways of guiding rockets and missiles. Space-exploration is also an area which can benefit greatly from improvements in automatic guidance systems. When exploring relatively remote planets such as Mars (or even the moon) it is impractical to rely solely on remote control of exploration vehicles from earth. One of the main problems is that it simply takes too long time for the signals to travel. Therefore robots which are able to learn an environment, and navigate on its own are of special interest. See [Burgess et al., 1997] for more details.

But a model can also have important implications for its originating system. By experimenting with a model it is often possible to make predictions about how the real system should behave in certain situations. When these predictions are paired with new examinations of the original system, this can lead to important discoveries.

Motivation for computation

There exists many tools in computer science for analyzing a model describing a complex system. The tools that are appropriate for one model are often not appropriate for another.

Models are often divided into continuous or discrete models. In continuous models there exist a continuous relationship between the input to the model and the model output. Typically, such models use a large number of linear or non-linear equations which can be solved by analytical or approximation methods. Models describing the currents in the North Sea is a typical example of a continuous model which depend on several continuous variables such as time and temperature.

In discrete models on the other hand there exist no continuous relation between the models input and output. The model output can only be evaluated at discrete values for its input parameters. Such problems are often difficult to solve, since many of the classical analytical tools developed for continuous models fail for discrete models. However, if the number of variables in the discrete model can be made small, then discrete models allows us to test the output for some especially selected values of its input parameters in a systematic manner.

This process of examining a models output for certain values of its input is called simulation. Simulation has the great advantage that data produced from the model can easily be compared to data gathered from the real system. This offers a clear way to falsify the model: if the model and the system produce "too different" results for the same input, then the model is false, and must be discarded. Otherwise, the model *may* be correct.

Since experimental data describing the real system behavior is available, we believe simulation is a good way to test our models. The experimental data is sampled at discrete points in time, and at a resolution which with the current equipment cannot be made finer. There is some evidence that parts of the underlying physiology in rats is discontinuous. Other characteristics of the rat is continuous, such as position and direction. Nevertheless it is generally believed that movement below a certain threshold value cannot be attributed to a conscious decision by the rat. This suggests that if we introduce a fine enough resolution, then we can capture the important aspects of movement, while excluding details most likely caused by randomness. All things considered we believe that a discrete model seems reasonable for the systems we would like to model.

Discrete event simulation (DES) is one classical and well understood method for examining a model. It has previously been showed that DES can be applied successfully for a large range of complex problems. DES is a method which can benefit greatly from object-orientation [Ziegler, 1991].

Java is one relatively new object-oriented computer programming language. We would like to examine if Java is a suitable language for implementing our models. The object-oriented features of the Java language should be ideal for the implementation of a DES. We hope to show that the models we will simulate can be easily implemented and simulated in Java.

1.2 Goals for this thesis

There exists several models describing physiological phenomenons in the rat ranging from neural networks and artificial intelligence to mathematical models involving large sets of equations. These models describe a complex system, which involves several components often not well understood (even by researchers in the field of physiology or medicine). We will in this thesis show that simulations implemented in the Java programming language is a good way of examining such systems.

In order for us to examine the models describing rat behavior, we first need to create a framework in which the simulations can take place. This should be done by implementing a discrete event simulator in the Java programming language. This framework should be general enough to be used for simulation of a large range of problems. We will also need some problem specific extensions to implement the rat simulations.

An important aspect of all simulations is to see whether the simulations indeed does provide reasonable results for the models they attempt solve. Given that experimental data from rats during exploration in an open environment is available, we should attempt to see if real rats could use the models we describe for navigation.

Therefore the goals for this thesis can be summarized as follows.

I. Create a robust and general framework for discrete event simulation in Java.

II. Use the general framework to create an extension for simulation of a rat moving freely in a cage.

III. Examine models describing rodent navigation by implementing them in our framework.

IV. Check whether the models can be used for describing rodent navigation in real rats.

There is one part of this thesis devoted to each of these goals.

1.3 Organization of the thesis

This thesis is divided into four parts. In Part I we start by examining a few examples of simulations, in order to determine their constituent parts. We then design and implement a general software library (or *framework*) for simulation from scratch. We will use this framework to implement a small version of a classical simulation; The ElevatorSimulation [Knuth, 1969].

In Part II we will extend the general framework created in Part I to be especially adopted to the simulation of a rodent moving freely in a cage. We will specify and implement each of the components in the rat simulation, and describe how we generate input in the form of vision, smell or touch for the simrat. After this part we will have a testbed in which we can experiment with various models describing both physical and cognitive rat behavior.

Part III is dedicated to modeling and simulation. We will consider two models [Tchernichovski and Benjamini, 1998, Trullier and Meyer, 2000] describing various aspects of rat behavior and implement them in the extended framework developed in Part II. We will give an overview of our implementation, and show results from the simulations performed with the two models.

In Part IV of the thesis we will examine experimental data gathered from real rats [Thorsnes, 2001]. We will give an overview of the data collection and refinement process involved. We will then use the data collected to adjust the free parameters of one of our models, so that the simrats internal representation of the environment is similar to those found in real rats. This adjusted model will then be simulated once more, and we discuss the results in comparison with those produced in Part III.

Part I

A general framework for simulation

We will start off this thesis by creating a general *framework* for discrete event simulation. This will be done by creating a custom class-library implemented in the Java programming language.

Although we have chosen to implement the framework in Java, the ideas presented in this thesis have been described as language-neutral as possible, so that the framework could easily be ported to a different object-oriented language, if the need for implementation in a different programming language should arise.

All the later simulations in this thesis will use this class-library, or framework as we will call it, extensively. In order for the framework to be as good as possible, we will first discuss what goals we strive for when creating computer programs. This will be done in Chapter 2 of the thesis. There we will also define a few quantities which are essential in discrete event simulation.

In Chapter 3 we will consider two examples of simulation; the elevator and the simrat simulation. We take a look at the elements (later referred to as *simulation entities*) which are needed in the two models. This chapter will introduce examples which we will later use as a basis for generalization.

Chapter 4 describes the structure for our simulation framework. Each element in a discrete event simulation will be described in detail. We also describe how each of the simulation components relates to each other in a class-hierarchy. Some notes on how efficient the framework is, and how it is implemented is also included.

The final chapter in Part I describes and implements the elevator simulation introduced in Chapter 3. The aim of this chapter is to exemplify the ideas described in the previous chapters. The results presented in this chapter will show that the framework we have created is sound and is working properly.

Chapter 2

Simulation and object-orientation

2.1 Models and simulation

Models and simulations of various types are tools for dealing with complex systems and the interactions of their constituent parts. Scientists and engineers have long used models to better understand the systems they are studying, for analysis and quantification, performance prediction and design.

According to Carson and Cobelli [Carson and Cobelli, 2001] a model is a representation of reality involving some degree for approximation. In classical scientific terms, modelling can be used to describe, interpret, predict or explain.

Simulation is the process of solving a model to examine its output behavior. When carried out on a model, simulation yields output responses that provide information on system behavior. Depending on the modelling purpose, this information assists in describing the system, predicting behavior or yielding additional insights (i.e., explanations).

Simulation offers a way forward in situations in which it might not be appropriate, convenient or desirable to perform particular experiments on the system. Such situations could include those in which experiments cannot be done at all, are too difficult, are too dangerous, are not ethical, or would take too long to obtain results. Therefore, we need an alternative way to experiment. Simulations offers an alternative that can overcome the preceding limitations. Such experimenting can provide information that, depending on the modelling purpose, aids description, prediction or explanation.

There are two fundamental goals that we try to achieve in all simulations.

- Correctness
- Efficiency

Correctness is important in all models describing reality. We say a model is correct if it can be validated. Validating a model is essentially examining whether it is good enough in relation to its intended purpose. This assumes that, in a Popperian sense, that it can be tested and be falsified; it must be readily capable of bringing about its own downfall. However; clearly no model can have absolute unbounded validity given that a model is essentially an approximation of reality.

As simulations becomes large, the need for efficiency becomes paramount. Often simulations will be run over and over again in order to examine how a model behaves and changes with the model parameters. Running time is here one measure of efficiency. Although the running time of a simulation is of extreme importance, it is not the only measure of efficiency. It is also important that the implementation of the model is simple to understand, so that verification and changes to the model can be done efficiently. Since simulation often involves alteration of the model implementation, it is vital that the simulation is implemented in such a way that changes does not produce unintended errors or require unreasonably long implementation time. If a simulation is inefficient, this may force us to accept results which are less than optimal because of time requirements imposed by outside factors.

In the next sections we will examine guidelines which will help us achieve our two goals of correctness and efficiency.

2.2 Software

The simulations we will create will all be implemented on a computer. We will create several software packages, and it is important that these packages will be written according to guidelines that will make them as good as possible. We will therefore next discuss what we expect from good computer software.

2.2.1 Desirable virtues

Whenever one sets out to create computer software, there are, according to Goodrich and Tamassia [Goodrich and Tamassia, 1998] a few basic implementation goals that one should always try to achieve

- Robustness
- Adaptability
- Reusability

Every good programmer wants to produce software that is *robust*, which means that a program produces the correct output for all input, and when run on different hardware platforms. Robustness is especially important when for instance a user tries to exceed the limitations of a given piece of software. It is then important that the software recovers gracefully from such an incident.

Software programs are typically expected to last for a long period of time, often over many years. In order for this to be possible the software needs to be able to evolve over time in response to changing conditions in its environment. These changing conditions could be anything from need for better utilization of a faster CPU to more functionality. It is important for good software to be *adaptable* enough to allow all this.

Developing software is both a very time-consuming and expensive process. If we allow for our software to support *reusability* both factors can be significantly reduced.

2.2.2 Object-Orientation

Many implementation methodologies have been developed that attempts to allow a programmer to easily produce software that meets the implementation goals listed above, while at the same time being powerful enough to allow a programmer to fully express efficient high-level solutions. Object-orientation is one such methodology, that promises both simplicity and power. Among the most important principles of object orientation are

- Abstraction
- Encapsulation
- Modularity

According to Goodrich and Tamassia [Goodrich and Tamassia, 1998], the notion of *abstraction* is an important concept in computer science. The main idea of this concept is to distill a complicated system down to its most fundamental parts and describe these parts in a simple, precise language. Typically, describing the parts of a system involves naming the different parts and describing their functionality. This functionality will become the abstract interface which other components can work with. This combination of clarity and simplicity benefits robustness, since it leads to understandable and correct implementations.

Another important principle of object-oriented design is the concept of *encapsulation* or *information hiding.* This principle states that different components of a software system should implement an abstraction, without revealing the internal details of implementation. One of the main advantages of this is that it give the programmer freedom to implement the details of the system as long as the abstract interface that the other components see is the same. Thus, encapsulation yields adaptability, since it allows the programmer to change only certain parts of the software without having to change all the other.

The final aspect of object-orientation that we will discuss is *modularity*. Modularity refers to an organization structure, in which different components of software systems are divided into separate functional units. This structure enables software reusability, for if software modules are written in an abstract way to solve general problems, it is likely that instances of these same general problems may arise in other contexts. Additionally, object-orientation allows for the organization of modules into an hierarchy, where it groups together common functionality at the most general level, and views specialized behavior as an extension of the general one. Thus the organization concepts of modularity and hierarchy enable software reusability.

2.2.3 Java

According to SUN Microsystems, Java is a "pure" object-oriented language, and it therefore supports all of the object-oriented principles mentioned in the previous section.

`Classes and objects` in Java presents to the outside world a concise and consistent view of the objects that are instances of a class, without going into too much unnecessary detail or giving access to the inner workings of the objects. This corresponds to the object-oriented principle of abstraction.

`Interfaces and strong typing` are important features of the Java language. Both at run-time and during compilation the virtual machine checks that parameters that are passed to methods rigidly conform to the type specified in the interfaces. Although having to define interfaces and them having those definitions enforced by the strong typing places an extra burden on the programmer, this burden is offset by the rewards that it provides, for it enforces the encapsulation principle.

`Inheritance and polymorphism` in Java allows one to simply create both general and more specialized classes. These aspects of the Java language corresponds directly to the object-oriented principles of modularity and hierarchy.

Additional advantages

Java's clear and unambiguous language will help us to avoid possible errors by disallowing some of the more "tricky" language features found in other object-oriented languages (for instance *multiple inheritance* and *operator overloading*). Although these features may be convenient in certain types of applications, they may also introduce errors if used without precautions. Java avoids these errors by not supporting such language features.

2.3 Terminology

Before we start discussing the various components of an discrete event simulation, we take time to define few terms that will be used often throughout this thesis.

2.3.1 Simulation entities

We will follow the definitions given in [Helsgaun, 2000]. We start by defining what we mean by *simulation.*

Definition 1 (Simulation)
Simulation is experimentation with a model in order to obtain information about the dynamic behavior of a system.

Instead of experimenting with the system, the experiments are performed with a model of the system.

The system components chosen for inclusion in the model are termed *entities.* Associated with each entity are zero or more *attributes* that describe the state of the entity.

Definition 2 (System state)
The collection of all attributes in a simulation at any given time is called the system state *at that time.*

We name changes to the state of a simulation a *state change.* Each state change will in our simulations be performed by an *Event.*

Definition 3 (Event)
Object especially created for the altering of the state of a simulation.

In the context of Java programming this means objects that implements the bergmann.simulation.Event interface.

2.3.2 Time

In all simulations, time plays an important part, and we therefore now define a few quantities related to time.

Definition 4 (User time)
User time, T_{user}, is time as perceived by an observer watching the simulation from outside the computer, independent of what goes on inside the computer.

The difference between start-time and end-time of the simulation (expressed in user time) is often called the running-time of an application. Often we would want to keep the running-time as small as possible. The term "wall-clock" is also often associated with the user-time. This term is used to emphasis that user-time progresses independently from the simulation.

Definition 5 (Simulation time)
Simulation time, T_{sim}, is time as perceived by the simulation entities.

Simulation time will in a discrete event simulation be discontinuous (hence the word discrete), and we call each update of the simulation time a *tick*.

Definition 6 (CPU time)
CPU time, T_{cpu}, refers to the amount of user-time, T_{user}, that passes between each tick of simulation-time, while an event is being executed.

Conversely, the accumulated CPU time over an entire simulation is the user-time it takes to execute the simulation, disregarding the overhead imposed by the simulation framework. The running-time of a simulation, on the other hand, is the amount of user-time it takes to execute the simulation altogether.

Definition 7 (Time complexity)
Time complexity is user time expressed as a function of the input size, n.

The input size n will, unless specified otherwise, refer to the total number of events in the simulation.

Time complexity is usually expressed using asymptotic notation ($O(\cdot)$ or $\Theta(\cdot)$). Asymthotic notation shows how user time varies if the size of the input, n, becomes large ($n \to \infty$). This implies that time complexity expressed in this notation is only valid as a measure of user time when the input size is large.

Corollary 1 *From the definitions 4, 5 and 6 we may express the following relation*

$$T_{user} = \sum_{i=1}^{n} T_{cpu}^{i} + f^{i}(n)$$

where n is the total number of ticks in the simulation and $f^i(n)$ is a value associated with finding and preparing event i for execution. T_{cpu}^i is the CPU time it takes to execute event i.

Although we would rarely need to calculate the user time from corollary 1, the formula is useful in determining which part of the system should be optimized if one would want to reduce running-time.

Chapter 3

Introducing the elements by two examples

I order for us to create a good framework for simulation we will, inspired by Jacobsen et al. [Jacobson et al., 1999], start off by looking at a few examples[1] in order to determine some of the structure and requirements we are faced with in discrete event simulation.

In Chapter 4 we will develop the final, more formal and general framework for these types of simulations.

3.1 An elevator example

The elevator simulation is a well known computer science problem, also discussed in the classic work "The Art of Computer Programming" [Knuth, 1968].

The elevator will reside in a building with many floor, and will be exclusively devoted to transporting people between floors. Based on this simulation we wish to determine the average time from when a person presses the elevator-button in one floor until he leaves the elevator in another floor. This time will include time waiting for the elevator, time entering and leaving the elevator and time inside the elevator. Among the parameters that the user of the simulation should be able to specify are the number of floors in the building, the number of people using the elevator and over how many time-units (of simulation time) the simulation will run over. Details of how the elevator will prioritize the different floors should be possible to specify (and change) easily.

Because the elevator example is one we all have first hand experience with from real life, it is ideal for validation of the general framework. By experimenting with elevator simulations of different sizes, we also hope to be able to determine how scalable the framework is. An important aspect in determining scalability is time complexity.

[1]also known as "use-cases" in various literature

A similar but more elaborate example over the same theme is discussed in Chapter 5.

3.1.1 The model

In accordance with the design principles presented in Section 2.2.2, we start making abstractions over the elements involved in a real-life elevator. The building will map to an object in the simulation. Noting that in real life, a building consists of several floors each (possibly) populated by people, we give the building a table called "floors". Next we give each floor a list of Persons. These people are the ones that will reside in that floor. In real life, an elevator resides inside the Building, so we will make the Building have a property called "elevator". Actually, now we have already introduced our two next abstractions; the Person and the Elevator.

The aspects of a person that are important for us in our simulation will now need to be addressed. Among them are which floor the person starts in and which floor the person wants to travel to. This person should also have a property which registers what (simulation-) time it is when he presses the elevator button, and what the time is when he exits the elevator. These values will later become important when one wants determine the average time between floors.

The next abstraction we describe will be the Elevator itself. The elevator will need to have a property called "peopleInsideElevator", that will be a list of all the people currently inside the elevator. We will need a value that will specify which floor the elevator is in and another value that specifies the current state of the elevator ("Moving", "LettingPeopleInOrOut" or "Stopped"). An important property which the elevator will also need is our final abstraction; the ElevatorController.

The ElevatorController is an abstraction over the real-life computer that controls real-life elevators. It is this objects job to determine which floor will be visited next. Whenever somebody presses the button to go to a certain floor, the Elevator will notify the controller object of this request. An answer will then be given back to the elevator, so that it can move or stand still, as specified by the controller. We note that this ElevatorController object could also strictly speaking have been "hard-coded" into the Elevator, but this would violate our design principles about encapsulation, and prevent modularity. Because of our choice of making the ElevatorController an abstraction, it can more easily be swapped with another ElevatorController which behaves differently, but allows the same operations.

3.1.2 The events

The Events in this simulation should attempt to capture the real-life events that can alter the state of the simulation. We start by noting that real-life events such as pressing the move-to-this-floor button or buttons inside the elevator most certainly would do so. We therefore should create Event-objects for these events.

The same is also true for all floors we stop in: people must get in or out of the elevator, and since this would also most certainly change the state of elevator, we should also create an Event-object here.

Finally, we need some way to move the elevator up or down. This has been solved by having the elevator generate a special move up or down Event-object a given time after the elevator reaches a floor. This MoveEvent will query the Elevator, which in turn will query the ElevatorController, about what to do. The Elevator will then move one floor up, one floor down or stand still depending on the answer. The abstraction of the MoveEvent is therefore our last Event in this simulation.

We should also be able to verify the elevator simulation. One possible way of doing this is by calculating average values for the quantities we need to produce the average waiting time. We will in Chapter 5 show how this can be done.

3.2 A small rat simulation

We now present a simplified version of the rodent navigation problem presented in Part II, to show the similarity with the elevator simulation, and to simplify the later discussion about rodent navigation.

We would like to create a model of a rat moving in a cage. The rat will become stimulated by its surroundings, and base its behavior on these stimulations. The purpose of this simulation is to see if one can develop a good description of the rat navigation strategy, so that the simulated rat behaves similar to real, living rats.

Notice how this specification also implies some sort of validation of the simulation results. This will be performed in Part IV, using data from real rat experiments.

3.2.1 The model

Our first abstraction in this model is that of the cage. The cage will need to have properties such as length and width. We will also allow for the cage to have obstacles placed in various places, through which the rat cannot move. Since, in real-life, the rat resides in a cage, we choose to make the rat position and which direction it is facing a property of the cage. What direction the rat is facing is important in determining what input the rat should be given (for instance what the rat will see).

The next abstraction we make is that of the rat. The rat will need to be able to change its position in the cage, and to turn around, making it face another direction.

As with the elevator simulation, we also make an abstraction out of the decision-making part of the active part, and create a RatController object. This Controller object will be responsible for deciding what to do (move or stand still), based on input from the environment. In this respect we hope that it will behave similar to the brain of real rats.

3.2.2 The events

Probably the most important type of event that would occur in this simulation is when the rat receives input from its environment. This will happen over and over again. Although a real rat will receive this information continuously (or at least as long as it has its eyes open, or it is listening), we hope that as long as we give input often enough per time unit, we will approximate a continually input source.

Although the input to the rat is continuous, there is some evidence presented by Walløe [Walløe, 1968] to suggest that the real rats does not process the input as a continuous stream of data. Instead all inputs are processed in the rat brain by nerve-cells which are either in an *active* or *non-active* state. If this is true, then a discrete representation of the input in our simulations would not seem unreasonable. Actually, Walløes work on neural processing in animals was one of the first applications of discrete event simulation.

Additionally the rat will need to determine what to do, at certain intervals. We have solved this by abstracting a special Move-object. It will also be called at regular intervals as with the InputEvent.

3.3 The timeline

As described in both examples above; a series of events will take place, all possibly altering the state of the simulation. But in order for these changes (or events) to take place and make sense the events will need to be handled sequentially, and one at a time. Humans are accustom to think and plan in terms of time, and we adopt this notion in the timeline. The timeline's mission is therefore to store each event that should happen in the future in an internal buffer, and then execute them sequentially and one at a time according to their registered execution time.

Conversely, the timeline will be used to keep track of which event will occur next.

Chapter 4

A general structure for discrete event simulation

According to the authors of "Design Patterns" [Gamma et al., 1995], one of the most difficult tasks of computing is that of creating a good and reusable general framework for the solution of a whole class of problems. When creating such a framework it is important that it be both robust, simple to use, fast, powerful and, perhaps most importantly of all, be general enough for it to be usable for a whole range of problems. The choices made about the architecture of a framework, will dramatically influence what the final program will be like.

4.1 The elements

There are a few elements that one can expect to be needed when creating a discrete event simulation. In the following sections we will describe the most important of them.

4.1.1 Timeline

The timeline is probably the single most important part of an discrete event simulator. It is truly the heart of the simulation and the one thing that brings the simulation forward. The timeline will need to be notified of all events and when they are to be executed. A method that will execute the "next" event (that is the one with the smallest time greater or equal than the current time) must also be defined.

The Timeline will typically use a PriorityQueue to keep track of the registered Events.

Program 4.1.1 From Timeline.java

```
public class Timeline {
  public void activate (Event event, Object time){...}
  public Event tick (){...}
  public Object currentTime (){...}
  ...
}
```

PriorityQueue

A priority queue is formally defined as an abstract data type (ADT) that imposes a structure upon its previously submitted elements. This ADT will upon request return the smallest element registered. This formal level of abstraction allows for a modular design pattern. By not tying ourselves down to any specific priority queue, we allow for parts of the program to be changed (even possibly at runtime), which in turn gives greater freedom of choice and easier problem specific adaption of the simulation framework. This can greatly improve performance.

Traditionally [Dahl and Nygaard, 1966] a linked-list has been the priority queue of choice for discrete event simulation. This because its ease of implementation and speed (under certain assumptions. See Section 4.3).

The problem of concurrency

Given that two events are scheduled to take place at the same time, which of them should occur first? Unfortunately there is no easy answer for this question, and there is no general strategy here that will always produce correct results. One might suggest always returning equal elements from the priority queue in the order in which they were inserted, (also referred to as *first-in/first-out*) but this would restrict our choice of priority queue to only those that could be implemented in such a manner (for instance *not* the heap).

Therefore we can *make* no *assumption about the order in which equal elements are returned in.* This means that if events should occur in a certain order then the user should not insert them into the queue with equal values.

Sometimes a situation might arise, during which you would want a series of events to occur sequentially, without any other events being executed in between them. Because of our above choice, the strategy of inserting all the events at the same time, will generally not work. There are many possible solutions to this problem, but we propose the following two. Firstly, the m elements may be "spread out" in time by scheduling them to occur at time $t_0, t_0 + \delta, t_0 + 2\delta, \ldots, t_0 + (m-1)\delta$ instead. For a small enough value of δ this should ensure that the events are processed uninterruptedly, in sequential order. However, now we cannot guaranty that other events won't be activate in between. The other approach is to create a special MultiEvent object, that keeps reference to the sequence of events you would want executed, and then register this object for activation. Upon its activation, it should in turn activate all events it has reference to. With this approach, however, one should keep in mind that the simulation time will stand still during execution of the internal event-list. For more information about this approach, see page 16.

4.1.2 Event

Events are crucial in an simulation since they are the ones that change the state of the simulation. There can be given no default action for the events, since how they actually will change its context will always be problem specific, and therefore left for its subclasses to implement. All events therefore only need to provide us with one functionality; how to activate it.

Program 4.1.2 From Event.java

```
public interface Event {
  public void activate ();
  ...
}
```

The states of an event

Upon realizing that, even for a medium sized simulation, the number of Event objects will be very large, and the noting that each single event often does not do very much, we suggest later (Section 4.3) a means of improving performance. In order for us to take advantage of this we should first make some simple changes to the suggested interface.

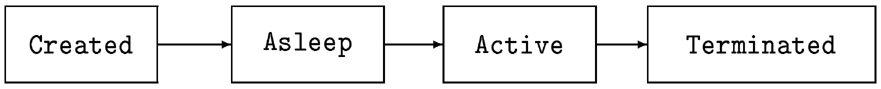

Figure 4.1: The life-cycle of an Event

Dahl and Nygaard [Dahl and Nygaard, 1966] suggest that each event has a life-cycle: It is first created, then given to the timeline for later activation, then activated and, eventually, terminated. If one were to keep track of these states one could, upon reaching a certain state (for instance terminated), perform some action. As we will later show, this could be used to improve performance for large simulations.

MultiEvent

As mentioned on page 14 sometimes situations arise, in which it would be convenient to treat a series of events as just one single Event. The MultiEvent does exactly this.

When creating a MultiEvent object, one has to supply as series of other events, in the order they should become activated. Then the MultiEvent can be scheduled for activation in the Timeline. When the MultiEvent is activated, it will in turn activate each of its registered events.

All of the events stored in the MultiEvent will be executed on the same tick of the timeline (the activation time of the MultiEvent itself) and we know that they will be activated in an uninterrupted sequence, without any other events in between.

ReoccuringEvent

The ReoccuringEvent is a special type of Event that is designed for reuse. When the ReoccuringEvent is created one must supply it with another Event which should be activated at regular intervals, as well as how long the intervals in between the activations should be. Then the ReoccuringEvent can be scheduled for activation in the Timeline.

When the ReoccuringEvent is activated it will activate the Event that it was supplied with. After activation (when it has reached the "Terminated" state) the ReoccuringEvent will schedule itself for activation again after a time period of the given interval. The ReoccuringEvent will thus be activated at regular intervals throughout the simulation, or until it is cancelled.

For flexibility, we also allow the interval between each activation of the ReoccuringEvent to be altered during runtime. This can be done by a call to its *setInterval(·)* method. This will later become useful when we will need to model *speed*.

One of the main advantages of using an ReoccuringEvent is that it will reuse the same Event object over and over again, and not have the overhead from creating new events.

4.1.3 Context

All simulations must take place in an environment. The important feature of this context is that it has the ability to be changed by events, or formally; to have its state altered. We can keep track of these states by allowing this object to store a (property, value) pair. This can be done easily using a dictionary ADT. An effective way to implement this is by using the hashtable ADT.

We should also, in designing this class, take into consideration that as time passes certain properties may not be applicable anymore, and the property in the context should become unset. We also note that this object may (and probably should) in turn be shared by several objects in a simulation. A context can obviously be divided up into individual objects (for instance the house or people in the elevator example), but doing so does not violate the idea of a context. The idea of a context merely provides us with an extra level of abstraction, so that our framework can be as general and simple as possible.

In our later simulations, we let the Context-class have references to all main simulation entities, so that all objects that are a part of the environment can be accessed from the Context.

4.1.4 Subject

There is often an active part present in a simulation. Most often this active part is the very thing we want to simulate, and from now on we formally refer to it as a Subject. A simulation may have any number of subjects involved.

The key property of the subject is that it is able to make decisions, and most often cause events to occur based on these decision. In order to do so it needs to have some internal state that drives its choices. This internal state is updated by events, and when a decision is to be made, the internal state is examined, and a choice takes place.

The elevator is one example of a typical subject. The elevator receives input in the form of the press of various buttons inside or outside the elevator, and the elevator stores these clicks in an internal data structure. When the MoveEvent queries the elevator of what to do, the elevator itself examines its internal data structure, and gives a reply depending on how it interprets the data.

Strategy

Often when modelling a subject, it is desirable to have a means through which behavior can be changed easily between (or perhaps even, during) runs. The best way to ensure this is to encapsulate all the decision-making into one part of the system, with a well defined interface to the subject. We

Program 4.1.3 From Strategy.java

```
public interface Strategy {
  public void processInput (int type, Object input);
  public int recomendMove ();
}
```

formally call this object the Strategy which the subject uses. The interface should be simple but still allow for all sorts of input and output. It will also later become evident that if we were to allow input to be of a particular type, this will simplify usage, and prevent unnecessary testing.

So when the state of the simulation changes in some way important to the Subject, the Subject is notified of this (by an event). The Subject then forwards this information to the Strategy which stores it. When the framework later queries the Subject of what to do, the Strategy examines its stored information, and returns an answer to the Subject. The Subject then returns this reply to the framework, which acts on it or ignores it (if it for instance is illegal).

4.2 The final framework

Based on the previous sections we suggest the class diagram presented in figure 4.2.

In creating these class diagrams, we have decided to use an "UML like" notation. These diagrams were created mainly to provide a simple overview of the classes and their relationship, and does not show all details surrounding the classes. We considered using standard UML for these diagrams, but decided that it would clutter up the diagrams with too much details.

We let each class be represented by a box divided into two parts. The top part displays the *class name* (for instance "Subject" or "Timeline"). The lower portion of the parts shows the *methods* which the class provides. Each line here represent one method. The first character in each line denotes the *visibility* of that method, + means *public*, # means *protected* and - means *private*. The word before the starting parenthesis is the *method name*. The *method parameters* is shown in the parenthesis behind the method name. The parameters are separated with a comma, and are on the form [name]:[type]. The word after the closing the parenthesis is the *return type*. If the method does not return a value, then this field is left empty.

Interfaces are represented in the class diagram in the same way as a class, except that the class name is preceded by the *interface* keyword. Our simple notation does *not* distinguish between abstract classes and (ordinary) classes.

The white-headed arrows between boxes represents an inheritance relationship. A class which implements all members of an inherited class is marked as an *implementation class*.

The black-headed diamond-arrows is in standard UML used to symbolize aggregation. We have in our notation relaxed this a little bit, and we let this symbol represent "*references*". The "diamond-relation" from Simulation to Subject indicates that the Simulation class holds a reference to an instance of the Subject class. The Subject class in turns holds a reference to an object which implements the interface Strategy, since there is a "diamond-relation" from Subject to Strategy.

The classes Simulation, Subject, Strategy and Context are all intended to be extended, implemented or overwritten before actual use in a simulation.

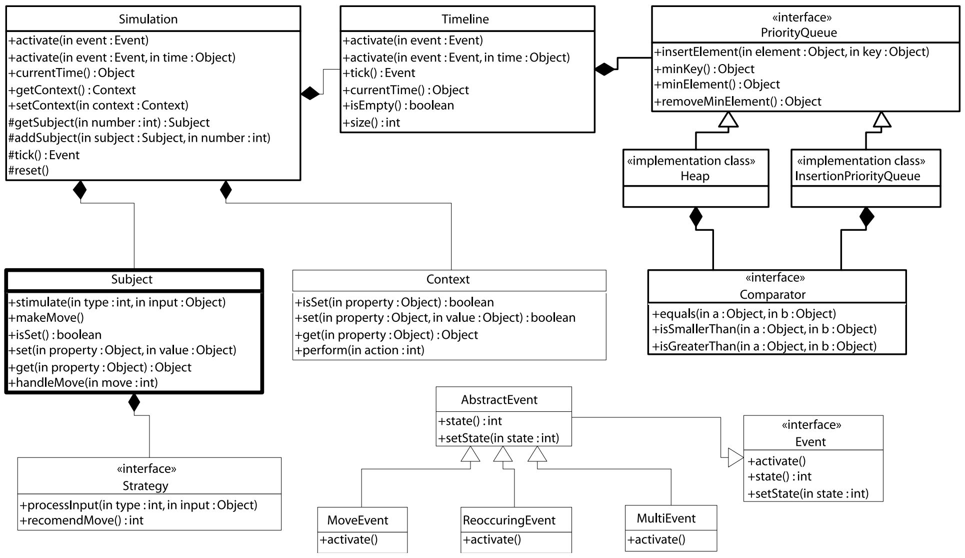
Simulation
+activate(in event : Event)
+activate(in event : Event, in time : Object)
+currentTime() : Object
+getContext() : Context
+setContext(in context : Context)
#getSubject(in number : int) : Subject
#addSubject(in subject : Subject, in number : int)
#tick() : Event
#reset()
Timeline
+activate(in event : Event)
+activate(in event : Event, in time : Object)
+tick() : Event
+currentTime() : Object
+isEmpty() : boolean
+size() : int
«interface»
PriorityQueue
+insertElement(in element : Object, in key : Object)
+minKey() : Object
+minElement() : Object
+removeMinElement() : Object
«implementation class»
Heap
«implementation class»
InsertionPriorityQueue
Subject
+stimulate(in type : int, in input : Object)
+makeMove()
+isSet() : boolean
+set(in property : Object, in value : Object)
+get(in property : Object) : Object
+handleMove(in move : int)
Context
+isSet(in property : Object) : boolean
+set(in property : Object, in value : Object) : boolean
+get(in property : Object) : Object
+perform(in action : int)
«interface»
Comparator
+equals(in a : Object, in b : Object)
+isSmallerThan(in a : Object, in b : Object)
+isGreaterThan(in a : Object, in b : Object)
AbstractEvent
+state() : int
+setState(in state : int)
«interface»
Event
+activate()
+state() : int
+setState(in state : int)
«interface»
Strategy
+processInput(in type : int, in input : Object)
+recomendMove() : int
MoveEvent
+activate()
ReoccuringEvent
+activate()
MultiEvent
+activate()

Figure 4.2: A class diagram showing the relationship between the various components in the simulation framework.

4.3 Efficiency considerations

The PriorityQueue

Corollary 1 states that here are two main factors that affect the speed of a simulation, for a given problem of a fixed size: the complexity of the events and the effectiveness of the priority queue.

For certain types of simulation, each individual event will only perform a constant number of operations. This could be simple testing and setting of only a constant number of properties. If so, then we may disregard the activation of the events contribution to running time as constant and the choice of priority queue will be important for the total running time.

Conversely the choice of priority queue determines, among other things how much user time passes between each execution of events. As mentioned in Section 4.1.1, the classical choice of the priority queue has often been a linked-list. The linked-list data structure has several desirable qualities that, under certain assumptions, makes it good for simulation. Being able to provide constant-time access and removal of the minimum element, is an optimal time-complexity for this task. This property implies that if using a linked-list driven timeline, the time between each execution of an event is constant (assuming no insertions into the timeline takes place during that event). Insertion, on the other hand, is not as quick using a linked-list. In order to determine how deep into the priority queue the next event should be inserted, $O(n)$ elements needs to be examined.

A good alternative to the linked-list priority queue is the heap. The heap

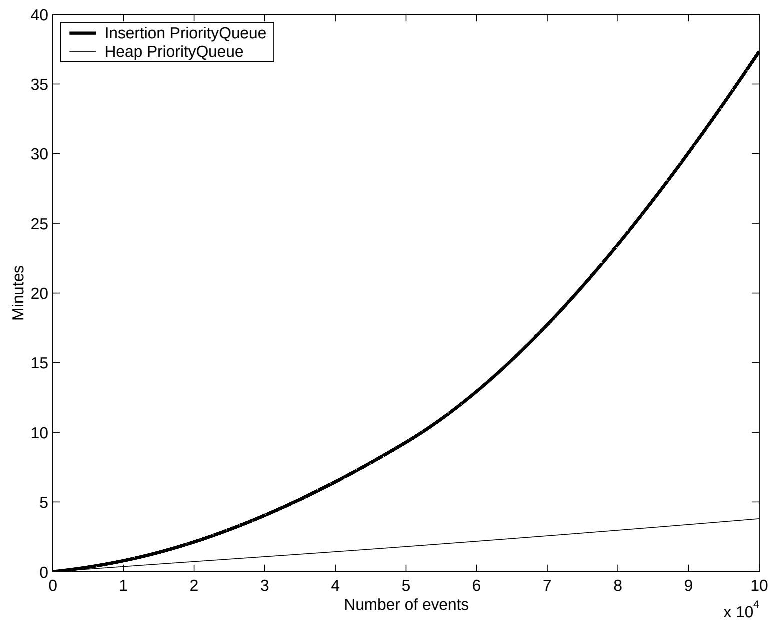

Figure 4.3: Example of the expected running time of a timeline using linked-list and heap priority queues. These results are interpolated from the results of several runs from the elevator simulation example in Section 5.4. Note that each Event only perform a constant number of operations and therefore the implementation of the priority queue has a a large impact on performance.

allows for both $\Theta(\log n)$ insertion an retrieval for elements. Although the heap takes $\Theta(\log n)$ time to retrieve the minimum element, as opposed to the linked-lists $O(1)$ time for the same task, the heap should still perform better on the whole because of its $\Theta(\log n)$ complexity for insertion versus the $O(n)$ for the linked-list.

Of course all of these observations are under the assumption that the size of the input (n) is large.

Optimality with respect to time complexity

Corollary 1 states that the overall running-time of a discrete event simulation equals $\sum_{i=1}^{n} T_{cpu}^{i} + f^{i}(n) = O(nf(n))$ if T_{cpu}^{i} is a constant. n is the total number of ticks in the simulation and $f(n) = \max f^{i}(n), i \in [1, \ldots, n]$. $f(n)$ is now a value determined by which priority queue you choose. It therefore follows from the discussion in the previous section that the overall time-complexity for using a linked-list is $O(n^2)$ and for a heap, $\Theta(n \log n)$. This is, as we will show, not always optimal.

We observe that if the events had always been kept in sorted order on increasing activation time, then finding the next event scheduled for activation would be simple; it would be the first element in the sorted sequence. Any sorting algorithm therefore seems to be a good choice for our priority queue –but since all the elements that needs to be sorted are not known at the beginning of the simulation (since some events will usually be added during simulation), we need to add one requirement; that the sorting algorithm does not need to resort all elements, if just one more element is added. Quick-sort and merge-sort are examples of common sorting algorithms which does *not* have this property.

If we accept this proposal, then we may view discrete event simulation as "sorting and execution of events". We can think of the linked-list based timeline as driven by an insertion sort, and the heap-based timeline as driven by a heap sort.

We note that the time complexity for the execution of n events in a discrete event simulation cannot be done quicker than the fastest sorting algorithm. According to the proof given by [Cormen et al., 1990] general sorting by comparison cannot be done faster than $O(n \log n)$. In this light, the heap implementation is optimal.

But, as also stated in the above reference, this lower bound on sorting can be improved if it is possible to make further assumptions about the input elements (and in the case of discrete event simulation, when they will occur). Interestingly enough, this may very well be the case in certain types of discrete event simulations. For instance, if one can assume that the largest and smallest element is known to begin with (that is the activation time of the first and the last Event), then it is possible to sort in $O(n)$ time using for instance bucket-sort. If needed, there should be no problems involved with implement an priority queue based on this sorting algorithm in our framework.

Recycling the events

One of the drawbacks of using objects (and object oriented programming as opposed to structured programming) is that objects take time to initialize, and when they are discarded, their resources need to be given back to the system. Often when dealing with many "small" objects, the time it takes to clean up after them (in Java known as "garbage collect"), may severely decrease system performance. If one therefore were to keep a record of terminated events (and thereby making then uneligable for garbage collect), these events could instead be reused and one would save computing time. This would of course require a bit more memory, but should have impact on performance, depending on the number of events and objects used in the simulation.

4.4 A few words about implementation

When implementing the classes used in this thesis, we have made extensive use of abstract data types (ADT). This has been done to ensure modularity and flexibility in the design. For instance, this allows us to freely change the details of the timeline implementation, without having to alter any other code. This is a great advantage that both shortens development time and keeps the design of the program simple.

The ADT's themselves, has been implemented from scratch in Java, in the bergmann.structure.* package. This package is loosely based on the ideas and examples presented in [Goodrich and Tamassia, 1998] and [Cormen et al., 1990]. Implementing ADTs may seem like a waste of time given the large number of free and commercial implementations available, but we have chosen to do so for several reasons:

1. Simplicity. The package provides an easy and understandable interface to all its functionality.

2. Readability. As follows from the previous item, a simple interface to the functionality makes the the code that uses it simpler to use, which is highly desirable in a thesis.

3. Extensibility. Having complete and free access to all the source code of the package is a great advantage. This allows us, among other things, to add more or change functionality if needed.

4. Interchangeability. Because the package focus on simplicity, it provides functionality that most other packages provides. If one needed to, it

would be simple to migrate to another ADT collection, since they would most certainly also provide all of the functionality of the basic ADTs.

5. Completeness. Although effort has been put on making the interfaces as easy to use as possible, several advanced structures such as several dictionaries, graphs and trees have been included, which are rarely a part of more commercial packages.

6. Uniformness. For reasons of complexity it is an advantage to limit the number of different packages used.

Using a good collection of ADTs is often a good way to ensure quick development, avoid program complexity and reduce the size of the overall program. Although we have chosen to use the bergmann.structure.* package, there are a great number of other ADT collection available that could also be used. ADTs are often a matter of personal taste. Which one you choose, should at least in theory, not matter.

4.5 Other simulation frameworks in Java

Although we have chosen to implement our simulation framework from scratch, there exists several Java based discrete simulation environments. They are mostly Java versions of existing simulation languages.

Simjava Simjava is a process based discrete event simulation package for building models of complex systems. It includes some graphical capabilities, which for instance allows simulation entities to be displayed on the screen as animated icons or "live diagrams" in web documents. The package has been designed for simulating fairly static networks of active entities which communicate by sending event objects via ports and is therefore appropriate for hardware and distributive software system modelling. Each of the simulation entities runs in their own thread, and a central system class controls all threads, advances simulation time and delivers the events from object to object. www.dcs.ed.ac.uk/home/hase/simjava

Silk Silk is a commercially available general-purpose simulation language based around a process-interaction approach and implemented in Java. Silk is designed as a tool for building self-contained, reusable modelling components and domain specific simulations. It provides a visual modelling environment where Silk-based modelling components can be graphically assembled using JavaBeans to create simulation applications in software environments such as Symantec's Visual Cafê, IBM's VisualAge and Microsoft's J++. www.threadtec.com

JavaSim JavaSim is a component-based, compositional simulation environment. It has been built upon the notion of the autonomous component programming model similar to COM/COM+, JavaBeans, or CORBA. This packages special architecture allows simulations to be interactively defined using a terminal environment similar to the Unix/Linux command prompt. Especially for large network simulations the JavaSim simulation environment has proven scalable both in terms of memory consumption and running time. www.javasim.org

One of the top most common uses for discrete event simulation today is network simulations. This is to a large degree also evident in many of the frameworks available for discrete event simulation. These frameworks often introduce abstractions such as "ports" on each of the simulation entities, and use events to pass information between ports on different entities. Such an approach is ideal for modelling networks of entities, and their communication patterns. Often the simulation frameworks are designed to handle multi-threading and each entity operates in its own thread.

Since there already exist powerful and specialized simulation environments, we have opted for the creation of a more general and "open" framework. One example of the generality of our simulation framework is how we represent time in the timeline. Most implementations we have examined represent time as a java.lang.double. Although a double value should be good enough for most practical applications, we have chosen to represent time by a general java.lang.Object. Our representation allows, in addition to representing time as a double, that time can be *any* java.lang.Object. It can be a java.util.Date (24/3-49, 28/7-77, etc), a java.math.BigInteger (integer larger than java.lang.Long.MAX_VALUE), or even a general java.lang.String ("Jan", "Feb", etc) as long as the user also provides a way to compare two object of the implemented type. The use of an Object rather than a double value also allows direct use of any standard PriorityQueue implementation.

Another difference between our framework, and many of the existing frameworks is that we have not included thread-support. Although it would be relatively simple to extend the framework to also be thread-compatible, we have chosen not to do so since it would incur an unnecessary overhead associated with handling the thread (and the infamously slow "synchronized" keyword) in simulations which does not need threads (such as the ones we have implemented in this thesis). Instead we allow our framework to be extended with thread-safe wrapper classes, if thread-safety is important. This technique is also used by Sun Microsystems in their implementation of the Collection package: the collections themselves are not thread-safe since that would incur an overhead if thread-safety is not needed, but there exists special wrapper classes which are thread-safe and "wraps" around the collections. The wrapper class provides the same functionality as the collection class (actually it is uses an instance of the collection) but also implements thread-safety in an appropriate manner.

A final note is that we would not like our base simulation package to become too large. According to Jacobson [Jacobson et al., 1999] one should always be careful before adding functionality which "might be nice to have" but is currently not really needed. One should bear in mind that more classes usually means more complexity and that it is usually a lot simpler to add functionality later on when it is needed than it is to remove uneccessary features, since removing something from a package often will break compability. It is a good general rule to be conservative about what to add to a package.

We aim to create a general framework, and believe it to be wrong to favor many features and complexity over simplicity and elegance in our implementation.

If more functionality really is needed, then we believe it would be better to create a new package and import and extend the bergmann.simulation framework. This way the packages becomes small, manageable and easier to learn and use. In the next part we will do exactly this when we create a package especially adapted to modelling a simulated rat in a cage. But first we will test our basic simulation package on a classical example.

Chapter 5

A comprehensive example

We will now examine the problem presented in Section 3.1 more closely. In addition to the problem specified there, we will also wish to test out different strategies for the elevator to use, so that one has more control over how the elevator moves. This could for instance be to give certain floors priority at certain times of the day, if one in advance knew that this floor would be especially crowded a that time.

For instance: In an office building in the morning, one would expect more people to come to work than leaving. Therefore we believe that it would be advantageous if the elevator went to the floor with the main entrance when empty, so that the next people to arrive at work did not have to wait for the elevator to arrive. We will in this chapter create a strategy which will handle this.

5.1 The elevator simulation

In order for us to take advantage of the code defined in the Simulation superclass, we choose to extend it, creating subclass ElevatorSimulation. This will ensure that the ElevatorSimulation class will have easy access to the Timeline.

The Elevator will most certainly be an important abstraction. The key property with an Elevator is that it will also be needing some choice-making-capabilities. We therefore find it most reasonable to extend Subject.

The other objects in the simulation (House and Person) are implemented and stored as objects in the Context. The choice of not making Person extend Subject is perhaps not obvious (since we are used to thinking in terms of relatively intelligent people) but in this simulation (where they may just as well have been boxes or cars), no intelligence is needed. If, however, the problem specification changed to also include that a person might grow tired of waiting after a certain time and take the stairs instead, then this would probably have to be changed.

We have decided to inherit from Strategy, and to create a common ElevatorStrategy. All implemented strategies in this simulation will in turn inherit from ElevatorStrategy. This is done for flexibility and to ensure that all strategies are uniform and adjusted to this simulation. Also, functionality can more easily be added to all of the strategies simultaneously this way.

As for events, all events inherit from Event directly. We have decided to create just a few events in order to ensure simplicity and readability. These are the PersonEvent, for when a person enters an Elevator, and PressButtonEvent, for when a Person presses a button (regardless of whether the button is outside or inside the elevator).

5.2 The protocol

In this simulation a number of people will move between floors using the elevator. We assume that all people and all strategies participating in the simulation adopts the following protocol.

1. Person P_i in floor j presses the "come to this floor" elevator button. The person logs the current time.
2. The elevator registers this request, and takes necessary actions.
3. Upon reaching floor j the elevator stops, and person P_i enters.
4. Person P_i pushes the button to go to floor k.
5. The elevator registers this request, and takes necessary actions.
6. The elevator reaches floor k. Person P_i exits the elevator.
7. Person P_i logs the current time.

The difference between the logged time will be used to calculate the average waiting time for each floor. It follows from the above protocol that the waiting time for a person moving between floors will be

$$\begin{aligned} T_{wait} &= 2 \cdot T_{enter\&leave} + (a+b) \cdot T_{between} + c \cdot T_{enter\&leave} \\ &= (a+b) \cdot T_{between} + (2+c) \cdot T_{enter\&leave} \end{aligned}$$

where $T_{enter\&leave}$ is the time the elevator stops in each floor, $T_{between}$ is the time it takes for the elevator to move one floor up or down. a is the number of floors person P_i wants to move, b is the number of floors the elevator has to move to get to floor j where person P_i is and c is the number of stops the elevator makes.

We here assume that the elevator travels at the same speed going up, as going down, regardless of how many people are in the elevator, and that it always takes the same amount of time to stop in each floor, regardless of how many people wants to enter or leave.

5.3 Choice of strategy

There are many possible strategies that can be implemented by an elevator, using the protocol from the previous section. We have here chosen to present a tour of a few significally different strategies. Their performance analysis will be discussed in the next section.

`UpDownStrategy` This strategy is based on the "pater noster" elevator. Move continuously from the first to the top floor and down again. For each floor you pass: if someone wants on or off, let them do so.

`ReturnStrategy` This strategy gives special priority to a permanent predetermined floor, called floor k. The elevator starts in floor k. It lets people on and registers to what floor they want to go to. It then visits those floors in a suitable order. Then the elevator returns to floor k. If new travelers entered while the elevators visited the other floors, the elevator transports them to their destination *after* floor k has been visited. If the elevator in floor k is empty, then the elevator picks up travelers from another floor and delivers them, before returning to floor k.

`NormalStrategy` Register all buttons that are pressed (clicks) in a "need-to-visit-floor" table. The indexes of this table refers to the floor where the buttons has been pressed. Then move to the floor associated with the largest index in the table that has been pressed, halting underway in floors, if needed. Then perform the same procedure moving downwards, to the floor associated with the smallest pressed button index in the table. Unmark each floor as they are visited in the table. If the table contains no clicks, then stop.

`EmptyReturnStrategy` This is an minor change to the NormalStrategy. It behaves the same way, but upon finding no more clicks in the table, returns to a predetermined floor, and stops there.

5.4 Results and analysis

In our simulations we have used that the travel time between floors (T_{wait}) is 2 time-units, and that the amount of time stopped in each floor ($T_{enter\&leave}$) is 5 time-units.

Using these constants and the strategies described in Section 5.3 produced the results presented in figures 5.1 and 5.2.

In fig 5.1 we have simulated 100 000 people inside a 10 floor building. The people has been uniformly distributed over all 10 floors. Over a time frame of approximately 10 000 000 time-units all people will travel from their starting floor to a random destination floor. The starting time of each person is chosen at random inside the specified simulated time-frame.

We let the elevator use one of the strategies described in the previous section, and we measure the average simulation time from when a person first presses the "come-to-this-floor" button to when he arrives in his destination floor.

It is possible to validate the results in fig 5.1 using the formula presented i Section 5.2. Using that $T_{enter\&leave} = 5$, $T_{between} = 2$, that the person wants to travel from first to sixth floor ($a = 5$) and that (now assuming NormalStrategy) on the average, the Elevator will be on the fifth floor ($b = 4$) whenever the person wants to use it. We also assume that the elevator makes no stops other than the ones for the person in question ($c = 0$). This gives us the following result:

$$\begin{aligned} T_{wait} &= (a+b) \cdot T_{between} + (2+c) \cdot T_{enter\&leave} \\ &= (5+4) \cdot 2 + (2+0) \cdot 5 \\ &= 28 \end{aligned}$$

Comparing this to the result obtained in fig 5.1 we see that our simulations are about right.

Under the same assumptions it is also possible to calculate the average number of people inside the elevator. We assume that in a building with 10 floors the average person using the elevator travels 5 floors. From our simulation and our calculated estimate we have concluded that the average time it takes to travel 5 floor with the elevator is 28 time-units. Therefore, on the average, each person will spend 28 time-units in the elevator. There are 100 000 who wants to use the elevator, so in 10 000 000 time-units each person has $\frac{10\ 000\ 000}{100\ 000} = 100$ time-units per person. Only 28 time-units out of these will actually be used, so there will be people inside the elevator 28% of the time.

Figure 5.2 shows the results from a similar simulation. We have again simulated 100 000 people inside a 10 floor building, but this time the simulation will run over only 1 000 000 time-units. Conversely this means that each person will have only $\frac{1}{10}$th of the time they had in to get to their destination compared to the previous simulation.

100 000 people are again uniformly distributed over the 10 floors in the building, and each one of them will at a random point in time start their trip towards a predetermined floor. Fig 5.2 shows the average time from each person presses the "come-to-this-floor" button to when he arrives in his destination floor.

We observe that if 100 000 people are to use the elevator over 1 000 000 time-units, then each person will one the average have only $\frac{1\ 000\ 000}{100\ 000} = 10$ time-units to use the elevator by himself. From our simulation we see that it

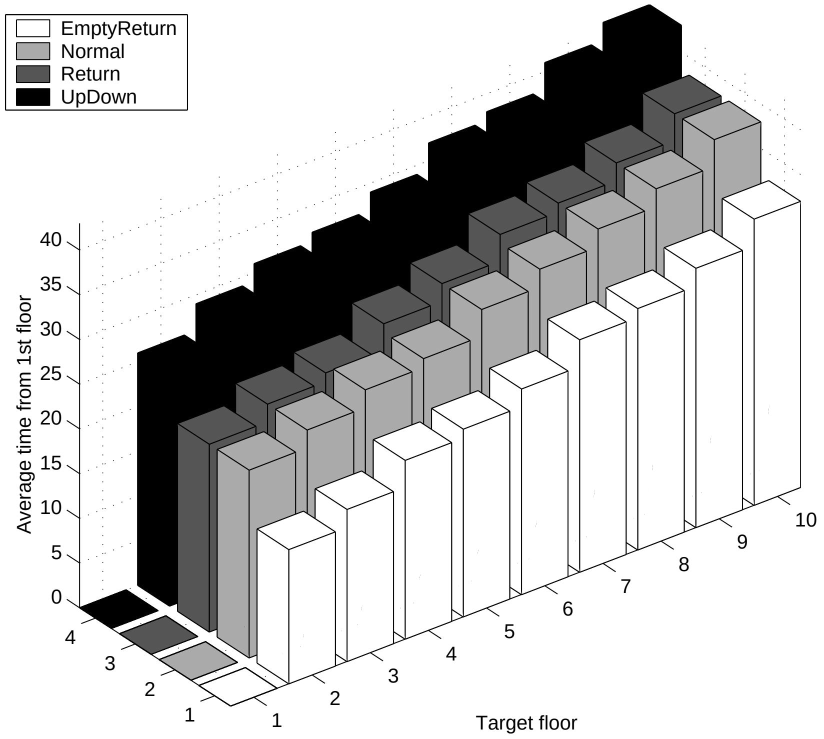

Figure 5.1: Graph of average waiting time from first floor to all the other floors. We see that it takes about 28 time-units to travel to the fifth floor with the NormalStrategy.

takes about 57 time-units to travel 5 floors when the elevator is this crowded. This implies that there on the average is $\frac{57 \cdot 100\ 000}{1\ 000\ 000} = 5.7$ people inside the elevator. We note that the elevator is a lot more crowded in this scenario. Reducing the available time to $\frac{1}{10}$th leads to about 20 times more people inside the elevator per time-unit.

As a curiosity we might mention that the average running time for each strategy was a little less than one hour[1], and that each run generated about 45 MB of output data[2]. Actually, the sample running time of different time-line implementations shown in figure 4.3 were interpolated from several runs of this simulation of various sizes.

As expected we see that the simple UpDown strategy usually results in longer average waiting-time compared to the other strategies, unless the

[1]On a SUN Microsystems Ultra SPARC 10 Creator 3D computer.

[2]Generating a grand total of about 300 MB of raw data in a little less than 8 hours of computing time. Fortunately all the simulations of the different strategies are independent of each other, and may be run in parallel if multiple computers are available.

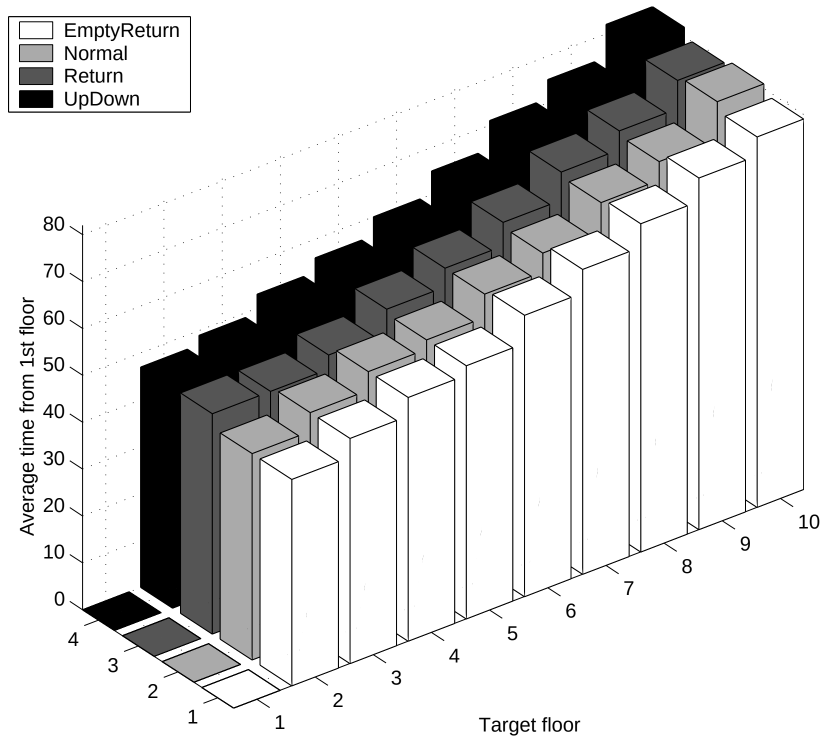

Figure 5.2: Graph of average waiting time from first floor to all the other floors. The same amount of people as in fig 5.1 is using the elevator in $\frac{1}{10}$th of the time. This means that there generally will be more than one person inside the elevator.

elevator is very crowded. Then it performs similarly to the other strategies. The ReturnStrategy seems to give low waiting-time in figure 5.1 and 5.2, but its strong priority of the ground floor results in longer waiting-time for the other floors (not shown in the figure). As expected, the NormalStrategy will ensure lower waiting-time, if no particular floor is given priority. The EmptyReturn strategy, always favoring the first floor (although never at the expense of the other floors), has the lowest average waiting time out of the strategies we have implemented in our simulation.

Although it is hard to determine a strategy that will perform optimally under all conditions we nevertheless expect, as our simulation has shown, that the Normal and EmptyReturn strategy generally will provide low waiting-time. Their average simulation waiting time is shown in figure 5.1 and 5.2.

Part II

An extension for rodent navigation simulation

In the first part of thesis, we created a general framework for discrete event simulation. Now, in Part II, we will extend[3] this general framework to create a framework which is specifically designed to handle rat exploration and navigation. The rest of this thesis will primarily describe the simulations performed in this framework.

Note that we have already introduced a simplified version of the rat simulation in Chapter 3.

In the first chapter in Part II we investigate the simulation entities needed in the rat simulation, and show how they will be implemented. We describe each of the entities, and some of the events that will be needed in this simulation.

The Rat will during exploration of the cage need to get stimuli from the environment in the form of touch, smell and vision. How these inputs are created is described in Chapter 7.

In Chapter 8 we will test out the framework for rat simulation we have created so far. We will develop basic strategies for moving the rat around the cage, and show plots of the output of the simulations. We hope that this chapter will help to convince the reader that the framework and stimuli generated by the extended framework are sound, and that small simulations in the framework provide reasonable results.

[3]We use the term "to extend" to describe the process of redefining old or adding new functionality to a package. Adding such functionality augments and enhances the original package. In this thesis we do this by introducing problem specific classes which specializes the original functionality of the framework. This is not to say that there was anything wrong with the original package, in fact we believe that it was correct enough to be *reused* in our extension. As stated in Section 2.2.1 reuse can be used to significantly reduce both complexity, development-time and development-cost in computer software.

In the context of Java programming, the keyword *extends* denotes inheritance or subclassing of another class.

Chapter 6

Adaptation of the framework

In this chapter we will attempt to use the general framework defined in Part I for creating a simulation as described in Chapter 3.2. Being able to use the same framework is a great advantage since we have already tested it (with the elevator simulation) and we therefore know that it works. Additionally, the framework has, from testing with elevator problems of different sizes proved itself to be quite scalable, with a good choice of priority queue. This is important since we don't know in advance how large the rat simulation will become.

6.1 Abstractions

As with the elevator-simulation in Chapter 5, we start by creating a subclass of the Simulation superclass. This will allow us to use the functionality defined there, and to extend it with more functionality which is specific to the rat simulation. For instance, the new RatSimulation class will probably be the best place to add support for a graphical user interface (GUI), since this class will be used to start, stop and pause the simulation.

Our next, and probably most interesting abstraction, is that of the *simrat* or simply *Rat*. The simrat will move around in its surroundings, and receive input from its environment. Since this object will need some guiding strategy in order to determine where to move, we choose to extend Subject. The notion of a Strategy object that controls how the Subject move also fits well with an abstraction of a "brain-object", which guides the Rat based on input. By encapsulating the decision-making into a single object, we can easily change how Rat behaves and responds to various input.

The details of the input will be defined later, and we are now only concerned with what types of input we would need. Both humans and rats share the same basic 5 types of input: Vision, smell, touch, hearing and taste. Out of these five, we assume that our strategies for navigation can be done successfully with only three of these: Vision, smell and touch, and therefore choose to ignore hearing and taste for now.

As suggested in Section 3.2.1, we need some sort of abstraction that will handle the environment the rat will move in. We extend the Context superclass from the framework, and to create a Cage abstraction. This allows us to reuse all functionality related to the general simulation framework while we can define new functionality which handles the simrat specific parts of the simulation.

The Cage class will be responsible for keeping track of where the Rat is at all times, that the Rat does not perform any illegal moves while in the cage (such as for instance walking through a wall, or outside the cage) and to keep a record of properties forced upon the environment (such as smell and visits per grid-point inside the cage). The Cage abstraction will also be important when determining what input the simrat should receive.

6.2 The model

6.2.1 Cage

The cage is a very important abstraction in our simulation. We choose to represent the cage as a two dimensional rectangular grid.

We let each grid-point (x, y) in the cage (also referred to as a *position*) hold a value. The values we allow for each position are one of the following: " " (a space, representing an empty position), "Start" (s), "Goal" (x), "Wall" (#) or a letter a through j, representing landmarks. For convenience we allow the position of all these values in the cage to be read from a text-file. A sample text-file can be seen in figure 6.1. The value of position (x, y) is returned by the *element (x,y)* method in the Cage-class.

Start and goal

The position in the map marked with "s" is the position in which the rat will be in at the beginning of the simulation. We will always assume that the rat is facing in the north direction at the start of a simulation.

The position marked with "x" is the place the rat should try to return to in the path integration phase. More on this later.

Obstacles

The obstacles, or walls in the cage, are marked with #'s in the map. It should be impossible for a simrat to visit a position marked as a wall, or to, from input alone, determine what lies directly behind a wall.

```
                               e
    d                                                          g









                     c                     f
                       ##################
                       ##################
                       ##################
                       ##################
                       ##################
                       ##################
                       ##################                      h
                       ##################
  b                    ##################
                       ##################
                       ##################
                       ##################
                       ##################
                       ##################
                       ##################
                       ##################   i
                       ##################
          a           x##################             s
```

Figure 6.1: Input file which defines a cage with obstacles, landmarks, start and goal positions.

Landmarks

Landmarks are in the map marked with the letters *a* through *j*. Each landmark has a distinct position in the cage, and for each position in the cage there is a most one landmark. These landmarks are the basic components which will be used for navigation.

Rat

The cage will keep a record of the current position and direction of the Rat in the cage. We only allow the simrat to face in the north, east, south or west direction. That we only allow the simrat to make turns of ± 90 degrees corresponds nicely with our implementation of the cage as a rectangular grid.

See next section for more information about the Rat.

6.2.2 Rat

There are two properties of the Rat that are important. The first is the opportunity to process input (from the environment), and the second is the ability to make choices based on those inputs. In this respect the Rat is not very different from the Elevator defined in the ElevatorSimulation.

However, it is important to remember that the Strategies that will guide the Rat are very different. After all it is the strategies that perform all of the analysis of the input, and that actually decides what the simrat should do.

Strategies

Each of the strategies implemented attempts to capture some specific aspect of the simrat. The aspects we would like to model can be anything from relatively simple behavior (such as the simrat always turning left at corners) to more complex (as is the case if we want the simrat to create an internal representation of the environment from input). We hope the framework is general enough to let the strategies implement just about any type of behavior that we could want to model.

The strategies must be able to handle the inputs that the framework will give. The strategies must also return well-defined values to indicate what moves the strategy recommends.

Handling input As for input we assume that a strategy will receive input of three different types ("vision", "smell" and "touch"). Note that the strategy will *not* receive its current (x, y) position in the cage as input. All information about the position of the rat in the cage will have to be deduced from the three types of input we allow. This is more difficult because there is the possibility that simrat misinterprets the input and believes it is in another position than it actually is. The exact details of the inputs will be discussed later.

Giving output The values that we allow the *recommendMove ()* method to the Strategy class to return should be one of the following:

MOVE_NORTH	FACE_NORTH	TURN_LEFT
MOVE_SOUTH	FACE_SOUTH	TURN_RIGHT
MOVE_EAST	FACE_EAST	MOVE_FORWARD
MOVE_WEST	FACE_WEST	NO_MOVE

Most of these legal moves should be self-explanatory, except perhaps for the TURN_LEFT and TURN_RIGHT return values. Both of these will take into consideration the direction the rat is currently facing, and then change the heading ± 90 degrees relative to that. Similarly, the MOVE_FORWARD takes into consideration the direction the rat is facing, and requests that the rat be moved in that direction.

Note however that the values returned by the *recommendMove ()* method to the environment are only requests. If the move requested is illegal for some reason (a move in the northern direction would force the rat through a wall), then that request will be ignored. In such a situation, the strategy should be robust enough to recognize the situation so that when it receives input the following time and finds that it has not moved, it should recommend a *different move.*

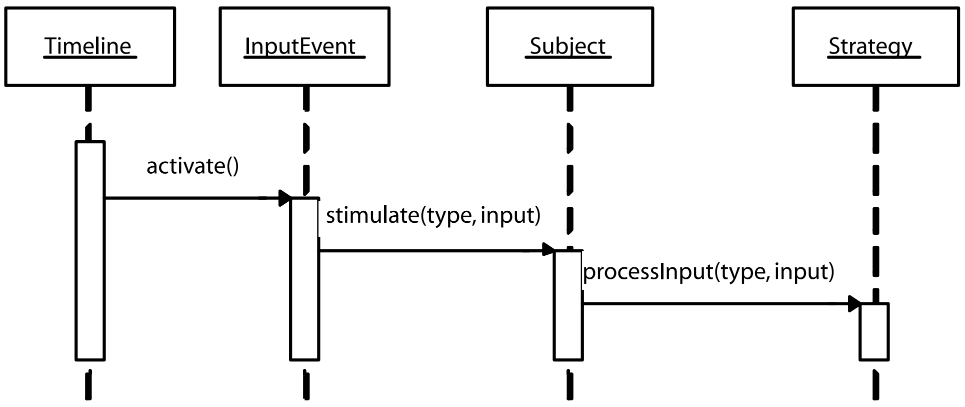

Figure 6.2: The interaction pattern for the InputEvent.

6.3 The events

We have now defined the two basic components in the simulation: the Rat and the Cage. Next we need a way to pass information between the two objects. Since we would like all information to pass through a common interface and to be generated in a standard way, we define Events to handle this task. This has the added bonus that if we wanted to add support for other types of inputs, the interface between the two objects would not have to be changed, only more Events added.

6.3.1 The InputEvents

We call all of the events that carries information from the environment to the simrat InputEvents. For each of the types of input we allow, we create one InputEvent: *VisionEvent*, *SmellEvent* and *TouchEvent*.

Each of these InputEvents will examine the state of the cage, and in turn pass selected pieces of information on to the Rat as input. The details of what information we will give to the rat for each of the three types of input is covered in the next chapter. The interaction pattern for all the InputEvents is the same, and shown in figure 6.2. Since we assume that all the input will happen at the same time and it may be convenient to have precise control over the exact order in which the input events will be activated, we choose to create a MultiEvent, as described in Section 4.1.2. This MultiEvent will, upon its activation, activate each of the input-events in a predetermined, sequential order.

6.3.2 The MoveEvent

After the simrat has been given input, the MoveEvent should be activated, to query the simrat what to do, or more specifically: where to move. The

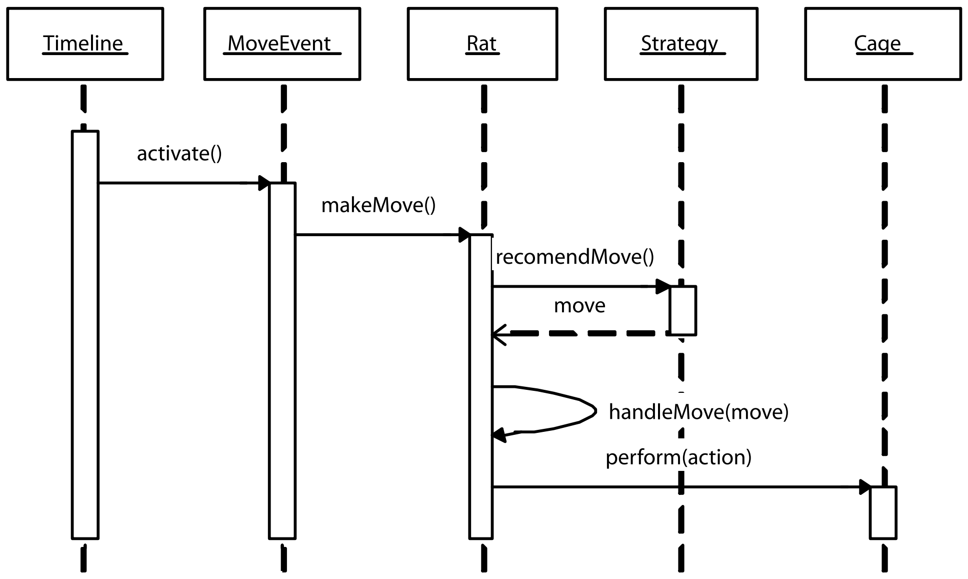

Figure 6.3: The interaction pattern for the MoveEvent.

simrat will now pass this request on to its Strategy-object, which then will make a decision. This decision will then be sent back to the simrat, which will further pass it back to the MoveEvent. The MoveEvent will now give this information to the Cage, which will alter it's state to reflect the move. The interaction pattern for the MoveEvent is shown in figure 6.3.

6.3.3 Reoccurence

An important feature of the RatSimulation is that the events will re-occur at certain intervals. First an InputEvent should be activated, to inform the simrat of the current state of the environment, and then the MoveEvent should be activated to cause the simrat to move. Next we would like new input to be given to the rat, a new decision to be made, and so on. This type of repetitive pattern can be efficiently handled by a ReoccuringEvent, as described in Section 4.1.2.

6.3.4 Handling speed

One of the strategies we will later discuss will need some way to represent speed in our framework, so that it can model a rat's *speed-distance* profile (see Chapter 11 for more information).

In our framework, we will model speed by manipulating the Reoccuring-Event, which in turn controls the MoveEvents.

Speed is defined as "change in position" per "time" ($\frac{\delta u}{\delta t}$). We observe that if we change the time between each MoveEvent, then the speed will also change (assuming that the strategy actually does recommend a move and not just recommends to stand still). So if we reduce the time between MoveEvents, then speed will increase, and if we let the MoveEvent be further apart in time, speed decreases.

For simplicity, we will only allow speed to assume 10 discrete values, 1 through 10. Speed 0 is not handled in our framework, but can be achieved by having the strategy recommend "NO_MOVE". Speed 1 means that the MoveEvent will become activated every 10th time unit. Speed 10 means that MoveEvent is activated once per time unit.

In our implementation we have handled this by creating a special Speed-Control object which the strategy has access to. The strategy then specifies which speeds it wants to have to the SpeedController (speed 1 through 10), and the SpeedController modifies the interval between each activation of an MoveEvent, by altering the ReoccuringEvents interval.

Chapter 7

Defining input

All of the types of input we would like to give to the Rat will depend on the current position of the Rat in the cage. Some inputs will also depend on the orientation of the Rat. Upon activation the InputEvents will examine the Cage, and call the *stimulate (type, input)* method of the Rat. The "type" parameter is simply which type of input we are giving (either "TOUCH", "SMELL" or "VISION"), and the "*input* parameter is the actual input. In the next sections we will examine more closely what this input, that the events should generate and give to the Rat, actually is.

7.1 Touch

Touch is probably the most basic type of input we have in our model. If the Rat is in position (x, y) then the input this InputEvent will give is the character determined by the cage method *element (x,y)*. Specifically; if the Rat is standing on landmark a, then the input given to the Rat is a. If no particular value is stored in this position (the map in figure 6.1 contains a space), then 0 is given as input.

7.2 Smell

Smell is a type of input which will become important when the Rat explore the environment. As with touch, the smell value is a value associated with a (x, y) coordinate in the cage. Initially the smell-value of each position in the cage is 0. This value can never become less than 0. Each time the Rat visits a position in the cage, the smell-value associated with that position is incremented by S_{increase}.

At regular intervals, the smell-value of all positions in the cage should be reduced with S_{decrease}. This is done with a DecreaseSmellEvent, which upon activation will subtract S_{decrease} from all smell-values in the cage, or set it to 0 if the resulting smell-value of a position becomes less than 0.

The Rat should give as input the smell-value of the position the Rat is currently in, the position in front of it, behind it, to the left and to the right of the Rat. This input can be used to determine where the Rat has been before.

7.3 Vision

Vision is one of the more advanced, and computationally heavy inputs to generate. The vision input should return the visible cage-positions, relative to the Rat's current position. By *visible* cage positions we refere to the positions in the cage that are inside the Rat's *vision radius* and are not behind any obstacles (i.e. there exist a *direct line of sight* to the position from the Rat). That we are interested in the *relative* position of the Rat, means that we should not be given the absolute position in the cage of the landmarks, but rather should express all landmark positions in the cage with the Rat's current position as the origin, and a predetermined direction as reference. This is so that the Rat cannot obtain global information (such as a landmarks position relative to a global origin) simply by looking at local information. Instead, all global information will have to be deduced from information learned about the relationship between various local landmarks. This is more difficult than being given global coordinates explicitly.

7.3.1 The problem defined

Let's assume that the Rat's current position is s in a cage containing obstacles (#'s) and landmarks (a and b). We now wish the VisionEvent to determine which of the positions in the cage are visible (is on a *direct line of sight* from the Rat), and which are not.

Finding candidates The first thing the VisionEvent should do is discard all points in the cage that are too far away from the current position of the Rat (i.e. outside of the vision-radius). This can be done by examining the Euclidean distance between each position in the cage, and the Rat's position. If this distance is less than the vision-radius, then this position is a candidate, and it must be stored for later analysis. Otherwise it can be discarded. If the cage is large, then this simple step can often prevent much unnecessary computation.

We now have a set of candidate position that may or may not be visible inside the cage from the Rat's current position. Therefore, the next thing we need to calculate is if each candidate position is hidden behind some obstacle or not.

Representation This problem can be solved by considering lines and intersection between lines in a plane. We start by transforming the map de-

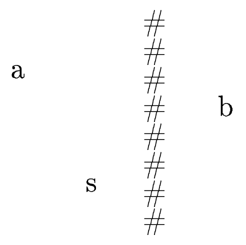

Figure 7.1: The map describing the scene

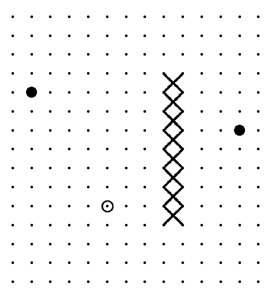

Figure 7.2: The point/line-representation of the scene

scribing the scene (figure 7.1) into points and lines in a plane (figure 7.2) in the following manner:

The positions We represent each position, (x, y) in the map, as a midpoint $(x + 0.5, y + 0.5)$ in the plane.

The obstacles We represent an obstacle in position (x, y) in the map, as a pair of line segment in the plane

1. the line in the plane from (x, y) to $(x + 1, y + 1)$
2. the line in the plane from $(x, y + 1)$ to $(x + 1, y)$

It is now possible to check if a direct line of sight exists from the position of the Rat to any candidate in the map, by checking whether the line from the Rat to the candidate intersects any of the obstacles. If there exist an intersection, then no direct line of sight exist.

Finding intersecting lines If we have two lines, ab and cd, both defined by their endpoints (a, b and c, d respectively, ordered by increasing x-value), then we can check for intersection between them by solving the following equation for the variables s and t

$$\begin{bmatrix}(a-b) & (d-c)\end{bmatrix}\begin{bmatrix}s\\t\end{bmatrix} = \begin{bmatrix}d-b\end{bmatrix}$$

Since we are only dealing with two dimensions (all points are described by a x and y coordinate) this equation turns into the following set of linear equations:

$$\begin{bmatrix}(a_x-b_x) & (d_x-c_x)\\(a_y-b_y) & (d_y-c_y)\end{bmatrix}\begin{bmatrix}s\\t\end{bmatrix} = \begin{bmatrix}d_x-b_x\\d_y-b_y\end{bmatrix}$$

This set of equations can be solved directly by substitution[1]. If the values of s and t both are in the range [0, 1], then the lines intersect.

Unfortunately, this set of equations does not necessarily have a solution. If the lines are parallel or collinear, the system becomes singular and other tests are needed; If the slope of the two lines are the same (i.e. they are parallel), then we should not use the equations given above directly. Now the two lines will fall into one of two categories:

Parallel and not collinear If so, then the lines do not intersect. This situation can be checked by testing whether the line from b to c also has the same slope as that of ab and cd. If so, then the lines are collinear and we must do more testing. Otherwise, they do not intersect.

Collinear If the lines are collinear then the lines intersect only if they overlap. They overlap only if any of the following is true:

- the point a or b lie on the line cd
- the point c or d lie on the line ab

Both of these cases can be tested by using the procedure described above (by testing if the line aa or bb lie on the line cd and vice versa).

7.3.2 From global to local coordinates

We now have a set of positions in the cage. With each position (x, y) in this set we should associate the touch value of position (x, y) in the cage. This gives us a new set containing both positions and their associated touch values.

Each position in the set is represented by *global coordinates*. This means that the coordinates of each positions is expressed relative to a global origin, i.e. $(0, 0)$. If we were to give this input to the Rat, then we would also have to supply the Rat with its position in global coordinates in order to position it relative to the input. Otherwise it would for instance not be able to tell what is in front of, or behind it. When we choose *not* to do this, it is because we would like to implement a biological system. It is not believed that a real rat operates by using a global coordinate system for navigation. We find it more plausible that real rats operate by observing distances, and angles between landmarks. This corresponds to a polar coordinate system, with the Rat as origin. But since there is a one-to-one correspondence between Cartesian coordinates and polar coordinates, and we have already used a Cartesian system in our implementation of the cage, we choose to give input in the form of Cartesian coordinates, with the Rat as origin. If needed,

[1]We have also implemented a Gauss-Jordan equation solver which can be used for this and larger sets of equations in bergmann.numerical.GaussJordan

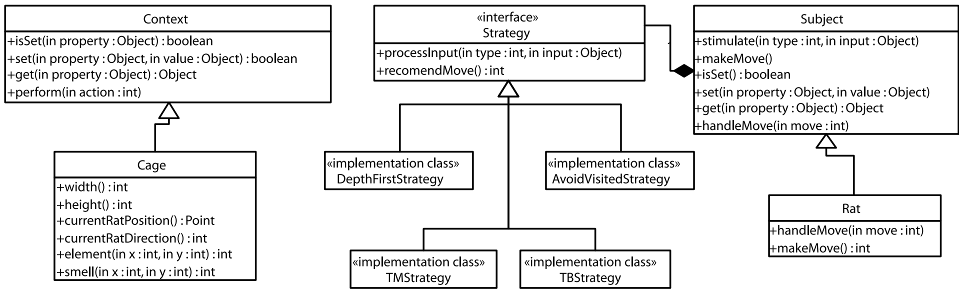
Context
+isSet(in property : Object) : boolean
+set(in property : Object, in value : Object) : boolean
+get(in property : Object) : Object
+perform(in action : int)
Cage
+width() : int
+height() : int
+currentRatPosition() : Point
+currentRatDirection() : int
+element(in x : int, in y : int) : int
+smell(in x : int, in y : int) : int
«interface»
Strategy
+processInput(in type : int, in input : Object)
+recomendMove() : int
«implementation class»
DepthFirstStrategy
«implementation class»
AvoidVisitedStrategy
«implementation class»
TMStrategy
«implementation class»
TBStrategy
Subject
+stimulate(in type : int, in input : Object)
+makeMove()
+isSet() : boolean
+set(in property : Object, in value : Object)
+get(in property : Object) : Object
+handleMove(in move : int)
Rat
+handleMove(in move : int)
+makeMove() : int

Figure 7.3: A class diagram showing the relationship between the various components in the rat extension of the framework.

the conversion from Cartesian to polar coordinates could be implemented trivially in the Rat.

After having found the set of all positions in the cage that are visible from the Rat's current position (with a touch value associated for each position) all positions in this set should be translated so that the Rat's current position becomes the origin in this set. If the Rat's current position is $(x_{\text{rat}}, y_{\text{rat}})$, then all positions (x_i, y_i) in the set should be set to $(x_i - x_{\text{rat}}, y_i - y_{\text{rat}})$.

This translated set of coordinates representing positions in the cage should be given as input to the Rat.

Note here that we are actually also giving the Rat the cardinal directions implicitly, since we for instance know that position $(0, 1)$ in the set is to the north of the Rat, position $(0, -1)$ is to the south etc. We could have solved this by rotating the positions in the set about the origin a random number of degrees, but since we later assume that the Rat is able to determine the cardinal directions from its vision input, we do not bother with this detail.

7.3.3 Efficiency considerations

Unfortunately the above procedure is not very efficient. It is essentially an $O(nm)$ algorithm, where n is the number of candidate positions in the cage and m is the number of obstacles [2]. In addition there is a relatively large constant involved with solving the linear equations (note however that this is only a constant since we only solve a set of two linear equations with two unknowns. This can be done in constant time).

[2] Actually there is some room for improvement here. We observe that it sometimes is possible to represent many neighboring obstacles as one large obstacle. Consider the cage in figure 6.1. It would be possible to consider the set of all walls (marked with "#") in that map as one large obstacle, defined by the #'s bounding box. Then, instead of then having to test for intersection between the line of sight and each of the lines in each of the walls, we could simply test for intersection between the line of sight and the four lines of the bounding obstacle box. However, it is not always fruitful searching for such shortcuts. In the worst case it is not possible to reduce the number of obstacles in the cage in this manner, and one will have to test for each one of them, as described earlier.

Fortunately there is room for optimization. Since we are dealing with a stationary environment, where the Rat will always see the same thing if it returns to the same position in the same map, it is therefore possible to *store* the input that should be given to the Rat every time the Rat returns to the same position. This way the above computation only has to be calculated once for each visitable position in each map.

By storing previously calculated results we were able to cut the running-time of the simulation down to a quarter of the original running-time.

Chapter 8

Evaluating the framework

8.1 Sample strategies

In this Section we will create a few basic strategies to guide our Rat. Each strategy is given the inputs defined in the previous sections but may or may not choose to use any one of them. The basic strategies will become the basis which we will later use for development of better and more advanced strategies.

8.1.1 Deterministic strategies

Deterministic strategies are the first group of strategies we now present. In the context of this thesis, we let the meaning of the term "deterministic strategy" be "a strategy that will not use any form of stochastic or pseudo-random criteria as a basis for choice-making". Conversely, this means that a deterministic strategy will generally produce the same result, if given the exact same input twice.

Depth-first strategy

This is an adaptation of the well known graph traversal algorithm by the same name, covered in e.g. [Cormen et al., 1990] and [Goodrich and Tamassia, 1998]. The basic idea of this algorithm is to systematically visit all of the visitable nodes in a graph exactly once, by keeping track of nodes that have previously been visited.

We have chosen to adopt the algorithm in the following manner: For each position visited in the cage, the Rat should always attempt to first move South, then East, then North and finally West, always disregarding positions that has been previously visited.

We also note that this strategy will eventually stop when it finds a position where all neighboring positions have been visited.

Avoid-visited strategy

Given that the Rat moves around in the Cage, and leaves a trail of smell-marks behind it, we can create a strategy that will attempt to avoid places it has previously visited.

For each point (referenced by an (x, y) pair) in the cage we associate a value, called the "smell" of this point. For each time-unit the Rat spends on this point, this smell-value is increased. We can then for each choice of which direction the Rat should move in, always choose to move in the direction where the "smell" value is the smallest. We hope that this over time will ensure that the Rat visits all points in the cage.

We also note that this strategy can be viewed as a discrete version of the well known "Method of steepest decent", covered in e.g. [Nocedal and Wright, 1999][1].

In the context of the RatSimulation framework we can use this algorithm for exploration of the environment.

8.1.2 Non-deterministic strategies

We name all strategies which will not necessarily generate the same output, given the same context and input, non-deterministic strategies. At the heart of these strategies often lies some sort of random number generator, which will somehow affect the choices made by the strategy. In the context of rat simulation, we may use the notion of randomness to account for impulsiveness, unpredictable behaviour or irrational choices made by the rat.

Probabilistic strategies

The probabilistic strategies (PS) can be formed by defining a set $\mathcal{S}$ of some strategies. The actual implementation of these strategies are irrelevant for the PS. A probability P_s is then associated with each strategy $s \in \mathcal{S}$.

When the PS is queried for what move the Rat should make, the PS should choose one of the strategies s in $\mathcal{S}$. The probability that s is chosen should be P_s. The PS then call the *recommendMove()* method of s and return the result to the Rat.

Input given to the PS should be forwarded to all of the strategies in $\mathcal{S}$ alike, so that all of them may give reasonable recommendation if they should later be chosen.

[1]There are of course some important differences. Firstly, the cost function we here attempt to minimize is determined by where the Rat has previously been, and will therefore change with time, as the Rat moves around in the Cage. Secondly, we note that the algorithm will never stop since if a point has been found which is currently optimal (that is, all smell values around it is greater) the Rat will stand still, causing the optimal value to increase, and eventually; no longer be optimal. This will then force the Rat to start moving again.

Now different types of behavior can be defined using the strategies $\mathcal{S}$ and different probabilities $P_s, s \in \mathcal{S}$.

Uniform The probability that strategy $s \in \mathcal{S}$ is chosen is $P_s = \frac{1}{|\mathcal{S}|}$. This means that every strategy has the same chance of being chosen.

Weighted This strategy assign each of the strategies $s \in \mathcal{S}$ a predetermined and constant weighted probability of P_s, so that $\sum_{s \in \mathcal{S}} P_s = 1$.

α-probability In some cases it is possible for each of the strategies in $\mathcal{S}$ to provide an indication of how "good" its recommendation is. Let α_s denote how "good" strategy $s \in \mathcal{S}$ rates itself. Then this strategy should have probability $P_s = \frac{\alpha_s}{\sum_{\hat{s} \in \mathcal{S}} \alpha_{\hat{s}}}$. This probability distribution will give those strategies which return high α-values higher probability of being chosen.

Vote Let each strategy $s \in \mathcal{S}$ recommend a move, and let the PS return the move that the most strategies recommends.

Sum Let each strategy $s \in \mathcal{S}$ recommend a direction to move, and let the PS return the vector-sum of these directions.

Weighted sum Let each strategy $s \in \mathcal{S}$ recommend a direction to move, and let the PS return the vector-sum of these directions weighted with each strategy's α-value.

Deviation The DeviationStrategy starts with one "Master strategy", $\hat{s} \in \mathcal{S}$, and a few other strategies which are dependent on $\hat{s}$. The other strategies in $\mathcal{S}$ will when queried for a move in turn query $\hat{s}$ and then each alter $\hat{s}$ recommendation in some way. One alteration for a strategy in $\mathcal{S}$ could be to recommend the direction that $\hat{s}$ recommended plus a random number of degrees. Another strategy might recommend the same move minus a random number of degrees. Each strategy $s \in \mathcal{S}$ is then assigned a predetermined probability weight of P_s, as in the Weighted PS.

Later on in this thesis we will use some of the probabilistic strategies described above in an attempt to create some more "realistic" (meaning less deterministic, and thus more biologically plausible) results from our simulations.

8.2 Validation of the components

The deterministic strategies suggested in Section 8.1.1 are ideal for simple validation of the various components in the simulation so far. Since we always know how the deterministic strategies will behave in a given situation, it is relatively easy to compare the output of the program to what we know should be the correct result. If the Rat walks through a wall, or starts moving outside the cage, hopefully the deterministic strategies will reveal this. Since the strategies are so regular and predictable, any unexpected behavior because of errors in the framework becomes easier to detect.

Figure 8.1 and 8.2 are examples of plots produced from such output. The plots show output from two simulations which use different strategies in a 100×100 grid-sized cage. The Rat starts in position $(0, 0)$ facing north (up) in both simulations.

The figures on page 52 shows what we from now on will refer to as a *smell map*. The grid in these types of maps represents the cage (at a given resolution), and each visitable point in the grid is marked with a color which indicates the "smell-value" of that point. Points with higher smell-value are marked with darker grey, and those with lower value with lighter grey. As mentioned earlier, the smell-values are additively increased for each time unit the subject is at a given point. All smell-values across the grid are reduced at regular and given time intervals, so that points recently visited will appear darker, and positions visited a long time ago will appear brighter.

Figure 8.1 is taken after about 10 000 steps with the DeterministicAvoid-SmellStrategy. Notice that the smell marks is stronger in a diagonal band between the corners. This is because the strategy will always use 1 extra time unit to turn the Rat 90 degrees. We also not that the smell mark decreases towards the edges (where it started), and is stronger nearer the center of the cage (where the Rat is now moving).

Similarly[2], figure 8.2, shows a plot of the smell map left by a subject using the DepthFirstStrategy. We note that the subject behaves as expected, always first attempting to go southwards, then eastwards then northwards, and finally westwards, and avoiding previously visited grid points.

8.2.1 Custom strategies for validation

To further ensure that the different various components behave as we expect them, we define the following strategies.

TestStrategy

The TestStrategy is created to monitor the input and output to another strategy. When you create a TestStrategy, you must supply another strategy that you would like to test.

Then, each time the TestStrategy is given input, it will display what input is given to the screen, and forward the input to the strategy being tested. Similarly, each time the TestStrategy is queried for a move, the

[2]Note that for illustration, the rate at which the smell is decreased is different in the two figures.

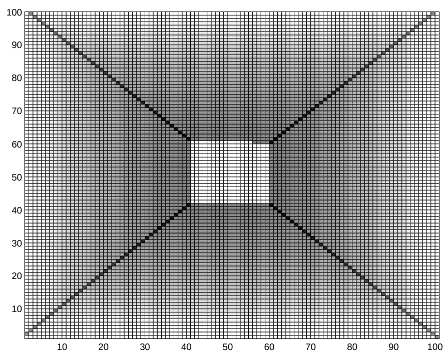
Figure 8.1: Plot of the smell marks left by a Rat using the DeterministicAvoidSmellStrategy. A darker color indicates a stronger (higher) smell-value.

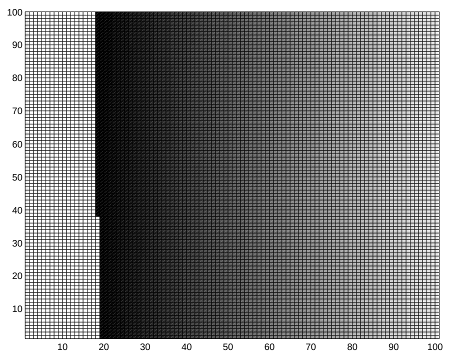
Figure 8.2: Plot of the smell marks left by a Rat using the DepthFirstStrategy.

TestStrategy queries the strategy being tested, and display the result, before it returns it to the Rat. The TestStrategy's interaction pattern is shown in 8.3. Although not a very large class, it has proven extremely useful in that it can in a simple manner be used to examine all input and output given to a strategy. This type of simple logging of what the strategy actually does is a good way to detect possible errors. This type of testing (where you only observe input and output to a software component) is commonly referred to as *black-box testing*.

ManualStrategy

One of the great advantages of discrete event simulation, is that simulation-time can be stopped, so that no further updates to the simulation will occur for an arbitrary amount for user-time. The ManualStrategy takes advantage of this fact.

Given the above strategy for logging of input and output, it would be convenient to see how a subject behaves in a certain situation. The ManualStrategy will upon each call to *recommendMove()* query the user to type (in using the keyboard) how the subject should move. While the simulation is waiting for the user to type in the answer, simulation-time is effectively stopped. This strategy can for instance be used to maneuver the subject into all positions and situations in the cage.

In the next part we will discuss and develop more advanced and sophisticated strategies, which model behavior in rat which is more "realistic". Although the implementation of these strategies will be a lot more complex than the strategies we have developed and shown so far, they will implement

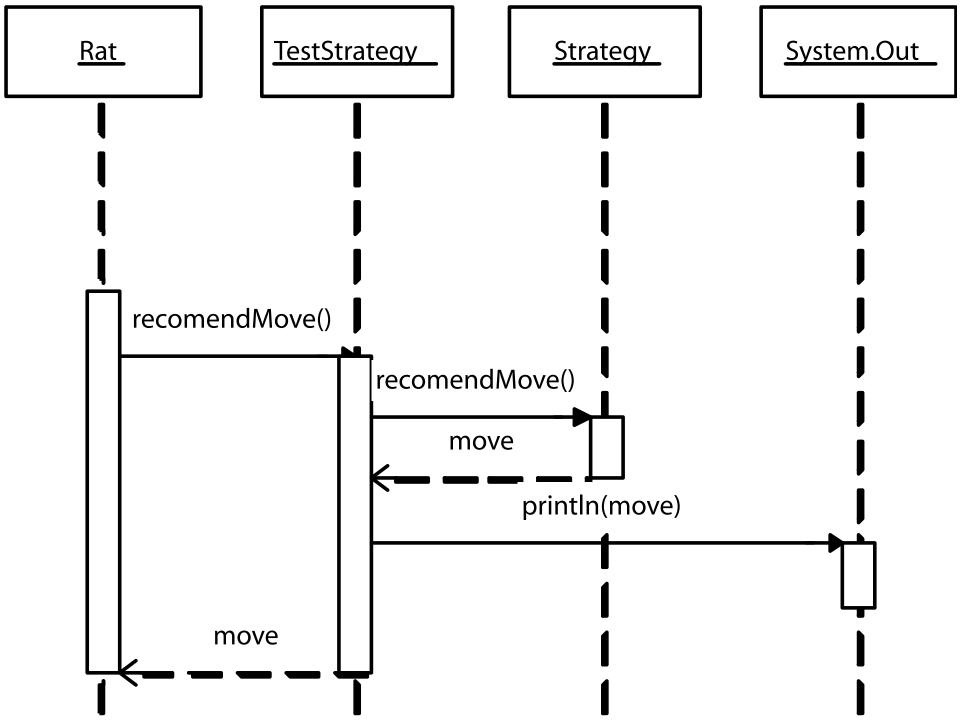

Figure 8.3: The interaction pattern of the TestStrategy.

the same simple bergmann.simulation.Strategy interface presented in Program 4.1.3. This ensures that exactly the same extended framework can be used in the simulations in the next part, as in this part.

Therefore, in the following, we will focus less on the extended framework itself (since it is the same as discussed in this part) and focus more on a precise presentation of how the strategies actually work. We have also chosen to focus less on the Java implementation of the strategies, since the strategies can be more easily described and understood when formulated in a more formal and mathematical language.

Part III

Simulation of rodent navigation

In Part III we will use the rat simulation framework created in Part II and develop strategies which model more realistic rat behavior.

We start off in Chapter 9 with a discussion about why we simulate the rat behavior on a computer and what we hope to achieve with the simulations.

In Chapter 10 we introduce two important aspects of rat behavior that we will examine; exploration and path integration. Exploration is the phase where the rat learns about the environment, and path integration is where the rat attempts to return to a goal in the cage.

Chapter 11 presents a published model which aims at describing exploration behavior in the rat. We implement the model as a strategy, simulate and present the results. We also do some comparison with the results presented in the original papers [Tchernichovski et al., 1998, Tchernichovski and Benjamini, 1998].

Our final strategy implements the Trullier-Meyer [Trullier and Meyer, 2000] model describing path integration in the rat. This is a relatively extensive model and there are some empirical evidence from real rat behavior to support it. This will be furthered studied in Part IV of the thesis. In Chapter 12 we present the model, we implement it and show sample output from the simulations.

Chapter 9

About the simulations

In this section we will use the framework for discrete event simulation developed in Part I and II of the thesis, for more advanced simulation of rodent navigation. Each of the two simulations we carry out will model different aspects of a rat's behavior.

9.1 Why simulation

As stated in Section 2.1, simulation is often used when it is too expensive, not ethical or simply not possible to experiment with the system we attempt to model. In the case of rat navigation, all of these reasons for simulation are to some degree applicable.

It is very difficult to alter how a free-running rat behaves when exploring an environment, although some progress has been made, as reported by IEEE Spectrum [Moore, 2002] recently ("The brain as a user interface"). As far as knowledge of the representation of the environment that the rat builds up goes, we can only rely on a relatively modest number of electro-physiological recordings from the rat brain. Currently, no good method for altering this internal representation of a rat's environment exists, and theories about how a rat stores and uses data can therefore only be tested out in some sort of simulated environment.

Given that we cannot perform extensive and invasive experiments on the brain in freely moving rat without running the risk of being unethical or mutilating the system we want to study, the question becomes how to best simulate the aspects of the rat we would like to model. Burgess [Burgess et al., 1997] propose using a mobile robot, placed inside a cage. Although they can show good results from their simulations, we believe that it is both easier, faster and cheaper to implement a simulation on a computer. We also believe that a computer simulation of rat navigation is easier to change, if the need should arise for simulation of other aspects of the rat than that originally intended or new empirical data needs to be taken into account. Easier implementation does not always provide better results, but it most certainly means quicker implementation and lower cost.

The final reason for not experimenting directly on animals, or at least reduce experiments, is one of empathy. It is a good principle that if an operation and examination poses a strong danger to the life or health of an animal, and there exists other ways of obtaining the same information, then the operation should not be carried out. This is especially true, if we do not know what effect the procedure will have on the animal in the first place, or whether the data so obtained will for sure improve scientific understanding of the system we are studying.

9.2 The purpose of these simulations

In the next two chapters we will show two published models describing two different aspects of rat behavior. These two models will both use the same simulation framework and the same simulation entities.

We will implement these two models as two different Strategies (implementing bergmann.simulation.Strategy). Therefore they will both, each in a separate simulation, dictate the behavior of the simrat, without needing any other changes made to the simulation framework itself. This is a good example of how the object-orientation principles of modularity and polymorphism saves us from having to do the same work twice (implement the simulation environment more than once).

What we hope to achieve by implementing these two models as strategies is to verify that these models are indeed sound, and that they are successful in describing the various aspects of what they attempt to describe. In order to help determine how good the simulations actually are, we will later introduce quantitative measures as a reference.

Chapter 10

Basic principles of rodent navigation

Rat navigation is often divided into two phases

- Exploration
- Path-integration

In the next chapters we will introduce one model which describes each of these two phases.

10.1 Exploration

Exploration is the process through which the rodent familiarizes itself with the environment. The rat makes excursions into the environment in order to learn more about it. According to [Tchernichovski et al., 1998] there are a few basic properties that hold true for rat exploration:

1. During exposure, the excursion length changes according to the following pattern
   (a) Each rat is characterized by a typical excursion length.
   (b) Excursion length increases during a session at a similar rate for all rats.
   (c) Excursion length increases from one session to the next at a similar rate for all rats.
2. The speed profile of the rat changes from a home base attraction profile to a home base repulsion profile. This change progresses from the home base and out.

The model introduced in the next chapter attempts to describe all of these features with a small algorithm.

10.2 Path integration

According to Redish [Redish, 1999], path integration is the ability to return directly to a starting point (sometimes called a *home base* or a *goal*) from any location in an environment, even in the dark or after a long circuitous route. Path integration, sometimes called *dead reckoning*, has been showed in hamsters, house mice, rats, birds, insects, dogs, cats and humans (see [Redish, 1999] for references).

Path integration in animals has been the subject of argument for more than a century, including a notable debate in 1873 between Alfred Wallace and Charles Darwin in which Wallace suggested that animals find their way back via sequences of smells and Darwin argued that animals must be using dead reckoning. The carefully controlled experiments of Mittelstaedt and Mittelstaedt (1980) and Etienne (1987) have demonstrated conclusively that this ability is a consequence of integrating internal cues from various input signals and movement information.

The basic idea of path integration is that if one knows one's location as well as one's speed and direction at time t_i, then the position at time t_{i+1} can be calculated. The main problem with path integration is that if your speed and direction is wrong, your representation of position will be increasingly inaccurate.

Path integration has been used in shipboard navigation for thousands of years. Polynesian navigators used path integration techniques to cross the Pacific Ocean over distances of thousands of miles, with no land in sight. Even as late as the eighteenth century, European navigators were still using dead reckoning to determine longitude, often with disastrous effects. With modern technology and much more accurate time pieces, however, submarines can navigate under the polar ice cap using dead reckoning with errors of less than a mile per week ([eb9, 1994]).

Because error in the path integrator can be corrected from local view, there may be some systematic error in the path integrator that is corrected by external cues. When Müller and Wehner (1988) examined desert ants and Seuinot, Maurer and Etienne (1993) tested hamsters, they found systematic error depending on specifics of the path taken. However, a path integrator that drifts too much will be useless.

In Chapter 12 we will describe and simulate a model for path integration which makes extensive use of vision for determining its position, and which makes it possible for a simrat to find its way back to a goal, from all positions in the cage.

Chapter 11

A model for exploration –Tchernichovski-Benjamini

Tchernichovski and Benjamini (TB) [Tchernichovski and Benjamini, 1998] suggests a discrete analytical model which describes exploratory behavior in the rat. The model attempts to describe the relationship between speed and distance from a "home base", and assumes that the rat has a basic motivation for exploration that will decrease as the distance to its home base becomes larger. This model represents both positions and the time frame in discrete values. This corresponds nicely with our framework.

In the next chapter this model will be used extensively to guide the simrat during exploration.

11.1 A model for exploratory behavior

The model starts by defining a point in its cage, u_0, known as the home base. The model assumes that the rat has a basic motivation, $M_i \leq 1$, which describes its willingness to explore. The rat will continue to advance as long as this motivation is positive, and will retreat towards the home base, u_0, when it is less than or equal to 0.

The index $i \in 1, 2, \ldots$ will refer to the time. Thus M_i will be the motivation of the rat at time i and u_i will be the position of the rat in the cage at time-step i. Speed also plays an important part in this simulation. See Section 6.3.4 for more information about how we handle speed in our framework.

The rat has a built-in "level of uncertainty" (or fear), $P(\|u_i - u_0\|)$, which depend on the distance from where the rat is, u_i, and the home base, u_0. $P(x)$ has it's minimum value at 0 and is convex. This function will be used as a basis for reducing the motivation M_i, in time-step i. A function $W(\cdot)$ is expressed to describes the change in the rat's "level of fear" from time-step $i-1$ to i, for $i \geq 1$. $\alpha(u_i)$ is a measure of how much accumulated time the simrat has spent in position u_i.

For each time-step i: u_i, M_i and W_i should be updated according to the following rules

$$u_{i+1} = \begin{cases} \text{advance}(u_i) & \text{if } M_i > 0 \\ \text{retreat}(u_i) & \text{if } M_i \leq 0 \end{cases}. \tag{11.1}$$

$$M_{i+1} = \begin{cases} M_i - \beta W(u_i) & \text{if } u_{i+1} = \text{advance}(u_i) \\ M_i & \text{if } u_{i+1} = \text{retreat}(u_i) \\ 1 & \text{if } u_{i+1} = u_0 \end{cases}. \tag{11.2}$$

$$W(u_i) = \begin{cases} \alpha(u_i)W(u_i) & \text{if } u_i \text{ has been visited before} \\ P(\|u_i - u_0\|) - P(\|u_{i-1} - u_0\|) & \text{otherwise} \end{cases} \tag{11.3}$$

$$\alpha(u_i) = \frac{c}{\text{Number of times the rat has visited } u_i} \tag{11.4}$$

where *advance* $(\cdot)$ should generally (this may not always be possible) ensure that $\|u_i - u_0\| > \|u_{i-1} - u_0\|$ and similarly *retreat* $(\cdot)$ that $\|u_i - u_0\| < \|u_{i-1} - u_0\|$.

11.2 Minor modifications of the model

In the original article, the TB model uses an angular component (between a wall and the simrat as seen from a corner in the cage) as a measure of the distance from the simrat to the home base. We have chosen not to use this measure, and instead opted to use the Euclidean distance since this is a standard measure already implemented in our framework. Furthermore, if we place a landmark in the position of the home base, then the simrat is able to calculate the distance from the home base to itself using its vision input.

Also, instead of using equation 11.4 directly as specified, and count the number of visits to each point, we will use the the smell value as a basis instead. This will make our model slightly more realistic, because we do not place on the simrat the responsibility of remembering the coordinates of all the positions that it has previously visited. The smell value will instead provide us with an estimate of the accumulated time the simrat has previously spent on each of the positions in the cage.

In our simulations we have used $\beta = 0.1$ and $c = 1$. The simrat starts facing north in the upper left hand corner of the cage. We use a UniformRandomStrategy which moves either to the south or east (with the same probability) when advancing, and a UniformRandomStrategy which moves north or west when retreating. See Section 8.1.2 more about the UniformRandomStrategy.

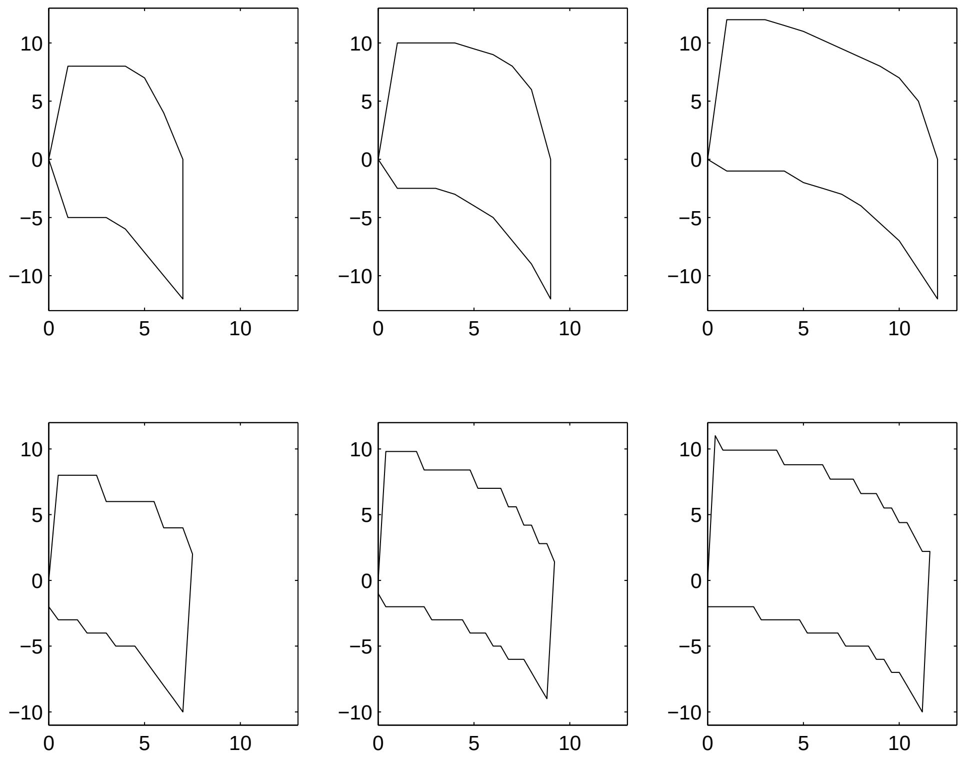

Figure 11.1: Plot of distance to home base vs speed for 6 excursions. Each plot shows an excursion starting at the home base u_0 and moving out into the environment, and ending as the simrat returns to u_0. The x-axis represents distance, and the y-axis, velocity. The top three excursions are the results presented in [Tchernichovski and Benjamini, 1998], and the lower three are generated from our simulations. Time in each of the two sets is increasing from the first plot (to the left) to the third plot (to the right), i.e. the leftmost excursion was take before the rightmost. Notice that the plots from our simulation is more "jagged" then the results from TB's simulations. This is because of the way our framework handles speed. See Section 6.3.4 for more information.

11.3 Results

Using the model mentioned in Section 11.1 we were able to produce the results in figure 11.1. As described in [Tchernichovski and Benjamini, 1998] we observe that the excursion length increases as a function of the excursion number. The movement pattern of the simrat changes concurrently with increases in excursion length: in the first ones which are short, the movement pattern indicates home base attraction; for every location visited by the simrat, speed is higher on the way back to the home base. This pattern is, according to TB, typical for early excursions in real rats.

During later excursions, the simrat leaves the home base at high speed, then the velocity decreases until it starts its return. While returning the speed is initially high, but when the simrat approaches the home base, its speed decreases again. Therefore the movement pattern of the simrat is reversed (becomes higher on in the way out) in the vicinity of the home base. There the velocity pattern falls into the category of home base repulsion pattern - it is higher on the way out. In the distant portion of each excursions, the velocity pattern remains primitive, i.e., it is higher on the way back. Both the change in excursion length as exploration progresses and the change in kinematics described above are, according to TB, quantitatively similar to the main findings observed in the real rats.

As shown in figure 11.1, we believe our results to be comparable to those produced by TB.

Chapter 12

Simrat navigation –The Trullier-Meyer model

Olivier Trullier and Jean-Arcady Meyer (TM) [Trullier and Meyer, 2000] presents a "model based on biologically plausible mechanisms ... that makes it possible for a simulated rat to navigate in an continuous environment containing obstacles". The strategy works in two phases: first *exploration* and later *path-integration*.

During the exploration phase, the simrat will move through the cage and attempt to create an internal representation of the environment as accurately as possible. It will build up a data structure that will become the basis on which later decisions are based on.

The actual movement of the simrat during the learning phase is determined by the Tchernichovski-Benjamini model for exploration, discussed in the previous chapter.

When the simrat later will wish to move from one position in the cage to another, it will enter the path-integration phase. It will now attempt to utilize the data it has learned during the exploration phase, and use these data to reach a *goal*.

12.1 The exploration phase and the learning model

During exploration of the environment, the rodent will attempt to build a "mental graph" representing the environment. This is in correspondence with how researchers believe that real rodents navigate ("The Hippocampus as a Cognitive Graph" [Muller et al., 1996]).

The basic idea of the Trullier-Meyer model for learning the environment is that the rat learns *distances* between itself and *landmarks*[1]. Landmarks are placed at various positions in the cage. As the simrat moves around in

[1]See Section 6.2.1 for definition of landmarks.

the cage, the simrat will attempt to learn the distances from the simrat to each landmark. For each one of the positions in the cage that the simrat visits, the distance to all visible landmarks are calculated. The simrat will store these sets of distances for each position. Each set of distances will identify one position in the cage.

The next step in the Trullier-Meyer model for navigation is to identify how each pair of positions learned (and stored as distances), relates to each other. During this step the simrat will build a (cognitive) graph where each vertex will correspond to a position, and an edge between a pair of vertices indicates that these two positions in the cage are physically close to each other.

As the simrat moves around, it will eventually find a *goal*. A goal is a position in the cage, that the rat should be able to return to from all other positions in the cage. When the simrat finds a goal, the vertices in the cognitive graph will be marked with the direction in which to move to reach the goal.

Eventually, the simrat will reach the path-integration phase. During this phase we would like the simrat to return to the goal found during the learning phase. This will be done by examining the cognitive graph in order to find the direction to the goal.

In the next sections we will describe the details of the Trullier-Meyer model.

12.1.1 Representing positions

We start off by defining how positions in the cage are represented in "rat memory".

This model uses vision as its primary source of input (see Section 7.3 for information about how we represent vision). The simrat's vision will provide the simrat with information about the position of the landmarks relative to the simrat's current position.

From the vision input we assume that the simrat is able to determine the distance to all visible landmarks, as well as the angle between each of the landmarks.

Let $u = \begin{bmatrix} x & y \end{bmatrix}$ be the position of the simrat in the cage. Let $d(u)$ be a function that returns a vector containing the Euclidean distance from u to each of the landmarks. If there are a total of n landmarks, then $d(u)$ will have n elements, and element i of this vector will contain the distance to landmark i. If landmark i is not visible then some predetermined value (in our implementation set to -0.1), will be used at position i in the vector instead. Under reasonable assumptions $d(u) : \Re^2 \rightarrow \Re^n$ will uniquely define the position u in the cage. See appendix A for further details.

Let $\theta(u) : \Re^2 \rightarrow \Re^{n \times n}$. This function returns a matrix with $n \times n$ elements, where element $(i, j) \in 1, 2, \dots, n$ is the angle between landmark i and landmark j from position u. If an angle cannot be determined, we assign a predetermined value (in our implementation set to ∞). $\theta(u)$ will later become important when we need to calculate which direction to move in to get to a given position.

Let i be a *PlaceCell*. We store the matrix returned by $\theta(u)$ in θ_i^*, and the vector returned by $d(u)$ in d_i^*. The PlaceCell is an important abstraction in our implementation. It is used for storing information about positions in the cage that has been previously visited. The PlaceCells are named after actual cells found in the Hippocampus area of the brain, and believed by Trullier-Meyer to have a similar function in real rats (storing spatial information)[2]. The PlaceCells in our simulations will model these real PlaceCells found in rats.

We name the set of all PlaceCells $\mathcal{P} = \{1, 2, \ldots\}$. We hope to generate just enough PlaceCells to accurately represent the cage, that is to make the set $\mathcal{P}$ as small as possible while still maintaining a good representation of the cage. Later on the set $\mathcal{P}$ will be examined extensively over and over again, so by minimizing $|\mathcal{P}|$ we not only save memory, we also make the algorithm (and simulation) faster.

The activation function

Let us now assume that the learning phase is complete and let the current position of the simrat be u.

In Chapter 7 we assumed that the simrat left a trail of "smell marks" and that the simrat could use these marks to tell whether it had been in a given position u before. We will now examine a more general method based on vision to detect whether the simrat has been in position u before. The method used in [Trullier and Meyer, 2000] for testing this is with an *activation function*.

An activation function, $\alpha_i : \Re^2 \to [0, 1]$, compares two distance vectors, and returns a scalar value between 0 and 1. The hypothesis is that if there is little difference between the two distance vectors, $d(u)$ and d_i^*, then the two positions in the cage, u and u_i, must be physically close together. Similarly, if there were large differences between $d(u)$ and d_i^*, then we believe that u and u_i are far apart.

The activation function Trullier-Meyer propose is

$$\alpha_i(u) = \exp(-\frac{\|d(u) - d_i^*\|^2}{\sigma^2}) \qquad (12.1)$$

where u is the current position of the rat in the cage and d_i^* is the distance vector corresponding to PlaceCell $i \in \mathcal{P}$ (we say that α_i is the activation function associated with PlaceCell i).

[2]See Chapter 13 for more information about the "biological" PlaceCells.

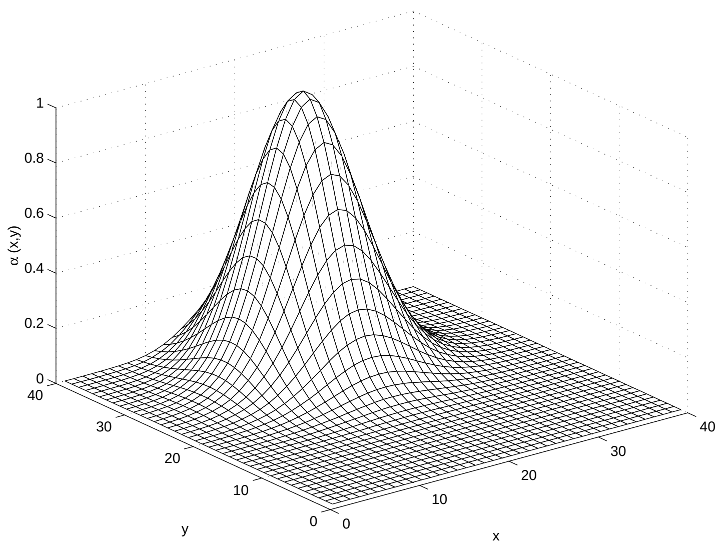

Figure 12.1: Activation function of PlaceCell i. The distance vector $d_i^* = d(\begin{bmatrix}20 & 30\end{bmatrix})$ and $\sigma = 0.1$. For each position $u = \begin{bmatrix}x & y\end{bmatrix}, (x,y) \in \{1, 2, \ldots, 40\}$, $\alpha_i(u)$ has been computed and plotted at position u. There are 4 landmarks, one located in each corner.

σ is a value which roughly determines the "width" of the activation function. A small value for σ will create a very steep and narrow shape, and a large value for σ will create a wide and large curve on the plot. Various values for σ can be seen on figure 12.2.

The activation function has the property that it will return a value equal to 1 if the two distance vectors d and d^* are equal, a value close to but less than 1, if the vectors are "similar", and close to but greater than 0, if the vectors are very "different".

Unwanted properties

This activation function also introduces some possible inaccuracies into our model. If two positions in the cage are physically close together, but from one of the position one cannot see landmark k, and from the other position one can[3]. If so, then one distance-vector will have the default value of -0.1 in element k, and the other distance-vector will have the actual distance, in element k of the vector. Depending on the distance to the visible landmark from the unobscured position, the difference between element k of the two distance vectors will vary. If the difference between the elements is large (landmark k is far away) then the value of $\alpha(\cdot)$ would be about as if landmark k was visible from both positions. However, if the difference is small

[3]This will be the case if the cage contains obstacles hiding the view to a landmark.

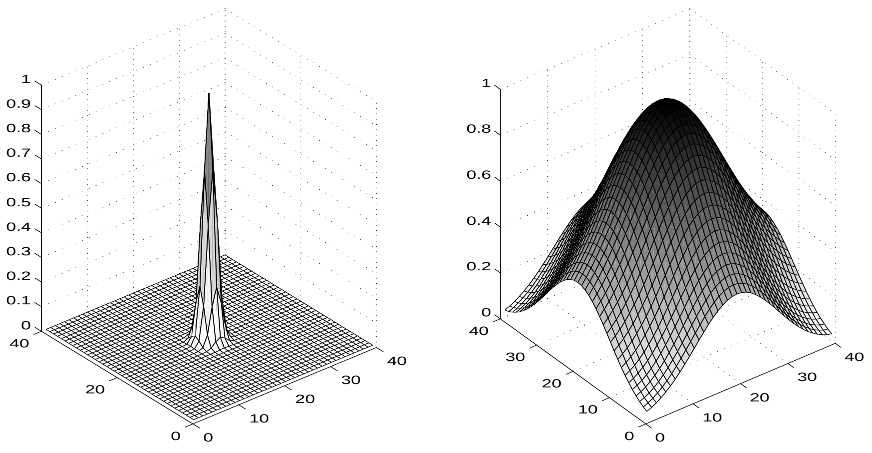

Figure 12.2: Same parameters as in figure 12.1, but the leftmost plot has $\sigma = 0.025$, and the rightmost plot has $\sigma = 0.25$.

(landmark k is close), then value of the activation function may change dramatically. This may introduce discontinuities in the shape of the activation function.

This situation only becomes worse if even more landmarks are not visible from the position represented by one of the PlaceCells, but visible from the other. The activation function might eventually classify the two positions as not physically close to each other, even though they are, if the differences in the vision input becomes too large. This is a possible source of error in our model.

A possible solution to this problem could be to only consider distances to landmarks visible from u and disregard all other landmarks. However, we believe that the discontinuity is an integral part of the navigation model and further clinical studies are needed.

12.1.2 Building a cognitive graph

Now the Trullier-Meyer strategy for rodent navigation attempts to build a explicit relationship between each of the PlaceCells. This is done by first building a (cognitive) graph, $G = (V, E)$, where the PlaceCells will correspond to vertices and an edge between two vertices indicate that the physical position the vertices represents, are close together.

For a given position u, $\alpha_i(u)$ and $\alpha_j(u)$ are measures of "closeness" to PlaceCell i and j. If both are large, then u is assumed to be physically close to u_i and u_j, and hence u_i and u_j must be close (i.e. $u_i \approx u_j$).

We now introduce a threshold value $0 < t_\alpha < 1$. Let $v_i \in V$ and $v_j \in V$ be the vertices in the cognitive graph G associated with PlaceCell i and j, respectively, and u be the current position of the simrat. We add an edge between v_i and v_j if $\alpha_i(u) \geq t_\alpha$ and if $\alpha_j(u) \geq t_\alpha$. We say that PlaceCell i is *active* at u if $\alpha_i(u) \geq t_\alpha$.

We are now ready to start to consider in detail what will happen during exploration.

The exploration phase

For each position, u, the simrat visits during exploration the simrat should attempt to find a set $\mathcal{M} \subset \mathcal{P}$ containing the m active PlaceCells $i \in \mathcal{P}$ with largest $\alpha_i(u)$. m is a given predetermined integer. In other words; if $\alpha_1(u), \ldots, \alpha_{|\mathcal{P}|}(u)$ are the values of the activation functions at u, let $\hat{\alpha}_1(u), \ldots, \hat{\alpha}_{|\mathcal{P}|}(u)$ be the sorted values in descending order. Then the *neighborhood* of u is the set $\mathcal{M} = \{j \in \mathcal{P} : \hat{\alpha}_j(u) \geq t_\alpha \text{ for all } j \leq m\}$.

In attempting to find $\mathcal{M}$, there are four possible situations that can arise, each relating to whether the *position*, or the *neighborhood of the position* has been previously visited. We say that a neighborhood of u is visited if the set $\mathcal{M}$ exists and each of the PlaceCells in $\mathcal{M}$ is active. We also say that a position u is known if $\exists i \in \mathcal{P} : \alpha_i(u) \geq T_\alpha$. T_α is a threshold-value which satisfies $0 < t_\alpha < T_\alpha \leq 1$.

The four possible situation that can arise, follows.

Unknown position, unvisited neighborhood The set $\mathcal{M} \subset \mathcal{P}$ does not exist, and no PlaceCell $i \in \mathcal{P}$ satisfies $\alpha_i(u) \geq T_\alpha$. Since the simrat has neither been in position u or in the neighborhood of u before, it should create a new PlaceCell, k. The following updates to should take place: $\mathcal{P} \leftarrow \mathcal{P} \cup \{k\}$, $\theta_k^* = \theta(u)$ and $d_k^* = d(u)$.

Known position, unvisited neighborhood The set $\mathcal{M} \subset \mathcal{P}$ does not exist, but PlaceCell $i \in \mathcal{P}$ satisfies $\alpha_i(u) \geq T_\alpha$. The simrat previously visited this position in the cage, and the position u is already represented by a PlaceCell. We should do nothing.

Known position, visited neighborhood The set $\mathcal{M} \subset \mathcal{P}$ exists, and PlaceCell $k \in \mathcal{P}$ satisfies $\alpha_k(u) \geq T_\alpha$.

We should now update the cognitive graph, $G = (V, E)$. For each PlaceCell $i \in \mathcal{M}$ we should create the vertex v_i if it does not already exist. If $v_i \notin V$, then $V \leftarrow V \cup \{v_i\}$. For each ordered PlaceCell pair $(i, j) \in \mathcal{M}, i \neq j$, we should create the edge $e = (v_i, v_j)$. If $e \notin E$, the $E \leftarrow E \cup \{e\}$.

After this procedure, all PlaceCells that are in the same neighborhood, $\mathcal{M}$, has corresponding edges between them in the cognitive graph. See the next section for more about adding edges to the graph.

Unknown position, visited neighborhood The set $\mathcal{M} \subset \mathcal{P}$ exists, but no PlaceCell $i \in \mathcal{P}$ satisfies $\alpha_i(u) \geq T_\alpha$

We do not know whether the simrat has been in position u before, but since this neighborhood is already represented well enough by m PlaceCells, we assume that we do not need to represent it further.

Instead we update the cognitive graph, $G = (V, E)$. For each PlaceCell $i \in \mathcal{M}$ we should create the vertex v_i if it does not already exist. If $v_i \notin V$, then $V \leftarrow V \cup \{v_i\}$. For each ordered PlaceCell pair $(i, j) \in \mathcal{M}, i \neq j$, we should create the edge $e = (v_i, v_j)$. If $e \notin E$, then $E \leftarrow E \cup \{e\}$. Also, see the next section for more about adding edges to the graph.

Creating an edge between PlaceCells

To every edge in the cognitive graph we will define a direction, called the *edge direction.*

The edge direction will later become important as we examine the cognitive graph. This direction should roughly indicate which direction to move from the position represented by the one PlaceCell to the position represented by the other.

We will assume that the simrat is able to determine the direction between two landmarks from its vision input. In other words, we assume that the simrat is able to tell cardinal directions ("North", "South", "East", "West"), as if it had a compass.

Unfortunately, determining the direction between PlaceCells might not be as easy as it sounds, since we assume that the simrat does not know the explicit cage coordinates to the positions that the PlaceCells represent.

Let i and j be two PlaceCells $\in \mathcal{P}$ that represents position u_i and u_j (i.e. $d_i^* = d(u_i)$ and $d_j^* = d(u_j)$). We would now like to calculate the direction $u_j - u_i$, only knowing the θ^* and d^* values of the two PlaceCells and the direction between two visible landmarks.

The figure shows two PlaceCells (1 and 2, at position u_1 and u_2 in the cage), and two landmarks (a and b). To simplify the discussion, we assume that the direction from a to b corresponds to the cardinal direction "East" [4].

We now need to determine the angle between the two lines u_1u_2 and ab. This would give us the direction from u_1 to u_2 relative to the "East" direction.

Calculating the direction between PlaceCells Assume that $|u_1a|$, $|u_1b|$, $|u_2a|$ and $|u_2b|$ are known (the distances to the landmarks, stored

[4]Actually, any consistent absolute direction in the cage would do.

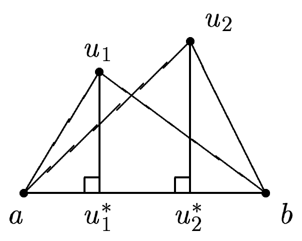

Figure 12.3: Two place-cells, u_1 and u_2, drawn in the position they represent in the cage, relative to two landmarks, a and b.

in d_1^* and d_2^*), as well as $\angle au_1b$ and $\angle au_2b$ (the angles between the landmarks, stored in θ_1^* and θ_2^*). From $\triangle au_1b$ with two known sides and one known angle, we are able to calculate $|ab|$ and $\angle u_1ab$. Knowing $|ab|$ we can also calculate $\angle u_2ba$. We arbitrarily choose a to be the origin, and place b at position $(|ab|, 0)$. Next we look at the *projection* of u_1 and u_2 onto ab. We name these projection-points onto ab u_1^* and u_2^*. The coordinate of u_1^* and u_2^* are then $(|au_1| \cos \angle u_1ab, 0)$ and $(|ab| - |bu_2| \cos \angle u_2ba, 0)$, respectively.

We now apply the Phytagorean theorem to $\triangle au_1u_1^*$ and $\triangle au_2u_2^*$ to find $|u_1u_1^*|$ and $|u_2u_2^*|$. This gives us that $|u_1u_1^*| = \sqrt{|au_1|^2 - |au_1^*|^2}$ and that $|u_2u_2^*| = \sqrt{|bu_2|^2 - |bu_2^*|^2}$. We now have enough information to express u_1 and u_2 relative to the line ab. u_1 is at point $(|au_1^*|, |u_1^*u_1^*|)$ and u_2 at $(|au_2^*|, |u_2^*u_2|)$.

The vector $u_1 - u_2$ will now give us the direction, relative to landmarks a and b, to move in from u_1 to u_2.

Our simulation does not need a precise direction, so all angles are classified relative to the "East" cardinal direction. Angles in the range $[-22.5, 22.5)$ are classified as "East" (class 1), in the range $[22.5, 67.5)$ as "NorthEast" (class 2), in $[67.5, 112.5)$ as "North" (class 3), and so on. Class s will contain angles in the range $[-22.5 + 45(s-1), 22.5 + 45(s-1))$. We store the classification of the angle between landmark i and j from the position u_k associated with PlaceCell k in $c_k(i,j)$. Note that we express angles so that $c_k(i,j) \in 1, 2, \ldots, 8$

12.1.3 Learning goals

To introduce a way for the simrat to learn positions that the simrat should be able to return to (from all positions in the cage), we introduce *goals*. In order to simplify the discussion, we will first consider the case when there is just one goal in the cage.

Let $g = \begin{bmatrix} x & y \end{bmatrix}$ be a position in the cage that the simrat should be able to return to. [Trullier and Meyer, 2000] introduces a set of directions $\mathcal{G}$ associated with g. We follow the notation used by TM and name each element in $\mathcal{G}$ a *GoalCell*. As with PlaceCells, GoalCells are named after actual cells found in the Hippocampus area of the brain, and believed by

TM to be used by the rat in determining its current position relative to a goal. The GoalCells in our model will serve a similar purpose.

Each GoalCell $i \in \mathcal{G} = \{1, 2, \ldots, 8\}$ will represent a direction relative to g. 1 will represent "East", 2 will represent "NorthEast", 3 will represent "North" and so on, giving a total of 8 directions.

In the previous section, the simrat created a cognitive graph $G = (V, E)$, which contained vertices $v_i \in V$ corresponding to PlaceCell i representing positions u_i. The simrat then added edges, $(v_i, v_j) \in E$ between the vertices associated with PlaceCell i and j, if it believed that u_i and u_j were physically close together. Now we would like to augment the graph G to also include information about each position u_i and its physical relationship to the goal g. This is done by introducing a vertex g_j to the graph G, for each of the GoalCells $j \in \mathcal{G}$. We add an edge (v_i, g_j) to E if u_i is in direction j from g. In this way we hope to classify all the PlaceCells $i \in \mathcal{P}$ relative to a goal g.

Let $G = (V, E)$ be the cognitive graph described in the previous section. When the simrat reaches g for the first time during exploration of the cage, it should create the set $\mathcal{G}$, and set $V \leftarrow V \cup \{g_j : j \in \mathcal{G}\}$. Next the simrat should find a PlaceCell $i \in \mathcal{P} : \alpha_i(g) = \max_{k \in \mathcal{P}} \alpha_k(g)$. For each of the GoalCells, $j \in \mathcal{G}$, the simrat should perform a depth-first graph-search starting at vertex $v_i \in V$, and only following edges $(v_x, v_y) \in E$ so that $c_x(x, y) = j$. For each of the vertices, v_z found in this manner, $E \leftarrow E \cup \{(v_z, g_j)\}$.

Finally, we define a function

$$\beta_j(u) = |\{i \in \mathcal{P} : \alpha_i(u) > t_\alpha, (v_i, g_j) \in E\}|$$

associated with GoalCell $j \in \mathcal{G}$. $\beta_j(u)$ returns the number of active PlaceCells which have an edge to the direction j at position u.

12.2 Returning to the goal

After having spent some time exploring and learning the environment, the simrat will attempt to return to a goal in the cage. The next sections describes how this can be done.

12.2.1 Single goal path-integration

We start by considering the base case when there is only one goal in the cage. The simrat will now attempt to return to this goal. This process is known as path-integration.

Let $u = \begin{bmatrix} x & y \end{bmatrix}$ be the current position of the simrat in the cage, and let $G = (V, E)$ be the cognitive graph constructed during exploration and learning of the goals in the previous sections.

Then the direction to the goal is a weighted sum of directions represented by the GoalCells $j \in \mathcal{G}$. Let $D(j)$ be a function which returns the direction

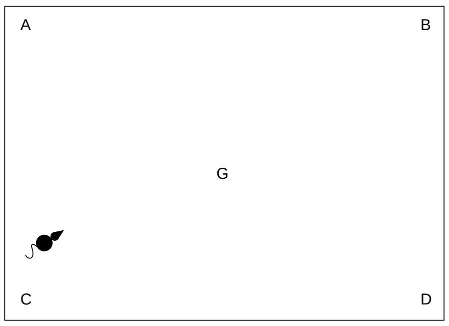

Figure 12.4: Cage with 4 landmarks, (A, B, C and D), a goal (G) and a simrat in the lower left corner.

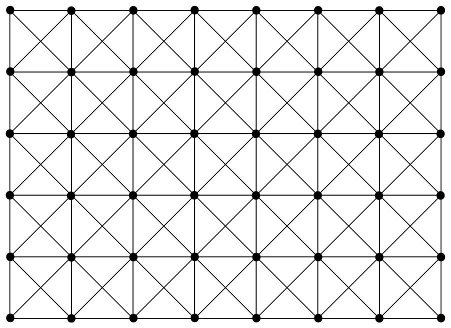

Figure 12.5: Idealized cognitive graph, showing the PlaceCells and edges corresponding to the cage. The lines represent edges, and the dots, PlaceCells. Simrat has completed the learning phase, so the cognitive map is relatively complete.

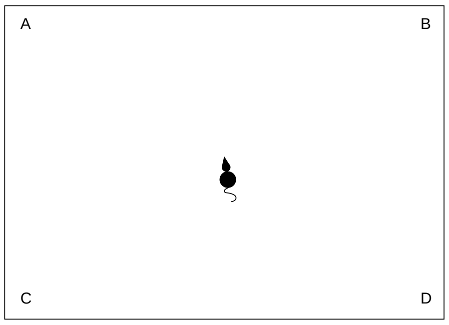

Figure 12.6: The simrat has moved to the center of the cage, where it has found goal G. It wishes to learn this position.

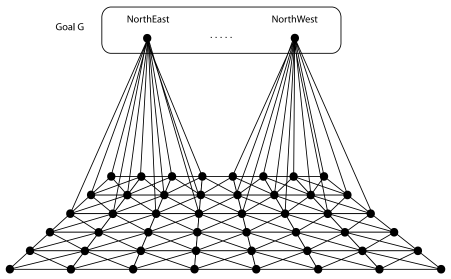

Figure 12.7: The rat creates a set of GoalCells for goal G. From the PlaceCell representing the current position (the ones in the center of the cage) it finds all PlaceCells reachable in each direction dictated by the GoalCells, and associate these with the corresponding GoalCell. For simplicity we have here only shown two GoalCells corresponding to the "NorthWest" and "NorthEast" direction.

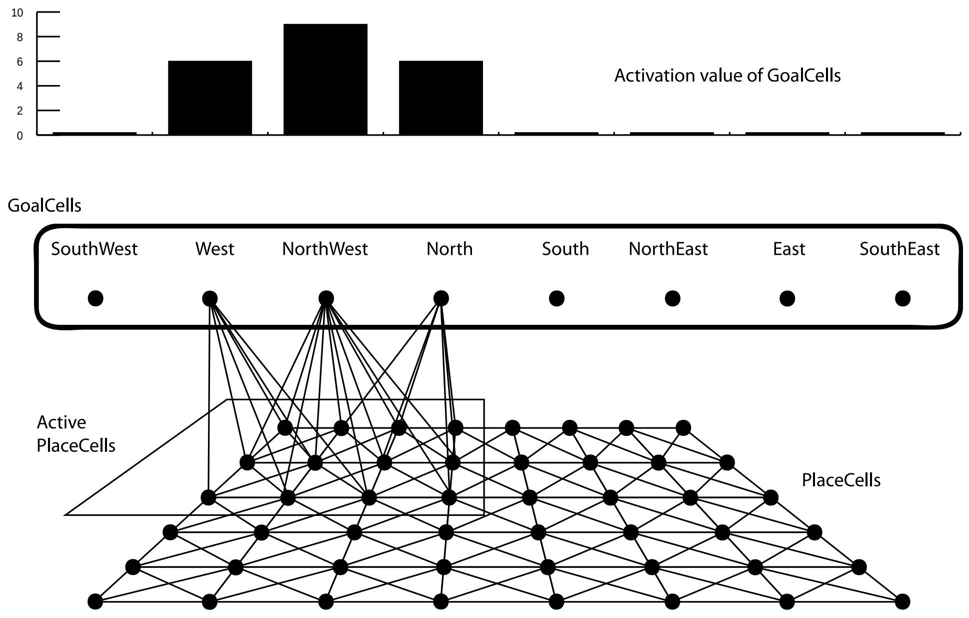

Figure 12.8: We see the active set of the PlaceCells, $\mathcal{M}$, and the GoalCells they are associated with. The bar-chart on top shows the current β values for each of the GoalCells. The simrat is currently in the upper left hand corner of the cage, as seen in figure 12.9.

that GoalCell j represent. Then the direction z to the goal is

$$z = -\sum_{j \in \mathcal{G}} \beta_j(u) D(j)$$

12.2.2 Multiple goals

In the previous section we only considered the case where there was one goal in the cage. The model extends to more than one goal. If more than one goal needs to be represented in the cognitive graph, then the above procedure must be repeated for each new goal.

Let $g^k = \begin{bmatrix} x_k & y_k \end{bmatrix}$ be one of the goals that we need to represent. Then, the first time the simrat visits position g^k, the simrat should create the set $\mathcal{G}^k$ containing 8 GoalCells representing directions relative to g^k. We also define a function $\beta_j^k(u)$ associated with GoalCell $j \in \mathcal{G}^k$. Now the procedure given in the previous sections can be followed by using $\mathcal{G}^k$ instead of $\mathcal{G}$, and $\beta_j^k(\cdot)$ instead of $\beta_j(\cdot)$.

Path-integration also works in the same way, but we work with the set $\mathcal{G}^k$ and function $\beta_j^k(\cdot)$, associated with goal g^k, instead of $\mathcal{G}$ and $\beta_j(\cdot)$ when

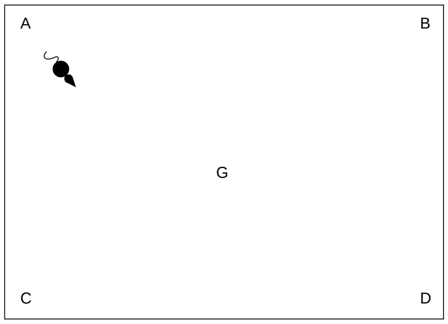

Figure 12.9: The simrat is in the upper left corner of the cage and wants to return to the goal G.

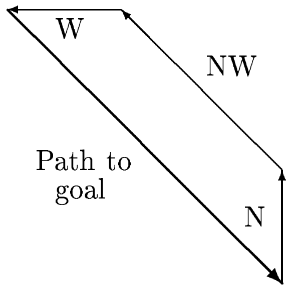

Figure 12.10: The direction to follow in order to return to goal G. The direction is found by taking the *opposite* of the vector-sum of the directions given by the GoalCells weighted by their respective β values. See figure 12.8 for more details.

evaluating z^k. Thus, the direction to goal k is

$$z^k = -\sum_{j \in \mathcal{G}^k} \beta_j^k(u) D(j)$$

12.2.3 Path-integration summary

Since it may be difficult to follow all the steps needed to evaluate the sum presented in Section 12.1.3, we provide the following summary. In order to simplify our notation, we only assume one goal in the cage.

Let $u = \begin{bmatrix} x & y \end{bmatrix}$ be the current position of the simrat in the cage. Let g be the goal in the cage. The simrat has already created the cognitive graph $G = (V, E)$ and learned goal g (see Section 12.1.3).

1. Find the set $\mathcal{S} = \{i \in \mathcal{P} : \alpha_i(u) > t_\alpha\}$. Notice that d_i^* is used to calculate $\alpha_i(u)$.

2. For each GoalCell $j \in \mathcal{G}$ count the number of edges $(v_i, g_j) \in E$ where $i \in \mathcal{S}$. Let this count for each GoalCell j be called β_j.

3. For each GoalCell $j \in \mathcal{G}$, find the direction $D(j)$ that j represents. $D(1) = \begin{bmatrix} 1 & 0 \end{bmatrix}$, $D(2) = \begin{bmatrix} \sqrt{2} & \sqrt{2} \end{bmatrix}$, $D(3) = \begin{bmatrix} 0 & 1 \end{bmatrix}$ and so on at 45° intervals around the circle of unity.

4. Evaluate the vector sum and find the direction z to the goal g: $z = -\sum_{j \in \mathcal{G}} \beta_j D(j)$.

12.3 Problems

Notice in Section 12.1.3 that the function $\beta_j(u)$ is defined to return the number of active PlaceCells which have an edge to GoalCell j.

If no PlaceCells are active or no active PlaceCells have edges to GoalCell j then $\beta_j(u)$ will return 0. If no GoalCells has and edge to an active PlaceCell, then $z = [0]$ and the simrat will not have any direction to the goal. This can happen if there are obstacles in the cage, which will cause some positions to not be represented in the map.

In this situation, the Trullier-Meyer model states that the simrat cannot take a direct route to the goal. Instead, a series of *subgoals* will need to be visited in sequence to reach the goal.

A subgoal is a goal in the cage, that the simrat will need to visit before it can go to its final goal.

12.3.1 Creating subgoals

Let the g^k be the goal that the simrat should return to, but no PlaceCells are active ($\beta_j(u) = 0$ for all GoalCells $j \in \mathcal{G}$) then $z_k = [0]$. In this situation TM states that the simrat should start to move at random in the cage, until it finally finds a position where, when calculated, $z_k \neq [0]$. At that position in the cage, a new (sub-)goal, l, should be created. This is done by following the procedure given i Section 12.1.3. Then the simrat should follow the direction given by z_k.

If this proves succesfull, and the simrat is able to reach goal g^k, then goal l should be used as a subgoal for goal k in the future: if $z_k = [0]$ at a given position, then the simrat should check if $z_l \neq [0]$. If this holds true, then the simrat should follow direction z_l, until it finally finds $z_k \neq 0$. Upon finding $z_k \neq 0$, the direction z_k should be followed to goal g^k.

If however, after creating subgoal l, the simrat follows the direction given by z_k, and this does not lead to goal k, then the subgoal l and $\mathcal{G}_l$ should be removed from the cognitive graph; it does *not* lie one the path to goal k as it should. This can happen if z_k leads the simrat to a position which is not represented by enough PlaceCells. Now the process must be repeated, the simrat should again move randomly about, until $z_k \neq 0$ is found, a new subgoal is created and so forth.

We hope that over time this procedure will generate enough subgoals for the simrat to return to goal k from all positions in the cage.

12.3.2 Navigation using multiple subgoals

Often when dealing with large cages with several obstacles, the simrat needs to visit several subgoals in turn before reaching our final goal g^k. In order to

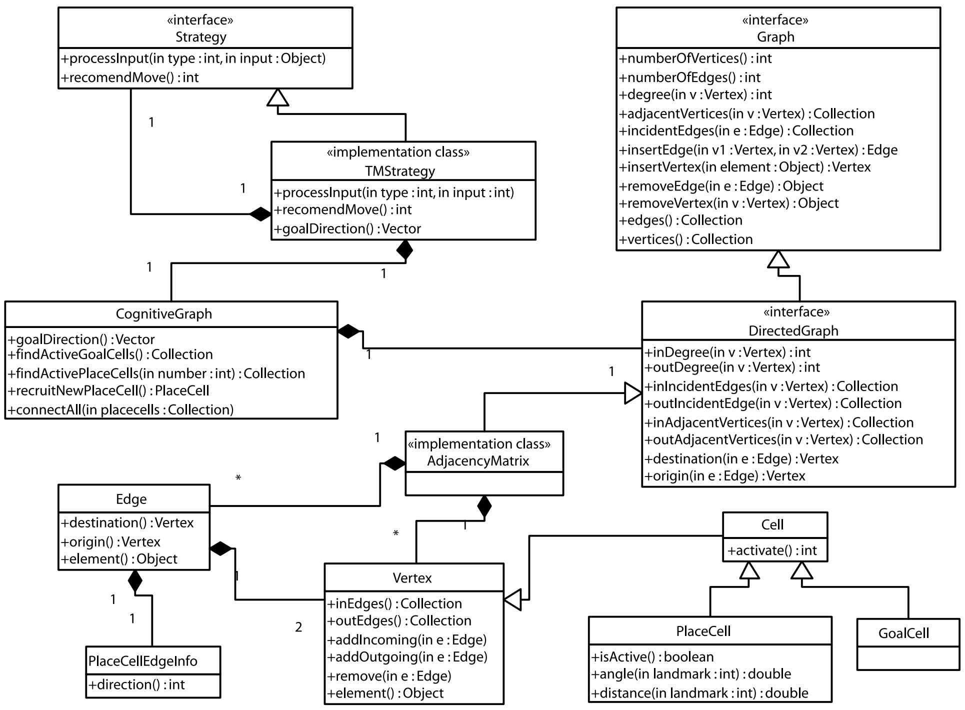
«interface»
Strategy
+processInput(in type : int, in input : Object)
+recomendMove() : int
«implementation class»
TMStrategy
+processInput(in type : int, in input : int)
+recomendMove() : int
+goalDirection() : Vector
«interface»
Graph
+numberOfVertices() : int
+numberOfEdges() : int
+degree(in v : Vertex) : int
+adjacentVertices(in v : Vertex) : Collection
+incidentEdges(in e : Edge) : Collection
+insertEdge(in v1 : Vertex, in v2 : Vertex) : Edge
+insertVertex(in element : Object) : Vertex
+removeEdge(in e : Edge) : Object
+removeVertex(in v : Vertex) : Object
+edges() : Collection
+vertices() : Collection
CognitiveGraph
+goalDirection() : Vector
+findActiveGoalCells() : Collection
+findActivePlaceCells(in number : int) : Collection
+recruitNewPlaceCell() : PlaceCell
+connectAll(in placecells : Collection)
«interface»
DirectedGraph
+inDegree(in v : Vertex) : int
+outDegree(in v : Vertex) : int
+inIncidentEdges(in v : Vertex) : Collection
+outIncidentEdge(in v : Vertex) : Collection
+inAdjacentVertices(in v : Vertex) : Collection
+outAdjacentVertices(in v : Vertex) : Collection
+destination(in e : Edge) : Vertex
+origin(in e : Edge) : Vertex
«implementation class»
AdjacencyMatrix
Edge
+destination() : Vertex
+origin() : Vertex
+element() : Object
Cell
+activate() : int
Vertex
+inEdges() : Collection
+outEdges() : Collection
+addIncoming(in e : Edge)
+addOutgoing(in e : Edge)
+remove(in e : Edge)
+element() : Object
PlaceCell
+isActive() : boolean
+angle(in landmark : int) : double
+distance(in landmark : int) : double
GoalCell
PlaceCellEdgeInfo
+direction() : int
1
1
1
1
1
1
1
*
*
1
1
1
2

Figure 12.11: The suggested implementation hierarchy.

facilitate this the simrat must order the subgoals according their respective distances to the goal.

It should assign goal g^k (which is its final goal) a distance of 0, subgoals which are close a "low" value, and subgoals which are far away a "large" value.

One possible way to find such a value for a subgoal is to consider the time this subgoal was created. Subgoals which are closer to the goal g^k, will probably have been created before subgoals which are further away from the goal, and therefore creation-time may be used as a measure of the distance[5].

The simrat should now always move towards the (sub-)goal j with the smallest distance to goal g^k which has $z_k \neq [0]$. If, at any point in time while on the way back, the simrat enters an area in which no GoalCells are active, the simrat should start creating subgoals, as described above.

Figure 12.11 shows a brief overview of our suggested Java implementation. Note that we make extensive use of the ADTs we have implemented in the bergmann.structure package. See Section 4.4 for more about our reasons for doing so. Both Graph, DirectedGraph, AdjacencyMatrix as well as Edge and

[5]Unfortunately, there are also several difficulties involved with this measure of distance. Several situations can be constructed for which this heuristic will fail. Other strategies for measuring distances must then be used.

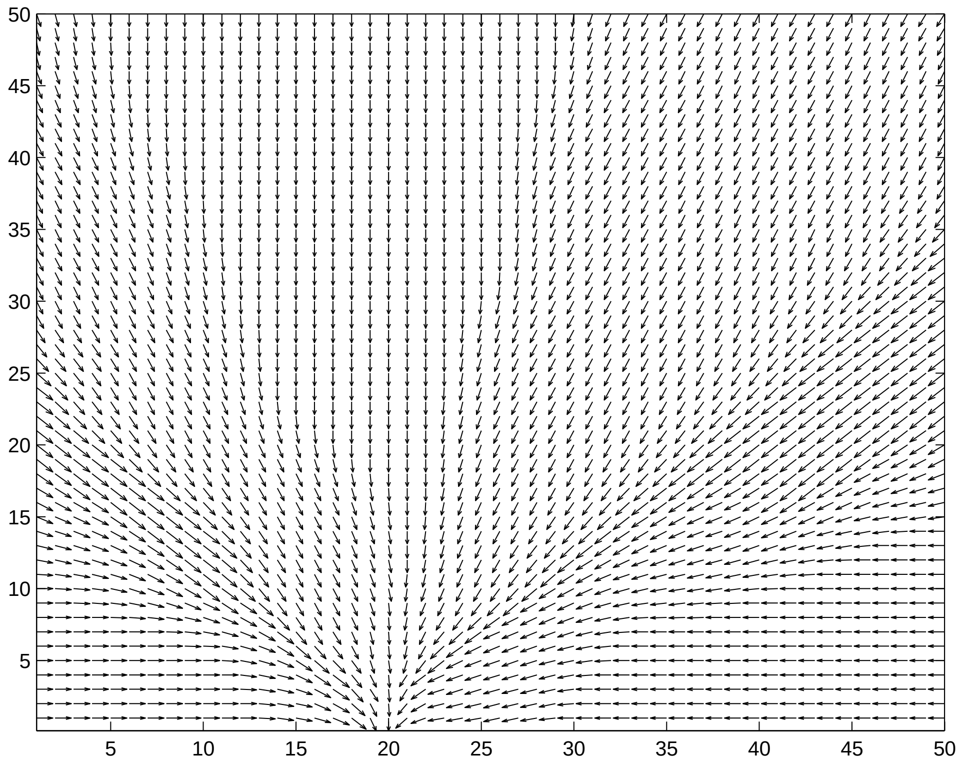

Figure 12.12: Plot of heading of the rat for all points in the cage. The cage is currently without obstacles of any type.

Vertex are implemented in the bergmann.structure, along with many other ADTs we have used in this thesis. We let the CognitiveGraph class hold a reference to a DirectedGraph implemented by the AdjacencyMatrix. We choose to extend the Vertex class, and create a common Cell class. This Cell class is then in turn extended by both PlaceCell and GoalCell. We let the single *activate(·)* method of Cell return α when implemented in PlaceCell, and β in GoalCell.

The TMStrategy is the "main" class which implements the bergmann. simulation.Strategy interface. The TMStrategy holds a reference to another strategy (in our implementation an instance of the TBStrategy class, discussed in the previous chapter). The TMStrategy makes extensive use of the CognitiveGraph class, which it also holds a reference to.

12.4 Results

From the algorithm presented in this chapter we were able to produce the following output.

Figure 12.12 shows a direction plot for a simrat attempting to return to

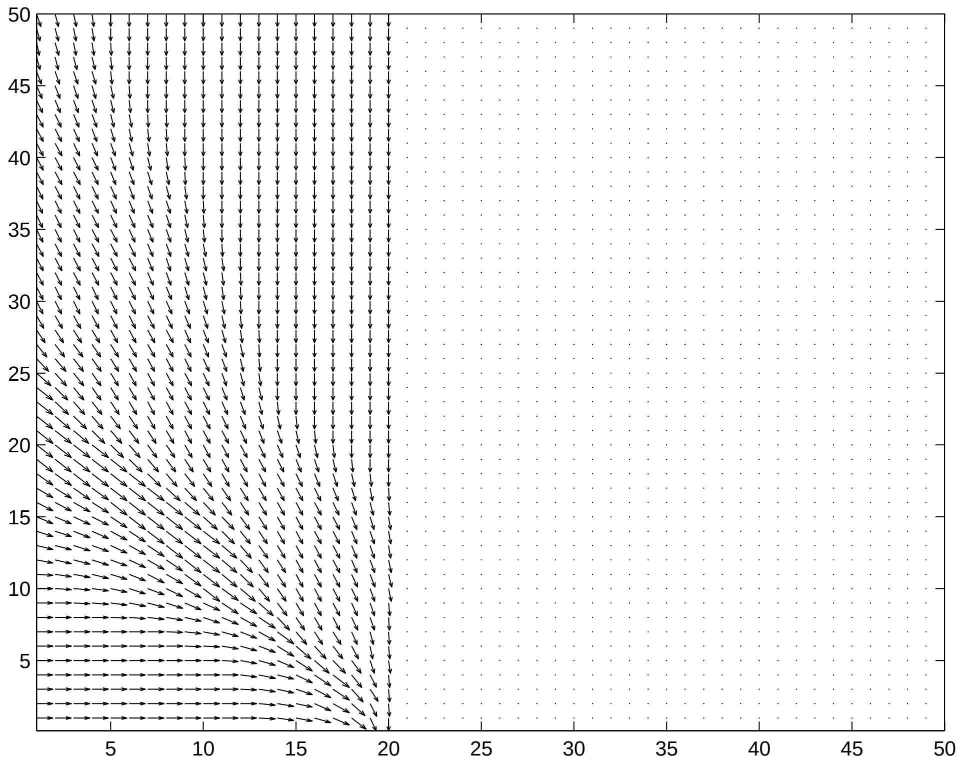

Figure 12.13: Plot of heading of the rat for all points in the cage. We have placed a wall in the lower part of the center of the cage. Notice that single goal path integration is unable to determine the path to the goal from anywhere behind the obstacle, although a path to the goal does exists. The solution to this problem is that we should now start to search for *subgoals.*

a goal in position $(20, 1)$. The 50×50 grid sized cage is without obstacles of any kind, and there is a landmark located in each corner of the cage. The arrows shows the direction the simrat believes the goal is in for each position in the cage. By visually inspecting the direction plot we observe that the simrat seems able to find its way back to the goal from all positions in the cage. In Chapter 18 we introduce a quantitative measure of how good this direction plot actually is.

Figure 12.13 shows a direction plot of the cage when an obstacle is introduced. The obstacle has the shape of a rectangle with its lower left hand corner in position $(21, 1)$, and its upper right hand corner in position $(31, 30)$. The edges of the obstacles are parallel to the corresponding edges of the cage. The goal is in position $(20, 1)$. We observe that the simrat is unable to find a direct path to the goal from positions behind the obstacle using single goal path integration. In order for the simrat to return to the goal, the simrat needs to introduce *subgoals* to the cognitive graph.

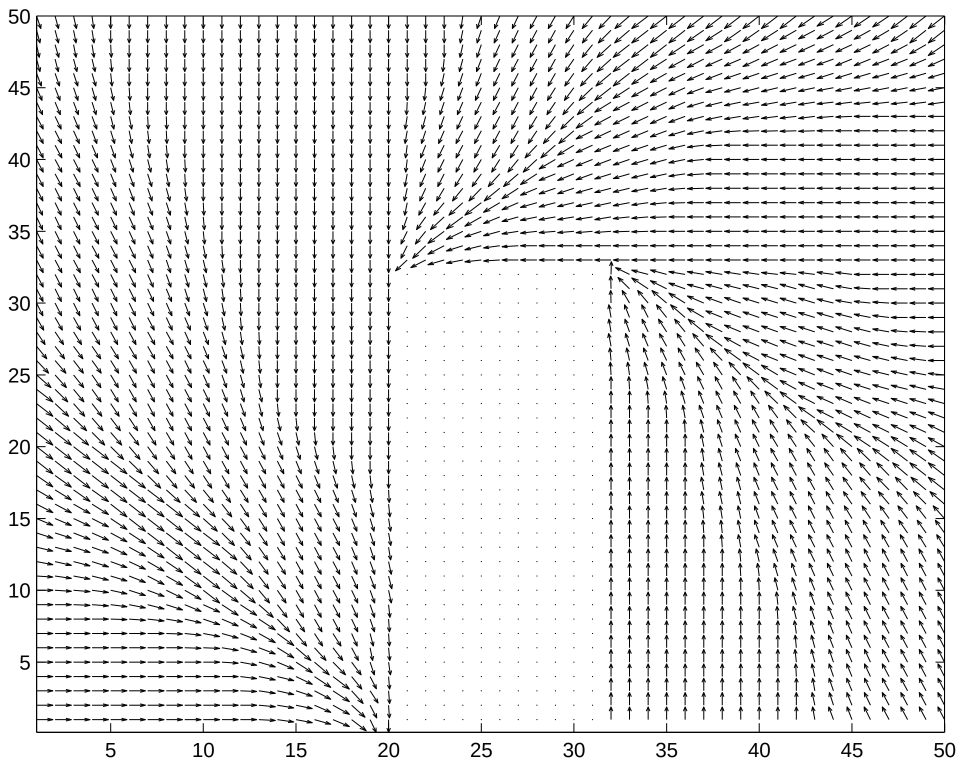

Figure 12.14: Plot of heading of the rat for all points in the cage. The simrat has now added subgoals at position $(20, 31)$ and $(32, 31)$. It is now able to return to the goal from all positions in the cage, using this subgoal.

Figure 12.14 shows a direction plot of the same cage as the one in figure 12.13. The simrat has now introduced several subgoals into the cognitive graph. There is one subgoal above the north-west corner, and one subgoal above the north-east corner of the obstacle. These subgoals ensure that the simrat is able to find its way back to the goal at position $(20, 1)$ from all positions in the cage.

Part IV

Experimental data and analysis

In Chapter 12 we introduced the Trullier-Meyer model for path integration. We also hinted that there were some empirical evidence from real rats to support it. In this part we examine data from real rats, and let our implementation of the Trullier-Meyer model use this data representation to learn a goal.

In Chapter 13 we give some background information about the areas of the rat brain that the Trullier-Meyer model deals with. We also introduce the "biological counterparts" of the abstractions Trullier-Meyer use.

Chapter 14 gives an account of how information is recorded from the areas of the rat brain discussed in Chapter 13 of the thesis.

After having collected the data from the rats in Chapter 14, we in Chapter 15 refine and analyze it and create the plots (or *maps* as we will call them) which will be used throughout the rest of this thesis when comparing data from real rats to the simrat.

In Chapter 16 we will take the data maps originating from the real rat experiments and adjust our internal representation in the Trullier-Meyer model to it. We wish to test whether the data representation we gathered from the rats is sufficient to guide the simrat in our simulation of directed behavior.

In Chapter 17 we again perform the same simulation done in Chapter 12, but we use the representation gathered from the rat as our simrat's internal representation. We present the result, and discuss them.

Now that we have created several simulations, we in Chapter 18 introduce a way of comparing output in order to provide descriptive statistic about how good a simulation really is.

In Chapter 19 we use probabilistic strategies developed earlier in the thesis to show sample plots of how the simrat behaves in the cage if we introduce an element of randomness into our simulations.

The final chapter, Chapter 20, is dedicated to summarizing our results, and to suggest possibilities for improvements and future work.

Chapter 13

Hippocampus and place cells

Hippocampus is also referred to as Ammon's horn, or *Cornu Ammonis*. This name refers to its resemblance in the human brain to ram horns (the ancient god Ammon who had rams head.) In humans, the Hippocampus is located at the floor of the lateral ventricle in the medial part of the temporal lobe. The entire extent of the Hippocampus is connected to a number of fore-brain structures.

Much of the present information known about brain functions has been learned from experimental lesions in the animals and brain injuries in humans. This is also the case with the Hippocampus.

Scoville and Milner [Scoville and Milner, 1957] have inspired many scientific studies with their study of a patient referred to as "H.M.". To alleviate his symptoms of severe temporal lobe epilepsy, he had his hippocampi, amygdala and parts of the temporal lobe surgically removed. After surgery, they discovered that H.M. was incapable of remembering new events and verbal content (known as *anterograde amnesia*). He also had *retrograde amnesia* (of things in the past) that was more severe close to the time of surgery. This specific finding lead to the hypothesis that the Hippocampus was largely responsible for storing new information.

Studies of other patients that had undergone similar surgery confirmed these findings. People with Alzheimers disease (which affects the hippocampal formation) exhibit spatial orientation deficits as one of their first symptoms. It has also been shown [Redish, 1999] that animals with lesions to the hippocampal formation exhibit deficits in different forms of spatial tasks.

A wealth of experimental and clinical data accumulated over the years has indicated that the Hippocampus must be involved in spatial orientation behavior. It is now a generally accepted hypothesis that the Hippocampus is a key component in the brain for the processing of spatial information.

For more details about the Hippocampus structure and functions, see [Redish, 1999].

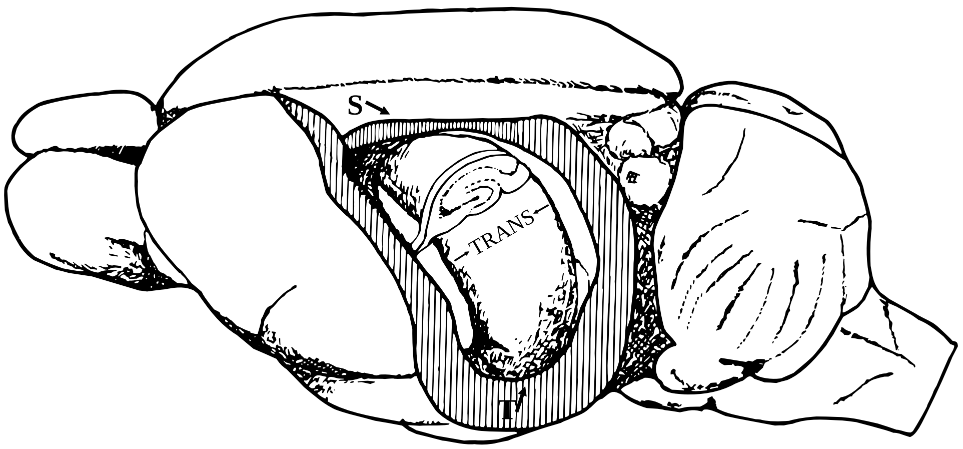

Figure 13.1: The Hippocampus in the rat brain. "S" marks the septal pole, "T" the temporal pole and "TRANS" the fibers connected to the Hippocampi, which run transverse to the fimbria-fornix. The Hippocampus is located in the medial part of the temporal lobe.

13.1 Place cells

O'Keefe and Dostrovsky [O'Keefe and Dostrovsky, 1971] presented in 1971 the first preliminary data that showed that there are cells in the Hippocampus area of the brain that only is active at specific places of the environment. These findings were important in that they eventually lead to the conclusions of what the Hippocampus area of the brain is used for. O'Keefe and Dostrovsky named these cells *place cells* and O'Keefe [O'Keefe, 1979] later introduced the following definition of them.

Definition 8 (Place cell)
"Cell whose firing rate or pattern consistently discriminate between different parts of an environment."

Observe that this definition uses the term *firing*. That a place cell fires means that it becomes active and emits a signal.

In the previous part we used the term PlaceCell to refer to an object in our simulation which was responsible for supplying a scalar value depending on the position of the simrat in the environment. We observe that our implementation of the PlaceCell also falls under definition 13.1.

In our simulations the PlaceCells supply a constant value when active. Real place cells, on the other hand, often exhibit a special firing pattern, known as complex-spike when active. We will use the following definition of a complex-spike firing pattern.

Definition 9 (Complex spike)
"A cell that sometimes has a spontaneously occurring burst of about 2-10

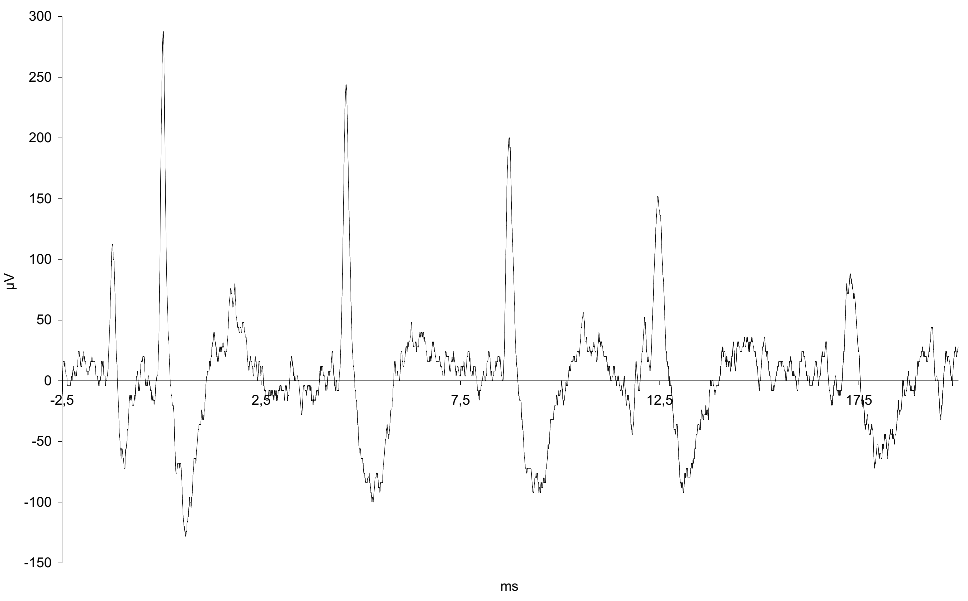

Figure 13.2: Example of a trace of a complex-spike firing pattern. The sample was collected using a digital oscilloscope with 100 kHz sampling rate. The x-axis shows time in milliseconds, and the y-axis voltage in microvolts.

action potentials of decreasing amplitude and increasing duration recorded extracellularly, with very short ($\leq$ 5 ms) inter-spike intervals."

Figure 13.2 shows one typical example of complex spike activity. We observe that when the place cell is active (or *fires*), it emits a series of signals over time, of decreasing amplitude. The value supplied by our simulation PlaceCells can be thought of as a scaled count of the number of spikes over a given time period.

Chapter 14

Experiment method and data acquisition

Since the reader may not have any prior experience with surgery protocol, we here present a short description of the steps involved in obtaining the experimental data for analysis. We hope that this chapter will help the reader understand the biological origin of the sampled data. In order to complete the analysis of the data in the next chapter, it is important to know how the acquired data relates to the actual experiments. We hope this will help us eliminate possible errors in the next chapter. Knowledge of the assumptions made during data acquisition can be vital for later analysis.

14.1 Rat surgery

For a detailed description of the surgery procedure, see [Thorsnes, 2001]. We here only present a short summary of the procedure.

14.1.1 Surgery

The rat was kept in a temporal animal housing in the lab for at least 5 days. If the animal during this period showed no symptoms to indicate any problems, it was prepared for surgery. The rat was given half of its daily portion of food before surgery.

The rat was deeply anaesthetised, its head shaved from behind the eyes to 3 cm behind the ears, and the head fixed to a stereotaxic apparatus. An incision 4 cm long was made in the skin with a scalpel, and the skull exposed (figure 14.1). The muscles surrounding the top of the scull were carefully retracted and the scull surface cleaned thoroughly. The scull was drilled open at several positions using an 1.8 mm burr, and three securing screws (with the sharp tip cutted off) were inserted in the skull to help support the implant (figure 14.2).

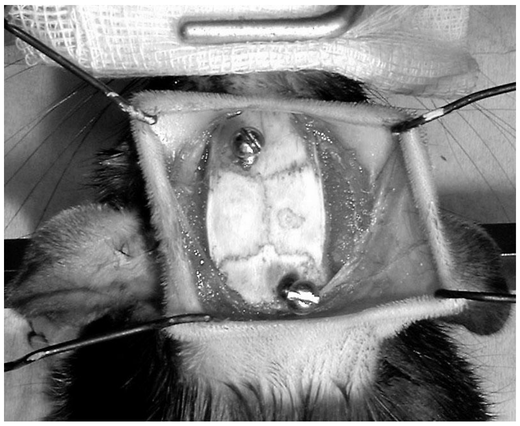

Figure 14.1: The skin retracted and the scull exposed. Screws has be attached to the skull to support the implant.

The electrode platform was placed in the position, and the canula with the ten micro-electrodes inserted into the brain. The electrodes were lowered to a depth of 1.3 mm from touchdown on the dura.

The openings in the skull were then filled, and the platform rods cemented to the skull using (biologically neutral) dental cement. The finished result can be seen on figure 14.3.

The rat was allowed at least ten days of recovery after surgery, during which the tissue surrounding the implant was smeared with anti-bacterial ointment.

14.1.2 Adjustment

After the rat had regained the weight lost after the implantation surgery, the descent of the electrode platform was started. The home cage of the rat was placed on the table in the recording room, and the electrodes on the implant attached by cable to a 10-channel FET preamplifier (see 14.2). All of the electrodes were then systematically checked for the presence of electrical activity resembling *complex-spike activity*. The signal displayed on a digital oscilloscope were used to determine the final electrode-waveform reference pair. If no units were found, the rat was detached from the cable, and the screw on the platform were turned by $\frac{1}{12}$ to $\frac{1}{8}$ of a rotation. This procedure was repeated for several weeks before stable units were found on several electrodes.

14.2 The testlab setup

The testlab is divided into two rooms, the testing room and the recording room, separated by a door. The testing room is where the actual experiments

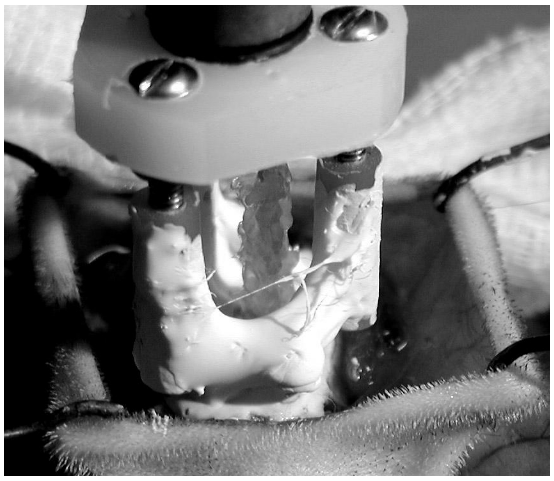

Figure 14.2: Attaching the platform to the scull. The platform can later be adjusted up or down by tightening the screws seen on top of the picture.

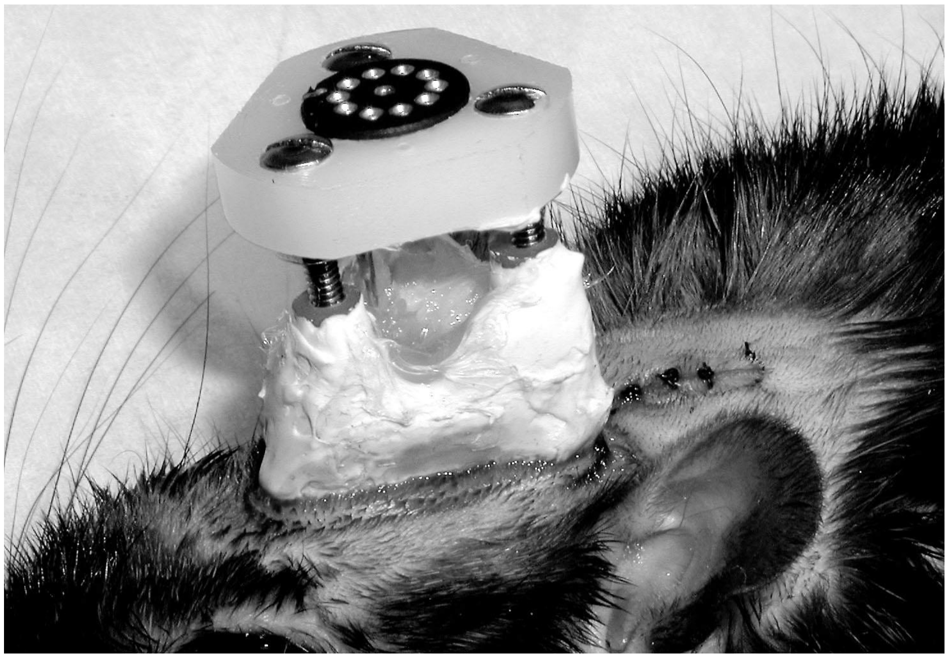

Figure 14.3: Picture of the rat after surgery. On top of the platform is a 10-pin jack which can be attached to the registration device.

take place, and signals acquired in the testing room during experiments are then sent to the recording room where they are stored on a computer for analysis.

The recordings we have used in this thesis were done in an open field in the testing room. This open field corresponds to the cage in our simulations. The open field is a 100 cm × 100 cm × 50 cm box without ceiling, elevated 20 cm above the floor. The walls of the cage were painted flat black to avoid reflections. The cage was placed in the center of a 100 cm×100 cm enclosure of black curtains. The curtains were suspended 250 cm above the floor. The walls above the curtains were painted flat black, as was the ceiling.

A black plastic disc, 80 cm in diameter was suspended from the ceiling, centered above the open field below. The disc is non-reflective, and is positioned at equal height with the top of the curtains. Six Halogen lights were distributed along the edge of the disc and the light cones adjusted to ensure even illumination of the open field. The Halogen lights were 20 W each, but the light intensity was dimmed with a potentiometer.

The custom made, lightweight recording wire from the rat was attached to a commutator in the center of the disc and suspended in a counter-weight system made in the lab. A black and white CCTV-camera was mounted to a modified camera-stand attached to the ceiling. The zoom and position of the camera lens was adjusted to cover the entire open field. An automatic pellet feeder was placed on top of the disc, and a hole made to allow the feeder to randomly drop small food pellets into the open field. The pellet feeder was connected to a control-panel in the equipment rack in the recording room, next door. This pellet feeder is activated if the rats stands still for a long period of time to stimulate movement. Figure 14.4 shows the connections and signal flow in the testlab.

The recording room contains a Analog/Digital conversion board which samples spike data at 300 kHz. Positional data is extracted from the video feed at 50 Hz.

14.3 A typical session

The rats are ordinarily kept in their cages placed on a series of shelves next to the door in the testing room. The rats cannot see the enclosure in which they will be tested from their home cage. When a rat is to be tested it and its cage is taken in to the recording room, and placed on a table. While the rat is still in its transportation cage, a wire is connected from its FET-implant to the registering equipment in the recording room. It is ensured that the recording equipment is able to pick up signals from several of the probes in the rat's brain.

The experiment is ready to begin if all seems okay at this point. The wire attached to the implant is removed, and the rat taken out of its cage and into

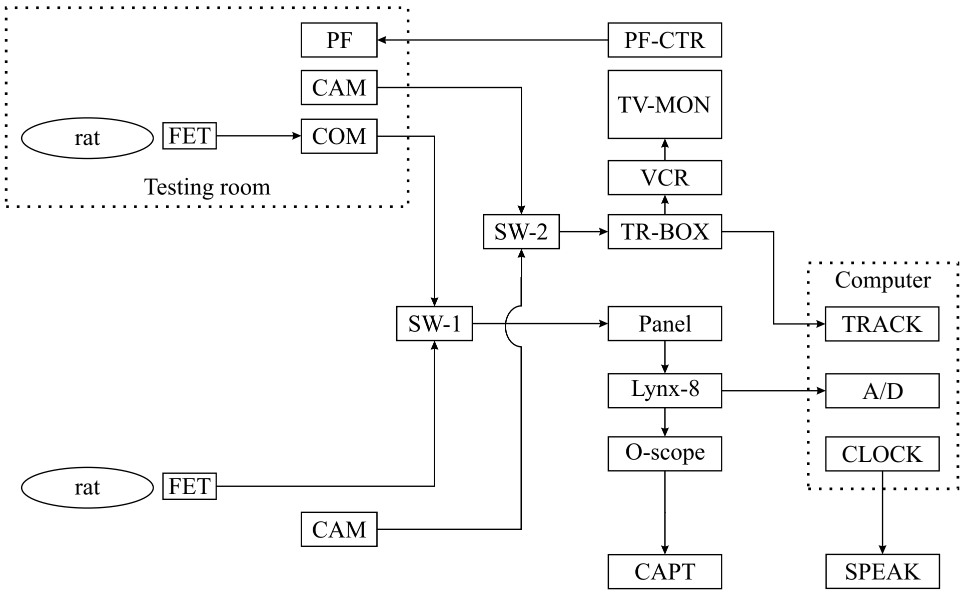

Figure 14.4: The connections and signal flow of the recording setup. Abbreviations: A/D, Analog/Digital conversion board; CAM, CCTV-camera; CLOCK, Clock-board; COM, Commutator; FET, Headstage with Field-Effect-Transistors; Lynx 8, Lynx 8 amplifier; O-scope, Digital storage oscilloscope; Panel, Panel for selecting electrodes and references; PF, Pellet Feeder; PF-CTR, Pellet Feeder Controller; SPEAK, Loudspeaker; SW 1, Switch for shielded cable input; SW 2, Switch for video signal; CAPT, Capture of oscilloscope traces on a computer; TV-MON, TV-Monitor; VCR, Video-cassette recorder.

the testing room. The rat is placed in the open field, which has been cleaned since the last experiment. The rat implant is attached to commutator in the ceiling by a wire. The curtains are drawn, and all researchers leave the testing room leaving the rat alone to explore the testing cage for 16 minutes.

The researchers watch the exploration from a monitor in the recording room. The exploration is also recorded by a standard VHS video recorder. A speaker is attached to the computer receiving data from the implant, and each time the computer receives a signal a clicking sound is heard from the speakers. At regular and frequent intervals the position of rat is recorded by a tracking box and forwarded to the computer. If the rat stops moving during exploration, the researchers press a button which controls the pellet dispenser in the cage. This causes small food pellets to be dropped at random places in the cage, and this usually encourages the rat to start moving again.

After 16 minutes the session is over, the rat is released from its wires and taken back to its home cage. The data collected by the computer is stored in a file. A log is also kept on written paper, and stored along with the video recordings.

In the context of this thesis we say that a session is a "good session" if the accumulated time the rat has spent in each part of the cage is about the same, and if one or more of the probes show complex spike activity in certain parts of the cage. We have only used "good sessions" in our analysis.

14.4 The data

The data acquired during a session is stored in a binary ".UFF" file format.

Each datafile is named after the ID number of the rat, and a session ID number. The UFF-file will contain all inputs received from the rat, as well as the positional data gathered by the monitoring camera in the cage.

All inputs which are gathered during the session, are marked with a *time stamp*, which indicates the time this input was received. Each input received at a given time is referred to as an *input event*. It is interesting to note that the data-acquisition software uses an abstraction analogous to our input-events in our simulation.

In our analysis of a rat sessions we will be concerned with only two types of input events; those describing cell-firing in the brain (*spike events*), and those describing the rat's position (*position events*). As mentioned above, these input events are marked with the time each input was recorded (a *time stamp*).

Chapter 15

Data analysis

As mentioned in the previous section, the data acquisition software will export all the results of each session to a ".UFF" file format. Each input event will be stored in the UFF-file as a *record*. This UFF-file will contain a large number of records, each consisting of several *fields*.

We will next describe the two types of records which we will utilize throughout the rest of this thesis.

15.1 Positional data

For each input event which describes movement in the cage (positional data), a record in the UFF-file is created. This record will contain several fields. Among the fields contained in this record are the time stamp field, x-position field and y-position field.

The x and y position fields will contain the x and y position of the rat at the time indicated by the time stamp. Ordinarily, the position of the rat in the cage is checked and logged every 200 millisecond.

Similarly, all other events which are received will be stored in different records.

15.2 Spike data

We use the term *spike data* to describe all events coming from the various probes implanted in the rat brain. The spike data record contains the following fields: a time stamp field, a probe number field, an optional cluster number (more on this later) and 32 discrete samples of the signal received on this probe number. We say that the 32 samples form a *waveform*.

When a place-cell (which is close enough to a probe) in the brain becomes active, the probe receives a signal from this place-cell. This signal is measured in microvolts(μV) and forwarded to the data acquisition software. The software creates a record in the UFF file for this input event. The time this signal was received on is stored in the time stamp field, the probe number is stored in the probe number field, and 32 discrete voltage-samples from the signals are saved in the last 32 fields in the record. The 32 samples will be distributed evenly in time after the signal was received. How far the samples are apart in time will be determined by the total length of the signal.

The sampling software is also able to sample several consecutive spikes (or *complex spikes*) correctly. Each spike will be stored in a separate record. For instance, the signal showed in figure 13.2, will be stored as 5 separate spikes in 5 separate records.

The cluster number field will at this time not be set.

Actually the term "spike data" comes from the graph which can be created by plotting the waveform (the 32 samples), where time is along the x-axis and the voltage, is along the y-axis. Typically, this signal will rise quickly once the signal begins, and the drop equally fast, once the signal has reached its peak, thus forming a "spike". A sample of several such consecutive spikes can be shown in figure 13.2.

15.2.1 Clustering

Unfortunately, the data collected from each probe may actually come from several different cells[1]. This is because the probe itself may be (and almost certainly is) in direct physical contact with several cells.

But since we would like to have information about each individual cell we need to perform some sort of filtering, to differentiate between signals from different cells received on the same probe.

Our goal is therefore to separate signals from the same cell into one group, or *cluster*. This will mean that we, for each probe, may have several clusters. We hope that all signals in each cluster originates from the same cell.

The filtering of a signal into several different clusters is done using a *principal component* analysis method implemented in the commercial software package "Off-line Sorter" by Plexon Inc (www.plexoninc.com). A brief discussion follows about how this software works.

The basic assumption is that signals originating from the same physical cell will have "similar" waveforms. The term *template-signature* is important when explaining what we mean by "similar".

From a set of signals received on one probe, the software is able to create a few *template-signatures*. Each signature has some key features that is shared by many of the signals. The assumption is that signals coming from the same cell will have many of the same features. With some help from the user, the software is able to assign each of the signals to the one template-signature that is the most similar. At the end of the filtering, each signature

[1]Note that there are several different types of cells in the rat brain. A few examples are the theta cells and place cells. The probes may pick up signals from either ones of these.

is assigned a positive cluster number. We hope that all signals belonging to one cluster (identified by a positive number) originates from the same cell. A few signals, however, does not fit any of the signature templates. These are assigned to cluster number 0 and will, in the rest of our analysis, be discarded as noise. For a more detailed description of the principal component analysis, see [Lewicki, 1998].

The software saves the results back in a ".UFF" file format. After this clustering is done, there exists a custom made computer software package called "Sess-anal" developed by Dr Matt Stead, New York, which can be used for calculating various parameters from the clustered data. Eirik Thorsnes has done some further development on this package, that allows it to export the data as (ASCII) text, which is needed for the rest of our analysis.

The text output, is further refined and parsed by a series of Bash and GNU Awk scripts created for this thesis. The final analysis is done with Matlab.

Before the Matlab analysis, the positional data is available in a text file on the following form

```
[timestamp] [x] [y]
```

and the spike data is available as follows

```
[timestamp] [probe]_[cluster] [sample1]...[sample32]
```

15.3 Maps

In order for us to use the data, we need to be able to present it in a understandable manner. In this section we show how to convert the raw data into 2D maps[2], which can be used for analysis throughout the rest of this thesis.

Originally, the positional data was sampled at 256×256 pixel grid. Unfortunately, it is difficult to utilize all of this tracking resolution in practice. This is, among other things, because of the problem of perspective (a rat standing on its rear legs might be classified by the software as actually outside of the cage). Also, if the resolution of the cage is too fine, other factors such as the rat shaking its head or chewing its food may begin to show up in our positional data, as movement. Clearly, we do not want this.

We will use the same resolution as in [Thorsnes, 2001] and throughout the rest of this thesis the original 256×256 grid map is reduced to a 50×50 grid. We will use this resolution both for the plots, and later on, for the simulations we intend to make and use in this thesis. This reduction in spatial resolution has the additional advantage that it increases the number

[2]By map we mean a two dimensional grid (or matrix). We use the term map, since the maps will always be related to positional data. Typically, each element in the map will contain information about something that occurred at the corresponding position in the cage.

of samples per grid position. The rat cage is originally 100 cm$\times$100 cm, so if we use a resolution of 50$\times$50 grid positions, then each grid point will correspond to a 2 cm$\times$2 cm area in the real cage.

In the next sections of this chapter we will show a few different ways to represent, summarize and visualize the spike and positional data gathered so far from the rat.

The plots in the next sections were generated from rat h11, session 26161, probe 5 and cluster 1.

15.3.1 Time-maps

The time-maps use information about the rat's position and the time stamps to show how much time the rat has spent in each grid position in the map.

Imagine attaching a pen to the rat. As the rat moved about in the cage, the pen would be pulled along the floor of the cage, creating a "track" where the rat has been. Where the rat has been more often, the lines would become thicker, and where the rat has only been a few times, the lines would be very faint.

A time-map is very similar to this rat movement outline in the floor of the cage. From the positional data, we will calculate how much time (in milliseconds) the rat has spent in each grid position in the cage. This is simply a matter of taking the difference in arrival-time and leave-time for each grid position, and then adding up all these differences for each grid position. We call such a map, a *time-map*.

15.3.2 Rate-maps

A rate-map will show information about in which parts of the cage a given cell is active. We will create *one rate-map for each cluster from each probe.* In order to create a rate-map, the spike data must therefore first be clustered. This procedure was described in Section 15.2.1.

In order to create a rate-map, we must first create a spike map. Since we will create only one rate-map (and only one spike map) for each cluster on each probe, we now consider only the spike events originating from one particular cluster on one particular probe.

As the rat moves around in the cage, the probe will register firing of the cells at various times. Each time one such firing occurs, and a spike is registered, the positional data is consulted to determine what the position the rat was in at that time. This is done by cross checking the time stamps in the positional data with the spike data. We then count each time a cell in a cluster on a probe fires on each grid position. We call the resulting map of spike counts on each position a spike map.

A rate-map is created by doing an *element-wise* division between the spike map and the corresponding time-map. A rate-map will show if a cell

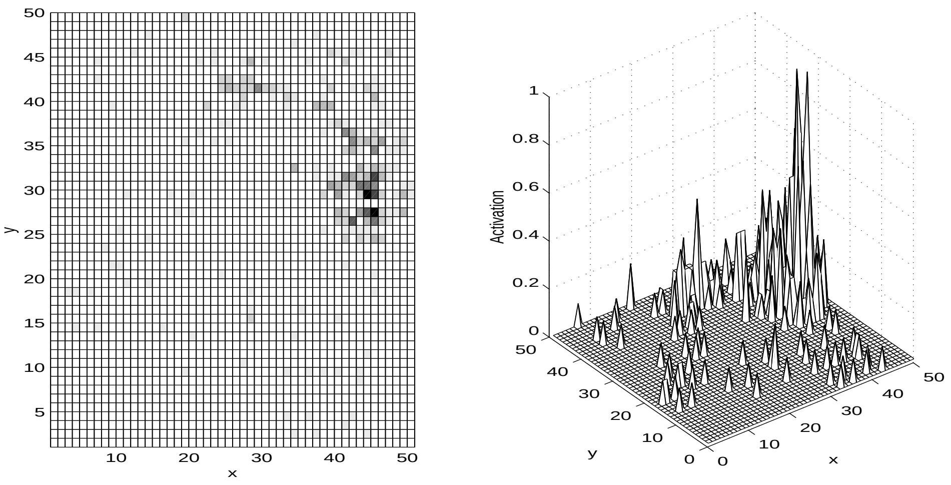

Figure 15.1: Sample of rate-map for rat h11, session 26161, probe 5 and cluster 1. Notice how this cluster seems to have a "home-base", where the cell has fired the most, in the area around pixel (33, 25).

has fired many times on a given position, relative to the time the rat has spent on that position.

If this rate of fire is high, this could be a sign that this cell is a place-cell for that position.

15.3.3 Smoothed rate-maps

A smoothed rate-map essentially shows the same information as a rate-map, but it attempts to "blur" out some of the irregularities which we typically find on a rate-map.

To create a smoothed rate-map one starts with a rate-map, as described in the previous section. One then performs a two dimensional convolution on the rate-map, with a average kernel. The value of each grid position in the smoothed rate-map will be equal to a weight on the corresponding grid position in the rate-map, plus a weighted sum of its neighboring values.

One might argue that it should not really be necessary to create a smoothed rate-map, since it does not really provide any more information than the ordinary rate-map, discussed in section 15.3.2. Although this is true, one should remember that the rate-maps are created from samples taken over a relatively short time period (the order of 16 minutes).

There are however practical difficulties involved with extending the time the rat has for exploration in the cage beyond the 16 minutes the rat currently has available. For instance, the rat will become tired as time progresses, and once it feels that it has explored the cage "enough", it will be come increasingly more passive and less willing to explore.

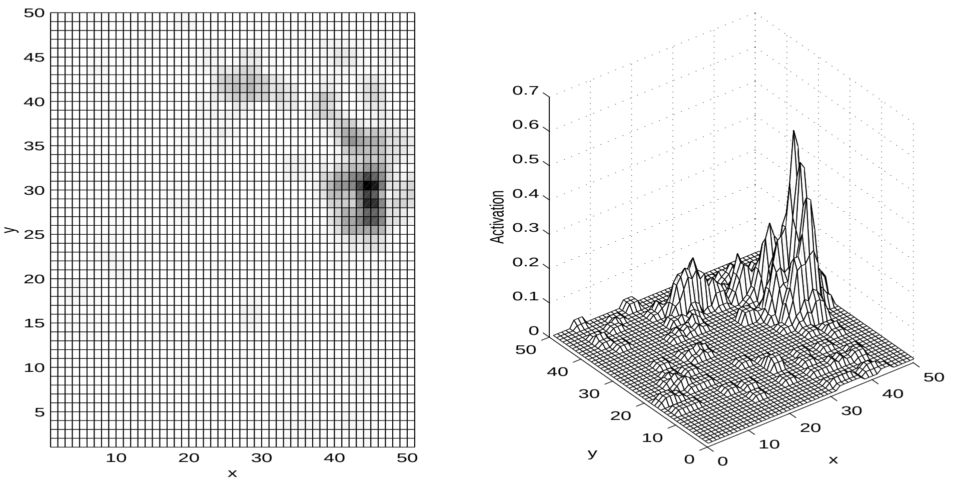

Figure 15.2: Sample of smoothed rate-map made from the rate-map in fig 15.1.

15.3.4 Field-maps

A field-map shows information about where a cell is active in many consecutive grid positions. This type of map can be used in determining the location of what we believe to be a place-cell.

Consider a rate-map. The value of a field-maps grid position is 1, if the corresponding grid position in the rate-map has k active neighbors, and 0 otherwise. By "active neighbors" we mean grid positions that are adjacent to the grid position in question, and that has a value higher than a given threshold value.

This map will discriminate against small groups of active grid positions. Larger groups of consecutively active grid positions will be shown. Our hypothesis here is that small groups of active grid positions will most likely be noise, and relatively large areas of active positions must come from a real place-cell.

Later on the field-maps will be valuable when we wish to create an initial guess of where a place-cell might be situated.

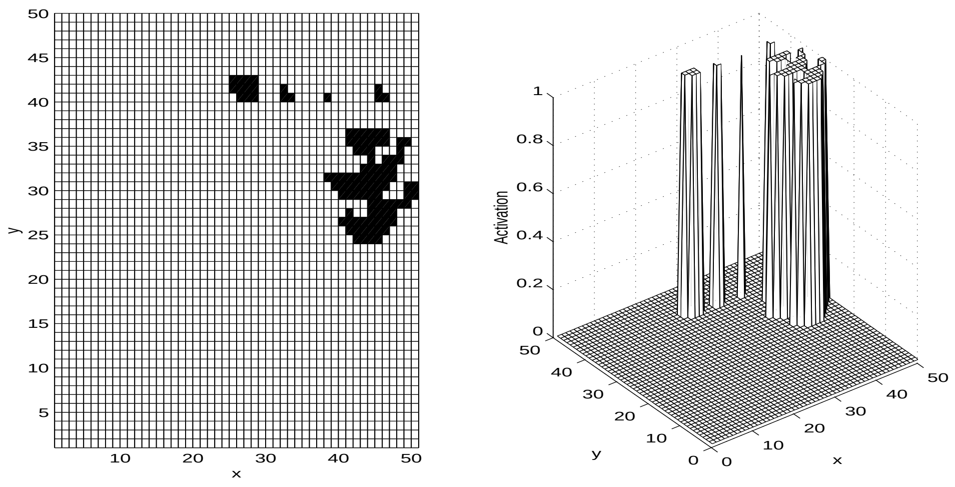

Figure 15.3: Sample of field-map made from the rate-map in fig 15.1.

Chapter 16

Fitting our model to real data

In the previous chapter we showed how to create-maps from the clustered data from the rat experiments. Each of the rate-maps we created described the firing pattern of one cell relative to the rat's position in the cage. Our hypothesis in this chapter will be that if the cell in question is a place-cell, then the cell will fire only when the rat is in certain positions in the cage, and that this firing pattern can be described by an activation function.

In this chapter we will attempt to see if an activation function can successfully have its parameters adjusted so that the activation function becomes a good approximation of the original firing pattern of the cell described in the rate-map.

16.1 The problem defined

From section 15.3.3 we have for each place-cell created a smoothed rate-map. Each smoothed rate-map shows where one place-cell in the rat brain is active relative to the rat position in the cage.

As stated in Chapter 12 the PlaceCells in our simulations models the real life place-cells found in rats. Both provide a value which indicates if they recognize a position in the cage.

We create a rate-map from the activation function of our simrat. This is done by evaluation the activation function for each position in the cage. Such a map is shown in figure 15.1. By changing the variables that our activation function depends on we are able to change the "shape" of the activation function, and thereby the map that can be created from that function.

We hope that by adjusting the parameters of the activation function we are able to create a map which is relatively similar to the one we found in the rat. If we are able to do that, then we say that our PlaceCells are a good model of the real place cells.

In order to do that, we must first define exactly what we mean by "similar". We will use the element-wise difference between the two maps, and an appropriate norm to express similarity.

We let $f(X)$ denote the error (or difference) function, and define it as follows.

$$f(X) = \sqrt{\sum_{u \in \mathrm{Cage positions}} (p[u] - g(X;u))^2} \tag{16.1}$$

where p is the smoothed rate-map found in Chapter 14, and $g(X;u)$ is our activation function, both evaluated at position $u = \begin{bmatrix} \hat{x} & \hat{y} \end{bmatrix}$ in the cage. For each position, u, in the cage, we take the difference between the smoothed rate-map value at that position, $p[u]$, and our activation function evaluated at that position, $g(X;u)$. Notice that the term `Cage positions` in our formula refers to all the discrete grid positions in the cage. The exact number of grid positions equals the resolution. If the resolution of the cage is wh, the sum runs over wh differences, one for each grid position in the cage.

We let X be our variables that we wish to adjust so that the difference between the two maps becomes as small as possible. Notice that if the two maps, $p[\cdot]$ and $g(X;\cdot)$, are equal, then the difference between each element in the two maps are 0, so therefore $f(X)$ will also be 0. If the maps are not equal, $f(X)$ will be greater than 0. Ideally, we would want $f(X)$ to be as small as possible, since that would mean that the two maps are as close as possible.

We let our variables, X, be on the following form

$$X = \begin{bmatrix} x & y & \omega_1 & \cdots & \omega_l \end{bmatrix}^T \tag{16.2}$$

where x and y represent the position that this place cell should represent in the cage. One interpretation of the values x and y, is where the "peak", or maxima, of the activation function will be[1].

ω_i is the weight applied to the distance to landmark $i \in 1, 2, \ldots, l$, out of a total of l landmarks. In our calculations we will always assume that there are only $l = 4$ landmarks, one in each of the four corners of the cage.

It is very difficult to guess exactly what landmarks the real rat actually use in the cage. But experiments with simulations with different number of landmarks, as well as landmarks in different positions has produced very little difference from our current setup, as long as the cage is without obstacles. Therefore we guess that for the rat the corners are the most easily recognizable landmarks in the cage, and that they are the only ones that the rat use.

[1]See Chapter 12 for more information about this, with examples.

We define our activation-function in the following way

$$g(X;u) = g(\begin{bmatrix} x \\ y \\ \omega_1 \\ \vdots \\ \omega_l \end{bmatrix};u) = \exp(-\sqrt{\sum_{i=1}^{l} \omega_i (d_i(u) - d_i(\begin{bmatrix} x \\ y \end{bmatrix}))^2}) \tag{16.3}$$

where $d(\cdot)$ is a function which returns a column-vector with l elements, containing the distances from the position represented by its arguments in the cage to each of the l landmarks. $d_i(\cdot)$ denotes the i'th element of this vector, where $i \in 1, \ldots, l$.

Observe that if the sum of the difference (between the two distance vectors squared) is 0, then $g(\cdot)$ will return 1. This will happen if the two distance vectors have exactly the same values. Otherwise $g(\cdot)$ will return a value between 0 and 1. The larger the sum is, the closer the value of $g(\cdot)$ will be to 0.

In the case where we only have $l = 4$ landmarks, one in each corner of the cage, $d(\cdot)$ is defined as follows

$$d(u) = d(\begin{bmatrix} \hat{x} \\ \hat{y} \end{bmatrix}) = \begin{bmatrix} \sqrt{\hat{x}^2 + \hat{y}^2} \\ \sqrt{(w - \hat{x})^2 + \hat{y}^2} \\ \sqrt{(w - \hat{x})^2 + (h - \hat{y})^2} \\ \sqrt{\hat{x}^2 + (h - \hat{y})^2} \end{bmatrix} \tag{16.4}$$

where w and h are the width and height of the cage, respectively. We observe that regardless of the values of $\hat{x}$ and $\hat{y}$, the distances that are returned from the $d(\cdot)$ function, are always positive.

Notice that our activation function $g(\cdot)$ is very similar to the one described in equation 12.1. The only difference is that we have now introduced a value, ω_i, for each landmark, i, instead of weighing all landmarks with the same value, σ. We observe that if $\omega_i = \sigma$ for $i \in 1, 2, \ldots, l$ then the two equations are equivalent. In the following section we will discuss g in more detail.

16.1.1 Discussion of the parameters

In our above model we have introduced 6 variables for each place cell, $\begin{bmatrix} x & y & \omega_1 & \cdots & \omega_4 \end{bmatrix}$, assuming that we have only 4 landmarks represented in the cage.

The parameters x and y are also used implicitly in equation 12.1 to represent the position that the place cell should learn. In both equations these variables have a very direct affect on the place cells since a change of $+1$ to x would result in the maximum (as well as the rest) of the activation function is moved $+1$ in x-direction on the map. The y variable has the same property in the y-direction on the map. Although values for x and y, that are outside the resolution of the grid (40×40 in our simulations, i.e. $x, y \notin [0, 40]$) would move the maximum of the activation function outside the cage, we still allow it, since it might actually be more optimal to do so. Additionally, there are a few examples in our sampled data, where the position of the place cells actually seems to be outside of the map, so that we only only see a small part of the side of the place cell peak.

Strictly speaking, the three variables used in equation 12.1 would be enough to get reasonable results in our model, but after noting that the real place cell activation functions are not always symmetrical around their respective maxima, we introduce the weight ω_i to the distance to landmark i, instead of a the same weight to all of them, σ. These variables will allow us to have more control over the exact shape of the activation functions, and their symmetrical properties. We require that the values of ω_i be greater or equal to 0, so that our $g(\cdot)$ function is positive semidefinite[2] for all allowed values of the variables. This is requirement will help make our activation function easier to analyze.

16.1.2 The final optimization problem

Using the requirements on the variables found in the previous section, and noting that we would like to minimize the difference (or error) between the two maps, this gives us the following minimization problem.

$$\begin{gathered} \min f(x, y, \omega_1, \ldots, \omega_4) \\ \text{subject to} \\ \omega_1 \geq 0 \\ \vdots \\ \omega_4 \geq 0 \end{gathered}$$

Solving this minimization problem is done relatively easy with the following Matlab program 16.1.1.

16.2 Results

From the output given to us by the minimization problem in the previous section, we can plot the results by evaluating the activation function for each discrete positions in the cage.

Let X_i be the optimal variables for smoothed rate-map i. The plots below shows $g(X_i; \begin{bmatrix} x & y \end{bmatrix}^T)$ evaluated for all positions in the cage, $(x, y) \in$

[2]That a quadratic function, $g(\cdot)$ is positive semidefinite means that $g(\cdot) \geq 0$ for all allowed values of its variables.

Program 16.1.1 From minimize.m

```
function [x] = minimize( filename )
  data = ratdata( filename );

  xdata = data( :, 1:2 );
  ydata = data( :, 3 );

  %            User variables
  %    [  x     y   w1  w2  w3  w4]
  ub = [ inf  inf inf inf inf inf];
  x0 = [  20   20  1   1   1   1 ];
  lb = [-inf -inf  0   0   0   0 ];

  options = optimset( 'Display', 'Iter');
  x = lsqcurvefit( @f, x0, xdata, ydata, lb, ub, options );
```

Program 16.1.2 From f.m

```
function [alpha] = f( X, xdata )
  for row = 1:length( xdata )
    % Cage position
    x = xdata( row, 1 );
    y = xdata( row, 2 );

    alpha( row ) = g( X, [x y], [50 50] );
  end
```

Program 16.1.3 From g.m

```
function [alpha] = g( X, pixel, size )
  % Position of PlaceCell
  x = X( 1 );
  y = X( 2 );

  % Weight for distances
  w = diag( X( 3:6 ) );

  v = d( pixel, size ) - d( [x y], size );
  alpha = exp( -sqrt( v' * w * v ) );
```

Program 16.1.4 From d.m

```
function [distance] = d( position, size )
  x = position( 1 );
  y = position( 2 );

  width  = size( 1 );
  height = size( 2 );

  distance( 1 ) = sqrt( x^2 + y^2 );
  distance( 2 ) = sqrt( (width-x)^2 + y^2 );
  distance( 3 ) = sqrt( (width-x)^2 + (height-y)^2 );
  distance( 4 ) = sqrt( x^2 + (height-y)^2 );

  distance = distance' / sqrt( width^2 + height^2 );
```

$[1, 40]$. The leftmost columns shows the original smoothed rate-map, and the rightmost column shows our approximation of the map.

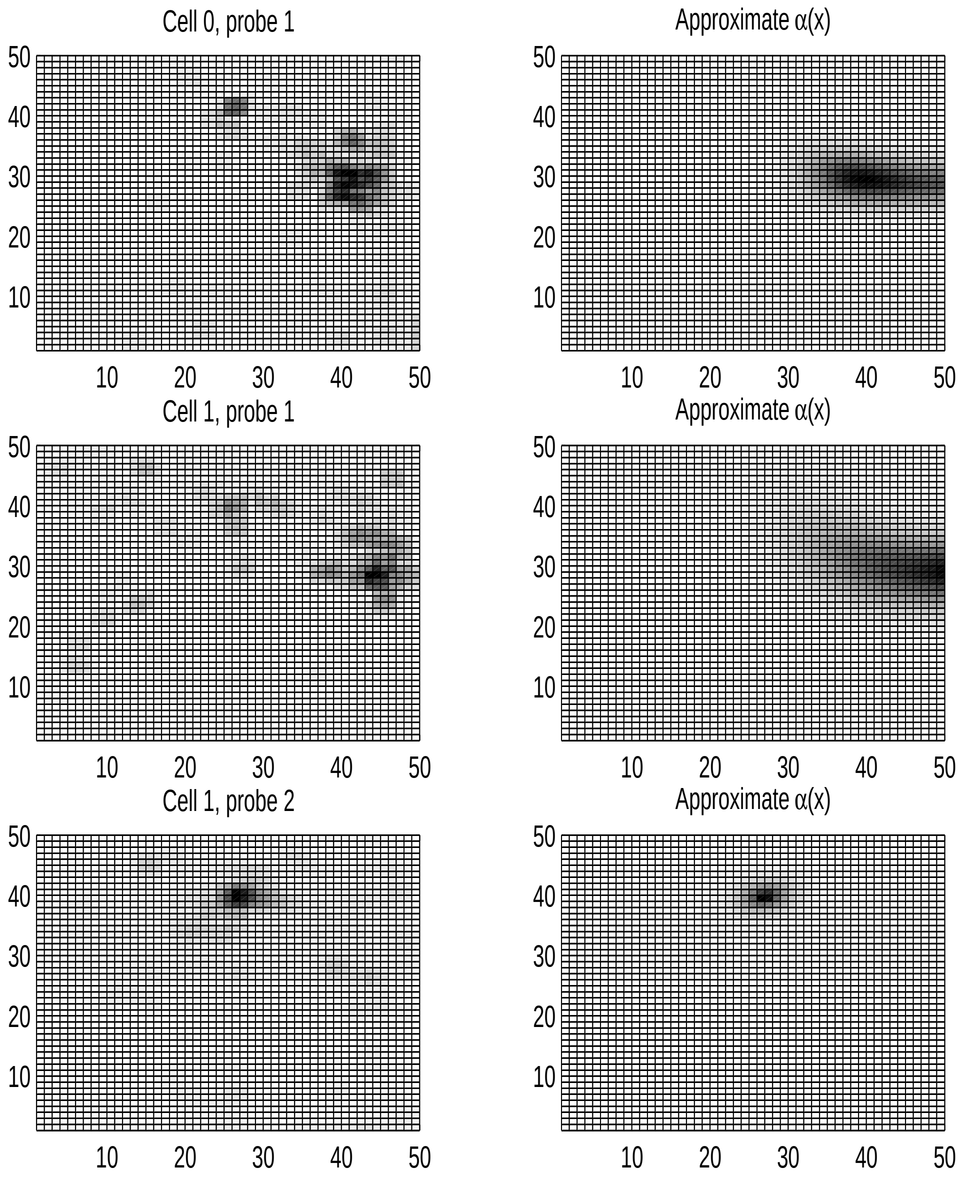

Figure 16.1: Comparison between real rat data, and the same plot created from the optimal solution to our optimization problem.

Chapter 17

Path integration with real data

In Chapter 16 we found the activation function which was the most similar to the rate-maps from the real rat. In the previous chapter we found values which gave the activation function a similar shape to those found in real rats. In this chapter we will attempt to use these values as our simrat's internal representation of the environment. We hope that this representation is enough to guide the rat back to a predetermined goal in the cage.

17.1 Field-maps

Remember from Chapter 12 that the Trullier-Meyer model for path integration is only able find its way back to a given position in the cage when it has active place-cells representing the position it is currently in. In order to determine whether a place cell is active, we need to check if its activation value is higher than a given activation threshold, t_α. A field-map is one way to plot where a given place-field is active. See Section 15.3.4 for more about the field-map.

In our simulations we need to determine whether *any* of all the place cells we have represented are active at a given position. This is done by creating a field-map F_i for each individual place-cell i, and then taking an element-wise OR operation between all the field-maps to create the final map F_{all}, containing information about where *any* of the place cells in *any* of the maps are active.

Since all elements in a field-map is either 0 or 1, we can perform the above mentioned OR operation by using, for each grid position, the element-wise max operator between all the field-maps. In other words,

$$F_{all} = \max(F_1, F_2, \ldots, F_n) \tag{17.1}$$

given that there are n field-maps.

For rat h11, session 11262, we have 9 rate-maps from the real rat showing 9 place-cells. We have approximated these rate-maps by 9 activation

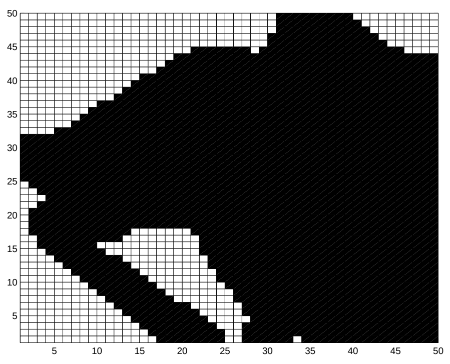

Figure 17.1: F_{all} plotted with $t_\alpha =$ 0.001

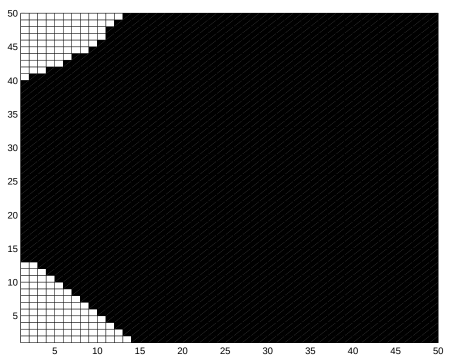

Figure 17.2: F_{all} plotted with $t_\alpha =$ 0.0001

functions. From these activation functions we create 9 field-maps. By performing the above mentioned OR operation between all the 9 corresponding field-maps, and using $t_\alpha = 0.001$ we created the plot shown in figure 17.1. Notice that this map does contain a few areas where not any of the place cells are active.

If we lower $t_\alpha = 0.00001$, we see in fig 17.2 that a larger portion of the cage has one or more place cells that are active. Ideally we would want to have as much of the cage as possible covered by an active place cell but there are, as we will later see, some serious drawbacks of lowering t_α in this way.

17.2 Deterministic results

By using our model fitted to the data from real rats, we were able to produce the results presented in fig 17.3 and 17.4.

Notice in fig 17.1 and 17.2 how there are a few areas in which there are no active place cells. These areas are white in the two plots. In this section we will compare the field-maps to the final path-integration maps shown in fig 17.3 and 17.4.

17.2.1 Handling silent areas

The "white spots" shown in the field-maps has an unfortunate affect on the path-integration maps. We call these white spots, where no place cells are active, *silent areas.*

Consider fig 17.1 and fig 17.3. We see that in the silent areas, the cognitive graph provides the rat with no direction in which to move in order to get to the goal. So what should the simrat do in this situation? As mentioned in Chapter 12, there is currently no good solution for this problem. The simrat should start to move about in an random fashion, and hope that it eventually finds an area where some of the place cells again are active. Upon doing

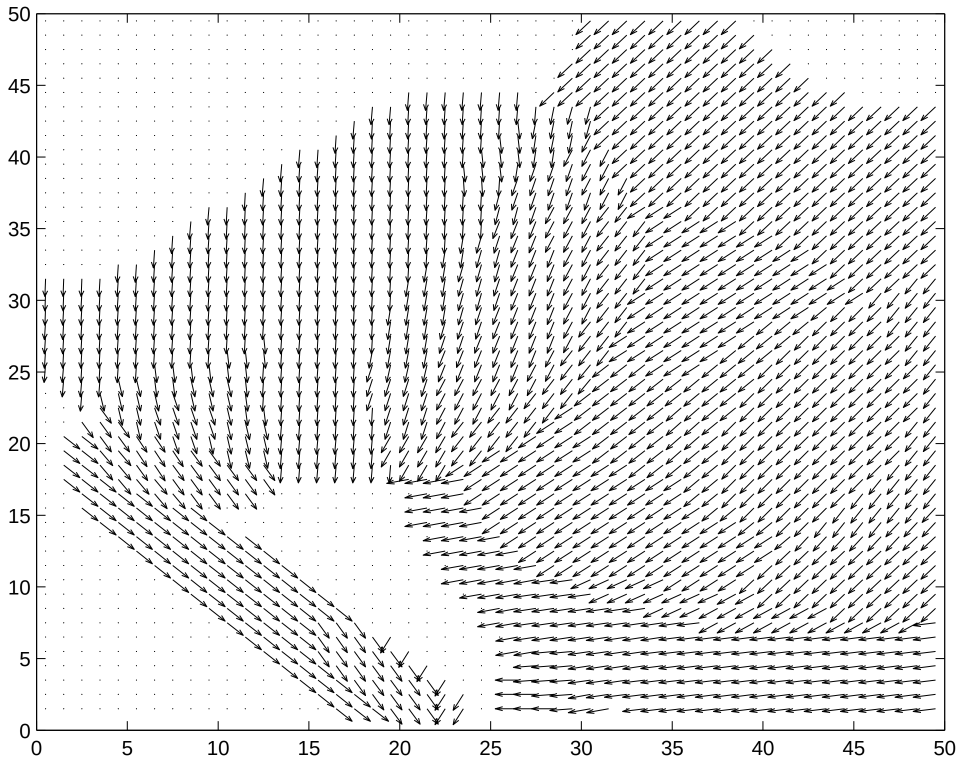

Figure 17.3: The direction the simrat would follow in order to return to grid point $(20, 1)$, using data from a real rat. We have here used $t_\alpha = 0.001$

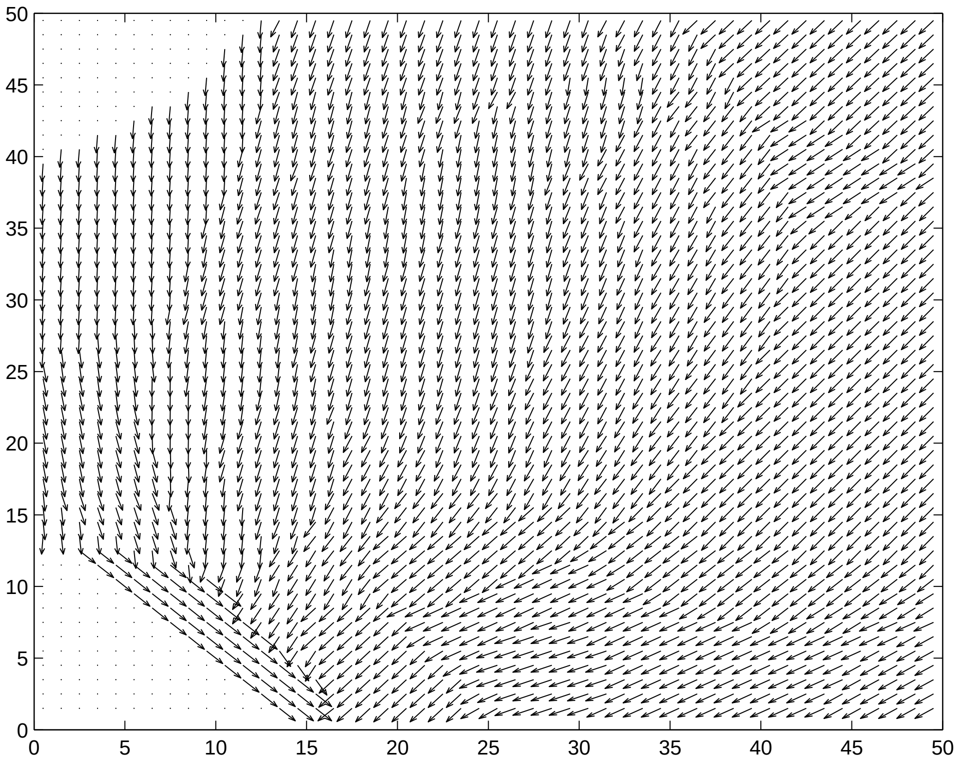

Figure 17.4: The direction the simrat would follow in order to return to the goal at grid-point $(20, 1)$, using data from a real rat. We have here used $t_\alpha = 0.00001$. Observe that if the simrat starts in position $(19, 1)$, it will never reach the goal. The same is true for many other positions in the cage. This is a problem which comes from lowering t_α this much.

so, the simrat should recruit a new place cell at that position, and hope that this new place cell will also be active in part of the silent area which it came from, so that the next time the simrat walks into this silent area, the silent area will be smaller, and it will be easier to find the way out. There exists some physiological evidence to suggest that real rats also dynamically creates new place cells when in a "silent area". See [Thorsnes, 2001] for details.

17.2.2 Inaccuracies and errors

In fig 17.2, we have lowered the threshold value t_α. We have done this hoping to find that a larger area of the cage will have active place cells, and that the rat therefore should be able to return to the goal from more positions in the cage.

From fig 17.4, we see that this is only partially true. Although the lowering of t_α actually does provide the simrat with a direction in which to go in order to return to the goal for a larger area of the cage, we also see that the direction is not always as correct as we would want it to be. For instance, if the simrat starts in position $(19, 1)$ then it actually moves away from the goal. This is because the lowering of t_α will effectively increase the size of each place cell's *active area*, and it is not always given that the direction dictated by that place cell which has just "expanded" is also good in the "expanded area". This is especially true when the rat is near the goal, where there often are large differences in the direction associated with different place cells. If however, the distance to the goal is large, one would expect the directions associated with the place cells to be more or less pointing in the same direction.

This suggests that there might be something to be said for having t_α be dependent on the distance to the goal, in some manner. This would cause areas around the goals to be much more heavily populated with place cells, and areas far from the goal much less populated, thus providing us with a a way to reduce the total number of place cells.

Indeed, there is some evidence ([Lever et al., 2002]) from real rats to indicate that areas where there are large changes in the environment (such as many obstacles) will be represented by more place-cells, and that areas with little change (such as open areas), is represented with fewer place-cells.

Chapter 18

Evaluating path integration

In the previous chapter we let the simrat navigate the cage using a PlaceCell distribution taken from real rats. We were able to produce plots which showed in which direction the simrat believed the goal was, for all positions in the cage.

In this chapter we will introduce a measure of how good a direction map is relative to a *shortest-path* direction map.

18.1 The shortest-path direction map

For each position u in the cage, we define the shortest-path direction to be the direction in which the shortest path to the ultimate goal g of the path integration is. An shortest-path direction map is then the shortest-path direction evaluated for each of the positions in the map. If the cage is without obstacles, the shortest path to the goal is a direct line and the direction from position u to goal g is $g - u$.

The shortest-path direction map for the cage used in the previous chapter is shown in figure 18.1.

18.2 Evaluating error

Having defined the shortest-path direction map, we may express the error of another direction map by taking the element-wise angular difference between the directions to the goal for each position in the two maps.

Let d_u be the direction to the goal from position u in one of the direction maps we created in the previous section. Let d_u^* be the direction to the goal in the shortest-path direction map. Then the error (measured in degrees) between the two maps at position u is

$$\mathfrak{E}_u = \cos^{-1}(\frac{d_u^T \cdot d_u^*}{|d_u||d_u^*|})$$

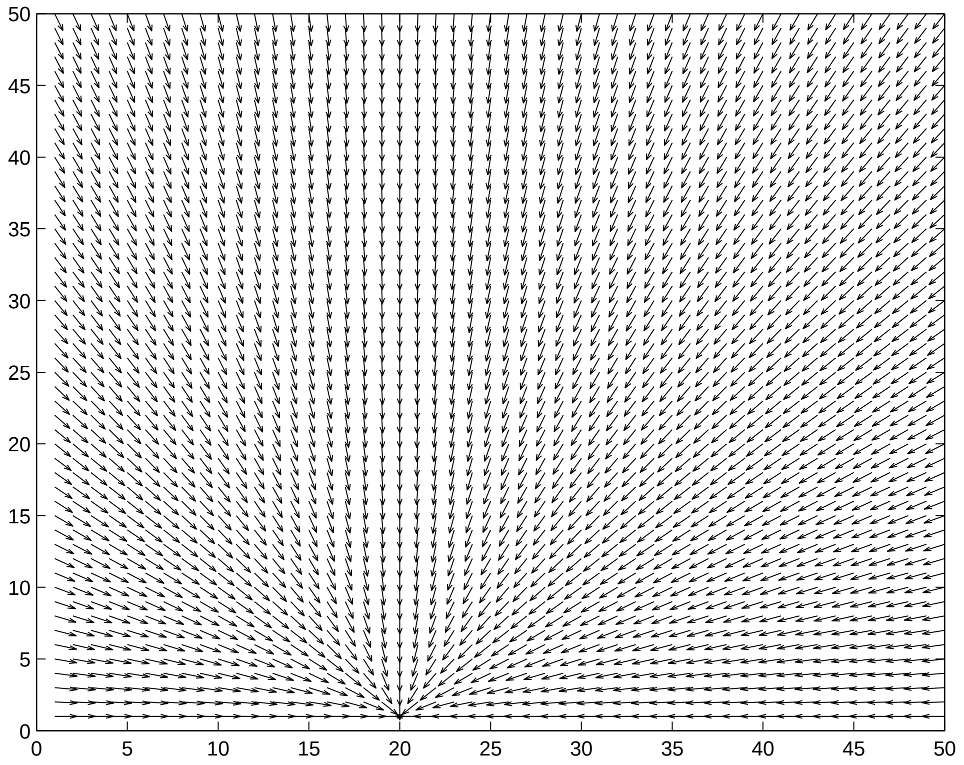

Figure 18.1: Shortest-path direction map for a cage without obstacles, with a goal in position (20,1).

Strategy K =	2.5°	5°	10°	15°	20°	25°
Original Trullier-Meyer	23.12%	46.32%	76.40%	92.32%	97.12%	99.52%
Adjusted Trullier-Meyer, $t_\alpha = 10^{-4}$	10.96%	20.32%	35.48%	47.56%	59.20%	69.72%
Adjusted Trullier-Meyer, $t_\alpha = 10^{-2}$	8.96%	18.44%	34.28%	46.40%	56.36%	63.00%

Figure 18.2: The number of positions u in the direction map where $|\mathfrak{E}_u| < K$, for various values of K. The adjusted Trullier-Meyer strategy is the Trullier-Meyer model using the PlaceCell positions taken from the lab experiments on the real rats.

$\mathfrak{E}_u$ will now be the angular difference between the direction to the goal suggested by the one map and the direction suggested by the shortest-path map.

Since we in our model only *approximate* the direction to the goal for each position, we should not expect $\mathfrak{E}_u$ to be 0 for all positions u. Instead we introduce a threshold value, K, and say that if the absolute value of the angular difference $\mathfrak{E}_u$ is less than K, then our direction map approximates the direction to the goal well enough.

We can now define the error map between two direction maps. We let element u of the error map be $\mathfrak{E}_u$ evaluated for each position u in the two maps.

18.3 Results

Based on the results presented in Chapters 12 and 17, we now show the error maps between the direction maps produced there and the shortest-path direction map as defined in the previous section.

Table 18.2 shows for how many position in the various direction maps that the absolute value of the angular difference is less than K, for different threshold values K.

We observe that the Trullier-Meyer strategy is able to produce the direction to the goal (within ±25 degrees) for 70% of the positions in the cage, using the PlaceCell representation from real rats. If we allow the Trullier-Meyer strategy to distribute the Placecells on its own, then it find the direction to the goal (within ±15) for 92% of the positions in the cage.

From the plots in figure 18.3, 18.4 and 18.5, we observe that the strategies are the most inaccurate around the goal at position (20,1). This is because around the goal the direction changes a lot with a small change in position. A move from position (19,2) to position (19,1) causes a change in the angle in the shortest-path map of 45°. As suggested earlier, we believe the problems with inaccuracies near the goal could be improved if we allowed the positions

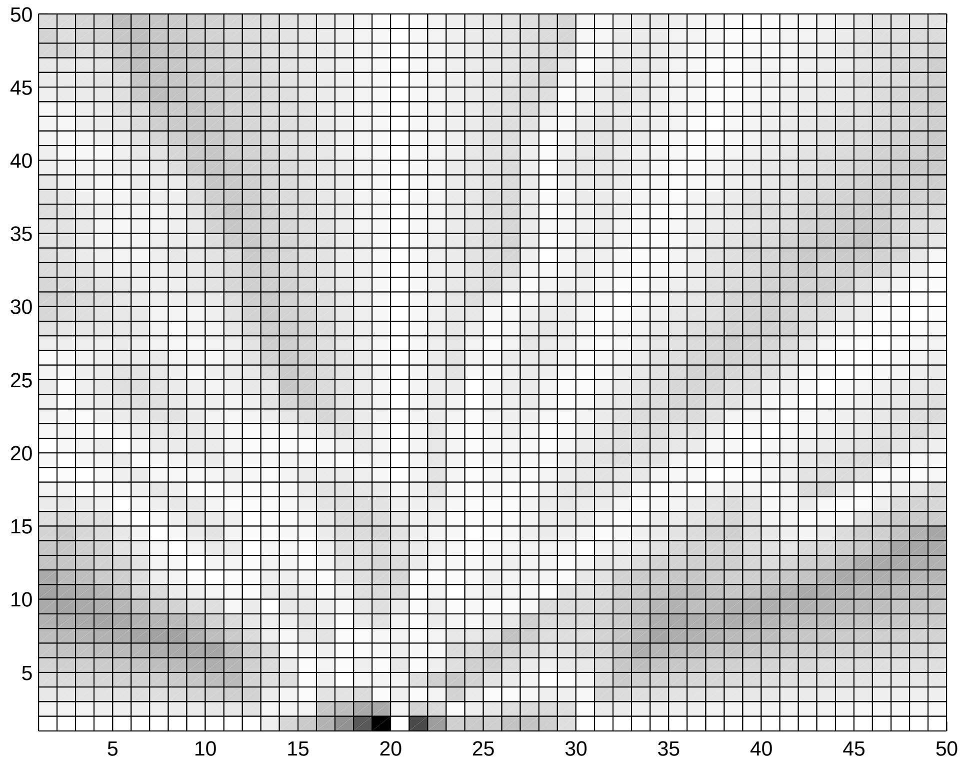

Figure 18.3: Plot of $\mathfrak{E}_u$ evaluated for all positions u in the cage, between the shortest-path direction map and the map from the Trullier-Meyer strategy presented in figure 12.12. A darker color indicates a large value for $\mathfrak{E}_u$ at position u, and a light color, a low value.

around the goal to be more heavily populated with PlaceCells.

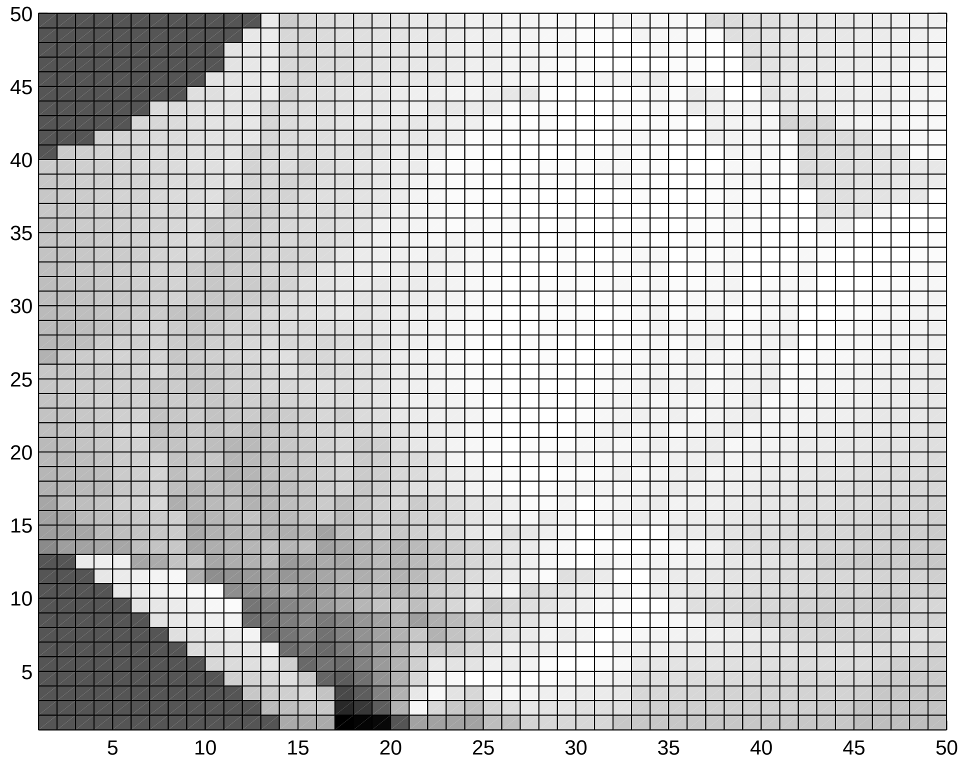

Figure 18.4: The difference map between the shortest-path direction map and the map from the adjusted Trullier-Meyer strategy with $t_\alpha = 0.00001$. Positions in which the direction map does not provide a direction has been assigned an arbitrary angle of 90°. This makes those positions medium gray, as seen in the upper and lower left-hand corner.

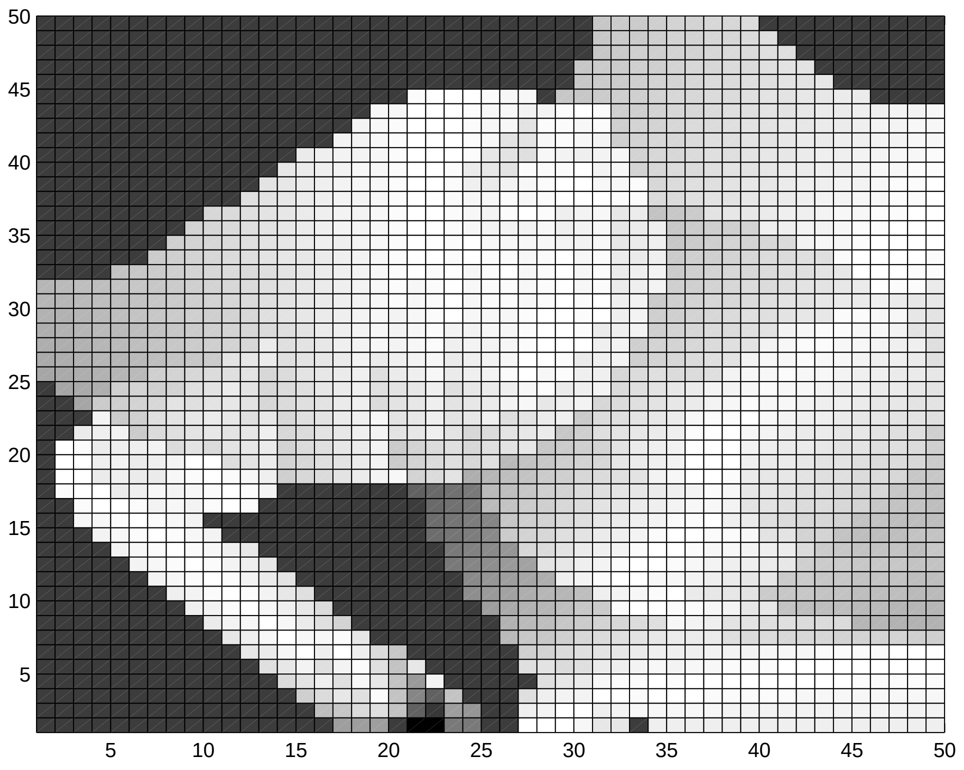

Figure 18.5: Plot of $\mathfrak{E}_u$ evaluated for all positions u in the cage, between the shortest-path direction map and the map from the adjusted Trullier-Meyer strategy with $t_\alpha = 0.001$.

Chapter 19

Verifying the results with simulation

In the previous chapters we have created several direction plots, which shows the direction the simrat believes the goal is in. In this chapter we will use theses direction plots to actual navigate the simrat from an arbitrary position in the cage to the goal.

Figure 19.1 shows how the rat moves in the cage, if following the direction suggested by the direction plot in figure 17.4. We have placed the simrat in a given position u, and let it for each position it visits, follow the direction given by the direction plot. We translate the angles suggested by the direction plots to movements in the following manner. Angles in the range $[-45, 45)$ means go East. Angles in the range $[45, 135)$ means go North, and so on.

Although these deterministic plots are interesting in themselves they are not really "realistic" in the sense that the movements seem to come from a real rat. However, we can remedy this situation by using some of the probabilistic strategies suggested in Chapter 8. Figure 19.2 shows how the simrat moves using a DeviationStrategy, as described in Chapter 8.

The DeviationStrategy uses the TMStrategy which produced the direction plot shown in figure 17.4 as its "Master strategy", $\hat{s}$. We define two other strategies, s^+ and s^-. Each of them will take the current direction specified by $\hat{s}$, alter it in some way, and return the resulting direction to the simrat, when queried. Specifically, s^+ will take the direction specified by $\hat{s}$ and add a random number[1] of degrees to the direction. Similarly, s^- subtracts a random number of the degrees from the direction specified by $\hat{s}$, when queried.

We assign $\hat{s}$ a probability of 10% of being chosen, and give s^+ and s^- a 45% probability each. Conversely this means that when the DeviationStrategy is queried for a direction, then there is a 45% chance that Deviation-

[1]We let the random number be positive and taken from a normal distribution with mean zero and variance one.

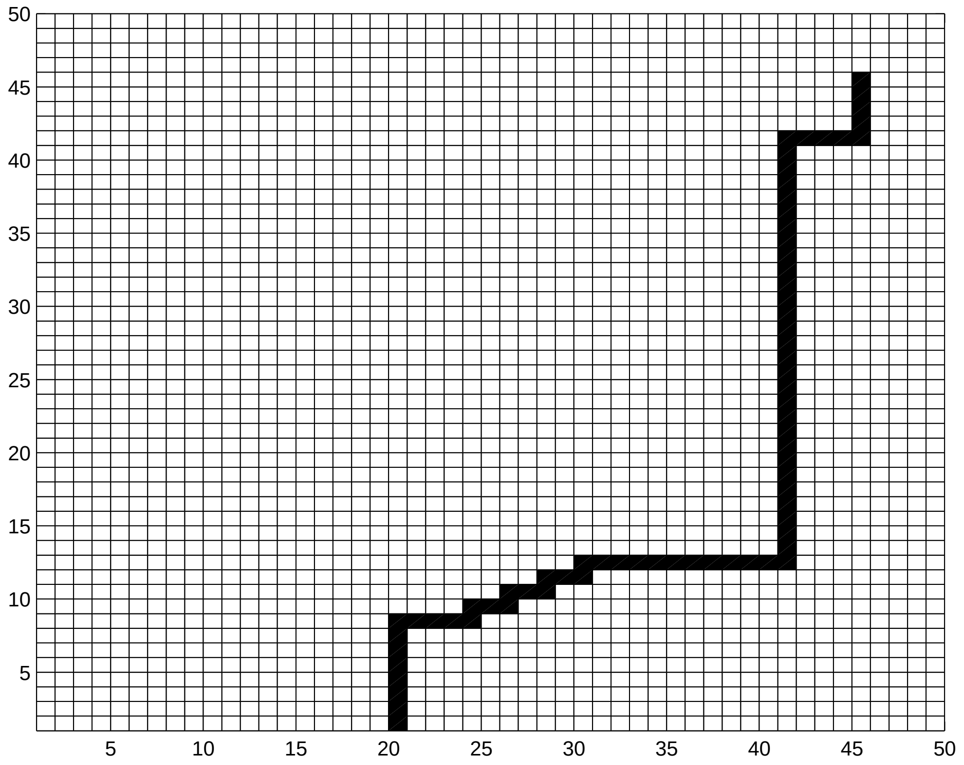

Figure 19.1: The deterministic path the simrat would follow from position $(45, 45)$ to $(20, 1)$, if using the direction map shown in figure 17.4.

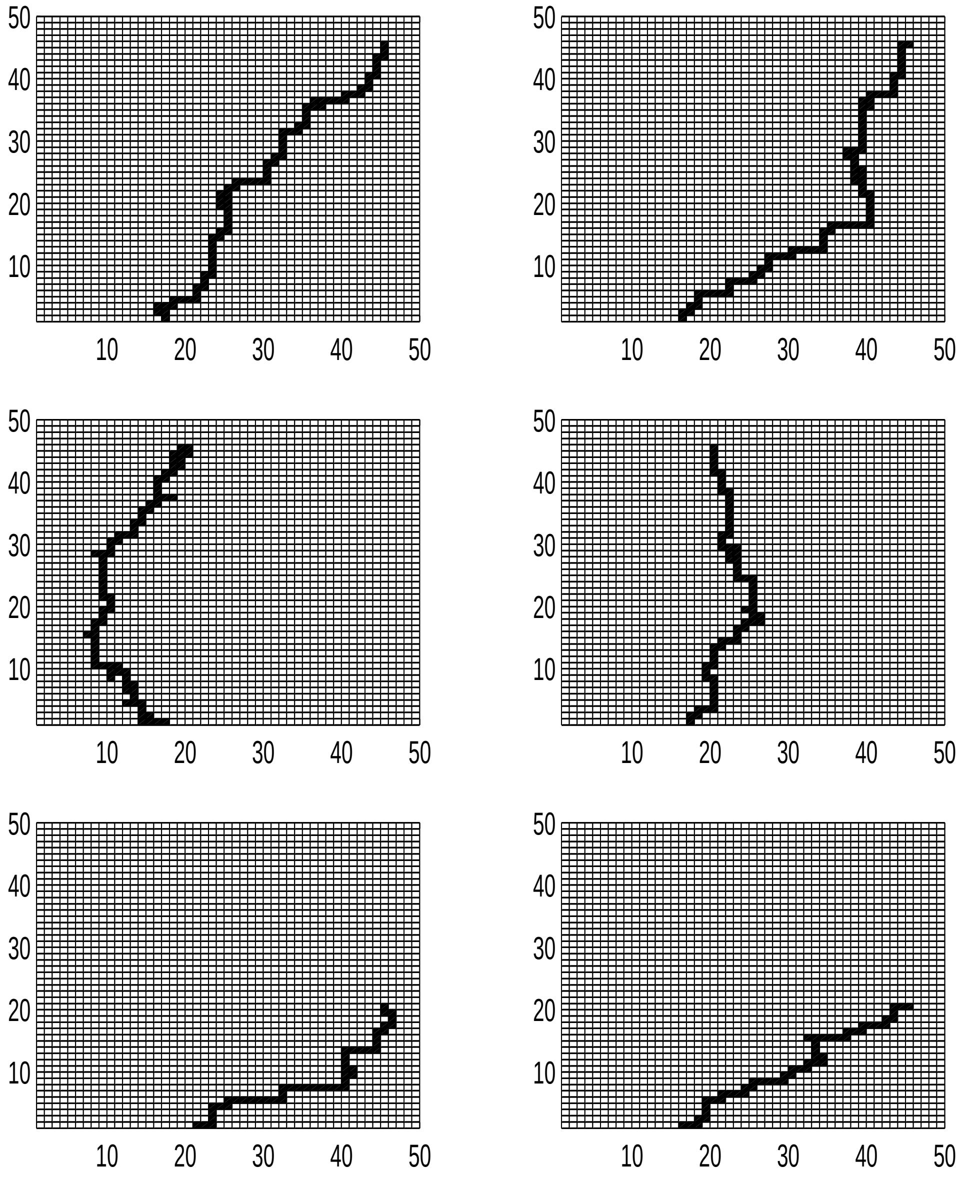

Figure 19.2: The sample paths taken by the simrat from various positions in the cage to the goal at position $(20, 1)$, using a probabilistic deviation strategy on the direction plot shown in figure 17.4.

Strategy will in turn query s^-, which then will return the direction in which to move.

From figure 19.2 we observe that the simrat does not always reach its intended goal at position $(20, 1)$ exactly. This is not surprising given the inaccuracies shown in figures 17.4 and 18.4. However, it usually produces reasonable results.

We stop the simulations when the simrat stays on the same position for 5 consecutive time units.

Chapter 20

Conclusion

In this thesis we have shown that discrete event simulation implemented in Java can be used successfully as a means for examining complex systems of biological phenomenons. We have created a general and open framework for simulation which has proved itself both scalable and easily extensible.

We started out with relatively simple examples showing basic functionality requirements, and gradually expanded our framework to be usable for simulation of advanced models and complex behavior in a simple and uniform manner. We believe that this is a good way of creating good software, first starting with the simple, then through a series of iterations gradually expand until the needed functionality is reached. This process involves first examining the goals of the project, and then consider each constituent part before implementing from the bottom up. Java supports such an approach, by allowing all parts of a program to be extended. Although we have implemented in the order of 6 000 lines of Java code in the course of this thesis, we believe that the Java language along with our architecture has made our frameworks simple and easy to use even for complex problems.

We have examined two models describing aspects of rat behavior, and produced results from our simulations comparable to those presented in the original papers by the creators of those models. In the case of the Trullier-Meyer (TM) strategy for navigation we have also clarified and formalized several aspects of the original model. This process has lead to both introducing new quantities and new calculations into the model. In addition we have introduced quantitative measures for evaluating how good a model for path integration actually is, relative to an "ideal map".

20.1 Results

We summarize the results we have obtained in this thesis in the following list.

- We have created a general and open framework for discrete event simulations in the Java programming language. This framework has been tested under several simulations, and has shown itself both scalable and easily extensible.

  - We have created an extension to the framework for discrete event simulation, which can be used for simulation of simrat navigation.
  - In the extended framework we have, under reasonable assumptions, been able to implement the models described in two published papers ([Trullier and Meyer, 2000] and [Tchernichovski and Benjamini, 1998]). We have produced results from these simulations which are comparable to those given in the original papers.

- Traditionally, cognitive functions and behavioural patterns in the rat has been simulated using neural networks. In this thesis we have demonstrated a version of the Trullier-Meyer (TM) strategy which can model these functions using an extensive algorithm.

  - We have formalized and clarified the TM strategy for navigation. For completeness, we have introduced new quantities and calculations, which are needed for implementation. We have given some conditions under which the model will fail.
  - We have shown that it is possible to use data from real rats as input to the TM model, and have a simrat successfully complete navigation in a simulated environment, using it. These results have not been shown before.

- We have introduced a quantitative measure of how good a simulation involving path-integration actually is, compare to an "ideal map".

20.2 Future work

The framework we have created for simulation is complete and ready for use. Although simple and powerful to use, it does not implement many of the more specialized features often found in discrete event simulation systems. A good continuation of the work presented in this thesis would be creating extensions for modelling communication systems, which is one of the top most important uses of discrete event simulation today. Among the things that should be introduced in such an extensions is support for threads, ports on each simulation entity and possibly a graphical user interface (GUI), which shows the state of the simulation.

In the process of developing the extended framework we made a few choices which may or may not impose restrictions on future use of the extension for rat simulation. We chose to represent the cage as a rectangular grid, and only allowed the rat to turn at $90°$ angles at a time. This was sufficient for the models we needed to implement in this thesis, but may hamper future models. One possible future improvement may include allowing the rat to represent both its position and direction as a decimal value. Nevertheless, our modular design will allow reuse of large portions of our current framework, and leave only relatively modest amounts of new code to be implemented.

For a while we considered implementing vision input as a 3D generated computer image, instead of our symbolic representation of vision, but concluded that that, along with the image recognition algorithms needed, would incur far to much work for implementation in this thesis. We leave such an implementation for the future.

In the Trullier-Meyer model for path integration, we used Trullier and Meyers assumption that the simrat was able to determine cardinal directions from its vision input. There has recently been discovered some evidence to suggest that there is a special type of cell, termed "head-direction cells", which are responsible for this task in the rat brain. The head-direction cells are active depending on the direction the rat is facing, and seems to be somewhat analogous to the place-cells which the Trullier-Meyer model describe. It would be interesting to examine the possibility of creating a unifying model which models both place-cells, goal-cells and head-direction cells in one complete model.

Appendix A

Mathematical details

In Chapter 12, we made a claim that a given set of distances $d_i^* = d(u_i)$ could uniquely define a position $u_i \in \Re^2$. In this chapter we will examine under which assumptions this holds true. But before we begin there is one result we will need to show first, which we will need later.

We start by considering two landmarks, x_1 and x_2 in the cage. Let u be the position of the simrat in the cage, and let $|ux_i|$ denote the Euclidean distance from position u to x_i.

Lemma 1
Let x_1 and x_2 be two distinct points $\in \Re^2$. Let $\mathcal{U}$ be the set of all points $u \in \Re^2$ satisfying $|ux_1| = |ux_2|$. Then all points $u \in \mathcal{U}$ will lie on a line ℓ.

Proof Let x_1 and x_2 be two points in $\Re^2$ and let $u \in \Re^2$ be a point that satisfies $|ux_1| = |ux_2|$. Now consider the triangle Δux_1x_2. Since $|ux_1| = |ux_2|$ then $\angle ux_1x_2 = \angle ux_2x_1$. Let $p \in \Re^2$ be the projection of u onto x_1x_2. Then $\angle upx_1 = \angle upx_2 = 90$.

Now, by trigonometry, we have the following relation

1. $|x_1u| \cos \angle ux_1p = |x_1p|$
2. $|x_2u| \cos \angle ux_2p = |x_2p|$

Since $|ux_1| = |ux_2|$ then $\angle ux_1p = \angle ux_2p = \alpha$ and we have that

$$\begin{aligned}
\frac{|x_1p|}{\cos\alpha} &= \frac{|x_2p|}{\cos\alpha} \\
|x_1p| &= |x_2p| \\
\sqrt{(p-x_1)^2} &= \sqrt{(x_2-p)^2} \\
2p &= x_2 - x_1 \\
p &= \frac{1}{2}(x_2 - x_1)
\end{aligned} \tag{A.1}$$

If $|ux_1| = |ux_2|$ we observe that

1. p lies on the midpoint between x_1 and x_2
2. up is perpendicular to x_1x_2

We now note that the second proposition holds true for all u satisfying $|ux_1| = |ux_2|$ and that therefore all up are perpendicular to x_1x_2; they are all *parallel.* From the first proposition it follows that all parallel lines up goes through point p. Therefore all the lines up are collinear; they lie in the same line ℓ.

■

Now we would like turn our attention towards the Trullier-Meyer (TM) model. TM assumes that a set of distances $d(u)$ to landmarks can *uniquely define* a position $u \in \Re^2$. We say that a position u is uniquely defined by $d(u)$ if there exists no other position $v \neq u$ so that $d(u) = d(v)$. If a position u is uniquely defined, then there is a one-to-one correspondence between u and $d(u)$.

We remember from chapter 12 that if there are n landmarks then $d(u) : \Re^2 \rightarrow \Re^n$, and the k'th element of $d(u)$, $d_k(u)$, is the Euclidean distance to landmark k. We say that landmark k is visible, if $d_k(u)$ is defined. Let m denote the number of visible landmarks from position u.

We would now like to answer whether a set of distances to landmarks $d(u)$ does uniquely define a position u, in the cage. We need to check whether there exist two positions u and v so that $d(u) = d(v)$. If so, then a set of distances does not uniquely define a position.

Theorem 1
If for every position u in the cage, there are at least 3 visible, non-collinear landmarks, then position u can be uniquely defined by the distance vector $d(u)$.

Proof Let a, b and c be three visible, unique and non-collinear landmarks in the cage. Let position $i \in \{a, b, c\}$ be the center in circle S_i, with radius r_i. Let u and v be the two intersection points between circles S_a and S_b. Position u and v are now the only positions not uniquely defined.

Assume that S_c also intersects S_a and S_b in u and v. If so then all positions u and v are still not uniquely defined.

But since then $|ua| = |va| = r_a$, $|ub| = |vb| = r_b$ and $|uc| = |vc| = r_c$ then, according to lemma 1, the three landmarks a, b and c must lie on the same line ℓ. But since they were defined to be non-collinear, our assumption must have been wrong. The circle S_c cannot exist so that it intersects both u and v, and have $|uc| = |vc|$. Therefore $d_c(u) \neq d_c(v)$, and $d(u) \neq d(v)$.

■

We observe that it would not be possible to represent each position in the cage uniquely if there are only two landmarks in the cage. Let a, b, S_a, S_b, u and v be defined as above. Then there are only two landmarks in the cage, and positions u and v are not uniquely defined by the distances to those landmarks. Therefore two landmarks are not always enough to represent all positions in a cage uniquely.

Trivially, just one landmark inside the cage can not represent the cage uniquely either, since any two positions lying on a circle with the landmark as center will have the same distance to that landmark.

Therefore we must have at least three landmarks in the cage for unique representation of all positions in the cage.

Appendix B

Image acknowledgements

I would like to thank the Hippocampus laboratory at PKI, notably Bolek Srebro and Eirik Thorsnes, for graciously allowing me to use images originating from their reasearch. Specifically, figure 13.1, 13.2, 14.1, 14.2, 14.3 and 14.4 were used, with permission from Eirik Thorsnes. For more information about these figures, see [Thorsnes, 2001].

Bibliography

[eb9, 1994] (1994). *Encyclopædia Britannica*, chapter "Navigation".

[Burgess et al., 1997] Burgess, N., Donnett, J. G., Jeffery, K. J., and O'Keefe, J. (1997). Robotic and neuronal simulation of the hippocampus and rat navigation. Technical report, Department of Computer Science, Manchester University, Oxford Road, Manchester, M13 9PL.

[Carson and Cobelli, 2001] Carson, E. and Cobelli, C., editors (2001). *Modelling methodology pf physiology and medicine.* Academic Press.

[Cormen et al., 1990] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms.* MIT Electrical Engineering and Computer Science. MIT Press.

[Dahl and Nygaard, 1966] Dahl, O. J. and Nygaard, K. (1966). SIMULA —an ALGO-based simulation language. *Communications of the ACM*, 9:671–678.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Booch, G. (1995). *Design Patterns —Elements of Reusable Object-Oriented Software.* Addison-Wesley, first edition.

[Goodrich and Tamassia, 1998] Goodrich, M. T. and Tamassia, R. (1998). *Data Structures and Algorithms in Java.* Worldwide Series in Computer Science. John Wiley & Sons.

[Helsgaun, 2000] Helsgaun, K. (2000). Discrete event simulation in java. Technical report, Department of Computer Science, Roskilde University, DK-4000 Roskilde, Denmark.

[Jacobson et al., 1999] Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process.* The Addison-Wesley Object Technology Series. Addison-Wesley.

[Knuth, 1968] Knuth, D. E. (1968). *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley.

[Knuth, 1969] Knuth, D. E. (1969). *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley.

[Lever et al., 2002] Lever, C., Wills, T., Cacucci, F., Burgess, N., and O'Keefe, J. (2002). Long-term plasticity in hippocampal place-cell representation of environmental geometry. *Nature*, 416:90–94.

[Lewicki, 1998] Lewicki, M. S. (1998). A review of methods for spike sorting: The detection and classification of neural action potentials. *Computational Neural Systems*, 9:R53–R78.

[Marr, 1982] Marr, D. (1982). *Vision: A computational investigation into the human representation and processing of visual information.* W.H. Freeman and Company.

[Moore, 2002] Moore, S. K. (2002). The brain as user interface. *IEEE Spectrum Online*, page http://www.spectrum.ieee.org/WEBONLY/resource/aug02/brainimplants.html.

[Muller et al., 1996] Muller, R. U., Stead, M., and Pach, J. (1996). The hippocampus as a cognitive graph. *Journal of General Physiology*, 107:663–694.

[Nocedal and Wright, 1999] Nocedal, J. and Wright, S. J. (1999). *Numerical Optimization.* Springer Series in Operations Research. Springer Verlag.

[O'Keefe, 1979] O'Keefe, J. (1979). A review of the hippocampal place cells. *Prog-Neurobiol*, 13(3):419-39.

[O'Keefe and Dostrovsky, 1971] O'Keefe, J. and Dostrovsky, J. (1971). The hippocampus as a spatial map. preliminary evidence from unit activity on the freely-moving rat. *Brain-Res*, 34(1):171-5.

[Redish, 1999] Redish, A. D. (1999). *Beyond the cognitive map: From place cells to episodoc memory.* The MIT Press.

[Scoville and Milner, 1957] Scoville, W. B. and Milner, B. (1957). Loss of resent memory after bilateral hippocampal lesions. *J-Neurol-Neurosurg-Psychiatry*, 20:11–21.

[Tchernichovski and Benjamini, 1998] Tchernichovski, O. and Benjamini, Y. (1998). The dynamics of long term exploration in the rat. *Biological Cybernetics*, 83:433–440. Part II. An analytical model of the minematic structure of rat exploratory behaviour.

[Tchernichovski et al., 1998] Tchernichovski, O., Benjamini, Y., and Golani, I. (1998). The dynamics of long term exploration in the rat. *Biological Cybernetics*, 78:423–432. Part I. A phase-plane analysis of the relationship between location and velocity.

[Thorsnes, 2001] Thorsnes, E. (2001). Emergence and stability of the hippocampal place cell activity on a new environment. –an experimental study on the rat.

[Trullier and Meyer, 2000] Trullier, O. and Meyer, J.-A. (2000). Animat navigation using a cognitive graph. *Biological Cybernetics*, 83:271–285.

[Walløe, 1968] Walløe, L. (1968). Transfer of signals through a second order sensory neuron. Technical report, Institute of Physiology, University of Oslo.

[Ziegler, 1991] Ziegler, B. P. (1991). Object-oriented modeling and discrete-event simulation. *Advances in Computers*, 33:67–114.