GASP: A Computing Framework For Markov Chain Monte Carlo Spatial Problems [1]

James Henstridge

October 2000

[1]Submitted in partial fulfilment of the requirements for the Bachelor of Science degree with Honours at the University of Western Australia

Abstract

In the area of Markov Chain Monte Carlo spatial problems, not many general purpose simulation systems exist. The GASP package is a framework for writing simulations. GASP stands for ***G****enerating* ***A****lgorithms for* ***S****patial* ***P****atterns*. It is designed to handle as many different problems as possible. A number of algorithms are distributed with the GASP framework, including a simulation of the Strauss Process using the Metropolis-Hastings Algorithm.

GASP includes a graphical user interface that aids visualisation of an algorithm, and sophisticated logging capabilities which allow for detailed post-simulation analysis. It is implemented as a share package for the computer algebra system GAP.

Acknowledgements

Thank you to my supervisors Adrian Baddeley and Alice Niemeyer.
Thank you to Alexander Hulpke for information about GAP.
Thank you to Ute Hahn for background information on some of the algorithms I considered.

Contents

Chapter 1

Introduction

1.1 Aims

In this project, I looked at the problem of implementing a framework for simulating processes related to spatial statistics using algorithms such as the Metropolis-Hastings algorithm.

The aim of the project was to produce a framework that could handle as many different algorithms in the class as possible, while remaining simple enough to be understood and easy to use. The result is the GASP package [6], which stands for ***G****enerating* ***A****lgorithms for* ***S****patial* ***P****atterns*.

1.2 The Choice of Language

Simulations of this type often require a lot of computation, so have traditionally been implemented in languages such as C or C++, due to the speed and flexibility they give. Using such a language has drawbacks:

- a program would be written specifically for a particular problem, so may require substantial modification to cover a different problem.
- knowledge of the C language, which includes many details, such as memory management, that are not directly related to the problem. This significantly raises the barrier for someone wanting to use such a program. Also, C is perceived as being difficult to learn, which reduces the value of a system which is aimed at mathematicians.
- as C is a compiled language, it takes more effort to make a change to the algorithm, which can slow down development.

Rather than having many programs, each implementing their own algorithm, it would be better to have a framework from which many different algorithms could be explored with. That was the aim of my project.

An interpreted language was chosen because of reduced development overhead, allowing more rapid experimentation and implementation of new algorithms.

1.3 GAP

The language chosen for the implementation was GAP [2]. While GAP was originally designed for work with group theory, a number of its features made it an interesting choice for this class of problem.

While S-Plus [10] may be considered the obvious choice of language, it suffers from some problems when used to run simulations that take a long time. Problems related to instability and memory leaks are magnified during long runs, which may lead to loss of data if the interpreter crashes. Also, to someone who does not already know S-Plus, the syntax can be difficult to learn.

Like S-Plus, GAP is an interpreted language. Its syntax was designed with mathematicians in mind, so is quite easy for a new user to pick up. GAP has some object oriented features, which were used when implementing the GASP framework.

As well as having the rapid development advantages of interpreted languages, GAP also has interactive debugging features, and support for designing graphical user interfaces that consist of menus and geometric shapes drawn on a canvas.

Using GAP's ability to pass functions as arguments to other functions, GASP allows the user to hook into many different parts of the simulation framework.

1.4 The Change Log

All the algorithms that I considered involved transitions between different states of a configuration of some type of object such as points in the plane. Quite often, it is these transitions, or changes in the configuration, that are interesting.

In the GASP framework, these changes form an important part of the simulation. Borrowing some ideas from the undo capabilities found in some

software, all changes made to the configuration are encapsulated inside objects that hold all information to apply and revert the change in state.

As the amount of memory required to encode these these changes is generally smaller than the amount required for the configuration itself, it was feasible to log a lot more information about what happened during the simulation than with systems where the complete state of the configuration was logged. This unique feature of GASP gives users the ability to analyse a simulation much more effectively, even after the simulation has completed.

1.5 Chapter Guide

The second chapter gives a brief overview of the mathematics behind some of the problems that the GASP framework covers.

The third chapter covers the features of GAP that were useful when designing GASP, and more detailed reasons why it was chosen as the language to implement GASP in.

The fourth chapter is an introduction to the functionality provided by the GASP framework. This chapter is aimed at users who are interested in using the existing functionality in GASP, rather than extending it to handle new algorithms.

The fifth chapter looks at the implementation of the framework, and how to extend its functionality – both the algorithms that are supported and the types of configurations that can be used in simulations.

Chapter 2

Background of Class of Problems

Some knowledge of the underlying probability theory is required to fully understand the purpose and functionality of the software we developed. So we will first look at some of the basic concepts and a few problems that relate to the program. Some of the background material in this chapter can be found in *Spatial Statistics* [7].

2.1 Markov Chains

2.1.1 Basics

A Markov Chain is a system with a number of possible states, and probabilistic rules for moving between those states. As we move through time, we make transitions between states. There are rules that govern to which state we can move from the current one. The restriction on the rules is that given the entire past series of states $X_1, X_2, \ldots, X_t$, the choice of next state X_{t+1} only depends on the current state X_t.

We can represent the probabilities for moving from a given state X_t to each of the other states in the system by the vector $P_t = (p_{t1}, p_{t2}, \ldots, p_{tn})$, where p_{ti} is the probability of moving from state X_t to state X_i. We can collect the probability vectors for transitions from each state into a matrix:

$$\mathbf{P} = \begin{pmatrix} P_1 \\ P_2 \\ \vdots \\ P_n \end{pmatrix} \tag{2.1}$$

This matrix, known as the *transition probability matrix*, is *stochastic*. That is the sum of the entries in each row is one (as the rows represent the complete set of probabilities for moving to any state in the state space).

If the current state is randomly distributed in the state space according to the probability row vector π_t, we can get the distribution for the next state by right multiplication by $\mathbf{P}$:

$$\pi_{t+1} = \pi_t \mathbf{P} \tag{2.2}$$

Similarly, we can get the distribution for the state after that by multiplying the probability distribution vector by $\mathbf{P}$ again.

Some systems will converge towards an *limiting* distribution as they progress in time. This distribution is often denoted π. The limiting distribution's definition is as expected:

$$\pi = \lim_{i \to \infty} \pi_0 \mathbf{P}^i \tag{2.3}$$

If such a limit exists, then under certain circumstances π is an equilibrium distribution, in which case it satisfies

$$\pi = \pi \mathbf{P} \tag{2.4}$$

This corresponds to the simple linear system $\pi(\mathbf{P} - \mathbf{I}) = 0$, which can be solved to find the possible equilibrium distributions for the system.

2.1.2 The Monte Carlo Method

The *Monte Carlo* method is a method of solving problems using random numbers. We can apply these ideas when simulating a Markov Chain system [11].

If we know the state transition probability matrix $\mathbf{P}$ for a system, it is possible to simulate a Markov Chain. The algorithm proceeds as follows:

Algorithm 1 (Monte Carlo)

1. *Pick an initial state X_0.*
2. *Given that the current state $X_t = x$, we generate the next state y from the random probability distribution $P(X_{t+1} = y) = p_{xy}$, or the yth column of the xth row of the transition probability matrix $\mathbf{P}$.*
3. *Having picked the new state X_{t+1}, we go to step 2 and repeat to generate the next state.*

2.1.3 Problems

Of course, Algorithm 1 is not practicable in all situations. Some possible problems are:

- the state transition matrix $\mathbf{P}$ is not known, or would be too expensive to calculate. In this case, we would still need to know the rule for calculating the next state.
- the number of possible states in the system is infinite or very large (for example, the states may represent the positioning of some points on the plane).

In these cases, we might not be able to find the equilibrium distribution for the system just by solving a system of linear equations, or may not get a useful result. However, we may be able to simulate the system, which is another way to get the desired information.

2.2 Using Simulation to Generate Distributions

For some probability distributions, it is not clear how to generate instances of the distribution directly. Instead, we sometimes use an iterative Markov Chain based algorithm that will converge to the desired distribution as equilibrium.

This is similar to shuffling cards in order to randomise their order. We know the distribution that we want – we just need to find a 'shuffling' algorithm that will get us to that distribution. One such algorithm is the Metropolis-Hastings Algorithm.

2.2.1 Metropolis-Hastings Algorithm

The *Metropolis-Hastings* algorithm is a Markov Chain system that can be used to generate a realisation of a given equilibrium distribution.

It uses the idea that if the probability of the transition from state x to state y, is the same as for the transition $y \to x$, then the equilibrium distribution will be maintained. This is known as *detailed balance*, as expressed in Equation 2.5.

$$\mathbf{P}(X_t = x \textbf{ and } X_{t+1} = y) = \mathbf{P}(X_t = y \textbf{ and } X_{t+1} = x) \tag{2.5}$$

It is achieved using the algorithm described below. It consists of a proposal step and an acceptance step.

Algorithm 2 (Metropolis-Hastings)

1. *An initial state x is chosen.*
2. *proposal: a transition $x \to y$ is proposed. The transition is chosen randomly according to some predefined random method with probability $p_{prop}(x \to y)$.*
3. *acceptance check: the probability of accepting this transition $p_{acc}(x \to y)$ is calculated. With probability $p_{acc}(x \to y)$, we move to state y. Otherwise we stay at x.*
4. *go to step 2 and repeat.*

According to these rules, the probability that the next state (x_{i+1}) will be y, given the current state (x_i) is x, is:

$$p_{trans}(x \to y) = p_{prop}(x \to y)p_{acc}(x \to y) \tag{2.6}$$

Applying the fact that $\mathbf{P}(A \textbf{ and } B) = \mathbf{P}(A \mid B)\mathbf{P}(B)$ to both sides of Equation 2.5 and substituting $p_{trans}(x \to y)$ we get:

$$p_{trans}(x \to y)\mathbf{P}(X_t = x) = p_{trans}(y \to x)\mathbf{P}(X_t = y) \tag{2.7}$$

Substituting from Equation 2.6:

$$p_{acc}(x \to y)p_{prop}(x \to y)\mathbf{P}(X_t = x) = p_{acc}(y \to x)p_{prop}(y \to x)\mathbf{P}(X_t = y) \tag{2.8}$$

And rearranging, we get:

$$\frac{p_{acc}(x \to y)}{p_{acc}(y \to x)} = \frac{p_{prop}(y \to x)\mathbf{P}(X_t = y)}{p_{prop}(x \to y)\mathbf{P}(X_t = x)} \tag{2.9}$$

We need to choose values $p_{acc}(x \to y)$ and $p_{acc}(y \to x)$ that will satisfy equation. The ratio $\frac{p_{acc}(x \to y)}{p_{acc}(y \to x)}$ is known as *Green's Ratio*, or $R_{x \to y}$. The probabilities $\mathbf{P}(X_i = y)$ and $\mathbf{P}(X_i = x)$ are given in the desired equilibrium distribution. The proposal probabilities $p_{prop}(x \to y)$ and $p_{prop}(y \to x)$ were chosen as part of the algorithm. Hence, we can calculate $R_{x \to y}$.

For the Metropolis-Hastings algorithm, we set $p_{acc}(x \to y)$ as:

$$p_{acc}(x \to y) = \min(\{R_{x \to y}, 1\}) \tag{2.10}$$

In the case that $R_{x \to y} > 1$, $R_{y \to x} = 1/R_{x \to y} < 1$, so $\frac{p_{acc}(x \to y)}{p_{acc}(y \to x)} = \frac{1}{1/R_{x \to y}} = R_{x \to y}$. When $R_{x \to y} < 1$, we get $\frac{p_{acc}(x \to y)}{p_{acc}(y \to x)} = \frac{R_{x \to y}}{1} = R_{x \to y}$. So this definition

of $p_{acc}(x \to y)$ is consistent. For a system with an infinite number of states, we can use the corresponding probability densities in place of the discrete measures, and the same argument applies.

Of course, the Metropolis-Hastings Algorithm does not guarantee that we will converge on the desired equilibrium distribution. Even in the case where it does converge, it does not give any indication when we have converged to the desired equilibrium distribution.

2.2.2 The Metropolis-Hastings Algorithm and Point Processes

Point processes are spatial models of a collection of points governed by some probability density [1]. Each state of the system is a configuration of points within some region (or window) of the plane. It can be very difficult to realise the process directly, so we use an algorithm that will converge to the desired distribution, such as the Metropolis-Hastings algorithm.

For the proposal part of the algorithm, we will limit the possible state transitions allowed to (a) adding a new point in a random location within the window W, and (b) removing an existing point. All other transitions are assigned a probability of zero. We will assign equal probability to adding and removing a point (unless the configuration is empty, in which case only an addition is possible).

We represent the state by the set of points on the plane $\mathbf{x} = \{x_1, \ldots, x_n\}$. The probability density of the proposal to add a particular point y to the configuration somewhere in the window of area $|W|$ is $p_{prop}(\mathbf{x} \to \mathbf{x} \cup \{y\}) = \frac{1}{2}\frac{1}{|W|}$. Similarly, the probability of proposing to remove a particular point y is $p_{prop}(\mathbf{x} \to \mathbf{x}\backslash\{y\}) = \frac{1}{2}\frac{1}{n(\mathbf{x})}$ where $n(\mathbf{x})$ is the number of points in the set $\mathbf{x}$.

Hence, for adding a point, Green's Ratio becomes:

$$R_{\mathbf{x}\to\mathbf{x}\cup\{y\}} = \frac{|W|}{n(\mathbf{x})+1}\frac{f(\mathbf{x}\cup\{y\})}{f(\mathbf{x})} \tag{2.11}$$

and for removing a point, we have:

$$R_{\mathbf{x}\to\mathbf{x}\backslash\{y\}} = \frac{n(\mathbf{x})}{|W|}\frac{f(\mathbf{x}\backslash\{y\})}{f(\mathbf{x})} \tag{2.12}$$

where $f(\mathbf{x})$ is the density function for the equilibrium distribution.

Hence, the acceptance probability p_{acc} used will be $\min(R_{\mathbf{x}\to\mathbf{x}\cup\{y\}}, 1)$ or $\min(R_{\mathbf{x}\to\mathbf{x}\backslash\{y\}}, 1)$ respectively.

2.3 The Strauss Process

The *Strauss Process* [8] is an example of a point process that can be simulated using the Metropolis-Hastings algorithm. The density function for the Strauss Process is:

$$f(\mathbf{x}) = \alpha\beta^{n(\mathbf{x})}\gamma^{s(\mathbf{x})} \tag{2.13}$$

where $\mathbf{x}$ and $n(\mathbf{x})$ are defined as in Section 2.2.2, and $s(\mathbf{x})$ is defined as the number of 'close' pairs of points separated by no more than a given distance r.

The symbols in the density function of Equation 2.13 are:

α is a constant to normalise the function to ensure that it is a density function.

β is a parameter that measures the 'intensity' . It affects the number of points we would expect to find in the configuration.

γ is the *interaction* parameter:

- If $\gamma = 1$, then the density function does not depend on neighbour relationships.
- If $0 < \gamma < 1$, then larger spacing of the points is favoured, as large $s(\mathbf{x})$ gives small $\gamma^{s(\mathbf{x})}$.
- If $\gamma > 1$, close clustering of a large number of points leads to a very large $\gamma^{s(\mathbf{x})}$. In fact, with $\gamma > 1$, we cannot find an α to normalise the density, so a Strauss Process does not exist for this case.

In order to simulate the Strauss Process using the Metropolis-Hastings algorithm, we must calculate the acceptance probabilities for proposed states. For this, we need to know Green's Ratio $R_{\mathbf{x_1}\to\mathbf{x_2}}$, which required the calculation of the ratio $f(\mathbf{x_2})/f(\mathbf{x_1})$. This ratio is:

$$\frac{f(\mathbf{x_2})}{f(\mathbf{x_1})} = \frac{\alpha\beta^{n(\mathbf{x_2})}\gamma^{s(\mathbf{x_2})}}{\alpha\beta^{n(\mathbf{x_1})}\gamma^{s(\mathbf{x_1})}} = \beta^{n(\mathbf{x_2})-n(\mathbf{x_1})}\gamma^{s(\mathbf{x_2})-s(\mathbf{x_1})} \tag{2.14}$$

In the case of adding a point, $n(\mathbf{x}\cup\{y\}) - n(\mathbf{x}) = 1$, and for removing a point, $n(\mathbf{x}\backslash\{y\}) - n(\mathbf{x}) = -1$. It is also fairly obvious that $s(\mathbf{x}\cup\{y\}) - s(\mathbf{x})$ is the number of points in $\mathbf{x}$ that are at most a distance of r from the new point y. We will denote this number $s(\mathbf{x}, y)$. We can see that $s(\mathbf{x}\backslash\{y\}) - s(\mathbf{x}) = -s(\mathbf{x}, y)$ if we do not count y when calculating $s(\mathbf{x}, y)$.

This is enough information to calculate the Green's Ratio for both adding and removing points, and hence the acceptance probabilities for the proposals:

- add point:

$$R_{\mathbf{x}\to\mathbf{x}\cup\{y\}} = \frac{|W|\beta\gamma^{s(\mathbf{x},y)}}{n(\mathbf{x})+1} \tag{2.15}$$

- remove point:

$$R_{\mathbf{x}\to\mathbf{x}\setminus\{y\}} = \frac{n(\mathbf{x})}{|W|\beta\gamma^{s(\mathbf{x},y)}} \tag{2.16}$$

2.4 The Widom-Rowlinson Bivariate Process

The *Widom-Rowlinson Bivariate Process* [12] is another process based on configurations of points that we consider. It can be thought of as a simulation of two different types of particles ('red' and 'black') in some region of the plane, and particles of opposite type cannot get closer than some predefined distance r. I refer to this process as the Red-Black process in later chapters.

In this system, we introduce the concept of *marking* the points in the configuration. A mark is an attribute of the point that is associated with a point. In many problems, we may think of a mark as a colour for an object. Of course, for some systems, we may have more than one non mutually exclusive mark for a point.

The configuration in this process is made up of two collections of points inside the simulation window. One collection is all the *red* points and one is the *black* points. The algorithm uses parameters μ and ν as means per unit area of Poisson distributions for the two collections of points.

The algorithm for this process can be represented as a two phase process:

Algorithm 3 (Widom-Rowlinson)

1. *remove all the black points from the configuration.*

2. *we pick a random number using a Poisson distribution of mean ν times the area of the window. This is the number of new black points we propose to add to the system.*

3. *for each of the points we propose to add, we choose a random location for the point within the simulation region. If the point is no closer than r to any of the red points, then we add it to the configuration. Otherwise, the point is discarded.*

4. *now repeat for the red points, using μ as the Poisson mean.*

This process is then repeated. As the reader can see, we end up with two collections of points separated by the distance r. The distribution of black points in all parts of the simulation window with no red point closer than r is Poisson distributed with a mean ν times the area of this region. Similarly, the red points are Poisson distributed in the region defined by the black points.

Once the simulation has completed, we may only be interested in one of the collections of points.

Once again, we are left with the question of when the simulation is 'finished'.

2.5 Glossary

Some of the terms introduced in this chapter are used later on when describing the software and the simulation algorithms we implemented. The important ones are listed below.

limiting distribution is the probability distribution that the system moves toward as the simulation continues. Such a distribution may not exist for a particular Markov Chain.

Monte Carlo method is a method of solving deterministic problems using random numbers.

Metropolis-Hastings algorithm is an algorithm used to implement a Markov Chain system that will converge on a desired probability distribution as the equilibrium distribution.

configuration refers to the arrangement of objects, along with any other state information that represents the state in the algorithm.

configuration objects (or just objects) are the objects that make up the configuration. In many of the problems, these are just points, but in others they may be lines or objects of some other shape.

marks are some attributes assigned to an object. In some problems, we may want to group the objects in the configuration. We do this by adding marks to the objects. These may be multi valued (such as the colour of the object), or boolean valued. An object may carry more than one mark.

proposals refer to changes proposed to be made to a configuration. This is the first part of the Metropolis-Hastings algorithm.

acceptance is the act of accepting a proposal, and modifying the state of the configuration accordingly.

rejection is the act of discarding a proposal and leaving the configuration in its existing state.

Chapter 3

Features of GAP

3.1 Requirements

When deciding on what language to use for the simulation system, there were a number of features that we were looking for.

3.1.1 Features

When choosing the language to implement GASP in, there were a number of required and desired features we looked for:

- **The language must allow rapid development**. It is important that the language be quick to use, so that end users of GASP can easily experiment with the algorithms they are working on. This requirement is most easily met by using an interpreted language.

- **must be easy to learn and have a clear syntax**. Ideally, it should be very easy for a new user to start extending GASP to handle new algorithms. Also, programs written in the language should be easy for others to be able to read without a full knowledge of the language.

- **must be fast and stable**. Programs written in the language must run fast. Stable means that programs written in the language should not crash half way through a simulation, causing all information up to that point to be lost.

- **should have some graphics capabilities**. It would be nice to be able to design a graphical user interface to run a simulation in.

- **should make it easy to debug programs**. The language should make it easy to debug errors. That is, when an error occurs, rather than just exiting, it would be nice if it were possible to find out what went wrong. This feature is more important to people trying to extend the GASP package, rather than those using existing algorithms.

3.1.2 The Choice of Language

The GAP language was chosen because it met these criteria. S-Plus was not chosen because it has had problems with long running programs in the past, where its memory consumption would increase with time, and S-Plus occasionally crashes under some conditions.

Another language that was considered was Python [9]. Python shares a number of these desirable features with GAP, but during tests it was found to be slower than GAP. This is most likely caused by the different types of memory management performed by the two languages. Also, the graphical capabilities of GAP are more convenient for use in GASP than those available for Python.

A more detailed summary of the relevant features of GAP follows in the next section.

3.2 Relevant Features of GAP

The aim of this section is to highlight some of the features of the system GAP, which we used to develop GASP. This section is not intended to be a substitute for the GAP documentation or as a tutorial for the language. Rather, it should familiarise the reader with some of the basic concepts that we use in the remaining chapters.

GAP stands for *Groups*, *Algorithms* and *Programming*. Development started in 1985 at Lehrstuhl D für Mathematik, RWTH-Aachen by a group led by Professor Joachim Neubüser. Since then it has gone through a number of rewrites, and is now maintained by an international group of developers coordinated from Saint Andrews. Although it was not specifically designed for the types of statistical problems we considered, a number of its features made it a very useful tool for our project. GAP is freely available from the program's web site [2].

Once installed, GAP can be started from the shell prompt with the following command:

```
$ gap
```

This will start up GAP, and print a gap> prompt where you can enter GAP commands to execute.

3.2.1 General Description

GAP was designed as a language for use by mathematicians, so has a syntax that corresponds fairly simply to written mathematics. This reduces the perceived learning curve required to use the system, which could be a deciding factor for a new user when considering whether or not to use GASP.

GAP consists of a number of components centered around a small kernel which is written in C. The kernel is implemented in the main GAP executable, and is comprises:

Language - the parser for the GAP language combined with the interpreter which is responsible for executing GAP programs. This also includes the base of the GAP object model, as described in Section 3.2.4.

Memory management - the code that handles all memory allocation and deallocation in GAP. GAP uses a form of garbage collection, so users do not have to concern themselves with explicitely allocating or freeing memory in their program.

Selected objects and operations - a number of functions and data types which are impossible or very inefficient to implement in the GAP language. These include implementations of the base data types such as integers, lists and some other data types, along with some of the basic operations for those types.

On top of this kernel are the standard library, some data sets and the documentation. The standard library takes the primitives defined inside the kernel and builds them up to create data types that can be used to work on various mathematical problems.

There are a number of data sets (the small groups library, etc) distributed with GAP, which can be accessed from the language. These are not relevant to the problems we are interested in here.

GAP contains a builtin documentation system, so the interpreter can extract a section from a specially formated TEX documentation file for a particular function. This is then displayed to the user, the format depending on the way GAP was invoked. This yields a very simple way to provide both online help for the functions in GAP, and produce a high quality typeset version of the library reference.

In addition to the standard library, GAP provides a way for any user to contribute packages known as *share packages* to be used. The use of share

packages makes it very easy to package together a number of functions and data structures together for distribution. Official share packages are refereed.

For GASP, most of the standard library and all the data sets were not used. However, a lot of the infrastructure these components were built on was used.

3.2.2 Interpreted Nature

GAP is an interpreted language. If a language is not interpreted, a user writes a program, then compiles the program to machine code, executes the program to test it, and debugs it if any errors occur. If errors are found, the user must recompile and test the application again. With an interpreted language however, the compile stage of the normal `write` → `compile` → `test` → `debug` cycle gets removed, which speeds up development.

GAP is also *interactive*. This means that a user can enter a command at the gap> prompt, GAP evaluates the command and prints the result. This is the usual `read` → `eval` → `print` loop found in many interpreted languages such as python or the UNIX shell.

Often, interactive languages reduce the experimentation part of the development process down to just `test` → `debug`. The above simplification of the development cycle leads to fast experimentation and fast development when combined with the fact that memory management can be ignored.

Of course, the interpreted nature of GAP can lead to less than optimal speed. In most cases, the tradeoff against shorter development time is sensible. In the cases it is not sensible, a GAP to C translator is available that allows compilation of GAP code for extra speed.

3.2.3 GAP Features

As well as the many features related to group theory, GAP contains many features found in most general purpose languages. Many of these were necessary to build up the mathematical features of the language. These include a number of data types including some aggregate data types, and many of the normal control structures found in most programming languages.

Basic Data types

As well as the more specialised group theory types found in the standard library, there are a number of general purpose types. These include integer, rational and floating point numbers. Integers and rational numbers convert to an arbitrary precision representation if they grow too large.

At the time of writing, the floating point support is not well integrated into the GAP system. While it is possible to perform binary operations on two floats, sometimes an error occurs when the same operation is applied to a float and a non-float. Also, not all functions accept a float where some other type of number is expected due to these problems. It is expected that these problems will be resolved in the future.

In addition to these basic types, GAP provides a number of aggregate data types. The main ones are lists and records.

Lists in GAP can be *heterogeneous*. That is, a list can contain elements of more than one type. Lists can be expanded and contracted as needed at runtime using the `Append` and `RemoveSet` functions. You can perform complex indexing operations on a list. For example, if we have a list initialised as `list := [7,8,9,10,42,11]`, then the expression `list{[1,3,5]}` returns a list containing the first, third and fifth elements of the initial list, which is `[7,9,42]`.

Records in GAP are similar to `struct`s in C or `record`s in Pascal. They provide a way to group a number of named variables together.

These aggregate data types can be nested to produce more complex types as required. GAP also provides a way to *objectify* lists and records, which gives you even more control over how they are handled. This is covered in the next section.

Control Structures

GAP provides many of the standard control structures found in most languages. These include the standard `if .. then .. else .. end if` conditional statements, as well as statements to iterate over the members of a list. There are also the more general `while` and `repeat .. until` loops.

For more information, see the GAP reference manual [3].

3.2.4 The GAP Object Model

GAP provides the ability to create hierarchies of types similar to what is found in many object oriented languages. We can create instances of these types by *objectifying* records or lists.

The main reason for creating these objectified types is GAP's operation system. Operations are analogous to methods in most object oriented languages and provide a way of creating *polymorphic* functions. That is, functions that perform different actions depending on the types of the arguments.

GAP also provides the ability to overload the standard operators. This means that you can implement the standard operators such as $+$, $-$, $*$, $=$

and $\leq$ for any new object types created in GAP. Operator overloading in GAP uses the operations system described above. This means that we can provide different implementations of the operations depending on the types of the operands.

Although GAP appears to be purely procedural, the combination of objectified types and operations gives as much power as found in more traditional object oriented languages. The difference in syntax is not expected to be a problem, as most users of GASP will probably not have much experience in object oriented languages to start with.

3.2.5 Functions

GAP treats functions like any other variables. They can be passed as an argument to another function, or returned from a function. In fact, the usual way of defining a new function is through an assignment to a variable.

Similar to functions in languages like Pascal, a GAP function has access to the variables in the environment where it was defined. If we define one function within the body of another function, it will have access to the local variables of the enclosing function as well as those in the *global name space*. A *name space* refers to a mapping between variable names and their corresponding values. Usually a particular name space is only used in certain contexts, but variables in the global name space are available from all parts of the program.

When combining access to the defining environment of a function with passing function variables as arguments to other functions, it is not quite clear what a language should do. Consider the following fragment of GAP code:

```
g := function()
    local i, h;
    i := 0;
    h := function();
        i := i + 1;
        return i;
    end;
    return h;
end;
f := g();
```

This creates the function `f`. When we exit from the call of `g`, the local variables that make up the environment in which the function `h` was defined are preserved. The other common behaviour in this situation is to not give the function `f` access to `g`'s local variables, or not preserve their values across

calls to `g`. GAP's behaviour was very convenient when writing parts of GASP.

Successive calls to `f` will result in a series of positive integers being returned. If we create another copy of `f` by calling `g` again, the series returned by the copy will be independent of the first `f`. In effect, each `f` has a copy of the variable `i`.

Such an `f` is known as a *closure*. That is, it is a combination of the function and the environment it was defined in.

We used this functionality fairly extensively inside GASP.

3.2.6 XGAP

XGAP is a combination of a share package for GAP along with a wrapper front end that provides graphical capabilities using the X Window System. XGAP can be started by running the `xgap` command. This will open an X window for the console and load up the share package part of XGAP.

The `xgap` share package provides the ability to create *graphic sheets* which act like a canvas on which various geometric figures such as rectangles, lines and circles can be drawn. These graphic sheets handle all redraws for the canvas, so a program does not need to worry about redrawing the shapes after the canvas gets obscured. XGAP also provides the capability to save the contents of a graphic sheet to an encapsulated PostScript (EPS) file which can then be printed or included in a document, as I have done so in Figure 5.1.

Graphic sheets can also take mouse input from the user, which is passed back to the program, and add extra menu items to the menu bar at the top of the graphic sheet.

XGAP opens a new window when you ask for help on a particular topic, and provides some cross referencing support so you can click on other topic names in the help window.

We use XGAP's graphics abilities for visualisation in GASP.

3.2.7 More Information

This section is not meant to be a complete description of the GAP system. Comprehensive documentation is distributed with the GAP system, consisting of tutorial, programming, extension and reference manuals and online help. The tutorial [4] and reference [3] manuals are good places to start learning about GAP.

Online help is available for almost all GAP functions by typing `?`*function-name* at the `gap>` prompt. GAP also supports tab completion, which can be used to find appropriate functions. By typing in part of a function name,

and pressing the `tab` key, either the name is completed or a list of names matching what was typed is listed.

3.2.8 Problems With GAP

When working with GAP, there were a few problems that surfaced. The first was the lack of floating point support in the released version. For work on GASP, I was able to obtain the unreleased development version of GAP, which has some floating point support built in. As floating point support is not very well integrated, into GAP at present, its use inside GASP has been quite limited.

The other problem with GAP is the lack of multiple global name spaces. This prevents the programmer from creating functions that are private to a particular share package. Although this did not cause any big problems when developing GASP, it means that people extending GASP have to be careful not to use function names that may be used in other GAP packages, as it may produce unpredictable results. Multiple global name space support is another feature that is planned for GAP in the future.

Although GAP has a number of problems, overall it proved to be a useful tool to use as a base for GASP.

Chapter 4

User Description of GASP

4.1 Introduction

This chapter gives an overview of GASP aimed at people who wish to use the existing functionality provided by GASP.

As GASP makes use of the graphics capabilities of the XGAP share package, the user must start XGAP in order to use the system. This is achieved by running the `xgap` command at the shell prompt. After XGAP has started and `gap>` prompt is shown, GASP must be loaded. This is achieved with the `RequirePackage` command:

```
gap> RequirePackage("gasp");
true
```

Here, the lines that do not start with `gap>` represent the output from GAP. At this point, the user may start using GASP.

4.2 An Example of GASP

To get a feeling of how to use GASP, it is best to look through an example. The following example brings up a window from which the user can simulate the Strauss Process using the Metropolis-Hastings algorithm. The example demonstrates setting up the algorithm for use with the simulation framework, and observing the behaviour through the *graphical user interface* (GUI).

```
gap> # create a 300x300 configuration of points
gap> config := PointConfiguration(0,0,300,300);
PointConfiguration( 0, 0, 300, 300 )
gap> # create the appropriate proposal function
```

```
gap> propose := CreateSimpleFlipPropose(1/2);
function( cnf ) ... end
gap> # create the appropriate acceptance check function
gap> check := CreateStraussCheck(1/900, 9/10, 15);
function( cnf, change ) ... end
gap> # create a window to run the simulation in
gap> GUISimulate(config, "Strauss", 300, 300, propose, check);
<graphic sheet "Strauss">
```

First we create an object `config` to hold the state of the simulation, which is a configuration (ie. finite, unordered list) of points on the plane. Next, proposal and acceptance check functions for the desired algorithm are created. The exact use of these is covered in Section 4.4.

Finally, we invoke the GASP command `GUISimulate`, which brings up a window from which we can observe what is going on. We also tell GAP to output log messages from the simulation to the console. The window should look similar to the one in Figure 4.1. The screen shot differs from the window initially shown by `GUISimulate` because the simulation has been running for a few iterations.

4.3 Using the Graphical User Interface

The *graphical user interface* (GUI) offers a nice way to experiment with the simulation framework. It is the window displayed when `GUISimulate` is called, as shown in Figure 4.1. The main area of the window displays the current state of the algorithm. In the example, the display area starts out empty, and as the simulation runs, it fills up with points.

There is a menu bar along the top of the window, from which a user can control the simulation. The first item on the menu bar is the standard XGAP 'Sheet' menu. It contains menu items to save the contents of the display area as a postscript file, and to close the window.

The second item opens the 'Iterate' menu, which is specific to GASP. It contains options to let the simulation run for various numbers of iterations. The display area updates after the desired number of iterations have been made.

In some cases, running the simulation for a particular number of iterations is not what is desired. Sometimes a user may want to run the simulation until a particular event occurs. This can be achieved with the 'Condition' menu. (To understand the condition menu fully, see Section 4.4 which describes the simulation framework). The conditions in the 'Condition' menu on which GASP can pause the simulation are:

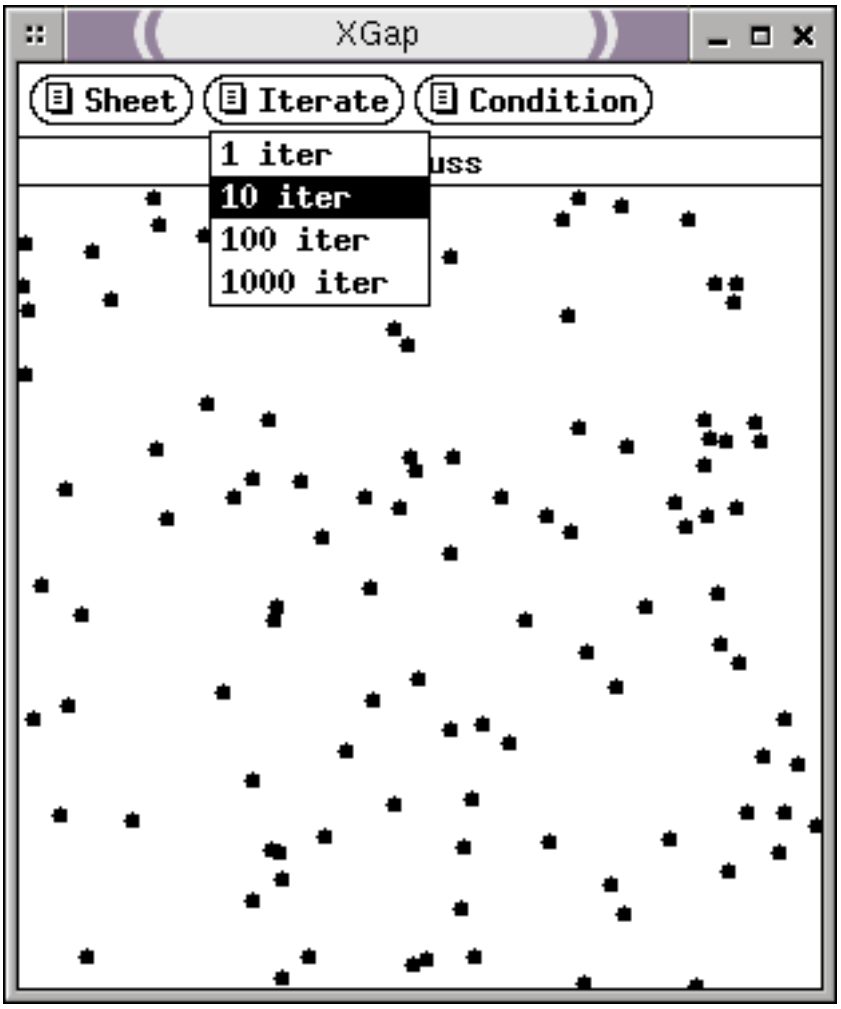

Figure 4.1: The simulation display window

Until first acceptance iterates until a proposition is accepted, after which the simulation stops.

Until first acceptance (***max 100 iter***) is like above, but also stops if 100 iterations are made without an acceptance. This option is useful in cases where it is not guaranteed that a proposition will ever be accepted.

Until first rejection iterates until a proposition is rejected, after which it stops.

Until first rejection (***max 100 iter***) is like above, but also stops if 100 iterations are made without a rejection.

Until propose add object (***max 100 iter***) iterates until we get an 'add object' proposal, or 100 iterations are made, whichever comes first

Until propose delete object (***max 100 iter***) iterates until we get a 'delete object' proposal, or 100 iterations are made, whichever comes first.

4.4 General Framework Used in Simulations

GASP provides a generic framework for running certain algorithms that can be expressed according to the flowchart describing the framework shown in Figure 4.2.

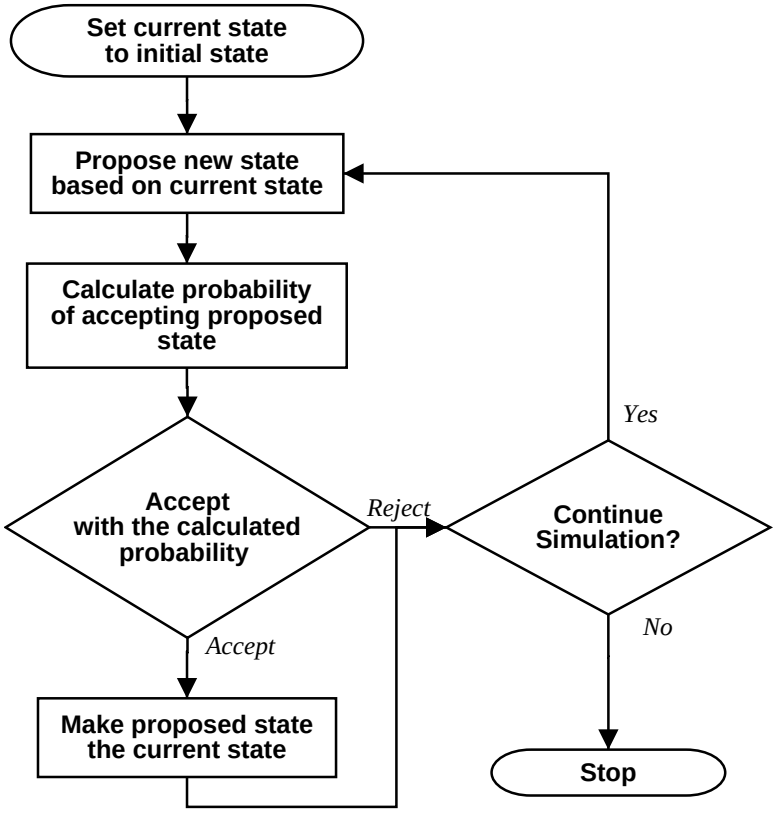

Figure 4.2: Simulation flow of control

The framework can also be described by the following set of rules:

1. propose a change to the current configuration.
2. calculate the acceptance probability for this proposed change. That is, the probability of applying the proposed change.

3. make a random decision whether to accept or reject the proposed change according to the acceptance probability calculated in step 2.

    - if it is accepted, apply the proposed change to the configuration.
    - if it is rejected, discard the proposed change.

4. check to see if the simulation should continue. If so, go to step 1 and repeat. If not, then the simulation stops.

The GASP simulation framework can run multiple types of algorithms. The different algorithms are defined by the combination of a *proposal* function and an *acceptance check* function. These two functions are passed to the `GUISimulate` function in order to run the algorithm.

In the case of the Strauss Process example above, we used a very simple proposal function which would either propose adding a random new point or removing a random existing point, with equal probability. Instead of writing the implementation of this function in full in the example, we used the `CreateSimpleFlipPropose` command, which returns a proposal function which adds a point with the desired probability.

For the acceptance check function, we want to use the reference formula used when simulating the Strauss Process with the Metropolis-Hastings algorithm, except that we calculate it in terms of the proposed next state, rather than the current one. Once again, we use a convenience function to implement this: `CreateStraussCheck`. This function returns the appropriate acceptance check function.

The other parameters passed into the main simulation function `GUI-Simulate` are the `Configuration` object, a continue checking function and optionally a file to log the progress of the simulation. The configuration object is simply the object that holds the state of the simulation. This is mainly the collection of objects (such as points) that make up the configuration. In the Strauss process example above, we used the `PointConfiguration` that is part of GASP as described in Section 4.5.1, which represents a configuration of points.

Deciding whether or not to continue the simulation is controlled from the GUI through the 'Iterate' and 'Condition' menus. More specialised control over deciding whether to continue is possible, provided the user is willing to write some simple GAP code.

It is possible to log the progress of the simulation if desired. This is achieved through the seventh optional argument to `GUISimulate`. To send the log messages to the GAP console, the user can pass `OutputTextUser()` as this argument. To append log messages to the file *filename*, the user can

pass `OutputTextFile(`*`filename`*`, true)` as the argument. If the argument is omitted, then log messages are not recorded.

4.5 Predefined Functionality

GASP supports a number of `Configuration` object types and provides several proposal and acceptance check functions. If the algorithm the user is interested in is supplied by GASP, then it is very easy to use GASP.

4.5.1 Configurations

A configuration object stores all information about the current state of a simulation. At a minimum, this is a collection of objects in the state, but may hold more information.

Configurations implement a number of operations that can be used to manipulate them (for example, counting how many objects are in the configuration or choosing a random object in the configuration). This makes it possible to implement algorithms that are independent of the actual configuration type used.

Currently, GASP provides configuration implementations for use in both point process and line process simulations. These are called the `PointConfiguration` type and `LineConfiguration` type.

The `PointConfiguration` object can be created using the following command:

```
config := PointConfiguration(sx,sy, swidth,sheight);
```

The first four parameters specify the top left corner (assuming the origin is in the top left corner of the GUI window, consistent with the coordinate system used for the computer screen) and the dimensions of the region the simulation will occur in.

The `LineConfiguration` object can be created in a similar fashion:

```
config := LineConfiguration(sx,sy, swidth,sheight);
```

The parameters have the same meaning as for `PointConfiguration`.

4.5.2 Proposal Functions

The proposal function for a particular algorithm is responsible for proposing changes to the state of a configuration. Given the current state of a configuration, it will return a proposed change (for example, adding a point). Usually it takes into account the current state of the configuration, but may also use the iteration number, some other counter or the output of a random

number generator to decide on the proposal. GASP provides a number of predefined proposal functions that may be useful for the user's simulation.

As most proposal functions have several parameters, it could potentially be time consuming to rewrite the function each time the user wanted to change a parameter. To save time, *function factories* are provided for the proposal functions that GASP provides. Function factories are simply functions that return functions. These factories provide an easy way to create members of families of proposal functions.

CreateSimpleFlipPropose

`CreateSimpleFlipPropose` creates proposal functions that choose between adding a randomly placed new point or removing an existing point at random. The probability of proposing to add a new point is passed in as a parameter to `CreateSimpleFlipPropose`. A proposal function of this type can be created like this:

```
propose := CreateSimpleFlipPropose(prob);
```

In the case that there are no points in the configuration, this proposal function always proposes to add a new point. This proposal function is appropriate for point processes using the Metropolis-Hastings algorithm.

CreateRedBlackPropose

`CreateRedBlackPropose` creates a proposal function that implements the Widom-Rowlinson 'Red-Black' point process as described in Section 2.4. For a full description of how the Widom-Rowlinson process is implemented in the GASP simulation framework, see the example in Section 5.2.5.

The user can create the proposal functions of this type with the following command:

```
propose := CreateRedBlackPropose(radius, mean_red,
                                 mean_black);
```

The first argument is the minimum required distance between points of different colours. The second is the mean for the Poisson distribution that is used to select the number of red points to add in one phase of the simulation. The third argument plays a similar role for the black points.

4.5.3 Acceptance Check Functions

The acceptance check function for an algorithm is responsible for calculating the probability of accepting the proposed new state for the configuration. This decision will generally be made based on the current and pro-

posed change. Based on the probability calculated by the acceptance check function, GASP will make a random decision whether or not to accept the proposed change.

There are a number of check functions provided with GASP. Similar to the proposal functions, factories are provided to make experimentation more convenient.

CreateStraussCheck

The function `CreateStraussCheck` creates acceptance check functions that implement the formula for the acceptance probability for the Strauss Process. The user can create a Strauss process acceptance check function with the following command:

```
check := CreateStraussCheck(beta, gamma, radius);
```

The arguments *beta* and *gamma* passed to this function are the β and γ parameters in the density function, and *radius* is the distance r, used when calculating $s(\mathbf{x}, y)$, as described in Section 2.3.

CreateFixedCheck

The function `CreateFixedCheck` creates an acceptance check function that always returns a particular probability regardless of the current state or proposed change. This type of acceptance check function may be created with the following command:

```
check := CreateFixedCheck(prob);
```

For this function, *prob* is the probability that is returned.

4.5.4 Problems Covered

By combining a proposal function and an acceptance check function as arguments to `GUISimulate`, a number of algorithms can be simulated when passed to `GUISimulate`. As seen in the example at the start of the chapter, `CreateSimpleFlipPropose` and `CreateStraussCheck` can be used to simulate a Strauss Process using the Metropolis-Hastings Algorithm.

The `CreateRedBlackPropose` and `CreateFixedCheck` functions can be combined to simulate the Widom-Rowlinson point process. In this case, we use a probability of 1 as the parameter for `CreateFixedCheck`.

4.6 Summary

With the information in this chapter, the reader should be able to use the existing proposal and acceptance check functions to simulate some algorithms. If an algorithm is not covered by the provided functions, then the user must be know how to write simple GAP functions and have a knowledge of the GASP programming interfaces. This is covered in the next chapter.

If the algorithm requires new configuration type, then a more detailed knowledge of GAP and the GAP object model is required. This is also covered in the next chapter.

Chapter 5

Implementation

This chapter covers the development of the GASP framework, along with information about how to extend GASP to handle new algorithms. It assumes some knowledge of programming in GAP.

5.1 Development

5.1.1 First Cut

When developing the GASP simulation framework, the aim was to have a simulation framework that could handle as many different spatial statistical problems as was reasonable while keeping the framework simple and easy to understand.

It is also desirable that code written for one problem should be usable in similar problems.

To achieve these goals, I first needed to write a general representation for configurations of geometric objects. This led to the creation of the `Configuration` and `PictureObject` types. The `PictureObject` type represents the geometric objects that make up the configuration (the name `PictureObject` was chosen because `Object` was already taken by GAP). Instances of these types cannot be created directly. They are *abstract*, so a user must use a derived type. For example, the `PointConfiguration` and `Point` types.

A first approach would be to adopt a simple framework that resembled the basic Metropolis-Hastings algorithm. The framework can be written in terms of the `Configuration` and `PictureObject` types. Parts of the framework that are specific to a particular algorithm are implemented as calls to functions a user must provide. This gave us the following basic

framework:

Given a configuration to store the state of the simulation, and user supplied functions `propose` and `acceptancecheck` that implement the algorithm, perform the following steps:

1. call the user supplied function `propose` to propose a change to the configuration.
2. call the user supplied function `acceptancecheck` function to calculate the probability of accepting the proposition.
3. make a random decision whether to to accept with the probability calculated in the previous step:
   - if accepted, make the change to the configuration
   - if rejected, do nothing
4. repeat from first step

At this point, the return value of the function `propose` is limited to proposing to create a new object or delete an existing object. These are represented by lists of length 2 – the first element being a constant representing the type of change, and the second being the object that we are adding or deleting.

This simple framework exhibits a number of limitations. For instance, the `propose` and `acceptancecheck` functions are implemented in terms of the particular `Configuration` used in the algorithm. This means that those functions would need to be modified if we wanted to run the same algorithm with a different `Configuration`. This framework also limited the allowable propositions to adding and removing points.

5.1.2 Virtualisation

In order to get around the first problem, I virtualised most of the actions that are performed on `Configuration`s and `PictureObject`s. This means that rather than accessing the configuration directly, the algorithm calls functions provided with the configuration implementation that perform the actions. We achieve this through a number of GAP operations. While it simplifies the job of the person implementing new algorithms, this change adds the burden of implementing a number of operations (such as calculating the distance between two objects, counting objects in a configuration, checking for objects close to a particular object, etc). As I expect that there will be

many more algorithms implemented for the framework than configuration types, the tradeoff seems appropriate.

This made it possible to implement new algorithms in such a way they could be used with different types of configurations without modification, which also gives more flexibility to change how various parts of the framework are implemented.

The list of operations for use when writing proposal or acceptance check functions is covered in Section 5.2

5.1.3 Change Objects and the Change Log

In order to lift the restriction on the allowed types of change that may be made to configurations, rather than using a two element list, I created a new type called a `Change` object. The idea behind `Change` objects in the simulation framework is very similar to the way undo capabilities are implemented in many applications. These objects package up all the information about how to *apply* themselves to the configuration, and how to *revert* any changes they made to the configuration.

We could now modify the simulation framework to look like:

1. call the user defined `propose` function, which returns a `Change` object representing the proposed change to the configuration.
2. apply the `Change` object to the configuration.
3. call the `acceptancecheck` function to calculate the probability of accepting the change.
4. make a random decision whether to accept the change with the calculated probability. If the proposal is rejected, revert the change.
5. go to step 1 and repeat.

All information about an iteration is written to a log file. This includes the change that was proposed, the calculated probability and whether the proposal was accepted. More information about the log file is found in Section 5.4.1

5.2 Implementing New Algorithms

This section covers the parts of the GASP application programming interface required to write new proposal and acceptance check functions. The user will

need to have a basic knowledge of the GAP syntax along with knowledge of the following programming interfaces. The GAP Tutorial [4] is a good introduction to using the GAP language. This information is necessary for simulating processes that are not covered by the existing algorithms distributed with GASP.

More specifically, the user will need to know how to write functions and use simple programming structures.

5.2.1 Configuration Operations

In order to implement the proposal and acceptance check functions, it will be necessary to make calculations based on a given configuration. A number of operations are provided to access the configuration so that the algorithms the user implements are not dependent on a particular configuration implementation.

These operations have to be implemented for each configuration type. Here is a list of the operations most useful when implementing proposal and acceptance checking functions:

ConfigWindowArea

The `ConfigWindowArea` operation can be used to calculate simulation region. This is used in a number of problems when calculating acceptance probabilities:

```
area := ConfigWindowArea(config);
```

Here, `config` is a `Configuration` object.

RandomNewObject

In the proposal function for a particular algorithm, it will usually be necessary to create new objects for the configuration. The operation `RandomNewObject` can be used to do this:

```
obj := RandomNewObject(config);
```

ChooseRandomObject

`ChooseRandomObject` picks a random existing object from a given configuration. This might be useful in a proposal function.

```
obj := ChooseRandomObject(config);
```

CountObjects

The operation `CountObjects` counts the total number of objects in a given configuration. It may be useful when calculating the acceptance probability.

```
num := CountObjects(config);
```

CountObjectsWithMark

For algorithms where we work with objects with marks attached (marks were defined in Section 2.5), it may be useful to count the objects that have a particular mark set. This is done with the operation `CountObjectsWithMark`:

```
num := CountObjectsWithMark(config, markname);
```

The argument *markname* is the name of the mark that must be set on the objects that are being counted. For more information on the marks interfaces, see Section 5.2.2.

GetObjectsWithMark

Some algorithms may require a list of all the objects that have a particular mark. This can be achieved with the operation `GetObjectsWithMark`:

```
object_list := GetObjectsWithMark(config, markname);
```

The argument *markname* for this function has the same meaning as the one in the `CountObjectsWithMark` operation.

CloseObjects

For some algorithms we require some information about how many objects in the configuration are within a certain distance of a particular object. The `CloseObjects` operation is used to calculate this:

```
num := CloseObjects(config, object, rad);
```

This is the function is basically the same as the function $s(\mathbf{x}, y)$ used in the calculation of the acceptance probability of the Strauss Process. As with $s(\mathbf{x}, y)$, if *object* is already part of the configuration, then it will not be included in the count.

CloseObjectsWithMark

For algorithms that involve marks, it may be necessary to perform a count as in `CountObjects`, but only counting objects that have a particular mark set. For this, the `CountObjectsWithMark` operation can be used:

```
num := CloseObjectsWithMark(config, object, rad,
                            markname);
```

5.2.2 The Marks Interfaces

For algorithms that involve marks, the user may need to know about the marks interfaces for objects in the GASP package. When creating new objects, it is necessary to set the appropriate marks on the object. When writing the acceptance check function, it may be necessary to check for a mark on an object.

The marks interfaces in GASP work on a set of boolean marks associated with an object. For multi-valued marks, it must be implemented as a set boolean marks for each possible value. The algorithm should make sure that it does not set conflicting marks on an object.

The marks for an object are manipulated with three different operations:

`AddMark(object, markname)` sets the mark *markname* on the given object.

`DelMark(object, markname)` unsets the mark *markname* on the given object.

`HasMark(object, markname)` returns true if the mark *markname* is set on the given object, and returns false otherwise.

5.2.3 The Proposal Function

The proposal function for an algorithm decides on a change to propose based on the current state of the configuration. The prototype for all proposal functions is:

```
change := propose(config);
```

The return value is a change object representing the change that is proposed. These *change objects* contain all the information required to move to the next state. There are a number of predefined change objects that can be used in the algorithm. It is a little more involved to implement new change object types.

AddObjectChange

The `AddObjectChange` type is used when proposing the addition of a new object. The user can create such a change with the following syntax:

```
change := AddObjectChange(object);
```

The object inside the change can be accessed with `change!.obj` if needed later.

DelObjectChange

Removing an object is achieved with the `DelObjectChange` type. This type is created with the following function call:

```
change := DelObjectChange(object);
```

Again, access to the object that is being removed is available as `change!.obj`.

ReplaceObjectChange

In the case of moving an object, a new object in the new position should be be created and a change that *replaces* the old object with the new one should be used. This type of change can be created with the following syntax:

```
change := ReplaceObjectChange(addobj, delobj);
```

The two objects in the change can be accessed as `change!.addobj` and `change!.delobj` if needed.

ReplaceMultipleObjectsChange

If the algorithm requires replacing one set of objects with another in the configuration, then `ReplaceMultipleObjectsChange` can be used.

```
change := ReplaceMultipleObjectsChange(addobjs, delobjs);
```

The two arguments to this function are lists of objects that are to be replaced.

5.2.4 The Acceptance Check Function

The acceptance check function for the algorithm calculates the probability of accepting the proposed state. The prototype for all acceptance check functions is:

```
probability := acceptancecheck(config, change);
```

When the acceptance check function is called, the configuration will already be in the proposed state. The change that was performed is passed as an argument in case that is needed when calculating the acceptance probability.

For some algorithms, it may be necessary to check what *type* of change was proposed. For each of the change objects, there is a boolean function to check for that type. These are just the name of the change type with 'Is' prepended. There are `IsAddObjectChange`, `IsDelObjectChange`, `IsReplaceObjectChange` and `IsReplaceMultipleObjectsChange`.

5.2.5 An Example

To put this into practice, we look at an example of implementing an algorithm from scratch with GASP. We reimplement the Widom-Rowlinson point process, which does not fit as neatly into the simulation framework as some other algorithms.

For this process, we implement the bulk of the algorithm in the proposal function. The proposal function performs one phase of the algorithm (that is, remove all points of a particular colour, and add new points of that colour). The acceptance check function always returns a probability of 1 to accept the change. We also keep some extra information in the configuration to keep track of whether we are supposed to be adding red or black points in the next iteration.

For this example, we use a 300 by 300 configuration, a minimum distance of 15 and 100 and 120 as means of the Poisson distributions for the number of red and black points respectively.

```
# create a configuration object
config := PointConfiguration(0,0,300,300);
# the first iteration will add black points
config!.isRedIter := false;

rad := 15;
# set up Poisson random number generators
red_rnd := CreatePoissonRNG(100);
black_rnd := CreatePoissonRNG(120);

# define the proposal function for the algorithm
propose := function(config)
    local del, add, obj, num, i, RandRat;

    # are we adding red points during this iteration?
    if config!.isRedIter then
        # we want to replace all existing red points
        del := GetObjectsWithMark(config, "red");
        add := [];
        num := red_rnd();  # number of points to add
        for i in [ 1 .. num ] do
            obj := RandomNewObject(config);
            AddMark(obj, "red");
            # add the object if it is far enough away
            # from the black points
            if CloseObjectsWithMark(config, obj, rad,
                                    "black") = 0 then
```

```
28                  Append(add, [ obj ]);
                fi;
            od;
        else
            # we want to replace all existing black points
            del := GetObjectsWithMark(config, "black");
            add := [];
            num := black_rnd();  # number of points to add
36          for i in [ 1 .. num ] do
                obj := RandomNewObject(config);
                AddMark(obj, "black");
                # add the object if it is far enough away
40              # from the red points
                if CloseObjectsWithMark(config, obj, rad,
                                        "red") = 0 then
                    Append(add, [ obj ]);
44              fi;
            od;
        fi;
        config!.isRedIter := not config!.isRedIter;
48      return ReplaceMultipleObjectsChange(add, del);
    end;

    # now the acceptance check function
52  acceptance_check := function(config, change)
        return 1;
    end;

56  # finally, start up the graphical user interface
    GUISimulate(config, "Red-Black", 300, 300,
                propose, acceptance_check);
```

This example will bring up the GUI where the user can run the simulation. On lines 8 and 9, we create Poisson random number generators used to calculate the number of points in the simulation. The rest of the example is fairly straight forward. The simulation should look something like Figure 5.1.

This particular algorithm can also be used through the existing `CreateRedBlackPropose`, and the `CreateFixedCheck` with a probability of 1.

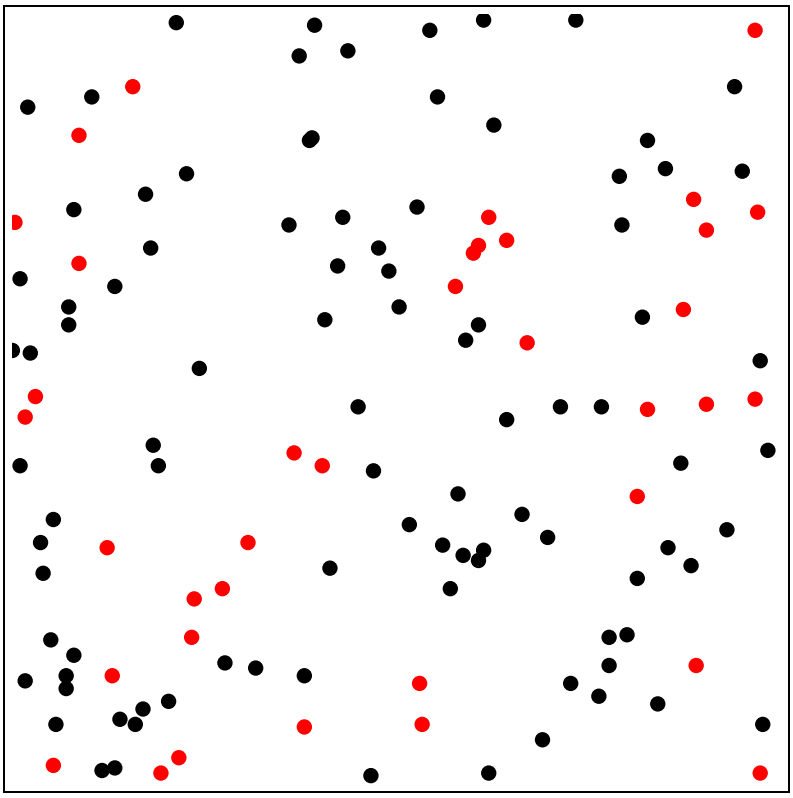

Figure 5.1: Red-Black Point Process

5.3 Creating New Configuration Types

For some problems, the supplied configuration types are not sufficient. Either the type of objects that make up the state are not covered by the provided configuration types or the existing configurations are not efficient enough.

As an example of the second case, if we had an algorithm that performed a lot of checks to count the objects close to a particular object, then it may be appropriate to use a different configuration type that was more optimised for that sort of operation. Such an optimisation would be to sort the objects into bins according to where they are located in the plane. Now to find all objects within a given radius r of a particular object y, we only have to check the bins that are within a radius r of the bin that contains y. While this internal representation for the configuration speeds up these distance checking operations, it slows down other operations such as counting all objects in the configuration.

The creation of new configuration types requires knowledge of the GAP object model. This is covered in the *Programming in GAP 4* manual [5] that

is distributed with GAP.

New configuration types must be derived from the abstract object category `IsConfiguration`. If a new object type is required, it must derive from `IsPictureObject`.

Any new configuration must implement all the operations listed in the previous section. In addition to those operations, the new configuration must implement a number of operations that are used to implement various parts of GASP. The extra operations are:

`AddObject` - this operation is used to add an object to the configuration. It is not designed to be called by algorithms directly. Instead, it is used to implement change objects that add objects to the configuration. The prototype for this operation is:

```
AddObject( config , object );
```

`DelObject` - this operation is used to delete an object to the configuration. As with `AddObject`, it is not designed to be called by algorithms directly. The prototype for this operation is:

```
DelObject( config , object );
```

`DrawConfiguration` - this operation is used to draw the configuration of objects to a graphic sheet. This will generally be implemented by calling the `DrawObject` method on each object in the configuration. The prototype is:

```
DrawConfiguration( config , graphicsheet );
```

The objects that make up the configuration need only implement a few operations. The two required operations are:

`SquareDistance` - this operation calculates the square of the distance between two objects. The prototype for `SquareDistance` is:

```
distance := SquareDistance( object , otherobject );
```

`DrawObject` - this operation is responsible for drawing the object to an XGAP graphic sheet. The prototype for `DrawObject` is:

```
DrawObject( object , graphicsheet );
```

Provided that the new object type is implemented as an objectified record, then the marks operations need not be implemented. Otherwise, those interfaces must be reimplemented for the object type.

For more information on implementing new configurations, see the implementation of the `PointConfiguration` type in the GASP package.

5.4 Change Objects

Change objects are used in the GASP framework to represent all changes made to configuration objects. GASP comes with a number of useful change object types, which were described in Section 5.2.3.

New change objects must derive from the abstract category `IsChange`. A change object contains all the information that is required to apply the change to the configuration, or to revert it once applied. These two actions on the configuration are handled by the following two operations:

```
ApplyChange( change, config );
RevertChange( change, config );
```

These operations must be implemented for each new change type. Usually a change object is implemented in terms of the `AddObject` and `DelObject` operations of the `Configuration` type, so that it may be used with more than one configuration type.

It is also desirable that the new change object type implement the standard GAP operation `PrintObj`, which is used to print a representation of the object to a file. This operation will be called when writing the type of change to the log file. For this reason, it is required that `PrintObj` write the code necessary to recreate the object. If this operation is not implemented correctly, the usefulness of the log file is greatly reduced.

5.4.1 The Log File

Provided that the user turned on logging, all changes to the configuration are written to a log file. The lines in the log file look similar to:

`[` *change*`,` *acceptprob*`,` *accept* `]`

Here, *acceptprob* is the calculated probability of accepting the change and *accept* is a boolean representing whether or not the change was actually accepted and applied.

The log file contains all the information required to replay the simulation to recreate the final state of the simulation. In fact, it is possible to replay the simulation up to any point by using the log. This could be used to play back the simulation at a faster rate.

The log is also a useful source of statistics about the run of the simulation. For instance, we can look for patterns in the types of changes that are proposed and the acceptance rates. Some interesting information the log could provide are:

- states for which proposals are given very low acceptance probability.

- lengths of runs of acceptances or rejections.
- changes in the acceptance probability over time.

I have not completed the framework to used to parse the log file for post simulation analysis. When the log parsing framework is complete, it will be added to the GASP package. However, parsing the log is very simple, as each line is a valid GAP expression. So it is possible to read the log before the log parsing framework is complete.

Chapter 6

Conclusion

The GASP package should prove to be a useful tool for simulation of spatial statistical processes. The package is designed to be as extendable as possible, I expect that it will become more useful in the future as more algorithms are added to the library accompanying GASP.

In hindsight, it may have been a better decision to choose a different language. Parts of the framework that required floating point support could only be tested by people who had access to the development version of GAP, as there are no publicly released versions of GAP with floating point support. Once the next version of GAP is released, this barrier will be removed.

If GASP is too slow to run a simulation, GAP comes with a GAP to C translator, which can increase the speed of a program. Alternatively, GASP can be used to explore an algorithm, before using another program to perform a longer simulation.

I plan to add some more features to GASP and submit it as a share package for GAP.

Bibliography

[1] D. Daley and D. Vere-Jones. *An introduction to the theory of point processes*. Springer, 1988.

[2] The GAP Group, Aachen, St Andrews. *GAP – Groups, Algorithms and Programming*, 1999.
`http://www-groups.dcs.st-and.sc.uk/~gap/`.

[3] The GAP Group, Aachen, St Andrews. *The GAP Reference Manual*.
`http://www-groups.dcs.st-and.ac.uk/~gap/Manual4/ref/`.

[4] The GAP Group, Aachen, St Andrews. *The GAP Tutorial*.
`http://www-groups.dcs.st-and.ac.uk/~gap/Manual4/tut/`.

[5] The GAP Group, Aachen, St Andrews. *Programming in GAP 4*.
`http://www-groups.dcs.st-and.ac.uk/~gap/Manual4/prg/`.

[6] James Henstridge. The GASP share package for GAP.
`ftp://ftp.daa.com.au/pub/james/uni/gasp.tar.gz`.

[7] B.D. Ripley. *Spatial statistics*. John Wiley and Sons, New York, 1981.

[8] D.J. Strauss. A model for clustering. *Biometrika*, 63:467–475, 1975.

[9] Guido van Rossum. The python language.
`http://www.python.org/`.

[10] W.R. Venables and B.D. Ripley. *Modern Applied Statistics with S-PLUS*. Springer, 3rd edition edition, 1999.

[11] S. Richardson W. Gilks and D.Spiegelhalter. *Markov Chain Monte Carlo in practice*. Chapman & Hall, 1998.

[12] B .Widom and J.S. Rowlinson. New model for the study of liquid-vapor transitions. *The Journal of Chemical Physics*, 52:1670–1684, 1970.