DEVELOPMENT OF A COLLISION TABLE FOR THREE DIMENSIONAL LATTICE GASES

PETER LAKE

SEPTEMBER 1992

NOTIFICATION OF SUBMISSION OF THESIS

This thesis is submitted to the Faculty of Science of the University of Cape Town, in fulfillment of the requirements for the degree of Master of Science in Applied Mathematics.

DECLARATION

I, Peter John Lake, hereby declare that this thesis is my own unaided work. It is submitted for the degree of Master of Science in Applied Mathematics to the University of Cape Town. This work has not been submitted before, for any degree or examination, to this or any other university.

Signed by candidate

Signature Removed

P. J. Lake

ABSTRACT

A lattice gas is a species of cellular automaton used for numerically simulating fluid flows. TransGas [9], the lattice gas code currently in use at the CSIR, is based on the FHP-I model [5], and is used to perform various two-dimensional flow simulations. In order to broaden the scope of the applications in which lattice gases can be used locally, the development of a three-dimensional lattice gas capability is required.

The first major task in setting up a three dimensional-lattice gas is the construction of an efficient collision rule generator which will determine collision outcomes. For suitability to local applications, the collision rules should be chosen in such a way as to maximise the Reynolds coefficient of the flow, while conserving quantities such as mass and momentum. Part of the task thus becomes an optimisation problem.

When expanding from two to three dimensions, the number of possible collision rules increases from 64 to 16777216. If a complete collision rule table is used for determining collision outcomes, storage problems are encountered on the available hardware. Selection and optimisation of collision rules cannot be done by hand when there are so many rules to choose from. Selection of rules is thus non-trivial.

The work outlined in this thesis provides the CSIR with a 3-D lattice gas collision table which is well suited to the available hardware capabilities. The necessary theoretical background is considered, and a survey of the literature is presented. Based on the findings of this literature study, various methods of collision outcome determination are implemented which are considered to be suitable to the local needs, while remaining within the constraints set by hardware availability. An isometric collision algorithm, and a reduced collision table are generated and tested.

A measure of the overall efficiency of a lattice gas model is determined by two factors, namely the computational efficiency and the implementation efficiency. In testing a collision table, the first is characterised by the rate at which post-collision states can be determined, and depends on the hardware and programming techniques. The second factor can be expressed by means of a number called the Reynolds coefficient, which is defined and discussed in the following chapters. The higher the Reynolds coefficient of a model, the greater the scope of flow regimes which may be simulated using it. Another advantage of having a high Reynolds coefficient is that the simulation time required for a given flow regime decreases as the Reynolds coefficient of the model increases.

The overall efficiency of the isometric model is too low to be of practical use, but a significant improvement is obtained by using the method of reduced tables. In the isometric case, the number of collision outcomes that can be determined per second is similar to that of the reduced table, but the Reynolds coefficient is very much lower. Simulation of a flow regime with a Reynolds number of about 100, on a lattice of size 128^3, over 20 thousand timesteps, making use of the isometric model, would take of the order of a few years to complete on the currently available hardware. Since the simulation parameters mentioned above are typical of the local requirements for lattice gas simulations, this method is obviously unsatisfactory. The isometric method does however serve as a useful introduction to three-dimensional lattice gas collision rule methods.

The reduced collision table has been constructed so that it maintains semi-detailed balance, and the Boltzmann Reynolds coefficient has been calculated. In the reduced collision table model, the efficiency is higher than the isometric case in respect of both the rate at which collision outcomes can be determined, and in terms of the Reynolds coefficient. As a result of these improvements, the simulation time for the exact case mentioned above would reduce to the order of days, on the same hardware. This simulation time is sufficiently low for immediate practical application in the local environment.

ACKNOWLEDGEMENTS

Thanks are due to Dr. I. M. A. Gledhill of the CSIR in Pretoria, South Africa for posing the problem and for technical advice and helpful discussions throughout the duration of this thesis.

Dr. R. Smart of the University of Cape Town, South Africa and Dr. I. M. A. Gledhill co-supervised this thesis, and I am indebted to them for this.

I am grateful to Mr. M. A. Honman of the CSIR, Pretoria, South Africa, for providing helpful advice as regards gaining improved computing performance from the local hardware system.

All tradenames and trademarks mentioned in the present text are respectfully acknowledged.

LIST OF CONTENTS

Page

APPENDICES

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Conventional computational fluid dynamics (CFD) methods are frequently unsuitable for the analysis of flow regimes such as turbulence, flow through porous media, multiphase flow, and phase transition. Complex geometries, and geometries which require highly complicated boundary conditions, also present difficulties.

The CFD facility of the CSIR is concerned with the analysis by computational methods of fluid flows in a variety of regimes. A deficiency exists in its capability to analyse complex flow situations of the types mentioned above, in three dimensions. One possible method of tackling these difficulties is provided by a newly emerging sub-field of the cellular automaton field, known as lattice gases.

It is common practice for a survey of relevant literature to be presented as an opening chapter in a thesis. A brief history of the field frequently appears in the introduction. It will however be necessary to introduce several lattice gas concepts as clarification to the reader, before the literature survey and historical overview can be embarked upon. For this reason, a summary of theoretical considerations relevant to the construction of collision tables is presented in chapter 1. The historical development of the lattice gas field and the literature survey is presented in chapter 2. In order to define the objective of this thesis, a brief description of the functioning of a lattice gas is required which necessitates the use of some of the ideas discussed in chapter 1. For brevity of the introduction, technical details and historical credits have been kept to a minimum here. Throughout the introduction, the reader is given references to later sections, where fuller definitions and discussions can be obtained if desired.

A cellular automaton is essentially a finite and discrete set of elements within a defined domain or lattice, and a set of rules which all of these elements obey. The elements in the domain are usually computationally represented by an array of Boolean values. At each discrete timestep, every element in the domain is updated simultaneously, according to the rules of the automaton. These rules usually specify the updated state of an element in terms of its previous state and those of its nearest neighbours. The reader is referred to Wolfram [32] for basic theory and applications of cellular automata.

Although the notion is spurned by Hénon [14], a lattice gas is generally accepted as being a special type of cellular automaton used for the modelling of fluid flows. This method of simulation traces the movement of particles, which are restricted to motion along a network of straight lines forming a regular lattice. The points where lattice lines cross one another are known as sites. A section of a lattice line joining two neighbouring sites is known as a link. All particles present at each site collide once per discrete timestep, and these collisions are governed by a set of collision rules. Under sufficient conditions of lattice symmetry and collision rule choices (which are discussed in sections 1.4 and 1.3 respectively), the macroscopic behaviour of the particles is seen to simulate the Navier-Stokes equations for certain flow regimes.

One alternative method of performing lattice gas simulations is the lattice Boltzmann approach, as pioneered by McNamara and Zanetti [34]. In this method, the lattice gas automaton is translated into a Boltzmann model. The site populations then become real numbers instead of Boolean numbers, and are controlled by a Boltzmann equation derived from the lattice gas model. This thesis will however concentrate on standard lattice gas techniques.

A lattice gas therefore simulates the particles of a fluid discretely in space at the sites of a regular lattice, with each site in the lattice being joined to its nearest neighbours by links. An exclusion principle allows each link to be occupied by at most one particle at any given time. The presence or absence of a particle on a link defines the state of that link. If the presence of a particle on a particular link is denoted by a 1, and the absence of such a particle by a 0, it can be seen that the state of the link can be represented by an element of the Boolean set $\{0;1\}$. Let the state of link number i be represented by s_i, which can assume the values 0 or 1. The state s of an entire lattice site can then be represented by an array of these Boolean numbers, one for each link, eg. $s=(s_1,s_2,s_3,...s_n)$, where n is the number of links joining a site to its nearest neighbours. Section 1.2 gives a more technical description of this material for the case of a famous lattice gas called the FHP model [5].

Lattice gas timesteps are discrete, and each consists of two phases, propagation and collision. During propagation, each particle moves from its current site to the neighbouring site in the direction of its velocity vector. During collision, all particles present at each site collide. The outcome of the collision phase is governed by a set of rules called the collision rules, by which the post-collision (output) state of every specified lattice site can be determined from its pre-collision (input) state.

The rules can be of different types and can be implemented by different means. If the rules are deterministic, then there is one explicit output state for each input state. Stochastic rules allow a choice of output states for certain input states. The reason for the fact that a choice is not allowed in the case of every state, is that for certain input states, there is only one valid output state. As far as implementation is concerned, collision rules can either be calculated as required during a lattice gas simulation (runtime rule generation), or they can be pre-calculated, and stored in the form of a lookup table called a collision table. This latter method is the more common, since it has an advantage in terms of reducing the time taken to obtain the collision outcomes. Runtime rule generation is slower, but requires far lower storage capabilities. This tradeoff is more fully discussed in sections 2.4, 2.5 and 2.8.

Since every possible input state of a site must be catered for in a collision table, the number of required rules depends on the number of different states which a site can assume. In the previously mentioned two-dimensional FHP model, there are six links per site, and since each of the links can assume a state of either 0 or 1, there are $2^6 = 64$ different possible site states. A two-dimensional lattice gas model, TransGas [9], is already in use at the CFD facility, based on the FHP model discussed in section 1.2.

Three-dimensional lattice gas capability is required locally in order to perform three-dimensional simulations of the previously mentioned flow regimes which are unsuited to conventional flow analysis methods. The first step towards this realisation is the generation of a three-dimensional collision rule set. For reasons of lattice symmetry (which will be discussed in section 1.3), the four-dimensional face centred hypercubic (FCHC) lattice structure is chosen from among other possibilities (mentioned in section 2.3). The FCHC is adapted for use in three-dimensional simulations.

In moving from a two-dimensional FHP type model to the FCHC model, the number of links joining each lattice site to its nearest neighbours rises from six to 24. As in the two-dimensional case, each link still has two possible states, 0 and 1. The number of possible states that a site can assume therefore rises to 2^{24}. Due to the fact that each possible pre-collision state must still be provided with an output state in a collision table, the number of collision table entries increases to the same enormous number, just under 17 million. The requirement of such a large collision table poses two major problems.

The first problem is one of storage requirements. A full collision table of 2^{24} entries would occupy more local memory than the available hardware can offer. The computer equipment on hand consists of a powerful engineering workstation, and a network of 40 Transputers, each with very limited local memory. More information concerning this hardware is given in section 1.2. Many methods mentioned in the literature (discussed in section 2.5) do involve the storage of the entire collision table, since in these cases, suitable hardware is available. Due to local hardware limitations, these methods are not within the capabilities of the present project, and other techniques must be investigated. The second problem concerns rule selection. Collision rules are chosen in such a way as to satisfy certain conditions (as discussed in section 1.4), and to minimise certain physical quantities for efficiency reasons (as laid out in section 1.7). In a two-dimensional case, rule selection can be performed manually. In the three-dimensional case, with nearly 17 million rules to choose from, optimisation and selection must be performed by computational methods. Rule selection is thus non-trivial.

For the above reasons, non-standard methods of collision rule generation need to be investigated. The aim of this thesis is to provide a means of obtaining collision outcomes which is well suited to the applications required in the CFD facility at the CSIR, and which are within the limitations imposed by the available hardware. The methods used are motivated throughout the present text.

Outline of Thesis:

In order to facilitate the reading of the literature survey, the first chapter of the thesis is devoted to discussing issues of the relevant theory, which require consideration during the construction of a lattice gas collision rule set.

The second chapter contains a survey of the literature available in the field of lattice gas collision table generation. A brief history of the development of the lattice gas field is provided. Due to the fact that this thesis concerns the construction of three-dimensional collision rule sets, very little is presented regarding two-dimensional lattice gases, other than of a brief historical nature. Lattice gas simulation results are also beyond the scope of this thesis except in cases where they are of direct relevance to the testing of a collision rule model. Some of the difficulties encountered in constructing computationally efficient rule tables are discussed. A survey of recent developments in the field of collision table construction for three-dimensional lattice gases is presented. At the close of the chapter, recommendations, based on this literature survey and the theoretical considerations of chapter 1, are given as to the proposed path that the project should follow.

The third chapter discusses a method of providing collision rules which is not reliant on the storage of a collision table at all. This is the isometric collision algorithm invented by Hénon [11]. The method involves the implementation of a procedure which generates appropriate collision rules as they are required, during the lattice gas simulation. This so-called runtime rule generation obviates the need for a cumbersome collision table. The first three-dimensional lattice gas collision rules in South Africa were provided by this implementation.

In the isometric method, the post-collision state of a site can always be obtained from its pre-collision state by applying suitable rotations and/or inversions of the coordinate axes to the input state. The method preserves momentum and mass, and attempts to minimise kinematic viscosity. Some of the relevant theory required to perform the task is considered. A detailed discussion of these methods, their implementation, and the findings are presented in this chapter.

The fourth chapter discusses the implementation of a collision rule table in which only some of the collision rules are stored. The rest of the rules are calculated as required at runtime, from the stored ones, by applying the basic symmetry of the problem. This method of storing a reduced collision table follows the work done by Hénon in [15] and [13]. The relevant theory is discussed, implementation methods and problems encountered are outlined, and the findings are presented.

The fifth chapter discusses improvements made to the local implementation of the reduced collision table of chapter 4. Some methods of increasing the rate of determination of post-collision states are discussed, one of which greatly reduces dependence on computer intensive operations such as matrix multiplication. The rate at which collision outcomes can be determined is improved by a factor of over two.

The reduced collision table constructed in chapter 4 violates semi-detailed balance, in that certain states appear as the output states for multiple input states. A modification to the optimisation process used in constructing the collision table is made, so that semi-detailed balance is maintained. The Boltzmann Reynolds coefficient for this reduced collision table is calculated, and inferences are drawn from the result.

In the sixth and final chapter, conclusions are drawn, and future possible work is discussed.

1 BASIC THEORETICAL CONSIDERATIONS

1.1 Advantages of Lattice Gas Techniques over Conventional CFD

Studying certain flow regimes (where low Reynolds numbers prevail) by using lattice gas methods has some advantages over using the more conventional techniques of computational fluid dynamics (CFD). Simulations of phenomena such as turbulence, flow through porous media, multiphase flow, and phase transition pose difficulties for conventional methods, but can be resolved by lattice gas techniques. The situation is similar for flow problems with complex geometries, and geometries which require highly complicated boundary conditions. In [26], Somers and Rem outline a few of the advantages of lattice gas simulations, but in general, it is accepted that:

- certain complex geometries and boundary conditions which pose problems for conventional CFD methods (e.g. flow through porous media) are more easily modelled by lattice gas methods.

- the use of integer calculations and Boolean number operations instead of real number arithmetic prevents numerical instabilities and rounding errors from becoming a factor.

- this use of simple Boolean operations, as opposed to the large amounts of real number arithmetic required by conventional techniques, results in a saving of computational resources.

- the way in which updating of the lattice can be done by direct bit manipulations makes the method intrinsically more suited to a digital computer environment.

The use of lattice gas simulation techniques also has its disadvantages. Examples of problems encountered are that the fixed mesh size prevents greater resolution from being obtained in regions of interest, lattice gas results are generally noisy, and difficulties are experienced when attempts are made to simulate additional physical effects, such as temperature. The use of lattice gases as a flow analysis tool is also severely limited by the low Reynolds numbers that can be simulated using these techniques. It is reported in [6] that the highest Reynolds number flow that can be simulated by three-dimensional lattice gas methods is roughly 2000, using present state of the art computer equipment. (Reynolds numbers of flows are discussed in section 1.7.)

1.2 TransGas and Problems Underlying Expansion from 2-D to 3-D

In 1986, Frisch, Hasslacher and Pomeau [5] showed that a simulation of the two-dimensional Navier-Stokes equations can be performed using lattice gas modelling techniques, with acceptable accuracy. Their model, now known as the FHP-I model, is used as the basis for the current CSIR two-dimensional lattice gas model, TransGas [9].

A brief description of the computer equipment available for local lattice gas simulations is provided for clarity. TransGas has been implemented on a network of 40 Transputers, each of which has very limited local memory. A Transputer is essentially a single-chip parallel processor which encompasses its own memory, communication primitives and floating point hardware. When connected in a network, several Transputers form a parallel computer. The lattice can be partitioned into several overlapping sections and the flow on each lattice section can be analysed on a separate Transputer. By simultaneously performing these smaller simulations, with communication between the Transputers handling adjacent lattice sections, a parallel simulation of the entire lattice can be performed. This parallel processing is discussed in [8].

A UNIX-based workstation is also available which is capable of approximately 22 MIPS. In floating point terms, it has about one sixteenth of the computing power of a Cray XMP. The computing performance of engineering workstations is increasing rapidly, and it is envisaged that a more powerful model will be purchased for the local environment in the near future. The workstation environment has been chosen for the implementation of three-dimensional collision rule generation, since the local memory per Transputer is too small to be of practical use in a problem of this magnitude.

In the TransGas model, fluid particles are modelled discretely in space and time, and are restricted to motion on a regular hexagonal lattice, a section of which is depicted in figure 1. The points where the lattice lines cross one another are known as sites, and each site is connected to its six nearest neighbours by segments of the lattice lines known as links. The unit direction vectors of the six links leaving each site are given by:

$$\left(\cos\frac{2\pi i}{6};\sin\frac{2\pi i}{6}\right);\ i=0,1,\ldots,5 \qquad (1)$$

Figure 2 shows one site in a FHP type lattice, situated at the intersection of the six links to its nearest neighbours. The link numbering convention taken as standard for this work is indicated. An exclusion principle allows at most one particle to reside on each link, and so each link has two possible states. The link either contains one particle, in which case its state can be represented by a Boolean 1, or else it is empty, and its state can be represented by a Boolean 0. Each six-linked site thus has $2^6 = 64$ possible states. The state s of a site can be represented by a vector of six Boolean values, one for each link, i.e. $s=(s_0,s_1,s_2,s_3,s_4,s_5)$ with $s_i \in \{0;1\}$.

Evolution of the gas takes place in discrete timesteps. Each timestep encompasses one propagation phase, and one collision phase. During propagation, every particle on the lattice moves to the nearest neighbouring site in the direction of the link which it is currently occupying. All propagating particles have unit speed.

A velocity can always be represented as a scalar speed multiplied by a unit direction vector. Since the scalar speed is always one in the FHP case, the velocity of a particle can be considered equivalent to the direction vector of the link which it is occupying. The velocity of a particle on link i, and the direction vector of the i^{th} link of a site are thus both represented by the same symbol, c_i.

During the collision phase, all particles arriving at each of the lattice sites are collided. The collision outcome is governed by a set of collision rules which ensures that the lattice gas displays physical properties similar to those of a real gas. These rules are chosen so as to conserve the number of colliding particles, and the total momentum, at every collision site, and determine a post-collision state from the pre-collision state. During any collision, the particles involved may be considered to acquire their updated velocities instantaneously.

In the FCHC lattice used for three-dimensional lattice gas simulations, each site is joined to its neighbours by 24 links. As each of these links can, as in the case of the FHP model, contain either zero or one particles, the number of possible states which a site can assume is 2^{24}. The state s of a FCHC site is now represented by a vector of 24 Boolean values, one for each link direction (c_i), i.e. $s=(s_0,s_1,\ldots,s_{22},s_{23})$ with $s_i \in \{0;1\}$.

Every possible pre-collision state must still be catered for by the collision table. The number of possible collision table entries thus rises from 64 to almost 17 million when converting from a FHP to a FCHC model. This vast increase in the number of rules causes two major problems.

Firstly, none of the hardware available for local lattice gas simulations is capable of handling a lookup table of this size. Various options of reducing the size of the collision table are investigated in the later sections.

Secondly, the rule selection becomes an issue. For each possible input state, an output state must be chosen which obeys several collision rule requirements (which will be discussed in a later section), and maximises a quantity known as the Reynolds coefficient. In the FHP case, these selections could be made by hand. An automated procedure must however be written for the FCHC case.

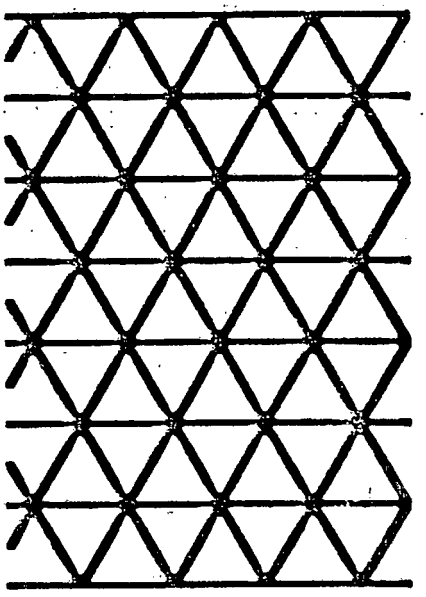

Figure 1: A section of a typical two-dimensional FHP lattice.

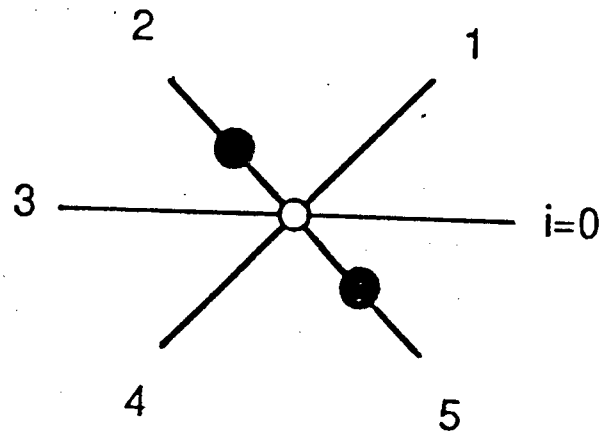

Figure 2: One site of an FHP lattice showing link direction numbering.

1.3 Lattice Symmetry Requirements: the FCHC

When we refer to accurately predicting the Navier-Stokes equations by lattice gas simulations, we mean ensuring that the large-scale and low speed behaviour of the lattice gas is similar to that in an incompressible real fluid. In order to achieve this, it is required that the non-linear term of the momentum equation be isotropic. This means that its components are invariant in any rotation of the co-ordinate axes.

In order for this constraint to be realised, the momentum flux tensors up to fourth order must be isotropic, as discussed in [12]. For a given integer r, $r \geq 1$, the r^{th} order tensor is of the form:

$$B_{\alpha_1 \alpha_2 \cdots \alpha_r} = \sum_i c_{i\alpha_1} c_{i\alpha_2} \cdots c_{i\alpha_r} \qquad (2)$$

Note that $\alpha_1, \alpha_2, \ldots$ or equivalently $\alpha, \beta, \ldots$ represent axis coordinates, and that $c_{i\alpha_k}$ represents the k^{th} component of the link direction c_i.

The tensors up to fourth order are thus given by:

$$B_\alpha = \sum_i c_{i\alpha} = 0 \tag{3.1}$$

$$B_{\alpha\beta} = \sum_i c_{i\alpha} c_{i\beta} = \frac{nc^2}{D}\delta_{\alpha\beta} \tag{3.2}$$

$$B_{\alpha\beta\gamma} = \sum_i c_{i\alpha} c_{i\beta} c_{i\gamma} = 0 \tag{3.3}$$

$$B_{\alpha\beta\gamma\delta} = \sum_i c_{i\alpha} c_{i\beta} c_{i\gamma} c_{i\delta} = \frac{nc^4}{D(D+2)}(\delta_{\alpha\beta}\delta_{\gamma\delta} + \delta_{\alpha\gamma}\delta_{\beta\delta} + \delta_{\alpha\delta}\delta_{\beta\gamma}) \tag{3.4}$$

with:

n	=	number of velocity components
c	=	velocity modulus i.e. link length ÷ propagation time
D	=	number of space dimensions
$\delta_{\alpha\beta}$	=	Kronecker delta symbol

From this it can be seen that the tensors depend only on the link directions, or velocities, c_i. Isotropy can thus be ensured by the correct form of the lattice. If the lattice has a large enough symmetry group G (which will be discussed in chapters 3 and 4), it is guaranteed that the prediction of the lattice gas will only differ from that of the Navier-Stokes equations in terms of a sufficiently high order as to be considered negligible. The reader is referred to [12] for further discussion of the symmetry requirements of the lattice.

If the requisite symmetries are insufficient, discrepancies appear in terms of a low enough order to cause unacceptable inaccuracies. In the two-dimensional HPP model mentioned later, the lattice symmetry cannot ensure the isotropy of the fourth order momentum flux tensor. In FHP-I, the fourth order tensors are ensured as isotropic by the triangularity of the lattice.

In extending the model to three dimensions, it was seen by Frisch *et. al.* [5] and others such as d'Humières *et. al.* and Wolfram [2,31] that there are no three-dimensional lattices with sufficient symmetry. The three-dimensional lattice with the highest degree of symmetry is the face centred cubic (FCC). This has twelve velocity directions. The fourth order tensors, however, include a term which disrupts the isotropy in the Navier-Stokes equations. The face centred hypercube (FCHC) in four dimensions displays the necessary symmetry characteristics, and its use in lattice gases was suggested in [2].

The nodes of the FCHC are the points having integer components (x_1,x_2,x_3,x_4) with the sum $x_1 + x_2 + x_3 + x_4$ even. Each node has 24 links to its nearest neighbours, 12 of which run from the centre of the hypercube (node) to the centre of each vertex, and 12 of which join the hypercube centre to each of the six faces, with two per face. One cell of this pseudo four-dimensional FCHC lattice is shown in figure 3. Note that the lines joining the node to the cell face centres all have non-zero components in the fourth dimension. Each link length is $\sqrt{2}$. The neighbouring nodes of the FCHC are the 24 points (±1,±1,0,0), (±1,0,±1,0), (±1,0,0,±1), (0,±1,±1,0), (0,±1,0,±1) and (0,0,±1,±1). To use this four-dimensional cell in three-dimensional simulations, the lattice consists of two three-dimensional layers of the four-dimensional FCHC with periodic boundary conditions in the fourth dimension, as discussed in [26].

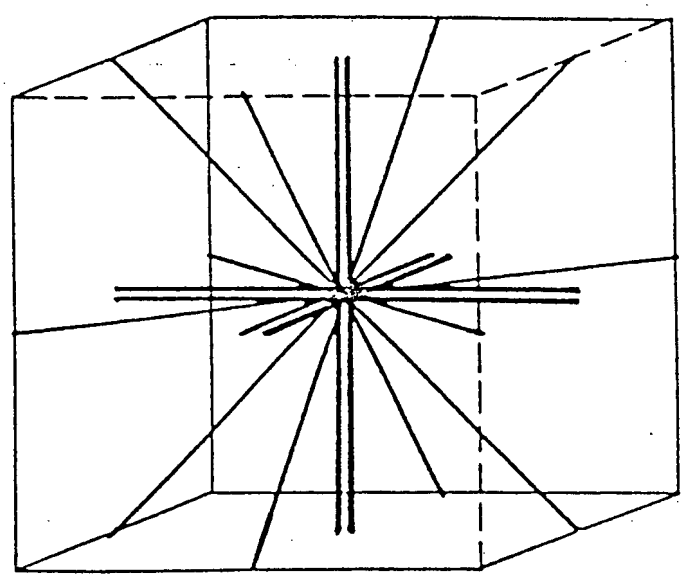

Figure 3: A single cell in the pseudo four-dimensional FCHC lattice.

1.4 Requirements for the Collision Rules, and Spurious Invariants.

Collision rules which give one explicit output state for each input state are said to be deterministic. In a case where there is a choice of output states for each input state, with a random or approximately random choice being made between these options, the rules are said to be stochastic. Storing more than one output state for each input state is more memory intensive than a deterministic case. Due to storage limitations, a deterministic table is usually opted for.

A list of the general conditions which should be satisfied by the collision rules is as follows.

1. The number of particles at each site (mass) should be conserved in all collisions.

2. The total momentum at each site should be conserved in every collision.

3. No other quantity should be conserved in all collisions, with the exception of energy conservation in models which are intended for thermodynamic simulations.

4. The exclusion principle should be maintained. This dictates that the post-collision velocities should be different from each other (implying at most one particle per link).

5. The collision rules must be invariant under each member of the set of isometries which preserves the set of velocities.

6. Collisions should satisfy semi-detailed balance (a very important issue, as discussed in section 1.8) if required.

Collision rules must thus *inter alia* conserve the number of particles, the momentum, and in some cases the energy, at any site where a collision takes place. No other quantities however should be conserved, as these undesirable conserved quantities, called "spurious invariants", may cause the model to behave in an unphysical manner. Attention should be paid to the possibility of such undesirable invariants appearing in a simulation, when constructing a collision table. The classes of invariants that can be encountered are summarised in [9].

It has been discovered by Kadanoff *et. al.* [17] that the FHP model has three extra conserved quantities called the staggered momentum densities. In [17], the origin of these invariants is illustrated by means of a very simple one-dimensional example: Consider a model in which all collision rules conserve momentum and particle numbers. If $g(x)$ is the linear momentum of the particles at site x, at a particular timestep t, then define, for that timestep:

G_e as total momentum of particles on even numbered sites, i.e. the sum of all $g(x)$ where x is even.

G_o as total momentum of particles on odd numbered sites, i.e. the sum of all $g(x)$ where x is odd.

Particles can only move from their present site to its nearest neighbour, and thus, in this one-dimensional case, G_e and G_o are exchanged at every timestep. This model therefore not only allows the conservation of particle number and momentum as imposed by the collision rules, but also conserves the quantity $(-1)^t(G_e-G_o)$ where t is the timestep number. In the FHP lattice gas, there are three such spurious invariants, one for each axis of symmetry of the hexagonal cell.

The collision rules affect the viscosity of the lattice gas. As discussed in section 1.7, it is desirable to minimise the viscosity in order that the Reynolds coefficient can be maximised. The rules should thus be chosen in such a way that the mean free path (average distance travelled by a particle before changing direction) is made as short as possible. To achieve this, the rules should cause the output state to be as different to the input state as possible.

1.5 The Boltzmann Approximation

The Boltzmann approximation assumes that there are no correlations between the particles entering a collision. This assumption means that the probability of a particle approaching a collision site along a specific link is independent of that for any other link. According to Dubrulle *et. al.* [4], the idea behind this approximation is that the propagation step immediately after the collision step removes most of the correlations which were generated in the collision.

If it is assumed that the Boltzmann approximation is valid, it is possible to predict the value of transport coefficients for a model with reasonable accuracy from the collision rules, by means of a calculation. The values calculated by means of the Boltzmann approximation are referred to as Boltzmann values, as opposed to measured or actual values. For the purposes of this work, it can be assumed that in the absence of specification as to whether a Boltzmann or measured value is being provided, the former holds true.

The reliable prediction of measured transport coefficients is only possible in cases where the collision rules satisfy semi-detailed balance (as discussed in section 1.8). In this case the measured values of the transport coefficients do not differ unreasonably from the values calculated using the Boltzmann approximation. In the comparison of variants of the FCHC lattice gases by Dubrulle *et. al.* in [4], it is seen that for all FCHC models which satisfy semi-detailed balance, the Boltzmann values were exact for equilibria, and also predicted the viscosities reasonably well. There is, however, a marked deterioration in the agreement between measured transport coefficients and those calculated using the Boltzmann approximation, when semi-detailed balance is violated.

Hénon [12] has shown that for models assuming semi-detailed balance, the Boltzmann viscosity is always positive. It has been reported in [7] (as referenced in [27]) that when semi-detailed balance is violated, together with the incorporation of rest particles (discussed in section 1.7), the Boltzmann viscosity can become negative. Dubrulle *et. al.* [4] report that no negative viscosities are in fact measured in models up to and including FCHC-8, even with the simultaneous addition of rest particles and the violation of semi-detailed balance. These discrepancies between Boltzmann and measured values are as a result of the lack of validity of the basic assumption when semi-detailed balance is violated.

An attempt at reducing the discrepancy between Boltzmann and measured values has been made by Hénon in [15] by means of implementing a simulation on several parallel lattices. A model with a measured negative shear viscosity (FCHC-9) is obtained by this means. At the beginning of a simulation, initial particle populations on several parallel lattices are calculated from a given density and velocity field, but using different random variations. All lattices display the same macroscopic properties. Random shuffling of corresponding lattice bits between the lattices is performed at the end of each time step. The microscopic properties of the lattices, which are different at the beginning of a simulation, remain different throughout the simulation due to the bit shuffling. The discrepancy between measured and Boltzmann values is seen to decrease slowly with increasing number of parallel lattices.

1.6 The Galilean Factor in Lattice Gases

The reader is referred to [31] for a complete derivation of the macrodynamical equations of a lattice gas. These differ from the standard incompressible Navier-Stokes equations:

$$\frac{\partial \overline{u}}{\partial t} + \overline{u} \cdot \nabla \overline{u} = -\nabla p + \nu \nabla^2 \overline{u} \tag{4.1}$$

$$\nabla \cdot \overline{u} = 0 \tag{4.2}$$

where

$\overline{u}$ is the velocity vector

p is the pressure

ν is the kinematic viscosity

by various terms. Some of these terms are negligible in the limit of low Mach and Knudsen numbers, providing that the symmetry group of the lattice is sufficient to ensure that the fourth-order momentum flux tensors are isotropic, as discussed in section 1.3. One extra term which is present in the lattice gas macrodynamical equations, is the Galilean factor *g(d)*. This term appears as a coefficient to the nonlinear term of the Navier-Stokes equations, $\overline{u} \cdot \nabla \overline{u}$. It is a function of the density of particles per oriented link *d*. The presence of the *g(d)* term is due to lack of Galilean invariance at the microscopic level. When speeds are low, the fluctuations in the density become irrelevant in all terms except the pressure term. The factor *g(d)* then becomes a constant and may be eradicated by rescaling the time variable. Thus in the asymptotic limit of low Mach number, the incompressible Navier-Stokes equations are recovered. The factor *g* can be obtained from the lattice topology, if semi-detailed balance is maintained, since the Fermi-Dirac particle distribution is then valid. For the FCHC lattice gas, *g* is given by:

$$g(d) = \frac{2}{3} \quad \frac{1-2d}{1-d} \tag{5}$$

1.7 The Reynolds Coefficient of a Lattice Gas

The Reynolds number of a flow can loosely be defined as the ratio of the inertial forces in the flow to the viscous forces. In lattice gases, it serves as a general measure of the complexity of a flow, since the size of the lattice needed to simulate a flow grows with Re^3, in three-dimensional simulations. In the case of lattice gases, the Reynolds number is given by

$$Re = R^* \; L \; V \tag{6}$$

Thus *Re* depends on a characteristic length *L*, which is the typical length of the experiment in lattice units, a characteristic velocity *V*, which is the velocity relative to the speed of sound, and the so called Reynolds coefficient R^*, which depends on the collision rules. The Reynolds coefficient is given by:

$$R^* = \frac{c_s \, g(d)}{\nu} \tag{7}$$

where c_s is the speed of sound, ν is the kinematic viscosity, d is the density per oriented link, and g is the Galilean factor, as discussed in section 1.6. It can be seen that the Reynolds coefficient is effectively the inverse of viscosity, but in dimensionless form. Note that throughout the present text, the term Reynolds coefficient implies Boltzmann Reynolds coefficient, unless it is specified that the actual measured Reynolds coefficient is being discussed. Section 1.5 gives fuller details as to the meaning of these terms.

A measure of the overall efficiency of a lattice gas model is determined by two factors, namely the computational efficiency and the implementation efficiency. In testing a collision table, the first can be determined by the rate at which post-collision states can be found. This factor is influenced by the method by which the collision outcomes are determined, and by the computing power of the hardware used. The second factor can be expressed by the Reynolds coefficient.

The way in which the Reynolds coefficient is used as a measure of the implementation efficiency of lattice gas models is as follows. The efficiency of a lattice gas flow simulation is proportional to the fourth power of R^* in a three-dimensional simulation with specified geometry, Mach and Reynolds numbers. The reason for this is that not only does the required lattice size grow as the cube of Re, as mentioned above, but the number of timesteps necessary for the simulation also grows linearly. Thus the computational time needed grows with Re^4. A high value of R^* is desirable so that flows with higher Reynolds numbers can be simulated, and so that the efficiency is improved. There are a number of ways in which the Reynolds coefficient can be increased, as discussed below.

1. The kinematic viscosity (referred to as viscosity in the present text) is inversely proportional to the Reynolds coefficient, and is dependent on the collision rules, as discussed in [12]. A standard way of increasing the Reynolds coefficient is by choosing collision rules in such a way that viscosity is minimised. A method of calculating the viscosity of a lattice gas, as well as a recipe for choosing collision rules which minimise the viscosity, is presented by Hénon in [12] and [13].

2. Another method of increasing the Reynolds coefficient is to increase the total number of possible collisions. The more collisions there are, the smaller the mean free path (MFP), and the lower the viscosity. More collisions can be obtained by adding stationary or rest particles. These are particles with zero velocity which reside at the site centres, and interact with the colliding particles. Certain pre-collision states cause the rest particles to move off towards a neighbouring site, while others leave the rest particles unaffected. The total number of interacting particles is therefore increased. The addition of rest particles also implies a greater number of possible collision table entries, and thus leaves more room for optimisation. This principle has been applied with success as can be seen from table 1 in section 2.5.

3. A third method for increasing the Reynolds coefficient is as follows. Violation of semi-detailed balance implies that the $g(d)$ factor becomes dependent on the collision rules, and the Reynolds coefficient can then be maximised by judicious selection of rules. Optimisation of the R^* value can thus be tackled by both minimising kinematic viscosity and maximising $g(d)$ simultaneously. Somers and Rem [27] have obtained a Boltzmann value of 40 for R^* by making use of these techniques, even without the use of rest particles. The measured Reynolds coefficient however will be far lower since Boltzmann values do not give a good prediction of measured values in the case of violation of semi-detailed balance. Due to the considerations mentioned in section 1.8, we prefer to maintain semi-detailed balance, and this method of increasing the Reynolds coefficient is thus not applicable to the local situation.

Notes regarding method 2.

Adding n rest particles increases the number of collision rules which need to be stored from 2^{24} to 2^{24+n}. The addition of one rest particle therefore doubles the required collision table size. On the other hand, the efficiency of a lattice gas increases with the power of four of any increase in Reynolds coefficient, as discussed above. Since the addition of rest particles benefits the Reynolds coefficient, and therefore greatly improves the efficiency of the lattice gas, this issue cannot be ignored. Addition of rest particles should not be considered for local implementation at this stage due to already severe memory constraints, but decisions regarding strategy should be made in such a way as to facilitate the addition of rest particles at a later stage.

It is important to note that reducing the number of possible collisions in any way lowers the Reynolds coefficient, in exactly the same way that increasing the number of collisions improves it. This becomes an issue when collision table splitting is discussed in a later chapter.

1.8 The Case for Semi-Detailed Balance

Semi-detailed balance is a generalisation of the micro-reversibility of the collisions. Maintaining semi-detailed balance in a collision table controls the correlations in the particle distribution: correlations not present in the pre-collision states will also not be present in the post-collision states. This control over the correlations is written as:

$$\sum_{s} A(s,s') = 1 \tag{8}$$

where $A(s,s')$ is the probability that a pre-collision state s will be transformed into a post-collision state s'.

As has been indicated in section 1.5, the validity of the Boltzmann approximation is dependent upon the maintenance of semi-detailed balance. In the absence of semi-detailed balance, the assumption of uncorrelated arrival of particles at a site breaks down. This has a few detrimental implications.

When semi-detailed balance is maintained, an analytical expression in the form of the Fermi-Dirac distribution is obtained for the mean populations, in terms of the mass and momentum conserved in the collisions. The Fermi-Dirac distribution is universal in that it is independent of the collision rules unless semi-detailed balance is violated. Expanding this distribution explicitly for low velocities is very useful in initialising the lattice, and for implementing general boundary conditions. Use of these expansions has already been made in the TransGas code.

In lattice gases violating semi-detailed balance, propagation can lead to severe decorrelation of the distribution of particles, causing the Fermi-Dirac distribution to be abandoned. The important uses of the equilibrium distribution mentioned above can then no longer be made. Maintenance of semi-detailed balance is thus essential for the derivation and application of the equilibrium distributions.

As has been previously noted in section 1.5, Dubrulle *et. al.* [4] show that for all FCHC models which violate semi-detailed balance, there is a marked deterioration in the agreement between the measured transport coefficients, and the values predicted using the Boltzmann approximation. Thus, when semi-detailed balance is violated, there is no easy way of reliably predicting measured values of the transport coefficients.

It should be also noted the large benefits (in terms of increasing the Reynolds coefficient) that can be obtained by violating semi-detailed balance as predicted by the Boltzmann approximation, are in actuality not so large. This is again as a result of the lack of agreement between measured and Boltzmann values when semi-detailed balance is violated. This fact tends to lessen the apparent appeal of the argument in favour of violating semi-detailed balance.

On the other hand, various authors eg. [3,12,27] have discussed the effective way in which the Reynolds coefficient can be further increased by violating semi-detailed balance. In a deterministic rule set, semi-detailed balance implies that no two different input states may have the same output state. This applies a limitation on the degree to which rule selection can be optimised in terms of reducing the viscosity. Violation of semi-detailed balance allows the collision rules to be individually optimised in terms of minimising viscosity. Under violation of semi-detailed balance, the output state for which viscosity is minimised can be selected, even if the same output state is the optimal choice for several different input states. The reduced viscosity resulting from this freedom, in turn increases the Reynolds coefficient.

The second means by which violation of semi-detailed balance increases the Reynolds coefficient is that it allows the Fermi-Dirac distribution to be abandoned as discussed above, and $g(d)$ becomes dependent on the collision rules. Collision rules can then be optimised to both minimise viscosity and maximise g in a single strategy (as discussed by Somers and Rem in [27]) yielding very effective optimisation of the Reynolds coefficient.

In conclusion it is seen that although violation of semi-detailed balance has advantages from the point of view of the Reynolds coefficient, drawbacks also exist. Violation of semi-detailed balance causes the universality of the equilibrium distribution function to be lost. Many uses are made of this in the current lattice gas, TransGas, such as in the specification of boundary conditions. Without semi-detailed balance, many currently used methods will have to be changed, causing undesirable delays. Semi-detailed balance also ensures that if all states have equal probabilities before collision, they will still be so after collision. Removing this property could be dangerous, as with certain states becoming eradicated, the behaviour of the automaton may change. It has been noted in section 1.5 and above that the predictions of transport coefficient values obtained with the Boltzmann approximation are not nearly as accurate when semi-detailed balance is violated. It is thus the opinion of the author that semi-detailed balance should be maintained.

2 GENERAL LITERATURE SURVEY

2.1 The Birth of Cellular Automata

Cellular Automata were invented by von Neumann in 1966 [30], and the reference is given although a copy of the paper could unfortunately not be obtained. A cellular automaton consists of a lattice or grid which is made up of sites, each joined to its neighbours by links. Each site can have a finite number of states. The states of all sites are simultaneously updated once per timestep, i.e. time is discrete. The updating is done in accordance with the rules of the automaton, which may be stochastic or deterministic. Wolfram recently "re-invented" cellular automata, and the reader is referred to [32] for general theory and applications. Lattice gases are a type of cellular automaton used for modelling fluid dynamics.

2.2 Early Lattice Gases

According to Frisch *et. al.* [6], the first lattice gas was that by Kadanoff and Swift in [33], for the modelling of sound waves. This model was continuous in time and made use of a master equation. The first lattice gas to use discrete time, position and velocity, together with a deterministic rule set, was that by Hardy *et. al.* This model and its characteristics are described in a series of three papers, culminating in [10]. In this model, known as the HPP lattice gas, a square lattice is used, but due to the lack of sufficient lattice symmetry, the lattice gas macroscopic equations differ unacceptably from the Navier-Stokes equations.

In 1986 Frisch, Hasslacher and Pomeau [5] demonstrated that the incompressible Navier-Stokes equations of fluid flow can be simulated in two dimensions using a lattice gas, as long as the form of the lattice used has suitable symmetries. Their model uses a triangular lattice in which each node is joined to its six nearest neighbours by links. The model, named after the initials of its authors, is known as the FHP-I model. The FHP model has been tested by various authors. In [17], Kadanoff, McNamara and Zanetti validate the FHP model using a two-dimensional version of laminar pipe flow as the test geometry. Verifications performed include studying the parabolic momentum density profile, the equation of state, and the logarithmic divergence in the viscosity. In the last case mentioned, full agreement was only obtained after extending the theory to allow for three extra invariants which were discovered by McNamara.

These invariants are the staggered momentum densities as previously discussed in section 1.4. The conclusion reached after these tests is that the lattice gas does simulate hydrodynamic equations, obeying equations which are potentially more complex than the two-dimensional Navier-Stokes equations. In [23] Rivet and Frisch determined macroscopic quantities such as the shear and bulk viscosities for FHP-type models called the simple hexagonal and the centre-hexagonal models. The latter includes extra collision laws to take the addition of rest particles into account.

This thesis concentrates on the construction of three-dimensional collision rule sets, and thus very little is presented regarding the vast amount of literature on two-dimensional lattice gases and the physical situations to which they have been applied. Lattice gas simulation results are beyond the scope of this thesis except in cases where they are of direct relevance to the testing of a collision rule model.

2.3 The Birth of 3-D Lattice Gases

d'Humières *et. al.* [2] introduced lattice gas models to simulate the three-dimensional Navier-Stokes equations. Since this time, the three-dimensional field has developed rapidly. In choosing a lattice for Navier-Stokes simulations in three dimensions, the symmetry as described in section 1.3 is of importance. It has been shown [2,5,31] that there are no sufficiently symmetrical three-dimensional Bravais lattices to accurately predict the Navier-Stokes equations. Various proposals were put forward concerning the solving of this problem:

1. FHP suggested that the lattice gas could be represented on two lattices, a cubic and a face centred cubic (FCC). The time evolution of the lattice gas would then be split onto these two lattices with the appropriate adjustment of the time ratios. In the opinion of d'Humières *et. al.*, this method would however be cumbersome and impractical.

2. d'Humières *et. al.* [2] investigated two different models for three-dimensional simulations. The first of these is a model utilising 19 bits per site, and three different particle speeds, of 0, 1 and $\sqrt{2}$. This model also conserves energy, and can be used for simulations which incorporate thermal effects. Isotropy is not built into the model, but the isotropy of the non-linear term of the Navier-Stokes equations can be guaranteed by selecting specific combinations of density and temperature. An advantage of this model is that it uses only 19 bits per site, but it has the drawback of being less robust at moderate hydrodynamic speeds due to the lack of built-in isotropy.

3. The second model investigated in [2] was a 24-bit FCHC model. This model occupies a four-dimensional lattice which has all the required symmetries for ensuring isotropy of the non-linear term of the Navier-Stokes equations. For the purposes of three-dimensional simulation, the lattice is made only one layer thick in the fourth dimension, giving it a three-dimensional spatial structure. Its major drawback is the fact that each site occupies 24 bits, making it expensive in memory. A major advantage is that there are many discrete velocities which result in more collisions, which in turn lead to higher Reynolds numbers. A lattice based on this geometry would appear to be the only viable option for use in three-dimensional simulations.

2.4 Isometric Collision Rules

In [11], Hénon discusses in some detail the method used to implement global rules on the FCHC lattice. These global rules are defined by a basic algorithm known as the isometric algorithm, which calculates the output state for a given input state, by using an appropriate isometry. An isometry can, for the purposes of this work, be described as a rotation about the origin and/or a mirror symmetry about a hyperplane. The basic method involves obtaining the post-collision state from the pre-collision state, by using an isometry which conserves mass and momentum, while minimising viscosity. This algorithm calculates the collision outcomes as needed during the lattice gas simulation, such that no storage of a collision table is necessary. This method provides a form of runtime rule generation, as opposed to collision rule lookup from a pre-stored table. The method maintains semi-detailed balance but yields a very low R^* value of approximately 2. This method has been implemented locally as a first-case three-dimensional collision rule generator. The algorithm consists of the following basic steps:

1. The four components of the momentum of a given input state are calculated.

2. Various isometries (rotations and mirror symmetries) are applied to the input state and to the momentum, until all momentum components are positive and in monotonically decreasing order.

3. All possible input states as modified in step 2 are divided into 12 classes. The derivation of the 12 classes is given in [11] and section 3.3.3, together with a list of "optimal isometries" for each class. These are optimal in the sense of minimising viscosity. A randomly chosen "optimal isometry" of the same class as the modified input state resulting from step 2 is then applied to this input state, and this constitutes the collision.

4. The same sequence of isometries as in step 2, but in reverse order, is applied to the post collision state. Thus an output state is generated from the input state by means of isometries.

Rivet [24] has implemented Hénon's isometric rules on a Cray-2 computer and has demonstrated by means of numerical experiments that the three-dimensional Navier-Stokes equations can be simulated with acceptable accuracy using a lattice gas. Results within a few percent of theoretically predicted values were obtained for kinematic shear viscosity using the FCHC. Similarly good results were seen when comparing lattice gas simulations of the Taylor-Green vortex with spectral simulations of the same. A lattice of 128^3 was used, yielding the ability to simulate flows with Reynolds numbers of about 100. It is seen that the collision phase, using the recipe outlined by Hénon in [11], is highly CPU-intensive, due to the repetition of a large number of arithmetic operations at each site.

This runtime rule generation algorithm is implemented and discussed in chapter 3, as a first attempt at obtaining three-dimensional collision rules.

2.5 Complete Collision Tables

Collision rules can be obtained by means of runtime rule generation (as discussed in the previous section), or by means of a pre-stored lookup table, called a collision table. The former method uses a global rule selection algorithm. In contrast, use of the latter method of obtaining collision rules involves carefully selecting an optimal output state for each input state, and this is known as detailed rule selection. As previously mentioned in section 1.2, a full collision table in the absence of rest particles would have 2^{24} entries. Such a collision table will be called a complete collision table for the purposes of this work.

In [13], Hénon uses detailed rules to define the collision outcomes in the form of a complete lookup table. The search for optimal rules is carried out in two stages. First, for each value of d (the average density per oriented link), rules are found which minimise the viscosity. Then, the d for which R^* is maximal is selected. This is shown in effect to be the optimisation problem of minimising

$$\mu_4 = \frac{1}{288} \sum_s \sum_{s'} A(s,s')\, d^{p-1} (1-d)^{n-p-1} \sum_\alpha \sum_\beta (Y_{\alpha\beta} + Y'_{\alpha\beta})^2 \tag{9}$$

with

n = number of links joining a site to its nearest neighbours.

p = number of particles present at the site.

$$Y_{\alpha\beta} = \sum_i s_i\, c_{i\alpha}\, c_{i\beta} - \frac{p}{2} \delta_{\alpha\beta}$$

subject to the constraints

$$0 \le A(s,s') \le 1 \text{ for all } s \text{ and } s' \tag{10.1}$$

$$\sum_{s'} A(s,s') = 1 \tag{10.2}$$

$$\sum_{s} A(s,s') = 1 \text{ (i.e. semi-detailed balance)} \tag{10.3}$$

Optimising the collision table thus involves pairing the states into input-output pairs in an optimal way. Since there are 17 million possible input states which require one-to-one mapping (in the case of maintenance of semi-detailed balance) onto 17 million output states in an optimal way, the size of this optimisation problem is formidable. The local hardware is incapable of tackling a problem of this magnitude due to storage limitations.

In order to reduce the complexity of the optimisation, Hénon divides all possible states into so-called packets. A packet contains all the states, and only the states, which contain p particles and have a momentum of $q = (q_\alpha, q_\beta, q_\gamma, q_\delta)$. In order that momentum and mass be conserved, it is necessary that a post-collision state comes from the same (p,q) packet as its pre-collision state. Optimising the collision table thus requires optimising within each packet, pairing only these states into input-output pairs. The single huge optimisation problem is broken up into 175225 smaller problems, as p and q can assume 25 and 7009 different values respectively. Since the factor $d^{p-1}(1-d)^{n-p-1}$ is constant within each packet, Hénon shows that the optimisation criterion is the separate minimisation, for each packet, of

$$\sum_{s}\sum_{s'} A(s,s')\,W(s,s') \tag{11.1}$$

with

$$W(s,s') = \sum_{\alpha}\sum_{\beta}(Y_{\alpha\beta}+Y'_{\alpha\beta})^2 \tag{11.2}$$

In order to further reduce the number of optimisation problems, and the size of the problems, Hénon uses a few more techniques which he calls "tricks". These involve use of the symmetries and duality (particle-hole swapping), and it is explained how packets with p equal to 11 or 12 can be optimised by simple computation. It is also obvious that empty packets, and those with only one element, do not need to be optimised. Due to the use of these techniques, the 175225 problems are reduced to only 154 different optimisation problems. The number of states N in each of these remaining problems ranges from two to 8952.

Hénon implemented two different techniques for performing the optimisations. An approximate optimisation was performed which made use of a "greedy algorithm". In this algorithm, the two states s and s' for which $W(s,s')$ is minimised, are matched. These two states are then removed from the packet, and the process is repeated until the packet is empty. It is this withdrawal of pairs from the packet which makes the greedy algorithm an approximate optimisation scheme. It is possible that an output state that has already been matched and withdrawn together with a specific input state, was also the optimal output state for another input state which has not yet been matched. Using the greedy scheme, the computation time for optimising a packet is proportional to N^2, and yielded a R^* of 7.13, with d equal to about one third.

An exact optimisation was also implemented by Hénon. This, in the case of deterministic rules, is in the form of a classical optimisation problem known as "the assignment problem" [1]. Using this scheme, computational time is proportional to N^3, and yet the best R^* value obtained (namely 7.57) is only 6% better than that obtained using the much faster greedy algorithm.

Rivet *et. al.* [25] investigated Hénon's FCHC model with detailed collision rules. The collision rules were obtained using the same method as outlined by Hénon in [13], making use of the greedy algorithm for optimisation of the rules. The result is a collision table of 2^{24} rules, optimised in such a way that the Reynolds coefficient is maximised. Using this collision table, simulations of flow past a circular plate at a Reynolds number of roughly 190 were performed on a Cray-2 computer. The size of the computational domain was 128 by 128 by 256 sites.

In the early validations of FCHC models with isometric collision rules [24], a maximum R^* value of 2.0 was achieved, and 0.5 million site updates could be performed per second. In contrast, use of the detailed collision rule table yields a value of 7.13 for R^*, and a capability of obtaining approximately 30 million site updates per second on the same computer. The reason for the improvement in speed is that in the latter case, collision outcomes are merely looked up in a collision table, whereas the isometric rules involve calculation of output states at each timestep, which involves a large number of arithmetic operations.

Dubrulle *et. al.* [4] have studied a class of FCHC lattice gas variants, with and without rest particles, and with or without maintenance of semi-detailed balance, with the aim of obtaining the highest possible Reynolds coefficient. They use a Hitchcock optimisation scheme for choosing collision rules which minimise the viscosity, in the cases where rest particles have been added. By violating semi-detailed balance, the Boltzmann predictions indicate that it would be possible to obtain negative viscosities, and hence almost infinite Reynolds coefficients.

The Boltzmann Reynolds coefficients appear in column 6 of table 1 below. True transport coefficients of some of the models have been measured after implementation, and reported in column 7 of table 1. It is found that a large discrepancy exists between the measured values and those predicted using the Boltzmann approximation, when semi-detailed balance is violated. It can be seen that the best measured value of R^* is 32.0, obtained by the addition of seven rest particles and simultaneous violation of semi-detailed balance. This result was obtained by utilising parallel lattices [15] as discussed in section 1.5.

A history of FCHC models is also summarised in [4]. Table 1, which has been adapted from this paper, and to which the results obtained in [15] have been added, is presented below for information. Note that the approximate and exact detailed optimisation models referred to above correspond to models FCHC-3 and FCHC-4 respectively, while the isometric algorithm discussed in section 2.4 corresponds to FCHC-1.

Table 1: Main characteristics of previously introduced FCHC models.

Name	Collision rules	Optimisation of ν	Number of rest particles	Semi-detailed balance	R_*^{max} Boltzmann	R_*^{max} Measured
FCHC-1	Algorithm	No	0	Yes	2.00	2.00
FCHC-2	Table	Approximate	0	Yes	6.44	-
FCHC-3	Table	Approximate	0	Yes	7.13	6.4
FCHC-4	Table	Exact	0	Yes	7.57	-
FCHC-5	Table	Exact	3	Yes	10.71	-
FCHC-6	Table	Exact	0	No	17.20	-
FCHC-7	Table	Exact	3	No	(∞)	7.9
FCHC-8	Table	Exact	3	No	99.7	13.5
FCHC-9	Table	Exact	7	No	-	32.0

It is desirable to maintain semi-detailed balance for the reasons discussed in section 1.8, and section 1.7 outlines the reason why addition of rest particles should be avoided at this stage. These considerations immediately preclude the immediate local implementation of models FCHC-5 through FCHC-9. Use of a collision table, as opposed to a collision algorithm, is however preferable for reasons of speed and flexibility. (The collision outcomes can be determined more quickly, and larger Reynolds number flows can be simulated.) Due to hardware limitations at the CFD facility, it is unfortunately impossible to store a complete lookup table of 2^{24} entries, and other strategies must be employed.

2.6 Split Collision Tables

Somers and Rem of Shell Laboratories in Amsterdam have a computer hardware system very similar to the local Transputer system, which is discussed in section 1.2. The system used by Somers and Rem, described in [26], has been utilised for three-dimensional lattice gas simulations, based on the FCHC lattice of d'Humières *et. al.*

In a Transputer system, memory is distributed among the processors, such that each Transputer only has direct access to its own very small local memory. In order to implement a lattice gas collision table on such a system, there are two options. The size of the collision table should be made small enough to fit into the local memory of each processor by some means, or the collision table should be stored in distributed form over all processors, and a strategy devised to allow each processor to access parts of the collision table which are not in its own local memory cache. The latter method increases the inter-processor communication time, and hence reduces the computational efficiency of the model.

In [27], Somers and Rem outline a strategy which enables them to store a collision table for a FCHC lattice gas in only 64 kilobytes of memory. This includes the use of three stationary particles in order to decrease the viscosity. Such a collision table would usually occupy over 100 Megabytes of memory if stored in full. These models satisfy semi-detailed balance and are optimised such as to minimise viscosity, and deliver R^* values ranging from 3.5 to 5.8.

The theory to calculate the viscosity of a lattice gas was invented by Hénon in [12], and extended to cater for rest particles in [13]. Somers and Rem present a different version of this theory in [27] for the case of the FCHC lattice gas with two rest particles, one of mass 2 and the other of mass 4. This is derived from the second order Boltzmann equation, and the first order expansion of the Fermi-Dirac distribution. The result of this derivation is:

$$\nu(\rho) = \frac{4 + Q_{\alpha\beta}}{24 - 6Q_{\alpha\beta}}; \alpha \neq \beta \tag{12.1}$$

where

$$Q_{\alpha\beta} = \sum_{ss'} W(s) A(s,s') X_{\alpha\beta}(s) X_{\alpha\beta}(s') \tag{12.2}$$

$\alpha, \beta \in \{x, y, z, t\}$ are the indices of the components of the direction vector

$$X_{\alpha\beta}(s) = \sum_{j} s_j c_{j\alpha} c_{j\beta} \tag{12.3}$$

$$W(s) = \frac{d^{M-1}(1-d)^{29-M}}{(d^2 + (1-d)^2)(d^4 + (1-d)^4)} \tag{12.4}$$

d = probability of a particle arriving along a link.

$$M = 4s_{25} + 2s_{24} + \sum_{j=0}^{23} s_j \tag{12.5}$$

s_{24} and s_{25}represent the presence or absence of the relevant stationary particles.

Using this result as a starting point, a strategy is presented in [27] for the development of an isotropic, non-deterministic collision table which satisfies semi-detailed balance, and has low memory requirements.

Using the invariance group of the FCHC lattice, G, (12.2) can be expressed as a function of C (the collision table) only. This can then be further simplified by taking the detailed structure of G into account. The derivation given by Somers and Rem is summarised here for clarification.

Define:

G as the invariance group of the FCHC lattice, and $p, q, r \in G$.
C as an anisotropic, deterministic collision table
s as the pre-collision state of a site, and s' is its post-collision state.

Consider the algorithm:

Select a random permutation $p \in G$

Let $s^p = p(s)$

Let $s'^p = C\{s^p\}$ (13)

Let $s' = p^{-1}s'^p$

Note that $C\{s^p\}$ is the stored output state of s^p in the collision table C, and that since p is an element of the invariance group G, p^{-1} is also an element of G.

This algorithm implies that

$$s' = p^{-1}(C\{p(s)\}) \tag{14}$$

Since G is a group, the collision algorithm (13) actually implements an isotropic collision operator.

For algorithm (13), it is seen that

$$A(s,s') = \frac{|\{p \in G : C\{p(s)\} = p(s')\}|}{|G|} \tag{15}$$

where $|G|$ is the number of elements in the invariance group G.

Substituting (15) and (12.4) into (12.2), and applying the equalities

$$\begin{aligned} &s' = C\{s\} \text{ for all } s \text{ and } s' \\ &W(q(s)) = W(s) \text{ for all } q \in G, \text{ and for all } s \text{ and } s' \\ &G^{-1} = G, \end{aligned} \tag{16}$$

the following equation is obtained:

$$|G| Q_{\alpha\beta} = \sum_{s} W(s) \sum_{q \in G} X_{\alpha\beta}(q(s)) X_{\alpha\beta}(q(c\{s\})) \tag{17}$$

Now, G can be decomposed into generating elements:

$$G = G_{perm} \circ G_{inv} \circ \tau_{\sigma} \tag{18}$$

where

G_{perm} and G_{inv} deal respectively with permutations and inversions of coordinates

τ_{σ} has 3 elements:

the identity
reflection in $x + y + z + t = 0$
inversion of x followed by reflection in $x + y + z + t = 0$

Since it can be shown that

$$X_{\alpha\beta}(q(s))X_{\alpha\beta}(q(C\{s\})) = X_{\alpha\beta}(s)X_{\alpha\beta}(C\{s\}) \text{ for all } q \in G_{inv}$$

$$\text{and } \sum_{\alpha\neq\beta} X_{\alpha\beta}(q(s))X_{\alpha\beta}(q(C\{s\})) = \sum_{\alpha\neq\beta} X_{\alpha\beta}(s)X_{\alpha\beta}(C\{s\}) \text{ for all } q \in G_{perm} \tag{19}$$

one can eliminate all transformations in subgroups G_{inv} and G_{perm} from the product $X_{\alpha\beta}(q(s))\ X_{\alpha\beta}(q(C\{s\}))$ in equation (17).

This leaves the equation

$$18Q = \sum_{s} W(s) \sum_{q\in\tau_\sigma} \sum_{\alpha<\beta} X_{\alpha\beta}(q(s))X_{\alpha\beta}(q(C\{s\})) \tag{20}$$

This equation can be used to generate a collision table C, such that Q is minimised. Apart from taking rest particles into account, it seems as if the resulting collision table would not be very different in size to that defined by Hénon [13]. Somers and Rem, however, make an observation which significantly reduces the storage required for such a collision table. This is as follows.

The set of 24 velocity directions in the FCHC lattice can be split into three subsets (E^0, E^1, and E^2) which each contain eight directions, in such a way that each term in (20) can be written as three terms, each as a function of velocity directions from one subset only.

The three subsets are as follows:

E^0:	E^1:	E^2:
(-1, -1, 0, 0)	(-1, 0, -1, 0)	(-1, 0, 0, -1)
(+1, +1, 0, 0)	(+1, 0, +1, 0)	(+1, 0, 0, +1)
(-1, +1, 0, 0)	(-1, 0, +1, 0)	(-1, 0, 0, +1)
(+1, -1, 0, 0)	(+1, 0, -1, 0)	(+1, 0, 0, -1)
(0, 0, -1, -1)	(0, -1, 0, -1)	(0, -1, -1, 0)
(0, 0, +1, +1)	(0, +1, 0, +1)	(0, +1, +1, 0)
(0, 0, -1, +1)	(0, -1, 0, +1)	(0, -1, +1, 0)
(0, 0, +1, -1)	(0, +1, 0, -1)	(0, +1, -1, 0)

(21)

Any pre-collision state s then has its bit-sequence ordered such that it can be written as a concatenation $s = s^0 + s^1 + s^2 + s^S$. Each of s^0, s^1 and s^2 are 8-bit sequences which show the presence of particles with velocities from E^0, E^1, and E^2 respectively. The term s^S is a bit-sequence showing the presence of the stationary particles. The outcome of the collision can thus be written as the concatenation $s' = C^0\{s\} + C^1\{s\} + C^2\{s\} + C^S\{s\}$. Equation (20) can thus be rewritten in terms of the short bit-sequences as

$$18Q = \sum_s W(s) \sum_{q \in \tau_\sigma} \left\{ \begin{array}{l} X_{xy}(q(s^0))\, X_{xy}(q(C^0\{s\})) \\ +X_{zt}(q(s^0))\, X_{zt}(q(C^0\{s\})) \\ +X_{xz}(q(s^1))\, X_{xz}(q(C^1\{s\})) \\ +X_{yt}(q(s^1))\, X_{yt}(q(C^1\{s\})) \\ +X_{xt}(q(s^2))\, X_{xt}(q(C^2\{s\})) \\ +X_{yz}(q(s^2))\, X_{yz}(q(C^2\{s\})) \end{array} \right\} \tag{22}$$

Leading on from this, Somers and Rem suggest that the 26-bit collision table could be broken up into three 10-bit collision tables. The 26-bit collisions could then be performed as a sequence of three 10-bit collisions, which are only correlated through the stationary particles. None of the transformations in the group G mixes the elements of E^0, E^1, and E^2. Breaking the collision up into three different parts thus causes the momentum to be conserved with respect to each velocity subset individually. Mass can only be exchanged via the rest particles.

The scope for optimisation in the three small tables is very limited in comparison to that for a single large table, and so Somers and Rem expect that the Reynolds coefficient achieved using this method would not be acceptable. To increase R^*, they suggest that a two-way split be made instead of a three-way split. This implies two larger collision tables which will mean more scope for optimisation. Somers and Rem have also investigated static splits and dynamic splits of collision tables. In the former it is decided beforehand which way the collision is to be split, while the latter is performed by deciding how the split should be made, for each collision, at runtime. A static 16/10 split has been implemented in [27] which combines E^0 and E^1 to form the 16-bit table. Each table is optimised independently by splitting (22) into two terms, giving

$$18Q = Q^{16} + Q^{10}$$

$$Q^{16} = \sum_{s \in \{0,1\}^{16}} W^{16}(s) \sum_{q \in \tau_\sigma} \begin{Bmatrix} X_{xy}(q(s))\, X_{xy}(q(C^{16}\{s\})) \\ +X_{zt}(q(s))\, X_{zt}(q(C^{16}\{s\})) \\ +X_{xz}(q(s))\, X_{xz}(q(C^{16}\{s\})) \\ +X_{yt}(q(s))\, X_{yt}(q(C^{16}\{s\})) \end{Bmatrix}$$

$$Q^{10} = \sum_{s \in \{0,1\}^{10}} W^{10}(s) \sum_{q \in \tau_\sigma} \begin{Bmatrix} X_{xt}(q(s))\, X_{xt}(q(C^{10}\{s\})) \\ +X_{yz}(q(s))\, X_{yz}(q(C^{10}\{s\})) \end{Bmatrix} \qquad (23)$$

with:

$$W^{16}(s) = \frac{d^{M-1}(1-d)^{15-M}}{(d^2+(1-d)^2)(d^4+(1-d)^4)},\ M = \sum_{i=0}^{15} s_i$$

$$W^{10}(s) = \frac{d^{M-1}(1-d)^{13-M}}{(d^2+(1-d)^2)(d^4+(1-d)^4)},\ M = 4s_9 + 2s_8 + \sum_{i=0}^{7} s_i$$

where s_9 and s_8 indicate the presence or absence of stationary particles.

They also use the method of splitting states into packets as did Hénon in [13]. The form of the optimisation of each part in (23) is known as the optimal marriage problem, with a processing time proportional to the number of states in the given packet.

In a private communication [28], Somers and Rem suggest a method of carrying out a dynamic 16/8 split on an FCHC model without rest particles. They propose using Hénon's formula (9) as a starting point, but in an alternative formulation:

$$\mu_4 = \frac{1}{288} \sum_s \sum_{s'} A(s,s') d^{p-1} (1-d)^{23-p} \sum_\alpha \sum_\beta \left\{ \sum_i (s_i + s_i')\, c_{i\alpha}\, c_{i\beta} - p\,\delta_{\alpha\beta} \right\}^2 \qquad (24)$$

The term splitting would be carried out on this formula. As there are no rest particles, it is not understood how the exchange of mass will occur. It is important to note that spurious invariants will be introduced as a result of breaking collisions into independent parts, and methods of dealing with this problem would have to be investigated if this approach were to be adopted.

The main problem with this method, however, is the degradation of the Reynolds coefficient that results from the fact that splitting of the collision table into smaller parts allows only some of the total number of possible collisions to occur. This seriously lowers the efficiency of the lattice gas, as outlined in section 1.7.

2.7 Reduced Collision Tables

For the purposes of this work, the term "reduced collision table" refers to a table in which only some of the collision rules are stored. The remaining rules are calculated from the stored rules at runtime, as required.

In [22] Somers and Rem outline (in very general terms) a single-phase model which runs on a Transputer network. A two-phase model and some preliminary results obtained with it are also presented.

In the one-phase model in [22], use is made of the large symmetry group (1152 elements) of the FCHC lattice, in order to partition the 2^{24} possible states into a group of equivalence classes. Only one collision rule is stored per class. The collision rules required for any given state are then determined from the stored collision rule of the same class by a relation through an isometry. A greatly reduced collision table is thus stored on the limited memory of the transputer network.

By allocating the same transition probability between any two states having equal mass and momentum, it is sufficient to know the mass and momentum of the input state to determine the output state of any collision. Making use of the subgroups and the duality, a collision table of this type can be reduced to 397 entries. Each entry has at most 20 different output states which are applied stochastically according to prescribed probabilities. Storage of this table takes up only 40 kiloBytes.

This method is feasible on local hardware, but Somers and Rem report that the measured viscosity obtained with this model is roughly three times higher than the reported optimal Boltzmann value for FCHC models. This in turn implies a seriously degraded Reynolds coefficient. It should be noted that semi-detailed balance is also lost in this reduction process, which would normally lead to the problems outlined in section 1.8. Somers and Rem report that despite this loss of semi-detailed balance, experiments show that the predicted viscosities remain valid.

Hénon, in [15], discusses a method of reducing the storage requirements of the collision table without adversely affecting the Reynolds coefficient. The set of all possible states is divided into subsets called species. Two states are said to be of the same species if the one can be determined from the other by means of applying an isometry. (The term isometry is discussed in detail in the next chapter.) The method then involves storing only one element, called the deputy, for each species of states. Let M be that isometry which transforms the input state into its deputy. The output state of a collision is calculated by applying the inverse of M to the output state stored for the deputy of the input state. In this way, the total storage requirements are greatly reduced by making use of the symmetry of the problem. This method is implemented, and since a full discussion is given in chapter 4, no further details are presented here.

Somers and Rem [29] have invented a strategy similar to the one by Hénon in [15]. A copy of this Somers and Rem paper could unfortunately not be obtained by the author prior to the printing of the present text, but the information presented here was obtained from [15]. A brief description of the method is given below.

As in [15], the determination of a collision outcome consists of the following steps. An isometry M is found which transforms an input state s to its representative in a reduced table, $\hat{s}$. The output state of the representative, $\hat{s}'$, is looked up in the reduced table, and the inverse isometry M^{-1} is applied to it to obtain the output state s' of the original input state s. The choice of representatives and the method of obtaining M is however quite different to that in [15].

The method of obtaining the representatives for which output states appear in the reduced table is as follows. Each set of opposite velocities in an input state is represented by a Boolean bit. The bit is set to 0 if both velocities are present or if both are absent. The bit is set to 1 if only one of the pair is present. The set of these 12 bits representing a state form a 12 bit integer s_d, which is normalised in some way by the application of isometries. Applying these normalisation isometries to the s_ds of all possible input states reduces the number of different states to only 106496. This gives the reduced collision table.

For a number of reasons, the rate at which collision outcomes can be determined by Somers and Rem's method is better than that of Hénon. Firstly, because the address of the reduced table entry can be calculated from the input state directly, only output states need to be stored, as in a complete collision table. In [15], both input and output states need to be stored, and the table is accessed by binary search. In the Somers and Rem scheme, direct addressing of the table is possible, which reduces the required lookup time. Secondly, the reducing isometries which depend on s_d can be read from a table, and thirdly, the entire collision can be programmed as a sequence of table lookups by the addition of a few extra tables.

The Somers and Rem scheme obtains output states approximately four times faster than the scheme in [15]. The major drawback of this method, however, is that the resulting reduced table is larger. This leaves less scope for the addition of rest particles (which is envisaged for the future), which would improve the Reynolds coefficient.

Since overall efficiency of a lattice gas is benefited to a greater extent by increased Reynolds coefficient than by increased rate of outcome determination (as shown in section 1.7), it is recommended that the method in [15] be implemented rather than that in [29].

2.8 Recommendations Based on the Literature Survey

In constructing a collision table for local three-dimensional lattice gas simulations, a number of issues should be considered. These considerations arise from the experience of other authors in the field, as ascertained by surveying the literature. These are as follows:

- In order to satisfy requirements of lattice symmetry, the only viable lattice geometry that can be used appears to be the face centred hypercube (FCHC).

- All collision rules must be chosen so that the requirements as listed in section 1.4 are satisfied.

- A deterministic collision table should be generated, since storing only one output state for each possible input state uses less computer memory than would a stochastic table.

- Semi-detailed balance should be maintained. Although the violation of semi-detailed balance allows greater scope for optimisation of the collision rules, and thus allows the Reynolds coefficient to be greatly increased, certain problems arise. Control over correlations of particle states is lost. The Fermi-Dirac equilibrium particle distribution becomes invalid, with the result that boundary condition specification and initialisation of the lattice, which are greatly facilitated by this distribution, become awkward. Loss of the ability to accurately predict transport coefficients within the framework of the Boltzmann approximation results, as reported in [4].

- The objective of maximising the Reynolds coefficient of the final model must be borne in mind at all times. Strategies which have the effect of increasing R^* should be employed whenever they are within the limitations imposed by the available hardware. Methods which degrade the Reynolds coefficient must be avoided.

- The addition of rest particles should not be considered at this stage, since the already severe storage problems would be exacerbated by doing so. The advantage of adding rest particles is that the total number of interacting particles increases, which lowers the mean free path and thus reduces viscosity. Also, since more particles are involved, the size of the collision table is increased, leaving more room for optimisation. These factors lead to a higher attainable Reynolds coefficient, which in turn leads to a considerably improved efficiency. Once the storage problems can be reduced (either due to successful implementation of one of the memory saving methods outlined in the previous sections, or by the procurement of better computing facilities), the addition of rest particles should be implemented. This will not form part of this thesis, but will be a future step. Strategies employed at this stage should however bear the future addition of rest particles in mind.

- Storage of the full collision table is impossible on local hardware, but an algorithmic collision rule (as opposed to a collision table) could be implemented. It is recommended that an isometric FCHC lattice gas model following Hénon [11] should be constructed. Expectations of efficiency are not high, as it has been reported that the arithmetic operations at each timestep are time-consuming, and the Reynolds coefficient attained with this model is low. The implementation of the global rules is however possible on the current hardware system, and it is a good first step.

- The method of splitting the collision table into two parts as suggested by Somers and Rem [28] would definitely seem to be a good solution, since Somers and Rem have a hardware system similar to one of those at the CFD facility. The major drawback of the Somers and Rem method is the issue of degrading the Reynolds coefficient. Splitting the collision table allows only a fraction of the possible collisions to occur. This reduced collision set causes an increase in the kinematic viscosity of the lattice gas, and hence a decrease in the Reynolds coefficient. This is obviously undesirable.

- It is therefore suggested that a better solution is to adapt the methods of reduced collision table storage, as discussed in section 2.7, to the available hardware. Although Hénon's scheme [15] is slower than the Somers and Rem method [29], it leaves more scope for the future addition of rest particles. There is thus a choice between a higher attainable Reynolds coefficient with the first method, or a greater rate of collision outcome determination with the second. Since the overall efficiency of a lattice gas model increases with the fourth power of the increase in Reynolds coefficient, and only linearly with increasing outcome determination rate, it is felt that the method in [15] is more suitable. This method of reducing storage requirements by making use of the symmetry of the problem, without lowering the Reynolds coefficient, is discussed and implemented in chapter 4.

3 IMPLEMENTATION OF THE ISOMETRIC COLLISION ALGORITHM

3.1 Motivation for Implementing the Isometric Collision Algorithm.

As discussed in the previous chapter, the first step towards acquiring a three-dimensional lattice gas capability is the generation of a three-dimensional collision rule set. In order to accurately simulate the Navier-Stokes equations, certain conditions of lattice symmetry have to be satisfied, and for this reason, the four-dimensional FCHC lattice is used for three-dimensional simulations. In moving from a two to a four-dimensional geometry, the number of possible states that a site can assume, and thus also the number of possible collision table entries, rises by a factor of roughly a quarter of a million. The hardware available for local lattice gas simulations is not capable of storing a lookup table of this size, and thus other methods have to be investigated.

One method of approach is not to store the collision rules at all, but instead to generate them as they are required, during runtime. An algorithm for performing this task has been invented by Hénon [11], and successfully implemented and tested by various authors such as Rivet [24]. Although Rivet's findings indicate that this algorithm is too inefficient (in terms of both update speed and Reynolds coefficient) for practical simulation purposes, it provides a good understanding of the symmetry of the problem, and supplies a benchmark against which other collision algorithms or tables can be compared. The algorithm provides a method of computing collision outcomes for three-dimensional simulations which is within the capabilities of the local hardware.

It is for the above reasons that one of the recommendations of the previous chapter is that an isometric FCHC lattice gas model, following Hénon's method, be constructed. The present chapter is in fulfilment of this recommendation. A discussion of the theory relevant to Hénon's method is provided in section 3.3, and the implementation of this theory is outlined. The preliminary findings have been presented by the author in [18]. Subsequent improvements to the coding have been made which have resulted in a significant speedup of the rate at which collision outcomes can be generated. These modifications, and the improved results, are reported in [19]. Even so, as reported by Rivet [25], it is indeed found that the algorithm is far too inefficient in terms of computing speed and Reynolds coefficient, and other methods must be implemented.

3.2 General Background to the Method

The first proposed collision rule set for use with the FCHC lattice was the isometric collision algorithm, suggested by Hénon [11]. The isometric rules together with the FCHC lattice formed the first FCHC isometric model, and provided the first method of simulating three-dimensional flows by lattice gas methods.

In [11] Hénon discusses in some detail the method used to implement these isometric rules (which are a specific instance of global rules) on the FCHC lattice. Global rules are defined by a basic algorithm which calculates the output state for a given input state, by using an appropriate isometry. An isometry can, in general terms, be described as a rotation about the origin plus an optional mirror symmetry, but is more fully discussed in later sections. The basic method involves obtaining the post-collision velocity set from the pre-collision velocity set, by using an isometry which conserves mass and momentum, while minimising viscosity.

Rivet has implemented these rules on a Cray-2 computer in [24], and has concluded that the three-dimensional Navier-Stokes equations can be reasonably simulated using this method. In these simulations, a lattice consisting of 128 cells in each cartesian direction is used. Flows with Reynolds numbers of about 100 have been simulated.

Rivet also notes the inefficiency of the scheme. The collision phase, using the recipe outlined by Hénon, is highly CPU-intensive, due to the repetition of a large number of arithmetic operations at each site. The number of collision updates per second for the implementation in [24] is reported in a later paper [25] as being half a million on the Cray-2, and the Reynolds coefficient obtained is of the order of 2.0. Even these "slow" results cannot be matched using the available hardware, but the basic algorithm that has been implemented (collision algorithm only) is essentially the same as that outlined by Hénon in [11]. The implementation has been geared specifically to the computer equipment available, and provides South Africa's first three-dimensional collision rule set.

3.3 Theory of the Isometric Method

3.3.1 Isometries and Isometric Collision Rules

Consider a set of particles present at a FCHC lattice site. Each particle present has a unique four-dimensional velocity, and we can consider the set of particles as a set of velocities. There are 24 different possible velocities, namely:

$$(\pm1,\pm1,0,0),(\pm1,0,\pm1,0),(\pm1,0,0,\pm1),(0,\pm1,\pm1,0),(0,\pm1,0,\pm1) \text{ and } (0,0,\pm1\pm1). \quad (25)$$

These velocities are arbitrarily numbered c_i, i=1,..,24, with $c_i = (c_{i1},c_{i2},c_{i3},c_{i4})$. Once the numbering of the velocities has been decided on, it must remain consistent throughout.

The input state of the site is represented by an array $s = \{s_1,s_2,...,s_{24}\}$ with s_i being a Boolean value 0 or 1. The value of s_i indicates the presence of a particle with velocity c_i if it is 1, or the absence of such a particle if it is 0. The output state of the site is denoted by a similarly Boolean-valued array $s' = \{s'_1,s'_2,...,s'_{24}\}$.

Let us define the state matrix as the 4 by 24 matrix constructed as follows: Write the 24 possible velocity vectors as 24 columns of a matrix. Because each velocity vector has four components, we get a 4 by 24 matrix. We now multiply every element in column i by the value (0 or 1) of s_i for all i = 1,..,24. This gives us our 4 by 24 element state matrix.

Consider the particles present at an FCHC lattice site to be a set of velocities. All velocities are distinct, since a maximum of one particle can reside on a link, and each link has a unique direction vector. An isometry of this set of velocities is essentially a rotation of the set about the origin of the site, plus an optional mirror symmetry. An isometry thus transforms the set of velocity vectors. Since the velocities are four-dimensional, an isometry which transforms the velocity set can be represented by a 4 by 4 matrix.

Let G be the set of all isometries which preserve the set of velocities. The sense in which the velocity set is preserved by an element of G is as follows. The isometry $M \in G$ is to be applied to a set of FCHC velocities, each of which is a distinct element of (25). After the operation of $M \in G$ on this FCHC velocity set, the resulting vectors are all still distinct elements of (25), ie the transformed set is still a valid set of FCHC velocities.

In terms of implementation detail, consider matrix multiplying a 4 by 4 isometry matrix M with a 4 by 24 state matrix. This process gives rise to a new 4 by 24 matrix, the transformed state matrix. If M is an element of G, then each non-zero column of the transformed state matrix will still be an element of (25), and all non-zero columns will remain distinct from one another. An isometry within G thus brings about a transformation of the set of velocity vectors which is such that all resulting vectors are still possible velocities within the FCHC. In this way, the FCHC velocity set is preserved.

As shown in [11], the set G is a mathematical group, and is known as the symmetry group of the FCHC. Let the elements of G be denoted by $M=(a_{ij}), i,j \in [1,..,4]$. We denote the isometry and the matrix by the same symbol M. If c is a velocity, then we denote the image of the velocity under the isometry by Mc.

According to d'Humières *et. al.* [2], the velocity set is preserved by permutations of coordinates, reversal of one or several coordinates, and also under the isometry $(x_1,x_2,x_3,x_4) \rightarrow (x_1-\Sigma, x_2-\Sigma, x_3-\Sigma, x_4-\Sigma)$ where Σ is given by $\Sigma=\frac{x_1+x_2+x_3+x_4}{2}$.

Some examples of isometries in G are:

(1) Changing the sign of one coordinate α, denoted by S_α. Note that this isometry is a reflection about the hyperplane $x_\alpha=0$ in four-dimensional space.

If we wish to change the sign of the second coordinate, then we would use

$$S_2=\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & -1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

which is a reflection about the hyperplane $x_2=0$.

(2) Swapping or permuting two coordinates α and β, $(\alpha \neq \beta)$, denoted by $P_{\alpha\beta}$. This isometry is a reflection about the hyperplane $x_\alpha=x_\beta$ in four-dimensional space.

If we wish to swap the second and third coordinates, then we would use

$$P_{23}=\begin{pmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{pmatrix}$$

which is a reflection about the hyperplane $x_2=x_3$.

(3) The isometry $(x_1,x_2,x_3,x_4)\to(x_1-\Sigma,x_2-\Sigma,x_3-\Sigma,x_4-\Sigma)$ can be written in matrix form as

$$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}\to\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}-\frac{1}{2}\begin{pmatrix}1&1&1&1\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{pmatrix}=\begin{pmatrix}\frac{1}{2}&-\frac{1}{2}&-\frac{1}{2}&-\frac{1}{2}\\-\frac{1}{2}&\frac{1}{2}&-\frac{1}{2}&-\frac{1}{2}\\-\frac{1}{2}&-\frac{1}{2}&\frac{1}{2}&-\frac{1}{2}\\-\frac{1}{2}&-\frac{1}{2}&-\frac{1}{2}&\frac{1}{2}\end{pmatrix}=\frac{1}{2}\begin{pmatrix}1&-1&-1&-1\\-1&1&-1&-1\\-1&-1&1&-1\\-1&-1&-1&1\end{pmatrix}$$

This isometry we will denote simply as Σ, where

$$\Sigma=\frac{1}{2}\begin{pmatrix}1&-1&-1&-1\\-1&1&-1&-1\\-1&-1&1&-1\\-1&-1&-1&1\end{pmatrix}$$

By combining $S_\alpha s$ with $P_{\alpha\beta}s$ and Σ, we get two isometries which turn out to be convenient to use. (Note that since G is a group, any combination of elements of G gives an isometry which is once again an element of G.)

We call these isometries

(4) $$\Sigma_1=P_{14}P_{23}S_1S_2S_3S_4\Sigma=\frac{1}{2}\begin{pmatrix}1&1&1&-1\\1&1&-1&1\\1&-1&1&1\\-1&1&1&1\end{pmatrix}$$

which is a reflection about the hyperplane $x_1+x_4=x_2+x_3$ in four-dimensional space,

and

$$(5)\qquad \Sigma_2=P_{14}P_{23}S_2S_3S_4\Sigma S_4=\frac{1}{2}\begin{pmatrix}1&1&1&1\\1&1&-1&-1\\1&-1&1&-1\\1&-1&-1&1\end{pmatrix}$$

which is a reflection about the hyperplane $x_1=x_2+x_3+x_4$ in four-dimensional space.

These isometries can be used to generate all elements of G (which has 1152 elements). According to Hénon in [7], the set $\{S_1P_{12}P_{13}P_{14}\Sigma_1\}$ is a minimal generating set for G.

The more convenient way of generating all elements M of G is to use the redundant set of 12 elements $\{S_1S_2S_3S_4P_{12}P_{13}P_{14}P_{23}P_{24}P_{34}\Sigma_1\Sigma_2\}$. This set is convenient because every isometry $M \in G$ has exactly one representation of the form

$$M=\begin{pmatrix}I\\S_4\end{pmatrix}\begin{pmatrix}I\\S_3\end{pmatrix}\begin{pmatrix}I\\S_2\end{pmatrix}\begin{pmatrix}I\\S_1\end{pmatrix}\begin{pmatrix}I\\P_{34}\end{pmatrix}\begin{pmatrix}I\\P_{23}\\P_{24}\end{pmatrix}\begin{pmatrix}I\\P_{12}\\P_{13}\\P_{14}\end{pmatrix}\begin{pmatrix}I\\\Sigma_1\\\Sigma_2\end{pmatrix} \qquad (26)$$

where one factor is chosen from within each set of parentheses, and I is the identity. This form is known as the normal form of the isometries.

In order to reduce the number of possible collisions still allowed by the general requirements for collision rules, we make use of isometries to impose three more restrictions on the collisions. These are called the isometric collision rules:

Rule 1: Every collision is an isometry, i.e. $s'=Ms$ where $M \in G$, i.e. the output state is deduced from the input state by means of applying an isometry $M \in G$.

Rule 2: The isometry used in the collision depends on the momentum only. This implies that isometries must be chosen so that momentum is preserved. This rule is imposed in order to satisfy the second general collision rule requirement, as specified in section 1.4, since Rule 1 does not cater for this condition. Note that all isometries automatically conserve mass.

Rule 3: The isometry chosen for the collision is randomly selected among all optimal isometries. The definition and identification of these optimal isometries is based on minimising the viscosity, and is discussed in section 3.3.2.

Notes:

- As discussed in [11], the number of possible values that the momentum can assume is 7009. By taking advantage of the symmetries, the number can be decreased to 37, divided into 12 classes. This will be discussed in section 3.3.3.

- Under these rules, the number of permitted isometries for any input state is always greater or equal to two, i.e. for any input state, there is always at least one isometry which preserves the momentum, and gives an output state which is different to the input state. One of the permitted isometries is always the identity.

3.3.2 Implications of Minimising the Viscosity

One criterion of rule selection is that the viscosity should be minimised, in order to increase the Reynolds coefficient of the lattice gas, and thus its efficiency. The viscosity depends upon the isometry M that is used.

Hénon [12] showed that the kinematic viscosity for a lattice gas can be written as:

$$\nu = \frac{\tau c^2}{2(D+2)} \frac{\mu_4}{1-\mu_4} \tag{27}$$

where μ_4 is the viscosity index; it is a dimensionless value $\in \{0;1\}$:

$$\mu_4 = 1 - \frac{D}{2(D-1)n} \sum_s \sum_{s'} A(s,s') d^{p-1} (1-d)^{n-p-1} \sum_i \sum_j s_i (s_j - s_j') \cos^2 \theta_{ij} \tag{28}$$

where τ = timestep length

c = velocity modulus

D = number of space dimensions

n = number of possible velocities

$A(s,s')$ = probability of transition from an input state s to an output state s'

d = density per oriented link

p = $\sum_i s_i$ = number of particles at a site

θ_{ij} = angle between velocity directions i and j.

For FCHC, $\tau = 1$

$c = \sqrt{2}$

$D = 4$

$n = 24.$

So, for FCHC,

$$\nu = \frac{1}{6}\frac{\mu_4}{1-\mu_4} \tag{29}$$

with

$$\mu_4 = 1 - \frac{1}{36}\sum_s\sum_{s'} A(s,s')\, d^{p-1}(1-d)^{n-p-1}\sum_i\sum_j s_i(s_j - s_j')\cos^2\theta_{ij} \tag{30}$$

We estimate the value of $\cos^2\theta_{ij}$ for a specific isometry M, by using average values.

In [12] it is seen that, for a given i,

$$\sum_j \cos^2\theta_{ij} = \frac{n}{D}$$

For an arbitrarily and independently chosen i and j, it follows that

$$<\cos^2\theta_{ij}> = \frac{1}{D} \tag{31}$$

Note that $< >$ denotes an average value over site states.

Using the isometry M, consider the case where $c_j = Mc_i$. In this case, i and j are not independent, and $<\cos^2\theta_{ij}>$ will depend on the isometry M: eg. If M is the identity, then $c_j = c_i$

so $$<\cos^2\theta_{ij}> = <\cos^2 0> = 1$$

Let the value of $<\cos^2\theta_{ij}>$ be w for isometry M, with $c_j = Mc_i$

then $$w = <\cos^2\theta_{ij}>_{c_j = Mc_i} \tag{32}$$

But recall that if we have 2 vectors $\bar{a}$ and $\bar{b}$ with an angle of θ between them

then $$\bar{a} \cdot \bar{b} = |a||b|\cos\theta$$

$$\Rightarrow \cos\theta = \frac{\bar{a} \cdot \bar{b}}{|a||b|}$$

So

$$\cos^2\theta_{ij} = \frac{(c_i \cdot c_j)^2}{|c_i|^2 |c_j|^2} = \frac{(c_i \cdot Mc_i)^2}{c^4}$$

since $c_j = Mc_i$

and $|c_i| = |c_j| = c$

thus

$$w = \frac{1}{nc^4} \sum_i (c_i \cdot Mc_i)^2 \tag{33}$$

Now consider $c_j \neq Mc_i$, i.e. c_j can assume any value other than Mc_i.

From equations (31) and (32) respectively, we have values for $<\cos^2\theta_{ij}>$ and $<\cos^2\theta_{ij}>_{c_j=Mc_i}$.

Thus, using simple weighted averages,

$$<\cos^2\theta_{ij}>_{c_j\neq Mc_i}=\frac{n<\cos^2\theta_{ij}>-<\cos^2\theta_{ij}>_{c_j=Mc_i}}{n-1}$$

$$=\frac{n\frac{1}{D}-w}{n-1}$$

so

$$<\cos^2\theta_{ij}>_{c_j\neq Mc_i}=\frac{1}{n-1}\left(\frac{n}{D}-w\right) \tag{34}$$

We wish to estimate the quantity

$$\sum_i\sum_j s_i(s_j-s_j')\cos^2\theta_{ij}=\sum_i\sum_j s_i s_j\cos^2\theta_{ij}-\sum_i\sum_j s_i s_j'\cos^2\theta_{ij}$$

$$= \quad Q^* \quad - \quad Q \tag{35}$$

The double sum $Q=\sum_i s_i\sum_j s_j'\cos^2\theta_{ij}$ contains only p^2 terms which are non-zero (i.e. the terms where $s_i=1$ and $s_j'=1$), since there are only p particles present at the site.

There are p terms which correspond to $c_j=Mc_i$, since the output state is derived from the input state through the isometry M.

Thus $p^2-p=p(p-1)$ terms correspond to $c_j\neq Mc_i$.

Therefore, using equations (32) and (34)

$$< Q > = pw + p(p-1)\frac{1}{n-1}\left(\frac{n}{D} - w\right)$$

$$= \frac{p(n-p)w}{n-1} + \frac{p(p-1)n}{(n-1)D} \tag{36}$$

Q^* is simply a special case of Q, where $s_j' = s_j$. This implies that M is the identity, and thus $w = 1$ from (33).

Substituting $w = 1$ into (36) gives us

$$< Q^* > = \frac{p(n-p)}{n-1} + \frac{p(p-1)n}{(n-1)D} \tag{37}$$

From equations (35), (37) and (36), we see that

$$< \sum_i \sum_j s_i(s_j - s_j') \cos^2 \theta_{ij} > = < Q^* - Q >$$

$$= \frac{p(n-p)}{n-1}(1-w) \tag{38}$$

Although this relationship is only approximate, it serves to show that by minimising w, we maximise $\sum_i \sum_j s_i(s_j - s_j') \cos^2 \theta_{ij}$ which in turn minimises μ_4 and hence ν.

Thus to optimise the viscosity, we should choose isometries M so that

$$w = \frac{1}{nc^4} \sum_i (c_i \cdot Mc_i)^2$$

is minimised.

If we let $c_{i_1}, \ldots, c_{i_4}$ be the coordinates of c_i,

and

$$M = \begin{pmatrix} a_{11} & \cdots & a_{14} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{41} & \cdots & a_{44} \end{pmatrix} \tag{39}$$

Then

$$\begin{aligned} w &= \frac{1}{nc^4} \sum_i \left(\sum_\alpha \sum_\beta a_{\alpha\beta} c_{i\alpha} c_{i\beta} \right)^2 \\ &= \frac{1}{nc^4} \sum_i \left(\sum_\alpha \sum_\beta a_{\alpha\beta} c_{i\alpha} c_{i\beta} \right) \left(\sum_\gamma \sum_\delta a_{\gamma\delta} c_{i\gamma} c_{i\delta} \right) \\ &= \frac{1}{nc^4} \sum_\alpha \sum_\beta \sum_\gamma \sum_\delta a_{\alpha\beta} a_{\gamma\delta} \sum_i c_{i\alpha} c_{i\beta} c_{i\gamma} c_{i\delta} \end{aligned} \tag{40}$$

From [12] we have that, by symmetry relations,

$$\begin{aligned} \sum_i c_{i\alpha}^2 c_{i\beta}^2 &= \frac{nc^4}{D(D+2)} \qquad \text{for } (\alpha \neq \beta) \\ \sum_i c_{i\alpha}^4 &= \frac{3nc^4}{D(D+2)} \\ \sum_i c_{i\alpha} c_{i\beta} c_{i\gamma} c_{i\delta} &= 0 \text{ for all other cases.} \end{aligned} \tag{41}$$

Substituting equations (41) into equation (40), we get:

$$\begin{aligned} w &= \frac{1}{D(D+2)} \left\{ \sum_\alpha \sum_{\alpha \neq \beta} (a_{\alpha\alpha} a_{\beta\beta} + a_{\alpha\beta}^2 + a_{\alpha\beta} a_{\beta\alpha}) + 3 \sum_\alpha a_{\alpha\alpha}^2 \right\} \\ &= \frac{1}{D(D+2)} \left\{ \sum_\alpha \sum_{\beta > \alpha} (a_{\alpha\beta} + a_{\beta\alpha})^2 + (\sum_\alpha a_{\alpha\alpha})^2 + 2 \sum_\alpha a_{\alpha\alpha}^2 \right\} \end{aligned} \tag{42}$$

This formula will be useful for determining which isometries of the form (39) minimise viscosity. Those isometries M for which w is minimised will be said to be optimal isometries within their class, the classes to be defined in the next section.

3.3.3 Class Differentiation of Collisions, and Optimal Isometries

The rules by which collisions are chosen have now been defined. In essence, the isometric rules ensure the following. Collision rules are defined as those isometries which preserve the momentum, while minimising kinematic viscosity. All isometries preserve the mass of a state, implying that it is not necessary to consider mass conservation in the selection of collision isometries. The fact that collisions are defined to be isometries ensures that the velocity set is preserved. From this informal definition of isometric rules, it is obvious that the permitted isometries depend on the coordinates of the momentum. If $q=(q_1,q_2,q_3,q_4)$ is the momentum of a pre-collision state, then the isometries that can be applied to the input state depend on the values of the coordinates. For example, if $q_2=0$, then S_2 may be applied, while if $q_3=q_4$, then P_{34} is a valid isometry.

If the momentum vector for every possible state is calculated, it can be seen that there are exactly 7009 distinct values. Hénon shows that these momenta can be 'normalised' by applying a change of coordinates to them, such that the 7009 values reduce to only 37 different 'normalised momenta'. Normalised momenta are defined as those which satisfy:

$$q_1 \geq q_2 \geq q_3 \geq q_4 \geq 0 \tag{43.1}$$

and

$$q_4 = 0 \quad \text{or} \quad q_1 + q_4 < q_2 + q_3 \tag{43.2}$$

Any momentum vector can be normalised by applying isometries to it. The term 'apply an isometry' implies matrix multiplying the isometry with the state matrix, and with the momentum vector of the state. The exact details of how to normalise a momentum vector are given in section 3.4.

Table 2 reproduced from Hénon gives a list of the 37 normalised momenta, the class of which each is a member, and the number r of momentum vectors which reduce to each of the normalised momenta after change of coordinates. Note that the total of this last column is 7009.

Table 2: Normalised momenta of the FCHC lattice.

q_1	q_2	q_3	q_4	Class	r
0	0	0	0	12	1
1	1	0	0	10	24
2	0	0	0	11	24
2	1	1	0	6	96
2	2	0	0	10	24
2	2	2	0	8	96
3	1	0	0	9	144
3	2	1	0	3	192
3	3	0	0	10	24
3	3	2	0	5	288
3	3	3	1	2	192
4	0	0	0	11	24
4	1	1	0	7	288
4	2	0	0	9	144
4	2	2	0	6	96
4	3	1	0	3	192
4	3	3	0	7	288
4	4	0	0	10	24
4	4	2	0	5	288
4	4	3	1	1	576
4	4	4	0	8	96
5	1	0	0	9	144
5	2	1	0	4	576
5	3	0	0	9	144
5	3	2	0	3	192
5	4	1	0	3	192
5	4	3	0	4	576
5	5	0	0	10	24
5	5	2	0	5	288
6	0	0	0	11	24
6	1	1	0	7	288
6	2	0	0	9	144
6	2	2	0	7	288
6	3	1	0	4	576
6	3	3	0	6	96
6	4	0	0	9	144
6	4	2	0	3	192

Systematic construction of such a table would proceed as follows. For each of the possible 16777216 different input states, the momentum vector would be calculated, and distinct momentum vectors saved. There would be 7009 different momentum vectors by the end of this exercise. Each of these vectors would then be normalised according to the method described in section 3.4. All momentum vectors which reduce to the same normalised momentum would be grouped together. The number of groups obtained in this way would be 37, and the number of original momentums in each of the groups would be the same as those listed in the last column of table 2. Obtaining the data in the column entitled "class" will be discussed below.

Let the subgroup H of G be the set of isometries which preserve a specific normalised momentum vector. The subgroup H is determined for each normalised momentum. This can be achieved by applying the 1152 isometries sequentially to the normalised momentum under consideration, and any isometry which leaves the normalised momentum unchanged is an element of H. In this way, a normalised momentum defines a subgroup. Normalised momenta which define the same subgroup are said to be in the same class. The classes listed in table 2 are determined in this way. After going through this rather tedious process, Hénon discovered that there are 12 different classes. Table 3 adapted from Hénon gives the 12 classes, their definitions, and the number of elements in the subgroups defined by them. This last column is thus the number of isometries which preserve all of the normalised momenta in each class. The numbering of the classes is arbitrary.

Table 3: The 12 classes of FCHC input states.

Class	Definition	|H|
1	$q_1 = q_2 > q_3 > q_4 > 0$	2
2	$q_1 = q_2 = q_3 > q_4 > 0$	6
3	$q_1 > q_2 > q_3 > q_4 = 0, q_1 = q_2 + q_3$	6
4	$q_1 > q_2 > q_3 > q_4 = 0, q_1 \neq q_2 + q_3$	2
5	$q_1 = q_2 > q_3 > q_4 = 0$	4
6	$q_1 > q_2 = q_3 > q_4 = 0, q_1 = 2q_2$	12
7	$q_1 > q_2 = q_3 > q_4 = 0, q_1 \neq 2q_2$	4
8	$q_1 = q_2 = q_3 > q_4 = 0$	12
9	$q_1 > q_2 > q_3 = q_4 = 0$	8
10	$q_1 = q_2 > q_3 = q_4 = 0$	48
11	$q_1 > q_2 = q_3 = q_4 = 0$	48
12	$q_1 = q_2 = q_3 = q_4 = 0$	1152

The value of w (as derived in section 3.3.2 and given by equation (42)) is calculated for each element of the subgroup H, for each class. If the minimum value of w obtained in this way is w_{min}, then we define all isometries in H for which $w=w_{min}$ to be optimal isometries for the class which H characterises. Table 4, which has been adapted from Hénon, gives the number of optimal isometries, lists the optimal isometries and gives the value of w_{min}, for each class. The isometries are in normal form, i.e. in the form given by equation (26).

Table 4: Optimal isometries for the various classes of the FCHC lattice.

Class	Number	Optimal Isometries	w_{min}
1	1	P_{12}	1/2
2	2	$P_{23}P_{12}$, $P_{23}P_{13}$	1/4
3	2	$S_4\Sigma_1$, $S_4\Sigma_2$	1/4
4	1	S_4	1/2
5	1	S_4P_{12}	1/3
6	4	$S_4\Sigma_1$, $S_4\Sigma_2$, $S_4P_{23}\Sigma_1$, $S_4P_{23}\Sigma_2$	1/4
7	1	S_4P_{23}	1/3
8	4	$P_{23}P_{12}$, $P_{23}P_{13}$, $S_4P_{23}P_{12}$, $S_4P_{23}P_{13}$	1/4
9	3	S_4S_3, S_3P_{34}, S_4P_{34}	1/3
10	6	$S_3P_{34}P_{12}$, $S_4P_{34}P_{12}$, $S_4S_3\Sigma_1$, $S_4S_3P_{34}P_{12}\Sigma_1$, $S_4S_3\Sigma_2$, $P_{34}P_1\Sigma_2$	1/6
11	6	$S_4S_2P_{23}$, $S_4S_3P_{23}$, $S_3S_2P_{24}$, $S_4S_3P_{24}$, $S_3S_2P_{34}$, $S_4S_2P_{34}$	1/6
12	12	$S_3S_1P_{34}P_{12}$, $S_4S_1P_{34}P_{12}$, $S_3S_2P_{34}P_{12}$, $S_4S_2P_{34}P_{12}$, $S_2S_1P_{24}P_{13}$, $S_4S_1P_{24}P_{13}$, $S_3S_2P_{24}P_{13}$, $S_4S_3P_{24}P_{13}$, $S_2S_1P_{23}P_{14}$, $S_3S_1P_{23}P_{14}$, $S_4S_2P_{23}P_{14}$, $S_4S_3P_{23}P_{14}$	0

The basic method of collision can thus be outlined as follows: For each input state, the momentum vector is calculated. By applying a relevant change of coordinates, the momentum vector is converted to a normalised momentum vector. The class of the input state is then ascertained by the form of the normalised momentum. The collision itself is then performed by applying an optimal isometry of the same class. In classes where there is more than one optimal isometry, the isometry to be applied is chosen at random from among them. The original form is then obtained by applying the change of coordinates in reverse order. The result of this transformation process is the output state of the collision. Note that the change of coordinates is effected by simply applying suitable isometries such as $P_{\alpha\beta}$, , S_α until the definition of normalised momenta as specified above is satisfied. As noted above, all isometries to be used are applied to both the state matrix and the momentum vector.

3.4 Method of Implementation

The preceding section, Theory of the Isometric Method, culminates in a method for determining the output state of a site, given the input state, by making use of isometries. A more detailed recipe for this isometric algorithm is as follows:

1. For a given input state, the four components q_1,q_2,q_3,q_4 of the momentum are calculated, and the state matrix is determined.

2. In order to obtain the normalised momentum, a change of coordinates is effected by applying various isometries to the input state (in the form of the state matrix) and to the momentum vector, until all momentum components are positive and in monotonically decreasing order, and $q_4=0$ or $q_1+q_4 < q_2+q_3$. Note that this is merely the required condition for normalised momenta as specified in (43). The isometries which are required to satisfy this condition are stored, along with the order in which they are applied.

3. The class of the normalised momentum (and thus of the collision) is determined according to table 3. An optimal isometry of this class is then applied, and this constitutes the collision. In a class where there is more than one optimal isometry, one of them is chosen at random. The optimal isometries are obtained from table 4.

4. The coordinates are changed back to their original form by applying the same sequence of isometries as in step 2, but in reverse order. This method is valid due to the fact that applying any isometry in G has an effect identical to that of applying its inverse. This is the reason for storing the required isometries and their order of application in step 2.

This recipe has been coded in FORTRAN on a unix-based workstation which has about one sixteenth of the computing capability of a Cray XMP. The implementation of each of the above steps is discussed in detail below. Many parts of the following discussion are repetitions of previous passages of the present text. The aim of these repetitions is to ensure that the scheme outlined below is self-contained, and can be implemented with little or no need to reference the rest of the document.

1 The 24 possible velocities in the FCHC lattice are

$(\pm1,\pm1,0,0)$, $(\pm1,0,\pm1,0)$, $(\pm1,0,0,\pm1)$, $(0,\pm1,\pm1,0)$, $(0,\pm1,0,\pm1)$ and $(0,0,\pm1,\pm1)$,

and are arbitrarily numbered c_i, i=1,..,24, with $c_i = (c_{i1},c_{i2},c_{i3},c_{i4})$. Consistent numbering of these velocity directions must remain throughout. The pre-collision or input state of a site is represented by an array $s = \{s_1,s_2,...,s_{24}\}$ with s_i being a Boolean value 0 or 1. The value of s_i indicates the presence of a particle with velocity c_i if it is 1, or the absence of such a particle if it is 0. The post-collision or output state of the site is denoted by a similar array $s' = \{s'_1,s'_2,...,s'_{24}\}$.

The input state is usually read in from the automaton, which is the computational module responsible for controlling the propagation step, and keeping track of the states of all lattice sites. For the purposes of this work, input states are generated at random, since no three-dimensional automaton exists at the CFD facility to date. All random numbers that are required for use in the work for this thesis are calculated by means of an adaptation of a random number generating subroutine written by Press *et. al.* [21].

1.a The four components q_1,q_2,q_3,q_4 of the momentum are calculated by

$$q = \sum_{i=1}^{24} s_i c_i = \sum_{i=1}^{24} s_i \begin{pmatrix} c_{i\,1} \\ c_{i\,2} \\ c_{i\,3} \\ c_{i\,4} \end{pmatrix} = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{pmatrix}$$

In the implementation, q is a 4 by 1 matrix with integer elements.

1.b The state matrix is calculated as the 4 by 24 matrix constructed as follows: The 24 possible velocity vectors are written as the 24 columns of a matrix. Because each velocity vector has four components, we get a 4 by 24 matrix. We now multiply each column i by the value (0 or 1) of s_i. This gives us our 4 by 24 element state matrix.

2 Whenever an isometry is applied to a state, it is applied to the state matrix and to the momentum vector. The operation in the group G is standard matrix multiplication, since all isometries M in G are 4 by 4 matrices. The isometries $P_{\alpha\beta}$ and S_α are produced as needed by applying the necessary permutations or sign changes to the 4 by 4 identity matrix. The isometries Σ, Σ_1, and Σ_2 are stored ready for use. To apply the isometries, these 4 by 4 matrices are then multiplied with the 4 by 1 momentum and 4 by 24 state matrices.

2.a If q_α is negative, we apply S_α, which inverts the sign of the corresponding coordinate. In practice, we go through a sequential test of all four momentum components, applying S_α for all α for which $q_\alpha < 0$. All q_α's are now positive or zero.

2.b $P_{\alpha\beta}$ is used to sort the q_α's into monotonically decreasing order. In practice, a bubble sort is used where, instead of swapping values α and β when necessary, $P_{\alpha\beta}$ is applied to the momentum vector and to the state matrix. We have now satisfied (43.1).

2.c If $q_4>0$ and $q_1+q_4=q_2+q_3$ then Σ_2 must be applied.
If $q_4>0$ and $q_1+q_4>q_2+q_3$ then Σ_1 must be applied.
If the resulting q_4 has become negative, then S_4 must be applied. The resulting momentum satisfies (43.2).

3 The isometry which constitutes the collision is then applied. This isometry is randomly chosen from among all pre-stored optimal isometries.

3.a The class of the normalised momentum (and thus of the collision) is determined according to table 3. This is done by means of a series of 12 IF statements, which compare the normalised momentum against the definitions of the classes given in table 3, until a match is found.

3.b If the momentum matches a specific class definition ICLASS, then the number of optimal isometries MAX pertaining to that class is set, and the isometries themselves are stored in an indexed array. A random number between 1 and MAX (inclusive) is generated, which is used as the index to the pre-stored optimal isometries of the relevant class.

3.c This isometry is then applied as the collision.

4. The coordinates are changed back to their original form by applying the same sequence of isometries as in step 2, but in reverse order. In order to implement this, it is necessary to store all isometries as they are applied (since the order is important) so that they can be re-applied backwards. In achieving this, the isometries had to be given unique numerical codes; after an isometry has been used, it is converted to a number by an arithmetical operation, and this number is then stored in the next available location in an array. When reversing the isometries, this array is accessed from highest index to lowest, the value at each successively lower array location being decoded from its number value into its unique isometry meaning.

3.5 Observations, Findings and Conclusions

The automaton is the module of a lattice gas which keeps track of the states of all sites within the lattice, and controls the propagation step between collisions. The pre-collision states of all sites are passed to the collision module once per timestep, in order that the post-collision states can be obtained. Due to the fact that there is no local three-dimensional automaton to date, the input states are generated randomly for use by the isometric algorithm.

Using these random input states, the collision algorithm has been tested. A continuous check is performed by the algorithm to ensure that all calculated output states satisfy mass and momentum conservation. An error message would be displayed if ever a post-collision state were to be generated which violated this requirement.

The computer used for this implementation is a unix-based workstation which runs at approximately 22 MIPS, and has about one sixteenth of the computing capability of a Cray XMP. Despite the reasonable efficiency of the hardware used, the runtimes are slow, even when compilation of the code is done utilising the full optimisation option. Rivet in [24] warns of the long computation time required to determine a post-collision state, which is due to the large number of arithmetic manipulations required. These manipulations can be listed as follows.

1. Calculation of the momentum components and the state matrix.

2. Several transformations of matrices, in order to generate the required isometries: All S and P isometries are calculated each time they are required, just before they are applied.

3. Numerous matrix multiplications in applying these isometries.

4. Evaluation of many logical and conditional tests in the form of FORTRAN IF statements, used for purposes of selection and testing.

5. The generation of a random number, used for choosing an optimal isometry.

6. Encoding and decoding of the isometries required for calculating the normalised momenta, such that they can be reused in reverse order for retransforming the output state.

7. Traversing backwards through the same sequence of isometries for retransformation.

The original performance results obtained with this implementation are reported in [18] and are summarised below. Since this first implementation, the hardware has been upgraded and a few improvements have been made to the coding in order to make better use of the hardware. These improvements, and their associated speedups, are as follows:

The original coding now takes 51 CPU seconds to generate 10000 output states, as opposed to the original 130 CPU seconds.

Since the floating point performance of the machine is far better than that for integer operations, all arrays have been changed from type INTEGER arrays to type REAL arrays.

Matrix multiplication is now performed by making use of assembler routines which are provided in a software library with the operating system of the hardware. The relevant calls are made to these routines in place of the FORTRAN routines written by the author, as used in [18]. The assembler routines can be executed far faster than compiled source code. Due to exceptional floating point performance noted when using the assembler 4 by 4 matrix multiplying routine, state matrices are decomposed into a set of six 4 by 4 matrices before multiplication. These modifications reduce the required time from 51 CPU seconds to 27 CPU seconds.

Since IF statements are costly in CPU time, an effort has been made to reduce the required number of such tests.

Random pre-collision states are generated as required, instead of being read in from a pre-stored file. This reduces the required runtime from around 27 CPU seconds to about 19.

The time taken to generate output states for 10 000 random input states as reported in [18] is about 130 CPU seconds, implying that roughly 73 collision outcomes could be calculated per second. With the implementation of the above modifications, the process has been speeded up by a factor of over 7, and 526 output states can now be generated per second. The code is modularised into several subroutines, and the final version incorporating the modifications as discussed above, appears in appendix A.

Even though this seems reasonably fast, it is too slow for practical applications. A typical lattice gas simulation of 20 thousand timesteps over a domain of 128^3 sites, would take almost two and a half years to complete, neglecting time needed to perform propagation and site updating. The Reynolds coefficient reported for this model is 2.0, which implies low efficiency of the gas (see section 1.7). Despite the resulting lack of applicability of the isometric algorithm to practical situations, it still serves as a good starting point in investigating the field of three-dimensional lattice gas collision rules, even if only as a basis for comparison against future collision rule generators and tables.

Due to the lack of efficiency of the algorithmic method, other methods must be employed. The aim is to utilise a method of collision outcome determination which improves the overall efficiency of the model. Increasing the efficiency could be achieved in one (or both) of two ways. Firstly, the rate at which the collision outcomes can be obtained, could be improved, and secondly, the Reynolds coefficient could be increased.

It is expected that the only way to achieve a practical solution for our purposes is to apply some sort of lookup table approach to determining collision outcomes. The lookup table method, in standard form, requires the generation of a collision table, which is a complicated and lengthy process. The advantage is that this process is only performed once; thereafter, output states are read from the rule table as required. Determination of output states at runtime is thus very fast. The drawback, of course, is the amount of memory required to store this table. Some sort of compromise between lookup and computation is thus required.

One possible approach is the collision table splitting method used by Somers and Rem [27], as previously discussed in section 2.6. In light of the fact that Somers and Rem have a hardware system similar to the local Transputer system, their method would seem attractive. One possible split suggested by them involves breaking the 24-bit collision of the standard FCHC into two pieces, a 16-bit collision and an 8-bit collision [28]. The 2^{24} entry collision table is replaced by two tables, one with 2^{16} entries, and the other with 2^{8} entries. The two tables are then separately constructed and optimised.

A large drawback of this method is that it allows only some of the original collisions to occur. This reduced collision set means that the Reynolds coefficient is lowered, which is obviously undesirable. Hénon in [15] discusses the implementation of a strategy which makes use of the symmetries of the problem to reduce the storage requirements, without adversely affecting the Reynolds coefficient. This method, as briefly outlined in section 2.7, is a compromise between the speed of a lookup method, and the compactness of a collision algorithm. It is thus strongly recommended that this method be investigated. The implementation is carried out in the following chapter.

4 USING SYMMETRIES TO REDUCE THE COLLISION TABLE SIZE

4.1 Motivation for Using the Method of Reduced Tables

The previous chapter demonstrated the lack of practical applicability of the algorithmic method. The low efficiency of the model can be seen both in terms of the low collision outcome determination rate, and the low Reynolds coefficient obtained. The requirement is to implement a method of collision outcome determination which has a higher overall efficiency than the isometric case. It is known that increasing the efficiency could be achieved in one (or both) of two ways. Firstly, the rate at which the collision outcomes can be obtained, could be improved. Secondly, the Reynolds coefficient could be increased. As can be deduced from the discussion in section 1.7, doubling the Reynolds coefficient improves the overall efficiency of a lattice gas model by a factor of 16.

One solution is to compromise between a lookup table method and a collision algorithm. Collision rules for only a subset of the possible input states are stored. Rules for the remaining input states are calculated, as required, from the rules which have been stored. A method of reducing the collision table which makes use of the symmetries of the FCHC, has been invented and implemented by Hénon [15]. The implementation of this method, based on Hénon's work, has been presented by Lake in [20] and is the subject of this chapter.

The structure of this chapter is as follows. A discussion of the basic method, and how the table is reduced, is presented in section 4.2. Section 4.3 explains the method of implementation, highlighting the small adaptations that were required, and some of the obstacles encountered. The optimisation of the output states for the reduced collision table is discussed in section 4.4, and the findings and recommendations are presented in section 4.5.

4.2 Reducing the Size of the Collision Table

4.2.1 Definitions

Use is made of the symmetries of the problem to reduce the collision table size without degrading the Reynolds coefficient. Some definitions are required:

Definition : Species

Two states are said to be of the same species if one can be derived from the other by means of applying an appropriate isometry.

Definitions : Isometry and Symmetry group

An isometry of the set of velocities present at a FCHC lattice site is a rotation of this set about the origin of the site, plus an optional mirror symmetry.

Let G be the set of all isometries which preserve the set of velocities, or equivalently preserve the set of link directions. The set G is a group, and is known as the symmetry group.

A full discussion of the isometries in G has already been presented in chapter 3.

4.2.2 The Basic Principle of the Method

The underlying philosophy of the reduction scheme is that only one collision rule need be stored for each species. In the terminology of Hénon, the input state of this collision rule is called the deputy of its species. By applying an appropriate isometry to any given input state s, it can be transformed into the deputy of its species, which is designated by $\hat{s}$. We can then make use of the collision rule listed in the reduced table. Applying the inverse isometry to the collision outcome will then provide the required output state.

Due to the fact that the resulting collision table is sparse, both the input and output states need to be stored. In the case of a full table, only the output state needs to be stored, since the input state is indirectly stored as the index of the array of output states. Getting the output state s' for a given input state s, using the reduced table, now consists of the following steps:

1. Find the isometry M which will reduce the input state s to its deputy input state $\hat{s}$, and apply M to s.

2. Look up the deputy output state $\hat{s}'$ which is stored for $\hat{s}$.

3. Apply the inverse isometry M^{-1} to $\hat{s}'$ and hence recover s', the desired output state of s.

4.2.3 Finding the Deputy Input States

In the case of the FCHC lattice gas, there are approximately 17 million different states which a site can assume. If any conditional isometry in G (other than the identity) is applied to all possible states, the result is a set of less than 17 million different states. For the purposes of this work, a conditional isometry is an isometry which is applied to all states which satisfy a certain condition. The procedure (if $q_1 > q_2$ apply S_{12}) is an example of a conditional isometry. Extending this principle, a set of well chosen conditional isometries can be applied to all possible states in such a way that a greatly reduced table can be obtained.

There are 18736 different species of states. If we applied all of the isometries in G to each of the possible states, we could reduce the 17 million possible states to only 18736 different states. This is impractical, especially in view of the fact that all of these isometries would have to be applied to each input state in calculating its output state. In practice therefore, a set of isometries is chosen which effectively reduces the number of different states as far as possible, in as few isometries as possible. Essentially, one has to settle for more than one collision rule per species. In [15], Hénon chooses a set of 23 conditional isometries which reduce the total number of different input states to 29312. These isometries are applied in a process called momentum normalisation followed by normalised ascent. Note that the definition of normalised momentum used by Hénon in [15] is slightly different to that used in [11], given by (43).

Momentum Normalisation

Momentum normalisation is a procedure in which the first 11 of the 23 chosen isometries are applied conditionally to each of the 17 million possible states. In essence, each of these isometries is applied to the input state under consideration only if it has the effect of helping to ensure that the momentum components obey Hénon's definition of normalised momentum: If q_1, q_2, q_3 and q_4 are the four components of the momentum vector, then if

$$q_1 \geq q_2 \geq q_3 \geq q_4 \geq |q_1 - q_2 - q_3|$$

is true, the momentum of the state is said to be normalised.

The 11 isometries which are required to ensure momentum normalisation are

1. If $q_1 < 0$, then apply S_1
2. If $q_2 < 0$, then apply S_2
3. If $q_3 < 0$, then apply S_3
4. If $q_4 < 0$, then apply S_4

This ensures that all momentum components are positive or zero.

5. If $q_1 < q_2$, then apply P_{12}
6. If $q_3 < q_4$, then apply P_{34}
7. If $q_1 < q_3$, then apply P_{13}
8. If $q_2 < q_4$, then apply P_{24}
9. If $q_2 < q_3$, then apply P_{23}

The momentum components are now sorted into non-increasing order.

10. If $q_1 + q_4 < q_2 + q_3$, then apply Σ_1.
11. If $q_2 + q_3 + q_4 < q_1$, then apply Σ_2.

Applying these 11 conditional isometries to all possible input states reduces the number of different states to 316301.

Normalised Ascent

Normalised Ascent is a procedure in which 12 further well chosen isometries are applied conditionally to each of the remaining 316301 different states. Each of these isometries is applied to the state under consideration only if doing so meets the following two requirements:

1. The momentum of the state remains normalised

2. The integer lying between 0 and 16777215 inclusive, which is the decimal equivalent of the 24-bit binary number representing the state s, increases. Let this integer be called the code of the state s, designated by $[s]$.

The following 12 conditional isometries were chosen by Hénon for this procedure, due to the fact that they are particularly effective in further reducing the number of different states:

1. If $[\Sigma_1 s] > [s]$ and $q_1 + q_4 = q_2 + q_3$, apply Σ_1.
2. If $[P_{34}s] > [s]$ and $q_3 = q_4$, apply P_{34}.
3. If $[\Sigma_2 s] > [s]$ and $q_2 + q_3 + q_4 = q_1$, apply Σ_2.
4. If $[S_4 s] > [s]$ and $q_4 = 0$, apply S_4.
5. If $[P_{23}s] > [s]$ and $q_2 = q_3$, apply P_{23}.
6. If $[P_{34}s] > [s]$ and $q_3 = q_4$, apply P_{34}.
7. If $[P_{12}s] > [s]$ and $q_1 = q_2$, apply P_{12}.
8. If $[S_3 s] > [s]$ and $q_3 = 0$, apply S_3.
9. If $[S_4 s] > [s]$ and $q_4 = 0$, apply S_4.
10. If $[P_{34}s] > [s]$ and $q_3 = q_4$, apply P_{34}.
11. If $[\Sigma_1 s] > [s]$ and $q_1 + q_4 = q_2 + q_3$, apply Σ_1.
12. If $[P_{23}s] > [s]$ and $q_2 = q_3$, apply P_{23}.

Applying these 12 conditional isometries to the remaining 316301 states reduces the number of different states to 29312. It is of course perfectly possible to simply apply all 23 conditional isometries to each of the original 17 million states, and the same 29312 reduced states would result. A great deal of computational effort is saved by first obtaining the reduced set of 316301 states, since it is then only necessary to apply the second group of 12 isometries to this reduced set.

4.3 Method of Implementation, and Problems Encountered

4.3.1 Implementation Decisions

Since the generation of a collision table need only be performed once, it has been decided not to parallelise the job over multiple CPU's, as the manpower effort involved cannot be justified. It has thus been decided to use the UNIX based workstation rather than the Transputer system for the collision table generation, due to its better performance. All applicable methods (such as utilising real arrays and assembler routine calls) that have been learnt in the previous chapter in order to reduce the runtimes, are incorporated. A binary search algorithm and an efficient sorting routine, which are adapted from routines written by Press *et. al.* [21], have been utilised where necessary in this implementation.

4.3.2 Overall Method

The overall method used in implementing the reduced table locally follows Hénon's work quite closely. A few differences in approach have however been made.

In Hénon's case, a fully optimised 2^{24} entry collision table was available, having already been successfully implemented on a Cray-2 computer (which is capable of handling a lookup table of this size). When implementing the FCHC lattice gas on a Connection Machine (CM-2) with relatively little local memory, the challenge was to find a way of storing these optimal collisions in less memory. The 29312 deputy input states were obtained by applying the isometries of momentum normalisation and normalised ascent, to the set of possible input states, as discussed in the previous section. Fully optimal deputy output states could then simply be obtained from the complete collision table, resulting in a fully optimised reduced collision table.

No complete optimised collision table has been generated locally to date. Such a table cannot even be stored in local memory on the available hardware. It might have been possible to generate such a table and store it in files on disk, and then proceed in exactly the manner described above. This would, however, be rather a waste of computational resources. The method which has been opted for is as follows.

The 29312 deputy input states are generated in exactly the same way as used by Hénon. The programs and subroutines used in order to determine these deputies are listed in appendix B. As mentioned in section 4.2.3, the reduction of states is performed in two stages. There are

thus two separate main programs, and a set of subroutines, most of which are common to both programs. Optimisation of output states is then performed only for these deputies. This obviously saves enormous amounts of computer time. Once this step is complete, an optimised reduced collision table is available.

4.3.3 Problems Concerning Link Numbering

In general, in the literature, certain numbering conventions are considered to be arbitrary. The only proviso is that once this numbering has been chosen, consistency must be maintained throughout the system. The numbering conventions concerned are:

- The numbering of the 24 FCHC links.
- The one-to-one correspondence between the bits of the state and the links, ie which bit of the 24-bit state refers to which of the links.
- The significance of the bits in determining the code of a state, ie which bit to multiply by 2^{23}, which to multiply by 2^{22}, and so on.

Difficulty was experienced in the process of trying to repeat Hénon's work. Applying the 11 isometries of momentum normalisation to all possible states yielded 316301 different states, as expected. When an attempt was made to further reduce this number by normalised ascent, the number of states reduced to 34696 instead of the expected 29312.

It was realised that the numbering conventions are in fact not arbitrary at all where the determination of the code of a state is concerned. All of the above-mentioned conventions need to be fully specified in order to ensure that the states reduce in a pre-determined way. The reason for this is that the reduction depends on the code, which in turn depends upon the bit significance, and the value (0 or 1) of the bit under consideration depends upon which link is associated with that bit. Any change in any one of these conventions causes the states to reduce differently.

The conventions used by Hénon in [15] are simply those which have historically always been used in his FCHC work, and no particular choice of numbering convention had been made for this implementation. The conventions used in [15] were obtained [16], and once implemented, yielded the expected results.

There are 24! different ways of assigning the bit significance in the determination of a code. Several hundred thousand of these were tried at random to see if, using the same set of 23 isometries, it would be possible to achieve a number of reduced states smaller than 29312. The smallest that was found was 30707. At first glance it seems as if Hénon was lucky with his numbering system, but this is obviously not the case. It must be borne in mind that this set of isometries was chosen (for its efficiency in reducing the number of different states), based upon the numbering convention that was being used.

4.4 Optimisation of the Deputy Output States

4.4.1 Derivation of the Optimisation Criterion

Having obtained the deputies, the next step is the optimisation of output states for these input states. The aim is to choose rules which minimise the kinematic viscosity and hence maximise the Reynolds coefficient. Informally, one can say that this involves choosing the output state which minimises the viscosity contribution for that collision, for each given input state.

In [13], Hénon describes a search for optimal rules. This is carried out in two stages: First, for each value of d (the average number of particles per oriented link), rules are found which minimise the viscosity. Then, the d for which the Reynolds coefficient is maximised, is selected. Optimising the collision table means pairing the states into input-output pairs in an optimal way. The optimisation problem is too large to handle in its original form, as there are about 17 million possible of each input and output states.

Hénon employs several methods to reduce the complexity of the optimisation problem. The most significant of these is the division of the problem into several packets. All states which have the same mass and momentum vector are said to belong to the same packet. Since collision rules must ensure that mass and momentum are conserved, the output state for a given input state must be chosen from within the same packet. This means that the optimisation can be carried out as a separate (and much smaller) problem for each packet. There are roughly 175000 packets, and once optimisation has been completed for all packets, a fully optimised collision table is obtained.

Using a form of approximate optimisation, the problem reduces to a simple greedy algorithm, which has computing complexity proportional to the square of the number of states in the packet under consideration. The output state for which the viscosity contribution is minimised (for a specific input state) is matched to that input state, and the pair is then removed from the packet. This continues until the packet is empty.

This algorithm is approximate in the following sense. A certain output state is often the best output state for more than one input state, but is only matched with the first of these input states, due to the removal of the matched pair immediately after this first matching has been made. The removal process ensures semi-detailed balance. The Reynolds coefficient obtained by Hénon using this method is 7.13.

In [13], exact optimisation is also performed, but only yields a Reynolds coefficient which is 6% better, at 7.57. This optimisation is in a classical form known as the assignment problem, and has computational complexity proportional to the cube of the number of elements in the packet.

The derivation of the optimisation criterion used in choosing a deputy output state $\mathcal{s}'$ for each given deputy input state $\mathcal{s}$, is presented below. This derivation is based on the work by Hénon in [13] and [12]. Since all symbols have the same meanings as previously defined in sections 1.5, 2.7 and 3.3.2, the definitions are not repeated here.

We wish to choose rules which maximise the Reynolds coefficient, which is given by

$$R^* = \frac{c_s g}{\nu} \tag{44}$$

To achieve this, we must minimise the viscosity ν. In [12], it is shown that ν can be written as

$$\nu = \frac{1}{6}\frac{\mu_4}{1-\mu_4} \tag{45}$$

This in turn implies that we must minimise μ_4, given in [12] as

$$\mu_4 = \frac{1}{288} \sum_s \sum_{s'} A(s,s') d^{p-1} (1-d)^{23-p} \sum_\alpha \sum_\beta (Y_{\alpha\beta} + Y'_{\alpha\beta})^2 \tag{46}$$

where $$Y_{\alpha\beta} = \sum_i s_i c_{i\alpha} c_{i\beta} - \frac{p}{2} \delta_{\alpha\beta} \tag{47}$$

Since optimisation is being carried out within a specific packet, $d^{p-1}(1-d)^{23-p}$ is a constant term for each packet. Hence it remains to minimise

$$\sum_s \sum_{s'} A(s,s') W(s,s') \tag{48}$$

where $$W(s,s') = \sum_\alpha \sum_\beta (Y_{\alpha\beta} + Y'_{\alpha\beta})^2 \tag{49}$$

We need to find an output state which minimises (48) for each given deputy input state. Suppose we are currently trying to find the optimal output state for the deputy input state $\hat{s}$. We need to search through all possible output states s' within the same packet, and choose only the best one of them to be the deputy output state, $\hat{s}'$.

Since the rules are to be deterministic, $A(s,s') \in \{0;1\} \forall\, s,s'$. This means that

$$A(\hat{s},s') = \begin{Bmatrix} 0\ \forall \text{possible } s' \text{ except } \hat{s}' \\ 1 \text{ for } \hat{s}' \end{Bmatrix}$$

Expression (48) thus reduces to

$$\sum_{s',s' \neq \hat{s}'} (0 \times W(\hat{s},s')) + 1 \times W(\hat{s},\hat{s}')$$

$$= W(\hat{s},\hat{s}')$$

This shows that for a given deputy input state $\hat{s}$, our job is simply to minimise $W(\hat{s},s')$.

From (49), we can see that

$$W(s,s') = \sum_{\alpha}\sum_{\beta} Y_{\alpha\beta}^2 + 2\sum_{\alpha}\sum_{\beta} Y_{\alpha\beta} Y'_{\alpha\beta} + \sum_{\alpha}\sum_{\beta} Y'^2_{\alpha\beta} \qquad (50)$$

The first and last terms of (50) are independent of the choice of collision rules as long as summation is performed over all possible collisions, in equation (46), and if semi-detailed balance is assumed. It is then sufficient to minimise

$$W_1(\hat{s},s') = \sum_{\alpha}\sum_{\beta} Y_{\alpha\beta} Y'_{\alpha\beta}$$

for each given deputy state $\hat{s}$.

We will call $W_1(\hat{s},s')$ the viscosity contribution of the collision which gives s' as an output state for $\hat{s}$.

The s' for which $W_1(\hat{s},s')$ is minimal will thus be chosen to be deputy output state $\hat{s}'$.

4.4.2 Algorithm for Optimising Deputy Output States

Initialisation phase

A preliminary collision table is produced, in which each deputy input state has itself as an output state.

The mass and momentum of each deputy input state is determined, and from this, a unique packet number is calculated. It is found that the 29312 deputies fall into only 448 different packets. The number of deputies per packet ranges from 0 to 671.

An attendance array is set up with all possible unique packet numbers as indices. The value of the array for a specific index, is the number of deputies present which have that index as their packet number. Most values of this array are thus zero.

An array of dimension 29312, containing the viscosity contribution (ie the value of W_1) for each deputy input state and its current best output state, is set up, with all of its values initialised to a value higher than the greatest attainable value of W_1.

Optimisation phase

The following algorithm is given in pseudo-code as clarification of the method used in the optimisation procedure.

```
For each state = 0 to 16777215
   Determine the unique number of the packet to which this state belongs.
   Set index = packet number.
   If attendance array (index) = 0 then
      This means: there are no deputies which belong to the same packet as this state.
      Therefore this state cannot be a candidate output state for any of the deputies.
      Therefore try next state.
   Else
      This means that one or more deputies belong to the same packet as this state.
      Set numdeps = attendance array(index).
      The current state is a candidate output state for numdeps different deputies.
      For each deputy = 1 to numdeps
         Calculate W_I using deputy as input state, and candidate state as output state.
         If W_I < value of viscosity contribution array for this deputy then
            Replace current output state with candidate output state.
            Replace viscosity contribution array entry for this deputy with W_I value.
         Endif.
      Next deputy.
   Endif.
Next state.
```

4.5 Results and Recommendations

The result of applying the optimisation procedure given in the previous section, is a reduced collision table of 29312 entries, for which the following statements hold:

The table is valid in the sense that mass and momentum are conserved by each collision.

The table has been optimised using a form of exact optimisation, in that the best possible output state has been chosen for each input state, irrespective of whether or not that same output state had already been chosen as the optimal output state for another input state. This is achieved by not withdrawing an input-output pair from the packet once matched. The Reynolds coefficient obtained should be higher than the 7.13 obtained by Hénon in his approximate optimisation algorithm.

Detailed balance and semi-detailed balance have been violated due to not removing matched pairs from the packet.

The procedures of obtaining the deputies, and optimising the choice of an output state for each of these deputies, are fairly computer intensive: The following table gives an indication of the amount of CPU time taken for the various phases of the work, on a UNIX based workstation, capable of about 22 MIPS:

Activity	**CPU time in seconds**
Momentum normalisation of all 16777216 states	43 000
Normalised ascent of the remaining 316301 states	1 200
Optimisation of choice of output states	60 000

The CPU times reported above relate to the construction of the collision table. This procedure is performed only once, whereafter the resulting table is used in determining the collision outcomes during subsequent simulations. The rate at which these outcomes can be obtained is discussed below.

Hénon reports a capability of determining roughly 170 collision outcomes per second per processor of the CM-2 (these results have been recalculated from the reported values, to exclude propagation time). Since there are 16*1024 processors, however, the rate at which outcomes are determined in simulations is impressive, at about $2.8*10^6$ per second.

In the local implementation, the collider routine has been written in such a way as to benefit from the peculiarities of the machine. This includes transforming most of the integer arrays to real arrays, such that use can then be made of assembler routines which are available for performing matrix multiplications and matrix transfers. In such a format, the collider can determine 270 collision outcomes per second during simulation.

Although this is even slower than the 526 collision outcomes per second achieved in the isometric case, the Reynolds coefficient should have improved by a factor greater than 3. The overall computational efficiency of this collision table should therefore be far greater than that obtained in the isometric case, and our objective in this regard has thus been reached.

5 IMPROVEMENTS TO THE REDUCED TABLE IMPLEMENTATION

5.1 Increasing the Rate of Collision Outcome Determination

The work in this section is aimed at obtaining a significant speedup of the rate of determination of collision outcomes. The intention is to sufficiently improve the efficiency of the local lattice gas model, such that it can be of immediate practical use in local lattice gas simulations.

A typical requirement for a local lattice gas simulation is the modelling of a flow regime with a Reynolds number of approximately 100. Using the isometric model, this simulation would require approximately 20000 timesteps, on a lattice of size 128^3. The simulation would take about 2.5 years of collision update time to complete on the local hardware, which is capable of approximately 22 MIPS, and which has been described in section 1.2.

In the format reported at the end of the previous chapter, the reduced table lattice gas model is capable of 270 collision outcome determinations per second, and (as seen from the literature [4,13]) should have a Reynolds coefficient of approximately 7. Using this model, simulation of a flow regime identical to the one discussed above would take approximately 12 days to complete (collision phase only). This enormous improvement in simulation time over the isometric model can be ascribed to the improved Reynolds coefficient, in the light of the considerations of section 1.7.

Although the above-mentioned simulation time is reasonable, any improvement in efficiency is desirable. Improving the Reynolds coefficient significantly, while maintaining semi-detailed balance, cannot be achieved without the addition of rest particles, which is beyond the scope of this thesis. The direction in which an immediate improvement can be sought is to increase the rate of collision outcome determination.

The first modification that has been made is to calculate and pre-store the required reducing isometries, in the form of four by four matrices, at the beginning of the computation. The isometries in this form are then made available to the subroutines where required, by means of FORTRAN COMMON blocks. This modification obviates the need to calculate the isometries by manipulation of the identity matrix, each time the isometry is required. Unfortunately, the speedup obtained from this modification was small, only raising the rate of collision outcome determination from 270 to 279 outcomes per second.

The second modification involves changing the manner in which the isometries are applied in reverse order to the deputy output state, in order to obtain the output state applicable to the original input state. Originally, the isometries, together with the order in which they were applied, were stored, and matrix multiplication was utilised to re-apply these isometries in reverse order. The modification involves constructing a four by four "reversal" matrix which is initialised to be the identity matrix. Each time an isometry is applied, the reversal matrix is multiplied with the isometry as well. In general, when applying an isometry M to a state s, the isometry is multiplied with the state in the order M by s. The reversal matrix and the isometry are however multiplied in the opposite order. Re-applying the isometries when required now only involves the multiplication of the reversal matrix with the deputy output state. The speedup obtained from this modification is less impressive than expected, but still raises the number of collision outcome determinations from 279 per second to 305 per second.

The collider algorithm in the form described in chapter 4 is heavily dependent on computer intensive operations such as matrix multiplication for the application of isometries. The third attempt at increasing the computational efficiency involves rewriting the collider to reduce the use of matrix multiplications and transfers. A totally different method of applying the isometries is implemented.

Consider a pre-collision lattice state in which there is only one particle, residing on a particular link i of an FCHC lattice site. For convenience of notation, let this particular input state be denoted Si. Let the result of applying an isometry $M \in G$ (where G is the symmetry group of the FCHC lattice as in the notation of chapter 3) to the single particle state Si be denoted by MSi. Application of M to Si would either leave the state unchanged (i.e. $MSi=Si$), or it would have the effect of moving the particle from link i to some other link j (i.e. $MSi=Sj$).

The essence of the new isometry method is that the effect of applying M to Si is pre-determined for all $i=1,...,24$ and for all $M \in G$. These results are stored in a table. Consider a two-dimensional table with Si, $i=1,..,24$ as row headings, and the different M isometries as column headings. The table would be filled with entries giving MSi for all the different M and Si values.

Determining the effect of applying an isometry M to a general state $s = \{s_1,s_2,...,s_{24}\}$ then involves reading MSi from the table for all $i=1,...,24$ for the particular isometry M in question, and setting s_{MSi} to the value of the pre-isometry link value s_i. Applying an isometry can thus be performed as a series of table lookups as opposed to a matrix multiplication.

In practice, not all 1152 different isometries of G (as discussed in section 3.3.1) need to be catered for in the table. In the reduced collision table method discussed in chapter 4, only 11 different reducing isometries are used, namely $\{ S_1, S_2, S_3, S_4, P_{12}, P_{13}, P_{23}, P_{24}, P_{34}, \Sigma_1, \Sigma_2 \}$ and only these have therefore to be catered for in attempting to speed up this method. The isometry P_{14} has however also been included, for completeness of the set for future possible applications. All isometries needed to recover the normal form of isometries, as given in equation (26), are therefore present. The redundant but convenient generating set for all isometries $M \in G$ is thereby made available. The isometry lookup table is thus 12 by 24 in size.

The final version of the program and subroutines which are used to determine the collision outcomes is presented in appendix C. This version contains all of the modifications for improving the efficiency, as discussed in this section and as mentioned in sections 4.3.1 and 4.5.

This method of applying the isometries indeed proved to be effective in increasing the rate of collision outcome determination. Over a random 10000 input states, output states could be determined at a rate of 590 per second, which more than doubles the overall efficiency of the lattice gas. The simulation referred to in the preceding paragraphs would require under 5.5 days to complete, as opposed to the previously-mentioned 12 days with the original isometry application method, still assuming a Reynolds coefficient of approximately 7.

It can be argued that the improvements discussed in this section could also be applied to the isometric case, and a speedup obtained. The Reynolds coefficient of the isometric model is so low in relation to the reduced table method however, that it is not advantageous to pursue the former method any further.

5.2 Method of Ensuring Maintenance of Semi-Detailed Balance

The reduced collision table reported at the end of chapter 4 had one serious failing in terms of the local requirements, and the recommendations outlined in section 2.8. The problem lies in the fact that semi-detailed balance is violated by this table. This section outlines the modifications made to the deputy output state optimisation routine, in order to ensure the maintenance of semi-detailed balance. The motivation for maintaining semi-detailed balance, and hence the reason for implementing these modifications, has been presented in section 1.8.

In the original deputy output state optimisation procedure, as discussed in section 4.4.2, the method involved the following. An initialisation phase is implemented, wherein an initial table is set up which simply lists each deputy input state (referred to as the deputy for simplification) as its own output state. An array containing an initial (impossibly high) value of *W1* for each of these deputy input-output pairs is stored, and another matrix (called the attendance matrix) containing the number of deputies present from within each particular packet (as defined in section 2.5) is also initialised.

The optimisation routine which violates semi-detailed balance is then carried out. Sequentially for every possible FCHC state, the packet in which this state lies is determined. If the attendance array (as set up in the initialisation phase) indicates that there are no deputies present from within the same packet, the next input state is considered. If, on the other hand, there are deputies (one or more) which lie within the same packet as the state under consideration, then that state becomes a candidate output state for each of the relevant deputies. The "viscosity contribution" *W1* (as derived in section 4.4.1) is calculated for each of the relevant deputies together with this candidate output state, in turn. For all deputies for which the newly calculated *W1* value is lower than the currently stored *W1* value, the candidate replaces the current output state, and the array of *W1* values is updated. This procedure is repeated until all possible states have been considered.

The problem with this procedure is that after consideration of all possible states, a certain candidate output state is frequently found to be the optimal deputy output state for multiple deputies. Although this routine does give a form of exact optimisation, this repetition of output states within the collision table leads to the violation of semi-detailed balance. This section will describe the modifications to the above-mentioned procedure, in order to ensure that semi-detailed balance is maintained. The programs and subroutines used in the process of selecting deputy output states, as well as a program for testing the resulting collision table, are listed in appendix D. For brevity, all subroutines concerned with merely writing out of collision tables have been omitted.

During the consideration of all possible states, a modification is required in order to satisfy semi-detailed balance. When a state is found to be a candidate output state for one or more deputies, its *W1* value (calculated for the deputy and the candidate) together with the candidate itself, is stored. Through the course of the routine, a list of candidates together with their *W1* values is constructed for each of the 29312 deputies. The list for each deputy is sorted into order of increasing *W1* value, after the discovery of a new candidate for that deputy. The lists

are thus in terms of best candidate to worst candidate for each deputy input state. After the completion of each sorting procedure of a list, the list is truncated such that it contains at most 40 elements. The rest of the routine, as described above, remains intact.

After consideration of all possible states, the same collision table (which violates semi-detailed balance) is produced, but in addition to this, a list of candidates, in order of preference as possible deputy output states, for each deputy input state, is available. The lists range in length from one choice, to the pre-set maximum of 40 optimal choices which are arranged in decreasing order of preference. The reason for this variation in list length is that some deputies will only have one possible output state, while some will have many. These lists make the construction of a collision table which maintains semi-detailed balance possible.

Before constructing a table which satisfies semi-detailed balance, the following brief investigation was performed. Each deputy input state is assigned its optimal candidate (i.e. the first entry on its list) as an output state. The resulting collision table obviously violates semi-detailed balance, due to the fact that several deputies have the same optimal candidate. The surprising result seen is that some entries in this table are different to the corresponding entries in the table generated by the original procedure. These differences are caused by the fact that the entries of the original table are only modified if an improvement in *W1* value is found. In contrast, the list of best optional candidates for each deputy has a new-found candidate added to it, even if the *W1* value of this new candidate is the same as that of another candidate already on the list. This addition to the list is of course subject to the proviso that the new candidate has a *W1* value which falls within the 40 best at this point in the algorithm. This implies that a candidate placed first on a list will be displaced to second position by a new candidate with the same *W1* value.

If the set of *W1* values for the entire original collision table is compared to the similar set for the table made up of the first choice candidates, it is found that the two sets are identical. These two tables which violate semi-detailed balance should thus be equally efficient in terms of Reynolds coefficient.

Some terminology must be introduced to facilitate the discussion which follows. Let the index of a candidate on a list of optional candidates be its position relative to the top of the list. The index of the optimal candidate for a specific deputy is thus 1, the index for the second best choice is 2, and so on.

The next phase involves using the lists of optional candidates to generate a reduced table which satisfies semi-detailed balance. This involves assigning an output state to each deputy in such a manner that no two deputy input states are assigned the same output state, and so that the set of candidates used as deputy output states lie as close to the top of the lists as possible. The implementation of this latter criterion is unfortunately degraded to a certain extent by a consideration discussed below, and it is suspected that this will have a detrimental effect on the Reynolds coefficient.

All deputies within the same packet are considered in a group, since it is only necessary to avoid repetition of output states within a packet. This is because an output state which is a candidate for deputies lying within a certain packet cannot be a candidate for any deputy lying outside that packet.

A certain difficulty has been experienced in trying to carry out optimisation which maintains semi-detailed balance on only the deputy input-output pairs. It has been found that it is necessary to assign output states to deputies within a packet, in order of increasing length of the lists of optional candidates. This implies that within the set of deputies lying in a specific packet, the deputy with the lowest number of optional candidates has its output state assigned first, and so on. This rule was found to be necessary in order to avoid situations from arising wherein a deputy has no options in its list of candidates, which have not already been assigned as deputy output states to other deputies. This situation would cause the method to fail.

One optimisation scheme would be to modify the order in which deputies have their output states assigned to them, until the sum of the indices of the candidates assigned as output states is minimised. It is expected that the required process of having to assign output states to deputies in a specific order will increase the viscosity to a certain extent, and thus reduce the Reynolds coefficient. This is the unfortunate consideration mentioned above. Another possible problem is that the achievement of a statistical isotropy requires a random choice as to which input state will have its optimal candidate assigned as its output state, as discussed in [13] and [25]. The method used in the present text may violate this achievement of statistical isotropy to some extent.

All deputies lying within the same packet are thus sorted into order of increasing number of listed candidate options. The deputy with the shortest list is automatically assigned its first choice of candidate as its deputy output state. No other deputy within the packet can then be assigned this output state. Each subsequent deputy within the packet is considered in order of increasing number of options, and assigned the first candidate on its list which has not

already been assigned to a previously considered deputy. Once all deputies (in groups according to the packets to which they belong) have been considered, a collision table is available which satisfies semi-detailed balance. This method has worked well in the case of 29312 deputies, but may fail (in the manner described in a previous paragraph) for a greater number of deputies, in which case other methods will need to be investigated.

If the set of *W1* values for this new table is compared to the set from the table which violates semi-detailed balance, it is seen that many discrepancies arise. For all table entries for which a difference can be noted, it is seen that the value of *W1* has increased in the new table. This is to be expected, since in the new routine, the optimal output states can no longer always be assigned, as repetition of output states has been prevented.

A routine for testing of collision tables has been written. This algorithm firstly tests that each table entry conserves mass and momentum. Secondly, the number of times that each output state appears in the table is determined, and the maximum of these numbers is printed. For a collision table to be valid, no errors in the first test may be reported. For maintenance of semi-detailed balance, the maximum number of times that any output state may appear in the table is one.

The collision table generated by the algorithm discussed above has been tested with this routine, and is found to be valid and to maintain semi-detailed balance. Collision outcomes can be determined from this collision table at the same rate as from the previous table. This fact is obvious, since the input states of the two tables are identical, and reduction thus occurs in the same manner. The value of the output state determined by the binary search also has no influence on the speed at which the search can be made. The issue of Reynolds coefficient and efficiency is addressed in the following section.

5.3 Calculation of the Reynolds Coefficient of the Reduced Collision Table

An assumption has been made throughout the present text (based on results in the literature), that the Boltzmann Reynolds coefficient of the reduced table model implemented in chapters 4 and 5, is approximately 7.

In this section, the method of calculating the Reynolds coefficient is discussed, based on the work of Hénon in [12],[13] and [15]. This calculation is then performed for the reduced collision table constructed in section 5.2. Inferences as to improvement in overall efficiency, as compared to the isometric model, are then made.

Due to the finding in [4] (as discussed in sections 1.5 and 1.8) which indicates that the Boltzmann Reynolds coefficients of tables which violate semi-detailed balance do not provide useful approximations of the measured values, no attempt is made here to perform the calculation for the collision table constructed in chapter 4, which itself violates semi-detailed balance.

Most of the following equations are repeated from section 4.4.1, and no explanation as to the meanings of symbols is therefore given.

The Reynolds coefficient can be calculated from

$$R^* = \frac{c_s g}{\nu}$$

where

$$\nu = \frac{1}{6}\frac{\mu_4}{1-\mu_4}$$

$$\mu_4 = \frac{1}{288}\sum_s\sum_{s'} A(s,s')d^{p-1}(1-d)^{23-p}\sum_\alpha\sum_\beta (Y_{\alpha\beta}+Y'_{\alpha\beta})^2$$

$$Y_{\alpha\beta} = \sum_i s_i c_{i\alpha} c_{i\beta} - \frac{p}{2}\delta_{\alpha\beta}$$

To simplify the notation, let

$$W = \sum_\alpha\sum_\beta (Y_{\alpha\beta}+Y'_{\alpha\beta})^2 \qquad (51)$$

In [13], it is seen that the optimal value of d is $d = \frac{1}{3}$. Maintenance of semi-detailed balance implies that $c_s = \frac{1}{\sqrt{2}}$, and

$$g(d) = \frac{2}{3}\frac{1-2d}{1-d} = \frac{1}{3}$$

By substituting these values and applying the simplification of notation above, the equations for calculating the Reynolds coefficient reduce to

$$R^* = \frac{(3\sqrt{2})^{-1}}{\nu} \tag{52}$$

where

$$\nu = \frac{1}{6}\frac{\mu_4}{1-\mu_4} \tag{53}$$

$$\mu_4 = \frac{1}{288}\sum_s\sum_{s'} A(s,s')\left(\frac{1}{3}\right)^{p-1}\left(\frac{2}{3}\right)^{23-p} W \tag{54}$$

The method used for calculating the Reynolds coefficient is as follows. The reducing isometries defined in section 4.2.3 are used to sequentially reduce each of the 16777216 possible states to its deputy. The number of states which reduce to each of the 29312 deputies is calculated and stored, and the value of μ_4 is initialised to zero. For each of the 29312 deputies, the following algorithm is performed.

The value of W in equation (51) and the mass p in equation (54) are calculated. The contribution to μ_4 of the deputy input-output pair is obtained by calculating the term $\left(\frac{1}{3}\right)^{p-1}\left(\frac{2}{3}\right)^{23-p}W$ from equation (54). This contribution is then multiplied by the number of states which reduce to the deputy under consideration, and the product is added to the current value of μ_4. Once all of the 29312 deputies have been considered, the μ_4 total can be divided by 288. The value of μ_4 as given by equation (54) is now available. The Reynolds coefficient can be determined from equations (53) and (52) above.

The Reynolds coefficient has been calculated for d values of $24d = 4$ to $24d = 12$ in integer steps, and then from $24d = 7.5$ to $24d = 8.5$ in steps of one tenth. It has thus been varified (as had been expected as a result of studying the literature) that the optimal Reynolds coefficient is obtained at $24d = 8$ or $d = \frac{1}{3}$. The procedure for calculating the Reynolds coefficient has been coded, and the program and subroutine listings are given in appendix E.

The calculated Reynolds coefficient for the reduced collision table which has been constructed to maintain semi-detailed balance, is 5.96. This is far lower than the values of 7.13 and 7.57 obtained by Hénon in [13]. It is suspected that this lower value is due to the difficulties experienced while attempting to perform exact optimisation, when limited to only optimising deputy input-output pairs while maintaining semi-detailed balance. These difficulties have been briefly discussed in section 5.2.

The Reynolds coefficient of 5.96 and a collision outcome determination rate of 590 (as discussed in section 5.1) implies that the overall efficiency of this model is almost 90 times better than the isometric model.

6 CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK

6.1 Satisfaction of Original Objective

The original objective of this thesis was to provide a collision rule set for use in local lattice gas simulations. This would be as a first step in the process of acquiring a local three-dimensional lattice gas capability which would facilitate modelling of fluid flow regimes which are unsuitable to analysis by conventional techniques. The work has involved investigating the available methods, and selecting and adapting suitable techniques to the requirements of the local environment. The aim was thus to provide a means of obtaining collision outcomes which fell within the constraints set by local hardware limitations, and which provided an overall lattice gas efficiency sufficient to be of immediate practical use in local simulations.

A measure of the overall efficiency of a lattice gas model is determined by the rate at which collision outcomes can be determined, and by the Reynolds coefficient. It is obvious how the former factor affects the efficiency, and the latter factor has been discussed in section 1.7. The nett effect of increasing the efficiency of a lattice gas is that the simulation time required to model a given flow regime is reduced. This work has aimed at providing a lattice gas collision table which is efficient enough to allow the simulation of a typical lattice gas problem to be completed in a sufficiently short time.

In selecting the methodology to be adapted for local use, a number of decisions have been taken, and a motivation for the acceptability or otherwise of each of the methods considered has been motivated throughout. As a result of theoretical considerations and a survey of the literature, an isometric lattice gas model and a reduced collision table technique have been implemented on the available hardware, with adaptations of the methods where required.

In particular, due to the inability of local hardware to store a complete optimised collision table, some fairly extensive deviations from the published methods have been necessary in the selection of deputy output states. Adaptations have been necessary in both the generation of a table which violates semi-detailed balance, and in the construction of one which maintains it. These modifications have saved considerable cost in terms of computer time and storage. Some further improvements have been made to the local implementation in order to increase the efficiency.

The overall efficiency of the isometric model is too low to be of practical use, but insight and knowledge has been gained from implementing this first case. In the isometric case in its present form, 526 collision outcomes can be determined per second, but improvements as discussed in chapter 5 could be applied to increase this rate. The low Reynolds coefficient of about 2 however makes it unattractive to expend more research effort on this method. If the simulation of a flow regime with a Reynolds number of about 100 is required to be performed on the available hardware, on a lattice of size 128^3, over 20 thousand timesteps, it would take 2.53 years to complete using the isometric method. This is a lattice gas simulation typical of the local requirement.

A significant improvement in efficiency over the isometric case has been obtained by adapting Hénon's method of reducing the size of the collision table, to the local computer environment. After modification of the implementation methods (as discussed in chapter 5) to suit the local hardware, the rate at which collision outcomes can be determined has been increased to 590 per second. The Reynolds coefficient has been calculated, and is found to be 5.96.

This value is only 80 per cent of the reported optimum for a collision table which has no rest particles and maintains semi-detailed balance (i.e 7.57 for the FCHC-4 model, as seen in table 1 of section 2.5). The local model has, however, been constructed at a much lower computational cost than would have been required by the methods in the literature, had these methods indeed been possible to implement on local hardware. The saving in computational resources was brought about by only optimising the choice of deputy output states.

Optimisation of the deputy input-output pairs in isolation presented a difficulty in the case of maintaining semi-detailed balance, as discussed in section 5.2. It was required that the optimisation scheme be compromised in order to overcome this problem, and it is suspected that this is the cause of the lowered Reynolds coefficient. The current implementation, however, although not optimal, is certainly well suited to the available hardware.

Considering only the improvement in computational efficiency obtained in moving from the isometric scheme to the reduced table method, the simulation time required to model the flow problem outlined in the example above, on the same hardware, would be reduced from 2.53 to about 2.22 years. The improvement of the Reynolds coefficient by a factor of approximately 3 implies that, in order to simulate the same flow regime, the required lattice size need only be 43^3, and the number of timesteps required would be decreased to approximately 6700. The simulation time would therefore reduce from over two years to 10.45 days. This simulation time is sufficiently low as a starting point for three-dimensional lattice gas simulations.

The lattice gas collision table that has been implemented in the CFD facility of the CSIR is comparable with , although less efficient than, the available state of the art, in the absence of violation of semi-detailed balance and addition of rest particles. The collision table which is now locally available can be implemented and used in its present form, with immediate effect.

6.2 Future Direction of Work

6.2.1 Coding of the Lattice Gas Automaton

Having obtained a collision table which is of sufficient overall efficiency to be practically applied to local lattice gas simulation problems, the next intended step is the development of a three-dimensional automaton with which three-dimensional lattice gas simulations can be performed.

The automaton is the software module which stores the current state of every lattice site in the computational domain of the region under consideration, and keeps track of the discrete timesteps. Once per timestep, the automaton initiates the propagation step, in which every particle on the lattice moves from its present site to the neighbouring site in the direction of its velocity vector. After the entire propagation phase has been completed and the new pre-collision states have been determined, the automaton addresses the collision table for each pre-collision state in the lattice, to obtain its post-collision state. This is the computer intensive part of each timestep. The whole process is then repeated until the required number of timesteps have been performed.

Since it is required that three-dimensional lattice gas simulations be performed in the near future, the writing of the automaton will be given priority over any other future work.

6.2.2 Addition of Rest Particles

The addition of rest particles allows the Reynolds coefficient to be increased in a number of ways. Since the overall number of interacting particles in a collision is increased, the mean free path of the particles is reduced, and the viscosity decreases. This leads to an increase in the Reynolds coefficient. The increased number of particles in the collisions also allows greater flexibility in selection of collision rules. Since the rules can therefore be optimised to a greater extent, there is a further beneficial effect on the Reynolds coefficient.

Increasing the Reynolds coefficient has a profound positive effect on the efficiency of a lattice gas. The number of lattice sites required per space dimension to discretise a given domain of interest, as well as the number of timesteps required to perform a given simulation, each decrease linearly with increasing Reynolds coefficient. This means that in a three-dimensional simulation, the total simulation time decreases with a power of four of the increasing Reynolds coefficient. As can be seen from the FCHC-5 model in table 1 of section 2.5, a Boltzmann Reynolds coefficient of 10.71 is attainable by adding three rest particles, without violating semi-detailed balance. This represents an increase in total efficiency, over the locally implemented scheme, by a factor greater than 10.

It is therefore strongly recommended that the addition of rest particles be implemented as the main future strategy in obtaining improved lattice gas efficiency, once the automaton is in place. All implementation decisions in this thesis have borne this future direction in mind, and a very small collision table has been constructed which allows considerable scope for the addition of rest particles on the local hardware.

6.2.3 A Temporary Method of Further Improving the Efficiency

If it is decided that an intermediate means of increasing lattice gas efficiency is required, as a temporary and partial "quick-fix" method before the addition of rest particles can be implemented, then the following recommendation is made.

A possible method of increasing the computational efficiency of the present reduced collision table lattice gas model, has been considered. It is possible to obtain a collision table which is less reduced than the 29312 entry version implemented in chapter 4, by applying fewer isometries to the original 17 million input states, during the reduction process. The collision outcome determination will then be faster, due to the fact that correspondingly fewer isometries will need to be applied to each input state to find its deputy. On the other hand, since the size of the sparse collision table is enlarged, the computational effort required for the binary search through the reduced collision table is increased. It is therefore expected that there will be an optimal collision table size, for this type of investigation.

It should be made clear at this juncture that this type of scheme would only be applied as a temporary method of improving the efficiency of the lattice gas model. The original decision to keep the size of the collision table as small as possible, in order to maximise the future

scope for adding rest particles, retains its priority, due to the fact that a greater improvement in overall efficiency is made available by increasing the Reynolds coefficient, than can be achieved by increasing the rate of collision outcome determination.

The reduced table method of Somers and Rem [29] (as discussed in section 2.7) would provide a reduced table of greater than 29312 entries, from which a known computational speedup of a factor of approximately four could be obtained, as reported in [15]. Since applying a less reduced collision table to achieve an increase in efficiency would only be used to provide a temporary improvement, prior to the addition of rest particles, the research time required to implement this totally different scheme would not be justified. A smaller deviation from the planned future route would be a temporary modification of the present implementation of Hénon's scheme, to increase the size of the reduced table.

The memory of the locally available computer is capable of storing a reduced table of far greater size than 29312 entries. A search for isometries which would reduce the complete table to this maximum available size in as few isometries as possible, would then be made. The less reduced table would then be constructed and optimised, and a temporary method of improving the computational efficiency of the local lattice gas model would be achieved. Investigative work has already been initiated for testing the viability of implementing this scheme.

REFERENCES

[1] Burkard, R.E. and Derigs, U. - "*Assignment and Matching Problems: Solution Methods with FORTRAN-Programs*", Lecture Notes in Economics and Mathematical Systems No 184, Springer, Berlin (1980).

[2] d'Humières, D., Lallemand, P. and Frisch, U., - "*Lattice Gas Models for 3D Hydrodynamics*". Europhys. Lett., 2 (4), pp. 291-297 (1986).

[3] Dubrulle, B. - "*Method of Computation of the Reynolds Number for Two Models of Lattice Gas Involving Violation of Semi-Detailed Balance*". Complex Systems, (2), pp. 577-609 (1988).

[4] Dubrulle, B., Frisch, U., Hénon, M. and Rivet, J.-P., - "*Low-Viscosity Lattice Gases*". Journal of Statistical Physics, vol 59, pp. 1187-1216 (1990).

[5] Frisch, U., Hasslacher, B. and Pomeau, Y., - "*Lattice Gas Automata for the Navier-Stokes Equation*". Phys. Rev. Lett., Vol 56, no 14, p. 1505 (1986).

[6] Frisch, U., d'Humières, D., Hasslacher, B., Lallemand, P., Pomeau, Y. and Rivet, J.-P. - "*Lattice Gas Hydrodynamics in Two and Three Dimensions*". Complex Systems, (1), pp. 649-707 (1987).

[7] Frisch, U., Hénon, M., Rivet, J.-P. and Dubrulle, B. - "*Progress Report on Lattice Gas Hydrodynamics*". November (1988).

[8] Gledhill, I.M.A. - "*Fluid Dynamics on Parallel Processors: the Cellular Automaton Approach*". Parallel Processing: Technology and Applications, H. Neishlos, ed. IOS, Amsterdam (1989).

[9] Gledhill, I.M.A. - "*Advances in Cellular Automata*". AERO 90/030 (1991).

[10] Hardy, J.-P., de Pazzis, O. and Pomeau, Y. - "*Molecular Dynamics of a Classical Lattice Gas: Transport Properties and Time Correlation Functions*". Phys. Rev A13, p. 1949 (1976).

[11] Hénon, M. - "*Isometric Collision Rules for the Four-dimensional FCHC Lattice Gas*". Complex Systems 1, pp. 475-494 (1987).

[12] Hénon, M. - "*Viscosity of a Lattice Gas*". Complex Systems 1, p. 762 (1987).

[13] Hénon, M. - "*Optimization of Collision Rules in the FCHC Lattice gas, and Addition of Rest Particles*". Proceedings of the Workshop on Discrete Kinetic Theory, Lattice Gas Dynamics and Foundations of Hydrodynamics, Torino, September 1988 pp. 146-159, World Scientific Publishing co. (1989).

[14] Hénon, M. - "*On the Relation between Lattice Gases and Cellular Automata*". Proceedings of the Workshop on Discrete Kinetic Theory, Lattice Gas Dynamics and Foundations of Hydrodynamics, Torino, September 1988 pp. 160-161, World Scientific Publishing co. (1989).

[15] Hénon, M. - "*Implementation of the FCHC Lattice Gas Model on the Connection Machine*". J. Stat. Phys. 68, pp 353-377 (1992).

[16] Hénon, M. - private communication, July (1992).

[17] Kadanoff, L.P., McNamara, G.R. and Zanetti, G. - "*From Automata to Fluid Flow: Comparisons of Simulation and Theory*". Physical Review A, Vol. 40, No 8, pp. 4527-4541, October (1989).

[18] Lake, P.J. - "*Construction of Collision Tables for 3-D Lattice Gases*". Proceedings of the Seventeenth South African Symposium on Numerical Mathematics, Umhlanga Rocks, 15-17 July (1991).

[19] Lake, P.J. - "*Methods of Obtaining Collision Outcomes for 3-D Lattice Gases*". To be published in Quaestiones Mathematicae 15 (1992).

[20] Lake, P.J. - "*Using Symmetries to Reduce a 3-D Lattice Gas Collision Table*". Proceedings of the Eighteenth South African Symposium on Numerical Mathematics, Durban, 13-15 July (1992).

[21] Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T. - "*Numerical Recipes The Art of Scientific Computing*". Cambridge University Press (1986).

[22] Rem, P.C. and Somers, J.A. - "*Cellular Automata on a Transputer Network*". Proceedings of the Workshop on Discrete Kinetic Theory, Lattice Gas Dynamics and Foundations of Hydrodynamics, Torino, September 1988 pp. 268-275, World Scientific Publishing co. (1989).

[23] Rivet, J.-P. and Frisch, U. - "*Lattice Gas Automata in the Boltzmann Approximation*". C. R. Acad. Sci. Paris II, 302, p. 267, (1986).

[24] Rivet, J.-P. - "*Three-dimensional Lattice Gas Hydrodynamical Simulations: First Results*". C. R. Acad. Sci. Paris II, 305, pp. 751-756, (1987).

[25] Rivet, J.-P., Hénon, M., Frisch, U. and d'Humières, D. - "*Simulating Fully Three-Dimensional External Flow by Lattice Gas Methods*". Europhys. Lett. 7, pp. 231-236, (1988).

[26] Somers, J.A. and Rem, P.C. - "*A Parallel Cellular Automata Implementation on a Transputer Network for the Simulation of Small Scale Fluid Flow Experiments*". Proceedings SHELL conference on Parallel Computing, LNCS Springer (1988).

[27] Somers, J.A. and Rem, P.C., "*The Construction of Efficient Collision Tables for Fluid Flow Computations with Cellular Automata*". Cellular Automata and Modelling of Complex Physical Systems, P.Manneville, ed. Springer Proceedings in Physics 46, pp. 161-177, (1989).

[28] Somers, J.A. and Rem, P.C. - "*Suggestion for 16-bit Collision Table Using Term-Splitting in Hénon's Formula*". Private Communication, September (1989).

[29] Somers, J.A. and Rem P.C. - "*Obtaining Numerical Results from the 3D FCHC-Lattice Gas*". Lecture Notes in Physics 398 pp 59-78 (1992).

[30] Von Neumann, J. - "*Theory of Self-Reproducting Automata*". University of Illinois Press (1966).

[31] Wolfram, S. - "*Cellular Automaton Fluids 1: Basic Theory*". J. Stat. Phys vol 45, pp. 471-526, (1986).

[32] Wolfram, S. (ed) - "*Theory and Applications of Cellular Automata*". World Scientific Publishing Co, Singapore (1986).

[33] Kadanoff, L.P. and Swift, J., - Phys. Rev., 165 p 310 (1968).

[34] McNamara, G.R. and Zanetti, G., - "*Use of the Boltzmann Equation to Simulate Lattice-Gas Automata*". Phys. Rev. Lett., vol 61 no 20 pp 2332-2335 (1988).

APPENDIX A: ISOMETRIC ALGORITHM

```
c*************************************************************
c     PROGRAM ISOMETRIC
c*************************************************************
c
      program main
c
c     THIS CODE CALCULATES THE OUTPUT STATES FOR numstates INPUT
c         STATES, WHICH ARE GENERATED AS REQUIRED
c     ISOMETRIES ARE GENERATED AS REQUIRED.
c
c     THE WHOLE CODE HAS BEEN MODIFIED FOR SPEEDUP AS FOLLOWS:
c     1.  INTEGERS CHANGED TO REALS
c     2.  USE OF ASSEMBLER ROUTINES FOR MATRIX MULTIPLICATION
c     3.  USE OF 4*4 MATRIX MULTIPLICATION WHERE POSSIBLE, INCL
c             DECOMPOSITION OF 4*24 MATRICES INTO 6 4*4 MATRICES EACH.
c     4.  GENERATING PRE-COLLISION STATES AS REQUIRED, IN
c             PLACE OF READING THEM IN FROM A FILE
c     5.  IN THE CASE OF AN AUTOMATON BEING PRESENT, THE CALL TO
c         POSTSTATE CAN BE OMITTED, RESULTING IN A GREAT SPEEDUP.
c
      common/d/icntmax
      common/f/isocount,isometry
      common/g/iseed
      common/h/c
      real II(4,4),q(4,1),s(4,24),State(24)
      real qin(4,1),inmass
      integer isometry(30)
c
c     last mentioned are the momentum components and the mass of
c     the input state, stored to check that the algorithm conserves
c     mass and momentum
c
c     isometry is an array which stores the isometries that
c     are applied to q and s, as well as the order in
c     which they are applied. A 3-digit number is assigned to the
c     next available array element each time an isometry is applied.
c
c     The first digit says what type of isometry: 1 for sigma
c                                                 2 for sighn
c                                                 3 for permute
c
c     If digit one is a 1, the second digit says: 0 for sigma
c                                                 1 for sigma1
c                                                 2 for sigma2
c                           the third digit is always a 0.
c
c     If digit one is a 2, the second digit gives the number of
c                           the row which has its sign changed
c                           the third digit is always a 0.
c
c     If digit one is a 3, the second and third digits give the
c                          numbers of the two rows to be inter-
c                          changed.
c
c     set II to the identity matrix
c
      call identity(II)
      iseed=133981
      icntmax=0
c
c     Input state normally passed to this routine from automaton
c     The pre-collision state is called: State = ( 0 or 1, 0 or 1, ...)
c     repeated 24 times to indicate presence or absence of
c     a particle with a specific velocity
c
      print *,'Input the number of input states to be processed'
      read  *, numstates
      do 100 istate=1,numstates
c
c        initialise necessary variables
c
         isocount=0
         do 10 i=1,30
            isometry(i)=0
   10    continue
c
         call makestate(State)
         call getstate(State,s,q,qin,inmass)
         call chngcoord(s,q,II)
         call orderq(q,s,II)
         call sumtest(q,s,II)
         call findclass(q,iclass)
         call choice(s,q,iclass,II)
         call backwards(s,q,qin,II)
         call poststate(s,inmass)
  100 continue
c
      stop
      end
c
c*************************************************************
c     SUBROUTINE SIGMA
c*************************************************************
c
      subroutine sigma(a)
c
c     produces the matrix sigma as in Henon and returns it in a
c
      common/f/isocount,isometry
      dimension a(4,4),isometry(30)
c
      isocount=isocount+1
      isometry(isocount)=100
c
      do 150 i=1,4
      do 150 j=1,4
      a(i,j)=-1
      if (i.eq.j) a(i,j)=1
  150 continue
c
c     but sigma has a factor of a 1/2 in front of it.
c     This will be taken into account after applying it
c
      return
      end
c
c*************************************************************
c     SUBROUTINE SIG1
c*************************************************************
c
      subroutine sig1(a)
c
c     generates the matrix sigma1 as per Henon, returning it in a
c
      common/f/isocount,isometry
      dimension a(4,4),isometry(30)
c
      isocount=isocount+1
      isometry(isocount)=110
c
      do 1000 i=1,4
      do 1000 j=1,4
         a(i,j)=1.0
         if (i+j.eq.5) a(i,j)=a(i,j)*(-1)
 1000 continue
```

```
c
c     but sigma1 has a factor of a 1/2 in front of it
c     This will be taken into account after applying it
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE SIG2
c*****************************************************************
c
      subroutine sig2(a)
c
c     generates matrix sigma2 as per Henon and returns it in a
c
      common/f/isocount,isometry
      dimension a(4,4),isometry(30)
c
      isocount=isocount+1
      isometry(isocount)=120
c
      do 1000 i=1,4
      do 1000 j=1,4
         a(i,j)=1.0
 1000 continue
c
      a(2,3)=-1.0
      a(2,4)=-1.0
      a(3,2)=-1.0
      a(3,4)=-1.0
      a(4,2)=-1.0
      a(4,3)=-1.0
c
c     but sigma2 has a factor of a 1/2 in front of it
c     This will be taken into account after applying it
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE IDENTITY
c*****************************************************************
c
      subroutine identity(a)
c
c     produces the four by four identity matrix
c
      real a(4,4)
      do 10 i=1,4
      do 10 j=1,4
      a(i,j)=0
      if (i.eq.j) a(i,j)=1
   10 continue
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE MATMULT
c*****************************************************************
c
      subroutine matmult(a,k,l,b,m,n,ab)
c
c     multiplies two matrices a and b to form the matrix ab.
c     Note matrix a has k rows and l columns (k*l), while b has m
c     rows and n columns (m*n). This means that ab has k rows
c     and n columns (k*n), and to work, l must equal m.
c
      dimension a(k,l),b(m,n),ab(k,n)
      integer h
      if (l.ne.m) print *,'matrix multiplication failed'
c
c     initialise ab(i,j) to zero
c
      do 17 i=1,k
      do 17 j=1,n
      ab(i,j)=0
   17 continue
c
      do 20 i=1,k
      do 20 j=1,n
      do 18 h=1,m
         ab(i,j)=ab(i,j)+a(i,h)*b(h,j)
   18 continue
   20 continue
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE TRANSFER
c*****************************************************************
c
      subroutine transfer(a,k,l,b,m,n)
c
c     overwrites the contents of matrix b with those of matrix a
c
      real a(k,l),b(m,n)
      if (k.ne.m.or.l.ne.n) print *,'matrix transfer failed'
      do 20 i=1,k
      do 20 j=1,l
         b(i,j)=a(i,j)
   20 continue
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE SIGHN
c*****************************************************************
c
      subroutine sighn(II,n,a)
c
c     produces the 4*4 matrix Sn from matrix II as in Henon
c     Isometric Rules, and stores the Sn of II in a.
c
      common/f/isocount,isometry
      real II(4,4),a(4,4)
      integer isometry(30)
c
      isocount=isocount+1
      isometry(isocount)=200+(10*n)
c
      do 125 i=1,4
      do 125 j=1,4
      a(i,j)=II(i,j)
  125 continue
      i=n
      do 130 j=1,4
      a(i,j)=(-1)*II(i,j)
  130 continue
c
      return
      end
c
```

```
c*************************************************************
c     SUBROUTINE PERMUTE
c*************************************************************
c
      subroutine permute(II,n,m,a)
c
c     produces the 4*4 matrix Pnm as in Henon's Isometric Rules,
c     and stores the Pnm of II in a
c
      common/f/isocount,isometry
      real a(4,4),II(4,4)
      integer isometry(30)
c
      isocount=isocount+1
      isometry(isocount)=300+(10*m)+n
c
      do 135 i=1,4
      do 135 j=1,4
      a(i,j)=II(i,j)
  135 continue
c
      do 140 j=1,4
      idum=a(m,j)
      a(m,j)=a(n,j)
      a(n,j)=idum
  140 continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE APPLYS
c*************************************************************
c
      subroutine applys(a,k,l,b,m,n)
c
c     This subroutine applies the isometry a (k*l but usually 4*4)
c     to the matrix s (which is usually 4*24, but has general
c     dimensions m*n) and returns s after its transformation.
c
      real SOFAR(4,24),a(4,4),b(4,24)
c
      do 100 irow=1,24,4
         call vec_$mat_mult(a,b(1,irow),SOFAR(1,irow))
  100 continue
      call transfer(SOFAR,k,n,b,k,n)
c
      return
      end
c
c*************************************************************
c     SUBROUTINE APPLYQ
c*************************************************************
c
      subroutine applyq(a,k,l,b,m,n)
c
c     This subroutine applies the isometry a (k*l but usually 4*4)
c     to the matrix q (which is usually 4*1, but has general
c     dimensions m*n) and returns q after its transformation.
c
      real SOFAR(4,1),a(4,4),b(4,1)
c
c     call matmult(a,k,l,b,m,n,SOFAR)
      call vec_$mat_multn(a,b,4,4,1,SOFAR)
      call transfer(SOFAR,k,n,b,k,n)
c
      return
      end
c
c*************************************************************
c     SUBROUTINE RAN
c*************************************************************
c
      subroutine ran(n,iseed,irand)
c
c     This subroutine produces an integer random number (irand) between
c     1 and n, and modifies the seed (iseed) for further use of
c     the random number generator.
c     Adapted from numerical recipes.
c
      parameter(ia=7141,ic=54773,im=259200)
      iseed=mod(iseed*ia+ic,im)
      rand=float(iseed)/float(im)
      rand=rand*n+1.0
      irand=int(rand)
      return
      end
c
c*************************************************************
c     SUBROUTINE MAKESTATE
c*************************************************************
c
      subroutine makestate(State)
c
c     generates one random pre-collision state
c     The pre-collision state is called:
c     State = ( 0.0 or 1.0,  0.0 or 1.0, ...)
c     repeated 24 times to indicate presence or absence of
c     a particle with a specific velocity
c
      real State(24)
c
      do 20 j=1,24
      call ran(2,iseed,irand)
      State(j)=irand-1
   20 continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE GETSTATE
c     step 1
c*************************************************************
c
      subroutine getstate (State,s,q,qin,inmass)
c
c     This subroutine returns the four components of momentum (q)
c     and the state matrix (s) for a given input state (State)
c
      common/h/c
      real c(24,4),State(24),s(4,24),q(4,1),qin(4,1)
c
c     velocity directions ci=(ci1,ci2,ci3,ci4) -- order is
c     arbitrary, but once the numbering order has been
c     selected, consistency in the numbering must be maintained.
c
      data (c( 1,j),j=1,4) /  1.0,  1.0,  0.0,  0.0/
      data (c( 2,j),j=1,4) / -1.0, -1.0,  0.0,  0.0/
      data (c( 3,j),j=1,4) / -1.0,  1.0,  0.0,  0.0/
      data (c( 4,j),j=1,4) /  1.0, -1.0,  0.0,  0.0/
      data (c( 5,j),j=1,4) /  1.0,  0.0,  1.0,  0.0/
      data (c( 6,j),j=1,4) / -1.0,  0.0, -1.0,  0.0/
      data (c( 7,j),j=1,4) / -1.0,  0.0,  1.0,  0.0/
      data (c( 8,j),j=1,4) /  1.0,  0.0, -1.0,  0.0/
```

```
      data (c( 9,j),j=1,4) /  1.0,  0.0,  0.0,  1.0/
      data (c(10,j),j=1,4) / -1.0,  0.0,  0.0, -1.0/
      data (c(11,j),j=1,4) / -1.0,  0.0,  0.0,  1.0/
      data (c(12,j),j=1,4) /  1.0,  0.0,  0.0, -1.0/
      data (c(13,j),j=1,4) /  0.0,  1.0,  1.0,  0.0/
      data (c(14,j),j=1,4) /  0.0, -1.0, -1.0,  0.0/
      data (c(15,j),j=1,4) /  0.0, -1.0,  1.0,  0.0/
      data (c(16,j),j=1,4) /  0.0,  1.0, -1.0,  0.0/
      data (c(17,j),j=1,4) /  0.0,  1.0,  0.0,  1.0/
      data (c(18,j),j=1,4) /  0.0, -1.0,  0.0, -1.0/
      data (c(19,j),j=1,4) /  0.0, -1.0,  0.0,  1.0/
      data (c(20,j),j=1,4) /  0.0,  1.0,  0.0, -1.0/
      data (c(21,j),j=1,4) /  0.0,  0.0,  1.0,  1.0/
      data (c(22,j),j=1,4) /  0.0,  0.0, -1.0, -1.0/
      data (c(23,j),j=1,4) /  0.0,  0.0, -1.0,  1.0/
      data (c(24,j),j=1,4) /  0.0,  0.0,  1.0, -1.0/
c
c     calculate state matrix (s) dependent on c and State
c
      inmass=0
      do 22 i=1,24
      inmass=inmass+State(i)
      do 22 j=1,4
         s(j,i)=State(i)*c(i,j)
  22  continue
c
c     Calculate momentum q=( q(1,1),q(2,1),q(3,1),q(4,1) )
c     from state matrix, noting that:
c     q(i,1) is equivalent to the sum of entries in ith row of
c     state matrix
c
      do 24 j=1,4
         q(j,1)=0
         qin(j,1)=0
  24  continue
c
      do 23 i=1,24
         q(1,1)=q(1,1)+s(1,i)
         q(2,1)=q(2,1)+s(2,i)
         q(3,1)=q(3,1)+s(3,i)
         q(4,1)=q(4,1)+s(4,i)
  23  continue
c
      do 25 j=1,4
         qin(j,1)=q(j,1)
  25  continue
c
      print *,'ORIGINAL INPUT STATE'
      call printt(state,1,24)
c
      return
      end
c
c*************************************************************
c     SUBROUTINE CHNGCOORD
c     step 2a
c*************************************************************
c
      subroutine chngcoord(s,q,II)
c
c     This subroutine performs the neccessary change of coordinates
c     so that the normalised momenta can be applied.
c
      real s(4,24),q(4,1),II(4,4),sn(4,4),c(4,24)
c
      do 30 i=1,4
         if (q(i,1).lt.0) then
            call sighn(II,i,sn)
            call applys(sn,4,4,s,4,24)
            call applyq(sn,4,4,q,4,1)
         end if
  30  continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE ORDERQ
c     step 2b
c*************************************************************
c
      subroutine orderq(q,s,II)
c
c     orders the components of the momentum into non-increasing
c     order
c
      real q(4,1),s(4,24),II(4,4),SOFAR(4,4)
c
      do 225 i=1,4
      do 225 j=1,4-i
      if (q(j,1).lt.q(j+1,1)) then
         call permute(II,j,j+1,SOFAR)
         call applys(SOFAR,4,4,s,4,24)
         call applyq(SOFAR,4,4,q,4,1)
      endif
  225 continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE SUMTEST
c     step 2c
c*************************************************************
c
      subroutine sumtest(q,s,II)
c
c     tests the sums as per step 2c in Henon
c
      real s(4,24),q(4,1),II(4,4),a(4,4)
c
      isum1=q(1,1)+q(4,1)
      isum2=q(2,1)+q(3,1)
c
      if (q(4,1).gt.0.and.isum1.eq.isum2) then
         call sig2(a)
         call applyq(a,4,4,q,4,1)
         call applys(a,4,4,s,4,24)
         do 10 i=1,4
            q(i,1)=q(i,1)*0.5
            do 10 j=1,24
               s(i,j)=s(i,j)*0.5
  10     continue
      endif
c
      if (q(4,1).gt.0.and.isum1.gt.isum2) then
         call sig1(a)
         call applyq(a,4,4,q,4,1)
         call applys(a,4,4,s,4,24)
         do 20 i=1,4
            q(i,1)=q(i,1)*0.5
            do 20 j=1,24
               s(i,j)=s(i,j)*0.5
  20     continue
      endif
```

```
c
      if (q(4,1).lt.0) then
         call sighn(II,4,a)
         call applys(a,4,4,s,4,24)
         call applyq(a,4,4,q,4,1)
      endif
c
      return
      end
c
c*************************************************************
c     SUBROUTINE FINDCLASS
c     step 3a
c*************************************************************
c
      subroutine findclass(q,iclass)
c
c     determines the class of a collision
c
      real q(4,1),qq(4)
c
c     note: to make notation simpler in this routine,
c     q(4,1) is written to qq(4) and used as such
c
      do 250 i=1,4
      qq(i)=q(i,1)
  250 continue
c
      if(qq(1).eq.qq(2).and.qq(2).gt.qq(3).and.qq(3).gt.qq(4).and
     #.qq(4).gt.0) then
         iclass=1
         return
      endif
c
      if(qq(1).eq.qq(2).and.qq(2).eq.qq(3).and.qq(3).gt.qq(4).and
     #.qq(4).gt.0) then
         iclass=2
         return
      endif
c
      if(qq(1).gt.qq(2).and.qq(2).gt.qq(3).and.qq(3).gt.qq(4).and
     #.qq(4).eq.0.and.qq(1).eq.qq(2)+qq(3)) then
         iclass=3
         return
      endif
c
      if(qq(1).gt.qq(2).and.qq(2).gt.qq(3).and.qq(3).gt.qq(4).and
     #.qq(4).eq.0.and.qq(1).ne.qq(2)+qq(3)) then
         iclass=4
         return
      endif
c
      if(qq(1).eq.qq(2).and.qq(2).gt.qq(3).and.qq(3).gt.qq(4).and
     #.qq(4).eq.0) then
         iclass=5
         return
      endif
c
      if(qq(1).gt.qq(2).and.qq(2).eq.qq(3).and.qq(3).gt.qq(4).and
     #.qq(4).eq.0.and.qq(1).eq.2*qq(2)) then
         iclass=6
         return
      endif
c
      if(qq(1).gt.qq(2).and.qq(2).eq.qq(3).and.qq(3).gt.qq(4).and
     #.qq(4).eq.0.and.qq(1).ne.2*qq(2)) then
         iclass=7
         return
      endif
c
      if(qq(1).eq.qq(2).and.qq(2).eq.qq(3).and.qq(3).gt.qq(4).and
     #.qq(4).eq.0) then
         iclass=8
         return
      endif
c
      if(qq(1).gt.qq(2).and.qq(2).gt.qq(3).and.qq(3).eq.qq(4).and
     #.qq(4).eq.0) then
         iclass=9
         return
      endif
c
      if(qq(1).eq.qq(2).and.qq(2).gt.qq(3).and.qq(3).eq.qq(4).and
     #.qq(4).eq.0) then
         iclass=10
         return
      endif
c
      if(qq(1).gt.qq(2).and.qq(2).eq.qq(3).and.qq(3).eq.qq(4).and
     #.qq(4).eq.0) then
         iclass=11
         return
      endif
c
      if(qq(1).eq.qq(2).and.qq(2).eq.qq(3).and.qq(3).eq.qq(4).and
     #.qq(4).eq.0) then
         iclass=12
         return
      endif
c
      end
c*************************************************************
c     SUBROUTINE CHOICE
c     steps 3b and 3c
c*************************************************************
c
      subroutine choice(s,q,iclass,II)
c
c     chooses a random optimal isometry dependent on iclass, and
c     applies this isometry
c
      common/d/icntmax
      common/f/isocount,isometry
      common/g/iseed
      real q(4,1),s(4,24),II(4,4),a(4,4),b(4,4)
      integer isometry(30),optiso(12,5)
c
c     there are a maximum of 12 optimal isometries per class, and
c     each optimal isometry is made of at most five elements
c
c     initialise optiso array
c
      do 300 i=1,12
      do 300 j=1,5
         optiso(i,j)=0
  300 continue
c
      if(iclass.eq.1) then
         optiso(1,1)=312
         nopts=1
      endif
c
      if(iclass.eq.2) then
         optiso(1,1)=323
```

```
          optiso(2,1)=323
          optiso(2,2)=313
          nopts=2
        endif
c
        if(iclass.eq.3) then
          optiso(1,1)=240
          optiso(1,2)=110
          optiso(2,1)=240
          optiso(2,2)=120
          nopts=2
        endif
c
        if(iclass.eq.4) then
          optiso(1,1)=240
          nopts=1
        endif
c
        if(iclass.eq.5) then
          optiso(1,1)=240
          optiso(1,2)=312
          nopts=1
        endif
c
        if(iclass.eq.6) then
          optiso(1,1)=240
          optiso(1,2)=110
          optiso(2,1)=240
          optiso(2,2)=120
          optiso(3,1)=240
          optiso(3,2)=323
          optiso(3,3)=110
          optiso(4,1)=240
          optiso(4,2)=323
          optiso(4,3)=120
          nopts=4
        endif
c
        if(iclass.eq.7) then
          optiso(1,1)=240
          optiso(1,2)=323
          nopts=1
        endif
c
        if(iclass.eq.8) then
          optiso(1,1)=323
          optiso(1,2)=312
          optiso(2,1)=323
          optiso(2,2)=313
          optiso(3,1)=240
          optiso(3,2)=323
          optiso(3,3)=312
          optiso(4,1)=240
          optiso(4,2)=323
          optiso(4,3)=313
          nopts=4
        endif
c
        if(iclass.eq.9) then
          optiso(1,1)=240
          optiso(1,2)=230
          optiso(2,1)=230
          optiso(2,2)=334
          optiso(3,1)=240
          optiso(3,2)=334
          nopts=3
        endif
c
        if(iclass.eq.10) then
          optiso(1,1)=230
          optiso(1,2)=334
          optiso(1,3)=312
          optiso(2,1)=240
          optiso(2,2)=334
          optiso(2,3)=312
          optiso(3,1)=240
          optiso(3,2)=230
          optiso(3,3)=110
          optiso(4,1)=240
          optiso(4,2)=230
          optiso(4,3)=334
          optiso(4,4)=312
          optiso(4,5)=110
          optiso(5,1)=240
          optiso(5,2)=230
          optiso(5,3)=120
          optiso(6,1)=334
          optiso(6,2)=312
          optiso(6,3)=120
          nopts=6
        endif
c
        if(iclass.eq.11) then
          optiso(1,1)=240
          optiso(1,2)=220
          optiso(1,3)=323
          optiso(2,1)=240
          optiso(2,2)=230
          optiso(2,3)=323
          optiso(3,1)=230
          optiso(3,2)=220
          optiso(3,3)=324
          optiso(4,1)=240
          optiso(4,2)=230
          optiso(4,3)=324
          optiso(5,1)=230
          optiso(5,2)=220
          optiso(5,3)=334
          optiso(6,1)=240
          optiso(6,2)=220
          optiso(6,3)=334
          nopts=6
        endif
c
        if(iclass.eq.12) then
          optiso(1,1)=230
          optiso(1,2)=210
          optiso(1,3)=334
          optiso(1,4)=312
          optiso(2,1)=240
          optiso(2,2)=210
          optiso(2,3)=334
          optiso(2,4)=312
          optiso(3,1)=230
          optiso(3,2)=220
          optiso(3,3)=334
          optiso(3,4)=312
          optiso(4,1)=240
          optiso(4,2)=220
          optiso(4,3)=334
          optiso(4,4)=312
          optiso(5,1)=220
          optiso(5,2)=210
          optiso(5,3)=324
          optiso(5,4)=313
```

```
          optiso(6,1)=240
          optiso(6,2)=210
          optiso(6,3)=324
          optiso(6,4)=313
          optiso(7,1)=230
          optiso(7,2)=220
          optiso(7,3)=324
          optiso(7,4)=313
          optiso(8,1)=240
          optiso(8,2)=230
          optiso(8,3)=324
          optiso(8,4)=313
          optiso(9,1)=220
          optiso(9,2)=210
          optiso(9,3)=323
          optiso(9,4)=314
          optiso(10,1)=230
          optiso(10,2)=210
          optiso(10,3)=323
          optiso(10,4)=314
          optiso(11,1)=240
          optiso(11,2)=220
          optiso(11,3)=323
          optiso(11,4)=314
          optiso(12,1)=240
          optiso(12,2)=230
          optiso(12,3)=323
          optiso(12,4)=314
          nopts=12
        endif
c
        icntmax=icntmax+isocount
c
        call ran(nopts,iseed,ioptadr)
c
        do 310 i=1,5
        if (optiso(ioptadr,i).eq.0) goto 310
        idig1=int(optiso(ioptadr,i)/100.0)
        idig2=int((optiso(ioptadr,i)-(idig1*100.0))/10.0)
        idig3=optiso(ioptadr,i)-(idig1*100.0)-(idig2*10.0)
c
        if (idig1.eq.1.and.idig2.eq.0)
       #   call sigma(a)
        if (idig1.eq.1.and.idig2.eq.1)
       #   call sig1(a)
        if (idig1.eq.1.and.idig2.eq.2)
       #   call sig2(a)
        if (idig1.eq.2)
       #   call sighn(II,idig2,a)
        if (idig1.eq.3)
       #   call permute(II,idig2,idig3,a)
c
        isocount=isocount-1
        call applys(a,4,4,s,4,24)
        call applyq(a,4,4,q,4,1)
        if (idig1.eq.1) then
           do 20 iii=1,4
              q(iii,1)=q(iii,1)*0.5
              do 20 jj=1,24
                 s(iii,jj)=s(iii,jj)*0.5
 20        continue
        endif
c
  310 continue
c
        return
        end
c*************************************************************
c       SUBROUTINE BACKWARDS
c       step 4
c*************************************************************
c
        subroutine backwards (s,q,qin,II)
c
c       applies all the previous isometries to s and q in reverse
c       order, with the exeption of those in step 3c (collision)
c
        common/f/isocount,isometry
        real a(4,4),s(4,24),q(4,1),qin(4,1),II(4,4)
        integer isometry(30)
c
        do 350 i=isocount,1,-1
        idig1=int(isometry(i)/100.0)
        idig2=int((isometry(i)-(idig1*100.0))/10.0)
        idig3=isometry(i)-(idig1*100.0)-(idig2*10.0)
c
        if (idig1.eq.1.and.idig2.eq.0)
       #   call sigma(a)
        if (idig1.eq.1.and.idig2.eq.1)
       #   call sig1(a)
        if (idig1.eq.1.and.idig2.eq.2)
       #   call sig2(a)
        if (idig1.eq.2)
       #   call sighn(II,idig2,a)
        if (idig1.eq.3)
       #   call permute(II,idig2,idig3,a)
c
        call applys(a,4,4,s,4,24)
        call applyq(a,4,4,q,4,1)
        if (idig1.eq.1) then
           do 20 iii=1,4
              q(iii,1)=q(iii,1)*0.5
              do 20 jj=1,24
                 s(iii,jj)=s(iii,jj)*0.5
 20        continue
        endif
        isocount=isocount-1
  350 continue
c
c       check that momentum has been conserved
c
        do 10 i=1,4
        if (q(i,1).ne.qin(i,1)) print *,'Comp ',i,' of q not conserved'
   10 continue
c
        return
        end
c
c*************************************************************
c       SUBROUTINE POSTSTATE
c*************************************************************
c
        subroutine poststate(s,inmass)
c
c       calculates the post-collision (output) state from the
c       retransformed state matrix.
c
        common/h/c
        real c(24,4),outstate(24),s(4,24)
c
        do 10 i=1,24
        outstate(i)=0
   10   continue
c
        do 30 i=1,24
```

```
      do 20 j=1,24
      if (s(1,j).ne.c(i,1)) goto 20
      if (s(2,j).ne.c(i,2)) goto 20
      if (s(3,j).ne.c(i,3)) goto 20
      if (s(4,j).ne.c(i,4)) goto 20
      outstate(i)=1
   20 continue
   30 continue
c
c     print *,' '
c     print *,'FINAL OUTPUT STATE'
c     call printt(outstate,1,24)
c
c     check if mass has been conserved
c
      newmass=0
      do 40 i=1,24
      newmass=newmass+outstate(i)
   40 continue
      if (newmass.ne.inmass) print *,'Mass not conserved'
c
      return
      end
c
c*************************************************************
c     SUBROUTINE PRINTT
c*************************************************************
c
      subroutine printt(a,m,n)
c
c     prints desired matrix to screen
c
      real a(m,n)
c
      do 56 i=1,m
         write (5,59)
         do 56 j=1,n
            write(5,60) a(i,j)
   56 continue
      write (5,59)
      return
   59 format()
   60 format(f4.1,'  ',$)
      end
```

APPENDIX B: DETERMINATION OF DEPUTIES

```
c*************************************************************
c     PROGRAM NORMALISER
c*************************************************************
c
      program main
c
c     THIS CODE CALCULATES THE 316301 NORMALISED MOMENTA NEEDED
c     TO DETERMINE THE 29312 DEPUTY INPUT STATES
c     AS DEFINED BY HENON IN HIS PAPER "IMPLEMENTATION OF
c     THE FCHC LG ON THE CONNECTION MACHINE"
c
c     ISOMETRIES ARE GENERATED AS REQUIRED.
c
      integer II(4,4),q(4,1),s(4,24),State(1,24)
      integer depcodes(320000),pqpakno(320000)
      integer code,packnum,p
c
c     set II to the identity matrix
c
      call identity(II)
      n=1
      depcodes(1)=0
      pqpakno(1)=14280
c
c     Go through all possible 16777216 input states
c     The pre-collision state is called: State = ( 0 or 1, 0 or 1, ...)
c     repeated 24 times to indicate presence or absence of
c     a particle with a specific velocity
c
      print *,' '
      print *,'input the number of input states to be processed'
      read *,numstates
c
c     numstates=16777216   Note that the 2**24 possible
c     different states are actually 0 to 16777215
c
      do 100 istate=0,numstates-1
c
c        initialise necessary variables
c
         call makestate(istate,State)
         call getstate(State,s)
         call getpq(State,s,p,q,packnum)
         call chngsign(s,q,II)
         call orderq(q,s,II)
         call sumtest(q,s,II)
         call reorder(s,State)
         call getcode(State,code)
         call getpq(State,s,p,q,packnum)
         call locate(depcodes,pqpakno,n,code,packnum)
         if (mod(istate,100000).eq.0) print *,istate,n
 100  continue
c
      open (unit=9,file='Dnormals',status='append')
      write (9,1)
      write (9,2) numstates
      write (9,3)
      write (9,4) n
      write (9,3)
c
      do 200 i=1,n
         write (9,*) depcodes(i),pqpakno(i)
 200  continue
      close (9)
c
      stop
 1    format ('ARRAY VALUES OF DEPCODES AND PQPAKNO AFTER')
 2    format ('CONSIDERING ',i8,' STATES')
 3    format (' ')
 4    format ('FINAL NUMBER OF DIFFERENT ARRAY ELEMENTS IS ',i6)
      end
c
c*************************************************************
c     PROGRAM ASCENDER (INTEGER VERSION)
c*************************************************************
c
      program main
c
c     THIS CODE DETERMINES THE 29312 DEPUTY INPUT STATES
c     AS DEFINED BY HENON IN HIS PAPER "IMPLEMENTATION OF
c     THE FCHC LG ON THE CONNECTION MACHINE",
c     FROM THE 316301 NORMALISED MOMENTA CALCULATED BY
c     GETNORM.FTN, AND STORED IN FILE 'Dnormals'.
c
c     ISOMETRIES ARE GENERATED AS REQUIRED.
c
      integer II(4,4),q(4,1),s(4,24),State(1,24)
      integer olddpcds(316301),oldpqpk
      integer depcodes(35000),pqpakno(35000)
      integer code,packnum
c
c     set II to the identity matrix
c
      call identity(II)
      n=1
      depcodes(1)=0
      pqpakno(1)=14280
c
c     Go through all of the 316301 input states as stored in file
c    'Dnormals'. These pre-collision states are called:
c     istate. istate is converted to State = ( 0 or 1, 0 or 1,...)
c     repeated 24 times to indicate presence or absence of
c     a particle with a specific velocity
c
      numstates=316301
      open (unit=9,file='Dnormals')
      read (9,*)
      read (9,*)
      read (9,*)
      read (9,*)
      read (9,*)
      do 5 i=1,numstates
         read (9,*)  olddpcds(i),oldpqpk
 5    continue
      close (9)
c
      do 100 icount=1,numstates
         istate=olddpcds(icount)
         call makestate(istate,State)
         call getstate(State,s)
         call getpq(State,s,p,q,packnum)
         call isometries(s,q,II)
         call reorder(s,State)
         call getcode(State,code)
         call getpq(State,s,p,q,packnum)
         call locate(depcodes,pqpakno,n,code,packnum)
         if (mod(icount,10000).eq.0) print *,icount,n
 100  continue
c
      open (unit=10,file='Dascends',status='append')
      write (10,1)
      write (10,2) numstates
      write (10,3)
      write (10,4) n
      write (10,3)
```

```
c
      do 200 i=1,n
         write (10,*) depcodes(i),pqpakno(i)
 200  continue
      close (10)
c
      stop
 1    format ('ARRAY VALUES OF DEPCODES AND PQPAKNO AFTER')
 2    format ('CONSIDERING ',i8,' STATES')
 3    format (' ')
 4    format ('FINAL NUMBER OF DIFFERENT ARRAY ELEMENTS IS ',i6)
      end
c
c
c*************************************************************
c     SUBROUTINE APPLYQ
c*************************************************************
c
      subroutine applyq(a,k,l,b,m,n)
c
c     This subroutine applies the isometry a (k*l but usually 4*4)
c     to the matrix q (which is usually 4*1, but has general
c     dimensions m*n) and returns q after its transformation.
c
      integer SOFAR(4,1),a(4,4),b(4,1)
c
      call matmult(a,4,4,b,4,1,SOFAR)
      call transfer(SOFAR,k,n,b,k,n)
c
      return
      end
c
c*************************************************************
c     SUBROUTINE APPLYS
c*************************************************************
c
      subroutine applys(a,k,l,b,m,n)
c
c     This subroutine applies the isometry a (k*l but usually 4*4)
c     to the matrix s (which is usually 4*24, but has general
c     dimensions m*n) and returns s after its transformation.
c
      integer SOFAR(4,24),a(4,4),b(4,24)
c
      call matmult(a,4,4,b,4,24,SOFAR)
      call transfer(SOFAR,k,n,b,k,n)
c
      return
      end
c
c*************************************************************
c     SUBROUTINE CHNGSIGN
c*************************************************************
c
      subroutine chngsign(s,q,II)
c
c     This subroutine performs the neccessary change of
c     sign to ensure that all momentum components are
c     positive or zero.
c
      integer s(4,24),q(4,1),II(4,4),sn(4,4)
c
      do 30 i=1,4
         if (q(i,1).lt.0) then
            call sighn(II,i,sn)
            call applys(sn,4,4,s,4,24)
            call applyq(sn,4,4,q,4,1)
         end if
 30   continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE GETCODE
c*************************************************************
c
      subroutine getcode(State,code)
c
c     calculates the 24-bit integer code (code) which represents
c     a State (State). Note that State(1) refers to the
c     presence or absence of a particle on the 24th link, and
c     State(24) refers to that for the 1st link.
c
      integer State(1,24)
      integer code
c
      code=0
      do 10 j=24,1,-1
         code=code*2+State(1,j)
 10   continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE GETPQ
c*************************************************************
c
      subroutine getpq(State,s,p,q,packnum)
c
c     calculates mass (p), momentum (q) and a unique packet
c     number (packnum) for an input state (s)
c
      integer State(1,24),s(4,24),p,q(4,1),packnum
c
      q(1,1)=0
      q(2,1)=0
      q(3,1)=0
      q(4,1)=0
      p=0
c
      do 10 j=1,24
         q(1,1)=q(1,1)+s(1,j)
         q(2,1)=q(2,1)+s(2,j)
         q(3,1)=q(3,1)+s(3,j)
         q(4,1)=q(4,1)+s(4,j)
         p=p+State(1,j)
 10   continue
c
      iqa=q(1,1)+6
      iqb=q(2,1)+6
      iqc=q(3,1)+6
      iqd=q(4,1)+6
c
c     check to ensure that this number is unique for p>12
c
      packnum= iqd+iqc*13+iqb*13**2+iqa*13**3+p*13**4
c
      return
      end
c
c*************************************************************
c     SUBROUTINE GETSTATE
c*************************************************************
```

```
c
      subroutine getstate (State,s)
c
c     This subroutine returns, for a given input state (State),
c     the state matrix (s)
c
      integer c(24,5),State(1,24),s(4,24)
c
c     velocity directions ci=(ci1,ci2,ci3,ci4)
c
      data (c( 1,j),j=1,5) /  1,  1,  0,  0,  1/
      data (c( 2,j),j=1,5) /  1, -1,  0,  0,  1/
      data (c( 3,j),j=1,5) / -1,  1,  0,  0,  1/
      data (c( 4,j),j=1,5) / -1, -1,  0,  0,  1/
c
      data (c( 5,j),j=1,5) /  1,  0,  1,  0,  1/
      data (c( 6,j),j=1,5) /  1,  0, -1,  0,  1/
      data (c( 7,j),j=1,5) / -1,  0,  1,  0,  1/
      data (c( 8,j),j=1,5) / -1,  0, -1,  0,  1/
c
      data (c( 9,j),j=1,5) /  1,  0,  0,  1,  1/
      data (c(10,j),j=1,5) /  1,  0,  0, -1,  1/
      data (c(11,j),j=1,5) / -1,  0,  0,  1,  1/
      data (c(12,j),j=1,5) / -1,  0,  0, -1,  1/
c
      data (c(13,j),j=1,5) /  0,  1,  1,  0,  1/
      data (c(14,j),j=1,5) /  0,  1, -1,  0,  1/
      data (c(15,j),j=1,5) /  0, -1,  1,  0,  1/
      data (c(16,j),j=1,5) /  0, -1, -1,  0,  1/
c
      data (c(17,j),j=1,5) /  0,  1,  0,  1,  1/
      data (c(18,j),j=1,5) /  0,  1,  0, -1,  1/
      data (c(19,j),j=1,5) /  0, -1,  0,  1,  1/
      data (c(20,j),j=1,5) /  0, -1,  0, -1,  1/
c
      data (c(21,j),j=1,5) /  0,  0,  1,  1,  1/
      data (c(22,j),j=1,5) /  0,  0,  1, -1,  1/
      data (c(23,j),j=1,5) /  0,  0, -1,  1,  1/
      data (c(24,j),j=1,5) /  0,  0, -1, -1,  1/
c
c     calculate state matrix (s) dependent on c and State
c
      do 22 i=1,24
      do 22 j=1,4
         s(j,i)=State(1,i)*c(25-i,j)
   22 continue
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE IDENTITY
c*****************************************************************
c
      subroutine identity(a)
c
c     produces the four by four identity matrix
c
      integer a(4,4)
      do 10 i=1,4
      do 10 j=1,4
      a(i,j)=0
      if (i.eq.j) a(i,j)=1
   10 continue
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE ISOMETRIES
c*****************************************************************
c
      subroutine isometries(s,q,II)
c
c     This subroutine performs the neccessary application
c     of isometries to increase the code of the current state
c     and reduce the number of different states
c
      integer s(4,24),stest(4,24),q(4,1),qtest(4,1),II(4,4),a(4,4)
      logical apply,sig
c
c     put the state matrix s into matrix stest for testing, and
c     put the momentum vector q into vector qtest for testing.
c
      call transfer(s,4,24,stest,4,24)
      call transfer(q,4, 1,qtest,4, 1)
      apply=.FALSE.
c
c     ISOMETRY 1
c
      call sig1(a)
      sig=.TRUE.
      call testcode(a,s,stest,q,qtest,apply,sig)
      absdiff=abs((qtest(1,1)+qtest(4,1))-(qtest(2,1)+qtest(3,1)))
      if (apply.and.(absdiff.eq.0)) then
         call transfer(stest,4,24,s,4,24)
         call transfer(qtest,4, 1,q,4, 1)
      else
         call transfer(s,4,24,stest,4,24)
         call transfer(q,4, 1,qtest,4, 1)
      endif
c
c     ISOMETRY 2
c
      call permute(II,3,4,a)
      sig=.FALSE.
      call testcode(a,s,stest,q,qtest,apply,sig)
      absdiff=abs(qtest(3,1)-qtest(4,1))
      if (apply.and.(absdiff.eq.0)) then
         call transfer(stest,4,24,s,4,24)
         call transfer(qtest,4, 1,q,4, 1)
      else
         call transfer(s,4,24,stest,4,24)
         call transfer(q,4, 1,qtest,4, 1)
      endif
c
c     ISOMETRY 3
c
      call sig2(a)
      sig=.TRUE.
      call testcode(a,s,stest,q,qtest,apply,sig)
      absdiff=abs((qtest(2,1)+qtest(3,1)+qtest(4,1))-qtest(1,1))
      if (apply.and.(absdiff.eq.0)) then
         call transfer(stest,4,24,s,4,24)
         call transfer(qtest,4, 1,q,4, 1)
      else
         call transfer(s,4,24,stest,4,24)
         call transfer(q,4, 1,qtest,4, 1)
      endif
c
c     ISOMETRY 4
c
      call sighn(II,4,a)
      sig=.FALSE.
      call testcode(a,s,stest,q,qtest,apply,sig)
```

```
      absdiff=abs(qtest(4,1)-0.0)
      if (apply.and.(absdiff.eq.0)) then
         call transfer(stest,4,24,s,4,24)
         call transfer(qtest,4, 1,q,4, 1)
      else
         call transfer(s,4,24,stest,4,24)
         call transfer(q,4, 1,qtest,4, 1)
      endif
c
c     ISOMETRY 5
c
      call permute(II,2,3,a)
      sig=.FALSE.
      call testcode(a,s,stest,q,qtest,apply,sig)
      absdiff=abs(qtest(2,1)-qtest(3,1))
      if (apply.and.(absdiff.eq.0)) then
         call transfer(stest,4,24,s,4,24)
         call transfer(qtest,4, 1,q,4, 1)
      else
         call transfer(s,4,24,stest,4,24)
         call transfer(q,4, 1,qtest,4, 1)
      endif
c
c     ISOMETRY 6
c
      call permute(II,3,4,a)
      sig=.FALSE.
      call testcode(a,s,stest,q,qtest,apply,sig)
      absdiff=abs(qtest(3,1)-qtest(4,1))
      if (apply.and.(absdiff.eq.0)) then
         call transfer(stest,4,24,s,4,24)
         call transfer(qtest,4, 1,q,4, 1)
      else
         call transfer(s,4,24,stest,4,24)
         call transfer(q,4, 1,qtest,4, 1)
      endif
c
c     ISOMETRY 7
c
      call permute(II,1,2,a)
      sig=.FALSE.
      call testcode(a,s,stest,q,qtest,apply,sig)
      absdiff=abs(qtest(1,1)-qtest(2,1))
      if (apply.and.(absdiff.eq.0)) then
         call transfer(stest,4,24,s,4,24)
         call transfer(qtest,4, 1,q,4, 1)
      else
         call transfer(s,4,24,stest,4,24)
         call transfer(q,4, 1,qtest,4, 1)
      endif
c
c     ISOMETRY 8
c
      call sighn(II,3,a)
      sig=.FALSE.
      call testcode(a,s,stest,q,qtest,apply,sig)
      absdiff=abs(qtest(3,1)-0.0)
      if (apply.and.(absdiff.eq.0)) then
         call transfer(stest,4,24,s,4,24)
         call transfer(qtest,4, 1,q,4, 1)
      else
         call transfer(s,4,24,stest,4,24)
         call transfer(q,4, 1,qtest,4, 1)
      endif
c
c     ISOMETRY 9
c
      call sighn(II,4,a)
      sig=.FALSE.
      call testcode(a,s,stest,q,qtest,apply,sig)
      absdiff=abs(qtest(4,1)-0.0)
      if (apply.and.(absdiff.eq.0)) then
         call transfer(stest,4,24,s,4,24)
         call transfer(qtest,4, 1,q,4, 1)
      else
         call transfer(s,4,24,stest,4,24)
         call transfer(q,4, 1,qtest,4, 1)
      endif
c
c     ISOMETRY 10
c
      call permute(II,3,4,a)
      sig=.FALSE.
      call testcode(a,s,stest,q,qtest,apply,sig)
      absdiff=abs(qtest(3,1)-qtest(4,1))
      if (apply.and.(absdiff.eq.0)) then
         call transfer(stest,4,24,s,4,24)
         call transfer(qtest,4, 1,q,4, 1)
      else
         call transfer(s,4,24,stest,4,24)
         call transfer(q,4, 1,qtest,4, 1)
      endif
c
c     ISOMETRY 11
c
      call sig1(a)
      sig=.TRUE.
      call testcode(a,s,stest,q,qtest,apply,sig)
      absdiff=abs((qtest(1,1)+qtest(4,1))-(qtest(2,1)+qtest(3,1)))
      if (apply.and.(absdiff.eq.0)) then
         call transfer(stest,4,24,s,4,24)
         call transfer(qtest,4, 1,q,4, 1)
      else
         call transfer(s,4,24,stest,4,24)
         call transfer(q,4, 1,qtest,4, 1)
      endif
c
c     ISOMETRY 12
c
      call permute(II,2,3,a)
      sig=.FALSE.
      call testcode(a,s,stest,q,qtest,apply,sig)
      absdiff=abs(qtest(2,1)-qtest(3,1))
      if (apply.and.(absdiff.eq.0)) then
         call transfer(stest,4,24,s,4,24)
         call transfer(qtest,4, 1,q,4, 1)
      else
         call transfer(s,4,24,stest,4,24)
         call transfer(q,4, 1,qtest,4, 1)
      endif
c
      return
      end
c
c*************************************************
c            SUBROUTINE LOCATE
c*************************************************
c
      subroutine locate(xx,yy,n,x,y)
c
c     this subroutine, adapted from LOCATE.FOR from
c     NUMERICAL RECIPES, determines whether code already appears
c     in array depcode or not. Id it does not, it inserts code
c     into depcodes at the correct location
c
```

```
      integer xx(n),yy(n),x,y
c
      jl=0
      ju=n+1
c
 10   if (ju-jl.gt.1) then
         jm=(ju+jl)/2
         if (x.gt.xx(jm)) then
            jl=jm
         else
            ju=jm
         endif
         goto 10
      endif
c
      j=jl
c
      if (xx(j).ne.x.and.xx(j+1).ne.x) then
c
c        code not yet in array depcodes, therefore insert it
c        into array depcodes at the correct location, and also
c        insert its corresponding packet number packnum into
c        array pqpakno at the correct location
c
         do 275 ii=n,j+1,-1
           xx(ii+1)=xx(ii)
           yy(ii+1)=yy(ii)
 275     continue
         xx(j+1)=x
         yy(j+1)=y
         n=n+1
      else
c
c        code is in array depcodes already, therefore do nothing
c
      endif
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE MAKESTATE
c*****************************************************************
c
      subroutine makestate(istate,State)
c
c     converts the current input state istate into the form:
c     State = ( 0.0 or 1.0, 0.0 or 1.0, ...) repeated 24 times
c     to indicate presence or absence of a particle with a
c     specific velocity. Note that State(1) refers to the
c     presence or absence of a particle on the 24th link, and
c     State(24) refers to that for the 1st link.
c
      integer State(1,24)
      kistate=istate
c
      do 10 j=1,24
         State(1,j)=mod(kistate,2)
         kistate=kistate/2
 10   continue
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE MATMULT
c*****************************************************************
c
      subroutine matmult(a,k,l,b,m,n,ab)
c
c     multiplies two matrices a and b to form the matrix ab.
c     Note matrix a has k rows and l columns (k*l), while b has m
c     rows and n columns (m*n). This means that ab has k rows
c     and n columns (k*n), and to work, l must equal m.
c
      integer a(k,l),b(m,n),ab(k,n)
      integer h
      if (l.ne.m) print *,'matrix multiplication failed'
c
c     initialise ab(1,j) to zero
c
      do 17 i=1,k
      do 17 j=1,n
      ab(i,j)=0
   17 continue
c
      do 20 i=1,k
      do 20 j=1,n
      do 18 h=1,m
         ab(i,j)=ab(i,j)+a(i,h)*b(h,j)
   18 continue
   20 continue
c
      return
      end
c
c
c*****************************************************************
c     SUBROUTINE ORDERQ
c*****************************************************************
c
      subroutine orderq(q,s,II)
c
c     orders the components of the momentum into non-increasing
c     order
c
      integer q(4,1),s(4,24),II(4,4),SOFAR(4,4)
c
      if (q(1,1).lt.q(2,1)) then
         call permute(II,1,2,SOFAR)
         call applys(SOFAR,4,4,s,4,24)
         call applyq(SOFAR,4,4,q,4,1)
      endif
c
      if (q(3,1).lt.q(4,1)) then
         call permute(II,3,4,SOFAR)
         call applys(SOFAR,4,4,s,4,24)
         call applyq(SOFAR,4,4,q,4,1)
      endif
c
      if (q(1,1).lt.q(3,1)) then
         call permute(II,1,3,SOFAR)
         call applys(SOFAR,4,4,s,4,24)
         call applyq(SOFAR,4,4,q,4,1)
      endif
c
      if (q(2,1).lt.q(4,1)) then
         call permute(II,2,4,SOFAR)
         call applys(SOFAR,4,4,s,4,24)
         call applyq(SOFAR,4,4,q,4,1)
      endif
c
      if (q(2,1).lt.q(3,1)) then
         call permute(II,2,3,SOFAR)
         call applys(SOFAR,4,4,s,4,24)
```

```
        call applyq(SOFAR,4,4,q,4,1)
      endif
c
      return
      end
c
c*************************************************************
c     SUBROUTINE PERMUTE
c*************************************************************
c
      subroutine permute(II,n,m,a)
c
c     produces the 4*4 matrix Pnm as in Henon's Isometric Rules,
c     and stores the Pnm of II in a
c
      integer a(4,4),II(4,4)
c
      do 135 i=1,4
      do 135 j=1,4
      a(i,j)=II(i,j)
  135 continue
c
      do 140 j=1,4
      idum=a(m,j)
      a(m,j)=a(n,j)
      a(n,j)=idum
  140 continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE PRINTT
c*************************************************************
c
      subroutine printt(a,m,n)
c
c     prints desired matrix to screen
c
      integer a(m,n)
c
      do 56 i=1,m
         write (5,59)
         do 56 j=1,n
            write(5,60) a(i,j)
   56 continue
      write (5,59)
      return
   59 format()
   60 format(i2,' ',$)
      end
c
c*************************************************************
c     SUBROUTINE REORDER
c*************************************************************
c
      subroutine reorder(s,State)
c
c     calculates the State from the state matrix s, which
c     has been updated, such that the state accurately
c     represents which velocities are present.
c
      integer s(4,24),State(1,24)
      integer clookup(-1:1,-1:1,-1:1,-1:1,1)
c
      clookup(  0,  0,  0,  0,  1) =   0
c
      clookup(  1,  1,  0,  0,  1) =   1
      clookup(  1, -1,  0,  0,  1) =   2
      clookup( -1,  1,  0,  0,  1) =   3
      clookup( -1, -1,  0,  0,  1) =   4
c
      clookup(  1,  0,  1,  0,  1) =   5
      clookup(  1,  0, -1,  0,  1) =   6
      clookup( -1,  0,  1,  0,  1) =   7
      clookup( -1,  0, -1,  0,  1) =   8
c
      clookup(  1,  0,  0,  1,  1) =   9
      clookup(  1,  0,  0, -1,  1) =   10
      clookup( -1,  0,  0,  1,  1) =   11
      clookup( -1,  0,  0, -1,  1) =   12
c
      clookup(  0,  1,  1,  0,  1) =   13
      clookup(  0,  1, -1,  0,  1) =   14
      clookup(  0, -1,  1,  0,  1) =   15
      clookup(  0, -1, -1,  0,  1) =   16
c
      clookup(  0,  1,  0,  1,  1) =   17
      clookup(  0,  1,  0, -1,  1) =   18
      clookup(  0, -1,  0,  1,  1) =   19
      clookup(  0, -1,  0, -1,  1) =   20
c
      clookup(  0,  0,  1,  1,  1) =   21
      clookup(  0,  0,  1, -1,  1) =   22
      clookup(  0,  0, -1,  1,  1) =   23
      clookup(  0,  0, -1, -1,  1) =   24
c
      do 10 j=1,24
         State(1,j)=0
   10 continue
c
c     look at each column j of the state matrix s:
c
      do 20 j=1,24
         i1=s(1,j)
         i2=s(2,j)
         i3=s(3,j)
         i4=s(4,j)
         index=clookup(i1,i2,i3,i4,1)
         if (index.lt.0.or.index.gt.24) print *,'error-impossible state'
         if (index.ne.0) then
c
c            this implies the presence of a particle on link
c            number index. Since State(1)=1 refers to the
c            presence of a particle on the 24th link, and
c            State(24) refers to that for the 1st link, we
c            must set State(25-index) to 1.
c
            State(1,25-index)=State(1,25-index)+1
            if (State(1,25-index).gt.1) print *,'error - Si>1'
         endif
   20 continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE SIG1
c*************************************************************
c
      subroutine sig1(a)
c
c     generates the matrix sigma1 as per Henon, returning it in a
c     note that the scalar multiplication by 1/2 must be
c     effected after application of this isometry
```

```
c
      integer a(4,4)
c
      do 1000 i=1,4
      do 1000 j=1,4
         a(i,j)=1
         if (i+j.eq.5) a(i,j)=a(i,j)*(-1)
 1000 continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE SIG2
c*************************************************************
c
      subroutine sig2(a)
c
c     generates matrix sigma2 as per Henon and reyurns it in a
c     note that the scalar multiplication by 1/2 must be
c     effected after application of this isometry
c
      integer a(4,4)
c
      do 1000 i=1,4
      do 1000 j=1,4
         a(i,j)=1
 1000 continue
c
      a(2,3)=-1
      a(2,4)=-1
      a(3,2)=-1
      a(3,4)=-1
      a(4,2)=-1
      a(4,3)=-1
c
      return
      end
c
c*************************************************************
c     SUBROUTINE SIGHN
c*************************************************************
c
      subroutine sighn(II,n,a)
c
c     produces the 4*4 matrix a from matrix II as in Henon
c     Isometric Rules, thus storing the Sn of II in a.
c
      integer II(4,4),a(4,4)
c
      do 125 i=1,4
      do 125 j=1,4
      a(i,j)=II(i,j)
  125 continue
c      i=n
c      do 130 j=1,4
c      a(i,j)=(-1)*II(i,j)
c  130 continue
c
      a(n,n)=-1
c
      return
      end
c
c*************************************************************
c     SUBROUTINE SUMTEST
c*************************************************************
c
      subroutine sumtest(q,s)
c
c     tests the sums as per isometries 10 and 11 Henon
c
      integer s(4,24),q(4,1),a(4,4),sum1,sum2
c
c
      sum1=q(1,1)+q(4,1)
      sum2=q(2,1)+q(3,1)
      if (sum1.lt.sum2) then
         call sig1(a)
         call applyq(a,4,4,q,4,1)
         call applys(a,4,4,s,4,24)
         do 10 i=1,4
            q(i,1)=q(i,1)*0.5
            do 10 j=1,24
               s(i,j)=s(i,j)*0.5
 10      continue
      endif
c
c
      sum1=q(2,1)+q(3,1)+q(4,1)
      sum2=q(1,1)
      if (sum1.lt.sum2) then
         call sig2(a)
         call applyq(a,4,4,q,4,1)
         call applys(a,4,4,s,4,24)
         do 20 i=1,4
            q(i,1)=q(i,1)*0.5
            do 20 j=1,24
               s(i,j)=s(i,j)*0.5
 20      continue
      endif
c
c
      return
      end
c
c*********************************************************
c           SUBROUTINE TESTCODE
c*********************************************************
c
      subroutine testcode(a,s,stest,q,qtest,apply,sig)
c
c     This routine determines whether the isometry I=a
c     should be applied or not.
c
      integer a(4,4),s(4,24),stest(4,24),q(4,1),qtest(4,1)
      integer State(1,24),Statetest(1,24)
      integer code,codetest
      logical apply,sig
c
      call applys(a,4,4,stest,4,24)
      call applyq(a,4,4,qtest,4,1)
c
      if (sig) then
         do 10 i=1,4
            qtest(i,1)=qtest(i,1)*0.5
            do 10 j=1,24
               stest(i,j)=stest(i,j)*0.5
 10      continue
      endif
c
      call reorder(stest,Statetest)
      call getcode(Statetest,codetest)
c
      call reorder(s,State)
```

```
      call getcode(State,code)
c
      if (codetest.gt.code) then
         apply=.TRUE.
      else
         apply=.FALSE.
      endif
c
      return
      end
c
c*************************************************************
c     SUBROUTINE TRANSFER
c*************************************************************
c
      subroutine transfer(a,k,l,b,m,n)
c
c     overwrites the contents of matrix b with those of matrix a
c
      integer a(k,l),b(m,n)
      if (k.ne.m.or.l.ne.n) print *,'matrix transfer failed'
      do 20 i=1,k
      do 20 j=1,l
         b(i,j)=a(i,j)
   20 continue
c
      return
      end
```

```
c
c     This subroutine applies the isometry a (k*l but usually 4*4)
c     to the matrix q (which is usually 4*1, but has general
c     dimensions m*n) and returns q after its transformation.
c
      real SOFAR(4,1),a(4,4),b(4,1)
c
      call matmult(a,4,4,b,4,1,SOFAR)
      call transfer(SOFAR,k,n,b,k,n)
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE APPLYS
c*****************************************************************
c
      subroutine applys(a,k,l,b,m,n)
c
c     This subroutine applies the isometry a (k*l but usually 4*4)
c     to the matrix s (which is usually 4*24, but has general
c     dimensions m*n) and returns s after its transformation.
c
      real SOFAR(4,24),a(4,4),b(4,24)
c
      call matmult(a,4,4,b,4,24,SOFAR)
      call transfer(SOFAR,k,n,b,k,n)
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE BACKWARDS
c*****************************************************************
c
      subroutine backwards (s,q)
c
c     applies all the previous isometries to s and q in reverse
c     order - note that the arguments of the call to matmult have
c     been reversed.
c
      common/j/isom,isig,F,II,sig1,sig2
c
      real s(4,24),q(4,1),isom(4,4),II(4,4),sig1(4,4),sig2(4,4)
      real s1(4,4),s2(4,4),s3(4,4),s4(4,4)
      real p12(4,4),p13(4,4),p14(4,4),p23(4,4),p24(4,4),p34(4,4)
c
      call applys(isom,4,4,s,4,24)
      call applyq(isom,4,4,q,4,1)
      if (isig.gt.0) then
         const=(0.5)**(isig)
         do 10 i=1,4
            q(i,1)=q(i,1)*const
            do 9 j=1,24
               s(i,j)=s(i,j)*const
    9       continue
   10    continue
      endif
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE CHECKPQ
c*****************************************************************
c
      subroutine checkpq(p1,p2,q1,q2)
c
c     checks that mass and momentum have been conserved
c
      integer p1,p2
      real q1(4,1),q2(4,1)
c
      if (p1.ne.p2) print *,'error: mass not conserved'
      do 10 i=1,4
         if (q1(i,1).ne.q2(i,1)) print *,'error: q not conserved'
   10 continue
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE CHNGSIGN
c*****************************************************************
c
      subroutine chngsign(State,q)
c
c     This subroutine performs the neccessary change of
c     sign to ensure that all momentum components are
c     positive or zero.
c
      common/j/isom,isig,F,II,sig1,sig2
      common/k/s1,s2,s3,s4,p12,p13,p14,p23,p24,p34
c
      real q(4,1),isom(4,4),State(1,24),SOFAR(4,4)
      real Nstate(1,24),II(4,4),sig1(4,4),sig2(4,4)
      real s1(4,4),s2(4,4),s3(4,4),s4(4,4)
      real p12(4,4),p13(4,4),p14(4,4),p23(4,4),p24(4,4),p34(4,4)
      integer ri,F(24,12)
c
c
      icount=1
      if (q(icount,1).lt.0) then
         ri=icount
         do 10 i=1,24
            Nstate(1,F(i,ri))=State(1,i)
   10    continue
         call transfer(Nstate,1,24,State,1,24)
         q(icount,1)=-1*q(icount,1)
         call applyi(s1,4,4,isom,4,4)
      end if
c
c
      icount=2
      if (q(icount,1).lt.0) then
         ri=icount
         do 20 i=1,24
            Nstate(1,F(i,ri))=State(1,i)
   20    continue
         call transfer(Nstate,1,24,State,1,24)
         q(icount,1)=-1*q(icount,1)
         call applyi(s2,4,4,isom,4,4)
      end if
c
c
      icount=3
      if (q(icount,1).lt.0) then
         ri=icount
         do 30 i=1,24
            Nstate(1,F(i,ri))=State(1,i)
   30    continue
         call transfer(Nstate,1,24,State,1,24)
         q(icount,1)=-1*q(icount,1)
         call applyi(s3,4,4,isom,4,4)
      end if
```

```
c
c
      icount=4
      if (q(icount,1).lt.0) then
         ri=icount
         do 40 i=1,24
            Nstate(1,F(i,ri))=State(1,i)
  40     continue
         call transfer(Nstate,1,24,State,1,24)
         q(icount,1)=-1*q(icount,1)
         call applyi(s4,4,4,isom,4,4)
      end if
c
c
      return
      end
c
c*************************************************************
c     SUBROUTINE GETCODE
c*************************************************************
c
      subroutine getcode(State,code)
c
c     calculates the 24-bit integer code (code) which represents
c     a State (State). Note that State(1) refers to the
c     presence or absence of a particle on the 24th link, and
c     State(24) refers to that for the 1st link.
c
      real State(1,24)
      integer code
c
      code=0
      do 10 j=24,1,-1
         code=code*2+State(1,j)
 10   continue
c
      return
      end
c*************************************************************
c     SUBROUTINE GETF
c*************************************************************
c
      subroutine getf(F)
c
      integer icount,i,ri,F(24,12)
      real s(4,24),II(4,4),a(4,4),sn(4,4)
      real State(1,24),OState(1,24)
c
      do 10 i=1,24
         do 10 ri=1,12
            F(i,ri)=0
 10   continue
c
c     set II to the identity matrix
c
      call identity(II)
c
c     for each different velocity
      do 100 icount=1,24
c        get the state State
         do 50 i=1,24
            State(1,i)=0
  50     continue
         State(1,icount)=1
c        for each different reducing isometry (ri)
c
c        ri=S1
         ri=1
            call getstate(State,s)
            call sighn(II,1,sn)
            call applys(sn,4,4,s,4,24)
            call reorder(s,OState)
            do 1 j=1,24
               if (OState(1,j).eq.1) F(icount,ri)=j
  1         continue
c
c        ri=S2
         ri=2
            call getstate(State,s)
            call sighn(II,2,sn)
            call applys(sn,4,4,s,4,24)
            call reorder(s,OState)
            do 2 j=1,24
               if (OState(1,j).eq.1) F(icount,ri)=j
  2         continue
c
c        ri=S3
         ri=3
            call getstate(State,s)
            call sighn(II,3,sn)
            call applys(sn,4,4,s,4,24)
            call reorder(s,OState)
            do 3 j=1,24
               if (OState(1,j).eq.1) F(icount,ri)=j
  3         continue
c
c        ri=S4
         ri=4
            call getstate(State,s)
            call sighn(II,4,sn)
            call applys(sn,4,4,s,4,24)
            call reorder(s,OState)
            do 4 j=1,24
               if (OState(1,j).eq.1) F(icount,ri)=j
  4         continue
c
c        ri=p12
         ri=5
            call getstate(State,s)
            call permute(II,1,2,a)
            call applys(a,4,4,s,4,24)
            call reorder(s,OState)
            do 12 j=1,24
               if (OState(1,j).eq.1) F(icount,ri)=j
  12        continue
c
c        ri=p13
         ri=6
            call getstate(State,s)
            call permute(II,1,3,a)
            call applys(a,4,4,s,4,24)
            call reorder(s,OState)
            do 13 j=1,24
               if (OState(1,j).eq.1) F(icount,ri)=j
  13        continue
c
c        ri=p14
         ri=7
            call getstate(State,s)
            call permute(II,1,4,a)
            call applys(a,4,4,s,4,24)
            call reorder(s,OState)
            do 14 j=1,24
               if (OState(1,j).eq.1) F(icount,ri)=j
  14        continue
```

```
c
c        ri=p23
         ri=8
            call getstate(State,s)
            call permute(II,2,3,a)
            call applys(a,4,4,s,4,24)
            call reorder(s,OState)
            do 23 j=1,24
               if (OState(1,j).eq.1) F(icount,ri)=j
 23         continue
c
c        ri=p24
         ri=9
            call getstate(State,s)
            call permute(II,2,4,a)
            call applys(a,4,4,s,4,24)
            call reorder(s,OState)
            do 24 j=1,24
               if (OState(1,j).eq.1) F(icount,ri)=j
 24         continue
c
c        ri=p34
         ri=10
            call getstate(State,s)
            call permute(II,3,4,a)
            call applys(a,4,4,s,4,24)
            call reorder(s,OState)
            do 34 j=1,24
               if (OState(1,j).eq.1) F(icount,ri)=j
 34         continue
c
c        ri=sig1
         ri=11
            call getstate(State,s)
            call sigma1(a)
            call applys(a,4,4,s,4,24)
            do 11 i=1,4
               do 9 j=1,24
                  s(i,j)=s(i,j)*0.5
  9            continue
 11         continue
            call reorder(s,OState)
            do 41 j=1,24
               if (OState(1,j).eq.1) F(icount,ri)=j
 41         continue
c
c        ri=sig2
         ri=12
            call getstate(State,s)
            call sigma2(a)
            call applys(a,4,4,s,4,24)
            do 21 i=1,4
               do 19 j=1,24
                  s(i,j)=s(i,j)*0.5
 19            continue
 21         continue
            call reorder(s,OState)
            do 42 j=1,24
               if (OState(1,j).eq.1) F(icount,ri)=j
 42         continue
c
 100  continue
      return
      end
c
c*************************************************************
c     SUBROUTINE GETPQ
c*************************************************************
c
      subroutine getpq(State,p,q)
c
c     calculates mass (p), momentum (q) for an input state (s)
c     Note that this c matrix is the same as in getstate, but
c     that the index is reversed to allow for easy use of matmult.
c
      real State(1,24),q(4,1),c(24,5),out(1,5)
      integer p
c
      data (c(24,j),j=1,5) /  1,  1,  0,  0,  1/
      data (c(23,j),j=1,5) /  1, -1,  0,  0,  1/
      data (c(22,j),j=1,5) / -1,  1,  0,  0,  1/
      data (c(21,j),j=1,5) / -1, -1,  0,  0,  1/
c
      data (c(20,j),j=1,5) /  1,  0,  1,  0,  1/
      data (c(19,j),j=1,5) /  1,  0, -1,  0,  1/
      data (c(18,j),j=1,5) / -1,  0,  1,  0,  1/
      data (c(17,j),j=1,5) / -1,  0, -1,  0,  1/
c
      data (c(16,j),j=1,5) /  1,  0,  0,  1,  1/
      data (c(15,j),j=1,5) /  1,  0,  0, -1,  1/
      data (c(14,j),j=1,5) / -1,  0,  0,  1,  1/
      data (c(13,j),j=1,5) / -1,  0,  0, -1,  1/
c
      data (c(12,j),j=1,5) /  0,  1,  1,  0,  1/
      data (c(11,j),j=1,5) /  0,  1, -1,  0,  1/
      data (c(10,j),j=1,5) /  0, -1,  1,  0,  1/
      data (c( 9,j),j=1,5) /  0, -1, -1,  0,  1/
c
      data (c( 8,j),j=1,5) /  0,  1,  0,  1,  1/
      data (c( 7,j),j=1,5) /  0,  1,  0, -1,  1/
      data (c( 6,j),j=1,5) /  0, -1,  0,  1,  1/
      data (c( 5,j),j=1,5) /  0, -1,  0, -1,  1/
c
      data (c( 4,j),j=1,5) /  0,  0,  1,  1,  1/
      data (c( 3,j),j=1,5) /  0,  0,  1, -1,  1/
      data (c( 2,j),j=1,5) /  0,  0, -1,  1,  1/
      data (c( 1,j),j=1,5) /  0,  0, -1, -1,  1/
c
      call matmult(State,1,24,c,24,5,out)
      q(1,1)=out(1,1)
      q(2,1)=out(1,2)
      q(3,1)=out(1,3)
      q(4,1)=out(1,4)
      p=out(1,5)
c
      return
      end
c
c*************************************************************
c     SUBROUTINE GETSTATE
c*************************************************************
c
      subroutine getstate (State,s)
c
c     This subroutine returns, for a given input state (State),
c     the state matrix (s)
c
      real c(24,5),State(1,24),s(4,24)
c
c     velocity directions ci=(ci1,ci2,ci3,ci4)
c
      data (c( 1,j),j=1,5) /  1,  1,  0,  0,  1/
      data (c( 2,j),j=1,5) /  1, -1,  0,  0,  1/
      data (c( 3,j),j=1,5) / -1,  1,  0,  0,  1/
      data (c( 4,j),j=1,5) / -1, -1,  0,  0,  1/
```

```
c
      data (c( 5,j),j=1,5) /  1,  0,  1,  0,  1/
      data (c( 6,j),j=1,5) /  1,  0, -1,  0,  1/
      data (c( 7,j),j=1,5) / -1,  0,  1,  0,  1/
      data (c( 8,j),j=1,5) / -1,  0, -1,  0,  1/
c
      data (c( 9,j),j=1,5) /  1,  0,  0,  1,  1/
      data (c(10,j),j=1,5) /  1,  0,  0, -1,  1/
      data (c(11,j),j=1,5) / -1,  0,  0,  1,  1/
      data (c(12,j),j=1,5) / -1,  0,  0, -1,  1/
c
      data (c(13,j),j=1,5) /  0,  1,  1,  0,  1/
      data (c(14,j),j=1,5) /  0,  1, -1,  0,  1/
      data (c(15,j),j=1,5) /  0, -1,  1,  0,  1/
      data (c(16,j),j=1,5) /  0, -1, -1,  0,  1/
c
      data (c(17,j),j=1,5) /  0,  1,  0,  1,  1/
      data (c(18,j),j=1,5) /  0,  1,  0, -1,  1/
      data (c(19,j),j=1,5) /  0, -1,  0,  1,  1/
      data (c(20,j),j=1,5) /  0, -1,  0, -1,  1/
c
      data (c(21,j),j=1,5) /  0,  0,  1,  1,  1/
      data (c(22,j),j=1,5) /  0,  0,  1, -1,  1/
      data (c(23,j),j=1,5) /  0,  0, -1,  1,  1/
      data (c(24,j),j=1,5) /  0,  0, -1, -1,  1/
c
c     calculate state matrix (s) dependent on c and State
c
      do 22 i=1,24
      do 22 j=1,4
         s(j,i)=State(1,i)*c(25-i,j)
   22 continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE IDENTITY
c*************************************************************
c
      subroutine identity(a)
c
c     produces the four by four identity matrix
c
      real a(4,4)
      do 10 i=1,4
         do 10 j=1,4
            a(i,j)=0
           if (i.eq.j) a(i,j)=1
   10 continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE INVENTSTATE
c*************************************************************
c
      subroutine inventstate(iseed,istate)
c
c     This subroutine produces a random state number (istate) between
c     0 and 16777215, and modifies the seed (iseed) for further use of
c     the random number generator.
c     Adapted from numerical recipes.
c
      parameter(ia=7141,ic=54773,im=259200)
      n=16777216
      iseed=mod(iseed*ia+ic,im)
      rand=float(iseed)/float(im)
      rand=rand*n
      istate=int(rand)
      return
      end
c
c
c*************************************************************
c     SUBROUTINE ISOMETRIES
c*************************************************************
c
      subroutine isometries(State,q)
c
c     This subroutine performs the neccessary application
c     of isometries to increase the code of the current state
c     and reduce the number of different states
c
      common/j/isom,isig,F,II,sig1,sig2
      common/k/s1,s2,s3,s4,p12,p13,p14,p23,p24,p34
c
      real q(4,1),newq(4,1),State(1,24),Nstate(1,24),out(4,4)
      real isom(4,4),sig1(4,4),sig2(4,4),SOFAR(4,4),II(4,4)
      real s1(4,4),s2(4,4),s3(4,4),s4(4,4)
      real p12(4,4),p13(4,4),p14(4,4),p23(4,4),p24(4,4),p34(4,4)
      integer ri,F(24,12),ncode,code
c
c     put the state matrix s into matrix stest for testing, and
c     put the momentum vector q into vector newq for testing.
c
c     ISOMETRY 1 : SIG1
c
      ri=11
      do 10 i=1,24
         Nstate(1,F(i,ri))=State(1,i)
   10 continue
      call getpq(Nstate,p,newq)
      call getcode(Nstate,ncode)
      call getcode(State,code)
      absdiff=abs((newq(1,1)+newq(4,1))-(newq(2,1)+newq(3,1)))
      if ((ncode.gt.code).and.(absdiff.eq.0)) then
         call transfer(Nstate,1,24,State,1,24)
         call transfer(newq,4, 1,q,4, 1)
         call applyi(sig1,4,4,isom,4,4)
         isig=isig+1
      endif
c
c     ISOMETRY 2 : P34
c
      ri=10
      indx1=3
      indx2=4
      do 20 i=1,24
         Nstate(1,F(i,ri))=State(1,i)
   20 continue
      call getpq(Nstate,p,newq)
      call getcode(Nstate,ncode)
      call getcode(State,code)
      absdiff=abs(newq(indx1,1)-newq(indx2,1))
      if ((ncode.gt.code).and.(absdiff.eq.0)) then
         call transfer(Nstate,1,24,State,1,24)
         call transfer(newq,4, 1,q,4, 1)
         call applyi(p34,4,4,isom,4,4)
      endif
c
c     ISOMETRY 3 : SIG2
c
      ri=12
```

```
      do 30 i=1,24
         Nstate(1,F(i,ri))=State(1,i)
   30 continue
      call getpq(Nstate,p,newq)
      call getcode(Nstate,ncode)
      call getcode(State,code)
      absdiff=abs((newq(2,1)+newq(3,1)+newq(4,1))-newq(1,1))
      if ((ncode.gt.code).and.(absdiff.eq.0)) then
         call transfer(Nstate,1,24,State,1,24)
         call transfer(newq,4, 1,q,4, 1)
         call applyi(sig2,4,4,isom,4,4)
         isig=isig+1
      endif
c
c     ISOMETRY 4 : S4
c
      ri=4
      indx1=4
      do 40 i=1,24
         Nstate(1,F(i,ri))=State(1,i)
   40 continue
      call getpq(Nstate,p,newq)
      call getcode(Nstate,ncode)
      call getcode(State,code)
      absdiff=abs(newq(indx1,1)-0.0)
      if ((ncode.gt.code).and.(absdiff.eq.0)) then
         call transfer(Nstate,1,24,State,1,24)
         call transfer(newq,4, 1,q,4, 1)
         call applyi(s4,4,4,isom,4,4)
      endif
c
c     ISOMETRY 5 : P23
c
      ri=8
      indx1=2
      indx2=3
      do 50 i=1,24
         Nstate(1,F(i,ri))=State(1,i)
   50 continue
      call getpq(Nstate,p,newq)
      call getcode(Nstate,ncode)
      call getcode(State,code)
      absdiff=abs(newq(indx1,1)-newq(indx2,1))
      if ((ncode.gt.code).and.(absdiff.eq.0)) then
         call transfer(Nstate,1,24,State,1,24)
         call transfer(newq,4, 1,q,4, 1)
         call applyi(p23,4,4,isom,4,4)
      endif
c
c     ISOMETRY 6 : P34
c
      ri=10
      indx1=3
      indx2=4
      do 60 i=1,24
         Nstate(1,F(i,ri))=State(1,i)
   60 continue
      call getpq(Nstate,p,newq)
      call getcode(Nstate,ncode)
      call getcode(State,code)
      absdiff=abs(newq(indx1,1)-newq(indx2,1))
      if ((ncode.gt.code).and.(absdiff.eq.0)) then
         call transfer(Nstate,1,24,State,1,24)
         call transfer(newq,4, 1,q,4, 1)
         call applyi(p34,4,4,isom,4,4)
      endif
c
c     ISOMETRY 7 : P12
c
      ri=5
      indx1=1
      indx2=2
      do 70 i=1,24
         Nstate(1,F(i,ri))=State(1,i)
   70 continue
      call getpq(Nstate,p,newq)
      call getcode(Nstate,ncode)
      call getcode(State,code)
      absdiff=abs(newq(indx1,1)-newq(indx2,1))
      if ((ncode.gt.code).and.(absdiff.eq.0)) then
         call transfer(Nstate,1,24,State,1,24)
         call transfer(newq,4, 1,q,4, 1)
         call applyi(p12,4,4,isom,4,4)
      endif
c
c     ISOMETRY 8 : S3
c
      ri=3
      indx1=3
      do 80 i=1,24
         Nstate(1,F(i,ri))=State(1,i)
   80 continue
      call getpq(Nstate,p,newq)
      call getcode(Nstate,ncode)
      call getcode(State,code)
      absdiff=abs(newq(indx1,1)-0.0)
      if ((ncode.gt.code).and.(absdiff.eq.0)) then
         call transfer(Nstate,1,24,State,1,24)
         call transfer(newq,4, 1,q,4, 1)
         call applyi(s3,4,4,isom,4,4)
      endif
c
c     ISOMETRY 9 : S4
c
      ri=4
      indx1=4
      do 90 i=1,24
         Nstate(1,F(i,ri))=State(1,i)
   90 continue
      call getpq(Nstate,p,newq)
      call getcode(Nstate,ncode)
      call getcode(State,code)
      absdiff=abs(newq(indx1,1)-0.0)
      if ((ncode.gt.code).and.(absdiff.eq.0)) then
         call transfer(Nstate,1,24,State,1,24)
         call transfer(newq,4, 1,q,4, 1)
         call applyi(s4,4,4,isom,4,4)
      endif
c
c     ISOMETRY 10 : P34
c
      ri=10
      indx1=3
      indx2=4
      do 100 i=1,24
         Nstate(1,F(i,ri))=State(1,i)
  100 continue
      call getpq(Nstate,p,newq)
      call getcode(Nstate,ncode)
      call getcode(State,code)
      absdiff=abs(newq(indx1,1)-newq(indx2,1))
      if ((ncode.gt.code).and.(absdiff.eq.0)) then
         call transfer(Nstate,1,24,State,1,24)
         call transfer(newq,4, 1,q,4, 1)
         call applyi(p34,4,4,isom,4,4)
```

```
      endif
c
c     ISOMETRY 11 : SIG1
c
      ri=11
      do 110 i=1,24
         Nstate(1,F(i,ri))=State(1,i)
 110  continue
      call getpq(Nstate,p,newq)
      call getcode(Nstate,ncode)
      call getcode(State,code)
      absdiff=abs((newq(1,1)+newq(4,1))-(newq(2,1)+newq(3,1)))
      if ((ncode.gt.code).and.(absdiff.eq.0)) then
         call transfer(Nstate,1,24,State,1,24)
         call transfer(newq,4, 1,q,4, 1)
         call applyi(sig1,4,4,isom,4,4)
         isig=isig+1
      endif
c
c     ISOMETRY 12 : P23
c
      ri=8
      indx1=2
      indx2=3
      do 120 i=1,24
         Nstate(1,F(i,ri))=State(1,i)
 120  continue
      call getpq(Nstate,p,newq)
      call getcode(Nstate,ncode)
      call getcode(State,code)
      absdiff=abs(newq(indx1,1)-newq(indx2,1))
      if ((ncode.gt.code).and.(absdiff.eq.0)) then
         call transfer(Nstate,1,24,State,1,24)
         call transfer(newq,4, 1,q,4, 1)
         call applyi(p23,4,4,isom,4,4)
      endif
c
      return
      end
c
c*************************************************************
c              SUBROUTINE LOOKUP
c*************************************************************
c
      subroutine lookup(xx,yy,n,x,y)
c
c     this subroutine, adapted from LOCATE.FOR from NUMERICAL
c     RECIPES, looks up the deputy output state (y) for a given
c     deputy deputy input state (x), given the sorted arrays
c     containing the deputy input states (xx), and deputy output
c     states (yy). n is the number of deputy input states (29312)
c
      integer xx(n),yy(n),x,y
c
      jl=0
      ju=n+1
c
 10   if (ju-jl.gt.1) then
         jm=(ju+jl)/2
         if (x.gt.xx(jm)) then
            jl=jm
         else
            ju=jm
         endif
         goto 10
      endif
c
      j=jl+1
      y=yy(j)
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE MAKESTATE
c*****************************************************************
c
      subroutine makestate(istate,State)
c
c     converts the current input state istate into the form:
c     State = ( 0.0 or 1.0, 0.0 or 1.0, ...) repeated 24 times
c     to indicate presence or absence of a particle with a
c     specific velocity. Note that State(1) refers to the
c     presence or absence of a particle on the 24th link, and
c     State(24) refers to that for the 1st link.
c
      real State(1,24)
      kistate=istate
c
      do 10 j=1,24
         State(1,j)=mod(kistate,2)
         kistate=kistate/2
 10   continue
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE MATMULT
c*****************************************************************
c
      subroutine matmult(a,k,l,b,m,n,ab)
c
c     multiplies two matrices a and b to form the matrix ab.
c     Note matrix a has k rows and l columns (k*l), while b has m
c     rows and n columns (m*n). This means that ab has k rows
c     and n columns (k*n), and to work, l must equal m.
c
      real a(k,l),b(m,n),ab(k,n)
c
      if (n.eq.24.or.n.eq.4) then
         do 100 irow=1,n,4
            call vec_$mat_mult(a,b(1,irow),ab(1,irow))
 100     continue
      elseif (n.eq.1) then
         call vec_$postmult(a,b,ab)
      else
         call vec_$mat_multn(a,b,k,l,n,ab)
      endif
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE ORDERQ
c*****************************************************************
c
      subroutine orderq(State,q)
c
c     orders the components of the momentum into non-increasing
c     order
c
      common/j/isom,isig,F,II,sig1,sig2
      common/k/s1,s2,s3,s4,p12,p13,p14,p23,p24,p34
c
```

```
      real q(4,1),isom(4,4),State(1,24),Nstate(1,24),SOFAR(4,4)
      real II(4,4),sig(4,4),sig2(4,4)
      real s1(4,4),s2(4,4),s3(4,4),s4(4,4)
      real p12(4,4),p13(4,4),p14(4,4),p23(4,4),p24(4,4),p34(4,4)
      integer ri,F(24,12),indx1,indx2,idum
c
      if (q(1,1).lt.q(2,1)) then
         ri=5
         indx1=1
         indx2=2
         do 20 i=1,24
            Nstate(1,F(i,ri))=State(1,i)
   20    continue
         call transfer(Nstate,1,24,State,1,24)
         idum=q(indx1,1)
         q(indx1,1)=q(indx2,1)
         q(indx2,1)=idum
         call applyi(p12,4,4,isom,4,4)
      endif
c
c
c
      if (q(3,1).lt.q(4,1)) then
         ri=10
         indx1=3
         indx2=4
         do 30 i=1,24
            Nstate(1,F(i,ri))=State(1,i)
   30    continue
         call transfer(Nstate,1,24,State,1,24)
         idum=q(indx1,1)
         q(indx1,1)=q(indx2,1)
         q(indx2,1)=idum
         call applyi(p34,4,4,isom,4,4)
      endif
c
c
c
      if (q(1,1).lt.q(3,1)) then
         ri=6
         indx1=1
         indx2=3
         do 40 i=1,24
            Nstate(1,F(i,ri))=State(1,i)
   40    continue
         call transfer(Nstate,1,24,State,1,24)
         idum=q(indx1,1)
         q(indx1,1)=q(indx2,1)
         q(indx2,1)=idum
         call applyi(p13,4,4,isom,4,4)
      endif
c
c
c
      if (q(2,1).lt.q(4,1)) then
         ri=9
         indx1=2
         indx2=4
         do 50 i=1,24
            Nstate(1,F(i,ri))=State(1,i)
   50    continue
         call transfer(Nstate,1,24,State,1,24)
         idum=q(indx1,1)
         q(indx1,1)=q(indx2,1)
         q(indx2,1)=idum
         call applyi(p24,4,4,isom,4,4)
      endif
c
c
c
      if (q(2,1).lt.q(3,1)) then
         ri=8
         indx1=2
         indx2=3
         do 60 i=1,24
            Nstate(1,F(i,ri))=State(1,i)
   60    continue
         call transfer(Nstate,1,24,State,1,24)
         idum=q(indx1,1)
         q(indx1,1)=q(indx2,1)
         q(indx2,1)=idum
         call applyi(p23,4,4,isom,4,4)
      endif
c
c
c
      return
      end
c
c*************************************************************
c     SUBROUTINE PERMUTE
c*************************************************************
c
      subroutine permute(II,m,n,a)
c
c     produces the 4*4 matrix Pmn as in Henon's Isometric Rules,
c     and stores the Pmn of II in a
c
      real a(4,4),II(4,4)
c
      do 135 i=1,4
         do 135 j=1,4
            a(i,j)=II(i,j)
  135 continue
c
      do 140 j=1,4
         idum=a(m,j)
         a(m,j)=a(n,j)
         a(n,j)=idum
  140 continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE PRINTT
c*************************************************************
c
      subroutine printt(a,m,n)
c
c     prints desired matrix to screen
c
      real a(m,n)
c
      do 56 i=1,m
         write (5,59)
         do 56 j=1,n
            write(5,60) a(i,j)
   56 continue
      write (5,59)
      return
   59 format()
   60 format(f4.1,' ',$)
      end
c
```

```
c*************************************************************
c     SUBROUTINE REORDER
c*************************************************************
c
      subroutine reorder(s,State)
c
c     calculates the State from the state matrix s, which
c     has been updated, such that the state accurately
c     represents which velocities are present.
c
      real s(4,24),State(1,24)
      integer clookup(-1:1,-1:1,-1:1,-1:1,1)
c
      clookup(  0,  0,  0,  0,  1) =   0
c
      clookup(  1,  1,  0,  0,  1) =   1
      clookup(  1, -1,  0,  0,  1) =   2
      clookup( -1,  1,  0,  0,  1) =   3
      clookup( -1, -1,  0,  0,  1) =   4
c
      clookup(  1,  0,  1,  0,  1) =   5
      clookup(  1,  0, -1,  0,  1) =   6
      clookup( -1,  0,  1,  0,  1) =   7
      clookup( -1,  0, -1,  0,  1) =   8
c
      clookup(  1,  0,  0,  1,  1) =   9
      clookup(  1,  0,  0, -1,  1) =   10
      clookup( -1,  0,  0,  1,  1) =   11
      clookup( -1,  0,  0, -1,  1) =   12
c
      clookup(  0,  1,  1,  0,  1) =   13
      clookup(  0,  1, -1,  0,  1) =   14
      clookup(  0, -1,  1,  0,  1) =   15
      clookup(  0, -1, -1,  0,  1) =   16
c
      clookup(  0,  1,  0,  1,  1) =   17
      clookup(  0,  1,  0, -1,  1) =   18
      clookup(  0, -1,  0,  1,  1) =   19
      clookup(  0, -1,  0, -1,  1) =   20
c
      clookup(  0,  0,  1,  1,  1) =   21
      clookup(  0,  0,  1, -1,  1) =   22
      clookup(  0,  0, -1,  1,  1) =   23
      clookup(  0,  0, -1, -1,  1) =   24
c
      do 10 j=1,24
         State(1,j)=0
 10   continue
c
c     look at each column j of the state matrix s:
c
      do 20 j=1,24
         i1=s(1,j)
         i2=s(2,j)
         i3=s(3,j)
         i4=s(4,j)
         index=clookup(i1,i2,i3,i4,1)
         if (index.lt.0.or.index.gt.24) print *,'error-impossible state'
         if (index.ne.0) then
c
c            this implies the presence of a particle on link
c            number index. Since State(1)=1 refers to the
c            presence of a particle on the 24th link, and
c            State(24) refers to that for the 1st link, we
c            must set State(25-index) to 1.
c
            State(1,25-index)=State(1,25-index)+1
            if (State(1,25-index).gt.1) print *,'error - Si>1'
         endif
 20   continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE SIGHN
c*************************************************************
c
      subroutine sighn(II,n,a)
c
c     produces the 4*4 matrix a from matrix II as in Henon
c     Isometric Rules, thus storing the Sn of II in a.
c
      real II(4,4),a(4,4)
c
      do 125 i=1,4
         do 125 j=1,4
            a(i,j)=II(i,j)
 125  continue
c
      a(n,n)=-1
c
      return
      end
c
c*************************************************************
c     SUBROUTINE SIGMA1
c*************************************************************
c
      subroutine sigma1(a)
c
c     generates the matrix sigma1 as per Henon, returning it in a
c     note that the scalar multiplication by 1/2 must be
c     effected after application of this isometry
c
      real a(4,4)
c
      do 1000 i=1,4
      do 1000 j=1,4
         a(i,j)=1
         if (i+j.eq.5) a(i,j)=a(i,j)*(-1)
 1000 continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE SIGMA2
c*************************************************************
c
      subroutine sigma2(a)
c
c     generates matrix sigma2 as per Henon and reyurns it in a
c     note that the scalar multiplication by 1/2 must be
c     effected after application of this isometry
c
      real a(4,4)
c
      do 1000 i=1,4
      do 1000 j=1,4
         a(i,j)=1
 1000 continue
c
      a(2,3)=-1
      a(2,4)=-1
```

```
      a(3,2)=-1
      a(3,4)=-1
      a(4,2)=-1
      a(4,3)=-1
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE SUMTEST
c*****************************************************************
c
      subroutine sumtest(State,q)
c
c     tests the sums as per isometries 10 and 11 Henon
c
      common/j/isom,isig,F,II,sig1,sig2
      common/k/s1,s2,s3,s4,p12,p13,p14,p23,p24,p34
c
      integer sum1,sum2,ri,p,F(24,12)
      real Nstate(1,24),sig1(4,4),sig2(4,4),II(4,4),q(4,1)
      real isom(4,4),State(1,24)
      real s1(4,4),s2(4,4),s3(4,4),s4(4,4)
      real p12(4,4),p13(4,4),p14(4,4),p23(4,4),p24(4,4),p34(4,4)
c
c
c
      sum1=q(1,1)+q(4,1)
      sum2=q(2,1)+q(3,1)
      if (sum1.lt.sum2) then
         ri=11
         do 20 i=1,24
            Nstate(1,F(i,ri))=State(1,i)
  20     continue
         call transfer(Nstate,1,24,State,1,24)
         call getpq(State,p,q)
         call applyi(sig1,4,4,isom,4,4)
         isig=isig+1
      endif
c
c
c
      sum1=q(2,1)+q(3,1)+q(4,1)
      sum2=q(1,1)
      if (sum1.lt.sum2) then
         ri=12
         do 30 i=1,24
            Nstate(1,F(i,ri))=State(1,i)
  30     continue
         call transfer(Nstate,1,24,State,1,24)
         call getpq(State,p,q)
         call applyi(sig2,4,4,isom,4,4)
         isig=isig+1
      endif
c
c
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE TRANSFER
c*****************************************************************
c
      subroutine transfer(a,k,l,b,m,n)
c
c     overwrites the contents of matrix b with those of matrix a
c
      call vec_$copy(a,b,m*n)
c
      return
      end
```

APPENDIX D: OPTIMISATION OF DEPUTY OUTPUT STATES

```
c
c*********************************************************************
c     PROGRAM OPTIMISE COLLISION TABLE KEEPING SDB
c*********************************************************************
c
      program main
c
      parameter(nstates=16777216,numprints=100000)
      parameter(numwrite=1000000,ndeps=29312)
      common/n/numdeps,numstates
      integer present(14280:699744),colltabl(ndeps,3)
      integer s(4,24),state(1,24),cops(1,24),q(4,1)
      integer packnum,istate,p,first,dumstate
      integer optnum(ndeps),choice2(ndeps,40),ch2W1(ndeps,40)
      integer ch2(40),ch2W(40)
      real w1min(ndeps)
      logical swap
c
      numdeps=ndeps
      numstates=nstates
c
      call table1(present,colltabl,w1min)
c
c     initialise arrays
c
      do 40 i=1,numdeps
         optnum(i)=0
         do 30 j=1,40
            choice2(i,j)=0
            ch2W1(i,j)=0
 30      continue
 40   continue
c
      do 100 istate=0,numstates-1
         if (mod(istate,numwrite).eq.0) then
            call prnttbl(istate,colltabl,w1min)
         endif
         if (mod(istate,numprints).eq.0) then
            print *,'NUMBER OF STATES ASESSED: ',istate
         endif
         call makestate(istate,cops)
         call getstate(cops,s)
         call getpq(cops,s,p,q,packnum)
         if (present(packnum).ne.0) then
            call findstart(colltabl,packnum,numdeps,first)
            do 50 i=first,first-1+present(packnum)
               dumstate=colltabl(i,2)
               call makestate(dumstate,State)
               W1p=w1min(i)
               call opt(State,cops,W1p,swap,W1c)
c
c              update the 2nd options arrays
c
               optnum(i)=optnum(i)+1
               choice2(i,optnum(i))=istate
               ch2W1(i,optnum(i))=W1c
c
c              create temporary arrays for sorting purposes
c
               N=optnum(i)
               do 11 j=1,N
                  ch2(j)=choice2(i,j)
                  ch2W(j)=ch2W1(i,j)
 11            continue
c
c              sort into ascending order of W1, ie best to worst
c
               call quicksort(N,ch2,ch2W)
c
c              return sorted values into 2nd option arrays (only top 40)
c
               if (optnum(i).gt.39) then
                  N=39
                  optnum(i)=39
               endif
               do 22 j=1,N
                  choice2(i,j)=ch2(j)
                  ch2W1(i,j)=ch2W(j)
 22            continue
c
c              update the collision table and W1 matrix if necessary
c
               if (swap) then
                  W1p=W1c
                  colltabl(i,3)=istate
                  w1min(i)=W1p
               endif
 50         continue
         endif
 100  continue
c
      call prnttbl(istate,colltabl,w1min)
      call prnopt(choice2,optnum)
c
      stop
      end
c
c*********************************************************************
c     PROGRAM MODIFY COLLISION TABLE TO  KEEP SDB
c*********************************************************************
c
      program main
c
      parameter(ndeps=29312,nstates=16777216)
      common/n/numdeps,numstates
      integer colltabl(ndeps,3)
      real w1min(ndeps)
      integer optnum(ndeps),choice2(ndeps,40),optnumk,used(ndeps)
      integer same(700),notallowed(700),noptsj(700),opt
      integer instate(1,24),outstate(1,24)
      logical ok
      numdeps=ndeps
c
c     read in files colltab, choice and optnum
c
      print *,'reading OPTNUM...'
      open(unit=10,file='OPTNUM')
      do 30 i=1,numdeps
         read (10,*) optnum(i)
 30   continue
      close(10)
      print *,'finished reading OPTNUM...'
c
      print *,'reading CHOICE...'
      open(unit=16,file='CHOICE')
      do 140 i=1,numdeps
         read (16,*) (choice2(i,j),j= 1,10)
         read (16,*) (choice2(i,j),j=11,20)
         read (16,*) (choice2(i,j),j=21,30)
         read (16,*) (choice2(i,j),j=31,40)
 140  continue
      close(16)
      print *,'finished reading CHOICE...'
c
      print *,'reading COLTAB...'
```

```
      open(unit=15,file='COLTAB')
      read (15,*) idum
      do 120 i=1,numdeps
         read (15,*) colltabl(i,1),colltabl(i,2),colltabl(i,3),wlmin(i)
 120  continue
      close(15)
      print *,'finished reading COLTAB...'
c
c     assign first choice of each input state as its output state
c     and write this temporary collison table out. Note that it has
c     output states which are sometimes different to those obtained in
c     Poptcol, but they are equivalent in Wl value, ie the tables
c     should be equally good in terms of R*. This non-SDB table thus
c     replaces the non-SDB table from Poptcol,and the file called COLTAB is
c     overwritten.
c
      do 150 i=1,numdeps
         colltabl(i,3)=choice2(i,1)
         call makestate(colltabl(i,2),instate)
         call makestate(colltabl(i,3),outstate)
         call getWl (instate,outstate,Wl)
         wlmin(i)=Wl
 150  continue
      istate=16777216
      call prnttbl(istate,colltabl,wlmin)
c
c
c     next section of coding to maintain SDB: it chooses first available
c     choice of output state for each input state, such that no output
c     states are repeated
c
c
      i=1
 600  if (i.ge.numdeps) goto 800
      ipq=colltabl(i,1)
      isam=1
      same(isam)=i
      noptsj(isam)=optnum(i)
      j=i+1
 700  jpq=colltabl(j,1)
      if (ipq.eq.jpq) then
         isam=isam+1
         same(isam)=j
         noptsj(isam)=optnum(j)
         j=j+1
         goto 700
      else
         if (isam.gt.1) then
            call quicksort(isam,same,noptsj)
            notallowed(1)=colltabl(same(1),3)
            used(same(1))=1
            numnot=1
            do 550 icount=2,isam
               k=same(icount)
               optnumk=optnum(k)
               ok=.true.
               do 510 m=1,numnot
                  if (colltabl(k,3).eq.notallowed(m)) ok=.false.
 510           continue
               if (ok) then
                  numnot=numnot+1
                  notallowed(numnot)=colltabl(k,3)
                  used(k)=1
                  goto 550
               else
                  opt=1
 500              ok=.true.
                  do 520 m=1,numnot
                     if(choice2(k,opt).eq.notallowed(m))ok=.false.
 520              continue
                  if (ok) then
                     colltabl(k,3)=choice2(k,opt)
                     numnot=numnot+1
                     notallowed(numnot)=colltabl(k,3)
                     used(k)=opt
                     goto 550
                  else
                     opt=opt+1
                     if (opt.gt.optnumk) then
                        print *,'disaster'
                        go to 560
                     endif
                     goto 500
                  endif
               endif
 550        continue
         else
            used(j-1)=1
         endif
 560     i=j
         goto 600
      endif
c
c     calculate updated values for Wl for writing in file
c
 800  do 160 i=1,numdeps
         call makestate(colltabl(i,2),instate)
         call makestate(colltabl(i,3),outstate)
         call getWl (instate,outstate,Wl)
         wlmin(i)=Wl
 160  continue
      call prnttblsdb(numdeps,colltabl,wlmin)
c
c     print out choice number for each output state
c
      max=1
      do 900 i=1,numdeps
         print *,used(i),choice2(i,used(i))
         if (used(i).gt.max) max=used(i)
 900  continue
      print *,max
c
      stop
 1    format(i9)
 2    format(3i9,f13.6)
      end
c
c**********************************************************************
c     PROGRAM TEST COLLISION TABLE
c**********************************************************************
c
      program main
c
      integer assocvarb(320000),sortvarb(320000)
      common/k/assocvarb,sortvarb,pp,N
      common/l/L,R,D,left,right,parpos
      real w
      integer colltabl(29312,3)
      integer Statein(1,24),qin(4,1),Stateout(1,24),qout(4,1)
      integer sin(4,24),sout(4,24)
      integer istatein,istateout,pin,pout,numdeps
      integer packnumin,packnumout
c
      numdeps=29312
      N=numdeps
```

```
c
c     enter name of file containing collision table to be
c     tested. Read in collision table.
c
      open(unit=16,file='COLTABSDB')
      read (16,*) idum
      do 100 i=1,numdeps
         read(16,*) colltabl(i,1),colltabl(i,2),colltabl(i,3),w
 100  continue
      close(16)
c
c     Test that mass and momentum and packet number are
c     all maintained
c
      print *,'ERRORS I.T.O  P AND Q  (IF ANY) WILL BE  REPORTED BELOW'
      do 200 i=1,numdeps
         istatein=colltabl(i,2)
         call makestate(istatein,Statein)
         call getstate(Statein,sin)
         call getpq(Statein,sin,pin,qin,packnumin)
c
         istateout=colltabl(i,3)
         call makestate(istateout,Stateout)
         call getstate(Stateout,sout)
         call getpq(Stateout,sout,pout,qout,packnumout)
c
         if (packnumout.ne.packnumin)      print *,'error1 on ',istatein
         if (packnumout.ne.colltabl(i,1))  print *,'error2 on ',istatein
         if (pout.ne.pin)                  print *,'error3 on ',istatein
         if (qout(1,1).ne.qin(1,1))        print *,'error4 on ',istatein
         if (qout(2,1).ne.qin(2,1))        print *,'error5 on ',istatein
         if (qout(3,1).ne.qin(3,1))        print *,'error6 on ',istatein
         if (qout(4,1).ne.qin(4,1))        print *,'error7 on ',istatein
 200  continue
      print *,'END OF REPORT ON ERRORS ITO MASS AND MOMENTUM'
c
c     Sorts collision table, arranging it in order of
c     increasing output state code.
c     The maximum number of input states with the same output state
c     is then determined. For SDB this should be 1. Any output states
c     occurring more than once are reported, together with the number of
c     times they appear.
c
      do 50 i=1,N
         assocvarb(i)=colltabl(i,2)
         sortvarb(i)=colltabl(i,3)
  50  continue
c
      call sort
c
      maxnum=1
      i=0
 4    i=i+1
      if (i.gt.N) goto 6
      icurrent=sortvarb(i)
      j=i+1
      num=1
 5    if (sortvarb(j).eq.icurrent) then
         num=num+1
         j=j+1
         i=i+1
         goto 5
      else
         IF (NUM.GT.1) PRINT *,SORTVARB(J-1),NUM
         if (num.gt.maxnum) maxnum=num
         go to 4
      endif
 6    continue
      print *,maxnum
c
      stop
      end
c
c*************************************************************
c           SUBROUTINE FINDSTART
c*************************************************************
c
      subroutine findstart(xx,x,n,first)
c
c     this subroutine, adapted from LOCATE.FOR from
c     NUMERICAL RECIPES, determines the location of the first
c     appearance of x in array xx which is of dimension n, and
c     returns this location in first
c
      integer xx(n,3),x,n,first
c
      jl=1
      ju=n+1
c
 10   if (ju-jl.gt.1) then
         jm=(ju+jl)/2
         if (x.gt.xx(jm,1)) then
            jl=jm
         else
            ju=jm
         endif
         goto 10
      endif
c
      do 20 j=jl-1,jl+1
         if(x.eq.xx(j,1)) goto 30
 20   continue
 30   continue
c
 40   if (((j-1).gt.0).and.(xx(j-1,1).eq.xx(j,1))) then
         j=j-1
         goto 40
      endif
c
      first=j
      return
      end
c
c*************************************************************
c     SUBROUTINE GETW1
c*************************************************************
c
      subroutine getW1 (State,cops,W1)
c
c     This subroutine returns the value of W1 for a given input
c     state (State) and candidate output state (cops)
c
      integer c(24,5),State(1,24),cops(1,24),p,p1,p2
      integer a,b,dab
      real po2dab,Yab,Yoab,Yabprod,W1
c
c     velocity directions ci=(ci1,ci2,ci3,ci4)
c
      data (c( 1,j),j=1,5) /  1,  1,  0,  0,  1/
      data (c( 2,j),j=1,5) /  1, -1,  0,  0,  1/
      data (c( 3,j),j=1,5) / -1,  1,  0,  0,  1/
      data (c( 4,j),j=1,5) / -1, -1,  0,  0,  1/
c
      data (c( 5,j),j=1,5) /  1,  0,  1,  0,  1/
      data (c( 6,j),j=1,5) /  1,  0, -1,  0,  1/
```

```
      data (c( 7,j),j=1,5) / -1,  0,  1,  0,  1/
      data (c( 8,j),j=1,5) / -1,  0, -1,  0,  1/
c
      data (c( 9,j),j=1,5) /  1,  0,  0,  1,  1/
      data (c(10,j),j=1,5) /  1,  0,  0, -1,  1/
      data (c(11,j),j=1,5) / -1,  0,  0,  1,  1/
      data (c(12,j),j=1,5) / -1,  0,  0, -1,  1/
c
      data (c(13,j),j=1,5) /  0,  1,  1,  0,  1/
      data (c(14,j),j=1,5) /  0,  1, -1,  0,  1/
      data (c(15,j),j=1,5) /  0, -1,  1,  0,  1/
      data (c(16,j),j=1,5) /  0, -1, -1,  0,  1/
c
      data (c(17,j),j=1,5) /  0,  1,  0,  1,  1/
      data (c(18,j),j=1,5) /  0,  1,  0, -1,  1/
      data (c(19,j),j=1,5) /  0, -1,  0,  1,  1/
      data (c(20,j),j=1,5) /  0, -1,  0, -1,  1/
c
      data (c(21,j),j=1,5) /  0,  0,  1,  1,  1/
      data (c(22,j),j=1,5) /  0,  0,  1, -1,  1/
      data (c(23,j),j=1,5) /  0,  0, -1,  1,  1/
      data (c(24,j),j=1,5) /  0,  0, -1, -1,  1/
c
c     calculate mass (p) of states
c
      p1=0
      p2=0
      do 10 i=1,24
         p1=p1+State(1,i)
         p2=p2+cops(1,i)
 10   continue
      if (p1.ne.p2) print *,'error1 - masses not the same'
      p=p1
c
c     CALCULATE W1 BY LOOPING OVER ALL ALPHA AND BETA
c
      W1=0.0
      do 50 a=1,4
         do 40 b=1,4
c
c           FOR A GIVEN ALPHA-BETA (a-b) COMBINATION, CALCULATE
c
c           (1)  p/2 * delta(alpha-beta)
c
            if (a.eq.b) then
               dab=1
            else
               dab=0
            endif
            po2dab=(p/2.0)*dab
c
c           (2)  Yab
c
            Yab=0.0
            do 20 i=1,24
               Yab=Yab+(State(1,i)*c(25-i,a)*c(25-i,b))
 20         continue
            Yab=Yab-po2dab
c
c           (3)  Y'ab (called Yoab)
c
            Yoab=0.0
            do 30 i=1,24
               Yoab=Yoab+(cops(1,i)*c(25-i,a)*c(25-i,b))
 30         continue
            Yoab=Yoab-po2dab
c
c           (4)  (Yab)(Y'ab)
c
            Yabprod=Yab*Yoab
c
c           W1= sum of all Yabprod
c
            W1=W1+Yabprod
 40      continue
 50   continue
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE GETPQ
c*****************************************************************
c
      subroutine getpq(State,s,p,q,packnum)
c
c     calculates mass (p), momentum (q) and a unique packet
c     number (packnum) for an input state (s)
c
      integer State(1,24),s(4,24),p,q(4,1),packnum
c
      q(1,1)=0
      q(2,1)=0
      q(3,1)=0
      q(4,1)=0
      p=0
c
      do 10 j=1,24
         q(1,1)=q(1,1)+s(1,j)
         q(2,1)=q(2,1)+s(2,j)
         q(3,1)=q(3,1)+s(3,j)
         q(4,1)=q(4,1)+s(4,j)
         p=p+State(1,j)
 10   continue
c
      iqa=q(1,1)+6
      iqb=q(2,1)+6
      iqc=q(3,1)+6
      iqd=q(4,1)+6
c
c     check to ensure that this number is unique for p>12
c
      packnum= iqd+iqc*13+iqb*13**2+iqa*13**3+p*13**4
c
      return
      end
c
c*****************************************************************
c     SUBROUTINE GETSTATE
c*****************************************************************
c
      subroutine getstate (State,s)
c
c     This subroutine returns, for a given input state (State),
c     the state matrix (s)
c
      integer c(24,5),State(1,24),s(4,24)
c
c     velocity directions ci=(ci1,ci2,ci3,ci4)
c
      data (c( 1,j),j=1,5) /  1,  1,  0,  0,  1/
      data (c( 2,j),j=1,5) /  1, -1,  0,  0,  1/
      data (c( 3,j),j=1,5) / -1,  1,  0,  0,  1/
      data (c( 4,j),j=1,5) / -1, -1,  0,  0,  1/
c
```

```
      data (c( 5,j),j=1,5) /  1,  0,  1,  0,  1/
      data (c( 6,j),j=1,5) /  1,  0, -1,  0,  1/
      data (c( 7,j),j=1,5) / -1,  0,  1,  0,  1/
      data (c( 8,j),j=1,5) / -1,  0, -1,  0,  1/
c
      data (c( 9,j),j=1,5) /  1,  0,  0,  1,  1/
      data (c(10,j),j=1,5) /  1,  0,  0, -1,  1/
      data (c(11,j),j=1,5) / -1,  0,  0,  1,  1/
      data (c(12,j),j=1,5) / -1,  0,  0, -1,  1/
c
      data (c(13,j),j=1,5) /  0,  1,  1,  0,  1/
      data (c(14,j),j=1,5) /  0,  1, -1,  0,  1/
      data (c(15,j),j=1,5) /  0, -1,  1,  0,  1/
      data (c(16,j),j=1,5) /  0, -1, -1,  0,  1/
c
      data (c(17,j),j=1,5) /  0,  1,  0,  1,  1/
      data (c(18,j),j=1,5) /  0,  1,  0, -1,  1/
      data (c(19,j),j=1,5) /  0, -1,  0,  1,  1/
      data (c(20,j),j=1,5) /  0, -1,  0, -1,  1/
c
      data (c(21,j),j=1,5) /  0,  0,  1,  1,  1/
      data (c(22,j),j=1,5) /  0,  0,  1, -1,  1/
      data (c(23,j),j=1,5) /  0,  0, -1,  1,  1/
      data (c(24,j),j=1,5) /  0,  0, -1, -1,  1/
c
c     calculate state matrix (s) dependent on c and State
c
      do 22 i=1,24
      do 22 j=1,4
         s(j,i)=State(1,i)*c(25-i,j)
  22  continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE MAKESTATE
c*************************************************************
c
      subroutine makestate(istate,State)
c
c     converts the current input state istate into the form:
c     State = ( 0.0 or 1.0, 0.0 or 1.0, ...) repeated 24 times
c     to indicate presence or absence of a particle with a
c     specific velocity. Note that State(1) refers to the
c     presence or absence of a particle on the 24th link, and
c     State(24) refers to that for the 1st link.
c
      integer State(1,24)
      kistate=istate
c
      do 10 j=1,24
         State(1,j)=mod(kistate,2)
         kistate=kistate/2
 10   continue
c
      return
      end
c
c*********************************************************
c     SUBROUTINE OPTIMISE OUTPUT STATE
c*********************************************************
c
      subroutine opt(State,cops,Wlp,swap,Wlc)
c
c     determines Wl(s,s') for
c         s = The deputy input state (State)
c         s'= The Candidate output state (cops).
c     If Wl(s,s') for the candidate is better (smaller)
c     than the value of Wl for the present output state
c     (pops) called Wlp, then Wlp is replaced with the
c     improved value, and swap is retuned as TRUE
c
      integer State(1,24),cops(1,24)
      real Wlp,Wlc
      logical swap
c
      call getWl(State,cops,Wlc)
      if (Wlc.lt.Wlp) then
         swap=.TRUE.
      else
         swap=.FALSE.
      endif
c
      return
      end
c ************************************************************
c                    SUBROUTINE QUICKSORT
c ************************************************************
c
      subroutine quicksort(num,varb1,varb2)
c
c     note that the array sortvarb is sorted into ascending order,
c     while assocvarb values just move around with them as they
c     are sorted. N is the number of elements to be sorted.
c
      integer varb1(30000),varb2(30000),num
      integer assocvarb(30000),sortvarb(30000),N
      common/k/assocvarb,sortvarb,pp,N
      common/l/L,R,D,left,right,parpos
c
      do 10 i=1,num
         assocvarb(i)=varb1(i)
         sortvarb(i)=varb2(i)
 10   continue
c
      N=num
      call sort
c
      do 20 i=1,num
         varb1(i)=assocvarb(i)
         varb2(i)=sortvarb(i)
 20   continue
c
      return
      end
c
c ************************************************************
c                    SUBROUTINE SORT
c ************************************************************
c
      subroutine sort
      integer assocvarb(30000),sortvarb(30000)
      integer R(35),L(35),D,N,left,right,Llen,Rlen,pp,parpos
      common/k/assocvarb,sortvarb,pp,N
      common/l/L,R,D,left,right,parpos
c
      D=1
      L(D)=1
      R(D)=N
      do while (D.gt.0)
          left=L(D)
          right=R(D)
          D=D-1
          if (left.lt.right) then
```

```
                call qcksrt (left,right)
                parpos=pp
                Llen=pp-left
                Rlen=right-pp
                call setsub(Llen,Rlen)
            end if
        end do
c
        return
        end
c
c ***************************************************************
c                    SUBROUTINE QCKSRT
c ***************************************************************
c
        subroutine qcksrt(left,right)
        common/k/assocvarb,sortvarb,pp,N
        integer pv,pp,R1,L1,dummy,left,right,spv
        integer assocvarb(30000),sortvarb(30000)
c
        pv=sortvarb(left)
        spv=assocvarb(left)
        L1=left+1
        R1=right
c
        do while (L1.le.R1)
c
            do while((sortvarb(L1).le.pv).and.(L1.le.right))
               L1=L1+1
            end do
c
            do while((sortvarb(R1).gt.pv).and.(R1.ge.left+1))
               R1=R1-1
            end do
c
            if (R1.gt.L1) then
                dummy=sortvarb(L1)
                sortvarb(L1)=sortvarb(R1)
                sortvarb(R1)=dummy
                dummy=assocvarb(L1)
                assocvarb(L1)=assocvarb(R1)
                assocvarb(R1)=dummy
            end if
c
        end do
c
        pp=R1
        sortvarb(left)=sortvarb(R1)
        sortvarb(R1)=pv
        assocvarb(left)=assocvarb(R1)
        assocvarb(R1)=spv
c
        return
        end
c
c ***************************************************************
c                    SUBROUTINE SETSUB
c ***************************************************************
c
        subroutine setsub(Llen,Rlen)
        common/1/L,R,D,left,right,parpos
        integer R(35),L(35),D,left,right,Llen,Rlen,parpos
c
        if (Llen.gt.Rlen)
       #   then
              D=D+1
              L(D)=left
              R(D)=parpos-1
              D=D+1
              L(D)=parpos+1
              R(D)=right
           else
              D=D+1
              L(D)=parpos+1
              R(D)=right
              D=D+1
              L(D)=left
              R(D)=parpos-1
        end if
        return
        end
c
c*********************************************************************
c     SUBROUTINE SHIFTUP
c*********************************************************************
c
        subroutine shiftup(choice2,i)
c
        integer choice2(29312,40)
c
c       This subroutine shifts all second choices up if running out of
c       space in the array
c
        do 10 j=1,39
           choice2(i,j)=choice2(i,j+1)
 10     continue
        choice2(i,40)=0
c
        return
        end
c
c*********************************************************
c     SUBROUTINE PRELIMINARY COLLISION TABLE
c*********************************************************
c
        subroutine table1(present,colltabl,wlmin)
c
c       Generates 3 arrays for further use in the program.
c
c       generates a very sparse array called present to
c       quickly determine whether a state is a candidate deputy
c       output state for any of the deputy input states. If
c       present(packetnum) is unequal to zero, then it means that
c       there are that many deputy input states for which the
c       current istate is a candidate output state. These deputy
c       input states are stored in consecutive i locations in array
c       colltabl(i,2).
c
c       generates the preliminary collision table consisting
c       of the 29312 deputy input states and preliminary
c       output states (just the input states themselves).
c       This array also contains the packetnumber.
c       For i=1,29312
c       colltabl(i,1) contains the packetnumber
c       colltabl(i,2) contains the deputy input state
c       colltabl(i,3) contains the current output state
c
c       generates an array containing the current optimal value
c       of W1. The value of W1 calculated using the deputy input
c       state in colltable(i,2), together with its present output
c       state in colltable(i,3), is stored in wlmin(i).
c
c
        common/n/numdeps,numstates
        real wlmin(29312)
```

```
      integer present(14280:699744),colltabl(29312,3)
      integer numdeps,depcode,packnum,numstates
c
      do 10 icount=14280,699744
            present(icount)=0
 10   continue
c
      do 20 icount=1,numdeps
            colltabl(icount,1)=0
            colltabl(icount,2)=0
            colltabl(icount,3)=0
 20   continue
c
      open(unit=11,file='Dsorted')
      do 30 icount=1,numdeps
         read (11,*) depcode,packnum
         present(packnum)=present(packnum)+1
         colltabl(icount,1)=packnum
         colltabl(icount,2)=depcode
         colltabl(icount,3)=depcode
         wlmin(icount)=10000.0
 30   continue
      close(11)
c
      return
      end
c
c*************************************************************
c     SUBROUTINE TRANSFER
c*************************************************************
c
      subroutine transfer(a,k,l,b,m,n)
c
c     overwrites the contents of matrix b with those of matrix a
c
      integer a(k,l),b(m,n)
      if (k.ne.m.or.l.ne.n) print *,'matrix transfer failed'
      do 20 i=1,k
      do 20 j=1,l
         b(i,j)=a(i,j)
   20 continue
c
      return
      end
```

APPENDIX E: CALCULATION OF REYNOLDS COEFFICIENT

```
c*************************************************************
c     PROGRAM COUNTER
c*************************************************************
c
      program main
c
c     THIS CODE TAKES ALL POSSIBLE INPUT STATES
c     (1) TRANSFORMS IT BY MEANS OF MOMENTUM NORMALISATION
c         AND NORMALISED ASCENT.
c     (2) THUS DETERMINES THE DEPUTY INPUT STATE
c     (3) COUNTS THE NUMBER OF ORIGINAL STATES WHICH REDUCE TO
c         EACH DEPUTY INPUT STATE
c     (4) PRINTS OUT THE TOTAL COUNT PER DEPUTY
c
c     ALL SPEEDUPS AS IN COLLIDER HAVE BEEN USED
c
      common/j/isom,isig,F,II,sig1,sig2
      common/k/s1,s2,s3,s4,p12,p13,p14,p23,p24,p34
c
      real II(4,4),qdep(4,1)
      real s(4,24),q(4,1),qin(4,1),State(1,24)
      real isom(4,4),sig1(4,4),sig2(4,4)
      real s1(4,4),s2(4,4),s3(4,4),s4(4,4)
      real p12(4,4),p13(4,4),p14(4,4),p23(4,4),p24(4,4),p34(4,4)
      integer icount,numstates,istate,p,pin,index,numdeps,pdep
      integer depinputs(29312),depindex(29312),F(24,12)
      integer redcount(29312),depinstate
c
c     set II to the identity matrix, and produce all isometries
c     for later use
c
      call identity(II)
      call sighn(II,1,s1)
      call sighn(II,2,s2)
      call sighn(II,3,s3)
      call sighn(II,4,s4)
      call permute(II,1,2,p12)
      call permute(II,1,3,p13)
      call permute(II,1,4,p14)
      call permute(II,2,3,p23)
      call permute(II,2,4,p24)
      call permute(II,3,4,p34)
      call sigma1(sig1)
      call sigma2(sig2)
      iseed=13989
      numdeps=29312
c
c     get array of isometry effects to lookup state updates
c
      call getf(F)
c
c     read reduced collision table from file COLLTABL
c
      open (unit=17,file='COLLTABL')
      do 10 i=1,numdeps
         read (17,*) depinputs(i),idummy
         depindex(i)=i
         redcount(i)=0
 10   continue
      close(17)
c
      print *,' '
      print *,'For how many input states do you require output states?'
      read *,numstates
c
c     Go through numstates input states 0 to numstates-1
c     The pre-collision state is called: State = ( 0 or 1,  0 or 1, ...)
c     repeated 24 times to indicate presence or absence of
c     a particle with a specific velocity
c
      do 100 istate=0,numstates-1
         if (mod(istate,100000).eq.0) then
            print *,'NUMBER OF STATES ASESSED: ',istate
         endif
         call transfer(II,4,4,isom,4,4)
         isig=0
         call makestate(istate,State)
         call getpq(State,p,q)
         call chngsign(State,q)
         call orderq(State,q)
         call sumtest(State,q)
         call isometries(State,q)
         call getcode(State,depinstate)
         call lookup(depinputs,depindex,numdeps,depinstate,index)
         redcount(index)=redcount(index)+1
 100  continue
c
      itotal=0
      open(unit=19,file='COUNTED',status='overwrite')
      do 110 i=1,numdeps
         write(19,*) i,redcount(i)
         itotal=itotal+redcount(i)
 110  continue
      write(19,*) itotal
      close(19)
c
      stop
      end
c
c*************************************************************
c     PROGRAM GETR*
c*************************************************************
c
      program main
c
c     THIS CODE TAKES READS IN THE ARRAY OF NUMBERS OF ORIGINAL
c     STATES WHICH REDUCE TO EACH DEPUTY INPUT STATE
c
c     THEN, FOR EACH DEPUTY INPUT STATE, DEPUTY OUTPUT STATE PAIR
c     (1) DETERMINES P AND W
c     (2) ADDS CONTRIBUTION OF THIS PAIR, MULTIPLIED BY
c         NUMBER OF STATES WHICH USE THIS COLLISION PAIR, TO MU4
c     HENCE CALCULATES MU4
c
c     THEN CALCULATES R*, ASSUMING OPTIMAL d=1/3
c
      integer icount,istate,ostate,numdeps,p
      integer depinputs(29312),depoutputs(29312)
      integer redcount(29312),distate(1,24),dostate(1,24)
      real mu4,contrib,visc,cs,g,Rstar
      real twothrd,onesxth,ovroot2,d
c
      twothrd=2.0/3.0
      onesxth=1.0/6.0
      ovroot2=1.0/(2.0**(0.5))
      numdeps=29312
c
c     read number of states which reduce to each deputy
c
      open(unit=19,file='COUNTED')
      do 10 i=1,numdeps
         read (19,*) l,redcount(i)
 10   continue
      close(19)
c
```

```
c     read deputy input and output states from file <filename>
c
      open(unit=17,file='COLTABSDB')
      read (17,*) idum
      do 20 i=1,numdeps
         read (17,*) idum,depinputs(i),depoutputs(i),dum
 20   continue
      close(17)
c
      do 200 id=4,12
         rd=id
         d=rd/24.0
         mu4=0.0
         do 100 icount=1,numdeps
            istate=depinputs(icount)
            ostate=depoutputs(icount)
            call makestate(istate,distate)
            call makestate(ostate,dostate)
            call getW(distate,dostate,W,p)
            contrib=(d**(p-1.0))*((1.0-d)**(23.0-p))*W
            mu4=mu4+redcount(icount)*contrib
 100     continue
c
         mu4=mu4/288.0
         visc=onesxth*(mu4/(1.0-mu4))
         g=twothrd*(1.0-2.0*d)/(1.0-d)
         cs=ovroot2
         Rstar=cs*g/visc
         print *,'For 24d= ',rd,' R*= ',Rstar
 200  continue
c
      stop
      end
c
c*************************************************************
c     SUBROUTINE GETW
c*************************************************************
c
      subroutine getW (State,cops,W,p)
c
c     This subroutine returns the value of W for a given input
c     state (State) and output state (cops)
c
      integer c(24,5),State(1,24),cops(1,24),p,p1,p2
      integer a,b,dab
      real po2dab,Yab,Yoab,W
c
c     velocity directions ci=(ci1,ci2,ci3,ci4)
c
      data (c( 1,j),j=1,5) /  1,  1,  0,  0,  1/
      data (c( 2,j),j=1,5) /  1, -1,  0,  0,  1/
      data (c( 3,j),j=1,5) / -1,  1,  0,  0,  1/
      data (c( 4,j),j=1,5) / -1, -1,  0,  0,  1/
c
      data (c( 5,j),j=1,5) /  1,  0,  1,  0,  1/
      data (c( 6,j),j=1,5) /  1,  0, -1,  0,  1/
      data (c( 7,j),j=1,5) / -1,  0,  1,  0,  1/
      data (c( 8,j),j=1,5) / -1,  0, -1,  0,  1/
c
      data (c( 9,j),j=1,5) /  1,  0,  0,  1,  1/
      data (c(10,j),j=1,5) /  1,  0,  0, -1,  1/
      data (c(11,j),j=1,5) / -1,  0,  0,  1,  1/
      data (c(12,j),j=1,5) / -1,  0,  0, -1,  1/
c
      data (c(13,j),j=1,5) /  0,  1,  1,  0,  1/
      data (c(14,j),j=1,5) /  0,  1, -1,  0,  1/
      data (c(15,j),j=1,5) /  0, -1,  1,  0,  1/
      data (c(16,j),j=1,5) /  0, -1, -1,  0,  1/
c
      data (c(17,j),j=1,5) /  0,  1,  0,  1,  1/
      data (c(18,j),j=1,5) /  0,  1,  0, -1,  1/
      data (c(19,j),j=1,5) /  0, -1,  0,  1,  1/
      data (c(20,j),j=1,5) /  0, -1,  0, -1,  1/
c
      data (c(21,j),j=1,5) /  0,  0,  1,  1,  1/
      data (c(22,j),j=1,5) /  0,  0,  1, -1,  1/
      data (c(23,j),j=1,5) /  0,  0, -1,  1,  1/
      data (c(24,j),j=1,5) /  0,  0, -1, -1,  1/
c
c     calculate mass (p) of states
c
      p1=0
      p2=0
      do 10 i=1,24
         p1=p1+State(1,i)
         p2=p2+cops(1,i)
  10  continue
      if (p1.ne.p2) print *,'error1 - masses not the same'
      p=p1
c
c     CALCULATE W BY LOOPING OVER ALL ALPHA AND BETA
c
      W=0.0
      do 50 a=1,4
         do 40 b=1,4
c
c           FOR A GIVEN ALPHA-BETA (a-b) COMBINATION, CALCULATE
c
c           (1)  p/2 * delta(alpha-beta)
c
            if (a.eq.b) then
               dab=1
            else
               dab=0
            endif
            po2dab=(p/2.0)*dab
c
c           (2)  Yab
c
            Yab=0.0
            do 20 i=1,24
               Yab=Yab+(State(1,i)*c(25-i,a)*c(25-i,b))
 20         continue
            Yab=Yab-po2dab
c
c           (3)  Y'ab (called Yoab)
c
            Yoab=0.0
            do 30 i=1,24
               Yoab=Yoab+(cops(1,i)*c(25-i,a)*c(25-i,b))
 30         continue
            Yoab=Yoab-po2dab
c
c           W = sum of all (Yab+Yoab)**2
c
            W=W+(Yab+Yoab)*(Yab+Yoab)
 40      continue
 50   continue
c
      return
      end
c
c*************************************************************
c     SUBROUTINE MAKESTATE
c*************************************************************
```

```
c
      subroutine makestate(istate,State)
c
c     converts the current input state istate into the form:
c     State = ( 0.0 or 1.0, 0.0 or 1.0, ...) repeated 24 times
c     to indicate presence or absence of a particle with a
c     specific velocity. Note that State(1) refers to the
c     presence or absence of a particle on the 24th link, and
c     State(24) refers to that for the 1st link.
c
      integer State(1,24)
      kistate=istate
c
      do 10 j=1,24
         State(1,j)=mod(kistate,2)
         kistate=kistate/2
 10   continue
c
      return
      end
```