SIMULATION OF AN INTERACTING SYSTEM USING A CELLULAR AUTOMATON

by

PETER ANDREW DONIS

SUBMITTED TO THE DEPARTMENT OF NUCLEAR ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1987

Signature of Author ......[illegible]..
Department of Nuclear Engineering
May 27, 1987

Certified by .... [illegible] ....
Kim Molvig
Thesis Supervisor

Accepted by ....[illegible]..
John E. Meyer
Chairman, Department Committee

SIMULATION OF AN INTERACTING SYSTEM USING A CELLULAR AUTOMATON

by

PETER ANDREW DONIS

Submitted to the Department of Nuclear Engineering on May 27, 1987 in partial fulfillment of the requirements for the Degree of Bachelor of Science in Nuclear Engineering

ABSTRACT

A cellular automaton is constructed to allow modeling of temperature equilibration and particle interactions producing energy exchange on a two-dimensional lattice, using two types of particles with different energy and momentum, and photons to mediate the interaction between the two particle types. The cellular automaton is constructed out of the possible basic processes involving these three classes of particles and their possible states at any given lattice point, the number of which was small. The exclusion principle is used, which stipulates that no two particles of the same type can occupy the same state at a given time. Out of these basic elements, an update rule is constructed which gives a unique output state for any input state at each lattice point.

The update rule is then put into Boolean form and coded into FORTRAN for implementation on the MicroVAX. The 32-bit processor allowed the treatment of 32 sites at a time, which increased the speed of the code. The initial state of the lattice is determined from the desired average particle densities. After a large number of time steps, the final state is then given to subroutines which output the results in a usable form. These results are then compared with the expected outcome based on physical considerations.

Thesis Supervisor: Dr. Kim Molvig

Title: Professor of Nuclear Engineering

I: Introduction

Simulations using cellular automata have only recently become objects of interest to physicists, even though the concept of a CA is by no means new. The first CA's, such as the Game of Life, did not pretend to mirror any large-scale physical processes. More recently, CA models to simulate basic fluid mechanics have been devised, but these are sharply limited by the types of states that they can deal with. All previous CA's have dealt with one particle type only, which can occupy a fixed number of states at each lattice site; this means that the macroscopic phenomena which they can simulate are limited.

One macroscopic variable in particular which cannot be simulated by such a system is temperature. The intention of the CA discussed in this thesis is to provide a framework for modeling temperature; to do so, the major change from previous work will be to use two particle "types" instead of one, with different energies and momenta, along with "photons" as the mediating particles of an interaction between the two types. Each type of particle will be able to occupy a fixed number of states at each lattice site. With such a scheme, one can see that the average energy per particle of the system, which can be called the "temperature", can be varied by varying the relative quantities of the two types of particles, whereas in former schemes, with only one type of particle, the concept of temperature was meaningless.

With a CA that can simulate temperature changes, it becomes possible to model a wide variety of processes that involve changes in temperature, rather than be restricted to "isothermal" processes. In particular, the CA discussed here will be the prototype for a CA which can model a gas-to-liquid phase transition.

The objective of this thesis, then, is to construct the basic CA which will serve as a framework for modeling temperature and thermal processes. This involves developing a description of the update rule which can be coded into FORTRAN, and additional sections of code which provide an easy way of testing the rule for physical validity.

II: Physical Background and CA Construction

As mentioned above, the CA used in this thesis uses two types of particles, which will be referred to as type I and type II. Interactions between the two particle types are mediated by particles of zero mass which we call photons, although the interaction they mediate is not necessarily electromagnetic; the only crucial point about this interaction is that it provides a mechanism for energy exchange between type I and type II particles. Both types of particles have unit mass, but they are distinguished by their momentum and energy, which are determined as follows.

The lattice on which these particles move is a two-dimensional rectangular grid, which can be of any size as far as physics is concerned, but whose actual dimensions will be constrained by the methods for storing bits in the computer's memory. Type I particles and photons move in the normal rectangular directions, parallel and perpendicular to the walls, while type II particles move "diagonally". The momentum of particles is then taken to be proportional to the length of the segments along which they move, so that type I particles and photons have unit momentum, while type II particles have momentum of 2. Since energy is proportional to the square of momentum, we then find that type I particles and photons have unit energy, while type II particles have energy of 2. We choose all of our units so that the numerical values of mass, momentum, and energy come out as stated above, i.e., all necessary normalization factors are included in the system of units we use, so that we can forget about them in subsequent analysis.

An interesting note here concerns the photons' momentum. For the model here described, the sign of photon momentum is positive; however, there is nothing physically paradoxical about choosing the photons' momentum to be negative. This does not affect their energy, but it does make a difference when particle interactions are discussed. When the CA is modified for simulating a phase transition, giving the photons negative momentum may well prove to be essential.

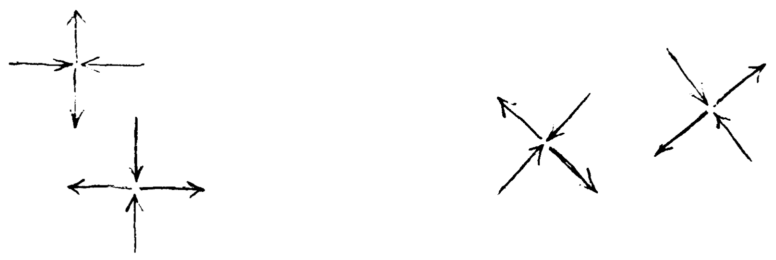
To return to our discussion, the various ways in which these particles may interact must now be considered. In the Game of Life, this problem was simple, since each site had only two possible states, on and off. All one needed to update the lattice from one time step to the next was a rule defining the conditions for a cell being on; otherwise, it would be off. In the CA's used to model fluid mechanics, the update rule was a little more involved, since "collisions" between particles moving in different directions at a given site had to be considered, so that the map from input states to output states had an added degree of complexity.

For our CA, not only collisions between particles of the same type, but also interactions which change a type I particle into a type II, and vice versa, involving photons, must be considered. Unlike the Game of Life, the CA discussed here will have to adhere to the constraints of physical laws which operate in the real world, which sets certain limits on the allowed processes, and therefore on the update rule. For present purposes, the most important of these physical laws are the conservation laws of mass, momentum, and energy, all of which must be obeyed microscopically. That is, at each lattice site, the mapping from input to output states must conserve mass, momentum, and energy.

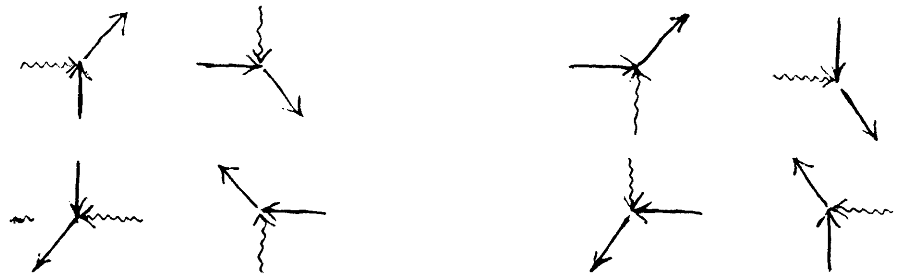
A comment on ways of viewing a CA's update rule seems appropriate here. From a purely mechanical point of view, a CA is nothing more, as has been stated, than a mapping from input states to output states, something like a logical truth table. In the Game of Life, this view seems most appropriate, since the update rule is thoroughly arbitrary and is not meant to mirror anything "real". However, in our CA, actual physical processes are being modeled, so that, from a physical point of view, the update rule is deciding what processes are taking place at a given lattice site, and determining their results. When trying to understand why a certain update rule has been chosen, this view is the only one that makes any sense.

It should not take long to convince oneself that the number of possible processes involving all these particle types, given the conservation constraints, is rather small. The processes which are used in the CA are diagrammed for greater clarity in Fig.1. Self-collisions can occur between two particles of type I or two particles of type II, provided they are moving in opposite directions, so that their momenta cancel. They will then "scatter", producing two particles of the same type moving along the two perpendicular directions. Type-II production processes can also occur when a type I particle and a photon meet at right angles; they will combine to

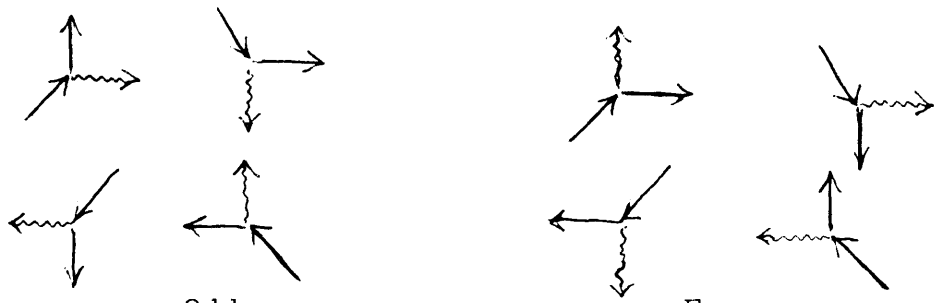
Self-Collisions:



Productions:



Decays:



(Particle with mass) ⟶

(Photon) ⇝

Fig. 1: Diagrams of Possible Processes

produce a type II particle moving in a direction halfway between their original directions. A type II particle, in the reverse of this process, can then decay into a type I and a photon moving at right angles. The production and decay processes both possess a definite parity, meaning that they can occur in two ways which are mirror images of each other and cannot be rotated to coincide. Self-collisions do not possess a definite parity. For the purposes of our CA, we do not consider self-collisions between photons; since they are meant to function only as exchange particles, we are interested only in their interactions with type I and II particles, not with their self-interactions. Also, we do not consider "scattering" interactions between photons and type I particles, even though they can be constructed to conserve the relevent quantities. Other possible processes are allowed, however, and it should not take long to convince oneself that no other processes which obey the conservation laws are possible.

A notation to express the different processes physically is now required. There are eight different direction vectors on the lattice; we number these starting with 1 for the direction "up"--vertically upward--and continue clockwise from there, so that "up and to the right" is 2, "to the right" is 3, and so on. Thus, odd-numbered directions are occupied by type I particles and photons, while even-numbered directions are occupied by type II particles. Next, we symbolize a particle moving in a given direction by the letter n with three indices, as follows:

$$n_{ij}^{t}$$

where i indicates particle type (1, 2, or p), j indicates direction (1 through 8) and t indicates the time step at which the particle occupies the state. For each lattice site, then, at each time step there are twelve possible states which may be occupied or not; each state is represented by a bit, so that twelve bits are required to represent each lattice site, rather than the one bit needed in the Game of Life. Our objective is then to construct an update rule that will give each nij(t+1) "going out" of a lattice site as a function of the nijt's "coming in".

The next task is to construct "operators" for each process at a given lattice site. All nij's are assumed to be "coming in" to the lattice site at time step t. Our operator notation also has three indices: the i and j are the same as for the n's above, but the third index indicates the type of process involved (s for self-collision, p for production, d for decay). The operator letter itself denotes creation (C) or annihilation (A). We then obtain operators such as the following:

$$Cijs = ni(j+2)ni(j-2)\underline{n}ij\underline{n}i(j+4) \quad (1a)$$

$$A1jp = n1jnp(j+2)\underline{n}2(j+1) \quad (1b)$$

where $\underline{n}$ is the negation of n, i.e., it denotes the absence of a particle in that state at time step t. The first operator, then, simply says that a self-collision producing a particle in direction j requires the presence of particles in directions j+2 and j-2 and the absence of particles in directions j and j+4. All sums are modulo 8, so that 7+2, for example, equals 1. The second operator says that a production process which destroys a type I particle in direction j requires that particle to collide with a photon in direction j+2, and that there be no particle of type II in direction j+1.

It is immediately obvious that there are relations between C and A operators, since each process both creates and destroys particles. For example, the two operators defined above could equally well have been "named" as follows:

$$Cijs = Ci(j+4)s = Ai(j+2)s = Ai(j+6)s \quad (2a)$$

$$A1jp = Ap(j+2)p = C2(j+1)p. \quad (2b)$$

Furthermore, there are certain pairs of operators which, though defined differently, share a common name: thus, there is another A1jp defined as follows:

$$A1jp = n1jnp(j-2)\underline{n}2(j-1) = Ap(j-2) = Cp(j-1) \quad (3)$$

Each of the names of this operator, and in fact of all production and decay operators, is shared with another production or decay operator. This arises because of the parity phenomenon mentioned above.

This becomes important when we consider what to do if two processes are possible at a given lattice site. Some pairs of processes will not "interfere" with each other, but some will, and we need a method of determining when this happens. The foolproof method is simply to list each operator, with all its possible names; then, to find out which other processes conflict with it, look down the list for any other operator names which differ from its names only in the process index. That is, to find out which processes conflict with Cijs, we need to find all other Cij's, Ci(j+4)'s, Ai(j+2)'s, and Ai(j+6)'s, no matter what their process index. Since the list of operators is finite, and not terribly long, this is a feasible process, and it was actually employed to compile a list, for each process, of all conflicting processes, which was then used to construct the update rule.

How the list is used in the update rule is not immediately clear, however. At present, the system here defined is "microscopically indeterminate", which simply means that one could not run the update rule backwards from a given end state and always reach the same starting state. It is obvious that the existence of any "conflicting" processes at a given time step will give rise to such indeterminacy, since there will then be two possible outcomes based on one of the two processes happening and the other not. Furthermore, however, the existence of parity means not only that the same input state can give rise to two different output states, but that the same output state could have been caused by two different input states. How are we to deal with this problem?

In the real world, which is quantum-mechanical microscopically, if two or more different output states are possible from a given input state, nature chooses randomly between them with a certain probability attached to each; similarly, if two or more different input states can produce the same output state, we will find that that output state arose from each possible input state with a certain probability over a large number of occurrences. This is possible to model; however, what we are dealing with is then, strictly speaking, no longer a CA, since it is probabilistic and not determinate. It might be called a "probabilistic CA" or a "microscopically indeterminate quasi-CA."

For our purposes, however, we do not need to model quantum effects directly, since they statistically average out when considering a model of something like temperature. We can, therefore, use any method which gives the same statistical results, even if microscopically it is unphysical, in order to model temperature correctly. This permits us to use methods which are much faster when implemented on a computer than a random number generator, which would be used in a strict quantum model.

It is thus expedient to adopt two conventions in defining the update rule for our CA. The first is to add an additional index to production and decay processes, making them po, pe, do, and de processes ("odd" or "even" parity). Thus, the two production operators defined above no longer share the same name, since the first is Aijpo and the second is Aijpe. The second is to index the time steps; most simply as odd or even, but also as "decay" or "non-decay", since we want to be able to vary the frequency of decay processes, not necessarily having them occur every time step (since they, unlike the other processes, have only one particle as input, they are not necessarily treated the same). Finally, in addition to these conventions, we assume that multiple processes at the same lattice site are rare enough that we can ignore them; thus, whenever an input state has two or more possible processes, we say that nothing happens, i.e., that the output state is the same as the input state (this is the same as simple propagation of particles). Since we will be dealing with low particle densities, this assumption is reasonable. We have therefore adopted the simple expedient of dealing with quantum effects by ignoring them.

Combining these three elements, the final physical description of the update rule is arrived at. On odd time steps, we only let odd processes happen, and on even time steps, we only let even processes happen; this eliminates the indeterminacy arising from parity. Decays only happen on decay steps. Whenever two processes "conflict" at a given lattice site, nothing happens at that site for that time step: every particle simply propagates.

Our list of conflicting processes is therefore useful in that, to complete the conditions for each process happening, we need only multiply its operator by the negation of all conflicting operators, signifying that only that process is possible. We represent any conflicting operator to either Aij or Cij by the notation Okl, where k is the particle type and l is the direction; O is then either C or A, as required for each conflicting operator name. The conditions for Aij or Cij actually happening, then, are given by

$$Aij\prod\underline{O}kl$$

$$Cij\prod\underline{O}kl$$

where the $\prod$ notation represents the product of all relevant terms, rather than the sum represented by the $\sum$ notation, and $\underline{O}$ is the negation of O.

We are now in a position to write the update rule in its physical form. We want to consider all possible processes which will give nij(t+1). There are two general types of processes. The first is propagation of nijt, which will happen provided that no Aij processes occur; we express this by the notation linking nij and all Aij's. The second is one or another of the Cij processes occurring, and we express this by summing over all the Cij terms. We must include the negation of all conflicting processes, in the notation above, for both the Aij and Cij processes, and we therefore arrive at:

$$nij(t+1) = nijt\pi not(Aij\pi \underline{Q}kl) + \Sigma Cij\pi \underline{Q}kl. \quad (4)$$

This is a physics equation, which is then converted to logical operations in order to implement it on the computer, by equating multiplication with "and" and addition with "or". If this is done, we can then exploit De Morgan's rule, which says that the product (and) of negations is equal to the negation of a sum (or), which will make the rule considerably simpler and faster when implemented:

$$nij(t+1) = nijt*not(\Sigma Aij\underline{\Sigma Okl}) + \Sigma Cij\underline{\Sigma Okl}. \quad (5)$$

There is thus an elegant correspondence between the creation and annihilation double sums.

A further consideration for the completion of the update rule concerns boundary conditions. For present purposes, a very simple treatment will suffice; this thesis is most concerned with deriving the update rule and proving that the code which implements it does so properly, and so the simpler the boundary conditions, the easier the testing becomes. In the actual code, then, a simple "quasi-reflection" condition was used, in which any particle hitting a wall had its direction changed by 180 degrees, so that the update equation would be:

$$n_{ij}(t+1)=n_{i(j+4)t}. \quad (6)$$

This is admittedly unphysical on a microscopic level; however, it is apparent that this condition does not affect the relative number of type I and type II particles, so that, essentially, it is preserving the "temperature" of the system at whatever level it equilibrates to by means of the update rule. The condition will affect the precise velocity distributions of type I and II particles, but for the present this is not an important concern.

Eventually, when the CA is modified to simulate temperature changes, the boundary conditions will become the means by which the temperature of the system is controlled. This will be done by weighting the probabilities of emitting type I and type II particles from the wall, so that the relative numbers going out are not necessarily equal to the relative numbers coming in. However, before this can be done, the update rule must be successfully implemented, and that is the next topic of discussion.

III: The FORTRAN Code

From the logical form of the update rule (equation (5)) it is a fairly straightforward task to code the rule into FORTRAN; the code used for this thesis is given in the Appendix. On the MicroVAX, system functions exist which can perform bitwise operations on words in memory:

these are iand, ior, ieor, not, ishftc, and the mvbits subroutine. The comments in the code explain how each operator is generated logically and then used to build the update rule.

The twelve bits for each lattice site are grouped into twelve words of 32 bits each in memory; thus each word stores one bit, denoting the presence or absence of one particle state, for each of 32 lattice sites. The VAX processor is a 32-bit processor, which is why this method of storing data is utilized; each of the twelve particle states can be updated 32 sites at a time for greater speed. The words are organized horizontally on the lattice, so that each word contains bits from 32 sites lying along a horizontal line. The only exceptions are the special left and right edge words, which are aligned vertically. Thus the vertical array size is a multiple of 32 sites, and the horizontal array size is a multiple of 32 plus 2 extra sites.

The first section of the code concerns initialization of the grid. Input parameters are given: the grid size in bits, the number of time steps desired, the frequency of decays, and the seed for the random number generator. Then, after calculating other necessary parameters, the program reads in the probabilities which will be used to initialize the grid. At each lattice site, there are twelve possible states which may be occupied, and the probabilities, when compared with the outputs from a random number generator, tell which of those states are actually occupied. Thus the initial states of all lattice points are generated, providing a starting array for the update rule.

The first part of the update process is the "movement phase". This phase is intended to make each of the twelve words at location i in the grid represent the bits for particles moving in that direction "into" the 32 lattice sites represented by that word. Thus word l1(i) would represent the bits for a particle moving in direction 1 "into" the 32 sites at location i in the grid. It is obvious from our discussion of the update rule above that this is the proper form for the words in memory in order to apply the rule.

There are two tools used to move the bits to their proper locations for applying the rule. The intrinsic function ishftc represents a "circular shift" of bits, each bit moving one place to the right or left (right was +1, left was -1 in the function arguments); this is used to move bits horizontally. Moving bits vertically amounts to shifting whole words at once, with the array reference variable shift used to denote the original location of the word. Since for greater speed we use only one index to reference our words in memory, this portion of the code is a bit obscure. Word i=1 is in the lower left corner of the grid; i increases to the right, and then upward, row by row, so that i=32 (for a 1026x1024 bit array) is in the lower right corner, and i=32768 (1024*32) is in the upper right corner. Therefore, moving a word vertically upwards means a shift from location i-32 to location i, and moving a word vertically downwards means a shift from location i+32 to location i. The actual array size is an input parameter.

The mvbits system subroutine is then used to move the end bits of words, which the circular shift would move to the other end of the word, to the end bits of the proper adjacent words. Thus, for a direction-3 move of word location i, we circular shift one unit to the right, meaning that the rightmost bit of word i (bit 31) would be shifted to bit 0 of word i. We then use mvbits to move that bit to bit 0 of word i+1. To make sure that we do not overwrite the wrong bits, we start rightward moves (directions 2, 3, and 4) from the largest value of i, and leftward moves (directions 6, 7, and 8) from the smallest value of i. The circular shift and mvbits subroutine are also used to move the bits in the left and right wall words to the necessary places for mvbits to be used between them and the interior words.

This completes the movement phase; the words are now all in their proper locations to apply the update rule. Boundary conditions and the rule itself are implemented directly from their logical forms, described in section II. above, and then optimization is done, although most of the optimization potential is in the movement phase.

However, optimization is not a primary issue at present; the code in the Appendix still has considerable inefficiency, but it does mirror the rule correctly.

The final section of the code is the output phase, which uses the raw data to calculate the total number of particles of type I and type II, and the sum of both, over the entire grid. This section of the code can be easily modified to give density or temperature distributions which can be analyzed to determine the macroscopic behavior of the system.

IV: Results

Having developed a working FORTRAN code, the final step is to determine whether it mirrors the update rule correctly, and also whether that rule, as mirrored in the code, provides the necessary framework for modeling temperature. Testing to see whether the rule is correctly implemented is straightforward, since the logical equations (5) and (6) can be directly translated into FORTRAN statements. However, there are still some possible problems: after all, the "movement phase" does not enter into the update rule equations, which assume that it has already been accomplished. Also, we have an input and output phase which are separate from the update process.

Testing the input and output is fairly straightforward; if the output consistently gives us the expected probabilities of particles in each state, then both sections of code are valid. For example, if there is a probability of 0.2 of finding a type I particle in a given direction at a given lattice site, then the probability of finding a type I at that site, in any of the 4 directions, is 4*0.2, or 0.8. Thus, if we take the total number of type I particles, as output from the program after 0 time steps, that should be 0.8 times the total number of lattice sites on the grid. Similar arguments apply to type II particles. This was checked and found to be correct to within the limits of the random number generator (statistical fluctuations).

To test other aspects of the code, physical considerations must be brought into play, bringing us to the second test, which is whether the update rule models the correct physics for our purposes. If the results that the code generates can be shown to do so, this will also be proof that the aspects of the code which cannot be directly checked against the update rule are nevertheless correct.

One aspect of the physics which can be checked is conservation of mass. Since photons have no mass, the conservation of mass simply implies that the total number of type I plus type II particles is constant over every time step. This was found to hold when the code was run with various initial conditions.

The CA was also constructed to allow for energy exchange between type I and type II particles, which can be checked by observing the relative numbers of the two at various time steps. It was found that, with initial conditions which had approximately equal numbers of both types, the number of type I's rose while the number of type II's dropped. This can be interpreted as thermal equilibration, since the numbers of both types were observed to converge, with oscillations caused by statistical fluctuations, to certain values which would determine the equilibrium "temperature" of the system.

Thus, it seems that the CA does indeed provide what is needed to model temperature. We have an update rule that conserves necessary quantities, and we have a mechanism for energy exchange which is shown to work. This is the needed foundation on which to build CA's for simulating actual thermal processes.

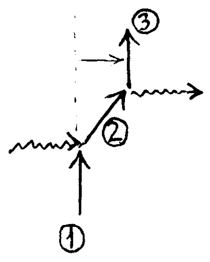
V: Suggestions for Further Research

The basic CA has now been constructed, and it appears to model the necessary physics correctly. However, it is indeed very basic: there is energy exchange between type I and type II particles, but there is no immediately obvious way of affecting the equilibrium temperature. We have tried to show here that, if any errors appear in the model when new considerations are added, they will not be due to the basic code which has been developed here. The next step in research will be to add these new considerations, a short list of which follows:

A: As was mentioned at the end of section II, the boundary conditions can be changed to allow the temperature of the system to be controlled through interactions with the walls. This could be used to set the wall temperature equal to a constant or to vary the wall temperature in time and observe the interior distribution, as well as other effects, such as a possible phase transition.

B: Also in section II the idea of giving photons negative momentum was mentioned. This might also be necessary to allow modeling of a phase transition.

C: It might be desirable to bias the photon distribution in a certain direction, thus creating a macroscopic "force field" in that direction. This effect can be seen by considering the effect of production and decay processes in sequence: these sequences can have the effect of moving a type I particle perpendicular to its direction of motion but parallel to the photon involved, as is shown in Fig. 2. Thus, if there is an overabundance of photons in a certain direction, their net effect will be to move type I particles in that direction. The more abundant type I's will then, via more production processes, create more type II's. This will be observed macroscopically as a force field, which might be useful when modeling a phase transition.



① - Type I starts at certain point, moving straight up

② - Production of Type II when right-moving photon hits Type I

③ - Decay now produces Type I displaced to the right from original Type I as if by a "force field"

Fig. 2: Effect of Production and Decay in Sequence

Appendix: A copy of the code, with comments included, is provided.

```
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c                  PROGRAM FOR CELLULAR AUTOMATON TO SIMULATE
c                            AN INTERACTING SYSTEM
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c      first make all variables except those at end of alphabet integer*4
c      so that the arrays will have 32-bit words
c
       implicit integer*4 (a-t)
c
c      reserve memory space: l arrays are particles, m are photons; the n
c      in the name means "new", used in the move loop below; lleft and
c      lright, mleft and mright are special words, vertically stacked,
c      for the left and right walls, for easier implementation of the
c      boundary conditions
c
       dimension l1(32768),l2(32768),l3(32768),l4(32768)
       dimension l5(32768),l6(32768),l7(32768),l8(32768)
       dimension m1(32768),m3(32768),m5(32768),m7(32768)
       dimension ln1(32768),ln2(32768),ln3(32768),ln4(32768)
       dimension ln5(32768),ln6(32768),ln7(32768),ln8(32768)
       dimension mn1(32768),mn3(32768),mn5(32768),mn7(32768)
       dimension lright2(32),lright3(32),lright4(32),lright7(32)
       dimension lright8(32),lleft8(32),lright6(32),lleft6(32)
       dimension lleft2(32),lleft3(32),lleft4(32),lleft7(32)
       dimension mright3(32),mright7(32),mleft3(32),mleft7(32)
       common icount
       open(unit=6,file='thesis.out',status='new')
c
cccccccccccccccccccccccccccccINPUT PARAMETERScccccccccccccccccccccccccccccccc
c
c      section to read in necessary parameters
c
       write(*,9900)
 9900  format(/'  Input max. time steps and array dimensions
     1          in bits.')
       read(*,*)tmax,imax,jmax
       write(*,9901)
 9901  format(/'  Input decay frequency.')
       read(*,*)m
       write(*,9902)
 9902  format(/'  Input random number seed.')
       read(*,*)seed
c
c      calculate other parameters to facilitate single-index array
c      references and proper number of words in memory.  The variables
c      are defined as follows:
c
c      tmax--maximum number of time steps
c      imax--vertical array size in bits; must be multiple of 32
c      jmax--horizontal array size in bits; mod(jmax,32) is 2
c      jm--number of words across one horizontal row, 32 bits per word
c      im--total number of words on the lattice, not counting left and
c          right edges
c      in--number of words along the left and right edges, vertically
c          aligned
c      im1,im2--index limits of words when top and bottom edge rows are
c          not included
c
c      bits are numbered left to right, so that bit 0 is leftmost and
c      bit 31 is rightmost in each word.  For the left and right edge
c      words, bit 0 is lowest and bit 31 is highest.  Words are indexed
```

```
c       starting in the lower left corner, numbered to the right, row by
c       row going up, each row increasing left to right.  The left and
c       right edge words are numbered bottom to top.
c
        jm=(jmax-2)/32
        im=imax*jm
        in=imax/32
        im1=im-jm
        im2=1+jm
c
ccccccccccccccccccccccccccccccINITIALIZATIONccccccccccccccccccccccccccccccc
c
c       section here to initialize the array
c
c       first input the probabilities which will be used to decide, for
c       each particle type and direction, whether that state is full or
c       empty initially at each lattice site
c
        write(*,9899)
 9899   format(/'  Input initial particle probabilities.')
        read(*,*)xd1,xd2,xdp
c
c        procedure for determining initial array state; the variable
c        iset is simply a word with a '1' in bit 0 and a '0' in all
c        other bits; we use it as a 'source' from which to set the proper
c        bits in each word to 1.
c
         iset=1
c
c        do interior points only: no initial wall particles
c
         do 10 i=im2,im1
         do 10 j=1,32
c
c        i is the word index, j is the bit index within the word.
c        We check each of the twelve possible states for each bit
c        (lattice site); probability of finding a particle in each
c        one is the initial density xd for that particle type.  We
c        check twelve times at each bit site, once for each state,
c        and if it is not occupied, we skip it, otherwise we fill it.
c
         jd=j-1
         ycheck=ran(seed)
         if(ycheck.gt.xd1)goto 11
         call mvbits(iset,0,1,l1(i),jd)
 11      ycheck=ran(seed)
         if(ycheck.gt.xd2)goto 12
         call mvbits(iset,0,1,l2(i),jd)
 12      ycheck=ran(seed)
         if(ycheck.gt.xd1)goto 13
         call mvbits(iset,0,1,l3(i),jd)
 13      ycheck=ran(seed)
         if(ycheck.gt.xd2)goto 14
         call mvbits(iset,0,1,l4(i),jd)
 14      ycheck=ran(seed)
         if(ycheck.gt.xd1)goto 15
         call mvbits(iset,0,1,l5(i),jd)
 15      ycheck=ran(seed)
         if(ycheck.gt.xd2)goto 16
         call mvbits(iset,0,1,l6(i),jd)
 16      ycheck=ran(seed)
         if(ycheck.gt.xd1)goto 17
         call mvbits(iset,0,1,l7(i),jd)
 17      ycheck=ran(seed)
```

```
        if(ycheck.gt.xd2)goto 18
        call mvbits(iset,0,1,l8(i),jd)
 18     ycheck=ran(seed)
        if(ycheck.gt.xdp)goto 21
        call mvbits(iset,0,1,m1(i),jd)
 21     ycheck=ran(seed)
        if(ycheck.gt.xdp)goto 23
        call mvbits(iset,0,1,m3(i),jd)
 23     ycheck=ran(seed)
        if(ycheck.gt.xdp)goto 25
        call mvbits(iset,0,1,m5(i),jd)
 25     ycheck=ran(seed)
        if(ycheck.gt.xdp)goto 10
        call mvbits(iset,0,1,m7(i),jd)
 10     continue
c
c       diagnostic writes
c
        write(6,5550)tmax,imax,jmax
 5550   format(/i5,' time steps on a',i5,' by',i5,' array.')
        write(6,5551)jm,im,in
 5551   format(/i5,' words per row,',i5,' words, and',i5,' side words.')
        write(6,5554)m
 5554   format(/'  Decays every',i5,' time steps.')
        write(6,5555)xd1,xd2,xdp
 5555   format(/'  Initial particle probabilities: ',3f5.2)
c
c       initialize time step counters
c
        tstep=0
        parity=0
c
cccccccccccccccccccccccccccccccMOVE LOOPcccccccccccccccccccccccccccccccccccc
c
c      do movement phase--first step in update process
c
c      these first three loops are designed to move bits in accordance with
c      the propagation rule for their direction, including bit shifts and
c      movements of an entire word; thus, for example, direction 1 words are
c      simply shifted as a whole one unit upwards, while direction 7
c      words are circle-shifted only.  The circle-shift is required to
c      prevent the loss of a bit.  Note that this is not really a complete
c      movement process; it is simply making it easier for the collision
c      operators below to be generated.
c
c      first check for last time step completed; if so, go to output.
c      This allows us to check the initial conditions by simply putting
c      tmax=0, so that it will go to output immediately, without doing the
c      update process.  After checking, we increment the time step variable.
c
 50    if(tstep.eq.tmax)goto 9999
       tstep=tstep+1
       do 100 i=1,im1
c
c      The variable shift gives the index number of the word directly
c      "above" word i
c
       shift=i+jm
       ln6(i)=ishftc(l6(shift),-1,32)
       ln5(i)=l5(shift)
       ln4(i)=ishftc(l4(shift),1,32)
       mn5(i)=m5(shift)
 100   continue
       do 110 i=im2,im1
```

```
      ln3(i)=ishftc(l3(i),1,32)
      ln7(i)=ishftc(l7(i),-1,32)
      mn3(i)=ishftc(m3(i),1,32)
      mn7(i)=ishftc(m7(i),-1,32)
 110  continue
      do 120 i=im2,im
c
c     The variable shift now gives the index number of the word directly
c     "below" word i
c
      shift=i-jm
      ln2(i)=ishftc(l2(shift),1,32)
      ln1(i)=l1(shift)
      ln8(i)=ishftc(l8(shift),-1,32)
      mn1(i)=m1(shift)
 120  continue
c
c     These next two loops do the bit shifts and movements for the
c     special wall arrays, which must be done prior to the main
c     mvbits loops
c
      do 125 i=1,in
      lleft2(i)=ishftc(lleft2(i),1,32)
      lleft4(i)=ishftc(lleft4(i),-1,32)
      lright6(i)=ishftc(lright6(i),-1,32)
      lright8(i)=ishftc(lright8(i),1,32)
 125  continue
      do 126 i=1,in-1
      shift=i+1
      call mvbits(lleft4(shift),31,1,lleft4(i),31)
      call mvbits(lright6(shift),31,1,lright6(i),31)
      ip=in-i
      ip1=ip+1
      call mvbits(lleft2(ip),0,1,lleft2(ip1),0)
      call mvbits(lright8(ip),0,1,lright8(ip1),0)
 126  continue
c
c     these next four loops finish the process of moving bits by
c     transferring bits which were circle-shifted from the end of a
c     word to their proper word -- thus, the 'left' bit of a direction-7
c     word, which got shifted to the 'right' bit of that word, is now
c     moved to the 'right' bit of the next word up.  This requires two
c     special 'rows' of words, left and right of the array, to take the
c     last bits of the left and right edge words.  Thus, the first loop
c     and the last loop of the four move the proper bits in and out of
c     these special arrays.
c
c     Our pattern is to move bits "into" the walls first, then to move
c     bits word by word, "sweeping" through the lattice so that each
c     bit, as it is moved into its new word, overwrites the bit that
c     was just moved out of that word.  Thus, the last step involves
c     moving bits "out of" the wall into the proper words.
c
      do 130 i=1,in
      kk=(i-1)*32
      do 130 j=1,32
c
c     the variable k is the 'counter' for bits along the matrix; since
c     the left and right special arrays are 'stacked' vertically, k
c     is needed to keep track of which bit in which array is being
c     dealt with.  The variables ik tell which word of the array to look
c     at to move bits to and from the special arrays; ikjm gives the index
c     numbers of the rightmost words, and ik1 gives the index numbers of
c     the leftmost words
```

```
c
      k=kk+j
      ikjm=k*jm
      ik1=ikjm-jm+1
      jd=j-1
c
c     the two if statements eliminate the proper mvbits statements for the
c     bottom and top walls: on the bottom, where k=1, we need only consider
c     directions 4 and 6; and on the top, where k=imax, we need only conside
c     directions 2 and 8.  This pattern continues in all loops dealing with
c     the bottom and top walls.  Neither wall has particles moving parallel
c     to it, so directions 3 and 7 are out for both.  What this all amounts
c     to is that, for each "corner" of the lattice, there is only one directi
c     that will move a particle "into" it; and there are two corners on the
c     bottom and two on the top, with directions as given above.  At these
c     places, no other directions need be considered.
c
      if(k.eq.imax)goto 131
      call mvbits(ln4(ikjm),0,1,lright4(i),jd)
      call mvbits(ln6(ik1),31,1,lleft6(i),jd)
      if(k.eq.1)goto 130
      call mvbits(ln3(ikjm),0,1,lright3(i),jd)
      call mvbits(ln7(ik1),31,1,lleft7(i),jd)
      call mvbits(mn3(ikjm),0,1,mright3(i),jd)
      call mvbits(mn7(ik1),31,1,mleft7(i),jd)
 131  call mvbits(ln2(ikjm),0,1,lright2(i),jd)
      call mvbits(ln8(ik1),31,1,lleft8(i),jd)
 130  continue
c
c     do mvbits for interior words only; bottom and top walls are
c     done in a separate loop
c
      do 140 i=im2,im1
c
c     if the word i is in column 32 (or the rightmost column) then we
c     cannot do directions 2,3,4; if the word i is in the leftmost
c     column we cannot do directions 6,7,8.  Therefore we must put in
c     the two if-statements to preclude this.  Also, since for directions
c     2,3,4 we must sweep 'backwards' through the lattice, for those
c     directions we use the array reference variable i1, which means that
c     as i sweeps up and to the right, i1 sweeps to the left and down, so
c     that we are always moving the correct bits.
c
      i1=im-i
      i11=i1+1
      id=i-1
      if(mod(i,jm).eq.0)goto 145
      call mvbits(ln4(i1),0,1,ln4(i11),0)
      call mvbits(ln3(i1),0,1,ln3(i11),0)
      call mvbits(ln2(i1),0,1,ln2(i11),0)
      call mvbits(mn3(i1),0,1,mn3(i11),0)
      if(mod(i,jm).eq.1)goto 140
 145  call mvbits(ln8(i),31,1,ln8(id),31)
      call mvbits(ln7(i),31,1,ln7(id),31)
      call mvbits(ln6(i),31,1,ln6(id),31)
      call mvbits(mn7(i),31,1,mn7(id),31)
 140  continue
c
c     now do mvbits for bottom and top walls
c
      do 150 i=1,jm-1
      itop=im1+i
      jtop=im-i
      jbot=jm-i
      i1=i+1
      itop1=itop+1
      jtop1=jtop+1
      jbot1=jbot+1
      call mvbits(ln4(jbot),0,1,ln4(jbot1),0)
      call mvbits(ln2(jtop),0,1,ln2(jtop1),0)
      call mvbits(ln6(i1),31,1,ln6(i),31)
      call mvbits(ln8(itop1),31,1,ln8(itop),31)
 150  continue
c
c     now move bits out of left and right walls into proper words; here
c     again, at the "corner" points, k=1 and k=imax, we only have two
c     directions to deal with, one for each corner on both the bottom
c     and top walls.  The directions are reversed from the first time
c     we did this because now we want the directions moving "out of" each
c     corner.
c
      do 160 i=1,in
      kk=(i-1)*32
      do 160 j=1,32
      k=kk+j
      ikjm=k*jm
      ik1=ikjm-jm+1
      jd=j-1
      if(k.eq.1)goto 161
      call mvbits(lleft4(i),jd,1,ln4(ik1),0)
      call mvbits(lright6(i),jd,1,ln6(ikjm),31)
      if(k.eq.imax)goto 160
      call mvbits(lleft3(i),jd,1,ln3(ik1),0)
      call mvbits(lright7(i),jd,1,ln7(ikjm),31)
      call mvbits(mleft3(i),jd,1,mn3(ik1),0)
      call mvbits(mright7(i),jd,1,mn7(ikjm),31)
 161  call mvbits(lleft2(i),jd,1,ln2(ik1),0)
      call mvbits(lright8(i),jd,1,ln8(ikjm),31)
 160  continue
c
c     this last loop 'clears' the ln and mn arrays so that they can be used
c     in the next phase of the update process; since the old values of the
c     l and m arrays are no longer needed, they are written over.
c
      do 170 i=1,im
      l1(i)=ln1(i)
      l2(i)=ln2(i)
      l3(i)=ln3(i)
      l4(i)=ln4(i)
      l5(i)=ln5(i)
      l6(i)=ln6(i)
      l7(i)=ln7(i)
      l8(i)=ln8(i)
      m1(i)=mn1(i)
      m3(i)=mn3(i)
      m5(i)=mn5(i)
      m7(i)=mn7(i)
 170  continue
c
cccccccccccccccccccccccccccccccBOUNDARY CONDITIONSccccccccccccccccccccccccccccc
c
c     update walls--simple condition to ensure conservation of particles
c     and preservation of the relative distributions, so that the wall
c     will not change the "temperature" of the system; we simply "reflect"
c     each incoming particle by 180 degrees off the wall, so that, for
c     example, a particle in direction 2 becomes a particle in direction
c     6, regardless of which wall it hits.  This is not valid physically
c     in a microscopic sense, but will affect only the velocity distribution
```

```
c      and not the relative numbers of type I's and type II's
c
       do 300 i=1,in
       lleft2(i)=lleft6(i)
       lleft6(i)=0
       lleft3(i)=lleft7(i)
       lleft7(i)=0
       lleft4(i)=lleft8(i)
       lleft8(i)=0
       mleft3(i)=mleft7(i)
       mleft7(i)=0
       lright8(i)=lright4(i)
       lright4(i)=0
       lright7(i)=lright3(i)
       lright3(i)=0
       lright6(i)=lright2(i)
       lright2(i)=0
       mright7(i)=mright3(i)
       mright3(i)=0
 300   continue
       do 350 i=1,jm
       itop=im1+i
       l8(i)=l4(i)
       l4(i)=0
       l1(i)=l5(i)
       l5(i)=0
       l2(i)=l6(i)
       l6(i)=0
       m1(i)=m5(i)
       m5(i)=0
       l4(itop)=l8(itop)
       l8(itop)=0
       l5(itop)=l1(itop)
       l1(itop)=0
       l6(itop)=l2(itop)
       l2(itop)=0
       m5(itop)=m1(itop)
       m1(itop)=0
 350   continue
c
cccccccccccccccccccccccccccccccccUPDATE RULEcccccccccccccccccccccccccccccccccc
c
c      update interior collisions using simple Boolean operators; there are
c      four different kinds of time steps involved, odd and even, decay and
c      non-decay, and each one has its own Boolean rule.  The bit-movement
c      process above has ensured that for each array element (i), all the
c      bits in that word, for each velocity direction, are the proper ones
c      to be 'plugged in' to the collision operators in order to obtain the
c      right output state for that array element.  Thus, as was said, the
c      above process was not really a movement phase; the actual propagation
c      operator is incorporated with all the others in the rule below.
c
c      check for decay or non-decay step
c
       chdecay=mod(tstep,m)
       if(chdecay.eq.0)goto 590
c
c      check for odd or even step
c
       chparity=mod(tstep,2)
       if(chparity.eq.0)goto 490
c
c      odd non-decay step
c
```

```
      do 400 i=im2,im1
c
c     define basic operators-necessary conditions.  What this means is
c     that these operators define the 'first-order' conditions for a
c     process to happen, namely, that the required input states are
c     present and the the required output states are open.  These operators
c     have nothing to do with any other process which might share an
c     input or output state with the given process; these conditions are
c     taken care of by the composite operators defined below.  Each basic
c     operator corresponds to a creation or annihilation operator in the
c     physical form of the rule.
c
c     Each operator has several physical 'names', since it represents a
c     creation of some particles and an annihilation of others.  For
c     consistency, for production processes I have chosen the unique
c     'creation' form, and for decays the unique 'annihilation' form
c     (since these processes create and annihilate, respectively, only
c     one particle).  For self-collisions, I used 'creation' form and
c     chose the lower-numbered of the two possible directions.  A table
c     of equivalences follows; the operator name as used in the program
c     appears on the left, and other equivalent names, which may help in
c     understanding the composite operators, are given on the right.
c
c     cs11    =cs15=as13=as17
c     cs22    =cs26=as24=as28
c     cs13    =cs17=as11=as15
c     cs24    =cs28=as22=as26
c     cpo22   =apo11=apop3
c     cpo24   =apo13=apop5
c     cpo26   =apo15=apop7
c     cpo28   =apo17=apop1
c     cpe22   =ape13=apep1
c     cpe24   =ape15=apep3
c     cpe26   =ape17=apep5
c     cpe28   =ape11=apep7
c     ado22   =cdo11=cdop3
c     ado24   =cdo13=cdop5
c     ado26   =cdo15=cdop7
c     ado28   =cdo17=cdop1
c     ade22   =cde13=cdep1
c     ade24   =cde15=cdep3
c     ade26   =cde17=cdep5
c     ade28   =cde11=cdep7
c
c     the operator notation corresponds to physics notation: the first
c     letter denotes either (c)reation or (a)nnihilation; the next one
c     or two letters are the process index: s for self-collision, po and
c     pe for odd or even productions, do and de for odd or even decays.
c     The last two characters are the particle type and direction: the
c     latter is always a digit from 1 to 8, with 1 being vertically upward
c     and the other directions numbered clockwise at 45 degree angles; and
c     the particle type is either 1, 2, or p for photon.
c
c     first, to save array references, we write the current
c     words into dummy variables.
c
 410  e1=l1(i)
      e2=l2(i)
      e3=l3(i)
      e4=l4(i)
      e5=l5(i)
      e6=l6(i)
      e7=l7(i)
      e8=l8(i)
      f1=m1(i)
      f3=m3(i)
      f5=m5(i)
      f7=m7(i)
c
c     now do basic operators for like-like collisions
c
      cs11=iand(e7,iand(e3,not(ior(e1,e5))))
      cs22=iand(e8,iand(e4,not(ior(e2,e6))))
      cs13=iand(e1,iand(e5,not(ior(e3,e7))))
      cs24=iand(e2,iand(e6,not(ior(e4,e8))))
c
c     since the like-like operators are the same for all processes, we
c     use the above lines of code in all four do-loops.  Thus, we need
c     the disjunct below: if the step is even we go to the even
c     production operators, otherwise we stay with the odd ones below
c
      if(chparity.eq.0)goto 530
c
c     production processes-these are also used in decay steps, thus the
c     disjunct after the four definitions to shift to the decay loop if
c     on a decay step, but only after the composite operators are defined
c     as well--since they are also the same for decay steps
c
 430  cpo22=iand(e1,iand(f3,not(e2)))
      cpo24=iand(e3,iand(f5,not(e4)))
      cpo26=iand(e5,iand(f7,not(e6)))
      cpo28=iand(e7,iand(f1,not(e8)))
c
c     define composite operators--link with negations of other basic op.
c     to generate conditions for the process actually happening--thus we
c     take into account all other processes which share either an input
c     or an output state with the given process, and negate them.  This
c     ensures that the system will be microscopically reversible, i.e.,
c     the mapping from input to output states at each point is one-to-
c     one.  These operators correspond to the summation/product terms in
c     the physical rule of update.  Note that each one is used both in
c     the creation sum and the annihilation sum (which is negated in the
c     free propagation term), but not necessarily in the rule for particles
c     of the same direction and type.
c
c     production processes
c
      nap2o=iand(cpo22,not(ior(cs22,cs13)))
      nap4o=iand(cpo24,not(ior(cs24,cs11)))
      nap6o=iand(cpo26,not(ior(cs22,cs13)))
      nap8o=iand(cpo28,not(ior(cs24,cs11)))
c
c     go to decay loop if on decay step
c
      if(chdecay.eq.0)goto 620
c
c     like-like collisions
c
      nas1=iand(cs11,not(ior(cpo24,cpo28)))
      nas2=iand(cs22,not(ior(cpo22,cpo26)))
      nas3=iand(cs13,not(ior(cpo22,cpo26)))
      nas4=iand(cs24,not(ior(cpo24,cpo28)))
c
c     use the above composite operators to generate the update rule:
c     the conjunction of all possible processes that produce a particle
c     in the given state.  Note that the operators for each process
c     are used both as creation and annihilation operators as context
c     requires.  I have here defined the extra dummy variables nn in
```

```
c       order to more clearly show how the rule is mirrored in the code;
c       the nn's correspond to the first term, which is the free propagation
c       times the negation of the annihilation summation terms.  In the
c       subsequent rule definitions, this extra dummy variable is omitted.
c
        nn1=iand(e1,not(ior(nas3,nap2o)))
        l1(i)=ior(nn1,nas1)
        nn2=iand(e2,not(nas4))
        l2(i)=ior(nn2,ior(nas2,nap2o))
        nn3=iand(e3,not(ior(nas1,nap4o)))
        l3(i)=ior(nn3,nas3)
        nn4=iand(e4,not(nas2))
        l4(i)=ior(nn4,ior(nas4,nap4o))
        nn5=iand(e5,not(ior(nas3,nap6o)))
        l5(i)=ior(nn5,nas1)
        nn6=iand(e6,not(nas4))
        l6(i)=ior(nn6,ior(nas2,nap6o))
        nn7=iand(e7,not(ior(nas1,nap8o)))
        l7(i)=ior(nn7,nas3)
        nn8=iand(e8,not(nas2))
        l8(i)=ior(nn8,ior(nas4,nap8o))
c
c       for photons, note that in a non-decay step there is only one process
c       that will produce a photon--propagation--and only one annihilation
c       operator that needs to be included--the production process
c
        m1(i)=iand(f1,not(nap8o))
        m3(i)=iand(f3,not(nap2o))
        m5(i)=iand(f5,not(nap4o))
        m7(i)=iand(f7,not(nap6o))
 400    continue
        goto 1000
c
c       even non-decay step
c
 490    do 500 i=im2,im1
c
c       goto like-like operator definitions
c
        goto 410
c
c       define basic production operators--the disjunct, as before,
c       shifts to the decay loop if on a decay step, but only after the
c       composite operators have also been defined
c
 530    cpe22=iand(e3,iand(f1,not(e2)))
        cpe24=iand(e5,iand(f3,not(e4)))
        cpe26=iand(e7,iand(f5,not(e6)))
        cpe28=iand(e1,iand(f7,not(e8)))
c
c       define composite production operators
c
        nap2e=iand(cpe22,not(ior(cs22,cs11)))
        nap4e=iand(cpe24,not(ior(cs24,cs13)))
        nap6e=iand(cpe26,not(ior(cs22,cs11)))
        nap8e=iand(cpe28,not(ior(cs24,cs13)))
c
c       go to decay loop if on decay step
c
        if(chdecay.eq.0)goto 720
c
c       define composite like-like operators
c
        nas1=iand(cs11,not(ior(cpe22,cpe26)))
```

```
      nas2=iand(cs22,not(ior(cpe22,cpe26)))
      nas3=iand(cs13,not(ior(cpe24,cpe28)))
      nas4=iand(cs24,not(ior(cpe24,cpe28)))
c
c     generate update rule
c
      l1(i)=ior(iand(e1,not(ior(nas3,nap8e))),nas1)
      l2(i)=ior(iand(e2,not(nas4)),ior(nas2,nap2e))
      l3(i)=ior(iand(e3,not(ior(nas1,nap2e))),nas3)
      l4(i)=ior(iand(e4,not(nas2)),ior(nas4,nap4e))
      l5(i)=ior(iand(e5,not(ior(nas3,nap4e))),nas1)
      l6(i)=ior(iand(e6,not(nas4)),ior(nas2,nap6e))
      l7(i)=ior(iand(e7,not(ior(nas1,nap6e))),nas3)
      l8(i)=ior(iand(e8,not(nas2)),ior(nas4,nap8e))
      m1(i)=iand(f1,not(nap2e))
      m3(i)=iand(f3,not(nap4e))
      m5(i)=iand(f5,not(nap6e))
      m7(i)=iand(f7,not(nap8e))
 500  continue
      goto 1000
c
c     check for odd or even before choosing decay loop
c
 590  if(chparity.eq.0)goto 690
c
c     odd decay step
c
      do 600 i=im2,im1
c
c     go to like-like and production operator definitions
c
      goto 410
c
c     define basic decay operators
c
 620  ado22=iand(e2,not(ior(e1,f3)))
      ado24=iand(e4,not(ior(e3,f5)))
      ado26=iand(e6,not(ior(e5,f7)))
      ado28=iand(e8,not(ior(e7,f1)))
c
c     define composite decay operators
c
c     also, composite production operators are the same, and have already
c     been defined by the goto statement above
c
      nad2o=iand(ado22,not(ior(cs24,cs11)))
      nad4o=iand(ado24,not(ior(cs22,cs13)))
      nad6o=iand(ado26,not(ior(cs24,cs11)))
      nad8o=iand(ado28,not(ior(cs22,cs13)))
c
c     define composite like-like operators
c
      nas1=iand(cs11,not(ior(ior(ado22,ado26),ior(cpo24,cpo28))))
      nas2=iand(cs22,not(ior(ior(ado24,ado28),ior(cpo22,cpo26))))
      nas3=iand(cs13,not(ior(ior(ado24,ado28),ior(cpo22,cpo26))))
      nas4=iand(cs24,not(ior(ior(ado22,ado26),ior(cpo24,cpo28))))
c
c     generate update rule
c
      l1(i)=ior(iand(e1,not(ior(nas3,nap2o))),ior(nas1,nad2o))
      l2(i)=ior(iand(e2,not(ior(nas4,nad2o))),ior(nas2,nap2o))
      l3(i)=ior(iand(e3,not(ior(nas1,nap4o))),ior(nas3,nad4o))
      l4(i)=ior(iand(e4,not(ior(nas2,nad4o))),ior(nas4,nap4o))
      l5(i)=ior(iand(e5,not(ior(nas3,nap6o))),ior(nas1,nad6o))
      l6(i)=ior(iand(e6,not(ior(nas4,nad6o))),ior(nas2,nap6o))
      l7(i)=ior(iand(e7,not(ior(nas1,nap8o))),ior(nas3,nad8o))
      l8(i)=ior(iand(e8,not(ior(nas2,nad8o))),ior(nas4,nap8o))
c
c     photons can now be produced by decays, hence the extra operator
c
      m1(i)=ior(iand(f1,not(nap8o)),nad8o)
      m3(i)=ior(iand(f3,not(nap2o)),nad2o)
      m5(i)=ior(iand(f5,not(nap4o)),nad4o)
      m7(i)=ior(iand(f7,not(nap6o)),nad6o)
 600  continue
      goto 1000
c
c     even decay step
c
 690  do 700 i=im2,im1
c
c     goto operator definitions in other loops
c
      goto 410
c
c     define basic decay operators
c
 720  ade22=iand(e2,not(ior(e3,f1)))
      ade24=iand(e4,not(ior(e5,f3)))
      ade26=iand(e6,not(ior(e7,f5)))
      ade28=iand(e8,not(ior(e1,f7)))
c
c     define composite operators
c
      nad2e=iand(ade22,not(ior(cs24,cs13)))
      nad4e=iand(ade24,not(ior(cs22,cs11)))
      nad6e=iand(ade26,not(ior(cs24,cs13)))
      nad8e=iand(ade28,not(ior(cs22,cs11)))
      nas1=iand(cs11,not(ior(ior(ade24,ade28),ior(cpe22,cpe26))))
      nas2=iand(cs22,not(ior(ior(cpe22,cpe26),ior(ade24,ade28))))
      nas3=iand(cs13,not(ior(ior(ade22,ade26),ior(cpe24,cpe28))))
      nas4=iand(cs24,not(ior(ior(cpe24,cpe28),ior(ade22,ade26))))
c
c     generate update rule
c
      l1(i)=ior(iand(e1,not(ior(nas3,nap8e))),ior(nas1,nad8e))
      l2(i)=ior(iand(e2,not(ior(nas4,nad2e))),ior(nas2,nap2e))
      l3(i)=ior(iand(e3,not(ior(nas1,nap2e))),ior(nas3,nad2e))
      l4(i)=ior(iand(e4,not(ior(nas2,nad4e))),ior(nas4,nap4e))
      l5(i)=ior(iand(e5,not(ior(nas3,nap4e))),ior(nas1,nad4e))
      l6(i)=ior(iand(e6,not(ior(nas4,nad6e))),ior(nas2,nap6e))
      l7(i)=ior(iand(e7,not(ior(nas1,nap6e))),ior(nas3,nad6e))
      l8(i)=ior(iand(e8,not(ior(nas2,nad8e))),ior(nas4,nap8e))
      m1(i)=ior(iand(f1,not(nap2e)),nad2e)
      m3(i)=ior(iand(f3,not(nap4e)),nad4e)
      m5(i)=ior(iand(f5,not(nap6e)),nad6e)
      m7(i)=ior(iand(f7,not(nap8e)),nad8e)
 700  continue
c
c     now go back to the line which checks to see if we are on the
c     last time step, and then start the update process again
c
 1000 goto 50
c
ccccccccccccccccccccccccccccccccccccOUTPUTcccccccccccccccccccccccccccccccccccc
c
c     use simple algorithms to calculate and output the final numbers
c     of type I, type II, and total particles (not including photons)
```

```
c
 9999   write(6,5999)
 5999   format(//'  OUTPUT PARTICLE COUNTS.')
        itotal=0
        itotal1=0
        itotal2=0
        do 1100 i=1,imax
        ijm=(i-1)*jm
        icount1=0
        icount2=0
        do 1110 j=1,jm
c
c       we loop row by row going upward; for each row, the 1110 loop
c       returns values of icount1 and icount2 which represent the total
c       numbers of type I's and type II's in that row, regardless of
c       direction (we are not concerned here with the velocities.) Then
c       we must also add in the bits from the left and right special
c       arrays, and finally, we add to the variables itotal, and the
c       1100 loop takes us to the next row.
c
        ij=ijm+j
        call countbits(l1(ij))
        icount1=icount1+icount
        call countbits(l2(ij))
        icount2=icount2+icount
        call countbits(l3(ij))
        icount1=icount1+icount
        call countbits(l4(ij))
        icount2=icount2+icount
        call countbits(l5(ij))
        icount1=icount1+icount
        call countbits(l6(ij))
        icount2=icount2+icount
        call countbits(l7(ij))
        icount1=icount1+icount
        call countbits(l8(ij))
        icount2=icount2+icount
 1110   continue
c
c       add in wall bits
c
        iword=int((i-1)/32)+1
        ibit=mod((i-1),32)
        iwall=0
        call mvbits(lleft3(iword),ibit,1,iwall,0)
        icount1=icount1+iwall
        call mvbits(lright7(iword),ibit,1,iwall,0)
        icount1=icount1+iwall
        call mvbits(lleft2(iword),ibit,1,iwall,0)
        icount2=icount2+iwall
        call mvbits(lleft4(iword),ibit,1,iwall,0)
        icount2=icount2+iwall
        call mvbits(lright6(iword),ibit,1,iwall,0)
        icount2=icount2+iwall
        call mvbits(lright8(iword),ibit,1,iwall,0)
        icount2=icount2+iwall
c
c       add to total particle sums
c
        itotal1=itotal1+icount1
        itotal2=itotal2+icount2
        itotal=itotal+icount1+icount2
 1100   continue
c
```

```
c       output the particle counts
c
        write(6,1197)itotal1,itotal2
        write(*,1197)itotal1,itotal2
 1197   format(/'  Numbers of particles: ',i10,' type I
     1              and',i10,' type II.')
        write(*,1198)tstep,itotal
        write(6,1198)tstep,itotal
 1198   format(/'  Total particles at time step ',i5,':',i15)
        stop
        end

cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c       subroutine which, given an input word iarg, returns a value of the
c       variable icount which is equal to the number of bits in the word
c       iarg which have the value 1
c
        subroutine countbits(iarg)
        implicit integer*4 (a-t)
        common icount
        icount=0
        iset=1
        do 2000 j=1,32
        itest=iand(iarg,iset)
        if(itest.eq.0)goto 2010
        icount=icount+1
 2010   iarg=ishft(iarg,-1)
 2000   continue
        return
        end
```