Documentation for Ising.h and Ising.c

Steven Andrews, © 2003
See the document "LibDoc" for general information about this and other libraries.

```
typedef struct LatticeType {
     int *g;
     char shape;
     int nx;
     int ny;
     int spc;
     int rot; }* Lattice;

Lattice SetUpIsing(char shape,int nx,int ny,int spc,int *nunit,int rot);
void FreeIsing(Lattice lt);
void DisplayIsing(Lattice lt);
void MetropolisIsingS(Lattice lt,int itmax,float *eps);
void MetropolisIsingH(Lattice lt,int itmax,float *eps);
void MetropolisIsingT(Lattice lt,int itmax,float *eps);
void MetropolisIsing(Lattice lt,int itmax,float *eps);
float EnergyAtIsingS(Lattice lt,float *eps,int ix,int iy,int spc1,int rot1);
float EnergyAtIsingH(Lattice lt,float *eps,int ix,int iy,int spc1,int rot1);
float EnergyAtIsingT(Lattice lt,float *eps,int ix,int iy,int spc1,int rot1);
float EnergyAtIsing(Lattice lt,float *eps,int ix,int iy,int spc1,int rot1);
float EnergyIsingS(Lattice lt,float *eps);
float EnergyIsingH(Lattice lt,float *eps);
float EnergyIsingT(Lattice lt,float *eps);
float EnergyIsing(Lattice lt,float *eps);
int CtClustersIsing(Lattice lt,int *count,int ncount);
void DisplayClustersIsing(Lattice lt);
float MinDistIsing(Lattice lt,int ix,int iy,int jx,int jy);
void RadCorrFnIsing(Lattice lt,int spc1,int spc2,float *bins,int nbins,float
     binsize);
void DisplayRCFIsing(Lattice lt,int spc1,int spc2,float binsize);
float LatticeSiteDistH(float *xptr,float *yptr,int *ixptr,int *iyptr,float sp);
```

Requires: `<math.h>`, `<stdio.h>`,`<stdlib.h>`
Example program: `cluster.c`

History: Written 12/02, improved 1/03. All parts of all routines have been tested sufficiently to verify that they produce reasonable results, although nothing has been tested rigorously. `LatticeSiteDistH` was added 3/11/04. Added `EnergyAtIsing` set of routines 3/12/04; they have not been tested yet. Fixed a bug in `EnergyAtIsingH`, found by Josh Adelman, on 6/16/04; also fixed bugs in Metropolis routines. Fixed a minor bug in SetUpIsing on 9/21/05, found by Aleksandar Donev.

These are routines for Metropolis Monte Carlo simulations of various two dimensional Ising type lattices, which can be square, hexagonal (each site has six neighbors), or triangular (each site has 3 neighbors). All lattices use periodic boundary conditions.

Data structure. The structure `LatticeType`, pointed to with `Lattice`, contains the infomation for a complete system. A lattice of any shape is based on a square array of integers, called `g` for grid, where the correspondance between the physical positions and this square array is explained below. The `shape` member is '`S`' for square, '`H`' for hexagonal, or '`T`' for triangular. The lattice array has width `nx` and height `ny`, with numbering from 0 to `nx-1` and 0 to `ny-1`. Both `nx` and `ny` are required to be integer powers of two for the Metropolis routines, but not for the other routines, to allow indexing with bit manipulations rather than requiring multiplications and divisions. They may be as small as 1 or as large as `RAND_MAX+1` (32768 on a Macintosh), although `ny` should be at least 2 for a hexagonal lattice and 4 for a triangular lattice for the boundary conditions to work properly. `spc` is the number of species on the lattice, such as the number of different types of proteins. It may be as small as 1 or arbitrarily large. `rot` is a boolean flag which is 1 to indicate that rotations of units should be considered and 0 if they should be ignored. If they are considered, it is possible to define specific bonds on each face of a unit.

Units (*e.g.* proteins) exist only as numbers on the grid. A value of 0 in a grid position is an empty site. If rotations are not considered, then a type 1 unit is represented with a 1, a type 2 with a 2, and so on up to `spc`. If rotations are considered, then the situation is somewhat more complicated, because a unit has both a type and an orientation. If the lattice is square or triangular, the orientation information, described below, is stored in the two lowest order bits of the number while the type is stored in the higher order bits; for a hexagonal lattice, the orientation information is stored in the three lowest order bits. For example, on a square lattice, a type 2 unit (10 in binary) in orientation 1 (01 in binary) is represented on the grid with a 9 (2*4+1, or 1001 in binary). On a hexagonal lattice, a type 2 unit in orientation 1 is represented with a 17 (2*8+1, or 10001 in binary). To convert a grid value `s` to its unit type and orientation, the respective operations are:

unit type = `s>>2` for square and triangular; `s>>3` for hexagonal
orientation = `s&3` for square and triangular; `s&7` for hexagonal

Regardless of whether there are rotations, occupied grid sites always have positive values. Using this fact, it is easy to temporarily mark a unit by replacing it with its negative value. If this is done, the values should be reset before other routines are called.

The grid, `g`, is created as a single array of integers, so the address of site (`ix`,`iy`) is

`(ix,iy)` address: `iy*nx+ix`

If `ix` or `iy` have been modified, so that they might be beyond the bounds of the lattice, the periodic boundary conditions can be considered using modulo arithmetic. If x and y have been increased, the actual x position (accounting for boundaries) is `ix%nx` and the actual y position is `iy%ny`, while a small decrease is accounted for by adding `nx` or `ny`: x is `(ix+nx)%nx` and y is `(iy+ny)%ny`. Alternatively, bit manipulations can be used, although before doing this a slight change is made: `nx` and `ny` are each decremented by 1, making them into bit masks. Also, the number `b` is defined as the power of two that `nx` was equal to. For example, an 8x8 array would have bit masks `nx` and `ny` each equal to 7, which is 0111 in binary, and `b` would equal 3. Using this, the address of (`ix`,`iy`) is

`(ix,iy)` address: `iy<<b|ix`

Now, periodic boundary conditions are addressed, for any increase or decrease in `ix` or `iy`, so that the actual x position is `ix&nx` and the actual y position is `iy&ny`. While these bit manipulations may be unfamiliar to some programmers, they produce simpler code than comparable operations using integer arithmetic, as well as running between two and three times faster. The only drawback of the bit manipulations is that they only work if `nx` and `ny` are integer powers of two.

Algorithms. The standard Metopolis Monte Carlo algorithm is used, in which a series of trial moves are attempted. A trial move is always accepted if the move produces an energy difference that is zero or negative, and it is also accepted, with probability exp(–ΔE/kT), if the energy difference is positive. If there is only one species on the grid, a trial move consists of moving a single randomly chosen unit into a random neighboring site. If the neighboring site is full, the trial move is rejected. For multiple species and/or rotations, trial moves are slightly different. In these cases, a trial move consists of *swapping* a randomly chosen unit with a randomly chosen neighbor, regardless of whether the neighbor is an empty site, identical to the first unit, or something else. The advantage of this method, compared to only moving to empty sites, is that it allows the efficient simulation of densely occupied lattices. Possible drawbacks are that it may be slower for dilute lattices and the kinetics may be more inaccurate. (A general fact about the Metropolis method is that it yields rigorously correct equilibrium distributions and fluctuations, but there is no assurance that the kinetics are correct).

Square lattice. Square lattices are represented with simple rows and columns, with no offsetting. Following is a listing of the row numbers for the lattice, along with the column numbers of the sites in each row, for an 8x8 lattice.

row	column numbers							
0	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
2	0	1	2	3	4	5	6	7
3	0	1	2	3	4	5	6	7
4	0	1	2	3	4	5	6	7
5	0	1	2	3	4	5	6	7
6	0	1	2	3	4	5	6	7
7	0	1	2	3	4	5	6	7

Each site has four nearest neighbors and four diagonal neighbors. In the routines, it is possible for a unit to move to any of these eight neighboring sites, although there is only bonding to the four nearest neighbors. For each site, the nearest neighbors are positioned at U, D, L, and R (up, down, left, and right). For a site with address (`ix`,`iy`) these neighbors and their addresses are:

direction	*x*	*y*	address without bits	address with bits
R	`ix+1`	`iy`	`iy*nx+(ix+1)%nx`	`iy<<b|(ix+1)&nx`
D	`ix`	`iy+1`	`(iy+1)%ny*nx+ix`	`((iy+1)&ny)<<b|ix`
L	`ix-1`	`iy`	`iy*nx+(ix-1+nx)%nx`	`iy<<b|(ix-1)&nx`
U	`ix`	`iy-1`	`(iy-1+ny)%ny*nx+ix`	`((iy-1)&ny)<<b|ix`

Note that `nx` and `ny` are each decremented in the last column, as explained above.

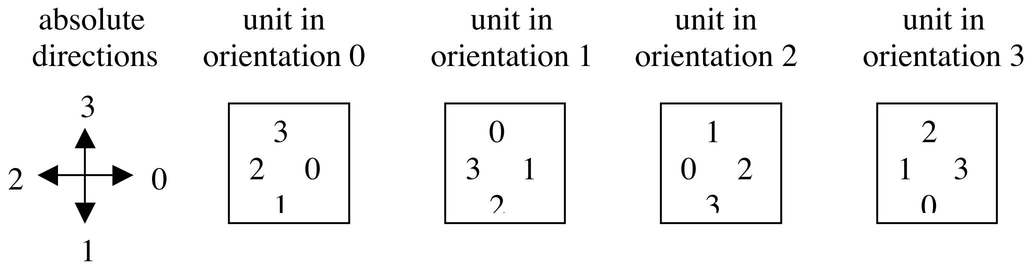
Rotations are fairly confusing. Absolute directions, *d*, are defined so that right is 0, down is 1, left is 2, and up is 3. Also, the sides of each unit, *s*, are numbered, which defines the types of bonds that may be formed with each side. One side is number 0; 1, 2, and 3 follow around in a clockwise direction. If the unit is oriented so that its 0 side is in the 0 absolute direction (and the 1 side is in the 1 direction, etc.), then the unit is said to be rotated, *r*, in the 0 orientation. Rotating the unit counter-clockwise by a quarter turn, so that the 1 side faces in the 0 direction, puts it in the 1 orientation. Using this, the relationship is

s=(r+d)%4 or s=r+d&3

For example, the side facing down (d=1) of a unit in orientation r=3 is side s=0. Because these relationships are used so frequently, they are in the code as macros. For example,

```
    #define DWNS(s) (s&~3|s+1&3)
```

Given `s` as the value of a grid site, `DWNS(s)` is the concatenation of the unit type in the high order bits (`s&~3`) with the number of the side that is facing down in the low order bits (`s+1&3`). All macros ending with ‘`S`’ are for a square lattice. Here are the four orientations of a unit on a square lattice:



In the Metropolis algorithm, after a unit is picked, a random number is used to create a trial move. If there are no rotations, then the move direction is chosen from the lowest 3 bits, which is then used in the `hrz` and `vrt` arrays, which code for the directions: R, RD, D, LD, L, LU, U, RU. If there are rotations, the lowest 4 bits are for the move direction, using arrays `hrzo` and `vrto`, and the unit is stationary if the upper two of these bits are both equal to 1: R, R, RD, D, D, LD, L, L, LU, U, U, RU, R, S, S, S, S. The next two higher bits, numbers 4 and 5, are used to create a trial orientation of the chosen unit. Finally, bits 6 and 7 are used to create a trial orientation of the neighboring unit.

The distance between a pair of lattice sites, at positions (`ix,iy`) and (`jx,jy`) is easy to calculate, by defining the variables `dx` and `dy` as the vector components from point i to point j: `dx=jx-ix` and `dy=jy-iy`. Periodic boundary conditions are accounted for by adding `nx` if `dx<nx/2` and subtracting `nx` if `dx>nx/2`, and similarly for `dy`. The result is that `dx` and `dy` are signed variables for the closest distances between the two points. The straight line distance is then just $(dx^2+dy^2)^{1/2}$.

The physical dimensions of a square lattice are simply `nx` wide by `ny` high.

Hexagonal lattice. Hexagonal lattices are also represented with a square array. Conceptually and in the display, odd numbered rows are offset to the right, so that each site has six nearest neighbors. Here are the column numbers in each row:

```
row     column numbers
 0      0  1  2  3  4  5  6  7
 1        0  1  2  3  4  5  6  7
 2      0  1  2  3  4  5  6  7
 3        0  1  2  3  4  5  6  7
 4      0  1  2  3  4  5  6  7
 5        0  1  2  3  4  5  6  7
 6      0  1  2  3  4  5  6  7
 7        0  1  2  3  4  5  6  7
```

Periodic boundary conditions work properly if there are an even number of rows and columns. Using directions from a square lattice, sites in even numbered rows of a hexagonal lattice (`iy&1==0`) have neighbors in the following positions: U, D, L, R, UL, and DL, while neighbors of a site in an odd numbered row are U, D, L, R, UR, DR. These can be combined into the following six directions:

direction	*x*	*y*
R	`ix+1`	`iy`
D	`ix`	`iy+1`
L	`ix-1`	`iy`
U	`ix`	`iy-1`
UL,UR	`ix+(iy&1?1:-1)`	`iy-1`
DL,DR	`ix+(iy&1?1:-1)`	`iy+1`

The addresses are identical to those for a square lattice, although the code gets slightly messier for the last two rows.

Including rotations is analogous to the square situation, although the low order three bits are used in this case. Comparable to the example above, the macro LDWH(s) returns the concatentation of the unit type in the high order bits with the side facing left-down in the low order bits,

```
#define LDWH(s) (s&~7|((s&7)+2)%6)
```

The unit type is from `s&~7`, while the side facing left-down is `((s&7)+2)%6`, where modulo arithmetic is now needed to account for the fact that there are only 6 possible positions, of the 8 values that can be represented in the three lowest bits.

In the Metropolis algorithm, the random number for the trial move is modified so that bit number 0 is equal to `iy&1`, making it is 0 for even rows and 1 for odd rows. Whether or not there are rotations, trial move directions are S, S, 0, 1, 2, 3, 4, 5, using the absolute directions defined above. The trial rotation of the chosen unit is found from bits 4 and higher of the random number, while the trial rotation of the neighbor is found from bits 6 and higher. This is only rigorously correct if `(rnd>>4)%6` is statistically independent of `(rnd>>6)%6`. It probably isn't, but I don't know for certain.

The distances between a pair of sites here is similar to the square situation, although the offsets of the odd rows need to be accounted for. `dx` is defined as above, as `dx=jx-ix`, and `distx` is the actual `x` distance. Here are the `distx` values for the possible row choices.

		`jy&1`	
		`0`	`1`
`iy&1`	`0`	`dx`	`dx+0.5`
	`1`	`dx-0.5`	`dx`

Also, distances along *y* are a factor of (√3)/2 larger than `dy` because of the lattice shape.

The physical dimensions of a hexagonal lattice are `nx` wide by (√3)/2 `ny` high, leading to a total area of (√3)/2 `nx ny`.

Triangular lattic. Triangular lattices are similar, although here, rows are offset in pairs.

```
row    column numbers
 0     0  1  2  3  4  5  6  7
 1     0  1  2  3  4  5  6  7
 2       0  1  2  3  4  5  6  7
 3       0  1  2  3  4  5  6  7
 4     0  1  2  3  4  5  6  7
 5     0  1  2  3  4  5  6  7
```

6	0	1	2	3	4	5	6	7
7	0	1	2	3	4	5	6	7

Periodic boundary conditions work properly if the number of rows is divisible by 4 and there are an even number of columns. Using directions from a square lattice, neighbors are found from the value of `iy&3`: if it is 0 then U, D, UL; if it is1 then D, DL, U; if it is 2 then UR, D, U; if it is 3 then DR, D, U. These combine into the following list of neighbors:

direction	*x*	*y*
U	`ix`	`iy-1`
D	`ix`	`iy+1`
UL,DL,UR,DR	`ix+(iy&2?1:-1)`	`iy+(iy&1?1:-1)`

Again, the addresses are the same as for a square lattice, although the code is messier for the last row.

Including rotations is again analogous to the square situation, although now it is more complicated because there are four kinds of triangles, for each of the possible values of `iy&3`. Direction number 0 is defined as being towards the right, which is actually right-up if `iy&3` is 0 or 2, and right-down if `iy&3` is 1 or 3. Other directions follow around in a clockwise direction.

Distances are significantly more complicated than before, but the same general principles hold. `dx` and `dy` are defined as above, and `distx` and `disty` are the actual *x* and *y* distances, accounting for lattice row offsets and differences in vertical spacing between the rows. Here are the `distx` values:

		jy&3			
		0	1	2	3
	0	√3 dx	√3 dx	√3 (dx+0.5)	√3 (dx+0.5)
iy&3	1	√3 dx	√3 dx	√3 (dx+0.5)	√3 (dx+0.5)
	2	√3 (dx-0.5)	√3 (dx-0.5)	√3 dx	√3 dx
	3	√3 (dx-0.5)	√3 (dx-0.5)	√3 dx	√3 dx

The `disty` values are a bit more complicated:

		jy&3			
		0	1	2	3
	0	3 dy/4	1+3⌊dy/4⌋	1.5+3⌊dy/4⌋	2.5+3⌊dy/4⌋
iy&3	1	2+3⌊dy/4⌋	3 dy/4	0.5+3⌊dy/4⌋	1.5+3⌊dy/4⌋
	2	1.5+3⌊dy/4⌋	2.5+3⌊dy/4⌋	3 dy/4	1+3⌊dy/4⌋
	3	0.5+3⌊dy/4⌋	1.5+3⌊dy/4⌋	2+3⌊dy/4⌋	3 dy/4

The term `⌊dy/4⌋` indicates the floor of the floating point value `dy/4`; for example, `⌊0.5⌋ = 0` and `⌊-0.5⌋ = -1`.

The physical dimensions of a triangular lattice are √3 `nx` wide by 3/4 `ny`, leading to a total area of 3(√3)/4 `nx ny`.

Functions

`SetUpIsing` allocates and initiallizes a lattice, with shape `shape` (‘`S`’, ‘`H`’, or ‘`T`’) and dimensions `nx` wide by `ny` high. Both dimensions need to be integer powers of two. `spc` is the number of species on the lattice, so it needs to be at least 1. `nunit` is a `spc` size list of the number of units for each type of species. For example, if sites are either occupied or open, then `spc` is 1 and `nunit` is a pointer to an integer. If sites can be occupied with either a type '1' or a type '2', then `spc` is 2 and `nunit` is an array of two integers: the number of '1's, and the number of '2's. `rot` is a flag equal to 1 if rotations of units are to be considered and 0 if they are to be ignored. Units are randomly positioned within the lattice. If rotations are considered, units have random orientations. If inputs are out of range, such as too many units or a lattice larger than $(\texttt{RAND\_MAX+1})^2$, or if memory could not be allocated, then the function frees any memory that was allocated and returns `NULL`. This function runs efficiently for any amount of lattice occupancy. Enter `NULL` for `nunit` to create an empty lattice.

`FreeIsing` frees all portions of a lattice structure. Partially allocated lattices are permitted.

`DisplayIsing` displays the current state of a lattice as a text array to the standard output. Occupied sites are shown with an 'x' if there is only one species or with other letters for more species. If rotations are considered, the orientation of each site is appended as a number. Open sites are shown with spaces.

`MetropolisIsingS` executes the Metropolis algorithm for a square lattice given in `lt`. `itmax` is the number of trial moves that should be attempted and `eps` is a pointer to the bond energy information. These interaction energies need to have been divided by kT; a negative number indicates a favorable interaction, such as a bond. This function is actually a concatenation of three completely independent routines, (*i*) considers rotations, (*ii*) is for multiple species but no rotations, and (*iii*) is for one species and no rotations. In (*i*), `eps` is a matrix of size $((\texttt{spc+1})*4)^2$. Row and column 0 are for interactions with empty sites; 1 to 3 are ignored; 4 to 7 are for sides number 0 to 3 of a type 1 unit; 8 to 11 are for sides number 0 to 3 of a type 2 unit, and so forth. `eps` must be symmetric, although this is not checked by the routine. For example, the number at the position of row 4 and column 9 is the interaction energy of side 0 of species 1 (4=1*4+0) with side 1 of species 2 (9=2*4+1). For (*ii*), `eps` is a diagonal matrix of size $(\texttt{spc+1})^2$. Again, row and column 0 are for interactions with empty sites; 1 is for type 1, 2 is for type 2, and so forth. Finally, for (*iii*), `eps` is just a pointer to a number, where that number is the bond energy and it is assumed that there is no interaction energy with empty sites. For routine (*i*), trial moves involve a swap of elements between a site and one of its neighbors as well as new random orientations for both units. The probability is 1/8 for swaps with each nearest neighbor, 1/16 for diagonal neighbors, and 1/4 for the same position. For (*ii*), trial moves are also swaps with neighbors, but this time the probability is 1/8 for each of the 8 neighbors. For (*iii*), an occupied site can only move to an open site; again, the probability is 1/8 for each neighbor.

`MetropolisIsingH` is similar to `MetropolisIsingS`, although it is for a hexagonal lattice. The probability for each trial move is 1/8 of swapping with each of the six neighbors and 1/4 of not moving. In this case, the energy matrix for rotations is size $((\texttt{spc+1})*8)^2$, to account for the additional sides. As before, row and column 0 are for empty sites, 1 to 7 are ignored, 8 to 15 are for type 1 units, and so on. For example, the number at row 11 and column 9 is the interaction of sides 3 and 1 of a type 1 unit (11=1*8+3 and 9=1*8+1).

`MetropolisIsingT` is similar to `MetropolisIsingS`, although it is for a triangular lattice. The probability of each trial move is 1/4 of swapping with each of the three neighbors and 1/4 of not moving. Rotations have been written but only tested very minimally.

`MetropolisIsing` is a very short routine that chooses the correct simulation routine from the shape information in the lattice structure. However, note that the `eps` matrix is lattice shape dependent if rotations are considered, so one can't casually change lattice shapes.

`EnergyAtIsingS` returns the bond energy of a unit of type `spc1`, rotated to orientation `rot1`, if it were to be placed at position (`ix`,`iy`). The current contents of position (ix,iy) is ignored and does not enter into the calculation. `eps` has the same structure as described above for `MetropolisIsingS`, although in this case it should not be divided by *kT*. `rot1` is ignored if the lattice was defined without rotations.

`EnergyAtIsingH` is the same as `EnergyAtIsingS`, except that it is for a hexagonal lattice.

`EnergyAtIsingT` is the same as `EnergyAtIsingS`, except that it is for a triangular lattice.

`EnergyAtIsing` is a very short routine that chooses the correct energy at routine for the lattice shape.

`EnergyIsingS` returns the total bond energy of a square lattice. `eps` has the same structure as described above for `MetropolisIsingS`, although in this case it should not be divided by *kT*.

`EnergyIsingH` is the same as `EnergyIsingS`, except that it is for a hexagonal lattice.

`EnergyIsingT` is the same as `EnergyIsingS`, except that it is for a triangular lattice.

`EnergyIsing` is a very short routine that chooses the correct energy routine for the lattice shape.

`CtClustersIsing` counts the cluster sizes on a lattice, for all lattice shapes. It only considers whether a site is occupied or empty, ignoring both the specific species and orientations in a site. Pass in a lattice, and an array of integers in `count` with total size `ncount`. The routine returns the largest size cluster and also fills in the `count` array with the number of clusters of each size. For example, `count[1]` is returned with the number of monomers, `count[2]` with the number of dimers, and so on. It is not an error if the largest cluster exceeds `ncount`, although then it obviously won't be included in the array.

`DisplayClustersIsing` is an easy way of calling `CtClustersIsing`, because it takes care of array allocation and display. This function prints out the number and concentration of all clusters in the system.

`MinDistIsing` returns the straight line distance on the lattice `lt` from point (`ix`,`iy`) to point (`jx`,`jy`), using the shortest path possible, with periodic boundary conditions. For all lattice types, the length between the centers of nearest neighboring sites is defined as 1.

`RadCorrFnIsing` determines the radial correlation function between species `spc1` and `spc2` on the lattice `lt` by finding the distances between all relevent pairs of units. The result is binned in the array `bins`, which has `nbins` elements, each with width `binsize`. The same unit length convention is used as in `MinDistIsing`. The array needs to have been allocated before running the routine, although it does not need to have been cleared. This function ignores any rotations of units. In the output, `bins[i]` includes all pairs that have distances within: `i*binsize` ≤ *dist.* < `(i+1)*binsize`. At the end of the function, the following conversion is executed:

```
bins[i]/=n1*n2*3.1415926535*((i+1)*(i+1)-i*i)*binsize*binsize/d;
```

In this equation, `bins[i]` starts as a simple count of pairs, `n1` is the total number of `spc1`, `n2` is the total number of `spc2`, and `d` is the total lattice area. The equation converts counts of pairs to a radial correlation function, $g(r)$. It follows the standard definition, which is that $\rho(r)=\rho g(r)$, where $\rho(r)$ is the density of species 2 as a function of r around species 1, ρ is the overall density of species 2, and $g(r)$ is the radial correlation function. If there are no units on the lattice of either `spc1` or `spc2`, the correlation function is meaningless and the function returns with all values of `bins` equal to 0. In using this function, note that the periodic boundaries make it meaningless to measure the correlation function over a distance that is more than half of the smallest lattice dimension, where these values are listed in a previous section.

`DisplayRCFIsing` is an easy way of calling `RadCorrFnIsing`, without having to worry about allocation or freeing of memory, or other details. It prints out the radial correlation function, from radius 0 to half of the smallest lattice dimension, using the `binsize` that was entered. The radius it displays is the center of the bin.

`LatticeSiteDistH` returns the distance to the nearest lattice point on an infinite hexagonal lattice, where this lattice has a point to point spacing of `sp`. Points are at (0,0), (`sp`,0), (`sp`/2,`sp`*√3/2), etc. `xptr` and `yptr` are sent in as pointers to an x,y position and are returned pointing to the location of the nearest lattice site. `ixptr` and `iyptr` values are ignored on input and are returned with the address of the nearest lattice site. The distance to that site is returned by the function. None of the pointers are allowed to be `NULL`.

Internal routines

`clsizeS` is an internal routine used recursively to count the number of uncounted units in the cluster that includes a unit at position (`ix`,`iy`), for a square lattice. Only nearest neighbors are considered as being part of a cluster. When units are counted, their value is replaced by the negative of the actual value to prevent double counting. If site (`ix`,`iy`) is empty or has already been counted, a `0` is returned. This routine uses random numbers to create branches in the counting trajectory in order to reduces the necessary recursion depth.

`clsizeH` is similar to `clsizeS` except that it is for a hexagonal lattice.

`clsizeT` is similar to `clsizeS` except that it is for a triangular lattice.