

# Term Project CS591.003 Fall 02 UNM

S Madhu <madhu@cs.unm>

December 8 2002

## Abstract

This report describes the design and results of a series of experiments in complexity in physics and critical phenomena . Experiments included the following:

1. Ising model simulations where the critical temperature of the phase transition and the correlation length were measured on 2D lattices.
2. Percolation simulations implented using the union find algorithm that estimate the size of the giant cluster and the critical exponents.
3. Experiments observing a phase transition in the complexity of NP-hard satisfiability problems using the davis-putnam procedure on random 3-SAT instances.

This report also doubles as a manual to the software written in [ANSI CL] <sup>1</sup>: An aim of the project was to construct a framework of reusable components for running, visualizing, and logging the results, of experiments. The components are implemented as a bunch of CLOS classes and generic function protocols

**Instructor:** Dr. Chris Moore  
**Dept of Computer Science**  
**University Of New Mexico**

---

<sup>1</sup>Written and tested on current versions (18d+) of CMU Common Lisp[CMUCL]. When available, the Common Lisp Interface Manager [CLIM] is used to display primitive graphics. The plots were done in gnuplot.

# Framework

This chapter documents some of the design of the framework used in implementing and running the experiments in the subsequent chapters.

⇒ `experiment-mixin` [Protocol Class]

The protocol class that represents, in abstract terms, a basic *experiment*. Defined in `experiment-mixin.lisp` The Experiment Protocols are generic functions defined in terms of this class. These protocols specify the tasks involved in running the experiment: setting up parameters and data, actually running the experiment for a number of time steps, logging progress, gathering the results, etc. As is typical in CLOS, the concrete behaviour is implemented in the “protocol methods” that specialize on appropriate subclasses of this class.

The behaviour specified in the standard protocol methods can be augmented by writing methods with `:before` and `:after` qualifiers. Note that The protocol methods on `experiment-mixin` use a variant of the CLOS standard method combination. In this variant, `:before` methods are invoked `:most-specific-last` and `:after` methods are invoked `:most-specific-first`.

⇒ `:stream` [Initarg]

keyword argument to `make-instance`. A stream to log messages from running the experiment. Defaults to `*standard-output*`.

⇒ `:nmax` [Initarg]

keyword argument to `make-instance`. The number of iterations to run the experiment. (`experiment-mixin` houses a slot `steps` whose value is the current iteration of the experiment.)

⇒ `update-experiment` *experiment-mixin* [Generic Function]

⇒ `run-experiment` *experiment-mixin* [Generic Function]

⇒ `simple-experiment-mixin` [Protocol Class]

Protocol class corresponding to an experiment with logging facility. Subclass of `clexperiment-mixin`. If you want to create a new class that behaves like a `simple-experiment-mixin`, it should be a subclass of `simple-experiment-mixin`. All instantiable subclasses of `simple-experiment-mixin` must obey the `simple-experiment-mixin` protocol.

⇒ `:logevery` [Initarg]

The periodic interval, in number-of-iterations, that specifies how often a log message should be written.

⇒ `write-experiment-log` *simple-experiment-mixin* [Generic Function]

Protocol method to write an experiment log.

⇒ `run-experiment-with-gui` *simple-experiment-mixin* [Generic Function]

Protocol method representing the CLIM hacks. Initializes a CLIM application frame which lets you call `run-experiment` through a GUI. Subclasses of `simple-experiment-mixin` would draw on `*lattice-pane*` during experiment protocols to get graphic output. `*lattice-pane*` is bound to the `clim` pane. Conditionally defined in `ising1.lisp`

**notes:** To run an experiment, first create an experiment object using the standard CLOS generic function, `make-instance`, and specify the experiment parameters as keyword arguments in the call to `make-instance`. Then call the `run-experiment`, or `run-experiment-with-gui` on the experiment object.

The standard CLOS protocol method `print-object` is used to display experiment objects.

**Packaging:** To load the the entire system with `mk:defsystem` `cd` to the directory containing the sources and type `(load“sysdcl.lisp”)`.

# The Ising model

The first section describes the implementation of

- single-spin-flip (monte-carlo) dynamics, and the
- Swendsen-Wang dynamics

simulations on 2D ising model The subsequent section describes the experiments which use these.

## 2.1 Implementation

### 2.1.1 The Ising Model

⇒ `basic-ising-mixin` [*Protocol Class*]

This protocol class corresponds to an abstract Ising model. Defined in `ising1.lisp`. If you want to create a new class that behaves like a `basic-ising-mixin`, it should be a subclass of `basic-ising-mixin`. All instantiable subclasses of `basic-ising-mixin` must obey the `basic-ising-mixin` protocol. This class houses slots for the total `spin-sum` and the actual lattice (which will be implemented by its subclasses).

⇒ `:lmax` [*Initarg*]

The lattice dimension, interpreted by the subclass of `basic-ising-mixin`. Default value is specified to be 1024.

⇒ `:J` [*Initarg*]

The exchange interaction. The default value is 1.00

⇒ `:H` [*Initarg*]

The external field. The default value is 0.00

⇒ `:k` [*Initarg*]

The boltzmann factor. The default value is 1.00

⇒ `:Temp` [*Initarg*]

Temperature, a single-float.

⇒ `:initially-random-p` [*Initarg*]

boolean specifying how the lattice should be initialized.

### 2.1.2 2D Ising Model

⇒ `ising2d-mixin` [*Protocol Class*]

This protocol class corresponds to the abstract Ising model in 2 dimensions. Protocol methods on this class support initialization, and correlation computations on the 2D lattice.

⇒ `$initial-conditions` (*obj ising2d-mixin*) [Method]

Protocol method to initialize the lattice. If `initially-random-p` is non-nil, the lattice is populated with a uniformly random distribution. Otherwise, it is populated with negative spins.

⇒ `$magnetization` (*obj ising2d-mixin*) [Method]

Returns the average magnetization of the 2D lattice.

### 2.1.3 Correlation Computations

These methods are defined on the `ising2d-mixin` class in `corr.lisp`

⇒ `$compute-correlations` (*obj ising2d-mixin*) [Method]

For the given lattice computes the correlation function  $C(r)$  and stores it in the `corr` slot, according to the following algorithm:

1. We define  $S(x)$  as a list of `lmax` numbers, each +1 or -1, corresponding to a row of the lattice at equilibrium.
2. We take the Fast Fourier Transform (2.1.6) of  $S(x)$  — this gives a list of `lmax` complex numbers,  $S \sim (w)$  where  $w$  is the frequency.
3. Now, for each  $w$ , take  $|S \sim (w)|^2$ . This gives a third list of numbers, all real, which we will call  $C \sim (w)$ . This happens to be the power spectrum.
4. Now take the inverse FFT (which in fact is the same as the FFT in this case) of  $C \sim (w)$  to obtain the auto correlation function. Essentially this calculates

$$\langle S(x)S(x+r) \rangle = \frac{1}{lmax} \sum_x S(x)S(x+r)$$

since the fourier transform turned out convolution of  $S(x)$  and  $S(-x)$  from the last step into the sum. To obtain  $C(r)$  we subtract  $m^2$  where  $m$  is the average magnetization per site of the lattice,  $(\langle S(x) \rangle \langle S(x+r) \rangle)$

⇒ `$gnuplot-corr` (*obj ising2d*) `&rest args &key &allow-other-keys` [Method]

Plots the correlation data computed in the `corr` slot. Plots  $C(r)$  vs.  $r$  using Derek Smith's excellent Common Lisp interface to `gnuplot`, `gnuplot/gnuplot.lisp`.

This method can accept any of the keyword arguments that the `gnuplot` function takes, and override those defaults.

This method also computes the linear regression of  $\log(r)$  and  $\log(C(r))$ . These values are also plotted on the graph along with the slope of the regression.

This method must be called AFTER the `corr` slot is computed (*i.e.* after calling `$compute-correlations`).

### 2.1.4 Single spin flip dynamics

Our monte carlo algorithm works as follows: At each step of the random walk: choose a lattice site  $(x, y)$  and flip its spin with probability

$$p(\Delta e) = \begin{cases} 1 & \Delta e \leq 0 \\ e^{-\frac{\Delta e}{kT}} & \Delta e \geq 0 \end{cases}$$

where  $T$  is the temperature,  $k$  is the boltzmann constant, (which we arbitrarily fix to be 1.0), and  $\Delta e$  is the change in energy<sup>1</sup> that would happen had we flipped the spin at that site.

⇒ `ising2d` [Protocol Class]

This class is used to implement single spin flip dynamics. Defined in `ising2.lisp`. This is a subclass of `ising2d-mixin`. This class houses slots for caching precomputed values of  $e^{-\Delta e/kT}$  for  $\Delta e = 4J, 8J$ .

⇒ `$initial-conditions` (*obj ising2d*) [Method]

⇒ `$single-spin-flip-dynamics` (*obj ising2d*) [Method]

Implements single flip dynamics (2.1.4) with periodic boundary conditions. We use an optimization based on detailed balance. Conditionally plots the new point if CLIM graphics is available.

---

<sup>1</sup>Note that the total energy of the lattice is given by  $E = H - J \sum_{i,j} s_i s_j$ , where sites  $i, j$  are neighbours,  $s_i$  is the spin ( $\pm 1$ ) at site  $i$ ,  $H$  is an external field (0), and  $J$  is the exchange interaction (another constant which we fix to be 1.0).

## 2.1.5 Swendson-Wang dynamics

The Swendson Wang algorithm works by building clusters of sites and flipping the entire cluster at a time. This algorithm gets to equilibrium much faster than single-flip dynamics for larger lattices. The basic algorithm is as follows: Choose a random site  $i$  with spin  $s_i$ , and build a single cluster of adjacent sites around  $i$ . A neighbouring site  $s_j$  gets added to the cluster with probability

$$p = \begin{cases} 0 & s_i, s_j \text{ are of opposite spins} \\ 1 - e^{-\frac{2J}{kT}} & s_i, s_j \text{ are of identical spins} \end{cases}$$

where  $T$  is the temperature,  $k$  is the Boltzmann constant (as usual, taken as 1).  $J$  is also fixed as 1.

⇒ `ising-swe` [Protocol Class]

This class is used to implement Swendson-Wang Algorithm on a ising 2d lattice. It is defined in `ising3.lisp`. This is a subclass of `ising-2d`. The slot `p/a` is used for caching the precomputed value of  $1 - e^{-2J/kT}$

⇒ `$swendson-wang-dynamics` (*obj* `ising-swe`) [Method]

Implements a step of the single cluster Swendson-Wang algorithm. This selects a random site  $i$  and builds a cluster around it in the `cluster` slot, adding bonds with neighboring sites if they are of the same spin, with probability  $1 - e^{J/kT}$ . Finally we flip the spins of all the sites in

We do processing in a non-recursive algorithm, using the following slots of `ising-swe` for book-keeping: `cluster`, for building the cluster; `stack`, to keep track of which neighbours need to be processed<sup>2</sup> and, `mask`, a bit array to mark sites that already have been added to the cluster.

## 2.1.6 The Fast Fourier Transform

⇒ `fft1` (*data nn isign*) [Function]

Implements a Common Lisp version of the discrete Fast Fourier Transform (adapted from Numerical Recipes in C[NRC]). Defined in `fft.lisp`

Replaces `data[1...2*nn]` by its discrete Fourier transform, if `isign` is input as 1; or replaces `data[1...2*nn]` by `nn` times its inverse discrete Fourier transform, if `isign` is input as -1. `data` is a complex array of length `nn` or, equivalently, a real array of length `2*nn`. `nn` MUST be an integer power of 2 (this is not checked for!)."

The real and imaginary parts of the zero frequency component  $F_0$  are in `data[1]` and `data[2]`; the smallest nonzero positive frequency has real and imaginary parts in `data[3]` and `data[4]`; the smallest (in magnitude) nonzero negative frequency has real and imaginary parts in `data[2*nn - 1]` and `data[2*nn]`.

Positive frequencies increasing in magnitude are stored in the real-imaginary pairs `data[5]`, `data[6]` up to `data[nn - 1]`, `data[nn]`. Negative frequencies of increasing magnitude are stored in `data[2*nn - 3]`, `data[2*nn - 2]` down to `data[nn + 3]`, `data[nn + 4]`.

Finally, the pair `data[nn + 1]`, `data[nn + 2]` contain the real and imaginary parts of the one aliased point that contains the most positive and the most negative frequency.

## 2.2 Results

We use the implementations to:

1. run experiments long enough to reach equilibrium on various lattice sizes.
2. run experiments varying the temperature, to calculate the equilibrium magnetization as a function of Temperature. Estimate  $T_c$  where magnetization goes to zero.
3. using the Fast Fourier Transform, we calculate the correlation function  $C(r)$  for various values of  $T$  and estimate correlation lengths as  $T \rightarrow T_c$  and power law coefficient.

### 2.2.1 Equilibrium Magnetization

Experiments were done on lattice sizes `lmax` = 64, 128, 256, 512, and 1024 for both `sse` and `swe` models, For various values of temperature (from 0.3 to 3.2) in steps of 0.1.

---

<sup>2</sup>The stacks are implemented using data structure `cstack` which can be found at `cstack.lisp`.

We notice that after a certain number of iterations, the average magnetization does not fluctuate much. We assume that the system has achieved equilibrium.

- ▷ We look for how quickly we reach equilibrium and make the following observations. For *ssd* on 256x256 lattices we reach equilibrium after about  $10^7$  iterations. For *ssd* on 512x512 lattices we reach equilibrium after about  $10^8$  iterations. On the other hand with *swe* for 256x256 lattices we reach equilibrium in just 2000 iterations.

The results are available for further analysis in the following directories:

**output.2/** *ssd*  $10^9$  iterations, non-random initial conditions.

**output.1/** *sse* 100000 iterations, initially-random conditions

Each directory has subdirectories {64, 128, 256, ...} which contain files, one bziped file per temperature for which the experiment was carried out. The files contain logs of experiments indicating average magnetization, and a final lattice configuration.

## 2.2.2 Critical temperature

We plot the equilibrium magnetization as a function of temperature for each experiment. For example we show the following figures: Figure 2.1 plots the function for *ssd* on a 512 square lattice (from the data in output.2/512/) with non initial random conditions. Figure 2.2 plots the function for *ssd* on a 1024 square lattice (from the data in output.2/1024/). Figure 2.3 plots the results of (from the data in output.1/256/) for *swe* on 256 square lattices.

- ▷ We look for the critical temperature  $T_c$  at which the equilibrium magnetization goes to zero by averaging out various experiments. Experimentally we notice it is between 2.2 and 2.4. (The theoretical value is 2.269). Finer measurements for *swe* on the 256 lattice indicate it is between 2.28 and 2.29. For both models we can also make the following observations:

If we start with non random initial state, (at  $T < T_c$ ) the equilibrium magnetization is non-zero, and is positive or negative depending on how the spins were initially aligned<sup>3</sup>. If we start off with a random initial state, the equilibrium magnetization (at  $T < T_c$ ) is again non-zero, but can be either positive or negative (randomly). At temperatures above  $T_c$  the equilibrium magnetization converges to zero, corresponding to a random arrangement. This happens independent of the starting state.

## 2.2.3 The Correlation Function

As we approach  $T_c$  from below, we observe large clusters that fluctuate slowly over time. We estimate the length of these correlations using the Fast Fourier Transform.

Using our implementation `fft1` of the Fast Fourier Transform (2.1.6), we calculate the correlation function  $C(r)$  for various values of  $T$ . The method `$compute-corretation` (2.1.3) does the hard work.

- ▷ For example, in figure 2.4 we plot  $C(r)$  vs.  $r$  for a few temperatures on the 256x256 *swe* lattice. We notice that this looks like an exponential, *i.e.*  $C(r) \sim e^{(-r/l)}$  where  $l$  is the correlation length. We also note that as  $T \rightarrow T_c = 2.269$  from below we notice that  $C(r)$  decreases more slowly as a function of  $r$ . Or equivalently that the correlation length diverges.
- ▷ In figure 2.5 we plot the same values on a semilog plot. After logscaling the y-axis the exponentials roughly look like straight lines - as  $r$  grows we quickly get into the noise, so we plot only a few values of  $r$ .

we can estimate the correlation length from the slopes: The slope of the lines for each temperature are equal to  $-1/l$ , for that temperature. We can see that As  $T \rightarrow T_c$  the correlation length  $l$  increases. At  $T = T_c$  we can assume that the correlation length goes to infinity (for large lattices), *i.e.* slope=0. (Warning: again, it is necessary to remember that in the figure, the lines  $T = 2.4$  and  $T = 2.3$  lie above  $T_c$ ).

At the transition,  $C(r)$  follows a power law,  $C(r) \sim r^{-a}$ .

Taking  $T_c$  to be 2.269185, we if plot  $C(r)$  vs.  $r$  we get an exponential curve. We ran both algorithms (swendson-wang and single-spin-flip) at  $T_c$ . Figure 2.6 shows the exponential and figure 2.7 shows the log-log plot for an *swe* experiment. Figure 2.8 shows the exponential  $C(r)$  vs.  $r$  for a few experiments near  $T_c$ .

---

<sup>3</sup> of course for *swe* the sign of the net magnetism fluctuates from step to step, so we consider only the magnitude for equilibrium

Figure 2.1: 512x512 ssd equil. mag

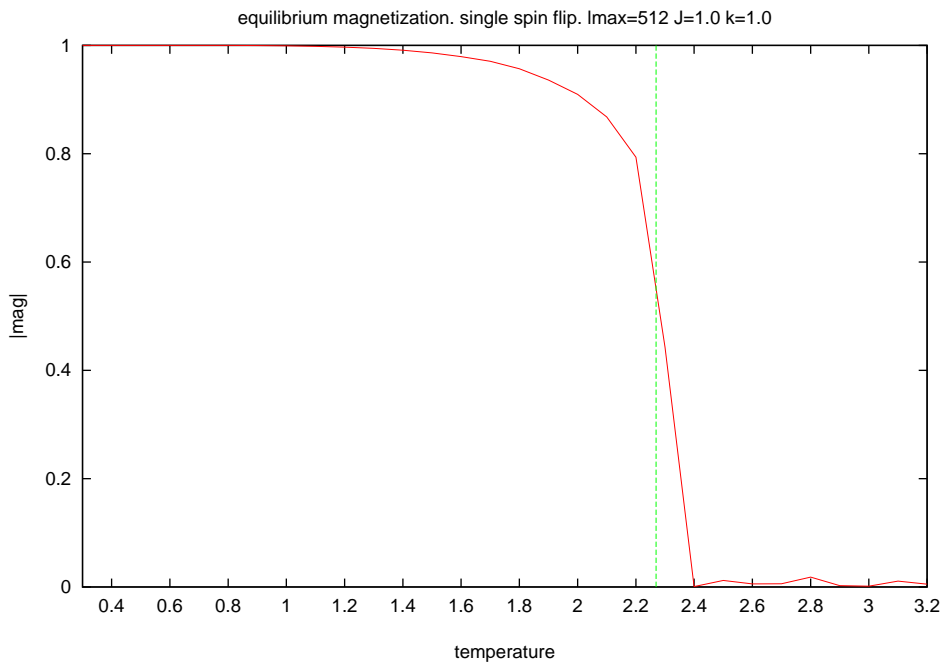


Figure 2.2: 1024x1024 ssd equil. mag

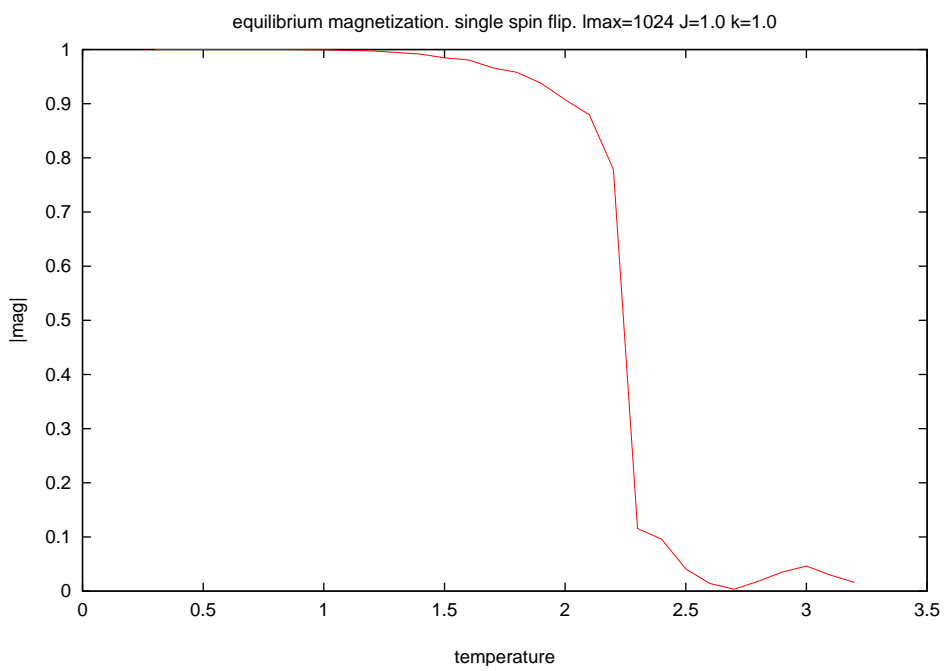


Figure 2.3: 256x256 swe equil. mag

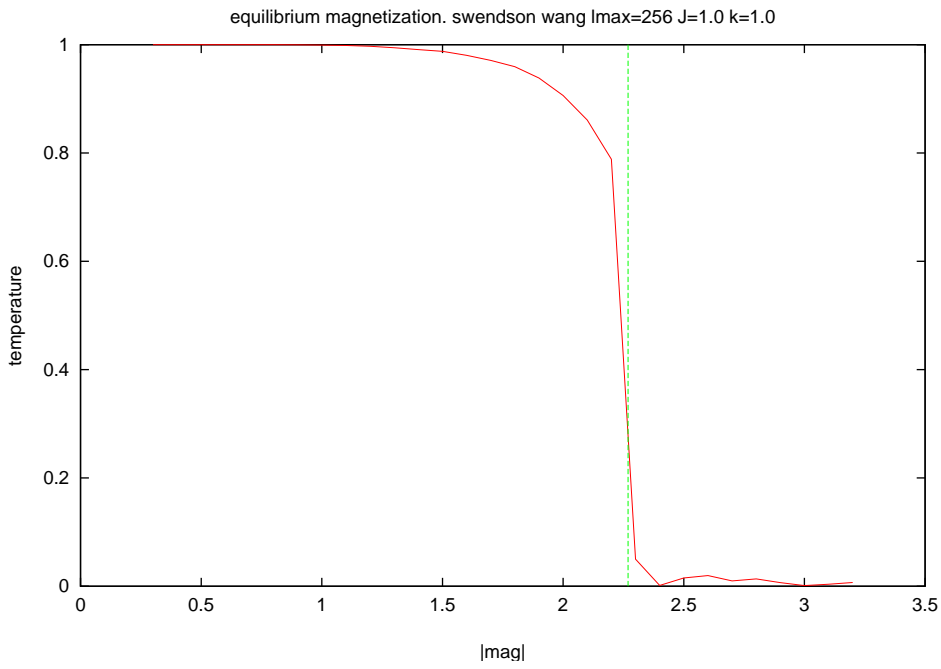


Figure 2.4: 256x256 swe  $C(r)$  vs  $r$

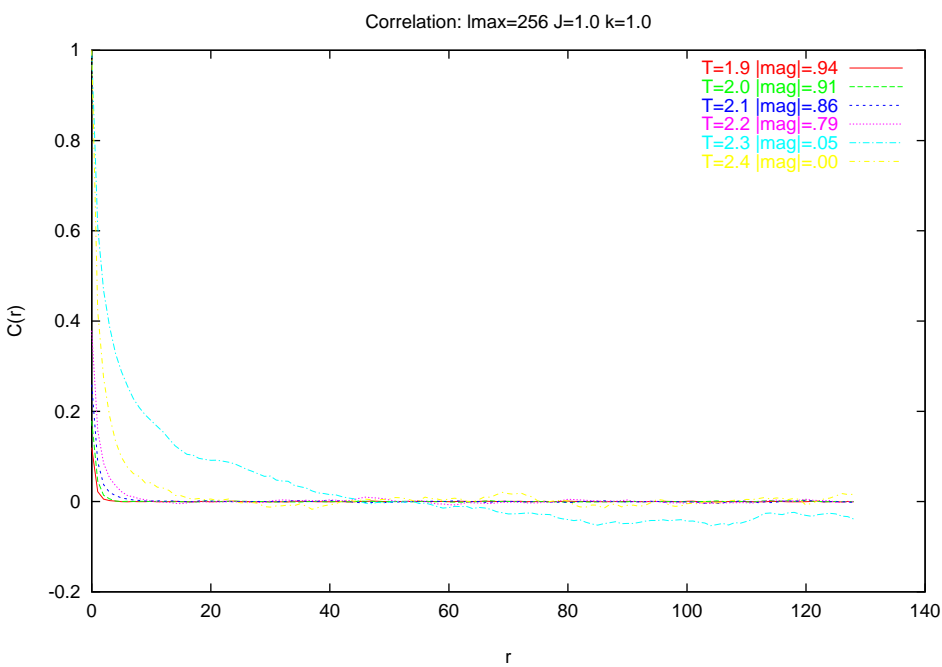


Figure 2.5: 256x256 swe logscale  $C(r)$  vs.  $r$

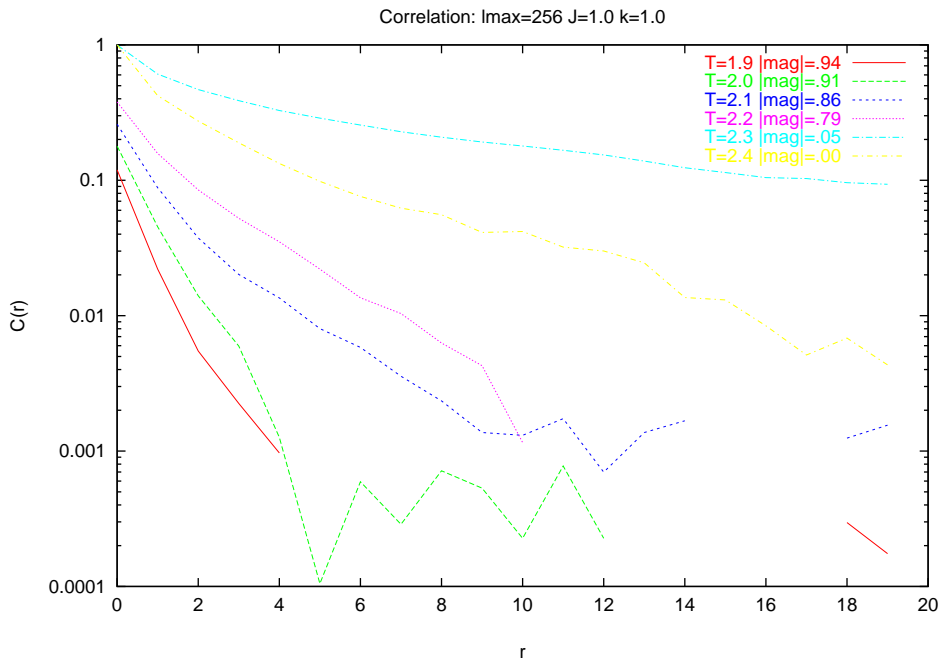


Figure 2.6: 256x256 swe  $C(r)$  vs.  $r$

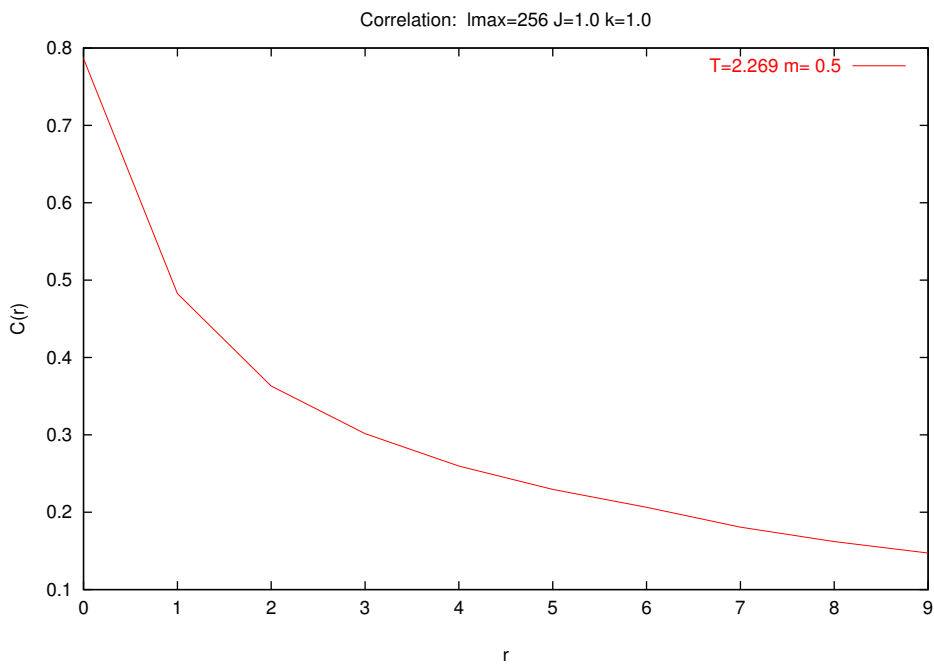


Figure 2.7: 256x256 swe log-log  $C(r)$  vs.  $r$

Correlation:  $I_{max}=256$   $J=1.0$   $k=1.0$

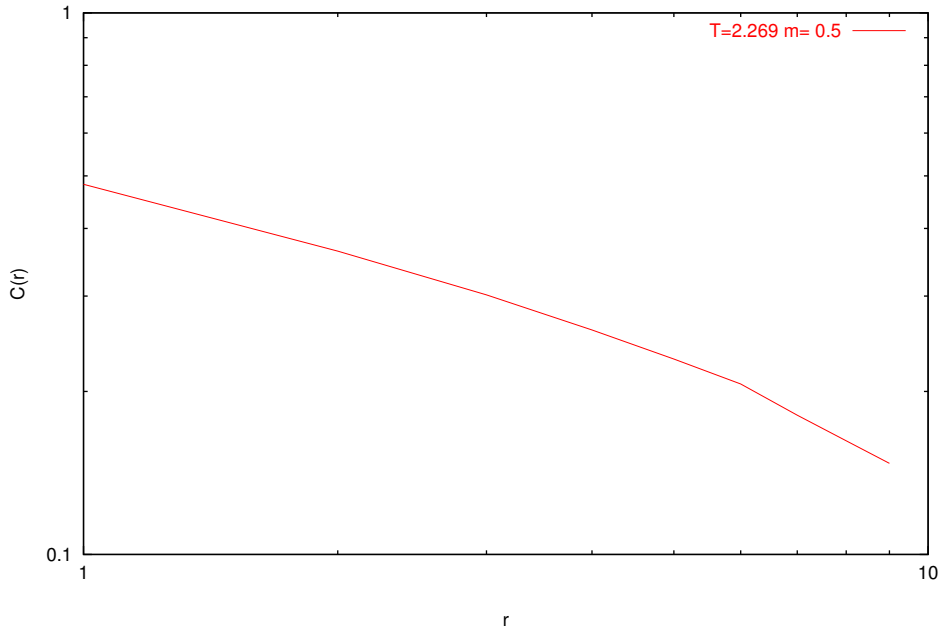
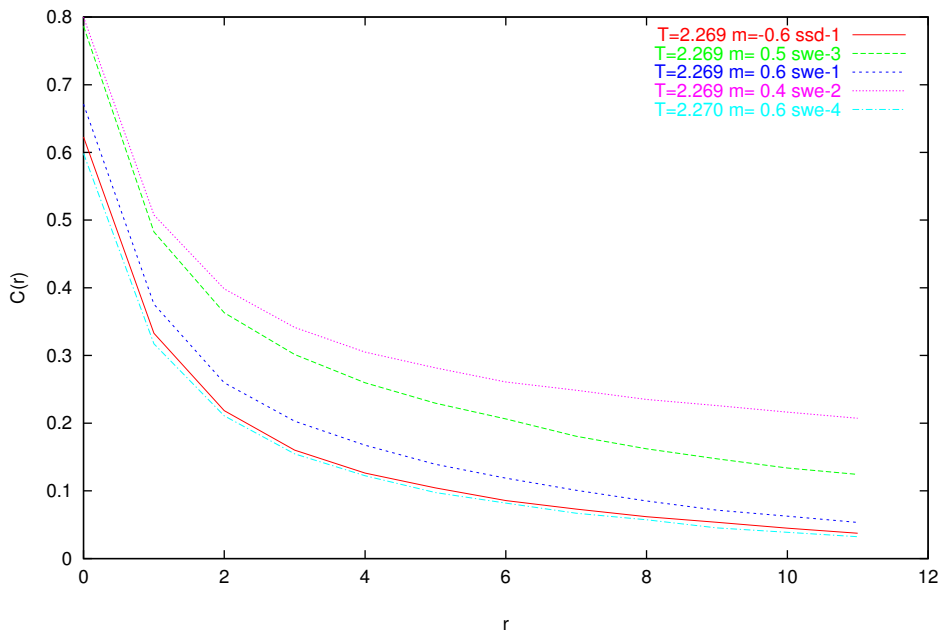


Figure 2.8: 256x256 ssd  $C(r)$  vs.  $r$

Correlation:  $I_{max}=256$   $J=1.0$   $k=1.0$



# Site Percolation in Square Lattices

We use a third party implementation of the union find algorithm `union-find.lisp` by Marco Antonietti from the CLOCC project from soresforge. <http://clocc.sourceforge.net>

## 3.1 Implementation

⇒ `percolation-mixin` [*Protocol Class*]

Support for 2D square lattice which gets occupied with site probability  $p$ . Defined in `perc1.lisp`

⇒ `:lmax` [*Initarg*]

lattice dimension

⇒ `:p` [*Initarg*]

probability with which a site is occupied.

## 3.2 Results

1. The largest lattice size for which one can carry out the simulation in a reasonable amount of time.
2. The average cluster size, maximum cluster size (including the giant component above the transition) and the cluster size distribution as a function of  $p$  on the square lattice. There is a transition.  $p_c$  is estimated at 0.5
3. Below the transition, the cluster size distribution fits an exponential. At  $p_c$  we notice a power law, and estimate the exponent from a log-log plot.
4. Observe percolation in sparse random graphs, in the vicinity of  $d = 1$  where the transition takes place.

# Random 3-SAT

We study how the hardness of an NP hard problem *viz.* 3-SAT varies on random instances. We use an implementation of the The Davis-Putnam procedure (LDPP) written by Mark E. Stickel of SRI International, `dp3/davis-putnam3.lisp`, to solve instances.

## 4.1 Implementation

The functions to generate random instances are defined in `3sat.lisp`.

⇒ `*number-of-variables*` [*Variable*]

The value of `*number-of-variables*` is initially set to 200. This is the default number of literals in random 3-SAT problems that we construct.

A variable is a number between 1 .. `*number-of-variables*`. A negated variable is the negative of that number.

⇒ `random-variable` `&optional` (`n` `*number-of-variables*`) [*Function*]

This function chooses a variable in the range  $1 \dots n$ , or its negation (in the range  $-n \dots -1$ ). Each variable (or negated variable) is selected with the same probability of  $1/2n$ .

⇒ `make-random-3sat-problem` (`r` `&optional` (`n` `*number-of-variables*`) (`*default-print-warnings*` `t`)) [*Function*]

This function returns a `dp-clause-set` structure containing  $m = (rn)$  3 variable clauses. We only create clauses with distinct variables and without tautological occurrences of any variable.

## 4.2 Results

We generate random problems of a given number (200) of variables, feed them to the solver, and measure the execution time. By varying the density,  $r = (\text{number of clauses}/\text{number of variables})$ , we notice a peak in computation time near  $r = 4.2$ .

# Bibliography

- [ANSI CL] [http://www.xanalys.com/software\\_tools/reference/HyperSpec/Front/](http://www.xanalys.com/software_tools/reference/HyperSpec/Front/) ANSI Programming Language Common Lisp Kent M Pitman (Ed.), X3J13 committee, 1994
- [CMUCL] <http://www.cons.org/cmucl/> Public Domain Implementation of Common Lisp (from CMU) on UNIX platforms
- [CLIM] <http://www.stud.uni-karlsruhe.de/~unk6/clim-spec-exp/> CLIM: Common Lisp Interface Manager CLIM II Specification
- [McCLIM] <http://www.bricoworks.com/~moore/clim-paper.pdf> Robert Strandh, Tim Moore presented at the International Lisp Conference Oct 2002. available at <http://clim.mikemac.com/>
- [NRC] Numerical Recipes in C: The art of scientific Computing Cambridge University Press ISBN 0521-43108-5 [http://www.ulib.org/webRoot/Books/Numerical\\_Recipes/bookcpdf.html](http://www.ulib.org/webRoot/Books/Numerical_Recipes/bookcpdf.html)