Computational Physics
Homework 10

Kieran Finn

December 9, 2014

1 Problem a

write a program to implement the single site Metropolis algorithm for the Ising model on an $L \times L$ square lattice with periodic boundary conditions.

1.1 description of the program

Most of the implementation of the program is stored in the function *metro* in *metropolis_functions.py*. This takes as arguments the value of β and L, the size of the grid, as well as the number of iterations and whether to start from a hot or cold initial state. There is also an option to continue a previous simulation, since it runs out of memory for more than about 10,000,000 iterations and hence larger simulations must be split into pieces.

after initialising the energy, magnetisation and a copy of the initial spin state the program enters the main loop. At each iteration the program picks a random site (x,y) and calculates what the difference in energy would be of flipping that site. This is easily done by considering the spins of the four nearest neighbours and it is not necessary to recalculate the energy at each step since the rest of the rest of the lattice stays fixed. This is a great speed up and, in particular, means that the speed of this algorithm per iteration is independent of grid size (although larger grid sizes require more iterations). In fact the slowest part of the program is writing it to a disk, especially in human readable form, which is why most of my data is pickled. This part, as well as reading the data from the disk, is so slow that in most cases it is quicker to calculate the data again from scratch.

1.2 Experimental results

See the video files which show how the Metropolis algorithm updates the lattice.

2 Problem b

make plots of E, M and M^2.

2.1 Experimental results

Figure 1 shows the results of the simulation for a 16×16 grid with $\beta = 0.1$ and a cold start.

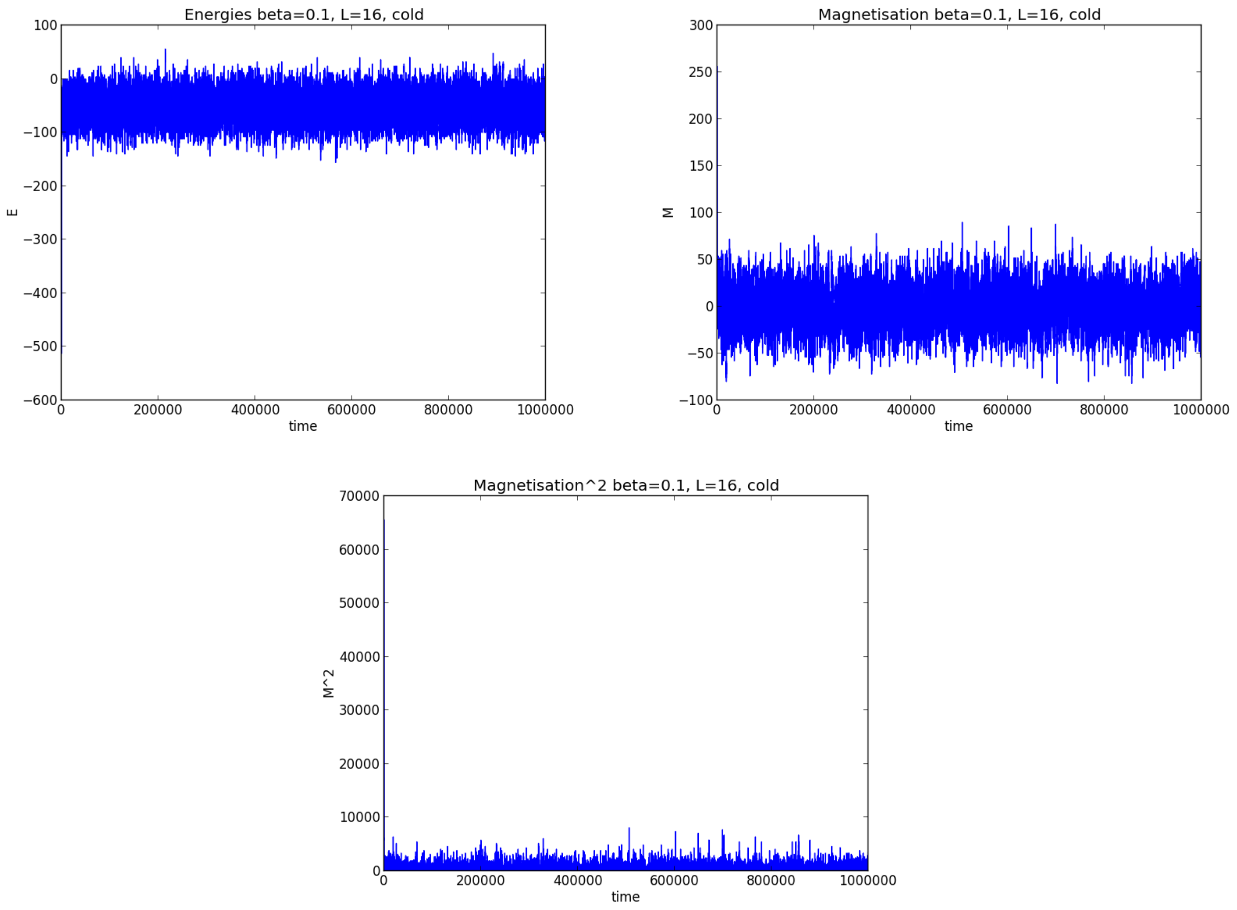

Figure 1: Plots of the how energy, magnetisation and M^2 change in the Metropolis simulation.

We see that the initial state is lost very quickly and we end up with a lot variation about the mean value.

3 Problem c

Analise E, M and M^2. calculate their averages, autocorrelation functions and autocorrelation time.

3.1 Description of the program

Most of the functionality is stored in *autocorrelate_functions.py* with a wrapper *autocorrelate.py*, which covers the initialisation, the number of iterations to drop and the plotting functions.

To calculate tau, the program first subtracts the mean from all data points since it is only the variance we are interested in. It then sums up all autocorrelation functions, $C_{ff}(u)$ within a given window, taking advantage ofthe fact that $C_{ff}(u) = C_{ff}(-u)$. In order to speed up the calculation of autocorrelation functions the program makes use of numpy's libraries. It takes a slice of the data excluding the first u entries and multiplies it with a slice that excludes the last u. This gives an array satisfying $C_i = (X_i - \overline{X})(X_{i+u} - \overline{X})$, the mean of which is the autocorrelation function $C_{ff}(u)$.

The autocorrelation time τ can then be calculated using the formula

$$\tau = \frac{\sum_{u=-M}^{M} C_{ff}(u)}{2C_{ff}(0)}. \tag{1}$$

3.2 Experimental results

Figure 2 shows the autocorrelation function with u, and shows that the window length of 700 is sufficient to calculate τ accurately.

4 Problem d

Consider a variety of values of β and L and make plots of the correlation time τ

4.1 Description of the program

The program used for this task, *calculate_tau.py* is essentially a wrapper, using the functions from *autocorrelate_functions.py*. As with most cases, it was actually quicker to calculate the Metropolis data from scratch rather than load it in from a file.

I also created the program *sweep.py* which created a large dataset of metropolis algorithms at various temperatures and betas

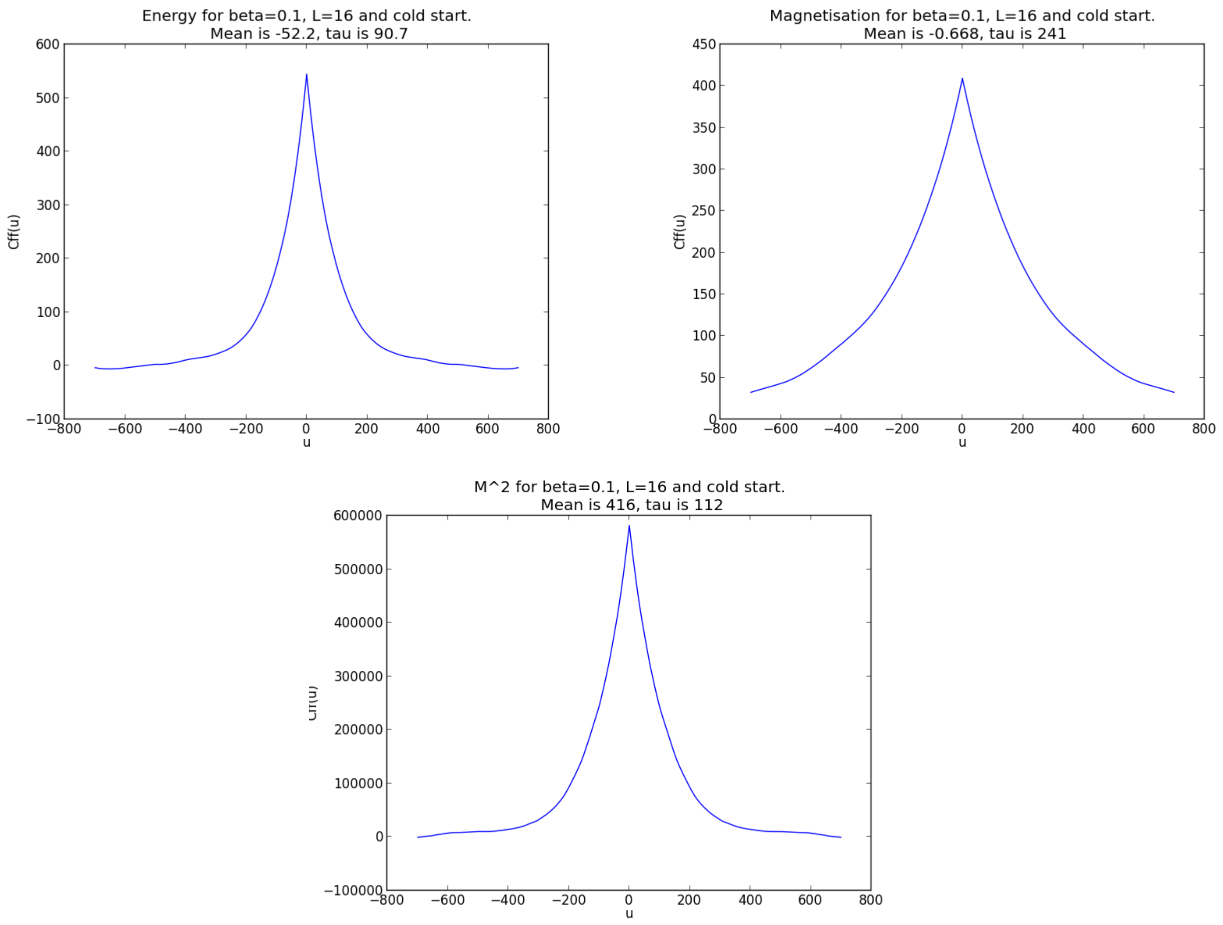

Figure 2: Autocorrelation fuction $C_{ff}(u)$ as a function of u.

4.2 Experimental results

Figures 3,4 and 5 show the resulting autocorrelation times. We see that τ blows up at the critical temperature and decreases away from it on either side. We also see that τ increases with increasing L.

5 Problem e

At the critical temperature calculate the magnetic susceptibility, $\chi = \frac{\langle M^2 \rangle}{L^2}$, as a function of L.

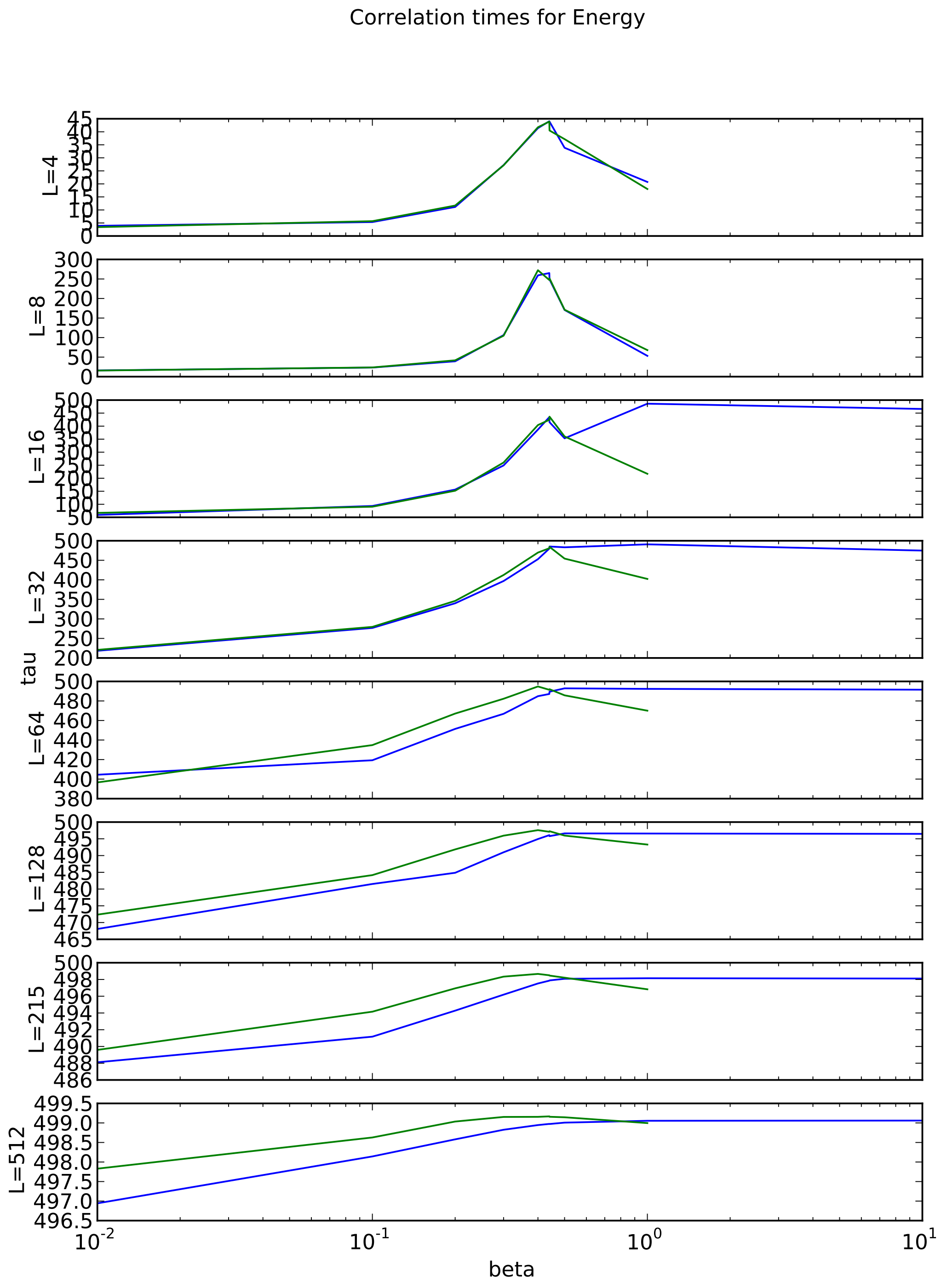

Figure 3: Autocorrelation time for the energy as a function of β for various grid sizes. The blue lines are for a hot start and the green lines are for a cold start.

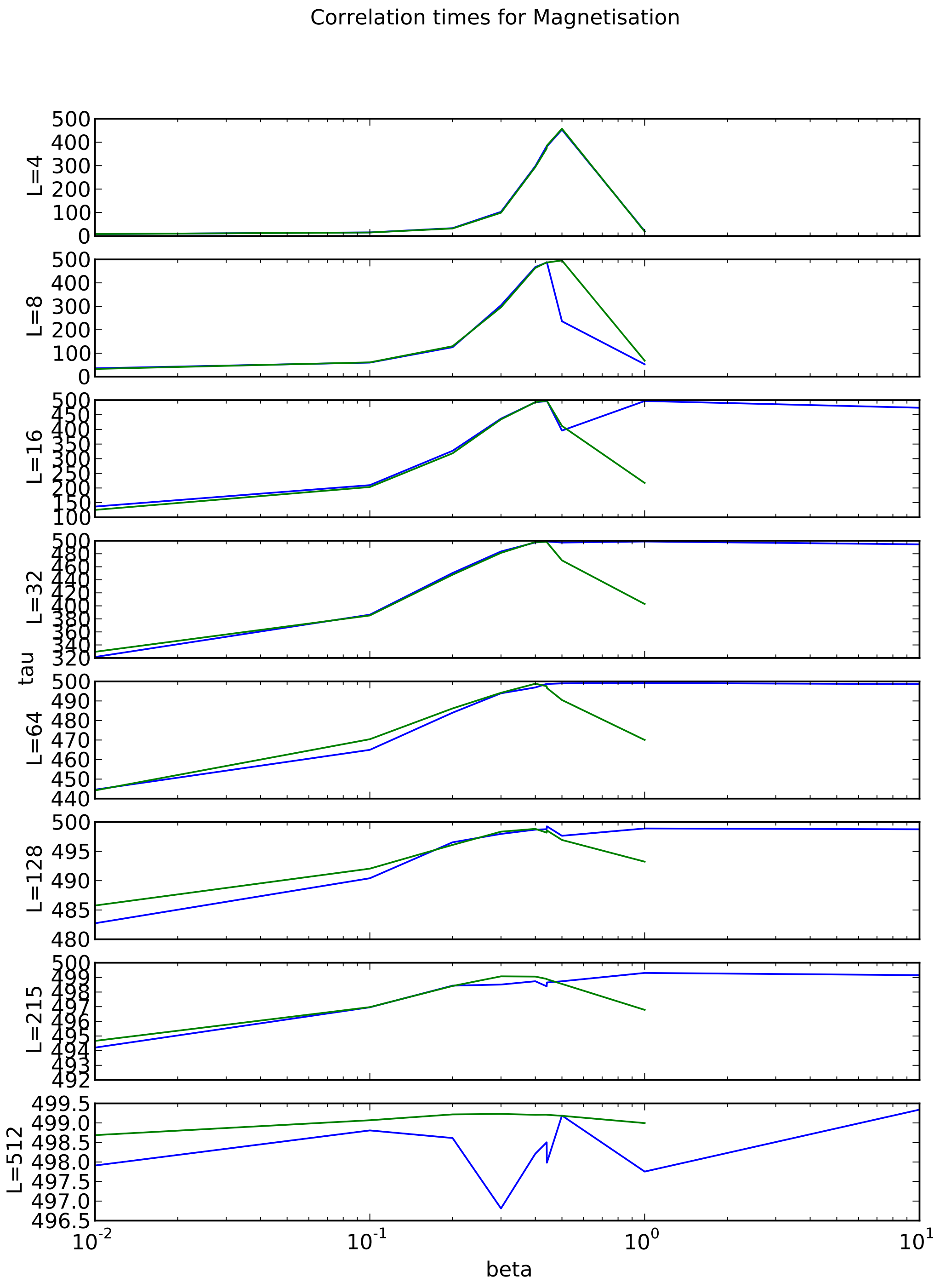

Figure 4: Autocorrelation time for the magnetisation as a function of β for various grid sizes. The blue lines are for a hot start and the green lines are for a cold start.

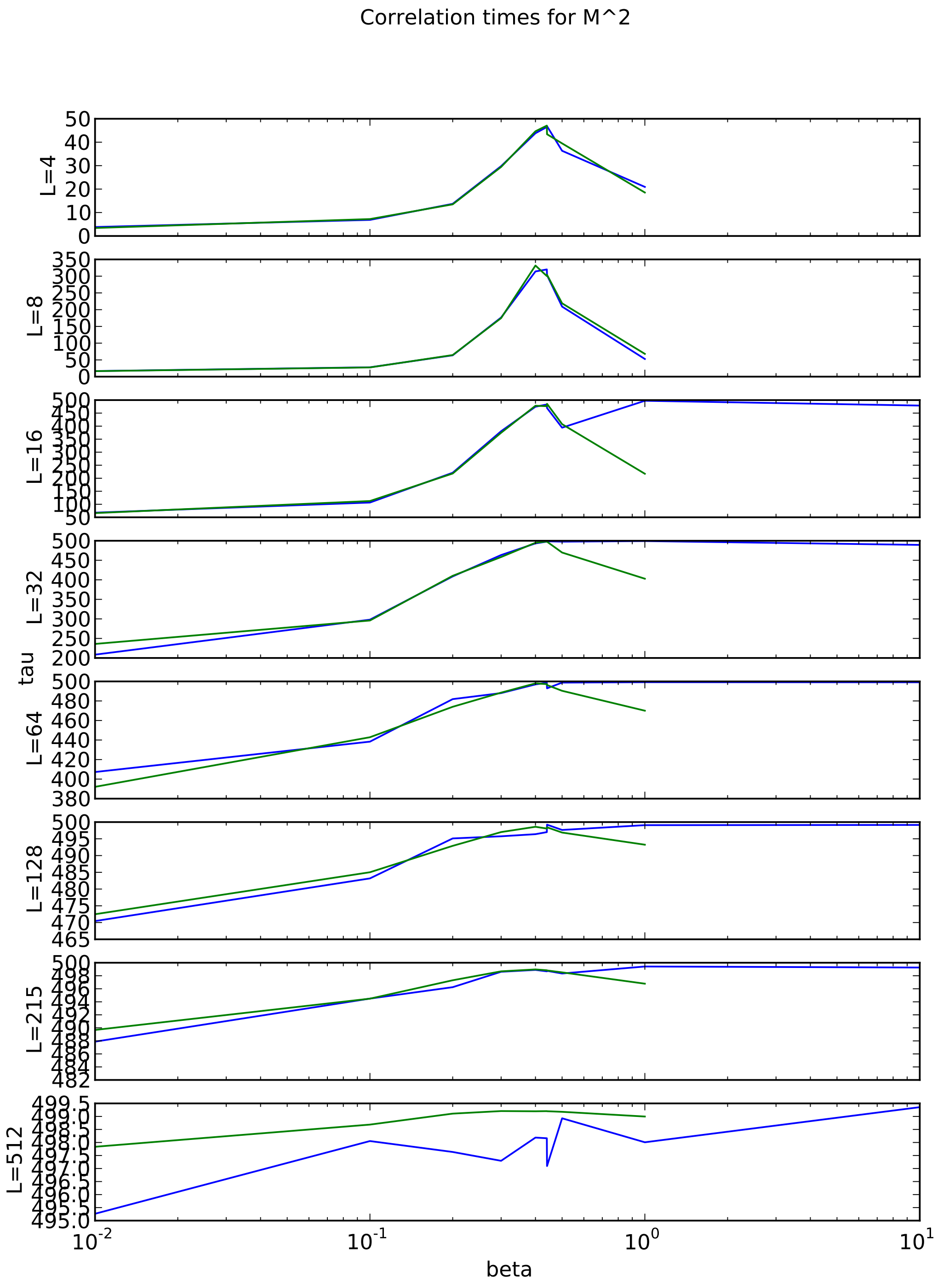

Figure 5: Autocorrelation time for M^2 as a function of β for various grid sizes. The blue lines are for a hot start and the green lines are for a cold start.

5.1 Description of the program

The program used is *susceptability.py* and it mainly relies on the metropolis function stored in *metropolis_functions.py*. The only thing that's changed is that it is generalised to allow a calculation to be split in half. i.e. metro can take the starting state as an input. In this way I am not limited to the $\sim 10,000,000$ iterations that *metro* can handle in one go without running into memory problems. For this algorithm I therefore give a number of iterations proportional to L^2.

5.2 Experimental results

Figure 6 shows the susceptibility as a function of grid size L. We see that the measured value of the critical exponent, $\frac{\gamma}{\nu} = 1.60$ is fairly close to the theoretical value of 1.75.

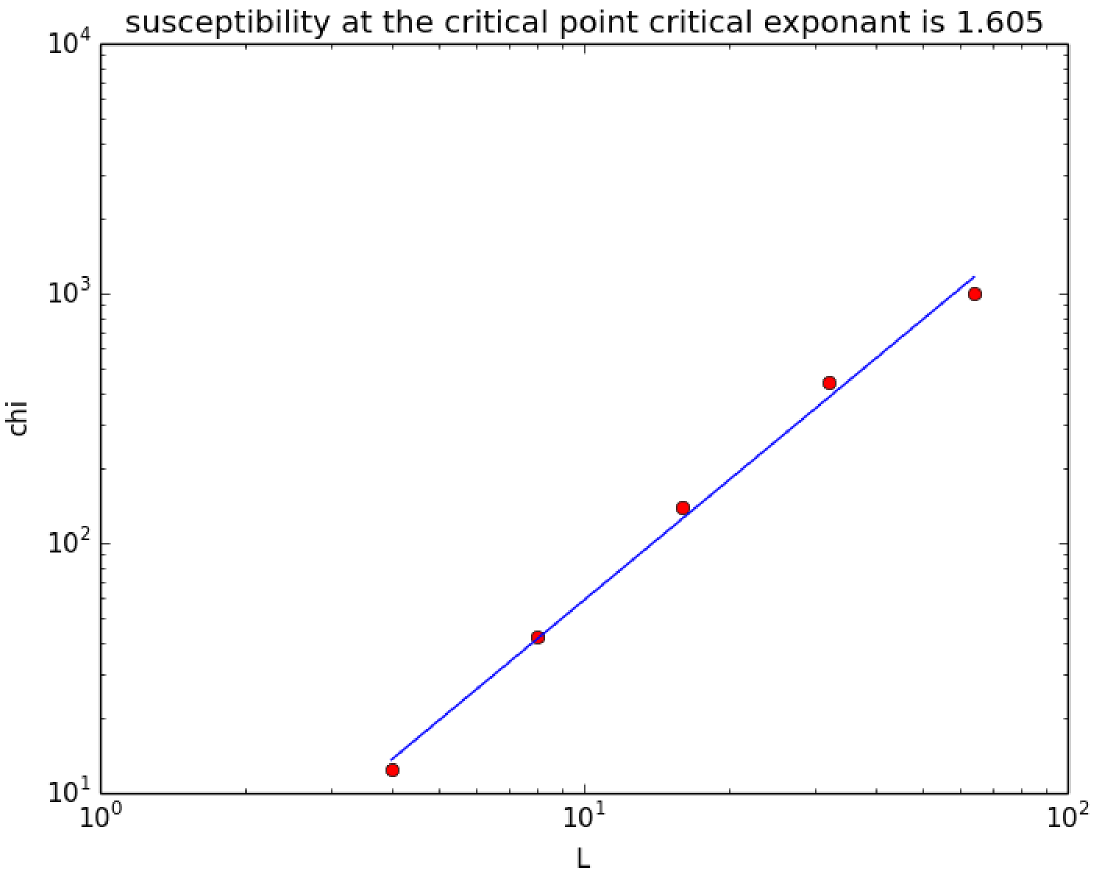

Figure 6: Susceptibility as a function of grid size, L. All simulations were started hot.