FY8904 Assignment 1

Filip Sund

Spring 2019

Notes on code

Most of my code is `C++` and make extensive use of the Armadillo `C++` matrix library[1].

- `Mat` is the base matrix type in Armadillo
- `Col` is the base column vector type in Armadillo
- `mat` is typedef for `Mat<double>`
- `vec` is typedef for `Col<double>`
- `umat` is typedef for `Mat<uword>`
- `imat` is typedef for `Mat<sword>`
- `uvec` is typedef for `Col<uword>`
- `ivec` is typedef for `Col<sword>`
- `uword` is typedef for an *unsigned* integer type
- `sword` is typedef for an *signed* integer type

The minimum width for `uword` and `sword` is 64 bits on 64-bit platforms when using `C++11` and newer standards, and 32 bits when using older `C++` standards.

2 The lattice

Task 2.1

See Listing 1 for a `C++` code snippet that, given a number of sites $N = m \times n$ returns a list of pairs of nodes connected by a bond in a square lattice of size (m, n) and periodic boundary conditions.

[1] See `http://arma.sourceforge.net/docs.html` for documentation.

```
umat makeRectangularLatticeBonds(const uword m, const uword n)
{
    // make bonds for rectangular lattice with dimensions (m, n)

    // allocate matrix with size (2, nBonds)
    // memory is not initialized
    umat bonds = umat(2, 2*m*n);

    for (uword i = 0; i < m; i++) // vertical index
    {
        for (uword j = 0; j < n; j++) // horizontal index
        {
            uword idx = j + i*n; // linear index
            uword right = (j + 1) % n + i*n; // periodic boundary conditions
            uword down = (idx + n) % (m*n);

            bonds(0, 2*idx) = idx;
            bonds(1, 2*idx) = right;
            bonds(0, 2*idx + 1) = idx;
            bonds(1, 2*idx + 1) = down;
        }
    }

    return bonds;
}
```

Listing 1: A C++ code snippet for generating the bonds in a rectangular lattice, using periodic boundary conditions.

I have omitted writing to a file, since I will be calling this code on the fly when the program runs. The code runs fast enough that there's no reason to generate the bonds/lattice ahead of time. Saving to a file is trivially achieved via for example `bonds.save("bonds.csv", arma::csv_ascii)`.

Example output for a square lattice of size $(3, 3)$

```
0  0  1  1  2  2  3  3  4  4  5  5  6  6  7  7  8  8
1  3  2  4  0  5  4  6  5  7  3  8  7  0  8  1  6  2
```

Task 2.2

In Fig. 1 the three different lattices are illustrated, with a simplified "unit cell" indicated in blue. By counting the number of bonds in each unit cell (including 1/2 bonds) we can figure out the number of bonds in a lattice of size N.

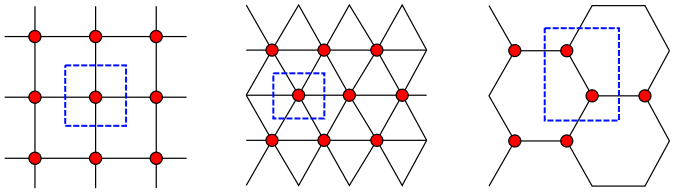


Figure 1: Illustrations of different lattices., with simplified unit cells drawn in blue.

In a *square* lattice we have $4 \times 1/2$ bonds in the unit cell, so for a lattice of size $N = L * L$ we have $2N$ bonds.

In a *trianglular* lattice we have $6 \times 1/2$ bonds in the unit cell, for a total of $3N$ bonds in a lattice of size N.

In a *honeycomb* lattice we have $1 + 3 \times 1/2$ bonds and 2 nodes in the unit cell, for a total of $3N/2$ bonds in a lattice of size N.

Task 2.3

See Listings 2 and 3 for code that generates the bonds for triangular and honeycomb lattices. The numbering schemes used are illustrated in Fig. 2.

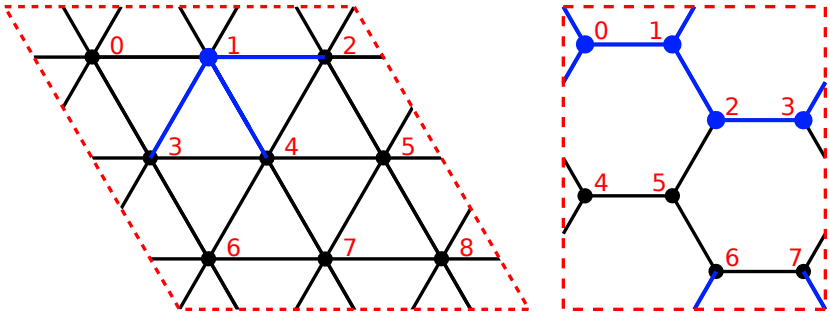


Figure 2: Illustrations a triangular lattice (left) and a honeycomb lattice (right) with periodic boundary conditions and the numbering scheme for the nodes/sites used in this report. The red dashed boundary illustrate the periodic boundaries, and the blue lines indicate the bonds that are added to the list of bonds at each step in Listings 2 and 3.

Example output for the lattices illustrated in Fig. 2 (a 3x4 triangular lattice and a 2x4 honeycomb lattice):

```
// 3x3 triangular lattice
0  0  0  1  1  1  2  2  2  3  3  3  4  4  4  5  5  5  6  6  6  7  7  7
1  4  7  2  5  4  3  6  5  0  7  6  5  0  3  6  1  0  7  2  1  4  3  2

// 2x4 honeycomb lattice
0  0  1  1  2  3  4  4  5  5  6  7
7  1  6  2  3  0  3  5  2  6  7  4
```

```
umat makeTriangularLatticeBonds(const uword m, const uword n)
{
    if (m < 1) throw runtime_error("n must be >= 1");
    if (n < 1) throw runtime_error("n must be >= 1");

    umat bonds(2, 3*m*n); // memory not initialized
    for (uword i = 0; i < m; i++) // vertical index
    {
        for (uword j = 0; j < n; j++) // horizontal index
        {
            uword right = (j + 1)%n + i*n;
            uword downRight = (i + 1)%m*n + j;

            // take modulo of j+n-1 instead of j-1, to avoid modulo of negative
            // number (which will happen at [i=0, j=0]), since this behaves
            // differently than Python
            uword downLeft = (i + 1)%m*n + ((sword(j + n) - 1))%n;

            uword idx = j + i*n;  // linear index
            bonds(0, 3*idx) = idx;
            bonds(1, 3*idx) = right;
            bonds(0, 3*idx + 1) = idx;
            bonds(1, 3*idx + 1) = downRight;
            bonds(0, 3*idx + 2) = idx;
            bonds(1, 3*idx + 2) = downLeft;
        }
    }

    return bonds;
}
```

Listing 2: Program that generates bonds for a triangular lattice with periodic boundary conditions.

```
umat makeHoneycombBonds(const uword m, const uword n)
{
    if (m < 1) throw runtime_error("n must be >= 1");
    if (n % 4 != 0) throw runtime_error("n must be divisible by 4");

    umat bonds(2, 3*m*n/2); // memory not initialized
    uword k = 0; // keep track of position in bonds matrix
    for (uword i = 0; i < m; i++) // vertical index
    {
        for (uword j = 0; j < n; j += 4) // horizontal index
        {
            // 6 bonds per "row" of 4 nodes
            uword idx = j + i*n;  // linear index
            uword node0 = idx;
            uword node1 = idx + 1;
            uword node2 = idx + 2;
            uword node3 = idx + 3;
            uword node;

            // node 0
            node = (i + m - 1)%m*n + (j + n - 1)%n;
            bonds(0, k) = node0;
            bonds(1, k) = node;
            k++;
            bonds(0, k) = node0;
            bonds(1, k) = node1;
            k++;

            // node 1
            node = (i + m - 1)%m*n + (j + 1 + 1)%n;
            bonds(0, k) = node1;
            bonds(1, k) = node;
            k++;
            bonds(0, k) = node1;
            bonds(1, k) = node2;
            k++;

            // node 2
            bonds(0, k) = node2;
            bonds(1, k) = node3;
            k++;

            // node 3
            node = i*n + (j + 3 + 1)%n;
            bonds(0, k) = node3;
            bonds(1, k) = node;
            k++;

            assert(k%6 == 0);
            if (k%6 != 0) throw runtime_error("k%6 != 0");
        }
    }

    return bonds;
}
```

Listing 3: Program that generates bonds for a honeycomb lattice with periodic boundary conditions.

3 Percolating the system

Task 3.1

Since I generate lattices on the fly instead of saving to files, I have omitted code for saving to and reading nodes/bonds from files. But using the Armadillo `C++` library, reading a matrix from a CSV-file is done using

```
umat bonds;
bonds.load("bonds.csv", arma::csv_ascii);
```

NB! Since the default `vec` type in Armadillo is a column vector, I store the bonds in a matrix of size $(2, M)$ instead of $(M, 2)$.

Task 3.2

I use a 32-bit Mersenne Twister from the `C++`-header `random`. This can be initialized to any seed via

```
static std::mt19937 rng(seed);
```

A uniform integer distribution which produces integer values in the (closed) interval $[a, b]$ is created as follows

```
std::uniform_int_distribution<uword> uniform(a, b);
```

And random numbers are generated using

```
for (uword i = 0; i < 10; i++)
{
    cout << uniform(rng) << endl;
}
```

This will generate the same 10 random numbers if the seed is the same. The RNG `rng` can be seeded using a random number as follows

```
static std::random_device rd;
static std::mt19937 rng(rd());
```

(`std::random_device` is platform specific, so if a non-deterministic source of randomness is not available this might not produce non-deterministic seeds. In our case this is not an issue, but check your implementations documentation to see what the case is for any work that requires non-deterministic seeds.)

Task 3.3

Code for shuffling the list of bonds can be seen in Listing 4. This code is not very optimized, since we create a new random distribution every step of the loop, but it does not appear to be any kind of bottleneck compared to the rest of the program.

```cpp
void shuffleBonds(umat& bonds)
{
    for (uword i = 0; i < (bonds.n_cols - 1); i++)
    {
        std::uniform_int_distribution<uword> uniform(i + 1, bonds.n_cols - 1);
        uword j = uniform(rng);

        bonds.swap_cols(i, j);
    }
}
```

Listing 4: A C++ code snippet for shuffling the columns in a Armadillo matrix, utilizing the `.swap_cols` function from Armadillo. Bonds are stored in a $(2, M)$ matrix.

Task 3.4

The status of each node is kept in a `ivec` (signed integer vector), and is initialized to -1 using

```
imat sites = zeros<ivec>(m*n) - 1
```

Task 3.5

I have implemented the root finding function as a member of a class `Sites`, defined as `uword findRoot(const uword site)`. The implementation can be seen in Listing 5.

```cpp
uword Sites::findRoot(const uword site)
{
    // recursive function
    if (m_sites(site) < 0)
    {
        // if sites[i] is negative, this is a root node with size sites[i]
        // return the index of the root node
        return site;
    }
    else
    {
        // this is not a root node, but a node belonging to a cluster with root
        // node sites[i]
        // call findroot again, update sites[i] to point to the root node, and
        // return the root node
        m_sites(site) = sword(findRoot(uword(m_sites(site))));

        return uword(m_sites(site)); // return after updating
    }
}
```

Listing 5: Member function of the class `Sites` that finds the root node of a site `site`. `m_sites` is a member variable of `Sites` with type `ivec`.

Task 3.6

The function for activating a bond (a member of the class `Sites`) is given in Listings 6 and 7.

```
void Sites::activate(const uvec& bond)
{
    uword node1 = bond(0);
    uword node2 = bond(1);
    uword root1 = findRoot(node1);
    uword root2 = findRoot(node2);

    if (root1 == root2)
    {
        // nodes belong to the same cluster -- do nothing
    }
    else
    {
        mergeClusters(root1, root2);
    }
}
```

Listing 6: Member function of the class `Sites` that activates a bond `bond`. See Listing 7 for the implementation of the function `mergeClusters`.

```
void Sites::mergeClusters(const uword root1, const uword root2)
{
    uword larger = root1;
    uword smaller = root2;
    if (m_sites(root2) < m_sites(root1)) // if root2 cluster is larger
    {
        larger = root2;
        smaller = root1;
    }

    // add size of smaller cluster to larger cluster
    m_sites(larger) += m_sites(smaller);

    // point root node of smaller cluster root node of larger cluster
    m_sites(smaller) = sword(larger); // use explicit cast to sword to avoid
                                      // warnings  about implicit casts

    updateLargestCluster(larger);
}
```

Listing 7: Member function of the class `Sites` that merges two clusters with root nodes `root1` and `root2`. See Listing 8 for implementation of `updateLargestCluster`.

Task 3.7

To keep track of the largest cluster I use the function `updateLargestCluster` given in Listing 8.

```
void Sites::updateLargestCluster(const uword root)
{
    if (uword(abs(m_sites(root))) > m_sizeOfLargestCluster)
    {
        m_largestClusterRoot = root;
        m_sizeOfLargestCluster = uword(abs(m_sites(root)));
    }
}
```

Listing 8: Member function of the class `Sites` that updates the size and root node of the largest cluster. `mm_largestClusterRoot` and `m_sizeOfLargestCluster` are members of `Sites` with type `uword`.

Task 3.8

To create an image I prefer using `Python`, so first I have to export an image from the `C++`-program. This is done via the function `makeImage` that creates a binary image from the biggest cluster. See Listing 9 for the implementation of this function. The matrix is saved to a CSV-file using

`image.save(<filename>, arma::csv_ascii)`

```
umat Sites::makeImage()
{
    umat image = zeros<umat>(m_m, m_n); // initialize to zero
    for (uword node = 0; node < m_sites.n_elem; node++)
    {
        if (findRoot(node) == m_largestClusterRoot)
        {
            // this node belongs to largest cluster
            uword i = node/m_n;
            uword j = node%m_m;
            image(i, j) = 1;
        }
    }

    return image;
}
```

Listing 9: Member function of the class `Sites` that creates a binary image illustrating the largest cluster.

The images are then produced using `matplotlib`'s `imshow` with the argument `aspect="equal"` to retain the aspect ratio. The result is shown in Fig. 3. From this it seems like a the spanning cluster appears somewhere between $p = 0.45$ and $p = 0.50$.

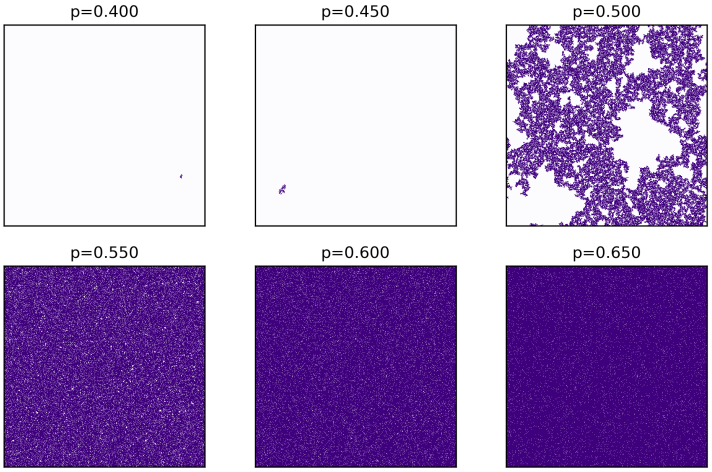


Figure 3: Illustration of the largest cluster at different probabilities p, for a lattice with size $(2000, 2000)$.

Task 3.10

To track the giant component P_∞, the average cluster size $\langle s \rangle$ and the susceptibility χ I rewrote `mergeClusters` and `updateLargestCluster`. See Listings 10 and 11 for the modified functions.

```
void Sites::mergeClusters(const uword root1, const uword root2)
{
    uword larger = root1;
    uword smaller = root2;
    if (m_sites(root2) < m_sites(root1)) // if root2 cluster is larger
    {
        larger = root2;
        smaller = root1;
    }

    // subtract the square of the size of the two clusters that are going to be
    // merged from the sum
    m_sizeSum -= pow2(m_sites(larger)) + pow2(m_sites(smaller));

    // add size of smaller cluster to larger cluster
    m_sites(larger) += m_sites(smaller);

    // point root node of smaller cluster root node of larger cluster
    m_sites(smaller) = sword(larger);

    // add the square of the size of the merged cluster to the sum
    m_sizeSum += pow2(m_sites(larger));

    updateLargestCluster(larger);

    if (m_sizeOfLargestCluster == m_N)
    {
        // avoid division by zero
        m_averageSquaredSize = 0;
    }
    else
    {
        // TODO: check signed-ness of this calculation
        m_averageSquaredSize = (m_sizeSum - pow2(m_N*m_giantComponent))
                /(m_N*(1 - m_giantComponent));
    }
}
```

Listing 10: Member function of the class `Sites` that merges two clusters. This is an updated version of the one shown in Listing 7, with the difference highlighted with dark grey background color.

```
void Sites::updateLargestCluster(const uword root)
{
    if (uword(abs(m_sites(root))) > m_sizeOfLargestCluster)
    {
        m_largestClusterRoot = root;
        m_sizeOfLargestCluster = uword(abs(m_sites(root)));
        m_giantComponent = m_sizeOfLargestCluster/double(m_N);
        m_giantComponentSquared = pow2(m_sizeOfLargestCluster/double(m_N));
    }
}
```

Listing 11: Member function of the class `Sites` that updates various parameters we want to track. This is an updated version of the one shown in Listing 8, with the difference highlighted with dark grey background color.

Task 3.11

See Listing 15 for the main body of the finished program, which includes a loop to sample over several iterations.

4 Convolution

Task 4.1

To avoid integer overflow we will calculate the logarithm of the convolution equation and taking the exponent in the end. We have

$$Q(q) = \sum_{n=0}^{M} \binom{M}{n} q^n (1-q)^{M-n} Q_n \tag{1}$$

$$\begin{aligned} \Rightarrow Q(q) &= \sum_{n=0}^{M} \exp\left(\log\left[\binom{M}{n} q^n (1-q)^{M-n} Q_n \right] \right) \\ &= \sum_{n=0}^{M} \exp\left(\log\binom{M}{n} + n\log(q) + (M-n)\log(1-q) + \log(Q_n) \right) \end{aligned}$$

The first factor can be rewritten as follows

$$\begin{aligned} \log\binom{n}{k} &= \log\left(\frac{n!}{k!(n-k)!} \right) \\ &= \sum_{i=1}^{n} \log i - \sum_{i=1}^{k} \log i - \sum_{i=1}^{n-k} \log i \end{aligned} \tag{2}$$

Task 4.2

If we study Eq. (2) we see that we can optimize the program a lot by tabulating the values of

$$\sum_{i=1}^{n} \log i \tag{3}$$

for all numbers n up to the largest number of bonds we are going to simulate. This is tabulated before the loop over iterations and lattice sizes via the short code shown in Listing 12. This is again implemented as part of a class `LogBinomCoeffGenerator` which is used to quickly evaluate $\log\binom{n}{k}$. The details of this class are shown in Listing 13.

Using the `LogBinomCoeffGenerator` we will tabulate $\log\binom{N}{q}$ for each lattice size N and each occupation number q inside the loop over lattice sizes. Tabulating the sums in Listing 12 takes around 70 milliseconds for lattice size $L = 1000$.

```
vec LogBinomCoeffGenerator::tabulateLogSum(const uword max)
{
    vec logSum(max + 1);
    for (uword i = 1; i <= max; i++)
    {
        logSum(i) = logSum(i-1) + log(i);
    }

    return logSum;
}
```

Listing 12: A function for tabulating $\sum_{i=1}^{n} \log i$ for all numbers up to `max`. This is defined as a static member function of `LogBinomCoeffGenerator` and the result stored in a const member variable, as shown in Listing 13.

```
#include <cassert>
#include <armadillo>

using namespace std;
using namespace arma;

class LogBinomCoeffGenerator
{
public:
    LogBinomCoeffGenerator(const uword max):
        m_logSum(tabulateLogSum(max))
    {}

    double operator()(const uword n, const uword k) const
    {
        // returns log of binomial coefficient (log(n choose k))
        if (!(n < m_logSum.n_elem)) throw
        ↪  runtime_error("!(n < m_logSum.n_elem)");
        assert(n < m_logSum.n_elem);
        if (!(n >= k)) throw runtime_error("!(n >= k)");
        assert(n >= k);

        double coeff = m_logSum(n) - m_logSum(k) - m_logSum(n - k);
        return coeff;
    }

private:
    const vec m_logSum; // this will contain sum_{i=1}^n log i
```

Listing 13: A class used to quickly evaluate $\log \binom{n}{k}$ using tabulated values. The implementation of `tabulateLogSum` is shown in Listing 12.

Task 4.3

The convolution is done by a double loop over the probabilities we want to sample, and the number of bonds/occupational numbers. The implementation is shown in Listing 14.

Since we use tabulated binomial coefficients and a calculate as many log values as we can outside the loops, this function is very efficient, using about 3 s to

```
vec Results::calcResults(const vec& Q, const vec& pvec)
{
    const vec logQ = log(Q); // calculate logarithm outside loops

    const uword nps = pvec.n_elem; // number of probabilities
    const uword M = Q.n_rows; // number of bonds
    vec Qp(nps); // results

    // loop over all probabilities we want to get results for
    for (uword i = 0; i < nps; i++)
    {
        const double p = pvec(i);
        const double logp = log(p); // calculate logarithms outside loop
        const double logOneMinusP = log(1 - p);

        // calculate Q(p)
        double Qpi = 0;
        for (uword n = 0; n < M; n++) // loop over all Q's
        {
            const double a = m_logBinomCoeff(n); // tabulated values
            const double b = n*logp;
            const double c = (M - n)*logOneMinusP;
            const double d = logQ(n);

            const double logTerm = a + b + c + d;
            Qpi += exp(logTerm);
        }
        Qp(i) = Qpi;
    }

    return Qp;
}
```

Listing 14: A function used to calculate the convolution of a quantity Q using Eq. (1).

calculate the convolution for 1000 values of p for a single quantity with lattice size $L = 100$, and 130 s for lattice size $L = 1000$.

The finished program uses around 15 minutes to simulate lattices 100, 200, 500, 700, and 1000, with 200 samples of each sample and convolutions at 1e4 different probabilities.

The program currently calls `calcResults` once for each quantity, so it could be optimized further by calculating for multiple quantities inside the loop, but most of the cpu time is spent elsewhere, so this isn't a huge issue.

NB! Averages of P_∞ etc. are calculated before the convolution, to save on memory usage (at lattice size $L = 1000$ we quickly run into issues with memory usage). This is done very conveniently by using the Armadillo class `running_stat_vec`, which calculates running statistics on the fly. A minimal example is as follows

```
running_stat_vec<vec> stats;
vec sample;
for(uword i=0; i<10000; ++i)
{
```

```
    sample = randu<vec>(5);
    stats(sample);
}
cout << "Mean values = " << endl << mean_values = stats.mean();
```

The details of how this is used in my program is shown in Listing 17.

5 Analysis of results

Task 5.1

See Figs. 4 to 6 for plots of the giant component, average cluster size and susceptibility as function of probability q, for different lattice sizes. 1000 iterations were performed for each lattice size, and the different measurements were sampled at 1000 different probabilities q.

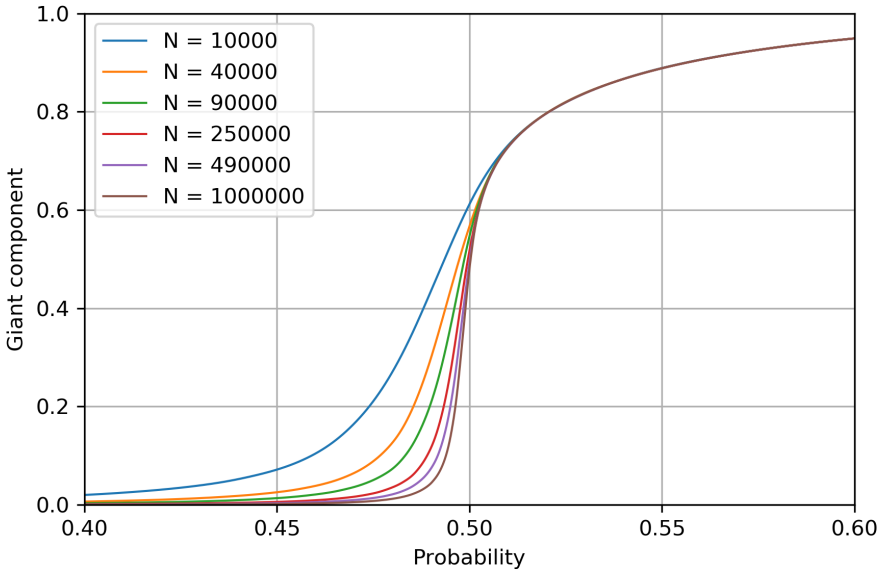


Figure 4: Plot of the giant component P_∞ for square lattices.

Task 5.2

From the sharp change in the giant component seen in Fig. 4, and the peaks observed in Figs. 5 and 6 I guess that the critical probability p_c will be somewhere around 0.5, perhaps a bit over 0.5.

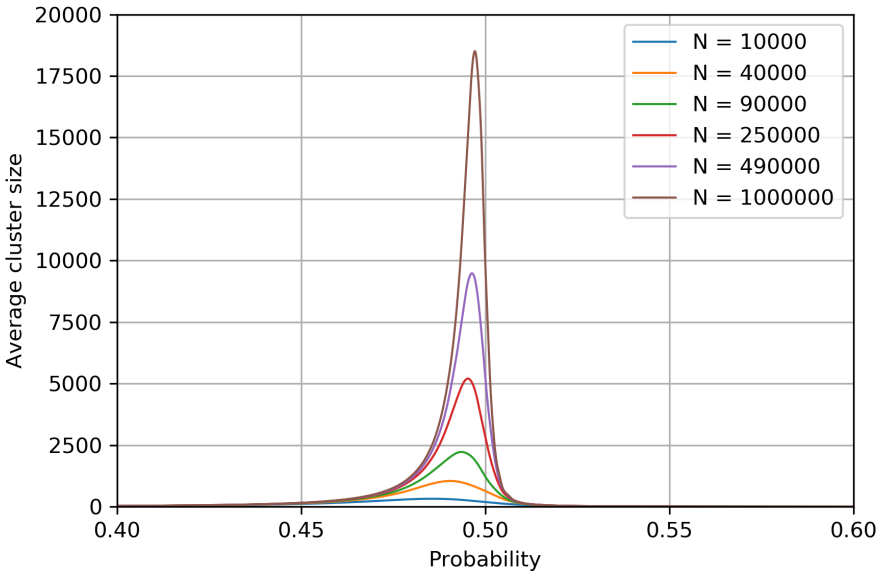


Figure 5: Plo of the average cluster size $\langle s \rangle$ for square lattices.

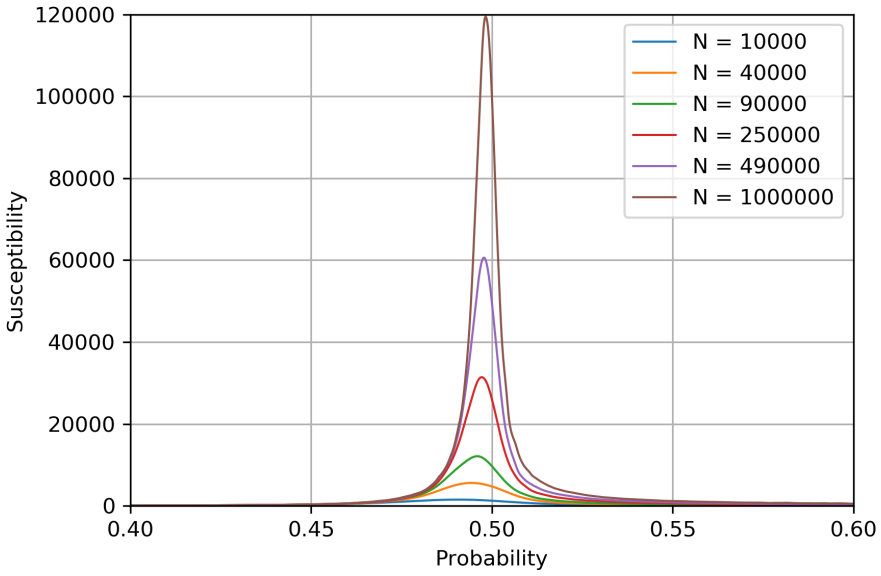


Figure 6: Plot of the susceptibility χ for square lattices.

Task 5.3

See Fig. 7 for a plot of the correlation coefficient R^2 against the probability q. The probability that maximizes R^2 is found to be $p_c = 0.5000$

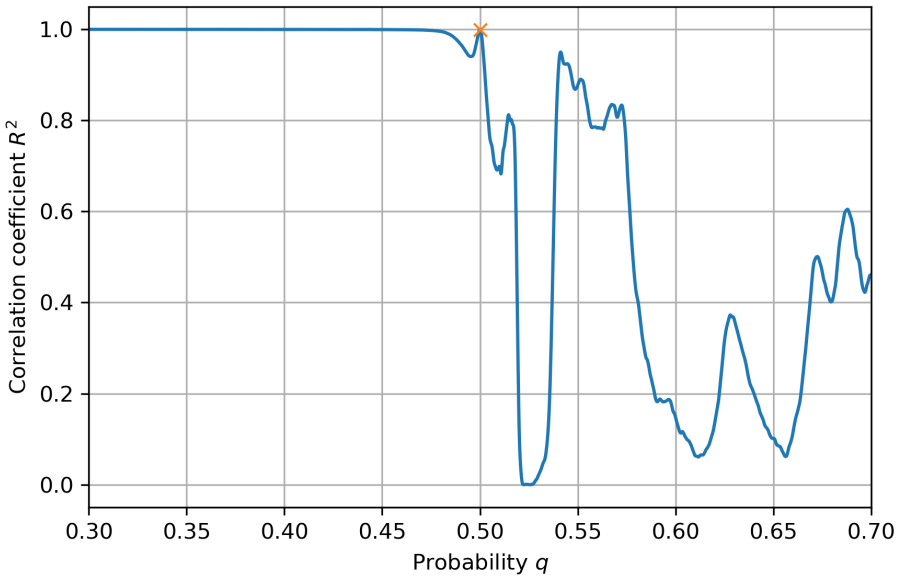


Figure 7: Plot of the correlation coefficient R^2 for different probabilities q. The maximum point is found at $q = 0.5000$, and is indicated with a orange cross.

In Fig. 8 the giant component P_∞ is plotted against the system size ξ at $q = p_c$, on a log-log scale. The linear fit has slope of -0.105820, which is equivalent to $-\beta/\nu$.

Task 5.4

In Fig. 9 is a plot of the maximum average cluster size $\max(\langle s \rangle)$ against the system size ξ, for each lattice size, on a log-log scale. The slope, which is equivalent to γ/ν, was found to be 1.752806.

In Fig. 10 is a plot of $|q_{\max} - p_c|$ against ξ on a log-log scale. The slope, which is equivalent to $-1/\nu$ is found to be -0.725174.

For the three critical exponents β, γ and ν we then have

$$-\frac{1}{\nu} = -0.725174$$
$$-\frac{\beta}{\nu} = -0.105820$$
$$\frac{\gamma}{\nu} = 1.752806$$

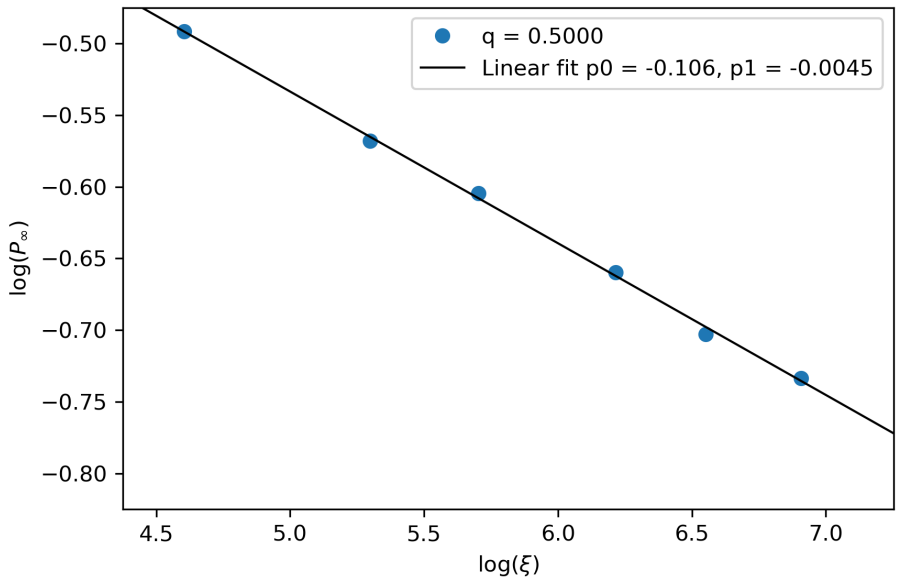


Figure 8: Plot of $\log(P_\infty)$ against $\log(\xi)$.

which leaves us with

$$\begin{aligned}\nu &= 1.3790\\ \beta &= 0.105820\nu = 0.1459\\ \gamma &= 1.752806\nu = 2.4170\end{aligned}$$

These values correspond well to the exact values, as given in Table 1. Increasing the number of iterations and the number of sampled probabilities would probably bring these numbers close to the exact values (we used 1000 iterations for each lattice size).

Table 1: Exact critical exponents for bond percolation on different lattices.

	Square	Triangular	Honeycomb
p_c	0.5	$2\sin(\pi/18) \approx 0.347$	$1 - 2\sin(\pi/18) \approx 0.653$
β	$5/36 = 0.13\overline{8}$	$5/36$	$5/36$
γ	$43/18 = 2.3\overline{8}$	$43/18$	$43/18$
ν	$4/3 = 1.\overline{3}$	$4/3$	$4/3$

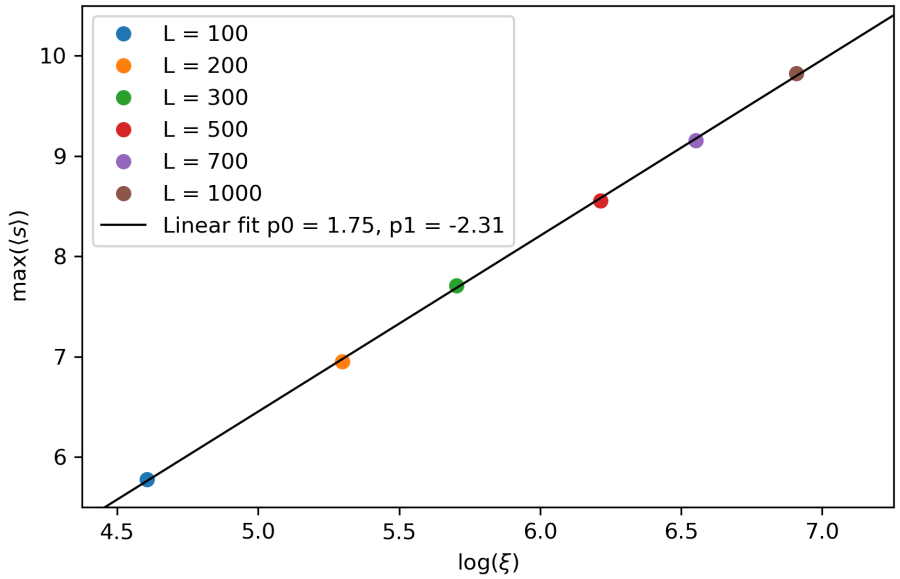


Figure 9: Plot the maximum value of $\langle s \rangle$ for each cluster size, against $\log(\xi)$.

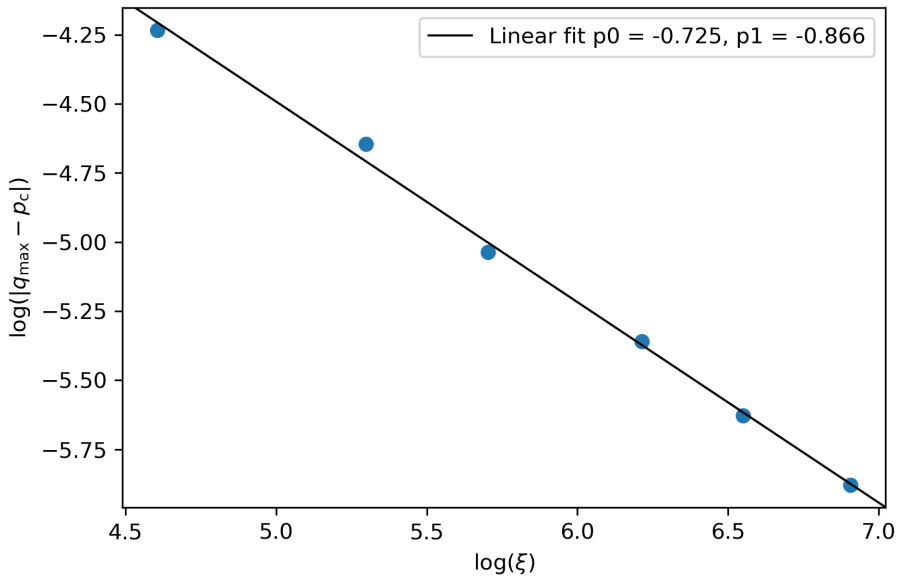


Figure 10: Plot of $\log(|q_{\max} - p_c|)$ against $\log(\xi)$. $q_{\max}$ is the probability at which $\langle s \rangle$ takes its max value.

Task 5.5

For the triangular lattice and the honeycomb lattice we find the critical exponents listed in Table 2. The simulations were performed with averaging over 1000 iterations for each lattice size.

The percolation threshold p_c match the exact values in Table 1 very well. The other critical exponents for the honeycomb lattice seems to correspond well to the exact values, but γ and ν for the triangular lattice diverges somewhat from the exact values.

See Section 5 for plots of the linear fits performed for triangular and honeycomb lattices.

Table 2: Experimental critical exponents for bond percolation.

	Triangular	Honeycomb	Exact
p_c	0.3477	0.6525	-
β	0.1349	0.1493	$5/36 = 0.13\overline{8}$
γ	2.7816	2.2191	$43/18 = 2.3\overline{8}$
ν	1.6076	1.2702	$4/3 = 1.\overline{3}$

Task 5.6

See Fig. 11 for proof that the triangular lattice is the dual graph of the honeycomb lattice, and that the square lattice is the dual graph of itself.

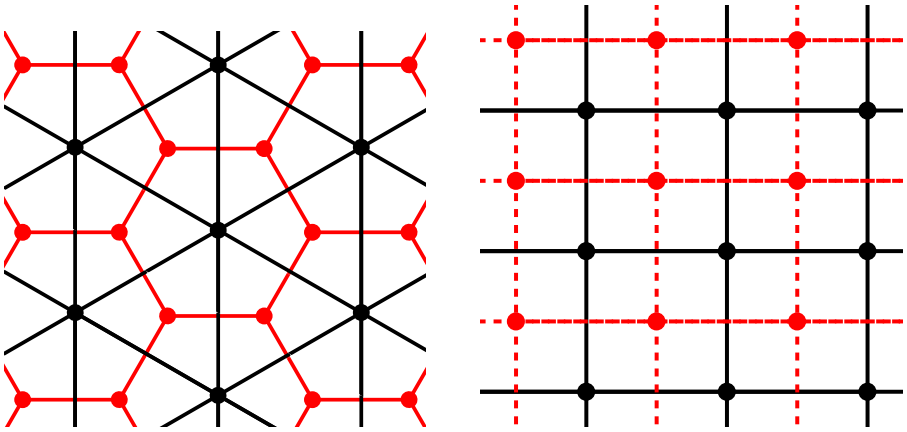
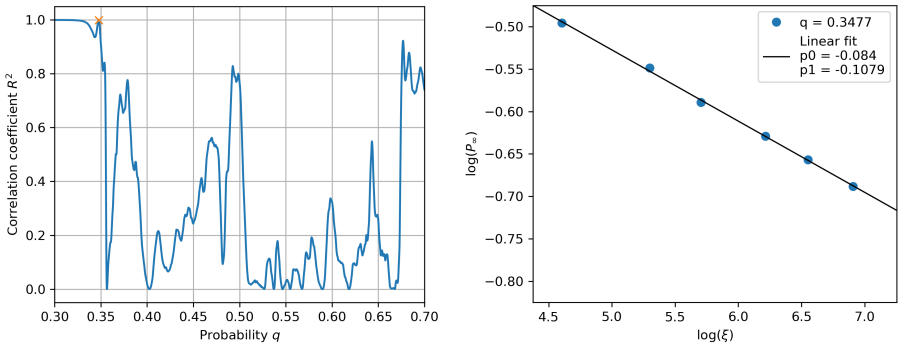
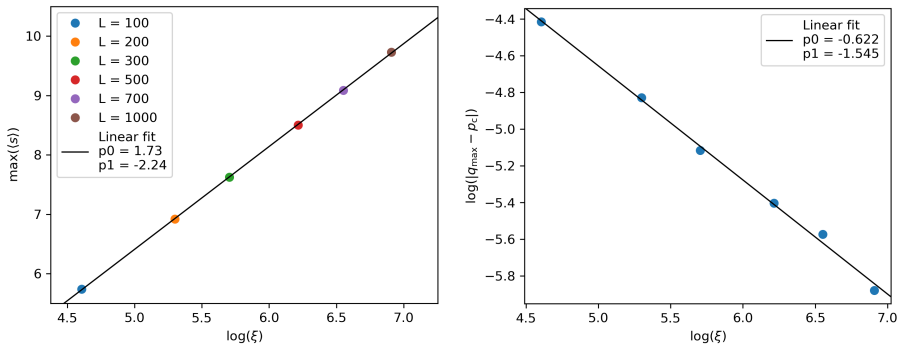


Figure 11: (left) The triangular lattice is the dual graph of the honeycomb lattice, and (right) the square lattice is the dual graph of itself.

Plots for honeycomb and triangular lattices



(a) **(b)**



(c) **(d)**

Figure 12: Results for triangular lattice.

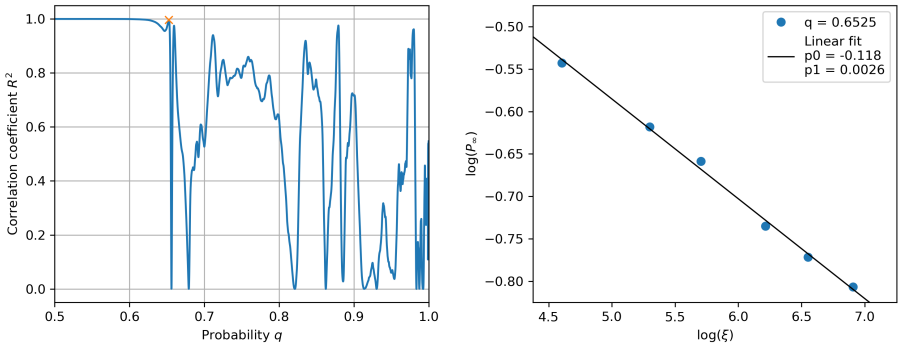


Figure 13: Results for honeycomb lattice.

The program

The main loop

```
    const uword maxNbonds = 2*sizes.max()*sizes.max();
    LogBinomCoeffGenerator gen(maxNbonds);

    for (uword L : sizes)
    {
        uword N = L*L;
        uword nBonds = 2*N;

        Results results(nBonds, N, gen);
        for (uword it = 0; it < its; it++)
        {
            umat bonds = makeRectangularLatticeBonds(L, L);
            shuffleBonds(bonds);

            Sites sites(L, L);
            for (uword i = 0; i < bonds.n_cols; i++)
            {
                sites.activate(bonds.col(i));
                results.sample(sites, it, i+1);
            }
        }

        // save results to file
        std::stringstream ss;
        ss << "L" << L << "_" << its << "iterations";
        string folder = "../cpp_results_square/" + ss.str();
        results.save(folder, 1e4);
    }

    return 0;
}

int runTriangular()
{
    const uvec sizes = {100, 200, 300, 500, 700, 1000};
//    const uvec sizes = {100, 200, 300};
```

Listing 15: The main program.

Results class

```cpp
#include <filesystem>
#include <armadillo>

#include "sites.hpp"
#include "logbinom.hpp"

using namespace std;
using namespace arma;

class Results
{
public:
    Results(const uword nBonds, const uword nSites,
            const LogBinomCoeffGenerator& generator):
        m_P(nBonds),
        m_Psquared(nBonds),
        m_s(nBonds),
        m_p(nBonds),
        m_logBinomCoeff(nBonds + 1),
        m_nSites(nSites),
        m_nBonds(nBonds),
        m_currentIteration(0),
        m_idx(0)
    {
        // tabulate log(n choose k) for all numbers of bonds
        for (uword i = 0; i <= nBonds; i++)
        {
            m_logBinomCoeff(i) = generator(nBonds, i);
        }
    }

    void sample(const Sites& sites, const uword iteration,
                const uword nActivatedBonds);
    void save(const string& folder, const uword nSamples=1e3);

private:
    vec m_P;
    vec m_Psquared;
    vec m_s;

    vec m_p; // probability

    running_stat_vec<vec> m_giant;
    running_stat_vec<vec> m_giantSquared;
    running_stat_vec<vec> m_averageSize;

    // table of log(n choose k) for n = nBonds and k in [0, nBonds]
    vec m_logBinomCoeff;

    uword m_nSites;
    uword m_nBonds;
    uword m_currentIteration;
    uword m_idx;

    vec calcResults(const vec& Q, const vec& pvec);
};
```

Listing 16: The `Results` class. The `sample` function is given in Listing 17 and the `calcResults` method is given in Listing 14.

```
void Results::sample(const Sites& sites, const uword iteration,
                     const uword nActivatedBonds)
{
    if (iteration > m_currentIteration)
    {
        // restart counter at beginning of each sample cycle
        m_idx = 0;
        m_currentIteration = iteration;

        // take sample
        m_giant(m_P);
        m_giantSquared(m_Psquared);
        m_averageSize(m_s);
    }

    const double p = nActivatedBonds/double(m_nBonds); // fraction of activated

    m_P(m_idx) = sites.giantComponent();
    m_Psquared(m_idx) = sites.giantComponentSquared();
    m_s(m_idx) = sites.averageSquaredSize();

    if (iteration == 0)
    {
        m_p(m_idx) = p;
    }

    m_idx++;
}
```

Listing 17: The `sample` method of the `Results` class.