Random numbers & high-dimensional integrals

It is very hard to implement a good random number generator because a sequence of trully random numbers can not be generated by deterministic computers. Only pseudo-random number generators can be coded. There are several excellent pseudo random number generators available in various libraries, which give very satisfactory results in combination with Monte Carlo methods or multidimensional integrations.

For every Monte Carlo application, it is crucial to use high quality random number generator. In practice, it is best to select a few random number generators with a good reputation, and make sure that results do not depend on the choice of a random number generator. A good source of excellent random numbers are

- GNU Scientific Library (GSL).
  - gsl_rng_mt19937 : a variant of the “Mersenne Twister” generator.

  - gsl_rng_ranlux389 : second generation ranlux algorithm developed by Lüscher.
  - gsl_rng_taus2 : Tausworthe generator by L'Ecuyer.

- mkl library :
  - VSL_BRNG_MT19937 : A Mersenne Twister pseudorandom number generator.
  - VSL_BRNG_MT2203 : A set of 6024 Mersenne Twister pseudorandom number generators.
  - VSL_BRNG_SFMT19937 : A SIMD-oriented Fast Mersenne Twister pseudorandom number generator.

How do random number generators work?

The simplest and fastest random number generators, which are unfortunately not of very high quality, are congruential algorithms

$$I_{j+1} = aI_j + c \bmod m. \tag{1}$$

For more complicated and much better methods, see Numerical Recipes book. If a fast generator of reasonable quality is desired, one can use standard methods implemented in c, such as `drand48` and `srand48` defined in `cstdlib`.

There alo exist very sophisticated tests for ”good” randomness or random number generators, and very few random numbers generators pass sophisticated tests. A nice collection of these tests is available at `http://www.phy.duke.edu/~rgb/General/dieharder.php`. A two simplest examples of tests are:

- random walk
- distribution of points on a square

Random walk: It is clear that the distance a particle can travel by performing N random

steps is proportional to $\sqrt{N}$. Most of good random number generators "obey" that constrain. To test that property, we can release a random walker from origin, and measure the distance it reaches from the origin after i steps, where i is any number between 1 and large N. It is clear than a single random walker will not end up $\sqrt{N}$ from the origin. To avoid fluctuations we need to repeat random walk many times or release a lot of random walkers and make an average over these random walkers. This should give us a perfect square-root curve.

Distribution of points on a square: This test is very easy to implement. Let's take two succesive random numbers as a point in 2D space (x, y). We can plot a large number of such points on a screen, and if one sees any patter in the plot, the generator is very bad.

Multidimensional integration

Multidimensional numeric integration in more than 4 dimensions is more appropriate for Monte-Carlo than one-dimensional quadratures. If the function is smooth enough, or we know how to transform integral to make it smooth, the integration can be performed with MC. The reason for MC success is that it's error, according to central limit theorem, is always proportional to $1/\sqrt{N}$ independent of dimension.

The error of one dimensional quadratures can be estimated: If the number of points used in each dimension is N_1, the number of all points used in d dimensions is $N = (N_1)^d$. The error for trapezoid rule was estimated to $1/(N_1)^2$ therefore the error in d dimensions is $1/N^{2/d}$. It is therefore clear than for $d = 4$ the Monte-Carlo error and the trapezoid-rule error are equal.

For more or less flat functions, the integration is straightforward

$$\int f dV \approx V\langle f\rangle \pm V\sqrt{\frac{\langle f^2\rangle - \langle f\rangle^2}{N}} \tag{2}$$

here

$$\langle f\rangle \equiv \frac{1}{N}\sum_{i=0}^{N-1} f(x_i) \tag{3}$$

$$\langle f^2\rangle \equiv \frac{1}{N}\sum_{i=0}^{N-1} f^2(x_i) \tag{4}$$

If the function f is rapidly varying, the variance is going to be large and precision of the integral vary bad. The scaling $1/\sqrt{N}$ is very bad! If one has a lot of computer time and patience, one might still try to use this method, because it is so straighforward to implement.

If the region V of the integral is complicated and is hard to generate distribution of points in volume V, one can just find a larger and simpler volume W which contains volume V. Then one samples over W and defines the function f to be zero outside V. Of course the error will increase because number of ”good” points is smaller.

Importance sampling

Usefulness of Monte Carlo becomes more appealing when importance sampling strategy is implemented. Of course one needs to know something about the function to implement the strategy, but one is rewarded with much higher accuracy.

It is simplest to ilustrate the idea in 1D. If one knows that function $f(x)$ to be integrated is mostly proportional to another function $w(x)$ in the region where the integral contains most of the weight, we might want to rewrite the integral

$$\int f(x)dx = \int \frac{f(x)}{w(x)} w(x)dx \tag{5}$$

If weight function $w(x)$ is a simple analytic function, which can be integrated analytically, and obeys the following constrains

- $w(x) > 0$ for every x
- $\int w(x)dx = 1$

we can define

$$W(x) = \int^x w(t)dt \qquad \rightarrow \qquad dW(x) = w(x)dx \tag{6}$$

and rewrite

$$\int f(x)dx = \int \frac{f(x)}{w(x)} dW(x) = \int \frac{f(x(W))}{w(x(W))} dW \rightarrow \left\langle \frac{f(x(W))}{w(x(W))} \right\rangle_{W\ uniform\ \in [0,1]}$$

If the function f/w on mesh W is reasonably flat, it can be efficiently integrated by MC. The error is now proportional to $\sqrt{\frac{\langle (f/w)^2 \rangle - \langle f/w \rangle^2}{N}}$ and is therefore greatly reduced.

To implement the algorith, we generate uniform random numbers r in the interval $r \in [0, 1]$ which correspond to variable W. We can solve the equaton for $x = W^{-1}(r)$ to get x and use it to evaluate $f(x)/w(x)$. The random numbers are therefore uniformly distributed on mesh W while they are non-uniformly distributed on mesh x.

This can also be writtens as

$$\left\langle \frac{f(x)}{w(x)} \right\rangle_{\frac{P(x)}{dx}=w(x)}$$

because the distribution of points x is $\frac{dP}{dx} = \frac{dP}{dW}\frac{dW}{dx}$ and since distribution $\frac{dP}{dW}$ is uniform, and $\frac{dW}{dx} = w(x)$, we have $\frac{dP}{dx} = w(x)$.

The archaic example is the exponential weight function

$$w(x) = \frac{1}{\lambda} e^{-x/\lambda} \qquad for\ x > 0 \tag{7}$$

This is equivalent to our exponentially distributed mesh points. Most of them are going to be close to 0 and only few at large x.

The integral is $W(x) = 1 - e^{-x/\lambda}$ which gives for the inverse $x = -\lambda \ln(1 - W)$. The integral

$$\int_0^\infty f(x)dx = \int_0^1 \frac{f(-\lambda \ln(1-W))}{w(-\lambda \ln(1-W))} dW = \int_0^1 f(-\lambda \ln(1-W)) \frac{\lambda dW}{1-W} \tag{8}$$

is easily evaluated with MC if $f(x)$ is exponentially falling function.

$$\int_0^\infty f(x)dx \to \langle \lambda \frac{f(-\lambda \ln(1-W))}{1-W} \rangle_{W\ uniform\in[0,1]} \tag{9}$$

Since $\frac{dP}{dW} = 1$ (W is uniformly distributed), the probability for x is $\frac{dP}{dx} = w(x)$, therefore x is exponentially distributed random number.

We could also write

$$\int_0^\infty f(x)dx \to \left\langle \frac{f(x)}{\frac{e^{-x/\lambda}}{\lambda}} \right\rangle_{\frac{dP}{dx}=e^{-x/\lambda}/\lambda} \tag{10}$$

Another very usefull weight function is Gaussian distribution

$$w(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(x-x_0)^2}{2\sigma^2}} \tag{11}$$

How do we get random number x to be distributed according to the above distribution? The integral gives erf function and its inverse is not simple to evaluate.

The trick is to use two random numbers to get Gaussian distrbution. Consider the following

algorithm

$$x_1 = x_0 + \sqrt{-2\sigma^2 \ln r_1}\cos(2\pi r_2) \tag{12}$$

$$x_2 = x_0 + \sqrt{-2\sigma^2 \ln r_1}\sin(2\pi r_2) \tag{13}$$

The distribution of r_1 and r_2 is uniform in the interval $[0, 1]$. The distribution of x_1 and x_2 is

$$\frac{d^2P}{dx_1 dx_2} = \frac{d^2P}{dr_1 dr_2}\left|\frac{\partial(r_1, r_2)}{\partial(x_1, x_2)}\right| = \frac{1}{\sqrt{2\pi}}e^{-(x_1-x_0)^2/2}\frac{1}{\sqrt{2\pi}}e^{-(x_2-x_0)^2/2} \tag{14}$$

therefore Gaussian. In this way, we can obtaine x to be distributed gaussian, and we can evaluate

$$\int f(x)dx = \left\langle \frac{f(x)}{\frac{1}{\sqrt{2\pi\sigma^2}}e^{-(x-x_0)^2/(2\sigma^2)}} \right\rangle_{\frac{dP}{dx}=\frac{1}{\sqrt{2\pi\sigma^2}}e^{\frac{(x-x_0)^2}{2\sigma^2}}} \tag{15}$$

What is the best choice for weight function w?

If function f is positive, clearly best w is just proportional to f. What if f is not positive everywhere? It turns out that the best choice is absolute value of f, i.e.,

$$w = \frac{|f|}{\int |f| dV} \tag{16}$$

The proof is simple. The MC importance sampling evaluates

$$\int f dV = \int \frac{f}{w} w dV \approx \left\langle \frac{f}{w} \right\rangle \pm \sqrt{\frac{\langle (\frac{f}{w})^2 \rangle - \langle \frac{f}{w} \rangle^2}{N}} \tag{17}$$

and the error is minimal when

$$\delta \left(\langle (\frac{f}{w})^2 \rangle - \langle \frac{f}{w} \rangle^2 + \lambda (\int w dV - 1) \right) = 0 \tag{18}$$

$$\delta \left(\int \frac{f^2}{w^2} w dV - \left(\int \frac{f}{w} w dV \right)^2 + \lambda (\int w dV - 1) \right) = 0 \tag{19}$$

$$\int \left(\frac{f^2}{w^2} - \lambda \right) dV = 0 \qquad \rightarrow \qquad w \propto |f| \tag{20}$$

If we know a good approximation for function f, we can use this information to sample the same function f to higher accuracy with importance sampling. The solution can thus be improved iteratively. This idea is implemented in Vegas algorithm (see below).

There is another set of algorithms to improve precision of Monte Carlo sampling. The idea is to divide the volume into smaller subregions and check in each subregion how rapidly is the function f varying in each subregion. The quantitative estimation can be the variance of the function is each subregion $\sqrt{\langle f^2 \rangle - \langle f \rangle^2}$. The idea is to increase the number of points in those regions where variance is big. The algorithm is called Stratified Sampling and is used in Miser integration routine. The idea seems simple and powerful, but is not very usefull for high-dimensional integration because the number of subregions grows exponentially with the number of dimensions therefore it is usefull only if we have some idea how to constract small number of subregions where variance of f is large. This last trick is also used in Vegas algorithm which is probabbly the best algorithm available at the moment.

Vegas

Peter Lepage, JOURNAL OF COMPUTATIONAL PHYSICS 27, 192-203 (1978).

The Vegas method is primary based on the importance sampling algorithm with the above mentioned self-adapting strategy. The basic idea is to use a separable weigh function. Thus instead of complicated $w(x, y, z, \cdots)$ one uses an ansatz $w = w_1(x)w_2(y)w_3(z)\cdots$.

The optimal separable weigh functions are

$$w_1(x) \propto \left[\int dy \int dz \cdots \frac{|f(x, y, z, \cdots)|^2}{w_1(x)w_2(y)w_3(z)\cdots}\right] \tag{21}$$

in close resembles with the 1-dimensional case above.

The power of Vegas is that by iteration, it can resolve any divergent point which is separable, i.e., it is parallel to any axis. However, when the divergency is along diagonal Vegas is similar to usual MC sampling. Note that separable ansatz avoids the explosion of stratified regions, which scale as K^d, while separable ansatz scales as $K \times d$. (K is number of points in one dimension, i.e., typically $K \approx 10^2$, $d \approx 10 \rightarrow$ ZettaByte versus KiloByte).

The algorithm starts with the separable weight function, which we will call a grid, $g_i(x)$. We want to evaluate

$$\int_0^1 dg_1 \int_0^1 dg_2 \cdots f(g_1, g_2, \cdots) =$$
$$\int_0^1 dx \int_0^1 dy \cdots f(g_1(x), g_2(y), g_3(z), \cdots) \frac{dg_1}{dx} \frac{dg_2}{dy} \cdots dxdydz \cdots \quad (22)$$

Note that g_1 depends only on x, and g_2 only on y, so that the Jacobian of such separable set of functions is just the product of all derivatives.

We first start with the grid functions $g_i(x) = x$, so that the integration at the first iteration is equivalent to the usual Monte Carlo sampling.

We generate a few thousand sets of random points (x, y, z) and evaluate f on these points. During the sampling we evaluate the integral

$$\langle f \rangle = \sum_{x,y,z} f(g_1(x), g_2(y), g_3(z)) \frac{dg_1}{dx} \frac{dg_2}{dy} \frac{dg_3}{dz}, \quad (23)$$

Note that random points are uniformly distributed on mesh x ($x \in [0, 1)$), on mesh y, etc, therefore the unit volume $\int dxdy \cdots = 1$.

We also save the value of the function on a grid, namely,

$$f_1(x) = \sum_{y,z} |f(g_1(x), g_2(y), g_3(z)) \frac{dg_1}{dx} \frac{dg_2}{dy} \frac{dg_3}{dz}|^2 \tag{24}$$

$$f_2(y) = \sum_{x,z} |f(g_1(x), g_2(y), g_3(z)) \frac{dg_1}{dx} \frac{dg_2}{dy} \frac{dg_3}{dz}|^2 \tag{25}$$

$$f_3(z) = \sum_{x,y} |f(g_1(x), g_2(y), g_3(z)) \frac{dg_1}{dx} \frac{dg_2}{dy} \frac{dg_3}{dz}|^2 \tag{26}$$

and then we normalize these projected functions

$$\widetilde{f}_i(x) = \frac{f_i(x)}{\int_0^1 f_i(x)dx}, \tag{27}$$

so that they map the interval $[0, 1] \rightarrow [0, 1]$.

Next we need to construct the refined grid functions $g_i(x)$ using the new sampled information ($\widetilde{f}_i(x)$). Just like in 1D case above

$$\int \widetilde{f}(g)dg = \int \widetilde{f}(g(x)) \frac{dg}{dx} dx, \tag{28}$$

we would like to have

$$\widetilde{f}(g(x))\frac{dg}{dx} = const \tag{29}$$

so that each interval of the grid contributes the same amount to the integral.

To determine the new grid g_l, we will determine the grip points numerically so that

$$\int_0^1 \widetilde{f}(g_{old})dg_{old} = I \tag{30}$$

$$\int_{g_{l-1}^{new}}^{g_l^{new}} \widetilde{f}(g_{old})dg_{old} = \frac{I}{N_g} \tag{31}$$

where N_g is the number of gridpoints. Hence, we require that there is exactly the same weight between each two consecutive points (between g_0 and g_1, and between g_1 and g_2,....).

Once we have a new grid, we can restart the sampling of the integral of f, just like in Eqs. 23 to 26, using g_{new}. We generate again a few thousand random set of points (x, y, z) and obtain $\widetilde{f}_i(x)$ functions. We then repeat this procedure approximately 10-times, and we can slightly increase the number of random points each time, as the grid functions becomes more are more precise. At the end, we can run a single long run with

10-times longer sampling, to reduce sampling error-bar.

In practice, we will implement a few more tricks, which make the algorithm stable.

First, when we compute the separable function $\widetilde{f}_i(x)$, we will smoothen it, because we want to avoid accessive discontinuities due to finite number of random points. We will just average over nearest neighbors

$$\widetilde{f}_i \leftarrow \frac{\widetilde{f}_{i-1} + \widetilde{f}_i + \widetilde{f}_{i+1}}{3} \tag{32}$$

Note, be careful at the endpoints, where you need to average over two points only.

Second, we will not use $\widetilde{f}(x)$ in Eq. 31 directly, but we will first transform it slightly. Namely, if $\widetilde{f}(x)$ is very small (very large) in some regions, we will keep transformed $\widetilde{f}(x)$ less small (less large). A good transformation is provided by the following function

$$t(r) = \left(\frac{r-1}{\log(r)}\right)^{3/2} \tag{33}$$

Note that $t(1) = 1$ and at small r it increases very fast $t(r) \approx (1/\log(1/r))^{3/2}$, while at large r it approaches $t(r) \approx \frac{3}{4}r$, which is less that r.

We will then use $t(\widetilde{f})$ instead of $\widetilde{f}$ when we build the new grid $g_{new}(x)$.

Finally, for the grid $g(x)$ we will have the grid of points distributed between $[0,1]$ so that $x_0 = \frac{1}{N}, x_1 = \frac{2}{N}, \cdots, x_{N-2} = \frac{N-1}{N}, x_{N-1} = 1$. We know that $g(x=0) = 0$, hence we do not need to save pont $x = 0$, but we need to be careful when interpolating at the point x_0. For such equidistant mesh, it is clear that given a point $x \in [0,1]$, we can compute $i = int(xN)$, and then we know that x appears between x_{i-1} and x_i, and linear interpolation gives

$$g(x) = g_{i-1} + (g_i - g_{i-1})\frac{(x - i/N)}{1/N} \tag{34}$$

$$g(x) = g_0 \; xN \quad if \quad i = 0 \tag{35}$$

Finally, we want to discuss the calculation of the error and confidence in the result. We will perform M outside iterations (which update the grid). Each such iteration will consist of n_i function evaluations (of the order of few thousand to ten thousand). From all these calculations $M * n$ we want to evaluate the best estimate of the integral, and its estimation for the error.

At each iteration, we will sample the following qualities:

$$\langle f_w \rangle_i = \frac{1}{n_i} \sum_{j=1}^{n_i} f(g_1(x_j), g_2(y_j), g_3(z_j)) \frac{dg_1}{dx}(x_j) \frac{dg_2}{dy}(y_j) \frac{dg_3}{dz}(z_j) \quad (36)$$

$$\langle f_w^2 \rangle_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \left(f(g_1(x_j), g_2(y_j), g_3(z_j)) \frac{dg_1}{dx} \frac{dg_2}{dy} \frac{dg_3}{dz} \right)^2 \quad (37)$$

Then the estimation for the variance-square of the MC-sampling is

$$\sigma_i^2 = \frac{\langle f_w^2 \rangle_i - \langle f_w \rangle_i^2}{n_i - 1} \quad (38)$$

Note that the variance of the sampled function is $\sqrt{\langle f_w^2 \rangle_i - \langle f_w \rangle_i^2}$, which is approaching a constant when the number of sampled points n_i goes to infinity. However, the variance of the MC-sampling is $\sigma_i \propto \frac{1}{\sqrt{n_i}}$, as expected.

From all accumulated evaluations of the function (in M iterations), we can construct the best estimate of the integral. Naively, we would just calculate $1/M \sum_i \langle f_w \rangle_i$. However, at the first few iterations the error was way bigger than in the last iteration, and therefore we want

to penalize those early estimates, which were not so good. We achieve that by

$$I_{best} = \frac{\sum_{i=1}^{M} \frac{\langle f_w \rangle_i}{\sigma_i^2}}{\sum_{i=1}^{M} \frac{1}{\sigma_i^2}} \tag{39}$$

Similarly, the error does not sum up, but is rather smaller than the smallest error (in the last iteration). We have

$$\frac{1}{\sigma^2}_{best} = \sum_{i=1}^{M} \frac{1}{\sigma_i^2} \tag{40}$$

and finally the χ^2 can be estimated by

$$\chi^2 = \frac{1}{M-1} \sum_{i=1}^{M} \frac{(\langle f_w \rangle_i - I_{best})^2}{\sigma_i^2} \tag{41}$$

The jupyter notebook implementation of Veags is available on the website

`http://www.physics.rutgers.edu/~haule/509/MC.html`.

The algorithm Vegas is also efficiently implemented in GNU SCIENTIFIC LIBRARY (`http://www.gnu.org/software/gsl/`).

For all major distributions of linux and also Windows simulation of linux (cigwin) the GNU SCIENTIFIC LIBRARY comes precompiled (rpm package). For mac, `brew install gsl` should work.

Unfortunately, the code is written in C and not C++. However, it is coded in modular way so that it is simple to write wrapper classes which hide the details of calls and data structures used. For details of the gsl library, see

`http://www.gnu.org/software/gsl/manual/html_node/Monte-Carlo-Integration.html`.

The wrapper is implemented in

`http://www.physics.rutgers.edu/~haule/509/src_numerics/Random/Vegas/`.

We will test MC by evaluating the three-dimensional divergent function

$$f(k_x, k_y, k_z) = \frac{1}{\pi^3} \frac{1}{1 - \cos k_x \cos k_y \cos k_z} \tag{42}$$

It is clear that this function should be simple for Vegas since divergencies are only in corners and not along diagonals.

Untitled−1 1

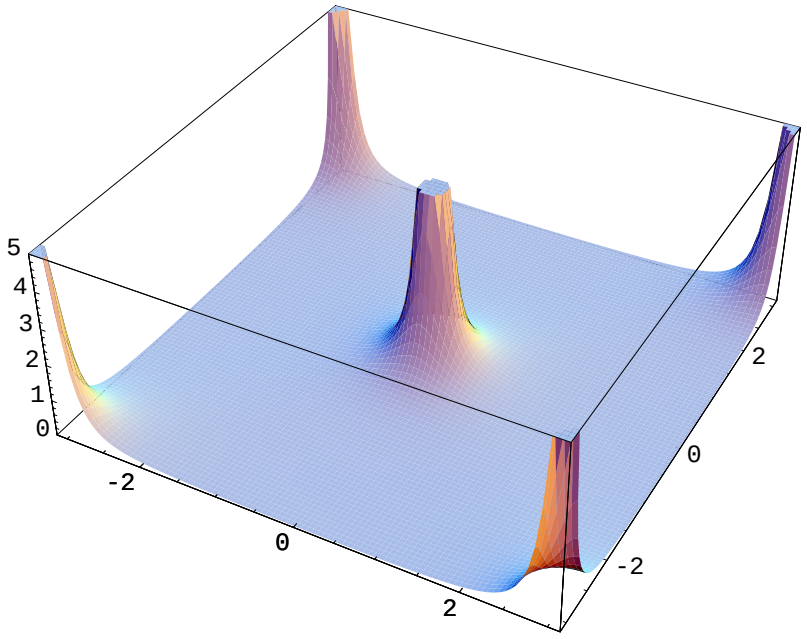

Figure 1: Plot of the two dimensional analog of the function $f(k_x, k_y, k_z)$.

Get familiar with gnu scientific library and check the implementation of the wrapper classes.

- Using the simple Monte-Carlo sampling and half a milion function evaluations, one gets error of the order of 2%.
- The Miser algorithm which uses Stratified sampling improves the accuracy for almost

one order of magnitude using the same number of function evaluations.

- Finally, Vegas further improves the accuracy for additional order od magnitude using even fewer function evaluations (100000 for warmup and 100000 for second call).

```
plain ==================
result = 1.41221
sigma  = 0.0134359
exact  = 1.3932
error  = 0.0190048 = 1.41448 sigma

miser ==================
result = 1.39132
sigma  = 0.00346056
exact  = 1.3932
error  = -0.00188235 = 0.543944 sigma
```

```
vegas warm-up ==================
result = 1.39267
sigma  = 0.00341041
exact  = 1.3932
error  = -0.000531339 = 0.155799 sigma

converging...
result = 1.39328 sigma=0.00036248 chisq/dof=
vegas final ==================
result = 1.39328
sigma  = 0.00036248
exact  = 1.3932
error  = 7.74556e-05 = 0.213682 sigma
```

Diffusion-limited aggregation

Random number generators can also be used in direct simulation: some processes of which we do not know the details and we model them by random generator.

An example of direct simulation is Diffusion-limited aggregation (DLA). It is a way to form objects with a special beauty. It takes place in non-living (mineral deposition, snowflake growth, lightning paths) or living (corals) nature - or within computers.

- Diffusion can be modeled by random motion (aka Brownian motion)
- When particles have the possibility to attract each other and stick together, they may form aggregates.

DLA is one of the most important models of fractal growth. It was invented by two physicists, T.A. Witten and L.M. Sander , in 1981. The growth rule is remarkably simple. We start with an immobile seed on the plane. A walker is then launched from a random position far away and is allowed to diffuse. If it touches the seed, it is immobilized instantly and becomes part of the aggregate. We then launch similar walkers one-by-one and each of them stops upon hitting the cluster. After launching a few hundred particles, a cluster with intricate branch structures results.

We can assign dimensionality to DLA cluster just like to a fractal. The definition is

$$m(r) \propto r^d \tag{43}$$

The fractal dimensionality of cluster in two dimensions is always less than 2. The objects which mass is increasing slower than r^2 must contain cracks or holes and the size of cracks and holes must increase with r.

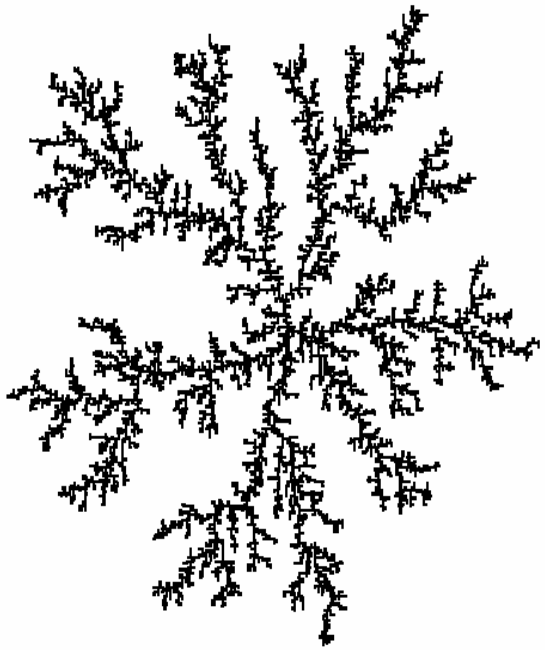

Figure 2: Typical DLA cluster

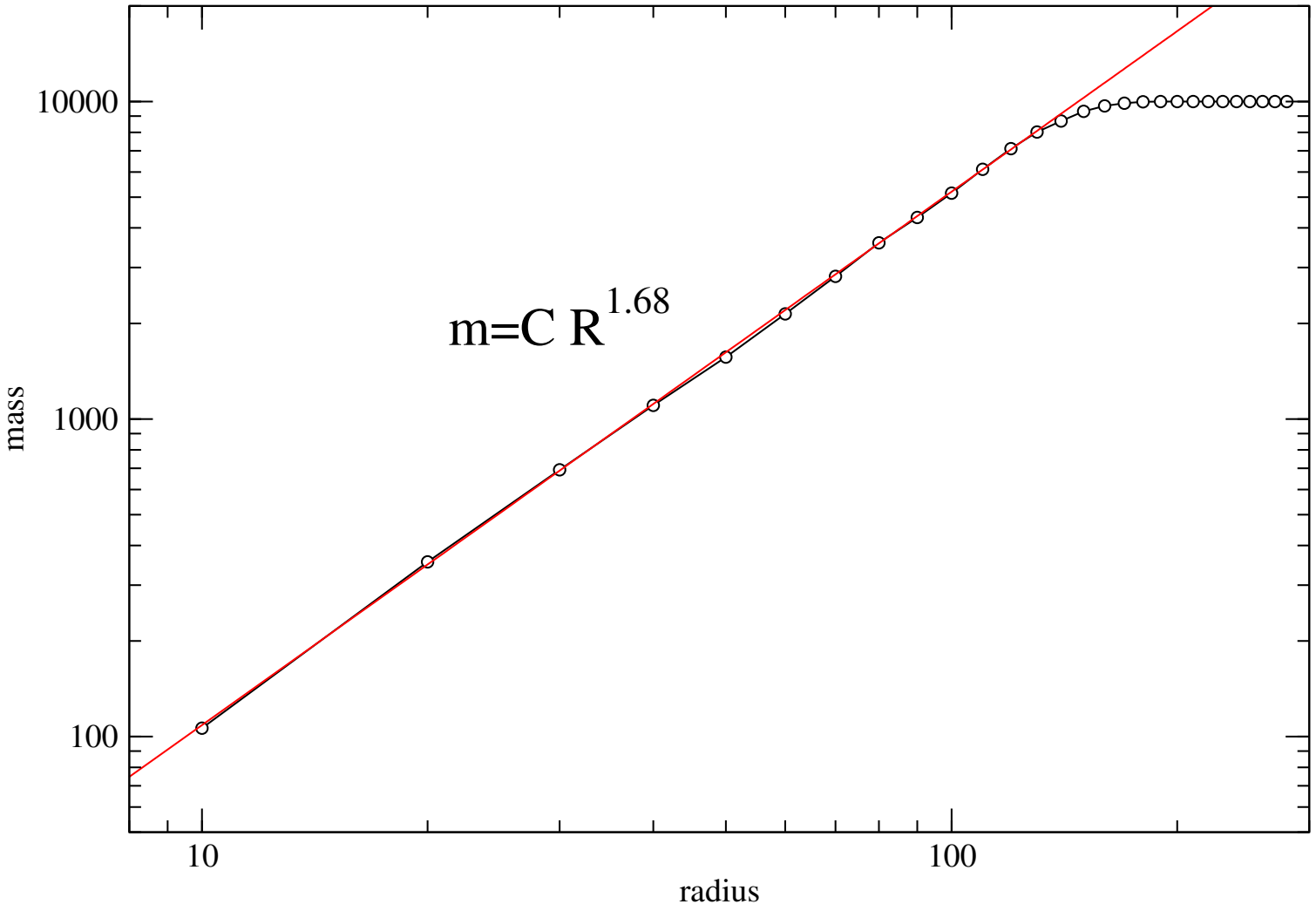

Figure 3: Mass of the DLA cluster as a function of radius R. The dimensionality of the cluster is appoximately 1.68.

```
class Lattice{
  int N;                                  // size of the lattice is NxN
  std::vector<std::vector<bool> > site; // matrix of positions, if it is false, it is empty
  std::pair<int,int> tpos, ppos;          // temporary and previous position of the particle
  std::pair<double,double> r0;            // average position <vec{r}>
  double r2;                              // average <r^2>
  static const int MAXRGB = 65535;
  int Nclust;                            // number of points in the cluster
public:
  Lattice(int N_);
  void ReleaseNew();
  int MakeStep();
};
 Lattice::Lattice(int N_): N(N_), site(N), r0(0,0), r2(0), Nclust(0)
{
  for (int i=0; i<site.size(); i++){
    site[i].resize(N);
    for (int j=0; j<site[i].size(); j++){
      site[i][j]=false;                // sites are empty at the beginning
    }
  }
  site[N/2][N/2]=true;// We put first point in the middle of the system
};

void Lattice::ReleaseNew()
{ // Random walker is released from the boundary
  do{
    int istart = static_cast<int>(drand48()*4*N); // boundary size is 4*N
    switch(istart/N){ // which face of a square the particle is put to?
    case 0: tpos = std::make_pair(istart,0); break;
    case 1: tpos = std::make_pair(N-1,istart%N); break;
    case 2: tpos = std::make_pair(istart%N,N-1); break;
    case 3: tpos = std::make_pair(0,istart%N); break;
    }
    // we have random site on the bundary. But is it empty?
  } while (site[tpos.first][tpos.second]);// just make sure that the site is empty
```

```
}

int Lattice::MakeStep()
{// Return codes: 1 - particle glued
  //              0 - particle moved
  std::pair<int,int> newpos(tpos);
  int ipos = static_cast<int>(drand48()*4); // particle can go in 4 directions: up,down,left,right
  switch(ipos){// Here we use peridoci boundary conditions in order that particle can not escape too fast
  case 0: newpos.first  = (tpos.first+1)%N;   break; // right
  case 1: newpos.second = (tpos.second+1)%N;  break; // up
  case 2: newpos.first  = tpos.first>0  ? (tpos.first-1) : N-1;   break; // left
  case 3: newpos.second = tpos.second>0 ? (tpos.second-1) : N-1;  break; // down
  }
  // Checking weather particle can be glued!
  // Here we do not take periodic boundary conditions since they put too many particles on the boundary
  bool r_neighbor = newpos.first<N-1   && site[newpos.first+1][newpos.second];
  bool u_neighbor = newpos.second<N-1  && site[newpos.first][newpos.second+1];
  bool l_neighbor = newpos.first>0     && site[newpos.first-1][newpos.second];
  bool d_neighbor = newpos.second>0    && site[newpos.first][newpos.second-1];

  if (r_neighbor || u_neighbor || l_neighbor || d_neighbor){ // If particle can be glued
    // Particle just glued
    site[newpos.first][newpos.second]=true; // lattice is now occupied
    r0.first += newpos.first-N/2;
    r0.second += newpos.second-N/2;
    r2 += sqr(newpos.first-N/2)+sqr(newpos.second-N/2); // average r^2
    Nclust++;                                  // number of particles in DLA cluster increases
    return 1;
  }
  ppos = tpos;
  tpos = newpos; // update position of the walker
  return 0;
}
```