

Random numbers & high-dimensional integrals

It is very hard to implement a good random number generator because a sequence of truly random numbers can not be generated by deterministic computers. Only pseudo-random number generators can be coded. There exist several excellent pseudo random number generators 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 good random numbers is GNU Scientific Library (GSL).

How do random number generators work?

The simplest and fastest random number generators, which are however not of very high

quality, are congruential algorithms

$$I_{j+1} = aI_j + c \text{ mod } m. \quad (1)$$

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 also 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 are 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 that 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 successive 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 pattern 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 its 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 $1/N^{2/d}$ and for Simpson's rule $1/N^{4/d}$. It is therefore clear than for $d = 4$ Monte-Carlo error and trapezoid-rule error are the same.

Here we should emphasize that the MC integration is not very sensitive to the quality of random number generator. The Markov chain simulation, however, is very sensitive.

For more or less flat functions, the integration is straightforward

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

here

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

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

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 you have a lot of computer time and patience, you might still try to use it because it is so straightforward to implement.

If the region V of your integral is complicated and is hard to find uniform deviates in the volume V , just find larger and simple volume W which contains volume V . Then sample over W and define 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 you need to know something about your function to implement it, but you will be rewarded with much higher accuracy.

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

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

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

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

we are almost done.

$$W(x) = \int^x w(t)dt \quad \rightarrow \quad dW(x) = w(x)dx \quad (6)$$

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

Function f/w on mesh W is reasonably flat therefore it can be successfully 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.

We generate uniform random numbers r in the interval $r \in [0, 1]$ which correspond to variable W . We can solve the equation 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 for W while they are non-uniformly distributed on x mesh.

The archaic example is the **exponential** weight function

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

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

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

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

$$\int_0^\infty f(x) dx \rightarrow \left\langle \lambda \frac{f(-\lambda \ln(1 - W))}{1 - W} \right\rangle_{W \text{ uniform} \in [0,1]} \quad (9)$$

It is easy to check that if random number $r \in [0, 1]$ is uniform, the resulting $x = -\lambda \ln(1 - r)$ is exponentially distributed random number.

Another very useful weight function is **Gaussian distribution**

$$w(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-x_0)^2}{2\sigma^2}} \quad (10)$$

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 distribution**. Consider the following algorithm

$$x_1 = \sqrt{-2 \ln r_1} \cos(2\pi r_2) \quad (11)$$

$$x_2 = \sqrt{-2 \ln r_1} \sin(2\pi r_2) \quad (12)$$

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

$$\frac{d^2 P}{dx_1 dx_2} = \frac{d^2 P}{dr_1 dr_2} \left| \frac{\partial(r_1, r_2)}{\partial(x_1, x_2)} \right| = \frac{1}{\sqrt{2\pi}} e^{-x_1^2/2} \frac{1}{\sqrt{2\pi}} e^{-x_2^2/2} \quad (13)$$

therefore Gaussian. In this way, we obtained the desired

$$\int f(x) dx = \left\langle \frac{f(x)}{\frac{1}{\sqrt{2\pi}} e^{-x^2/2}} \right\rangle_{x \text{ distributed Gaussian}} \quad (14)$$

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.,

$$w = \frac{|f|}{\int |f| dV} \quad (15)$$

The proof is simple. The MC importance sampling evaluates

$$\int f dV = \int \frac{f}{w} w dV \approx \left\langle \frac{f}{w} \right\rangle \pm \sqrt{\frac{\langle (\frac{f}{w})^2 \rangle - \langle \frac{f}{w} \rangle^2}{N}} \quad (16)$$

and the error is minimal when

$$\delta \left(\langle (\frac{f}{w})^2 \rangle - \langle \frac{f}{w} \rangle^2 + \lambda \left(\int w dV - 1 \right) \right) = 0 \quad (17)$$

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

$$\int \left(\frac{f^2}{w^2} - \lambda \right) dV = 0 \quad \rightarrow \quad w \propto |f| \quad (19)$$

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 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 in each subregion $\sqrt{\langle f^2 \rangle - \langle f \rangle^2}$. The idea is to increase number of point 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 useful for high-dimensional integration because the number of subregions grows exponentially with the number of dimensions therefore it is useful only if we have some idea how to construct small number of subregions where variance of f is large. This last trick is also used in Vegas algorithm which is probably the best algorithm available at the moment.

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 $w(x_1, x_2, \dots)$ one uses an ansatz $w = w_1(x_1)w_2(x_2) \dots$.

Just like we showed above that best weight function for any function f is its absolute value $|f|$, it can be shown that the optimal separable weigh function is

$$w_1 \propto \left[\int dx_2 \int dx_3 \cdots \frac{f^2(x_1, x_2, \cdots)}{w_2(x_2)w_3(x_3) \cdots} \right]^{1/2} \quad (20)$$

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 **not better than usual MC sampling**.

The algorithm **Vegas** is somewhat tedious to implement therefore we will use **GNU SCIENTIFIC LIBRARY** (<http://www.gnu.org/software/gsl/>) instead.

For all major distributions of linux and also Windows simulation of linux (cigwin) the **GNU SCIENTIFIC LIBRARY** comes precompiled (rpm package). If you can not find binary, you might want the compile it yourself.

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.

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

$$f(k_x, k_y, k_z) = \frac{1}{\pi^3} \frac{1}{1 - \cos k_x \cos k_y \cos k_z} \quad (21)$$

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

Untitled-1

1

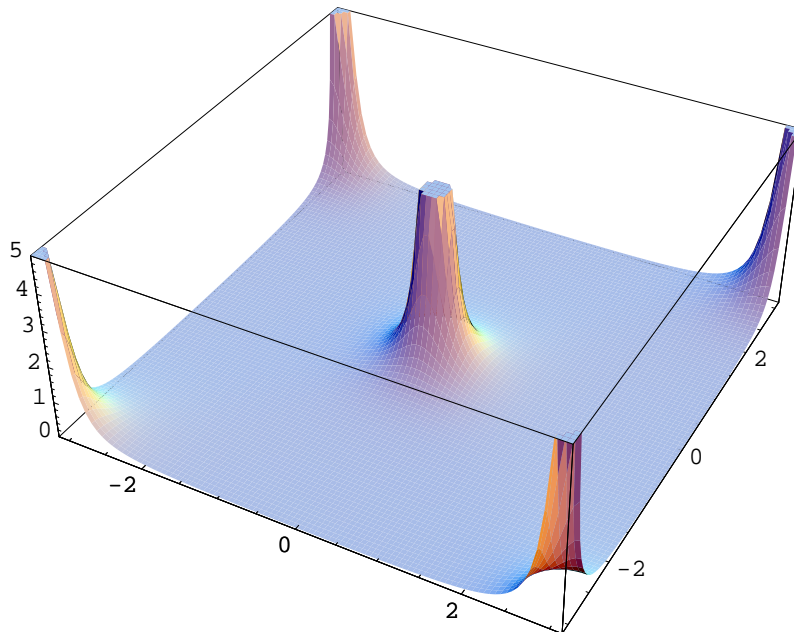


Figure 1: Plot of the two dimensional analog of the function $f(k_x, k_y, k_z)$.

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

$$m(r) \propto r^d \quad (22)$$

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 .

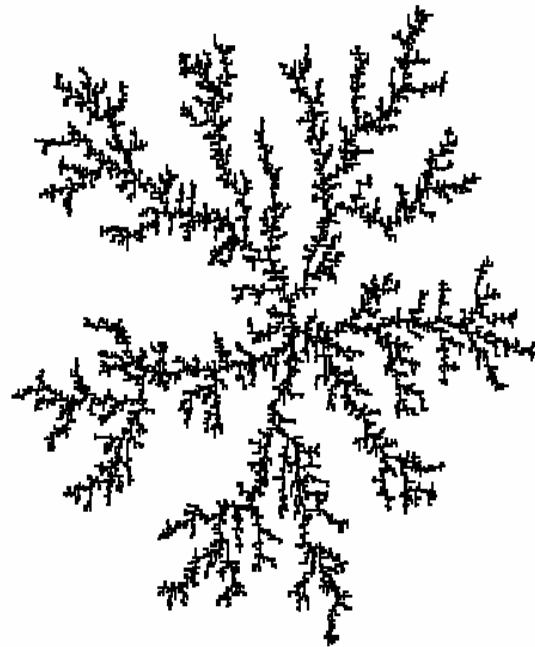


Figure 2: Typical DLA cluster

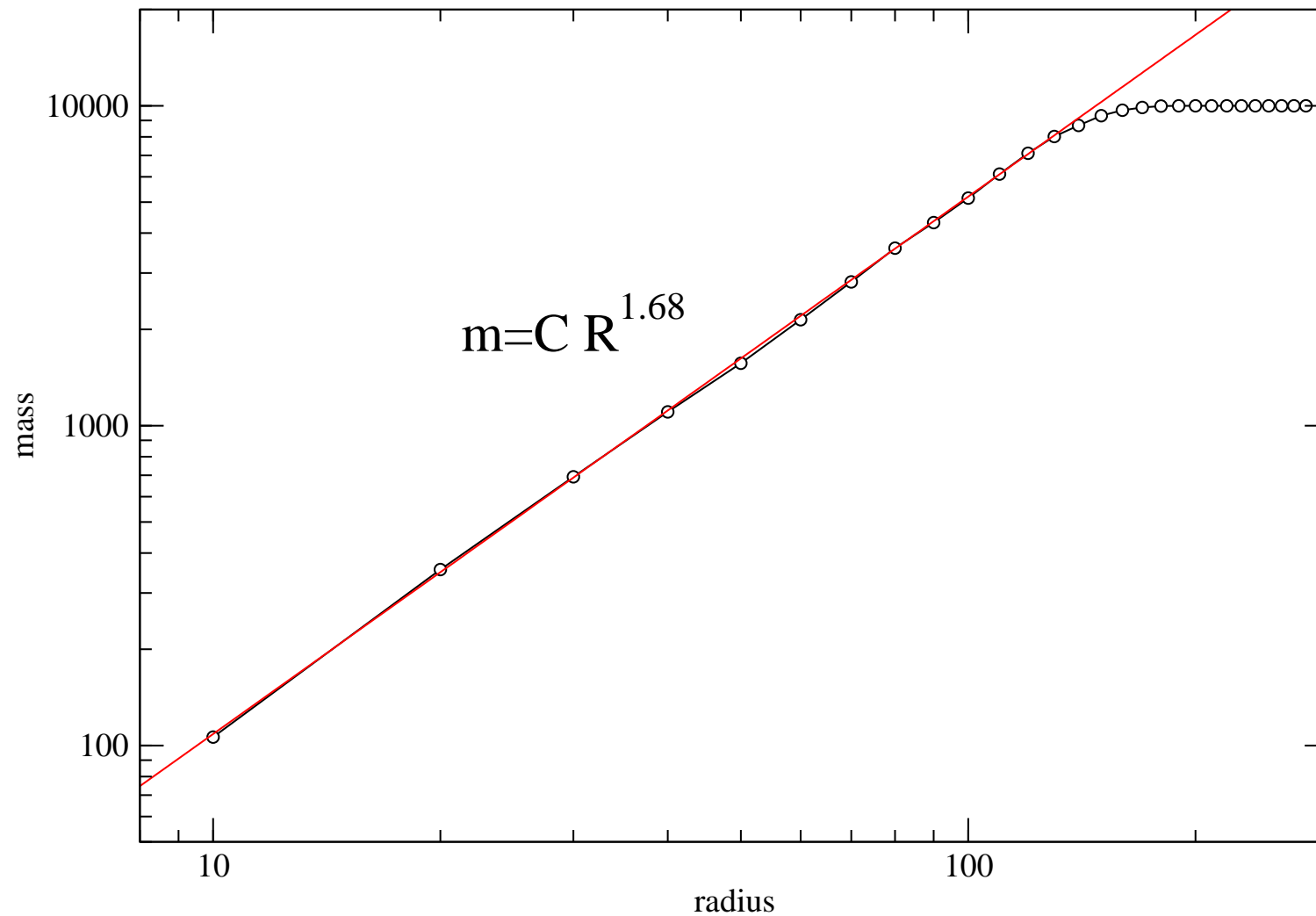


Figure 3: Mass of the DLA cluster as a function of radius R . The dimensionality of the cluster is approximately 1.68.

```

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 boundary. 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 periodic 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 whether 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;
}

```