Tutorial 5: *Simple and importance sampling. Random walks.*

Michael Kopp (2439093)

1 GSL

Install and test Fortunately the GSL was already installed on my system...

I could thus use the GSL for a first program quite easily – this is what `gsl-demo` produced using shell variables as input:

```
$ GSL_RNG_TYPE=taus GSL_RNG_SEED=123 ./gsl-demo
GSL_RNG_TYPE=taus
GSL_RNG_SEED=123
generator type: taus
seed = 123
first value = 2720986350
0.38613
0.55660
0.69627
0.52846
0.06751
0.21717
0.32694
0.50671
0.66234
0.00937
```

Random number generators ("RNG") in GSL Include the header file `gsl_rng.h`

Two structs are used: gsl_rng_type where you can specify what generator you want to use and gsl_rng which is the generator itself. It must be initialized (actually: Allocate memory) via gsl_rng* foo = gsl_rng_alloc(bar); where bar is a gsl_rng_type*. You don't need to use the variable bar but instead you can just use one of the types directly:

```
gsl_rng* foo = gsl_rng_alloc( gsl_rng_taus ) ;
```

which will make foo a Taus-RNG.

GSL uses a generic interface for all different generators: To draw a random number use

```
double r_d = gsl_rng_uniform( foo ) ;
int r_i = gsl_rng_get( foo ) ;
int r_i2 = gsl_rng_uniform_int( foo , 234 ) ;
```

asdf will then be in $[0, 1)$, r_i an integer – the range depending on the generator used, but the range-ends can be determined using gsl_rng_max(foo) resp. gsl_rng_min(foo) – and r_i2 will be an integer from 0 to 233.

To free the memory use

```
gsl_rng_free( foo ) ;
```

in the end.

Shell variables The type (GSL_RNG_TYPE) and seed (GSL_RNG_SEED) of the generator can be given as shell variables (see first paragraph in this section); use gsl_rng_env_setup() to make GSL evaluate the variables and then

```
gsl_rng_type * bar = gsl_rng_default;
```

to actually adopt the type.

2 Calculating π

We randomly select n points in a square of length l containing a circle of diameter d. The probability to hit the circle within the square should be the ratio of the areas

$$p = \frac{\pi(d/2)^2}{l^2} = \frac{\pi}{4} \cdot \left(\frac{d}{l}\right)^2 ,$$

since the possibility to hit any point is equal. So if we count the points within the circle – m – the ration m/n should be close to p.

We use this to "evaluate" – rather *estimate* – π:

$$\pi \approx 4 \left(\frac{l}{d}\right)^2 \cdot \frac{m}{n} \tag{1}$$

This is illuatrated by this short `octave` script using $l = d = 1$ and only one quarter of the square reps. the circle:

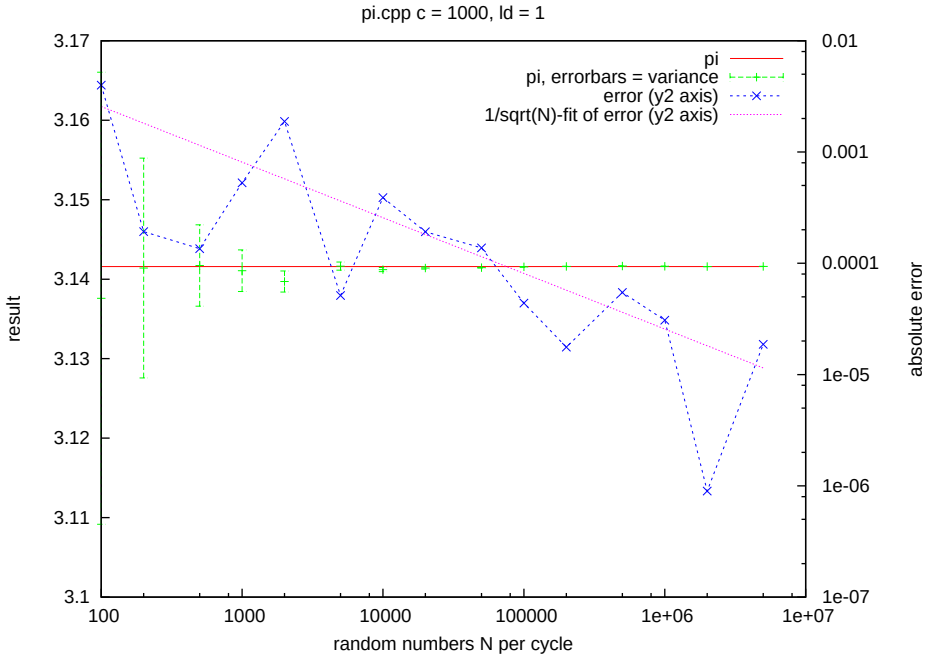


Figure 1: Error of the Program `pi.cpp` for different N.

```
octave:5> r = rand(2,10000) ;
octave:6> p = ( r(1,:).^2 + r(2,:).^2 < 1 ) ;
octave:7> sum(p)/10000
ans =  0.78500
```

As we can see using 10000 random numbers we get quite near to $\pi/4 = 0.785398\ldots$.

I wrote the C++-Program `pi.cpp` which computes π using this method several times ("cycles") with N random numbers each and estimates the error by the variance of the different πs.

In figure 1 you can see the error for differen N. As you can see (cf. the fit $\propto 1/\sqrt{N}$) the error vanishes approximatly with $1/\sqrt{N}$. The large leaps in the error even for large N is (partly) due to the logarithmic scaling of the y2-axis.

In figure 2(a) you can see the behaviour for different ratio l/d. Since the behaviour – the meandering about and below π – looks somewhat strange, I did the same loops using two different RNG – the *MT19937* and the *ranlux* (version ranlxs2) – and the behaviour was reproduced. Then I used a different seed (in fact the nanoseconds of the current second) and this ended the

strange meandering behaviour – cf the blue data in fig. 2(a). So it seems to have been an effect due to the poorly used RNG.

In 2(b) (only the data without meander-effect was used by setting a new seed for each l/d ratio) you can observe that the error increases exponentially (this is a lin-log-plot!) – cf the fit-function. So it's best not to touch the l/d parameter but leave it at $l/d = 1$ since values smaller that 1 don't make sense (you cannot put a circle of diameter d into a square of diameter $l < d$).

Precision and Alternatives The MC method is not good choice to determine π with high accuracy. MC methods can act out their superiority concerning integrals of higher dimensions, but in only two/three dimensions "regular" integration schemes (e.g. polynomial approximation and then integration of these polynomials) can outstrip MC.

An alternative is to compute $\pi = 4 \cdot \text{atan}(1)$. Since sin and cos are declared geometrically, it's no problem to construct/compute values of them with arbitrary precision. What we actually want to know is the angle α where $\cos\alpha = \sin\alpha$ since then $\tan\alpha = \sin\alpha/\cos\alpha = 1$ and thus $\alpha = \text{atan}(1) = \pi/4$.

Although very high precision can be achieved using this method, it's probably not easy since intersection of two functions ($\sin\alpha = \cos\alpha$) is not easy.

Have a look at [2] for different – more promising – methods ;-)

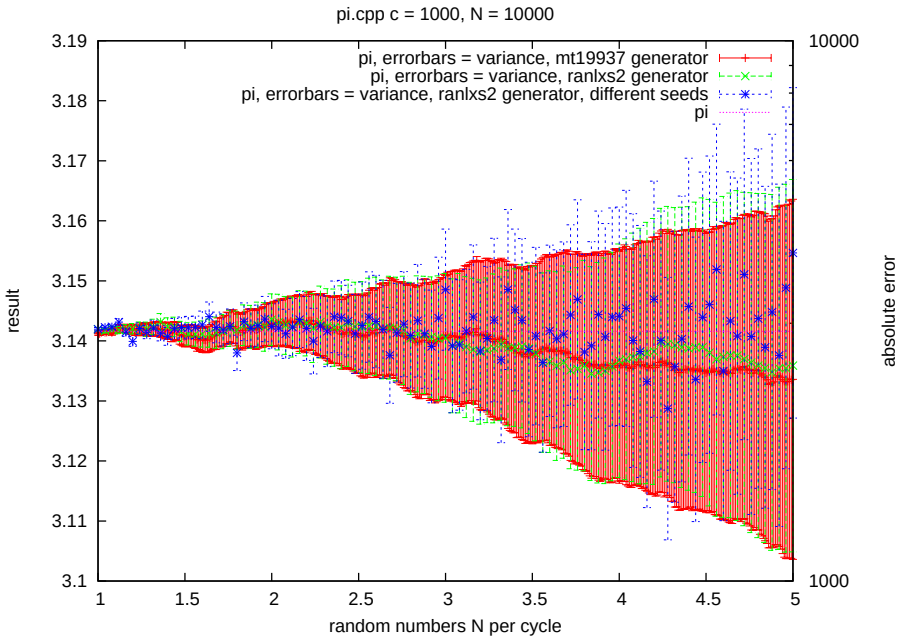
Hypersphere To compute the volume of a hypersphere, we'll do pretty much the same as in the previous chapter but instead of a square, we now consider a hypercube of dimension D containing a hypersphere of the same dimension. From what we learned from two paragraphs before, we'll use a ration of $l/d = 1$.

We must now consider the D dimensional metric

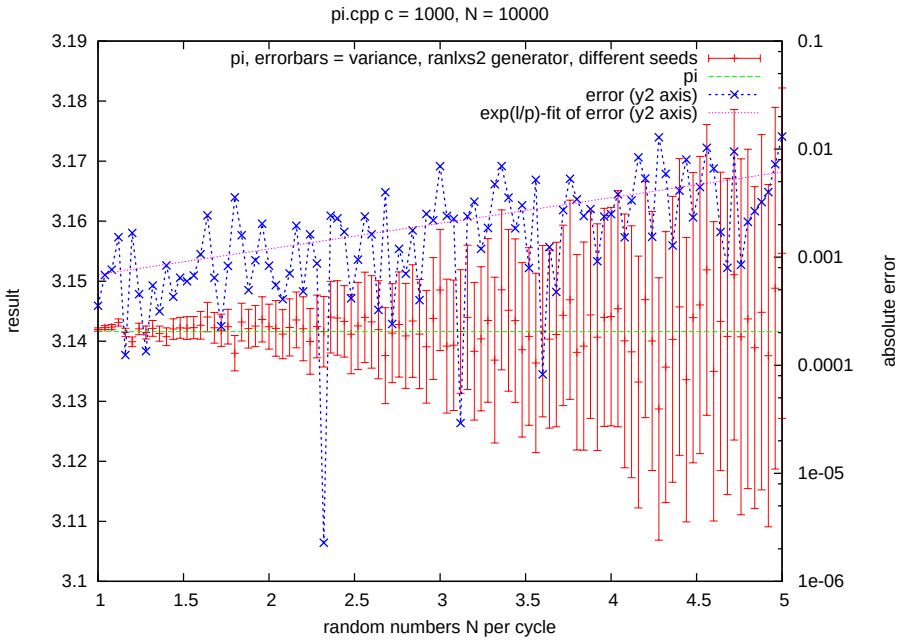
$$d(\boldsymbol{x}, \boldsymbol{y}) = \left(\sum_{i=1}^{D} (x_i - y_i)^2 \right)^{-1/2} ,$$

and bear in mind to create D random numbers for the D coordinates x_i of a random position in space $\boldsymbol{x}$. As our hypercube has (per definition) volume 1 we must only determine the ration m/n and this already is the volume of the hypersphere.

The realization of these considerations is `sphere.cpp`. It uses only random numbers from $[0, 1)$ and thus $V \approx 2^d \cdot m/n$ since only half the possible space *in each dimension* is used.



(a) The generators produce a meandering structure for subsequent ratios l/d if used with the same (the default) seed.



(b) The error increases exponentially.

Figure 2: Error and outcome of the Program `pi.cpp` for different l/d.

According to [1] the volume of an d-sphere of radius $r = 1$ is

$$\frac{\pi^{d/2}}{(d/2)!}$$

so for $d = 6$ the analytical result is approx. 5.1677 and the program `sphere.cpp` outputs $5.16816 \pm 2.819 \cdot 10^{-3}$, so the analytical solution is within the estimated error range of the computed result.

```
$ ./sphere -d 6 -N 100000 -c 1000
The result after      1000 circles with    100000 iterations
    is: 5.16816000
the variance of the volumess is 2.819e-03
```

3 Importance Sampling

If one integrates a function f over $[a, b]$ the standard Monte Carlo idea is to approximate

$$I = \int_a^b f(x)\mathrm{d}x \approx (b - a) \cdot \langle f \rangle \tag{2}$$

where $\langle f \rangle$ denotes the mean

$$\langle f \rangle = \frac{1}{N} \sum_{i=1}^{N} f(x_i) \quad \text{with uniformly distributed random } x_i \in [a, b] \; . \tag{3}$$

This scheme is inefficient if e.g. f varies rapidly or has a large slope etc. Then one has to use a large N to get enough random numbers x_i to the *important* part of f. The more flat f is, the better this approximation works.

The idea is, to use a weighting function h with $h(x) > 0$ for $x \in \operatorname{supp} f$ which behaves roughly similar to f, so that f/h does not have these slopes, vibrations etc. any more. In oder to keep the integral unchanged, the new integrand is $f = f/h \cdot h$. Up to now, nothing's won.

The trick is now, to turn $h\,\mathrm{d}x$ into a new variable $\mathrm{d}\xi$:

$$h(x)\,\mathrm{d}x = \mathrm{d}\xi \Rightarrow \xi = \xi(x) = H(x) = \int_a^x h(x')\mathrm{d}x' \; , a \leq H(x) \leq b \; . \tag{4}$$

H shall then be used to substitute x in the integrand:

$$x \to x(\xi) = H^{-1}(\xi) \; .$$

Since $x \in [a,b]$, h must be scaled properly so that this holds for $x(\xi)$ with $\xi \in [H(a), H(b)]$. Using a scaling factor A in h (so using $A\,h$ instead of h) allows to scale $[h(a), H(b)]$ to $[0,1]$. This is useful, since one does not have to think about the factor $(\beta - \alpha)$ in (5) as it's $(1-0) = 1$.

We thus get

$$I = \int_a^b f(x)\mathrm{d}x = \int_a^b \frac{f(x)}{g(x)}\mathrm{d}H(x) = \int_{\alpha:=H(a)}^{\beta:=H(b)} \frac{f(\xi)}{g(\xi)}\mathrm{d}\xi \approx (\beta-\alpha)\left\langle \frac{f}{g} \right\rangle_h \quad (5)$$

where the index "h" means, that the random numbers are not distributed uniformly but with the distribution function $H(x)$.[1]

Now the area with large h will be sampled more often, so h should be chosen so that it's large where f is large etc. – in principal it would be perfect to use $h(x) \propto f(x)$, but then we would need to know the integral in (4) to estimate the integral... This would be rather stupid ;-) If integrating a function with rapid oscillations, it's wise to use an h that's large near these oscillations.

So h must be chosen to approximate f rather closely, it should be rather easy to integrate analytically and the integral should be possible to invert.

Example: Gauss' Bell We want to integrate

$$f(x) = \exp(-x^2/2) \text{ over } x \in [0,b] \,.$$

As a weighting function we choose

$$h(x) = A\exp(-x/\lambda)$$

since it has an easy integral

$$H(x) = A\lambda\,(1-\exp(-x/\lambda)) \;\Rightarrow\; H^{-1}(\xi) = -\lambda \ln\left(1 - \frac{\xi}{A\lambda}\right) \,.$$

As we see, $H(0) = 0$, so we choose A so that H satisfies $H(b) = 1$ – thus

$$A = \frac{1}{\lambda}\,\frac{1}{1-\exp(-b/\lambda)} \,.$$

These functions are shown in figure 3.

This was implemented in `importance.cpp`.

To compare the solutions, f was computed for different b using `octave`:

[1] In the application, however, the random variables *will* be uniform – but the uniform x_i will be processed using H: $\xi_i = H(x_i)$.

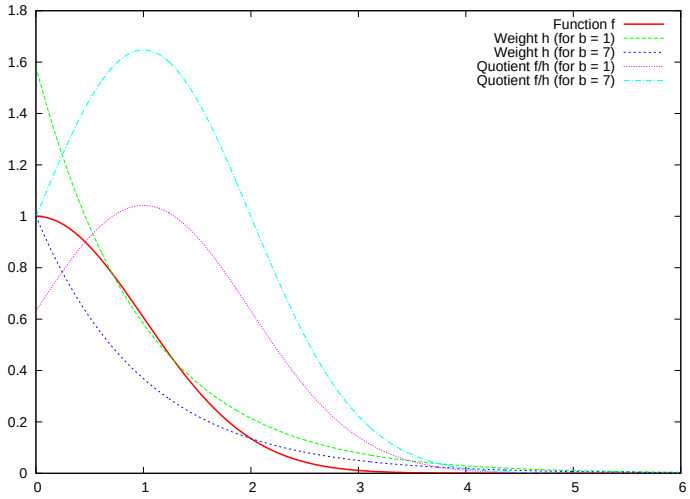


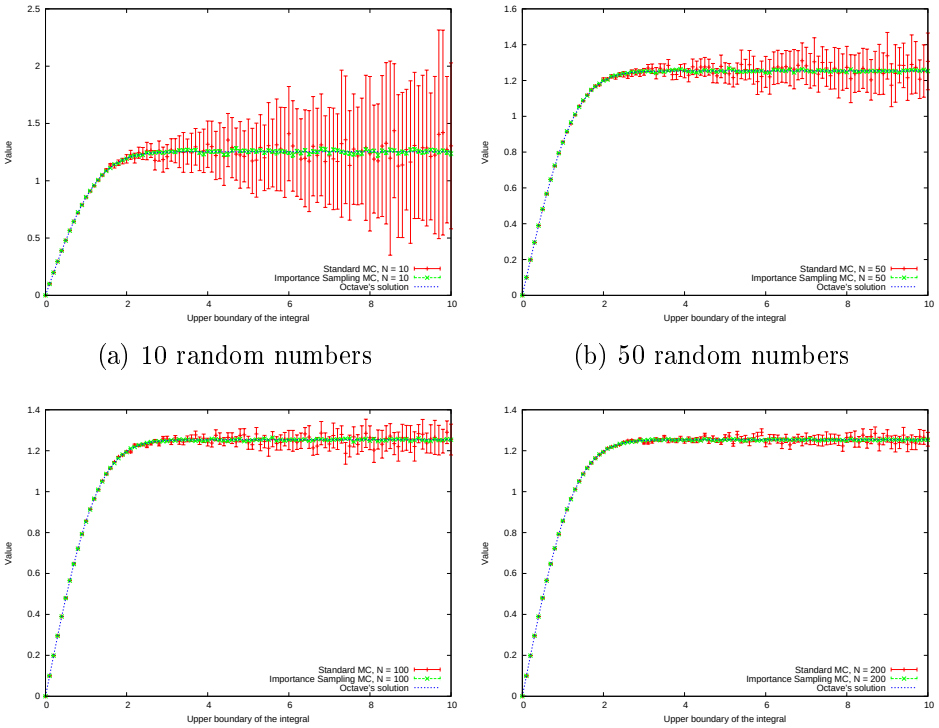
Figure 3: The functions f and h and the quotient f/h for different As (chosen for different b).

```
octave:1> function y = f(x)
> y = exp(-x.^2/2 ) ;
> end
octave:2> I = 0 ;
octave:3> b = [0:0.1:10] ;
octave:4> for j = b
> I( length(I)+1 ) = quad( "f" , 0 , j ) ;
> end
octave:5> b = [0 b] ;
octave:6> integral = [ b' I' ] ;
octave:7> save( "-ascii" , "exp_oct" , "integral" ) ;
```

The results are presented in figure 4. The simulation was conducted for different N, each time running 100 cycles. Of these cycles the mean and variance were computed – this is what you can find in the figure (the variance is plotted as error bars).

You can see clearly that the standard MC values (red) are less accurate: The mean is more often further away from octave's solution than the importance sampled mean (green) and the variance is much higher.

You can also observe, that the variance (thus the insecurity) for larger b gets bigger and bigger for the standard algorithm. This is due to the fact, that for larger b more function values are very small. A look at figure 3 shows, that the quotients f/h does not get that small that fast.



(a) 10 random numbers

(b) 50 random numbers

(c) 100 random numbers

(d) 200 random numbers

Figure 4: Comparison of Monte Carlo with and without importance sampling. Octave's results were used as reference.

The choice of h was probably not the best – but it's better using it than not doing importance sampling at all.

So obviously the technique of importance sampling is superior...

4 Random Walk in one dimension

My randowm Walk Program for one dimension can be found in `rw1d.cpp`.[2]

Mean end position I had to performe quite some cycles to get the desired result of a mean end distance of zero; cf the program's output

```
$ GSL_RNG_SEED=$(date +%N) ./rw1d -N 1000 -b 50 -c 600000
GSL_RNG_SEED=4
```

[2] And yes: I read the provided one and did understand it ;-)

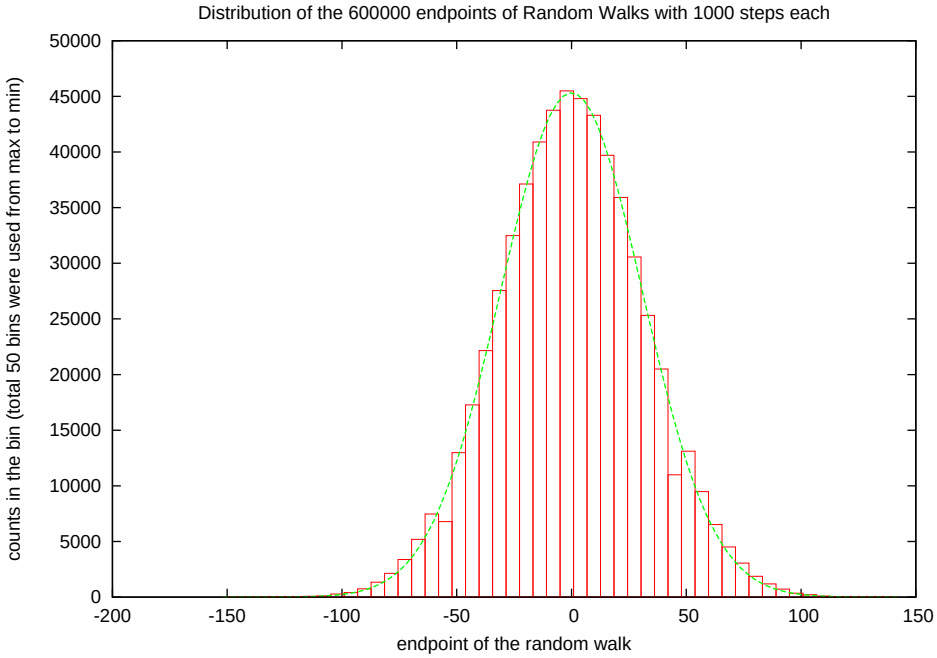


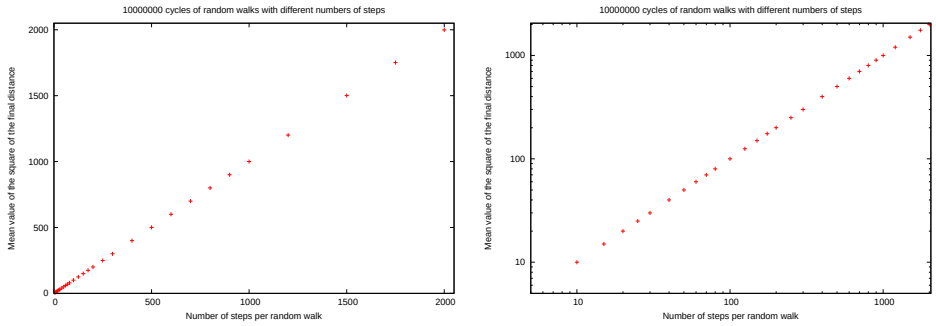
Figure 5: Histogram of the endpoints after several (600000) random walks with 1000 steps each. Width: 953

```
#After   600000 cycles of      1000 walks with p =
    0.50000:
# the mean endpos is 1.6350000e-02 +/- 9.992643e+02
# the mean of the squared endpos is 9.9926286e+02
```

Here the estimate of the error (in fact that's the variance) is bigger than the displacement of the mean value from zero. See also figure 5. Here you can see, that the distribution is centered nicely around zero

Mean of the squared end position The "experiment" was conducted several times with different number of steps each time (and a different seed each time). In figure 6 you can see that the mean of the squared end position is proportional to N – using `gnuplot`'s fit-algorithm I was able to determine the constant of proportionality; it's 1.00016 ± 0.000106. This is an accuracy of 0.0106% since the analytical[3] solution is simply 1.

[3]In some other tutorial we showed this analytically...



(a) lin-lin (b) log-log

Figure 6: Mean of squared end position for different numbers of steps for the one dimensional random walk.

Different Probability Now the probability to move "right" was set to $p = 0.8$. In figure 7 you can see, that the endpoints are still distributed in a Gaussian way – compare fig. 5.

To check the behaviour of the random walks for different probabilities, RWs for different p were made. In figure 8 you can clearly see that the mean endpoint shifts for different p. This is totally understandable as we expect a final point of

$$p \cdot N - (1-p) \cdot N \ :$$

Walk right with probability p and thus in the opposite direction with probability $(1-p)$ and this holds for each single step...

The functional dependency of the variance of the mean values seems to be $4Np(1-p)$, which coincides perfectly with the analytical solution.[4]

A qualitative interpretation is the following: For $p = 0.5$ the particle does not follow any "destination". It travels totally randomly around and thus generates a rather broad Gaussian bell. If $p = 1$ (or near to 1), it's more and more destined where to go. For $p = 1$ it knows exactly that it has to travel every step right – thus the Gaussian bell is quite small (in fact it won't be a Gaussian any more but a simple box of width 1). In between for $p \in [0.5, 1]$ the Gaussian distribution becomes leaner the more p differs from $p = 0.5$. This matches exactly the "measured" outcome.

This "model" can also explain why the mean of the square distance grows rapidly for values of p different from 0.5: The particle is "drawn" into one (more or less) particular direction. The more it's destined, the farther it gets

[4] In fact, this solution is only valid for large N as it uses the inequality of Stirling – but this should be satisfied here.

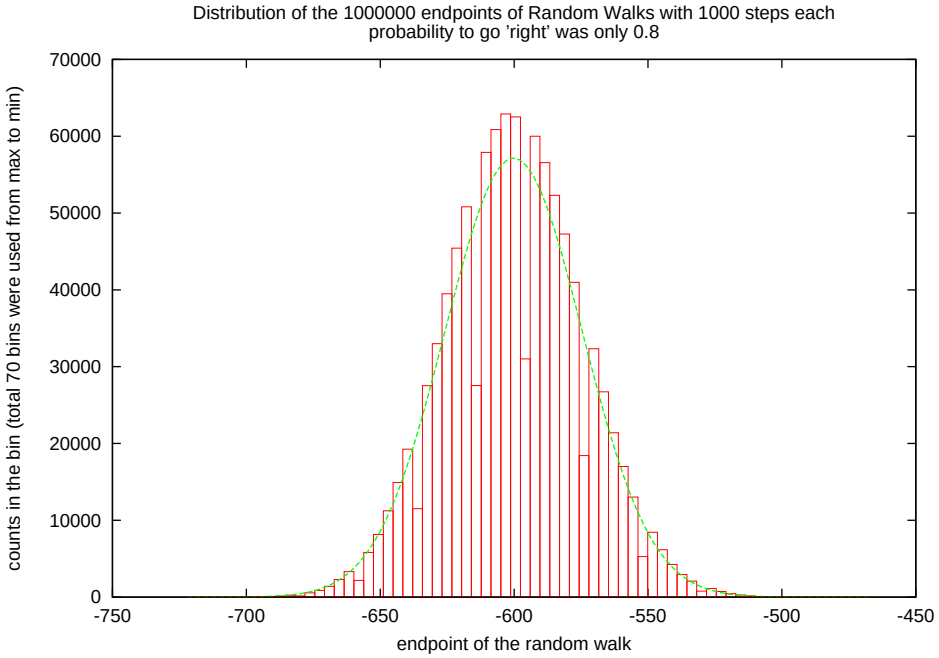


Figure 7: Like fig 5 but with $p = 0.8$. Width: 654

away from the origin etc... A short discussion in [3] makes it clear to me that I am totally satisfied with this hand waving "explanation"...

5 Random Walk in two dimensions

I wrote the program `rw2d.cpp`. It uses a *board* (see file `board.h`) of integers and when the random walk encounters one of these fields, the integer associated to it is incremented. So if I'm running in SAW[5] mode, I can check, whether the field has already been visited and in that case simply move into another direction.

One must, however, be careful as it may happen, that a SAW has no possibility to go on: When every neighbouring field has already been visited once, the SAW must be finished. In my implementation it is then discarded.

[5] self avoiding walk

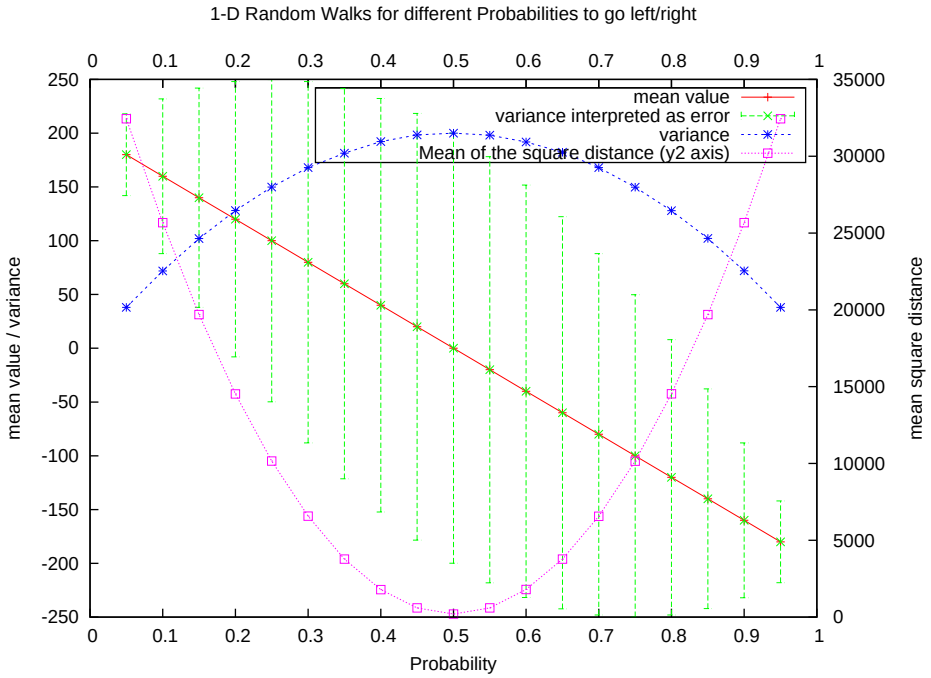


Figure 8: Characteristics of 1-D-RWs for different probability to go right/left.

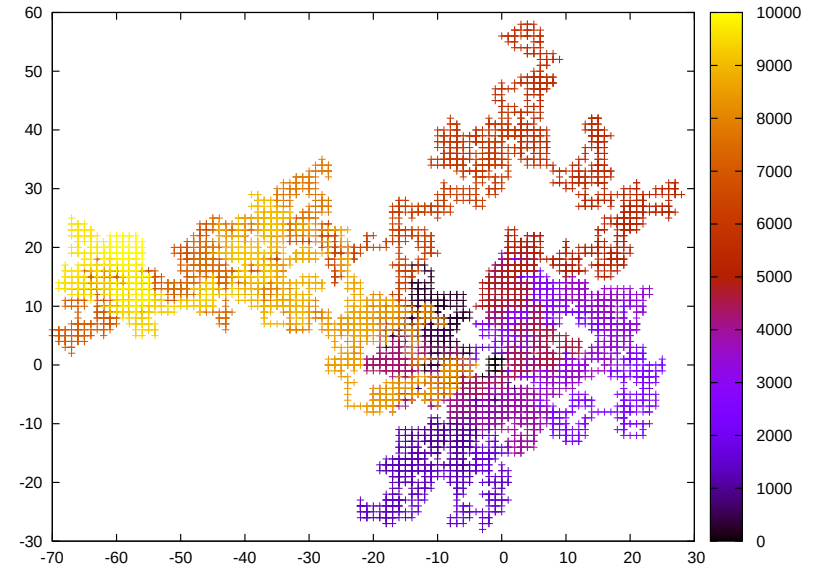
Contact or no contact First I wanted to know, how long a random walk has to be so that it contains a contact[6]. For larger random walks this is obviously almost always true; in fact it's very hard to find SAWs for larger N (the number of random steps to go). I was patient enough to find some for $N \sim 400$, but not much higher. See figure 9 for some nice, colourful plots.

The really meaningful plot is in figure 10(a). It shows, that the probability to find a SAW (without specially preparing the algorithm to find some) in a set or random walks decreses like $\exp(-N/\alpha)$ with some $\alpha > 0$. So as you can see, it's really hard to find SAWs for $N \gtrsim 30$.

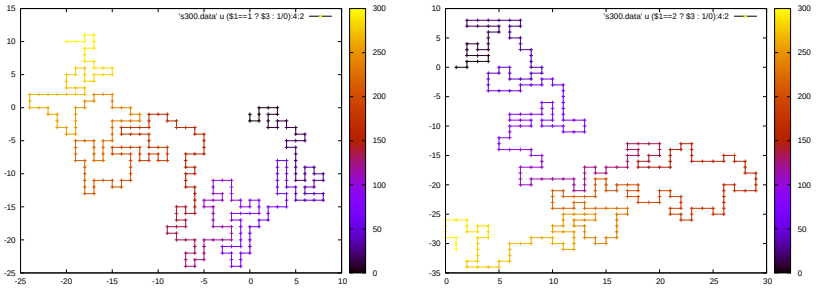
For larger N see figure 10(b). As you can see the linear trend from figure 10(a) continues: The number of contacs is proportional to the number of steps.

End-to-end distance In figure 11 you find the (squared) end-to-end distance for random walks. For each length, 400 RWs were computed. The fits

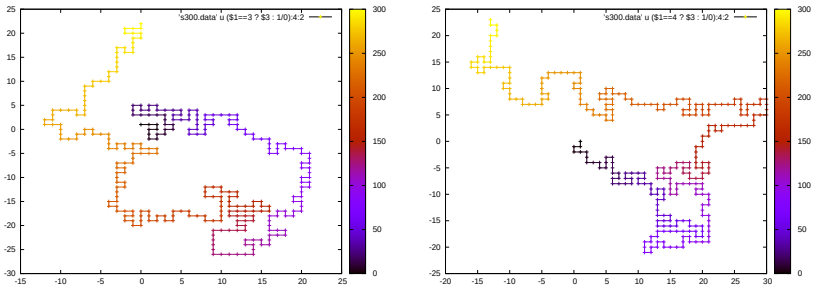
[6]In the following text, "contact" will refer to a trajectory which crosses at least one point several times.



(a) RW with contacts: $N = 10000$

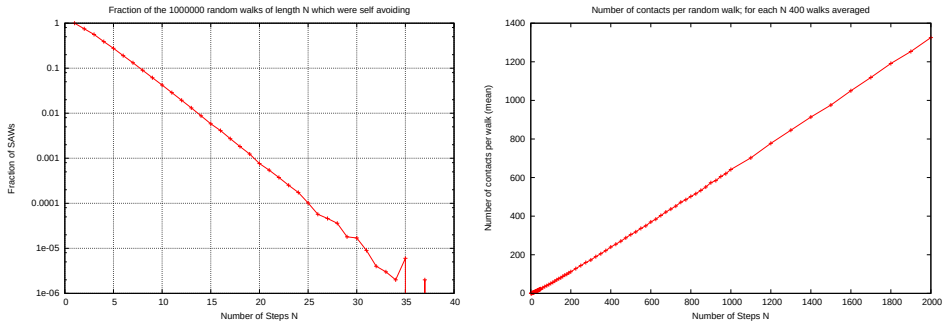


(b) SAW $N = 300$ (c) SAW $N = 300$



(d) SAW $N = 300$ (e) SAW $N = 300$

Figure 9: Random walks: Its far more easy to simulate RSs with contacts than Self Avoiding Walks. The colour scale is a timescale.



(a) How many contacs do the random walks have for different N: Depicted here is the fraction of SAWs out of 1000000 walks.

(b) Contacts per random walk for different N in 2 dimensions.

Figure 10: Contacts in the 2 dimensional random walk.

there are

$$d(N) = 0.841696N^{0.508425} \text{ and } d^2(N) = 0.783768N^{1.0369} \ .$$

So we have a linear dependency of d^2 – like in the one dimensional case. What has changed is the constant of proportionality; it's now almost $1/\sqrt{2}$.

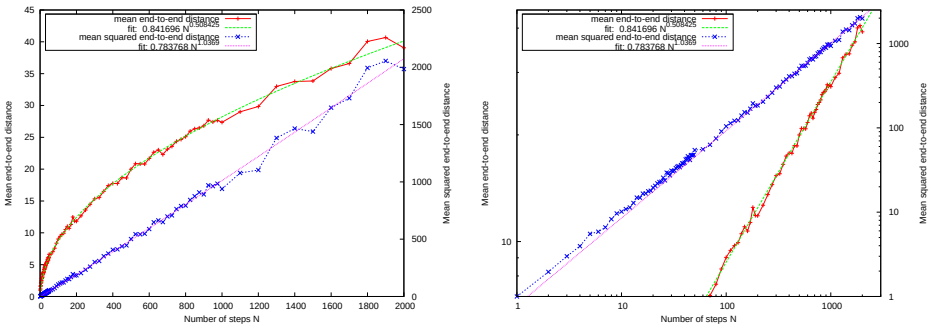
Possible step length for SAWs To check, what range of N does make sense to work with for, I conducted several simulations generating only SAWs with different N, for $N \in [0, 200]$, 10000 walks each, for $N \in [200, 300]$ only 100 walks each since the computational time increased very fast for large N. In fact it did increase that fast, that I did not try to get walks larger than $N \sim 400$ (or at least only single ones, not whole bunch of walks).

In figure 12 you can see, that the numbers of walks which had to be aborted since they confined theirselves "to death" increases exponentially. This is what urges the computational time higher and higher.

You can see there, too, that the length (in N) of the aborted SAWs seem to saturate at about ~ 70. So it seems, that many of them "die" young – the projected length is 350; 5 times more!

End-to-end length of SAWs In figure 13 the (squared) end-to-end length of SAWs is plotted for different N. The plots show, that the length behaves almost linear (with offset: $y = mx + \mathbf{c}$) for larger N ($N \in [70, 200]$), but the log-log plot reveals, that it's a power functional dependency.[7]

[7] As $\log y = \alpha \log x + \beta$ it's $y = x^\alpha \cdot \mathrm{e}^\beta$.



(a) lin-lin (b) log-log

Figure 11: Mean (squared) end-to-end distance of two dimensional random walks for different N.

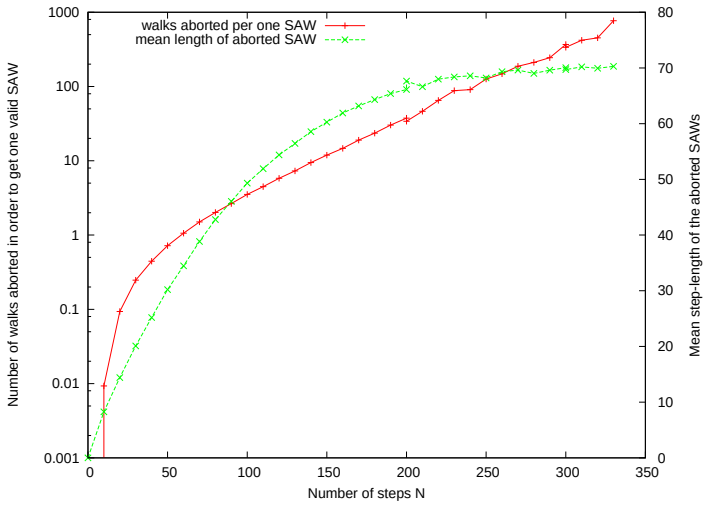
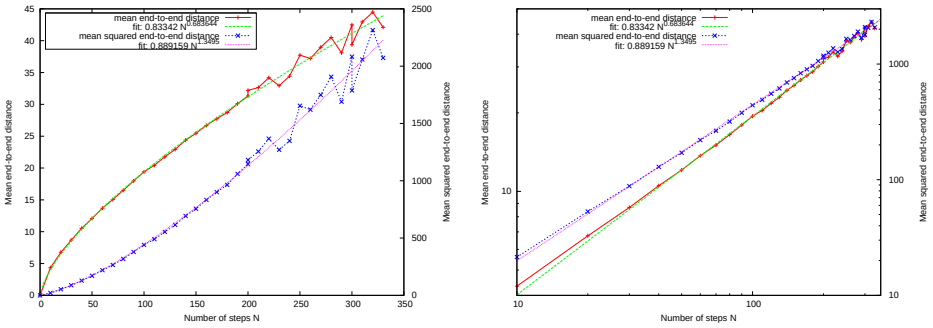


Figure 12: Effort to produce SAWs.



(a) lin-lin (b) log-log

Figure 13: Mean (squared) end-to-end distance of SAWs for different N.

As you can see in the figures, some power functions were fitted with

$$d(N) = 0.83342N^{0.683644} \text{ and } d^2(x) = 0.889159N^{1.3495} .$$

References

[1] Wikipedia: N-sphere. http://en.wikipedia.org/wiki/N-sphere

[2] Wikipedia: Numerical approximation of π. http://en.wikipedia.org/wiki/Numerical_approximations_of_pi

[3] Wolfram Alpha: Random Walk–1-Dimensional. http://mathworld.wolfram.com/RandomWalk1-Dimensional.html