[Monte Carlo Direct Method]
[Monte Carlo Error Analysis]
[Estimating the Monte Carlo Error]
[Basic Calculus with Mathematica]
[Stochastic Processes]

Monte Carlo Direct Method

Monte Carlo integration uses random sampling of a function to numerically compute an estimate of its integral.
Suppose that we want to integrate the one-dimensional function f (x) from a to b:

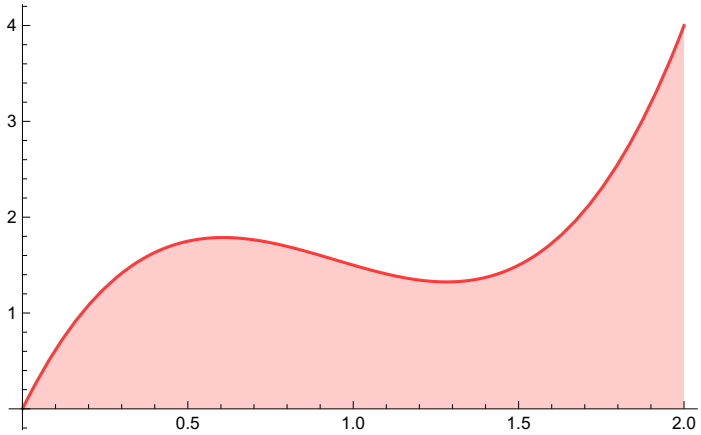
$$F = \int_a^b f(x)\, dx$$

The integral sign $\int$ represents integration.
The symbol dx, called the differential of the variable x, indicates that the variable of integration is x.
The function f(x) to be integrated is called the integrand. A function is said to be integrable if the integral of the function over its domain is finite.
The points a and b are called the limits of the integral. An integral where the limits are specified is called a definite integral.
The integral is said to be over the interval [a, b].

Now let's first approach it with classical integration.

Our integrand is the function : $7x - 8.5x^2 + 3x^3$ and its domain $\in$ R[0,2].

```
Clear[f, x];
f[x_] = 7 x - 8.5 x^2 + 3 x^3;
Plot[f[x], {x, 0, 2}, Filling → Axis,
 PlotStyle → {Red, Opacity[0.75]}, PlotRange → Full]
```



Integration is generally referred as to find the area under the fnc curve.

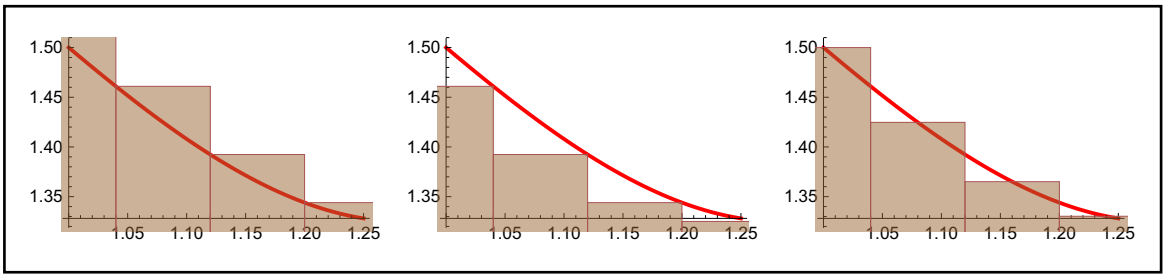
Now the general approach for numerical deterministic integration is to :
- subdivide the area under the curve by many rectangular bins,
- calculate for each their respective area and
- eventually sum then all up

```
Clear[a, b, di, intervals];
a = 0; b = 2; (* domain *)
nbins = 25; (* integration steps *)
di = (b - a) / nbins; (* base *)
intervals = Range[a, b, di]; (* intervals *)
```

Above result is actually the base length of our rectangular bins. In fact, we subdivided the x-axis extension of our integrand by n-steps.
Now we need to multiply the base by the height to find the area our rectangles.
What is the height of our rectangles ? It's the integrand evaluation at that point.

One little caveat is that we can start from the left of our basis or from the right or from the center.

```
Framed@GraphicsRow[Plot[f[x], {x, 1, 1.25}, PlotStyle → {Red, Thick},
    Epilog → {Opacity[.5], Brown, EdgeForm[Darker@Pink],
      Rectangle @@@ (Transpose /@ Transpose[{Partition[intervals, 2, 1],
          Partition[Riffle[f /@ #@intervals, 0, {1, -2, 2}], 2]}])},
    ImageSize → Small] & /@ {Most, Rest, MovingAverage[#, 2] &}]
```




```
leftSum = Sum[di * f[i], {i, Most@intervals}];
rightSum = Sum[f[i] di, {i, Rest@intervals}];
middleSum = Sum[f[i + di / 2] di, {i, Most@intervals}];
N@{leftSum, rightSum, middleSum}
```

```
{3.1744, 3.4944, 3.3328}
```

Above is the result of nbins integration =25 (i.e. we subdivided the x by 25).
Let's use Mathematica symbols to calculate the correct true answer and compare it with our.
We see the middle point sum method is more precise than the other twos.

```
NIntegrate[f[x], {x, 0, 2}]
```

```
3.33333
```

We can still use Mathematica and NIntegrate to do what we've done manually above.

```
NIntegrate[f@x, {x, 0, 2},
    Method → {"RiemannRule", "Type" → #, "Points" → 25}] & /@ {"Left", "Right"}
```

NIntegrate: NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near {x} = {1.87492}. NIntegrate obtained 3.333251461338975` and 0.00015961071549491825` for the integral and error estimates.

NIntegrate: NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near {x} = {1.875}. NIntegrate obtained 3.33341529051659` and 0.0001593560859425433` for the integral and error estimates.

```
{3.33325, 3.33342}
```

If we keep increasing the integration steps we'll keep getting closer and closer to the correct answer.
To what point we should keep increasing our number of steps ? To have them an infinitesimal width dx.

When the step width approaches dx we don't talk anymore about summation but about integration and use the integral symbol.

$$\int_0^2 \left(7\,x - 8.5\,x^2 + 3\,x^3\right) \mathrm{d}x$$

```
3.33333
```

==========

==

Now let's approach this with Monte Carlo.

We can approximate this integral by averaging samples of the function *f* at uniform random points within the interval.
Given a set of N uniform random variables $X_i \in [a,b)$ with a corresponding PDF of 1/(b – a) (i.e. the pdf of a uniform distribution)
the Monte Carlo estimator for computing F is

$$\left\langle F^N \right\rangle = (b-a)\,\frac{1}{N-1}\sum_{i=0}^{N} f\,(X_i)$$

(take care that the above is actually the sample mean of X_i, aka the mean estimate, also called the *primary* MC estimator)
The random variable $X_i \in [a,b)$ can be constructed by :
- warping a canonical random number uniformly distributed between zero and one, $\xi_i \in [0, 1)$:

$$X_i = a + \xi_i\,(b-a)$$

With that, our estimator becomes :

$$\left\langle F^N \right\rangle = (b-a)\,\frac{1}{N}\sum_{i=0}^{N-1} f\,(a + \xi_i\,(b-a))$$

Note that $\langle F^N \rangle$ means 'approximating *f* with N samples' and since $\langle F^N \rangle$ is a function of X_i, it's itself a random variable.

```
n = 600;
(b - a) 1/n Sum[f[a + RandomReal[] * (b - a)], {i, 0, n}] // N
Sum[f[a + RandomReal[] * (b - a)], {i, 0, n}] * (2 / (n - 1))

(* while the above procedures are the same they
 will return different results due to the random numbers *)
```

```
3.36787
```

```
3.38917
```

```
f[a + RandomReal[] * (b - a)]
```

```
1.40848
```

==========

[LT->Theory->2014 - Monte Carlo Integration Tecniques]

It is easy to show that the expected value $\langle F^N \rangle$ is in fact F.

First recall the proofs about Expected Values and Variance.
The expected value of a random variable Y=f(X) over a domain μ(x) is

$$E[Y] = \int_{\mu(x)} f(x)\, pdf(x)\, dx \tag{1}$$

while its variance is

$$\sigma^2[Y] = E\left[(Y - E[Y])^2\right] = E\left[(Y - \mu_y)^2\right] \tag{2}$$

where σ, the standard deviation, is the square root of the variance

$$\begin{aligned} E[aY] &= aE[Y] \\ \sigma^2[aY] &= a^2\, \sigma^2[Y] \end{aligned} \tag{3}$$

furthermore, the expected value of a sum of random variables Y_i is the sum of their expected values

$$\begin{aligned} &E\left[\sum_i Y_i\right] = \\ &\sum_i E[Y_i] \quad (* \text{ this is called, the linearity of the expectation value } *) \end{aligned} \tag{4}$$

From there we derive a simpler expression for the variance

$$\begin{aligned} &\sigma^2[Y] = E\left[Y^2\right] - E[Y]^2 \quad (* \text{ see expectation algebra in part1 } *) \\ &\sigma^2\left[\sum_i Y_i\right] = \sum_i \sigma^2[Y_i] \quad (* \text{ for uncorrelated randoms } *) \end{aligned} \tag{5}$$

With the above in mind we can demostrate that $\langle F^N \rangle$ is in fact F

(so that the error here would be zero.. ie $\langle F^N \rangle$ is an unbiased estimator)

$$E\left[\langle F^N \rangle\right] = E\left[(b-a)\, \frac{1}{N} \sum_{i=0}^{N-1} f(X_i)\right]$$

$$= (b-a)\, \frac{1}{N} \sum_{i=0}^{N-1} E[f(X_i)] \quad (*\text{from 4 and 3a}*)$$

$$= (b-a)\, \frac{1}{N} \sum_{i=0}^{N-1} \int_a^b f(x)\, pdf(x)\, dx \quad (*\text{from 1}*)$$

$$= \frac{1}{N} \sum_{i=0}^{N-1} \int_a^b f(x)\, dx \quad (*\text{since pdf}(x)=1/(b-a)*)$$

$$= \int_a^b f(x)\, dx$$

$$= F \mid \langle\langle F \rangle\rangle$$

Eventually due to 'the law of large numbers' the estimator $\langle F^N \rangle$ becomes closer and closer to F

$$\text{Pr}\left\{\lim_{N\to\infty} \langle F^N \rangle = F\right\} = 1$$

$$\text{Pr}\left\{\lim_{N\to\infty} \langle \bar{A} \rangle = \langle A \rangle\right\} = 1$$

So with Mathematica code we can approach this with :

```
m = 100;
1/m Sum[(b - a) 1/n Sum[f[a + RandomReal[] * (b - a)], {i, 0, n}], {i, 0, m}] // N
```

```
3.37866
```

==========

==

[LT->Theory->Sampling->MCErrorAnalysis->Estimating Errors Reliably in Monte Carlo Simulations]
(This also anticipates the MC error derivation we'll find below)

However take care that in some literature something is the estimate $\bar{F}$ and something else is ⟨F⟩ which is the true expectation,
there, the above relation is defined as $\langle \bar{A} \rangle$=⟨A⟩ , so that the deviation of $\langle \bar{A} \rangle$ from the true expectation ⟨A⟩ is a fluctuating quantity $\triangle_A$.

So below,
$\bar{A}$ is the mean estimate of A while
⟨A⟩ is the true expected value of A (sometimes written like ⟨⟨A⟩⟩). And
$\langle \bar{A} \rangle$ is the expected mean estimate of A.

More generally it would be written as $\hat{A}$=A (where the hat^ stays for expectation as ⟨ ⟩. However then we don't have the upper space for another symbol so to write the expected mean value of A with the ^ notation, we use the μ that stays for mean.. $\hat{\mu}$ is the expected mean value of A, ie. $\langle \bar{A} \rangle$

And just to make things easier ;) we may add the distinction between an estimate and an estimator.
If we write $\hat{\mu} = \bar{x}$; we want to say that the estimate μ is the mean $\bar{x}$. Formally this is written as $\hat{\theta} = \bar{x}$
But if we wanna talk about the estimator (not the estimate) then we should use the uppercase theta Θ.

Anyway :) due to the linearity of the expectations

$$\left\langle \bar{A} \right\rangle = \left\langle \frac{1}{N} \sum_{i=1}^{N} A_i \right\rangle = \frac{1}{N} \sum_{i=1}^{N} \langle A_i \rangle = \frac{1}{N} \sum_{i=1}^{N} \langle A \rangle = \langle A \rangle$$

where ⟨A⟩

$$\langle A \rangle = \sum_{x} A\,(x)\; p\,(x)$$

and

$$A_i \equiv A\,(x_i)$$

Similar reasoning allows the calculation of the average of the square of the sample mean

$$\left\langle \bar{A}^2 \right\rangle = \left\langle \left(\frac{1}{N} \sum_{i=1}^{N} A_i \right)^2 \right\rangle = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N} \langle A_i\, A_j \rangle$$

$$= \frac{1}{N^2} \sum_{i=1}^{N} \left\langle A_i^2 \right\rangle + \frac{N-1}{N} \langle A \rangle^2$$

$$= \frac{1}{N} \left\langle A^2 \right\rangle + \frac{N-1}{N} \langle A \rangle^2 \qquad (6)$$

where we have inserted the definition of the average,

$$\bar{A} = \frac{1}{N} \sum_{i=1}^{N} A_i$$

then used the linearity of the expectation value,
and also exploited the fact that for independent samples x_i and x_j, the expectation value for i≠j factorizes as

$$\langle A_i\, A_j \rangle = \langle A_i \rangle \langle A_j \rangle = \langle A \rangle^2$$

The statistical error $\triangle_A$, the root-mean-square deviation of the sample mean $\bar{A}$ from the true expectation value ⟨A⟩,
is thus given by

$$\triangle_A^2 \equiv \left\langle \left(\bar{A} - \langle A \rangle \right)^2 \right\rangle = \frac{1}{N^2} \sum_{i=1}^{N} \left\langle A_i^2 \right\rangle - \frac{1}{N} \langle A \rangle^2$$

$$= \frac{1}{N} \left(\left\langle A^2 \right\rangle - \langle A \rangle^2 \right) \equiv \frac{1}{N} \sigma_A^2$$

which is the basis of the central limit theorem.

It is, however, more useful to express the error in terms of the sampled A_i 's.

A naïve guess would be to estimate the variance as $\bar{A^2} - \bar{A}^2$, where

(one is the average of the squared A and the other is square of the averaged A)

$$\bar{A^2} \equiv \frac{1}{N} \sum_{i=1}^{N} A_i^2$$

If we calculate the expectation values via (6) we get

$$\left\langle \bar{A^2} - \bar{A}^2 \right\rangle = \frac{N-1}{N} \sigma_A^2$$

Thus the estimator is

$$\sigma_A^2 \Big| \text{Var}[A] \approx \frac{N}{N-1} \left(\bar{A^2} - \bar{A}^2 \right)$$

where the (small) fluctations in the right-hand side have been ignored.

Eventually, taking the square root , we obtain the final result

$$\triangle_A = \sqrt{\frac{\text{Var}[A]}{N}} \approx \sqrt{\frac{\bar{A^2} - \bar{A}^2}{N-1}} \tag{7}$$

The -1 in the denominator, which is irrelevant for large values of N, reflects the loss of one piece of information in calculating the sample mean.

==========

==

Monte Carlo Error Analysis

Btw, with standard integration we are almost there with just 25 'samples' ...

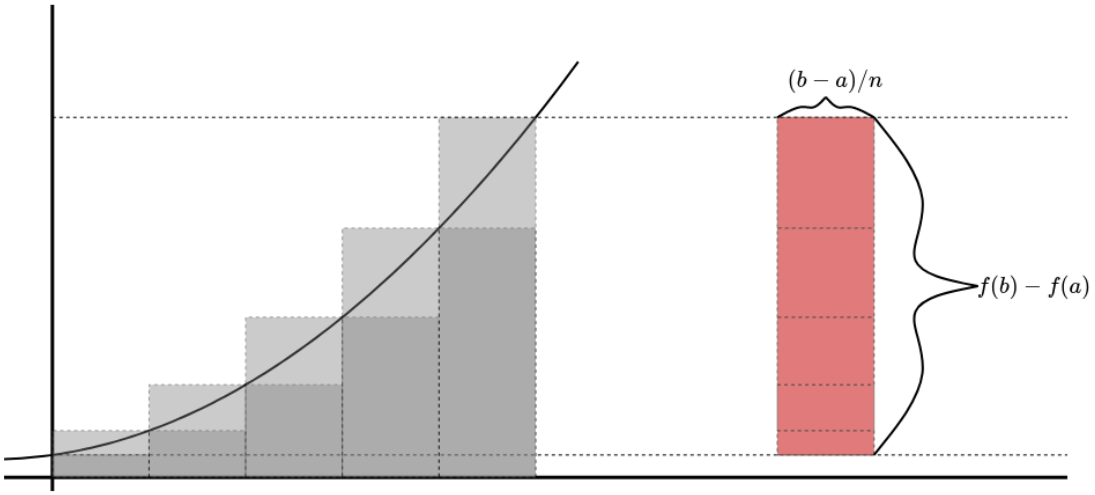
how it comes that with Monte Carlo we need 50000 samples to get there ?

(Or how can a random array be better than a grid ?)

Now the error for the midpoint method (Riemann sum) is

$$\propto \frac{b - a}{N^d}$$

this is because for a monotone function in a-
b



and more generally :

$$Er \propto M \frac{(b - a)^2}{n^d}$$

==========

==

[LT->Theory->Sampling->MCErrorAnalysis->Error Estimates for the Monte Carlo Method]

while for the error with Monte Carlo method

we first draw some introduction. We know the following relationship between our estimation and true value

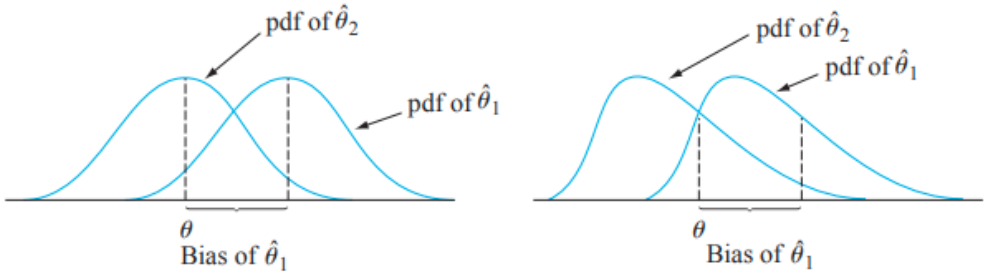
$$\hat{\theta} = \theta + \text{error}$$

We saw just above the potentially for an unbiased estimator the error will be 0 because

$$\hat{\theta} - \theta = 0$$

That is, $\hat{\theta}$ is unbiased if its probability sampling distribution is 'centered' at the true value of the parame-

ter (outcome).



We have the following entities for our setup :

$\bar{x}$ is the simulated (or the estimated) output (ie. the above $\bar{A}$, F^N or the below $\hat{\mu}_n$)
μ_x is the true expected output, (ie. the above ⟨A⟩ or the below μ)
x_i is the generated sample where *i=1 to N,* *(ie. replicates (A_i) of the outcome of a certain f*[x])
$\vec{x}_i$ is the configurator (also called the locator)

Practically it is like this :

$\bar{x}$ = `(b - a) 1/n Sum[f[a + RandomReal[] * (b - a)], {i, 0, n}]`
μ_x = `Integrate[7 x - 8.5 x^2 + 3 x^3, {x, 0, 2}]`
x_i = `f[a + RandomReal[] * (b - a)]`
$\vec{x}_i$ = `RandomReal[]`

And it works like this..
every time we run the Mathematica code for $\bar{x}$ we'll get different results (just go above and run and re-run the code :).
Now the fluctuating quantity from the result will deviate a certain $\triangle_x$ from the true expected result and that is the Monte Carlo Error.
With other words, according to the Central Limit Theorem, $\bar{x}$ is normally distributed around μ_x with a standard deviation $\triangle_x$.
Or also that, the rate of convergence of our simulation for N is $\triangle_x$.

This can be seen also as

$$\bar{x}_1 = f\left(\vec{x}\right)$$

when $\vec{x}$ is distributed uniformly, the estimator (called primary estimator) is unbiased so that its expected value is the integral.

$$E\left[\bar{x}_1\right] = I$$

Now the function $f\left(\vec{x}\right)$ is itself a random variable with an arbitrary distribution and typically a large variance.

A more practical MC estimator is obtained by averaging a fixed number of (say N) primary estimates

$$\bar{\mu}_N = \sum f\left(\bar{x}_i\right) \Big/ N$$

Such secondary estimators are known to be unbiased and approach a Gaussian distribution as N→∞ and if the primary estimator has finite var.

So, when we refer to the estimation of a primary estimator we use lowercase $\bar{x}$

when we refer to the estimation of a secondary estimator we use uppercase $\bar{X}$, so the single outcome $X_i = \bar{x}$

$$\bar{X} \mid \bar{\mu}_N = \frac{1}{N}\sum_{i=1}^{N}\left(\bar{x} \mid X_i\right) = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{n}\sum_{i=1}^{n} x_i\right)$$

Now, let's see the derivation for the above.

Below we are subtracting the true expected value from the mean of the simulation to see how accurate is $\bar{x}$ as an estimate of μ_x

$$E\left[\bar{X} - \mu_x\right] = E\left[\bar{X}\right] - \mu_x = E\left[\frac{1}{N}\sum_{i=1}^{N} X_i\right] - \mu_x = \frac{1}{N}\sum_{i=1}^{N} E\left[X_i\right] - \mu_x$$

Since x_i occurs from random sampling, then the expectation of x_i $\left(\bar{x}\right)$ equals the true expected value μ_x , thus we have

$$E\left[\bar{X} - \mu_x\right] = \frac{1}{N} N\mu_x - \mu_x = 0$$

This result shows that on average, the error in using $\bar{x}$ to approximate μ_x is zero.

(when an estimator gives an expected error of zero, it is called an unbiased estimator)

Next, to quantify the variability in $\bar{X}$, we use the variance of $\left(\bar{X} - \mu_x\right)$,

noting that the (variance of μ_x) is zero as it is a constant (because comes from the true expected value ?)

$$Var\left[\bar{X} - \mu_x\right] = Var\left[\bar{x}\right] - Var\left[\mu_x\right] = Var\left[\bar{X}\right]$$

$$= Var\left[\frac{\sum_{i=1}^{N} X_i}{N}\right] = \frac{1}{N^2} Var\left[\sum_{i=1}^{N} X_i\right]$$

Because the Monte Carlo method draws independent, random samples,

the variance of the sum of the samples x_i, is the sum of their variances (see Variance algebra on part1).

Therefore

$$\text{Var}\left[\bar{X} - \mu_x\right] = \frac{1}{N^2}\left[\sum_{i=1}^{N} \text{Var}[X_i]\right] = \frac{1}{N^2} N\sigma_x^2 = \frac{\sigma_x^2}{N}$$

$$\mu_x \equiv E\left[\bar{X}\right] = \mu_x, \quad \sigma_x^2 \equiv E\left[\left(\bar{X} - \mu_x\right)^2\right] = \frac{\sigma_x^2}{N} = \sigma_x \frac{1}{\sqrt{N}}$$

==========

==

[LT->Theory->Sampling->MCErrorAnalysis->Basics of Direct Monte Carlo]

In a more straightforward way.

Since all the random variables X_i (ie. the generated samples) in our sample have mean μ, the mean of the sample mean is

$$E[\hat{\mu}_n] = \mu$$

In words, we can say that given a particular design (ie. $f_x()$), let
μ denote some target quantity of interest (the true expected value) and
$\hat{\mu}_n$ denote the Monte Carlo expected estimate of μ from a simulation with N replicates (where each generated replicate is X_n)

(so it could be written just as $\langle\mu\rangle=\mu$ that belongs to the notation above as $\langle\bar{A}\rangle=\langle A\rangle$ or $\langle F^N\rangle = F$)

In the language of statistics, $\hat{\mu}_n$ is said to be an unbiased (mean) estimator of μ.

We assume that the X_i have finite variance.
Since they are identically distributed, they have the same variance σ.
Since they are also independent, we have (from variance algebra, that the variance of the sum of the samples x_i, is the sum of their variances)

$$\text{Var}\left[\sum_{i=1}^{N} X_i\right] = N\sigma^2$$

(aka the variance of the sum of all the outcomes of X_i is proportional to the variance of the sample times Nsamples (ie. sum of equal variances))

and so the variance of the sample mean is

$$\text{Var}[\hat{\mu}_n] = \frac{1}{N^2}\text{Var}\left[\sum_{i=1}^{N} X_i\right] = \frac{\sigma^2}{N}$$

Thus the difference of μ_N from μ should be of order $\sigma/\sqrt{N}$
This is written also as

$$\sigma\left[\left\langle F^N\right\rangle\right] \propto \frac{1}{\sqrt{N}}$$

==========

==

In other words, we're sayin that for infinite (N) samples the mean of the simulation is equal to true expected value of the simulation,
where for Nsamples instead it is just proportional to and the difference is the MC error .. that is the same as saying that
the rate of convergence of our Monte Carlo simulation to the exact value is, $1/\sqrt{N}$.

Here we see that for standard deterministic quadratures the error is $1/N^d$ while for Monte Carlo direct method the error is $1/\sqrt{N}$.
This means that the standard approach has a faster rate of convergence than Monte Carlo.
For example it approaches 0.04 error with just 25 samples where for Monte Carlo we need around 600.

```
N[1/25]
N[(1/Sqrt[600]), 1]
```

```
0.04
```

```
0.04
```

Another thing we see is that, - to halve the error in a Monte Carlo simulation we need x4 samples (that is the square of the improvement)
(or in other words, that the standard error decreases with the square root of the sample size)

```
{N[(1/Sqrt[600 * 4]), 1], "halved the above error"}
```

```
(* if we want 10 time less the actual error(0.04) ?..
 we need a *10^2 Nsamples boost *)
N[(1/Sqrt[600 * 10^2]), 1]
```

```
{0.02, halved the above error}
```

```
0.004
```

However Monte Carlo error does not belong to the dimensions involved while the standard error does, effectively standard integration techniques suffer from the curse of dimensionality, where the convergence rate becomes exponentially worse with increased dimensions. It can be seen that only for integrals with more than 6 dims the MC error starts to be better than the standard. The smoothness of the integral can also play a part .. where it's narrow, standard quadratures have more problem sampling it than Monte Carlo.

Take care that being the usual approach to estimate σ as well as μ and then take your error magnitude to be $\sigma/\sqrt{n}$, the MC error is just a probabilistic error bound (there is no guarantee that the expected accuracy is achieved in any particular case). Same goes for "convergence" that means that you will never get an exact answer from Monte-Carlo but increasingly good approximations.

==========

==

[LT->Theory->Sampling->MCErrorAnalysis->Basics of Direct Monte Carlo]

The central limit theorem gives a more precise statement of how close $\hat{\mu}_n$ is to μ. It says :
- Let X_n be an iid sequence such that $E[\,|\,X,\ n\,|^2]<\infty$.
- Let $\mu = E[X_n]$ and
- Let σ^2 be the variance of X_n ie. $\sigma^2 = E[X_n^2] - E[X_n]^2$

then

$$\frac{1}{\sigma\sqrt{n}}\sum_{k=1}^{n}(X_k-\mu)$$

converges in distribution to a standard normal random variable.
This means that

$$\lim_{n\to\infty} P\left(a \le \frac{1}{\sigma\sqrt{n}}\sum_{k=1}^{n}(X_k-\mu) \le b\right) = \int_a^b \frac{1}{\sqrt{2\pi}} e^{-x^2/2}\,dx$$

In terms of the sample mean, the central limit theorem says that $(\hat{\mu}_n - \mu)\sqrt{n}\ \sigma$ converges in distrbution to a standard normal distribution.

However the statement of the central limit theorem involves σ^2. It is unlikely that we know σ^2 if we don't even know μ.
So we must also use our sample to estimate σ^2.

The usual estimator of the variance σ^2 is the sample variance.
It is typically denoted by s^2, but we will denote it by s_n^2 to emphasize that it depends on the sample size.
It is defined to be

$$s_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i-\hat{\mu}_n)^2$$

An application of the strong law of large numbers shows that $s_n^2 \to \sigma^2$. Since $(\hat{\mu}_n - \mu)\sqrt{n}\,/\sigma$ converges in distribution to a standard normal distribution and that $s_n^2 \to \sigma^2$ converges almost surely (a.s.) to σ^2, a standard theorem in probability implies that $(\hat{\mu}_n - \mu)\sqrt{n}\,/s_n$ converges in distribution to a standard normal. (In statistics the theorem being used here is usually called Slutsky's theorem.)

So we have the following variation on the central limit theorem :
- Let X_n be an iid sequence such that $E[\,|\,X,\ n\,|^2]<\infty$.
- Let $\mu = E[X_n]$ and
- Let σ^2 be the variance of X_n ie. $\sigma^2 = E[X_n^2] - E[X_n]^2$

then

$$\frac{(\mu_n - \mu)\ \sqrt{n}}{s_n}$$

converges in distribution to a standard normal random variable.
This means that

$$\lim_{n\to\infty} P\left(a \le \frac{(\mu_n - \mu)\ \sqrt{n}}{s_n} \le b\right) = \int_a^b \frac{1}{\sqrt{2\,\pi}}\, e^{-x^2/2}\, dx$$

The central limit theorem can be used to construct confidence intervals for our estimate $\hat{\mu}_n$ for μ.

We want to construct an interval of the form $[\hat{\mu}_n-\epsilon, \hat{\mu}_n+\epsilon]$ such that
the probability μ in this interval is $(1-\alpha)$ where the confidence level $(1-\alpha)$ is some number close to 1, ie. 95%.

So let Z be a random variable with the standard normal distribution.
Then note that μ belongs to $[\hat{\mu}_n-\epsilon,\hat{\mu}_n+\epsilon]$ if and only if $\hat{\mu}_n$ belongs to $[\mu-\epsilon,\mu+\epsilon]$.

The central limit theoreom says that

$$P\ (\mu - \epsilon \le \hat{\mu}_n \le \mu + \epsilon)\ = P\ (-\epsilon \le \hat{\mu}_n - \mu \le \epsilon)$$

$$= P\left(-\frac{\epsilon\ \sqrt{n}}{s_n} \le \frac{(\hat{\mu}_n - \mu)\ \sqrt{n}}{s_n} \le \frac{\epsilon\ \sqrt{n}}{s_n}\right) \approx P\left(-\frac{\epsilon\ \sqrt{n}}{s_n} \le Z \le \frac{\epsilon\ \sqrt{n}}{s_n}\right)$$

Let z_c be the number such that $P(-z_c \le Z \le z_c) = 1 - \alpha$.
Then we have $\frac{\epsilon\ \sqrt{n}}{s_n} = z_c$ ie. $\epsilon = z_c\, s_n/\sqrt{n}$.
Thus our confidence interval for μ is

$$\hat{\mu}_n \pm \frac{z_c\ s_n}{\sqrt{n}}$$

(btw, Z_c stays for Z critical value).

Common choices for $(1-\alpha)$ are 95% and 99%, for which $z_c \approx 1.96$ and $z_c \approx 2.58$, respectively.
(The central limit theorem is only a limit statement about what happens as the sample size n goes to infinity.
How fast the distribution in question converges to a normal distribution depends on the distribution of the original random

variable X).

It's easy to see that if we are looking for an error around 1std, being the estimated distribution a standard normal distribution (see part1 for the 68-95-99.7 rule) we'll fall into just 68% of the distribution or in other words that we have around 68% probability to be there within 1std error.

On the other side as seen above, if we really want a strict confidence interval, ie. we wanna be almost (ie. 99.7%) sure that our estimate is almost the true expectation than we'll have to take a fairly big potential error, ie. around 3std.

Eventually keep in mind that from a practical point of view, what is important is not how many samples are needed to achieve a given level of accuracy, but rather how much CPU time is need to achieve that accuracy. Suppose we have two Monte Carlo methods that compute the same thing. They have variances σ_1^2 and σ_2^2. Let τ_1 and τ_2 be the CPU time needed by the methods to produce a single sample. With a fixed amount T of CPU time we can produce Subscript[N, i] = T / τ_i samples for the two methods. So the errors of our two methods will be

$$\frac{\sigma_i}{\sqrt{N_i}} = \frac{\sigma_i \sqrt{\tau_i}}{\sqrt{T}}$$

Thus the method with the smaller $\sigma_i^2 \tau_i$ is the better method.

On a final note, take care that when

$$\lim_{n\to\infty} F_n \to \mathcal{N}(0, 1)$$

we're saying that the limiting distribution for a large set of sample means is the Standard Normal Distribution.
In other words, a sample mean can have any arbitrary distribution.. when we go for repeated simulation that arbitrary distribution tends to a standard normal distribution, because of that is called limiting distribution or also asymptotic distribution.

==========

[LT->Theory->Sampling->MCErrorAnalysis->ProbabilityAndStatisticsForEngineering]

Another way to put it is that, we know from part1 (Central Limit Theorem) that to 'standardize' a normal distribution we have

$$z = \frac{\bar{X} - \mu}{\sigma \sqrt{n}}$$

Then for example, because the area under the standard normal curve between and 1.96 is 0.95, we can write

$$P\left(-1.96 < \frac{\bar{X} - \mu}{\sigma\sqrt{n}} < 1.96\right) = 0.95$$

Let's now manipulate the inequalities inside the parentheses so that they appear in the (confidence) interval form.

Basically we're rewriting the eq in terms of μ.

Multiply both sides by $\sigma/\sqrt{n}$

$$-1.96\,\frac{\sigma}{\sqrt{n}} < \bar{X} - \mu < 1.96\,\frac{\sigma}{\sqrt{n}}$$

Subtract $\bar{X}$ from each term

$$-\bar{X} - 1.96\,\frac{\sigma}{\sqrt{n}} < -\mu < -\bar{X} + 1.96\,\frac{\sigma}{\sqrt{n}}$$

Eventually multiply through by -1 to make μ without the minus in front (that will reverse the direction of each inequality)

$$\bar{X} + 1.96\,\frac{\sigma}{\sqrt{n}} > \mu > \bar{X} - 1.96\,\frac{\sigma}{\sqrt{n}}$$

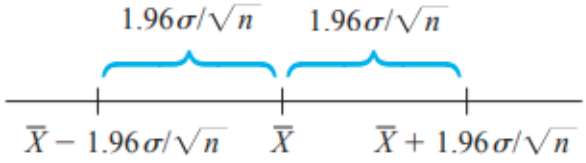
that is

$$\bar{X} - 1.96\,\frac{\sigma}{\sqrt{n}} < \mu < \bar{X} + 1.96\,\frac{\sigma}{\sqrt{n}}$$

with the interval

$$\left\{\bar{X} - 1.96\,\frac{\sigma}{\sqrt{n}},\ \bar{X} + 1.96\,\frac{\sigma}{\sqrt{n}}\right\}$$

This can be paraphrased as "there's a probability of .95 that the random interval includes or covers the true value of μ".



However take care that the confidence level is not so much a statement about any particular interval. Instead it pertains to what would happen if a very large number of intervals were to be constructed using the same CI formula... aka it applies to a long sequence of replications of an experiment rather than just a single replication.

==========

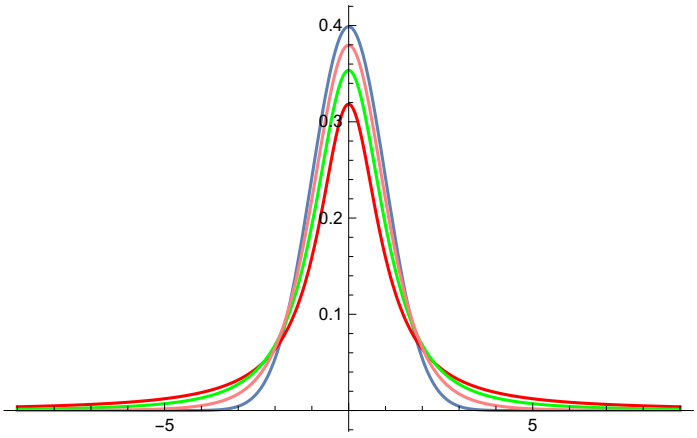
==

Another thing regard Confidence Intervals is that if the number of samples of the mean is small, typically less than about 25, then the confidence coefficients given in Table 1 cannot be used. Instead of using the normal distribution, the so-called (Student's) T distribution has to be used. This distribution has a similar shape to the normal distribution, but is a function of an additional parameter called the degrees of freedom, *v*.

Here our lower and upper confidence bounds are given by

$$T = \bar{X} \pm \frac{t_c \, S_n}{\sqrt{n-1}}$$

```
Show[
 Plot[PDF[NormalDistribution[0, 1], x], {x, -9, 9}, PlotRange → Full ],
 Plot[PDF[StudentTDistribution[1], x], {x, -9, 9},
  PlotStyle → {Red, Opacity[1]} , PlotRange → {{0, 1}, {-5, 5}}],
 Plot[PDF[StudentTDistribution[2], x], {x, -9, 9},
  PlotStyle → {Green, Opacity[1]} , PlotRange → {{0, 1}, {-5, 5}}],
 Plot[PDF[StudentTDistribution[5], x], {x, -9, 9},
  PlotStyle → {Pink, Opacity[1]} , PlotRange → {{0, 1}, {-5, 5}}]
]
```



==========

==

Eventually by considering the confidence interval to represent twice the maximum error, we can write

$$\text{error}_{max} = \frac{z_c \, S_n}{\sqrt{n}}$$

The percentage error of the mean becomes

$$Er = 100 * \frac{z_c \, S_n}{\bar{X} \sqrt{n}}$$

And so by transforming for n, we get that the total number of samples to be drawn for a certain CI is

$$n = \left(\frac{z_c \, S_n}{Er \, \bar{X}} \right)^2$$

This reads :
if the simulation is run for n samples, we are ie. 95% confident that the calculated mean will not differ by more than 5% from the true value.
To know the total n of samples (n*m) we can just use CI instead of Er$\bar{X}$.

$$n = \left(\frac{z_c \, S_n}{CI} \right)^2$$

Note that on both eqs, n will increase if :
- CI decreases (i.e. we require greater precision)
- σ increases (i.e. there's more dispersion in the population)
- α decreases (i.e. we require greater accuracy)

Example (Confidence Intervals)

You want to rent an unfurnished one-bedroom apartment in Durham, NC next year.
The mean monthly rent for a random sample of 60 apartments advertised on Craig's List is $1000.
Assume a population standard deviation of $200. Construct a 95% confidence interval.

```
Clear[x, σ, n, α, Z];
x = 1000;
σ = 200;
n = 60;
α = 95 / 100;
Z = Abs[InverseCDF[NormalDistribution[0, 1], (1 - α) / 2]];
```

$$\{\text{"Confidence Interval->"}, x - Z \frac{\sigma}{\sqrt{n}}, x + Z \frac{\sigma}{\sqrt{n}}\} \text{ // N}$$

```
{Confidence Interval->, 949.394, 1050.61}
```

Estimating the Monte Carlo Error

Let's say we have this function to integrate over $(0,\pi)$

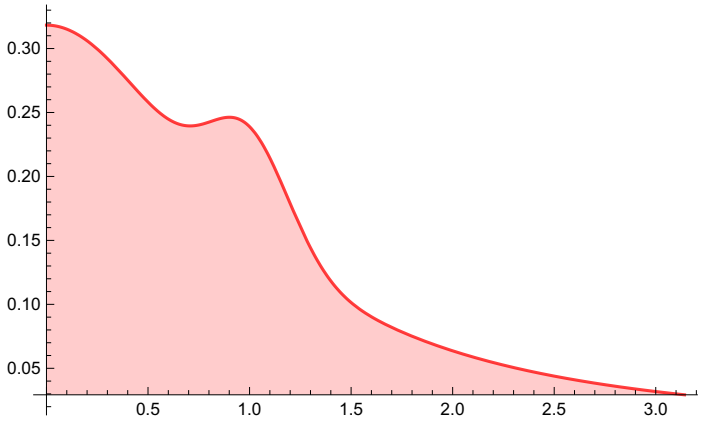
```
Clear[f, x, a, b, w, n, tValue, eValue];
```

$$f[x\_] = \frac{2}{2\left(1+x^2\right)\pi} + \frac{2\, e^{-\frac{25}{2}(-1+x)^2}}{10\sqrt{2\pi}};$$

```
a = 0;
b = π;
w = b - a;
Plot[f[x], {x, a, b}, Filling → Axis,
 PlotStyle → {Red, Opacity[0.75]}, PlotRange → Full]
```

$$tValue = \int_a^b f[x]\, dx \;//\; N;$$

```
{"True value", tValue}
```




```
{True value, 0.441907}
```

Let's calculate our estimated result with Monte Carlo.

```
n = 10^4;
SeedRandom[12345];

(* 3 equivalent ways with Mathematica *)
w 1/n Sum[f[a + RandomReal[] * w] // N
```

```
Sum[f[a + RandomReal[] * w], {i, 0, n}] * (w / n)
```

```
ests = Table[f[a + RandomReal[] * w], {i, n}] * w;
(* with this one we keep the estimates to deal with its error analysis *)
Mean[ests]
```

```
0.437377
```

```
0.446679
```

```
0.440544
```

[LT->Theory->Sampling->MCErrorAnalysis->Using Simulation Studies to Evaluate Statistical Methods]

Now, let's deal with errors aka Monte Carlo Analysis that will return a serie of Monte Carlo statistics (MCS) of interest.

Constructing MCSs is simply a matter of organizing a "generate–analyze–summarize" scheme so that particular datasets and analysis procedures can be emulated and studied over a wide number of random replications.

Let's start by using Mathematica buildin StandardDeviation and Variance symbols for our list of estimates

```
Variance[ests]
StandardDeviation[ests]
```

```
0.0987918
```

```
0.314312
```

We know that an estimate of the variance from Monte Carlo samples is given by the average of each sample's error squared:

(where $\bar{x} = \mu = \mu_X$ = true mean result and $\hat{X}_n$=estimated mean)

$$\sigma^2 \approx \frac{\sum_{n=1}^{N} \left(\hat{x}_n - \bar{x}\right)^2}{N}$$

This is the Mean Squared Error (MSE) and the Monte Carlo Standard Error based on that is

$$MCSE = \sqrt{\frac{\sum_{i=1}^{N} \left(\left(\hat{x}_n - \bar{x} \right)^2 - MSE \right)^2}{N \, (N-1)}}$$

```
Clear[sqr, MSE];
sqr = Sum[((ests[[i]] - tValue)^2, {i, n}];    (* numerator above *)
MSE = √(sqr / n)
```

$$MCSE = \sqrt{\frac{\text{Sum}\left[\left((\text{ests}[[i]] - \text{tValue})^2 - \text{MSE} \right)^2, \{i, n\} \right]}{n * (n-1)}}$$

```
0.314299
```

```
0.00227576
```

If we don't have the true outcome (most likely) we can use a fixed estimate as mean.
Because we lose a degree of freedom in using the estimate as opposed to use the true value, we use N=N-1

$$\sigma^2 \approx \frac{\sum_{n=1}^{N} \left(\hat{x}_n - \hat{x} \right)}{N-1}$$

This is called 'empirical standard error' (Empirical SE)

$$\sigma \approx \sqrt{\frac{\sum_{n=1}^{N} \left(\hat{x}_n - \hat{x} \right)}{N-1}}$$

And the Monte Carlo standard error based on that is

$$MCSE = \frac{EmpiricalSE}{\sqrt{2 \, (N-1)}}$$

```
Table[f[a + RandomReal[] * w], {i, n}] * w;
eValue = Mean[%];
√(Sum[((ests[[i]] - eValue)^2, {i, n}] * (1 / (n - 1))
```

```
MCSE = % / √(2 (n - 1))
```

```
0.314313
```

```
0.00222264
```

BTW, if we wanna compare two error returns (from the same error model), we can use this formula

$$100 \left(\left(\frac{MCSE_1}{MCSE_2} \right)^2 - 1 \right)$$

```
Quantity[100 * ((0.0021999 / 0.0021947)^2 - 1), "Percent"]
```

```
0.47443%
```

Technically we got an increase in relative precision by a 0.47% ! ;)

Anyway, we see our estimated standarddeviations is coherent with the previous true standarddeviation.

Now let's say it's not practical for us to deal with the expected mean before running our simulation, as seen above (7) we can use the following to compute our standarddeviation. This works also 'inline' ie. for 'running summations' without having to keep the estimates in memory.

[LT->Theory->Sampling->MCErrorAnalysis->MonteCarloAnalysis]

$$\sqrt{\frac{\bar{X^2} - \bar{X}^2}{N-1}} = \frac{N}{N-1}\left(\frac{\sum_{n=1}^{N} \hat{X}_n^2}{N} - \left(\frac{\sum_{n=1}^{N} \hat{X}_n}{N}\right)^2\right)$$

```
Clear[k, y];
k = Sum[(ests[[i]])^2, {i, n}] * (1 / n);
y = (Sum[(ests[[i]]), {i, n}] * (1 / n))^2;
σ̂ = Sqrt[n / (n - 1) * (k - y)]
 (* note that because it's an estimated error we use the ^hat over the σsigma *)
0.314312
```

==========

Till now we saw error analysis for a single simulation.

Let's see how to deal with multiple replications of a Monte Carlo simulation.

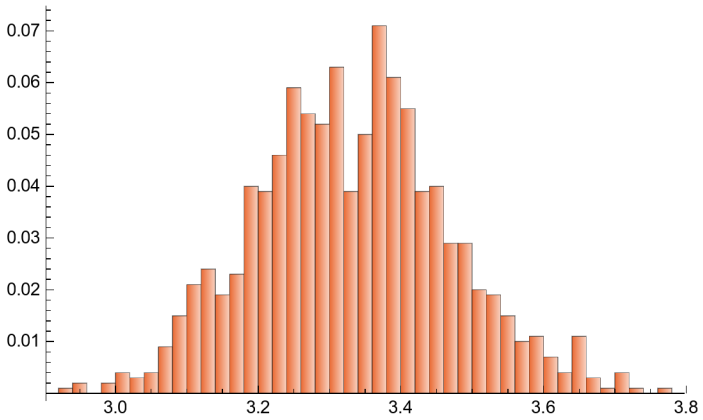
```
Clear[n, m, a, b, w, data00, data01, data02, f, x, tValue];
n = 100;                    (* n samples *)
m = 1000;                   (* n simulations *)
f[x_] = 7 x - 8.5 x^2 + 3 x^3;  (* integrand *)
a = 0; b = 2;               (* domain *)
w = b - a;                  (* 'quadrature' width *)

{"true result", tValue = ∫_a^b f[x] dx }

(* (b-a) 1/(n*m) ∑_{i=0}^{n*m} f[a+RandomReal[]*(b-a)]; *)
data00 = ParallelTable[f[a + RandomReal[] * w], {i, n * m}] * w;
Rnm = Mean[data00]
```

```
(* 1/m ∑_{i=0}^{m} ( w 1/n ∑_{i=0}^{n} f[a+RandomReal[]*w] ); *)
data01 := Table[f[a + RandomReal[] * w], {i, n}] * w;
(* same as the symbolic above but we keep the estimates *)
data02 = ParallelTable[data01, m];
(* eval m times data01 (that itself evals fx *n times) *)
data03 = {};
(* all replication means will be here *)
For[i = 1, i < m + 1, i++,
 (* dunno why but if we do Mean[data02] that should return m means.. *)
 data = data02[[i]];
 (* instead it return n means !? effectively we have m items ! *)
 mm = Sum[data[[j]], {j, n}] * (1 / n);
 (* let's do it manually ..................................... *)
 AppendTo[data03, mm]
]
RnXm = Mean[data03]
(* mean of the replication means, ie. final average *)

Histogram[data03, 32, "Probability", PlotTheme -> "WarmColor",
 ImageSize -> Medium, ChartElementFunction → "FadingRectangle"]

(* the latter, approaches a standard normal distribution
  (thanks to CLT) as we can see when m is big enough but .. *)
(* if m is not big enough we can still get confidence intervals from a student-
 t distib where the degrees of freedom is (m-1) *)
Show[
  Plot[PDF[NormalDistribution[0, 1], x], {x, -9, 9}, PlotRange → Full ],
  Plot[PDF[StudentTDistribution[16], x],
   {x, -9, 9}, PlotStyle → {Red, Opacity[0.75]} ]];
(* with enough degrees of freedom a student-
 t distribution approaches a gaussian distribution *)
(* left at 16 or at 20 the two plots almost coincide.. that's
   it ! With at least 20 repeats we have student-t → stdgauss *)
```

```
{true result, 3.33333}
```

```
3.32943
```

```
3.33195
```



So what are we doing here ? We're not here to spank frogs, right ? :))

We first run a single simulation (with n*m samples=10^7).
Then we run multiple simulations (each with n samples=10^4) * (m=10^3) replicates.

At least in Mathematica, - the latter approach is always faster than the former (use AbsoluteTiming to check this)

Let's first quantify the relative % of our simulations.

```
Interpreter["Percent"][100 * ((Rnm / RnXm) - 1)]
```

```
-0.075735%
```

Let's start our analysis with data00, ie. the one from the n*m simulation,
using Mathematica buildin stuff for that

```
Variance[data00]
StandardDeviation[data00]
```

```
1.74917
```

```
1.32256
```

Now go for the usual approach where we pretend to have the true value to compare against

```
Clear[sqr];

sqr = Sum[(data00[[i]] - tValue)^2, {i, n * m}];
{"variance", VARt = sqr / (n * m)}
{"std", STDt = √VARt }
```

$$\left\{\text{"MCSEt"}, \sqrt{\frac{\text{Sum}\left[\left(\left(\text{data00}[[i]] - \text{tValue}\right)^2 - \text{STDt}\right)^2, \{i, n * m\}\right]}{(n * m) * ((n * m) - 1)}}\right\}$$

```
{variance, 1.74916}

{std, 1.32256}

{MCSEt, 0.0118599}
```

Let's take a full sim estimated outcome and use it as true value for our error estimation (aka empirical error)

```
Clear[eValue];
eValue = data03[[RandomInteger[100]]]; (* point estimator *)
{"variance", VARe = Sum[(data00[[i]] - eValue)^2, {i, n * m}] * (1 / (n * m - 1))}
{"std", STDe = √VARe }
```

$$\left\{\text{"MCSEe"}, \text{MCSE} = \text{STDe} \Big/ \sqrt{2\,((n * m) - 1)}\right\}$$

```
{variance, 1.78152}

{std, 1.33474}

{MCSEe, 0.00298458}
```

```
{"Theoretical VS Estimated STDerror",
 Interpreter["Percent"][100 * ((STDt / STDe) - 1)]}
```

```
{Theoretical VS Estimated STDerror, -0.912285%}
```

Here instead let's pretend we don't have a clue of what's the true outcome will be, so we estimate the variance and std

```
Clear[k, y];
k = Sum[(data00[[i]])^2, {i, n}] * (1 / n);
y = (Sum[(data00[[i]]), {i, n}] * (1 / n))^2;
{"variance", VARes = n/(n - 1) * (k - y)}
{"std", STDes = √VARes}
```

```
{variance, 1.57566}
```

```
{std, 1.25525}
```

We're a bit less precise but jeez really enough ;) considering we ain't using the true nor the estimated μ !

```
{"Against theoretical STDerror", Interpreter["Percent"][100 * ((STDt / STDes) - 1)]}
{"Against empirical STDerror", Interpreter["Percent"][100 * ((STDe / STDes) - 1)]}
```

```
{Against theoretical STDerror, 5.36201%}
```

```
{Against empirical STDerror, 6.33206%}
```

==========

==

Let's now see the the m replicated simulation statistics.

```
Variance[data03]
StandardDeviation[data03]
```

```
0.0186834
```

```
0.136687
```

Take care these are the variance and std of all the estimated means we got back from replicating the simulation m times.
So they're very low because of this (they are all means and very close each) compared to the variance of the iid samples of the single simulation.

This variance is called 'between variance' that measures between group/repetitions variation.
The former variance is called 'within variance' that measures the samples variation.
With a bigger 'within variance' we're expected to have a bigger 'between variance' because the internal variation will spread outside.

Total variance is between variance + within variance.

There's a so called Ftest that does between variance / within variance

```
Ftest = Variance[data03] / Variance[data00]
```

```
0.0106813
```

Let's compute the crude bias with

$$\text{Bias} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{X}_i - \mu\right)$$

Let's note that we just miss the squared integrand, with that we would have a standard variance. Now, each replicate estimated mean is in data03 so

```
{"Bias", Sum[data03[[i]] - tValue, {i, m}] * (1 / m)}
```

```
{Bias, -0.00138169}
```

This is the RMSE. Smaller RMSE values are preferable because they indicate better recovery of the population parameters.

$$\text{RMSE} = \sqrt{\frac{1}{m}\sum_{m=1}^{n}\left(\bar{X} - \mu\right)^2}$$

```
RMSE = Sqrt[Sum[(data03[[i]] - tValue)^2, {i, m}] * (1 / m)]
```

```
0.136626
```

We see RMSE resembles the standard deviation. Squaring the RMSE yields the mean-squared error (MSE).

```
MSE = RMSE^2
```

```
0.0186666
```

Because MSE values are interpretable as variance terms, a final method that is often useful for comparing the sampling efficiency between different estimators (distinguished using the subscript i) is the relative efficiency (RE) statistic

$$\text{RE}_i = \text{MSE}_i \,/\, \text{MSE}$$

This statistic is used when we use different estimations coming from different approaches so one estimator is treated as the reference statistic and all other MSEs are evaluated in reference to it. Values greater than 1 indicate less efficiency (i.e., more variability) than the reference statistic, while values less than 1 indicate greater efficiency and therefore greater precision in recovering μ.

The model-based standard error is computed by averaging the estimated variances for each replication:

$$\text{ModelSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m} \text{Var}[X_i]}$$

All the n*samples blocks (m*replications) are in data02,
we take the variance of each of them and plug it in the eq above

```
variancedata = {};
For[i = 1, i < m + 1, i++,
 data = data02[[i]];
 mm = Variance[data]; (* compute variance for the m_th simulation *)
 AppendTo[variancedata, mm]
]
{"std", ModelSE = √(Total[variancedata] / m)}
```

```
{std, 1.32885}
```

==========

==

Now, let's see Confidence Level, Confidence Limits and Intervals.

We already know that we can be 68% confident that a random sample will lie within $\pm\sigma$ of the mean.
95.5% 2σ and 99.77% 3σ.
The percentage value is the confidence level, and the interval within which the value of x is expected to fall is the confidence limit.
This range may be expressed in the form of an upper(U) and a lower(L) bound where

$$U = \bar{x} + z_c\,\sigma$$
$$L = \bar{x} - z_c\,\sigma$$

We also know that to compute confidence intervals we need the standard deviation to relate it to the estimated output from multiple simulation because they approach a standard normal in distribution.
And we know we can't have a std while we're still calculating the expected result.
We can however calculate an estimated std like this

$$s_n^2 = \frac{1}{n-1}\sum_{i=1}^{n} \left(X_i - \hat{\mu}_n\right)^2$$

```
μ̂ = data03[[45]](*point estimate*);
μ̂ = tValue;
STDest = √(Sum[(data03[[i]] - μ̂)^2, {i, m}] * (1 / m))
```

```
0.136626
```

We know also that our confidence interval is calculated like this

$$\hat{\mu}_n \pm \frac{z_c\, S_n}{\sqrt{n}}$$

Where Zc is the Z value of the standard normal distribution calculated over the α (percentile).
So that

```
Clear[α];
α = 95.5 / 100;
(* this is the std for the 95% of a std normal dist ≃2std *)
{"STD", Z = Abs[InverseCDF[NormalDistribution[0, 1], (1 - α) / 2]] // N}
```

$$\left\{\text{"CI"},\ Ci = \frac{Z * STDest}{\sqrt{m}}\right\}$$

$$\left\{\text{"Error\%"},\ Er = 100 * \frac{Z * STDest}{Mean[data03]\ \sqrt{m}}\right\}$$

```
{STD, 2.00465}
```

```
{CI, 0.00866109}
```

```
{Error%, 0.259941}
```

Here we can state that we are 95% confident that the true mean is within $\bar{x}_i \pm$ Ci.
That of course means for our estimated mean, that we can be 95% sure that it will fall ± Ci from the true expectation.
Or also that at the 95% confidence level, any value of μ between ±Ci is plausible.

```
{Mean[data03], "-> Estimated Mean"}
{Mean[data03] - Ci, "-> Estimated Mean - Ci"}
{tValue, "-> True Value"}
{Mean[data03] + Ci, "-> Estimated Mean + Ci"}
```

```
(Mean[data03] - Ci ≤ tValue ≤ Mean[data03] + Ci)
```

```
{3.33195, -> Estimated Mean}
```

```
{3.32329, -> Estimated Mean - Ci}
```

```
{3.33333, -> True Value}
```

```
{3.34061, -> Estimated Mean + Ci}
```

```
True
```

However, what about all the other estimated means we got by replicating our simulation ? Keep reading !

Coverage is another key property of an estimator.
It is defined as the probability that a confidence interval contains the true value, and computed as:

$$\text{Coverage} = \frac{1}{m}\sum_{i=1}^{m} I\left(\hat{X}_{i,\text{low}} \le \mu \le \hat{X}_{i,\text{upp}}\right)$$

I(·) is the indicator function.

The Monte Carlo standard error is then computed as

$$\text{MCSE}(\text{Coverage}) = \sqrt{\frac{\text{Coverage} * (1 - \text{Coverage})}{m}}$$

```
Coverage =
  Sum[Boole[(data03[[i]] - Ci ≤ tValue ≤ data03[[i]] + Ci)], {i, m}] * (1 / m) // N;
{"Coverage", Interpreter["Percent"][100 * Coverage]}
```

$$\left\{\text{"MCSE(Coverage)"}, \text{MCSEcov} = \sqrt{\frac{\text{Coverage} * (1 - \text{Coverage})}{m}} \text{ // N}\right\}$$

```
{Coverage, 3.8%}
```

```
{MCSE(Coverage), 0.00604616}
```

What we have here ? The coverage over the confidence interval of our expected means.

Now, one would think that if we low down the confidence level we may up the coverage, right ? Wrong !!
If we put the confidence level to 68% we actually shrink the width of our interval.. because at 68% we have only ±1std, while at 95 we had ±2 !
Same goes if we up it to 99.7.. the width becomes then of ±3std so our coverage will be bigger .. because we have room for more error.
(That's also why a 95.5% is generally chosen most of the time so it lays in between).
Effectively if we check the Er for 95.5 and 99.7% we see that the latter is bigger than the former.
That eventually means that when α decreases we are requiring greater accuracy and so our coverage is gonna be shrinked.

```
α = 68 / 100;
(* this is the std for the α of a std normal dist ≃1std *)
{"STD", Z = Abs[InverseCDF[NormalDistribution[0, 1], (1 - α) / 2]] // N}
```

$$\left\{\text{"CI"},\ Ci = \frac{Z * STDest}{\sqrt{m}}\right\}$$

$$\left\{\text{"Error\%"},\ Er = 100 * \frac{Z * STDest}{Mean[data03]\ \sqrt{m}}\right\}$$

```
Coverage =
  Sum[Boole[(data03[[i]] - Ci ≤ tValue ≤ data03[[i]] + Ci)], {i, m}] * (1 / m) // N;
{"Coverage", Interpreter["Percent"][100 * Coverage]}
```

$$\left\{\text{"MCSE(Coverage)"},\ MCSEcov = \sqrt{\frac{Coverage * (1 - Coverage)}{m}}\ \text{// N}\right\}$$

```
{STD, 0.994458}
{CI, 0.00429655}
{Error%, 0.12895}
{Coverage, 1.8%}
{MCSE(Coverage), 0.00420428}
```

One last thing would be that we know we did shoot $100*1000 = 10^7$ to get here.
What about the other way around ... to achive a 99.7% confidence level how many samples should we shoot ?

$$TOTsamples = \left(100 * \frac{Z * STDest}{Ci}\right)^2$$

$$TOTsims = \left(100 * \frac{Z * STDest}{Er * Mean[data03]}\right)^2$$

```
1. × 10^7
1000.
```

For better estimations of the number of samples eventually check these:

[LT->Theory->Sampling->MCErrorAnalysis->Monte Carlo Simulations: Number of Iterations and Accuracy.pdf (A Priori Estimate of Number of MC Iterations, pg.7)]
[LT->Theory->Sampling->MCErrorAnalysis->Probability and Statistics for Engineering and the Sciences.pdf (A Confidence Interval for a Population Proportion, pg.280)]

[and following that needs Power of MC simulations not analized here but to be found on the latter pdf above]

Computes the number of simulations B to perform based on the accuracy of an
estimate of interest, using the following equation:

$$B = \left(\frac{(Z_{1-\alpha/2} + Z_{1-\theta})^{\sigma}}{\delta} \right)^2$$

where
δ is the specified level of accuracy of the estimate of interest (i.e.the permissible difference from the true value β),
$Z_{1-\alpha/2}$ is the $(1 - \alpha/2)$ quantile of the standard normal distribution,
$Z_{1-\theta}$ is the $(1 - \theta)$ quantile of the std normal dist with
$(1 - \theta)$ being the power to detect a specific difference from the true value as significant
σ^2 is the variance of the parameter of interest.

==========

Now, let's see what's the impact of using a general uniform sampler VS a specialized one.

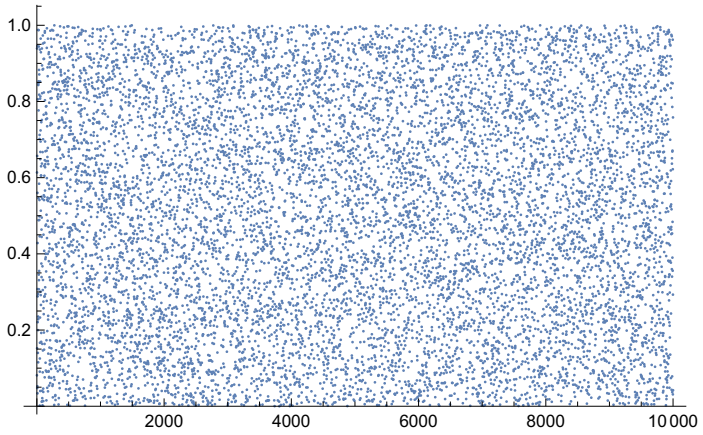
For general uniform sampler we intend white noise. For a specialized one let's take a Low Discrepancy Sequence (LDS).
White noise tends to 'clump' samples in some zones (about $\sqrt{N}$ out of N points lie in clumps), while LDS has optimal distributed numbers over the sampling interval 0-1.

An LDS is also called quasirandom sequence. That also means they are not fully random. Which in turn means we can't just apply the same error metrics we do for Monte Carlo error analysis and simply because this is not Monte Carlo but Quasi Monte Carlo. In quasi-Monte Carlo methods, the error satisfies deterministic bounds, as given by the Koksma-Hlawka inequality. [LT->Theory->1998 - Monte Carlo and Quasi-Monte Carlo Methods]

However here we'll deal with low discrepancy hashed sequences to see if they are still better than crude white noise.
When we hash a sequence we're turning it into a seeded random sequence.

```
AbsoluteTiming[dww = Table[RandomReal[], {i, 10 000}];]
ListPlot[dww]
```

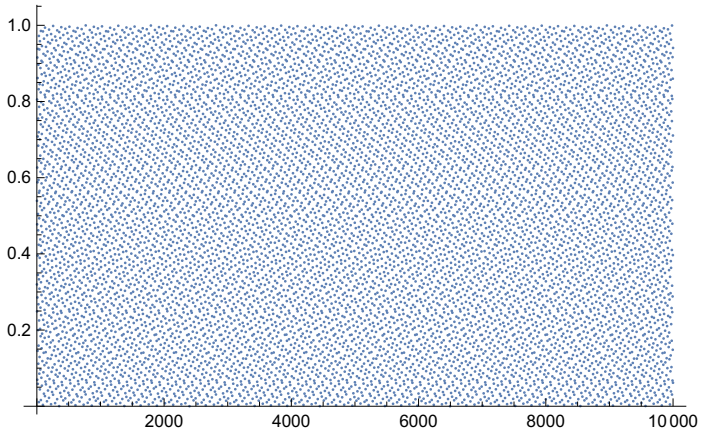
```
{0.000469, Null}
```




```
(* this works if github directory structure is the same as the local one,
if not adjust accordling *)
(* eventually take care that here we'll use 'proof of concept' samplers,
that are really slow *)
(* scroll down where we sample fncs and plot
 errors to see how to use compiled (same) samplers *)

(*NotebookEvaluate[
   FileNameJoin[{NotebookDirectory[],"Samplers/ProofsOfConcept","Halton.nb"}]]
  LDSHsampler[w_]:=haltonSampler1D[w];*)

(*NotebookEvaluate[FileNameJoin[
     {NotebookDirectory[],"Samplers/ProofsOfConcept","R2Jittered.nb"}]]
   LDSHsampler[w_]:=r2Sample1D[w]; *)

(*NotebookEvaluate[FileNameJoin[{NotebookDirectory[],
    "Samplers/ProofsOfConcept","Correlated-Multi-Jitter.nb"}]]
 LDSHsampler[w_]:=cmjSampler1D[w]; *)

NotebookEvaluate[FileNameJoin[
  {NotebookDirectory[], "Samplers/ProofsOfConcept", "OwenScrambledSobol.nb"}]]
LDSHsampler[w_] := owenScrambledSobol1D[w];
```

```
AbsoluteTiming[dos = ParallelTable[LDSHsampler[i], {i, 10 000}];]
ListPlot[dos]
```

```
{2.21179, Null}
```



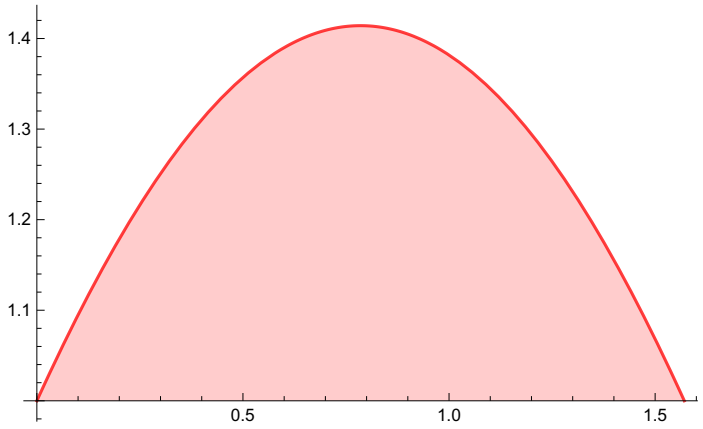
See how the rectangle is more evenly filled with OwenScrambled(OS) than with WhiteNoise(WN) ? Yeah, OS is really slower compared to WN.

So, let's see how it goes by sampling a function with our two different samplers.

```
Clear[f, x, a, b, w, n, tValue, eValue];
f[x_] = Cos[x] + Sin[x];
a = 0;
b = π / 2;
w = b - a;
n = 10 000;
tValue = ∫_a^b f[x] dx // N;
{"True value", tValue}
```

```
Plot[f[x], {x, a, b}, Filling → Axis,
 PlotStyle → {Red, Opacity[0.75]}, PlotRange → Full]
```

```
{True value, 2.}
```




```
estsWN = ParallelTable[f[a + RandomReal[] * w], {i, n}] * w;
meanWN = Mean[estsWN]
Variance[estsWN]
StandardDeviation[estsWN]
```

```
2.0034
```

```
0.0375404
```

```
0.193753
```

```
estsLDS = ParallelTable[f[a + LDSHsampler[i] * w], {i, n}] * w;
meanLDS = Mean[estsLDS]
Variance[estsLDS]
StandardDeviation[estsLDS]
```

```
2.00001
```

```
0.0381945
```

```
0.195434
```

To measure the impact of using a different sampler we of course will employ the error metrics we used above.

So let's start by getting the standard deviation and its assoaciated Monte Carlo standard error and bias statistics.

```
Clear[sqr, MSE];
sqr = Sum[(estsWN[[i]] - tValue)^2, {i, n}];
{"STD WN", MSEwn = Sqrt[sqr / n]}(* aka standard deviation *)
```

$$\left\{\text{"MCSE WN"},\ \text{MCSEwn} = \sqrt{\frac{\text{Sum}\left[\left(\left(\text{estsWN}[[i]] - \text{tValue}\right)^2 - \text{MSEwn}\right)^2, \{i, n\}\right]}{n * (n - 1)}}\right\}$$

```
{"Bias WN", Sum[estsWN[[i]] - tValue, {i, m}] * (1 / m)}
```

```
{STD WN, 0.193774}
```

```
{MCSE WN, 0.00160997}
```

```
{Bias WN, 0.00306064}
```

```
Clear[sqr, MSE];
sqr = Sum[(estsLDS[[i]] - tValue)^2, {i, n}];
{"STD LDShash", MSElds = Sqrt[sqr / n]}
```

$$\left\{\text{"MCSE LDShash"},\ \text{MCSE} = \sqrt{\frac{\text{Sum}\left[\left(\left(\text{estsLDS}[[i]] - \text{tValue}\right)^2 - \text{MSElds}\right)^2, \{i, n\}\right]}{n * (n - 1)}}\right\}$$

```
{"Bias LDShash", Sum[estsLDS[[i]] - tValue, {i, m}] * (1 / m)}
```

```
{STD LDShash, 0.195424}
```

```
{MCSE LDShash, 0.00162208}
```

```
{Bias LDShash, 0.000278994}
```

First let's just note that with the STD and MCSE because of the squaring and then the square-rooting, - our errors look not so much different.
Bias instead, being an order of magnitude less for the OS sampler, gives us a somehow better statistic.

We can however also use a derivation (dependent t-test for paired samples) of the so called T-test that gives us a nice value that effectively seems characterezing a bit more our error.

$$\text{DTestwn} = \frac{\text{Abs}[\text{meanWN} - \text{tValue}]}{\text{MSEwn}}$$

$$\text{DTestlds} = \frac{\text{Abs}[\text{meanLDS} - \text{tValue}]}{\text{MSElds}}$$

```
0.0175651
```

```
0.0000435081
```

We can also use another derivation of the T-test called Welch's t-test to compare directly our two estimates

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}}$$

$$\text{Twelch} = \frac{\text{meanWN} - \text{meanLDS}}{\sqrt{\frac{\text{MSEwn}^2}{n} + \frac{\text{MSElds}^2}{n}}}$$

```
1.23367
```

Eventually the T-test is given by

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2+s_2^2}{2}} * \sqrt{\frac{2}{n}}}$$

$$\frac{\text{meanWN} - \text{meanLDS}}{\sqrt{\frac{\text{MSEwn}^2+\text{MSElds}^2}{2}} * \sqrt{\frac{2}{n}}}$$

```
1.23367
```

Result is the same of the Welch's t-test because n is the same for both estimations there while it could be different as opposed to the T-test.

==========

==

Now this error estimates do tell us something when compared against different estimators of the same parameter (here mean).
What they don't tell us is the variability of our result compared to successive runs of the same simulation.

```
dataWN01 = estsWN;
dataWN02 = ParallelTable[f[a + RandomReal[] * w], {i, n}] * w;
dataWN03 = ParallelTable[f[a + RandomReal[] * w], {i, n}] * w;
dataWN04 = ParallelTable[f[a + RandomReal[] * w], {i, n}] * w;
dataWN05 = ParallelTable[f[a + RandomReal[] * w], {i, n}] * w;

meanWN
meanWN2 = Mean[dataWN02]
meanWN3 = Mean[dataWN03]
meanWN4 = Mean[dataWN04]
meanWN5 = Mean[dataWN05]
meansWN = {meanWN, meanWN2, meanWN3, meanWN4, meanWN5};
```

```
2.0034
```

```
2.00061
```

```
1.99885
```

```
2.00122
```

```
1.99739
```

```
(* the i+n...+n thing is needed here because OSS gives back the same outputs for
  the same inputs so we need to extend the sequence to get new random numbers *)
Clear[k];
ProgressIndicator[Dynamic[k]](* progress indicator *)
dataLDS01 = estsLDS;
dataLDS02 = ParallelTable[With[{i = i + n}, f[a + LDSHsampler[i] * w]], {i, n}] * w;
k = 0.25;
dataLDS03 = ParallelTable[With[{i = i + n + n}, f[a + LDSHsampler[i] * w]], {i, n}] * w;
k = .5;
dataLDS04 =
  ParallelTable[With[{i = i + n + n + n}, f[a + LDSHsampler[i] * w]], {i, n}] * w;
k = .75;
dataLDS05 =
  ParallelTable[With[{i = i + n + n + n + n}, f[a + LDSHsampler[i] * w]], {i, n}] * w;
k = 1;

SetPrecision[meanLDS, 18]
SetPrecision[meanLDS2 = Mean[dataLDS01], 18]
SetPrecision[meanLDS3 = Mean[dataLDS03], 18]
SetPrecision[meanLDS4 = Mean[dataLDS04], 18]
SetPrecision[meanLDS5 = Mean[dataLDS05], 18]
meansLDS = {meanLDS, meanLDS2, meanLDS3, meanLDS4, meanLDS5};
```

```
2.00000850254700913

2.00000850254700957

1.99999596668954038

2.00000252054640093

2.00000333994209489
```

Here we are not interested in the simulation mean but in the simulation variance which will tell us something about the variability of our estimators that in turn will tell us something about the impact of our random numbers generators as that's the only thing different.

```
{"Var whitenoise", ScientificForm[Variance[meansWN]]}
{"Var LDSH", ScientificForm[Variance[meansLDS]]}

{"STD whitenoise", ScientificForm[StandardDeviation[meansWN]]}
{"STD LDSH", ScientificForm[StandardDeviation[meansLDS]]}
```

$\{\text{Var whitenoise}, 5.29315 \times 10^{-6}\}$

$\{\text{Var LDSH}, 2.68579 \times 10^{-11}\}$

$\{\text{STD whitenoise}, 2.30068 \times 10^{-3}\}$

$\{\text{STD LDSH}, 5.18246 \times 10^{-6}\}$

Here we go instead for the ModelSE.

```
variancedata = {Variance[dataWN01], Variance[dataWN02],
   Variance[dataWN04], Variance[dataWN05], Variance[dataWN05]};
{"STD WN ModelSE", ModelSE = √(Total[variancedata] / m)}

variancedata = {Variance[dataLDS01], Variance[dataLDS02],
   Variance[dataLDS03], Variance[dataLDS04], Variance[dataLDS05]};
{"STD LDS ModelSE", ModelSE = √(Total[variancedata] / m)}
```

$\{\text{STD WN ModelSE}, 0.0137609\}$

$\{\text{STD LDS ModelSE}, 0.0138203\}$

Now, because we have done just 5 repetitions of our simulation we saw that we can't use the z_c from the Normal distribution but we need to use the t_c from the Student's t distribution.

```
α = 95 / 100;
m = 5;
t = Abs[InverseCDF[StudentTDistribution[m - 1], (1 - α) / 2]] // N;
STD = √(Total[variancedata] / m);
```

$$\text{Ci} = \frac{\text{t} * \text{STD}}{\sqrt{\text{m}}}$$

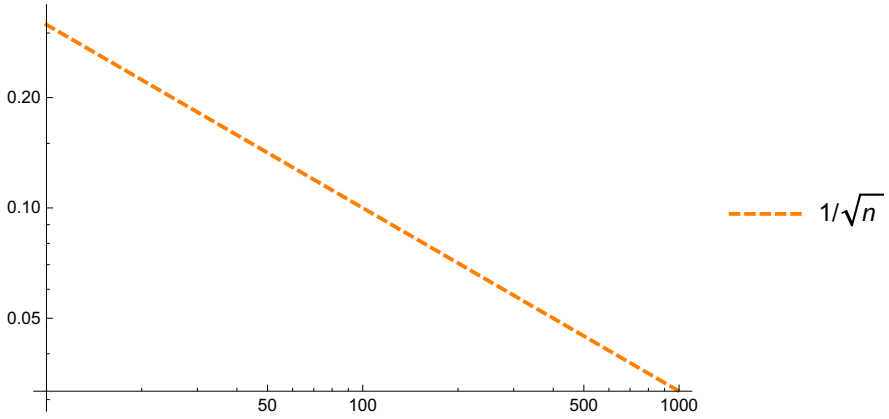
0.242681

Of course with just 5 repetitions our CI is rather big.

==========

==

```
(* ploting with loglog will straight our n^-0.5 graph *)
LogLogPlot[z^-0.5, {z, 10, 1000}, PlotRange → Full,
 PlotLegends → {"1/√n "}, PlotStyle → {Orange, Dashed, Thick}]
```




```
(* check it for iteration in LDsequences *)
(*
Clear[k,wtf,kn,km];
kn=10;
km=20;
wtf[q_]:=Module[{},
  dd=Table[{j=i+(kn*q),k=((kn*km))+(i+(kn*q))},{i,kn}];
  dd
 ];
drt=Table[wtf[i],{i,km}]
*)

(* the default behaviour of the random number generator can be changed *)
SeedRandom[12,
  Method → {"Congruential", "Multiplier" → 11, "Increment" → 0, "Modulus" → 63}];
RandomReal[1]
RandomReal[1]

SeedRandom[Method → "Legacy"]
RandomReal[1]
RandomReal[1]
```

```
0.0952381
```

```
0.047619
```

```
0.214347
```

```
0.539981
```

```
sample[n_, m_] := Module[{kn = n, km = m},
  Clear[samplingf];
```

```
  samplingf := Table[f[RandomReal[], RandomReal[]], {i, kn}];
  datasamples = ParallelTable[samplingf, km];
  Block[{},
   datatrials = {};
   For[i = 1, i < km + 1, ++i,
    datax = datasamples[[i]];
    AppendTo[datatrials, Sum[datax[[j]], {j, kn}] * (1 / kn)];
   ]
  ];
  MSE = √(Sum[(datatrials[[i]] - tval)^2, {i, km}] / km);
  {"Estimated Value", Mean[datatrials] // N, "MSError", MSE}
 ]
sampleError[n_, m_] := Module[{kn = n, km = m},
  Clear[samplingf];
  samplingf := Table[f[RandomReal[], RandomReal[]], {i, kn}];
  datasamples = ParallelTable[samplingf, km];
  Block[{},
   datatrials = {};
   For[i = 1, i < km + 1, ++i,
    datax = datasamples[[i]];
    AppendTo[datatrials, Sum[datax[[j]], {j, kn}] * (1 / kn)];
   ]
  ];
  MSE = √(Sum[(datatrials[[i]] - tval)^2, {i, km}] / km);
  (*{Mean[datatrials]//N,MSE}*)
  {kn, MSE}(*error as a fnc of the number of samples*)
 ]
sampleErrorCustom[n_, m_] := Module[{kn = n, km = m},
  samplingwtf[kq_] := Table[With[{j = i + (kn * kq), k = ((kn * km)) + (i + (kn * kq))},
     f[RandomRealCustom[j], RandomRealCustom[k]]], {i, kn}];
  datasamples = ParallelTable[samplingwtf[k], {k, km}];
  Block[{},
   datatrials = {};
   For[r = 1, r < km + 1, ++r,
    datax = datasamples[[r]];
    AppendTo[datatrials, Sum[datax[[j]], {j, kn}] * (1 / kn)];
   ]
  ];
  MSE = √(Sum[(datatrials[[p]] - tval)^2, {p, km}] / km);
  (*{Mean[datatrials]//N,MSE}*)
  {kn, MSE}
 ]
plotError[n_, m_, s_: 10] := Block[{},
```

```
  trig = True;
  dataerr = Table[(k = i; sampleError[i, m]), {i, n / s, n, n / s}];
  trig = False;
  dt = Transpose[dataerr];
  Show[
   ListLogLogPlot[dataerr, AxesLabel → {"Nsamples", "MSE"},
    LabelStyle → Directive[Orange, Bold], ImageSize → Large,
    PlotRange → {{n / s, n}, {Min[dt[[2]]] / 2, Max[dt[[2]]] * 2}}],
   LogLogPlot[z^-0.5, {z, n / s, n}, PlotRange → Automatic,
    PlotLegends → {"√n "}, PlotStyle → {Orange, Dashed, Thick}]]
  (*we plot in LogLog so we're able to better see the y-error axis*)
 ]
plotErrorX[n_, m_, s_: 10] := Block[{},
  trig = True;
  dataerrh = Table[(k = i;
     sampleErrorCustom[i, m]), {i, n / s, n, n / s}];
  dataerr = Table[(k = i; sampleError[i, m]), {i, n / s, n, n / s}];
  trig = False;
  dt = Transpose[dataerr];
  Show[
   ListLogLogPlot[dataerr, AxesLabel → {"Nsamples", "MSE"},
    LabelStyle → Directive[Orange, Bold], ImageSize → Full, PlotRange →
     {{n / s, n}, {Min[dt[[2]]] / 2, Max[dt[[2]]] * 2}}, PlotLegends → {"WhiteNoise"}],
   ListLogLogPlot[dataerrh, AxesLabel → {"Nsamples", "MSE"},
    LabelStyle → Directive[Orange, Bold], ImageSize → Full,
    PlotRange → {{n / s, n}, {Min[dt[[2]]] / 2, Max[dt[[2]]] * 2}},
    PlotStyle → {Green, Dashed}, PlotLegends → {"Halton"}],
   LogLogPlot[z^-0.5, {z, n / s, n}, PlotRange → Automatic, PlotLegends → {"√n "},
    LabelStyle → Directive[Orange, Bold], PlotStyle → {Cyan, Dashed, Thick}]]
 ]

(* Take care of this, it's used in sampleErrorCustom *)
Needs["CCompilerDriver`"]

randomRealHaltonLib = CreateLibrary[
  {"/home/max/git/Miscellaneous/MathematicaNotebooks/Samplers/C++/LibraryLink/
     randomRealHalton.cpp"}, "randomRealHalton",
  "Debug" → False, "CompileOptions" → "-O3"]
RandomRealHalton = LibraryFunctionLoad[randomRealHaltonLib,
  "RandomRealHalton", {Integer}, Real]
RandomRealHalton[123 688]

(*randomRealCMJLib=CreateLibrary[
  {"/home/max/git/Miscellaneous/MathematicaNotebooks/Samplers/C++/LibraryLink/
```

```
    randomRealCMJ.cpp"},"randomRealCMJ",
  "Debug"→False, "CompileOptions"→"-O3"];
RandomRealCMJ=LibraryFunctionLoad[randomRealCMJLib,
  "RandomRealCMJ",{Integer},Real];*)

RandomRealCustom[k_] := RandomRealHalton[k]; (* plug-in *)
```

```
/home/max/.Mathematica/SystemFiles/LibraryResources/Linux-x86-64/randomRealHalton.
  so
```

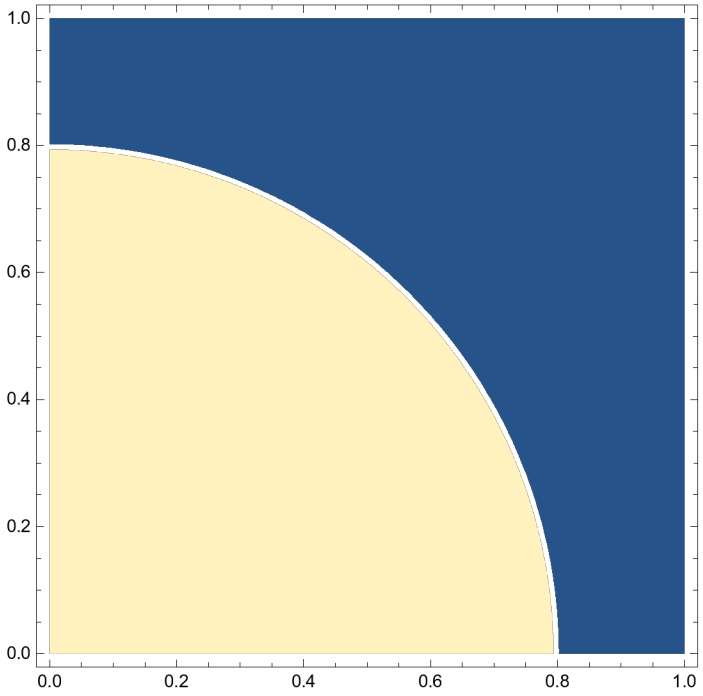
```
LibraryFunction[ Function name: RandomRealHalton
                 Argument count: 1 ]
```

```
0.30087
```

```
Clear[f, tval];
f[x_, y_] := Boole[(x^2 + y^2) < (2/π)] (* disk *)
DensityPlot[f[x, y], {x, 0, 1}, {y, 0, 1}]
{"True Value", tval = NIntegrate[f[x, y], {x, 0, 1}, {y, 0, 1}]}
```

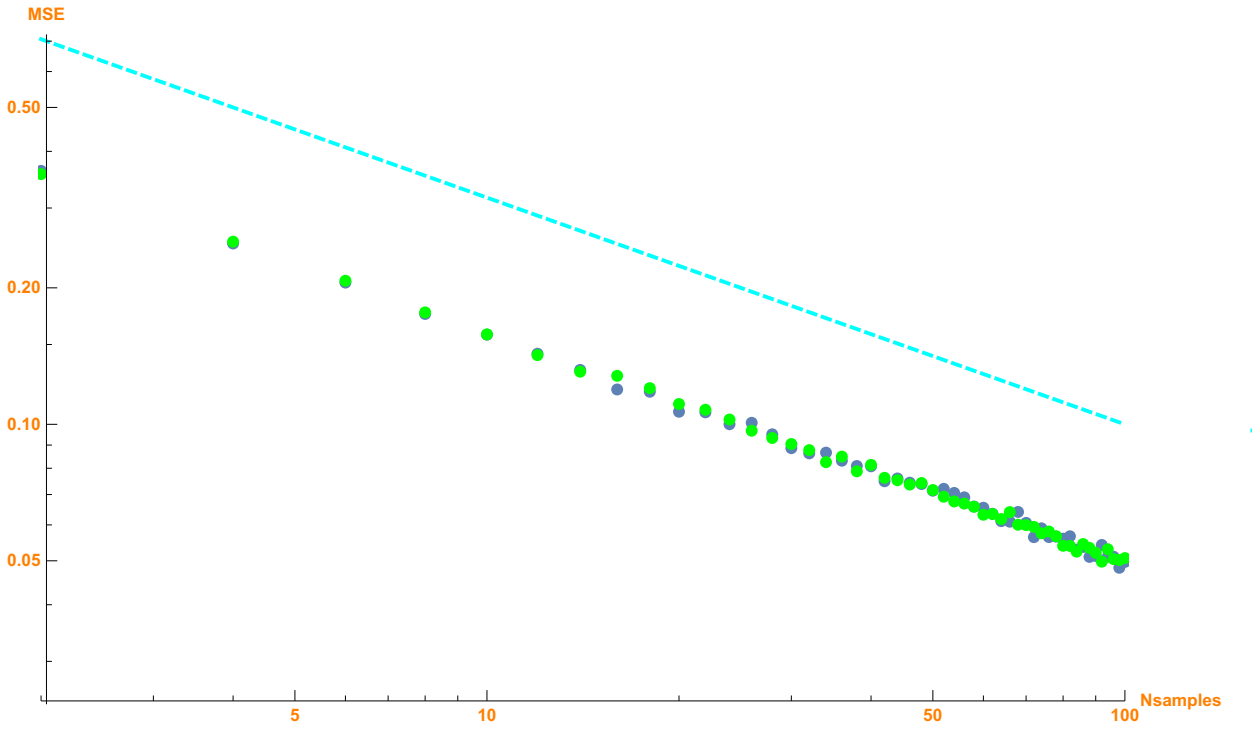
```
n = 100;
m = 1000;
s = 50;
```

```
sample[n, m]
Dynamic[If[trig, ProgressIndicator[Dynamic[k], {n / s, n}], ""]]
plotErrorX[n, m, s]
```

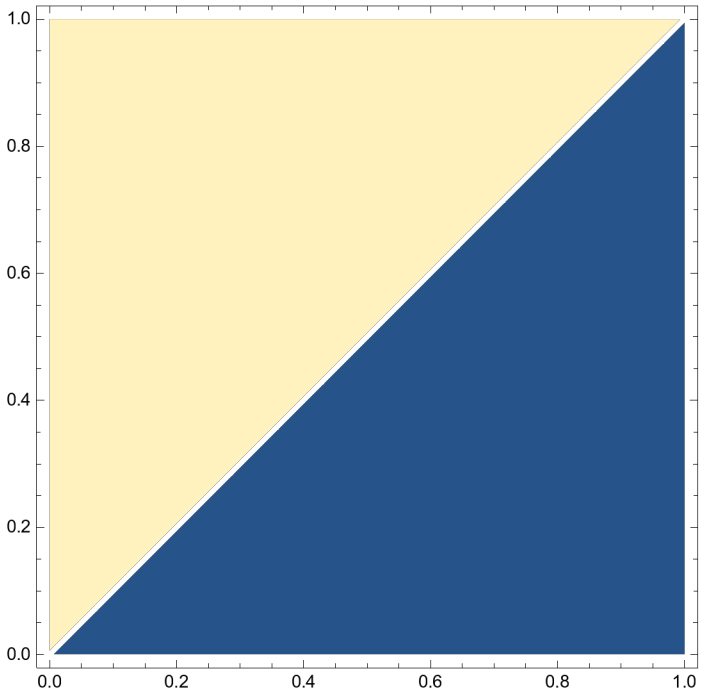


```
{True Value, 0.5}
```

```
{Estimated Value, 0.49602, MSError, 0.0504282}
```

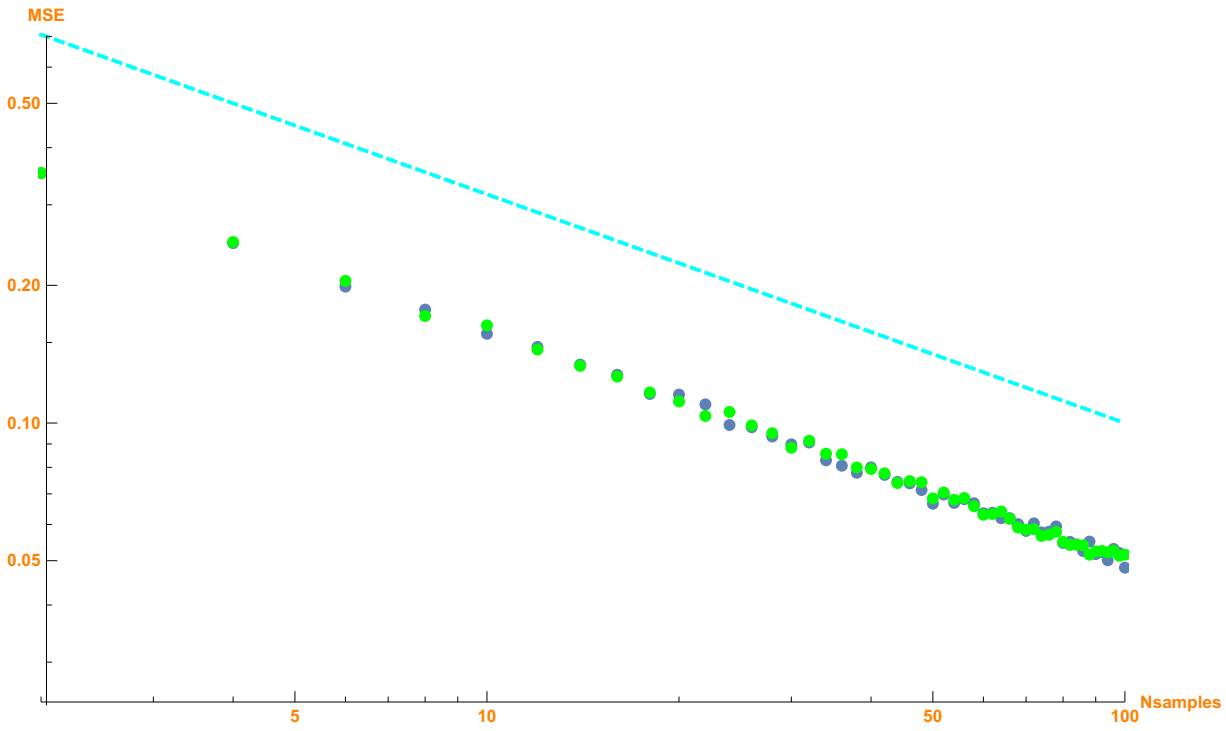



```
Clear[f, tval];
f[x_, y_] := Boole[x < y] (* triangle *)
DensityPlot[f[x, y], {x, 0, 1}, {y, 0, 1}]
{"True Value", tval = NIntegrate[f[x, y], {x, 0, 1}, {y, 0, 1}]}

n = 100;
m = 1000;
s = 50;

sample[n, m]
Dynamic[If[trig, ProgressIndicator[Dynamic[k], {n / s, n}], ""]]
plotErrorX[n, m, s]
```

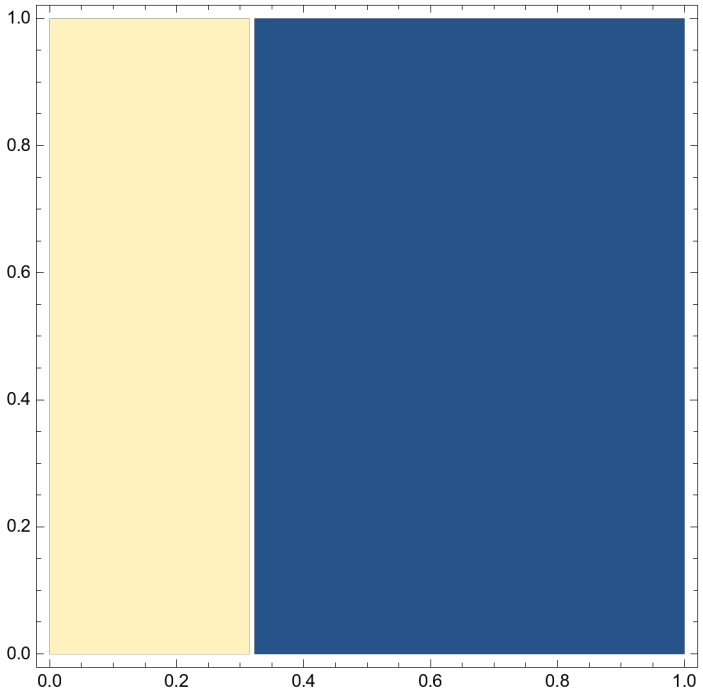


{True Value, 0.5}

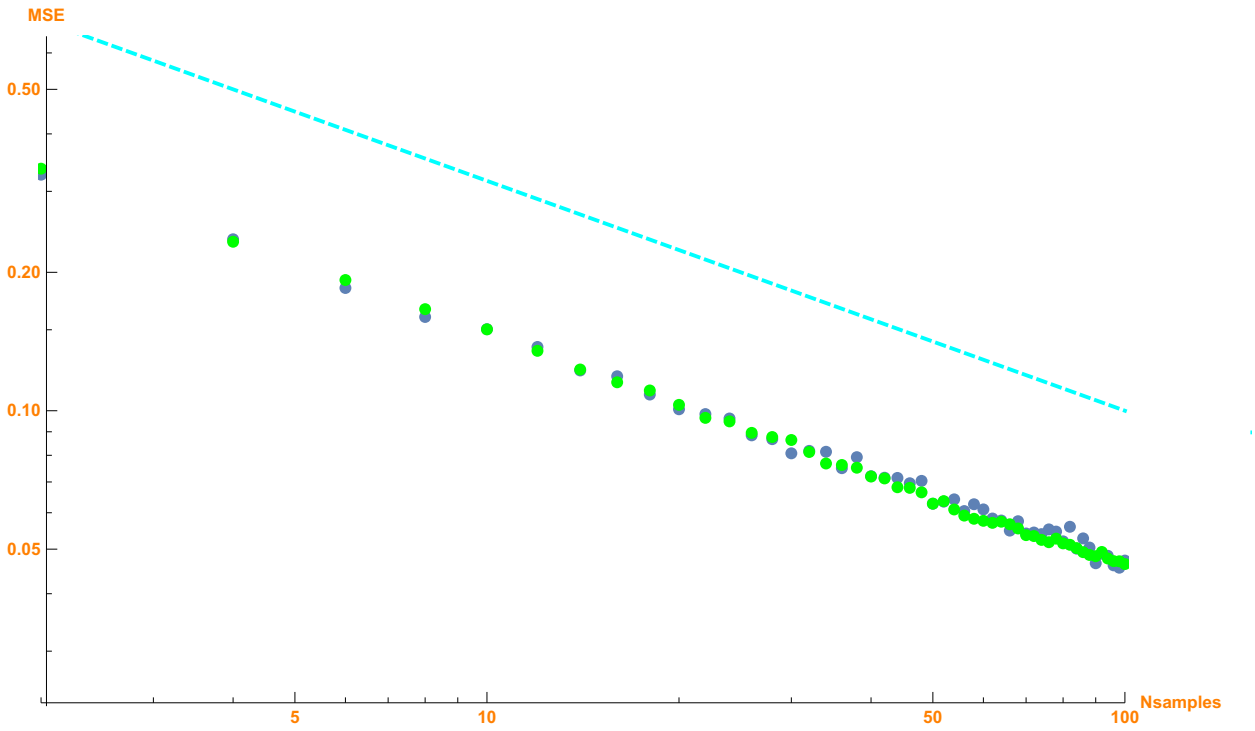
{Estimated Value, 0.5021, MSError, 0.0506814}



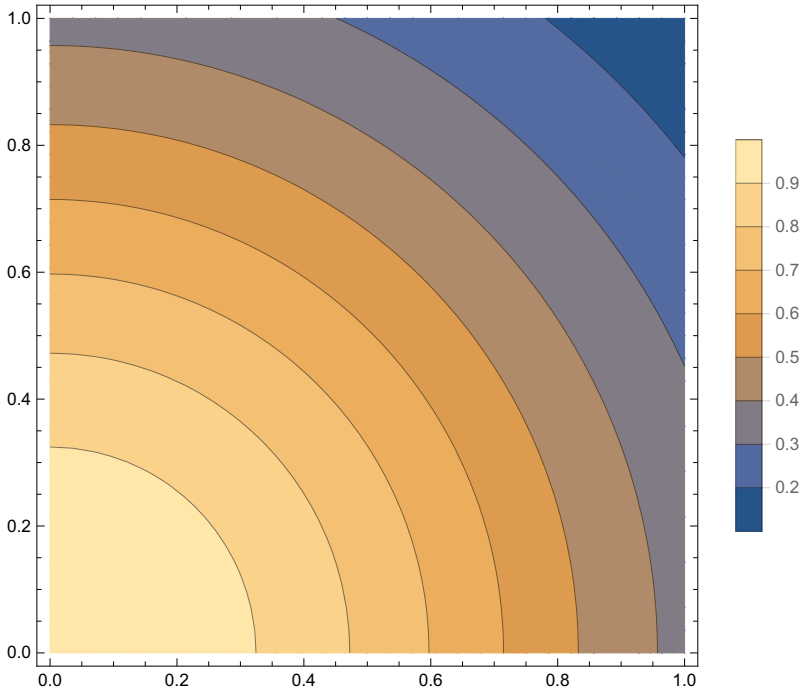
```
Clear[f, tval];
f[x_, y_] := Boole[x < 1 / π] (* step *)
DensityPlot[f[x, y], {x, 0, 1}, {y, 0, 1}]
{"True Value", tval = NIntegrate[f[x, y], {x, 0, 1}, {y, 0, 1}]}

n = 100;
m = 1000;
s = 50;

sample[n, m]
Dynamic[If[trig, ProgressIndicator[Dynamic[k], {n / s, n}], ""]]
plotErrorX[n, m, s]
```




```
{True Value, 0.31831}

{Estimated Value, 0.31751, MSError, 0.0458251}
```

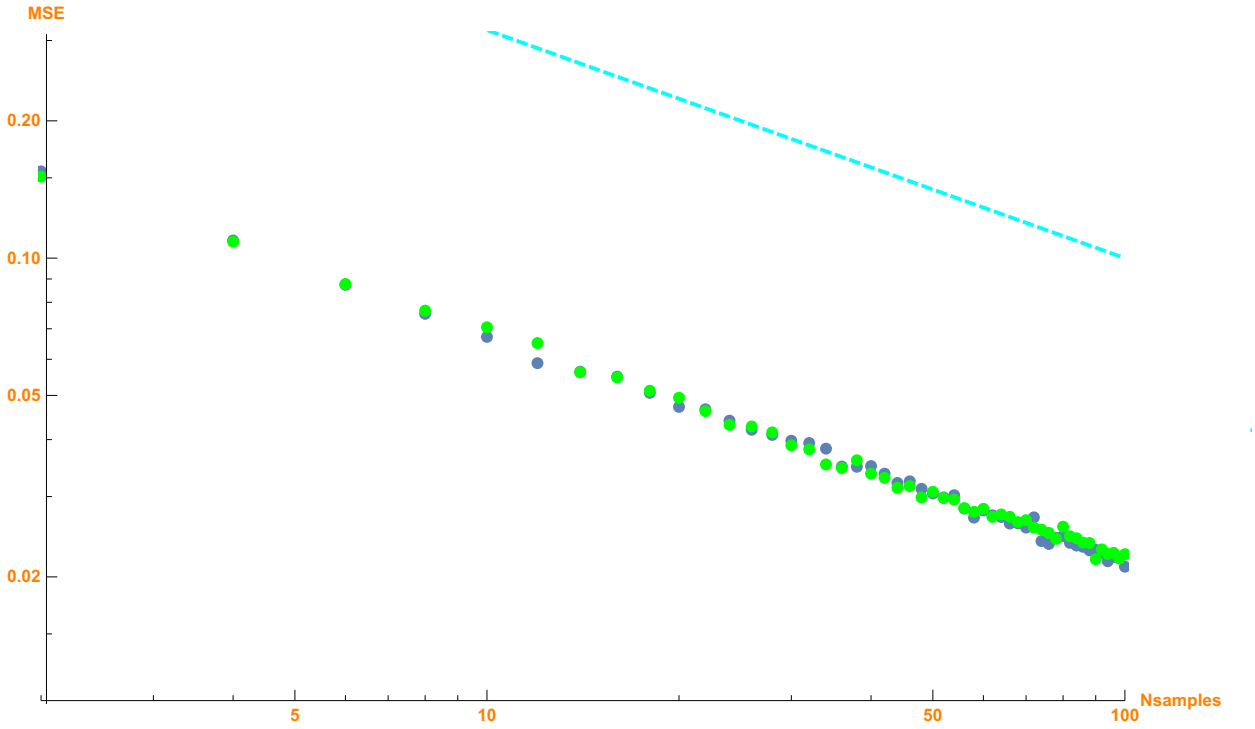



```
Clear[f, tval];
f[x_, y_] := e^(-x^2-y^2)  (* gaussian *)
ContourPlot[f[x, y], {x, 0, 1}, {y, 0, 1}, PlotLegends → Automatic]
{"True Value", tval = NIntegrate[f[x, y], {x, 0, 1}, {y, 0, 1}]}

n = 100;
m = 1000;
s = 50;

sample[n, m]
Dynamic[If[trig, ProgressIndicator[Dynamic[k], {n / s, n}], ""]]
plotErrorX[n, m, s]
```

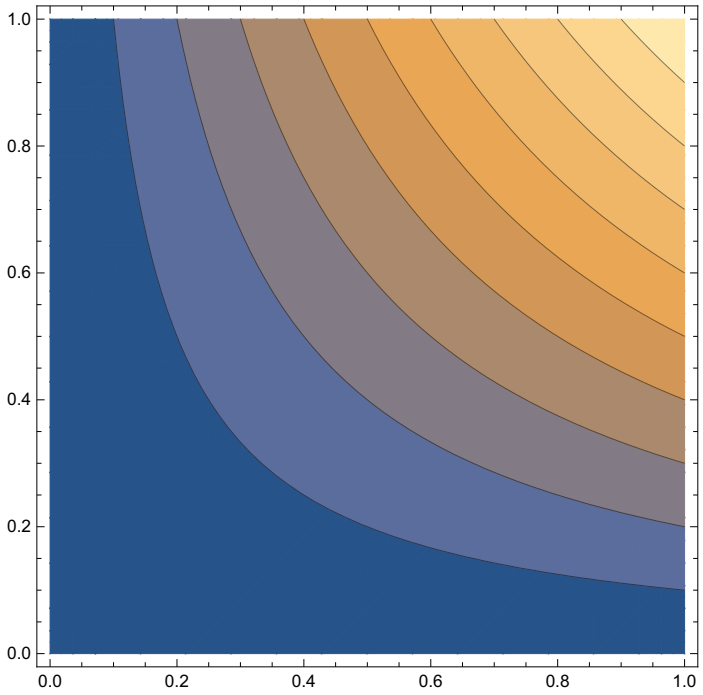


{True Value, 0.557746}

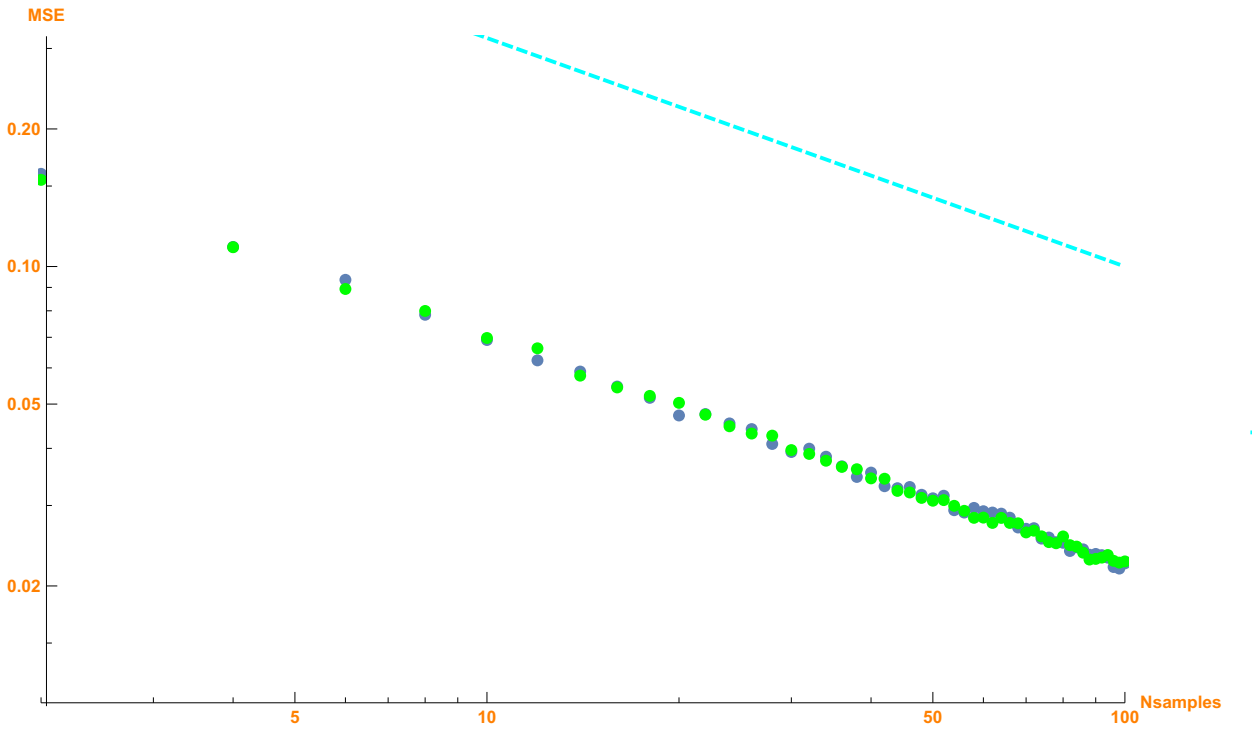
{Estimated Value, 0.558577, MSError, 0.0222912}



```
Clear[f, tval];
f[x_, y_] := x * y    (* bilinear *)
ContourPlot[f[x, y], {x, 0, 1}, {y, 0, 1}]
{"True Value", tval = NIntegrate[f[x, y], {x, 0, 1}, {y, 0, 1}]}

n = 100;
m = 1000;
s = 50;

sample[n, m]
Dynamic[If[trig, ProgressIndicator[Dynamic[k], {n / s, n}], ""]]
plotErrorX[n, m, s]
```




```
{True Value, 0.25}

{Estimated Value, 0.249787, MSError, 0.0229584}
```



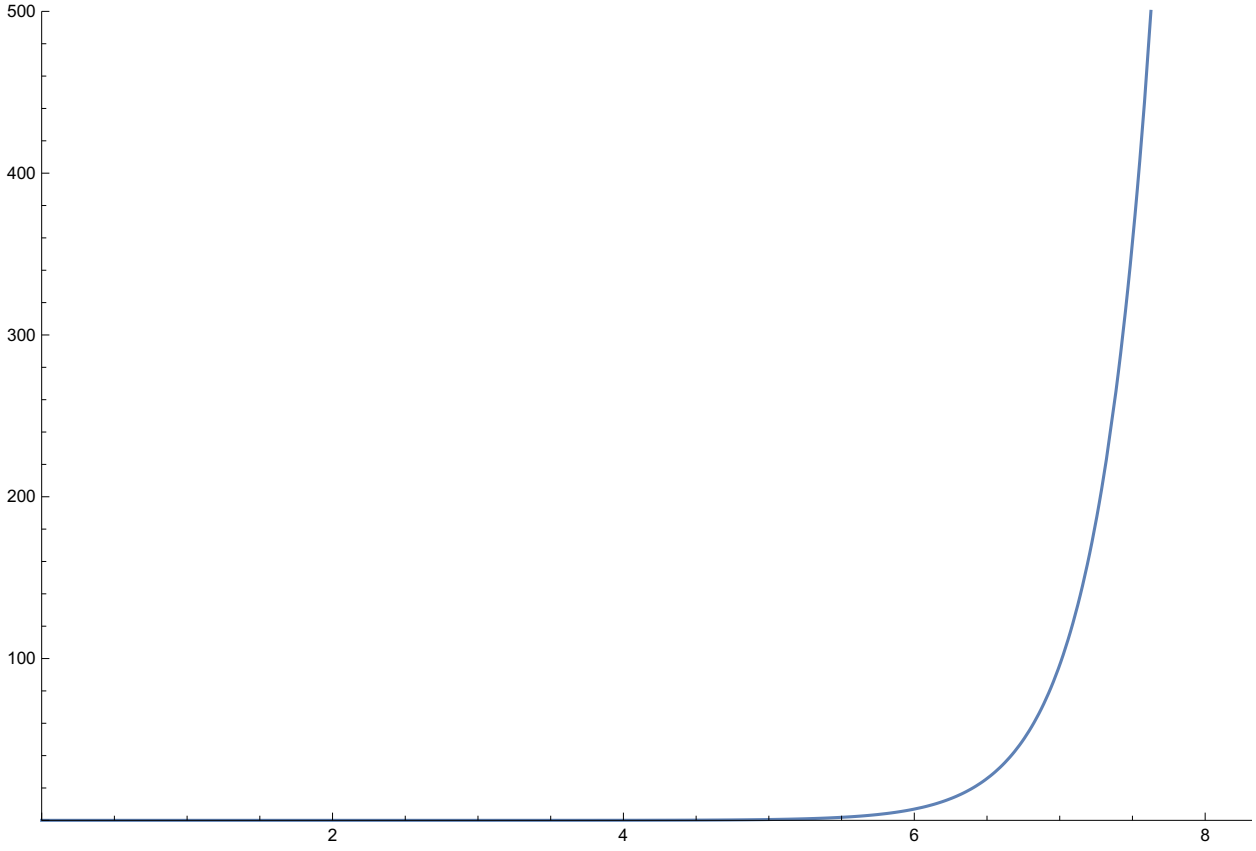
==========

==

Theoretically for classical LDS their discrepancy upper bound is the QMC error bound and so naively we can just compare it with the MC errror bound: logN^dim /N < 1/N^0.5 . For 1mil samples already at dim=3 the deterministic bound begins to fall below the probabilistic one.

```
Clear[n, d, qmcErrorBound, mcErrorBound, kd, kn]
n = 10^6;
d = 3;

qmcErrorBound[n_, d_] := Log[n]^d / n // N;
mcErrorBound[n_]      := 1 / Sqrt[n] // N;
mcErrorBoundX[n_]     := 1 / n^0.5 // N;

qmcErrorBound[n, d]
mcErrorBoundX[n]

Plot[qmcErrorBound[n, kd], {kd, 0, 50}, PlotRange → {{0, 9}, {0, 500}}]
```

```
0.00263694

0.001
```



Basic Calculus with Mathematica

```
(*Integration Basics*)

(*Use 'ESC int ESC' to enter∫ .. and 'ESC dd ESC 'to enter d*)
(*Use CTRL_ to enter the lower limit,then CTRL% for the upper limit*)
(*or use the Basic Math Assistant palette*)
Clear[x];
```

$$\int_0^1 1 / (x\text{^}3 + 1)\, dx$$

```
% // N

Integrate[1 / (x^3 + 1), {x, 0, 1}]
% // N
```

$$\frac{1}{18}\left(2\sqrt{3}\,\pi + \text{Log}[64]\right)$$

```
0.835649
```

$$\frac{1}{18}\left(2\sqrt{3}\,\pi + \text{Log}[64]\right)$$

```
0.835649
```

```
(*Calculate Pi(π) with integration*)
(*see LightTransport→theory→
  2014MonteCarloIntegration to integrate it with rnd samples.. aka 'crude qMC'*)
Clear[f];

f[x_] = 4 / (1 + x^2);

Integrate[f[x], {x, 0, 1}]
% // N
```

$$\int_0^1 f[x]\, dx$$

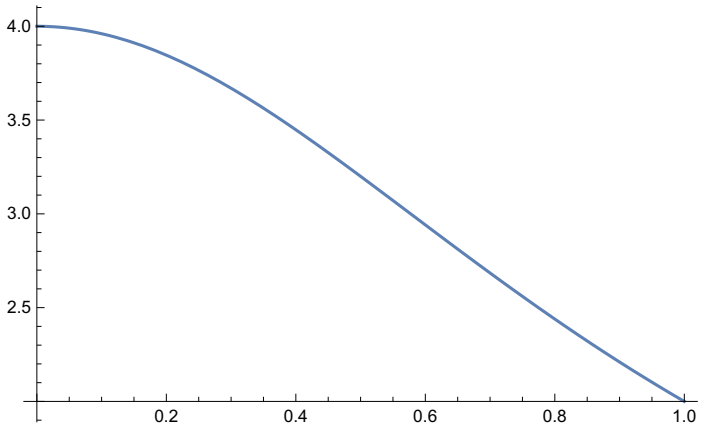
```
N[%, 20] (*Pi with 20 digits precision*)

Plot[f[x], {x, 0, 1}]
```

π

3.14159

π

3.1415926535897932385



```
(* Integration Strategies *)
(*https://
 reference.wolfram.com/language/tutorial/NIntegrateIntegrationStrategies.html*)

Timing[NIntegrate[E^-(x^4+y^4), {x, -2, 2}, {y, -2, 2}, Method → "MonteCarlo"]]
Timing[
 NIntegrate[E^-(x^4+y^4), {x, -2, 2}, {y, -2, 2}, Method → "QuasiMonteCarlo"]]
Timing[NIntegrate[E^-(x^4+y^4), {x, -2, 2}, {y, -2, 2},
  Method → {"RiemannRule", "Type" → "Right"}, PrecisionGoal → 2]]
Timing[NIntegrate[E^-(x^4+y^4), {x, -2, 2}, {y, -2, 2},
  Method → {"NewtonCotesRule", "Type" → "Closed"}]]
```

```
{0.018491, 3.2474}
```

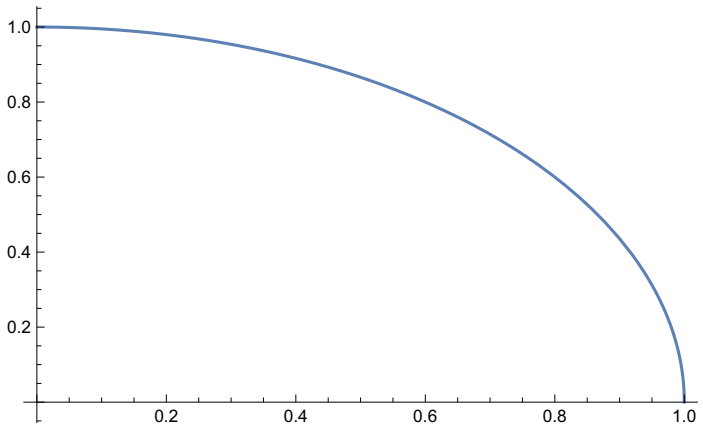
```
{0.054829, 3.28632}
```

```
{0.049942, 3.25423}
```

```
{0.118182, 3.28626}
```

```
Clear[f];
f[x_] := Sqrt[1 - x^2]
Plot[f[x], {x, 0, 1}]
```

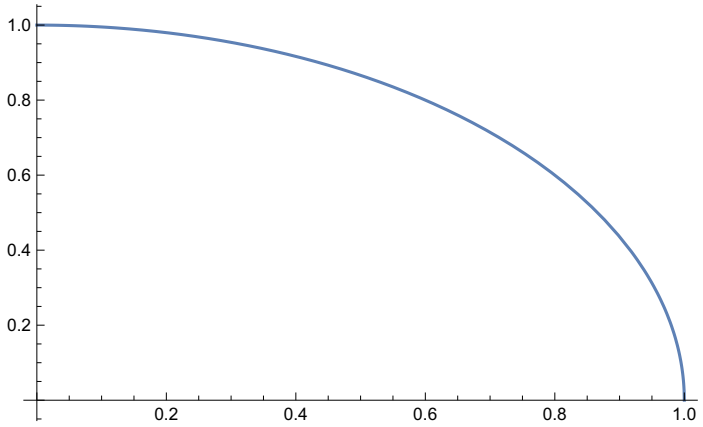
```
Integrate[f[x], {x, 0, 1}]
% // N
```



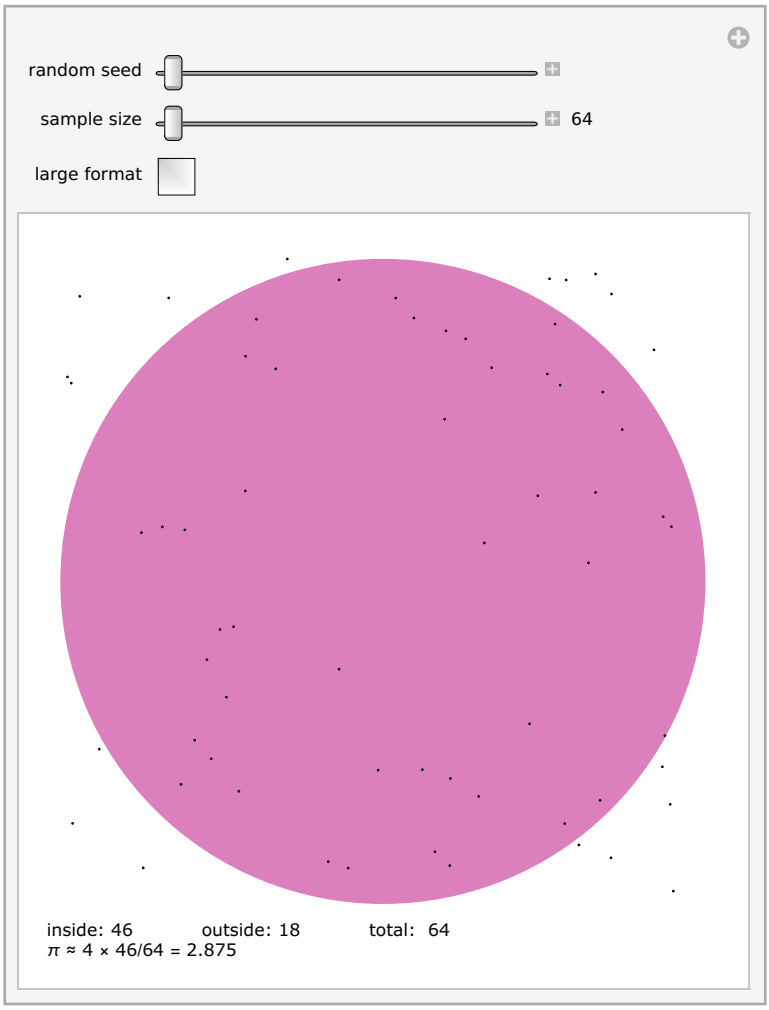
$$\frac{\pi}{4}$$

```
0.785398
```

see this: https://www.maplesoft.com/applications/view.aspx?sid=4009&view=html

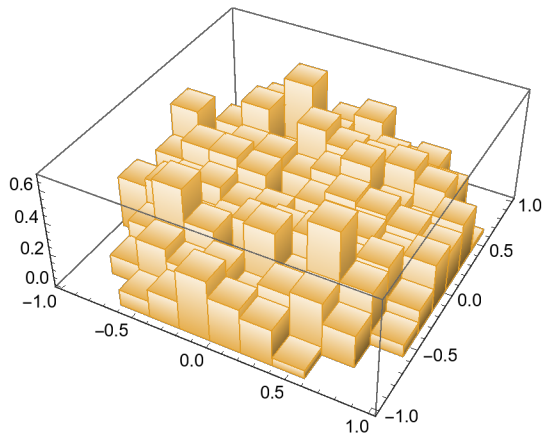


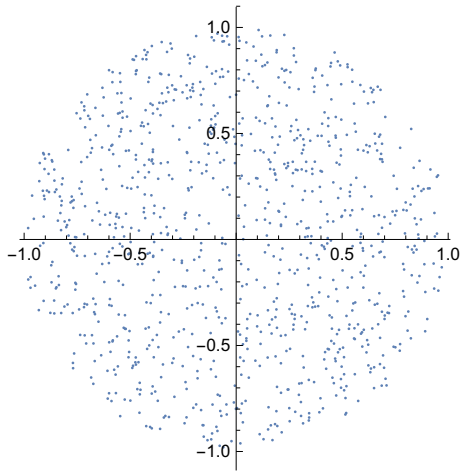

```
(* Hit-or-Miss Monte Carlo method to calculate π*)

(*The area ratio of a unit circle Ac over a unit square As is: Ac/As=
 πr^2 / 4r^2 = π/4 *)
(*We shoot (n) rnd pts over the square(-1,+1) and consider
 a hit for those pts (m) that satisfy: x^2+y^2≤1*)
(*Hence π = 4m/n ..ie. ratio of hit vs shot time 4*)

(*see 2013-Math Numerics and Programming (for mechanical engineers) ... *)
(*or MonteCarlo Integration in a Nutshell for better area estimation tecniques*)

Manipulate[
 Module[{data, inside, insidepts}, SeedRandom[n];
  data = RandomReal[{-1, 1}, {m, 2}];
  insidepts = Cases[data, {x_, y_} /; x^2 + y^2 < 1];
  inside = Length[insidepts];
  Text@Style[Column[{Graphics[{PointSize[0.004], RGBColor[0.86, 0.5, 0.74],
        Disk[{0, 0}, 1], Black, Point[data]}, ImageSize → If[format, 500, 350]],
     Row[{"inside: ", inside, "\toutside: ", m - inside, "\ttotal:  ", m}],
     Row[{"π ≈ 4 × ", inside, "/", m, " = ", 4. inside/m}]}], "Label"]
 ],
 {{n, 1, "random seed"}, 1, 1000, 1},
 {{m, 64, "sample size"}, 64, 8192, 64, Appearance → "Labeled"},
 {{format, False, "large format"}, {True, False}}, AutorunSequencing → {1, 2}]
```

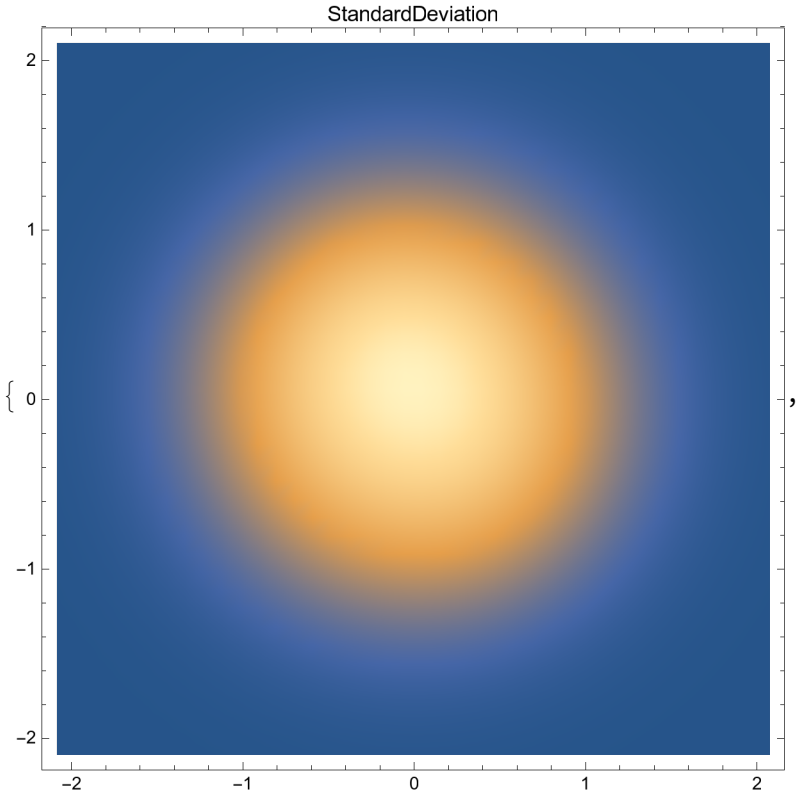

random seed
sample size
64
large format
inside: 46
outside: 18
total: 64
π ≈ 4 × 46/64 = 2.875

```
Clear[f, tf];
f[] := Block[{u, t, r},
  u = RandomReal[] + RandomReal[];
  t = RandomReal[] 2 Pi;
  r = If[u > 1, 2 - u, u];
  {r Cos[t], r Sin[t]}
 ]
tf = Table[f[], {1000}];

Mean[tf]
Histogram3D[tf, Automatic,
 "ProbabilityDensity", ChartElementFunction → "FadingCube"]
(*
SmoothHistogram[tf,50,"PDF",PlotRange→Automatic]
 SmoothHistogram[tf,50,"CDF"]
*)
ListPlot[tf, AspectRatio → Automatic]
Table[SmoothDensityHistogram[tf, name, ImageSize → 400, PlotLabel → name],
 {name, {"StandardDeviation", "StandardGaussian"}}]
```

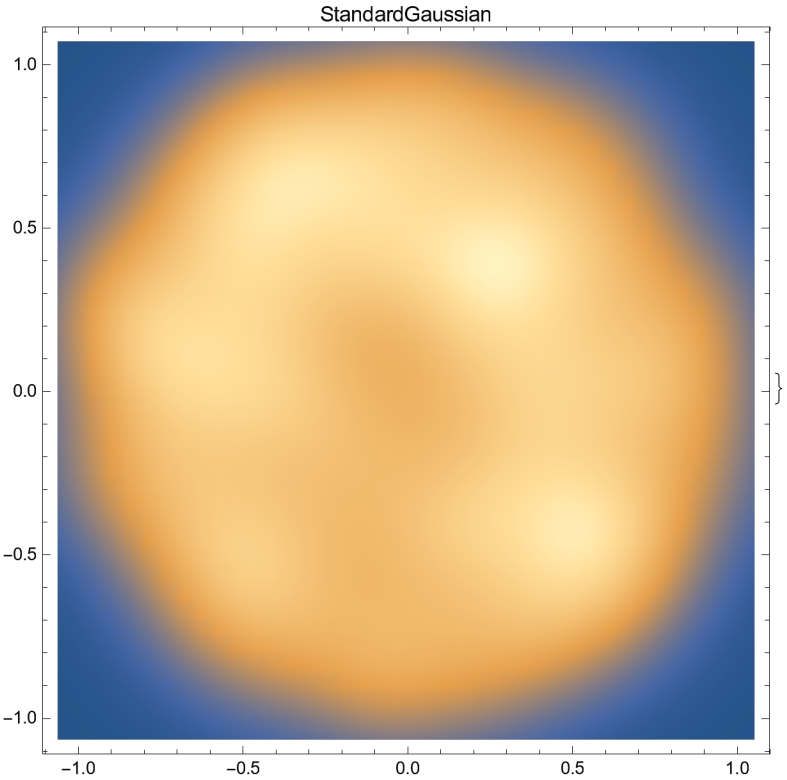
```
{-0.0153579, 0.0305612}
```



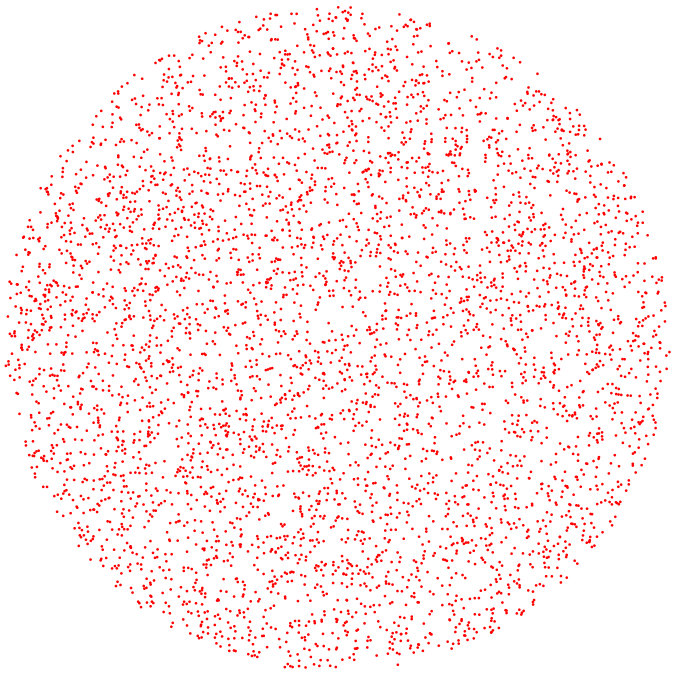

1.0
0.5
−1.0
−0.5
0.5
1.0
−0.5
−1.0


StandardDeviation
2
1
0
−1
−2
−2
−1
0
1
2

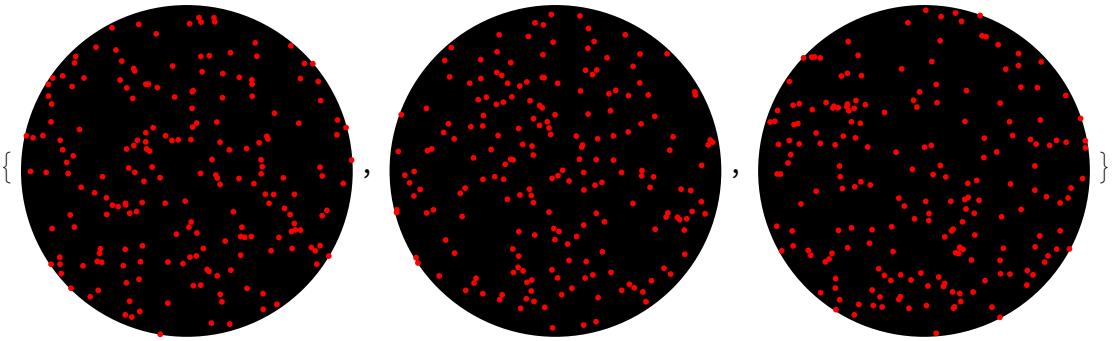
,


StandardGaussian
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

```
(*Generate a list of points in a unit disk.. new in M-11*)
pts = RandomPoint[Disk[], 5000];
Graphics[{PointSize[Tiny], Red, Point[pts]}]
```

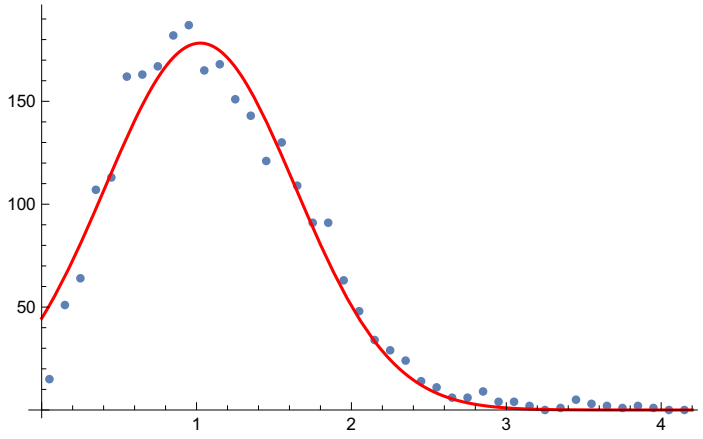


```
(*Generate multiple lists of points for a unit disk region*)
ℛ = Disk[];
pl = RandomPoint[ℛ, {3, 200}];
Table[Graphics[{ℛ, Red, Point[pts]}], {pts, pl}]
```



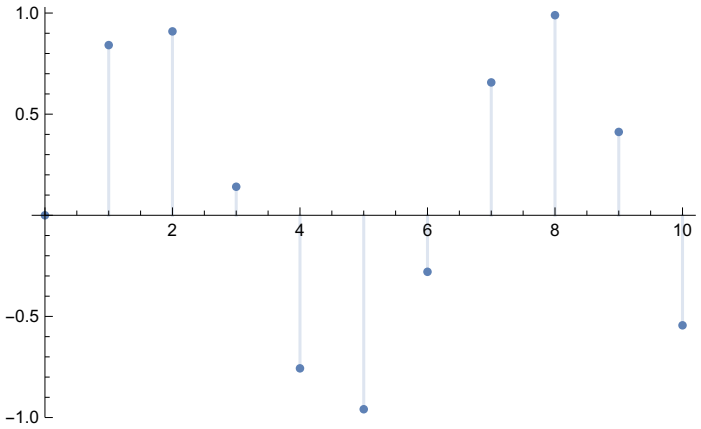
```
(*Gauss Fit to List of 2D Points*)
Clear[data, fit, sigma, x, ampl];
data = {{0.05, 15}, {0.15, 51}, {0.25, 64}, {0.35, 107}, {0.45, 113},
   {0.55, 162}, {0.65, 163}, {0.75, 167}, {0.85, 182}, {0.95, 187},
   {1.05, 165}, {1.15, 168}, {1.25, 151}, {1.35, 143}, {1.45, 121}, {1.55, 130},
   {1.65, 109}, {1.75, 91}, {1.85, 91}, {1.95, 63}, {2.05, 48}, {2.15, 34},
   {2.25, 29}, {2.35, 24}, {2.45, 14}, {2.55, 11}, {2.65, 6}, {2.75, 6},
   {2.85, 9}, {2.95, 4}, {3.05, 4}, {3.15, 2}, {3.25, 0}, {3.35, 1}, {3.45, 5},
   {3.55, 3}, {3.65, 2}, {3.75, 1}, {3.85, 2}, {3.95, 1}, {4.05, 0}, {4.15, 0}}
model[x_] = ampl Evaluate[PDF[NormalDistribution[x0, sigma], x]];
fit = FindFit[data, model[x], {ampl, x0, sigma}, x]

(*{ampl→274.765,x0→1.02404,sigma→0.614853}*)

Show[ListPlot[data], Plot[model[x] /. fit, {x, 0, 4.2}, PlotStyle → Red]]
```

```
{{0.05, 15}, {0.15, 51}, {0.25, 64}, {0.35, 107}, {0.45, 113}, {0.55, 162}, {0.65, 163},
 {0.75, 167}, {0.85, 182}, {0.95, 187}, {1.05, 165}, {1.15, 168}, {1.25, 151},
 {1.35, 143}, {1.45, 121}, {1.55, 130}, {1.65, 109}, {1.75, 91}, {1.85, 91},
 {1.95, 63}, {2.05, 48}, {2.15, 34}, {2.25, 29}, {2.35, 24}, {2.45, 14}, {2.55, 11},
 {2.65, 6}, {2.75, 6}, {2.85, 9}, {2.95, 4}, {3.05, 4}, {3.15, 2}, {3.25, 0}, {3.35, 1},
 {3.45, 5}, {3.55, 3}, {3.65, 2}, {3.75, 1}, {3.85, 2}, {3.95, 1}, {4.05, 0}, {4.15, 0}}

{ampl → 274.765, x0 → 1.02404, sigma → 0.614853}
```

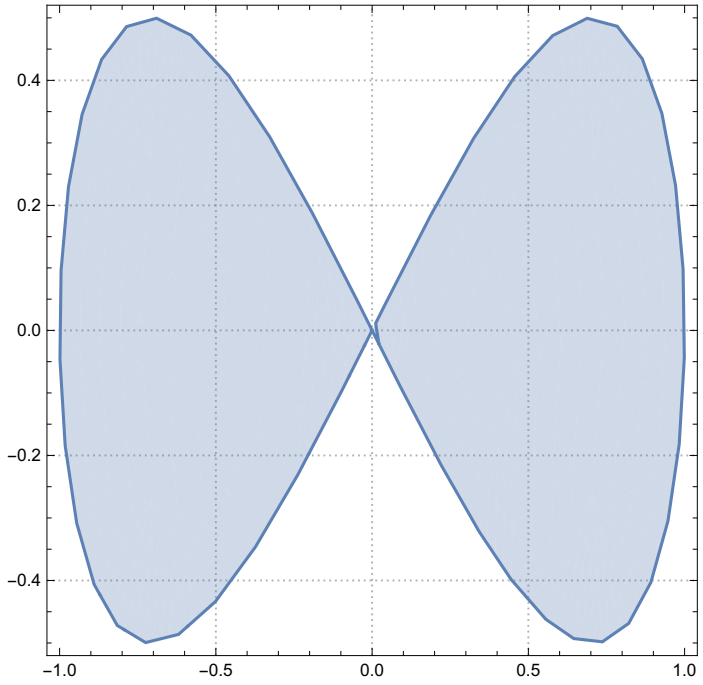


```
(*DiracComb to sample a function*)
Clear[x, a, b, c, e, f];
f[x] = ae^-((x-b)^2/2 c^2) (*gaussian fnc*)
DiscretePlot[Sum[DiscreteDelta[t-n] Sin[t], {n, Infinity}], {t, 0, 10}]
```

$$a e^{-\frac{1}{2} c^2 (-b+x)^2}$$



```
(*Integrate a fnc over a parametric domain ... new in M-11*)
Clear[f, x, y, region];
region = ParametricRegion[{{s, s t}, s^2 + t^2 ≤ 1}, {s, t}];
RegionPlot[region, ImageSize → Medium, PlotTheme → "Detailed", PlotLegends → None]

f[{x_, y_}] := x^3 - 2 x^2 y + 4 x^6 - y^5;
val = NIntegrate[f[{x, y}], {x, y} ∈ region]

(*visualize the convergence of the Monte
 Carlo statistic as the sample size increases*)
(* uncomment it as it takes time to get resolved *)
(*
ListLogLogPlot[Transpose[{2^Range[20],ParallelTable[
    RegionMeasure[region] Mean[f/@RandomPoint[region,2^n]],{n,1,20}]}],
 PlotRange→All,PlotTheme→"Detailed",ImageSize→Medium,Joined→True,Filling→val]
*)
```




```
0.812698
```

```
(* Use Monte Carlo sampling to calculate
 the area of the white area in the figure below *)
pannter = ImageData[Image[ColorConvert[  , GrayLevel], "Bit"]];

n = 100 000;
Total[Table[If[pannter[[RandomInteger[{1, Dimensions[pannter][[1]]}],
        RandomInteger[{1, Dimensions[pannter][[2]]}]]] == 1, 1, 0], {n}]] / n // N

(* Compare with a direct count of bright points,1s,with dark points (0s): *)
Count[Flatten[pannter], 0];
N[Count[Flatten[pannter], 1] / Length[Flatten[pannter]]]
```

```
0.36752
```

```
0.366691
```

Stochastic Processes

Till now what we've seen it's more about general statistical methods powered by probability theory. The average life expectancy of a population, what's the probability of having a female baby after two males ;) etc. If you're still interested in that, better go elsewhere to study them in detail.
From now on we'll focus on Computer Science and Computer Graphics probabilistic/stochastics approaches.

Let's start trying to understand what is a determinist approach vs a stochastic one.

A deterministic model generally does not include elements of randomness. Every time one runs a model with the same initial conditions he will get the same results. Most simple mathematical models of everyday situations are deterministic, for example, the height (h) in metres of an apple dropped from a hot air balloon at 300m could be modelled by h = - 5t2 + 300, where t is the time in seconds since the apple was dropped.

However let's say that we can't exactly measure the height from where the apple is dropped and neither exactly the time it take to fall on the ground, or better let's say the those measure are slighty different every time we redo the simulation. How to deal with that ? With a probabilist approach. A probabilistic model includes elements of randomness. Every time one runs the model, he's likely to get different results, even with the same initial conditions. A probabilistic model is one which incorporates

some aspect of random variation.

A deterministic model is a model where:
1 - the material properties are well known, i.e. deterministic. none of them is random
2 - The applied load are also deterministic

A Stochastic model has on the other hand:
1 - random properties; or normal distribution (of a given mean or standard deviation)
2 - the applied load is random variable, e.g. wind Load, earthquake (vibration of random amplitude and displacement)

A deterministic model is one that uses numbers as inputs, and produces numbers as outputs.

A stochastic model includes a random component that uses a distribution as one of the inputs, and results in a distribution for the output. These distributions may reflect the uncertainty in what the input should be (e.g. a deterministic input plus noise), or may reflect a random process (i.e. a stochastic input).

For example we can use a deterministic approach if all we want is to shoot a couple of rays and analyse their hit.
However it's better we use a stochastic approach if you wanna run a simulation with milions of rays and evaluate their intersections.

The notion of a **stochastic processes** is very important both in mathematical theory and its applications in science, engineering, economics, etc.
It is used to model a large number of various phenomena where the quantity of interest varies discretely or continuously through time in a non-predictable fashion.
Every stochastic process can be viewed as a function of two variables - t and ω. For each fixed t, $\omega \rightarrow X_t(\omega)$ is a random variable, as postulated in the definition. However, if we change our point of view and keep ω fixed, we see that the stochastic process is a function mapping ω to the real-valued function $t \rightarrow X_t(\omega)$. These functions are called the trajectories of the stochastic process X.

Now, let's start to see a bit more in detail a stochastic process.

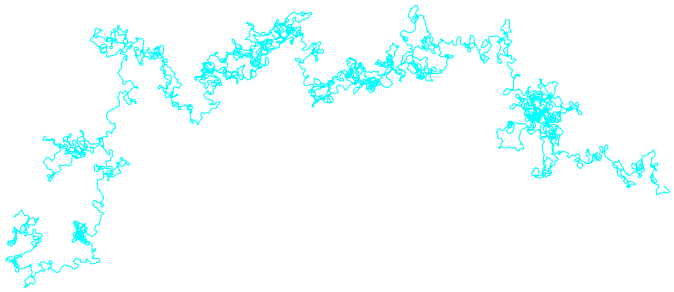
A stochastic process $\{X_n\}_{n \in \mathbb{N}_0}$ is called a simple random walk if :
- $X_0 = 0$,
- the increment $X_{n+1} - X_n$ is independent of $(X_0, X_1, \ldots, X_n)$ for each $n \in \mathbb{N}_0$, and
- the increment $X_{n+1} - X_n$ has the coin-toss distribution

On the other side, a random walk is called 'simple' when the size of each step is fixed (equal to 1) and it

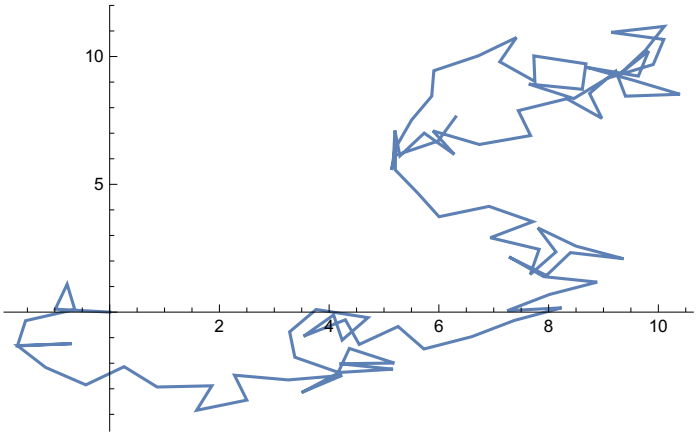
is only the direction that is random.

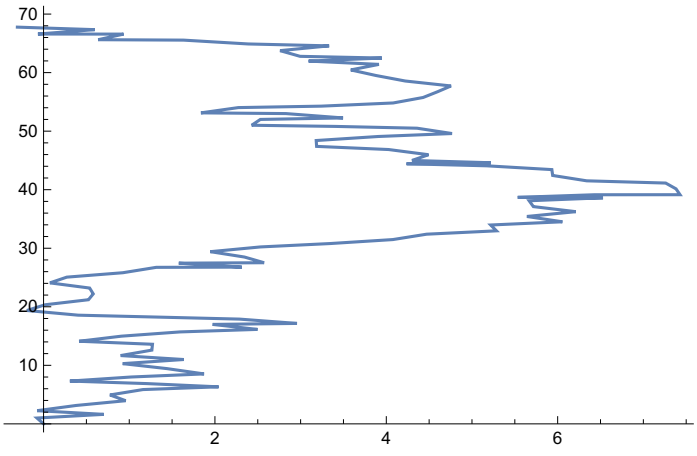
Below are all applications of a simple random walk using different Mathematica approaches that have all a common thing ...
a random variate to generate the next direction :

```
Block[{size = 0.5},
 l = Line[
   Accumulate[
    Function[x, size * {Re[x], Im[x]}, Listable]
     [Exp[I ×
       Accumulate[
        RandomVariate[
         NormalDistribution[0, Pi / 4],
         10^4]]]]]];
 Graphics[{Cyan, l}]
]
```

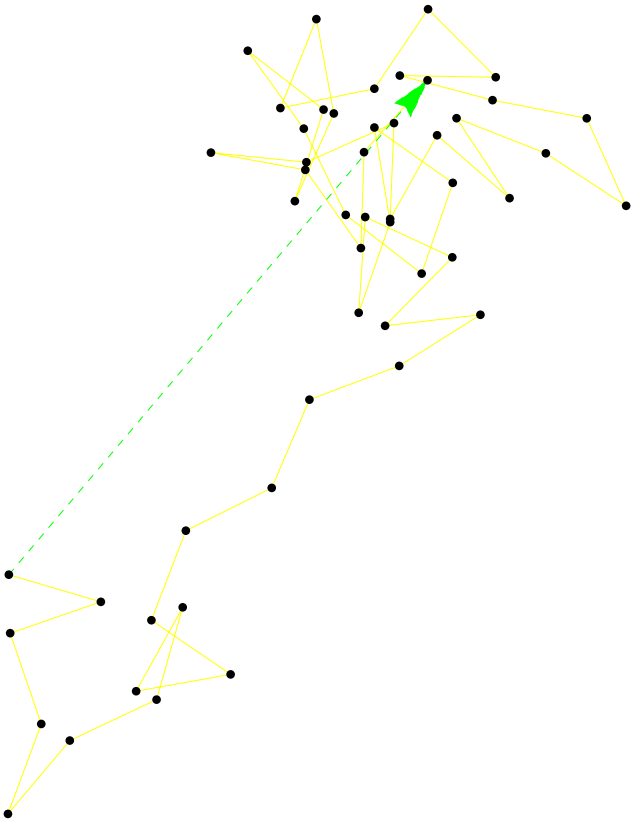


```
(* random walk for ω ∈ {0,2π}, ie. at any new segment we can turn up to 360°*)
randomWalk2D[t_] := Accumulate[Prepend[
    RandomPoint[DiscretizeRegion[Circle[]], t],
    {0, 0}]] // ListLinePlot
randomWalk2D[100]

(* forward biased random walk for ω ∈ {0,π},
ie. at any new segment we can turn up to 180°, aka never look back *)
hemicircle = Circle[{0, 0}, 1, {0, Pi}];
randomWalk2Dforward[t_] := Accumulate[Prepend[
    RandomPoint[DiscretizeRegion[hemicircle], t],
    {0, 0}]] // ListLinePlot
randomWalk2Dforward[100]

(* use 'sphere' instead of 'circle' and {0,0,0} instead of {0,0} for 3D walks *)
```





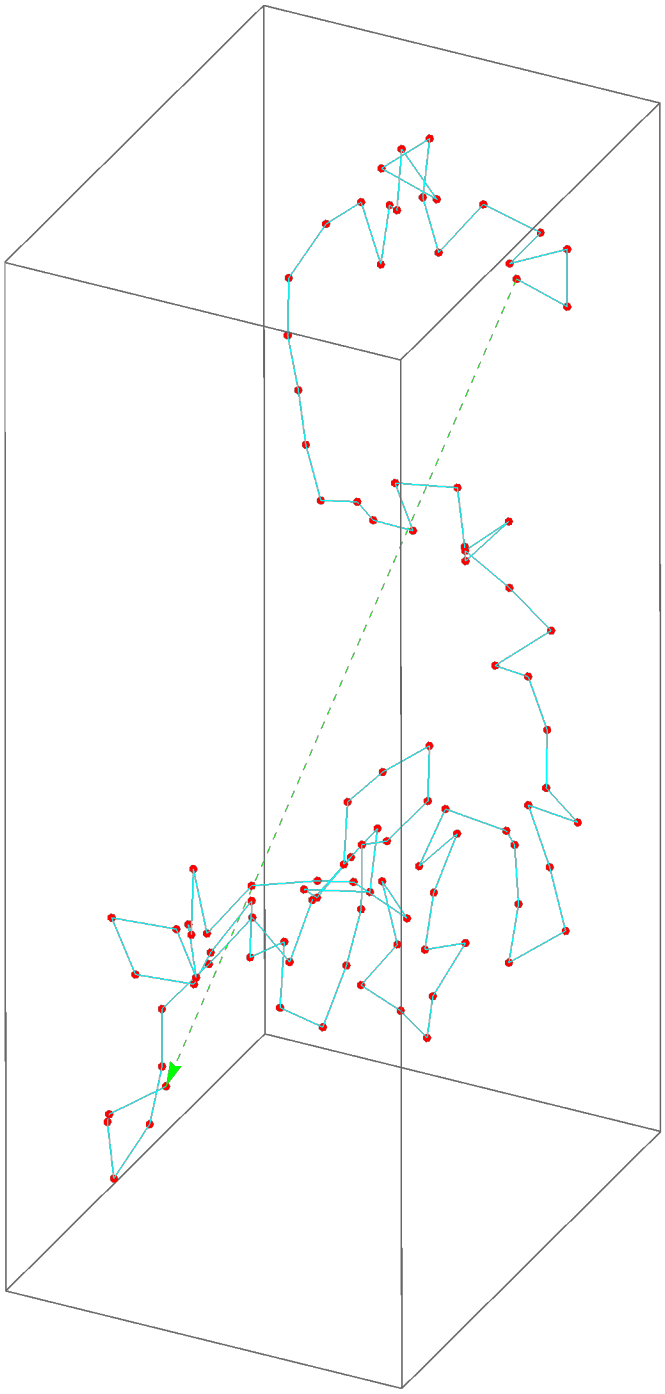
```
(* 2D simple random walk *)

step[position_] := With[{t = 2 Pi RandomReal[]}, position + {Cos[t], Sin[t]}];
walk[n_, origin_: {0, 0}] := NestList[step, origin, n];
distance[aWalk_] := EuclideanDistance @@ aWalk[[{1, -1}]];
(* calc dist between first and last pt in the walk list *)
draw[aWalk_] := Graphics[
   {Yellow, Line[aWalk], Dashed, Green, Arrowheads[Large], Arrow[aWalk[[{1, -1}]]],
    Black, PointSize[Medium], Point[aWalk]}, ImageSize → Medium];

step[{10, 0}];
w = walk[50];
draw[w]
distance[w]
```



```
6.75244
```

```
(* 3D simple random walk *)

Clear[step, walk, distance, draw];

(* comment this out to have everytime a different random walk as above *)
SeedRandom[12 345];

(* change t to Pi for forward only directions for example *)
step[position_: {0, 0, 0}] := With[{t = 2 Pi RandomReal[], p = Pi RandomReal[]},
   position + {Cos[t] Sin[p], Sin[t] Sin[p], Cos[p]}];

walk[n_, origin_: {0, 0, 0}] := NestList[step, origin, n];
distance[aWalk_] := EuclideanDistance @@ aWalk[[{1, -1}]];
(* calc dist between first and last pt in the walk list *)
draw[aWalk_] := Graphics3D[{Cyan, Line[aWalk], Dashed, Green, Arrowheads[Large],
    Arrow[aWalk[[{1, -1}]]], Red, PointSize[Medium], Point[aWalk]},
   ImageSize -> Large, ViewProjection → "Orthographic", ViewPoint → {Pi, Pi / 2, 2}];

distance[w]
draw[walk[100]]
```

```
6.75244
```



Simulate and plot multiple paths of the following stochastic difference equation,
for any initial condition $\pi 0 \in (0,1)$: $\pi t+1=\gamma \pi t$ with probability $1/3 \pi t$ and $\pi t+1=(1-\gamma)\pi t$ with probability $2/3 \pi t$, where $\gamma \in (0,1)$ is a parameter.

```
Clear[pi0, gamma];
SeedRandom[1234];
pi0 = RandomReal[]
gamma = RandomReal[]
nextpi[pi_] :=
 RandomChoice[{1 / 3 * pi, 2 / 3 * pi, 1 - pi} → {gamma * pi, (1 - gamma) * pi, pi}]
trajectory = NestList[nextpi, pi0, 20];
ListPlot[trajectory]
```

```
0.681162
```

```
0.854217
```