Simulation

You might have noticed that we are using as example many *simulated data sets*. In theory, these are computer generated data sets, from a given distribution, that are just like what you would obtain by observing real data that followed that distribution.

There are several tricky parts in the statement above. First of all, to say that "real data follow a specific distribution" is quite a leap in mathematical "realism". Our distributions are mathematical *models*, that we hope come close to describing actual real-life observations. That is fine, we will correct the statement to mean that the computer generated data will be the same as we would observe if we observed a *mathematical system* that obeyed our mathematical model.

Even this is a bit tricky. First, it's not quite clear what the statement actually means precisely. But, even if we accept it at intuitive value, the problem lies in the "computer-generated" part. In fact, (current, digital) computers are anything but random. What a random number generator in computer programming actually is, it is a perfectly deterministic routine that creates a string of numbers starting from a *seed* according to some perfectly determined algorithm. While the seed can be thought of as random (e.g., the number of seconds elapsed since the last boot of the system), from then on the sequence is perfectly determined. However, this routine is chosen so that, even though it is hardly random, it *behaves as if* it was random: the sequence is so complicated that it looks random, and, with some tweaking, it will pass at least some statistical tests that try to check for randomness.

How to come up with this smoke-and-mirrors act is a big topic in computer science, and is constantly investigated, but experience shows that, strange enough, we are not doing too bad in this respect.

It should be noted that before the advent of digital computers (in the 1940s) physical phenomena were used to generate *analog* random numbers, based on the well established fact that many physical phenomena are essentially random, with theoretically established features. The problem with these tools was their relative slowness, and the enormous speed of digital computers made the trade-off for a less solid "randomness" worth the price. Very recently, there has been a revival in analog random generators, so that, for highly delicate simulations, we might be be able to have a "genuine" random source, coupled with acceptable speed.

1 How Do I set Up A Simulation?

The answer depends on what you really mean by "Simulation". A full-fledged simulation can be a major programming project. In our limited scope, we may simply want to simulate repeated, independent, identically distributed observation of a random variable, with a given distribution. This can be done by writing a short program in most any programing language, but here we will focus on using a spreadsheet.

All spreadsheets (just like most all programing languages) have built-in functions that produce a "random number" each time they are called. In principle, these random numbers are produced independently of each other, all from a fixed distribution. Like we discussed above, the real story is quite different, but it turns out that the results are indeed satisfactory for all but the most demanding circumstances. Hence, we will pretend that what you get is what you have been promised.

1.1 The Easy Way

Your spreadsheet will have, under "functions" a subsection listing random number generators. Usually the names are fairly self-explanatory, but just looking at their description will tell you what distribution they are simulating, and what parameters they need.

Gnumeric has an amazing array of choices, including all the ones that you are most likely going to need. Of these, the following only are also available in Excel and in Open Office/LibreOffice:

- `rand`: generate random numbers uniformly distributed between 0 and 1
- `randbetween`: generate integer random numbers between two given values

Disturbingly enough, this list does not include a Gaussian generator.

1.2 The Hard Way

Actually, most programming languages only have a built-in function equivalent to `rand` above. Fact is, at least in theory, that's enough to generate any other distribution, with a little programing savvy. The reference trick is the following:

> Suppose X is a random variable, uniform between 0 and 1, we can, in principle, construct another random variable, $f(X)$, with any prescribed distribution.
>
> Indeed, consider a distribution with cumulative distribution function F—that is, such that a random variable with that distribution, Y, would be such that $P[Y \leqslant y] = F(y)$. Since F is non-decreasing, we can always construct its inverse function F^{-1} (if F is constant here and there, we need to make some adjustments, but, for simplicity, let's assume that F is always increasing). Also notice that $0 \leqslant F(y) \leqslant 1$, always.
>
> The trick consists in programming the random variable $F^{-1}(X)$:
>
> $$P\big[F^{-1}(X) \leqslant y\big] = P[X \leqslant F(y)] = F(y)$$

This is nice, but only works for distributions that allow us to compute F^{-1} explicitly. The exponential distribution is one such: if Y is exponential with parameter λ,

$$\begin{aligned} P[Y \leqslant y] &= 1 - e^{-\lambda y} = F_Y(y) \\ 1 - e^{-\lambda y} = z &\text{ implies } e^{-\lambda y} = 1 - z \\ y = -\frac{1}{\lambda}&\ln(1-z) \,(\text{note that } 0 \leqslant z \leqslant 1) \end{aligned}$$

In other words, if U is a random variable, uniformly distributed over $[0, 1]$, $X = -\frac{1}{\lambda}\ln(1-U)$, will have exponential distribution, with parameter λ.

Interestingly, another distribution that can be managed this way is the Cauchy distribution. It turns out that its cumulative distribution function is $F(x) = \frac{1}{2} + \frac{1}{\pi}\tan^{-1}(x)$, so that, using the same trick as for the exponential distribution, $W = \tan\Big(\pi\Big\{U - \frac{1}{2}\Big\}\Big)$, will have Cauchy distribution.

These two examples are used in real life, but they illustrate a drawback, as far as their practical implementation: we need to evaluate logarithms or trigonometric functions, operations that are slow, and prone to much higher rounding errors than algebraic operations.

This problem is greatly magnified by one of the many tricks proposed to simulate the Gaussian distribution (we cannot obviously use the inversion of the cumulative distribution functions, since we have no simple explicit way for it). A trick that is often quoted relies on a remarkable fact:

> Assume X and Y are two independent standard normal variables, and consider them as coordinates in the plane. The corresponding polar coordinates, say R, and θ, have simple distributions:
>
> - R^2 is exponential with parameter $\frac{1}{2}$
> - θ is uniform over $[0, 2\pi]$
>
> Hence, reversing the argument, if E, is an exponential random variable with parameter $\frac{1}{2}$, and G is uniform over $[0, 2\pi]$,
>
> $$X = \sqrt{E}\cos(G), Y = \sqrt{E}\sin(G)$$
>
> are two independent standard normal random variables. Since we can create them as
>
> $$E = -2\ln(1-U), G = 2\pi V$$
>
> where U and V are independent variables, uniform over $[0, 1]$, we have our algorithm.

Unfortunately, this algorithm has the problem of using complicated functions in spades, and is not much loved by practicing simulators. In fact, there are several research papers about good ways to generate Gaussian random numbers (*Gnumeric* uses a famous one). As mentioned elsewhere in this course, an old IBM routine generated standard normal variables by adding 12 uniformly distributed numbers, as generated by a `rand`-like function. For our purposes, you could use this method, which compares favorably, in terms of speed, with other popular choices.

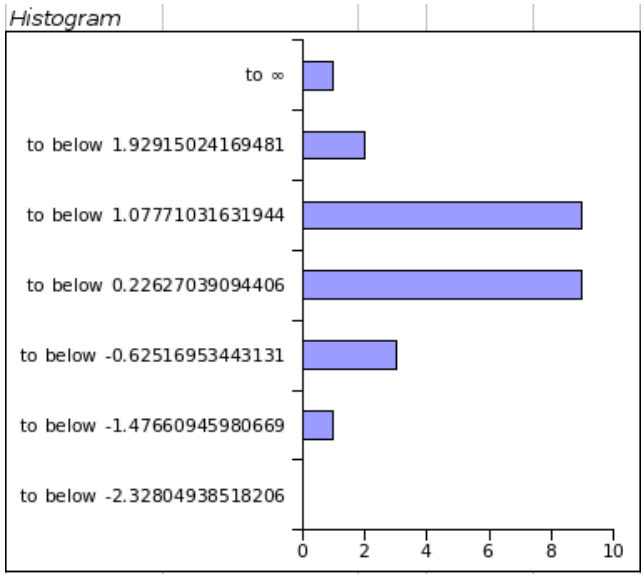
1.3 Trying This Out

As mentioned, `rand` will result in a pseudo-random number that will look like it was obtained from a uniform distribution between 0 and 1.

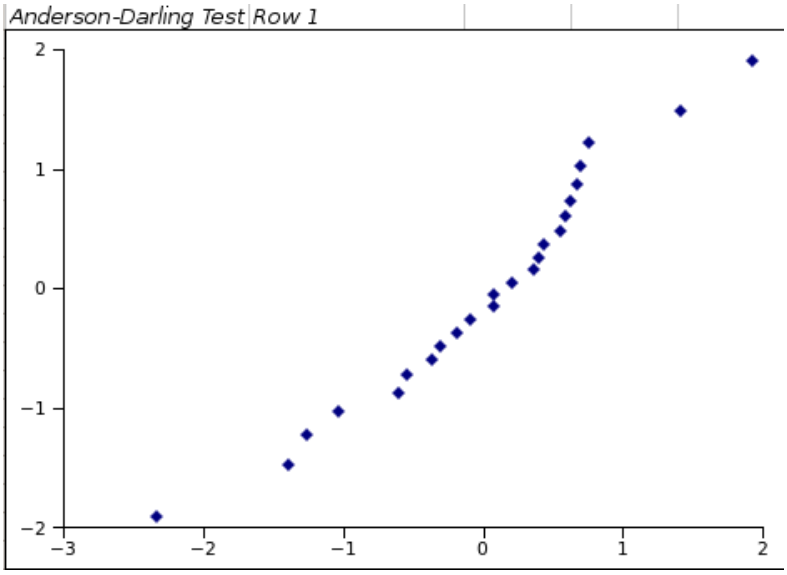
Hence `rand - 0.5`, will produce a uniform random number between $-\frac{1}{2}$, and $\frac{1}{2}$. Let's see what happens if we produce 12 of such numbers, and repeat this a few times (this example was produced with *Gnumeric*). We get the following table (if you try this at home, you will get different numbers, of course: these are supposed to be *random*, after all)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	0.241	0.159	0.144	0.115	0.024	0.446	0.069	0.371	0.029	0.152	0.445	0.107	0.273	0.354	−0.3
2	0.421	0.134	−0.040	0.041	0.035	0.450	0.267	0.276	0.082	−0.289	−0.224	−0.192	−0.421	0.494	0.0
3	0.087	−0.221	0.233	−0.315	0.458	−0.210	−0.497	0.498	0.419	0.196	−0.124	−0.391	−0.214	−0.379	0.0
4	0.332	0.266	0.308	0.395	0.400	0.043	0.079	0.295	−0.376	−0.080	0.172	0.042	−0.168	0.026	−0.2
5	0.233	−0.259	−0.450	−0.498	0.172	0.336	−0.152	0.453	−0.492	−0.407	−0.037	0.095	−0.401	−0.143	0.0
6	0.227	−0.247	0.288	0.453	0.493	0.147	−0.089	0.282	0.074	−0.331	−0.458	0.137	0.147	−0.094	−0.0
7	−0.330	−0.218	−0.278	−0.074	0.379	−0.212	0.268	0.371	−0.434	−0.308	−0.250	−0.497	−0.319	0.394	0.3
8	−0.274	0.482	−0.497	−0.420	−0.369	−0.012	0.279	−0.175	0.359	0.067	−0.120	0.150	−0.086	0.170	0.4
9	0.235	0.069	0.176	0.276	0.222	−0.412	−0.293	−0.337	0.476	0.150	0.097	0.380	0.274	−0.066	−0.3
10	−0.118	0.363	0.277	0.227	0.010	−0.296	0.125	−0.004	0.255	0.322	0.132	0.243	0.244	0.051	0.0
11	−0.253	−0.183	−0.239	0.179	0.026	0.173	0.079	−0.113	0.018	−0.493	−0.131	−0.037	0.163	0.282	−0.4
12	0.452	0.089	−0.222	0.023	−0.436	−0.087	0.455	0.012	0.209	−0.012	−0.045	0.037	0.145	−0.399	−0.0
13	**1.254**	**0.433**	**−0.301**	**0.402**	**1.414**	**0.366**	**0.591**	**1.929**	**0.620**	**−1.033**	**−0.541**	**0.075**	**−0.363**	**0.691**	**−0.5**

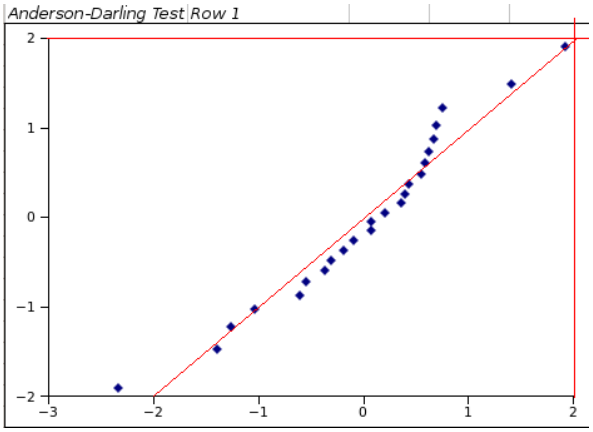
Each column has 12 "uniform random numbers", and the 13th row is their sum. If we draw a histogram with six bins for this row of data, we obtain this:



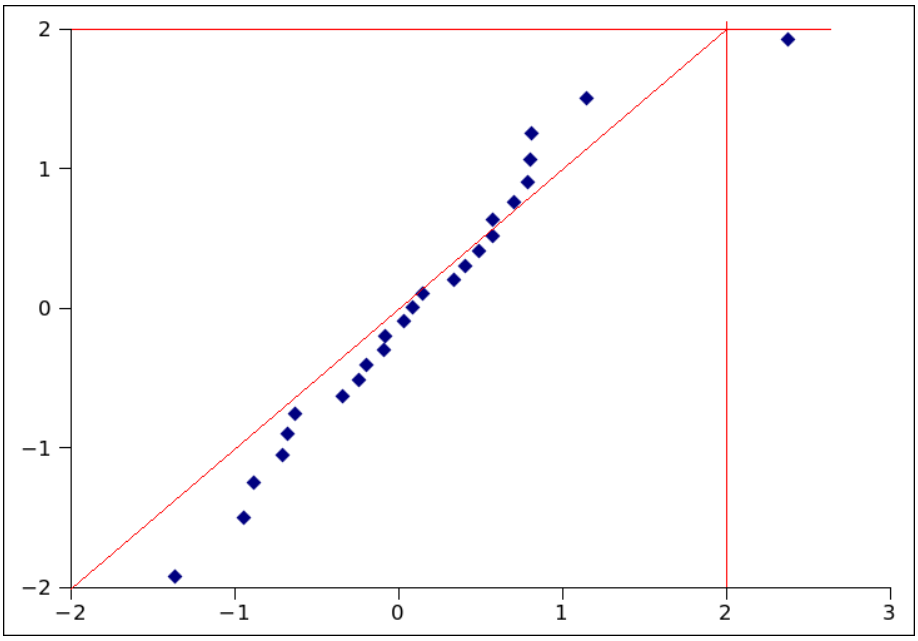
This might seem similar to a standard normal distribution, or not to you, so here is the result of a standard "goodness-of-fit" test (this is a *non parametric* test, which we are not discussing in this course, and it tries to determine whether a sequence of numbers can be thought as normally distributed.. The following is the graph of a "normality test": in theory, a sequence of independent normal numbers would be aligned along the diagonal.



Of course, we are using a subjective criterion here – looking at a picture. Here is the same scatterplot, with the diagonal superimposed. As you can see, the higher values are a bit too high. Still, since we are dealing with only 25 values here, this is still relatively good:



To have a reference, here is the corresponding diagram using Gnumeric's own normal simulation routine: as you can see, the 25 data points seem to be too small in the lower half, and too big in the higher half. There are quantitative ways to assess the result of this test, but they are outside our scope.


2
1
0
−1
−2
−2
−1
0
1
2
3