Litter Power Package Documentation

Litter Power Package

Design
Development
Documentation
Quality Assurance
Graphics
Marketing
Sales
Chief cook
Bottle-washing:
Peter Castine

Contact:
e-mail: 4-15@kagi.com
Fax: +49 (30) 4367 1058
WWW: http://www.bek.no/~pcastine/

What is the Litter Power Package?

The short answer: Random number generators, noise generators, interval mutation, and much, much more.

Litter began life as a set of objects implementing a large collection of random number distributions, including but not limited to all the generators documented in Denis Lorrain's seminal article "A Panoply of Stochastic 'Cannons'." In the original Litter Package these were implemented as patchers ("abstractions"). In the new Litter Power Package all of these generators have been implemented as external objects, resulting in execution speeds in the area of ten to twenty times faster than in previous incarnations. The generators include basic random distributions for the MIDI/control domain ("white" noise, Brownian motion, and 1/f noise) as well as "cannons" for generating continuous and discrete values according to many well-known distributions (binomial, linear, exponential, Poisson, and Gauss, to name a few).

Recognizing the limitations of the core random number generator provided by the Mac OS and C standard libraries, Litter Power uses more modern methods of generating random numbers that are both faster and more robust than the old linear congruence method used in Max. The random number generators in Litter Power produce 32-bit numbers with all bits random and cycles of 2^{88} and higher. With Litter Power you *can* use the least significant bit as a random coin toss, something that does not work well with the standard random object. Random distributions new to the Litter Power package include log-normal; a very flexible general-purpose implementation of the finite urn model; support for non-integral parameters to the Gamma distribution; plus an *I Ching* implementation supporting both coin and yarrow stick methods of appealing to the oracle.

Reflecting the significance of MSP in many current Max applications, "tilde" variants of the noise generators have been added, providing efficient generators for white, pink, brown, and gray noise, as well as sources for "popcorn" noise and more. Speaking of linear congruence, there is a parametric linear congruence module, with which you can generate a variety of sounds ranging from noise to pitched sounds. Find the shortest linear congruence cycle!

Also included with Litter Power are implementations of Interval Mutation algorithms, providing features analogous to the mutation functions in SoundHack and Argeïphontes Lyre, as well as supporting mutation of MIDI and other control data.

Finally, a number of utilities have been included in the package. Skew and kurtosis have been added to the statistical data calculated in previous versions of Litter (count, minimum, maximum, mean, and standard deviation). A novel group of objects for mapping and limiting values and signals have been added to your arsenal of Max tools. Phase unwrapping needs have been provided for. And more…

Enjoy.

Which Version of Litter Power?

Litter Power is distributed in the following forms: a Litter Power Starter Pack and a Litter Power Professional Bundle. The Professional Bundle is available in two alternate license forms, “Artistic” and “Institutional”

In brief:

1) The Litter Power Starter Pack is distributed free of charge, but is limited to a subset of the entire package and is for personal use only. You may not resell any of the components.

1) The Litter Power Professional Bundle is the full set of Litter Power objects, and may be used for research, artistic work, development, etc. The artistic license also allows you to resell Litter components in the context of original work (music projects and other artwork, or as part of Collectives). The institutional license is a multi-user license.

Please refer to your license for further specifics about usage. The overview above is a summary, the license text is definitive.

Site licenses are also available, please contact 4-15 Music & Technology for further details <mailto: 4-15@kagi.com>.

The Litter Power object overview below indicates which objects are included in the Starter Pack and which objects are only available in the Professional Bundle.

In terms of version numbering, all Litter Power objects include standard Mac OS version (‘vers’) resources, so that version information is displayed in The Finder as well as when using the Max Get Info… command. This should make it easier for you to compare versions of individual objects when package updates become available.

What's in Litter Power?

Discrete Distributions

lp.bernie	Bernoulli distribution, binary choice	All Bundles
lp.dicey	Dice (variable number of faces)	Pro Bundles only
lp.ernie	Arbitrary distributions ("finite urn" model)	Pro Bundles only
lp.ginger	I Ching (yarrow stick and coin oracles, calculate changes)	All Bundles
lp.lili	Parametric linear congruence pseudo-random number generator	Pro Bundles only
lp.pfishie	Poisson distribution	Pro Bundles only
lp.tata	Tausworthe 88 pseudo-random number generator. Cycle approximately 2^{88}.	All Bundles
lp.titi	Matsumoto's TT800 pseudo-random number generator. Cycle approximately 2^{800}.	Pro Bundles only

Continuous Distributions

lp.abbie	Arc sine and beta distributions	Pro Bundles only
lp.chichi	Chi square distribution	Pro Bundles only
lp.coshy	Cauchy distribution (the symmetrical or "bilateral" Cauchy distribution as well as positive and negative variants)	Pro Bundles only
lp.expo	Exponential and Laplace (bilateral exponential) distributions	Pro Bundles only
lp.fishie	Fisher distribution	Pro Bundles only
lp.gammer	Gamma and Erlang distributions	Pro Bundles only
lp.grrr	"Gray" noise	Pro Bundles only
lp.hyppie	Hyperbolic cosine distribution	Pro Bundles only
lp.linnie	Linear and triangular distributions	All Bundles
lp.loggie	Logistic distribution	Pro Bundles only
lp.lonnie	Log-normal distribution	Pro Bundles only
lp.norm	Normal (Gaussian) distribution	All Bundles
lp.pfff	$1/f^2$ distribution ("Brownian" noise)	All Bundles
lp.shhh	Uniform distribution ("white" noise)	All Bundles
lp.sss	1/f distribution ("pink" noise) using the Voss/Gardner algorithm	All Bundles
lp.stu	Student's *t* distribution.	Pro Bundles only
lp.y	Weibull/Rayleigh distribution	Pro Bundles only
lp.zzz	1/f distribution ("pink" noise) using the McCartney algorithm.	Pro Bundles only

Signal Generators

lp.frrr~	Low frequency noise	Pro Bundles only
lp.grrr~	Gray noise	Pro Bundles only
lp.lll~	(Parametric) linear congruence noise	Pro Bundles only
lp.pfff~	Brown noise	All Bundles
lp.phhh~	Black noise	Pro Bundles only
lp.ppp~	Popcorn noise ("dust")	Pro Bundles only
lp.shhh~	White noise	All Bundles
lp.sss~	Pink (1/f) noise, using the Voss/Gardner algorithm	All Bundles
lp.zzz~	Pink (1/f) noise, using the McCartney algorithm.	Pro Bundles only

Mutation Processors

lp.frim~	Frequency-domain (spectral) interval mutation	Pro Bundles only
lp.tim~	Time-domain interval mutation	All Bundles
lp.vim	Interval mutation of numeric values	Pro Bundles only

Utilities

lp.c2p~	Convert pairs of signals from Cartesian representation (e.g., **fft~** output) to polar form (e.g., for processing by **lp.frim~**)	All Bundles
lp.kg	Map I Ching output (in the range 1 to 64) to other ranges.	Pro Bundles only
lp.p2c~	Convert pairs of signals from polar representation (e.g., **lp.frim~** output) to Cartesian form (e.g. for processing by **ifft~**).	All Bundles
lp.i	Read texts of I Ching oracles	Pro Bundles only
lp.scampf	Scale, offset, and limit numbers; output floating-point values.	Pro Bundles only
lp.scampi	Scale, offset, and limit numbers; output integers.	All Bundles
lp.scamp~	Scale/map/limit signals to a user-specified output range.	Pro Bundles only
lp.stacey	Statistics: count, minimum, maximum, mean, standard deviation, skew, and kurtosis	All Bundles
lp.grl~	Phase unwrapping	Pro Bundles only

Installing (and Removing) Litter Power

Copy the Litter Power Package to a hard disk attached to your computer. The Litter externals can be located on anywhere you want; just make sure the Max File Preferences include a path that will lead to them. If you prefer, you can copy the content of the Litter Objects folder to your the main Max externals folder (this is normally a folder called *externals* inside the Max folder) or the Max Startup Items folder (normally *max-startup*, also in the Max folder). The contents of the Litter Help folder must be copied into the Help folder specified in Max' File Preferences. This is normally the folder *max-help* in the Max folder.

All Litter Power objects begin with the sequence of characters **lp.**. If you, for some reason, should need to remove them, use Sherlock (the Find… command in The Finder's File menu) to search for all items beginning with these three characters and remove them.

Using Litter Power

All Litter Power external objects respond (in an unlocked Patcher window) to the Get Info… command by displaying a small Alert Box with version and copyright information. The Get Info… command will also print information about current object settings in the Max window. This state information can also be generated by double-clicking on a Litter Power external object in a locked Patcher window. This has proven to be a useful debugging device while developing patches using Litter Power external objects. A few of the Litter Power external objects have no state information, with these objects a double-click will generate no information in the Max window (at least, nothing useful).

All Litter Power external objects generate a brief message in the Max window when they are loaded. This is to help remind you, should you forget, where they came from.

The random number distributors are based on the Tausworthe 88 random number generator. This algorithm is not only faster than the standard Linear Congruence algorithm used otherwise in Max, it is also much more robust. Tausworthe 88 generates 32-bit random numbers, with all bits exhibiting random properties (i.e., the bits show no correlations or 'patterns'), and has a cycle of approximately 2^{88} deviates before repeating (this is about 75,000,000,000,000,000 times longer than the longest possible 32-bit linear congruence cycle).

All random number generators auto-seed themselves, based on the current date and time of day, time since starting up your Mac, and other values garnered from the operating system. This guarantees that you will get a different set of random values every time you run any patch using Litter Power externals. This also guarantees that all Litter Power externals in your patch will be mutually independent and that no undesired correlations occur. However, if you wish a "random" pattern to be replicable, you can specify a non-zero seed as the final initialization argument to any of the random-value generating objects when you create it. You can "re-seed" objects that were created with a seed at any time. This may be useful for testing a patch, some people may also find this helpful for a kind of "controlled (pseudo-) randomness."

All random number generators respond to a bang by sending a random number from the given distribution out their leftmost outlet. Several of these objects can also be used as *mapping* objects, generating an output value that depends, in some way, on the input. Presuming that the input values follow a uniform distribution in the range from zero to one, the output values will follow the given random distribution. For instance, **lp.expo** will map uniformly distributed input values to an exponential distribution. Of course, if the input values are not uniformly distributed, the distribution of the output values will be something weirdly different from the named distribution. Still, this may be fun. You may find the **lp.scampi** object useful for mapping MIDI input to values to the unit range in this context. See the documentation of the individual objects for further details.

Several of the noise and random number generators have a parameter called an NN factor. This parameter controls the number of bits of noise generated. The parameter is always set to zero by default, but positive integral values will cause low-order bits to be cleared, generating "low quality" noise. Very small values of the NN factor do not perceptibly change the sound of noise generated, but values close to the maximum of 31 can effect the audio signal quite significantly. The name of this factor bears no resemblance to any fictional character invented by Dostoyevsky.

Talking about names…

Why "Litter"?

The use of the term 'cannon' to describe an algorithm for generating random numbers was introduced by Iannis Xenakis. The motivation for this term comes from the French idiom *tirer au hasard* (choose at random). Obvious, isn't it?

In the same vein, when the original Litter Package was developed, I could not resist the temptation to name it after a relatively obscure pun on the German word for dice. (*Würfeln* = throw dice, but conceivably a diminutive of *Wurf* = litter of puppies.)

Despite obscure etymology, the name has gained acceptance. Long live Litter! Over time, a number of objects have been developed that go beyond the generation of random distributions, but seem in some way or another related to using random processes in music (as well as some deterministic processes). These now take their place in the Litter canon.

With only a few exceptions, Litter follows the longstanding Max tradition of, shall we say, whimsical object names. The logic behind the more obscure derivations is explained in the documentation of the individual objects (see the "What's in a name?" sections). However, all object names are prefixed **lp.** (for Litter Power) in an effort to maintain an independent name space. The Litter Power Thesaurus should be an aid in finding your way to the object you need, no matter how surprising the name may appear.

Acknowledgements

Much of the development work for the Litter Power package was done during a residency in Bergen, Norway, as a guest of the Bergen senter for elektronisk kunst (BEK) and Kulturhuset USF. Thanks particularly to Jørgen, Trond, and Christian of BEK for their encouragement and enthusiasm.

Several of the ideas for different kinds of noise generators were inspired by James McCartney's SuperCollider application. James took the time to discuss details of different algorithms for generating random numbers despite my working on a "competing" project; I am indebted to him both as a scholar and a gentleman.

Thanks also to Larry Polansky and Tom Erbe for their support while developing the Interval Mutation objects.

Without the work of Miller Puckette, David Zicarelli, and all the folks at Cycling 74 there would be no Max, no MSP, and no Litter. Thanks.

Finally, thanks are due to the Litter Pro Beta Testers as well as the numerous people who provided feedback on the original Litter package.

Bibliography

I Ging, trans. Richard Wilhelm (Munich: Eugen Diederichs, 1973).

I Ching or *Book of Changes*, trans. by Cary F. Baynes (from the German translation with commentaries by Richard Wilhelm). (Princeton, New Jersey: Princeton University Press, 1967).

Ahrens, Joachim. H. and Ulrich Dieter, "Computer Methods for Sampling from Gamma, Beta, Poisson, and Binomial Distributions," *Computing* 12 (1974): 223-246.

Ahrens, Joachim H. and Ulrich Dieter, "Generating Gamma Variates by a Modified Rejection Technique," *Communications of the ACM* 25, no. 1 (1982): 47-54.

Ames, Charles, "A Catalog of Statistical Distributions: Techniques for Transforming Determinate, Random, and Chaotic Populations," *Leonardo Music Journal* 1, no. 1 (1991): 55-70.

Ames, Charles, "A Catalog of Sequence Generators: Accounting for Proximity, Pattern, Exclusion, Balance and/or Randomness," *Leonardo Music Journal* 2, no. 1 (1992): 55-72.

Austin, Larry. "An interview with John Cage and Lejaren Hiller." *Source* 4, no. 2 (1968): 11-19. Reprinted in *Computer Music Journal* 16(4), pp. 15-29, 1993.

Behnen, Konrad and Georg Neuhaus, *Grundkurs Stochastik*, *Teubner Studienbücher Mathematik* (Stuttgart: Teubner, 1984).

Cage, John. *Silence*. (London: Marion Boyars, 1987).

Cheng, Russel C. H., "Generating Beta Variates with Non-Integral Shape Parameters," *Communications of the ACM* 21 (1978): 317-322.

Dodge, Charles and Thomas A. Jerse, *Computer Music*, *Synthesis*, *Composition*, *and Performance* (New York: Schirmer, 1985).

L'Ecuyer, Pierre, "Maximally Equidistributed Combined Tausworthe Generators," *Mathematics of Computation* 65 (1996): 203-213.

Gardner, Martin, "Mathematical Games: White and Brown Music, Fractal Curves, and One-over-*f* Fluctuations," *Scientific American* 1978, 16-31.

Hawking, Stephen W., *A Brief History of Time.* (London/New York: Bantam 1988)

Hiller, Lejaren A. and Leonard Isaacson, *Experimental Music* (New York: McGraw-Hill, 1959).

Huang, Kerson and Rosemary Huang, *I Ching* (New York: Workman, 1987).

Jou, Tsung Hwa, *Tao of I Ching: Way to Divination* (Boston: Tuttle, 1985)

Knuth, Donald E., *The Art of Computer Programming*, Vol. 2 *Semi-Numerical Algorithms*. (Reading, Mass.: Addison-Wesley, 1972).

Lorrain, Denis, “A Panoply of Stochastic ‘Cannons’,” in *The Music Machine*, ed. Curtis Roads (Cambridge, Massachusetts: MIT, 1989), 351-379.

Matsumoto, Makoto and Yoshiharu Kurita, “Twisted GFSR Generators II,” *ACM Transactions on Modelling and Computer Simulation* 4, no. 3 (1994): 254-266.

Polansky, Larry, “Morphological Metrics: An Introduction to a Theory of Formal Distances” (paper presented at the International Computer Music Conference, Champaign-Urbana, 1987), 197-204.

Polansky, Larry and Tom Erbe, “Spectral Mutation in *SoundHack*: A Brief Description” (paper presented at the International Computer Music Conference, Banff, Canada, 1995), 307-314.

Polansky, Larry, “Morphological metrics,” *Journal of New Music Research (formally Interface)* 25 (1996): 289-368.

Salkind, Neil J., *Statistics for People Who (Think They) Hate Statistics.* (Thousand Oaks, California: Sage, 2000)

Voss, Richard F. and John Clarke, “1/*f* Noise in Music: Music from 1/*f* Noise,” *Journal of the Acoustic Society of America* 63, no. 1 (1978): 258-263.

Xenakis, Iannis, *Formalized Music* (Bloomington, Indiana: Indiana University Press, 1971).

Thesaurus

1/f distribution	**lp.sss, lp.zzz, lp.sss~, lp.zzz~**
Amesian feedback	**lp.ernie**
Arc sine distribution	**lp.abbie**
Bernoulli Trials	**lp.bernie**
Beta distribution	**lp.abbie**
Bilateral exponential distribution	**lp.expo**
Binary Choice	**lp.bernie**
Black noise	**lp.phhh~**
Brown noise	**lp.pfff, lp.pfff~**
Brownian motion	**lp.pfff, lp.pfff~**
Cartesion to Polar coordinates	**lp.c2p~**
Cauchy distribution	**lp.coshy**
Chi-Square distribution	**lp.chichi**
Clipping values to range	**lp.scampf, lp.scampi, lp.scamp~**
Coin tosses	**lp.bernie, lp.ginger**
Conversion	**lp.c2p~, lp.p2c~**
Count of events	**lp.stacey**
Dice	**lp.dicey**
Dust noise	**lp.ppp~**
Erlang distribution	**lp.gammer**
Exponential distribution	**lp.expo, lp.y, lp.gammer**
First Law of Laplace	**lp.expo**
Fisher distribution	**lp.fishie**
Floating-point interval mutation	**lp.vim**
Fractal noise	**lp.pfff, lp.pfff~**
Frequency-domain interval mutation	**lp.frim~**
Gamma distribution	**lp.gammer**
Gauss distribution	**lp.norm**
Gray noise	**lp.grrr, lp.grrr~**
Hyperbolic cosine distribution	**lp.hyppie**
I Ching	**lp.ginger, lp.kg, lp.cass**
Interval mutation	**lp.frim~, lp.tim~, lp.vim**
Kurtosis	**lp.stacey**
Laplace distribution	**lp.expo**
Limiting to range	**lp.scampf, lp.scampi, lp.scamp~**
Linear congruence	**lp.lili, lp.lll~**
Linear distribution	**lp.linnie**
Logistic distribution	**lp.loggie**
Log-normal distribution	**lp.lonnie**
Low frequency noise	**lp.frrr~**
Map values	**lp.expo, lp.hyppie, lp.linnie, lp.loggie, lp. scampf, lp.scampi, lp.scamp~, lp.kg**
Maximum	**lp.stacey**
McCartney Pink noise	**lp.zzz, lp.zzz~**
Mean	**lp.stacey**
Minimum	**lp.stacey**
Morphological mutation	**lp.frim~, lp.tim~, lp.vim**
Negative Cauchy distribution	**lp.coshy**
Negative exponential distribution	**lp.expo**

Noise	**lp.frrr~**, **lp.grrr~**, **lp.pfff~**, **lp.phhh~**, **lp.ppp~**, **lp.shhh**, **lp.shhh~**, **lp.sss~**, **lp.tata**, **lp.titi**, **lp.zzz~**
Normal distribution	**lp.norm**
Parametric linear congruence	**lp.lili**, **lp.lll~**
Phase unwrapping	**lp.grl~**
Pink Noise	**lp.sss**, **lp.zzz**, **lp.sss~**, **lp.zzz~**
Poisson distribution	**lp.pfishie**
Polar to Cartesian coordinates	**lp.p2c~**
Popcorn noise	**lp.ppp~**
Positive Cauchy distribution	**lp.coshy**
Random walk	**lp.pfff**, **lp.pfff~**
Range limiting	**lp. scampf**, **lp.scampi**, **lp.scamp~**
Rayleigh distribution	**lp.y**
Reflecting values into range	**lp.scampf**, **lp.scampi**, **lp.scamp~**
Sample-and-hold noise	**lp.frrr~**
Scale values	**lp. scampf**, **lp.scampi**
Skew (statistical)	**lp.stacey**
Spectral mutation	**lp.frim~**
Standard deviation	**lp.stacey**
Statistics	**lp.stacey**
Student's "T" distribution	**lp.stu**
"T" distribution	**lp.stu**
Tausworthe 88 random number algorithm	**lp.tata**
Time domain mutation	**lp.tim~**
Triangular distribution	**lp.linnie**
TT800 random number algorithm	**lp.titi**
Uniform distribution	**lp.shhh**, **lp.shhh~**, **lp.tata**, **lp.titi**
Urn model	**lp.ernie**
Voss/Gardner algorithm	**lp.sss**, **lp.sss~**
Voss/McCartney algorithm	**lp.zzz**, **lp.zzz~**
Weibull distribution	**lp.y**
White noise	**lp.titi**, **lp.shhh**, **lp.shhh~**
Wrapping values into range	**lp. scampf**, **lp.scampi**, **lp.scampi~**

The beta distribution generates random numbers in the range $0 < x < 1$. It has two parameters, *a* and *b*. These parameters are sometimes referred to in the literature as α and β or ν and τ. The parameters control the shape of the distribution. Loosely speaking, values of *a* closer to zero increase the probability of small deviates (i.e., random values less than 0.5) being generated; values of *b* closer to zero increase the probability of large deviates (i.e., random values larger than 0.5).

The arc sine distribution is a special case of the beta distribution, with $a = b = 0.5$.

Note that if both parameters are set to one, the beta distribution degenerates to a uniform distribution. Both parameters, by definition, must be greater than zero. The values zero and one will be generated by **lp.abbie** when invalid parameter values are set.

Input

bang — Generate a random number from a beta distribution and send it out the outlet.

float — In the middle inlet: set the *a* parameter

In the right inlet: set the *b* parameter

seed — The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

You can initialize an **lp.abbie** object with up to three optional arguments. You must specify the first argument if you want to specify the second and you must specify the second to specify the third. The arguments, in order, are:

float — Two floating-point arguments specify initial values for the parameters *a* and *b*. Both parameters default to 0.5 (i.e., **lp.abbie** defaults to the arc-sine distribution).

int — Specify a seed for the core random number generator. The generator is auto-seeded if this value is 0 (the default).

Output

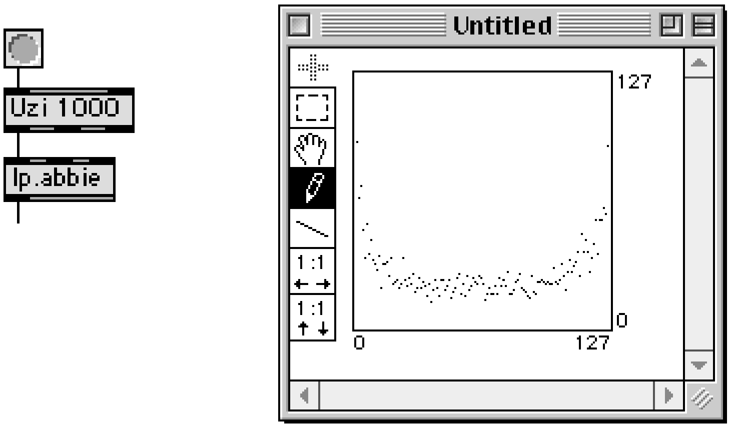
float — A random value from a beta distribution.

lp.abbie

Pro Bundles

Generate random numbers from beta and arc sine distributions

Examples



Generate random values from an arc sine distribution

What's in a name?

"Ah" is for arc sine; "bee" is for beta. Put them together and what do you get?

See Also

lp.tata Generate random numbers using the Tausworthe 88 algorithm
lp.scampi Scale, offset, and limit numbers; output integers

Cheng, Russel C. H., "Generating Beta Variates with Non-Integral Shape Parameters," *Communications of the ACM* 21 (1978): 317-322.

The Bernoulli distribution is based on a model of n independent trials, each of which has a probability p of succeeding. The result of a Bernoulli test is the number of successful trials, which must lie in the range $0 \le x \le n$.

Input

bang	Generate a random number from a Bernoulli distribution and send it out the outlet.
int	In the middle inlet: set n, the number of trials in a Bernoulli test. Negative values are invalid and treated as zero.
float	In the right inlet: set p, the probability of any single trial "succeeding." This should be a number in the range $0 \le p \le 1$. Negative values are treated as zero; positive values outside the valid range are treated as one.
seed	The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

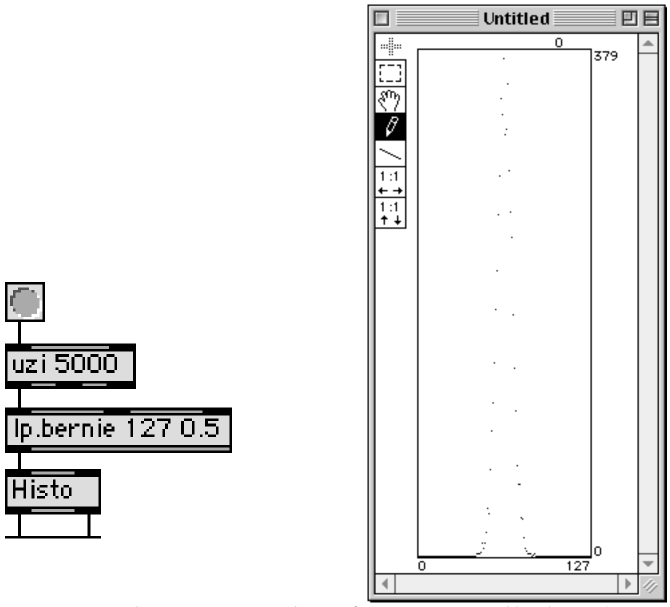
You can initialize an **lp.bernie** object with up to three optional arguments. You must specify the first argument if you want to specify the second and you must specify the second to specify the third. The arguments, in order, are:

int	Specify an initial value for n. Negative values are ignored. The default value is one.
float	Optional value in the range $0 \le x \le 1$. Set the initial value for the parameter p. Invalid values are ignored. The default value is 0.5
int	Optional. Specify a seed for the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

int	A random value from a Bernoulli distribution.

Examples



Generate random MIDI values from Bernoulli distribution

What's in a name?

This is one of the (relatively few) names retained from the original Litter package.

See Also

lp.pfishie	Generate random numbers from a Poisson distribution
lp.dicey	Throw dice
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

Convert Cartesian to Polar coordinates

lp.c2p~

All Bundles

This object was developed prior to the availability of the **cartopol~** object in MSP version 2. It is retained in the Litter Power package to allow older Patchers that required this object to run unaltered and for users of older MSP versions. Conveniently, the interfaces of **lp.c2p~** and **cartopol~** are identical.

Input

signal — In left inlet: The real component of a frequency domain signal.

In right inlet: The imaginary component of a frequency domain signal.

Arguments

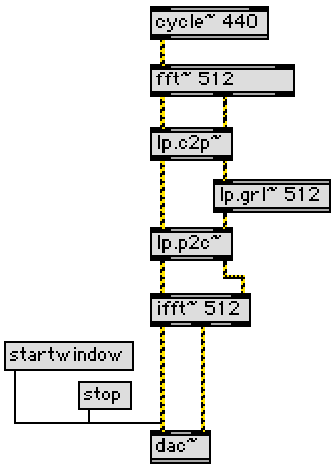
None.

Output

signal — Left outlet: The magnitude (i.e., amplitude) component of the polar representation equivalent to the incoming signal pair.

Right outlet: The phase (in radians) of the polar representation equivalent to the incoming signal pair.

Examples



See Also

fft~	Fast Fourier transform
ifft~	Inverse fast Fourier transform
lp.p2c~	Convert polar to Cartesian coordinates
lp.grl~	Phase unwrapping

lp.chichi

Generate random numbers from a chi-square distribution

The chi-square (χ^2) distribution has one parameter, *f*, the degrees of freedom. It is defined as the sum of *f* squared uniform deviates and hence all values from a χ^2 distribution are greater than or equal to zero. The mean of the χ^2 distribution converges to *f* and its standard deviation is $\sqrt{2f}$.

The chi-square distribution is frequently used in statistical tests.

Input

bang — Generate a random number from a χ^2 distribution and send it out the outlet.

int — In the right inlet: set the degrees of freedom parameter, *f*.

seed — The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

You can initialize an **lp.chichi** object with up to two optional integer arguments. You must specify the first argument if you want to specify the second.

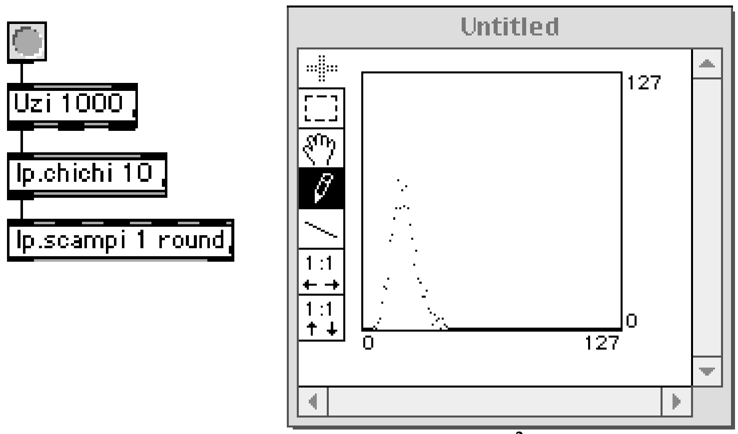
int — The first argument specifies an initial value of the degrees of freedom parameter, *f*. The default value is one.

The second argument specifies a seed for the core random number generator. The generator is auto-seeded if this value is 0 (the default).

Output

float — A random value from a χ^2-distribution.

Examples



Generating random numbers with a χ^2 distribution

What's in a name?

Think of "chichi" as "chi times chi."

See Also

lp.fishie	Generate random numbers from a Fisher distribution
lp.shhh	Generate random numbers from a “white” distribution
lp.stu	Generate random numbers from Student’s *t* distribution
lp.shhh	Generate random numbers from a “white” distribution
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

The Cauchy distribution has a single parameter, τ. The standard Cauchy distribution is symmetrical. In some literature reference is made to a positive Cauchy distribution, and the **lp.coshy** object can optionally produce this variant. Although not found in the literature, **lp.coshy** supports a negative variant for the sake of symmetry to the positive form.

Although the standard Cauchy distribution is symmetrical around zero, its mean does *not* converge but is undefined. Sooner of later one gets used to this.

Input

bang	Generate a random number from a Cauchy distribution and send it out the outlet.
float	In right inlet: sets the value of the parameter τ. Although the Cauchy distribution is only defined for positive values of τ, negative values and zero are allowed by the **lp.coshy** object. Negative values invert the sign of the resulting deviate. This has little impact on the (symmetrical) Cauchy distribution but note that this inverts the sign of the positive and negative variants. When τ is zero, the distribution degenerates to the constant zero.
sym pos neg	These messages specify which variant of the Cauchy distribution to use. The sym message causes standard (symmetrical) Cauchy-distributed deviates to be generated, the pos message causes positive Cauchy-distributed numbers to be generated, and the neg message causes negative Cauchy-distributed numbers to be generated.
seed	The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

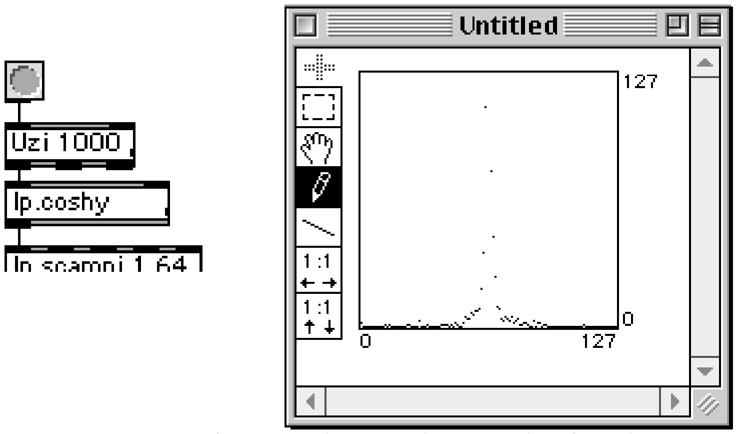
You can initialize an **lp.coshy** object with up to three optional arguments. You must specify the first argument if you want to specify the second and you must specify the second to specify the third. The arguments, in order, are:

float	Specify an initial value for τ. The default is one.
sym pos neg	Any of these specifies which variant of the Cauchy distribution to use. The default is sym.
int	Specify a seed for the core random number generator. The generator is auto-seeded if this value is zero (the default).

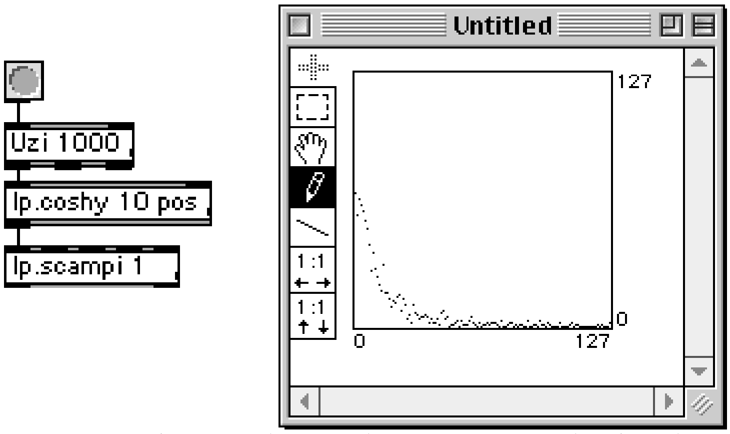
Output

float	A random value from a Cauchy distribution.

Examples



Generating random numbers with a Cauchy distribution



Generating random numbers with a positive Cauchy distribution

What's in a name?

I'm told some people pronounce it that way.

See Also

lp.tata Generate random numbers using the Tausworthe 88 algorithm

lp.dicey

Pro Bundles

Throw any number of "dice" with any number of faces.

Given n dice, each having f faces numbered from one to f, the resulting value will be in the range $1 \le x \le nf$.

If n is sufficiently large, the distribution will approaches a Gaussian distribution with a mean value of $\frac{nf+1}{2}$.

Input

bang — Throw the dice (i.e., choose one face from each die at random), add up the dots, and send the result out the outlet.

int — In the middle inlet: set the number of dice, n. This value should be positive. If n is zero, the distribution degenerates to a constant zero. Negative values of n are treated as zero

In the right inlet: set the number of faces for each die. This value should be two or greater, otherwise the distribution will degenerate to a constant equal to the number of dice. (This is equivalent to having single-faced dice, something of a topological nightmare.)
Note that using two-faced dice (in other words, a coin) is the equivalent of performing a Bernoulli test with a probability of 0.5 for "success."

seed — The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

You can initialize an **lp.dicey** object with up to three optional arguments. You must specify the first argument if you want to specify the second and you must specify the second to specify the third. The arguments, in order, are:

int — The first argument specifies an initial value for the number of dice. Negative values and zero are ignored. The default value is two.

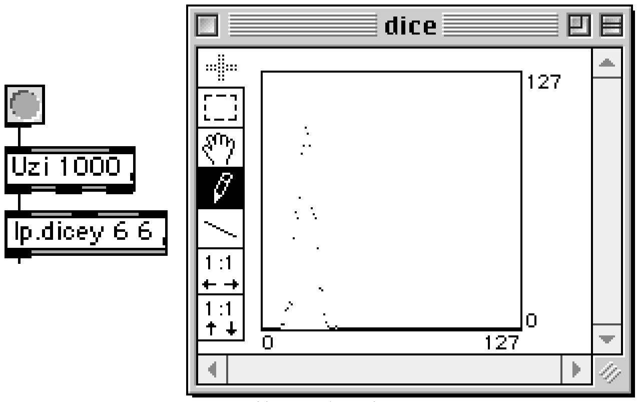
The second argument specifies an initial number of faces for each die. Negative values and zero are ignored. The default value is six.

The third argument sets the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

int — The sum of the dots on the faces chosen from the individual dice.

Examples



Rolling the dice

See Also

lp.bernie	Generate random numbers from a Bernoulli distribution
lp.norm	Generate random numbers from a normal (“Gaussian”) distribution
lp.ernie	Select items from an urn (“Finite urn” probability model)
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

The **lp.ernie** object implements a very flexible general-purpose “finite urn” model.

The typical finite urn model deals with colored balls in an urn. For instance, there might be three red balls, two black balls, and four white balls. Balls are removed one at a time. They are not replaced after removal. As balls are removed, the probabilities of picking each color change. The characteristics of the urn model are the concern with the changing probabilities and the fact that the total number of balls is known.

Instead of colors, an **lp.ernie** object deals “balls” numbered starting at zero. The model described above might be represented in **lp.ernie** by three balls with the value 0, two balls with the value 1, and four balls with the value 2. When an **lp.ernie** object receives a bang message, a ball is taken at random from the urn and its value is sent out the outlet. The ball is not returned to the urn until all balls have been used up (that is, the urn automatically refills itself when it has been emptied). In a sense, the standard **table** object can be used for implementing an infinite urn model; **lp.ernie** implements the finite urn model in an analogous manner.

Typically, the distribution of the balls in the urn will be read in from a **table** object using the refer message, but there are other messages for controlling the state of an **lp.ernie** object.

Input

bang	Remove a ball from the urn and send the ball’s value out the outlet. If the urn is empty it will automatically reset (cf. the reset message) and a bang is sent out the right outlet.
refer	The symbol refer followed by the name of a **table** object will cause **lp.ernie** to read values from the named table. The value in the table for zero will determine how many balls numbered zero are in the urn, and so on for each kind of ball. Typically, the table size will be the same as the size of the **lp.ernie** object. If the table size is smaller than the **lp.ernie** object, the ball types not defined will all be set to zero. If the table size is larger, the table values higher than the last kind of ball will be ignored.
set	The symbol set, followed by a list of numeric values, sets the number of each kind of ball in the urn. The first numeric value defines the number of balls of type zero, the second numeric value defines the number of balls of type one, and so on. Any symbols interspersed in the list will be interpreted as zero.
clear zero	The symbol clear empties the urn: the count of every king of ball is set to zero. The symbol zero is a synonym for clear.
const	The symbol const followed an integer will set the number of every kind of ball to the value specified. If no integer is explicitly specified, the default value zero is used (thereby providing yet another synonym for the clear message).
reset	Refills the urn. That is, the **lp.ernie** object is returned to the state defined by the last refer, set, clear, or const message.

size — The symbol size followed by an integer sets the number of different kinds of balls. It does not change the counts of the kinds of ball remaining in the urn. If the number of different kinds of ball is increased (that is, the size parameter is greater than before), the new kinds of ball are set to zero count.

seed — The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

You can initialize an **lp.ernie** object with up to two optional arguments. You must specify the first argument if you want to specify the second. The arguments, in order, are:

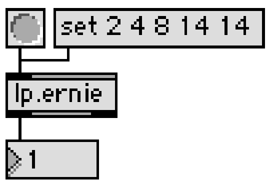
int — The first argument specifies the "size" of the urn (that is, how many different kinds of ball to use). The default value is 128, following the MIDI-oriented convention of the **table** object.

The second argument sets the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

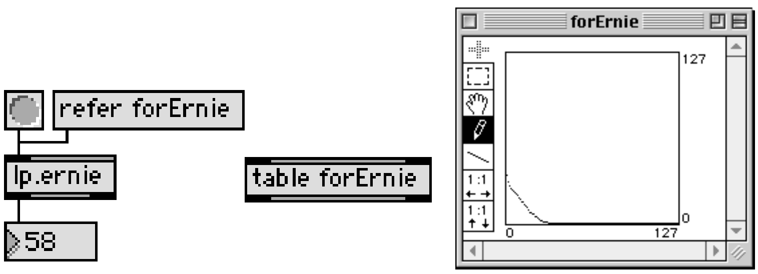
Output

int — The value of a ball chosen at random from the urn. The ball is not returned to the urn until all balls have been used up.

Examples



Set distribution and draw balls at random



Reading values for Ernie from a table

What's in a name?

A deliberate misspelling.

See Also

table	Store and graphically edit an array of numbers
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

The exponential distribution is typically used to model waiting times between Poisson-distributed events. It has one parameter, the mean time between events, generally named λ. In the literature the distribution is sometimes described in terms of mean density of events (i.e., 1/λ) and the parameter may be named δ or τ. The density is what is actually used while calculating the distribution, but to maintain consistency with the interface to the **lp.pfishie** object, **lp.expo** object interprets its parameter as the mean. There is a tau message to allow direct specification of the density.

The exponential distribution only produces non-negative values. A variant form, known variously as the "bilateral exponential" or "Laplace" distribution, generates a distribution symmetrical around zero. Both of the bilateral and standard exponential distributions can be generated with **lp.expo**. Out of symmetry to the standard "positive" variant, a negative variant can also be generated.

Input

bang — Generate a random number from an exponential distribution.

float — In the left inlet: A value in the range 0 < x ≤ 1 will be transformed using the formula

$$f(x) = \frac{-1}{\lambda}\log(x)$$ for the symmetry option pos,

$$f(x) = \frac{1}{\lambda}\log(x)$$ for the symmetry option neg, and

$$f(x) = \begin{cases} \frac{-1}{\lambda}\log(2x) & if\, 0 < x \leq 0.5 \\ \frac{1}{\lambda}\log(2x-1) & if\, 0.5 < x \leq 1.0 \end{cases}$$ for the symmetry option sym.

Assuming the input values are uniformly distributed, the output values will be exponentially (or bilaterally) distributed according to the current value for λ.

In the right inlet: specify the mean value (λ). Note that in the case of the bilateral distribution, the mean is always 0, regardless of the current value of λ. However, λ still determines the variance ("spread") of the distribution.

sym
pos
neg — These symbols set the symmetry option. The symbol pos causes **lp.expo** to produce deviates with the standard exponential distribution. The symbol sym generates a bilateral exponential ("Laplace") distribution. The symbol neg produces the negative reflection of the standard exponential distribution.

tau — The symbol tau followed by a floating point value allows you to directly specify the distribution density. Sending the tau message with a value of 1/λ is equivalent to sending the value λ to the right inlet.

seed — The symbol seed followed by an integer reseeds the internal random number generator.

lp.expo

Pro Bundles

Generate random numbers from an exponential distribution

Arguments

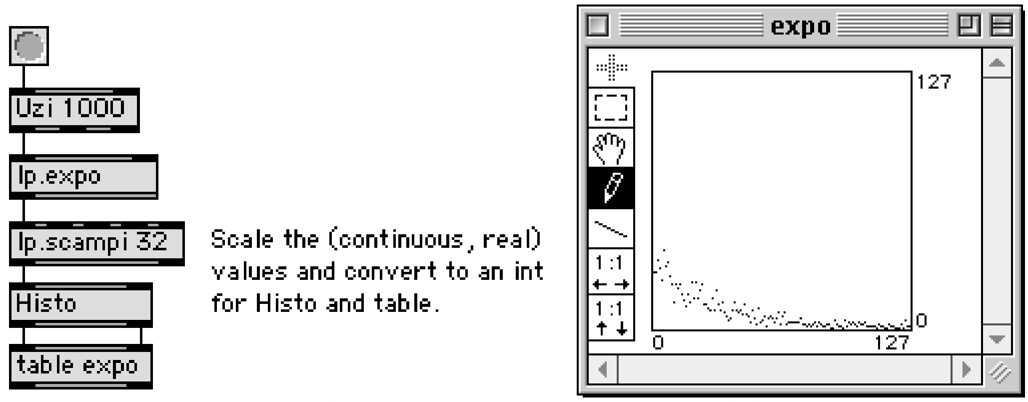
You can initialize an **lp.expo** object with up to three optional arguments. You must specify the first argument if you want to specify if you want to specify the second and you must specify the second to specify the third. The arguments, in order, are:

float	Specify an initial value for the mean value (λ) of the distribution. The default is one.
sym pos neg	Any of these symbols will specify an initial value for the symmetry option. The default value is pos.
int	Set the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

float	A random value from an exponential distribution.

Examples



Generating random numbers with an exponential distribution

See Also

lp.pfishie	Generate random numbers from a Poisson distribution
lp.gammer	Generate random numbers from Gamma and Erlang distributions
lp.hyppie	Generate random numbers from a hyperbolic cosine distribution
lp.linnie	Generate random numbers from linear and triangular distributions
lp.loggie	Generate random numbers from a logistic distribution
lp.lonnie	Generate random numbers from a log-normal distribution
lp.ppp~	Popcorn (dust) noise
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

The Fisher distribution has two “degrees of freedom” parameters, normally designated f_1 and f_2. The distribution is generated by dividing values taken from two independent chi-square distributions and can take on arbitrary floating point values.

The distribution is commonly used in statistical tests.

Input

bang — Generate a random number from a Fisher distribution and send it out the outlet.

int — In the middle inlet: Set the value of the f_1 parameter.

In the right inlet: Set the value of the f_2 parameter.

seed — The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

You can initialize an **lp.fishie** object with up to three optional integer arguments. You must specify the first argument if you want to specify the second and you must specify the second to specify the third. The arguments, in order, are:

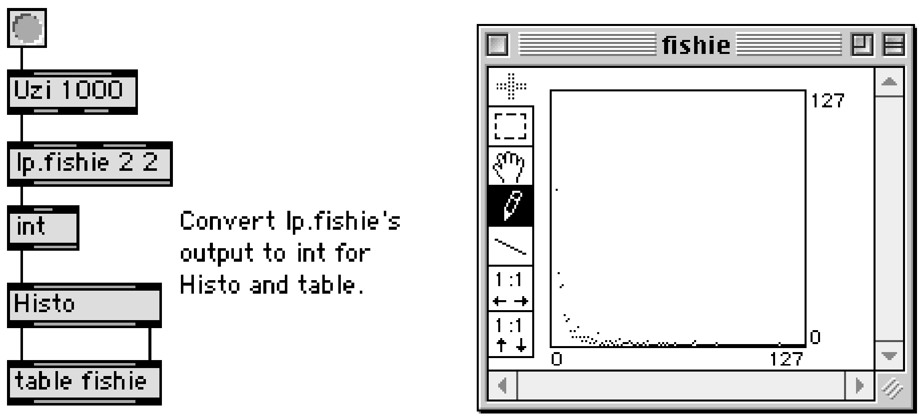
int — The first and second integers specify initial values for the f_1 and f_2 parameters, respectively. The default value for both is 1.

A third integer sets the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

float — A random value from a Fisher distribution.

Examples



Generating random numbers with a Fisher(2, 2) distribution

What's in a name?

This one seems sort of obvious.

See Also

lp.chichi	Benerate random numbers from a chi-sqaure distribution
lp.stu	Generate random numbers from Student's *t* distribution
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

The Frequency-domain Interval Mutator is a variant of the Time-domain Interval Mutator (**lp.tim~**) object, modified in two ways to better handle frequency-domain signals (for instance, signals produced by the **fft~** object).

First, the **fft~** object produces two signals, the real and imaginary components of a Fourier Transform, that need to be handled in parallel. While this could be handled adequately for uniform mutations with two **lp.tim~** objects, with irregular mutations it is normally desirable to synchronize the mutation of both components. So, **lp.frim~** provides pairs of inlets for Source and Target signals, and all mutations are performed equally on both components.

Second, when performing any mutation on relative intervals, the **lp.tim~** object calculates intervals between successive samples from the Source and Target signals. But the interval between adjacent samples of a signal output by **fft~** represent values of adjacent bins in the same FFT sample frame, whereas the interval that is normally desired is the interval from bin-to-bin across sample frames. In other words, a **lp.tim~** object would calculate the interval function Δ(S[frame, bin], S[frame, bin-1]), whereas what is wanted is Δ(S[frame, bin], S[frame-1, bin]). In terms of MSP signals, this means that one needs to specify a distance between interval samples equal to the sample size parameter used by the .**fft~** object generating the FFT signals to be mutated. The **lp.frim~** object allows you to specify this distance as an initialization parameter.

Note that, unlike the **fft~/ifft~** objects, **lp.frim~** makes no automatic corrections for Sample Size arguments; it is quite possible that you may wish to experiment with effects produced by “non-standard” values. Note also that if no interval distance argument is specified, the interval distance defaults to zero, which indicates a “hard-wired” use of absolute intervals (that is, you will not be able to switch to relative intervals).

Finally, those familiar with the implementation of Spectral Mutation as found in SoundHack should be aware that **fft~** represents the spectral signal using complex Cartesian coordinates (real and imaginary pairs) as opposed to a representation of amplitude and phase, used most spectral processing software,. The **lp.c2p~** and **lp.p2c~** objects, included in the Litter package, perform this conversion, additionally the **lp.grl~** object will perform the phase unwrapping typically calculated as part of the Fourier Transform. The effect of most mutations can be quite different with these two different representations. Irregular mutations on absolute intervals should be identical between both representations.

Input

signal — In 1st Inlet and 2nd Inlets, the mutation source (real and imaginary components respectively. Both signals are mandatory if you want anything to happen).
In 3rd and 4th Inlets, the mutation target (real and imaginary components respectively; both signals mandatory if you want anything to happen)
In 5th Inlet, a time-varying Mutation Index (defaults to float input or object argument if no signal). Mutation Index is limited to the range $0 \leq \Omega \leq 1$.
In 6th Inlet, a time-varying Delta Emphasis value (defaults to float input or object argument if no signal; ignored if the object is using absolute intervals). Delta Emphasis is limited to the range $-1 \leq \delta \leq 1$.
In 7th Inlet, a time-varying Clumping Factor (defaults to float input or object argument if no signal; ignored if the object is performing a uniform mutation). Clumping Factor is limited to $0 \leq \pi < 1$. For practical purposes in this implementation, the maximal value for π is clipped to 0.9990234375, which means that you can expect an irregular mutation with a mutation index of 0.5 to change state between mutated and non-mutated forms about once every thousand samples or so.)

float — In 5th Inlet: sets the Mutation Index (but is overridden if a signal is present).
In 6th Inlet: sets the Delta Emphasis This value is overridden when a signal is present and ignored if the object is using absolute intervals.
In 7th Inlet, sets the Clumping Factor. This value is overridden when a signal is present and ignored if the object is using absolute intervals.
Sending a float to any of the first four inlets elicits an error message in the Max window.

usim
isim
uuim
iuim
wcm
lcm — Set the mutation algorithm to Linear Contour Modulation, Uniform Signed Interval Modulation, etc.

rel — Use relative intervals for calculating the mutant. This is the default setting. You can include a float with this message to set Delta Emphasis (the default value is zero).
If the **lp.frim~** object was initialized with no Interval Distance argument you can not use Relative Interval.

abs — Use absolute intervals for calculating the mutant.

Note that, unlike interval mutation in SoundHack and other implementations, the **lp.frim~** object does not support source and target reference values. If you want source or target intervals to be calculated against a reference other than zero, you need to send the signals through **+~**, ***~**, or other objects to suit your needs. This gives you greater flexibility and control than anything **lp.frim~** could offer.

obands The obands message causes frequency bins belonging to the same interval band to be treated as a unit during irregular mutations.

The obands message takes an optional integer parameter in the range $0 \leq b \leq 15$. The parameter indicates how many divisions of the octave are to be treated as a band. The value three produces third-octave bands. The default value is zero, which indicates that each frequency bin is mutated independently.

clear Resets the stored values of previous source, previous target, and previous mutant to zero. This is often helpful after a mutation has gotten chaotic.

Arguments

symbol The symbols usim, isim, uuim, iuim, wcm, and lcm can be used to specify the initial mutation algorithm to use. The default is usim.

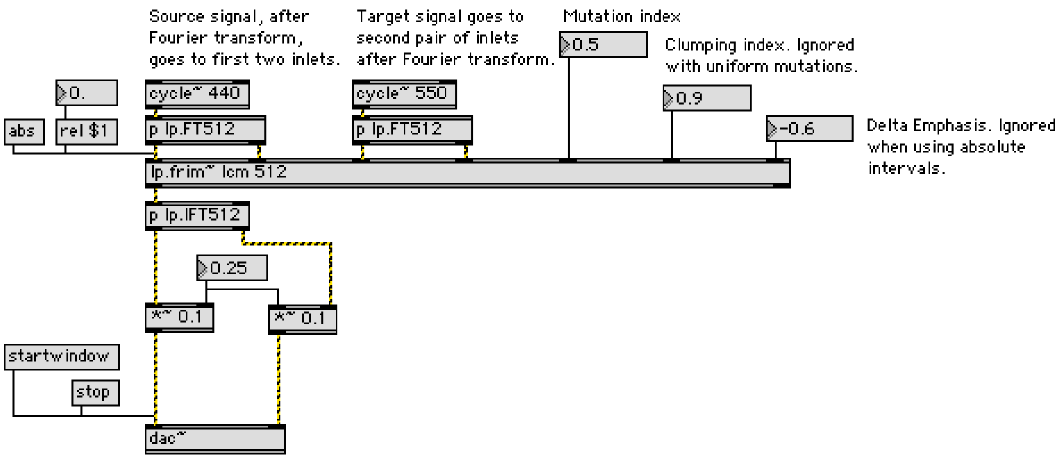
int Interval Distance to use when calculating relative intervals. Normally you would set this to the same value as the Sample Size in the **fft~** objects generating the Source and Mutant signals. Unlike the **fft~** objects, however, **lp.frim~** does not "auto-correct" values to the closest power of 2, allowing you to experiment with unusual interval distances. Like **fft~**, a maximum setting of 2048 is enforced.
Note that **lp.frim~** requires over twice as much memory as an **fft~** object using the same sample size. A complete set of two **fft~**s, a **lp.frim~**, and an **ifft~** with all objects set to 2048 will require memory allocation of over 80 kB. Aren't you glad that memory is cheaper than it used to be?

float Up to three float arguments can be included to specify (in order) Mutation Index (Ω), Delta Emphasis (this is ignored when absolute intervals are used), and Clumping Factor (this is only used by irregular mutations). All default to 0.0.

Output

signal Mutant signals out of the left and middle outlets. The signal in the left outlet is the mutant of the 1st and 3rd inlets; the signal in the middle outlet is the mutant of the 2nd and 3rd inlets.

Examples



Using lp.frim~ to perform spectral mutation.
Fast Fourier Transform with Cartesian-to-polar coordinate conversion and phase unwrapping is encapsulated in the patchers lp.FT512; conversion back to Cartesian coordinates and inverse FFT is encapsulated in the lp.IFT512 patcher.

What's in a name?

Abbreviation for **FR**equency domain **I**nterval **M**utation

See Also

lp.c2p~	Convert Cartesian to Polar coordinates
lp.grl~	Phase unwrapping
lp.p2c~	Convert polar to Cartesian coordinates
lp.tim~	Time domain interval mutation
lp.vim	Interval mutation of numeric values

Polansky, Larry, "Morphological Metrics: An Introduction to a Theory of Formal Distances" (paper presented at the International Computer Music Conference, Champaign-Urbana, 1987), 197-204.

Polansky, Larry and Tom Erbe, "Spectral Mutation in *SoundHack*: A Brief Description" (paper presented at the International Computer Music Conference, Banff, Canada, 1995), 307-314.

Polansky, Larry, "Morphological metrics," *Journal of New Music Research (formally Interface)* 25 (1996): 289-368.

Low-frequency noise is generated from a sequence of random values chosen at a constant rate slower than the sampling rate. In its simplest form, it functions as a noise generator passed through a sample-and-hold module. However, **lp.frrr~** also allows the samples between the randomly generated values to be interpolated, either linearly or quadratically.

The rate at which random values are generated is specified in Hz; **lp.frrr~** adjusts the actual rate of generation to be an integral number of samples.

The **lp.frrr~** object can also be used to good effect for control signals.

Input

signal — Signal processing provided for the benefit of **begin~** / **selector~** configurations.

float — In left inlet: set the base frequency. Note that the actual frequency used may be adjusted by **lp.frrr~** to match an integral sub-harmonic of the sampling rate.

int — In right inlet: zero, one, or two. Zero indicates no interpolation between generated values, one indicates linear interpolation, and a value of two indicates quadratic interpolation. Negative values are treated as zero; values larger than two are treated as two.

Arguments

You can initialize an **lp.frrr~** object with up to two optional arguments. You must specify the first argument if you want to specify the second. The arguments, in order, are:

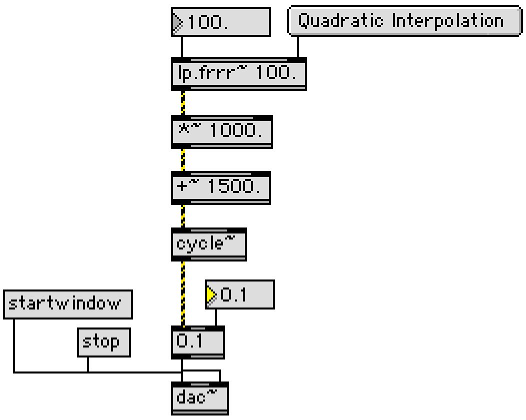
float — The first argument specifies an (approximate) initial setting in Hz for the base frequency at which new random values are generated. The default value is 100.

int — The second argument specifies an initial value for interpolation, which should be either zero, one, or two. The default value is zero (no interpolation).

Output

signal — Low frequency noise.

Examples



Using low-frequency noise as a control signal.

What's in a name?

Low-**FRRR**equency noise.

See Also

lp.grrr~	"Gray" noise
lp.lll~	Parametric linear congruence "noise"
lp.pfff~	"Brownian" ($1/f^2$) noise
lp.ppp~	Popcorn (dust) noise
lp.shhh~	White noise
lp.sss~	"Pink" noise (Voss/Gardner algorithm)
lp.zzz~	"Pink" noise (McCartney algorithm)
noise~	Another source of noise
pink~	Another source of pink noise

The Gamma distribution has two parameters, generally referred to as *order* and *location*. It produces an asymmetrical distribution of positive random values, and is often used in musical contexts for generating rhythms. The order parameter must be positive, the location parameter may be zero or positive.

The Erlang distribution is a special case of the Gamma function when the order parameter is an integer. This distribution is modeled on a process consisting of several independent exponentially-distributed sub-processes. In this case, the order parameter indicates the number of sub-processes. An Erlang distribution with order of one is equivalent to an exponential distribution.

Input

bang — Generate a random value from a Gamma or Erlang distribution.

float — In the middle inlet: Set the value of the order parameter

In the right inlet: Set the value of the location parameter.

seed — The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

You can initialize an **lp.gammer** object with up to three optional arguments. You must specify the first argument if you want to specify the second and you must specify the second to specify the third. The arguments, in order, are:

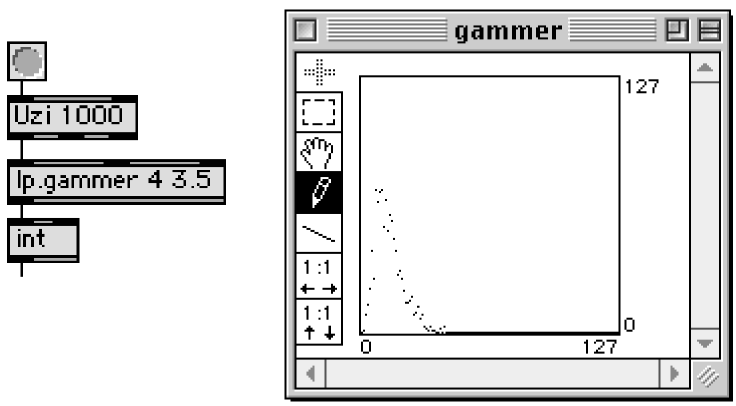
float — The first two arguments specify initial values for the order and location parameters (respectively). Both parameters default to one.

int — Set the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

float — A random value from a Gamma distribution.

Examples



Generating random numbers with a Γ(4, 3.5) distribution

What's in a name?

Portmanteau word derived from the first syllables of the two distributions implemented.

See Also

lp.expo	Generate random numbers from an exponential distribution
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

Ahrens, Joachim H. and Ulrich Dieter, "Generating Gamma Variates by a Modified Rejection Technique," *Communications of the ACM* 25, no. 1 (1982): 47-54.

Choose numbers in the range $1 \leq x \leq 64$ using the methods from the *I Ching*, the *Book of Changes*.

Traditionally, there are two different methods for consulting the *I Ching*. The more commonly used one consists of tossing three coins six times, with each toss of three coins determining the value of a "line", which may either be yang (unbroken) or yin (broken). The six tosses generate six lines that, taken together, form a hexagram. Depending on how the coins fall, each line may be either stable or instable. Instable lines change their value between the present and the future, thereby resulting in two hexagrams.

The older method of consulting the *I Ching* consists of throwing yarrow sticks to divide them into two piles. The number of sticks in each pile determines the value of a single line (yin or yang, stable or instable).

The **lp.ginger** object allows you to use either of these two methods. Those concerned with mysticism will be relieved to know that **lp.ginger** follows the instructions stated in the *I Ching* as far as possible, including such details as having 50 yarrow sticks at its disposal but only throwing 49 of them. The **lp.ginger** object does everything except burn incense for you (and we're working on that).

It is worth noting that the method modeled on throwing yarrow sticks does *not* result in a flat distribution. In particular, there are some striking correlations between present and future values. This is as it should be.

The **lp.ginger** object sends both present and future oracles as numbers out the left and middle outlets, respectively. It also sends a list indicating exactly how the coins fell (or how the yarrow sticks were divided) out the right outlet.

Input

bang — Consult the *I Ching*. Coins will be tossed or yarrow sticks thrown, divided into piles, and counted. The resulting values of the hexagrams are determined and sent out the two left outlets. Additional details are sent out as lists through the right two outlets.

A bang message should typically be preceded by meditating on the question you wish to have answered, but this is optional.

coin
yarrow — Generate a new oracle using the method specified. The coin message causes the method based on tossing coins to be used; the yarrow message causes the method based on throwing yarrow sticks to be used. The method specified becomes the method to be used by bang messages.

set — The symbol set followed by either coin or yarrow determines the method to be used to consult the *I Ching*. No oracle is generated.

zen — You may use this message to meditate.

Arguments

symbol — Either of the symbols **coin** or yarrow may be used to specify the initial method to be used for generating oracles. This argument is optional; coins are tossed by default.

Output

int Left outlet: a number in the range from 1 to 64 indicating the hexagram determined by the six lines

Middle outlet: a number in the range from 1 to 64 indicating the hexagram determined by the six lines after any instable lines have changed from yin to yang (or vice versa).

list Right outlet: A list consisting of eighteen values. The individual values will be either two (representing yang) or three (representing yin). If the method of tossing coins is used, these will represent the individual coin tosses (the first three items representing the coins tossed to determine the first line, etc.). For yarrow sticks, this represents the result of counting the sticks in each pile after each throw of the sticks.

Examples

Dipslay oracle of the present and the future

What's in a name?

The transliteration of the Chinese for *Book of Changes*, used by Richard Wilhelm in his seminal translation, is *I Ging*.

See Also

lp.i Text of I Ching oracles
lp.kg Map I Ching values to non-standard ranges

Peter Elsea's **Lobjects** may be useful for processing the details of the oracle.

I Ging, trans. Richard Wilhelm (Munich: Eugen Diederichs, 1973).

I Ching or *Book of Changes*, trans. by Cary F. Baynes (from the German translation with commentaries by Richard Wilhelm). (Princeton, New Jersey: Princeton University Press, 1967).

Cage, John. *Silence*. (London: Marion Boyars, 1987).

This is a utility object, designed primarily for calculating the phase unwrapping typically performed as part of the Fourier Transform in spectral analysis. It could, conceivably, be used for other purposes.

Input

signal (Wrapped) phase.

float Sets the threshold value for unwrapping.

pi, π Special messages for setting the threshold to π or some multiple thereof. If followed by an integer, the threshold will be set to that multiple of π. (For instance, to set the threshold to 2π, send the message 'pi 2'.)

clear Clears the buffer of stored previous samples, setting them all to zero.

Arguments

int Distance between samples to compare before unwrapping; typically equal to the sample size used by the corresponding **fft~** object.

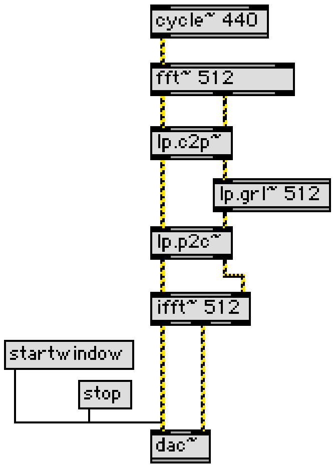
Note that, unlike the **fft~** object, **lp.grl~** does not automatically adjust this parameter to a legal FFT sample size. Furthermore, if no value is specified, this argument defaults to 1 (which, in terms of a Fourier Transform, would result in "unwrapping" phase comparing adjacent bins). The maximum value for this parameter is arbitrarily set to 2048. This is, perhaps conveniently, the maximum sample size supported by the fft~ object.

float Threshold allowed between pairs of samples. If the interval between a pair of samples (at the distance defined by the integer argument) is larger than this threshold, the signal is unwrapped (that is, the complementary value threshold -∂ is used). The default value is π.

Output

signal Unwrapped phase

Examples



Unwrapping phase while converting Cartesian to polar coordinates

What's in a name?

In honor of one of the great unwrappers of all time: **G**ypsy **R**ose **L**ee.

See Also

lp.c2p~	Convert Cartesian to polar coordinates
lp.p2c~	Convert polar to Cartesian coordinates
fft~	Fast Fourier transform
ifft~	Inverse fast Fourier transform

This is a control-domain version of the **lp.grrr~** signal generator. It generates values in the range $0 \leq x \leq 1$.

Input

bang	Generate a random value with "gray" distribution.
seed	The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

int	Set the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

float	A random value in the range $0 \leq x \leq 1$.

What's in a name?

See **lp.grrr~**.

See Also

lp.grrr~	"Gray" noise
lp.pfff	Generate random numbers from a $1/f^2$ ("Brownian") distribution
lp.shhh	Generate random numbers from a "white" distribution
lp.sss	Generate random numbers from a 1/f ("pink") distribution
lp.zzz	Generate random numbers from a 1/f ("pink") distribution
lp.scampi	Scale, offset, and limit numbers; output integers
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

lp.grrr~

Pro Bundles

"Gray" noise
(Signal)

Gray noise results from flipping random bits of an integer representation of the sample signal on a sample-to-sample basis. The spectrum is stronger towards lower frequencies.

Input

signal Signal inlet provided solely for the benefit of **begin~ / selector~** configurations.

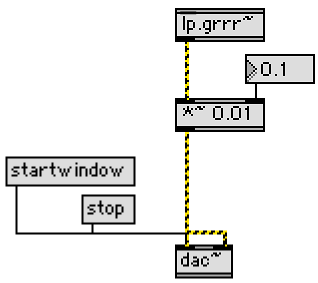
Arguments

None

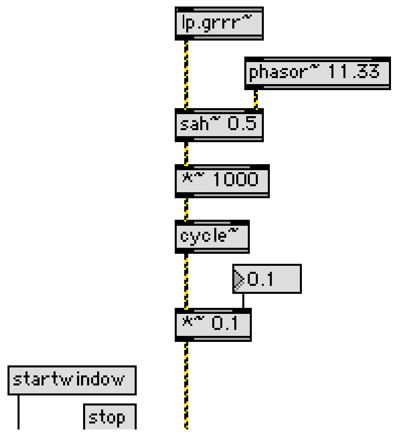
Output

signal Gray noise

Examples



Using lp.grrr~ as a noise source.



Using lp.grrr~ as a control signal

What's in a name?

GRRRay noise.

This particular "colored" noise seems to have been named not so much due to associations with light (as is the case of white or pink noise) but because of a certain similarity to the Gray Code.

See Also

lp.frr~	
lp.grrr	"Gray" noise (control domain)
lp.lll~	Parametric linear congruence "noise"
lp.pfff~	"Brownian" ($1/f^2$) noise
lp.ppp~	Popcorn (dust) noise

lp.shhh~	White noise
lp.sss~	“Pink” noise (Voss/Gardner algorithm)
lp.zzz~	“Pink” noise (McCartney algorithm)

Input

bang — Generate a random number from a hyperbolic cosine distribution and send it out the outlet.

float — A value in the range 0 < x < 1 will be transformed using the formula

$$f(x) = \log(\tan(x))$$

Values outside the range are ignored.

Assuming that input values are uniformly distributed, the output values will follow a hyperbolic cosine distribution.

seed — The symbol seed followed by an integer reseeds the internal random number generator.

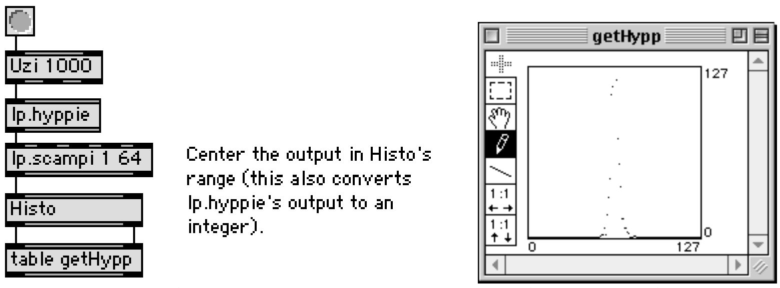
Arguments

int — Set the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

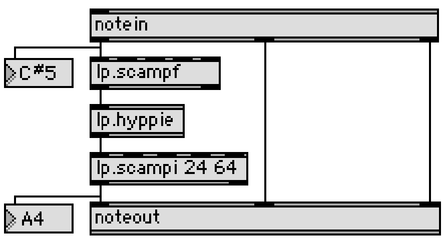
Output

float — A random value from a hyperbolic cosine distribution.

Examples



Generating random numbers with a hyperbolic cosine distribution



Mapping MIDI input.
Decide for yourself if this is more fun with local control on or off

What's in a name?

Hypperbolic cosine is hypp. Get it?

See Also

lp.linnie	Generate random numbers from linear and triangular distributions
lp.loggie	Generate random numbers from a logistic distribution
lp.lonnie	Generate random numbers from a log-normal distribution
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

The discussion presumes some familiarity with the use of *I Ching*. An in-depth introduction is beyond the scope of this document, please consult the bibliographic references in the **See Also** section for further background.

Input

bang — Generate a new fortune and send the texts of the oracle out the outlet as a sequence of symbols.

Note that if **lp.i** was initialized with explicit hexagram numbers as arguments, bang outputs text without generating a new fortune.

int — In left inlet: An integer between 1 and 64 will set a new value for the main hexagram and cause **lp.i** to send the texts of the fortune out the outlet. All other values are ignored.

In right inlet: An integer between 1 and 64 will set a new value for the future hexagram. All other values are ignored.

set — The symbol set followed by an integer changes the current value of the main hexagram without causing the text of the fortune to be output.

Optionally, a second integer may be included with the set message. If included, it will set the value of the future hexagram.

Values outside the range of 1 to 64 are ignored.

name — The symbol name causes a list of symbols for the name of the main hexagram to be sent out the outlet. Optionally, an integer may be included in the message, specifying which components of the name to include ("name options"). This integer is the sum of the following values:

1: Include hexagram number
2: Include name of hexagram in transliterated Chinese
4: Include name of hexagram in English

You can also use the value -1 to indicate all three components.

Each component is represented as a single symbol.

The name options will remain in effect until changed with another name message. The **lp.i** object will not allow you to suppress all name options; at least one component will always be generated.

trigrams

The symbol trigrams causes a sequence of lists of symbols to be sent out the outlet. These symbols describe the trigrams contained in the main hexagram. Optionally, an integer may be included in the message, specifying which trigrams to include (“trigram options”). This integer may take on the following values:

0: Suppress all trigrams
1: Output the main (top and bottom) trigrams
2: Output all trigrams (also the two middle trigrams)

You can also use the value -1 to indicate all trigrams.

Each trigram is represented as a list of three symbols. The first symbol indicates the position of the trigram (above, below, upper middle, lower middle). The second symbol is the name of the trigram (in transliterated Chinese). The third symbol is the meaning of the symbol (in English)

The trigram options will remain in effect until changed with another trigrams message.

judgement

The symbol judgement causes a sequence of symbols representing the judgement of the current main hexagram to be sent out the outlet. The first symbol is The Judgement. It is followed by symbols with the text of the judgement, one line of text per symbol.

image

The symbol image causes a sequence of symbols representing the image of the current main hexagram to be sent out the outlet. The first symbol is The Image. It is followed by symbols with the text of the image, one line of text per symbol.

lines The symbol lines causes a sequence of lists of symbols to be sent out the outlet. These symbols describe lines from the main hexagram. Optionally, an integer may be included in the message, specifying which lines to include ("line options"). This integer may take on the following values:

0: Suppress all lines
1: Output texts of changing lines only
2: Output texts of governing lines only
3: Output all lines

You can also use the value -1 to indicate all lines.

Each hexagram line is represented as a sequence of lists of symbols. Each hexagram line is introduced by a list of four symbols indicating the line position and its value. The first symbol indicates if the hexagram line is a ruling line: if the symbol contains the character • the line is a constituting ruler; if it contains the character ° the line is a governing ruler. The second symbol is the value of the hexagram line (either Six or Nine). The third symbol indicates which line is currently being described (at the beginning, in the second place, etc.). The final symbol is a static text (means:). This is followed by a sequence of symbols, one symbol per line of text from *I Ching*.

The line options will remain in effect until changed with another lines message.

Arguments

int Two optional ints, specifying initial values for the main and future hexagrams.

If no initialization arguments are specified, **lp.i** generates new hexagrams every time it receives a bang message.

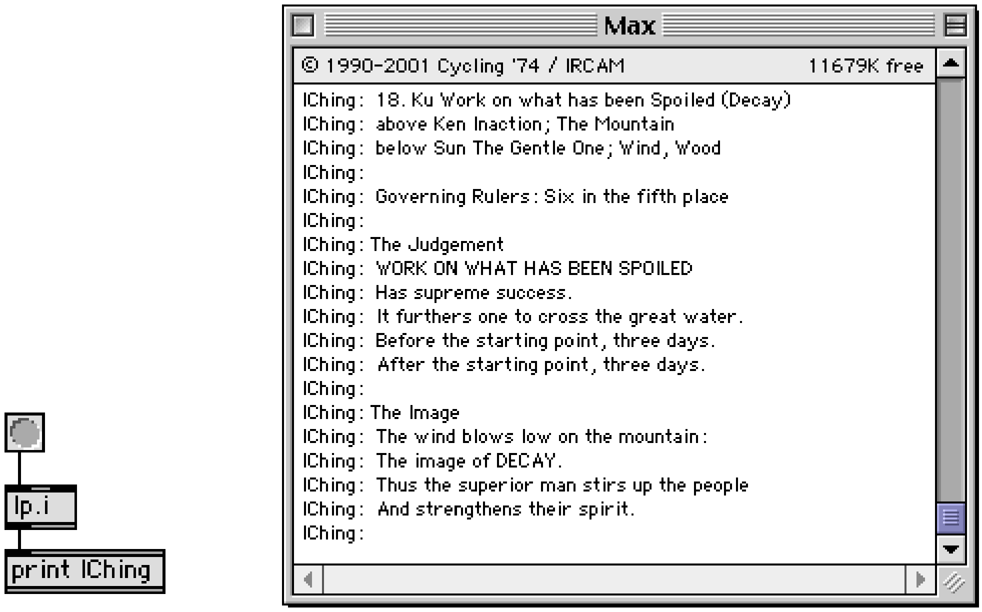
If one argument is specified, it sets an initial value for both the main and future hexagrams (that is, the oracle has no changing lines). Bang messages do not generate new hexagrams; you must explicitly send int or set messages to change the hexagram values.

If two arguments are specified, the first argument sets an initial value for the main hexagram and the second sets an initial value for the future hexagram. Bang messages do not generate new hexagrams; you must explicitly send int or set messages to change the hexagram values.

Output

Texts of the *I Ching*.

Examples



Consulting the I Ching

What's in a name?

This puts the "i" back into the *I Ching*.

See Also

lp.ginger I Ching

I Ging, trans. Richard Wilhelm (Munich: Eugen Diederichs, 1973).

I Ching or *Book of Changes*, trans. by Cary F. Baynes (from the German translation with commentaries by Richard Wilhelm). (Princeton, New Jersey: Princeton University Press, 1967).

Huang, Kerson and Rosemary Huang, *I Ching* (New York: Workman, 1987).

Jou, Tsung Hwa, *Tao of I Ching: Way to Divination* (Boston: Tuttle, 1985)

One common way to consult the *I Ching* when there are less than 64 alternatives is to ask the *I Ching* how much weight to assign the individual choices. The *I Ching* is consulted for each of the choices wanted, and the hexagram values chosen are assigned as weights to each choice. Then a range of hexagram values proportional to the weight of each choice will be assigned. Thereafter, when a hexagram number is chosen from the *I Ching*, it is mapped to the choice with the relevant range.

This process is computed for both “present” and “future” *I Ching* values.

In **lp.kg** the choices are numbered starting at one.

Input

int — In the left inlet: map this value and a pending “future” input value. Send the results out the two outlets. If there is no pending future input value, the present value is used for both the present and future mappings.

In the right inlet: set the value of the pending “future” input.

bang — Calculate a new set of mappings.

size — The symbol size, followed by positive integer less than 64, will reset the number of choices available and generate a new set of mappings. The values generated will range from one to the value specified in the size message.

Arguments

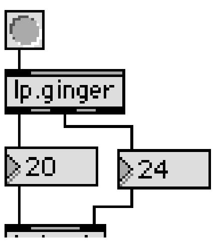
int — Specify an initial size (number of choices) for **lp.kg**’s mapping. If no argument is specified,

Output

int — Out the left outlet: mapping of the present input value.

Out the right outlet: mapping of the future input value.

Examples



Mapping I Ching decisions to the values one to four

What's in a name?

The technique described here was described by John Cage and used in works such as *HPSCHD*.

See Also

lp.ginger	I Ching
lp.i	Text of I Ching Oracles
lp.scampi	Scale, offset, and limit numbers; output integers

Austin, Larry. "An interview with John Cage and Lejaren Hiller." *Source* 4, no. 2 (1968): 11-19. Reprinted in *Computer Music Journal* 16(4), pp. 15-29, 1993.

Cage, John. *Silence*. (London: Marion Boyars, 1987).

The linear congruence algorithm has been the standard method of generating pseudo-random numbers since the late 1950s. More recent statistical literature (i.e., since about 1960) has pointed out numerous shortcomings with the algorithm. Despite this, linear congruence remains the method provided by practically all operating systems and programming libraries. Max is no exception.

With carefully chosen parameters, LC can produce sequences of numbers that at least appear random at first glance. However, even with the most carefully chosen parameters, LC shows a number of correlations that are not in any sense random. For this reason, the Litter Power Package uses more modern methods that are more measurably random and robust. The algorithm used by default in the Litter Power Package is faster, to boot.

The **lp.lili** object was created not to bury LC, but to investigate it. You can set the individual parameters of the formula:

$$x = x_{prev} \cdot f + a \bmod m$$

By adjusting the parameters x_{prev}, f, a, and m (referred to in the following as seed, mul, add, and mod, respectively), you can generate sequences of numbers of greater and lesser apparent randomness. The default values mirror the parameters used by the Max random object. Start from there and see what you can get. The length of the cycle of numbers will be (at most!) equal to mod, but if you're clever at setting the other parameters you can get much shorter cycles.

A note about integer representation: Max uses signed 32-bit values (i.e., integers in the range from -2,147,483,648 to 2,147,483,647). The **lp.lili** object uses unsigned arithmetic exclusively, interpreting negative numbers as *unsigned* 32-bit values. This allows generation of pseudo-random values over the entire range of 32-bit integers, but the results may seem a little strange at first sight. If you worry about this, restrict yourself to parameter values in the range 0 ≤ mod ≤ 2,147,483,647. One nice trick: setting the mod parameter to zero will generate random numbers across the entire range of 32-bit values.

Input

bang — Generate a new pseudo-random number in the range 0 ≤ x < mod and send it out the outlet.

int — In the left inlet: Set a new value for seed and generate a new pseudo-random number. The new value is sent out the outlet.

In the left middle inlet: set a new value for the mul parameter.

In the right middle inlet: set a new value for the add parameter.

In the right inlet: set a new value for the mod parameter.

set
seed — The symbol set followed by an integer sets a new value for the seed without sending a number out the outlet. You may use seed as a synonym for set. The set message follows general Max conventions; the seed message follows the usage of other Litter Power random-number generators.

Arguments

Four integer arguments, all of which are optional. However, you must explicitly specify the first argument if you want to set the second one, and so on.

int The first argument specifies the initial value of the mul parameter. The default value is 65,539.

The second argument specifies the initial value of the add parameter. The default value is 0.

The third argument specifies the initial value of the mod parameter. The default value is 0, which is interpreted as 4,294,967,296 (that is, the entire 32-bit range is used).

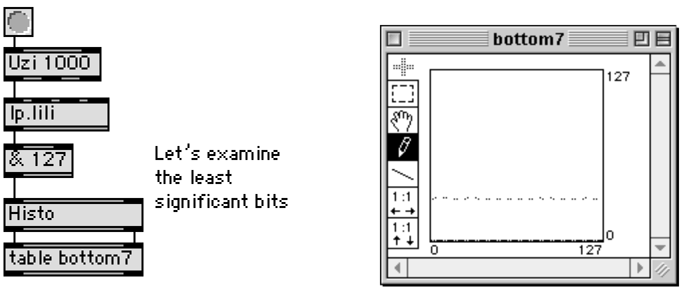
The fourth argument specifies the initial value of the seed parameter. The default value is 1.

The default values are taken from a common implementation of the linear congruence algorithm. There has been some informal indication from Cycling '74 that these are the same parameters as used by **random**.

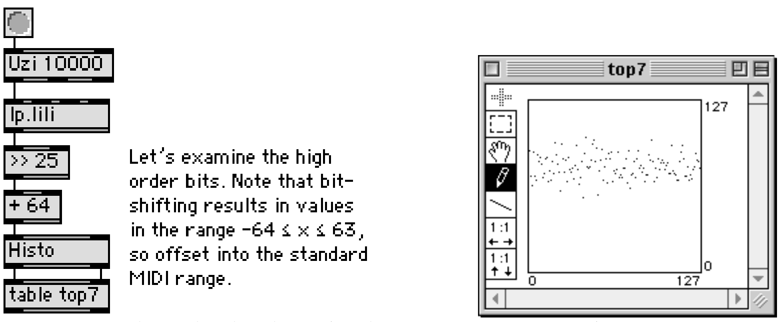
Output

int A more-or-less random value in the range 0 ≤ x < mod. (But cf. the notes on signed vs. unsigned representations above).

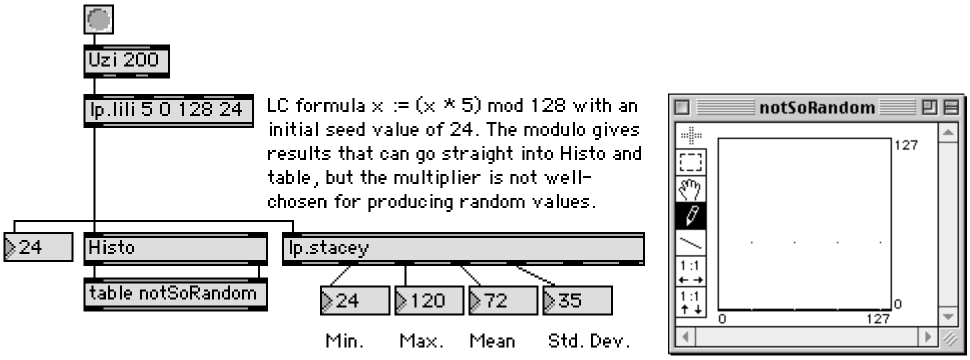
Examples



The lowest bits of numbers produced with the Linear Congruence method are not very random…



…but the high order bits are pretty random…



…unless you choose inappropriate parameters.

What's in a name?

See **lp.tata**.

See Also

lp.scampi	Scale, offset, and limit numbers; output integers
lp.tata	Generate random numbers using the Tausworthe 88 algorithm
lp.titi	Generate random numbers using the TT800 algorithm
lp.lll~	Parametric linear congruence "noise"
random	Hard-wired linear congruence pseudo-random number generator

Knuth, Donald E., *The Art of Computer Programming*, Vol. 2 *Semi-Numerical Algorithms*. (Reading, Mass.: Addison-Wesley, 1972).

The **lp.linnie** object wraps linear and triangular distributions into one neat package. You can choose which variant to use through the symmetry option described below.

Input

bang — Generate a random number from a linear or triangular distribution.

float — A floating point value in the range $0 \le x \le 1$ will be transformed using the formulae

$$f(x) = 1 - \sqrt{1 - x}$$ for the symmetry option pos,

$$f(x) = \sqrt{x}$$ for the symmetry option neg, and

$$f(x) = \begin{cases} \sqrt{2x} & \text{if } 0 \le x \le 0.5 \\ 1 - \sqrt{2x - 1} & \text{if } 0.5 < x \le 1.0 \end{cases}$$ for the symmetry option sym.

Assuming the input values are uniformly distributed, the output values will be from either a linear or triangular distribution.

sym
pos
neg — These symbols set the symmetry option. The symbol pos causes **lp.linnie** to produce deviates with a linear distribution with positive slope (i.e., values closer to one have a higher probability of occurring). The symbol neg generates a linear distribution with negative slope. The symbol sym produces a triangular distribution.

Note that all distributions are in the range $0 \le x \le 1$.

It may be helpful to notice that the negative and positive options refer to the slope of the density function.

seed — The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

You can initialize an **lp.linnie** object with up to two optional arguments. You must specify the first argument if you want to specify the second. The arguments, in order, are:

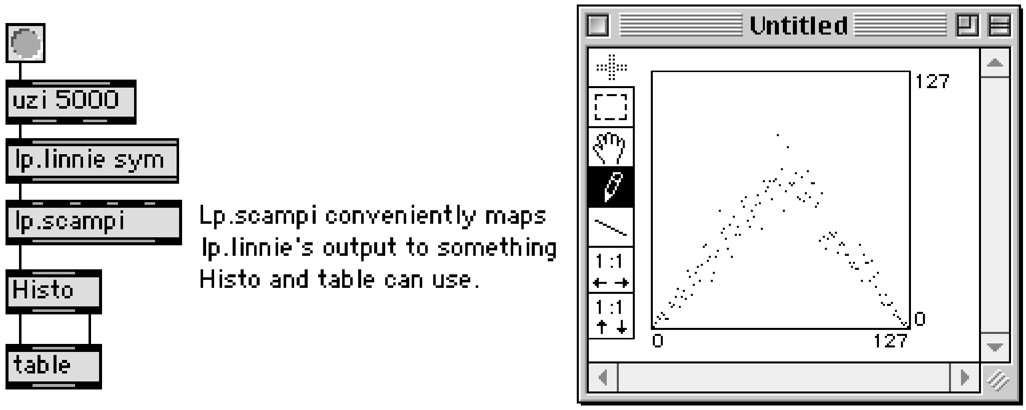
symbol — Any of the symbols sym, pos, or neg specifies an initial value for the symmetry option. The default value is neg.

int — Set the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

float — A random value from a linear distribution.

Examples



Using lp.linnie to generate random numbers from a triangular distribution

See Also

lp.expo	Generate random numbers from an exponential distribution
lp.hyppie	Generate random numbers from a hyperbolic cosine distribution
lp.loggie	Generate random numbers from a logistic distribution
lp.lonnie	Generate random numbers from a log-normal distribution
lp.scampi	Scale, offset, and limit numbers; output integers
lp.shhh	Generate random numbers from a “white” distribution
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

"White" noise using the Linear Congruence algorithm, while allowing you to specify values for the LC parameters. See **lp.lili** for more information on parametric linear congruence. The **lp.lll~** object works very much like **lp.lili**, except that the integral values produced are scaled to the range $-1 \le x. \le 1$ for signals. Note that the scaling factor is calculated relative to the mod parameter, so the maximum power range is always produced (except for LC cycles that get stuck at a constant… this can happen!).

For many parameter combinations, this cycle of numbers generated may be very short. In other words, the result may be much closer to pitch than noise. There are many intermediate signals.

Input

signal — Signal processing provided for the benefit of **begin~** / **selector~** configurations.

int — In the left inlet: Set a new value for the seed parameter.

In the left middle inlet: Set a new value for the mul parameter.

In the right middle inlet: Set a new value for the add parameter.

In the right inlet: Set a new value for the mod parameter.

Arguments

int — Four integer arguments, all of which are optional. However, you must explicitly specify the first argument if you want to set the second one, and so on.

The first argument specifies the initial value of the mul parameter. The default value is 65,539. The second argument specifies the initial value of the add parameter. The default value is 0. The third argument specifies the initial value of the mod parameter. The default value is 0, which is interpreted as 4,294,967,296 (that is, the entire 32-bit range is used). The fourth and final parameter specifies the initial value of the seed parameter. The default value is 1.

The default values are taken from a common implementation of the linear congruence algorithm. There has been some informal indication that these are the same parameters as used by **noise~**.

Output

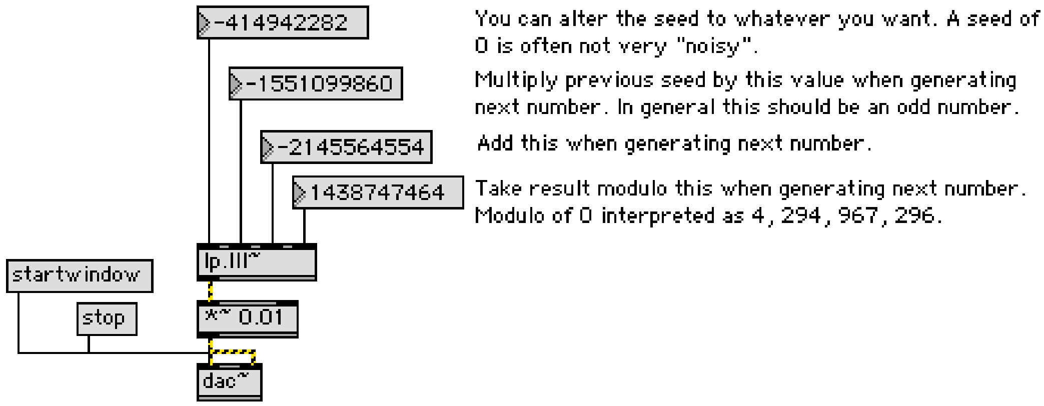
signal — Depending on the current parameters, anything from vaguely white noise, through noisy pitched signals, to pure pitch.

lp.lll~

Pro Bundles

Parametric Linear Congruence “noise”

Examples



You may want to experiment with different values.

What’s in a name?

I don’t know what got into me the day I named this one.

See Also

lp.frrr~	Low-frequency noise
lp.grrr~	“Gray” noise
lp.pfff~	“Brownian” ($1/f^2$) noise
lp.ppp~	Popcorn (dust) noise
lp.shhh~	White noise
lp.sss~	“Pink” noise (Voss/Gardner algorithm)
lp.zzz~	“Pink” noise (McCartney algorithm)
noise~	Another source of noise

Knuth, Donald E., *The Art of Computer Programming*, Vol. 2 *Semi-Numerical Algorithms*. (Reading, Mass.: Addison-Wesley, 1972).

The logistic distribution has two parameters, a location parameter named α and a scale parameter named β (in some literature, the equivalent Roman letter names are used). It is symmetric around its mean value (-β/α).

Input

bang Generate a random number from a logistic distribution.

float In the left inlet: A floating point value in the range 0 < x < 1 will be transformed using the formula

$$f_{\alpha,\beta}(x) = \frac{-\beta - \ln(\frac{1-x}{x})}{\alpha}$$

Assuming the input values are uniformly distributed, the output values will have a logistic distribution.
Input values outside the defined range are ignored.

In the middle inlet: set the value of α.

In the right inlet: set the value of β.

seed The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

You can initialize an **lp.loggie** object with up to three optional arguments. You must specify the first argument if you want to specify the second and you must specify the second to specify the third. The arguments, in order, are:

float Specify an initial value for the α parameter. The default value is one.

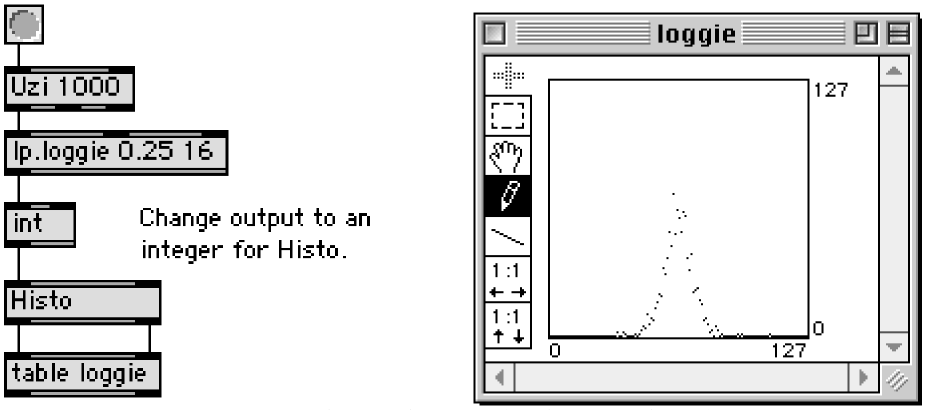
float Specify an initial value for the β parameter. The default value is zero.

int Set the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

float A random value from a logistic distribution.

Examples



Generating random values with a logistic distribution

See Also

lp.expo	Generate random numbers from an exponential distribution
lp.hyppie	Generate random numbers from a hyperbolic cosine distribution
lp.linnie	Generate random numbers from linear and triangular distributions
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

The log-normal distribution is derived from the normal (or "Gaussian") distribution. By definition, if the logarithm of a set of random variables has a normal distribution, then the variable has a log-normal distribution. Conceptually, one can think of the log-normal distribution is as the product of many independent uniform distributions (in contrast to the normal distribution, which is derived from the notion of summing independent uniform distributions). The log-normal distribution is often used to model characteristics such as income distribution, distribution of grain sizes in geological contexts, and distribution of weight or height in biological contexts.

The log-normal distribution has two parameters: mean and standard deviation. Values from a log-normal distribution are positive and skewed to the right (i.e., the median is greater than the arithmetic mean.)

Input

bang — Generate a random number from a log-normal distribution.

float — In the middle inlet: set the mean.

In the right inlet: set the standard deviation.

seed — The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

You can initialize an **lp.lonnie** object with up to three optional arguments. You must specify the first argument if you want to specify the second and you must specify the second to specify the third. The arguments, in order, are:

float — Specify an initial value for the mean. The default value is one.

float — Specify an initial value for the standard deviation. The default value is one.

int — Set the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

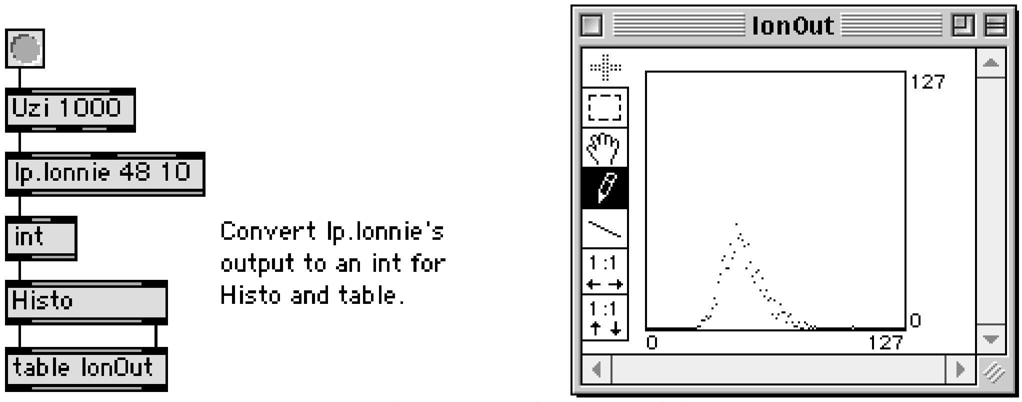
float — A random value from a log-normal distribution.

lp.lonnie

Pro Bundles

Generate random numbers from a log-normal distribution

Examples



Generating numbers with a log-normal distribution. The blip at x=102 is Bill Gates.

What's in a name?

LOg-**N**ormal would be lon, but lonnie sounds friendlier.

See Also

lp.bernie	Generate random numbers from a Bernoulli distribution
lp.norm	Generate random numbers from a normal ("Gaussian") distribution
lp.scampi	Scale, offset, and limit numbers; output integers
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

Generate random numbers from a normal ("Gaussian") distribution

lp.norm

All Bundles

This is the standard statistical "bell curve."

The normal distribution has two parameters: mean and standard deviation.

Input

bang	Generate a random number from a logistic distribution.
float	In the middle inlet: set the mean.
	In the right inlet: set the standard deviation.
seed	The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

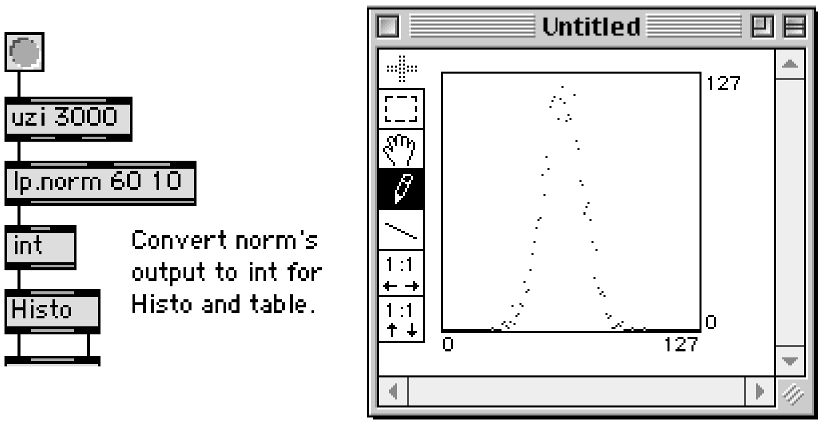
You can initialize an **lp.norm** object with up to three optional arguments. You must specify the first argument if you want to specify the second and you must specify the second to specify the third. The arguments, in order, are:

float	Specify an initial value for the mean. The default value is zero.
float	Specify an initial value for the standard deviation. The default value is zero.
int	Set the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

float	A random value from a normal distribution.

Examples



Generating random numbers with a Gaussian distribution

See Also

lp.bernie	Generate random numbers from a Bernoulli distribution
lp.lonnie	Generate random numbers from a log-normal distribution
lp.shhh	Generate random numbers from a "white" distribution
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

This object was developed prior to the availability of the **poltocar~** object in MSP version 2. It is retained in the Litter Power package to allow older Patchers that required this object to run unaltered and for users of older MSP versions. Conveniently, the interfaces of **lp.p2c~** and **poltocar~** are identical.

Input

signal — In left inlet: The amplitude component of a frequency domain signal.

In right inlet: The phase component (in radians) of a frequency domain signal.

Arguments

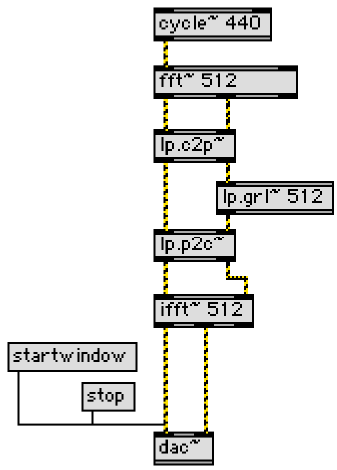
None.

Output

signal — Left outlet: The real component of the Cartesian representation equivalent to the incoming signal pair.

Right outlet: The imaginary component of the Cartesian representation equivalent to the incoming signal pair.

Examples



Converting from Cartesian to polar coordinates

See Also

fft~	Fast Fourier transform
ifft~	Inverse fast Fourier transform
lp.c2p~	Convert Cartesian to polar coordinates
lp.grl~	Phase unwrapping

This is a control-domain version of the **lp.pfff~** signal generator. It generates values in the range $0 \le x \le 1$.

Input

bang Generate a random value from a Brownian ($1/f^2$) distribution.

seed The symbol seed followed by an integer reseeds the internal random number generator. (Only available if the object was initialized with a seed parameter.)

int In second inlet: sets the NN factor. This is a value in the range $0 \le nn \le 31$ that controls the "granularity" of the random numbers. For a NN factor of zero (the default), all bits of the random numbers are random. For other values, NN indicates the number of low-order bits to mask out before converting to a floating-point value.

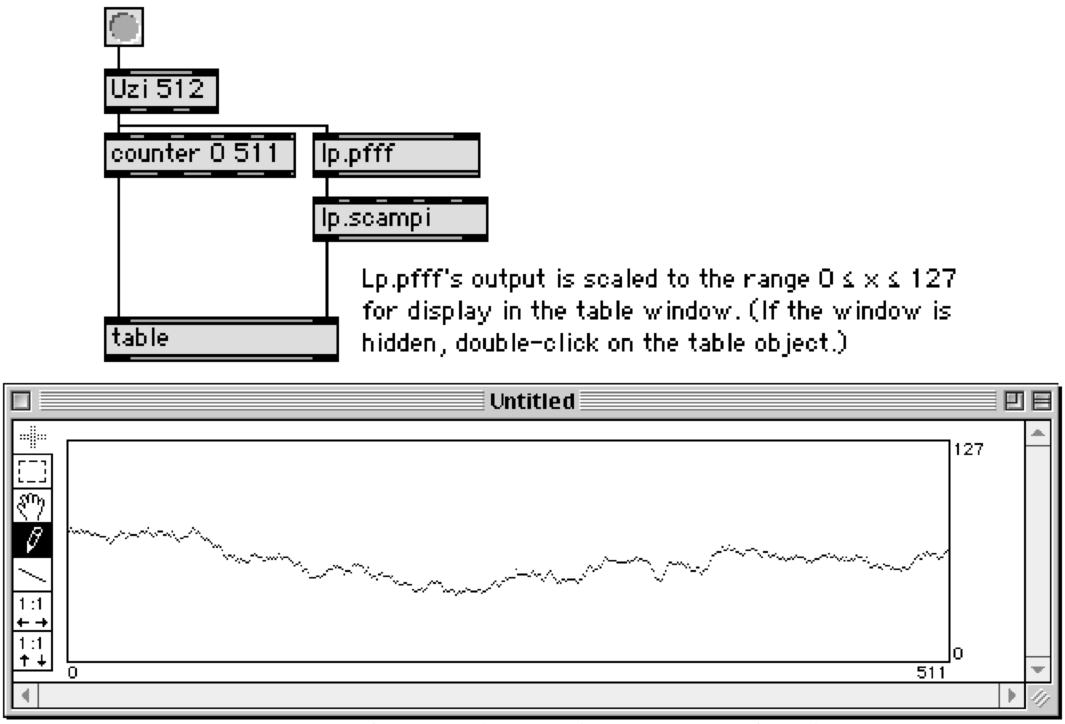
Arguments

int Set the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

float A random value in the range $0 \le x \le 1$.

Examples



Generating random numbers with a Brownian distribution

What's in a name?

See **lp.pfff~**.

See Also

lp.grrr	"Gray" noise (control domain)
lp.pfff~	"Brownian" ($1/f^2$) noise
lp.scampi	Scale, offset, and limit numbers; output integers
lp.shhh	Generate random numbers from a "white" distribution
lp.sss	Generate random numbers from a 1/f ("pink") distribution
lp.zzz	Generate random numbers from a 1/f ("pink") distribution
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

Brownian noise is fractal noise with a falloff of about 12 dB per octave.

Input

signal — Signal processing provided for the benefit of **begin~**/**selector~** configurations.

int — NN factor: specifies the number of low-order bits to clear before converting the integer representation to the floating-point value used in signal connections. The NN factor for the **lp.pfff~** object may be in the range $0 \le nn \le 31$.

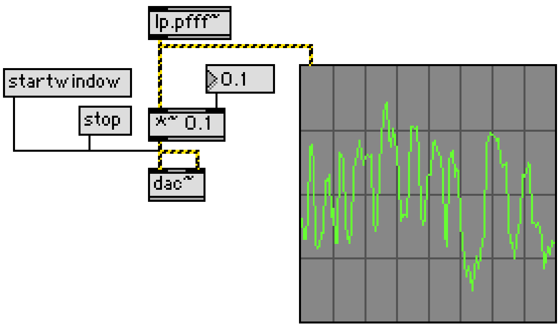
Arguments

int — Optional value to set the initial NN factor. This is zero (no masking) by default.

Output

signal — Brown noise

Examples



Brownian noise

What's in a name?

Onomatopoeia.

See Also

lp.frrr~	Low-frequency noise
lp.grrr~	"Gray" noise
lp.lll~	Parametric linear congruence "noise"
lp.pfff	Generate random numbers from a $1/f^2$ ("Brownian") distribution
lp.phhh~	"Black" ($1/f^3$) noise
lp.ppp~	Popcorn (dust) noise
lp.shhh~	White noise
lp.sss~	"Pink" noise (Voss/Gardner algorithm)
lp.zzz~	"Pink" noise (McCartney algorithm)
noise~	"White" noise
pink~	"Pink" noise

The Poisson distribution has one parameter, λ, which happens to be both the expected mean and variance. (Standard deviation is therefore $\sqrt{\lambda}$). The Poisson distribution generates non-negative integers only. It is defined for positive real values of λ.

The Poisson distribution was originally developed as an efficient means of approximating the Bernoulli distribution for special cases (to wit, when the product *np* is small even when *n* is large). It has gained considerable popularity for use in algorithmic composition, particularly due to the influence of Iannis Xenakis, who used it extensively.

Input

bang — Generate a random number from a Poisson-distributed distribution and send it out the outlet.

float — In the middle inlet: set the value of λ.

seed — The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

You can initialize an **lp.pfishie** object with up to two optional arguments. You must specify the first argument if you want to specify the second.

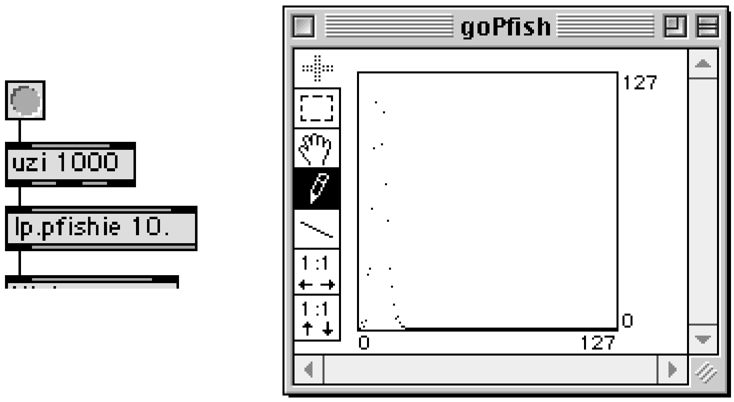
float — The first argument specifies an initial value for λ. The default value is one.

int — The second argument specifies a seed for the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

int — A random value from a Poisson-distribution.

Examples



Generating random numbers from a Poisson distribution

What's in a name?

The obvious name for this would have been **lp.fishie**, but that was already taken.

See Also

lp.bernie	Generate random numbers from a Bernoulli distribution
lp.expo	Generate random numbers from an exponential distribution
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

Black noise is fractal noise that is even "darker" than Brownian nosie. It is characterized by a falloff of about 18 dB per octave.

Input

signal	Signal processing provided for the benefit of **begin~** / **selector~** configurations.
int	NN factor: specifies the number of low-order bits to clear before converting the integer representation to the floating-point value used in signal connections. The NN factor for the **lp.phhh~** object may be in the range $0 \leq nn \leq 31$.

Arguments

int	Optional value to set the initial NN factor. This is zero (no masking) by default.

Output

signal	Black noise

What's in a name?

Onomatopoeia.

See Also

lp.frrr~	Low-frequency noise
lp.grrr~	"Gray" noise
lp.lll~	Parametric linear congruence "noise"
lp.pfff~	"Brownian" ($1/f^2$) noise
lp.phhh~	"Black" ($1/f^3$) noise
lp.ppp~	Popcorn (dust) noise
lp.shhh~	White noise
lp.sss~	"Pink" noise (Voss/Gardner algorithm)
lp.zzz~	Pink noise (McCartney algorithm)
noise~	Another source of noise
pink~	Another source of pink noise

lp.ppp~

Pro Bundles

This noise generator, known variously as popcorn or dust noise, generates exponentially distributed pulses of varying amplitude and pulse width. It resembles kinds of noise frequently found in telecommunications lines and sometimes in radio broadcast. Curiously, in most naturally occurring circumstances, the pulses are all of the same sign, either positive or negative. The **lp.ppp~** object supports both, as well as a symmetrical variant in which positive and negative pulses are mixed at random.

When the density of pops becomes high and pulse width also increases, it becomes possible for pops to overlap. The current implementation makes no provision for overlapping pops; one pop must be completed (i.e., the signal must return to 0) before the next one can begin. Thus, the actual frequency of pops may fall slightly underneath the specified mean.

Input

signal	Signal processing provided for the benefit of **begin~** / **selector~** configurations.
float	In the left inlet: Set the mean density of impulses. This is specified in Hz.
int	In the right inlet: Set the width of impulses in samples. This must be a non-negative value that specifies the length of the upward and downward ramps.
sym pos neg	These messages set the symmetry option. The message pos causes only positive impulses to be generated, the neg message causes only negative impulses to be generated, and the sym message causes both positive and negative impulses to be chosen at random.

Arguments

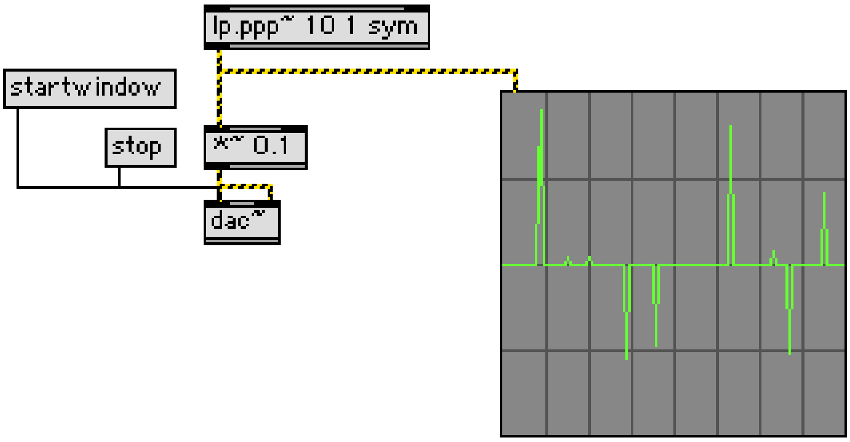
You can initialize an **lp.ppp~** object with up to three optional arguments. You must specify the first argument if you want to specify the second and you must specify the second to specify the third. The arguments, in order, are:

float	Specify the initial mean density of impulses in Hz. The default value is 10.
int	Specify the initial ramp width of impulses. The default value is one sample.
symbol	Any of the symbols sym, pos, or neg, will specify the initial value of the symmetry option. The default value is pos.

Output

signal	Popcorn noise

Examples



Generating popcorn noise

What's in a name?

P-P-Popcorn,
yummy good popcorn,
You're the b-b-b-best noise any guy could have!
When the d-dust cracks
On the ph-phone line,
I'll be sampling to an ei-ei-ei-eightbit .WAV!
Sung to the tune of "K-K-Katie"

See Also

lp.frrr~	Low-frequency noise
lp.grrr~	"Gray" noise
lp.lll~	Parametric linear congruence "noise"
lp.pfff~	"Brownian" ($1/f^2$) noise
lp.shhh~	White noise
lp.sss~	"Pink" noise (Voss/Gardner algorithm)
lp.zzz~	Pink noise (McCartney algorithm)
noise~	Another source of noise
pink~	Another source of pink noise

The **lp.scampf** object wraps ***** and **+** into one convenient object with range-correction and splitting capabilities.

The core duty of **lp.scampf** is to multiply incoming values by a scaling factor and then to add an offset. In the following discussion this is referred to as *mapping*. Additionally, values may be constrained to a given range. Optionally, out-of-range values may be routed to a second outlet. If you prefer, you may specify that an arbitrary message be sent out the right outlet whenever mapped values exceed the specified range.

Input

float
int

In the first (left) inlet: the number is mapped by the current scale and offset values. The resulting value may be constrained to a given range (see below) and will be sent out one of the outlets. If no range correction or splitting option is in effect the value will be sent out the left outlet. Also, as long as the calculated output value is within the current range, the value will be sent out the left outlet. If range correction is in effect *and* the calculated value is outside the current range, *and* splitting is on, then the calculated value (or, optionally, another message) is sent out the right outlet.

In the second inlet: set the scale parameter.

In the third inlet: set the offset parameter.

In the fourth inlet: set the range lower bound.

In the fifth inlet: set the range upper bound.

Setting the lower bound to a value greater than or equal to the upper bound generates an invalid range. In this case, mapped values cannot be corrected and all input values will be treated as out-of-range by the splitting option.

bang

The result of the last input value is recalculated to reflect current scale, offset, and range settings. This result is sent out the appropriate outlet.

set

The symbol set followed by a number sets the input value without producing any output.

split — The symbol split followed by an integer parameter sets the splitting option. If the integer is zero, no splitting takes place and all values are sent out the left outlet. If the integer is one, any value that, after scaling and offset, is outside the current range will be routed to the right outlet. Note that this does not change the range correction setting: any current clipping, wrapping, or reflection calculations will be performed before the final calculated value is sent out the right outlet.

Any other integer parameter will be interpreted as split 1, but making use of this feature is deprecated and may not be compatible with future versions of the **lp.scampf** object.

The symbol split followed by a symbol or float parameter will cause the *parameter* to be sent out the right outlet when the scaled and offset value is out of range. A typical idiom would be the message split bang to cause **lp.scampf** to send a bang out the right outlet instead of the corrected value.

clip
wrap
reflect
stet — Set the current range correction option.

The symbol clip causes mapped values to be clipped to the current range. Similarly, the symbols wrap and reflect cause the mapped value to be wrapped or reflected (respectively) into range before output.

The symbol stet turns off range correction. The range bounds are not effected; this is useful if you want to turn off range correction while leaving the splitting option in effect.

None of these symbols effects the splitting option.

All of these symbols may be followed by up to two optional numeric parameters to set the range.

If no parameters are included with the range correction message, then the current range remains in effect.

If one value is specified, it defines a range with zero as one of the endpoints. If the parameter is positive, the value is taken as the upper bound and the lower bound is set to zero. If the parameter is negative, it is taken as the lower bound and the upper bound is set to zero. If the parameter is zero, range correction is turned off.

If two values are specified in the message, they define the range. The smaller of the two values is taken as the lower bound and the larger is taken as the upper bound.

Invalid range settings (that is, setting the lower bound to a value greater than *or equal* to the upper bound) are ignored, so don't do that.

Arguments

You can initialize an **lp.scampf** object with up to six optional arguments: four integer or floating-point arguments and two symbol arguments. The symbol arguments may be in any order and may be arbitrarily interspersed among the numeric arguments (or left out altogether—they *are* optional). The numeric arguments must, however, be in the order given below. Also, you must specify the first numeric argument if you want to specify the second, and so on for the subsequent numeric arguments.

int
float
The first numeric argument specifies an initial value for the scaling factor. The default value is 0.0078125 (this is 1/128, chosen because it neatly maps MIDI input into the range $0 \le x < 1$).

The second numeric argument specifies an initial value for the offset. The default value is zero.

The third and fourth numeric arguments specify initial values for the range bounds. If no values are specified, the range is set to $0 \le x < 1$ (but this will be without consequence unless range-correction or splitting are turned on). If only the third numeric value is specified and it is positive, it will be taken as the range upper bound; if it is negative it will be taken as the range lower bound; in either case zero will be taken, by default, for the other range bound. If both of the final numeric arguments are specified, the smaller value will be taken as lower bound and the larger value as upper bound. If you explicitly specify two equal values, they will be ignored entirely and the default range will be used.

symbol
You can initialize the range-correction option with one of the symbols clip, wrap, reflect, or stet. If none of these symbols is listed, no range-correction will take place initially (that is, stet is the default option).

If you include the symbol split in the argument list, this is initializes the splitting option to split 1.

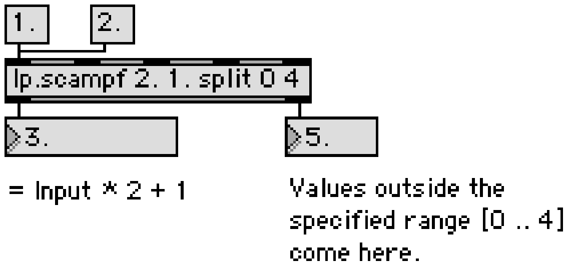
Output

float
Out the left outlet: the mapped input value. If range-correction is on but splitting is off, out-of-range values will be corrected back into range.

Out the right outlet: If the splitting option is set to one, out-of-range values will be sent out the right outlet (possibly after range correction).

symbol
If the split symbol option is in effect, a symbol will be sent out the right outlet whenever a mapped value is outside the current range.

Examples



Values are scaled and offset; out-of-range results are sent to the right outlet

What's in a name?

Portmanteau word from scale and map, with a few letters getting displaced in the process. The final 'f' indicates floating point.

See Also

*****	Multiply two numbers, output the result
+	Add two numbers, output the result
clip	Limit numbers to a specified range
lp.scampi	Scale, offset, and limit numbers; output integers
lp.scamp~	Scale, offset, and limit signals
split	Look for a range of numbers

This is your one-stop source for scaling, offsetting, and limiting integer values to a different output range. The **lp.scampi** object wraps ***** and **+** into one convenient object with range-correction and splitting capabilities. Since **lp.scampi** may also be used for calculating integer values from floating-point input, it also provides facilities for rounding.

The core duty of **lp.scampi** is to multiply incoming values by a scaling factor and then to add an offset. In the following discussion this is referred to as *mapping*. Additionally, values may be constrained to a given range. Optionally, out-of-range values may be routed to a second outlet. If you prefer, you may specify that an arbitrary message be sent out the right outlet when mapped values exceed the specified range. Finally, you can specify how floating-point results are to be converted to integers. The options cover: conventional rounding, floor, ceiling, truncation (i.e., "to zero"), and "to infinity."

Input

int
float

In the first (left) inlet: the number is mapped by the current scale and offset values. If necessary, the resulting value is converted to an integer, following the current rounding settings. This result may be constrained to a given range (see below) and will be sent out one of the outlets. If no range correction or splitting option is in effect the value will be sent out the left outlet. Also, as long as the calculated output value is within the current range, the value will be sent out the left outlet. If range correction is in effect *and* the calculated value is outside the current range, *and* splitting is on, then the calculated value (or, optionally, another message) is sent out the right outlet.

In the second inlet: set the scale parameter.

In the third inlet: set the offset parameter.

In the fourth inlet: set the range lower bound. Note that the range lower bound is stored as an integer; incoming floats are rounded to integers by **lp.scampi** following the current rounding option.

In the fifth inlet: set the range upper bound. Note that the range lower bound is stored as an integer; incoming floats are rounded to integers by **lp.scampi** following the current rounding option.

Setting the lower bound to a value greater than or equal to the upper bound generates an invalid range. In this case, mapped values cannot be corrected and all input values will be treated as out-of-range by the splitting option.

bang

The result of the last input value is recalculated to reflect current scale, offset, and range settings. This result is sent out the appropriate outlet.

set

The symbol set followed by a number sets the input value without producing any output.

round
floor
ceiling
toinf
tozero
trunc

These messages set the rounding method used in converting non-integral floating-point values to integers. This conversion takes place immediately after the mapping calculation, before splitting and range-correction.

The round message sets conversion to conventional rounding (i.e., if the fractional portion of a non-integral floating-point value is greater than or equal to 1/2, the value is rounded up to the next integer, otherwise the value is rounded down to the next lowest integer).

The floor message causes all non-integral floating point values to be rounded downwards to the next integer.

The ceiling message causes all non-integral floating-point values to be round upwards to the next integer.

The toinf message causes positive non-integral floating-point values to be rounded upwards and negative values to be rounded downwards.

The tozero message causes positive non-integral floating-point values to be rounded downwards and negative values to be rounded upwards. This behavior is often referred to as *truncation* and is the way float-to-integer conversion is normally handled in Max.

The message trunc is a synonym for tozero.

split

The symbol split followed by an integer parameter sets the splitting option. If the integer is zero, no splitting takes place and all values are sent out the left outlet. If the integer is one, any value that, after scaling and offset, is outside the current range will be routed to the right outlet. Note that this does not change the range correction setting: any current clipping, wrapping, or reflection calculations will be performed before the final calculated value is sent out the right outlet.

Any other integer parameter will be interpreted as split 1, but making use of this feature is deprecated and may not be compatible with future versions of the **lp.scampi** object.

The symbol split followed by a symbol or float parameter will cause the *parameter* to be sent out the right outlet when the scaled and offset value is out of range. A typical idiom would be the message split bang to cause **lp.scampi** to send a bang out the right outlet instead of the corrected value.

clip
wrap
reflect
stet

Set the current range correction option.

The symbol clip causes mapped values to be clipped to the current range. Similarly, the symbols wrap and reflect cause the mapped value to be wrapped or reflected (respectively) into range before output.

The symbol stet turns off range correction. The range bounds are not effected; this is useful if you want to turn off range correction while leaving the splitting option in effect.

None of these symbols effects the splitting option.

All of these symbols may be followed by up to two optional integer parameters to set the range.

If no parameters are included with the range correction message, then the current range remains in effect.

If one integer is specified, it defines a range with zero as one of the endpoints. If the parameter is positive, the value is taken as the upper bound and the lower bound is set to zero. If the parameter is negative, it is taken as the lower bound minimum and the upper bound is set to zero. If the parameter is zero, range correction is turned off.

If two integers are specified in the message, the smaller of the two values is taken as the lower bound and the larger is taken as the upper bound.

Invalid range settings (that is, setting the lower bound to a value greater than *or equal* to the upper bound) are ignored.

Arguments

You can initialize an **lp.scampi** object with up to seven optional arguments: four integer or floating-point arguments and three symbol arguments. The symbol arguments may be in any order and may be arbitrarily interspersed among the numeric arguments (or left out altogether—they *are* optional). The numeric arguments must, however, be in the order given below. Also, you must specify the first numeric argument if you want to specify the second, and so on for the subsequent numeric arguments.

int
float

The first numeric argument specifies an initial value for the scaling factor. The default value is 128 (chosen because it, together with other default settings, conveniently maps the output of many Litter Power objects into the MIDI-friendly range $0 \le x \le 127$).

The second numeric argument specifies an initial value for the offset. The default value is zero.

The third and fourth numeric arguments specify initial values for the range bounds. If no values are specified, the range is set to $0 \le x \le 127$ (but this will be without consequence unless range-correction or splitting are turned on). If only the third numeric value is specified and it is positive, it will be taken as the range upper bound; if it is negative it will be taken as the range lower bound; in either case zero will be taken, by default, for the other range bound. If both of the final numeric arguments are specified, the smaller value will be taken as lower bound and the larger value as upper bound. If you explicitly specify two equal values, they will be ignored entirely and the default range will be used.

Range bounds are stored by **lp.scampi** as integers. If you specify any of these values as a floating point number, they will be rounded to integer by **lp.scampi** following the initial rounding option

symbol

You can initialize the float-to-integer conversion method with one of the symbols round, floor, ceiling, toinf, or tozero. If none of these symbols is included in the argument list, all floating-point values will be rounded towards zero (i.e., truncated).

You can initialize the range-correction option with one of the symbols clip, wrap, reflect, or stet. If none of these symbols is listed, no range-correction will take place initially (that is, stet is the default option).

If you include the symbol split in the argument list, this initializes the splitting option to split 1.

Output

int

Out the left outlet: the mapped input value. If range-correction is on but splitting is off, out-of-range values will be corrected back into range.

Out the right outlet: If the splitting option is set to one, out-of-range values will be sent out the right outlet (possibly after range correction). If the split symbol option is in effect, a symbol will be sent out the right outlet instead.

symbol

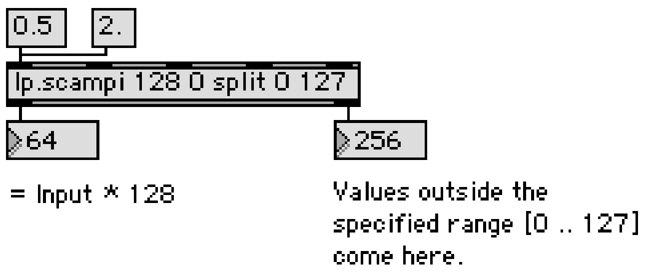
If the split symbol option is in effect, a symbol will be sent out the right outlet whenever a mapped value is outside the current range.

lp.scampi

All Bundles

Scale, offset, and limit integer values

Examples



Values are scaled and (optionally) offset
Out-of-range results are sent out the right outlet

What's in a name?

Portmanteau word from scale and map, with a few letters getting displaced in the process. The final 'i' indicates integer.

*****	Multiply two values, output the result
+	Add two values, output the result
clip	Limit numbers to a specified range
lp.scampf	Scale, offset, and limit floating-point values
lp.scamp~	Scale, offset, and limit signals
split	Look for a range of numbers

This is your one-stop source for scaling, offsetting, and limiting signals to a different output range. The **lp.scamp~** object wraps ***~** and **+~** into one convenient object with range-correction and splitting capabilities.

The core duty of **lp.scampf** is to multiply incoming values by a scaling factor and then to add an offset. In the following discussion this is referred to as *mapping*. Additionally, values may be constrained to a given range. You can poll an **lp.scamp~** object to find out how many samples were out of range.

Input

signal — In the first (left) inlet: samples are mapped by the current scale and offset values. The resulting samples may be constrained to a given range (see below). After any range correction the final result is sent out the left outlet.

In the second inlet: set the scale parameter, overriding any floating-point parameter value.

In the third inlet: set the offset parameter, overriding any floating-point parameter value.

In the fourth inlet: set the range lower bound, overriding any floating-point parameter value.

In the fifth inlet: set the range upper bound, overriding any floating-point parameter value.

If a signal sets the lower bound to a value greater than or equal to the upper bound, the range bounds are reversed.

float — In the second inlet: set the scale parameter.

In the third inlet: set the offset parameter.

In the fourth inlet: set the range lower bound.

In the fifth inlet: set the range upper bound.

Setting the lower bound to a value greater than or equal to the upper bound reverses the range bounds.

bang — Sends the current count of how many samples were out of range out the right outlet. As a side effect, the count is reset to zero.

clip
wrap
reflect
stet

Set the current range correction option.

The symbol clip causes mapped values to be clipped to the current range. Similarly, the symbols wrap and reflect cause the mapped value to be wrapped or reflected (respectively) into range before output.

The symbol stet turns off range correction. The range bounds are not effected.

All of these symbols may be followed by up to two optional numeric parameters to set the range.

If no parameters are included with the range correction message, then the current range remains in effect. You may also find this form of the range correction messages most convenient when signals are connected to any of the range bounds inlets.

If one value is specified, it defines a range with zero as one of the endpoints. If the parameter is positive, the value is taken as the upper bound and the lower bound is set to zero. If the parameter is negative, it is taken as the lower bound and the upper bound is set to zero. If the parameter is zero, range correction is turned off.

If two values are specified in the message, they define the range. The smaller of the two values is taken as the lower bound and the larger is taken as the upper bound.

Invalid range settings (that is, setting the lower bound to a value greater than or equal to the upper bound) are ignored.

Arguments

You can initialize an **lp.scamp~** object with up to five optional arguments: four integer or floating-point arguments and one symbol argument. The symbol argument, if used, may be included anywhere among the numeric arguments. The numeric arguments must, however, be in the order given below. Also, you must specify the first numeric argument if you want to specify the second, and so on for the subsequent numeric arguments.

int
float

The first numeric argument specifies an initial value for the scaling factor. The default value is 0.5488116361 (this is equivalent to a gain of -6dB).

The second numeric argument specifies an initial value for the offset. The default value is zero.

The third and fourth numeric arguments specify initial values for the range bounds. If no values are specified, the range is set to $-1 \leq x \leq 1$. If only the third numeric value is specified **lp.scamp~** will assume that you want a symmetrical range with the specified value and its negative as the range bounds. If both of the final numeric arguments are specified, the smaller value will be taken as lower bound and the larger value as upper bound. You can specify two equal values to generate a constant signal, but there are less processor-intensive ways of doing this.

symbol — You can initialize the range-correction option with one of the symbols clip, wrap, reflect, or stet. If none of these symbols is listed, no range-correction will take place initially (that is, stet is the default option).

Output

signal — Out the left outlet: the mapped input value. If range-correction is on but splitting is off, out-of-range values will be corrected back into range.

int — Out the right outlet: a count of the number of out-of-range samples since the last time a count was sent out.

What's in a name?

Portmanteau word from scale and map, with a few letters getting displaced in the process. The final '~' is in honor of Miller Puckette (or maybe the Milwaukee airport), but you knew that anyway.

See Also

***~**	Multiply two signals
+~	Add signals
clip~	Limit signal amplitude
lp.scampf	Scale, offset, and limit numbers; output floating-point values
lp.scampi	Scale, offset, and limit numbers; output integers
lp.grl~	Phase unwrapping
pong~	Variable range signal folding

lp.shhh

All Bundles

Generate random numbers from a "white" distribution

This is a control-domain version of the **lp.shhh~** signal generator. It generates values in the range $0 \le x \le 1$.

Input

bang — Generate a random value with "white" (i.e., uniform) distribution.

int — In the right inlet: sets the NN factor, specifying the number of low-order bits to clear before converting the integer representation to floating-point. The NN factor may be in the range $0 \le nn \le 31$.

seed — The symbol seed followed by an integer reseeds the internal random number generator.

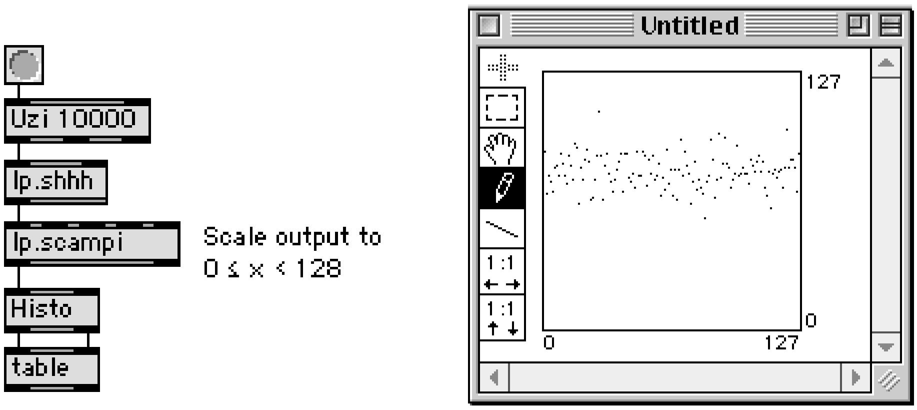
Arguments

int — Set the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

float — A random value in the range $0 \le x \le 1$.

Examples



Generating random numbers

What's in a name?

See **lp.shhh~**.

See Also

lp.grrr	"Gray" noise (control domain)
lp.pfff	Generate random numbers from a $1/f^2$ ("Brownian") distribution
lp.shhh~	White noise
lp.sss	Generate random numbers from a 1/f ("pink") distribution
lp.zzz	Generate random numbers from a 1/f ("pink") distribution
lp.scampi	Scale, offset, and limit numbers; output integers
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

This is the "whitest" white noise available for Max/MSP, taking about $2.2 \cdot 10^{14}$ years to repeat its cycle. That's an order of magnitude longer than the estimated age of the universe since the Big Bang.

Based on the **lp.tata** random number generator, it should also use a little less processing power than other white noise implementations.

Input

signal — Signal processing provided for the benefit of **begin~** / **selector~** configurations.

int — NN factor, a value in the range $0 \leq nn \leq 31$, specifying the number of low-order bits of the randomly generated integers to set to zero before converting to a signal.

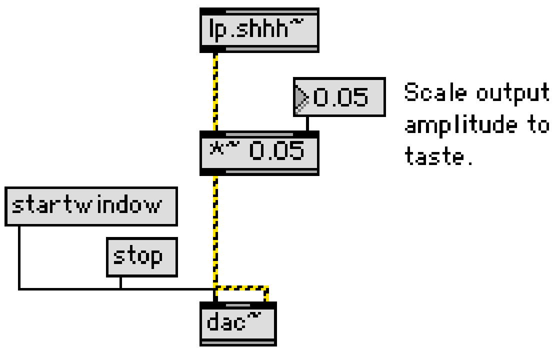
Arguments

int — Optional initial value for the NN factor. Zero by default.

Output

signal — Noise.

Examples



Generating white noise

What's in a name?

Onomatopoeia.

See Also

lp.frrr~ Low-frequency noise
lp.grrr~ "Gray" noise
lp.lll~ Parametric linear congruence "noise"
lp.pfff~ "Brownian" ($1/f^2$) noise
lp.phhh~ "Black" ($1/f^3$) noise
lp.ppp~ Popcorn (dust) noise

lp.sss~ "Pink" noise (Voss/Gardner algorithm)
lp.zzz~ Pink noise (McCartney algorithm)
noise~ Another source of noise
pink~ Another source of pink noise

Hawking, Stephen W., *A Brief History of Time*. (London/New York: Bantam 1988)

This is a control-domain version of the **lp.sss~** signal generator. It generates values in the range 0 ≤ x ≤ 1. It uses the Voss/Gardner algorithm, first published in Martin Gardner's "Mathematical Games" section of Scientific American (cf. the Bibliography).

Input

bang	Generate a random value with 1/f distribution.
int	In the right inlet: sets the NN factor, specifying the number of low-order bits to clear before converting the integer representation to floating-point. The NN factor may be in the range 0 ≤ *nn* ≤ 31.
seed	The symbol seed followed by an integer reseeds the internal random number generator.

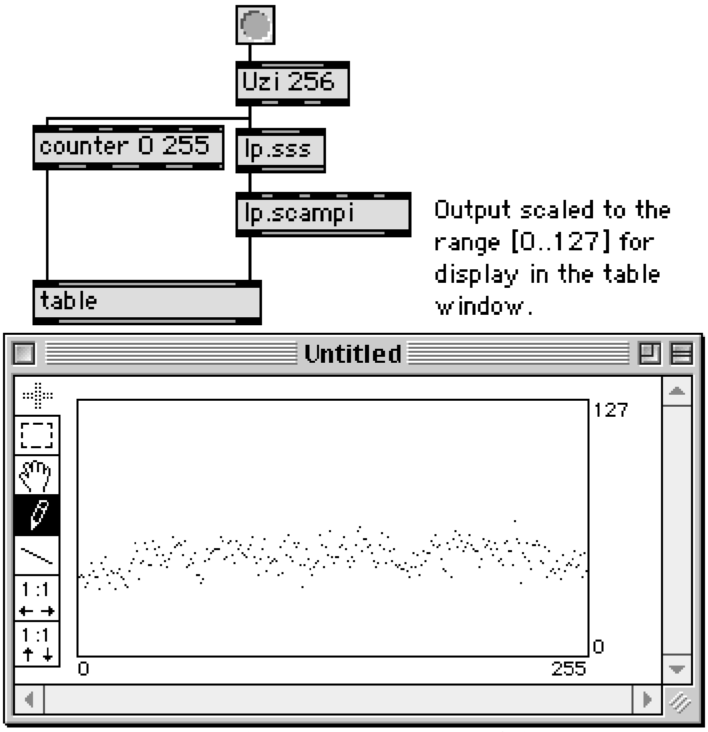
Arguments

int	Set the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default if nothing is specified).

Output

float	A random value in the range 0 ≤ x ≤ 1.

Examples



Generating random numbers with 1/f distribution

Generate random numbers from a 1/f (“pink”) distribution (Voss/Gardner algorithm)

What’s in a name?

See **lp. shhh~**.

See Also

lp.grrr	“Gray” noise (control domain)
lp.pfff	Generate random numbers from a $1/f^2$ (”Brownian”) distribution
lp.shhh	Generate random numbers from a “white” distribution
lp.sss~	“Pink” noise (Voss/Gardner algorithm)
lp.zzz	Generate random numbers from a 1/f (“pink”) distribution
lp.scampi	Scale, offset, and limit numbers; output integers
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

Gardner, Martin, “Mathematical Games: White and Brown Music, Fractal Curves, and One-over-*f* Fluctuations,” *Scientific American* 1978, 16-31.

Pink noise generated based on the original Voss/Gardner algorithm for generating 1/*f* distributed random numbers.

Input

signal — Signal processing provided for the benefit of **begin~**/**selector~** configurations.

int — NN factor: specifies the number of low-order bits to clear before converting the integer representation to the floating-point value used in signal connections. The NN factor for the **lp.sss~** object may be in the range $0 \le nn \le 31$.

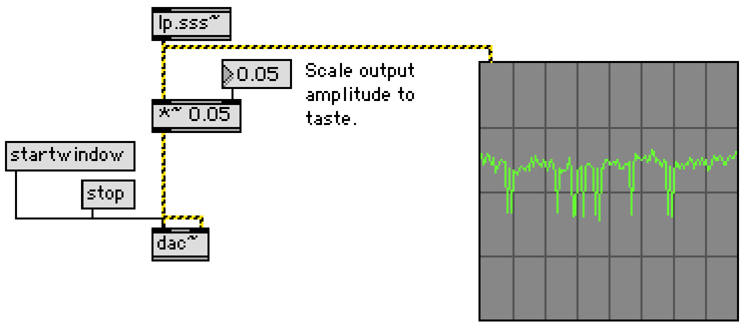
Arguments

int — Optional value to set the initial NN factor. This is zero (no masking) by default

Output

signal — Pink noise.

Examples



Generating pink noise

What's in a name?

Onomatopoeia.

See Also

lp.frrr~	Low-frequency noise
lp.grrr~	"Gray" noise
lp.lll~	Parametric linear congruence "noise"
lp.pfff~	"Brownian" ($1/f^2$) noise
lp.phhh~	"Black" ($1/f^3$) noise
lp.ppp~	Popcorn (dust) noise
lp.shhh~	White noise
lp.zzz~	"Pink" noise (McCartney algorithm)
pink~	Another source of pink noise (algorithm not known)

lp.stacey

All Bundles

Collect statistics

Count the input values, track cumulative minimum and maximum, and calculate mean, standard deviation, skew, and kurtosis.

Input

float int	Any numeric value is added to the cumulative statistics. The resulting statistics are output through the outlets.
bang	Send the current value of all statistics out the outlets.
clear	Remove all statistical data stored in the object. In keeping with the standard behavior of the clear message in other Max objects, no data are sent out the outlets in response to this message.
clearbang	Remove all statistical data stored in the object *and* send zeros through all the outlets. This behavior is often desired, hence the merging of clear and bang into a single message.

Arguments

None.

Output

int	Out the first (leftmost) outlet: statistical count of the number of values that are being statistically evaluated.
float	Out the second outlet: minimum value received.

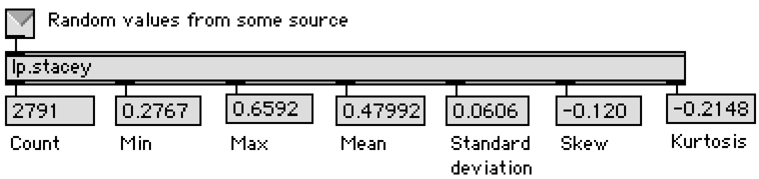
Out the third outlet: maximum value received.Out the fourth outlet: mean.

Out the fifth outlet: sample standard deviation

Out the sixth outlet: skew

Out the seventh outlet: kurtosis.

Examples



Collecting statistics

What's in a name?

The convention in the Litter Package of abbreviating the function to a nickname would have given us *statsie*. I thought *stacey* sounded better.

See Also

Histo	Make a histogram of numbers received
table	Store and graphically edit an array of numbers

Behnen, Konrad and Georg Neuhaus, *Grundkurs Stochastik, Teubner Studienbücher Mathematik* (Stuttgart: Teubner, 1984).

Salkind, Neil J., *Statistics for People Who (Think They) Hate Statistics.* (Thousand Oaks, California: Sage, 2000)

…and the entire rest of the Litter Package

lp.stu

Pro Bundles

Generate random numbers from Student's t distribution

The *t* distribution has one parameter, referred to as *degrees of freedom*. It produces an asymmetrical distribution of positive deviates. The degrees of freedom parameter is a positive integer.

The *t* distribution was developed by the statistician William Gosset. At the time of publication Gosset was employed by the Guinness brewery, which did not allow employees to publish, so Gosset wrote under the pseudonym of Student. The rest is history.

Input

bang — Generate a random value from a *t* distribution.

int — In the right inlet: set the degrees of freedom

seed — The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

int — The **lp.stu** object can be initialized with two optional arguments. You must specify the first argument if you want to specify the second.

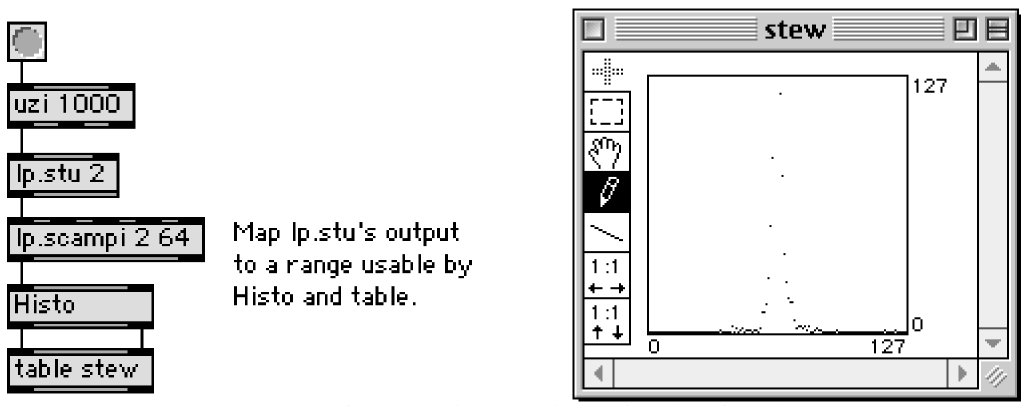
The first argument sets an initial value for the degrees of freedom parameter. The default is one.

The second argument sets the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

float — A random number from a *t* distribution.

Examples



Generating random numbers with Student's *t* distribution.

What's in a name?

This is one of the few names inherited from the original Litter package.

See Also

lp.norm	Generate random numbers from a normal ("Gaussian") distribution
lp.chichi	Generate random numbers from a chi-square distribution
lp.fishie	Generate random numbers from a Fisher distribution
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

The **lp.tata** object implements the Tausworthe 88 random number generator. This is currently the fastest algorithm that passes all standard statistical tests for randomness. It has a cycle of approximately 2^{88} (that's about $3 \cdot 10^{26}$) and generates random values across the entire range of 32-bit numbers (i.e., $-2{,}147{,}483{,}648 \leq x \leq 2{,}147{,}483{,}647$).

The **lp.tata** object allows you to scale the output to a given range.

Input

bang — Generate a random number and send it out the outlet.

int — In the middle inlet: set minimum value (inclusive) to generate.

In the right inlet: set the maximum value (inclusive) to generate.

seed — The symbol **seed** followed by an integer reseeds the internal random number generator.

Arguments

You can initialize an **lp.tata** object with up to three optional arguments. You must specify the first argument if you want to specify the second and you must specify the second to specify the third.

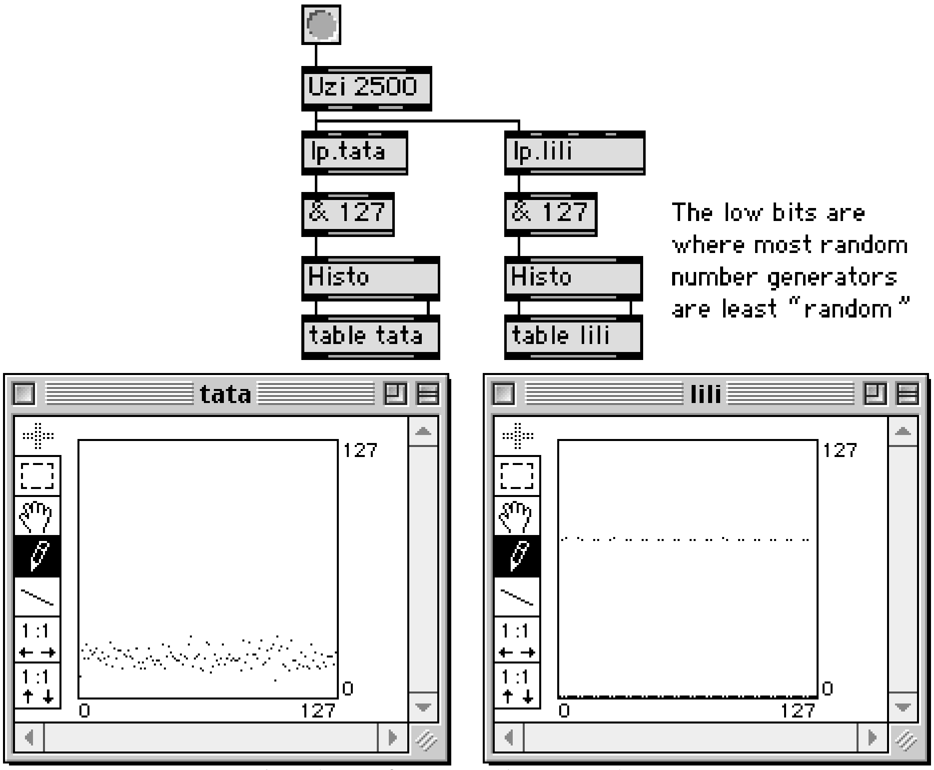
int — The first two arguments specify the range. If you specify only one argument, the range will run from zero to the value specified. If this is a negative value, the value is taken as the minimum and zero as the maximum. If the argument is a positive value, it is the maximum and zero is taken as the minimum. If you specify both of the first two arguments, the first is taken as minimum and the second as maximum. If you specify a minimum larger or equal to the maximum, no scaling will take place. This is the default situation.

A third argument, if specified, seeds the random number generator. The generator is auto-seeded if this value is zero (the default).

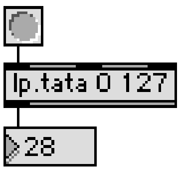
Output

int — A random number.

Examples



Compare low bits of two random number generators



As a replacement for random

What's in a name?

All basic random number generators are named based on a repeated short syllable drawn from the name commonly found in the statistical literature. The habit started with the TT800 algorithm (the first one implemented). It seemed like a nice idea at the time.

See Also

lp.lili	Parametric linear congurence method
lp.titi	Generate random numbers using the TT800 algorithm
random	Hard-wired linear congrunce pseudo-random number generator

L'Ecuyer, Pierre, "Maximally Equidistributed Combined Tausworthe Generators," *Mathematics of Computation* 65 (1996): 203-213.

The Time-domain Interval Mutator is an implementation of Larry Polansky's "Morphological Mutations" designed for mutating audio signals in the time domain.

Input

signal In 1st Inlet, the mutation source (mandatory if you want anything to happen).

In 2nd Inlet, the mutation target (mandatory if you want anything to happen)

In 3rd Inlet, a time-varying Mutation Index (defaults to float input or object argument if there is no signal). Mutation Index is limited to the range $0 \leq \Omega \leq 1$.

In 4th Inlet, a time-varying Delta Emphasis value. This defaults to float input or object argument if there is no signal. It is ignored if the object is using absolute intervals. Delta Emphasis is limited to the range $-1 \leq \delta \leq 1$.

In 5th Inlet, a time-varying Clumping Factor (defaults to float input or object argument if no signal; ignored if the object is performing a uniform mutation). Clumping Factor is limited to $0 \leq \pi < 1$. The meaning of a Clumping Factor when $\pi = 1$ is indeterminate when the length of a mutation is unknown. For practical purposes in this implementation, the maximal value is clipped to 0.9990234375, which means that you can expect an irregular mutation with a mutation index of 0.5 to change state between mutated and non-mutated forms about once every thousand samples or so.)

float In 3rd inlet: sets the Mutation Index. This is overridden if a signal is present.

In 4th inlet: sets the Delta Emphasis. This is overridden if a signal is present and ignored if the object is using absolute intervals.

In 5th Inlet, sets the Clumping Factor. This is overridden if a signal is present and ignored if the object is performing a uniform mutation.

Sending a float to either of the first two inlets elicits an error message in the Max window.

usim
isim
uuim
iuim
wcm
lcm Set the Mutation algorithm to Linear Contour Modulation, Uniform Signed Interval Modulation, etc.

rel Use relative intervals for calculating the mutant. This is the default setting. You can include a float with this message to set Delta Emphasis. The default value is zero abs — Use Absolute Intervals for calculating the mutant.

Note that, unlike interval mutation in SoundHack and other implementations, the **lp.tim~** object does not support source and target reference values. If you want to source or target intervals to be calculated against a reference other than zero, you need to send the signals through **+~**, ***~**, or other objects to suit your needs. This gives you greater flexibility and control than anything **lp.tim~** could offer.

clear — Resets the stored values of previous source, previous target, and previous mutant to 0.0. This is often helpful after a mutation has gotten chaotic.

Arguments

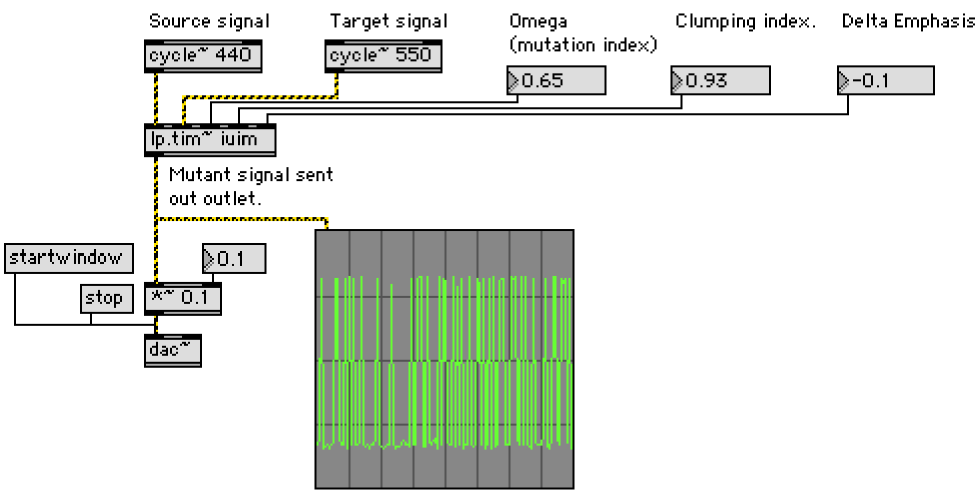
symbol — The symbols usim, isim, uuim, iuim, wcm, and lcm can be used to specify the initial mutation algorithm to use. The default is usim.

float — Up to three float arguments can be included to specify (in order) Mutation Index (Ω), Delta Emphasis (this is ignored when absolute intervals are used), and Clumping Factor (this is only used by irregular mutations). All default to 0.0.

Output

signal — Mutant signal out of the left outlet

Examples



Mutate two signals to get a surprising result

What's in a name?

Time-domain **I**nterval **M**utation.

See Also

lp.frim~	Frequency domain interval mutation
lp.vim	Interval mutation of numeric values

Polansky, Larry, "Morphological Metrics: An Introduction to a Theory of Formal Distances" (paper presented at the International Computer Music Conference, Champaign-Urbana, 1987), 197-204.

Polansky, Larry, "Morphological metrics," *Journal of New Music Research (formally Interface)* 25 (1996): 289-368.

The **lp.titi** object implements the TT800 random number generator proposed by Makoto Matsumoto and Yoshiharu Kurita. This algorithm passes all standard statistical tests for randomness. It has a cycle of 2^{800} - 1 (that's approximately $6 \cdot 10^{240}$) and generates random values across the entire range of 32-bit numbers (i.e., from -2,147,483,648 to 2,147,483,647).

The **lp.titi** object allows you to scale the output to a given range.

Input

bang — Generate a random number and send it out the outlet.

int — In the middle inlet: set minimum value (inclusive) to generate.

In the right inlet: set the maximum value (inclusive) to generate.

seed — The symbol seed followed by an integer reseeds the internal random number generator.

Arguments

int — Three optional arguments.

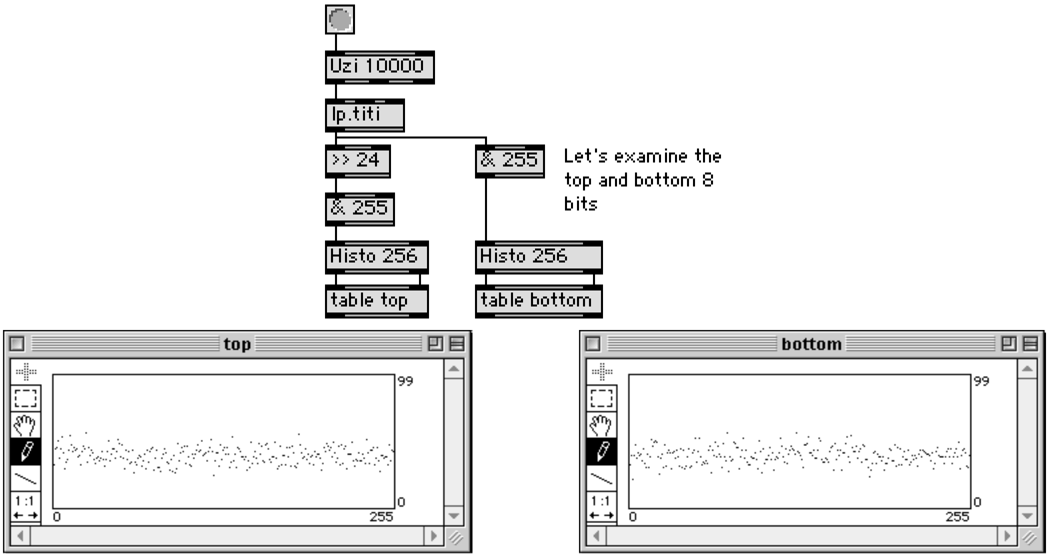
The first two arguments specify the range. If you specify only one argument, the range will run from zero to the value specified. If this is a negative value, the value is taken as the minimum and zero as the maximum; if the argument is a positive value, it is the maximum and zero is taken as the minimum. If you specify both of the first two arguments, the first is taken as minimum and the second as maximum. If you specify a minimum larger or equal to the maximum, no scaling will take place. This is the default situation.

A third argument, if specified, seeds the random number generator. The generator is auto-seeded if this value is zero (the default).

Output

int — A random number

Examples



High order and low-order bits of numbers generated with lp.titi are all random. You can also rely on the other bits to be random.

What's in a name?

See **lp.tata**.

See Also

lp.lili	Parametric linear congurence method
lp.tata	Generate random numbers using the Tausworthe 88 algorithm
random	Hard-wired linear congrunce pseudo-random number generator

Matsumoto, Makoto and Yoshiharu Kurita, "Twisted GFSR Generators II," *ACM Transactions on Modelling and Computer Simulation* 4, no. 3 (1994): 254-266.

Value Interval Mutation is an implementation of Larry Polansky's "Morphological Mutations" designed for mutating sequences of discrete pairs of source and target numbers. The numbers may be either integer or floating point.

Input

int
float

In the left inlet: computes a mutant value based on the integer received (the "source") and the current "target" (received in the second inlet). The mutant value is output through the outlet.

In the second inlet: sets the current target value

In 3rd Inlet, sets the Mutation Index (Ω).
The mutation index is constrained to the range $0 \leq \Omega \leq 1$. Note that the only valid integer values are zero and one.

In 4th Inlet, sets the Delta Emphasis (δ). This value is ignored if the object is using absolute intervals.
The mutation index is constrained to the range $-1 \leq \delta \leq 1$. Note that the only valid integer values are one, zero, and negative one.

In 5th Inlet, sets the Clumping Factor (∏). This value is ignored if the object is performing a uniform mutation.
The clumping factor is constrained to the range $0 \leq \prod < 1$. Note that the only truly valid integer value is zero. All positive integers are clipped to the maximum value for ∏. In this implementation, the maximum value is set to 0.9990234375, which means you can expect an irregular mutation with $\mu = 0.5$ to change state between mutated and non-mutated forms about once every thousand events or so.

set

The symbol set followed by an integer sets the current target value.

bang

Sends the current mutant value through the left outlet and the state of range-checking through the right outlet.

usim
isim
uuim
iuim
wcm
lcm

Set the Mutation algorithm to Linear Contour Modulation, Uniform Signed Interval Modulation, etc.

rel

Use Relative Intervals for calculating the mutant. This is the default setting. You can include a float with this message to set Delta Emphasis (default 0.0).

abs

Use Absolute Intervals for calculating the mutant.

Note that, unlike interval mutation in SoundHack and other implementations, the **lp.vim** object does not support source and target reference values. If you want to source or target intervals to be calculated against a reference other than zero, you need to send the signals through **+~**, ***~**, or other objects to suit your needs. This gives you greater flexibility and control than anything **lp.vim** could offer.

clear Resets the stored values of previous source, previous target, and previous mutant to 0. This is often helpful after a mutation has gotten chaotic.

Arguments

symbol The symbols usim, isim, uuim, iuim, wcm, and lcm can be used to specify the initial mutation algorithm to use. The default is usim.

float Up to three float arguments can be included to specify (in order) Mutation Index (Ω), Delta Emphasis (this is ignored when absolute intervals are used), and Clumping Factor (this is only used by irregular mutations). All default to 0.0.

Output

float The current mutant value.
Most integer objects will accept a float and convert by truncating any fractional part. You can use **lp.scampi** for rounding and forms of floating-point to integer conversion.

What's in a name?

Abbreviation for **I**nterval **M**utation of numeric **V**alues. The letters got shuffled around, but that happens with interval mutation a lot.

See Also

lp.scampi	Scale, offset, and limit numbers; output integers
lp.tim~	Time domain interval mutation
lp.frim~	Frequency domain interval mutation

Polansky, Larry, “Morphological Metrics: An Introduction to a Theory of Formal Distances” (paper presented at the International Computer Music Conference, Champaign-Urbana, 1987), 197-204.

Polansky, Larry, “Morphological metrics,” *Journal of New Music Research (formally Interface)* 25 (1996): 289-368.

The Weibull distribution has two parameters, generally referred to as *scale* and *curve*.

The Rayleigh distribution is a special case of the Weibull distribution.

The Weibull distribution is widely used in the study of reliability.

Input

bang	Generate a random value from a Weibull or Rayleigh distribution.
float	In the middle inlet: Set the value of the scale parameter
	In the right inlet: Set the value of the curve parameter.
seed	The symbol seed followed by an integer reseeds the internal random number generator.

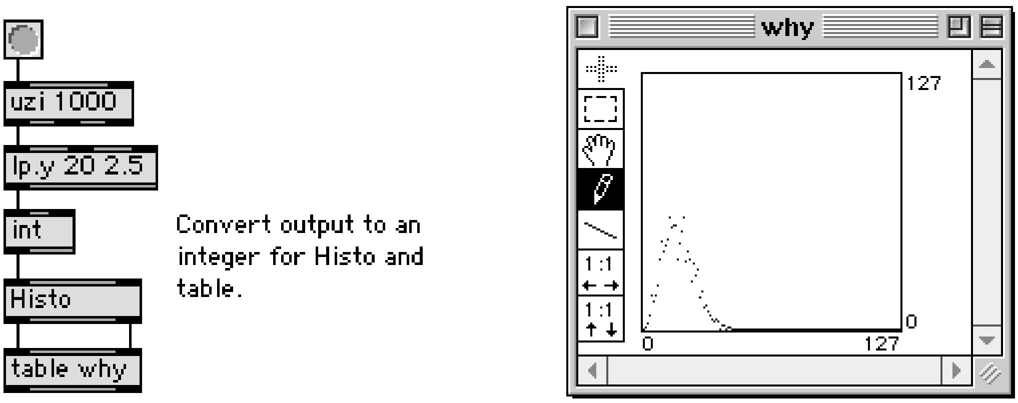
Arguments

float	The first two (optional) arguments specify initial values for the scale and curve parameters (respectively). Both parameters default to one.
seed	Set the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

float	A random value from a Weibull distribution.

Examples



Generating random numbers with a Weibull distribution

What's in a name?

Why not?

See Also

lp.tata	Generate random numbers using the Tausworthe 88 algorithm

This is a control-domain version of the **lp.zzz~** signal generator. It generates values in the range $0 \leq x \leq 1$. It is based on a variant of the classic Voss/Gardner algorithm developed by James McCartney.

Input

bang — Generate a random value with “pink” distribution.

seed — The symbol seed followed by an integer reseeds the internal random number generator.

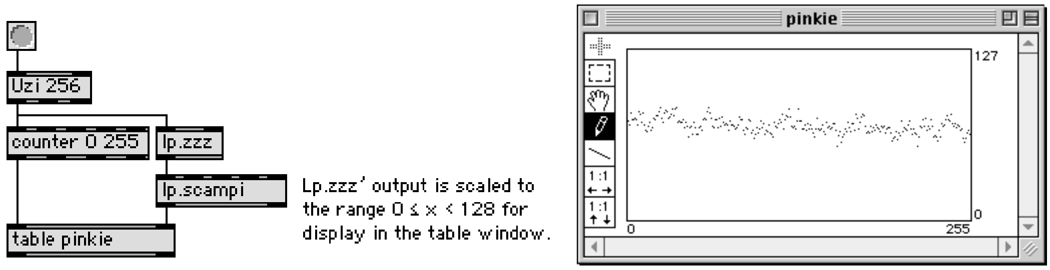
Arguments

int — Set the value for the seed of the core random number generator. The generator is auto-seeded if this value is zero (the default).

Output

float — A random value in the range $0 \leq x \leq 1$.

Examples



Generating random numbers with a 1/f (pink) distribution

What’s in a name?

See **lp.zzz~**

See Also

lp.grrr	“Gray” noise (control domain)
lp.pfff	Generate random numbers from a $1/f^2$ (”Brownian”) distribution
lp.shhh	Generate random numbers from a “white” distribution
lp.sss	Generate random numbers from a 1/f (“pink”) distribution
lp.zzz~	“Pink” noise (McCartney algorithm)
lp.scampi	Scale, offset, and limit numbers; output integers
lp.tata	Generate random numbers using the Tausworthe 88 algorithm

Pink noise generated using James McCartney's improved version of the original Voss/Gardner algorithm. McCartney's algorithm is somewhat more efficient and, perhaps more importantly, distributes processor load more evenly. Also, it is possible to prove that the algorithm produces the desired power fall-off of 3dB/octave.

Input

signal	Signal processing provided for the benefit of **begin~** / **selector~** configurations.
int	NN factor: specifies the number of low-order bits to clear before converting the integer representation to the floating-point value used in signal connections. The NN factor for the **lp.zzz~** object may be in the range $0 \le nn \le 31$.

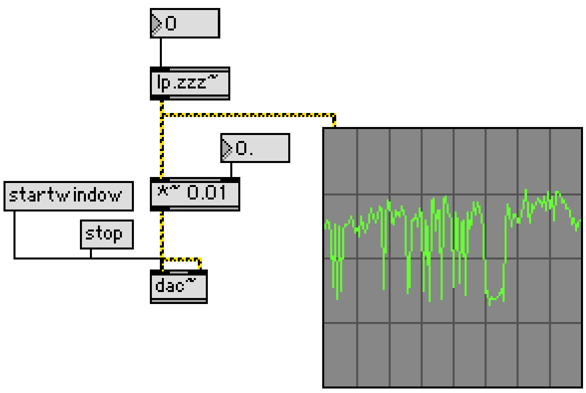
Arguments

int	Optional value to set the initial NN factor. This is zero (no masking) by default

Output

signal	Pink noise.

Examples



Generating Pink noise

What's in a name?

Onomatopoeia

See Also

lp.frrr~	Low-frequency noise
lp.grrr~	"Gray" noise
lp.lll~	Parametric linear congruence "noise"
lp.pfff~	"Brownian" ($1/f^2$) noise
lp.phhh~	"Black" ($1/f^3$) noise

lp.ppp~	Popcorn (dust) noise
lp.shhh~	White noise
lp.sss~	"Pink" noise (Voss/Gardner algorithm)
pink~	Another source of pink noise (algorithm not known)