The available random number generators are:

RNG	**Description**
1	Linear Congruential Generator by L'Ecuyer with added Bayes-Durham shuffle. The algorithm is described in *Numerical Recipes* as the function `ran2()`. This option has nearly 23^2 distinct values and the sequence of random numbers has a period in excess of 10^{18}. This RNG is provided for backward compatibility only. New code should use RNG=3 (Xoshiro256**).
2	Mersenne Twister by Matsumoto and Nishimura. It is claimed to have better distribution properties and period of 2^{19937}-1. See Matsumoto, M., and T. Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, *ACM Trans. on Modeling and Computer Simulation*, *8*, 3-30, 1998. This RNG is provided for backward compatibility only. New code should use RNG=3 (Xoshiro256**).
3	Xoshiro256** by David Blackman and Sebastiano Vigna. It has a period of 2^{256}-1, is 4 dimensionally equidistributed and passes all statistical tests at the time of writing. For technical details, see "Scrambled linear pseudorandom number generators", 2019 (http://vigna.di.unimi.it/ftp/papers/ScrambledLinear.pdf).

Use of enoise With Complex Numbers

enoise returns a complex result if *num* is complex or if it is used in a complex expression.

```
Function DemoENoiseComplex()
   // enoise(rv) in a complex expression returns a complex result
   // and is equivalent to cmplx(enoise(rv),enoise(rv))
   Variable rv = 1                  // A real variable
   SetRandomSeed 1
   Variable/C cv1 = enoise(rv)      // Real parameter, complex result
   SetRandomSeed 1
   Variable/C cv2 = cmplx(enoise(rv),enoise(rv))     // Equivalent
   Print cv1, cv2

   // enoise(cv) is equivalent to cmplx(enoise(real(cv)),enoise(imag(cv)))
   Variable/C cv = cmplx(1,1)       // A complex variable
   SetRandomSeed 1
   Variable/C cv3 = enoise(cv)      // Complex parameter, complex result
   SetRandomSeed 1
   Variable/C cv4 = cmplx(enoise(real(cv)),enoise(imag(cv)))   // Equivalent
   Print cv3, cv4
End
```

Example

```
// Generate uniformly-distributed integers on the interval [from,to] with from<to
Function RandomInt(from, to)
   Variable from, to
   Variable amp = to - from
   return floor(from + mod(abs(enoise(100*amp)), amp+1))
End
```

See Also

SetRandomSeed, **gnoise**, **Noise Functions**, **Statistics**

EqualWaves

EqualWaves(*waveA*, *waveB*, *selector* [, *tolerance*])

TheEqualWaves function compares *waveA* to *waveB*. Each wave can be of any data type. It returns 1 for equality and zero otherwise.

Use the *selector* parameter to determine which aspects of the wave are compared. You can add *selector* values to test more than one field at a time or pass -1 to compare all aspects.