7 Random numbers

Functions:

ran0: random deviates, improve an existing generator
ran1: random deviates, uniform
ran2: random deviates, uniform
ran3: random deviates, uniform subtractive method
expdev: random deviates, exponential
gasdev: random deviates, normally distributed Box-Muller method
gamdev: random deviates, gamma-law distribution
poidev: random deviates, Poisson distributed
bnldev: random deviates, binomial distributed
irbit1: generate a random bit sequence
irbit2: generate a random bit sequence
ran4: random deviates, uniform using Data Encryption
des: encryption
ks: encryption key standard
cyfun: cyfer function for the Data Encryption Standard

---

ran0

Description:

This function improves the internal Common Lisp random number generator "random".

ran0(idum) ⟶ value

Where:

idum is an integer representing the seed which should be set to any negative number to initialize or reinitialize the generator,
value is a floating point number between 0.0 and 1.0 being the random deviate.

Example:

Four values of `ran0` are computed and then the value of π using a Monte Carlo method.

```
-> idum : -11;
-> ran0(idum)?
0.095095580
-> ran0(idum)?
0.095095580
-> ran0(6)?
0.258454014
-> ran0(6)?
0.227311250
-> Statistically calculate pi, using ratio of areas: circle/square #
-> dist(x,y) := x*x + y*y;
-> prop : 0;
-> for i to 100 do
  if dist(ran0(-idum),ran0(-idum)) < 1.0 then prop : prop + 1;
-> 4*prop/100.0?
2.840000000
-> pi?
3.141592654
```

ran1

Description:

This function returns a random number chosen from the uniform distribution using three independent congruential generators.

ran1(idum) $\longrightarrow$ **value**

Where:

idum is an integer representing the seed which should be set to any negative number to initialize or reinitialize the generator,
value is a floating point number between 0.0 and 1.0 being the random deviate.

Example:

Four random numbers are computed using `ran1` and then a Monte Carlo method used to compute the area of the unit circle.

```
-> idum : -11;
-> ran1(idum)?
0.848440189
-> ran1(idum)?
0.848440189
-> ran1(3)?
0.248953574
-> ran1(3)?
0.926415854
-> Statistically calculate pi, using ratio of areas: circle/square #
-> dist(x,y) := x*x + y*y;
-> prop : 0;
-> for i to 100 do
  if dist(ran1(-idum),ran1(-idum))<1.0 then prop : prop + 1;
-> 4*prop/100.0?
3.240000000
-> pi?
3.141592654
```

ran2

Description:

This function returns a random number chosen from the uniform distribution using one congruential generator and the shuffling technique used by the function **ran0**.

ran2(idum) ⟶ value

Where:

idum is an integer representing the seed which should be set to any negative number to initialize or reinitialize the generator,
value is a floating point number between 0.0 and 1.0 being the random deviate.

Example:

Five random numbers are computed using `ran2` and then the area of the unit circle is computed.

```
-> idum : -11;
-> ran2(idum)?
0.773019159
-> ran2(idum)?
0.773019159
-> ran2(5)?
0.492419738
-> ran2(5)?
0.769618718
-> ran2(9)?
0.220887226
-> Statistically calculate pi, using ratio of areas: circle/square #
-> dist(x,y) := x*x + y*y;
-> prop : 0;
-> for i to 100 do
  if dist(ran2(-idum),ran2(-idum))<1.0 then prop : prop + 1;
-> 4*prop/100.0?
4.000000000
-> pi?
3.141592654
```

ran3

Description:

This function returns a random number chosen from the uniform distribution using the subtractive method.

ran3(idum) $\longrightarrow$ value

Where:

idum is an integer representing the seed which should be set to any negative number to initialize or reinitialize the generator,
value is a floating point number between 0.0 and 1.0 being the random deviate.

Example:

Four random numbers are computed using `ran3` and then the area of the unit circle estimated.

```
-> idum : -11;
-> ran3(idum)?
0.941262328
-> ran3(idum)?
0.941262328
-> ran3(1)?
0.636222818
-> ran3(1)?
0.352192397
-> Statistically calculate pi, using ratio of areas: circle/square #
-> dist(x,y) := x*x + y*y;
-> prop : 0;
-> for i to 100 do
  if dist(ran3(-idum),ran3(-idum))<1.0 then prop : prop + 1;
-> 4*prop/100.0?
3.120000000
-> pi?
3.141592654
```

expdev

Description:

This function returns a random number from the exponential distribution, positive and with unit mean, using **ran1** as the source of random deviates.

expdev(idum) $\longrightarrow$ value

Where:

idum is an integer representing the seed,
value is a floating point number being the random deviate.

Example:

Four random numbers are computed using `expdev` and then a comparison made with a normalized exponential distribution.

```
-> ee : float(%e)?
2.718281828
-> idum : -50;
-> expdev(idum)?
0.464043856
-> expdev(idum)?
0.464043856
-> expdev(1)?
0.055540669
-> expdev(1)?
0.265924209
-> x : const(0,20);
-> for i to 50 do
  sprog(y : expdev(i),
    for j to 20 do if (y>j/20.0) and (y<(j+1)/20.0) then
      x : put(x,j,proj(x,j)+1));
-> total : 0;
-> for i in x do total : total + i;
-> 1.0/total*x?
[0.064516130, 0.064516130, 0.161290320, 0.032258064, 0.096774190, 0.096774190,
0.064516130, 0.129032250, 0.064516130, 0.000000000, 0.064516130, 0.000000000,
0.129032250, 0.000000000, 0.000000000, 0.000000000, 0.000000000, 0.032258064,
0.000000000, 0.000000000]
-> 0.05*ee/(ee-1)*map(exp,-1.0*listn(1,20))?
[0.029098837, 0.010704863, 0.003938099, 0.001448746, 0.000532964, 0.000196066,
0.000072129, 0.000026535, 0.000009762, 0.000003591, 0.000001321, 0.000000486,
0.000000179, 0.000000066, 0.000000024, 0.000000009, 0.000000003, 0.000000001,
0.000000000, 0.000000000]
```

gasdev

Description:

This function returns a random number chosen from the normal distribution using the Box-Muller technique. The random deviates have zero mean and unit variance and use **ran1**.

gasdev(idum) ⟶ value

Where:

idum is an integer representing the seed,
value is a floating point number being the random deviate.

Example:

A crude plot of the normal distribution is formed using `gasdev`.

```
-> gasdev(-900)?
-1.670149551
-> Crude plot of output: #
-> unbind(x);
-> npts : 50; sample size = 50 (say) #
-> dist : const(0,20);
-> for i to npts do
  sprog(y : trunc(gasdev(i)) + 1,
    if (y>=1) and (y=<21) then dist : put(dist,y,proj(dist,y)+1));
-> for i to 20 do print(const(x,proj(dist,i)));
[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x,
x, x, x, x, x, x, x, x, x, x]
[x, x, x, x]
[x]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
```

gamdev

Description:

This function returns a random number chosen from the gamma-law distribution using **ran1** as the source of random deviates.

gamdev(ia, idum) ⟶ value

Where:

ia is an integer representing the integer order, a waiting time to the *ia*'th event in a Poisson process of unit mean,
idum is an integer representing the seed,
value is a floating point number being the random deviate.

Example:

A crude plot of the gamma distribution is formed using `gamdev`.

```
-> gamdev(2,-1200)?
3.949475326
-> unbind(x);
-> ia : 3; integer order = 3 (say) #
-> npts : 50; sample size = 50 (say) #
-> dist : const(0,20);
-> for i to npts do
  sprog(y : trunc(gamdev(ia,900)) + 1,
    if (y>=1) and (y=<21) then dist : put(dist,y,proj(dist,y)+1));
-> for i to 20 do print(const(x,proj(dist,i)));
[x, x, x, x, x, x]
[x, x, x, x, x, x, x, x, x]
[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
[x, x, x, x, x, x, x, x, x]
[x, x, x, x, x]
[x, x]
[x]
[x]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
```

poidev

Description:

This function returns a random number chosen from the Poisson distribution.

poidev(xm, idum) ⟶ value

Where:

xm is a real number being the mean of the Poisson distribution,
idum is an integer representing the seed,
value is a floating point number with an integer value being the random deviate.

Example:

A crude plot of the Poisson distribution is formed using `poidev`.

```
-> poidev(1.0,-300)?
2.000000000
-> unbind(x);
-> xm : 3; poisson mean = 3 (say) #
-> npts : 50; sample size = 50 (say) #
-> dist : const(0,20);
-> for i to npts do
  sprog(y : trunc(poidev(xm,300)) + 1,
    if (y>=1) and (y=<21) then dist : put(dist,y,proj(dist,y)+1));
-> for i to 20 do print(const(x,proj(dist,i)));
[x, x]
[x, x, x, x, x, x, x, x, x, x]
[x, x, x, x, x, x, x, x, x]
[x, x, x, x, x, x, x]
[x, x, x, x, x, x, x, x, x, x, x, x]
[x, x, x, x, x, x, x]
[x, x, x]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
```

bnldev

Description:

This function returns a random number chosen from the binomial distribution and using **ran1** as the source of uniform random deviates.

bnldev(pp, n) ⟶ **value**

Where:

pp is a real number being the probability of success of an individual trial in the binomial distribution,
n is an integer representing the number of trials in the binomial distribution,
value is a floating point number representing the integral number of successes in a random trial.

Example:

A crude plot of the binomial distribution is formed using `bnldev`.

```
-> bnldev(0.5,3,-300)?
0.000000000
-> unbind(x);
-> pp : 0.3; (say) #
-> n : 20; (say) #
-> npts : 50; (say) #
-> dist : const(0,20);
-> for i to npts do
  sprog(y : trunc(bnldev(pp,n,400)) + 1,
    if (y>=1) and (y=<21) then dist : put(dist,y,proj(dist,y)+1));
-> for i to 17 do print(const(x,proj(dist,i)));
[]
[]
[x]
[x, x, x]
[x, x, x, x, x]
[x, x, x, x, x, x, x, x, x]
[x, x, x, x, x, x, x, x, x]
[x, x, x, x, x, x, x]
[x, x, x, x, x, x, x, x, x, x, x]
[x, x, x]
[x]
[x]
[]
[]
[]
[]
[]
```

irbit1

Description:

This function generates a random bit sequence by returning as an integer a random bit based on the 18 low-significant bits in an internal value of *iseed* which is modified for the next call in the sequence.

irbit1() ⟶ value

Where:

value is an integer being the random bit.

Example:

The start of a random bit sequence is computed using `irbit1`.

```
-> irbit1()?
0
-> irbit1()?
0
-> irbit1()?
1
```

---

irbit2

Description:

This function generates a random bit sequence based on the 18 low-significant bits in an internal value of *iseed* which is modified for the next call in the sequence.

irbit2() ⟶ value

Where:

value is an integer being the random bit.

Example:

The start of a random bit sequence is computed using `irbit2`.

```
-> irbit2()?
1
-> irbit2()?
0
-> irbit2()?
1
```

---

ran4

Description:

This function returns a random number chosen from the uniform distribution using the Data Encryption Standard. This routine is too slow to be of practical use.

$$\textbf{ran4(idum)} \longrightarrow \textbf{value}$$

Where:

idum is an integer representing the seed which should be set to a negative number to initialize or reinitialize the generator,
value is a floating point number between 0.0 and 1.0 being the random deviate.

Example:

Two random numbers are computed using `ran4`. Note that this algorithm is very slow indeed.

```
-> ran4(-900)?
0.425257027
-> ran4(-600)?
0.673944294
```

des

Description:

This function performs encryption using the Data Encryption Standard.

des(input, key, newkey, isw) ⟶ jotput

Where:

input is a list of 64 “bits” represented by 0 or 1 being the data to encypt,
key is a list of 64 “bits” represented by 0 or 1 being the encryption key to be used,
newkey is an integer being 1 when the key is new,
isw is an integer being 0 for encryption and 1 for decryption,
jotput is an integer list of length 64 being the encypted/decrypted message.

Example:

A sequence of 64 ”bits” is encrypted using a given key.

```
-> input : [0,1,1,0,0,1,0,0,1,0,1,1,1,0,0,1,
            0,1,0,0,0,1,1,1,0,0,1,0,0,0,1,0,
            1,1,0,1,1,1,0,0,1,0,0,1,1,1,0,0,
            1,0,0,0,1,0,1,1,0,1,1,1,0,0,1,0];
-> key :   [0,1,1,0,0,1,0,0,1,0,1,1,1,0,0,1,
            0,1,0,0,0,1,1,1,0,0,1,0,0,0,1,0,
            1,1,0,1,1,1,0,0,1,0,0,1,1,1,0,0,
            1,0,0,0,1,0,1,1,0,1,1,1,0,0,1,0];
-> newkey : 1;
-> isw : 0;
-> des(input,key,newkey,isw)?
[0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1,
1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1,
0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0]
```

ks

Description:

This function provides a key schedule for the Data Encryption Standard. The function must be called with n having successive values, 1 through 16 inclusive.

ks(key, n) ⟶ kn

Where:

key is a list of 64 "bits" represented by 0's or 1's,
n is an integer between 1 and 16 inclusive,
kn is an integer list of length 48 being the key schedule.

Example:

A key schedule for encryption is computed using `ks`.

```
-> key : [0,1,1,0,0,1,0,0,1,0,1,1,1,0,0,1,
          0,1,0,0,0,1,1,1,0,0,1,0,0,0,1,0,
          1,1,0,1,1,1,0,0,1,0,0,1,1,1,0,0,
          1,0,0,0,1,0,1,1,0,1,1,1,0,0,1,0];
-> ks(key,1)?
[0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0,
0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1]
-> ks(key,2)?
[1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0,
1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1]
-> ks(key,3)?
[0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0,
0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0]
```

cyfun

Description:

This function provides a cipher function for the Data Encryption Standard.

$$\textbf{cyfun(ir, k)} \longrightarrow \textbf{iout}$$

Where:

ir is an integer list of length 32,
k is an integer list of length 48,
iout is an integer list of length 32 being the cypher function of *ir* and *k*.

Example:

A cypher function is computed using `cyfun`.

```
-> ir : [0,1,1,0,0,1,0,0,1,0,1,1,1,0,0,1,
         0,1,0,0,0,1,1,1,0,0,1,0,0,0,1,0];
-> k :[0,1,1,0,0,1,0,0,1,0,1,1,1,0,0,1,
       0,1,0,0,0,1,1,1,0,0,1,0,0,0,1,0,
       1,1,0,1,1,1,0,0,1,0,0,1,1,1,0,0];
-> cyfun(ir,k)?
[1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1,
0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```