Simulation

Random numbers

Random numbers

- ”Anyone who considers arithmetic methods of producing random digits is, of course, in a state of sin”, John v. Neumann
- Only seemingly random (pseudo random numbers) are used in simulation
- Random numbers should be
  - Reproducable and efficiently generated
  - Reflect the desired properties of the intended truly random sequence (apparent independency, statistics)
- Intended use dictates which features are important

History

- Need to generate random numbers boomed after invention of computers
  - Simulation of nuclear reactions, Los Alamos
- Simplicity and computational efficiency were emphasized in the beginning
  - Simple arithmetics, simple parameters
- Later portability and quality issues
  - Efficient implementation with high level languages
  - Statistical properties

Generation of random numbers

- Divided in two stages
  - Generation of Uniform (0,1) random numbers
    - Generate uniformly (0,m-1) distributed integers and divide with m
    - Requires deep analysis for statistical properties
  - Generation of random numbers with given probability density function
    - Is done using Unif(0,1) random streams
    - Mainly a technical exercise

Modelling of randomness

- Consider generation of pseudo random numbers as a case of simulation.
  - We go through the steps of simulation modelling process

Modelling randomness

- Recognition of the system/problem
    - Which statistical properties of a truly random sequence we have to reproduce?
    - Right probability density function (easy part)
    - Sufficient (!) statistical independence between sampled values
    - Long enough sequences
    - Case: Sequences of millions of independent Unif(0,1) random numbers

Modelling randomness

- Model design
  - System components and their interactions
  - Deterministic model with fixed parameters, (large but finite) state that is updated and fixed transform for output
  - $X_n = F(X_{(n-1)})$, $R_n = f(X_n)$
- Data collection and parameter estimation
  - Not relevant for U(0,1)

Lehmer generator

- Developed in 40s (D Lehmer) for first computers (Eniac)
- Basic operations: addition, multiplication and taking reminder
  - X= (a X+ c) mod m, R=X/m
  - Parameters a, c and m influence the properties of the sequence
  - Original generator was implemented as a separate physical unit. Random stream was read when needed (-> additional randomness)

Lehmer generator

- Original Eniac generator
    - m= 10^8 +1
    - A= 23
    - C= 0
  – Simple and efficient to implement

Lehmer generator

- – Next X is uniquely defined from the previous value.
  - Sequence starts to repeat at first reoccurence of X
  - Domain of X:n defines the theoretical maximum for the length of sequence (=m)
- – Conditions for reaching the maximum cycle are known
  - If q divides m (being prime or 4), a-1 =0 mod q
  - C and m have no common divisors (and c is nonzero)

Modelling randomness

- Software design
  - Description model structures and interaction patterns
    - Set up phase and iterator delivering the next instance
- Software implementation
  - Actual programming of the simulator
    - Portability + handling the intermediate large integers
- Software testing
  - Debugging

Lehmer generator

```
real(dp),parameter :: m=2._dp**31-1._dp

m_1=1._dp/m
a=16807._dp

real(wp) function random()
seed=modulo(seed*a,m)
random=seed*m_1
return
end function random
```

Modelling randomness

- Model validation
  - Qualitative/quantitative analysis of the model (comparisons to observation, intuitive expectations, simplified test cases, dependency of uncertain parameters)
  - Counter example (mid square)
- Model experimentation
  - Does the sequence appear as random?
  - In what sense we can prove that the sequence is valid (for our purposes)?
  - What kind of experiments are needed?

Mid square method

```
integer,parameter :: m0=100,m1=10000
integer :: seed

real function random()
seed=seed*seed
seed=seed/m0
seed=modulo(seed,m1)
random=real(seed)/real(m1)
return
end function random
3456
0.9439 9.47000E-02 0.8968 0.425 6.25000E-02 0.3906 0.2568 0.5946
   0.3549 0.5954 0.4501 0.259 0.7081 0.1405 0.974 0.8676 0.2729
   0.4474 1.66000E-02 2.75000E-02 7.56000E-02 0.5715 0.6612 0.7185
   0.6242 0.9625 0.6406 3.68000E-02 0.1354 0.8333 0.4388 0.2545
   0.477 0.7529 0.6858 3.21000E-02 0.103 6.09000E-02 0.3708 0.7492
   0.13 0.69 0.61 0.21 0.41 0.81 0.61 0.21 0.41 0.81 0.61 0.21
```

Model validation

- ”All models are wrong but some may still be useful”
  - We can not prove models to be ”right”
  - Goal is to find models that resist our attempts to prove them wrong (in given regime at least)
  - For stochastic models the basic technique is hypothesis testing

Testing of randomness

- Easy tests
  - Test distribution of x_i under condition $x_{(i-1)}$ from [a,b]
  - Test distribution of k successive values within the unit cube of R^k or distribution of $\max(x_i,\dots,x_{(i+k-1)})$ in R.
  - Try these to original Lehmer generator

Testing of randomness

– More elaborated tests
  - See Knuth vol II for history
  - DIEHARD (classical test pattern from 1995, see http://www.phy.duke.edu/~rgb/General/rand_rate.php)
  - Big Crush (collection of 100+ tests, see http://www.iro.umontreal.ca/~simardr/testu01/tu01.html for tutorial + software downloads)

Lehmer generator

– Popular basic generators in practice
– Conceptually simple arithmetics
– 2^31-1 (maxint) is prime
– Portable implementation simple (using double precision arithmetics and small a if 64 bit integers are not supported)
– Well studied and known
  - Too short cycle for modern needs

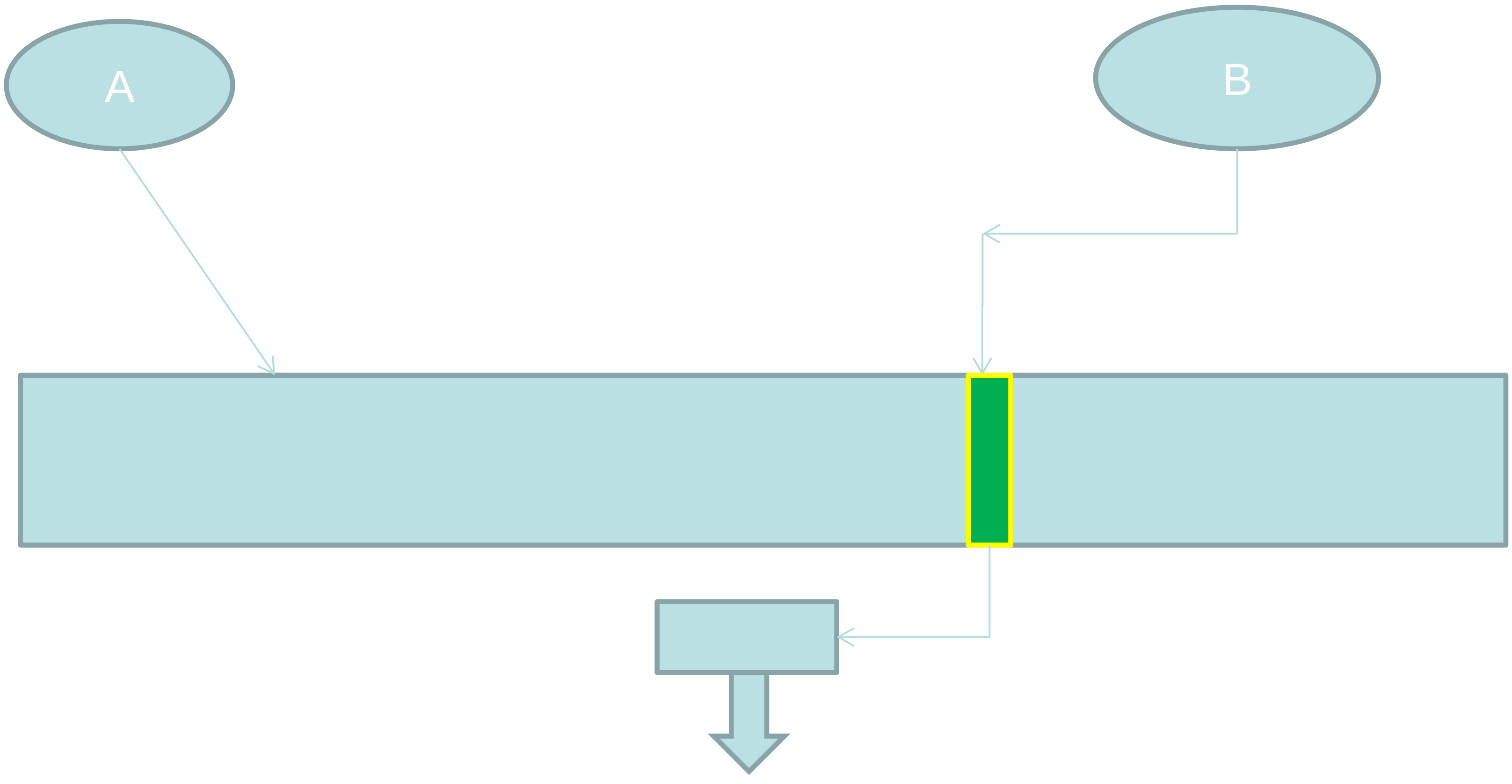
Combined generators

- Developed in the era of 16-bit processors, (Wichman-Hill)
- Uses several generators with short cycles
  - Take cycles m_1, m_2 ja m_3
  - Produce streams X_i and U_i= X_i/m_i
  - Set U= U_1+U_2+U_3 mod 1
- With appropriate choices the cycle is m_1*m_2*m_3
  - Fully standard (32-bit) arithmetics (if m_i<2^14)

Shuffled generators

- Used both for longer cycles and reduced serial correlation
  - Generate random numbers with method A to a table
  - Using generator B to select value from the table (for output) and replace it with new value from A
  - Requires an initialization, some memory and two random numbers for each output value
  - Cycle can be longer (but how much)

Shuffled generator



Modern RNGs

- Current de facto standard is Mersenne Twister
  - Developed at late1990s
  - Very long cycle (2^ 19937 -1)
  - Needs a working memory (and initialization) of 624-words
  - Available for several languages
  - Some serial correlation problems
    - Slow escape of ”zero state”

Mersenne twister

- The main ideas
  - X_(N+1) = F(X_N,…, X_(N-623))
    - ”State vector” has 624*32 = 19968 bits
    - Theoretical maximal cycle would go through all states
    - Ruling out some bits of X_(N-623) and the zero state from possible states we get the wanted length of theoretical maximal cycle (Mersenne prime which gives the name)

Mersenne twister

- We need an F, that
  - Is computationally light
  - Leads to reaching the maximal cycle
- Can be found in the family of
  - $X_{(N+1)} = X_N * A_0 + \dots X_{(N-k)} * A_k$
  - A_i:s are coefficient matrices
  - The family has theory for maximum cycles
  - Found F with only three A:s with non zero values
    - I.e. only three distinct old X values are used on each round.

Mersenne twister

- Method produces a very long cycle
- Is computationally relatively light
- Serial correlation has to be addresed
  - This can be affected shuffling bits in the output
  - Use Y=BX as output (B permutates the least correlated bits to be the most significant)
- More recent versions (WELL) with improved serial correlation available

Xorshift generators

- Simple generators based on efficient bit-level shift and XOR operations
  - Marsaglia (2003)
  - Three successive right/left shifts and XORs
  - Full cycle for selected parameters, good properties
  - Standard int/long operations for 32/64 bits

```
y ^= (y<<13); y ^= (y>>17); return y ^= (y<<5);
```

  - For longer cycles few ints needed

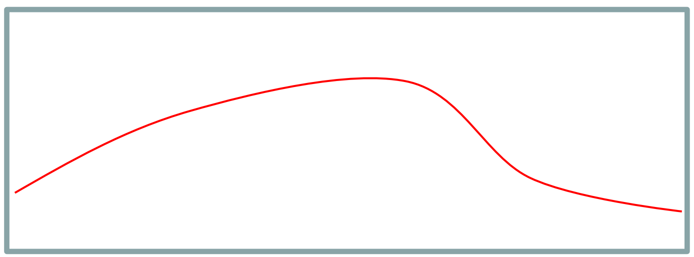
```
tmp=(x^(x<<15)); x=y; y=z; z=w;
return w=(w^(w>>21))^(tmp^(tmp>>4));
```

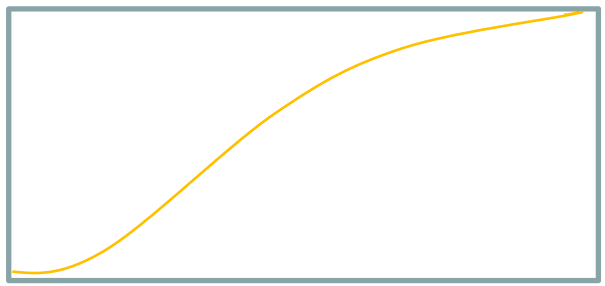
Summary

- Generation of random numbers has over 60-years of history
  - Tested and known generators are available
  - Don't try to do it yourself
  - Do not use unknown and undocumented generator (details, references missing) without testing (vs the ”secret” generator of IBM PC:s Basic language)
  - You have to understand the generator to make controlled replications
    - Initialization, ensuring independent streams

Random numbers and probability distributions

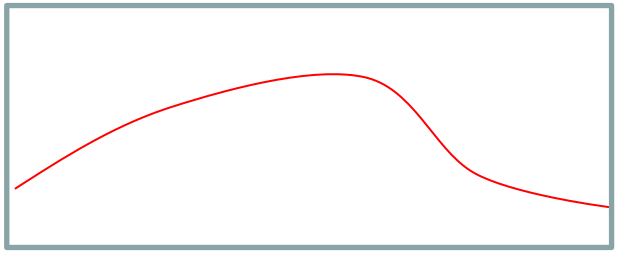
- How to generate random numbers with given probability distribution function (pdf).
- Method of inverse probability
  - Let f be a given pdf. It has a cumulative probability function F: x-> (0,1).

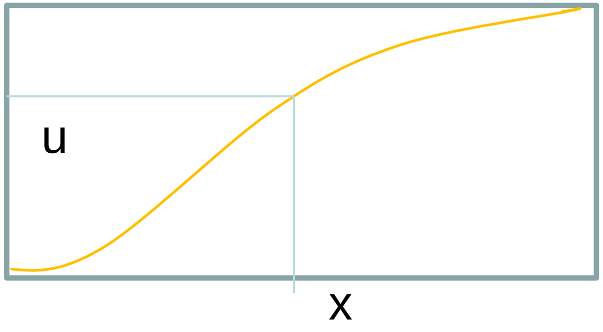




Inverse probability method

- Pick u from Unif (0,1)
- Set x = F^(-1) (u).
- Pdf of x is f.
- We have to know F^(-1) in closed form



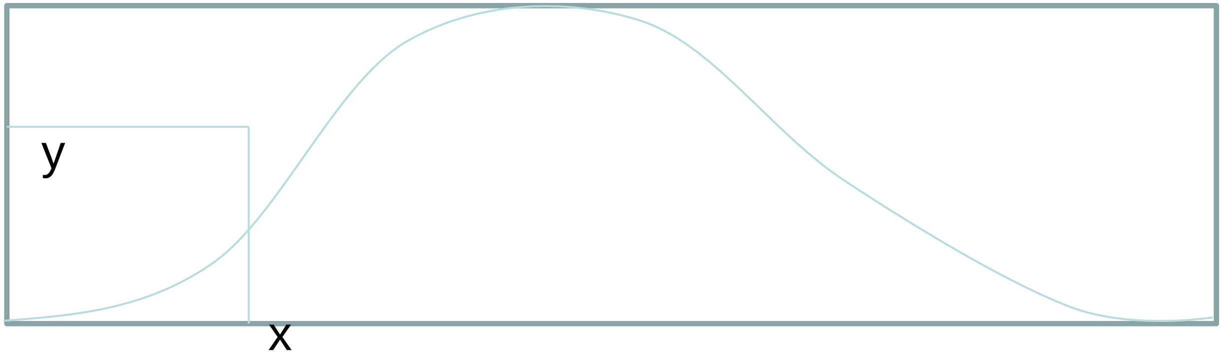


Inverse probability method

- Consider the exponential distribution
  - Pdf f. is f(x) = a e^(-ax)
  - Cumulative pf is F(x) = 1- e^(-ax)
  - So F^(-1) (U) = - ln(1-U)/a
  - Numbers obeying exponential pdf are obtained generating U ~ Unif(0,1) and reporting
    - Either –ln(1-U)/a
    - Or –ln (U)/a if U>0 always

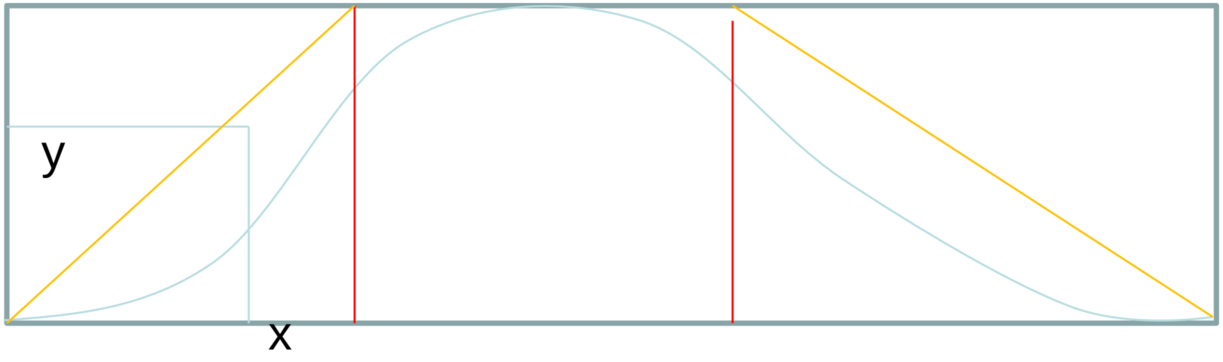
Elimination method

– General method that requires only pdf values
  - Let f be a pdf supported on (a,b) with values 0<f<c.
  - Pick x in Unif(a,b), y in Unif(0,c).
  - If y< f(x), accept x.
  - Else reject x and pick new values for x,y



Elimination method

– Method is most efficient when there is least amount of rejections
  - One can divide (a,b) to subintervals and/or change the pdf of y to approximate f better.
  - If f< cg (on some subinterval), g is a known pdf, pick x from g-distribution and y from Unif(0, cg(x))



Elimination method

– When using subintervals
  - First one has to draw which subinterval to select for x (probabilites computed beforehand)
  - Then draw x from g corresponding to subinterval and y Unif(0,cg(x)) and test for y<f(x).
  - Subdivision of interval can be an art (Marsaglia, cf Knuth vol II)