Theory

- Introduction to simulation
- Flow of simulation -- generating process realizations
- Random number generation from given distribution
  - Introduction and generation of pseudo random numbers
  - Different methods for generating random numbers with a given distribution
  - Simulating the Poisson process
- Collection and analysis of simulation data
- Variance reduction techniques

Drawing a random number from a given distribution

Generation of random variables is too important a task to be left for chance!

- *Good statistics* (i.e. obeys the distribution) and independence
  - excludes “drawing from the hat”
- *Repeatable sequence*
  - excludes real lottery or the use of physical random processes (such as radioactivity)
- *Long sequence of different numbers*
  - excludes the use of precomputed tables
- *Fast generation*
  - excludes the use of certain digits of π or e or some other well-known number

Drawing a random number from a given distribution

- Based on the generation of so called **(pseudo) random numbers**
  - aims at generating independent random numbers from the uniform distribution U(0,1)
- Given that random numbers from U(0,1) are available, random numbers with any distribution can be generated by using an appropriate “transformation” method (more on these later…)
- Programming languages, libraries and tools supporting simulation usually have routines available for the generation of random numbers from the most common distributions

Generation of random numbers

- A random number generator refers to an algorithm which produces a sequence of (apparently) random integers Z_i in some range 0,1,…,m-1
  - the sequence is always periodic (one aims at as long a period as possible)
  - strictly speaking, the numbers generated by an algorithm are not random but on the contrary deterministic; the sequence just looks random, i.e. is pseudorandom
  - in practice, such numbers will do provided that the generation is done "carefully"
- The "randomness" of the generated numbers have to be checked by appropriate statistical tests
  - the uniformity of the distribution in the range {0,1,…,m-1}
  - the independence of the generated numbers (often just uncorrelatedness)
- Simplest algorithms are so called *linear congruental generators*. A subclass of these are formed by the so called multiplicative congruential generators.
  - In both methods, each generated random number is determined by the previous one through a deterministic mapping, $Z_{i+1} = f(Z_i)$.
- Other methods:
  - more general congruential generators, where $Z_{i+1} = f(Z_i, Z_{i-1},\ldots)$
  - Mersene-Twister: a recent generator with huge period, large linear-feedback register

Linear congruential generator (LCG)

- Linear congruential random number generator produces random integers Z_i in the range {0,1,…,m-1} using the following formula (the period is at most m):

$$\boxed{Z_{i+1} = (aZ_i + c) \bmod m}$$

  - The generator is fully defined given the parameters a, c and m
  - In addition, one needs the seed Z_0
  - The parameters must be chosen carefully; otherwise the sequence can be anything but random
- Under some conditions the sequence has the maximum period m
  - for instance, m of the form 2^b, c odd, a of the form 4*k +1 (k is any integer > 0)
  - note that even with a full period the quality of the sequence may not be good
  - in practise, the values of a, c and m are defined in the implementation and user can only choose the initial seed Z_0
- Used in standard C/C++ library
  - rand() has typically sequence length 2^32 (~ $4*10^9$ numbers, which is not that much)
  - drand48() has sequence length 2^48 (~ $2*10^{14}$)

Multiplicative congruential generator (MCG)

- Multiplicative congruential random number generator produces integers Z_i in the range {0,1,…,m-1} using the following formula:

$$\boxed{Z_{i+1} = (aZ_i) \bmod m}$$

- This is a special case of LCG with the choice c = 0
- The algorithm is fully defined given the parameters a and m
- In addition, one needs the seed Z_0
- Also now, the parameters have to be chosen carefully in order to get pseudo random numbers
- By the choice m = 2^b (with any b) the period is at most 2^(b-2)
- In the best case the period can be m - 1
  - for instance when m is prime and a is chosen appropriately

Mersenne-Twister generator[1]

- Current state-of-the-art generator implemented in many simulation tools
  - Period length 2^19937 – 1, excellent independence properties
- Idea is based on using a cache of initial pseudo-random numbers with depth *n* (each 32 bits) and applying the following linear recurrence equation to generate number *k*+*n* (| = bitwise OR, ⊕ = bitwise XOR)

$$\mathbf{x}_{k+n} := \mathbf{x}_{k+m} \oplus (\mathbf{x}_k^u | \mathbf{x}_{k+1}^l)A, \quad (k = 0, 1, \cdots)$$

  - w : word length (e.g., $w = 32$ bits)
  - n : degree of recurrence (size of “number cache“, e.g., $n = 624$)
  - m : constant, $1 \leq m \leq n$ (e.g., $m = 297$, controls degree of parallel sequences)
  - $\mathbf{x}^u$: upper r bits of $\mathbf{x}$
  - $\mathbf{x}^l$: lower $(w - r - 1)$ bits of $\mathbf{x}$
  - A : rotation matrix chosen so that matrix multiplication requires only bit-shift operations
- Finally, another transformation is done to produce the pseudo-random number
  - Again this is actually a matrix-multiplication operation, but the elements are chosen so that it can be done with shift operations

[1] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 (1998)

Generation of random numbers from the uniform distribution U(0,1)

- If a (pseudo) random number generator produces a (pseudo) random integer I in the range {0,1,…,M-1}, then the normalised variable U = I/M approximately obeys the uniform distribution U(0,1) in the range (0,1)

The use of the seed numbers
(depending on the rand-algorithm used some of the instructions are relevant / irrelevant)

- Don’t use the seed 0
- Avoid even values
- Don’t use one random number sequence for several purposes
  - e.g. if the numbers (u_1, u_2, u_3, …) have been used to generate interarrival times, the same sequence should not be used to generate service times
- When using many sequences (streams), make sure the sequences don’t overlap
  - if the seed of one sequence is included in another sequence, then from that point on the sequences are identical
  - if you have to generate 10000 interarrival times and 10000 service times, the sequence of random numbers (u_1, u_2,...,u_{10000}) used for the generation of interarrival times can be computed starting from the seed u_0
  - service times can be generated using the numbers (u_{10001},…,u_{2000}), i.e. the seed is u_{10000}
  - if the sequences are not stored but the numbers are generated as they are needed you have to figure out the seed u_{10000} in advance by computing all the numbers (u_1, u_2,...,u_{10000}); these have to be recomputed during the simulation for the generation of the first sequence
- In practise, within a single simulation run by using a single stream of random numbers one does not need to worry about overlaps
  - with repeated runs, one must still be careful!

Theory

- Introduction to simulation
- Flow of simulation -- generating process realizations
- Random number generation from given distribution
  - Introduction and generation of pseudo random numbers
  - Different methods for generating random numbers with a given distribution
  - Simulating the Poisson process
- Collection and analysis of simulation data
- Variance reduction techniques

Different sample generation methods overview

- Based on the generation of so called **(pseudo) random numbers**
  - produces independent random numbers from the uniform distribution U(0,1)

- Random numbers from any distribution can be generated, provided that random numbers from U(0,1) are available, by using some of the following methods:
  - **discretisation** (=> Bernoulli(p), Bin(n,p), Poisson(a), Geom(p))
  - **rescaling** (=> U(a,b))
  - **inverse transformation (inversion of the cdf function)** (=> Exp(λ))
  - **other transformations** (=> N(0,1) => N(μ,σ^2))
  - **rejection method** (works for any distribution)
  - **characterisation of the distribution** (reduction to other distributions) (=> Erlang(n,λ), Bin(n,p))
  - **composition method**

Generation of a discrete random variable

- Let U ~ U(0,1)
- Let X be a discrete r.v. with the range S = {0,1,2,…,n} or S = {0,1,2,...}
- Denote F(i) = P{X ≤ i}
- Then the random variable Y, where

$$\boxed{Y = \min\{i \in S \mid F(i) \geq U\}}$$

obeys the same distribution as X (Y ~ X).

- This is called the discretization method. In fact it is the inverse transformation method for a discrete distribution
- Example: Bernoulli(p) distribution:

$$\boxed{Y = 1_{\{U > 1-p\}} = \begin{cases} 0, & U \leq 1-p, \\ 1, & U > 1-p. \end{cases}}$$

Generation of a discrete random variable (continued)

- Then one needs to compute the CDF F(x) and store it in an appropriate data structure
  - Making many comparisons, however, can be computationally slow (the generations are done in the inner loops of the simulator and must be fast)

- Simple search mechanisms:
  - store values of F(x) in a vector and perform a linear search from the beginning
  - improved search: any more advanced search, e.g., a binary search
  - improved binary search: store the value of the index where F(x) equals roughly 0.5, whence one can find with one comparison on which ”side” the result lies, and then perform a binary search (or linear search)

- Optimal search (minimum number of comparison operations)
  - use a Huffman tree to represent F(x)

Using a Huffman tree to generate discrete rv:s

- Store the distribution in a Huffman tree, so that the most probable candidates have the shortest path.
- The Huffman tree is created with the following algorithm:
  - Create an ordered list of nodes. The elements in the list represent values of the rv and are ordered by probability. These nodes will be the leaves in the tree.
  - Take the first two items from the list (two smallest probabilities) and create a parent node for them so that the smaller child is on the left. The probability of the new node will be the sum of the children. The new node is added to the list in the right position.
  - Repeat the previous step until the list has only one element, which is the root of the tree.
- Now the sample generation goes as follows:
  - Generate a random number U ~ U(0,1) and compare it to the probability of the left child.
  - If the value is smaller, move to the left child. If it is greater, subtract the probability of the left child from the random number U and move to the right child. Compare the random number to the probability of the new left child.
  - Repeat the previous step until the current node is a leaf.

Example of using a Huffman tree

value	f(x)	F(x)
1	0.10	0.10
2	0.30	0.40
3	0.20	0.60
4	0.15	0.75
5	0.25	1.00

```
Iteration 1
-----------
List: (0.1, 1), (0.15, 4), (0.20, 3), (0.25, 5), (0.30, 2)
Tree:
         0.25:*
         /     \
     0.10:1   0.15:4

Iteration 2
-----------
List: (0.20, 3), (0.25,*), (0.25, 5), (0.30, 2)
Tree:
         0.45:*
         /    \
     0.20:3  0.25:*
             /     \
          0.10:1  0.15:4

Iteration 3
-----------
List: (0.25, 5), (0.30, 2), (0.45,*)
Tree:
          0.55:*
         /     \
      0.25:5  0.30:2

Iteration 4
-----------
List: (0.45,*), (0.55,*)
Tree:
              _____1.0:*______
             /                \
         0.45:*             0.55:*
         /    \             /      \
     0.20:3  0.25:*      0.25:5   0.30:2
             /     \
          0.10:1  0.15:4
```

Generation of a discrete random variable (continued)

- The described method, where the value of the random variable U ~ U(0,1) is compared consecutively with the values of the cdf is general
- In some simple cases the method based on comparisons can be replaced by a method where the value of the random number to be generated can directly be computed from the value of U using a simple formula
- Some examples are given in the following table

- In the following $U,\ U_1, \ldots,\ U_n$ denote independent random variables $\sim$U(0,1)
- $\text{int}(X) = \lfloor X \rfloor$ = integer part of X

Distribution	Expression for generation
Symmetric bivalued {0,1} distribution $\text{P}\{X=0\} = \text{P}\{X=1\} = 0.5$	$\text{int}(2U)$ or $\text{int}(U+0.5)$
Symmetric bivalued {0,1} distribution $\text{P}\{X=0\} = 1-p,\ \text{P}\{X=1\} = p$	$\text{int}(U+p)$
Bivalued {-1,1} distribution $\text{P}\{X=0\} = 1-p,\ \text{P}\{X=1\} = p$	$2\ \text{int}(U+p) - 1$
Trivalued {0,1,2} distribution probs:t $= \{1-p_1-p_2,\ p_1,\ p_2\}$	$\text{int}(U+p_2) + \text{int}(U+p_1+p_2)$
Uniform discrete distribution $\{0,\ 1,\ 2, \ldots, n-1\}$	$\text{int}(n\,U)$
Uniform discrete distribution $\{1,\ 2,\ 3, \ldots, n\}$	$\text{int}(n\,U) + 1$
Binomial distribution $\text{Bin}(n,p)$	$\sum_{i=1}^{n} \text{int}(U_i + p)$

Generation from geometric distribution

- The point probabilities of a discrete random variable X obeying the geometric distribution $\mathrm{Geom}(p)$ are

$$\mathrm{P}\{X = i\} = p_i = p(1-p)^i \qquad i = 0, 1, 2, \ldots$$

- The generation of samples of X can be done with the following simple procedure
- Algorithm

$$X = \left\lfloor \frac{\log U}{\log(1-p)} \right\rfloor$$

where $U \sim \mathrm{U}(0,1)$

- In fact, this represents generation of samples from the distribution $\mathrm{Exp}(-\log(1-p))$ and discretization to the closest integer smaller then or equal to that value

Inverse transformation method

- Let U ~ U(0,1)
- Let X be a continuous r.v. in the range S = [0,∞)
- Assume that the cdf of X, F(x) = P{X ≤ x}, is monotonously growing and has the inverse function $F^{-1}(y)$
- Then the random variable Y, where

$$\boxed{Y = F^{-1}(U)}$$

obeys the same distribution as X (Y ~ X).

- This is called the inverse transformation method (inversion of the cdf function)
- Proof: Since P{U ≤ z} = z for all z in the range [0,1], then the following holds

$$\boxed{P\{Y \le x\} = P\{F^{-1}(U) \le x\} = P\{U \le F(x)\} = F(x)}$$

Thus Y ~ X.

Generation of a random variable from the exponential distribution

- Let U ~ U(0,1)
- Let X ~ Exp(λ)
- The cdf of X, $F(x) = P\{X \le x\} = 1 - \exp(-\lambda x)$, is monotonously increasing; thus it has an inverse function $F^{-1}(y) = -(1/\lambda) \log(1 - y)$
- Since U ~ U(0,1), then also 1 - U ~ U(0,1)
- Thus the inverse transformation method gives us
- Algorithm

$$\boxed{X = F^{-1}(1-U) = -(1/\lambda)\log U}$$

Note! log-function means the logarithm with base e (i.e., the ln-function)

Generation of a random variable from the Weibull distribution

- The Weibull distribution $W(\lambda,\beta)$ is a generalisation of the exponential distribution
- The cdf of $X \sim W(\lambda,\beta)$ is $F(x) = P\{X \leq x\} = 1 - \exp(-(\lambda x)^{\beta})$
- This has the inverse function $F^{-1}(y) = (1/\lambda)[-\ln(1 - y)]^{1/\beta}$
- Algorithm

$$\boxed{X = F^{-1}(1-U) = (1/\lambda)(-\log U)^{1/\beta}}$$

Generation of a random variable from the uniform distribution

- Let $U \sim U(0,1)$
- Then $X = a + (b - a)U \sim U(a,b)$
- An arbitrary uniform distribution is thus obtained simply by scaling

Polar coordinate transformation: generation of random variable from the normal distribution

- Let U and V be independent random variables from U(0,1)
- According to so called Box-Müller method the random variables X and Y, where

$$\boxed{\begin{aligned} X &= \sqrt{-2\ln U}\cos(2\pi V) \\ Y &= \sqrt{-2\ln U}\sin(2\pi V) \end{aligned}}$$

  are independent random variables obeying the distribution N(0,1)
- Proof:
  - Consider *X* and *Y* in polar coordinates $R^2 = X^2 + Y^2$ and $\tan\Theta = Y / X$
  - The joint pdf of *X* and *Y* is $f(x,y) = (1/2\pi)\, e^{-(x^2+y^2)/2}$
  - Now we do a change of variables $d = x^2 + y^2$ and $\Theta = \operatorname{atan}(y / x)$. Thus,

    $$J(d,\Theta) = \left|\frac{\partial(x,y)}{\partial(d,\Theta)}\right| = \frac{1}{2} \Rightarrow f(d,\Theta) = f(x(d,\Theta), y(d,\Theta)) \cdot J(d,\Theta) = \frac{1}{2\pi}\cdot\frac{1}{2}e^{-d/2}$$
  - Thus, R^2 and Θ are independent with $R^2 \sim \text{Exp}(1/2)$ and $\Theta \sim U(0, 2\pi)$
  - Given (U,V) we generate $R^2 = -2 \log U$ and $\Theta = 2\pi V \Rightarrow X = R\cos\Theta$ and $Y = R\sin\Theta$
- An arbitrary normal distribution is obtained by scaling: $\mu + \sigma X \sim N(\mu,\sigma^2)$

More sample generation methods

- Continue with the material from “Generation of random variables”, part 2

Theory

- Introduction to simulation
- Flow of simulation -- generating process realizations
- Random number generation from given distribution
  - Introduction and generation of pseudo random numbers
  - Different methods for generating random numbers with a given distribution
  - Simulating the Poisson process
- Collection and analysis of simulation data
- Variance reduction techniques

Generation of arrivals from a Poisson process

- The most important model for the arrival processes used in the analysis of telecommunication systems is the **Poisson process**
- Poisson process with arrival intensity λ can be characterized as process with interarrival times which are independent and obey the exponential distribution Exp(λ)
  - Denote the arrival time of customer n by t_n
  - Arrival instants can be generated using the formula $t_{n+1} = t_n + X_n$, where $X_n \sim \text{Exp}(\lambda)$

$$\boxed{t_{n+1} = t_n - \frac{1}{\lambda} \log U}$$

- Another way to generate a realization of a Poisson process in the interval (0,T):
  - draw the total number of arrivals N from Poisson distribution $N \sim \text{Poisson}(\lambda T)$
  - draw the location of each arrival instant t_n in (0,T) from the uniform distribution $t_n \sim U(0,T)$
  - the instants $t_1, t_2, \ldots, t_n$ (in order) constitute a realisation of the Poisson process

More general arrival processes

- GI (generally distributed, independent) arrivals
  - only affects the interarrival time distribution!
- More general arrivals (non-independent)
  - must explicitly take into account the dependency structure
  - for example an MMPP (Markov Modulated Poisson Processes) where Poisson arrivals occur with different rates depending on the state of the modulating Markov chain

Generation of arrivals from a nonstationary Poisson process (1)

- Sometimes the assumption of a constant arrival rate for the Poisson arrivals is not valid (cf. time controlled public votings or competitions)
  - a Poisson process for which the arrival rate is a function of time, $\lambda(t)$, is called a nonstationary Poisson process
- Two methods: thinning or inverse transform
- Thinning method
  - Poisson process with intensity λ from which arrivals are accepted with a probability p (rejected with prob. 1-p) is still a Poisson process, but with intensity $p\lambda$
  - In simulation $\lambda(t)$ is known and it is possible to determine an upper bound $\lambda^* \geq \lambda(t), \forall t$
  - Idea: One generates arrivals with intensity λ^* and a particular arrival at time t' is accepted with probability $\lambda(t')/\lambda^*$
- Algorithm: let t_n be the arrival instant of the nth customer
  1. Set $\tau = t_n$
  2. Generate U_1, $U_2 \sim U(0,1)$
  3. Set $\tau = \tau - (1/\lambda^*) \ln U_1$
  4. If $U_2 \leq \lambda(\tau)/\lambda^*$ return $t_{n+1} = \tau$, else goto step 2
- Problem: Generation of ”extra” arrivals
  - inefficient if λ^* is large compared with the majority of the values of $\lambda(t)$ (for example, high and narrow spikes in the shape of the rate function)

Generation of arrivals from a nonstationary Poisson process (2)

- Inverse transform: comparable with the inverse transform method of the cdf
- Define the function $\Lambda(t)$

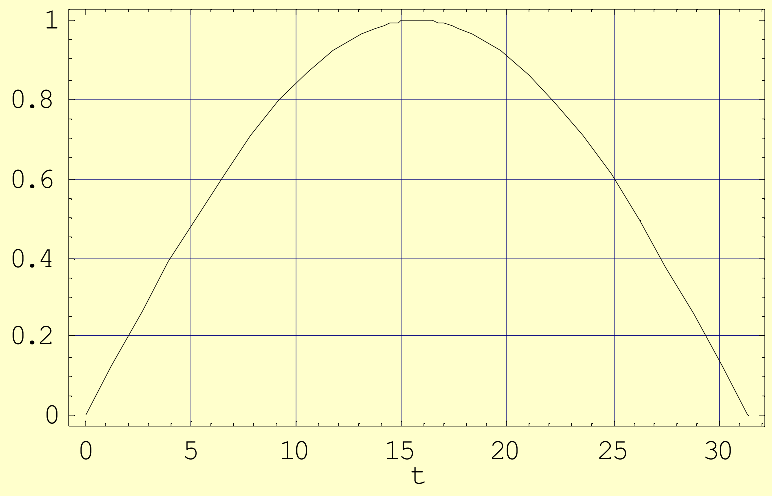
$$\boxed{\Lambda(t) = \int_0^t \lambda(y)\,dy}$$

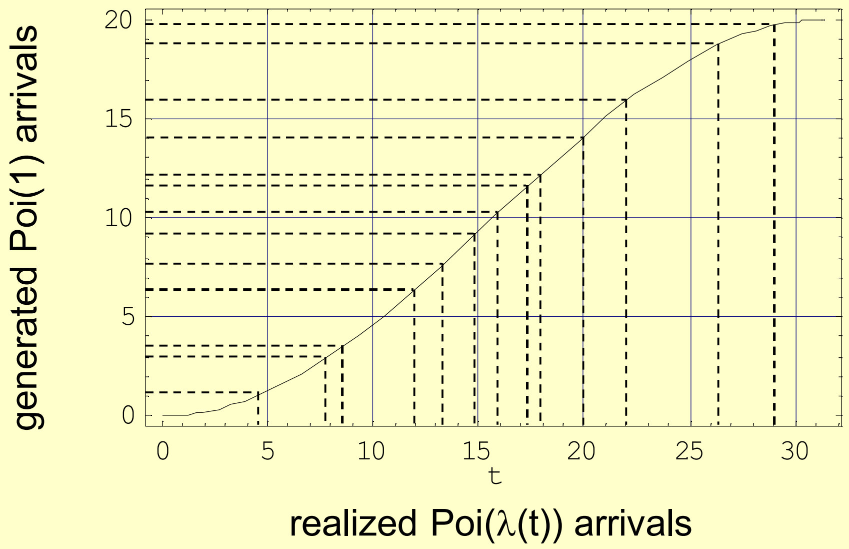
  - $\Lambda(t)$ denotes the average number of arrivals in the interval [0, t]
- Idea: Generate first arrival instants τ_n with Poisson intensity 1 and perform the transformation $t_n = \Lambda^{-1}(\tau_n)$, where $\Lambda^{-1}(y)$ is the inverse function of $\Lambda(y)$. Then, arrivals t_n constitute a Poisson process with intensity $\lambda(t)$
- Algorithm: let τ_n denote the arrival instant of the nth customer generated with intensity 1, t_n denotes the actual arrival time with intensity $\lambda(t)$
  1. Generate $U_1 \sim U(0,1)$
  2. Set $\tau_{n+1} = \tau_n - \ln U_1$
  3. Return $t_{n+1} = \Lambda^{-1}(\tau_{n+1})$
- Benefit: all arrival events can be utilized
  - Problem: solving the inverse function may be difficult/impossible

Generation of arrivals from a nonstationary Poisson process (3)

- Example of the inverse transform method

$$\lambda(t) = \sin(t/10) \quad , \quad t \in [0, 10\pi] \quad \Leftrightarrow \quad \Lambda(t) = 10(1 - \cos(t/10))$$


1
0.8
0.6
0.4
0.2
0
0
5
10
15
20
25
30
t


generated Poi(1) arrivals
t
realized Poi(λ(t)) arrivals