Discrete Stochastic Simulation

Pedro Barahona
DI/FCT/UNL

Métodos Computacionais

1st Semestre 2018/2019

Random Processes

- Many "systems" do not have an analytical model from which we may study their behaviour over time, as well as making decisions about their design. Nevertheless, for many such systems, their behaviour may be analysed by simulation.
- An important source of uncertainty is the occurrence of non-deterministic events, affecting such behaviour, but for which there is no exact information about them.
- In this case, studying these systems requires the consideration of **stochastic processes**, i.e. phenomena that evolve over time or space taking into account a sequence of events. The timing of these events can be approximated given the incomplete information that may be known, such as the patterns observed in the past of their occurrence.
- These patterns are typically modelled by probability distributions that fit the observations, as studied in Statistics.
- Here we will thus consider nondeterministic processes where events follow some probability distribution, discrete or continuous, and study how to model systems subject to this type of events.

(Pseudo-) Random Numbers

- As will be seen briefly, any nondeterministic process that follows a known probability distribution may be simulated by means of a **random generator function**, that generates numbers in the interval **0..1** with a **uniform distribution**.
- In most computer languages and tools (as in MATLAB) this random generator is available through a system defined function **rand()**.
- Based on this function any nondeterministic process, defined by a known **probability density function** (**PDF**), **p**, can be simulated.
- Informally, this function is defined over a domain, discrete or continuous, of the values that a probabilistic variable can take. We will assume here a numerical domain ranging in the interval **a..b**.
- Remind that the **cumulative distribution function** (**CDF**), **P**, can be defined as

Discrete Domain

$$P(x) = \sum_{v=a}^{v=x} p(v)$$

Continuous Domains

$$P(x) = \int_{v=a}^{v=b} p(v)\,dv$$

Inverse Method

- The inverse method takes into account that, for a random variable taking values in the domain **a .. b**, it is

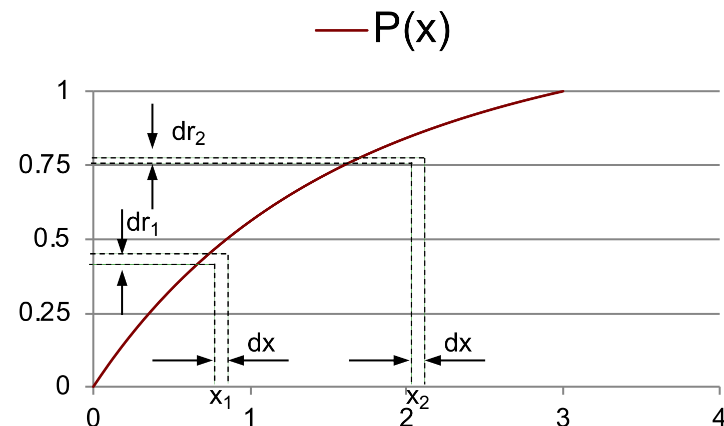
$$P(a) = 0 \quad \text{and} \quad P(b) = 1$$

- Then, the random variable may be implemented by the inverse method in the two following steps:

  1. Generate a random number **r**, with uniform distribution in the interval **0 .. 1**;
  2. Return $\mathbf{x = F^{-1}(r)}$

- In fact the probability $\mathbf{p_i}$ of generating a number in interval $\mathbf{x_i}$ **..** $\mathbf{x_{i+dx}}$, i.e. the probability that the variable takes an approximate value $\mathbf{x_i}$ is, **dx**. Since,

  - $\mathbf{p_1} = \mathbf{dr_1} = dP(x_1)/dx \bullet dx = \mathbf{p(x_1)\ dx}$;
  - $\mathbf{p_2} = \mathbf{dr_2} = dP(x_2)/dx \bullet dx = \mathbf{p(x_2)\ dx}$;

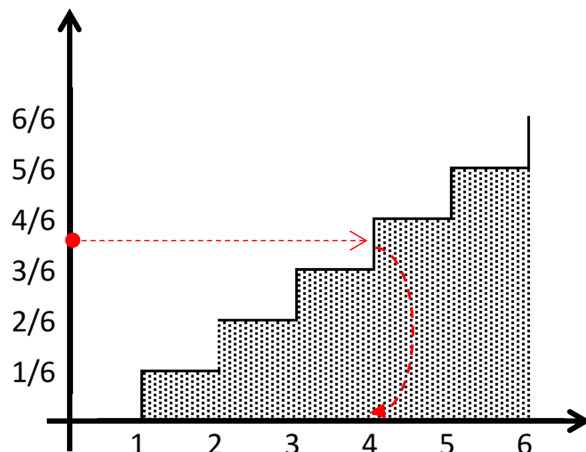


- Hence the probabilities of two values in the domain being generated is proportional to the value of their probability density function.

Inverse Method

Example: Simulate the throwing of a dice

- In this discrete distribution, each of the values 1 to 6 occurs with probability 1/6.
- The probability distribution **P(x)**, is the step function shown in the figure;
- The inverse function, $\mathbf{P^{-1}(x)}$, can be computed by finding the step (1..6) of the probability function that corresponds to the random number r, generated by function **rand()**, as implemented in function **dice()**.

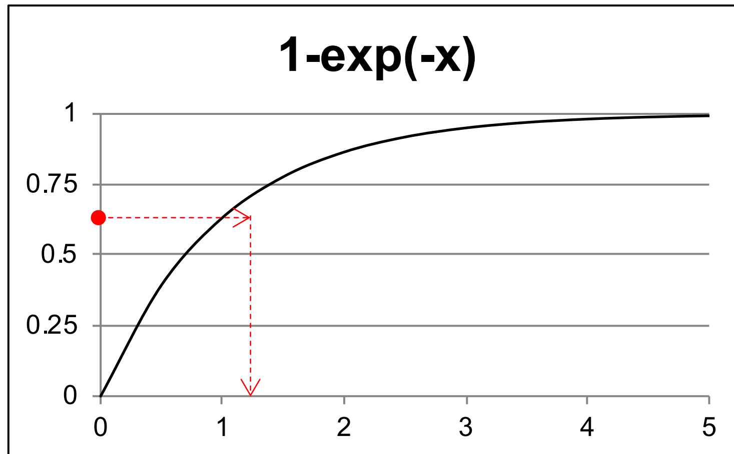



```
function v = dice();
   r = rand();
   if      r <= 1/6 v = 1;
   elseif r <= 2/6 v = 2;
   elseif r <= 3/6 v = 3;
   elseif r <= 4/6 v = 4;
   elseif r <= 5/6 v = 5;
   else             v = 6;
   end
end
```

Inverse Method

Example: Simulate the next arrival of a stochastic process following an exponential distribution, with mean time $\mathbf{m = 1/\lambda}$

- This is a continuous distribution where $\mathbf{p(x) = \lambda e^{-\lambda x}}$, ranging from 0 to ∞.
- The probability function $\mathbf{r = F(x) = (1- e^{-\lambda x})}$ (shown for $\lambda = 1$)
- The inverse function is then $\mathbf{x = F^{-1}(r) = - \ln(1-r) / \lambda}$
- Hence, these arrivals can be modelled by a variable obtained through function **exp_inv(lambda)**, shown below parameterised by the value of λ.




```
function x = exp_inv(lambda);
   r = rand();
   x = -log(1-r)/lambda;
end
```

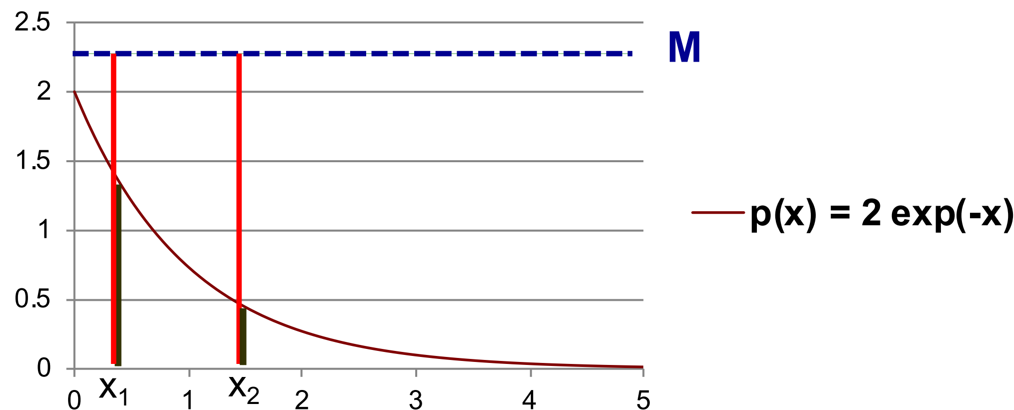
Accept/Reject Method

- Of course, the inverse method assumes that it is possible to obtain a $\mathbf{F^{-1}}$, the inverse of the cumulative distribution function **F**.
- When a closed form of $\mathbf{F^{-1}}$ is not available, the random variable may be implemented by the **accept/reject method**. Assuming
  - The domain of the variable is **a .. b**, and
  - The probability density function in the domain is always less or equal to **m**

  then the random variable may be implemented in the following steps:
  1. Generate a random number **x**, with uniform distribution in the interval **a .. b**;
  2. Generate a random number **r**, with uniform distribution in the interval **0 .. m**;
  3. Accept **x**, if $\mathbf{r \leq p(x)}$, reject it otherwise
- In some cases, the domain of a continuous random variable is infinite. In this case, one may truncate the domain so that the values truncated have a “very low probability”

Accept/Reject Method

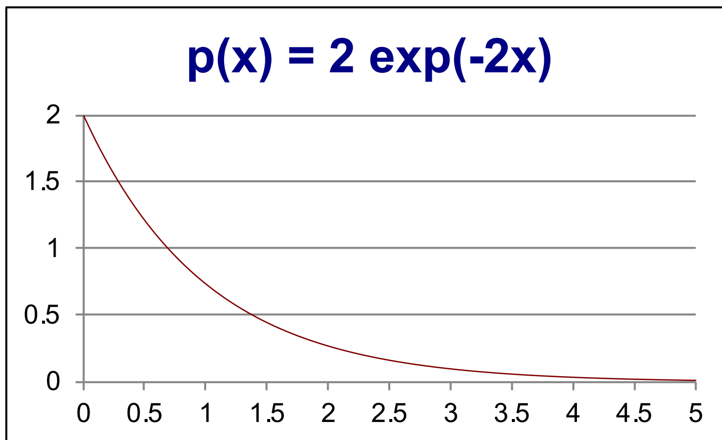
- The probability that a value x_i in the domain **a..b** is accepted is thus
  - Probability that $\mathbf{x_i}$ is generated, i.e. the value is between $\mathbf{x_i}$ and $\mathbf{x_i + dx}$;
  - Probability that the value is subsequently accepted, i.e. $\mathbf{p(x_i) \leq r}$.
- Given two values $\mathbf{x_1}$ and $\mathbf{x_2}$, the first probability is the same for both (**dx** is the same) .
- Since **r** is generated in the range **0..M**, their acceptance probability is, respectively, $\mathbf{p(x_1)/M}$ and $\mathbf{p(x_2)/M}$.
- Hence the probability of generating two values $\mathbf{x_1}$ and $\mathbf{x_2}$ is proportional to the value of their probability density function



Accept/Reject Method

Example: Simulate the next arrival of a stochastic process following an exponential distribution

- This is a continuous distribution where $\mathbf{p(x) = \lambda\, e^{-\lambda x}}$, ranging from 0 to ∞.
- The domain must then be truncated to some value **T** (**T=5** in the figure).
- The function is always less or equal to λ (so we can use **M = λ)** .
- Hence, these arrivals can be modelled by a variable obtained through function **exp_ar(lb,t)**, shown below parameterised by the values of λ and **k**.




```
function x = exp_ar(M,T);
   accept = false;
   while ! accept
      x = T * rand();
      r = M * rand();
      accept = (r <= M*exp(-M*x))
   end
end
```

Simulation of Stochastic Systems

- A stochastic system has a behaviour that depends on a stochastic process, i.e. a sequence of non-deterministic events that evolve over time or space.
- Here we assume that the nondeterministic events may be modelled by random variables specified by some probability distribution.
- At any time, the system is characterised by its **state**, i.e. the value of the set of **state variables** that completely specifies it.
- Whenever an **event** occurs, it causes some (possibly empty) change of the system to a new state, possibly yielding some **output**.
- Such a system can thus be modelled by an **automaton**, defined informally as
  - A set of **states**, some of which might be the initial states
  - A set of **transitions**, between two states, caused by some **event**, and possibly yielding some output.
  - **Monitoring data**, that gathers extra information during the simulation useful to analyse the behaviour of the system.
- The behaviour of the system is modelled by simulating the state transitions of the automaton given a set of events, until a stop condition holds.

Example: Random Walk

- Once defined the state variables and the events that may cause state transitions, an automaton may thus be specified by a table that enumerates all possible state transitions.
- In addition the initial state must be specified, as well as the terminating conditions.
- One such automaton may be considered to simulate a **random walk**, i.e. the movement of an object composed of a sequence of random steps (this is a very simplified model of the Brownian Motion problem arising in physics - cf. https://en.wikipedia.org/wiki/Brownian_motion)
- In particular we will consider the steps to be either forward or backwards, occurring with equal probability, and causing the position of the object to increase or decrease, respectively, its current position.
- The automaton for the Random Walk process may thus be specified in the following slide.

Example: Random Walk

State Variables:

- A state `st` consists of two fields, **pos**, stating the position of the object, and **stp**, the number of steps already done

Events

- An event is either a move forward or backwards, and is represented by a variable, **dir**, with randomly generated values values 1 or -1.

Initial State

- We may assume that the object starts in **pos = 0** at **stp = 0**.

Termination Condition

- We may want to know whether the object reaches a certain distance **d** fom the initial position and thus set the termination condition to be pos >= d (or better, if we consider both directions, **abs(pos) >= d**.

Monitoring Data

- In this case, we do not need extra information. All useful information is required is modelled in the state variables.

Example: Random Walk

- The state transition table of this automaton can thus be specified as follows:

event	current state		next state		output
dir	stp	pos	stp	pos	next event
1	s	p	s+1	p+1	random dir
-1	s	p	s+1	p-1	random dir

- Notice that, somewhat artificially we consider as output the generation of the next event. This is done so that it allows the specification of a generic algorithm to specify these automata, where some side information may be considered in general.
- We will also consider as output the update of some **monitoring** information that is needed in more complex automata and simulations.

Simulation of Stochastic Systems

- The simulation of a system, i.e. the behaviour of the corresponding automaton, may be specified through the following generic function

```
function m = simulate (...)
   s = initial_state(...);
   e = initial_event(...);
   m = initial_monitor(...);
   while !stop(...)
      [s, e, m] = transition(s, e, m);
   end
end
```

- The function returns the information gathered during the simulation for monitoring the behaviour of the system.
- To make it generic, all the information gathered in the states, events and monitoring, should be modelled in appropriate structures, **s**, **e** and **m**, respectively.
- The function requires the specification of 5 auxiliary functions:

Simulation of Stochastic Systems

```
function m = simulate (...)
   s = initial_state(...);
   e = initial_event(...);
   m = initial_monitor(...);
   while !stop(...)
      [s, e, m] = transition(s, e, m);
   end
end
```

- **initial_state(…)**: sets the state variables in the initial state.
- **initial_state(…)**: sets the event variables applicable to the initial state.
- **initial_state(…)**: sets the monitoring variables prior to the initial state.
- **stop(…)**: checks the stopping condition for the simulation.
- **transition(s,e,m):** given the state **s**, and an event obtained from **e**, determines the next state, and updates the monitoring data, **m**, as well as the event data from where the next event can be obtained.

Example: Random Walk

- We may now instantiate the generic functions identified before.
- The initial state has both steps and position set at zero

```
function s = initial_walk_state()
    s.pos = 0;
    s.stp = 0;
end
```

- The initial event is simply either -1 or +1 randomly generated (with equal probability).

```
function e = initial_walk_event()
    e.dir = -1 + (random()< 0.5) * 2;
end
```

- In this case, we will not consider extra monitoring information. All the useful information lies in the state variables

```
function m = initial_walk_monitor(s)
    m = s;
end
```

Example: Random Walk

- The stopping condition just checks if distance k was reached, within k steps

```
function b = stop_walk (s, d, k);
   b = (abs(s.pos) >= d || s.stp > k);
end
```

- We may now specify the transition function, given the previously discussed transition.

event	current state		next state		output
dir	stp	pos	stp	pos	next event
1	s	p	s+1	p+1	random dir
-1	s	p	s+1	p-1	random dir

```
function [s, e, m] = transition_walk(s, e, m);
   s.stp = s.stp + 0;
   s.pos = st.pos + e.dir;
   e.dir = -1 + (random()< 0.5) * 2;
   m = s;
end
```

Example: Random Walk

- To sum up, the simulation function is instantiated for the case of a random walk system, i.e. the behaviour of the corresponding automaton, may be specified through the following generic function

```
function m = simulate_walk (d, k)
   s = initial_walk_state();
   e = initial_walk_event();
   m = initial_walk_monitor(s);
   while ! stop_walk (s, d, k);
       [s, e, m] = transition_walk(s, e, m);
   end
end
```

Example: Queueing Systems

- Queuing systems are systems relying on the occurrence of requests that are to be serviced, if possible, by a number of existing resources.
- Examples of these systems are everywhere, ranging from traditional supermarket tills or petrol stations, to more "present day" call centres or computers servers.
- Broadly, these systems are characterised by a number of parameters, namely:
  - the number of servers that are available (in parallel);
  - the timing of the request arrivals
    - typically following some probability distribution
  - the queuing discipline used
    - a simple queue or different queues
  - the maximum size of a queue
    - if full, a new arriving request is rejected
  - the service time for each request
    - typically following some probability distribution

Example: Queueing Systems

- These systems are now illustrated with a very simple system with the following characteristics:
  - One single server
  - Service time following a uniform distribution between 2 and 8 secs;
  - No queue buffer: if a request arrives when the server is busy, it is rejected.
  - Requests arrive with an exponential distribution with mean time of 5 seconds.
- Note that although the mean time between arrivals and the mean time of service is the same (5 secs) a number of features are not be easily computed analytically.
- In particular, simulation (for a sufficient large time) may be used to estimate the value of a number of features of this system, namely
  - What is the percentage of time the server is busy
  - What is the percentage of requests that are rejected

Example: Queueing Systems

- The simulation of these systems can use the previous scheme, taking into account the characteristics of the specific queuing system.
- In particular, the state, **s**, of a system should indicate whether a request is being served, and at what time it arrived. Moreover it should encode the simulation time. Hence, **s** may be encoded as a structure with two fields:
  - **s.time (time)**: the time elapsed since the beginning of the simulation;
  - **s.entry_server_time (est):** a number specifying whether the server is busy. If the server is busy it should represent the time the request has been accepted. Otherwise, the value is encoded as +inf.
- The event, **s**, should indicate the timing of the next arrival of a request, as well as the timing of the next completion of a served request. :
  - **e.next_arrival_time (nat)**: the timing of the next arrival of a request;
  - **e.next_exit_time (net)**: the timing of the next exit from the server.
  - If the servers are empty the **next_exit_time** should be encoded as **+inf.**

Example: Queueing Systems

- The system should be monitored so as to maintain at any time the number of requests so far (total, accepted and rejected). Hence, **m** may be encoded as a structure with 3 fields:
    - **m.server_busy_time (sbt)**: the time the server has been busy so far;
    - **m.number_accepted_requests (nar)**: Number of requests accepted so far;
    - **m.number_rejected_requests (nrr)**: Number of requests rejected so far;
- Given these assumptions the initial state should be encoded as
    - **s.time = 0; s.net = inf.**
- The initial events should be
    - **e.nat = x; s.net = inf.** where x is obtained from the exponential distribution
- The initial monitoring data should be
    - **m.nar = 0; m.nrr = 0; m.sbt = 0**

Example: Queueing Systems

- The stopping condition could be specified in a number of ways. One possibility is to simulate the queueing system until some final time, i.e. until
  - **s.time > final_time**

- Finally, the state transitions can be caused by the arrival of requests or exit from servers, and can be described in the following transition table

	event		current state		next state		next event		monitor		
	nat	net	lts	est	lts	est	nat	net	nar	nrr	sbt
arrival while empty	a	inf	t	inf	a	a	exp a	uni a	+1	=	=
arrival before exit	a	b (> a)	t	c	a	c	exp a	b	=	+1	=
exit before arrival	a	b (< a)	t	c	b	inf	a	inf	=	=	+(b-c)

where **next a** and **next b** are random times generated, respectively, according to the exponential distribution of mean time **m**, and a uniform distribution between **lo** and **up**.

Example: Queueing Systems

- Given the above specifications we can implement the queueing system, with 1 server and max queue of 0 as follows:

```
function s = initial_s1q0_state()
   s.latest_system_time = 0;
   s.entry_time_in_server = inf;
end
```

```
function e = initial_s1q0_event(mean)
   e.next_arrival_time = expo_distr (mean);
   e.next_exit_time = inf;
end
```

```
function e = initial_s1q0_monitor()
   m.number_rejected_services = 0;
   m.number_accepted_services = 0;
   m.server_busy_time = 0;
end
```

```
function e = stop_s1q0(s,max_t)
    s.latest_system_time =< max_t;
end
```

Example: Queueing Systems

- Finally, the state transitions can be implemented taking into account the previous transition table

	event		current state		next state		next event		monitor		
	nat	net	lts	est	lts	est	nat	net	nar	nrr	sbt
arrival while empty	a	inf	t	inf	a	a	exp a	uni a	+1	=	=
arrival before exit	a	b (> a)	t	c	a	c	exp a	b	=	+1	=
exit before arrival	a	b (< a)	t	c	b	inf	a	inf	=	=	+(b-c)

```
function [s,e,m] = transition_s1q0(s,e,m,lo,up,mean);

% arrival while empty
   if e.next_exit_time == inf
      s.latest_system_time = e.next_arrival_time;
      s.entry_time_in_server = e.next_arrival_time;
      e.next_arrival_time = s.latest_system_time + expo_distr(mean);
      e.next_exit_time = s.latest_system_time + unif_distr(lo,up);
      m.number_accepted_services = m.number_accepted_services + 1;
  ....

end
```

Example: Queueing Systems

	event		current state		next state		next event		monitor		
	nat	net	lts	est	lts	est	nat	net	nar	nrr	sbt
arrival while empty	a	inf	t	inf	a	a	exp a	uni a	+1	=	=
arrival before exit	a	b (> a)	t	c	a	c	exp a	b	=	+1	=
exit before arrival	a	b (< a)	t	c	b	inf	a	inf	=	=	+(b-c)

```
function [s,e,m] = transition_s1q0(s,e,m,lo,up,mean);

....
% arrival before exit
   elseif e.next_arrival_time <= e.next_exit_time
      s.latest_system_time = e.next_arrival_time ;
      e.next_arrival_time = s.latest_system_time + expo_distr(mean);
      m.number_rejected_services = m.number_rejected_services + 1;
....

end
```

Example: Queueing Systems

	event		current state		next state		next event		monitor		
	nat	net	lts	est	lts	est	nat	net	nar	nrr	sbt
arrival while empty	a	inf	t	inf	a	a	exp a	uni a	+1	=	=
arrival before exit	a	b (> a)	t	c	a	c	exp a	b	=	+1	=
exit before arrival	a	b (< a)	t	c	b	inf	a	inf	=	=	+(b-c)

```
function [s,e,m] = transition_s1q0(s,e,m,lo,up,mean);
...
%exit before arrival
   elseif e.next_exit_time < e.next_arrival_time
      s.latest_system_time = e.next_exit_time;
      aux = e.next_exit_time - s.entry_time_in_server;
      m.server_busy_time = m.server_busy_time + aux;
      e.next_exit_time = inf;
      s.entry_time_in_server = inf;
   else
      printf("unforeseen situation!!!");
end
```