Random Variables; (Monte Carlo) Simulation

Pedro Barahona
DI/FCT/UNL

Métodos Computacionais

1st Semester 2016/2017

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

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

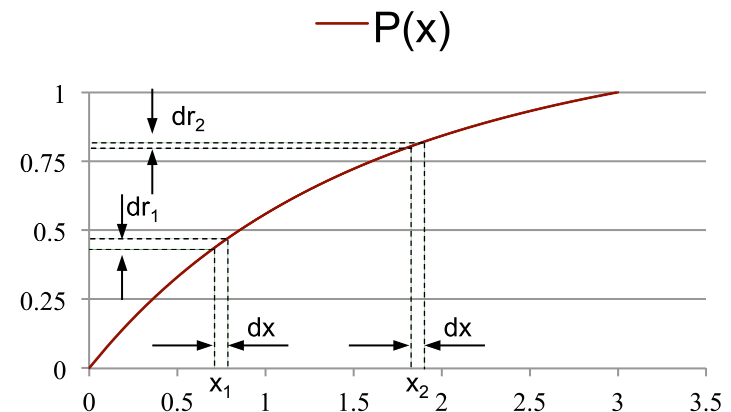
Inverse Method

- 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 p_i of generating a number in interval x_i .. x_i+dx, i.e. the probability that the variable takes an approximate value x_i is, dx. Since,

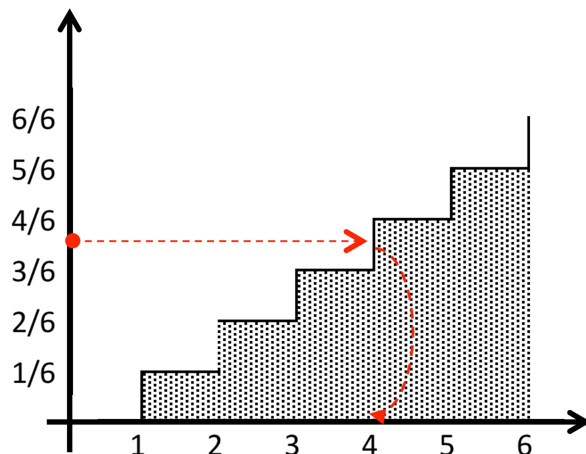
- $\mathbf{p_1} = \mathbf{dr_1} = d\,P(x_1)/dx * dx = \mathbf{p(x_1)\ dx}$;
- $\mathbf{p_2} = \mathbf{dr_2} = d\,P(x_2)/dx * dx = \mathbf{p(x_2)\ dx}$;
- Hence the probabilities of two values in the domain being generated is proportional to the value of their probability density function.



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, **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()**.

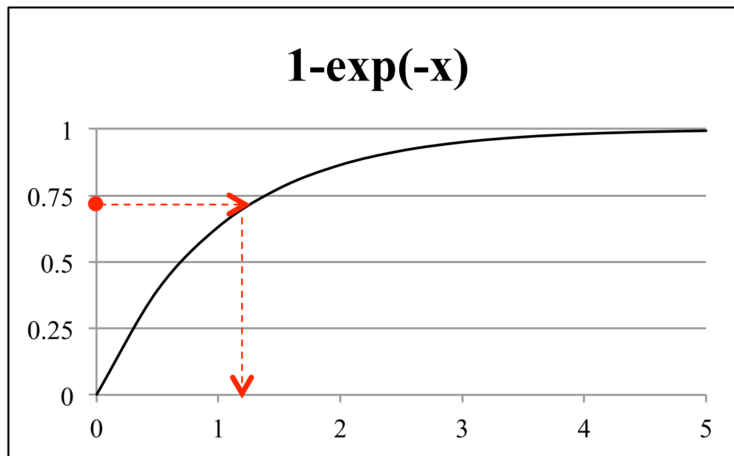



```
function v = dice();
   r = rand();
   if      r <= 1/6 v = 1;
   elseif r <= 2/6 v = 2;
   elseif r <= 3/6 v = 3;
   elseif r <= 4/6 v = 4;
   elseif r <= 5/6 v = 5;
   else             v = 6;
   endif
endfunction
```

Inverse Method

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

- This is a continuous distribution where $p(x) = \lambda\, e^{-\lambda x}$, ranging from 0 to ∞.
- The probability function $r = F(x) = (1 - e^{-\lambda x})$ (shown for $\lambda = 1$)
- The inverse function is then $x = F^{-1}(r) = -\ln(1-r) / \lambda$
- 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 = -ln(1-r)/lambda
endfunction
```

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**.

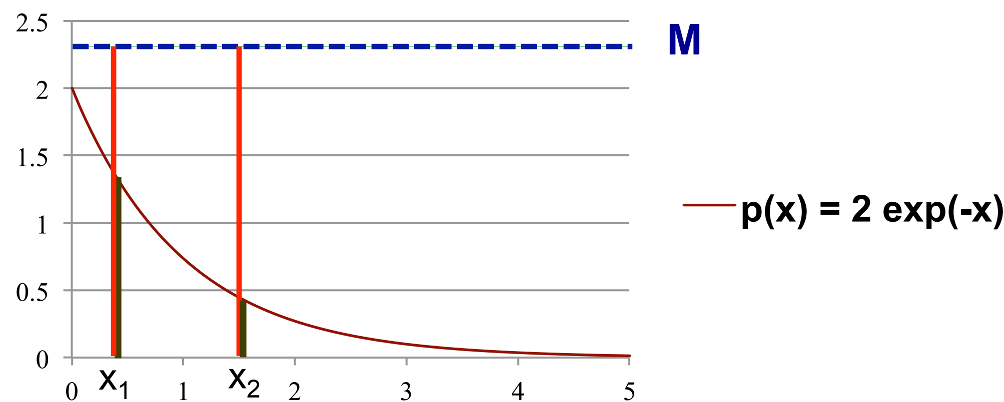
- When a closed form of $\mathbf{F^{-1}}$ is not available, the random variable may be implemented by the **accept/reject method**. Assuming
  - The domain of the variable is **a .. b**, and
  - The probability density function in the domain is always less or equal to **m**

- 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

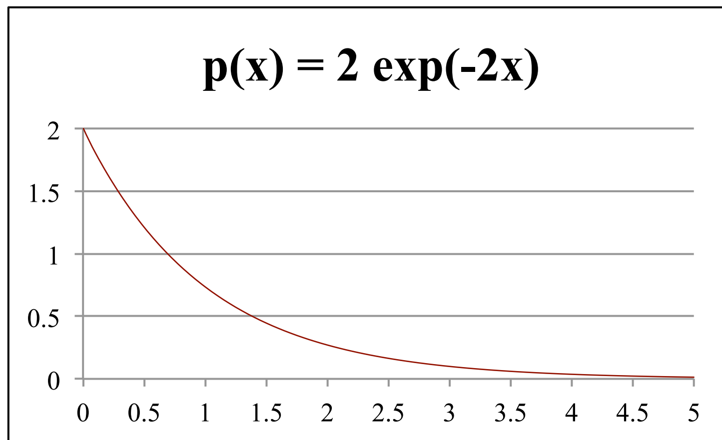
- The probability that a value x_i in the domain a..b is accepted is thus
  - Probability that x_i is generated, i.e. the value is between $\mathbf{x_i}$ and $\mathbf{x_i + dx}$;
  - Probability that the value is subsequently accepted, i.e. $\mathbf{p(x_i) \leq r}$.
- Given two values $\mathbf{x_1}$ 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(lb,T);
   accept = 0;
   while ! accept
      x = T*rand();
      r = lb*rand();
      accept = (r <= lb*exp(-lb*x))
   endwhile
endfunction
```

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 specify it.
- Whenever an **event** occurs, it causes some (possibly empty) change of the sistem to a new state.
- 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.
- The behaviour of the system is modelled by subjecting the automaton to a set of “external” events, i.e. those that are independent of the system.

Note: In general an automaton produces an output. Here we assume that the output of interest, may be obtained from the state variables,

Simulation of Stochastic Systems

- The simulation of a system, i.e. the behaviour of the corresponding automaton, may be specified through the following generic algorithm, where
  - $R = <r_1, r_2, \ldots, r_m>$ is a sequence of "requests" (external to the system)
  - $S = <s_0, s_1, \ldots, s_n>$ is a sequence of states

```
function S = simulate(s,R);
   e = "dummy"; S = [];
   while e != "end"
      e = next_event(s,R);
      [s,R] = update(s,R,e);
      S = [S,s];
   endwhile
endfunction
```

- Function **`next_event`** simply detects the event occurring next, but does not affect neither the sequence of requests nor the state of the automaton.
- Function **`update`** uses this event to change the state of the automaton, and possibly remove requests from the sequence.
- The new state is added to the sequence of states, and the loop proceeds until an ***end*** event is detected.

Example: Random Walk

- This generic algorithm must be instantiated by specifying the adequate **next_event** and **update** functions. We illustrate this with a simple example: **a random walk**, i.e. the movement of an object composed of a sequence of **random steps**.
- 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 move, respectively, +1 or -1 positions from its current position.
- **State Variables**:
  - The state **s** consists a single integer variable, stating the position of the object.
  - We may assume that the initial state corresponds to **s = 0**.
- **Events**
  - There are two types of “real” events, **move_forward** and **move_backwards**. In addition we create a virtual event “**end**” to stop the simulation.
- **Transitions**
  - Whenever a **move_forward** event occurs, **s** increases by 1.
  - Whenever a **move_backwards** event occurs, **s** decreases by 1.
  - When the virtual **end** event occurs, the similation finishes.

Example: Random Walk

- For convenience, we may encode both the requests to steps forward and backwards and the corresponding events by integers +1 and -1, respectively, and the **end** event by integer **0**.
- Moreover, the sequence of n events is encoded as an n element array, whose elements are the numbers +1 and -1, generated with equal probability.
- This can be done with function **`generate_random_walk_events`**

```
function R = generate_random_walk_events(n)
   R = zeros(1,n+1);
   for i = 1:n
      R(i) = 2*(round(rand())-0.5)
                  % 0/1 with probability = 0.5
   endfor
   R(n+1) = 0;
endfunction
```

Example: Random Walk

- Function next_event detects the type of the next event, which in this case is simply the value of the first element of the sequence of events, as shown

```
function e_type = next_event(s,R)
   e = E(1)
endfunction;
```

- In this simple example, function **update** simply removes the event from the sequence, and applies it to the state variable **s**.

```
function [s,E] = update(s,R,e);
   R = R(2:end)
   s = s+e
endfunction
```

- These functions can be used to obtain a simulation of a random walk process, as shown below

```
function S = simulate_random_walk(n);
   R = generate_random_walk_events(n);
   S = simulate(s,E);
endfunction
```

Monte Carlo Simulation

- A simple simulation does not provide might not provide sufficient information for the characterization of the behaviour of a system. Many of these characteristics do not have a deterministic value but only a probabilistic one.
- In this case, many different and independent simulations may be performed so that such statistic values (e.g. averages and variances) may be collected from the results obtained in the different simulations.
- This is the basic idea behind **Monte Carlo Simulation**:
  - Run a sufficient number of simulations
  - Aggregate the values of interest from the results of the different simulations
- In general these should be sufficient to guarantee the significance of the results, but this topic is beyond the scope of this course.
- Of course if we are interested in some particular feature we may adapt the simulation function, so as to simply provide the results we are interested in.

Monte Carlo Simulation: Random Walk

- For a **random walk**, we may be interested in obtaining the **probability** that distance **d** from the initial position is reached in **n** (or less) steps.
- This probability can be approximated by the frequency in which this occcurs, given a sufficient large number of runs of the random_walk process.
- We may thus perform **ns** simulations and count the number **ps** of those that reached the distance **d** in **n** steps (or less), as shown below.

```
function p = random_walk_reach_probability(n,d)
   ns = 1000; ps = 0;
   for i = 1:ns
      S = simulate_random_walk(n,d);
      if check(S,d) ps = ps+1; endif
   endfor
   p = ps/ns;
endfunction
```

- Here we assumed a function check that analyses the sequence S of states generated in a simulation and detects whether a distance ± d was reached. Of course, it would be more efficient to stop a simulation as soon as distance ±d is reached and return a Boolean indicating whether this was the case.
- ***This is left as an exercise***.

Example: Queuing 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 the number of servers that are available (in parallel), the queuing discipline used (i.e. a simple queue or different queues, the maximum size of a queue – if full, a new request is rejected) and the the service provided. In particular, the service time is usually characterised by some probability distribution.
- The behaviour of these queuing systems depends of course on the arrivals of requests that can also be characterised by some probability distribution. The simulation of these systems can use the previous scheme, taking into account that
  - Requests are external events, that queue to be serviced
  - Whenever a server is free, a request from a queue is moved to a server;
  - Whenever a service is completed, a terminating event occurs, denoting that the corresponding server became free.

Example: Queuing Systems

- To simulate such queuing systems, we must model the state of the system and the events it is subject to. There are several possible variants of these systems so we will consider the following

Example:

- A system with a single server and accepting one request on the waiting. If more requests arrive, they are rejected. We also assume
  - Arrivals events follow an exponential distribution with mean time **m** between arrivals **m** (for example, **m = 5 secs**)
  - Services are processed in some constant time, **p** (for example, **p = 4 secs**)
- For this problem, we are interested in studying:
  a. What percentage of time the server is busy
  b. The percentage of requests that are rejected.
- To study this system, we must of course specify
  a. The type of events to be considered
  b. A model of the system, namely the variables that represent a current state of the system, as well as well as additional information about the past behaviour used to answer the questions posed.

Example: Queuing Systems

State Variables:

- We model the state of the system by a structure **s**, with the following fields
  - **s.busy** – a Boolean denoting whether the server is busy
  - **s.finish_t** – the time the server will finish the service (when busy is 1)
  - **s.waiting** – a Boolean denoting whether there is a request already waiting
  - **s.arrival_t** – the time the waiting request has arrived (when waiting is 1)
  - **s.n_accepted** – the number of requests accepted so far
  - **s.n_rejected** – the number of requests rejected so far
  - **s.working_t** – the time the server has been busy so far

Note: The 3 last fields of the last state answer the questions we are interested in.

Events:

- In this case, events will be of three types, in addition to a virtual **end** event:
  - **arrival** – a request is transferred to the queue of the system
  - **start** – a request in the queue is transferred to the server
  - **term** – a service terminates being served

Example: Queuing Systems

- We simulate the system by means of a function **simulate_simple_queue**, that will use the previous functions adapted for the mentioned encodings of the current state and the request sequence that

```
function [p_rjct,p_busy] = simulate_simple_queue(n,m);
   R = generate_exp_arrivals(n,m);
   s = initial_state();
   S = simulate(s,R);
   acpt = S(end).n_accepted;
   rjct = S(end).n_rejected;
   p_rjct = rjct/(acpt+rjct);
   p_busy = S(end).working_t / S(end).finish_t;
endfunction
```

- The simulation itself is similar to that explained before, repeated below

```
function S = simulate(s,R);
   e = "dummy";
   while e != "end"
      e = next_event(s,R);
      [s,R] = update(s,R,e);
      S = [S,s];
   endwhile
endfunction
```

Example: Queuing Systems

Generation of the Sequence of Requests:

- We will study this system for an interval of time that should be large enough to gather significant statistical information.
- For example we may assume that 1000 requests are a sufficient large number, and so create a sequence of 1000 requests, each stored with the time in which it is made, with the following program

```
function R = generate_exp_arrivals(n,m);
% n should be a number large enough
% m is the mean time
   R(1) = 0;
   for i = 2:n_repeat
      delta = exp_inv(1/m); % exponential distribution
      R(i) = R(i-1) + delta;
   endfor;
endfunction
```

Example: Queuing Systems

- For this problem, the selection of the **next_event** can be done as follows:

```
function e = next_event(s,R)
   if s.busy
      if length(R) > 0 && s.finish_t > R(1)   e = 1; %"arrival"
      else                                    e = 3; % "term"
      endif
   else
      if ! s.waiting % && ! s.busy
         if length(R) == 0           e = 4; % "end"
         else                        e = 1; % "arrival"
         end
      else   % s.waiting && ! s.busy
                                     e = 2; % "start
      endif
   endif
endfunction
```

Example: Queuing Systems

Generation of the Sequence of Requests:

- We may now define the function **update** that changes the state of the system, and possibly, the state of the request sequence.
- The function may be implemented by means of different functions, one for each of the three type of events (**arrival**, **start** and **term**) that are detected

```
function [s,R] = update(s,R,e);
   if e == 1              % "arrival"
      [s,R] = arrival(s,R);
   elseif e == 2          % "start"
      [s,R] = start_service(s,R);
   else                   % "term"
      [s,R] = term_service(s,R);
   endif
endfunction
```

Example: Queuing Systems

Event Processing

Arrivals:

- If the event is an **arrival**, a request is **transferred** from the request sequence to the queue maintained by the system.
- Upon an arrival event, either the request is accepted or rejected (if the queue is full). Otherwise the time of arrival is recorded and the number of accepted / rejected messages is updated.

```
function [s,R] = arrival(s,R);
   if s.waiting == 1
      s.n_rejected = s.n_rejected + 1;
   else
      s.waiting = 1;
      s.arrival_t = R(1);
      s.n_accepted = s.n_accepted + 1;
   end;
   R = [R(2:end)];
endfunction
```

Example: Queuing Systems

Event Processing

Starts:

- If the event is an **start**, this means the server is not **busy**, and it must become so.
- Moreover, the queue becomes empty.
- The start of the service is either the time when the last service has terminated, or the time the request arrived into the queue (both recorded in the current state).
- The expected service time should be added to the start time to become the current finish time (in this case we use a constant time = 4), which updates the finishing time and the overall working time of the server

```
function [s,E] = start_service(s,E);
   s.busy = 1;
   s.waiting = 0;
   start_time = max(s.finish_t, s.arrival_t);
   service_time = 4;
   s.finish_t = start_time + service_time;
   s.working_t = s.working_t + service_time;
endfunction
```

Example: Queuing Systems

Event Processing

Terms:

- If the event is a **term**, which means that the **busy** state variable has been true, then all that is required is to change the value of this busy variable.

```
function [s,R] = term_service(s,R);
   s.busy = 0;
endfunction
```

Example: Queuing Systems

Simulation

- To start a simulations, it is still necessary to define the initial state

```
function s = initial_state();
   s.busy = 0;
   s.finish_t = 0;
   s.waiting = 0;
   s.arrival_t = 0;
   s.n_accepted = 0;
   s.n_rejected = 0;
   s.working_t = 0;
endfunction
```

- With all the previous function in place, all that is needed to know the percentage of requests that are rejected, as well as the percentage of time the server is busy is obtained with the call

```
>> [p_rej,p_busy] = simulate_simple_queue(n,m);
```