Universidade Nova de Lisboa
OMNIS CIVITAS CONTRA SE DIVISA NON STABIT
Faculdade de Ciências e Tecnologia

# Discrete Stochastic Simulation

Pedro Barahona
DI/FCT/UNL
Métodos Computacionais
1st Semestre 2017/2018

# 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

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

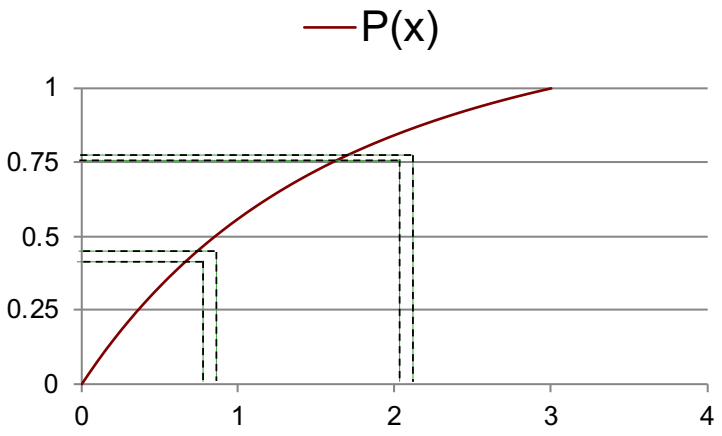
- Then, the random variable may be implemented by the inverse method in the two following steps:

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,

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

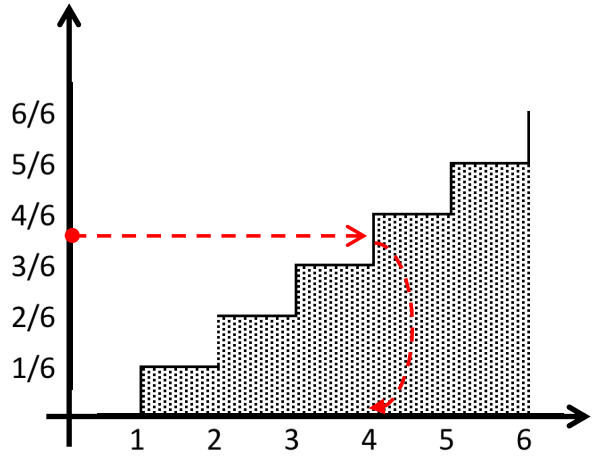
- Hence the probabilities of two values in the domain being generated is proportional to the value of their probability density function.



# 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()**.

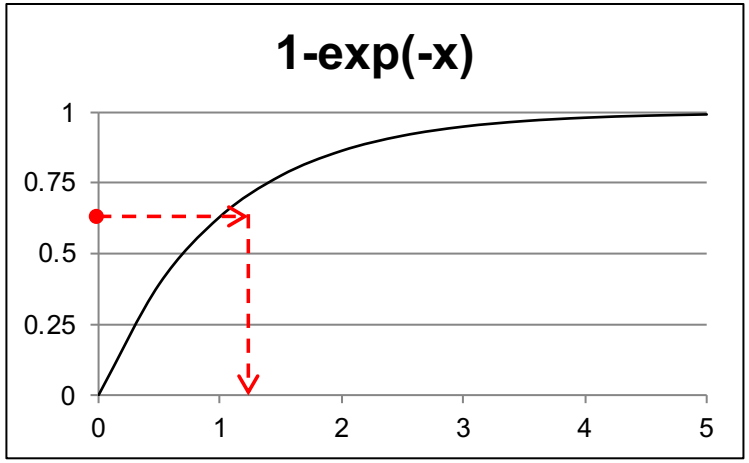



```
function v = dice();
   r = rand();
   if      r <= 1/6 v = 1;
   elseif  r <= 2/6 v = 2;
   elseif  r <= 3/6 v = 3;
   elseif  r <= 4/6 v = 4;
   elseif  r <= 5/6 v = 5;
   else             v = 6;
   end
end
```

# Inverse Method

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

- 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 λ = 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 = -log(1-r)/lambda;
end
```

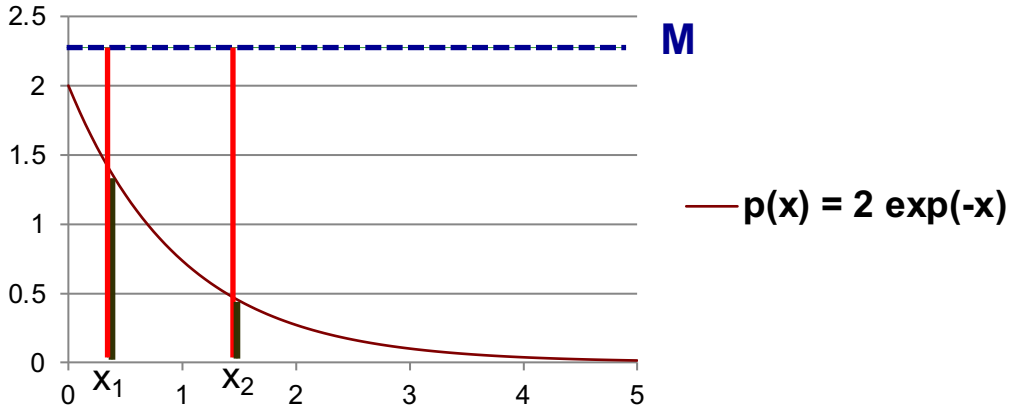
# Accept/Reject Method

- 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 **r ≤ 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

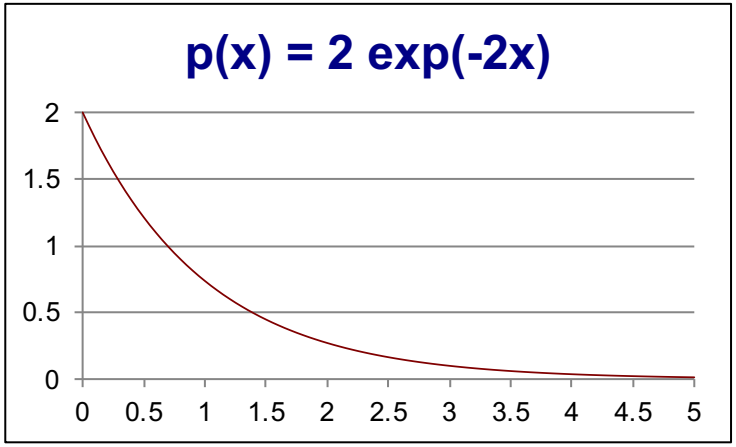
- The probability that a value $x_i$ in the domain **a..b** is accepted is thus
  - Probability that $\mathbf{x_i}$ is generated, i.e. the value is between $\mathbf{x_i}$ and $\mathbf{x_i + dx}$;
  - Probability that the value is subsequently accepted, i.e. $\mathbf{p(x_i) \leq r}$.
- Given two values $\mathbf{x_1}$ 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 **p(x) = λ $e^{-\lambda x}$**, 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.
- 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 simulating the state transitions of the automaton given a set of events, until a stop condition holds.

**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 function

```
function Sts = simulate (<params>)
   st = initial_state(<params>);
   Sts = [st];
   while !stop(Sts(end), <params>)
      st = update(st);
      Sts = [Sts, st];
   end
end
```

- The function returns the *sequence of states* **Sts** given a set of problem specific parameters, **<params>**, and requires the specification of 4 auxiliary functions:
  - **initial_state(<params>)**
  - **stop(st, <params>)**
  - **update(st)**

# Example: Random Walk

- To use the generic simulation algorithm both states and events are encoded by means of *structures*, with problem specific fields.
- This generic algorithm is now instantiated to simulate **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 devrease, respectively, its current position. This leads to the following fields:

**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 single field variable, **dir**, that takes values 1 or -1, respectively.

# Example: Random Walk

- We may now instantiate the 4 functions identified before.
- The initial state has both steps and position set at zero

```
function st = initial_state()
   st.pos = 0;
   st.stp = 0;
   printf("starting\n"); % trace - can be ommited
end
```

- The stopping condition depends on the purpose of the simulation. Here we are interested to know whether a certain distance is achieved within a certain number of steps. Hence:

```
function b = stop(st, max_stp, max_pos)
   b = (abs(st.pos) >= max_pos || st.stp > max_stp);
end
```

- The max_stp and max_pos are the <params> that are input to the simulation function.

# Example: Random Walk

- The generic simulation function can thus be adapted for the random walk problem as follows:

```
function Sts = simulate_walk(max_stp, max_pos)
   st = initial_state();
   Sts = [st];
   while !stop(Sts(end), max_stp, max_pos)
      st = update(st);
      Sts = [Sts, st];
   end
end
```

- We now turn to the remaining function,
  - **update(st,ev)**

# Example: Random Walk

- The **update** function adds to the current position a step in the direction of the event.

```
function st = update(st)
   st.stp = st.stp + 1;
   st.pos = st.pos + next_step();
   % printf("step #%3i : position =%3i\n",
                                  st.stp, st.pos);
end
```

- Additionally, and for debugging purposes, it may print to the terminal the new state that was achieved, thus tracing the sequence of states.
- Of course, this statement can be commented (as shown), if one is not interested in the trace.

# Example: Random Walk

- The **next_step** function is straightforward.
- It simply generates, with equal probability, either a forward step or a backward stepm, and returns it as the field **dir** of the event.
- This random variable, taking values in the set {-1,1} can be implemented as discussed before

```
function step = next_step();
   r = rand();
   if r < 0.5          % step = 2*(round(rand())-0.5)
      step = -1;
   else
      step = +1;
   end
end
```

# Example: Random Walk

- The simulation of this random walk can be illustrated with the following call.

```
>> Sts = simulate_walk(10,3)
starting
step #  1 : position =  1
step #  2 : position =  0
step #  3 : position = -1
step #  4 : position =  0
step #  5 : position =  1
step #  6 : position =  2
step #  7 : position =  3
Sts =  1x8 struct array containing the fields:
   pos    stp
```

- In this case, the walk reached the maximum distance of 3 at step #7, and hence the simulation did not last for the maximum of 10 steps.
- But the trace shows that the system first move forward, then back (to position -1) and then eventually started to move forward until it reached position 3 and stopped.

# Example: Random Walk

- Of course, a random process may return different walks fo different simulations.
- For example:

```
>> Sts = simulate_walk(10,3)
Starting
step #  1 : position = -1
step #  2 : position = -2
step #  3 : position = -3
Sts =  1x4 struct array containing the fields:
    pos    stp
```

- In general, we may be interested to know what is the probability that a certain distance (in absolute values) is reached within a certain number of steps.
- This can be obtained by simulating the system a sufficient large number of times and count the percentage of times this distance is reached.

# Example: Random Walk

- This computation can be performed by the function below, where the parameter n is a "**sufficiently large**" number

```
function p = probability_walk(max_stp, max_pos, n)
   c = 0
   for i = 1:n
      Sts = simulate_walk(max_stp, max_pos);
      if length(Sts) < max_stp || abs(Sts(max_stp)) == max_pos
         c = c + 1;
      end
   end
   p = c / n;
end
```

- Now the probability can be estimated as

```
>> p = probability_walk(10,3,100)
   p =  0.68000
>>
```

# Example: Random Walk

- Of course, the probability is simply estimated since what is returned is the frequence.process may return different walks fo different simulations.

```
>> p = probability_walk(10,3,100)
   p =  0.68000
>> p = probability_walk(10,3,100)
   p =  0.74000
>> p = probability_walk(10,3,100)
   p =  0.67000
>> p = probability_walk(10,3,1000)
   p =  0.66200
>> p = probability_walk(10,3,1000)
   p =  0.67700
>> p = probability_walk(10,3,1000)
   p =  0.66800
>> p = probability_walk(10,3,5000)
   p =  0.69200
>> p = probability_walk(10,3,5000)
   p =  0.67780
>> p = probability_walk(10,3,5000)
   p =  0.67360
>>
```

# 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, a state, **st**, of a system may be represented by the following fields
  - **st.time_stamp**: the time the system entered the state
  - **st.server_busy:** a Boolean indicating whether the server is busy
  - **st.end_service:** the time the request being served terminates
  - **st.next_arrival:** the time of arrival of the next request
  - **st.n_accept:** the number of requests already accepted
  - **st.n_reject:** the number of requests already rejected
- The last 2 fields are not strictly needed, since their values can be computed from the sequence of states, but they are useful to answer some questions discussed before (percentage of rejected requests) and can be computed during the simulation.
- Of course, the other question (percentage of busy time) must be computed from the sequence of states that is returned by the simulation.

# Example: Queueing Systems

- Regarding the events, there are 3 types of events in this simple queueing system
  - **Accept**: a request arrives and is served (the server is idle)
  - **Reject**: a request arrives and is rejected (the server is busy)
  - **Served**: a request finishes being served
- Hence the update function has to take into account 3 types of transitions, which depend of course on the state of the system (namely, the server)
- We now discuss the adaptation of the simulation function for this system

# Example: Queueing Systems

- The generic simulation function can be adapted for the simple queueing system with a server and no queues

```
function Sts = simulate_server(max_time)
    st = initial_state();
    Sts = [st];
    while !stop(Sts(end), max_time)
        st = update(st);
        Sts = [Sts, st];
    end
end
```

- We now address the 3 auxiliary functions.
  - `initial_state()`
  - `stop(st, max_stp, max_pos )`
  - `update(st,ev)`

# Example: Queueing Systems

- The initial state has both steps and position set at zero

```
function st = initial_state()
    st.time_stamp = 0;
    st.server_busy = false;
    st.end_service = +inf; % not applicable
    st.next_arrival = next_arrival()
    st.n_accept = 0;
    st.n_reject = 0;
    printf("starting: first arrival at %6.2f\n", …
st.next_arrival); % trace - can be ommited
end
```

- The **next_arrival** auxiliary function returns the time of the next arrival, following an exponential distribution with mean time of 5 (lambda = 1/5)

```
function t_next = next_arrival()
% an exponential distribution with lambda = 0.2
% i.e. mean time between arrivals of 1/lambda = 5
    lambda = 0.2; r = rand();
    t_next = -log(1-r)/lambda;
end
```

# Example: Queueing Systems

- Before addressing the most interesting function we specify the functions
  - to stop the simulation when a state **time_stamp** exceeds the specified **max_time**

```
function b = stop(st, max_time)
   b = (st.time_stamp >= max_time);
end
```

  - To obtain a service time with a uniform distribution between 2 and 8 secs.

```
function t_busy = service_time()
% a uniform distribution between 2 and 8
   t_busy = 2 + 6*rand(); % average time = (2+8)/2 = 5
end
```

# Example: Queueing Systems

- It still remains to specify the function **update**, that encodes the state transitions. For each event type, an auxiliary function adresses the corresponding transition

```
function st = update(st)
   if       st.server_busy && ...
            st.next_arrival < st.end_service
      st = reject_event(st);      % a reject event
   elseif   st.server_busy && ...
            st.end_service <= st.next_arrival
      st = serviced_event(st);    % a serviced event
   elseif   !st.server_busy
      st = arrival_event(st);     % an arrival event
   else
      printf("error - unclear transition in state:\n")
      x = st
      st.time_stamp = + inf;
   end;
end
```

- Note the **else** statement for debugging purposes.
- Now each of the transitions may be encoded separately.

# Example: Queueing Systems

- First, a transition following an accept event.

```
function st = arrival_event(st)
   t = st.next_arrival;
   st.time_stamp = t;
   st.server_busy = true;                % server becomes busy
   st.end_service = t + service_time(); % new end service time
   st.next_arrival = t + next_arrival();% new next arrival time
   st.n_accept = st.n_accept + 1;        % increase accepted requests
   st.n_reject = st.n_reject;            % maintain rejected requests
   printf("arriving at %6.2f: busy until %6.2f; next arrival \
at %6.2f\n", t, st.end_service, st.next_arrival);
end
```

# Example: Queueing Systems

- Alternatively, a transition following a reject event.

```
function st = reject_event(st)
   t = st.next_arrival;
   st.time_stamp = t;
   st.server_busy = true;                % server is kept busy
   st.end_service = st.end_service;      % end service time is kept
   st.next_arrival = t+next_arrival(); % compute next arrival
   st.n_accept = st.n_accept;            % maintain accepted requests
   st.n_reject = st.n_reject + 1;        % increase rejected requests
   printf(”rejected at %6.2f: busy until %6.2f; next arrival \
at %6.2f\n", t, st.end_service, st.next_arrival);
end
```

# Example: Queueing Systems

- Finally, the transition following a serviced event.

```
function st = serviced_event(st,t)
   t = st.end_service;
   st.time_stamp = t;
   st.server_busy = false;              % server becomes not busy
   st.end_service = +inf;               % serving time not applicable
   st.next_arrival = st.next_arrival;   % new next arrival time
   st.n_accept = st.n_accept;           % maintain accepted requests
   st.n_reject = st.n_reject;           % maintain rejected requests
   printf("serviced at %6.2f: busy until %6.2f; next arrival \
at %6.2f\n", t, st.end_service, st.next_arrival);
end
```

- It is now possible to simulate the behaviour of the system, and fom this behaviour to answer the questions asked
  - What is the percentage of time the server is busy
  - What is the percentage of requests that are rejected

# Example: Queueing Systems

- First, a simulations for some (short) period of time.

```
>> S = simulate_server(25)
starting: first arrival at 13.83
accepted at 13.83: busy until 20.83; next arrival at 17.31
rejected at 17.31: busy until 20.83; next arrival at 22.18
accepted at 22.18: busy until 28.39; next arrival at 22.57
rejected at 22.57: busy until 28.39; next arrival at 29.51
accepted at 29.51: busy until 32.29; next arrival at 43.95
>>
```

- What is the percentage of requests that are rejected?
  - This question can be answered by inspecting the last state (though the statistics is poor – a larger period of time should be used )

```
>> a = S(end).n_accept
 a = 2
>> r = S(end).n_reject
 r = 3
>> p = r / (a + r)
 p = 0.6
```

# Example: Queueing Systems

- What is the percentage of time the server is busy
  - Since during the simulation the times in which the sderver is busy were not accumulated (as were the number of accepted and rejected requests) this question can not be answered so easily.
- Two possible alternatives to answer this question
  1. Improve the representation of the state of the system to accumulate the busy and idele times of the server; or
  2. Analyse the structure vector and sum the periods of time in which the server is busy, i.e.
     - Find all elements **k** of the structure vector S where the server is busy, i.e.

```
st(k).server busy == true
```

     - For each of these k, compute the time they remain in the state

```
st(k+1).time_stamp - st(k+1).time_stamp
```

     - Accumulate all these values starting in k = 1 and ending in k = end-1
- Both alternatives are left as an exercise.