Stochastic Simulation

Pedro Barahona
DI/FCT/UNL
Métodos Computacionais
1st Semestre 2020/2021

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 dependent on events that follow known probability distributions may be simulated if it is available a **random generator function**, that generates numbers in the interval **0..1** with a **uniform distribution**.
- This random generator is available in all programming languages. In Python, it is available through function **random()** from library **random.**
- Based on this function any nondeterministic distribution, 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 Domains

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

Continuous Domains

$$P(t) = \int_{x=a}^{b} p(x)dx$$

Probability Distributions - Inverse Method

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

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

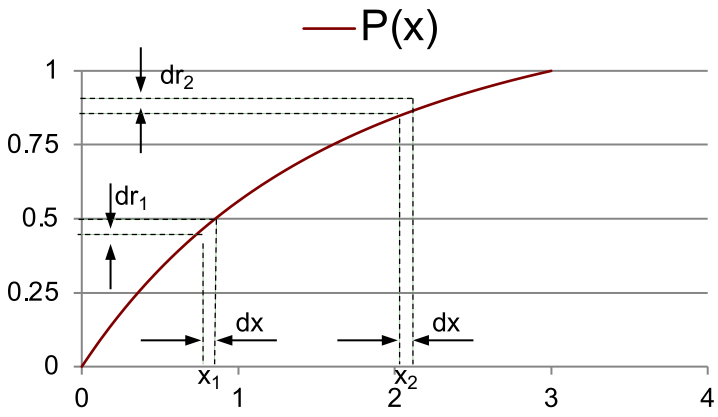
- Then, the random variable may be implemented by the inverse method in the two following steps:

  1. Generate a random number **r**, with uniform distribution in the interval **0 .. 1**;
  2. Return **x = P^{-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**. Hence,

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

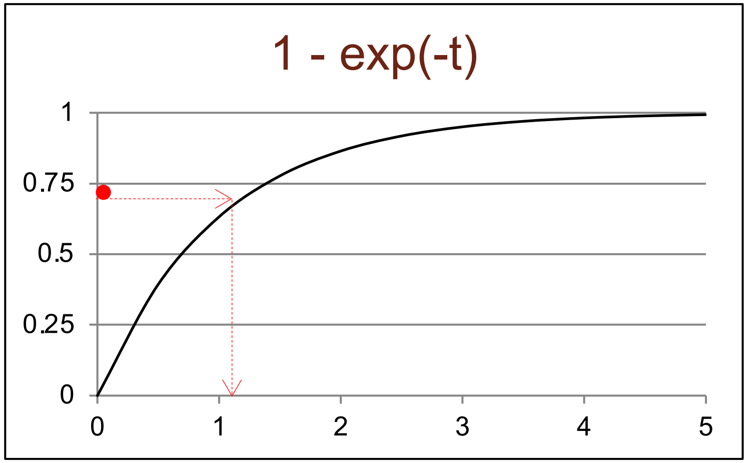
- Thus, the probabilities of two values in the domain being generated is proportional to the value of their probability density function.



Probability Distributions - 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(t) = λ $e^{-\lambda t}$**, where t ranges from 0 to ∞.
- The probability function **F(t) = (1- $e^{-\lambda t}$)** (shown for λ = 1)
- The inverse function is then **t = F^{-1}(r) = - log(1-r) / λ = - m * log(1-r)**
- Hence, these arrivals can be modelled by a variable obtained through function **exp_inv(m)**, shown below parameterised by the value of **m**.



```python
def exp_inv(mean):
    """computes the timing of an event
    according to an exponential
    distribution with the given
    mean time"""
    r = random.random()
    t = - mean * math.log(1-r)
    return t
```

Probability Distributions - Accept/Reject Method

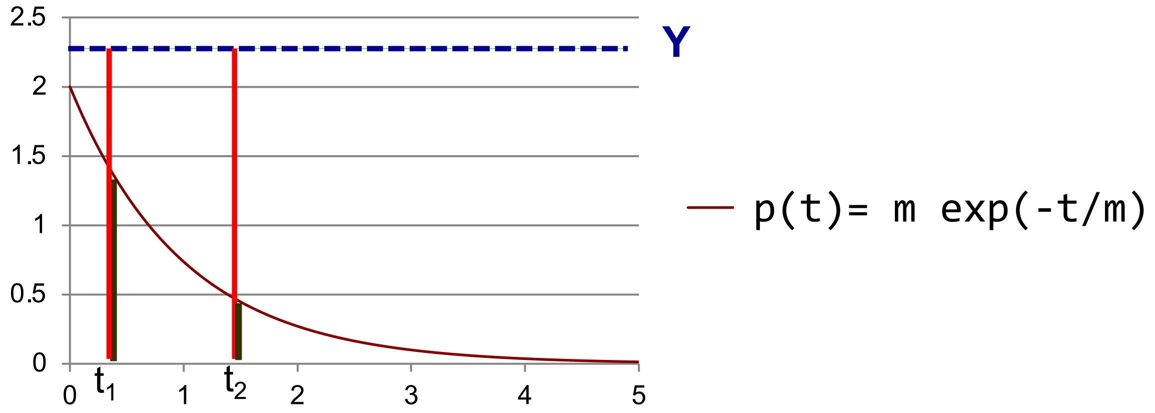
- Of course, the inverse method assumes that it is possible to obtain a closed form for $\mathbf{F^{-1}}$, the inverse of the cumulative distribution function **F**.
- When such a closed form of $\mathbf{F^{-1}}$ is not available, the random variable may be implemented by the **accept/reject method**.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

  then the random variable may be implemented in the following steps:

  1. Generate a random number **t**, with uniform distribution in the interval **a .. b**;
  2. Generate a random number **r**, with uniform distribution in the interval **0 .. Y**;
  3. Accept **x**, if **r ≤ p(t)**, 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”

Probability Distributions - Accept/Reject Method

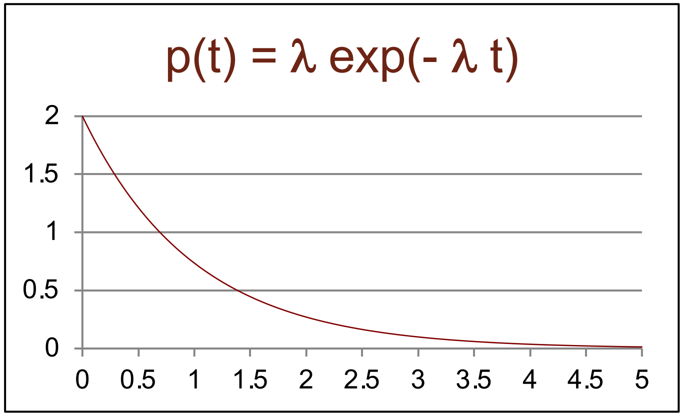
- The probability that a value t in the domain **a..b** is accepted is thus
  - Probability that **t** is generated, i.e. the value is between **x** and **x +dx**;
  - Probability that the value is subsequently accepted, i.e. **p(t) ≤ r**.
- Given two values $\mathbf{t_1}$ and $\mathbf{t_2}$, the first probability is the same for both (**dx** is the same) .
- Since **r** is generated in the range **0..Y**, their acceptance probability is, respectively, $\mathbf{p(t_1)/Y}$ and $\mathbf{p(t_2)/Y}$.
- Hence the probability of generating two values $\mathbf{t_1}$ and $\mathbf{t_2}$ is proportional to the value of their probability density function



Probability Distributions - Accept/Reject Method

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

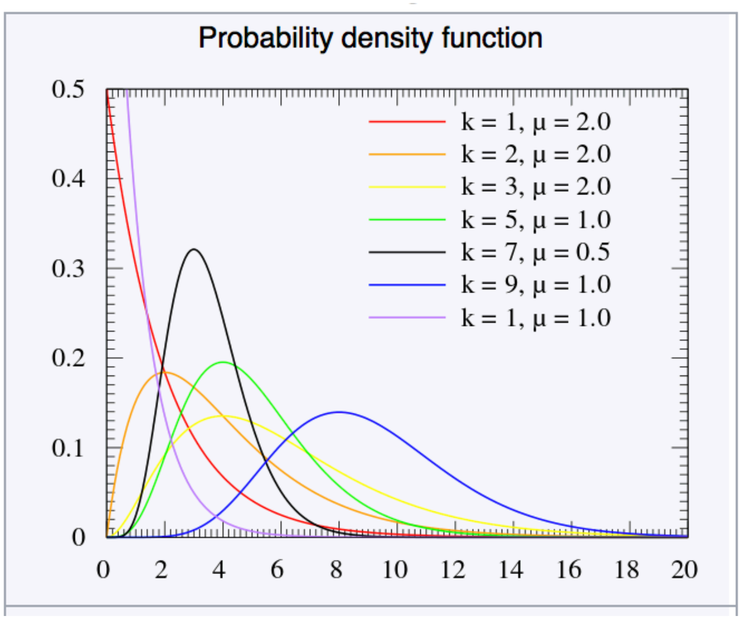
- This is a continuous distribution where **$p(t) = \lambda\ e^{-\lambda t}$**, ranging from 0 to ∞.
- The domain must then be truncated to some value **T** (**T=5** in the figure).
- The function is always less or equal to λ (we use **$Y = \lambda = 1/m$** = 2 in the figure).
- Hence, these arrivals can be modelled by a variable obtained through function **exp_ar(mean)**, shown below parameterised by the value of m = 1/λ**.**

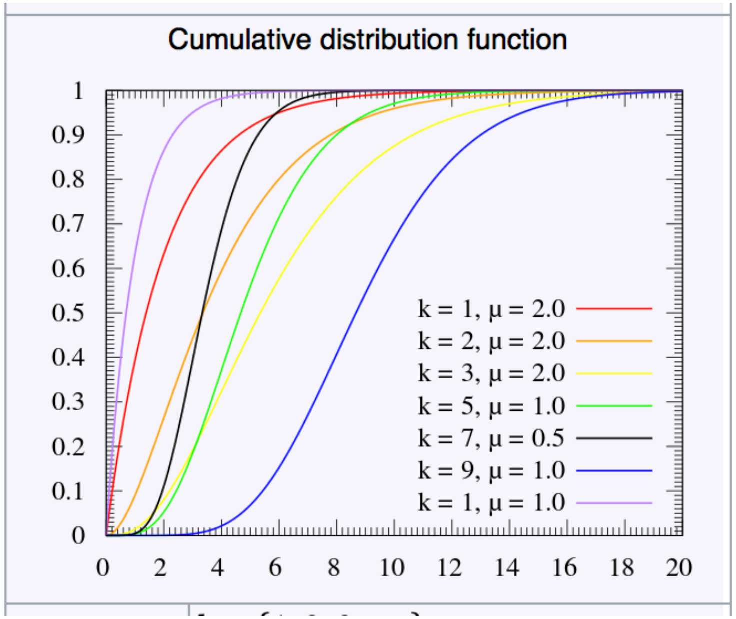


```
def exp_ar(mean):
  """computes ..."""
  accept = False
  while not accept:
    t = 10 * mean * random.random() # t < = 10*mean
    r = random.random() / mean      # Y =  1 / mean
    y = math.e**(-t/mean) / mean    # y <= 1 / mean
    accept = (r <= y)
  return t
```

Example - Erlang distribution

- The Erlang distribution is the distribution of the sum of ***k*** *independent and identically distributed random variables*, each having an *exponential distribution with mean **m***.





- Source: https://en.wikipedia.org/wiki/Erlang_distribution

Erlang distribution

- The Erlang distribution is the distribution of the sum of ***k*** *independent and identically distributed random variables*, each having an *exponential distribution with mean **m***.

- Its pdf (probability density function) is the following:

$$\boldsymbol{f(x;k,m) = \frac{x^{k-1}e^{-x/m}}{m^k(k-1)!}}$$

- Hence, a significant difference with respect to the uniform and exponential distribution is that it cannot be generated by the inverse method (that requires obtaining **x** as a function of **f**).

- Hence it can be obtained by the general accept-reject method, assuming that it is truncated at some convenient **x** (for example, $\mathbf{x_{max}}$ = **10*k*m**) and max value (it depends on **k** and **m**, but for k > 1 and m > 0.2, $\mathbf{f_{max}}$ **= 2** is a “safe” value).

- Of course, given the definition above it can be simulated as the sequence of **k** exponential distributions, each with a mean **m**.

Erlang distribution

$$f(x; k, m) = \frac{x^{k-1} e^{-x/m}}{m^{k}(k-1)!}$$

- Adopting the accept-reject method the distribution can be obtained by adapting the generic ar function (seen before) to the Erlang pdf, as follows

```
def erlang_ar(k,mean):
    """generates events with an Erlang (k,m) distribution,
    adopting the ccept-reject method. """
    accept = False
    while not accept:
        t = 10 * k * mean * random.random()  # t <= 10 * k * mean
        r = 2 * random.random()                # Y = 2
        num = t**(k-1) * math.e**(-t/mean)
        den = mean**k * math.factorial(k-1)
        y = num/den
        accept = (r <= y)
    return t
```

- In this case, we generate values of x, up to a maximum 10*k*m. In this range of values for x, the values of the pdf are all below 2 (as discussed)

Erlang distribution

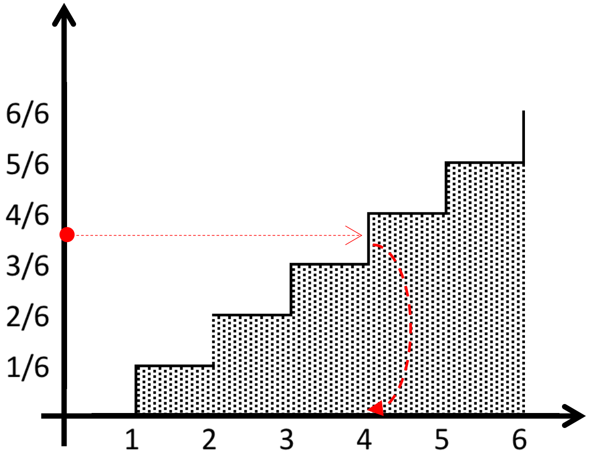
- Since the Erlang distribution corresponds to the the sum of *k independent and identically distributed random variables*, each having an *exponential distribution with mean m/k*, its generator can be also obtained alternatively as:

```
def erlang_ke(k,mean):
    """generates events with an Erlang (k,m) distribution, taking
    it account that this distribution corresponds to a sequence of
    k independent exponential distibutions with mean m. """
    t = 0
    for i in range(k):
        t = t + exp_inv(mean);
    return t
```

Discrete Probability Distributions

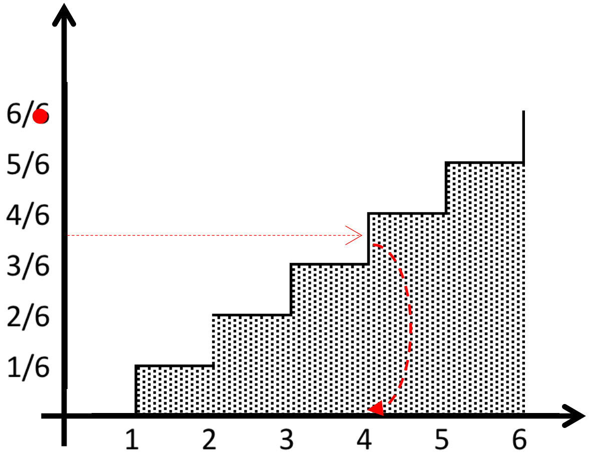
- In many cases, the events are modelled by discrete numbers, namely integers. If all the numbers have the same probability the method to generate these numbers is straightforward, and adapts the inverse method seen before.

- This can be illustrated with 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 a random number **r**, generated randomly.



Discrete Probability Distributions

- The discrete events representing the throwing of a dice can be thus modelled as:



```
import random
def dice():
    r = random.random();
    if   r <= 1/6:
        return 1
    elif r <= 2/6:
        return 2
    elif r <= 3/6:
        return 3
    elif r <= 4/6:
        return 4
    elif r <= 5/6:
        return 5
    else:
        return 6
```

- Or better still by the equivalent, and more compact definition

```
def dice():
    r = random.random();
    return math.ceil(6*r)
```

Discrete Probability Distributions

- This technique may be generalised to non equal probable events. If we know the relative probability of events we may adopt the inverse method.
- For example if we have 3 possible outcomes, with probabilities 50%, 30% and 20%, we can still generate a number 1 to 3 as follows:
- Generate a random number, **r**, in the range 0..1
    - if r <= 0.5 return 1
    - if 0.5 < r <= 0.8 return 2
    - if 0.8 < r return 3
- We leave as an exercise to generate a number **k**, from 1 to n from given a list **P** with **n** numbers representing the relative probabilities of the **n** possible outcomes

```
def proportional_random(P):
    ...
   return k
```

- Note: Use this function to implement function **select_among_index** discussed above in the TSP

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.

Simulation of Simple Queuing System

- We illustrate the simulation of this type of processes, with a (very) simple queuing system with the following characteristics:
  1. Clients arrive at some server, according to a exponential distribution with mean time m_1.
  2. The server dispatches a client with an Erlang distribution (k, m_2), i.e. a sequence of k exponential tasks with average time of m_2 minutes each.
  3. Every time a client arrives and finds the server busy, it gives up from being serviced.
- The goal is to compute the percentage of clients that are not served.
- To obtain the percentage of clients lost, we can simulate the behaviour of this system for a sufficient long time, and monitor the number of clients that are served as well as the number of clients that are lost (rejected)
- Hence, throughout the simulation, we should maintain state variables (that represent the state of the system) and monitoring variables (that represent the information that we are interested in studying).

Simulation of Simple Queuing System

For study such a system we can define the following modelling variables:

- **States:**
  - The server may be busy or idle, and this may be represented by a Boolean variable **busy**;
- **Monitoring:**
  - The number of clients accepted and rejected may be represented by two integer variables: **na** and **nr**, respectively.
- **Events:**
- Two types of events are considered:
  - a) arrival of a client.
    - The arrival time of a client is generated after the arrival of the previous client
  - b) leaving of a (served) client.
    - The leaving time is generated when a client starts to be served.
- **Transitions:**
  - How to change the above variables upon occurrence of the above events.

Simulation of Simple Queuing System

- The programming of this system can thus be made representing the following transition table:

State Variables	event (at time t)	variables update			next event (s)	
		state variables	monitoring variables			
busy	type	busy	n_accepted	n_rejected	type	time
FALSE	arrival	-> TRUE	+1	=	arrival leaving	t + exp(λ) t + Er(n,λ)
TRUE	arrival	=	=	+1	arrival	t + exp(λ)
TRUE	leaving	-> FALSE	=	=	-	-

Simulation of Simple Queuing System

- The programming of this system can thus be made in Python by
- Stating the initial conditions (variables busy, n_accepted, n_rejected)
- Specify a termination condition and
    - Encode the transition table
- Return the intended results

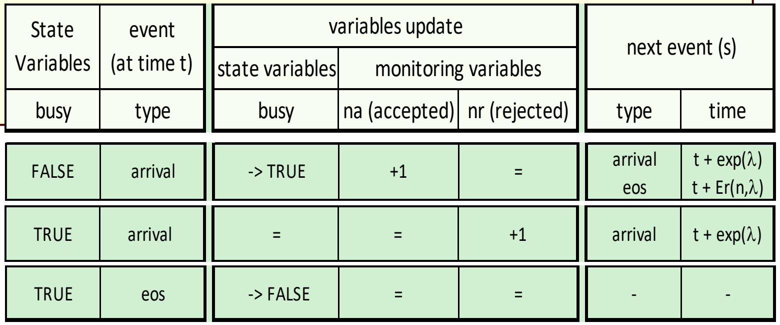
State Variables	event (at time t)	variables update			next event (s)	
		state variables	monitoring variables			
busy	type	busy	na (accepted)	nr (rejected)	type	time

```
def simple_queue_simulation (mean_1, k, mean_2):
    “”” computes the percentage of rejected clients in a system with
    one server, where clients arrive according to an exponential
    distribution with parameter lbd_1, and are served according to an
    erlang distribution with parameters k and lbd_2”””
    busy = False
    n_accepted = 0
    n_rejected = 0
    next_arrival_time = 0
    next_leaving_time = 0
    while n_accepted + n_ rejected < 1000   # simulate 1000 clients
        ...
    return 100.0 * n_rejected/(n_accepted + n_rejected)
```

Simulation of Simple Queuing System

- Programming the simulation loop follows the transition table discussed:

```
while n_accepted + n_rejected < 1000:   # simulate 1000 clients
  if not busy or next_arr_time < next_eos_time:
    if not busy:
       busy = True
       n_accepted = n_accepted + 1
       next_leaving_time = next_arrival_time + erlang_ke(k, mean_2)
    else:
       n_rejected = n_rejected + 1
    next_arrival_time = next_arrival_time + exp_inv(mean_1)
  else:
    busy = false
```



State Variables	event (at time t)	variables update			next event (s)	
		state variables	monitoring variables			
busy	type	busy	na (accepted)	nr (rejected)	type	time
FALSE	arrival	-> TRUE	+1	=	arrival eos	t + exp(λ) t + Er(n,λ)
TRUE	arrival	=	=	+1	arrival	t + exp(λ)
TRUE	eos	-> FALSE	=	=	-	-

- Of course, the distributions of the arrival and leaving times must be encoded with the appropriate functions, discussed earlier.

Example: Traveling Salesperson Problem

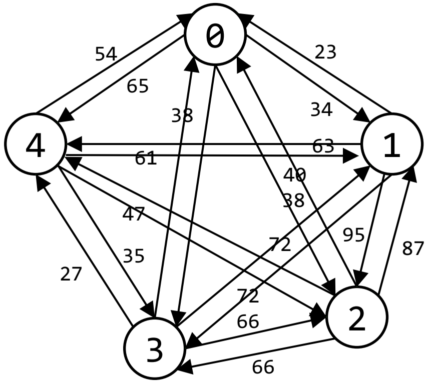
- Random numbers can also be used in the solution of hard problems, although in an approximate way. We illustrate this technique with the TSP – Traveling Salesperson Problem.

TSP Problem:

- Given a number of cities where a saleswoman operates, the goal is to find the sequence of visits such that the cost of travelling is minimal.

- The problem may be modelled by a graph where the edges have costs (e.g. distances, fuel, or time), and an example is shown below.

Graph



Adjacency Matrix

	0	1	2	3	4
0	-	34	38	42	65
1	23	-	95	72	63
2	40	87	-	66	51
3	38	72	66	-	27
4	54	61	47	35	-

Example: Traveling Salesperson Problem

- A solution of the problem, is a sequence of the cities, **Visited**, that represents the order in which they are visited.

- If the cities are selected randomly, and the solving process is repeated several times, different solutions are obtained.

- Eventually good solutions, or even the best solution, is obtained.

- Hence this problem can be solved, with the following function

```
def tsp(AdjMat):
    """the function solves the TSP problem in a graph given by
    its Adjacency matrix representation. It returns the sequence
    of nodes that are visited in the circuit, the cost of all the
    edges of the circuit, and their sum."""
    ....
    return(Visited, Edges, sum(Edges))
```

Example: Traveling Salesperson Problem

- The solution is found by starting with:
  - obtaining **n**, the number of nodes in the Adjacency matrix
  - a list, **Visited**, with the initial node (any node will do but we use node 0); and
  - a list, **ToVisit**, of all the nodes still to visit (i.e. [1,2…,n-1]); and
  - an empty list of the **Edges** already travelled through;
- Then a while cycle is performed to fill the Visited and Edges list, until there are no more nodes to visit.

```
def tsp(AdjMat):
    ""”...""”
    n = len(AdjMat);
    Visited = [0]
    ToVisit = [i for i in range(1,n)]
    Edges = []
    while len(ToVisit) > 0:
        ...
    return(Visited, Edges, sum(Edges))
```

Example: Traveling Salesperson Problem

- The cycle iterates the following instructions:
    - It sets the last visited node as the x node; and
    - It selects a node to visit, from the ToVisit nodes, given x, and the graph; and
    - It sets the next node to visit as the y node; and
    - Appends the cost of the edge x -> y to the Edges list; and
    - Appends node y to the Visited list; x -> y to the Edges list; and finally
    - Removes the y node from the nodes to visit;
- In the end of the cycle, edge from the last visited node to node 0 is appended to the Edges list, and the results are returned.

```
    ...
    while len(ToVisit) > 0:
        x = Visited[-1]
        i = select_random_index(x, ToVisit, AdjMat)
        y = ToVisit[i]
        Edges.append(AdjMat[x][y])
        Visited.append(y)
        ToVisit = remove_node(i, ToVisit)
     Edges.append(AdjMat[Visited[-1]][0])
     return(Visited, Edges, sum(Edges))
```

Example: Traveling Salesperson Problem

- Removing the node from the ToVisit list is made with the obvious list operations

```
def remove_node(i, L):
    """ returns list L with the element in position i removed"""
    P = L[0:i]
    P.extend(L[i+1:])
    return P
```

- Of course the key to obtain a good solution is to select a good “next node” y

```
    i = select_index(x, ToVisit, AdjMat)
```

and a naïve strategy is simply to obtain an arbitrary index, as in

```
def select_random_index(x,L, Matrix):
    """ selects an arbitrary index from list L"""
    k = len(L)
    r = random.random()
    z = math.floor(k*r)
    return z
```

Example: Traveling Salesperson Problem

- Although it looks very naïve, and it is, this random selection of the next node, if the solving is repeated a large number of times it may eventually obtain good or even optimal (minimal solutions). Other strategies are possible though.

- The node to be selected from the ToVisit list might be that closer to node x

```
    i = select_closer_index(x, ToVisit, AdjMat)
```

- This is an eager selection, and apart from ties in some edges, it always provides the same solution. It is usually a fairly good solution, but not optimal in most cases, namely when the graphs are large.

- A better strategy is to allow some randomness in the selection. This can be done by selecting, not the closest node, but selecting arbitrarily one of the neighbour nodes of x with a probability that is inversely proportional to its cost:

```
    i = select_likely_index(x, ToVisit, AdjMat)
```

- Programming these alternative functions is left as an exercise.