Author: Roberto Rossi Email: robros@gmail.com Website: https://gwr3n.github.io

Roberto Rossi, *Inventory Analytics*. Cambridge, UK: Open Book Publishers, 2021, https://doi.org/10.11647/OBP.0252

Please direct all enquiries to the author.

ISBN Paperback: 978-1-80064-176-1
ISBN Digital (PDF): 978-1-80064-177-8
DOI: 10.11647/OBP.0252

Appendix

https://doi.org/10.11647/OBP.0252.07

Introduction

In this Appendix, we provide relevant formal background on Poisson processes, Discrete Event Simulation, and Stochastic Dynamic Programming.

Topics

Poisson processes

In this section we discuss the nature and properties of Poisson processes.

A typical example of an experiment that can be modelled via a Bernoulli random variable is the toss of a coin; assuming the coin is fair, then $p = 0.5$.

Definition 28. *A Bernoulli random variable takes value* 1 *with probability* p*, and* 0 *with probability* $q \triangleq p - 1$.

Definition 29. *A Bernoulli stochastic process is a collection* $\{X_1, X_2, \ldots\}$ *of Bernoulli random variables.*

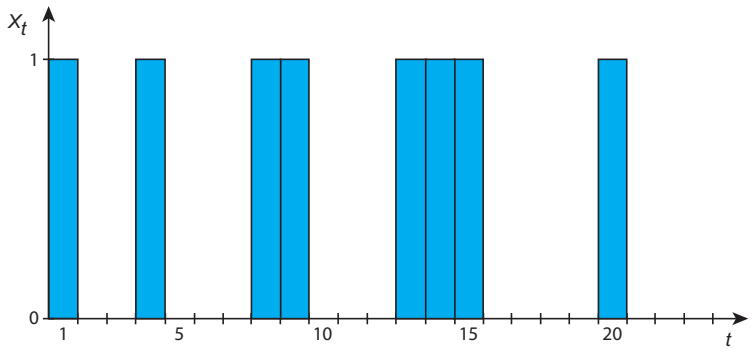


Fig. 120 A Bernoulli stochastic process.

In Fig. 120 we illustrate the dynamics of a Bernoulli process.

Definition 30. *A binomial random variable with parameters* n *and* p *is the sum of* n *Bernoulli random variables with parameters* p.

Let X be a binomial random variable, its probability mass function is then

$$\Pr(X = k) = \binom{n}{k} p^k q^{n-k};$$

its cumulative distribution function is

$$F(x) = \sum_{k=0}^{x} \binom{n}{k} p^k q^{n-k}.$$

Definition 31. *A binomial stochastic process is a collection* $\{X_1, X_2, \ldots\}$ *of Binomial random variables.*

In Fig. 121 we illustrate the dynamics of a binomial process.

In inventory control, consider a series of time periods and a setting in which in each time period it is possible to observe at most one unit of demand with probability p. A Bernoulli random variable models the occurrence of one unit of demand in any given period. A binomial random variable models the total demand observed in a sequence of n independent periods.

Definition 32. *A Poisson random variable with parameters* λ *is the limiting case of a binomial random variable when* $n = \infty$ *and* $np = \lambda$.

Let X be a Poisson random variable, its probability mass function is then

$$\Pr(X = k) = \frac{\lambda^k e^{-\lambda}}{k!};$$

its cumulative distribution function is

$$F(x) = \sum_{k=0}^{x} \frac{\lambda^k e^{-\lambda}}{k!}.$$

A Poisson random variable therefore models a system with a large number of possible events, each of which is rare; events occur with a known constant mean rate λ and independently of the time since the last event occurred.

Unfortunately, it is difficult to interpret a Poisson process in a way similar to that used for a binomial process. In fact, units of demand may occur at arbitrary positive times, and the probability p of a unit of demand at any particular instant is infinitely small. This means that there is no very clean way of describing a Poisson process in terms of the probability of an arrival at any given instant.

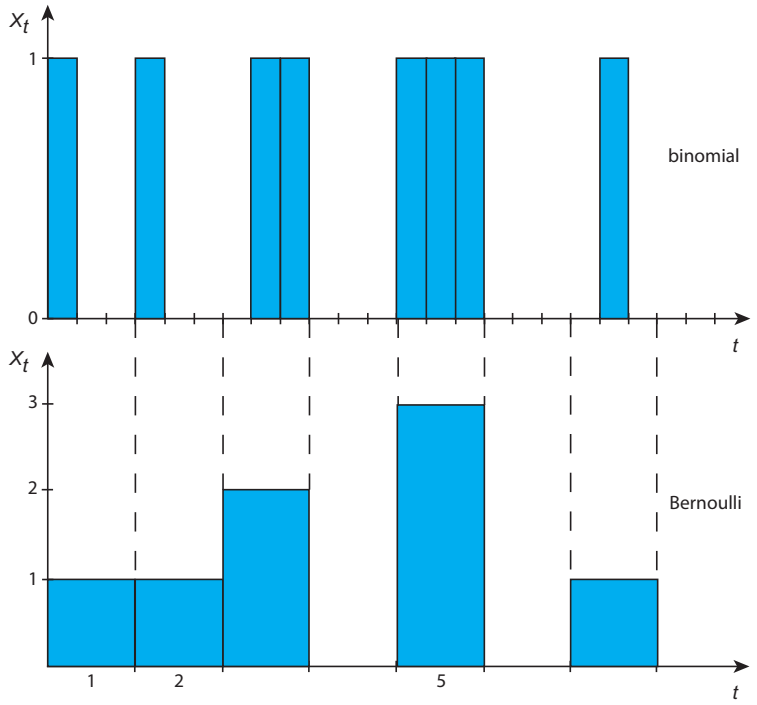


Fig. 121 A binomial stochastic process where $n = 3$, and its underpinning Bernoulli stochastic process.

It is more convenient to define a Poisson process variable in terms of the sequence of interarrival times, $X_1, X_2, \ldots$, which are defined to be independently and identically distributed random variables.

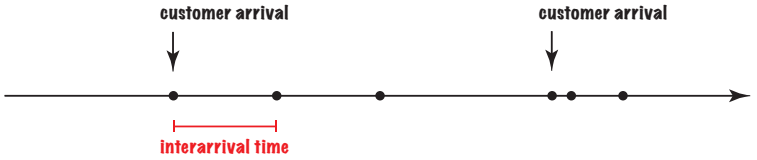


Fig. 122 A Poisson stochastic process seen as an arrival process in terms of interarrival times between successive events.

Definition 33. *An arrival process $\{S_1, S_2, \ldots\}$ is a sequence of increasing random variables, that is $0 < S_1 < S_2 < \ldots$, which are called arrival epochs. In this process, random variable $X_i \triangleq S_{i+1} - S_i$, which is called the interarrival time between event i and event $i+1$, is a positive random variable, that is $Pr(X_i \leq 0) = 0$.*

Condition $\Pr(X_i \leq 0) = 0$ effectively means that events cannot occur simultaneously.

Definition 34. *A renewal process is an arrival process for which the sequence of interarrival times is a sequence of independent and identically distributed random variables.*

Definition 35. *A Poisson process is a renewal process in which the interarrival intervals follow an exponential distribution; i.e. for some real $\lambda > 0$, each X_i has the probability density function $f(x) = \lambda e^{-\lambda x}$.*

The parameter λ is called the rate of the process; for any interval of size t, λt is the expected number of arrivals in that interval. Thus λ is called the arrival rate of the process.

Let X be an exponential random variable, its probability density function is

$$f(x) = \lambda e^{-\lambda x};$$

its cumulative distribution function is

$$F(x) = 1 - e^{\lambda x}.$$

Definition 36. *A random variable X possesses the memoryless property if it is positive, i.e. $Pr(X \leq 0) = 0$, and for every $s \geq 0$ and $t \geq 0$*

$$Pr(X \geq s + t | X > s) = Pr(X \geq t). \tag{44}$$

Lemma 58. *A continuous (resp. discrete) random variable X is exponential (resp. geometric) if and only if it possesses the memoryless property.*

Proof. This proof will only cover the continuous case.

To show that ($\rightarrow$) if X is exponential, then it possesses the memoryless property, observe that

$$\begin{aligned}
\Pr(X \geq s + t | X > s) &= \frac{\Pr(X \geq s + t \cap X > s)}{\Pr(X > s)} \\
&= \frac{\Pr(X \geq s + t)}{\Pr(X > s)} \\
&= \frac{e^{-\lambda(s+t)}}{e^{-\lambda s}} \\
&= e^{-\lambda t} \\
&= \Pr(X \geq t)
\end{aligned}$$

We next show that ($\leftarrow$) if X possesses the memoryless property, then it is exponential. Observe that $\Pr(X \geq s + t | X > s)\Pr(X \geq s) = \Pr(X \geq s + t)$, then

$$\Pr(X \geq s + t) = \Pr(X \geq t)\Pr(X \geq s). \tag{45}$$

Let $h(x) = \ln(\Pr(X \geq x))$ and observe that since $\Pr(X \geq x)$ is nonincreasing in x, $h(x)$ is also. Moreover, Eq. 45 implies that $h(s + t) = h(s) + h(t)$ for all $s \geq 0$ and $t \geq 0$. These two statements imply that $h(x)$ must be linear in x, and $\Pr(X \geq x)$ must be exponential in x. □

Let X be the waiting time until some given arrival, then Eq. 44 states that, given that the arrival has not occurred by time t, the distribution of the remaining waiting time (given by x on the left side of Eq. 44) is the same as the original waiting time distribution (given on the right side of Eq. 44), i.e. the remaining waiting time has no "memory" of previous waiting.

From Definition 35 and Lemma 58, it immediately follows that the portion of a Poisson process starting at an arbitrary time $t > 0$ is a probabilistic replica of the process starting at 0; that is, the time until the first arrival after t is an exponentially distributed random variable with parameter λ, all subsequent arrivals are independent of this first arrival and of each other, and all have the same exponential distribution.

Lemma 59. *Let X and Y be independent Poisson random variables with rates λ_X and λ_Y, respectively; then $X + Y$ is a Poisson random variable with rate $\lambda_X + \lambda_Y$.*

Proof. Let $Z = X + Y$ and $\lambda = \lambda_X + \lambda_Y$, then $\Pr(Z = z) = \sum_{j=0}^{z} \Pr(X = j)\Pr(Y = z - j) = \frac{e^{-\lambda}}{z!} \sum_{j=0}^{z} \binom{z}{j} \lambda_X^j \lambda_Y^{z-j} = \frac{e^{-\lambda}}{z!}(\lambda_X + \lambda_Y)^z$. The last step, which was obtained by using binomial expansion, concludes the proof: $\Pr(Z = z) = \frac{e^{-\lambda}}{z!}\lambda^z$. □

Discrete Event Simulation

A QUEUEING SYSTEM is a generic model that captures a variety of real-world scenarios: ticket offices, call centers, etc.

We consider a single teller who provides a service requiring a certain **service time** to be completed; customers arrive randomly and, if the teller is already busy, wait in a queue. The dynamics of the system are shown in Fig. 123.

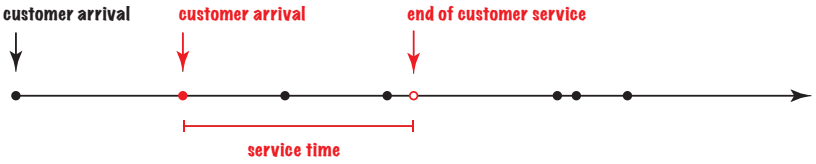
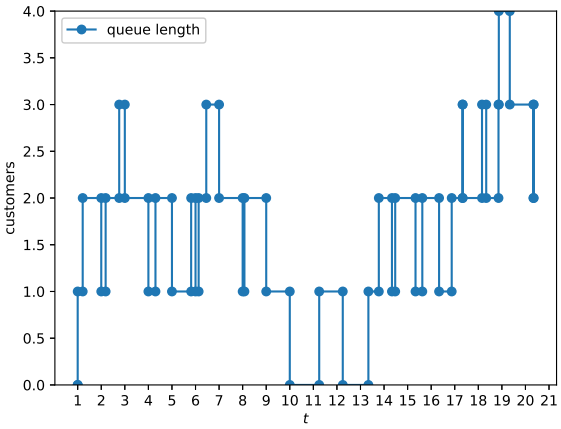


Fig. 123 The dynamics of the queueing system.

THE QUEUE displays the behaviour shown in Fig. 124.




```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

def plot_queue(values, label):

   # data
   df=pd.DataFrame({'x':
         np.array(values)[:,0], 'fx':
         np.array(values)[:,1]})

   # plot
   plt.xticks(range(len(values)),
            range(1,len(values)+1))
   plt.ylim(min(np.array(values)[:,1]),
         max(np.array(values)[:,1]))
   plt.xlabel("t")
   plt.ylabel("customers")
   plt.plot( 'x', 'fx', data=df,
         linestyle='-', marker='o',
         label=label)
```

Listing 75 Plotting the queue length in Python.

Fig. 124 The behaviour of the queue simulated in our numerical example.

DISCRETE EVENT SIMULATION (DES) is a modelling framework that can be used to model queueing systems. The DES model for the queueing system previously described is captured by the flow diagram[69] in Fig. 125.

THE QUEUE is the scarce resource modelled by class `Queue`: it features a `push` method and a `pop` method to add and remove customers. It also features a `review_queue` method to record the length of the queue at a given point in time.

THE DES ENGINE is captured in class `DES`. The engine comprises a method `start` that initates the loop by which the engine extracts events from the event stack `events`, and executes method `end` of for each of them. Finally, the engine comprises a method `schedule` to schedule an event after a given time lag.

[69] Arnold H. Buss. A tutorial on discrete-event modeling with simulation graphs. In C. Alexopoulos, I Kang, W. R. Lilegdon, and D. Goldsman, editors, *Proceedings of the 1995 Winter Simulation Conference ed.*, Arlington, Virginia, 1995.

Listing 76 A DES to model a queueing system in Python.

```
from queue import PriorityQueue

############################
##    QUEUEING SYSTEM   ##
############################

class Queue:
   def __init__(self, initial_size):
      self.s = 0
      self.levels = [[0, self.s]]

   def push(self):
      self.s += 1

   def pop(self):
      self.s -= 1

   def queue_empty(self):
      return self.s == 0

   def review_queue(self, time):
      self.levels.append([time, self.s]) # review queue

########################
##     DES ENGINE     ##
########################

class Event():
   pass

class EventWrapper():
   def __init__(self, event):
      self.event = event

   def __lt__(self, other):
      return self.event.priority < other.event.priority # low numbers denote high
           priority

class DES():
   def __init__(self, end):
      self.events, self.end, self.time = PriorityQueue() , end, 0

   def start(self):
      while True: # cycle until self.time < self.end
         event = self.events.get() # extract an event
         self.time = event[0]
         if self.time < self.end:
            event[1].event.end() # call method end() for the event
         else:
            break

   def schedule(self, event: EventWrapper, time_lag): # schedule an event
      self.events.put((self.time + time_lag, event))

########################
##       EVENTS       ##
########################

class CustomerArrival(Event):
   def __init__(self, des: DES, service_time: float, arrival_rate: float, queue:
        Queue):
      self.des, self.t, self.r, self.q = des, service_time, arrival_rate, queue
      self.priority = 1 # lowest priority

   def end(self):
      self.q.review_queue(self.des.time)
      if self.q.queue_empty():
         self.des.schedule(EventWrapper(EndOfCustomerService(self.des, self.t,
              self.q)), self.t)
      self.q.push()
      self.q.review_queue(self.des.time)
      self.des.schedule(EventWrapper(self), np.random.exponential(1.0/self.r)) #
           schedule another arrival

class EndOfCustomerService(Event):
   def __init__(self, des: DES, service_time: float, queue: Queue):
      self.des, self.t, self.q = des, service_time, queue
      self.priority = 0 # highest priority

   def end(self):
      self.q.review_queue(self.des.time)
      self.q.pop()
      if not(self.q.queue_empty()):
         # schedule end of customer service after self.t time periods
         self.des.schedule(EventWrapper(EndOfCustomerService(self.des, self.t,
              self.q)), self.t)
      self.q.review_queue(self.des.time)
```

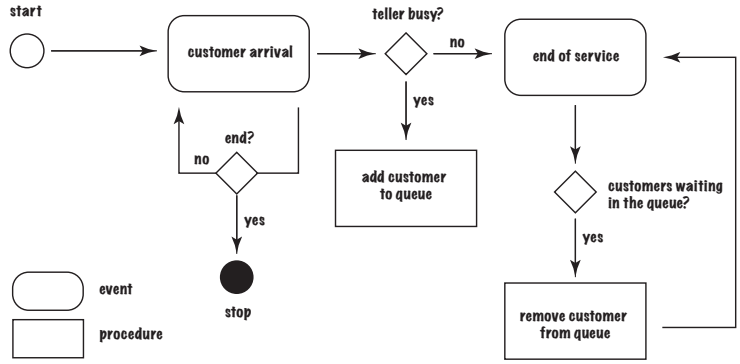


Fig. 125 DES flow diagram for the queueing system.

Events are necessary to properly capture the dynamics of the system. `Event` is an empty class capturing a generic event, `EventWrapper` is a wrapper class that allows us to express relative priorities between event types. For instance, suppose that in an inventory system with no stock available, an order receipt event is scheduled at the same time as a customer demand, which event should be processed first? The field `priority` in each event class allows us to break such a tie.

There are two main events we need to consider: `CustomerArrival` and `EndOfCustomerService`. `CustomerArrival` models the random arrival of a customer. The `end` method schedules an `EndOfCustomerService` event if the queue is empty, otherwise it increments the number of customers in the queue. Finally, it triggers a new `CustomerArrival` event after a time interval exponentially distributed with rate parameter equal to `customer_arrival_rate`. The `end` method of `EndOfCustomerService` event, decrements the number of customers in the queue, and if the queue is not empty, triggers another `EndOfCustomerService` event after `customer_service_time` time periods.

The Python code for simulating our example is shown in Listings 75, 76, and 77. After running the code, we find that the mean queue length is 1.75 customers (standard deviation: 0.93) and that the maximum observed queue length over the simulation horizon is 4 customers.

Listing 77 Simulating a queueing system in Python.

```
np.random.seed(1234) # set a random seed to ensure replicability
q = Queue(0) # create an empty queue
N = 20 # simulation horizon
des = DES(N)
customer_service_time, customer_arrival_rate = 1, 1
d = CustomerArrival(des, customer_service_time, customer_arrival_rate, q)
des.schedule(EventWrapper(d), 0) # schedule the first arrival
des.start()

print("Mean queue length:\t"+ '%.2f' % np.average(np.array(q.levels)[:,1]))
print("St. dev. queue length:\t"+ '%.2f' % np.std(np.array(q.levels)[:,1]))
print("Max queue length:\t"+ '%.2f' % max(np.array(q.levels)[:,1]))

plot_queue(q.levels, "queue length")
plt.legend()
plt.show()
```

Stochastic Dynamic Programming

DYNAMIC PROGRAMMING is a framework for modeling and solving sequential decision making problems. The framework was originally introduced by Bellman in his seminal book *Dynamic Programming*[70] to deal with multistage decision processes under uncertainty. The framework takes a "functional equation" approach to the discovery of optimum policies. Although originally devised to deal with problems of decision making under uncertainty, dynamic programming can also solve deterministic problems.

[70] Richard Bellman. *Dynamic Programming*. Princeton Univ. Pr., 1957.

TO MODEL AND SOLVE a problem via dynamic programming, one has to specify:

- a **planning horizon** comprising n periods;
- the finite set S_t of possible **states** in which the system may be found in period t, for $t = 1, ..., n$;
- the finite set A_s of possible **actions** that may be taken in state $s \in S_t$;
- the **state transition function** $g_t : S_t \times A_s \to S_{t+1}$ that identifies the state $s' \in S_{t+1}$ towards which the system transitions if action $a \in A_s$ is taken in state $s \in S_t$;
- the **immediate cost** (resp. profit) $c_t(s, a)$ incurred if action $a \in A_s$ is taken in state $s \in S_t$ of period t;
- the **functional equation** $f_t(s)$ denoting the minimum total cost (resp. maximum total profit) incurred over periods $t, t+1, \ldots, T$ when the system is in state $s \in S_t$ at the beginning of period t.

Without loss of generality in what follow we will consider a cost minimisation setting.

IN DETERMINISTIC DYNAMIC PROGRAMMING one usually deals with functional equations taking the following structure

$$f_t(s) = \min_{a \in A_s} c_t(s, a) + f_{t+1}(g_t(s, a)),$$

where the boundary condition of the system is $f_{T+1}(s) \triangleq 0$, for all $s \in S_{T+1}$. Let the initial state of the system at the beginning of the first period be s, the goal is to determine $f_n(s)$.

Given the current state s and the current action a in period t, we know with certainty the cost during the current stage and — thanks to the state transition function g_t — the future state towards which the system transitions.

In practice, however, even if we know the state of the system at the beginning of the current stage as well as the decision taken, the state of the system at the beginning of the next stage and the current period reward are often random variables that can only be observed at the end of the current stage.

STOCHASTIC DYNAMIC PROGRAMMING deals with problems in which the current period reward and/or the next period state are random, i.e. with multi-stage stochastic systems. The decision maker's goal is to maximise expected (discounted) reward over a given planning horizon. In their most general form, stochastic dynamic programs deal with functional equations taking the following structure

$$f_t(s) = \min_{a \in A_s} c_t(s,a) + \alpha \sum_{j \in S_{t+1}} p_{sj}^a f_{t+1}(j).$$

where

- $c_t(s,a)$ is the **expected immediate cost** (resp. profit) incurred if action $a \in A_s$ is taken in state $s \in S_t$ of period t;
- α is the **discount factor**;
- p_{sj}^a be the **transition probability** from state $s \in S_t$ towards state $j \in S_{t+1}$, when action $a \in A_i$ is taken;
- $f_t(s)$ is the minimum **expected total cost** (resp. maximum total profit) that can be attained during stages $t, t+1, \ldots, n$, if the system is in state s at the beginning of period t.

Let the initial state of the system at the beginning of the first period be s, once more the goal is to determine $f_n(s)$.

Example 40. *Consider a 3-period inventory control problem. At the beginning of each period the firm should decide how many units of a product should be produced. If production takes place for x units, where $x > 0$, we incur a production cost $c(x)$. This cost comprises both a fix and a variable component: $c(x) = 0$, if $x = 0$; $c(x) = 3 + 2x$, otherwise. Production in each period cannot exceed 4 units. Demand in each period takes two possible values: 1 or 2 units with equal probability (0.5). Demand is observed in each period only after production has occurred. After meeting current period's demand holding cost of \$1 per unit is incurred for any item that is carried over from one period to the next. Because of limited capacity the inventory at the end of each period cannot exceed 3 units. All demand should be met on time (no backorders). If at the end of the planning horizon (i.e. period 3) the firm still has units in stock, these can be salvaged at \$2 per unit. The initial inventory is 1 unit.*

The problem described in the previous example can be implemented in Python as shown in the `InventoryControl` class below. Additional classes used are shown in Listing 78 and Listing 79. The following code captures the instance described.

```
instance = {"T": 3, "K": 3, "v": 2, "h": 1, "s": 2, "pmf": [[(1, 0.5),(2, 0.5)] for
     i in range(0,3)], "C": 3}
ls = InventoryControl(**instance)
t = 0    # initial period
i = 1    # initial inventory level
print("f_1("+str(i)+"): " + str(ls.f(i)))
print("b_1("+str(i)+"): " + str(ls.q(t, i)))
```

```
import functools

class memoize(object):

   def __init__(self, func):
      self.func = func
      self.memoized = {}
      self.method_cache = {}

   def __call__(self, *args):
      return self.cache_get(
           self.memoized, args,
         lambda: self.func(*args))

   def __get__(self, obj, objtype):
      return self.cache_get(
           self.method_cache, obj,
         lambda: self.__class__(
              functools.partial(
              self.func, obj)))

   def cache_get(self, cache, key,
         func):
      try:
         return cache[key]
      except KeyError:
         cache[key] = func()
         return cache[key]

   def reset(self):
      self.memoized = {}
      self.method_cache = {}
```

Listing 78 The `Memoize` class; **memoization** is a technique for storing the results of expensive function calls and returning the cached result when the same inputs occur again.

```
class State:
   '''the state of the inventory system
   '''

   def __init__(self, t: int, I:
         float):
      '''state constructor

      Arguments:
         t {int} -- time period
         I {float} -- initial inventory
      '''
      self.t, self.I = t, I

   def __eq__(self, other):
      return self.__dict__ ==
            other.__dict__

   def __str__(self):
      return str(self.t) + " " +
            str(self.I)

   def __hash__(self):
      return hash(str(self))
```

Listing 79 `State` class.

```
class InventoryControl:
   '''the inventory control problem

   Returns:
      [type] -- [description]
   '''

   def __init__(self, T:int, K: float, v: float, h: float, s: float, pmf:
        List[List[Tuple[int, float]]], C: float):
      '''inventory control problem constructor

      Arguments:
         T {int} -- periods in planning horizon
         K {float} -- fixed ordering cost
         v {float} -- per item ordering cost
         h {float} -- per item holding cost
         s {float} -- per item salvage value
         pmf {List[List[Tuple[int, float]]]} -- probability mass function
         C {float} -- capacity of the warehouse
      '''

      self.max_demand = max([max(i, key=lambda x: x[0])[0] for i in pmf])
      self.min_demand = min([min(i, key=lambda x: x[0])[0] for i in pmf])
      self.max_order_qty = C + self.min_demand

      # initialize instance variables
      self.T, self.K, self.v, self.h, self.s, self.pmf, self.warehouseCapacity = T,
           K, v, h, s, pmf, C

      # lambdas
      self.ag = lambda s: [i for i in range(max(self.max_demand - s.I, 0),
                                 min(self.max_order_qty - s.I, self.max_order_qty)
                                      + 1)] # action generator
      self.st = lambda s, a, d: State(s.t+1, s.I+a-d) # state transition
      L = lambda i,a,d : self.h*max(i+a-d, 0)         # immediate holding cost
      S = lambda i,a,d : self.s*max(i+a-d, 0)         # immediate salvage value
      self.iv = lambda s, a, d: ((self.K + self.v*a if a > 0 else 0) +
                L(s.I, a, d) -
                (S(s.I, a, d) if s.t == T - 1 else 0)) # immediate value function

      self.cache_actions = {}                         # cache with optimal
           state/action pairs

   def f(self, level: List[float]) -> float:
      s = State(0,level)
      return self._f(s)

   def q(self, period: int, level: List[float]) -> float:
      s = State(period,level)
      return self.cache_actions[str(s)]

   @memoize
   def _f(self, s: State) -> float:
      #Forward recursion
      v = min(
         [sum([p[1]*(self.iv(s, a, p[0])+                  # immediate cost
                 (self._f(self.st(s, a, p[0])) if s.t < self.T - 1 else 0)) #
                      future cost
              for p in self.pmf[s.t]])                     # demand realisations
          for a in self.ag(s)])                            # actions

      opt_a = lambda a: sum([p[1]*(self.iv(s, a, p[0])+
                          (self._f(self.st(s, a, p[0])) if s.t < self.T - 1 else 0))
                       for p in self.pmf[s.t]]) == v
      q = [k for k in filter(opt_a, self.ag(s))]           # retrieve best action
           list
      self.cache_actions[str(s)]=q[0] if bool(q) else None  # store an action in
           dictionary

      return v                                             # return expected total
           cost
```

To implement Stochastic Dynamic Programming in Python we make extensive use of `lambda` expressions (also known as Anonymous functions) to define: the function that generates the set of feasible actions for a given state, the state transition function that determines the future state given a present state and an action, and the immediate value function for an action taken in a given state.

Markov decision processes[71] represent a special class of stochastic dynamic programs in which the underlying stochastic process is a stationary process that features the Markov property.

[71] Martin L. Puterman. *Markov decision processes: Discrete stochastic dynamic programming*. J. Wiley & Sons, 1994.