```python
### Cloning_randomtimes_constant-pop.py

import random, math, pylab, heapq
                # heapq provides the Heap Queue structure which allows to manipulate efficiently sorted list.
                # Here we use a list of copies of the system, sorted according to their next time of evolution.
c=.25           # c = creation rate 0->1 ; 1-c = annihilation rate 1->0
s=-0.30          # s conjugated to the activity K
t=0.            # initial and maximal time
popsize=400     # population size
populat=[]
                # initial state of the population :
                # each member of (or "copy" in) the population is described by a 4-uple
                # (time,dt,state,K)
                #   time  = next time at which it will evolve
                #   dt = time since last evolution
                #   state = 0 or 1 = empty or occupied
                #
tmax=2000.      # maximal time
tmin=tmax/2     # measures start at tmin
step=0          # counter for the number of steps in the "mutation/selection" process


# s-modified escape rate
def sescape(n):
    return math.exp(-s)*(n+c)

# s-dependent cloning rate
def cloningrate(n):
    return (math.exp(-s)-1.)*(n+c)

# procedure to remove a element of index i from a heap, keeping the heap structure intact
def heapq_remove(heap, index):
    # Move slot to be removed to top of heap
    while index > 0:
        up = (index + 1) / 2 - 1
        heap[index] = heap[up]
        index = up
    # Remove top of heap and restore heap property
    heapq.heappop(heap)

# lists to sample the logarithm of the cloning ratios Y a function of time
samplestime,samplesY,samplesYint,samplesK,samplescs=[],[],[],[],[]
Y,Yint=0.,0.

# initialization of the population
populat=[ (0.,0.,random.randint(0,1),0) for count in range(popsize)]
heapq.heapify(populat)  # orders the population into a Heap Queue


while t<tmax:
    # we pop the first element of populat, which is always the next to evolve
    (t,dt,state,K)=heapq.heappop(populat)
    # the copy we poped out is to be replaced by p copies ; random.random() is uniform on [0,1[
    cloningfactor= math.exp(dt*cloningrate(state))
    p=int( cloningfactor + random.random() )

    if p==0:    # one copy chosen at random replaces the current copy
        toclone=random.choice(populat)
        heapq.heappush(populat,toclone)
    elif p==1:  # the current copy is evolved without cloning
        newstate=state+1
        if random.random() < (state*1.)/(state*1.+c):
            newstate=state-1
        Deltat=random.expovariate(sescape(newstate)) # interval until next evolution
        toclone=(t+Deltat,Deltat,newstate,K+1)
        heapq.heappush(populat,toclone)
    else: # p>1 : make p clones ; population size becomes N+p-1 ; remove p-1 clones uniformly
        pcount=p
        while pcount>0:
            pcount-=1
            newstate=state+1
            if random.random() < (state*1.)/(state*1.+c):
                newstate=state-1
```

```
            Deltat=random.expovariate(sescape(newstate)) # interval until next evolution
            toclone=(t+Deltat,Deltat,newstate,K+1)
            heapq.heappush(populat,toclone)
        # we first chose uniformly the p-1 distinct indices to remove, among the N+p-1 indices
        listsize=popsize+p-1;indices=random.sample(xrange(listsize),p-1)
        # the list of indices to remove is sorted from largest to smallest, so as to remove largest
        indices first
        indices.sort(reverse=True)
        for i in indices:
            heapq_remove(populat,i)

    if t>tmin:
        Yint+=math.log((popsize+p-1.)/(1.*popsize))
        Y   +=math.log((popsize+cloningfactor-1.)/(1.*popsize))
        if step%5 == 0:
            samplestime.append(t)
            samplesY.append(Y)
            samplesYint.append(Yint)
            samplesK.append(sum([K/t for (t0,Deltat0,state0,K) in populat]))
            samplescs.append(sum([state0 for (t0,Deltat0,state0,K) in populat]))
        step+=1

# Bulk numerical estimation of psiK(s) from population size ~ e^(t psiK(s) )
psiK=Y/(t-tmin)
psiKint=Yint/(t-tmin)
listK=[K for (t,Deltat,newstate,K) in populat]
liststate=[newstate for (t,Deltat,newstate,K) in populat]

# better estimation by fitting log(popsize(t)) starting from a given threshold so as to isolate the
large-time
# exponential behaviour popsize(t) ~ e^(t psiK(s) )

psiKintfit,const = pylab.polyfit(samplestime,samplesYint,1)
psiKfit,const    = pylab.polyfit(samplestime,samplesY,   1)

print 'final total population size = ', len(populat)
print '         theoretical psi(s) = ', c*(math.exp(-2.*s)-1.)
print '      bulk numerical psi(s) = ', psiK
print '(int) bulk numerical psi(s) = ', psiKint
print 'fitted numerical fit psi(s) = ', psiKfit
print '(int) ---------- fit psi(s) = ', psiKintfit
print ' '
print 'theoretical mean activity(s)= ', 2.*c*(math.exp(-2.*s))
print ' numerical mean activity(s) = ', sum(listK)/tmax/len(populat)
print '       running average K(s) = ', sum(samplesK)/len(populat)/len(samplesK)
print ' '
print '       theoretical mean c(s)= ', c*(math.exp(-s))
print ' numerical mean activity(s) = ', (sum(liststate)*1.)/len(populat)
print '       running average c(s) = ', (sum(samplescs)*1.)/len(populat)/len(samplescs)

pylab.plot(samplestime,samplesY, 'r')
pylab.plot(samplestime,samplesYint, 'b')
pylab.plot(samplestime,[const+psiKfit*samplestime[i] for i in range(len(samplestime)) ], 'g-')
pylab.show()
```