CS 131: Project 1 Simulating a Network

Matthew McLaughlin

April 24, 2018

Overview of the document

First I give an overview of the code.
After I explain some important calculations in the program.
I then give an analysis of the results as well as the plots
Finally, the last pages show code for three files: hw1.py, helper.py, and system.py.
I have endeavored to make the code as readable by putting comments before every

hw1.py

This is the main file of the program. The code in this file is responsible for conducting the simulations, gathering the results of the simulations, and plotting those results. A simulation is on a fixed size network. Thus, the main function is responsible for calling simulation with various sized Network.

helper.py

This file contains an assortment of functions useful to the hw1.py. It has functions that assist in plotting, randomly assigning a unit to a memory slot, and determining when to end the simulation. In addition, constants are defined and standard library imports occur here.

system.py

This file defines the Unit,UnitList,Module,and Network classes. A Network object is what the simulation function creates and controls during its deliberation. A Network object is composed of N units (proccesssors) and a memory module with M slots (memory cells). These values stay fixed until the simulation for that particular Network object ends.

Unit

A Unit maintains two pieces of information, its *wait_time* and its *module_index*. The wait_time for a unit tracks how many cycles the unit has been waiting to connect to a particular slot. Once it is connected to said slot, this value is reset to 0. The only purpose of this attribute is to determine a unit's priority in requesting a slot.
The module_index is set to the index of the slot the unit is waiting to connect to. If it is not waiting to connect to any slot this value is None

UnitList

The UnitList is simple data structure that stores a collection of Units and is able to order the units according to their priorit. Units with a high wait_time will appear earlier in the UnitList

Module

A Module object is a data structure used to reprsent the memory in a Network. A Module is composed of individual slots and it is initiailzed with M of them. Each slot can either be free or occupied depending on wether or not a unit has connected to that particular slot during a given cycle. At the end of every cycle, all of the Modules slots are freed. Inside the Module class are methods and attributes that perform all the tasks afformentioned.

Network

A Network object is for in essence a container and book-keeper. It is a container because it is responsible for both initializing and storing a UnitList and Module object. Thus, in order to initialize a Network you must pass it N and M.

The Network is a book-keeper because it maintains information about the simulation. It tracks the total number of requests that have been made by units on the network as well as the *total_wait_time*. ANY time one of the units request is denied, the Network object notes this, and increments total_wait_time. Note that this attribute is completly independent from the unit attribute *wait_time* which is used soley for determining its priority.

Finally, a Network object is also capable of calculating the average access time up to a particular point in the simulation. This value is simply ...

$$\text{average access time} = \frac{total\_wait}{total\_requests}$$

This avg in effect tell us, out of all the requests that have been made up to this point in the simulation, what portion of these requests were units waiting?

Calculations

In this section I explain my methods for randomly generating requests using a uniform and gaussian distribution. The way this is done in my program is not immediatly clear so I include the following code snippet from helper.py

```
def choose_random_function(M: int, choice: str):
        if choice == 'G':
                # Use normal distribution to
                # generate memory requests
                # guass(mu, sigma) returns random number
                # with mean mu and standar dev sigma
                # note that the mean is randomly chosen
                arg1 = randint(0, M-1)
                arg2 = ceil(M / C)
                return gauss, arg1, arg2
        if choice == 'U':
                # Use a uniform distribution
                # randint(a, b) returns random int in range
                # [a, b]
                arg1 = 0
                arg2 = M-1
                return randint, arg1, arg2
        else:
```

I wanted to write the simulate function once, however I needed to be able to do two different types of simulations; those with a uniform and those with a gaussian distribution. This led me to write the choose_random_function which takes a choice string as a parameter as well as the number of slots in the given Network.
If the choice is Gaussian then I prepare the arguments to the guass function. First I choose uniformaly at random a mean that will be used used throughout the simulation. The mean is a random value between 0 and M. To determine the standar deviation I set $\sigma = \lceil \frac{M}{C} \rceil$ where $C = 5$ so that the standard deviation scales with the number of units. After the arguments are prepared I return the guass function (defined in random.py) as well as the two arguments.

If the choise is a unfiform distribution I can simply use the standard randint(a,b) function which returns a pseudorandom number n where $a \leq n \leq b$. Hence, I return this function as well as the arguments with $a = 0$ and $b = M - 1$ (since Module uses 0 based indexing).

Why am I returning random functions with arguments

If you look at this snippet from the main function followed by the signature of the simulate function....

```
        for num in UNIT_NUM:
                uni_avgs = [num]
```

```
3                   gau_avgs = [num]
4                   for i in range(1,MODULES + 1):
5                           uni_avgs.append(simulate(num, i, 'U'))
6                           gau_avgs.append(simulate(num, i, 'G'))
```

```
def simulate(N,M, choice:str):
```

it is hopefully more clear what is going on. We run a series of simulations varying the number of units and slots.Inside the innermost forloop we make two calls to the simulate function, one specifying that we want to simulate using a universal distribution (line 5) and the other with a gaussian (line 6). These values are then added to lists storing all the averages of the simulations which is later used to plot.

Transform

One problem for our simulation when using a gaussian distribution is that the guass function can return negative number, numbers outside the boundary of the Module, and real numbers, or some combination of the three. The boundary issues is due to the fact that the mean can be sometimes be left or right centered making it more likely for outliers to be generated outside of the boundaries of the Module.
To address this issue I created the transform function.

```
def transform(num:float,M:int):
        #x = abs(floor(num))
        x = floor(num)
        if x >= M or x < 0: return x % M
        return x
```

The idea behind this function was to treat Module as a circular array. That is to say, if a number is generated outside the boundaries of the Module, that just means that the index should wrap around. For example, if a number was generated greater than M than this index should be transformed so that is is now one of the Modules lower order indices.

Analysis of Results

First, observe below the plots of the average access time. You will note that I have supplied four figures total. This is because the graphs with modules varying from 1 to 2048 are almost unreadable. Because of this, I have provided addition graphs of the same simulation, just limited to values between 0 and 100 as this is the range in which all the interesting stuff happens.

Please note that the full range graphs are labeld from 0 to 2000 however the results of the simulation were as speciefied from 1 to 2048. I just styled my graph this way so that the tick marks didn't look ugly.

Observations

More modules means faster access time

In both graphs we see the trend that as the number of slots increase, the average access time decreases. This is expected behavior as more slots available decreases the probability of a collision (two units requesting the same slot). Also as expected, when the number of units is low, the access time approaches 0 more quickly than when the number of units is high. Take for example the blue line (units= 2). Even when the memory of modules is only two it is unlikely that there will be a collosion. Assuming we each unit chooses independently and uniformily at random, the probability of a collison is $\frac{1}{20}$!.

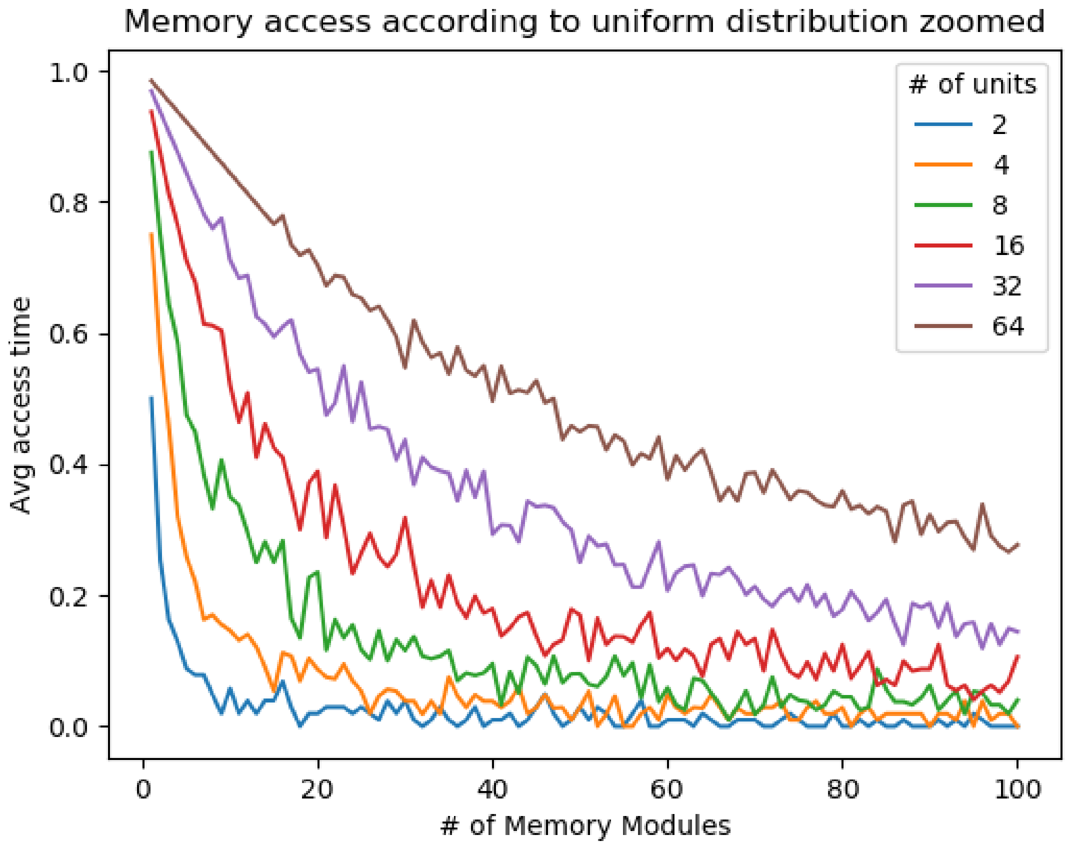
Random choice according to a Uniform distribution seems to converge to 0 more quickly and with less variance than choice according to Gaussian distribution

If you look at the two graphs, you will note that they are not perfect slopes. That is because in some cases we get unlucky and even though we increased the number of slots the access time worsens then from some other simulation when we used less slots. This phenomenum is evident in both the Gaussian and Uniform plots. However, you will notice that these so called hiccups vary in intensity. Suppose we used a modules during a simulation and got access time y_a. Later, suppose another simulation using $a + c$ modules (where c is some constant) with access time y_{a+c}, is such that $y_c > y_a$. The severity of the hiccup, H, is how large $H = y_{a+c} - y_a$ is. You can see from the graphs that all the H in the guassian plot tend to be larger then the H in the uniform plot. To see this, look at the various tick marks on the zoomed graphs, it is pretty much always the case that the gaussian plots have much larger hiccups at these points then the uniform plots

A possible explanation for larger hiccups in Gaussian plots

The problem with choosing slots according to a gaussian distribution is that all the requests will tend to be centered around the mean. This means (not a joke) there are likely to be more collisions. This raises the question of why then does the guassian curve eventually approach 0 as the number of modules increases? Let's examine when $M = 2048$ and $N = 64$. In this case the standard deviation is approximately 409 using the calculation above. Assuming in the worse case that all requests always fall within the range of 409 of the mean, that is still

Figure 1: Zoomed figures of gaussian and uniform



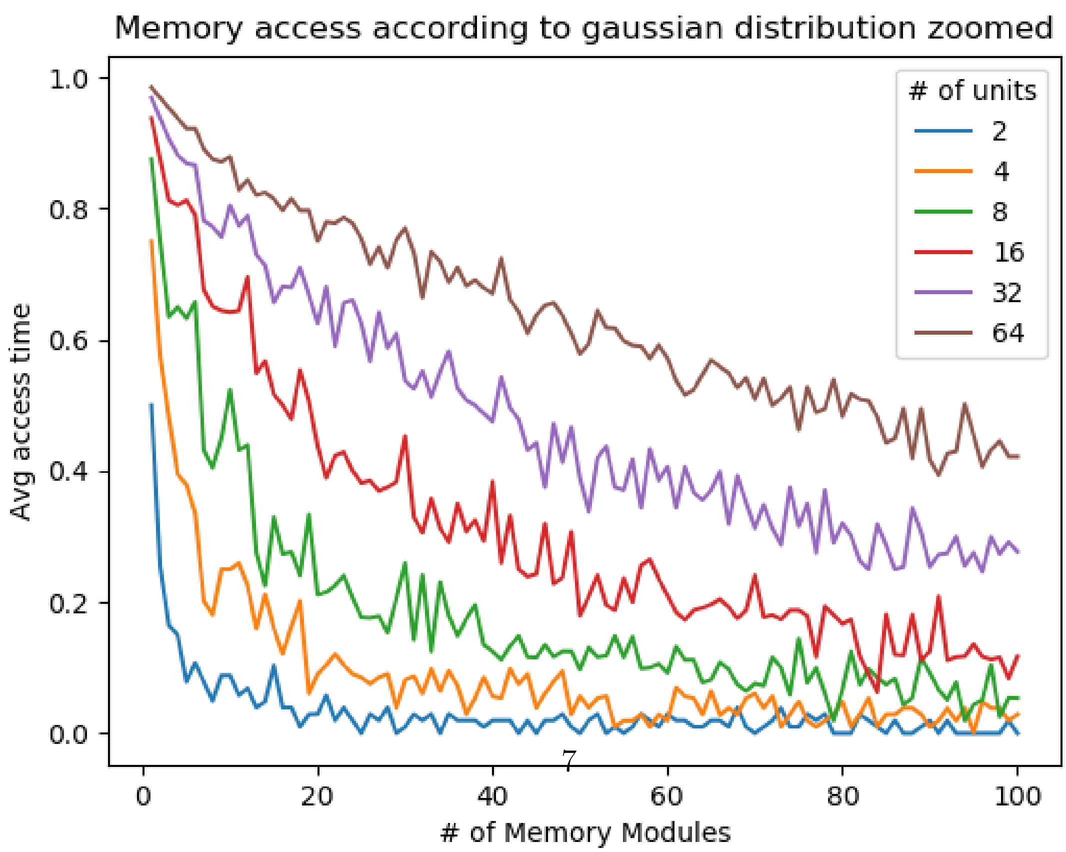
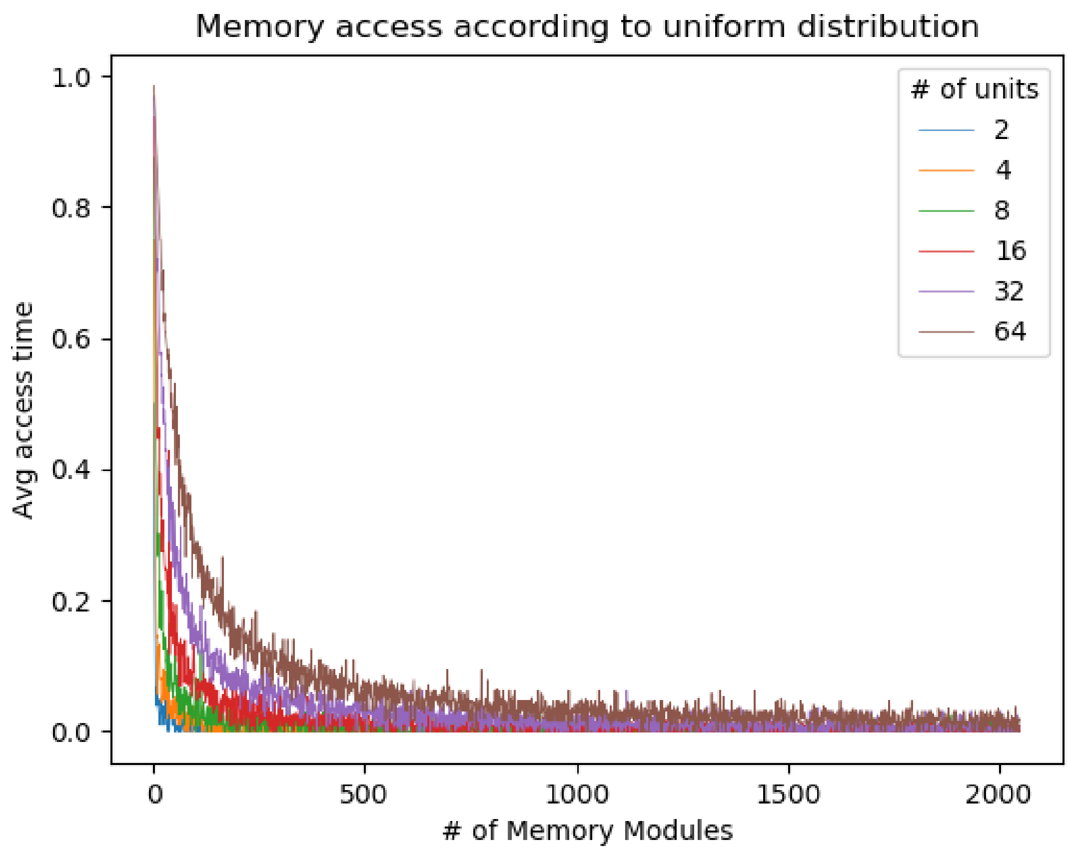
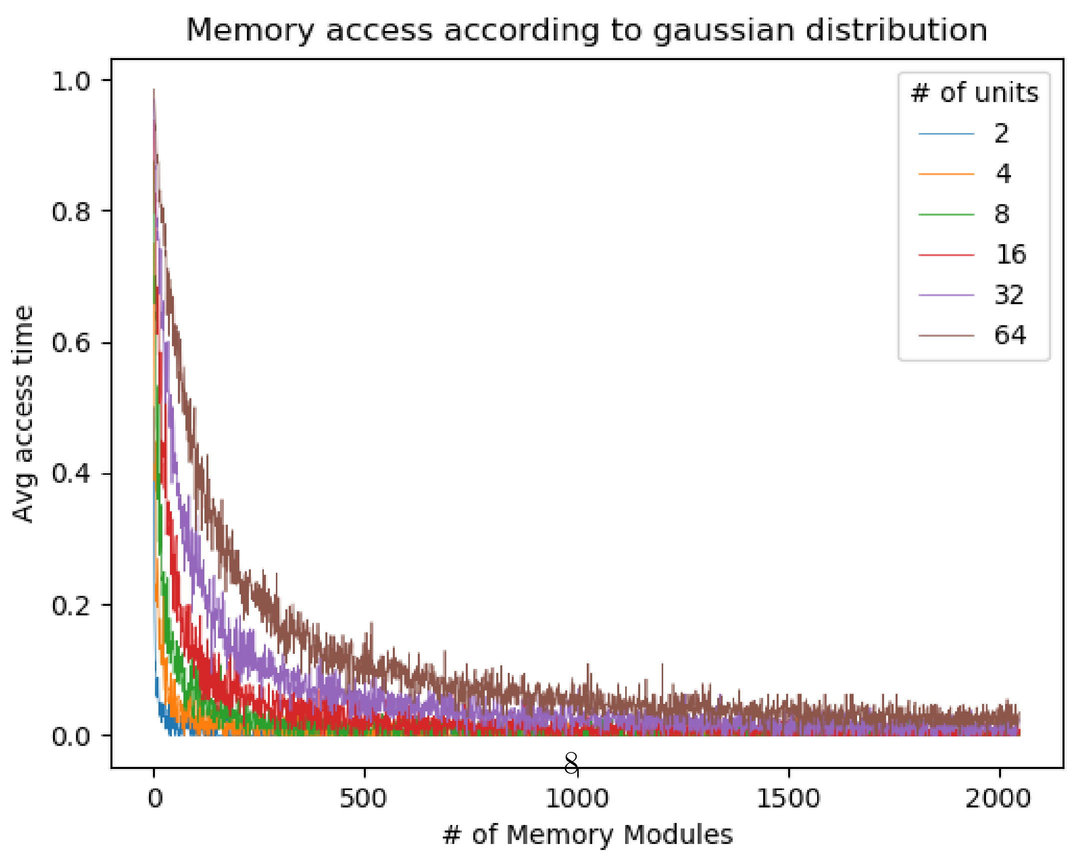


Figure 2: Full range of gaussian and uniform





a considerable amount of breathing room for the units. Using a variation of the birtday paradox, we could determine exactly how likely it would be for any of the units to share the same (birthday or slot).

```
# Author: Matthew McLaughlin
# email:  matthedm@uci.edu
# id:     34026707

##############################################
#                          hw1.py
# this file contains the main function
# as well as the simulate function
##############################################
'''
NOTES TO THE GRADER
*I will provide information about
functions, classes, constants, etc
in comments above the actual implementation
above the implementation of said objects.
I prefer this format for documenting my code.

*If this code doesn't make sense I reccomend
reading through System.py
'''
# User defined Modules
# these statements import
# everything both libraries
from helper import *
from system import *

'''
N: Number of units to use
M: Number of slots to use
Returns the avg access time for
a network with N units and M slots

'choose_random_function' will store in f
a function that generates random numbers
according to either a guassian or uniform
distribution depending on the value of
the string 'choice'. I did this so that
I didnt need to write the simulate function
twice.

transform ensures that the randomly chosen
index is within the indices of the
module list
'''
```

```
def simulate(N,M, choice:str):
        f, arg1, arg2 = choose_random_function(M, choice)
        Net = Network(N,M)
        old_avg = float('-inf')
        cur_avg = float('inf')
        for counter in range(MAX_CYCLES):
                # Generate requests for each unit in network
                for unit in Net.UnitList:
                        # CASE1: Unit was waiting
                        if unit.is_waiting():
                                i = unit.module_index
                                # if memory slot is free
                                if Net.Module.is_free(i):
                                        unit.reset_wait_time()
                                        unit.free_unit()
                                        Net.Module.occupy(i)
                                else: # memory slot wasnt free
                                        unit.increment_wait_time()
                                        Net.increment_total_wait()
                        # CASE2: Unit was not waiting
                        else:
                                # randomly map a unit to a slot
                                i = transform(f(arg1,arg2), M)
                                if Net.Module.is_free(i):
                                        Net.Module.occupy(i)
                                else:
                                        unit.bind_unit(i)
                                        unit.increment_wait_time()
                                        Net.increment_total_wait()
                #END OF CYCLE UPDATES
                Net.update_total_requests()
                Net.UnitList.update_priority()
                Net.Module.free_slots()
                ## Decide if it is time to end
                ## the simulation
                old_avg = cur_avg
                cur_avg = Net.average_access_time()
                if end_sim(Net.total_requests,old_avg,cur_avg):
                        return cur_avg
        #### Program only reaches here, if maxcycles reached
        return cur_avg

'''
Runs the simulation, gathers the averages,
then plots them. Note that two simulations
```

```
are performed, for varying sizes
of proccessors and memory modules.

Two simulations are performed. To map
units to slots, two different distributions
are used
'U' uniform distribution
'G' guassian distribution
'''
def main():
        uni = list()
        gau     = list()
        # run simulations with networks
        # varying the number of units
        # and the number of modules
        for num in UNIT_NUM:
                uni_avgs = [num]
                gau_avgs = [num]
                for i in range(1,MODULES + 1):
                        uni_avgs.append(simulate(num, i, 'U'))
                        gau_avgs.append(simulate(num, i, 'G'))
                uni.append(uni_avgs)
                gau.append(gau_avgs)
        # Plot the results of the simulations
        x_axis = [i for i in range(1,MODULES+1)]
        plot(uni,x_axis,'Uniform Distribution')
        plot(gau,x_axis, 'Gauss Distribution')

main()
```

```
##################################################
#                           helper.py
#  This file contains functions useful
# to the main function and the
# simulate method in hw1.py
#
# It defines the constants
# used in this file,
# hw1.py and System.py
#
# It imports useul library function
# used through the files afformentioned
##################################################

# Standard library imports
import matplotlib.pyplot as plt
import matplotlib.pyplot as plt
from math import floor, ceil
from random import randint, gauss
from numpy import arange

'''
Constant Definitions
UNIT_NUM: A list that tells for each simulation
                    how many units a network should have

MAX_CYCLES: cycles is the most number of cycles
              to do before aborting the simulation

MIN_CYCLES: min number of cycles before
                            ending simulation

STOPVAL: is the constant that tells simulation
                  when to stop the simulation. It does so
                  when the previous average and current
                  average differ by <= STOPVAL

C:      is a value to help calculate the
            standard deviation when
            generating random module requests
            according to a normal distrib
'''
UNIT_NUM = [2,4,8,16,32,64]
MODULES = 2048
```

```
MAX_CYCLES = 10**8
MIN_CYCLES = 100
STOP_VAL = .002
C  = 6

'''
M: number of slots
choice: a string representing
                    representing the userts choice
                    of which random function to sue

The purpose of this function is to
choose the random function that will be
used during a simulation based on the
users choice. The purpose of this function
is to moduralize so that
two different simulations for the
different distributions dont need to be
written. this function returns
a) function to be used
b) its arguments
'''
def choose_random_function(M:int, choice:str):
        if choice == 'G':
                # Use normal distribution to
                # generate memory requests
                # guass(mu,sigma) returns random number
                # with mean mu and standar dev sigma
                # note that the mean is randomly chosen
                arg1 = randint(0, M-1)
                arg2 = ceil(M / C)
                return gauss,arg1,arg2
        if choice == 'U':
                # Use a uniform distribution
                # randint(a,b) returns random int in range
                # [a,b]
                arg1 = 0
                arg2 = M-1
                return randint,arg1,arg2
        else:
                raise ValueError

'''
num: a randomly generated number
           from gauss or independent distrib
```

```

M: is the number of slots

This function transforms
a randomly generated number so that
the following conditions hold
a) num is positive
b) num is an integer
c) 0 <= num < M

Note that if num was generated
from a uniform distribution this
the transformatino has no effect
'''
def transform(num:float,M:int):
        #x = abs(floor(num))
        x = floor(num)
        if x >= M or x < 0: return x % M
        return x

'''
Returns true if it is time
to end the simulation

r:    is number of requests
old   the old average
cur   the current averag
'''
def end_sim(r:int, old:float, cur:float):
        return (r > MIN_CYCLES) and \
                   (abs(cur - old) <= STOP_VAL)

'''
Super imposes the plots of the
averages for each each network
and generates a png file.

avg_list: a list of averages. avg_list[i]
                 is the average access time
                 of the ith simulation

x_axis: is a list of values from 1..M
               where M=2048. x_axis is same
               regardles of how many units used
               thus we treat it like a constant
```

```

title: is the title of the plot
'''
def plot(avg_list, x_axis, title:str):
        for L in avg_list:
                plt.plot(x_axis,L[1:])
        plt.title(title)
        plt.xlabel("# of Memory Modules")
        plt.ylabel("Avg access time")
        plt.legend(title="# of units")
        plt.savefig(title + '.png')
        plt.show()
```

```
##############################################
#                         system.py
# file contains class defs for
# Unit:  reprsentation of proccessor
#
# UnitList: a collection of Units
#
# Module:  reprsentation of memory
#              in a network
#
# Network: composed of a UnitList
#                  and a Module. Tracks
#                  information about
#                  the current state of
#                  the network
##############################################

'''
A unit is nickname for a proccessor.
I just didnt want to type proccessor
all the time because it is long
and easy to misspell.

wait_time:   the accumulated number of cycles a unit
                        has spent waiting to access a
                        particular module. This value
                        is only relevant for determining
                        the priority scheme and NOT
                        for caclulating the average.

module:       If a Unit waited for a particular memory,
                        module, module is set to the index
                        of that memory module. Otherwise
                        the value is set to None indicating
                        that the unit can be assigned to
                        a new module on the next cycle
'''
class Unit():
        def __init__(self):
                self.wait_time = 0
                self.module_index = None
        '''
        Called whenever a units
        request was denied by a module
```

```
    '''
    def increment_wait_time(self):
        self.wait_time += 1

    '''
    reset_wait_time and
    free_unit are called whenever
    a unit that was waiting connects
    to the module it was waiting for
    '''
    def reset_wait_time(self):
        self.wait_time = 0

    def free_unit(self):
        self.module_index = None

    '''
    Binds the unit to the module at index i
    This only should happen if unit
    was denied its request to module i
    '''
    def bind_unit(self,i):
        self.module_index = i

    '''
    returns true if the unit is waiting
    '''
    def is_waiting(self):
        return self.module_index != None

'''
A UnitList is a data structure that
stores a collection of units. It
reprsents the proccessors in a Network
The primary job of a  UnitList is to order
itself according to the units with
the lowest wait time

unit_list: a list storing N units
unit_count: number of units in network
'''
class UnitList():
    def __init__(self,N:int):
        self.unit_list = [Unit() for i in range(N)]
        self.unit_count = N
```

```

    '''
    this method makes it possible to iterate
    directly overy a UnitList
    '''
    def __iter__(self):
        return iter(self.unit_list)

    '''
    Reorders the WaitList so that modules with higher
    total wait time appear at the front of the
    wait List. After each cycle this method
    is called
    '''
    def update_priority(self):
        f = lambda unit : unit.wait_time
        self.unit_list.sort(key=f, reverse=True)

'''
A module object is a data structure
used to represent the memory of a Network.
A module is composed of what i call 'slots',
or individual memory cells.

module_count: number of modules in network
'''
class Module():
    def __init__(self,M):
        self.Module = [None for i in range(M)]
        self.slot_count = M

    '''
    returns true if the slot at index i is
    currently available
    '''
    def is_free(self,i:int):
        return self.Module[i] == None

    '''
    Slot i becomes occupied by a unit
    Called whenever a unit generates
    a request to a free module
    '''
    def occupy(self,i):
        self.Module[i] = 1
```

```

        '''
        resets the module so that each slot is
        unnocupied. This method is called
        at the end of each cycle
        '''
        def free_slots(self):
                for i in range(self.slot_count):
                        self.Module[i] = None

'''
A network is composed of
a) modules
b) units
c) information about state of network

The primary role of a network object is
to do book keeping and manage its
memory and units. It delegates
the tasks of tracking which
slots in the module are free
and ordering of units
to the Module & UnitList
objects resp.

It tracks the number of cycles (requests)
made in addition to computing the average
access time.

Initialization
N: is the number of units in the netowrk
M: is the number of memory modules available in the network

Attributes
UnitList:       A list of all the units in the
                        network ordered first by
                        units with high wait time
                        and secondly by lowered
                        indexed units

Module: the representation of memory in the network

total_wait:     at a given point during the simulation
                        'total_wait' reflects the total
                        amount of time spent waiting by
```

```
                            each unit up to the current cycle.
                            Note that this is not the same as
                            unit.wait_time which reflects the
                            amount of time a unit has spent trying
                            to access a particular module.
                            Put simply, any time a unit waits,
                            this value is incremented

total_requests: is the number of hits/misses
                generated by units Since each of the
                k units generates a request every cycle
                in my simulation, this number is k*C
                where C > 0 and denotes number of cycles
'''
class Network():
        def __init__(self,N:int, M:int):
                self.UnitList = UnitList(N)
                self.Module = Module(M)
                self.total_wait = 0
                self.total_requests = 0

        '''
        Any time a units request is denied
        this method is called
        '''
        def increment_total_wait(self):
                self.total_wait += 1

        '''
        This is only updated at the end of a cycle
        since at the end of a cycle, each of the N
        units will have generated a request
        and so we can simply increase the total
        number of requests by N
        '''
        def update_total_requests(self):
                self.total_requests += self.UnitList.unit_count

        '''
        At the end of each cycle this function
        gives the average access time. Note here
        That self.total_wait time already encapsulates
        what TA refers to as 'total still wait time'
        since total_wait time is incremented
        any time a unit waits.
```

```
        '''
    def average_access_time(self):
        return self.total_wait / float(self.total_requests)
```