12-746 Fundamental Python Prototyping for Infrastructure Systems

Project Progress Report:

Monte-Carlo Generator for Bridge Traffic Configurations Data

Name: Jingxiao Liu **Andrew ID**: jingxial

Date: Oct 2016 **Email**: jingxial@andrew.cmu.edu

1. Brief Description

Recently, the Weigh-in-Motion (WIM) systems measure vehicle weights accurately for many months or even years of real traffic. Statistical analyses of the WIM data provides an efficient way to predict future load effects. However, the cost of time and money for collecting and measuring limits the amount of traffic data. In order to extend the quantity of traffic data, artificial traffic data is generated depending on analyses of traffic characteristics by using Monte-Carlo simulation. Then, the generated traffic data is used for estimate extreme load effects during the infrastructure lifetime. In this project, the parameters calculated from multimodal distribution (Tri-modal Lognormal, Normal, and Weibull distribution) model are used to generate bridge traffic configurations data by Python.

2. Main Progress

During these days, I have finished the main Object-Oriented Classes and functions development. There are four classes in my program, which are *Vehicle*, *Pattern*, *Date*, and *Time*. This program can model gross vehicle mass, axle weight, axle spacings

1.1. *Vehicle* Class

Class *Vehicle* is the most important class, and it contains the *Pattern*, *Date* and *Time* Classes. Following is the core codes of this class.

```
from Generator import *
from IO.ParameterReader import *

class Vehicle:

    def __init__(self, d, t, h, s, dire, la, tl, pat):
        self.date = d
        self.time = t
        self.head = h # default head 1001
        self.speed = s # default speed 10
        self.patt = pat
```

```
        self.axlen = pat.axlen
        self.pattern = pat.patstr
        self.direction = dire # default direction 1
        self.lane = la # default lane 1
        self.tlit = tl #default TLiT 0
        self.gvm = genGVM(pat.patternPara)
        self.axlew = genAW(pat.axlewPara, pat, self.gvm)
        self.axles = genAS(pat.axlesPara, pat)
        self.length = self.length(self.axles)

    def length(self, axles):
        len = 0
        for i in axles:
            len += i
        return len
```

Explanations on properties of *Vehicle* are as follows:

- ***Time & Date (Vehicle.time & Vehicle.date)***

The Time and Date class were developed to indicate arrival time of Vehicles on the bridge. The properties of Time are hour, minute and second, the properties of Date are year, month and date. Also The Time and Date class are properties of physical vehicle in the Vehicle class. In addition, operators, equal (=); minus (-); compare (sort), are overloaded for calculating Date efficiently.

- ***GVM (Vehicle.gvm)***

Gross Vehicle Mass is the first parameter that is considered for generating artificial vehicles. One function was established to output samples using Monte Carlo method. It can generate tri-modal mixed distribution (Lognormal, Normal and Weibull distribution) samples efficiently.

Function:

lnwGenerator: This function is used to generate samples with tri-modal distribution. A distribution with three different modes is called tri-modal distribution. The tri-modal distribution has three piles of data and its *PDF* is defined as:

$$f(x) = p_1 f_1(parameters) + p_2 f_2(parameters) + (1 - p_1 - p_2) f_3(parameters).$$

where the parameters p_1, p_2 and $(1 - p_1 - p_2)$ are the proportions of the distribution, and $f_1(parameters)$, $f_2(parameters)$ and $f_3(parameters)$ are the *PDF* with different parameters. This tri-modal distribution contains Log-Normal, Normal, and Weibull distributions. Log-Normal and Weibull distributions are defined as follows:

Log-normal distribution function

Log-normal distribution of a random variable *X*, whose logarithm $(\ln(X))$ is normally distributed, its *PDF* and *CDF* are defined as:

$$f_X(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln x - \mu^2)}{2\sigma^2}},$$

$$\text{and } F_X(x) = \frac{1}{2} + \frac{1}{2}\text{erf}\left[\frac{\ln x - \mu}{\sqrt{2}\sigma}\right].$$

where erf is the error function defined as:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

Weibull distribution function

The Weibull distribution is a widely used distribution in reliability. For modelling time to fail and material strength, Weibull distribution has good results and universal use. Its *PDF* and *CDF* are defined as:

$$f_X(x) = \begin{cases} \frac{k}{\lambda}\left(\frac{x-\delta}{\lambda}\right) e^{-[(x-\delta)/\lambda)^k]} & x \geq 0 \\ 0 & x < 0 \end{cases},$$

$$\text{and } F_X(x) = \begin{cases} 1 - e^{-[(x-\delta)/\lambda]^k} & x \geq 0 \\ 0 & x < 0 \end{cases}.$$

- ***Pattern (Vehicle.patt)***

Pattern Class will be explained later.

- ***Length (Vehicle.length)***

Length of vehicle is equal to wheelbase, which is the sum of axle spacings. So program does not need function for generating length, but directly accumulates axle spacings of each vehicle.

- ***Number of Axles (Vehicle.axlen)***

This parameter is generated in *Pattern* Class. For example, if the pattern of vehicle is '123', the number of axles should be 6.

- ***Number of Axle Groups (Vehicle.axlegn)***

This parameter is generated in *Pattern* Class. For example, if the pattern of vehicle is '123', the number of axle groups should be 3.

- ***Axle Weights (Vehicle.axlew)***

Axle Weights property is a 20 length list each vehicle. For each pattern class vehicles, axle group weights percentages are generated firstly using LNW distributions and copula function. In order to ensure that the sum of axle group weight percentages is equal to the GVW, they are scaled pro-rata. The individual axle loads are calculated by equally dividing the axle group weight.

- ***Axle Spacings (Vehicle.axles)***

In the *Vehicle* class, Axle Spacings are generated as a list, which contains 19 values the same as BeDIT (APPENDIX A) file. Initially, the intra axle group spacings are directly defined as 1.3 meters. For the axle spacings between each axle group are generated from distributions fit by tri-modal Mixed (Lognormal, Normal, Weibull distribution) distribution.

1.2. *Pattern* Class

Pattern Class's codes are shown below:

```
from IO.ParameterReader import *

class Pattern:

    def __init__(self, pts):
        self.patstr = pts
        self.axlen = 0
        self.genan()
        self.axlegn = 0
        self.genagn()
        self.patternPara = parameterReader(pts)
        self.axlewPara = self.axlewParaReader(self.axlegn, self.patternPara)
        self.axlesPara = self.axlesParaReader(self.axlegn, self.patternPara)
        self.proportion = self.patternPara[2][3+self.axlegn]

# Generate Number of Axle Groups
    def genagn(self):
        self.axlegn = len(self.patstr)

# Generate Number of Axles
    def genan(self):
        self.axlen = 0
        for i in range(0, len(self.patstr)):
            self.axlen += int(self.patstr[i])

# Read Axle Weight Parameters
    def axlewParaReader(self, axlegn, patternPara):
        para = []
        for i in range(0, axlegn):
            fname = patternPara[0][3 + i]
            para.append(parameterReader(fname))
```

```
        return para

#Read Axle Spacing Parameters
    def axlesParaReader(self, axlegn, patternPara):
        para = []
        for i in range(0, axlegn-1):
            fname = patternPara[2][3 + i]
            para.append(parameterReader(fname))
        return para
```

There are seven properties in *Pattern* Class:

- ***Pattern string (Pattern.patstr)***

This property is used as the "fingerprint" in order to define physical configurations of vehicles. Different pattern string indicates different distributions of GVM, axle group weight percentage and axle spacings.

- ***Proportion (Pattern.proportion)***

Proportion of certain pattern vehicles will be read from *.csv files and assigned to this property.

- ***Parameters of pattern, axle group weight, and axle spacings (Pattern.patternPara & Pattern.axlewPara & Pattern.axlesPara)***

The *ParameterReader* (1.3) function will be used to read these parameters.

1.3. Input / Output Functions

Input Traffic Files

Three input files define GVW, axle group weight percentage, axle spacing, number of certain type vehicles and pattern name. The values are separated by commas and stored in *csv files.

Pattern file stores the GVW distribution parameters, AGWP model names, correlation parameters, axle spacing model names, pattern name and the proportion for each pattern. The AGWP and axle spacing model names indicate particular AGWP and AS files that contain the AGWP and AS distribution parameters; the copula parameters are stored for generating copula functions; and the proportion input defines the percentage of these pattern vehicles in whole. Explanations of these three input files for a four axle group vehicle pattern viewed in tabular form are:

Pattern.csv

p_1	μ_L	σ_L	AGWP model number	AGWP model number	AGWP model number	AGWP model number
p_2	μ_N	σ_N	Copula parameter	Copula parameter	Copula parameter	Pattern
c	a	b	Axle spacing model number	Axle spacing model number	Axle spacing model number	Proportion

AGWP.csv

p_1	μ_L	σ_L
p_2	μ_N	σ_N
c	a	b

AGWS.csv

p_1	μ_L	σ_L
p_2	μ_N	σ_N
c	a	b

Function *ParameterReader* is used to read '*.csv' files and save them into list *Str*. Its codes are shown as follows:

```
import csv

def parameterReader(pattern):
    Str = []
    fname = str(pattern)+'.csv'
    with open(fname, 'rb') as f:
        reader = csv.reader(f, delimiter=',', quotechar='|')
        for row in reader:
            Str.append(row)
    return Str
```

Output Functions

Function *WriteBeDIT* (shown under this paragraph) is developed to output vehicle configurations. And the output format is BeDIT, shown in Appendix A.

writeBeDIT function:

```
def writeBeDIT(vehicle):
    f = open('simulation.txt', 'w')
    count = 1
    for i in vehicle:
        f.write('{:>4}'.format(str(i.head)))
        f.write('{:>2}'.format(str(i.date.date)))
        f.write('{:>2}'.format(str(i.date.month)))
        f.write('{:>2}'.format(str(i.date.year)))
        f.write('{:>2}'.format(str(i.time.hour)))
```

```
        f.write('{:>2}'.format(str(i.time.minute)))
        f.write('{:>2}'.format(str(i.time.second)))
        f.write('{:>2}'.format(str(int(i.time.second/60*100))))
        f.write('{:>3}'.format(str(int(round(i.speed)))))
        f.write('{:>4}'.format(str(int(round(i.gvm*10)))))
        f.write('{:>3}'.format(str(int(round(i.length*10)))))
        f.write('{:>2}'.format(str(int(round(i.axlen)))))
        f.write('{:>1}'.format(str(int(round(i.direction)))))
        f.write('{:>1}'.format(str(int(round(i.lane)))))
        f.write('{:>3}'.format(str(int(round(i.tlit)))))
        for j in range(0,19):
            f.write('{:>3}'.format(str(int(round(i.axlew[j]*10)))))
            f.write('{:>3}'.format(str(int(round(i.axles[j]*10)))))
        f.write('{:>3}'.format(str(int(round(i.axlew[19]*10)))))
        if count != len(vehicle):
            f.write('\n')
            count += 1
    f.close()
```

The output data will be tested separately using MATLAB.

1.4. Usage

This program does not have an interface right now. Functions will be called in the main script called *GenVeh*. Codes is presented below:

```
from traffic.Vehicle import Vehicle
from traffic.Pattern import Pattern
from arrivalTime.Time import Time
from arrivalTime.Date import Date
from IO.WriteBeDIT import writeBeDIT
import random
import time

print "How many vehicles do you want to generate?"
number = int(raw_input('-->'))
print "What type of Vehicles does the network have? (Enter vehicle patterns divided
by comma)"
pattern = raw_input('-->')
start = time.clock()
pattern = pattern.split(",")

v = []
for i in pattern:
    pattern = Pattern(i)
    for j in range(0, int(number*float(pattern.proportion))):
        v.append(Vehicle(Date(15, 6, 1), Time(1, 1, 1), 1001, 10, 1, 1, 0,
pattern))

random.shuffle(v)
writeBeDIT(v)

end = time.clock()
print "Run time: %f s" %(end - start)
```

For using this program, firstly, users should write parameter files in the project's folder. Two vehicle patterns, '123' and '1211', were used as an example.

After that, when users run the program it will ask they to type in the number of vehicles that will be simulated:

```
C:\Python27\python.exe C:/Users/jingx/Dropbox/Work&Study/CMU/12746/TermProject/GenVeh/GenVeh.py
How many vehicles do you want to generate?
-->
```

Further, users should define the composition of traffic:

```
C:\Python27\python.exe C:/Users/jingx/Dropbox/Work&Study/CMU/12746/TermProject/GenVeh/GenVeh.py
How many vehicles do you want to generate?
-->100000
What types of Vehicles do the network have? (Enter vehicle patterns divided by comma)
-->
```

After typing 123, 1211, program will write traffic into 'simulation.txt' and calculate running time:

```
C:\Python27\python.exe C:/Users/jingx/Dropbox/Work&Study/CMU/12746/TermProject/GenVeh/GenVeh.py
How many vehicles do you want to generate?
-->100000
What types of Vehicles do the network have? (Enter vehicle patterns divided by comma)
-->123,1211
Run time: 16.681525 s
```

It can be seen that *GenVeh* can generate 100,000 vehicles in 16.7 seconds. Screenshot of part of the output file is shown as follows:

```
1001 1 615 1 1 1 0 10 223152 611  0 57 37 46 13 46 77 24 13 24 13 24  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 174163 511  0 33 39 32 13 32 43 43 68 35  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 222128 611  0 39 49 46 13 46 40 30 13 30 13 30  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 179142 611  0 22 27 33 13 33 75 30 13 30 13 30  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 202145 611  0 32 34 45 13 45 72 27 13 27 13 27  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 359144 611  0 96 40 67 13 67 65 43 13 43 13 43  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 285129 611  0 43 30 56 13 56 60 44 13 44 13 44  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 225158 611  0 59 35 37 13 37 84 31 13 31 13 31  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 271136 611  0 78 32 50 13 50 66 31 13 31 13 31  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 172160 511  0 22 30 30 13 30 49 33 68 56  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 293143 611  0 56 33 53 13 53 71 44 13 44 13 44  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 253135 511  0 62 41 53 13 53 56 34 25 49  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 325140 611  0 55 29 59 13 59 72 51 13 51 13 51  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 184149 611  0 20 34 41 13 41 76 27 13 27 13 27  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 317139 511  0 38 45 62 13 62 56 79 25 77  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 196153 611  0 44 30 32 13 32 84 29 13 29 13 29  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 212169 511  0 60 33 35 13 35 55 45 69 37  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 175170 511  0 24 43 29 13 29 47 50 67 43  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 204185 511  0 25 37 37 13 37 68 55 66 50  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1001 1 615 1 1 1 0 10 231109 611  0 55 32 41 13 41 39 31 13 31 13 31  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

This file has 100,000 lines in BeDIT format (APPENDIX A). As a result, my generator is efficient.

3. Future Work

The main part of my program has been done. In next weeks, I will revise the copula function library to apply copula correlations to axle group weights (Core code is shown under this paragraph.). As is known, correlation should be applied to total samples after generating, it cannot be calculated with marginal distribution sampling, so this process would increase generator's running time. Also, timing complexity will be increased, caused by reading samples from multi-dimension lists for several times. After that, a visualized interface will be developed

for better user experience. And I will compare simulation results with parametric distributions in the interface. Finally, more pattern (37 types) input files will be set for real traffic simulation.

CopulaGenerator Function:

```
def CopulaGenerator(n, theta):
    if theta <= 1:
        raise ValueError('the parameter for GUMBEL copula should be greater than
1')
    if theta < 1 + sys.float_info.epsilon:
        U = np.random.uniform(size=n)
        V = np.random.uniform(size=n)
    else:
        u = np.random.uniform(size=n)
        w = np.random.uniform(size=n)
        w1 = np.random.uniform(size=n)
        w2 = np.random.uniform(size=n)

        u = (u - 0.5) * np.pi
        u2 = u + np.pi / 2
        e = -np.log(w)
        t = np.cos(u - u2 / theta) / e
        gamma = (np.sin(u2 / theta) / t) ** (1 / theta) * t / np.cos(u)
        s1 = (-np.log(w1)) ** (1 / theta) / gamma
        s2 = (-np.log(w2)) ** (1 / theta) / gamma
        U = np.array(np.exp(-s1))
        V = np.array(np.exp(-s2))
    return U,V
```

APPENDIX A

BeDIT File Format

In the table below, the Format column gives the storage type of the data. IX refers to an integer of X number of digits, including leading or trailing zeros.

Record	Unit	Format
Head		I4
Day		I2
Month		I2
Year		I2
Hour		I2
Minute		I2
Second		I2
Second/100		I2
Speed	dm/s	I3
Gross Vehicle Weight - GVW	kg/100	I4
Length	dm	I3
Number of Axles		I2
Direction (zero-based)		I1
Lane		I1
Transverse Location In Lane	dm	I3
Weight Axle 1	kg/100	I3
Spacing Axle 1 - Axle 2	dm	I3
Weight Axle 2	kg/100	I3
Spacing Axle 2 - Axle 3	dm	I3
⋮	⋮	⋮
Spacing Axle 19 - Axle 20	dm	I3
Weight Axle 20	kg/100	I3