12-746 Fundamental Python Prototyping for Infrastructure Systems

Project Final Report:

GenVeh
A Monte-Carlo Generator for Bridge Traffic Configurations Data

Name: Jingxiao Liu **Andrew ID**: jingxial
Date: Oct 2016 **Email**: jingxial@andrew.cmu.edu

1. Introduction

1.1. Problem Description

The transportation system is regarded as blood system in human's body. An efficient and safe transport infrastructure, especially bridges, that is responsible for moving goods and people. 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 provide 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. 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.

During last 8 months, I have almost finished the probability descriptions and modelling of traffic physical configurations, including gross vehicle weight (GVW), axle weights, axle spacings and number of axles. That model developed by the MATLAB uses multimodal distribution and Copula correlation functions (for more detail, please contact me for my thesis). In this project, I would like to use the parameters calculated from that mentioned model to generate bridge traffic configurations data by Python 2.7.

1.2. Objectives for Project

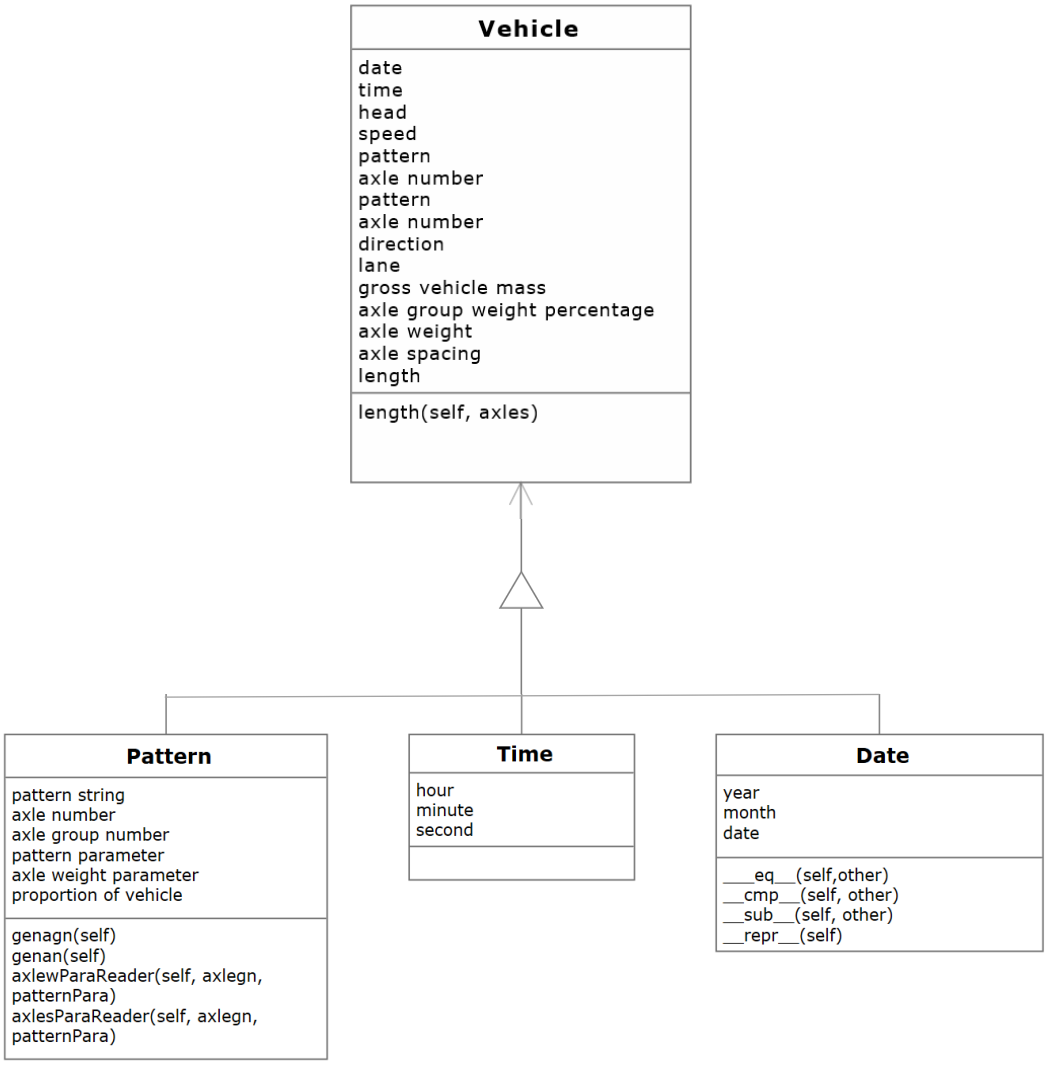
This Monte-Carlo generator (GenVeh) should:

- Be a packaged objected-oriented programming project;
- Have legible and reasonably structured inputs;
- Output artificial traffic efficiently.

This application can be used by bridge designers and researchers. It can do prediction of future (even 100 years) traffic composition by changing those distribution parameters. Users can apply these generated data to bridge finite-element models with scientific traffic flow model for monitoring during bridges' operation time.

1.3. Overview of *GenVeh*

This project used object-oriented (OO) method, it contains four classes: The *Vehicle* class, the *Pattern* class, the *Date* class and the *Time* class. As an example, properties of the physical vehicle (Gross Vehicle Mass (GVM), number of axles, axle spacings, axle weights etc.) are programmed into the *Vehicle* class. Also, *Time, Date* and *Pattern* are included in *Vehicle*, *Vehicle* can return its *Time*, *Date* and *Pattern* class of arrival on the bridge by calling functions. Input function will be used to read distribution and function parameters from *.csv files, and one output function will be developed for writing BeDIT file (Appendix A). Following is the class diagram of this project.



In addition, class *GenVehGUI* is developed to create components of graphical user interface, function *ResultPlot* is used for plotting result figures of gross vehicle weight and axle group weight percentages.

1.4. Organization of the rest of the report

In this report, the main classes *Vehicle* and *Pattern* will be explained firstly with their important properties and algorithms. Then, the structure and format of input and output files and their functions will be presented. Before assessment of this application, I will describe some of the graphical user interface and results plot function.

2. Detailed features developed

The properties and functions of those classes will be described as follows.

2.1 *Vehicle* Class (Codes in APPENDIX B)

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

Explanations on properties of *Vehicle* are as follows:

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

The Time and Date classes 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* is defined as:

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

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* is defined as:

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

- ***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. For each pattern class vehicle, axle group weights percentages are generated firstly using LNW distributions and copula function. 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.

To apply copula correlations to axle group weights, I used some codes from the copula function library (copulalib). 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. For axle group weight percentage, the Gumbel copula function (Codes in APPENDIX B) is used, and it is defined as:

Families and properties	Functions and ranges
Gumbel (1960)	
$\phi_\alpha(v)$	$-\ln\frac{\exp(-\alpha v)-1}{\exp(-\alpha)-1}$
Range for α	$(-\infty,0)\bigcup(0,+\infty)$
$C^{Arch}(u,v)$	$-\frac{1}{\alpha}\ln\left(1+\frac{(\exp(-\alpha u)-1)\exp(-\alpha v)-1)}{\exp(-\alpha)-1}\right)$

- ***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.

2.2 *Pattern* Class

Pattern Class's codes are shown in APPENDIX B:

There are seven properties in *Pattern* Class:

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

This property is used as the "fingerprint" 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* function (APPENDIX B) will be used to read these parameters.

2.3 Input / Output Files and 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 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 in APPENDIX B:

Output Functions

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

writeBeDIT function:

2.4 Graphical User Interface

Class *GenVehGUI* is developed to create the graphical user interface's components. It contains 'run' button, 'result' list, 'patterns' checkbox, 'file directory' button and labels, shown as follows:

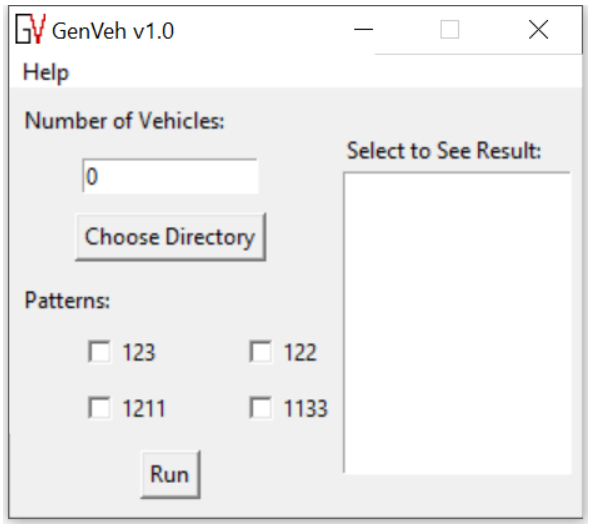


Figure 1: Graphical User Interface of GenVeh

After selecting and running the application, a result list will appear on the right-hand list box. By selecting the result list box, result figures for gross vehicle weight and axle group weight percentage will be shown in other windows. They (Figure 2 and Figure 3) are plotted by using the 'matplotlib' library.

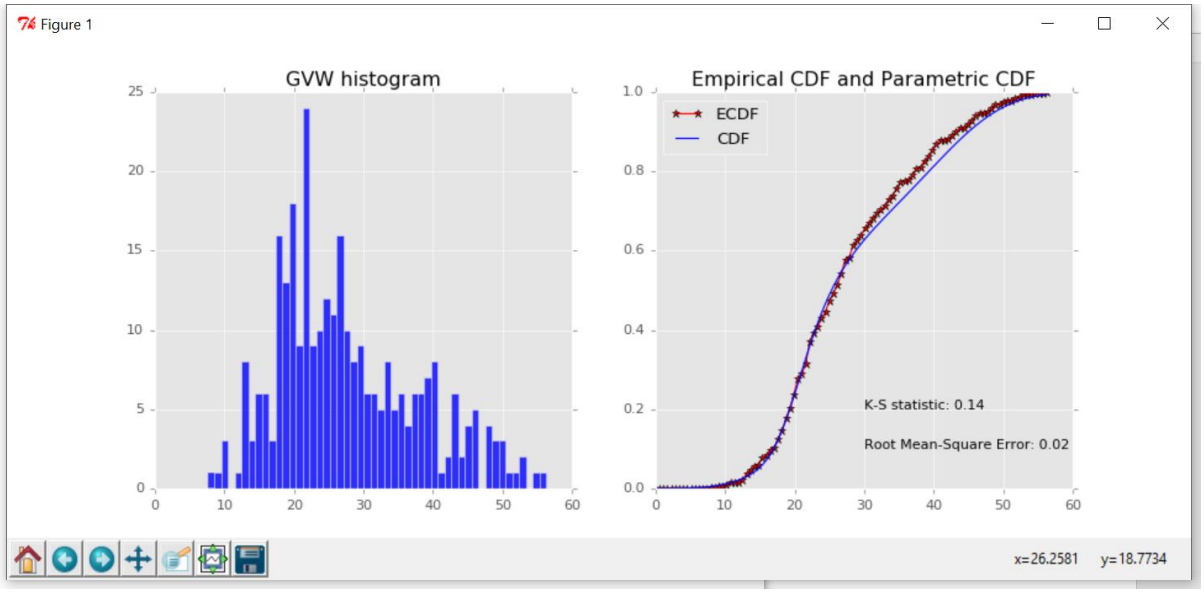


Figure 2: Gross Vehicle Weight result graph.

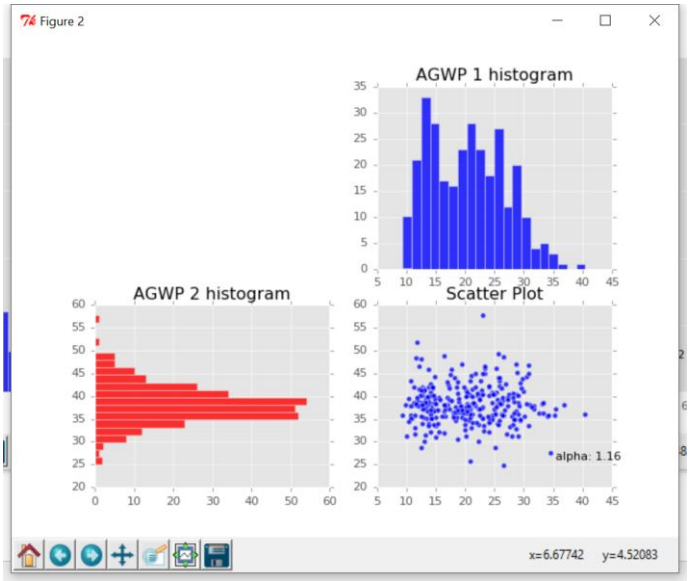


Figure 3: Axle Group Weight Percentage result graph.

3. Application Assessment

All Functions will be called in the script *GenVeh*. (Codes is presented in APPENDIX B.) This script can achieve application's main functions, such as generating vehicle samples and outputting them into *csv files. The main function *GenVehApp* imports the GUI class and creates interface for users.

Firstly, users should write parameter files in the project's folder. Four vehicle patterns, '123' '122', '1133' and '1211', were used as example.

After that, when user runs the program they should entry number of vehicles (for example 100,000) in the network, select simulation result file's directory, and vehicle's patterns included in the network. By pressing button 'Run', *GenVeh* can generate 100,000 vehicles in 20 seconds. Screenshot of part of the output file is shown as follows:

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

To verify the result, The K-S statistic and root mean-square error between empirical cumulative distribution function and parametric cumulative distribution function for gross vehicle weight are calculated. The K-S statistics, root mean-square error for 100,000 '1133' pattern vehicles are 0.07, 0.00, respectively. Thus, my application is accurate.

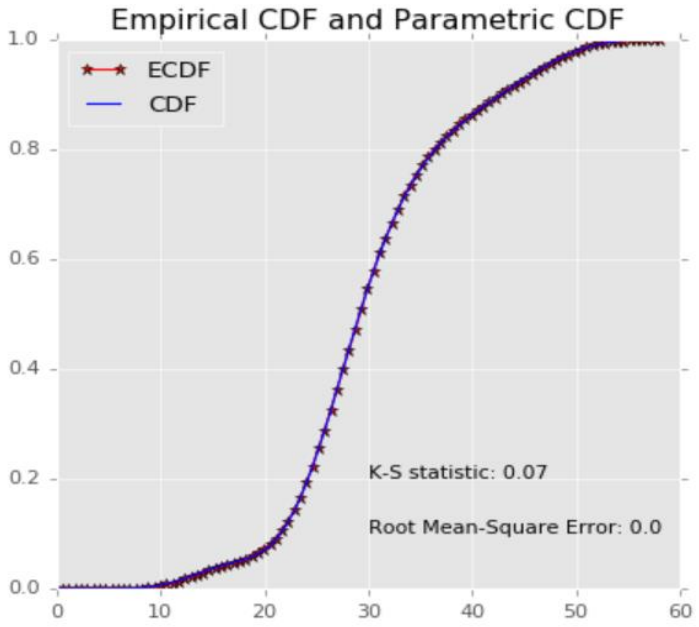


Figure 4: Verification for gross vehicle weight.

4. Future Work

There are several works in the future. Actually, there are more than 30 types of vehicles in the network, so I should define all the input files later. Furthermore, scientific traffic flow model will be developed later and be applied to this application for generating arrival time of vehicles. (They are default time and cannot be generated right now.) If possible, the finite element model can be added to this application for directly using it to calculate load effects and do monitoring of bridges.

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

APPENDIX B

Vehicle Class:

```
from Generator 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.axlegwp = genAGWP(pat)
        self.axlew = genAW(self.axlegwp, pat, self.gvm)
        self.axles = genAS(pat)
        self.length = self.length(self.axles)

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

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
```

Pattern Class

```
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][2+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
```

parameterReader Function

```
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
```

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()
```

GenVeh Script

```
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
from Generator import ApplyCopula
import os.path
import random
import time

def GenVeh(number, pattern, directory):

    fileName = os.path.join(directory, "simulation result.txt")
    start = time.clock()

    v = []
    axlegwp = []
    gvm = []
    parameter = []
    #Generate by pattern
    for i in pattern:
        veh = []
        pattern = Pattern(i)
        parameter.append(pattern.patternPara)
        agwp = []
        g = []
        for j in range(0, int(number*float(pattern.proportion))):
            veh.append(Vehicle(Date(15, 6, 1), Time(1, 1, 1), 1001, 10, 1, 1, 0,
pattern))
        for i in range(0, veh[0].patt.axlegn):
            agwp.append([])
            for j in range(0, len(veh)):
                agwp[i].append(veh[j].axlegwp[i])
        for j in range(0, len(veh)):
            g.append(veh[j].gvm)
        # Apply Copula Correlation
        v += ApplyCopula(veh[len(veh)-
int(number*float(pattern.proportion)):len(veh)])

        axlegwp.append(agwp)
        gvm.append(g)
    random.shuffle(v)
    writeBeDIT(v, fileName)

    end = time.clock()
```

```
    # print "Run time: "+str(end - start)+" s"
    return axlegwp, gvm, parameter
```