Priyanka x20192037 Project CA

Modelling, Simulation and Optimisation (H9MSO)

The “Future Cities” Project in Trinity College Dublin investigates a “Self-Organising Motorway” in SimPy v.4

Requirements¶

Your task is to investigate for suitable points along the motorway segment

the traffic density k (the number of vehicles per km across all available lanes), and the mean speed of the vehicles u, and for the overall motorway segment the traffic volume q (i.e. the total number of vehicles passing per hour across all available lanes), and the average total travelling time. depending on certain input parameter: motorway speed limits settings (60km/h, 80km/h, 100km/h, and 120km/h), inter-arrival time distributions (directly influencing the traffic density), and traffic mix (percentage of heavy goods vehicles, electrical, autonomous cars) Report the highest traffic volume the motorway segment can accommodate and articulate your recommendations regarding the setting of speed limits on the motorway segment.

Importing all required libraries

In [1]:
```
#Importing all required libraries
import pandas as pd
import numpy as np
import math
import random
import statistics
import matplotlib as mpl
import matplotlib.pyplot as plt
import scipy as sp
import scipy.stats as stats
import statsmodels.api as sm
import seaborn as sns
import simpy
import simpy.events as evt
import random
from random import random
```

In [2]:
```python
# Utilities
def isZero(x, ε = 0.00001):
    return abs(x)<ε
```

Bell Curve Random Generator

In [3]:
```
gen_x_tab = None
gen_y_tab = None

def pdfv(x, mean=0, sd=None, delta=None):

    if sd is None and delta is None:
        sd = 1
    elif sd is None and delta is not None:
        sd = delta/math.sqrt(6)*0.9275

    τ = math.sqrt(6)
    x = (x-mean)/sd*0.9275
    return np.where(abs(x)<τ, 5/(8*τ*sd)*(1-x**2/2+x**4/24+1/2)*0.9275, 0)

def rv(mean=0, sd=None, delta=None, alternating=None, rounding=None):

    if sd is None and delta is None:
        sd = 1
    elif sd is None and delta is not None:
        sd = delta/math.sqrt(6)*0.9275

    global gen_count, gen_x_tab, gen_y_tab

    if gen_x_tab is None:

        # initialise global variables
        gen_x_tab = list(np.linspace(-math.sqrt(6), -1.51, 500)) + \
                    list(np.linspace(-1.5, 0, 151))

        τ = math.sqrt(6)

        def cdf(x):
            return 5/(8*τ)*(x**5/120-x**3/6+x*3/2)+5/(8*τ)*(τ**5/120-τ**3/6+τ*
3/2)

        gen_y_tab = [ cdf(x) for x in gen_x_tab ]

    y, sign = random.random(), 1
    if y>0.5:
        y, sign = 1-y, -1
    if alternating is not None:
        sign = 1 if alternating%2==0 else -1

    for i in range(1, len(gen_x_tab)):
        if y < gen_y_tab[i]:
            x0 = gen_x_tab[i]
            y0 = gen_y_tab[i]
            x1 = gen_x_tab[i-1]
            y1 = gen_y_tab[i-1]
            x = mean+sign*((x0+(y-y0)*(x1-x0)/(y1-y0))*sd/0.9275)
            if type(rounding) is int and rounding>0:
                return round(x, rounding)
            else:
                return x
```

Entities

In [4]:
```
def normaliseDirection(d):
    d = d.lower()
    if d=='r' or d=='fast':
        return 'fast'
    elif d=='l' or d=='slow':
        return 'slow'
    else:
        return None
```

Lanes Entity

In [5]:
```
LANE_ID = 0

class Lane:

    def __init__(self, rec, length, speedLimit=None, merge=None):

        global LANE_ID
        self.id = LANE_ID
        LANE_ID += 1

        self.rec = rec
        self.x0 = 0
        self.length = length

        # speed limit on this lane segment in km/h
        self.speedLimit = speedLimit

        # indicates that at the end of the lance traffic has to merge to the l
eft or right
        self.merge = merge

        self.vehicles = []

        self.next = None
        self.prev = None

        # lane attached to the left/right
        self.left = None
        self.right = None

        rec.recordLane(self)

    # defines generic str() method for Lanes
    def __str__(self):
        l = " L:--" if self.left is None else f" L:{self.left.id:0>2d}"
        r = " R:--" if self.right is None else f" R:{self.right.id:0>2d}"
        s = "" if self.speedLimit is None else f" speedLimit:{self.speedLimit:
d}km/h"
        return f"[{self.id:0>2d} x={self.x0:3.1f}m l={self.length:3.1f}m"+l+r+
s+"]" + \
                    ("\\" if self.merge=='R' else "") + \
                    ("/" if self.merge=='L' else "") + \
                    ("-" + str(self.next) if self.next is not None else "")

    def getLane(self, direction):
        if direction=='slow':
            return self.left
        elif direction=='fast':
            return self.right
        else:
            return None

    ## additional code
    # adding parallel lane on right side
    def attachRight(self, lane):
        self.right = lane
```

```
        lane.left = self
        lane.x0 = self.x0
        lane.speedLimit = self.speedLimit

    ## additional code
    # adding parallel lane on right side
    def attachLeft(self, lane):
        self.left = lane
        lane.right = self
        lane.x0 = self.x0
        lane.speedLimit = self.speedLimit

    ## additional code
    # constructs a number of lane segments of the same length
    # and attaches them to the right
    def widenRight(self):
        lane = self
        newLane = Lane(self.rec, lane.length)
        lane.attachRight(newLane)
        while lane.next is not None:
            lane = lane.next
            newLane = Lane(self.rec, lane.length)
            lane.attachRight(newLane)
            newLane.prev = lane.prev.right
            newLane.prev.next = newLane
        return self.right

    ## additional code
    # constructs a number of lane segments of the same length
    # and attaches them to the right
    def widenLeft(self):
        lane = self
        newLane = Lane(self.rec, lane.length)
        lane.attachLeft(newLane)
        while lane.next is not None:
            lane = lane.next
            newLane = Lane(self.rec, lane.length)
            lane.attachLeft(newLane)
            newLane.prev = lane.prev.left
            newLane.prev.next = newLane
        return self.left

    # defines concatenation of lanes
    def extend(self, lane):
        l = self
        while l.next is not None:
            l = l.next
        l.next = lane
        lane.x0 = l.x0+l.length
        lane.prev = l
        return self

    def totalLength(self):
        total = self.length
        l = self
        while l.next is not None:
            l = l.next
```

```
            total += l.length
        return total

    def between(self, posBack, posFront):
        # make sure that the position of all cars is accurate
        # at this point in time
        for v in self.vehicles:
            v.updateOnly()
                # normally the list should be sorted, but just in case
        self.vehicles.sort(key=lambda v: v.pos)
        res = []
        for v in self.vehicles:
            if posBack < v.pos and v.pos-v.length < posFront:
                res.append(v)
        # if the required distance reaches over the end of the lane segment
        if posFront > self.length and self.next is not None:
            return res + self.next.between(0, posFront-self.length)
        elif posBack < 0 and self.prev is not None:
            return self.prev.between(self.prev.length+posBack, self.prev.lengt
h) + res
        else:
            return res

    def inFront(self, pos, far):
        # make sure that the position of all cars is accurate
        # at this point in time
        for v in self.vehicles:
            v.updateOnly()
        # normally the list should be sorted, but just in case
        self.vehicles.sort(key=lambda v: v.pos)
        for v in self.vehicles:
            if v.pos > pos:
                return v if v.pos-pos<far else None
        # there is none in front in this lance
        # if the free lane in front is long enough or there is no next lane
        if self.length-pos>far or self.next is None:
            return None
        else:
            return self.next.inFront(0, far-(self.length-pos))

    def behind(self, pos, far):
        # make sure that the position of all cars is accurate
        # at this point in time
        for v in self.vehicles:
            v.updateOnly()
        # This time we sort in reverse order
        self.vehicles.sort(key=lambda v: v.pos, reverse=True)
        for v in self.vehicles:
            if v.pos < pos:
                return v if pos-v.pos<far else None
        # there is none behind in this lance
        # if the free lane in behind is long enough or there is no previous la
ne
        if pos>far or self.prev is None:
            return None
        else:
            return self.prev.behind(self.prev.length, far-pos)
```

```python
    def enter(self, vehicle, pos=0, laneChange=False):
        self.vehicles.insert(0, vehicle)
        vehicle.pos = pos
        vehicle.lane = self
        vehicle.rec.record(vehicle, event="enter lane"+(f" {self.id:d}" if laneChange else ""))

    def leave(self, vehicle, laneChange=False):
        vehicle.rec.record(vehicle, event="leave lane"+(f" {self.id:d}" if laneChange else ""))
        vehicle.lane = None
        self.vehicles.remove(vehicle)
```

Vehicle Entity

In [6]:

```python
def isRunning(p):
    return p is not None and p.running

def isCrashed(p):
    return p is not None and p.crashed
```

```
In [7]: VEHICLE_ID = 0
#vehicle_dict=dict of carname: (max_speed,max_acceleration,length and ideally
 width)


#VEHICLE_DICT={"Nissan Saloon":(240,4.67,5.12),"Renault Clio Hatchback":(161,
1.6,4.05),"Ford Fiesta Hatchback":(200,2.9,4.068),"Electric car":(160,2.3,3.8
9),"HGV":(140,1.8,15)}

class Vehicle:
    def __init__(self, rec,
                 startingLane=None, startingPos=0,
                 t0=0, x0=0, dx0=0, ddx0=0, dddx0=0,
                 vmax=None, vmax_var=None,a_max=None,
                 t=[], v=[],VEHICLE_MODEL="car"):

        global VEHICLE_ID
        self.id = VEHICLE_ID
        VEHICLE_ID += 1
        #self.vehicle=list(VEHICLE_DICT.keys())[VEHICLE_ID-1]

        self.VEHICLE_MODEL=VEHICLE_MODEL

        if self.VEHICLE_MODEL=="car":
            self.a_max=2.5
            self.a_brake = -4.0 # [m/s²]
            self.a_coast = -0.6 # [m/s²]
            self.length = 4

        elif self.VEHICLE_MODEL=="HGV":
            self.a_max=1.8
            self.a_brake = -2.5 # [m/s²]
            self.a_coast = -1.3 # [m/s²]
            self.length = 15

        elif self.VEHICLE_MODEL=="electric car":

            self.a_max= 4.6
            self.a_brake = -8.0 # [m/s²]
            self.a_coast = -1.8 # [m/s²]
            self.length = 4.6

        else:
            self.a_brake = -4.0 # [m/s²]   Tesla: -8.0 # [m/s²]
            self.a_coast = -0.6 # [m/s²]   Tesla: -1.8 # [m/s²]
            self.a_max= 2.5
            self.length = 4.6



        self.a_max =    a_max # [m/s²]   Tesla:  4.6 # [m/s²]
                            # Note: 2.5m/s² corresponds to 0-100km/h om 11s
      #  self.length = 4 # [m]  Length of the vehicle
        self.VEHICLE_LENGTH=4

        self.absolute_speed_time_tolerance = 2 # [s]
```

```
        self.relative_speed_time_tolerance = 5 # [s]
        self.far_away_in_front = 200 # [m]
        self.far_away_in_back = 80 # [m]
        self.min_action_time = 0.5 # [s]

        self.lane_change_time = 3 # [s]
        self.min_speed_diff = 2 # [m/s]

        ## if not None the preferred max free velocity
        self.vmax = vmax
        self.vmax_variation = vmax_var if vmax_var is not None else 0.05
        self.timing = 100 # [s]
        self.timing_variation = 0.30

        self.env = rec.env
        self.rec = rec

        self.startingLane = startingLane
        self.startingPos = startingPos
        self.lane = None
        self.pos = 0

        # speedLimit once recognised
        self.speedLimit = None

        ## second lane reference during changing of lanes
        self.oldLane = None

        self.alternator = 0

        self.t0 = t0
        self.x0 = x0    # [m] the reference point is on the front of the vehic
le
                        # the body of the vehicle of the given length is behin
d.
        self.dx0 = dx0
        self.ddx0 = ddx0
        self.dddx0 = dddx0

        self.t_target = t
        self.v_target = v
        # while there is an adjustment of velocity in progress
        # the current_target is set
        self.current_target = None

        # trace flags
        self.traceAdjustVelocity = False
        self.traceInterrupt = False
        self.traceBraking = False
        self.traceCrash = False
        self.traceSurround = False
        self.traceOvertaking = False

        # start process
        self.running = False
        self.crashed = False
```

```
        # Flags used for temporarily exclusive behaviour
        self.laneChangeControl = False
                # This flag is used to block automatic overtaking
                # while processing explicit lane change control
        self.braking = False
                # This flag is used to prevent interrupting braking for brakin
g
        self.changingLane = False
                # This flag is used to prevent interrupting lanechanging

        self.processRef = None
        self.env.process(self.process())

    def trace(self, message):
        print(f"t={self.t0:7,.1f}s x={self.x0:7,.1f}m v{self.id:d}", message)

    def isNotFasterThan(self, other):
        return True if other is None else self.dx0 <= other.dx0

    def isNotSlowerThan(self, other):
        return True if other is None else other.dx0 <= self.dx0

    ## adapted to handle lane information
    def updateOnly(self):
        t = self.env.now
        if t < self.t0 or not self.running:
            return False
        if t > self.t0:
            dt = t - self.t0
            ddx = self.ddx0 + self.dddx0*dt
            dx = self.dx0 + self.ddx0*dt + self.dddx0*dt*dt/2
            Δx = self.dx0*dt + self.ddx0*dt*dt/2 + self.dddx0*dt*dt*dt/6
            x = round(self.x0 + Δx, 2)
            self.t0, self.x0, self.dx0, self.ddx0 = t, x, dx, ddx

            ## begin additional code
            self.pos = round(self.pos+Δx, 2)
            # update lane information if necessary
            if self.pos >= self.lane.length:
                nextPos = self.pos - self.lane.length
                nextLane = self.lane.next
                self.lane.leave(self)
                if nextLane is None:
                    self.oldLane = None
                    self.rec.record(self, event='end')
                    self.running = False
                    return False
                else:
                    nextLane.enter(self, pos=nextPos)
            ## end additional code

        return True

    ## minor change only
    def update(self):

        self.surround = Surround(self)
```

```
        ## instead of direct link, call method
        inFront = self.surround.front

        # if there is a running or crashed vehicle in front and
        # we have crashed into the rear end of that vehicle
        if inFront is not None and inFront.x0-inFront.length < self.x0+self.VEHICLE_LENGTH:
            self.crash(inFront)

        # if the speedLimit changes...
        elif self.speedLimit != self.lane.speedLimit:
            self.speedLimit = self.lane.speedLimit
            if self.speedLimit is None:
                # end of speed limit, return to default behaviour
                if self.vmax is not None and self.vmax>self.dx0:
                    Δt = max(self.min_action_time, (self.vmax-self.dx0)/self.a_max)
                    self.setTarget(Δt, self.vmax)
            else:
                vmax = self.speedLimit/3.6
                vmax = rv(mean=vmax, delta=self.vmax_variation*vmax)
                if vmax<self.dx0:
                    # when the new speedLimit is lower than current speed, coast down
                    Δt = max(self.min_action_time, (self.dx0-vmax)/self.a_coast)
                    self.setTarget(Δt, vmax)
                else:
                    # when the new speedLimit is higher than current speed, accelerate at half max a_max
                    Δt = max(self.min_action_time, 2*(vmax-self.dx0)/self.a_max)
                    self.setTarget(Δt, vmax)

        # if at the end of the lane merging is required
        elif self.lane.merge is not None and \
                not self.braking and not self.laneChangeControl and not self.changingLane:
            if self.lane.merge == 'R':
                if self.surround.rightLane is not None and \
                        self.surround.right is None and \
                        self.isNotFasterThan(self.surround.rightFront) and \
                        self.isNotSlowerThan(self.surround.rightBack):
                    self.setTarget(self.lane_change_time, 'fast')
            elif self.lane.merge == 'L':
                if self.surround.leftLane is not None and \
                        self.surround.left is None and \
                        self.isNotFasterThan(self.surround.leftFront) and \
                        self.isNotSlowerThan(self.surround.leftBack):
                    self.setTarget(self.lane_change_time, 'slow')

        # start overtaking maneuver by changing into fast lane
        elif inFront is not None and \
                not self.braking and not self.laneChangeControl and not self.changingLane and \
                self.dx0 > inFront.dx0 + self.min_speed_diff and \
```

```
                self.x0 + (self.lane_change_time+self.absolute_speed_time_tole
rance)*self.dx0 > inFront.x0-inFront.length and \
                self.surround.rightLane is not None and \
                self.surround.right is None and \
                self.isNotFasterThan(self.surround.rightFront) and \
                self.isNotSlowerThan(self.surround.rightBack):
            self.setTarget(self.lane_change_time, 'fast')

        # if there is a vehicle in front
        # which drives slower and
        # the distance to that vehicle in front is at current speed
        # less than the critical time tolerance of this driver
        elif inFront is not None and \
                not self.braking and not self.laneChangeControl and not self.c
hangingLane and \
                self.dx0 > inFront.dx0 and \
                self.x0 + self.absolute_speed_time_tolerance*self.dx0 > inFron
t.x0:
            # note that self.a_coast is a negative acceleration
            Δt = max(self.min_action_time, (inFront.dx0-self.dx0)/self.a_coast
)
            if self.current_target is None or \
                    not isZero(self.current_target-inFront.dx0):
                self.setTarget(Δt, inFront.dx0)

        # end overtaking by returning to slow lane
        elif self.surround.leftLane is not None and \
                not self.braking and not self.laneChangeControl and not self.c
hangingLane and \
                self.surround.leftLane.totalLength()-self.pos>100 and \
                self.surround.left is None and \
                self.isNotSlowerThan(self.surround.leftBack) and \
                self.surround.leftFront is None:
            self.setTarget(self.lane_change_time, 'slow')

    ## code remains unchanged
    def setTarget(self, t, v):
        self.t_target = [ t ] + self.t_target
        self.v_target = [ v ] + self.v_target
        self.interruptProcess()

    ## minor change only
    def process(self):

        # delay start to the given time t-
        if self.t0>self.env.now:
            yield self.env.timeout(self.t0-self.env.now)
        self.t0 = self.env.now
        self.running = True
        self.rec.startRecording(self)

        ## begin additional code
        ## enter the first lane
        self.startingLane.enter(self)
        ## end additional code

        while self.updateOnly():
```

```
            ## instead of direct link, call method
            inFront = self.lane.inFront(self.pos, self.far_away_in_front) \
                        if self.lane is not None else None

            # if the car in front is slower and we are a bit too near on its h
eals...
            if inFront is not None and inFront.dx0 < self.dx0 and \
                    inFront.x0-inFront.length < \
                        self.x0 + self.relative_speed_time_tolerance*(self.dx0
-inFront.dx0):
                yield from self.emergencyBraking(inFront.dx0)
                if not isZero(self.dx0-inFront.dx0):
                    # after emergency breaking adjust to the speed of the car
 in front...
                    Δt = 1
                    self.setTarget(Δt, inFront.dx0)
                continue

            if len(self.t_target)>0:

                t_target = self.t_target[0]
                v_target = self.v_target[0]
                self.t_target = self.t_target[1:]
                self.v_target = self.v_target[1:]

                if type(v_target) is str:

                    # explicit overtake commands temporarily
                    # disable overtaking
                    if v_target=='R':
                        self.laneChangeControl = True
                    elif v_target=='L':
                        self.laneChangeControl = False

                    direction = normaliseDirection(v_target)
                    if self.lane.getLane(direction) is not None:
                        yield from self.changeLane(inFront, direction, t_targe
t)

                ## the rest is what was there before
                else:
                    v0 = self.dx0
                    v1 = v_target
                    Δt = t_target
                    if isZero(v1-v0):
                        yield from self.continueAtSameSpeed(Δt)
                    else:
                        yield from self.adjustVelocity(v1-v0, Δt)

            ## additional code
            elif self.vmax is not None:

                # if the explicit control strategy is exhausted and
                # a target maximum speed is set the vehicle will follow
                # a random speed around the target maximum speed
```

```python
                ## additional code
                vmax = self.vmax
                if self.speedLimit is not None:
                    vmax = self.speedLimit/3.6
                    vmax = rv(mean=vmax, delta=self.vmax_variation*vmax)

                # as long there is no vehicle in front or the vehicle in front
                # is far enough ahead
                if inFront is None or \
                    self.x0 + self.relative_speed_time_tolerance*(self.dx0-inF
ront.dx0) < inFront.x0:
                    v0 = self.dx0
                    self.alternator += 1
                    v1 = rv(mean=vmax,
                            delta=self.vmax_variation*vmax,
                            alternating=self.alternator,
                            rounding=2)
                    Δt = rv(mean=self.timing,
                            delta=self.timing*self.timing_variation,
                            rounding=1)

                    yield from self.adjustVelocity(v1-v0, Δt)
                else:
                    yield from self.continueAtSameSpeed(10)
            else:
                yield from self.continueAtSameSpeed(10)

        self.rec.stopRecording(self)

    def crash(self, other):

        def recordCrash(v):
            v.rec.record(v, 'crash')
            v.running = False
            v.crashed = True
            # stop anything your are currently doing...
            v.interruptProcess()
            v.dx0 = 0
            v.ddx0 = 0
            v.dddx0 = 0

        if self.running:
            if self.traceCrash:
                self.trace(f"Crashed into v{other.id:d} at x={self.x0:7.1f}m")
            recordCrash(self)
            if other.running:
                recordCrash(other)

    def emergencyBraking(self, v):

        def emergencyBrakingProcess(v):

            if self.traceBraking:
                self.trace(f"Braking from v={self.dx0:4.1f}m/s to {v:4.1f}m/s"
)

            self.rec.record(self, 'brake')
```

```
            minΔt = 0.2
            self.dddx0 = (self.a_brake-self.ddx0)/minΔt
            yield self.env.timeout(minΔt)

            self.updateOnly()
            self.dddx0=0
            self.ddx0=self.a_brake
            v = min(v, self.dx0-2)
                # the brake time estimate is for perfect timing for
                # autonomous cars. For manual driving leave out the
                # -minΔt/2 or use a random element.
            Δt = max(0.5, (v-self.dx0)/self.ddx0 - minΔt/2)
            yield self.env.timeout(Δt)

            self.updateOnly()
            self.dddx0 = -self.ddx0/minΔt
            yield self.env.timeout(minΔt)

            self.updateOnly()
            self.ddx0 = 0
            self.dddx0 = 0
            self.rec.record(self, 'brake end')

            if self.traceBraking:
                self.trace(f"Braking end  v={self.dx0:4.1f}m/s")

        ## The 'braking' bit prevents the interruption of an emergency breakin
g process
        self.braking = True
        self.processRef = self.env.process(emergencyBrakingProcess(v))
        try:
            yield self.processRef
        except simpy.Interrupt:
            if self.traceInterrupt or self.traceBraking:
                self.trace(f"Braking interrupted at v={self.dx0:4,.1f}m/s")
            pass
        self.processRef = None
        self.braking = False

    def changeLane(self, inFront, direction, Δt):

        def changeLaneProcess(oldLane, newlane, Δt, trace=True):
            # self.updateOnly()
            if trace and self.traceOvertaking and inFront is not None and  dir
ection=='fast':
                other = f"v{inFront.id:d} " if inFront is not None else ""
                self.trace(f"Overtaking {other:s} at x={self.x0:7,.1f}m")

            self.rec.record(self, 'change '+direction)
            self.oldLane = oldLane
            newLane.enter(self, pos=self.pos, laneChange=True)
            self.ddx0 = 1
            self.dddx0 = 0
            yield self.env.timeout(Δt)

            self.updateOnly()
            if not self.running:
```

```
                return
            currentLane = self.lane
            self.oldLane.leave(self, laneChange=True)
            self.lane = currentLane
            self.oldLane = None
            self.ddx0 = 0
            self.dddx0 = 0
            self.rec.record(self, 'done change '+direction)

            if trace and self.traceOvertaking and direction=='slow':
                self.trace(f"Overtaking done return to slow lane at x={self.x0:7,.1f}m")
            # self.updateOnly()

        ## keep record of current lane, as in case of aborting
        ## the lane change
        ## when interrupted go back into original lane
        self.changingLane = True
        self.updateOnly()
        oldLane = self.lane
        newLane = self.lane.getLane(direction)
        try:
            self.processRef = self.env.process(changeLaneProcess(oldLane, newLane, Δt))
            yield self.processRef
            self.processRef = None
        except simpy.Interrupt:
            if self.traceInterrupt and self.traceOvertaking:
                self.trace(f"Overtaking interrupted at x={self.x0:7,.1f}m")
            # if interrupted go quickly back into old lane
            # but this is not interruptible
            self.updateOnly()
            # it is possible that there was an intermittent change lane event
            while self.lane != newLane and newLane is not None:
                newLane = newLane.next
                oldLane = oldLane.next
            self.processRef = None
            self.env.process(changeLaneProcess(newLane, oldLane, Δt/4, trace=False))
        self.changingLane = False

    # smoothly adjust velocity by Δv over the time Δt
    def adjustVelocity(self, Δv, Δt):

        # smoothly adjust velocity by Δv over the time Δt
        def adjustVelocityProcess():

            self.updateOnly()

            if self.traceAdjustVelocity:
                self.trace(f"Adjusting Velocity from v={self.dx0:4,.1f}m/s by Δv={Δv:4,.1f}m/s over {Δt:4,.1f}s")

            minΔt = 0.1*Δt
            a = Δv/(Δt-minΔt)
            tt = Δt-2*minΔt
```

```
            self.dddx0 = (a-self.ddx0)/minΔt
            yield self.env.timeout(minΔt)

            self.updateOnly()
            self.dddx0 = 0
            self.ddx0 = a
            yield self.env.timeout(tt)

            self.updateOnly()
            self.dddx0 = -a/minΔt
            yield self.env.timeout(minΔt)

            self.updateOnly()
            self.dddx0 = 0
            self.ddx0 = 0

            if self.traceAdjustVelocity:
                self.trace(f"Adjusted  Velocity to   v={self.dx0:4,.1f}m/s")

        # start process
        self.current_target = self.dx0 + Δv
        self.processRef = self.env.process(adjustVelocityProcess())
        try:
            # wait for the process to finish
            yield self.processRef
        except simpy.Interrupt:
            # in case this has been interrupted
            if self.traceInterrupt:
                self.trace(f"Adjusting Velocity interrupted at v={self.dx0:4,.
1f}m/s")
            pass
        self.current_target = None
        self.processRef = None

    def continueAtSameSpeed(self, Δt):

        def continueAtSameSpeedProcess():
            # don't change the current velocity
            self.dddx0 = 0
            self.ddx0 = 0
            yield self.env.timeout(Δt)

        # start process
        self.processRef = self.env.process(continueAtSameSpeedProcess())
        try:
            # wait for the process to finish
            yield self.processRef
        except simpy.Interrupt:
            # in case this has been interrupted
            if self.traceInterrupt:
                self.trace(f"Constant Velocity interrupted at v={self.dx0:4,.1
f}m/s")
            pass
        self.processRef = None

    def interruptProcess(self):
        if self.processRef is not None and self.processRef.is_alive:
```

```
            self.processRef.interrupt('There are more important things to d
o...')
```

Property Surround

In [8]:

```
class Surround:

    def __init__(self, vehicle):

        # For each of the directions None means that there is no
        # vehicle in the immediate vicinity.
        # We initialise to a 'safe' value which can be easily detected
        # if something goes wrong

        self.vehicle = vehicle

        self.leftBack = None
        self.left = None
        self.leftFront = None
        self.back = None

        self.front = None
        self.rightBack = None
        self.right = None
        self.rightFront = None

        lane = vehicle.lane
        posFront = vehicle.pos
        posBack = vehicle.pos-vehicle.length
        if lane is not None:
            self.lane = lane
            self.front = lane.inFront(posFront, self.vehicle.far_away_in_front
)
            self.back = lane.behind(posBack, self.vehicle.far_away_in_back)

            self.rightLane = lane.right
            if self.rightLane is not None:
                if vehicle.oldLane == lane.right:
                    # drifting left
                    self.right = vehicle
                    self.rightFront = self.rightLane.inFront(posFront, self.ve
hicle.far_away_in_front)
                    self.rightBack = self.rightLane.behind(posBack, self.vehic
le.far_away_in_back)
                else:
                    right = self.rightLane.between(posBack-10, posFront+10)
                    if len(right)==0:
                        # self.right = None
                        self.rightFront = self.rightLane.inFront(posFront+10,
self.vehicle.far_away_in_front)
                        self.rightBack = self.rightLane.behind(posBack-10, sel
f.vehicle.far_away_in_back)
                    else:
                        self.right = right[0]
                        # self.rightFront = None
                        # self.rightBack = None

            self.leftLane = lane.left
            if self.leftLane is not None:
                if vehicle.oldLane == lane.left:
                    # drifting right
```

```
                    self.left = vehicle
                    self.leftFront = self.leftLane.inFront(posFront, self.vehi
cle.far_away_in_front)
                    self.leftBack = self.leftLane.behind(posBack, self.vehicle
.far_away_in_back)
                else:
                    left = self.leftLane.between(posBack-10, posFront+10)
                    if len(left)==0:
                        # self.left = None
                        self.leftFront = self.leftLane.inFront(posFront+10, se
lf.vehicle.far_away_in_front)
                        self.leftBack = self.leftLane.behind(posBack-10, self.
vehicle.far_away_in_back)
                    else:
                        self.left = left[0]
                        # self.leftFront = None
                        # self.lefttBack = None

        if vehicle.traceSurround:

            def s(vehicle):
                if vehicle is None:
                    return " "
                elif type(vehicle) is list:
                    if len(vehicle)==1:
                        return s(vehicle[0])
                    else:
                        res = "["
                        for v in vehicle:
                            if len(res)>1:
                                res += ','
                            res+=s(v)
                        res += "]"
                        return res
                else:
                    return f"{vehicle.id:d}"

            print(f"surround t={self.vehicle.env.now:6.2f} " +
                  "|" +
                  ("" if self.leftLane is None else
                      f"|{s(self.leftBack):s}>{s(self.left):s}>{s(self.leftFro
nt):s}") +
                  f"|{s(self.back):s}>{s(self.vehicle):s}>{s(self.front):s}|"
+
                  ("" if self.rightLane is None else
                      f"{s(self.rightBack):s}>{s(self.right):s}>{s(self.rightF
ront):s}|") +
                  "|"
                 )
```

Recorder

In [9]:
```
class Recorder:

    def __init__(self, startTime=0, stopTime=0, timeStep=1):

        global VEHICLE_ID, LANE_ID
        VEHICLE_ID = 0
        LANE_ID = 0

        self.env = simpy.Environment()
        self.ps = []
        self.startTime = startTime
        self.stopTime = stopTime
        self.timeStep = timeStep
        self.data = pd.DataFrame(columns=['t', 'x', 'v', 'a', 'id', 'lane', 'o
ldLane', 'pos', 'event', 'Vehicle'])

    # runs the simulation
    def run(self):
        self.env.process(self.process())
        self.env.run(self.stopTime+self.timeStep)

    def process(self):
        yield self.env.timeout(self.startTime-self.env.now)
        while self.env.now <= self.stopTime:
            self.record()
            yield self.env.timeout(self.timeStep)

    def startRecording(self, p):
        self.ps.append(p)
        self.record(p, event='start')

    def stopRecording(self, p):
        self.ps.remove(p)

    def record(self, p=None, event='timer'):
        if p is not None:
            if p.updateOnly() or event!='timer':
                laneId = -1 if p.lane is None else p.lane.id
                oldLaneId = -1 if p.oldLane is None else p.oldLane.id
                ix = len(self.data)
                self.data.loc[ix]=[self.env.now, p.x0, p.dx0, p.ddx0, p.id, \
                                   laneId, oldLaneId, p.pos, event, p.VEHICLE_
MODEL]
                if event=='timer':
                    p.update()
        else:
            for p in self.ps:
                self.record(p, event)

    ## additional code: record lane information
    def recordLane(self, lane):
        ix = len(self.data)
        self.data.loc[ix]=[0, 0, 0, 0, 0, lane.id, -1, lane.length, 'lane inf
o', 'No vehicle yet']

    ## additional code: record lane information
```

```
    def getLaneLength(self, laneId):
        lanes = self.data[self.data.event=='lane info']
        lane = lanes[lanes.lane==laneId]
        return lane.pos.iloc[0]

    def saveData(self, filename):
        self.data.to_csv(filename, index=False)

    def getData(self):
        return self.data.copy(deep=True)

    def getEvents(self):
        return self.data[self.data.event!='timer'].copy(deep=True)

    ## new code: computes traffic flow in vehicles/h at the end of a given lan
e
    ## based on 'leave lane' events proper (i.e. excluding overtaking)
    def flow(self, lane):
        df = self.data[self.data.event=='leave lane']
        df = df[df.lane==lane]
        if len(df)>1:
            # compensate for lane filling late or early running dry
            f = (len(df)-1)*3600/(df.t.max()-df.t.min())
            return round(f, 2)
        else:
            return 0

    ## new code: computes the average travelling time through a lane segment
    def avgTravelTime(self, laneId):
        d0 = self.data[self.data.event=='enter lane']
        d0 = d0[d0.lane==laneId]
        d1 = self.data[self.data.event=='leave lane']
        d1 = d1[d1.lane==laneId]
        times = []
        id0 = d0.id.unique()
        id1 = d1.id.unique()
        for id in id0:
            t0 = d0.t[d0.id==id].min()
            if id in id1:
                t1 = d1.t[d1.id==id].max()
                times += [ t1-t0 ]
        avg = 0
        if len(times)>0:
            avg = round(sum(times)/len(times), 2)
        return avg

    def avgSpeed(self, laneId):
        return round(3.6*self.getLaneLength(laneId)/self.avgTravelTime(laneId
),2)


    ## new code: computes traffic density in vehicles/km for a given lane
    ## at a given moment in time. When no time is specified it returns a
    ## list of traffic densities over time.
    def density(self, laneId, time=None):
        timerEvents = self.data[self.data.event=='timer']
        times = timerEvents.t.unique()
```

```
        laneEvents = timerEvents[timerEvents.lane==laneId]
        laneLength = self.getLaneLength(laneId)

        if time is None:
            densities = []
            for t in times:
                events = laneEvents[laneEvents.t==t]
                d = len(events)*1000/laneLength
                densities.append(round(d,2))
            return sum(densities)/len(densities)

        # find the timestamp nearest to the requested time
        if time in times:
            t = time
        else:
            # take the nearest point in time
            diff = list((times-time)**2)
            t = times[diff.index(min(diff))]
        events = laneEvents[laneEvents.t==t]
        d = len(events)*1000/laneLength
        return round(d,2)

    def plot(self, x, y,
             vehicles=None, style='', lw=1, decoration=True,
             x0=None, x1=None, y0=None, y1=None, fillColor=None,
             xmin=None, xmax=None, ymin=None, ymax=None):
        columns = ['t', 'x', 'v', 'a']
        labels = ['Time [s]', 'Position [m]', 'Velocity [m/s]', 'Acceleration
[m/s²]']
        xindex = columns.index(x)
        yindex = columns.index(y)

        plt.figure(figsize=(5, 3), dpi=120)
        if xmin is not None and xmax is not None:
            plt.xlim((xmin, xmax))
        if ymin is not None and ymax is not None:
            plt.ylim((ymin, ymax))

        if vehicles is None:
            vehicles = list(self.data.id.unique())
        for id in vehicles:
            df = self.data[self.data.id==id]
            plt.plot(x, y, style, lw=lw, data=df)
            plt.xlabel(labels[xindex])
            plt.ylabel(labels[yindex])

            if not decoration:
                continue

            # use small red circle to indicate emergency braking
            dc = df[df.event=='brake']
            for i in range(len(dc)):
                X = dc.iloc[i, xindex]
                Y = dc.iloc[i, yindex]
                plt.plot([X], [Y], 'ro')

            db = df[df.event=='brake end']
```

```
        for i in range(len(db)):
            X = db.iloc[i, xindex]
            Y = db.iloc[i, yindex]
            plt.plot([X], [Y], marker='o', mec='r', fillstyle='none')

        # use black 'x' as crash indicator
        dc = df[df.event=='crash']
        for i in range(len(dc)):
            X = dc.iloc[i, xindex]
            Y = dc.iloc[i, yindex]
            plt.plot([X], [Y], 'xk')

        # use black Diamond to indicate that
        # a vehicle ran out of track
        dc = df[df.event=='end']
        for i in range(len(dc)):
            X = dc.iloc[i, xindex]
            Y = dc.iloc[i, yindex]
            plt.plot([X], [Y], 'Dk')

        # use black right pointing full triangle
        # to indicate that a vehicle started
        #  changing into the fast lane
        dc = df[df.event=='change fast']
        for i in range(len(dc)):
            X = dc.iloc[i, xindex]
            Y = dc.iloc[i, yindex]
            plt.plot([X], [Y], '>k')

        # use black right pointing hollow triangle
        # to indicate that a vehicle has finished
        # changing into the fast lane
        dc = df[df.event=='done change fast']
        for i in range(len(dc)):
            X = dc.iloc[i, xindex]
            Y = dc.iloc[i, yindex]
            plt.plot([X], [Y], marker='>', mec='k', fillstyle='none')

        # use black left pointing full triangle
        # to indicate that a vehicle started
        #  changing into the slow lane
        dc = df[df.event=='change slow']
        for i in range(len(dc)):
            X = dc.iloc[i, xindex]
            Y = dc.iloc[i, yindex]
            plt.plot([X], [Y], '<k')

        # use black left pointing hollow triangle
        # to indicate that a vehicle has finished
        # changing into the slow lane
        dc = df[df.event=='done change slow']
        for i in range(len(dc)):
            X = dc.iloc[i, xindex]
            Y = dc.iloc[i, yindex]
            plt.plot([X], [Y], marker='<', mec='k', fillstyle='none')

    # fill area with background color
```

```
        if fillColor is not None:
            if x0 is None:
                x0=self.data[x].min()
            if x1 is None:
                x1=self.data[x].max()
            if y0 is None:
                y0=self.data[y].min()
            if y1 is None:
                y1=self.data[y].max()
            plt.fill_between( [x0, x1], [y0, y0], [y1, y1], color=fillColor)

        plt.grid(True)
```

Testing the functionality

In [10]:
```
import random
VMAX = 120/3.6

N_l = 10 # number of vehicles in the left lane
IAT_l = 3 # average interarrival time left lane

N_c = 10 # number of vehicles in the central lane
IAT_c = 3 # average interarrival time central lane


N_r = 10 # number of vehicles in the right lane
IAT_r = 3 # average interarrival time right lane

random.seed(13)

iat_l = [ rv(IAT_l, delta=0.4*IAT_l) for i in range(N_l) ]
iat_r = [ rv(IAT_r, delta=0.4*IAT_r) for i in range(N_r) ]
iat_c = [ rv(IAT_c, delta=0.4*IAT_c) for i in range(N_c) ]

rec = Recorder(0, 100, 1)

c = Lane(rec, 1000)
c1 = Lane(rec,700)
c2= Lane(rec, 300)
c3= Lane(rec, 1000)

c.extend(c1)
c.extend(c2)
c.extend(c3)

r = c.widenRight()

l = Lane(rec, 1000)
l1 = Lane(rec, 700)
l.extend(l1)

mergeLane = Lane(rec, 300, merge='R')
l.extend(mergeLane)

c.attachLeft(l)

print("L:", l)
print("C:", c)
print("R:", r)



# fill left lane
t = 0
for i in range(N_l):
    t += iat_l[i]
    v = Vehicle(rec, startingLane=l, t0=t, dx0=VMAX, vmax=VMAX,a_max=2.5)

# fill centre lane
t = 0
for i in range(N_c):
```

```python
    t += iat_c[i]
    v = Vehicle(rec, startingLane=c, t0=t, dx0=VMAX, vmax=VMAX,a_max=3)


# fill right lane
t = 0
for i in range(N_r):
    t += iat_r[i]
    v = Vehicle(rec, startingLane=r, t0=t, dx0=VMAX, vmax=VMAX,a_max=5)

rec.run()
```

```
L: [08 x=0.0m l=1000.0m L:-- R:00]-[09 x=1000.0m l=700.0m L:-- R:--]-[10 x=17
00.0m l=300.0m L:-- R:--]\
C: [00 x=0.0m l=1000.0m L:08 R:04]-[01 x=1000.0m l=700.0m L:-- R:05]-[02 x=17
00.0m l=300.0m L:-- R:06]-[03 x=2000.0m l=1000.0m L:-- R:07]
R: [04 x=0.0m l=1000.0m L:00 R:--]-[05 x=1000.0m l=700.0m L:01 R:--]-[06 x=17
00.0m l=300.0m L:02 R:--]-[07 x=2000.0m l=1000.0m L:03 R:--]
```

In [11]:
```python
rec.getData()
```

Out[11]:

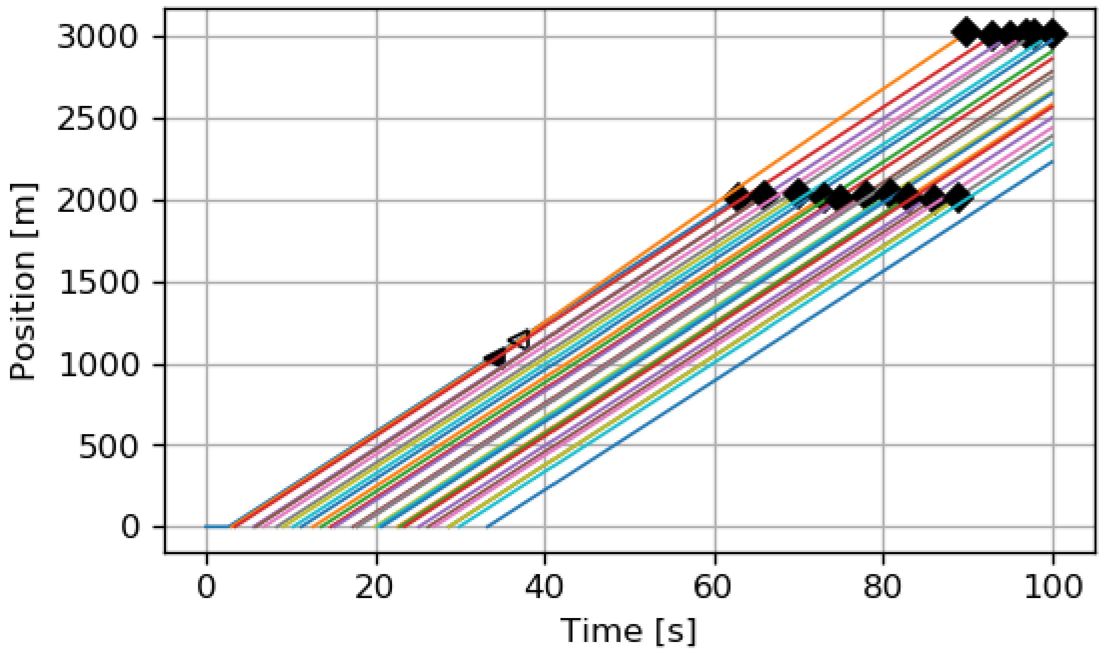
	t	x	v	a	id	lane	oldLane	pos	event	Vehicle
0	0	0	0	0	0	0	-1	1000	lane info	No vehicle yet
1	0	0	0	0	0	1	-1	700	lane info	No vehicle yet
2	0	0	0	0	0	2	-1	300	lane info	No vehicle yet
3	0	0	0	0	0	3	-1	1000	lane info	No vehicle yet
4	0	0	0	0	0	4	-1	1000	lane info	No vehicle yet
...	...	...	...	...	...	...	...	...	...	...
2495	100	2499.44	33.4965	0.00234936	28	7	-1	499.44	timer	car
2496	100	2438.59	33.464	0.00193117	17	3	-1	438.59	timer	car
2497	100	2388.38	33.72	0.00585171	29	7	-1	388.38	timer	car
2498	100	2340.25	33.5047	0.00265042	18	3	-1	340.25	timer	car
2499	100	2229.82	33.5453	0.00341758	19	3	-1	229.82	timer	car

2500 rows × 10 columns

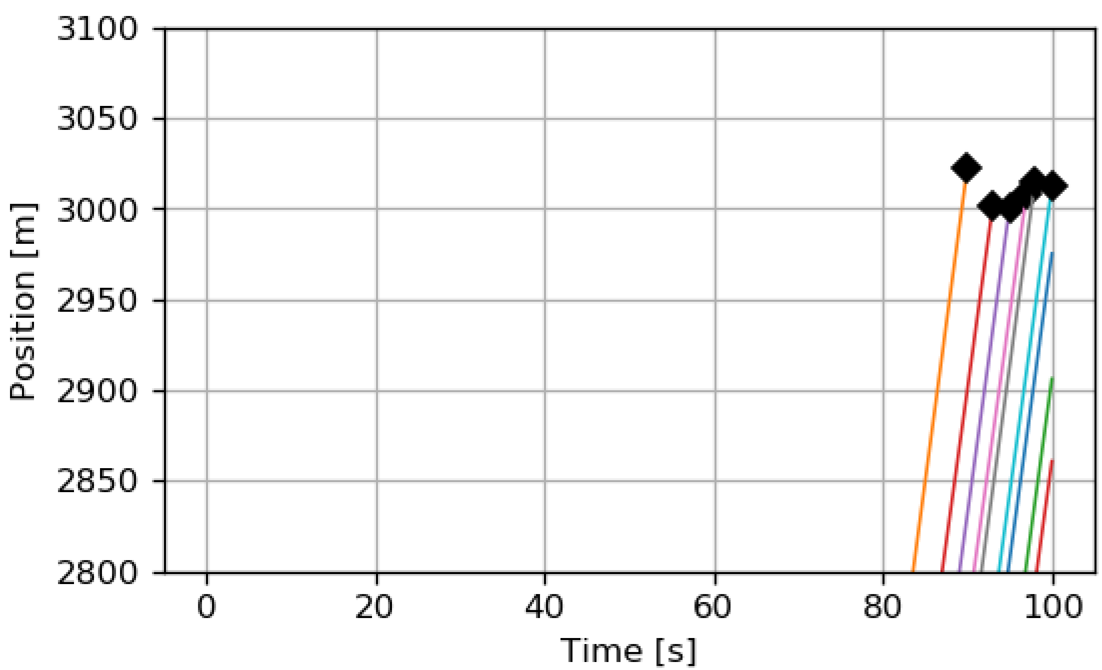
In [12]:
```python
rec.data['Vehicle'].unique()
```

Out[12]:
```
array(['No vehicle yet', 'car'], dtype=object)
```

In [13]:
```
rec.plot('t', 'x')
```



In [14]:
```
rec.plot('t', 'x', ymin=2800, ymax=3100)
```



Traffic Flow in veh/h

The traffic flow at the end of lane segments for the **left lane**:

In [15]:
```
rec.flow(6), rec.flow(7) #rec.flow(4)
```
Out[15]:
```
(1252.17, 1200.0)
```

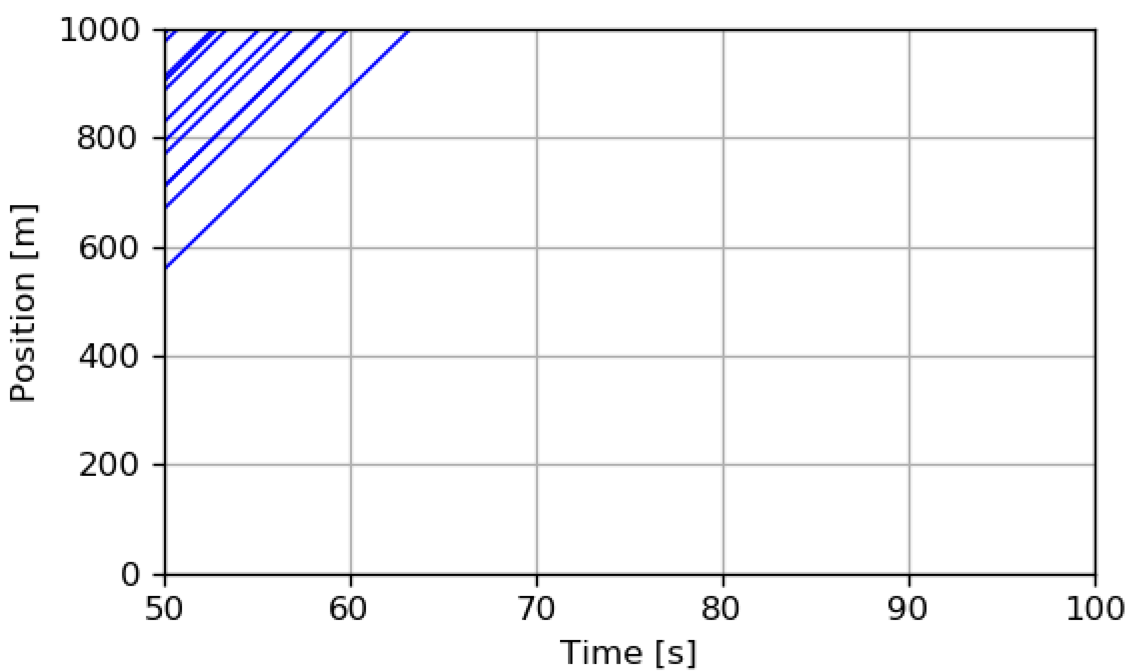
The traffic flow at the end of lane segments for the **right lane**:

In [16]:
```
rec.flow(3), rec.flow(4), rec.flow(5)
```
Out[16]:
```
(1080.0, 1296.0, 1252.17)
```

The traffic flow at the end of lane segments for the **centre lane**:

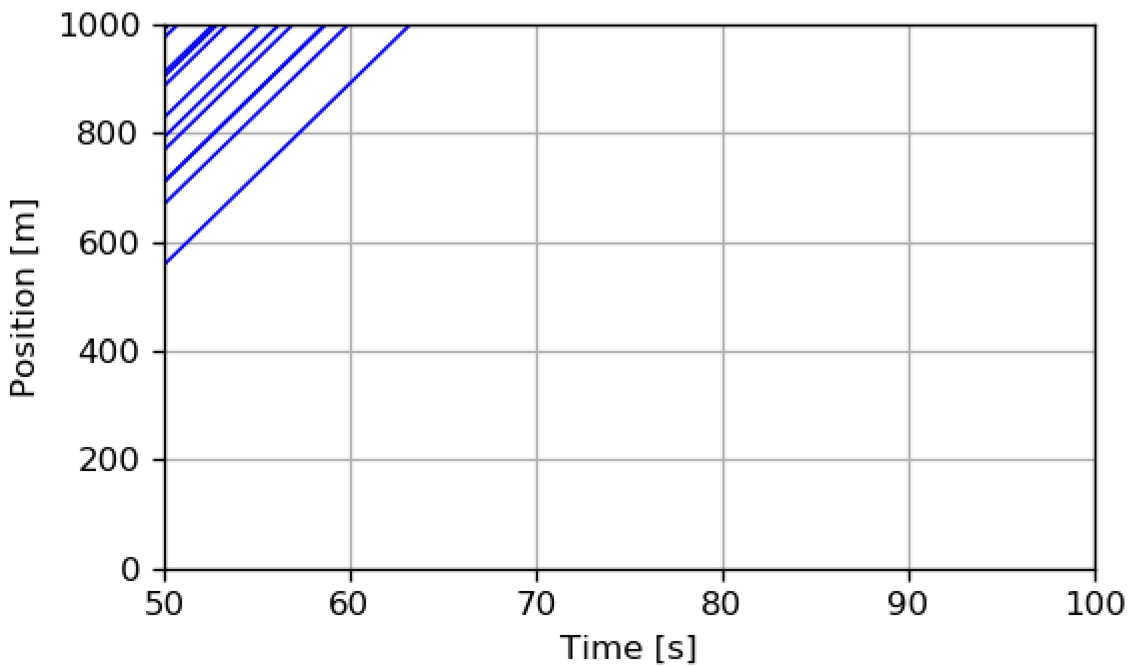
In [17]:
```
rec.flow(0), rec.flow(1), rec.flow(2)
```
Out[17]:
```
(1080.0, 1125.0, 1090.91)
```

In [18]:
```
rec.plot('t', 'x', xmin=50, xmax=100, ymin=0, ymax=1000, style='b')
```



Traffic Density in veh/km

In [19]:
```
rec.plot('t', 'x', xmin=50, xmax=100, ymin=0, ymax=1000, style='b')
```



In [20]:
```
#The statistics reports:
print(rec.density(0, 50) + rec.density(1, 50)+rec.density(2, 50))
print(rec.density(3, 50) + rec.density(4, 50)+rec.density(5, 50))
print(rec.density(6, 50) + rec.density(7, 50))
```

```
13.57
11.57
0.0
```

Average Travel Times in s

In [21]:
```
rec.avgTravelTime(0)
```

Out[21]: 30.38

Average Speed in km/h

In [22]:
```
rec.avgSpeed(0)
```

Out[22]: 118.5

Fundamental Law

In [23]:
```
rec.density(0, 50)*rec.avgSpeed(0) + rec.density(3, 50)*rec.avgSpeed(3)
```
Out[23]:
```
592.5
```
In [ ]:

Testing Simulation with different Vehicle types

Case 4

1000 Vehicles, 300 in left Lane, 400 in center lane and 300 in Right Lane

Inter-Arrival Time Distributions = 3 sec for each lane

Traffic Mix = 45% of "Car", 20% of "Electric Car", 35% of " heavy goods vehicles"

Motorway speed Limit Setting = 60Km/h = 100/3.6 m/s

In [24]:
```
import random
VMAX = 100/3.6

N_l = 300 # number of vehicles in the left lane
IAT_l = 3 # average interarrival time left lane

N_c = 400 # number of vehicles in the central lane
IAT_c = 3 # average interarrival time central lane

N_r = 300 # number of vehicles in the right lane
IAT_r = 3 # average interarrival time right lane

random.seed(13)

iat_l = [ rv(IAT_l, delta=0.4*IAT_l) for i in range(N_l) ]
iat_r = [ rv(IAT_r, delta=0.4*IAT_r) for i in range(N_r) ]
iat_c = [ rv(IAT_c, delta=0.4*IAT_c) for i in range(N_c) ]

rec = Recorder(0, 100, 1)



c = Lane(rec, 1700)
c1 = Lane(rec,300)
c2= Lane(rec, 1000)
c.extend(c1)
c.extend(c2)


r = Lane(rec, 1700)
r1 = Lane(rec,300)
r2= Lane(rec, 1000)
r.extend(r1)
r.extend(r2)
c.attachRight(r)

#r = c.widenRight()
l = Lane(rec, 1700)
mergeLane = Lane(rec, 300, merge='R')
l.extend(mergeLane)
c.attachLeft(l)



# c = Lane(rec, 1000)
# c1 = Lane(rec,700)
# c2= Lane(rec, 300)
# c3= Lane(rec, 1000)

# c.extend(c1)
# c.extend(c2)
# c.extend(c3)



# r = Lane(rec, 1000)
```

```
# r1 = Lane(rec,700)
# r2= Lane(rec, 300)
# r3= Lane(rec, 1000)

# r.extend(r1)
# r.extend(r2)
# r.extend(r3)

# c.attachRight(r)


# #r = c.widenRight()





# l = Lane(rec, 1000)
# l1 = Lane(rec, 700)
# l.extend(l1)

# mergeLane = Lane(rec, 300, merge='R')
# l.extend(mergeLane)

# c.attachLeft(l)

print("L:", l)
print("C:", c)
print("R:", r)


#for right
t=0
for i in range(N_r):
    t += iat_r[i]
    v = Vehicle(rec, startingLane=r, t0=t, dx0=VMAX,vmax=VMAX)


t = 0
# fill left lane
for i in range(N_l):
    t += iat_l[i]
#    v = Vehicle(rec, startingLane=l, t0=t, dx0=VMAX, vmax=VMAX,a_max=2.5)

    r=random.random()
    if r<=0.2:#electric car
        v = Vehicle(rec, startingLane=l, t0=t, dx0=VMAX, vmax=VMAX,a_max=2.5,
VEHICLE_MODEL='electric car')
    elif r<=0.55:
        v = Vehicle(rec, startingLane=l, t0=t, dx0=VMAX, vmax=VMAX,a_max=2.5,
VEHICLE_MODEL='HGV')
    else:
        v = Vehicle(rec, startingLane=l, t0=t, dx0=VMAX, vmax=VMAX,a_max=2.5)
```

```
# fill centre lane
t=0
for i in range(N_c):
    t += iat_c[i]
#    v = Vehicle(rec, startingLane=c, t0=t, dx0=VMAX, vmax=VMAX,a_max=2.5)

    r=random.random()
    if r<=0.2:
        v = Vehicle(rec, startingLane=c, t0=t, dx0=VMAX, vmax=VMAX,a_max=2.5, 
VEHICLE_MODEL='electric car')
    elif r<=0.55:
        v = Vehicle(rec, startingLane=c, t0=t, dx0=VMAX, vmax=VMAX,a_max=2.5, 
VEHICLE_MODEL='HGV')
    else:
        v = Vehicle(rec, startingLane=c, t0=t, dx0=VMAX, vmax=VMAX,a_max=2.5)




#     r=random.random()
#     if r<=0.2:
#         v = Vehicle(rec, startingLane=r, t0=t, dx0=VMAX, vmax=VMAX,a_max=2.
5, VEHICLE_MODEL='electric car')
#     elif r<=0.4:
#         v = Vehicle(rec, startingLane=r, t0=t, dx0=VMAX, vmax=VMAX,a_max=2.
5, VEHICLE_MODEL='HGV')
#     else:
#         v = Vehicle(rec, startingLane=r, t0=t, dx0=VMAX, vmax=VMAX,a_max=2.
5)



rec.run()
```

```
L: [06 x=0.0m l=1700.0m L:-- R:00]-[07 x=1700.0m l=300.0m L:-- R:--]\
C: [00 x=0.0m l=1700.0m L:06 R:03]-[01 x=1700.0m l=300.0m L:-- R:--]-[02 x=20
00.0m l=1000.0m L:-- R:--]
R: [03 x=0.0m l=1700.0m L:00 R:--]-[04 x=1700.0m l=300.0m L:-- R:--]-[05 x=20
00.0m l=1000.0m L:-- R:--]
```

In [25]:
```
rec.getData()
```

Out[25]:

	t	x	v	a	id	lane	oldLane	pos	event	Vehicle
0	0	0	0	0	0	0	-1	1700	lane info	No vehicle yet
1	0	0	0	0	0	1	-1	300	lane info	No vehicle yet
2	0	0	0	0	0	2	-1	1000	lane info	No vehicle yet
3	0	0	0	0	0	3	-1	1700	lane info	No vehicle yet
4	0	0	0	0	0	4	-1	300	lane info	No vehicle yet
...	...	...	...	...	...	...	...	...	...	...
5078	100	90.41	27.7798	0.00122169	31	3	-1	90.41	timer	car
5079	100	89.68	27.778	0.000113134	331	6	-1	89.68	timer	electric car
5080	100	29.3	27.7782	0.00079295	632	0	-1	29.3	timer	car
5081	100	26.32	27.7778	1.79819e-05	32	3	-1	26.32	timer	car
5082	100	11.24	27.7778	0.000219815	332	6	-1	11.24	timer	car

5083 rows × 10 columns

In [ ]:

Statistics of self organising motorway scenarios

Before Merging at 1700m

In [26]:

```
# rec.flow(6)+rec.flow(7)
# rec.flow(0)+ rec.flow(1)+ rec.flow(2)
# rec.flow(3)+ rec.flow(4)+ rec.flow(5)


# The traffic Volume at the end of lane segments for the different lanes (veh/
hr):

print("***************************** After 1700m (1KM)***********************
****")
print("The traffic Volume at 1700m the left lane : ", rec.flow(6))
print(" The traffic Volume at 1700m the right lane :", rec.flow(3))
print(" The traffic Volume at 1700m the centre lane", rec.flow(0))
print(" \n The traffic Volume at 1700m in all lane", rec.flow(0)+ rec.flow(3)+
rec.flow(6))

print("\n ----------------------------------------------------")
# Traffic Density at the end of lane segments for the different lanes (veh/k
m):

print("The traffic density at 1700m the left lane : ", rec.density(6, 50))
print(" The traffic density at 1700m the right lane :", rec.density(3, 50))
print(" The traffic density at 1700m the centre lane", rec.density(0, 50))
print(" \nThe traffic density at 1700m in all lane", rec.density(6, 50)+ rec.d
ensity(0, 50)+ rec.density(3, 50))

print("\n ----------------------------------------------------")
# Average travel time:

print("Average Travel travel at 1700m the left lane :", rec.avgTravelTime(6))
print("Average Travel travel at 1700m the right lane :", rec.avgTravelTime(3))
print("Average Travel travel at 1700m the centre lane :", rec.avgTravelTime(0
))
print("\nAverage Travel travel at 1700m in all lane :", (rec.avgTravelTime(0)+
rec.avgTravelTime(6)+ rec.avgTravelTime(3))/3)

print("\n ----------------------------------------------------")

# Average Speed

print("Average speed at the 1700m the left lane :", rec.avgSpeed(6))
print("Average speed at the 1700m the right lane :", rec.avgSpeed(3))
#print("Average speed at the 1700m the centre lane :", rec.avgSpeed(0))
print("Average speed at the 1700m the all lane :", (rec.avgSpeed(0)+ rec.avgSp
eed(3)+ rec.avgSpeed(6))/3)
```

```
***************************** After 1700m (1KM)****************************
The traffic Volume at 1700m the left lane :  1270.59
 The traffic Volume at 1700m the right lane : 1164.71
 The traffic Volume at 1700m the centre lane 1234.29

 The traffic Volume at 1700m in all lane 3669.59

 ----------------------------------------------------
The traffic density at 1700m the left lane :  10.0
 The traffic density at 1700m the right lane : 8.82
 The traffic density at 1700m the centre lane 9.41

The traffic density at 1700m in all lane 28.23

 ----------------------------------------------------
Average Travel travel at 1700m the left lane : 61.33
Average Travel travel at 1700m the right lane : 61.45
Average Travel travel at 1700m the centre lane : 61.36

Average Travel travel at 1700m in all lane : 61.379999999999995

 ----------------------------------------------------
Average speed at the 1700m the left lane : 99.79
Average speed at the 1700m the right lane : 99.59
Average speed at the 1700m the all lane : 99.70666666666666
```

After Merging 3 lane to 2 lane at 2000m

In [27]:

```
# The traffic Volume at the end of lane segments for the different lanes (veh/
hr):

print("***************************** After 2000m (****************************
\n")
print("The traffic Volume at 2000m the left lane : ", rec.flow(7))
print(" The traffic Volume at 2000m the right lane :", rec.flow(4))
print(" The traffic Volume at 2000m the centre lane", rec.flow(1))
print(" \n The traffic Volume at 2000m in all lane", rec.flow(7)+ rec.flow(1)+
rec.flow(4))

print("\n ----------------------------------------------------")
# Traffic Density at the end of lane segments for the different lanes (veh/k
m):

print("The traffic density at 2000m the left lane : ", rec.density(7, 100))
print(" The traffic density at 2000m the right lane :", rec.density(4, 100))
print(" The traffic density at 2000m the centre lane", rec.density(1, 100))
print(" \nThe traffic density at 2000m in all lane", rec.density(1, 100)+ rec.
density(7, 100)+ rec.density(4, 100))

print("\n ----------------------------------------------------")
# Average travel time:

print("Average Travel travel at 2000m the left lane :", rec.avgTravelTime(7))
print("Average Travel travel at 2000m the right lane :", rec.avgTravelTime(4))
print("Average Travel travel at 2000m the centre lane :", rec.avgTravelTime(1
))
print("\nAverage Travel travel at 2000m in all lane :", (rec.avgTravelTime(1)+
rec.avgTravelTime(4)+ rec.avgTravelTime(7))/3)

print("\n ----------------------------------------------------")

# Average Speed

#print("Average speed at the 2000m the left lane :", rec.avgSpeed(7))
#print("Average speed at the 2000m the right lane :", rec.avgSpeed(4))
#print("Average speed at the 2000m the centre lane :", rec.avgSpeed(1))
#print("Average speed at the 2000m the all lane :", (rec.avgSpeed(1)+ rec.avgS
peed(4)+ rec.avgSpeed(7))/3)
```

```
**************************** After 2000m (****************************

The traffic Volume at 2000m the left lane :  1252.17
 The traffic Volume at 2000m the right lane : 1145.45
 The traffic Volume at 2000m the centre lane 1200.0

 The traffic Volume at 2000m in all lane 3597.62

 ----------------------------------------------------
The traffic density at 2000m the left lane :  13.33
 The traffic density at 2000m the right lane : 13.33
 The traffic density at 2000m the centre lane 13.33

The traffic density at 2000m in all lane 39.99

 ----------------------------------------------------
Average Travel travel at 2000m the left lane : 10.62
Average Travel travel at 2000m the right lane : 10.75
Average Travel travel at 2000m the centre lane : 10.67

Average Travel travel at 2000m in all lane : 10.68

 ----------------------------------------------------
```

In [ ]:

At the end of the motorway section at 3000m

In [28]:
```
# The traffic Volume at the end of lane segments for the different lanes (veh/
hr):

print("***************************** After 3000m (****************************
\n")

print(" The traffic Volume at 3000m the right lane :", rec.flow(5))
print(" The traffic Volume at 3000m the centre lane", rec.flow(2))
print(" \n The traffic Volume at 3000m in all lane", rec.flow(2)+ rec.flow(5))

print("\n ----------------------------------------------------")
# Traffic Density at the end of lane segments for the different lanes (veh/k
m):

print(" The traffic density at 3000m the right lane :", rec.density(5, 50))
print(" The traffic density at 3000m the centre lane", rec.density(2, 50))
print(" \nThe traffic density at 3000m in all lane", rec.density(5, 50)+ rec.d
ensity(2, 50))

print("\n ----------------------------------------------------")
# Average travel time:

print("Average Travel travel at 3000m the right lane :", rec.avgTravelTime(5))
print("Average Travel travel at 3000m the centre lane :", rec.avgTravelTime(2
))
#print("\nAverage Travel travel at 3000m in all lane :", (rec.avgTravelTime(5)
+ rec.avgTravelTime(2))/2)

print("\n ----------------------------------------------------")

# Average Speed

#print("Average speed at the 3000m the right lane :", rec.avgSpeed(5))
#print("Average speed at the 3000m the centre lane :", rec.avgSpeed(2))
#print("Average speed at the 3000m the all lane :", (rec.avgSpeed(5)+ rec.avgS
peed(2))/2)
```

```
***************************** After 3000m (****************************

 The traffic Volume at 3000m the right lane : 0
 The traffic Volume at 3000m the centre lane 0

 The traffic Volume at 3000m in all lane 0

 ----------------------------------------------------
 The traffic density at 3000m the right lane : 0.0
 The traffic density at 3000m the centre lane 0.0

The traffic density at 3000m in all lane 0.0

 ----------------------------------------------------
Average Travel travel at 3000m the right lane : 0
Average Travel travel at 3000m the centre lane : 0

 ----------------------------------------------------
```

Part 2 Vehicle Type with Driver behaviour

In [ ]:

In [29]:
```
free_speed = [ 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170 ]
quantiles = np.cumsum([0, 0.002, 0.015, 0.025, 0.178, 0.372, 0.390, 0.119
, 0.148, 0.03, 0.001])
def random_speed():
  u = random.random() # uniformly distributed random number generated between
 0 and 1
  for i in range(len(quantiles)):
    if u<quantiles[i+1]:
      p = (u-quantiles[i])/(quantiles[i+1]-quantiles[i])
      return free_speed[i]*p+free_speed[i+1]*(1-p)
```

In [30]:
```
random.seed(0)
speeds = [ random_speed() for i in range(210000)]
kernel = stats.gaussian_kde(speeds)
```

In [31]:
```
v1 = np.arange(60, 170) #generate spread out from 60 170
q1 = [ kernel.integrate_box_1d(30, i) for i in v1 ]
def freeMotorwaySpeed():
  u = random.random() # generates uniformly distributed random number between
 0 and 1
  for i in range(len(q1)):
    if u<q1[i+1]:
      p1 = (u-q1[i])/(q1[i+1]-q1[i])
      # return (free_speed[i+1]+free_speed[i])/2*p+(1-p)*(free_speed[i+1]+free
_speed[i+2])/2
      return v1[i]*p1+v1[i+1]*(1-p1)


freeMotorwaySpeed()
```

Out[31]: 84.19903863508866

In [32]:

```
free_speed_bins = [ 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160]
free_speed_quantiles = np.cumsum([0, 0.006, 0.014, 0.052, 0.148, 0.27, 0.309,
0.143, 0.048, 0.01])

def free_speed_distribution():
    u = random.random() # generates uniformly distributed random number betwee
n 0 and 1
    for i in range(len(free_speed_quantiles)):
        if u<free_speed_quantiles[i+1]:
            p = (u-free_speed_quantiles[i])/(free_speed_quantiles[i+1]-free_sp
eed_quantiles[i])
            return free_speed_bins[i]*p+free_speed_bins[i+1]*(1-p)

free_speed_data = [ free_speed_distribution() for i in range(1200)]
free_speed_kernel = stats.gaussian_kde(free_speed_data)

free_speeds = np.arange(30, 191)
free_speed_cdf = [ free_speed_kernel.integrate_box_1d(30, i) for i in free_spe
eds ]

def random_free_speed():
    u = random.random() # generates uniformly distributed random number betwee
n 0 and 1
    for i in range(len(free_speeds)):
        if u<free_speed_cdf[i+1]:
            p = (u-free_speed_cdf[i])/(free_speed_cdf[i+1]-free_speed_cdf[i])
            return free_speeds[i]*p+free_speeds[i+1]*(1-p)
```

In [33]:

```
speeds = [ random_free_speed() for i in range(100000) ]
```

```
In [34]: def VehiclePlusDriverType(HGV, electric, car, autonomous = 0, d_behaviour=1):
  w = random.random()

  if w <= autonomous:
    vdrivers = 'Human'
  else:
    vdrivers = 'auto'

  d = random.random()
  if d < d_behaviour:
    behaviour = 'normal'
  elif d > d_behaviour:
    behaviour = 'drunk'
  else:
    behaviour='rash'

  vFAR_AWAY_IN_FRONT = 200
  vFAR_AWAY_IN_BACK = 80

  u = random.random()
  if u <= HGV:
    vtype = 'HGV'
    vstartspeed = freeMotorwaySpeed()/3.6 #### vstartspeed = freeMotorwaySpeed
Lorry()/3.6
    vlength = 15
    va_max = 0.96

    if vdrivers == 'Human':
        if behaviour== "drunk":
            # fixed 0.4 difference
            va_min = -1.28 + 0.125*random.random()
        elif behaviour == "normal":
            va_min = -0.88 + 0.125*random.random()
        elif behaviour == "rash":
            # fixed 0.25 difference
            va_min = -0.63 + 0.125*random.random()


    else:
      va_min = -0.88
      vFAR_AWAY_IN_FRONT = 200
      vFAR_AWAY_IN_BACK = 110

  elif u <= HGV + electric: #######??????
    vtype = 'Electric Car'
    vstartspeed = freeMotorwaySpeed()/3.6
    vlength = 6
    va_max = 2.47
    if vdrivers =='Human':
         if behaviour== "drunk":
            va_min = -4.37 + 0.5*random.random()
         elif behaviour == "normal":
            va_min = -3.97 + 0.5*random.random()
         elif behaviour == "rash":
            va_min = -3.72 + 0.125*random.random()
```

```
    else:
      va_min = -3.97
      vFAR_AWAY_IN_FRONT = 200
      vFAR_AWAY_IN_BACK = 80
  else:
    vtype = 'Car'
    vstartspeed = freeMotorwaySpeed()/3.6
    vlength = 4
    va_max = 2.47
    if vdrivers == 'Human':
        if behaviour== "drunk":
            va_min = -2.15 + 0.4*random.random()
        elif behaviour == "normal":
            va_min = -2.55 + 0.4**random.random()
        elif behaviour == "rash":
            va_min = -2.8 + 0.125*random.random()

    else:
      va_min = -3.97
      vFAR_AWAY_IN_FRONT = 200
      vFAR_AWAY_IN_BACK = 80

  return vtype, vstartspeed, vlength, va_max, va_min, vFAR_AWAY_IN_FRONT, vFAR
_AWAY_IN_BACK, vdrivers, behaviour
```

In [35]:
```
SLOW_CYCLE = 100

def randomIntervals(cycles):
  # return [ random.expovariate(1.0/SLOW_CYCLE)+10 for i in range(cycles) ]
  return [ max(0, random.normalvariate(SLOW_CYCLE, SLOW_CYCLE/3)) for i in ran
ge(cycles) ]
```

In [36]:
```
def cyc(): # random element to cycles
  cycles = 5 + round(5*(random.random()))
  return cycles
cyc()
```

Out[36]:
```
10
```

In [37]:
```
SPEED_VARIATION = 0.05 #To get an idea about the time intervals and the speed
 approached we look at the first few random values:
random.seed(13)

def randomSpeedVariation(vmax, cycles, cv=SPEED_VARIATION):
  return [ vmax + (-1)**i*abs(random.normalvariate(0, vmax*cv)) for i in range
(cycles) ]
```

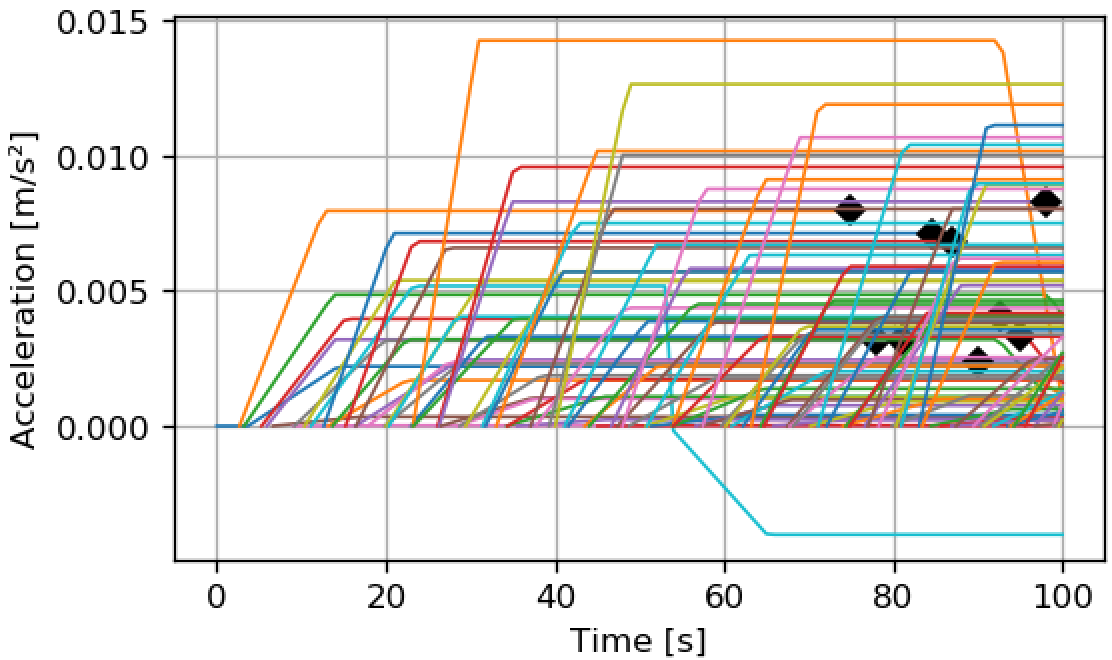
In [ ]:

In [ ]:

Simulation of Part 2

Case 1

In [38]:
```
rec.plot('t', 'a')
```



In [ ]:

Statistics of self organising motorway scenarios

Before Merging at 1700m

In [39]:

```python
# The traffic Volume at the end of lane segments for the different lanes (veh/
hr):

print("**************************** After 1700m (1KM)***************************")
print("The traffic Volume at 1700m the left lane : ", rec.flow(6))
print(" The traffic Volume at 1700m the right lane :", rec.flow(3))
print(" The traffic Volume at 1700m the centre lane", rec.flow(0))
print(" \n The traffic Volume at 1700m in all lane", rec.flow(0)+ rec.flow(3)+
rec.flow(6))

print("\n ----------------------------------------------------")
# Traffic Density at the end of lane segments for the different lanes (veh/k
m):

print("The traffic density at 1700m the left lane : ", rec.density(6, 50))
print(" The traffic density at 1700m the right lane :", rec.density(3, 50))
print(" The traffic density at 1700m the centre lane", rec.density(0, 50))
print(" \nThe traffic density at 1700m in all lane", rec.density(6, 50)+ rec.d
ensity(0, 50)+ rec.density(3, 50))

print("\n ----------------------------------------------------")
# Average travel time:

print("Average Travel travel at 1700m the left lane :", rec.avgTravelTime(6))
print("Average Travel travel at 1700m the right lane :", rec.avgTravelTime(3))
print("Average Travel travel at 1700m the centre lane :", rec.avgTravelTime(0
))
print("\nAverage Travel travel at 1700m in all lane :", (rec.avgTravelTime(0)+
rec.avgTravelTime(6)+ rec.avgTravelTime(3))/3)

print("\n ----------------------------------------------------")
```

```
***************************** After 1700m (1KM)****************************
The traffic Volume at 1700m the left lane :  1270.59
 The traffic Volume at 1700m the right lane : 1164.71
 The traffic Volume at 1700m the centre lane 1234.29

 The traffic Volume at 1700m in all lane 3669.59

 ----------------------------------------------------
The traffic density at 1700m the left lane :  10.0
 The traffic density at 1700m the right lane : 8.82
 The traffic density at 1700m the centre lane 9.41

The traffic density at 1700m in all lane 28.23

 ----------------------------------------------------
Average Travel travel at 1700m the left lane : 61.33
Average Travel travel at 1700m the right lane : 61.45
Average Travel travel at 1700m the centre lane : 61.36

Average Travel travel at 1700m in all lane : 61.379999999999995

 ----------------------------------------------------
```

After Merging 3 lane to 2 lane at 2000m

In [40]:
```
# The traffic Volume at the end of lane segments for the different lanes (veh/
hr):

print("****************************** After 2000m (****************************
\n")
print("The traffic Volume at 2000m the left lane : ", rec.flow(7))
print(" The traffic Volume at 2000m the right lane :", rec.flow(4))
print(" The traffic Volume at 2000m the centre lane", rec.flow(1))
print(" \n The traffic Volume at 2000m in all lane", rec.flow(7)+ rec.flow(1)+
rec.flow(4))

print("\n ----------------------------------------------------")
# Traffic Density at the end of lane segments for the different lanes (veh/k
m):

print("The traffic density at 2000m the left lane : ", rec.density(7, 100))
print(" The traffic density at 2000m the right lane :", rec.density(4, 100))
print(" The traffic density at 2000m the centre lane", rec.density(1, 100))
print(" \nThe traffic density at 2000m in all lane", rec.density(1, 100)+ rec.
density(7, 100)+ rec.density(4, 100))

print("\n ----------------------------------------------------")
# Average travel time:

print("Average Time travel at 2000m the left lane :", rec.avgTravelTime(7))
print("Average Time travel at 2000m the right lane :", rec.avgTravelTime(4))
print("Average Time travel at 2000m the centre lane :", rec.avgTravelTime(1))
print("\nAverage time travel at 2000m in all lane :", (rec.avgTravelTime(1)+ r
ec.avgTravelTime(4)+ rec.avgTravelTime(7))/3)

print("\n ----------------------------------------------------")

# Average Speed

#print("Average speed at the 2000m the left lane :", rec.avgSpeed(7))
#print("Average speed at the 2000m the right lane :", rec.avgSpeed(4))
#print("Average speed at the 2000m the centre lane :", rec.avgSpeed(1))
#print("Average speed at the 2000m the all lane :", (rec.avgSpeed(1)+ rec.avgS
peed(4)+ rec.avgSpeed(7))/3)
```

```
***************************** After 2000m (****************************

The traffic Volume at 2000m the left lane :  1252.17
 The traffic Volume at 2000m the right lane : 1145.45
 The traffic Volume at 2000m the centre lane 1200.0

 The traffic Volume at 2000m in all lane 3597.62

 ----------------------------------------------------
The traffic density at 2000m the left lane :  13.33
 The traffic density at 2000m the right lane : 13.33
 The traffic density at 2000m the centre lane 13.33

The traffic density at 2000m in all lane 39.99

 ----------------------------------------------------
Average Time travel at 2000m the left lane : 10.62
Average Time travel at 2000m the right lane : 10.75
Average Time travel at 2000m the centre lane : 10.67

Average time travel at 2000m in all lane : 10.68

 ----------------------------------------------------
```

In [ ]:

At the end of the motorway section at 3000m

In [41]:

```
# The traffic Volume at the end of lane segments for the different lanes (veh/
hr):

print("***************************** After 3000m (****************************
\n")

print(" The traffic Volume at 3000m the right lane :", rec.flow(5))
print(" The traffic Volume at 3000m the centre lane", rec.flow(2))
print(" \n The traffic Volume at 3000m in all lane", rec.flow(2)+ rec.flow(5))

print("\n ----------------------------------------------------")
# Traffic Density at the end of lane segments for the different lanes (veh/k
m):

print(" The traffic density at 3000m the right lane :", rec.density(5, 50))
print(" The traffic density at 3000m the centre lane", rec.density(2, 50))
print(" \nThe traffic density at 3000m in all lane", rec.density(5, 50)+ rec.d
ensity(2, 50))

print("\n ----------------------------------------------------")
# Average travel time:

print("Average Travel travel at 3000m the right lane :", rec.avgTravelTime(5))
print("Average Travel travel at 3000m the centre lane :", rec.avgTravelTime(2
))
print("\nAverage Travel travel at 3000m in all lane :", (rec.avgTravelTime(5)+
rec.avgTravelTime(2))/2)

print("\n ----------------------------------------------------")

# Average Speed

#print("Average speed at the 3000m the right lane :", rec.avgSpeed(5))
#print("Average speed at the 3000m the centre lane :", rec.avgSpeed(2))
#print("Average speed at the 3000m the all lane :", (rec.avgSpeed(5)+ rec.avgS
peed(2))/2)
```

```
**************************** After 3000m (****************************

 The traffic Volume at 3000m the right lane : 0
 The traffic Volume at 3000m the centre lane 0

 The traffic Volume at 3000m in all lane 0

 ---------------------------------------------------
 The traffic density at 3000m the right lane : 0.0
 The traffic density at 3000m the centre lane 0.0

The traffic density at 3000m in all lane 0.0

 ---------------------------------------------------
Average Travel travel at 3000m the right lane : 0
Average Travel travel at 3000m the centre lane : 0

Average Travel travel at 3000m in all lane : 0.0

 ---------------------------------------------------
```

In [42]:
```
rec.saveData("Data_x20192037_priyanka_simulation1_100kmperhour.csv")
```

In [ ]:

In [ ]: