Packages

In [1]:
```
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import scipy.stats as stats
import math
import numpy as np
import random

import simpy
```

In [2]:
```
ε = 0.00001

def isZero(x):
    return abs(x)<ε
```

Entities

Here we have chnaged the constant values as per Car

In [3]:
```
# Time tolerance: when at current speed difference a crash might occur within that number of seconds
CRITICAL_TIME_TOLERANCE = 4  # [s]
LANE_CHANGE_TIME = 4 # [s]
MIN_TIME_DIFF = 1

MIN_SPEED_DIFF = 4 # [m/s] min speed diff to trigger overtaking

CAR_LENGTH = 5 # [m]

FAR_AWAY_IN_FRONT = 180 # [m]  distance at which a car in front can be ignored
FAR_AWAY_IN_BACK = 70  # [m]  distance at which a car behind can be ignored
```

Lanes

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
```

In [5]:
```
LANE_ID = 0

class Lane:

    ## some additional code
    def __init__(self, length, speedLimit):

        global LANE_ID
        self.id = LANE_ID
        LANE_ID += 1

        self.length = length
        self.speedLimit = speedLimit
        self.vehicles = []

        self.next = None
        self.prev = None

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

    # defines generic str() method for Lanes
    # extends the method with list of vehicles on the lane
    def __str__(self):
        l = "" if self.left is None else f" L:{self.left.id:d}"
        r = "" if self.right is None else f" R:{self.right.id:d}"
        vs = "" if len(self.vehicles)==0 else " "
        for v in self.vehicles:
            vs += str(v)
        return f"[{self.id:d} {int(self.length):d}m"+l+r+vs+"]" + \
                    ("-" + 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

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

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

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

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

    ## additional code
    ## new generalised access method needed to calculate sideway view
    ## returns all vehicles between pos+distFrom and pos+distTo
    def at(self, pos, distFrom=-CAR_LENGTH/2, distTo=CAR_LENGTH/2):
        # 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 pos+distFrom < v.pos and v.pos < pos+distTo:
                res.append(v)
        # if the required distance reaches over the end of the lane segment
        if pos+distTo > self.length and self.next is not None:
            res = res + self.next.at(0, distFrom=0, distTo=distTo-(self.length-pos))
        if pos+distFrom < 0 and self.prev is not None:
            res = self.prev.at(self.prev.length, distFrom=pos+distFrom, distTo=0) + res
        return res

    def inFront(self, pos, far=FAR_AWAY_IN_FRONT):
        # 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=far-(self.length-pos))

    def behind(self, pos, far=FAR_AWAY_IN_BACK):
        # 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 lane
        if pos>far or self.prev is None:
            return None
        else:
            return self.prev.behind(self.prev.length, far=far-pos)

    def enter(self, vehicle, pos=0):
        self.vehicles.insert(0, vehicle)
        vehicle.pos = pos
        vehicle.lane = self
        vehicle.rec.record(vehicle, event="enter lane")

    def leave(self, vehicle):
        vehicle.rec.record(vehicle, event="leave lane")
        vehicle.lane = None
        # in the meantime the vehicle may have have moved
        # to one of the next lane segments...
        lane = self
        while lane is not None:
            if vehicle in lane.vehicles:
                lane.vehicles.remove(vehicle)
                break
            else:
                lane = lane.next
```

Vehicles

In [6]:

```
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

class Vehicle:
    def __init__(self, env, rec,
                 startingLane=None, startingPos=0,
                 t0=0, x0=0, dx0=0, ddx0=0, dddx0=0,
                 t=[], v=[]):

        global VEHICLE_ID
        self.id = VEHICLE_ID
        VEHICLE_ID += 1

        self.a_min = -3 # [m/s²]
        self.a_max = 2 # [m/s²] corresponds to 0-100km/h om 12s

        self.env = env
        self.rec = rec

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

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

        self.t0 = t0
        self.x0 = x0
        self.dx0 = dx0
        self.ddx0 = ddx0
        self.dddx0 = dddx0

        self.t = t
        self.v = v
        self.t_target = []
        self.v_target = []

        self.running = False
        self.crashed = False
        self.braking = False
        self.changingLane = False

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

        ## this allows to trigger trace messages for
        ## the new feature Surround
        self.traceSurround = False
        self.traceOvertake = False
        self.traceBrake = False

    def __str__(self):
        return f"({self.id:d})"

    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

    def updateOnly(self):
        if self.crashed:
            return False
        t = self.env.now
        if t < self.t0:
            return False
        if self.running and t > self.t0:
            dt = t - self.t0
            ddx = self.ddx0 + self.dddx0*dt
            dx = round(self.dx0 + self.ddx0*dt + self.dddx0*dt*dt/2,4)
            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

            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.rec.record(self, event='end')
                    self.running = False
                    return False
                else:
                    nextLane.enter(self, pos=nextPos)
        return True

    def update(self):
        active = self.updateOnly()
        if not active:
            return False

        self.surround = Surround(self)

        ## instead of direct link, call method
        inFront = self.surround.front
        if (isRunning(inFront) or isCrashed(inFront)) \
               and inFront.x0 < self.x0 + CAR_LENGTH:
            self.crash(inFront)
            return True

        if inFront is not None and not self.braking and \
                self.dx0 > inFront.dx0 and \
                self.x0 + CRITICAL_TIME_TOLERANCE*self.dx0 > inFront.x0:
            Δt = max(MIN_TIME_DIFF, (inFront.x0-self.x0)/self.dx0)
            self.setTarget(Δt, inFront.dx0)
            self.interruptProcess()
            return True

        ## new code: start overtaking maneuver by changing into fast lane
        if inFront is not None and \
                not self.braking and not self.changingLane and \
                self.dx0 > inFront.dx0 + MIN_SPEED_DIFF and \
                self.x0 + (LANE_CHANGE_TIME+CRITICAL_TIME_TOLERANCE)*self.dx0 > inFront.x0 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):
            if self.traceOvertake:
                print(f"t={self.t0:7,.1f}s Overtaking v{self.id:d} overtakes v{inFront.id:d} at x={self
.x0:7,.1f}m")
            self.setTarget(LANE_CHANGE_TIME, 'fast')
            self.interruptProcess()
            return True

        ## new code: end overtaking by returning to slow lane
        if self.surround.leftLane is not None and \
                not self.braking and not self.changingLane and \
                self.surround.left is None and \
                self.isNotFasterThan(self.surround.leftFront) and \
                self.surround.leftBack is None:
            if self.traceOvertake:
                print(f"t={self.t0:7,.1f}s Overtaking v{self.id:d} returns to slow lane at x={self.x0:
7,.1f}m")
            self.setTarget(LANE_CHANGE_TIME, 'slow')
            self.interruptProcess()
            return True

    def setTarget(self, Δt, v):
        self.t_target = [ Δt ] + self.t_target
        self.v_target = [ v ] + self.v_target

    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 = env.now
        self.running = True
        self.rec.startRecording(self)
        self.startingLane.enter(self, pos=self.startingPos)

        while self.running:
            self.updateOnly()

            self.surround = Surround(self)

            inFront = self.surround.front
            if inFront is not None:

                # if the car in front is slower and we are a bit too near on its heals...
                if inFront.dx0 < self.dx0 and \
                        inFront.x0 < self.x0 + CRITICAL_TIME_TOLERANCE*self.dx0:
                    if self.traceBrake:
                        print(f"t={self.t0:7,.1f}s Braking v{self.id:d} v={self.dx0:4.4f}m/s to {inFron
t.dx0:4.4f}")


                    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 = 2
                        self.setTarget(Δt, inFront.dx0)
                    continue

            if len(self.t_target)==0:
                self.t_target = self.t.copy()
                self.v_target = self.v.copy()

            if len(self.t_target)>0:

                ## add code for explicit change of lane
                if type(self.v_target[0]) is str:
                    direction = normaliseDirection(self.v_target[0])
                    t = self.t_target[0]
                    self.t_target = self.t_target[1:]
                    self.v_target = self.v_target[1:]
                    if self.lane.getLane(direction) is not None:
                        yield from self.changeLane(direction, t)

                ## the rest is what was there before
                else:
                    v0 = self.dx0
                    v1 = self.v_target[0]
                    t = self.t_target[0]
                    self.t_target = self.t_target[1:]
                    self.v_target = self.v_target[1:]
                    if isZero(v1-v0):
                        yield from self.wait(t)
                    else:
                        yield from self.adjustVelocity(v1-v0, t)
            else:
                yield from self.wait(10)

        self.rec.stopRecording(self)

    def emergencyBraking(self, v):

        def emergencyBrakingProcess(v):
            self.rec.record(self, 'brake')
            minΔt = 0.2
            self.dddx0 = (self.a_min-self.ddx0)/minΔt
            yield self.env.timeout(minΔt)

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


        ## The 'braking' bit prevents the interruption of an emergency breaking process
        self.braking = True
        self.processRef = self.env.process(emergencyBrakingProcess(v))
        try:
            yield self.processRef
        except simpy.Interrupt:
            pass
        self.processRef = None
        self.braking = False

    ## make changeLane robust against interrupt:
    def changeLane(self, direction, Δt):

        # smoothly adjust velocity by Δv over the time Δt
        def changeLaneProcess(oldLane, newlane, Δt):
            self.updateOnly()
            self.rec.record(self, 'change '+direction)
            self.oldLane = oldLane
            newLane.enter(self, pos=self.pos)
            self.ddx0 = 1
            self.dddx0 = 0
            yield self.env.timeout(Δt)
            self.oldLane.leave(self)
            self.lane = newLane
            self.oldLane = None
            self.rec.record(self, 'done change '+direction)
            self.updateOnly()
            self.ddx0 = 0
            self.dddx0 = 0

        ## keep record of current lane, as in case of aborting
        ## the lane change
        ## when interrupted go back into original lane
        oldLane = self.lane
        newLane = self.lane.getLane(direction)
        self.changingLane = True
        try:
            self.processRef = self.env.process(changeLaneProcess(oldLane, newLane, Δt))
            yield self.processRef
            self.processRef = None
        except simpy.Interrupt:
            # if interrupted go quickly back into old lane
            # but this is not interruptible
            self.processRef = None
            self.env.process(changeLaneProcess(newLane, oldLane, Δt/4))
        self.changingLane = False

    def adjustVelocity(self, Δv, Δt):

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

        self.processRef = self.env.process(adjustVelocityProcess())
        try:
            yield self.processRef
        except simpy.Interrupt:
            self.dddx0 = 0
            pass
        self.processRef = None

    def wait(self, Δt):

        def waitProcess():
            yield self.env.timeout(Δt)

        self.processRef = self.env.process(waitProcess())
        try:
            yield self.processRef
        except simpy.Interrupt:
            pass
        self.processRef = None

    def interruptProcess(self):
        if self.processRef is not None and self.processRef.is_alive:
            self.processRef.interrupt('change')

    def crash(self, other):

        def recordCrash(self):
            self.rec.record(self, 'crash')
            self.running = False
            self.crashed = True
            self.dx0 = 0
            self.ddx0 = 0
            self.dddx0 = 0

        if self.running:
            print(f"Crash p{self.id:d} into p{other.id:d} at t={self.t0:7.3f} x={self.x0:7.1f}")
            recordCrash(self)
            if other.running:
                recordCrash(other)
```

In [8]:

```
class Surround:

    def __init__(self, vehicle):

        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}"

        # 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.leftBack = vehicle
        self.left = vehicle
        self.leftFront = vehicle
        self.back = vehicle
        self.vehicle = vehicle
        self.front = vehicle
        self.rightBack = vehicle
        self.right = vehicle
        self.rightFront = vehicle

        lane = vehicle.lane
        pos = vehicle.pos
        if lane is not None:
            self.lane = lane
            self.front = lane.inFront(pos)
            self.back = lane.behind(pos)

            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(pos)
                    self.rightBack = self.rightLane.behind(pos)
                else:
                    right = self.rightLane.at(pos)
                    if len(right)==0:
                        self.right = None
                    elif len(right)==1:
                        self.right = right[0]
                    else:
                        self.right = right

                    if self.right is None:
                        self.rightFront = self.rightLane.inFront(pos)
                        self.rightBack = self.rightLane.behind(pos)
                    else:
                        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(pos)
                    self.leftBack = self.leftLane.behind(pos)
                else:
                    left = self.leftLane.at(pos)
                    if len(left)==0:
                        self.left = None
                    elif len(left)==1:
                        self.left = left[0]
                    else:
                        self.left = left

                    if self.left is None:
                        self.leftFront = self.leftLane.inFront(pos)
                        self.leftBack = self.leftLane.behind(pos)
                    else:
                        self.leftFront = None
                        self.lefttBack = None

        if vehicle.traceSurround:
            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.leftFront):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.rightFront):s}|") +
                  "|"
                 )
```

Recorder

In [9]:

```
class SimpleRecorder:

    def __init__(self, env, startTime, stopTime, timeStep):

        global VEHICLE_ID, LANE_ID
        VEHICLE_ID = 0
        LANE_ID = 0

        self.env = env
        self.startTime = startTime
        self.stopTime = stopTime
        self.timeStep = timeStep
        self.vehiclesToTrace = []
        self.vehicles = []
        self.data = pd.DataFrame(columns=['t', 'x', 'v', 'a', 'id', 'lane', 'oldLane', 'pos', 'event'])

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

    def startRecording(self, p):
        self.vehicles.append(p)

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

    def record(self, p=None, event='timer'):
        if p is not None:
            if p.updateOnly():
                laneId = None if p.lane is None else p.lane.id
                oldLaneId = None if p.oldLane is None else p.oldLane.id
                if p.running or event!='timer':
                    ix = len(self.data)
                    self.data.loc[ix]=[self.env.now, p.x0, p.dx0, p.ddx0, p.id, laneId, oldLaneId, p.po
s, event]
                    if event=='timer':
                        p.update()
        else:
            for p in self.vehicles:
                self.record(p)

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

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

    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 plot(self, x, y,
             vehicles=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=(6, 4), 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, '', data=df)
            plt.xlabel(labels[xindex])
            plt.ylabel(labels[yindex])

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

            # 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 right pointing triangle
            # to indicate that a vehicle
            # was 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 left pointing triangle
            # to indicate that a vehicle
            # was 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], '<k')

            # 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')

        plt.grid(True)
```

In [10]:

```
#def randomIntervals(cycles, length=100):
#    return [ max(0, random.normalvariate(length, length/3)) for i in range(cycles) ]
#def randomSpeedVariation(vmax, cycles, cv=0.02):
#    return [ vmax + (-1)**i*abs(random.normalvariate(0, vmax*cv)) for i in range(cycles) ]
```

In [11]:

```
#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 range(cycles) ]
#SPEED_VARIATION = 0.05
#def randomSpeedVariation(vmax, cycles):
#    return [ vmax + (-1)**i*abs(random.normalvariate(0, vmax*SPEED_VARIATION)) for i in range(cycles)
 ]
```

Genreating Free Speeds using RSA graphs Quantiles

[illegible]

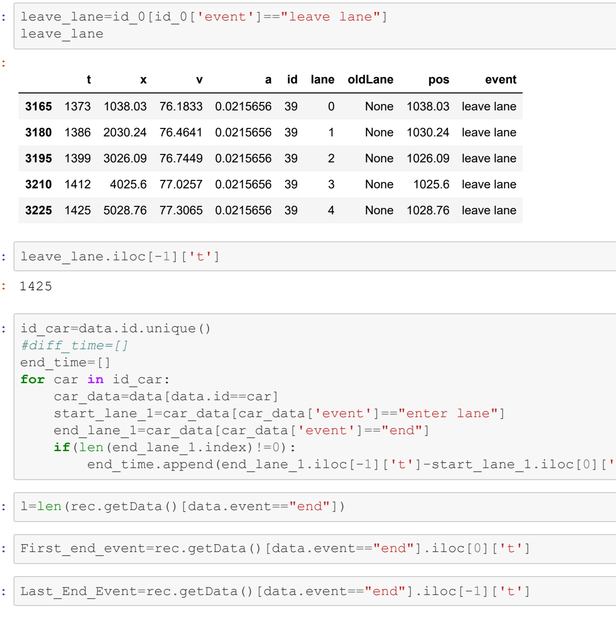
Simulation for 50 Cars

[illegible]

	t	x	v	a	id	lane	oldLane	pos	event
0–36	[illegible]								
37	29	123.03	73.0296	0.035056	16	0	None	123.03	timer
38	30	2509.92	83.2784	-0.0259185	0	2	None	509.92	timer
39	30	196.08	73.0751	0.0558606	16	0	None	196.08	timer
40	31	2593.19	83.2525	-0.0259185	0	2	None	593.19	timer
41	31	269.19	73.1414	0.0766653	16	0	None	269.19	timer
42	32	2676.43	83.2266	-0.0259185	0	2	None	676.43	timer
43	32	342.37	73.2285	0.09747	16	0	None	342.37	timer
44	33	2759.64	83.2007	-0.0259185	0	2	None	759.64	timer
45	33	415.65	73.3364	0.118275	16	0	None	415.65	timer
46	34	2842.83	83.1748	-0.0259185	0	2	None	842.83	timer
47	34	489.05	73.4651	0.139079	16	0	None	489.05	timer
48	34.5776	0	88	0	1	0	None	0	enter lane
49	35	2925.99	83.1489	-0.0259185	0	2	None	925.99	timer

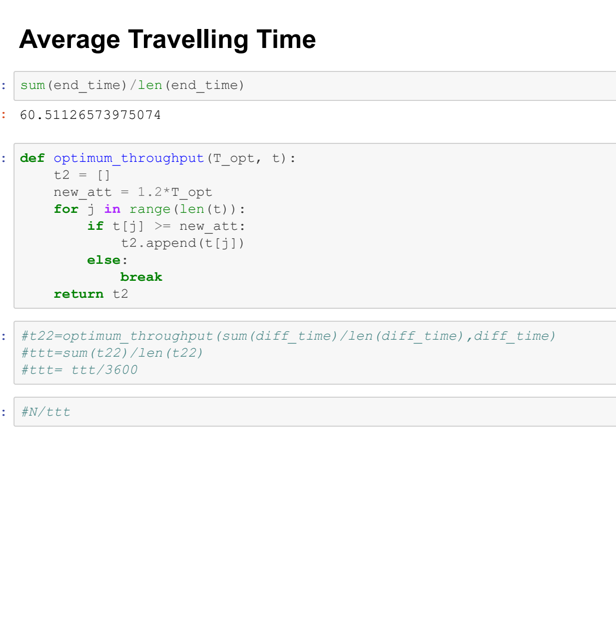
Position vs Time Graph

In [18]:
```
rec.plot('t','x')
```



Velocity vs Time Graph

In [19]:
```
rec.plot('t', 'v')
```



Time Vs Acceleration Graph

In [20]:
```
rec.plot('t', 'a')
```

In [21]:
```
id_0.groupby(id_0['event']).nunique()
```

Out[21]:

event	t	x	v	a	id	lane	oldLane	pos	event
end	1	1	1	1	1	0	0	1	1
enter lane	5	5	5	2	1	5	0	5	1
leave lane	5	5	5	1	1	5	0	5	1
timer	65	65	65	11	1	5	0	65	1

In [22]:
```
ent_lane=id_0[id_0['event']=="enter lane"]
ent_lane
```

Out[22]:

	t	x	v	a	id	lane	oldLane	pos	event
3151	1359.35	0	76	0	39	0	None	0	enter lane
3166	1373	1038.03	76.1833	0.0215656	39	1	None	38.03	enter lane
3181	1386	2030.24	76.4641	0.0215656	39	2	None	30.24	enter lane
3196	1399	3026.09	76.7449	0.0215656	39	3	None	26.09	enter lane
3211	1412	4025.6	77.0257	0.0215656	39	4	None	25.6	enter lane

In [23]:
```
ent_lane.iloc[0]['t']
```

Out[23]: 1359.3531861666538

In [24]:
```
leave_lane=id_0[id_0['event']=="leave lane"]
leave_lane
```

Out[24]:

	t	x	v	a	id	lane	oldLane	pos	event
3165	1373	1038.03	76.1833	0.0215656	39	0	None	1038.03	leave lane
3180	1386	2030.24	76.4641	0.0215656	39	1	None	1030.24	leave lane
3195	1399	3026.09	76.7449	0.0215656	39	2	None	1026.09	leave lane
3210	1412	4025.6	77.0257	0.0215656	39	3	None	1025.6	leave lane
3225	1425	5028.76	77.3065	0.0215656	39	4	None	1028.76	leave lane

In [25]:
```
leave_lane.iloc[-1]['t']
```

Out[25]: 1425

In [44]:
```
id_car=data.id.unique()
#diff_time=[]
end_time=[]
for car in id_car:
    car_data=data[data.id==car]
    start_lane_1=car_data[car_data['event']=="enter lane"]
    end_lane_1=car_data[car_data['event']=="end"]
    if(len(end_lane_1.index)!=0):
        end_time.append(end_lane_1.iloc[-1]['t']-start_lane_1.iloc[0]['t'])
```

In [52]:
```
l=len(rec.getData()[data.event=="end"])
```

In [53]:
```
First_end_event=rec.getData()[data.event=="end"].iloc[0]['t']
```

In [54]:
```
Last_End_Event=rec.getData()[data.event=="end"].iloc[-1]['t']
```

Throughput

In [67]:
```
#THROUGHPUT
l/((Last_End_Event-First_end_event)/3600)
```

Out[67]: 95.94882729211088

Average Travelling Time

[illegible]